EXPLOITING LOCALITY AND PARALLELISM WITH HIERARCHICALLY TILED ARRAYS

BY

JIA GUO

B.Eng., Tsinghua University, 2001
M.S., University of Illinois at Urbana-Champaign, 2004

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois

Abstract

The importance of tiles or blocks in mathematics and thus computer science cannot be overstated. From a high level point of view, they are the natural way to express many algorithms, both in iterative and recursive forms. Tiles or sub-tiles are used as basic units in the algorithm description. From a low level point of view, tiling, either as the unit maintained by the algorithm , or as a class of data layouts, is one of the most effective ways to exploit locality, which is a must to achieve good performance in current computers given the growing gap between memory and processor speed. Finally, tiles and operations on them are also basic to express data distribution and parallelism.

Despite the importance of this concept, which makes inevitable its widespread usage, most languages do not support it directly. Programmers have to understand and manage the low-level details along with the introduction of tiling. This gives place to bloated potentially error-prone programs in which opportunities for performance are lost. On the other hand, the disparity between the algorithm and the actual implementation enlarges.

This thesis illustrates the power of Hierarchically Tiled Arrays (HTAs), a data type which enables the easy manipulation of tiles in object-oriented languages. The objective is to evolve this data type in order to make the representation of all classes for algorithms with a high degree of parallelism and/or locality as natural as possible. We show in the thesis a set of tile operations which leads to a natural and easy implementation of different algorithms in parallel and in sequential with higher clarity and smaller size.

In particular, two new language constructs *dynamic partitioning* and *overlapped tiling* are discussed in detail. They are extensions of the HTA data type to improve its capabilities to express algorithms with a high abstraction and free programmers from programming tedious low-level tasks.

To prove the claims, two popular languages, C++ and MATLAB are extended with our HTA data type. In addition, several important dense linear algebra kernels, stencil computation kernels, as well as some benchmarks in NAS benchmark suite were implemented. We show that the HTA codes needs less programming effort with a negligible effect on performance.

To higher productivity.

Acknowledgments

First and foremost, my deepest gratefulness goes to my adviser Professor David Padua. He taught me how to do research: how to simplify a problem from its complexity, how to understand a problem from the fundamental level, how to improve the performance relentlessly, and how to turn obstacles into valuable assets. What I learned from him is beyond research: to be faithful to truth, joyful to life, and grateful to people. He has provided me with a perfect balance of guidance and freedom, which allows me to pursue my ideas along the right direction. I will also miss his wisdom and jokes which have accompanied me for six years.

I would like to thank my co-adviser Professor Basilio Fraguela, who began to advise me since 2003. He guided me in every aspect such as programming techniques, writing skills, system tools, and optimizations in my design. He always acknowledged every small progress I achieved and tolerated the mistakes I made. Although half of the discussions were through emails, his responsiveness, crystal clear writing, along with funny jokes, made the communication pleasant and enjoyable.

Many thanks to Professor Maria J. Garzaran who helped me since I started my master work. I still remember vividly, in a sunny afternoon, she sat with me, side by side, manually modifying an assembly code to test our conjecture. She gave many suggestions on my work, my presentations, and the papers we wrote.

I thank the honorable members of my thesis committee Professor Michael Heath and Professor Laxmikant Kale for their helpful discussions and suggestions on the ideas in my work.

I would like to extend my special thanks to my colleague Ganesh Bikshandi, Christoph

von Praun (IBM T.J. Watson Research Center) and Gheorghe Almasi (IBM T.J. Watson Research Center). The inspiring discussions with them help me generate part of the ideas in the thesis.

I am grateful for being a member of Polaris group. My colleagues are like shining stars whose thoughts enlightened my mind. I thank Jianxin Xiong, Jianjing Zhu, Zehra Sura, David Wong, Xiaoming Li, for their help. I also want to thank Sheila Clark for her fast administrative work.

Finally, I am indebted to my parents for years of love and support. Their devotion and guidance are the source of my achievements. Last but not least, I thank Chao for bringing me happiness, encouragement, and a baby boy, David.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Tiling

Tiling [32] is ubiquitous in scientific and engineering programs for several reasons. At the high level, tiles are a natural way to express iterative and recursive, serial and parallel, algorithms. Some algorithms are described more naturally in terms of tiles [43]. Projects such as FLAME [7] have worked extensively on generating blocked algorithms and proposed programming APIs that exploit this feature. Programs expressed in term of tiles allow an easier comparison of different algorithmic variants, and reduce the probability of errors as indexing expressions appear less often. Still, most research on tiling focuses on it as a program transformation in order either to distribute data and computations [40] or to enhance locality [47] [20]. The usage of tiled layouts in order to exploit locality effectively on matrix computations has also been an object of study [25][16].

1.1.1 Dense linear algebra

McKellar and Coffman [37] were the first to use a partitioned matrix representation to reduce the number of page faults with respect to naive algorithms. One of their partition schemes, row storage, partition the matrix into sets of rows, each of which fit in one page. Another partition, submatrix storage, partitions the matrix into submatrices with each submatrix occupying one page. By analyzing different matrix operations like addition, transpose, multiplication and Gaussian elimination, it was shown that the number of page faults can be significantly reduced by sequencing the individual computations appropriately. It was also shown that the storage by submatrices is superior to storage by rows for all algorithms they considered with the possible exception of forward elimination.

The ATLAS library generator [46], which incorporates the best techniques used to implement BLAS routines, makes use of the McKeller and Coffman's approach to increase the locality of matrix matrix multiplication. But their focus is cache memories and general purpose registers. ATLAS assumes multiple levels of memory hierarchy and utilizes register blocking and cache blocking at least for the level 1 (L1) cache. Blocking is applied in order to partition the original problem to make the working set small enough to fit into the L1 cache. To implement matrix matrix multiplication $C = A * B$, ATLAS partitions A, B and C into submatrices and then searches for the largest size of the submatries so that the working set of their multiplication (mini-MMM code) fits into the cache. A second level of blocking, register blocking, is used for temporal locality in the code where floating point computation is intensive. The register blocking assigns a_r and b_r registers for submatrices of A, B respectively and $a_r \times b_r$ registers for elements of C. Thus if N_r is the number of registers available to the algorithm, ATLAS will evaluate the performance of all values a_r and b_r that satisfy the condition $a_r b_r + a_r + b_r \leq N_r$.

A modern extension to the McKeller and Coffman's strategy is cache oblivious algorithms [22], which do not have to be tuned for optimality as a function of hardware parameters such as cache size and cache-line length. Those algorithms are recursive algorithms and have divide-and-conquer structure in the code. To match the algorithm, the data structure is also recursively tiled. The main difference with McKeller and Coffman as implemented in ATLAS is that cache-oblivious algorithms are self adjusting and there is no need to determine tile sizes a priori.

The FLAME project [7] follows a systematic approach to derive high-performance dense linear algebra algorithms. It uses *precondition* and *postcondition* to generate algorithms. A *precondition* in FLAME describes the dimension, size requirements of input and output variables. If a variable Z is both an input and output variable, $\hat{Z}$ is typically used to denote the original content of Z and $Z = \hat{Z}$ is included in the precondition. A *postcondition* describes the relationship between the original and modified matrix after the operations performed on different variables. One of the important steps in the systematic derivation is to partition a variables in the precondition representing a matrix into several submatrices. Then, the partitioned variable is substituted into the postcondition to determine the partitioning of the other variables. Different partitions lead to a family of algorithms for the same problem. Those algorithms can be unblocked (by specifying the block size of 1), blocked, iterative, or recursive. The description of the algorithm in FLAME raises the level of abstraction at which data (matrices) are expressed so that indexing details are hidden. However, once a correct algorithm has been derived, it is necessary to manually translate this algorithm into code since the FLAME notation does not constitute a programming language.

1.1.2 Sparse computations

Tiling is also important for sparse computations. Experimental studies have shown that cache blocking and the register blocking are critical to the performance in sparse matrix computations. The SPARSITY generator [31] incorporates widely used techniques for important computational kernel sparse matrix-vector multiplication.

Due to the irregular memory access pattern that sparse matrices exhibit, blocked algorithms for sparse codes are more difficult to write than those of dense computations. For example, consider the operation of sparse matrix and dense matrix multiplication $D = S \times B$ where S is a sparse matrix and B,D are dense matrices. The sparse matrix S uses the Compressed Row Storage (CRS) format with three vectors: vector A contains the sparse matrix non-zero elements; vector C stores the column of each entry; vector R points to the location in vector A and C where a new row of the sparse matrix starts. The code in Figure 1.1 extracted from [21] shows the tiled version. For simplicity, we omit the register tiling part. Figure 1.2 shows the pictorial view of cache tiling used in the code. Matrices `S`, `B`, and `D` are partitioned along dimension K and J so that the blocks of B fit into cache. In lines 9 and 18, vector R2 is used to access to the sub-rows of sparse matrix. It points to the location in vectors A and C where these sub-rows start. The irregular access of B is shown in line 14 in Figure 1.1. Although the partition shown in Figure 1.2 is regular, the code has to manage irregular data structure.

```
 1 DO J2=0,H,BJ
 2   LIMJ = J2+MIN(BJ, H-J2+1)-1
 3   DO I=1, M+1
 4     R2(I)=R(I)
 5   ENDDO

 6   DO K2=1, N, BK
 7     LIMK = K2+MIN(BK,N-K2+1)
 8     DO I=1, M
 9       K=R2(I)
10       LK=R(I+1)
11       DO WHILE (K<LK AND C(K)<LIMK)
12         DO J=J2, LIMJ
13           ind = C(K)
14           D(I,J)=D(I,J)+ A(K)*B(ind,J)
15           K=K+1
16         ENDDO
17       ENDDO

18       R2(I)=K
19     ENDDO
20   ENDDO
21 ENDDO
```

Figure 1.1: A sparse matrix-dense matrix multiplication with blocking at the cache level.

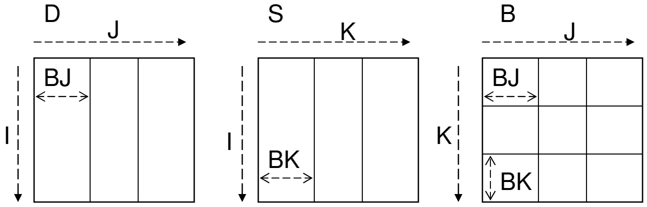


Figure 1.2: Pictorial view of cache tiling for sparse matrix-dense matrix multiplication

1.1.3 Data layout

Another effort to enhance the locality based on tiling is to study the data layout of the programs. We call the default row major or column major layout *linear* layout, and other data layout *non-linear* layout.

To create a non-linear layout of the memory, Chatterjee et al. [16] chose a $t_R \times t_C$ block of the $m \times n$ array so that if stored in contiguous memory locations, it would fit in cache without self-interference. Then, it conceptually views the $m \times n$ array as a $\lceil \frac{m}{t_R} \rceil \times \lceil \frac{m}{t_C} \rceil$ array of $t_R \times t_C$ tiles. Within each tiles, the layout is linear. If the tiles are also placed in a linear layout, the resulting layout function is *4D* layout [16]. Figure 1.3 (a) shows an example of *4D* layout where the layout inside the tiles is column-major and the layout across tiles is row-major. Another non-linear layout is the *Morton* layout [42]. One version, the Z-Morton layout, divides the original matrix into four quadrants, and places these submatrices in memory in the linear order NW, NE, SW, SE. Each representing $k_R \times k_C$ submatrix as long as $k_R > t_R$ and $k_C > t_C$ is laid out recursively using the Morton layout. When the partitions reach the size $t_R \times t_C$, the resulting matrix is stored in conventional linear layout. This layout is shown in Figure 1.3 (b).

In [16], it is claimed that certain non-linear data layouts provide significant performance benefits for dense matrix kernels. The execution time is reduced by factors between 1.1 and 2.5. on UltraSPARC-II and Alpha 21164 in 1999.

1.1.4 The current limitations

Despite the importance of tiles, most programming languages do not provide support for them. Rather, compilers are expected to find those loops that can benefit from tiling and transform them automatically, something that too often is beyond their capabilities. For this reason, programmers are typically forced to proceed manually and add new loops and index computations and/or manually redefine their data structures to achieve tiling. Tiling

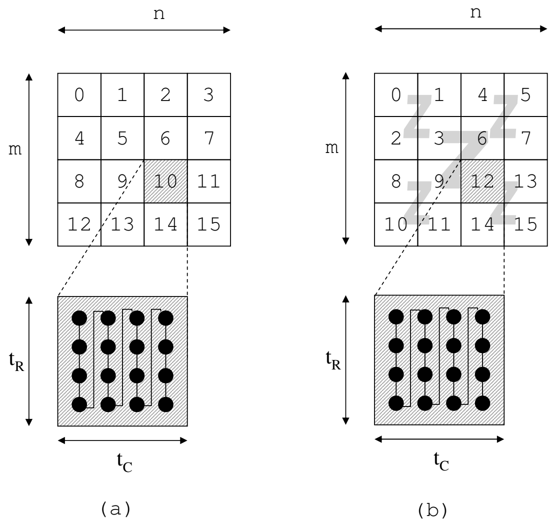


Figure 1.3: Graphical description of layout functions. Arrays are $m \times n$; tiles are $t_R \times t_C$. (a) *4D* layout. (b) *Morton* layout

is frequently applied because it is the way to distribute data and computations and to exploit locality in many scientific codes. The price to pay is reduced programmer productivity and an increased complexity of the resulting applications, which makes them less readable and thus more error-prone.

Language extensions that express tiling have most of the times limited themselves to specify that data should be distributed in blocks on a mesh of processors (HPF [28], UPC [13]). Even those languages that allow explicit references to the blocks in which an array is divided, such as Co-Array Fortran [38], exhibit many limitations for their definition and manipulation.

1.2 Our approach and contributions

It is desirable to have language extensions which address tiling explicitly. These extensions should satisfy the following requirements. First, they should provide a higher level abstraction of the problem. The explicit tile notation and the operations defined for tiles can simplify programs by avoiding the need for inner nest loops introduced for tiling. Also, the higher level abstraction should be recursive. At each memory level, such abstraction is still valid. Second, the extensions should hide as much as possible machine-dependent details of a program. Although the target architecture could change, the tiled algorithm itself should not be affected. Because the architecture-related parameters such as tile size and the levels of tiling usually must be modified for each new architecture, they should be easy to change. Third, the language extension should also provide good performance along with the clean syntax.

The Hierarchically Tiled Array (HTA) [10] data type described in this thesis is the result of our pursuit for an easy, thorough and consistent way to integrate tiles and their manipulation both in sequential and parallel programs. This type describes arrays partitioned into tiles, which can be themselves lower level HTAs or final conventional arrays.

The contributions of this thesis are summarized in the following five bullets below. The first two contributions are a joint work with other colleagues. The other contributions are based on an individual effort.

- *The design of the HTA data type.* We designed the HTA data type and its related operations. The rules for accessing HTAs, operating HTAs and assigning to HTAs were formally defined. We also defined the semantics and implementation of HTAs so that programs based on them have a global view of the data with a single-threaded programming model. Such design improves readability and ease of development.

- *The implementation of the HTA library.* We implemented the HTA data type in two different languages, MATLAB and C++. The first implementation in MATLAB followed client-server mode and provided a good debugging and interactive environment for parallel programming and programming for locality. Then, we changed to SPMD execution model to minimize the communication overhead. Still, we found that the inherent constraints imposed by MATLAB hampered the performance sometimes. With several optimizations in htalib, our C++ implementation, the HTA runtime system successfully matches the performance of hand optimized codes or library routines.

- *The study of HTA programs for locality.* One of the most important objectives of the HTA project is to evolve the HTA data type in order to enable it to represent all classes of algorithms with a high degree of locality and/or parallel as natural as possible. The strengths of the HTA data type were discussed and summarized through several important kernels in different algorithm categories. Meanwhile, we identified two necessary language constructs that would be of interest for a broad range of algorithms.

- *The identification of two language constructs.* We designed and implemented the two language constructs to facilitate programming with HTAs. The first one, *dynamic partitioning* allows to modify at run-time the tiling of an HTA in a completely flexible way. HTAs extended with this mechanism seem to be the most powerful and easy way to express both iterative and recursive algorithms expressed in terms of tiles. The second extension, *overlapped tiling*, hides all the details of the management of shadow regions, which usually arise when tiling is applied to stencil computations. This thesis mostly focuses on the tiled algorithms for locality. However, the second construct is also extended for the parallel case. Our validation shows that both mechanisms greatly improve programmer productivity while their impact on performance is negligible.

- *The evaluation of the HTAs* focusing on the new language constructs resulting from the previous study. The evaluation of the HTA approach fell in two areas. First, the programming productivity was evaluated by several metrics: the programming effort [26], the program complexity [36], and the source lines of code. Our experiments indicate the HTA data type effectively relieves the programmer's burden; reduces the program complexity and the source lines of code. Second, several sequential and parallel benchmarks were implemented with HTAs and were compared with their original implementations in both MATLAB and C++. The HTA programs were able to obtain a similar performance in all the cases.

1.3 Thesis organization

The thesis is organized as follows.

In Chapter 2 describes in detail the definition of HTA data type, the classification of different types of HTAs, the construction syntax and useful operations are provided. The implementations of HTA in two languages are shown in Chapter 3. In Chapter 4, we discuss through three examples that how HTA facilitates the tiled algorithms for locality. In Chapter 5, we introduce *dynamic partitioning*, a new language construct to change the tile structure at runtime. The syntax and implementation are explained. Then, we show how dynamic partitioning can be applied to different types of algorithms. In Chapter 6, another language construct *overlapped tiling* is presented. We introduce the motivation, the syntax and implementation challenges. Later, we discuss how overlapped tiling can be applied to stencil computations. In Chapter 7, experiments are conducted on the improvement of programming productivity and the performance evaluation. In Chapter 8, we examine the related research efforts to help programming tiled algorithms. Finally, we conclude in Chapter 9.

Chapter 2

An Overview of Hierarchically Tiled Arrays

The objective in this chapter is to provide a clear and consistent description for the HTA syntax and operations, since they are frequently used in the foregoing discussion. We start from the definition, classification, and creation of HTAs. Then, we describe the operations on HTAs: the indexing, binary operations, assignments, and other methods.

HTAs have been implemented as a class both in MATLAB [10] and in C++ [11]. We will describe the details of the two implementations in the second part of this chapter. We will use the HTA syntax provided by the C++ implementation with small simplifications. The HTA implementation library in C++ is called htalib.

2.1 Hierarchically Tiled Arrays

We define an array is a **tiled array** if it is partitioned into sub-arrays in such a way that adjacent sub-arrays have the same size along the dimension of adjacency. Although the literature usually assumes that array tiles have the same shape (the same number of dimensions and size of each dimension), we do not require this in our definition because there are important cases where using tiles of different sizes is advantageous. Furthermore, "randomly" partitioned arrays, where two adjacent tiles have different size along the adjacent dimension, do not fall into the category of tiled arrays. Figure 2.1 shows an illegal tiled array.

We define **Hierarchically Tiled Arrays (HTAs)** to be tiled arrays where each tile is either an unpartitioned array or a hierarchically tiled array.

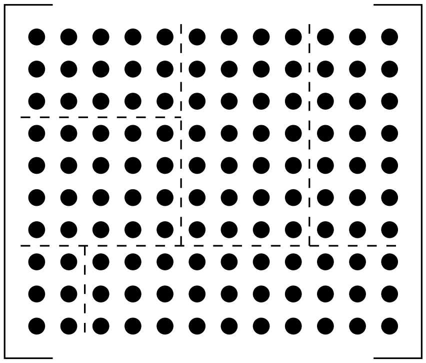


Figure 2.1: An illegal tiled array.

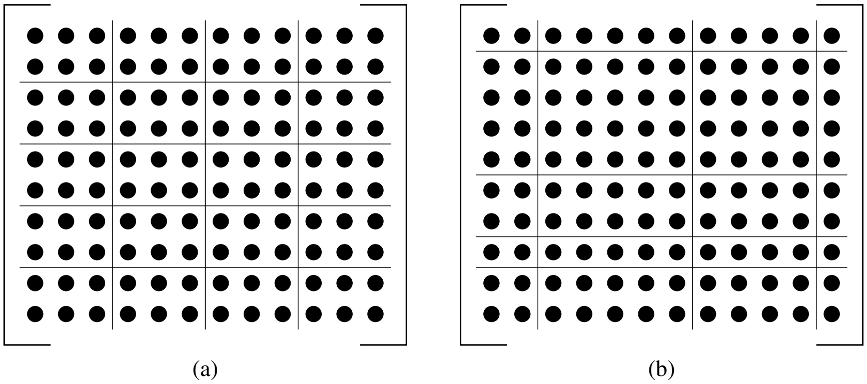


Figure 2.2: HTAs with one level of tiling. (a) Regular (b) Irregular

2.1.1 Classification

We define an HTA to be a **regular HTA** if all the tiles in that HTA at any given level have the same size. Otherwise, the HTA is said to be **irregular**.

Figure 2.2 (a) is a regular HTA with all tiles the same size. Figure 2.2 (b) shows an irregular HTA where the tile sizes are different. However, the size along the adjacent dimension is still the same for all adjacent tiles. Note the regularity of an HTA is based on the size at a certain level. In general, HTAs can have more than one level of tiling. The sizes of the tiles at each level can be different. So the regular HTA requires each level of the HTA to be regular. Otherwise it is an irregular HTA.

We define an HTA to be **homogeneous** if the number of levels of tiling is the same for all the tiles, and at each level, the sizes of the partitions are the same along the adjacent dimensions for all adjacent tiles. If these conditions are not satisfied, the HTA is a **heterogeneous** HTA.

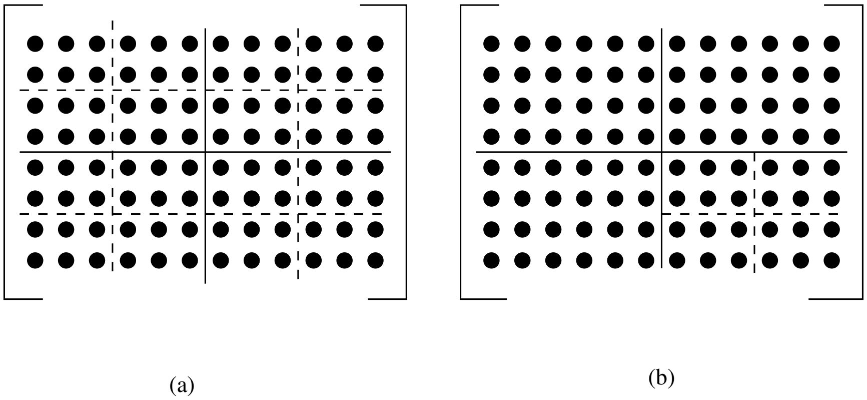


Figure 2.3: HTAs with two levels of tiling. (a) Homogeneous (b) Heterogeneous

The HTAs we discussed in Figure 2.2 are homogeneous HTAs. Also, Figure 2.3 (a) gives an example of another homogeneous HTA. It has two levels of tiling. The solid lines represent the first level of tiling, while the dashed lines represent the second level of tiling. For all adjacent tiles, the partition position is the same along the adjacent dimensions. Figure 2.3 (b) is a heterogeneous HTA because the tile at the southeast corner has one more level of tiling than the other tiles.

We define an HTA to be a **sequential** HTA if all its tiles are stored in a single machine and operated sequentially. If its tiles are distributed across processors in a distributed-memory machine and the tiles can be operated in parallel, the HTA is called a **distributed** HTA. HTA can also be operated in a shared memory node or a cluster of shared memory nodes. However, in this thesis, the main focus of parallelism is on distributed memory systems.

The classification of HTAs helps the programmer to decide which particular types of HTA to use in different applications. The interfaces for creating different kinds of HTAs are provided and discussed in this chapter. Also the tiling structure of HTAs can be changed dynamically from one to the other. This will be discussed in detail in Chapter 5.

2.1.2 Symbols, notations, and terminologies

We can represent the tiling structure of an HTA with multiple levels of tiling in Figure 2.4 (a) through a tree as the one illustrated in in Figure 2.4 (b). The root of the tree is the top level of the HTA. The circles filled with black at the bottom are scalars, and the nodes in between represent tiles partitioned at different levels. Scalars form the leaf tiles of the HTA tree.

We define *the level of an HTA* as the length of the longest path from the root to the leaf tiles in its tree representation. In Figure 2.4 (b), for example, the HTA has two levels. We number the levels from the root to the leaf tiles starting with zero. Thus, the root of the HTA is at level 0, while the leaf tiles are at level 2. The notation *n-th level of tiling* refers to the partition at level *n*. To simplify our explanation, we also include two variables *LEAF_LEVEL* and *ROOT_LEVEL* to indicate the level of the leaves and the root, respectively.

We define *the tile size for level n* as the number of components (tiles or scalars) in each dimension for one tile in level n. In Figure 2.4 (a), the tile size for the root level is 1×2 and the tile size for the leaf level is 1×3. We define *the absolute tile sizes for level n* as the total number of scalars a tile at level n contains. The absolute tile size for the root level is

1 × 12 and the absolute tile size for level 1 is 1 × 6 in Figure 2.4. These notations are only applicable for homogeneous HTAs.

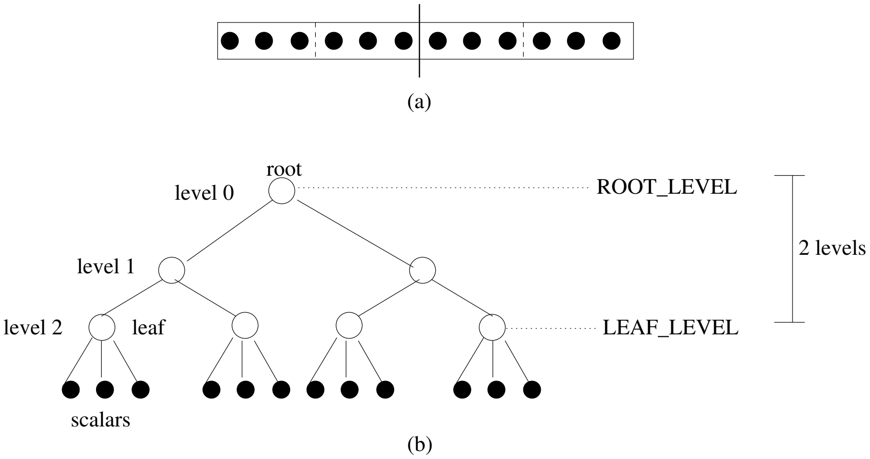


Figure 2.4: Naming and numbering conventions for various levels in an HTA. (a) A HTA with two levels of tiling (b) The HTA tree representation

To simplify the representation of HTA creation and operations, we use a comma separated list of integers *(x, y, ...)* to represent a tuple rather than the actual `Tuple<NDIMS>(x, y, ...)` used by htalib. A tuple is an n-dimensional value from Z^n. Similarly we will use the *low:step:high* notation as in MATLAB to represent a range of integers instead of the real `Triplet(low,high,step)`. In both cases the *step* can be omitted if it is 1. If only " :" is used in the range, it indicates that all the elements in that range is selected.

2.2 Construction of HTAs

2.2.1 One level of tiling

The syntax for creating a regular HTA with one level of tiling is

```
HTA<type, dim>::alloc(int level, Seq tiling,
                 Distribution<dim>* dist=NULL, Order dataLayout=ROW);
```

The function `alloc` is a general function that works for one level and multiple levels of tiling. The data type `type` and the number of dimensions `dim` of the HTA are template parameters of the HTA class.The first parameter specifies the level of tiling to be `1`. The second parameter is the specification of the tiling sequence. The sequence records the tile sizes for all the levels from the leaf level to the root level. The third parameter is the distribution of the HTA if it needs to be distributed. The last parameter is the data layout we want to have for the leaf tiles. We will discuss the last two parameters later.

For example, to create the HTA shown in Figure 2.2 (a) using the `double` data type, we have

```
Seq tiling = ((2,3), (5,4));
HTA<double, 2>::alloc(1, tiling);
```

The first parameter gives the levels of tiling: 1. A 10 × 12 matrix is to be allocated and partitioned into an array of 5 × 4 tiles of 2 × 3 elements each, as specified in the `tiling` parameter in the first statement. The second statement creates the 5 × 4 HTA with the specified `tiling` parameter. The default values of `dist` and `dataLayout` are used since they are not explicitly specified.

For irregular HTAs, the creation follows the same way except the specification of the `tiling` parameter. It now consists of vectors recording the positions of partition lines in each dimension. For instance, we define the `tiling` for Figure 2.2 (b) as:

```
Seq tiling = (Vector(1,5,7,8), Vector(2,7,11));
```

2.2.2 Multiple levels of tiling

Two types of construction methods can be used to build HTAs with multiple levels: *top-down* construction and *bottom-up* construction.

The top-down construction builds the HTA from the top and successively refine each partition. It can generate both homogeneous and heterogeneous HTAs. First, an empty HTA is created with using

```
HTA<type, dim>::allocShell(Tuple<dim> tiling, Distribution<dim>* dist=NULL);
```

The parameter `tiling` describes the tile sizes for the root level of partitioning. After the empty top level HTA is created, each one of its tiles can be filled with either an HTA or a matrix.

Before presenting the example of top-down creation of HTAs, we need to describe how to address the tiles in an HTA. The top level tiles of an HTA can be addressed using subscripts enclosed within parenthesis. More details on the indexing operation on HTA will be provided later. Figure 2.5 shows the C++ like pseudo code of top-down HTA construction for the HTA illustrated in Figure 2.3 (a). The HTA in Figure 2.3 (b) can be built similarly by filling the first three tiles a 4 × 6 matrix each and the last tile an HTA of 2 × 2 tiles with 2 × 3 elements each.

Top-down creation is preferred for parallel programs, as it does not require the entire array to be allocated before the HTA is created; each processor can allocate only the tiles it owns.

```
HTA<double,2> h = top_down(2);

HTA<double,2> top_down(int level)
{
  if(level == LEAF_LEVEL){
    return rand(3,3);
  }else{
    HTA<double,2> h = HTA<double,2>::allocShell((2,2));
    for (int i=0; i<2; i++)
      for(int j=0; j<2; j++)
        h(i,j) = top_down(level-1);
    return h;
  }
}
```

Figure 2.5: Top-down creation of HTA.

The bottom-up construction tiles the matrix from the lowest level of the hierarchy first and then proceed by tiling the resulting tiles up to the highest level . The function `alloc` follows a bottom-up construction approach.

```
HTA<type, dim>::alloc(int level, Seq tiling,
                      Distribution<dim>* dist=NULL, Order dataLayout=ROW);
```

The `level` specifies the number of levels in HTA. The `tiling` provides the sequence of sizes for each level of tiling from the bottom leaf level to the root level. The construction of the HTA in Figure 2.3 can also be:

```
Seq tiling = ((2,3), (2,2), (2,2));
HTA<double, 2>::alloc(2, tiling);
```

2.3 Distribution and data layout

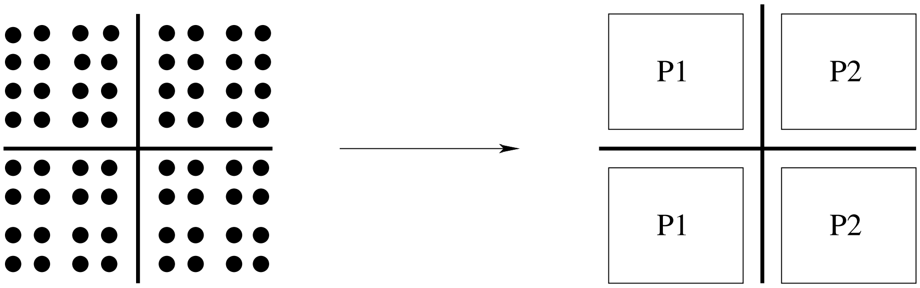
The distribution of an HTA specifies where the tiles of the HTA are allocated in a distributed system. Currently, we only distribute the top level of tiles in distributed HTA. The distribution is expressed by instances of the class `Distribution`, which specifies the *home location* of the scalar data for each one of the tiles of an HTA.

The default distribution is *row cyclic*. That is, the tiles of an HTA are assigned one-to-one to the available processors in row major order. If the number of processors is fewer than the number of tiles, then the assignment of remaining tiles starts from processor 0 and proceeds again in row major order. The other valid types of distribution are *column cyclic*, *block cyclic*, and *double cyclic*. In Figure 2.6 (a), we show the code to define a row cyclic distribution in the second statement. The tuple `1,2)` in the distribution specifies the logical arrangement of processors, which we call processor topology. If the topology is not specified, the processors are assumed arranged linearly. It distributes cyclically the tiles onto two processors. Figure 2.6 (b) gives the pictorial view for the cyclic distribution.

The data layout specifies the memory mapping of the data on each processor or on a sequential machine. Currently HTA supports linear data layouts (row and column major) and a *tile* data layout for the leaf tiles. The `tile` data layout stores the elements within a

```
Seq tiling = ((4,4), (2,2));
CyclicDistribution dist((1,2));
HTA<double,2> h =  HTA<double,2>::alloc(1, tiling, dist, ROW);
```

(a)



(b)

Figure 2.6: To create distributed HTA. (a) The code. (b) The illustration

tile continuously in memory in row major. The layout across tiles is also row-major. Figure 2.7 (a) shows the tile data layout. Currently the data layout is only specified for leaf tiles. However, it is also possible to extend the definition of the data layouts for the upper levels of tiles. For example, in a two level HTA in Figure 2.7 (b), if the tile data layout is specified for both the leaf level and the upper level, the overall data layout is the *Morton* data layout [16]. Nevertheless, currently we only define the data layout for the leaf level. This is because the tile data layout defined only for the leaf level proves to lead to high performance for certain computations as good as the Morton data layout [16].

In order to specify a data layout, an additional parameter should be added in the creation function of HTAs, as shown in the third statement in Figure 2.6 (a). Programmers can try different data layouts for performance tuning by changing the data layout at the creation time. If no data layout is specified, the default row major layout is used.

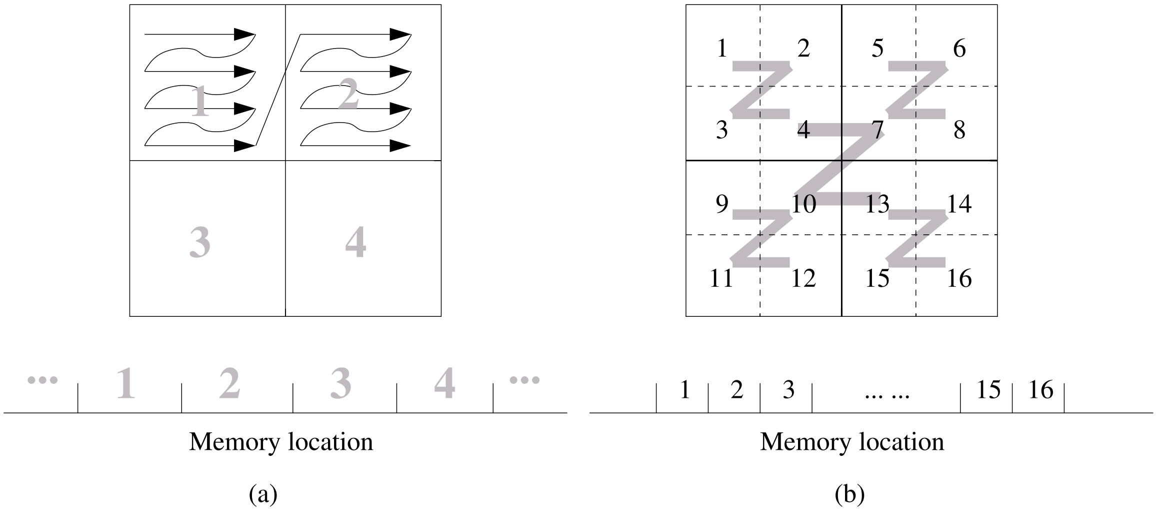


Figure 2.7: Data layout. (a) Tile (b) Morton

2.4 Accessing HTAs

HTA indexes are 0-based in `htalib` as it is built in C++. We overload the operator `()` to access tiles and he operator `[]` to index elements in an HTA disregarding the tiling structure. In any kind of indexing, a range of element positions or tiles may be chosen in each dimension using triplets of the form *begin:step:end*. Figure 2.8 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, :)[0:2, :]` selects the first three rows in tiles (0,0) and (0,1).

We can also use boolean arrays as HTA subscripts When this logical indexing is applied every true element in the boolean array is used as a subscript that designates a tile of the HTA based on its position in the boolean array. As illustrated in Figure 2.9, logical indexing allows the selection of arbitrary, banded diagonal or upper triangular tiles of an HTA.

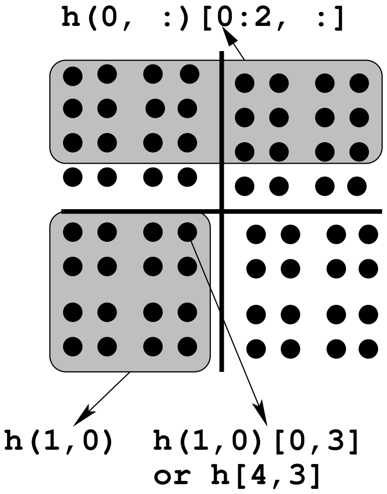


Figure 2.8: Different ways to index HTAs

2.5 Binary operations, assignments and methods

We generalize the notion of conformability of Fortran 90 to define the legality of operations on HTAs. 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 can also be conformable to an untiled array if each innermost tile of the HTA is conformable with the array. The untiled array will be operated with each 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. References to local HTAs do not involve communication. However, in distributed HTAs assignments between tiles located in different processors involve communication.

$$K = \begin{bmatrix} true & false & true \\ false & false & false \\ true & false & true \end{bmatrix} \qquad I = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} \quad J = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}$$

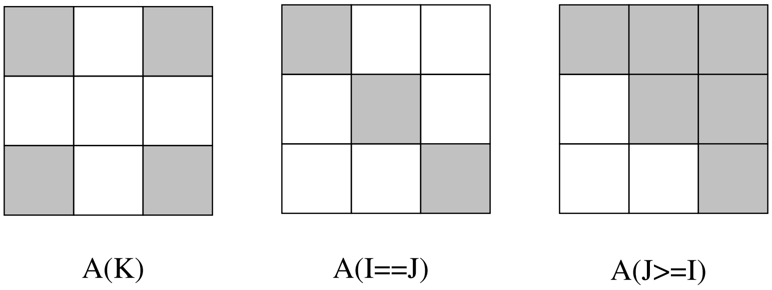


Figure 2.9: Logical indexing of HTAs

We have implemented frequently-used functions on arrays such as `circshift`, `transpose`, `permute`, or `repmat`, and we 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 express the algorithm at the tile level. A more detailed description of HTA operations can be found in [9].

2.6 A paradigm with a global unified view

The HTA programming model corresponds to that of a global view language where the programmer specifies the behavior of the algorithm as a whole, without worrying about the program's behavior on a per processor basis. As a result, the sequential and distributed HTA versions of a program share great similarity. Data are also global and are handled in a unified way. The HTA programming model provides a deterministic semantics: each statement is completed before the next one begins its execution; and the right hand side of assignments is evaluated before assignment takes place. As a result, programmers need not specify synchronization or worry about deadlocks, race conditions, etc. Thus, HTA syntax and semantics are simpler and cleaner than those of SPMD approaches. The downside is that asynchronous overlap cannot be explicitly stated using the current HTA operations, but much of this overlap can be achieved automatically with the appropriate implementation.

HTAs improve programmers' productivity, since they can use familiar programming languages and sequential modules, perhaps with small changes. So, programmers can write parallel programs the same way they write sequential programs and they can even gradually migrate sequential applications to parallel forms.

The HTA syntax makes communication explicit, since computations involving tiles located in different processors will result in movement of tiles. This provides a simple, but powerful performance model like the what-you-see-is-what-you-get (WYSIWYG) performance model in ZPL [17]. This is something missing in other global view parallel programming approaches like HPF [28, 34].

Chapter 3

Two HTA Implementations

We have developed implementations of the HTA class for two object oriented languages: MATLAB and C++. MATLAB provides a natural and clean syntax for HTAs, while C++ provides good opportunities for performance improvement. We implemented HTAs as a class using the OO capabilities of each of these two languages. Although a single implementation could be used across languages, we decided to develop a different prototype implementation for each language mainly because our first implementation, developed for MATLAB, demonstrated that, for many computations it is difficult to obtain reasonable absolute performance. To avoid the inefficiencies of MATLAB, we reimplemented the library in C++, which achieves reasonably good performance for both sequential and parallel HTA programs. Although the two implementations are different in syntax and performance, they share an important thing in common: the execution model.

In this chapter, we first show the execution model used in both implementations. Then, we describe the MATLAB and C++ implementations in turn.

3.1 An SPMD execution model

In both implementations, we follow an SPMD execution model for performance while still keeping the global view for the programmers.

All processors execute the same program. Scalar variables, arrays and non-distributed HTAs are replicated in each processor. Distributed HTAs are also created in every processor, including in those that do not own tiles of the HTA. In this last case, processors will only keep information about the structure of the HTA.

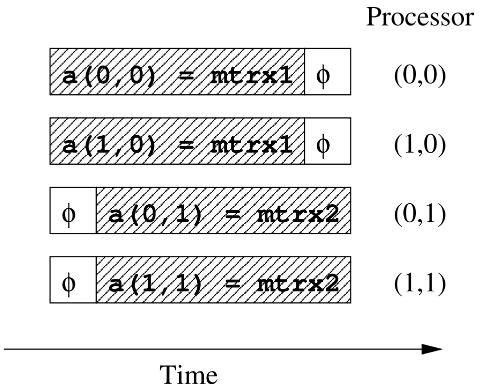
When an invocation to a method on a distributed HTA is performed, each processor applies *the owner computes rule.* It conducts the corresponding operation to the tiles of the HTA it owns and stores the corresponding tiles of the resulting HTA. Since every processor has all the information on the structure and distribution of each distributed HTA, they are completely autonomous for determining when and which tiles to sand and receive. The tiles of the right operand not co-located with the tiles of the left operand, are copied (using MPI primitives). Then, the processors that own the tiles of the left HTA perform the desired operation and store the resulting tiles. In this way, the HTA resulting from a binary operation has always the same mapping as that of the left operand. Of course, when the HTAs to operate are mapped in the same way, there is no communication. Similar communication occurs when the right hand side (RHS) of an assignment is not mapped in the same way as the left hand side (LHS). Every tile of the RHS HTA of the assignment is copied to the processor that owns the corresponding tile of the LHS HTA to which it is assigned. With these simple rules, the programmers know at all times where and how communication takes place.

In our implementation, we use an idea similar to that of integer sets in the dHPF compiler [4] to generate SPMD codes from a global view program. In [4], general formulations of computation partitioning and communication analysis tasks are presented as abstract operations on sets of integer tuples. The primitive sets are data indexes, loop indexes and processor indexes. The mappings are from processor indexes to data indexes, and to loop indexes, and from loop indexes to data indexes. In our work, it is much simpler because of the tiling expressions. The primitive set in our case is the tile indexes, which capture the relation of data indexes and loop indexes. The mapping is from processor indexes to tile indexes. This greatly simplifies the formulation of integer sets used in SPMD codes.

Our implementation also allows asynchronous execution of statements without affecting the sequential deterministic semantics of HTAs. Although there is no explicit synchroniza-

```
(1) a(0:1, 0) = mtrx1;
(2) a(0:1, 1) = mtrx2;
```

(a)



(b)

Figure 3.1: An example of asynchronous execution. (a) The code. (b) The execution timeline

tion after each instruction, our implementation makes sure there are no deadlocks. If some of the processors participating in the execution of an HTA method need to synchronize, only a partial synchronization on those processors takes place. As a result, computation and/or communication may be overlapped in different processors even for different statements.

Let us consider for example the code in Figure 3.1 (a), where `a` is a 2×2 HTA distributed on a grid of 2×2 processors so that tile(i,j) is on processor (i,j). As a result, only processors (0,0) and (1,0) are involved in the execution of the assignments in statement (1). Processors (0,1) and (1,1) do not own tiles affected by this assignment, so they exit this method and continue to execute the next instruction. In the second instruction, processors (0,1) and (1,1) assign tiles with `mtrx2`, while processors (0,0) and (1,0), which reach the statement after finishing the previous one, complete the second instruction soon, since they do not own tiles that the second assignment affects. As shown in Fig. 3.1 (b), the execution time (grayed) of the first instruction on processors (0,0) and (1,0) is overlapped with that of the second instruction on processors (0,1) and (1,1). Letter ϕ denotes the time used by a processor to check that an instruction does not affect the tiles it owns.

We believe our new parallel programming paradigm with a single thread programming model and an SPMD style implementation has many advantages. Current approaches either rely on the programmers to write efficient programs in an SPMD way [23, 13, 38] or rely on the compiler to generate SPMD codes [14, 3]. Our approach releases the burden of both the programmers and the compiler. It lets the programmers write programs very similar to the sequential ones, and relies on the library-based HTA methods to generate efficient SPMD codes automatically at runtime.

3.2 A MATLAB implementation

The purpose of our first implementation on MATLAB was to demonstrate that a conventional sequential language could be easily extended for parallel computation or programming locality using HTAs. MATLAB was a natural choice for this experiment because of its array syntax and OO capabilities. We found that MATLAB 's syntax for cell array accesses, generalized with triplet notation and extended to allow operations between components, was convenient to represent HTA accesses and therefore we adopted it. Table 3.1 summarize the major syntax differences between the implementations in MATLAB and in C++.

HTA constructs	MATLAB syntax	C++ syntax
class	hta	HTA<type, dimension>
tuple	$(i_0,i_1,...,i_{n-1})$	Tuple<n> $(i_0,i_1,...,i_{n-1})$
range	low:step:high	Triplet(low, high, step)
tile index	{}	()
scalar index	()	[]
operations	+ - *	+ - *
constructor	hta(...)	HTA<type, dimension>::alloc(...)
transpose(h)	'	h.transpose()
map(h)	feval(func,h, ...)	h.map(op, ...)
reduce(h)	reduce(func,h, ...)	h.reduce(op, ...)

Table 3.1: Different syntax for HTA implementation in MATLAB and C++

HTAs were implemented as a MATLAB toolbox programmed in both C and MATLAB with invocations to MPI primitives. The MATLAB toolbox mechanism proved adequate to implement with reasonable efficiency and natural syntax all needed HTA operations except for the `forall` construct. However, we were able to develop an elegant implementation of all the codes we studied without `forall`. MATLAB also provides a native method interface called MEX, which allows functions to be implemented in languages like C and Fortran. The bulk of the HTA implementation is actually written in C and interfaces with MATLAB through MEX. Communication is implemented using the MPI [23] library. The cell array, which allows any content in each cell, is used in our HTA class to hold the tiles.

Our first MATLAB implementation of HTAs followed the client/server model in which the main thread is executed on a workstation and HTAs are stored and manipulated in a distributed system that operates as a co-processor. Although this approach facilitates program understanding, it requires too much communication between the workstation and the processors in the background parallel machine. We decided to change the implementation to follow the SPMD execution model mentioned earlier.

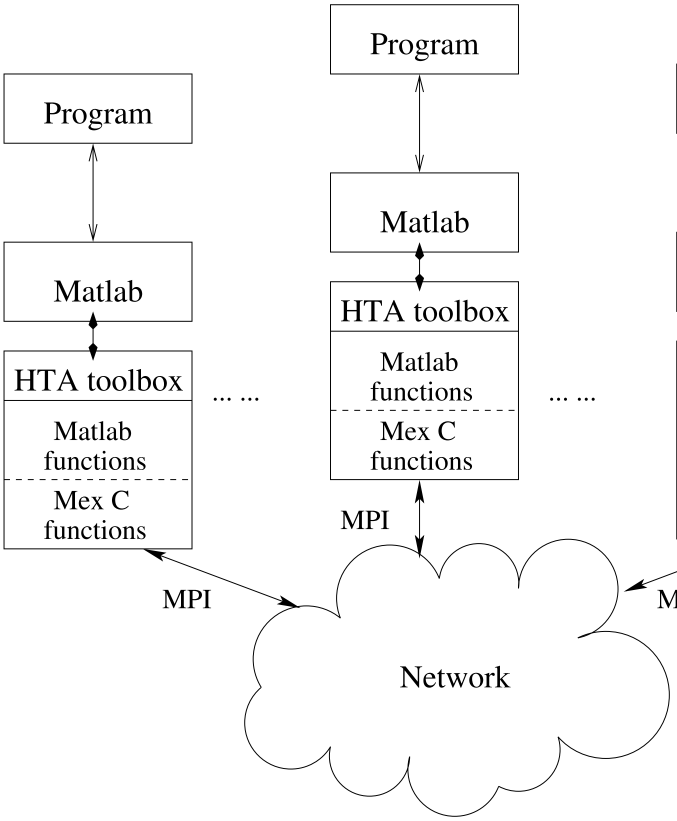


Figure 3.2: HTA implementation in MATLAB with SPMD execution model.

The second MATLAB implementation, shown in Figure 3.2, follows the SPMD execution model. The program is run on all the processors with MATLAB installed. Each processor invokes the HTA methods operating on the tiles it owns. The switch from the first implementation to the second one is simple. Those functions without communications are untouched. The functions with communications in the first implementation are always coupled, one for client side and one for the server side. In the new implementation, the function for the client side is simply removed, while the server side is kept with minor modifications. The lower layers of the HTA toolbox in Figure 3.2 take care of the communication requirements, while the higher layers implement the syntax expected by MATLAB users.

3.2.1 Constructor

By default, the constructor in MATLAB creates non-distributed HTAs, in which all the contents reside in the memory of the processor that requests the construction of the HTA. The constructor will distribute the top-level tiles of the HTA to build on a mesh of processors when the last argument of the call to the constructor is a vector. This vector describes the sizes of the mesh of processors on which to map the tiles. The distribution of the top-level tiles on the mesh is currently fixed to be block-cyclic.

Besides the different HTA constructions described in Section 2.2, a copy constructor and a HTA-like constructor are also implemented in MATLAB.

HTA provides a copy constructor that replicates an object. If we want to make a copy of HTA A into a new HTA B we can do it in two ways:

```
B = A;
B = hta(A);
```

The difference is that in the first case, B is not really a copy of A, but an alias, so there is no replication involved. Still, since MATLAB implements copy-on-write semantics, if either A or B happen to modify their contents, then a replication will take place and they

Operation	Function	Description
a + b	plus	Binary addition
a - b	minus	Binary subtraction
a * b	mtimes	Matrix multiplication
a .* b	times	Element-wise multiplication
a \ b	mldivide	Matrix left division
a / b	mrdivide	Matrix right division
a ./ b	rdivide	Right element-wise division
a ∧ b	mpower	Matrix power
a .∧ b	power	Element-wise power

Table 3.2: Overloaded MATLAB binary arithmetic operators

will become different HTAs; one of them with the original contents, and another one with the new values. The usage of the HTA copy constructor, as in the second line, guarantees that `B` is a separate independent HTA which has the same contents as `A`.

Sometimes we want to build an HTA that has the same shape and distribution as another one, but whose contents come from a different array or another HTA. The HTA-like constructor receives as first parameter the matrix or HTA from which the data are to be taken, and as second parameter the HTA that provides the frame for the structure and distribution of the HTA we want to build. The condition for the HTA-like constructor to be successful is that the size in terms of elements/scalars of the first and second arguments are the same in every dimension. So if we have the HTA `B`, which has a given tiled structure and distribution and which contains an array of 1000 × 1000 × 4 elements, an HTA with exactly the same structure and distribution can be built by executing

```
A = hta(rand(1000, 1000, 4), B);
```

where the contents of the new HTA `A` will be extracted from the 1000 × 1000 × 4 random matrix generated by `rand`.

Function	Description
uminus(-)	Unary minus
abs	Absolute value
max	Maximum value
min	Minimum value
sum	Reduction sum

Table 3.3: Overloaded MATLAB unary,binary and reduction functions

3.2.2 Overloaded operators

Table 3.2 enumerates the standard MATLAB binary arithmetic operators/functions that are overloaded in the HTA class. The output of the computation has always the same distribution as the left operand.

Table 3.3 enumerates the standard MATLAB unary, binary, and reduction functions overloaded by the class. The output HTA has the same distribution as the input one. In the case of reduction functions like `max` or `sum`, it is possible to provide a second argument that specifies the dimension along which we want the reduction to take place, as in MATLAB . The output HTA has each tile containing the result of the reduction performed in the corresponding tile in the input HTA. If no dimension is provided, the output HTA contains only one scalar in each tile, that corresponds to the reduction of the associated tile in the input HTA in every dimension.

Users can also define their own HTA operators/functions, which performs an operation on each tile of an HTA. They should define an HTA method that applies to corresponding tiles in two HTAs and generate a new tile with the output data. Such method is put in the hta toolbox and the content of the file will look like:

```
function r = myfun(a, b)
% HTA/myfun implementation
r = binaryOp(@myfun, a, b);
```

where `myfun` designates the pointer to the function as defined for standard MATLAB arrays. The HTA toolbox fencing `binaryOp` will take care of applying this function to the corresponding tiles in two HTAs to be operated. The user defined unary operator can be created in a similar fashion.

The definition of reduction functions is a bit more complex because we can choose to perform reductions along a specific dimension or on every dimension. Let us use as example for our explanation the implementation of the `sum` reduction function for HTAs:

```
function r = sum(h,ndim)
% HTA/sum
if nargin == 1
  ndim = -1;
end
r = unaryOp(@sum, h, ndim);
```

where `nargin` is a variable provided by MATLAB that specifies how many arguments the call to the function received. If this variable is 1, it means the user did not provide the argument `ndim`, which is the dimension along which to make the reduction. In this case we define it to be -1, which means the reduction will take place in every dimension, in such a way that the output will be a scalar with the addition of all the elements in the input array. If `nargin` is not one, the `ndim` provided by the user is passed to `unaryOp`, which will take care of performing the reduction in the specified dimension.

3.2.3 Strengths in the MATLAB implementation

One of the great advantages of using MATLAB to implement the HTA data type is the concise representation of linear algebra operations. For example, : for the triplet notation, $'$ for transpose, * for matrix multiply, $circshift$ for circular shift, *repmat* for replicating matrix, etc. When the HTA implementation overloads those operators and functions, it inherits the conciseness from MATLAB.

On the other hand, MATLAB provides a rich set of scientific operations which can be easily incorporated in the HTA toolbox. We have shown in Section 3.2.2 the simple and systematic way to add new operations into the HTA toolbox by adding two lines of code.

The MATLAB implementation also provides an interactive mode for the programmers. Although the client/server model suffers from performance bottlenecks, it is an ideal environment for translating existing MATLAB codes to serial or parallel HTA codes and for debugging in a sequential machine and in parallel. The two implementations, the client/server model and the SPMD model we described, have the same functionality. Consequently, a programmer can write the HTA code and debug, test on the client/server model interactively and finally run the correct code using the implementation with the SPMD model.

Further, the main data type matrix in MATLAB and the HTA data type are similar. The major operators and functions that support matrices are overloaded to support HTAs as well. Matrices can also be thought of as a leaf level HTAs. Therefore, the gap between the conventional data type in MATLAB and the HTA data type is small. This helps the programmer with the incremental translation from a MATLAB code with matrices to a code with the HTA data type.

The incorporation of HTAs in MATLAB produced an explicitly parallel programming extension of MATLAB that integrates seamlessly with the language. Most other parallel MATLAB extensions either make use of extraneous primitives (MultiMATLAB [45]) or do not allow explicit parallel programming (Matlab*P [30]). Also, the incorporation of HTAs gives MATLAB a mechanism to access and operate on tiles much more powerful than that provided by their native `cell arrays`.

3.2.4 Overhead in the MATLAB implementation

The main disadvantage of the implementation is that the immense overhead of the interpreted MATLAB limits the efficiency of many applications. The main sources of this overhead are:

1. *Excessive creation of temporary variables.* MATLAB creates temporaries to hold the partial results of expression, including the result of assignment instructions, which significantly slows down the programs.

2. *Frequent replication of data.* MATLAB passes parameters by value and assignment statements replicate the data. This way, our implementation with MATLAB must make a full copy of the LHS of any indexed HTA assignment operation. This constitutes a potential crippling overhead if copying happens in every assignment. MATLAB itself nevertheless does not suffer from this overhead because the indexed assignment operation is a built-in function to which arguments are passed by reference.

3. *Interpretation of instructions.* The overhead resulting from the interpretation of instructions is more pronounced when the computation relies mainly on scalar operations.

4. *Compiler's limitation.* The MATLAB JIT (Just In Time) compiler did not work on loops that contained vectorized and non-vectorized portions of code.

However, the sources of overheads are mostly due to implementation issues or the limitations in MATLAB, they are not inherent constraints of the HTA approach. To boost the performance of HTA code, we experimented on a C++ implementation of the HTA data type.

3.3 A C++ implementation: htalib

In the C++ implementation, HTAs are represented as composite objects with methods to operate on both distributed and sequential HTAs. Two communication layers are available: MPI and UPC. The implementation follows the SPMD execution model while the programming model is still single-threaded.

The core data structures of the htalib API fall into four categories:

1. *Logical index space.* Classes used to define index space and tiling of an HTA are `Tuple<N>`, an N-dimensional index value; `Triplet`, a 1-dimensional range with optional stride `(low:high:step)`; and `Region<N>`, an N-dimensional rectangular index space spanned by N triplets. Arithmetic, shift, and iterator functionality are implemented and compatible with the STL library. Instances of `Tuple<N>` and `Triplet` are values, i.e., once defined, their value cannot change.

2. *HTA.* Class `HTA<T,N>` defines an HTA with scalar elements of type `T` and `N` dimensions. The data type implements scalar access (`operator[]`), tile access (`operator()`) and built-in array operations, e.g., `transpose`, `permute`, `dpermute`, `reduce` that are described in earlier publications [10]. An HTA is part of a hierarchical structure of recursively composed HTAs.

3. *Machine mapping.* The machine mapping of an HTA specifies (i) where the HTA is allocated in a distributed system and (ii) the memory layout of the scalar data array underlying the HTA. The former aspect is captured by instances of class `Distribution` that specifies the *home location* of the scalar data for each of the tiles of an HTA. The latter aspect is represented by instances of class `MemoryMapping` that specify the layout (row-major across tiles, row-major per tile etc.), size and stride of the flat array data underlying the HTA.

   The machine mapping is accessed internally by the htalib, for example, to orchestrate implicit communication. The machine mapping is also available through the API of the library to facilitate direct access and communication of array data in case the programmer intends to bypass the access mechanisms provided by HTAs.

4. *Operator framework.* htalib provides a powerful operator framework following the design of the STL operator classes. This framework consists of routines that evaluate specific operators on HTAs and base classes that serve as a foundation for user-defined operators. A detailed discussion of the operator framework is given in [9].

3.3.1 Useful optimizations in htalib

Performance is one of the main goals of our C++ implementation. We optimized htalib in the following ways and effectively reduced the overhead incurred inside htalib.

1. *Automatic memory management.* HTAs are allocated through factory methods on the heap for automatic memory management. The methods return a handle which is assigned to a (stack allocated) variable. All accesses to the HTA occur through this handle, which itself is small in size and typically passed by value across procedure boundaries to avoid unnecessary copy overhead. Once all handles to an HTA disappear from the stack, the HTA and its related structures are automatically deleted from memory. This design permits sharing of sub-trees among HTAs and also precludes deallocation errors. Moreover, the temporary arrays that are for instance created during the partial evaluation of expressions, are handled through this mechanism and deleted automatically, as early as possible.

2. *Template class.* Template classes are used intensively in `htalib` to handle data with different types and dimensions. The template class provides both the flexibility and opportunities for optimizations at the compile time. The data type of an HTA can be of any type: a double, an integer, or any user defined data type. On the other hand, the dimension specified for template class places more work to be done at compiler time instead of runtime. For example, in HTA operations, the conformability check for HTAs with same dimensions is done by compiler since the operations are written for HTAs with same dimensions. Another example is data type such as `Tuple<DIM>`, can be allocated statically.

3. *Specialized methods.* Methods were optimized and whenever possible specialized for specific cases. For example, a specialized method that avoids multiplication by the stride is invoked when the data being accessed is known to be stored in consecutive locations.

```
htalib::async();
B(1:n)[0] = B(0:n-1)[d];
B(0:n-1)[d+1] = B(1:n)[1];
htalib::sync();
```

Figure 3.3: An example of relaxing sequential evaluation order to facilitate overlap of communication and computation.

4. *Inlined hot methods.* Inlining is performed to methods that are used frequently. For example, the tile access functions and scalar access functions are carefully inlined to reduce the overhead of function calls.

5. *Lazy evaluation.* In a HTA assignment where the RHS has more than one variables, htalib uses lazy evaluation to avoid or reduce the temporary variables generated from one or more binary operations. Then, if LHS and RHS have no data dependency, the assignment is directly evaluated on the LHS.

6. *Relaxation of serial evaluation semantics.* htalib provides a mechanism to temporarily relax the serial evaluation ordering, helping the overlapping of different communications and of communication with computation. For example, Figure 3.3 shows the boundary exchanges in the 1D Jacobi. As there is no data dependence among the assignments, both statements can proceed concurrently. This is achieved through the runtime calls to `async` and `sync`.

Chapter 4

Writing Tiled Algorithms with HTAs

In this section, we discuss how the HTA data type can help present the tiling and data layout in different algorithms. The algorithms discussed here are matrix matrix multiplication, matrix transposition, and LU factorization. For each example, we first describe the algorithm, and then show our HTA implementations with clean syntax. Finally, a summary highlights the nice features of HTA to simplify the programs.

4.1 Matrix matrix multiplication

Figure 4.1 shows the recursive blocked matrix matrix multiplication using HTAs. HTAs `A`, `B`, and `C` are declared and initialized outside the `mult` function. We only show the creation of `A` in Line 3. However, `B` and `C` are created in the similar fashion. The first parameter in `alloc` specifies two levels of tiling for `A`. The second parameter, an array of tuples, provides the sizes of the tiles for each level. The last parameter decides the data layout to be `TILE`. We show that the configuration of an HTA object is easy to manipulate. The levels of tiling, the size of tiles, and the data layout can be changed with minimal changes in the program. This enables an easy exploration of the search space for the parameters mentioned above.

In the recursive function `mult`, if the input HTA `A` reaches the leaf level, it calls a HTA wrapper function for mini-MMM code (Line 9) generated by ATLAS [46], which is optimized for smaller 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. The recursive nature of HTA data type facilitates the writing of recursive algorithms by providing the lower level

```
typedef HTA<double,2> HTA;

HTA A = HTA::alloc(2, ((ts,ts),
     (nt2,nt2), (nt1,nt1)), TILE);
... ...
void  mult(HTA A, HTA B, HTA C)
{
  if(A.level()==LEAF_LEVEL){
    ATLAS_mini_MMM(A, B, C);
  }else{
    for (int i = 0; i< A.size(0); i++)
      for (int k = 0; k < B.size(0); k++)
        for (int j = 0; j< C.size(0); j++)
          mult(A(i,k), B(k,j),C(i,j));
  }
}
```

Figure 4.1: Recursive blocked matrix multiplication using HTAs

tiles for the next level of recursion. Furthermore, the multi-dimensional tile indexing gives a precise indication of which range of data is selected in the computation.

4.2 Matrix transposition

The problem of matrix transposition is defined as follows. Given an $m \times n$ matrix stored in a certain layout, compute and store its transposition matrix of size $n \times m$ also in the same data layout. Figure 4.2 shows the recursive blocked matrix transpose algorithm using HTAs. HTA `A` and `B` are created in Line 2-3 with two levels of tiling. If the function reaches the leaf tiles, the scalars in each tile is transposed using the scalar access operator `[ ]` in Line 14. Otherwise, it recursively calls the transpose function on every tile of A and stores the result in the corresponding tiles of B. As shown from the code, the indexing of tiles and scalars is simple and straightforward without additional calculation.

```
typedef HTA<double,2> HTA;
HTA A = HTA::alloc(2, ((t1,t2),
    (n1,n2), (n3,n4)), ROW);
HTA B = HTA::alloc(2, ((t2,t1),
     (n2,n1), (n4,n3)), ROW);
... ...
void transpose (HTA A, HTA B){
  int M = A.size(0);
  int N = A.size(1);

  if(A.level()== LEAF_LEVEL){
    for(int i=0; i<M; i++)
      for(int j=0; j<N; j++)
        B[i,j]= A[j,i];

  }else{
    for(int i=0; i<M; i++)
 for(int j=0; j<N; j++)
        transpose(A(i,j), B(j,i));
  }
}
```

Figure 4.2: Recursive blocked matrix transpose using HTAs

4.3 LU factorization

In this section, we present the algorithm of blocked LU factorization with pivoting. To make a distinction of other types of LU algorithms discussed later, we call this algorithm *the iterative static LU algorithm*, since the partitions are made statically before factorization. The goal is to show the algorithm and different flavors of code representations with and without HTAs.

We first describe the right-looking LU algorithm with partial pivoting [44]. The algorithm factors an $M \times N$ matrix A into an $M \times M$ permutation matrix P and 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 an array of $m \times n$ tiles T_{ij} of size $b \times b$, where $0 \leq i \leq m-1$, $0 \leq j \leq n-1$. The algorithm iterates over the tiles in diagonal. In the kth iteration, we group the tiles into 9 parts:

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

where A_{11} is T_{kk} of order b, A_{00} contains all the tiles to the north and west of A_{11}. A_{22} contains all the tiles to the south and east of A_{11}. Figure 4.3 shows the groups of tiles for kth iteration. Matrices P, L, U are partitioned and grouped in a similar way. In the kth iteration, the algorithm performs the following steps.

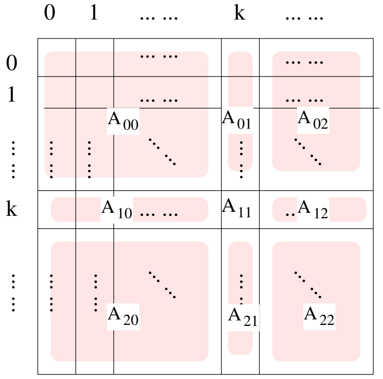


Figure 4.3: LU factorization in kth iteration

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 the triangular system $L_{11}U_{12} = A_{12}$.

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

In [25], Gustavson discussed the combination of tiled algorithms with a special data layout. He proposed a new data structure (NDS) which stores the matrix as a collection of submatrices with continuous data storage in each submatrices. Figure 4.4 shows the data structure for NDS. It is built from conventional arrays in Fortran with an additional dimension for tile indexing. Figure 4.5 shows a snapshot of the implementation using NDS for LU factorization. Here `A`, the NDS data format is implemented using a three dimensional array in Fortran so that the first two dimensions are local coordinates within a tile and the third dimension addresses linear tile index. We changed a few variable names to make the naming consistent with the algorithm description. We only present step 5 (update the trailing matrix using `nds_dgemm`) and its related statements.

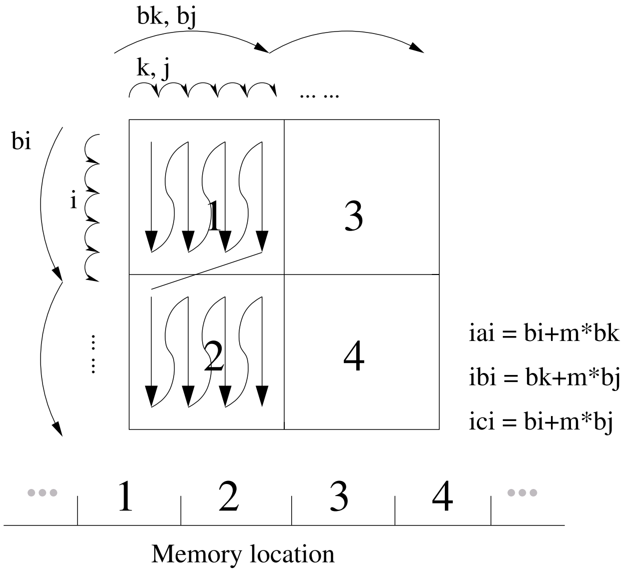


Figure 4.4: An illustration of NDS in [25]

Lines 4,7 and 12 are loops that iterate over tiles in different dimensions. The function `nds_dgemm` in step 5 is called in Line 21 inside the three loops to perform the update of the trailing matrix. These statements are necessary to represent the algorithm. However, more statements and variables are used in Figure 4.5 in order to deal with the management of tiles. Other than the scalar indices `i, j, k`, the linearized tile indices `iai, ibi, ici` are introduced and calculated in Line 9,19 and 20; the tile indices for each dimension `bi, bj, bk` are updated in Line 3,5,8,11 and 16. The code also keeps track of the size for each tile in Line 17 and 18.

In the overall implementation from [25], out of 51 lines of code, 12 statements are used to calculate the tile indexes; 11 statements are used to obtain the information for tile sizes. The situation is also true in the implementation of LAPACK[5], which follows the same algorithm but uses a column major layout. 29 operations are used to calculate the indexing. We will show the detailed analysis in Chapter 7. In summary, It requires non-trivial effort from programmer to keep indexing correct. On the other hand, the intricate indexing may obscure the kernel operations and the structure of the algorithms.

We also show the HTA implementation of the iterative static LU algorithm in Figure 4.6. The input and output parameters `A` and `ipiv` are one level HTAs. They are statically partitioned and passed to function `LU`. The `ipiv` has similar functionality as the permutation matrix P. Each step inside of the iteration is completed by one function call. The notion `hta_func` is a wrapper function for `func`, a basic BLAS or LAPACK library function. The wrapper functions allow the programmers to pass HTAs as parameters to the functions of such libraries. Furthermore, it moves some parameters (the data type of the array, the sizes of the array, the length of the leading dimensions, etc.) into the wrapper, since they can be easily obtained from the HTA variables. In this way, the programmer can focus more on the algorithm itself rather than the parameter details.

The wrapper functions for BLAS and LAPACK are implemented in the following way. If the HTA data layout is linear, we get the pointer to the HTA data with all other related information and call the library routine directly. If the HTA data layout is `TILE`, and the input HTA has more than one tile, we have to implement the blocked version. Some studies [7][25] mention the blocked implementation of common library routines. Inside of each block, the BLAS and LAPACK routines can be reused. Programmers also can easily

```
 subroutine dgefb(M, N, A, b, ipiv, info)
 ...
 bk = 0
 do k=0, N-1, b
   bj = bk
   ...
   do j = k+b, N-1, b
     bj = bj + 1
     ibi = bk+m*bj
      ...
     bi = bk
     do i= k+b, M-1, b
*
*    Update the trailing matrix
*
       bi = bi+1
       ms = b
       if(bi .eq. m-1) ms=m2
       iai = bi+m*bk
       ici = bi+m*bj
       call nds_dgemm(ms, ks, b, A(0,0,iai), b,
   &              A(0,0,ibi), b, A(0,0,ici), b)
     enddo
   enddo
   ...
 enddo
 return
 end
```

Figure 4.5: A snapshot of NDS implementation of iterative static LU algorithm extracted from [25]

write HTA wrapper functions themselves since the parameters such as data type, tile sizes and data layout can be easily derived from the HTA data type.

Compared to Figure 4.5, it demonstrates the clean syntax in the HTA implementation as follows. First, the use of multi-dimensional tile indexing avoids unnecessary variables and extra indexing calculations. Second, the access of a range of tiles can be expressed easily in HTA program using triplet notation. For example, $A(k : m, k)$ in Line 9 are tiles grouped by A_{11} and A_{21} in Figure 4.3. Such vectorized code eliminates the loops, thus makes the control flow of the program simpler. Finally, the information about the size of the tile, the memory mapping of the underlying array, is hided in the wrapper functions provided by

```
void LU(HTA<double,2>  A, HTA<int,1> ipiv)
{
 int m = A.size(0)-1;
 int n = A.size(1)-1;
 int k;

 for( k=0; k<=n; k++){
   // step 1. Factor
   hta_getf2(A(k:m, k), ipiv(k));

   if(k<n){
     // step 2. Permute
     hta_laswp(A(k:m, k+1:n), ipiv(k));

     // Step 3. Permute backward*/
     hta_laswp(A(k:m, 0:k-1), ipiv(k));

     // step 4. Solve the triangular system
     hta_trsm(Right, Upper, NoTrans, Unit,
                   1.0, A(k), A(k, k+1:n));
   }
   if(k>1)
     // step 5.  Update trailing matrix
     hta_gemm(NoTrans, NoTrans, -1.0, A(k+1:m,k),
              A(k, k+1:n), 1.0, A(k+1:m, k+1:n));

 }
}
```

Figure 4.6: HTA implementation of iterative static LU factorization algorithm

htalib. The interfaces with less parameters become simpler.

4.4 Summary

In summary, we show how to write HTA programs for recursive and iterative algorithms. The recursive nature of HTA makes the implementation easy by going down one level of tiling. Furthermore, the multi-dimensional tile indexing is convenient and concise. With the capability of selecting a range of tiles, loops can be simply avoided. As showed through the three examples, the HTA programs are always implemented at the tile level which matches the description the tiled algorithm. It also facilitates performance tuning to search for the best parameters of tile sizes, the number of levels of tiling, the data layout, etc. with minimal changes of the code.

Chapter 5

Dynamic Partitioning

We have shown several examples of HTA programs with *static partitioning*, which means that the tiling is fixed at creation time and it does not change throughout the program. However, many algorithms such as cache oblivious algorithms [22] or algorithms derived from FLAME [7] require changing the tiling layout on a given matrix several times. This motivates the extension of the HTA data type with a new language construct, *dynamic partitioning*, which allows programmers to change the tiling structure of HTAs on the fly. With dynamic partitioning, such algorithms can be expressed in a simple and direct manner.

In this chapter, we first introduce the syntax and semantics of the dynamic partitioning. Then, we show how it is implemented in htalib, the runtime system for HTA. Next, several examples from algorithms generated by FLAME and from cache oblivious algorithms are shown to illustrate the power of the dynamic partitioning. And we conclude at the end.

5.1 Syntax and semantics

We first describe the partition numbering scheme we used in the syntax of dynamic partitioning. After an HTA is created, the lines that separate the tiles are called *partition lines*. These partition lines include the implicit ones at the position 0 and after the last element of the array in each dimension, as well as the explicit partition lines specified by the user. We number the partition lines from 0 in ascending order. For example, in Figure 5.1(a), the dotted lines (implicit partition lines) along with the solid lines are numbered from 0 to 2.

A set of partition lines, one along in each dimension, is called a *partition*. We identify

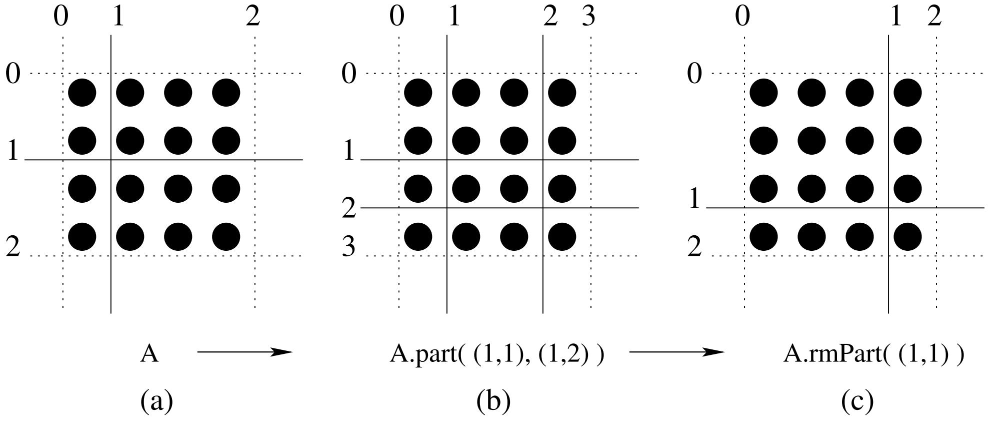


Figure 5.1: Illustration of dynamic partitioning in HTAs

a partition by a tuple with the number of partition line it consists of in each dimension, or the special symbol `NONE` for those dimensions along which no partition line belongs to the partition. Since in most algorithms the change of the tiling structure involves only one partition at a time, we provide an interface to add or remove one partition in one operation. We denote P_i the partition number in dimension i and p_i the absolute position of P_i, while Q_i is the the new partition line in dimension i and q_i is the absolute position of Q_i. o_i is the offset of the new partition with respect to P_i. Then to add a partition, we have

$$\text{Partition after } (P_1, P_2, ..., P_n) \text{ with offset } (o_1, o_2, ..., o_n) \Rightarrow$$

$$Q_i = \begin{cases} NONE & \text{if } P_i = NONE \\ P_i + 1 & \text{if } o_i \geq 0 \\ P_i & \text{if } o_i < 0 \end{cases}$$

$$q_i = \begin{cases} NONE & \text{if } P_i = NONE \\ p_i + o_i & \text{Otherwise} \end{cases}$$

To remove an existing partition, only the tuple of partition numbers for that partition should be provided. In `htalib`, HTA dynamic partitioning mechanism is based on two methods: `part` for adding a partition and `rmPart` for removing a partition. Their syntax in C++ is as follows:

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

The first parameter `sourcePartition` in method `part` is the already existing partition relative to which the partition lines of the new partition will be added. No partition lines will be added for those dimensions that do not participate in the `sourcePartition`, i.e., those labeled as `NONE`. `DIM` is the same as in HTA, the number of dimensions of the array. The second parameter `offset` states the relative offset length in each dimension with respect to the `sourcePartition` of the new partition to create, measured in array elements. Both positive and negative offsets are supported, so that partition lines can be added after or before the target partition line in every dimension.

Figure 5.1(b) gives one example. A request is made of create a new partition in a location relative to partition `(1,1)`, namely that with relative offset `(1,2)`. Notice that since partition lines are numbered consequently the new partition can be identified as `(2,2)` and now the implicit partition lines at the end of the array in each dimension have number 3.

When a partition is to be removed, all is needed is an invocation to `rmPart` with the tuple identifying it. Only explicit partition lines can be removed. Calling `rmPart` with no arguments removes all the explicit partitions. Figure 5.1(c) illustrates the removal of partition `(1,1)`. Notice that the elimination of a partition line in a dimension renumbers the partition lines that followed it to accommodate to the new situation.

By adopting this partition numbering scheme, programmers can perform dynamic partitioning at the tile level without needing to know the absolute positions in the array of the partition lines.

5.1.1 Empty tiles

One of two policies can be followed when a new partition line is requested with offset 0 with respect to an already existing partition line: ignore the request (i.e. both are the same partition line) or indeed add a new partition line, giving place to tiles that are empty in between the original line and the new one. We have decided to follow the second approach because empty tiles simplify the writing of array algorithms. In general, the simplification occurs in the initial and final steps of an algorithm, which normally require special consideration. With empty tiles, they are automatically included in the general case. We show such examples in Section 5.2.

Operations and assignment with empty tiles in HTAs still follow the rule of conformability. That is, empty tiles cannot operate with or be assigned non-empty tiles, since the sizes are not equal. The methods that take empty tiles as input will immediately exit since no element will be used.

5.1.2 Implementation

Dynamic partitioning is implemented in htalib as methods of the HTA class. After the programmer specifies the partition to add or remove, the tiling structure is recalculated. Then, the information associated to new tiles such as tile size, memory mapping, etc. is updated. htalib records the types of HTA and use a different data structure to represent the tiling topology. If a regular HTA changes to irregular HTA after dynamic partitioning, the data structure for tiling topology is automatically changed. Furthermore, the dynamic partitioning does not change the number of levels of current HTA except that when it is performed on a leaf HTA with level 0, the HTA level changes to 1.

For different data layouts, dynamic partitioning implies different overhead. For row and column major layouts, it only needs to change the tiling structure of the top level. However, for Tile data layout, rearranging the data so that it is stored continuously in new tiles can be a significant cost. The programmer is advised to use row or column major when dynamic partitioning is used frequently.

5.2 Examples

In this section, we study two types of algorithms: algorithms derived from FLAME [7] and cache oblivious algorithms [22]. We then present how the new language construct helps express such algorithms.

5.2.1 Representation of FLAME algorithms

Iterative dynamic LU factorization

The FLAME project [7] uses a systematic approach to the derivation of families of high-performance algorithms for a large set of frequently encountered dense linear algebra algorithms or operations. The algorithms that result from the derivation process present themselves a very rigid format. We illustrate this format in Figure 5.2, which depicts the algorithm for blocked LU factorization with partial pivoting, denoted as *iterative dynamic LU algorithm.* The algorithm is divided into three parts by two double horizontal lines. In the first part, arrays A and p are initialized with empty tiles to the north and west. At the top of the `while` loop body, A and p are updated by adding new partitions to the south and east (the latter for A only). The T, B, L, R in subscripts stand for Top, Bottom, Left, and Right respectively. The distance between the new partition (thin line) and the old partition (thick line) is b for both dimensions. The update of the newly created tiles is performed in the second part. For the LU factorization, it consists in the same five steps mentioned in Chapter 4. The third part removes the old partitions (shown now in thin line) for the current iteration and the remained partition becomes the old one in the next iteration. Finally, the desired result has been computed when A_{TL} encompasses all of A and the loop condition becomes false. Here $n(A)$ returns the number of columns of A.

```
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 $A_{TL}$ is $0 \times 0$, $p_T$ has 0 elements
while $n(A_{TL}) < n(A)$ do
    Repartition

        $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$,
        $\left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$
        where $A11$ is $b \times b$ , $p_1$ is $b \times b$

    $[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), p_1] := LU_{PIV} \left(\begin{array}{c} A_{11} \\ \hline A_{12} \end{array}\right)$
    $\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right) := P(p_1) \left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right)$
    $A_{12} := L_{11}^{-1} A_{12}$
    $A_{22} := A_{22} - A_{21} A_{12}$

    Continue with

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

Figure 5.2: FLAME's iterative dynamic LU algorithm

```
void lu(HTA<double,2> A, HTA<int,1> p,int b)
{

  A.part((0,0), (0,0));
  p.part((0), (0));

  while( A(0,0).lsize(1) < A.lsize(1) ){
    A.part((1,1), (b,b));
    p.part((1), (b));

    hta_getf2(A(1:2,1), p(1));
    hta_laswp(A(1:2,0), p(1));
    hta_laswp(A(1:2,2), p(1));
    hta_trsm(Right, Upper, NoTrans,
             Unit, 1.0, A(1,1),A(1,2));
    hta_gemm(NoTrans, NoTrans, -1.0,
             A(2,1), A(1,2), 1.0, A(2,2));

    A.rmPart((1,1));
    p.rmPart((1));
  }
 }
```

Figure 5.3: HTA implementation of the iterative dynamic LU algorithm in Figure 5.2

In summary, the first part and the third part, which dictate how the partition sweeps through the matrices, share the common actions *partition*, *repartition*, and *continue with* across different algorithms. It is how the tiles in matrices are updated in the second part that differ.

The actions in the first and third part can be represented using dynamic partitioning in a straight forward way. Using `part` for both *partition* and *repartition* and `rmPart` for *continue with*, this is a one to one correspondence from the actions described in algorithms to the HTA methods in the code as shown in Figure 5.3. Here HTA `A` and `p` are HTAs with a single tile when function `lu` is called. Also, `lsize` is an HTA method that returns a tuple with the number of elements in each dimension of the corresponding HTA. All the functions used in the second part in Figure 5.3 are the same as in Figure 4.6 in Chapter 4. As we see, the HTA program does not use as many variables as those in the algorithm description, since tiles are easy to access in HTAs through tile indexing, either for a single tile or a range of tiles.

The triangular Sylvester equation

Another example is triangular Sylvester equation [6]. Sylvester equation has numerous applications in control theory, signal processing, filtering, etc. It solves the equation

$$AX + XB = C \tag{5.1}$$

where A is an $m \times m$ matrix, B is $n \times n$, C and X are $m \times n$ and X is the soughtafter solution. We use

$$X = \Omega(A, B, C)$$

to represent the solution to the equation. In particular, we focus on the triangular case, where both A and B are upper triangular matrices. The solution of the triangular case arises as an intermediate subproblem in the Sylvester equation solver described in [6].

```
Algorithm: Solve $X = \Omega(A, B, C)$ where $X$ satisfies $AX + XB = C$
Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array}\right)$ , $B \rightarrow \left(\begin{array}{c|c} B_{TL} & B_{TR} \\ \hline 0 & B_{BR} \end{array}\right)$ , $C \rightarrow \left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right)$ ,
    where $A_{BR}$ , $B_{TL}$ and $C_{BL}$ are $0 \times 0$,
while $C_{TR}$ is not $0 \times 0$ do
    Determine block sizes $b_m$ and $b_n$
    Repartition

        $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline 0 & A_{11} & A_{12} \\ \hline 0 & 0 & A_{22} \end{array}\right)$,
        $\left(\begin{array}{c|c} B_{TL} & B_{TR} \\ \hline 0 & B_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} B_{00} & B_{01} & B_{02} \\ \hline 0 & B_{11} & B_{12} \\ \hline 0 & 0 & B_{22} \end{array}\right)$,
        $\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right)$
        where $A_{11}$ is $b_m \times b_m$ , $B_{11}$ is $b_n \times b_n$ , $C_{11}$ is $b_m \times b_n$
    ================================================================
    $C_{10} := \Omega(A_{11}, B_{00}, C_{10})$
    $C_{11} := C_{11} - C_{10}B_{01}$
    $C_{21} := \Omega(A_{22}, B_{11}, C_{21})$
    $C_{11} := C_{11} - A_{12}C_{21}$
    $C_{11} := \Omega(A_{11}, B_{11}, C_{11})$
    $C_{00} := C_{00} - A_{01}C_{10}$
    $C_{01} := C_{01} - A_{01}C_{11}$
    $C_{01} := C_{01} - A_{02}C_{21}$
    $C_{12} := C_{12} - C_{10}B_{02}$
    $C_{12} := C_{12} - C_{11}B_{12}$
    $C_{22} := C_{22} - C_{21}B_{12}$
    ================================================================
    Continue with

        $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline 0 & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline 0 & A_{11} & A_{12} \\ \hline 0 & 0 & A_{22} \end{array}\right)$,
        $\left(\begin{array}{c|c} B_{TL} & B_{TR} \\ \hline 0 & B_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} B_{00} & B_{01} & B_{02} \\ \hline 0 & B_{11} & B_{12} \\ \hline 0 & 0 & B_{22} \end{array}\right)$,
        $\left(\begin{array}{c|c} C_{TL} & C_{TR} \\ \hline C_{BL} & C_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} C_{00} & C_{01} & C_{02} \\ \hline C_{10} & C_{11} & C_{12} \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right)$
endwhile
```

Figure 5.4: The algorithm to solve the triangular Sylvester equation in FLAME [39].

FLAME derives a family of blocked algorithms[39] and Figure 5.4 shows one of them. The result of X is stored in C. The algorithm is a hybrid of iterative and recursive algorithms. Note that the partitioning of A, B, and C differs. A's new partition is added to the top-left tile; B's new partition is added to the bottom-right tile; and C's new partition is added to the top-right tile.

The HTA implementation is presented in Figure 5.5. The parameters `bm_algs` and `bn_algs` are arrays of absolute tile sizes along the first and second dimensions. THey describe the size of new tiles created (`A(1,1)`, `B(1,1)`, and `C(1,1)`). `nblks` is the number of levels of tilings to perform. The HTAs are partitioned step by step in the `while` loop (line 14) for a certain level. If the current level is not the leaf level, then `Syl` is performed recursively on the corresponding tiles of `A`, `B` and `C` (line 20, 22, 28). When leaf level is reached, `Syl_base`, a library routine for the base case, is called (line 6).

We show In line 17-19, the new partitions are added easily using the dynamic partitioning construct. Since the offsets can be either positive numbers or negative numbers, the new partition can be positioned in different direction with respect to the old partition. In line 45-47, all the partitions added in the function are removed in order to keep the structures of `A`, `B` and `C` unchanged after the recursive function call.

5.2.2 Representation of cache oblivious algorithms

Cache oblivious algorithms [22] are usually recursive algorithms with a divide-and-conquer strategy. When this strategy applies to matrices, it implies partitioning them into submatrices and performing the operations on each submatrix recursively.

Matrix multiplication

The cache oblivious matrix multiplication algorithm for a $m \times m$ matrix A and a $m \times m$ matrix B is described in [49]. It uses the divide-and-conquer strategy to partition in half all three dimensions. Then the partitioned submatrices are multiplied recursively. Bilardi

```
typedef HTA<double,2> HTA;
int * bm_algs, *bn_algs, nblks;
void Syl(HTA A, HTA B, HTA C, int n)
{
  if(n ==0){
     Syl_base(A, B, C);
  return;
  }
  int bm, bn;
  A.part((1,1), (0,0));
  B.part((0,0), (0,0));
  C.part((1,0), (0,0));

  while(B(1,1).size(0)!=0 || A(0,0).size(0)!=0){
    bm = min(bm_algs[nblks-n], A(0,0).size(0));
    bn = min(bn_algs[nblks-n], B(1,1).size(0));
    A.part((1,1), (-bm, -bm));
    B.part((1,1), (bn, bn));
    C.part((1,1), (-bm, bn));

    Syl(A(1,1), B(0,0), C(1,0), n-1);

    Syl(A(2,2), B(1,1), C(2,1), n-1);

    HTA_Gemm( NoTrans, NoTrans, -1.0, A(1,2), C(2,1), 1.0, C(1,1) );

    HTA_Gemm( NoTrans, NoTrans, -1.0, C(1,0), B(0,1), 1.0, C(1,1) );

    Syl( A(1,1), B(1,1), C(1,1), n-1);

    HTA_Gemm( NoTrans, NoTrans, -1.0, A(0,1), C(1,0), 1.0, C(0,0) );

    HTA_Gemm( NoTrans, NoTrans, -1.0, A(0,1), C(1,1), 1.0, C(0,1) );

    HTA_Gemm( NoTrans, NoTrans, -1.0, A(0,2), C(2,1), 1.0, C(0,1) );

    HTA_Gemm( NoTrans, NoTrans, -1.0, C(2,1), B(1,2), 1.0, C(2,2) );

    HTA_Gemm( NoTrans, NoTrans, -1.0, C(1,0), B(0,2), 1.0, C(1,2) );

    HTA_Gemm( NoTrans, NoTrans, -1.0, C(1,1), B(1,2), 1.0, C(1,2) );

    A.rmPart((2,2));
    B.rmPart((1,1));
    C.rmPart((2,1));
  }
  A.rmPart();
  B.rmPart();
  C.rmPart();
  return;
}
```

Figure 5.5: The HTA implementation for the triangular Sylvester equation.

et al. [12] pointed out it is possible to optimize memory hierarchy performance by using a Cray code order to schedule the 8 subsequent recursive calls so that the submatrix shared between consecutive calls is maximum. The algorithm with such a sequence is illustrated in Figure 5.6. The tiles shared in the recursive call $C = A \times B$ is maximum for C (4 cases), medium for A (2 cases) and minimum for B (1 case).

The HTA implementation is shown in Figure 5.7. Input parameters `A`, `B` and `C` are HTAs with a single tile of size $m \times m$. Lines 8-9 show the dynamic partitioning to create submatrices. Line 21-23 removes all the partitions to restore the HTAs `A`, `B` and `C` as they are at the beginning of the recursion. Such restoration guarantees every recursive function call is performed on HTAs with a single tile. Also, removing the partitions at the end of recursion deallocates the space for tiling structure in a timely fashion. Suppose the space for tiling structure of one tile is t, and the recursion level is r (starting from 0). The maximum space allocated for tiling structures in Figure 5.7 is

$$3t(1 + \sum_{i=1}^{r} 4) = 3t(1 + 4r)$$

If the partitions are not removed from the end and assume the code still works with some modifications, the maximum space allocated is equal to the pre-partitioned HTAs with r levels of tiling. It is

$$3t(1 + \sum_{i=1}^{r} 4^i) = t(4^{r+1} - 1)$$

Apparently, it requires much more space for not removing partitions at the end.

LU factorization

A more complicated example of cache oblivious algorithm is the recursive LU factorization proposed by Toledo [44]. The algorithm, denoted as *recursive LU algorithm* is shown in Figure 5.8, with matrices A, p being the same as in the one in FLAME.

In this algorithm, if A has more than one column, it is partitioned in half in each di-

```
**Algorithm:** $C = A \times B$
if $A$ and $B$ are scalar
    $C = A * B$
else
        $A \rightarrow \left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right)$
        $B \rightarrow \left(\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right)$
        $C \rightarrow \left(\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right)$
        $C_{00} := A_{00} \times B_{00} + A_{01} \times B_{10}$
        $C_{01} := A_{01} \times B_{11} + A_{00} \times B_{01}$
        $C_{11} := A_{10} \times B_{01} + A_{11} \times B_{11}$
        $C_{10} := A_{11} \times B_{10} + A_{10} \times B_{00}$
endif
```

Figure 5.6: Cache oblivious matrix multiplication algorithm [49]

```
typedef HTA<double, 2> HTA;
void co_mult(HTA A, HTA B, HTA C)
{
  int m = A.lsize(0);
  if(m ==1){
     C[0,0]=A[0,0]*B[0,0];
  }else{
     A.part((0,0),(m/2,m/2));
     B.part((0,0),(m/2,m/2));
     C.part((0,0),(m/2,m/2));

     co_mult(A(0,0), B(0,0), C(0,0));
     co_mult(A(0,1), B(1,0), C(0,0));
     co_mult(A(0,1), B(1,1), C(0,1));
     co_mult(A(0,0), B(0,1), C(0,1));
     co_mult(A(1,0), B(0,1), C(1,1));
     co_mult(A(1,1), B(1,1), C(1,1));
     co_mult(A(1,0), B(0,0), C(1,0));
     co_mult(A(1,1), B(1,0), C(1,0));

     A.rmPart();
     B.rmPart();
     C.rmPart();
   }
 }
```

Figure 5.7: HTA implementation of cache oblivious matrix multiplication [49]

mension and the factorization is performed recursively on the left half until a single column is reached. The factorization of a single column takes place in function LU_{INNER}. When the algorithm returns from the recursive call in (*), the factorization for the left half of the portion of A considered in the current invocation of the algorithm has already been completed, and it proceeds to apply the appropriate permutation and updates to the right half of the matrix. Then another recursive invocation in (**) 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 to be returned.

One of the challenges to implement the algorithm is how to perform the partition action. If we 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 recursion. Once we reach a single column, the number of horizontal partition lines we added will be equal to the number of recursions performed in (*). However, the horizontal partition lines will not be used until the algorithm goes to solve the triangular system. Consequently, adding the horizontal partition line at the beginning only complicates the tiling data structure and indexing for the column panels. We show the slightly modified algorithm in Figure 5.9. The statements (1),(2) show the partition lines are added when it is necessary. Statement (3) removes all the partitions added for the current recursion level so that it restores the tiling structure of HTAs as they were passed to the function originally.

Figure 5.10 shows our implementation of this cache oblivious LU algorithm. It first partitions A vertically in line 8 . In line 11, the left half of A is factorized recursively. The horizontal partition is added in line 14. At the end of the function, in line 25-26, again, we remove all the partitions created in the current recursion for the reasons discussed above.

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

Figure 5.8: Toledo's recursive LU algorithm [44]

```
**Algorithm:** $[A,p] :=\ LU_{REC}(\mathrm{A})$
**if** $n(A) = 1 \quad [A,p] = LU_{INNER}(A)$
**else**
  $A \rightarrow \left(\begin{array}{c|c} A_0 & A_1 \end{array}\right) \qquad (1)$
  $p \rightarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \end{array}\right)$
   **where** $A_0$ is $n/2 \times n/2$,
       $p_0$ has $n/2$ elements
   $[A_0, p_0] := LU_{REC}(A_0)$
   $A_1 := P(p_0)(A_1)$
   $\left(\begin{array}{c|c} A_0 & A_1 \end{array}\right) \rightarrow \left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right) \qquad (2)$
     **where** $A_{00}$ is $m/2 \times m/2$
   $A_{01} := L_{00}^{-1} A_{01}$
   $A_{11} := A_{11} - A_{10} A_{01}$
   $[A_{11}, p_1] := LU_{REC}(A_{11})$
   $A_{10} := P(p_1)(A_{10})$
   $A \leftarrow \left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right) \qquad (3)$
   $p \leftarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \end{array}\right)$
**endif**
```

Figure 5.9: Modified recursive LU algorithm

```
void lu(HTA<double,2> A, HTA<int,1> p)
{
  int m = A.lsize(0);
  if( m <= 1) {
     lu_inner (A, p);
  } else {
    int b = m/2;
    A.part(NONE,0), (NONE,b));
    p.part((0), (b));

    lu(A(0,0), p(0));
    hta_laswp(A(0,1), p(0));

    A.part((0,NONE),(b,NONE));

    hta_trsm(Right, Upper,NoTrans,
       Unit, 1.0, A(0,0), A(0,1));
    hta_gemm(NoTrans, NoTrans, -1.0,
      A(1,0), A(0,1), 1.0, A(1,1));

    lu(A(1,1), p(1));
    hta_laswp(A(1,0), p(1));
    p(1) += b;

    A.rmPart();
    p.rmPart();
  }
}
```

Figure 5.10: HTA implementation of recursive LU factorization [44]

5.3 Summary

Dynamic partitioning helps to express the locality algorithms in a straightforward way. Compared with static partitioning, it always partitions the current level as needed. It uses simpler indexes, only constants 0, 1, and 2 for each dimension. By contrast, the code generated from static partitioning such as LU code in Figure 4.6, usually use variables for indexing. For example, in Figure 4.6, line 23, `A(k+1:m, k+1:n)` is used to index the same range of data as `A(1,1)` in Figure 5.10, line 19. From the implementation point of view, it also saves allocation space since it only allocates 3×3 tiles instead of $(m-k) \times (m-k)$ tiles at any point of program. The use of method `rmPart()` also helps to limit the footprint of HTA in the memory because partitions, along with their associated data structures, are deleted. Nevertheless, programmers should use dynamic partitioning properly by choosing correct data layouts. Row and column major layouts only modify the top level of tiling structure. Hence the overhead is relatively low. However, the dynamic partitioning for Tile layout indicates data rearrangement. It should be used carefully so that the incurred overhead can be amortized by the amount of computation.

Chapter 6

Overlapped Tiling

6.1 Introduction

Partial Differential Equation (PDE) solvers constitute a large fraction of scientific computations such as heat diffusion, electro-magnetics, and fluid dynamics. They often use iterative finite differencing techniques which 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. A simple example of a stencil computation is Jacobi iterations shown in Figure 6.1.

```
double A[N], B[N];
while(!converged){
  for(int i=1; i<N-1; i++)
    A[i] = 0.5 * (B[i-1]+ B[i+1]);
  ... ...
}
```

Figure 6.1: 1D Jacobi iterations

```
HTA<double,1> A, B;
double S = 0.125;
Seq tiling = ((n),(d+2));
A=HTA<double,1>::alloc(1, tiling, TILE);

while (!converged) {
  // boundary exchange
  B(1:n)[0] = B(0:n-1)[d];
  B(0:n-1)[d+1] = B(1:n)[1];
  // stencil computation
  A()[1:d] = S*(B()[2:d+1] + B()[0:d-1]);
  ...
}
```

Figure 6.2: 1D Jacobi iterations using HTA

Tiling [32] is often used to optimize stencil codes for parallelism and data locality. When tiling is used to distribute the data in stencil programs onto different processors, shadow regions or ghost regions are created either implicitly by a compiler or explicitly by the programmers. The purpose of using shadow regions is to accommodate neighboring nonlocal elements needed by the computations. Shadow regions are updated through communications when the values they reflect are changed remotely.

Compilers such as the Fortran D compiler [29] and the HPF compiler [28] try to parallelize stencil programs and add shadow region manipulations automatically. This requires non-trivial dependence analysis and loop transformations. Unfortunately, compiler technology is not always capable of generating code that matches the performance of hand-written code.

On the other hand, programmers can handle explicitly the shadow regions and have direct control on the performance. However, the stencil code, tiled and extended with the shadow region manipulations can be cumbersome. For example, the indexing range of the array is different from that of the sequential codes in order to accommodate tiling and the addition of the shadow regions. Figure 6.2 shows the HTA program for 1D Jacobi. It can be used sequentially or in parallel if the distribution is specified. HTA `A`, `B` have n tiles each with d+2 elements among which, 2 elements are used in shadow regions. First, statements for boundary exchanges have to be added (lines 8,9). Then, the indexing in line 11 `A()[1:d]` is used instead of `A` since the elements in the shadow regions have to be excluded.

We propose an approach that falls in between those of the compiler and the explicit manipulation. We propose a new language construct called *overlapped tiling* to facilitate the programming of stencil codes. This construct can be used both in sequential and parallel. It is an extension of our Hierarchically Tiled Array (HTA) data type [10], which provides a natural representation for tiled codes with high degree of parallelism and locality. We rely on the user to specify the shadow regions through the overlapped tiling construct. This high level abstraction helps our HTA runtime system automate the process of managing the defined shadow regions.

The code written in HTA extended with the overlapped tiling construct, is simpler than what can be written in C or C++ and consistent with the algorithms for stencil codes because the shadow regions are transparent to the programmers. The implementation in the runtime system is simple, since no data dependence analysis is needed. Our experiments with non-trivial stencil computation benchmarks show that it greatly improves readability with a negligible effect on performance.

The organization of this chapter is as follows. First, we present the motivating example for the need of the overlapped tiling. Second, the syntax and semantics of the overlapped tiling are introduced, especially the shadow region consistency and its implementation in htalib. Then, we demonstrate the improvement of productivity through the comparison of HTA codes with and without the overlapped tiling. Finally we summarize the pros and cons of the overlapped tiling.

6.2 A motivating example

Tiling[32] combines strip-mining and loop permutation to reorder the computations in an iteration space in order to exploit data locality and/or enable effective parallelization.

In the sequential case, tiling helps to reduce the cache misses and increase data reuse for stencil computations. For example, in 2D version of a simple Jacobi kernel, as shown in Figure 6.3. Three adjacent rows of array B are accessed together. If N, the size of second dimension, is very large, the cache line from one row is evicted before it is read again. We denote C the number of elements in one cache line, then the total number of cache misses for reading B is

$$\frac{N}{C} \times 3(M-2).$$

If the tiling is performed along the second dimension so that every three rows in one tile fit into the cache. Then total number of cache misses for reading B tile by tile is

$$\frac{N}{C} \times M.$$

As we see, stencil computation without tiling leads to more cache misses. Tiling helps even more if further neighbors of `B`, `B[i-2][j]` and `B[i+2][j]` for instance, are accessed together. Another situation that tiling is more effective is on arrays with more dimensions [27].

```
double A[M][N}, B[M][N];
while(!converged){
  for(int i=1; i<M-1; i++)
    for(int j=1; j<N-1; j++)
      A[i][j] = 0.25 * ( B[i-1][j]+ B[i+1][j]
                        +B[i][j-1]+ B[i][j+1]);
  ... ...
}
```

Figure 6.3: 2D Jacobi iterations in C++

In the parallel case, the fact that the computation for each point in the array depends on its neighbors implies that the computations associated to the data in the border of each tile require data from the neighboring tiles. As a result, stencils parallelized in distributed memory environments usually resort to the usage of ghost or shadow regions.

Stencils usually operate repetitively within a time-step loop, updating the array corresponding to the next iteration with the results of the array corresponding to the current one. As a result, shadow regions also help parallelization by isolating the changes in the input tiles. This justifies their usage also in shared memory environments, and even in sequential codes that update the input tile before proceeding to process the next one in order to further exploit data locality.

Unfortunately, the periodic update also implies that shadow regions need to be updated with the new values for the input array before next iteration of the time-step loop begins, which complicates their management. We illustrate the complexity of this process with the `comm3` routine of the MG NAS MPI benchmark, which performs a multi-grid computation on a discrete 3D grid whose boundaries are periodic. This routine, shown in Figure 6.4 is responsible for the update of the shadow regions in most of the stencil routines of this benchmark. It first checks whether the update is really needed (line 7) and then it proceeds to perform the exchange in each of the three dimensions of the grid. It also has to handle

```
subroutine comm3(u,n1,n2,n3,kk)

integer n1, n2, n3, kk
double precision u(n1,n2,n3)
integer axis

if( .not. dead(kk) )then
  do  axis = 1, 3
    if( nprocs .ne. 1) then
      call ready( axis, -1 )
      call ready( axis, +1 )
      call give3( axis, +1, u, n1, n2, n3, kk )
      call give3( axis, -1, u, n1, n2, n3, kk )
      call take3( axis, -1, u, n1, n2, n3 )
      call take3( axis, +1, u, n1, n2, n3 )
    else
      call comm1p( axis, u, n1, n2, n3, kk )
    endif
  enddo
else
  call zero3(u,n1,n2,n3)
endif
return
end
```

Figure 6.4: `comm3` routine in Fortran from MG NAS benchmark

Versions	comm3	comm1p	Inversion	Residual
MPI	193	134	24	23
CAF	168	132	25	23
OpenMP	29	0	27	26

Table 6.1: Summary of SLOCS for communications and computations in different versions of MG

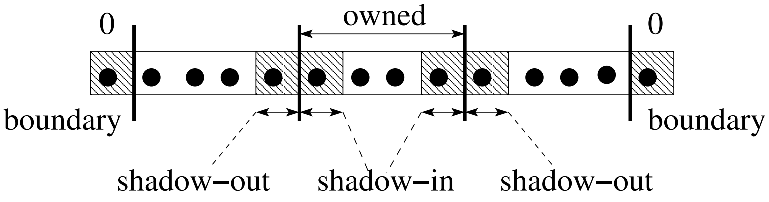
the special case that only one processor is running the code (line 17). When this is not the case, buffers are set up for the exchange both in the forward and backward direction and non-blocking receives are issued in ready, data is copied to the buffers and sent in give3 and, finally, take3 waits for the receives to complete and copies the data received from the temporary buffers to the shadow regions in the arrays used in the computations. Routines give3 and take3 are not included here because of their lengths (86 and 65 lines, respectively).

Altogether, the code to update the shadow regions in the general case requires a total of 193 Source Line of Codes (SLOCs), which rise to 327 if we include comm1p. In contrast, the computational part of the stencils is much shorter. For the two intra-grid operations inversion and residual computation, the source line of codes are only 24 and 23 respectively. Table 6.1 shows the SLOCS for shadow region management (comm3 and comm1p) and stencil computations (inversion and residual computations).

The impact of shadow regions in programming environments that provide a global view of the data is smaller, but still noticeable. For example, the Co-Array Fortran (CAF) [38] version of MG requires 168 SLOCs to update these regions, 302 if we include comm1p. Finally, none of the MG stencils updates the input array within the stencil. It enables them to avoid the shadow regions between the tiles in programming environments like openMP [15] that do not require (explicit) tiling. But still in this favorable situation shadow regions can generate non-negligible overheads: the OpenMP version of MG uses a periodic boundary shadow region which needs a comm3 routine of 29 SLOCs to keep it updated. Table 6.1 shows the

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

(a)



(b)

Figure 6.5: Example of overlapped tiling. (a) Construction of an HTA with tiles that overlap in both directions and boundary regions. (b) Pictorial view.

SLOCs for the Co-Array Fortran (CAF) and OpenMP versions.

As we see, the code for shadow region management sometimes is much longer than the computational kernel they support. Programming the shadow regions is error-prone due to the number of memory regions and complexity of the indexing expressions and data interchange patterns involved. Nevertheless this management is conceptually mechanic. Thus, it is ideal to be delegated to a library or a compiler, provided enough information is furnished by the user.

6.3 Syntax and semantics

We designed a new approach in htalib to tackle the problems mentioned in Section 6.2 for tiled stencil codes: overlapped tiling. It consists in specifying at construction time that the contents of each tile of the HTA overlap with a given number of elements of each neighboring tile. The HTA becomes responsible then for creating shadow regions of storage if they are needed, and updating them as necessary.

6.3.1 Creation

We first give the definitions of *owned* region, *shadow-out* region, *shadow-in* region and *boundary* region before we show the creation of overlapped tiling. When a tile T is defined in the creation of an HTA, we say that the region T comprises is *owned* by T. The size of this owned region for each tile is given by the tiling parameter, the first parameter in the HTA constructor in Figure 6.5(a). The *shadow-out* region refers to the shadow region we defined earlier. They are extended regions outside T that can be accessed by the processor operating on T. Inside T, there are regions that other tiles are allowed to access. We call those *shadow-in* regions of T. For example, in Figure 6.5(b), we show the shadow-in and shadow-out regions for the second tile of A defined in Figure 6.5(a). *Boundary* regions are regions allocated around the complete original array. They provide the boundary values of original array. The size of the boundary regions is assumed to be the same as that of the shadow regions. The boundary region can be a periodic boundary which reflects the data at the opposite end. It can also be a non-periodic boundary which is simply an extended area filled with some initial value.

HTAs with overlapped tiling, which we call *overlapped HTAs*, are created by using an Overlap object as last parameter in the HTA constructor. An object of type Overlap is built using the constructor

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

The DIM is the same as the number of dimensions in the HTA we want to create. 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 the first two parameters define the size of the overlap and the size of the boundary region. The mode parameter specifies the nature of the boundary regions in the original array. It can be either `zero` (non-periodic boundary filled with zeros) or `periodic` (periodic boundary).

The example code in Figure 6.5(a) creates an 1×3 HTA `A` with 4 elements per tile in which each tile overlaps with its neighbors one element in both directions. The boundary regions for the array consist of one element in each direction and they are initialized automatically with zeros. The resulting HTA is depicted in Figure 6.5(b). As a result of the overlap, the shadow-out regions for the second tile are the last element of the first tile and the first element of the third tile.

6.3.2 Indexing, assignment and operations for overlapped HTAs

Overlapped tiling provides flexible indexing to HTAs so that each tile can index the neighboring elements in adjacent tiles. The indexing for the owned region of each tile remains the same as if no overlapped tiling had been applied. The index for the owned data starts from 0 to the maximum size of the tile minus one in each dimension. The indexing for the elements across neighboring tiles extends the owner's range in both positive and negative directions in each dimension. For example, let us consider a one dimensional tile `T` of size 4, with overlapped tiling in both directions with length 1 as shown in Figure 6.5 (c). The region owned by `T` can be indexed as `T[0:3]`. The left shadow region can be indexed as `T[-1]` and the right shadow region can be indexed as `T[4]`. We define the symbol `ALL` inspired by ZPL [14] to index all elements in the region owned by `T`. The expression `T[ALL+1]` indexes a region which shifts `T` to the right by 1. It can also be thought as adding value `1` to each index in `T[ALL]`.

The HTAs created by indexing overlapped HTAs do not allow overlapped indexing anymore. If the HTA `A` in Figure 6.5 is indexed as `C = A()[1:2]`, the newly created HTA `C` is a normal HTA of 1×3 tiles with two elements each. Thus, indexing such as `C(0)[-1]` is not allowed.

Figure 6.7 shows the same code as Figure 6.2, but it uses overlapped tiling. Compared with the original code, it helps simplify the program by requiring only one statement in the computation. Furthermore, the simpler indexing scheme makes the code more readable.

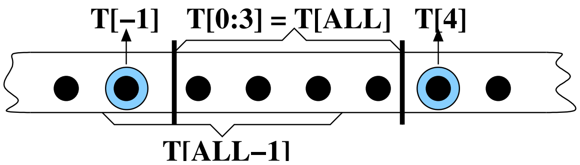


Figure 6.6: Example of indexing for overlapped HTA.

```
HTA<double,1> A,B;
Seq tiling =((d), (n));
Overlap<1> ol((1), (1), zero));
B = HTA<double,1>::alloc (tiling, matrix, ol);

while(!converged){
  A()[ALL] = 0.5 * ( B()[ALL-1] + B()[ALL+1]);
  ... ...
}
```

Figure 6.7: HTA code for 1D Jacobi with overlapped tiling

When a tile of an overlapped HTA is operated with a scalar, an array, or another HTA, only the owned region of each tile participates in the operation, that is, the shadow-out regions and the boundary regions do not participate. The same rule applies to assignment. When an overlapped HTA is on the left side of an assignment, only the values in the owned region will be modified. These rules enable the interaction of overlapped HTAs with non-overlapped HTAs, or overlapped HTAs with different overlap sizes.

In summary, users can read the overlapped shadow-out regions of an overlapped HTA by indexing; however, they are only allowed to write to the owned regions, just as in standard HTAs. Making the shadow regions transparent to the programmer makes the code simpler and cleaner. However, htalib has to take care of more aspects of the shadow regions, and particularly, their consistency.

6.3.3 Shadow region consistency

htalib provides mechanisms to enforce consistency in shadow regions, as when the shadow-in region of a certain tile is modified, the corresponding shadow-out region should be consistent with this change before the following read.

Consistency is trivial for some data layouts such as row or column major in the case of non distributed HTAs. This is because the shadow-in regions of one tile and the shadow-out regions of its neighbors are physically the same place. Any updates in the shadow-in regions of one tile will be immediately seen by its neighbors. As for the periodic boundary, since our current implementation replicates the data to corresponding boundary regions, these boundary regions need explicit copying to keep them consistent.

For some data layouts such as the *tile* data layout, or the data layout in distributed HTAs, proper updates should be performed by htalib in order to keep all the copies of an overlapped region consistent. For the *tile* data layout, the update is done through copying. In distributed HTAs, the update requires communications. Except the different actions for updating the shadow regions, the strategies of where and when to update for tile data layout and the data layout in distributed HTAs are the same. We now present the update strategy used for distributed HTAs in htalib.

Update policy for distributed overlapped HTAs

Programmers can rely on htalib for the automatic update of shadow regions. They can also perform manual update themselves particularly if they choose to disable the automatic update mechanism, as we will explain later. The library provides an interface for manual update as follows. At any point in the program, the programmer can specify a synchronized update `A.update()` or a pair of asynchronous update function calls `A.asyncUpdate()` and `A.sync()` for an overlapped HTA `A`.

There are two possible ways to perform an automatic update after the shadow-in region of one tile is modified. One is to update immediately after an assignment operation to one or more elements of the shadow-in region. We call it *update on write* policy. The other alternative is just before an access operation to one or more elements of the shadow-out regions of the neighboring tiles. We call it *update on read* policy.

The *update on write* policy has the advantage of sending the modified data as early as possible. However, it does not know which neighbor(s) will perform a read next. So the modified data must be sent to every neighbor whose shadow-out region overlap with the modified area. On the contrary, the *update on read* policy only updates the neighboring tiles that are needed, although bookkeeping for the status of each tile is required, and the update is delayed until the next read of neighboring tiles.

We chose the *update on read* policy to keep the shadow region consistent for several reasons. First, it leads to a minimal number of communications: only the tiles needed will be updated. This is important for some applications with wavefront computations. Second, the bookkeeping mechanism this policy involves is also required by the manual update which can be performed by the user. The *update on read* policy for automatic update may suffer from delays if, between the write to shadow-in regions and the read of corresponding shadow-out regions, other computations take place and occupy a large amount of time. However, in that case, the user can overlap the communications with computations by placing a manual asynchronous update for the neighboring tiles immediately after the write and a synchronization call before the read.

6.3.4 Implementation in the library

We now present the design details for the update of shadow regions in htalib. First, we introduce the implementation of the *update on read* policy. Then we show a few changes in the library in order to make overlapped tiling more flexible.

The implementation for the *update on read* policy is as follows. A small table records for each tile the state of its shadow-in and shadow-out regions. The list of each tile's neighbors is also recorded along with the shadow-out regions. Since the HTA class provides a global

Operations	Owner shadow-in region status	Corresponding neighbor shadow-out region status
Owner writes in shadow-in region	Set status to *inconsistent.*	Set status to *inconsistent*
Neighbor reads from shadow-out region	If *inconsistent*, send the update, set status to *consistent.*	If *inconsistent*, receive the update, set status to *consistent.*

Table 6.2: Actions for each shadow-in region and shadow-out region on read and write

view and a single logical thread of execution, each tile knows every operation performed on it or its neighbors. For instance, in Figure 6.7, the statement in line 7 calls the overloaded function for operator "=", in which every processor examines the input HTA `A()[ALL]` to see 1). if it owns one of the tiles; 2). if its neighboring tiles are inside the input HTA. Such examination is done without communications. The contents of each entry in the table and the associated actions for read and write operations are shown in Table 6.2. Each entry consists of two fields: shadow-in region status and shadow-out region status. The values in the two fields will be changed accordingly upon the read and write operations. Since a two sided communication layer is used in `htalib`, we keep track of shadow regions from two sides: the sender side (shadow-in region) and the receiver side (shadow-out region). If one sided communication is used, only the receiver side (shadow-out region) needs to be recorded. When there is a need for an update, the whole shadow-in region will be sent to its corresponding neighbor.

We placed the update actions shown in the bottom row of Table 6.2 in all the HTA functions or overloaded operators that can take an overlapped HTA as an input, such as operator +, -, =, and HTA function `mapReduce()`, `hmap()` [11]. Those functions will update the input HTAs before operating. On the other hand, functions or operators that write to HTA should invoke the bookkeeping actions listed in the front row of Table 6.2. For example, operator = and `hmap()` add the bookkeeping functions at the end.

By inserting update functions before the reads and placing bookkeeping functions after the writes, htalib provides automatically updated shadow regions.

We also accommodated several changes in htalib to make overlapped tiling more general and flexible. One of the changes consisted in adding a parent field to the HTAs resulting from accessing overlapped HTAs. The HTAs created by such indexing are normal HTAs with a new mapping over the raw data. To prevent writes to the shadow-out regions of the data through these HTAs, they are associated with the overlapped HTAs from which they were generated through the parent field. This enables the legality checks, necessary updates and bookkeeping to be done through the parent pointer on the overlapped HTAs. For example, in a 2×2 HTA `A`, there is a write to the indexed HTA `A(0)`:

```
A(0)= 1;
```

The library checks whether the write to `A(0)` modifies the owned regions of its parent `A`. Since the shadow-in region in the parent HTA `A` is modified, the status for the shadow-out regions of the neighboring tiles `A(1), A(2)` should be changed accordingly. Even if the neighboring tiles do not appear in the assignment, through the parent pointer, the library is able to perform the update or bookkeeping successfully.

Another change made in the library was to allow users to disable the automatic update in overlapped HTAs. In stencil computations, some overlapped HTAs are just temporaries which do not need updates. In this case, we can disable the automatic update by adding a fourth parameter **false** in the constructor of an `Overlap` object.

6.4 Example

Let us examine a Jacobi relaxation code with periodic boundary 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 6.8. If distribution `dist` is specified, it can also be used in parallel case. 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. The first

```
int ntiles = N / d;
Tuple<2>::seq tiling =((d+2,d+2), (ntiles, ntiles));

v = HTA<double, 2>::alloc(1, tiling, dist, ROW);
u = HTA<double, 2>::alloc(1, tiling, dist, ROW);
...
while( dif < epsilon) {
  if(ntiles > 1){
    v(1:ntiles-1, :)[0, :]    = v(0: ntiles-2, :)[d, :];
    v(0:ntiles-2, :)[d + 1, :]= v(1:ntiles-1, :)[1, :];
    v(:, 1:ntiles-1)[:, 0]    = v(:, 0: ntiles-2)[:, d];
    v(:, 0:ntiles-2)[:, d + 1]= v(:, 1:ntiles-1)[:, 1];
  }
  v(0, :)[0, :] = v(ntiles-1, :)[d, :];
  v(ntiles-1, :)[d+1, :] = v(0, :)[1, :];
  v(:, 0)[:, 0] = v(:, ntiles-1)[:, d];
  v(:, ntiles-1)[d+1, :] = v(:, 0)[:, 1];

  u(:,:)[1:d, 1:d] = k * (v(:,:)[1:d, 0:d-1] +
                          v(:,:)[0:d-1, 1:d] +
                          v(:,:)[1:d, 2:d + 1] +
                          v(:,:)[2:d + 1, 1:d] );
  dif = maxdif(u(:,:)[1:d,1:d], v(:,:)[1:d,1:d]);
  v = u;
}
```

Figure 6.8: Stencil code with explicit shadow region exchange

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

Figure 6.9: Stencil code using overlapped tiling

four statements in the loop are shadow region exchanges between the neighboring tiles. The next four statements are periodic boundary updates. The actual computations use only local data.

The version with overlapped tiling in Figure 6.9 does not need either to declare explicitly the shadow regions or update them. In line 4, the object of `Overlap` is constructed with periodic boundary mode with automatic update. Also the data that actually belongs to the tiles is indexed from 0 to $d-1$ in every dimension, as in regular HTAs. The shadow regions are in indexes -1 and d in each dimension. Notice that we could have further simplified the indexing with using `ALL` and operating it with tuples. For example `v(:,:)[0:d-1, -1:d-2]` can be also expressed as `v(:,:)[ALL-(0, 1)]`. Finally, it is interesting to notice that the left hand side of the assignment in the stencil no longer requires selecting the area to be overwritten, as it is the total one actually owned by each tile, just as in non-overlapped HTAs.

6.5 Summary

Overlapped tiling for HTAs has two major advantages. First, it releases the programmer from creating and updating shadow regions explicitly. Overlapped tiling tries to eliminate all operations related to shadow regions or boundary regions in the program and have them handled by the library. Therefore, the code is simpler and cleaner.

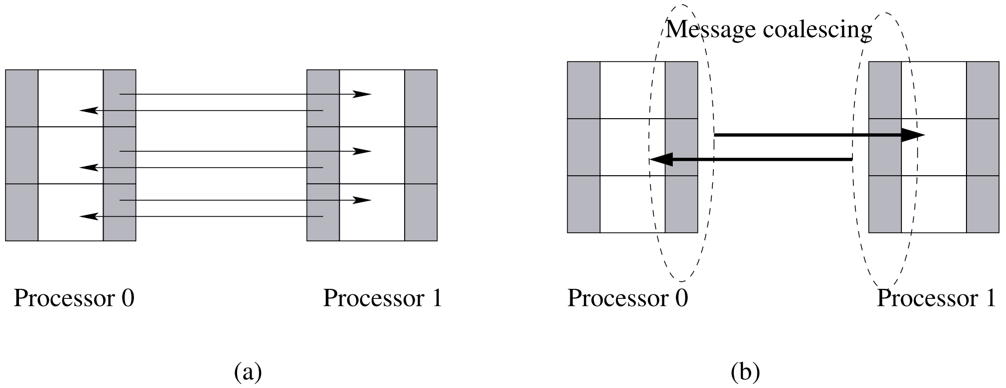


Figure 6.10: Message coalescing for overlapped HTA. (a) Before. (b) After

Second, the library can handle the update of shadow regions in a more optimized fashion than naive implementations. Some optimizations applied in HPF compilers [24][29] can be applied here. For example, the library can possibly perform message coalescing which combines several messages together before sending them to a certain processor. Figure 6.10 shows the example without message coalescing in (a). The HTA is distributed along the second dimension on two processors processor 0 and 1. The shadow region update can be sent separately, but it requires 3 messages from each processor. Figure 6.10 (b) illustrates the effect of message coalescing. The shadow regions sent to one destination processor are merged first and sent as one message. Another possible optimization is multi-dimensional shift communication [24] which reduces the number of messages from $3^d - 1$ to $2d$, where d is the number of dimensions. Figure 6.11 (a) shows the example for 2D case. Without the optimization, a tile needs to update its shadow regions from 8 neighbors. However, if the updates are arranged in two steps as depicted in Figure 6.11 (b), we can reduce the number

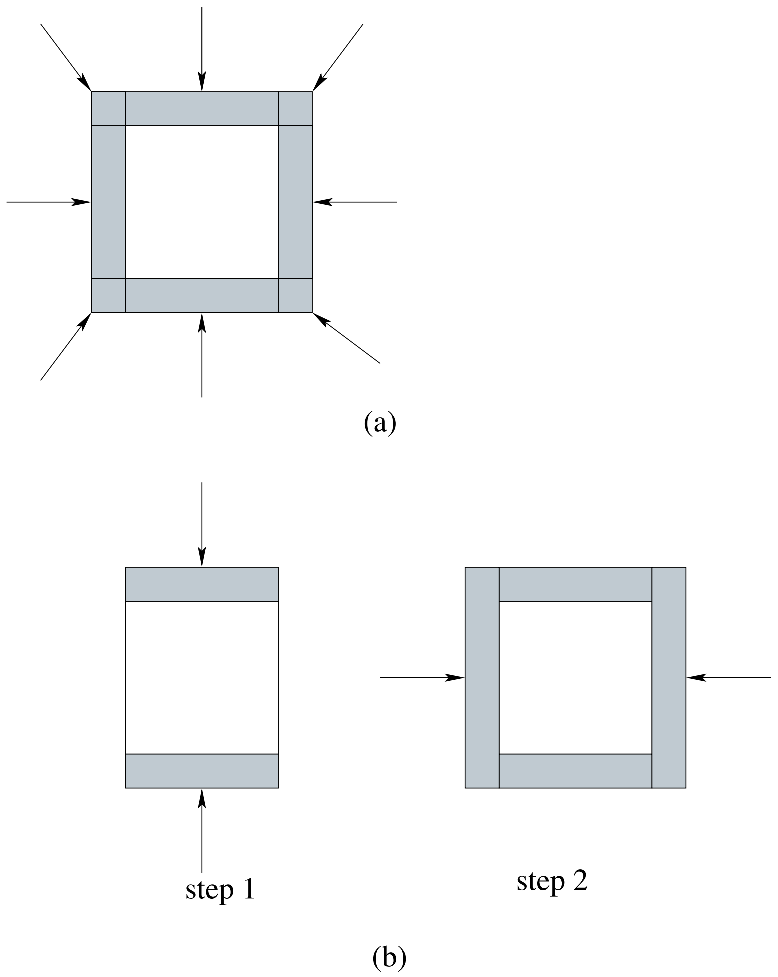


Figure 6.11: Multi-dimensional shift communication for overlapped HTA. (a) A tile needs updates from 8 neighbors (before). (b) A tile only needs updates from 4 neighbors (after)

of messages to 4. Note that a barrier should be placed between the two steps to make sure step 1 finishes before step 2 starts.

However, overlapped tiling has some disadvantages. One is associated with the library approach. The choices of when and where to update the shadow regions may not be globally optimal since the library has no idea about the overall program. For instance, our automated updates cannot be triggered until an attempt to read the inconsistent shadow region, which may delay the update in the critical path. In these situations user intervention is required to improve performance, in this particular case probably by means of manual asynchronous updates.

Chapter 7

Evaluation

In this chapter, we examine the validity of our approach to write codes using the HTA data type both in terms of performance and programming productivity. The experiments were conducted on a series of sequential benchmarks and parallel benchmarks using the HTA data type: matrix matrix multiplication, matrix transposition, LU factorization, triangular Sylvester equation, 3D Jacobi, vpenta from NAS kernel benchmark, a routine to invert three pentadiagonals simultaneously, as well as MG and LU from NAS parallel benchmarks [2]. The benchmarks, along with their descriptions are shown in Table 7.1. For most of the benchmarks, we focus on the application of the two new language constructs. In most benchmarks, we observe little performance degradation compared to the hand optimized codes. This is because the additional cost of tiling structure and tile manipulation functions at run time can be amortized over enough computations for each tile. Nevertheless, the readability is improved greatly thanks to the simplification of indexing, function interfaces, and concise representation of the algorithms.

This chapter is structured as follows. In Section 7.1, we present the several sequential benchmarks using MATLAB implementation. In the rest sections, the experiments were conducted in htalib. Section 7.2 shows the performance results of our HTA codes using htalib and their counterparts for a series of sequential benchmarks. Next, we rewrote two parallel benchmarks LU Solver and MG from NAS benchmark suite and applied the overlapped tiling to both of them. The performance numbers are shown in Section 7.3. Last but not least, The systematic analysis for the improvement of productivity is discussed in Section 7.4.

	Benchmark	Description	Implementation Version
Sequential	MMM	matrix matrix multiplication	MATLAB and htalib
	Transpose	transpose a matrix	MATLAB and htalib
	Jacobi	stencil computation on 3D data with 6 neighbors	MATLAB and htalib
	vpenta	inverting the three pentadiagonals simultaneously	in MATLAB
	staticLU	the iterative static LU factorization	in htalib
	dynamicLU	the iterative dynamic LU factorization	in htalib
	recLU	the cache-oblivious LU factorization	in htalib
	Sylvester	the solver of triangular Sylvester equation	in htalib
Parallel	NAS MG	Multi Grid V-cycle algorithm	in htalib
	NAS LU	Navier Strokes equation solver	in htalib

Table 7.1: Descriptions of the benchmarks used in experiments.

7.1 Sequential benchmarks using the MATLAB implementation

In this section, we show some benchmarks that are able to achieve good locality and clean syntax using HTA toolbox in MATLAB. The MATLAB version we use is 7.0.4.352 (R14) and the experiments were conducted on on a 3.0 GHz Intel Pentium 4 with 16KB L1, 1MB L2 and 1GB RAM.

7.1.1 Matrix matrix multiplication

MATLAB uses operator "*" for matrix matrix multiplication (MMM) which calls the highly optimized library routine. In our HTA implementation, we tried one level of tiling and two levels of tiling. At the leaf level, "*" is used for multiplication of leaf tiles. The performance reported is a result of search for the best tiling combinations that generate the best performance for each matrix size. Please note that in MATLAB, the leaf tiles are stored in pseudo-tile layout: the data inside one tile are stored continuously in memory. Nevertheless, the leaf tiles are not stored together since MATLAB controls the allocation process. Another two versions, the MMM in MKL library (MKL) and the MMM using the library provided by MATLAB (MATLAB lib) are also shown for completeness. It indicates

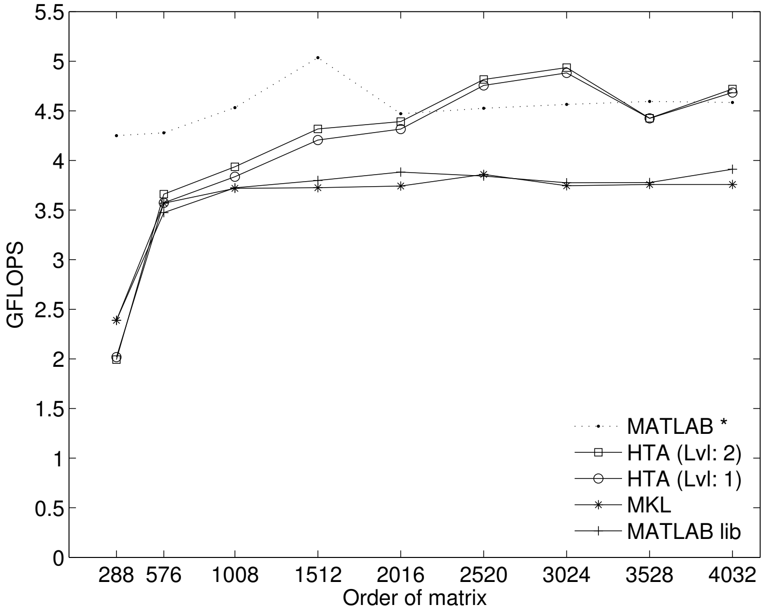


Figure 7.1: Performance of MMM in the MATLAB implementation

that the MATLAB BLAS library is MKL since the two lines overlap. Further, the MMM provided by MATLAB is better than that from MKL. However, with the searching in our HTA, we can achieve matched performance and sometimes, better performance.

7.1.2 Matrix transposition

In MATLAB, the transposition of a matrix a is represented as a'. We overloaded this operator for the HTA toolbox. Therefore, the transposition of an HTA A is A'. The syntax is more concise than that of htalib. We show the native transposition in MATLAB and our HTA transposition in Figure 7.2. Pseudo MFLOPS which measures one million assignment per second since the code executes no floating point operations. Again, we searched for the best performance point for each matrix size. The performance of HTA transposition is able to achieve in average 27% better performance than the MATLAB native transposition.

It is worth noting that if we partition the matrix along each row/column, and leave the other dimension unpartitioned, the HTA transposition is achieved without incurring data movement. For example, a 3600 × 3600 matrix can be partitioned into a 3600 × 1 HTA

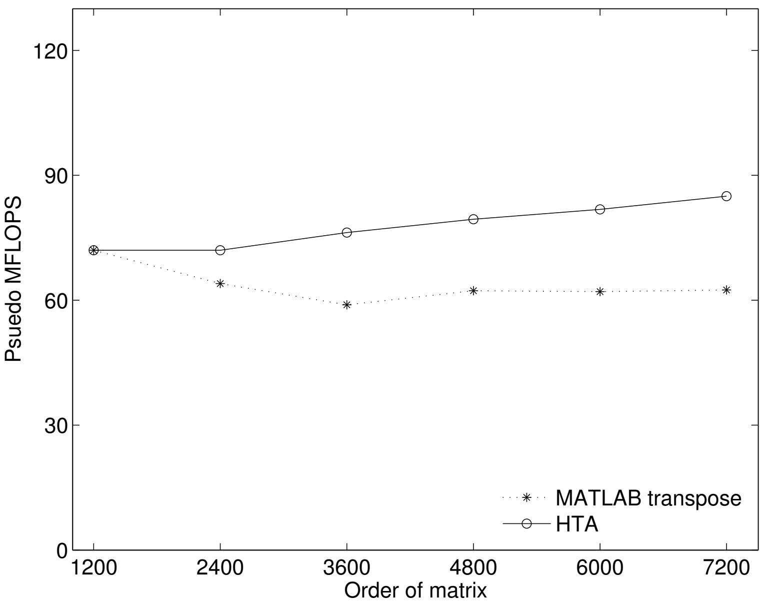


Figure 7.2: Performance of Transpose in the MATLAB implementation

with each tile of size 1 × 3600. Since the data of each leaf tile is one dimensional and stored continuously, the transposition of such leaf tile is not really performed by MATLAB as an optimization. Therefore, transposing an HTA in this case is only operated on tiling structures, not on the data in the leaf tiles. The performance of this case is 3.4-6.8 times faster than the best performance of HTA we found in Figure 7.2.

7.1.3 3D Jacobi

We implemented in MATLAB both the untiled 3D Jacobi version and the HTA version. We show the performance in Figure 7.3. Compared with the latter experiment in Figure 7.12, both versions in MATLAB suffers from poor performance. This is because MATLAB interprets the instructions. However, the best performance of HTA code we searched is in average 27% better than the untiled MATLAB code since it enhances the locality.

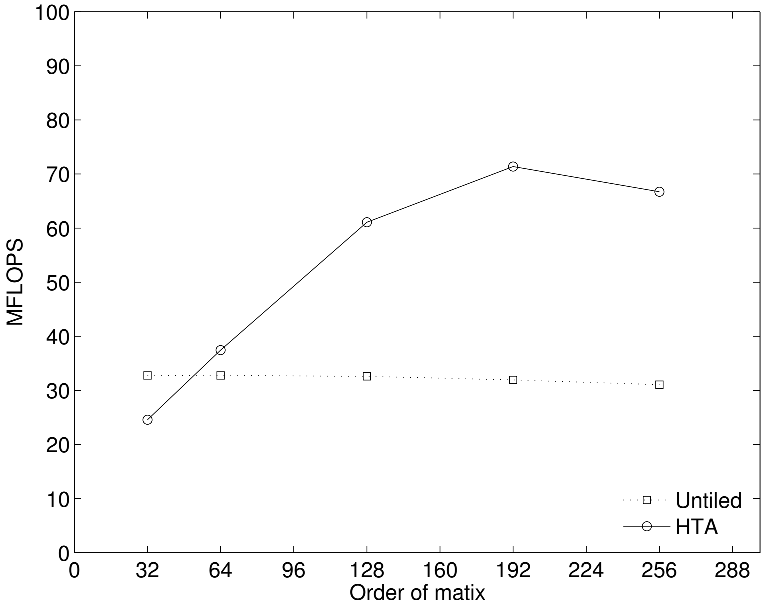


Figure 7.3: Performance of 3D Jacobi in the MATLAB implementation

7.1.4 Vpenta

Vpenta is one of the kernels in NASA7, a program in the SPEC92 floating-point benchmark suite. This kernel simultaneously inverts three pentadiagonal matrices. This program operates on a set of two-dimensional and three-dimensional arrays. The program has poor locality since the innermost loop operates on the row dimension while the data are stored in column dimension. It can be improved by partitioning both the first and second dimensions of the arrays.

The performance results are shown in Figure 7.4. We implemented the untiled version and the HTA version. Since most of codes are interpreted by MATLAB, the absolute performance of both versions are low. However, the HTA code with high locality outperforms the untiled version by 34.8%.

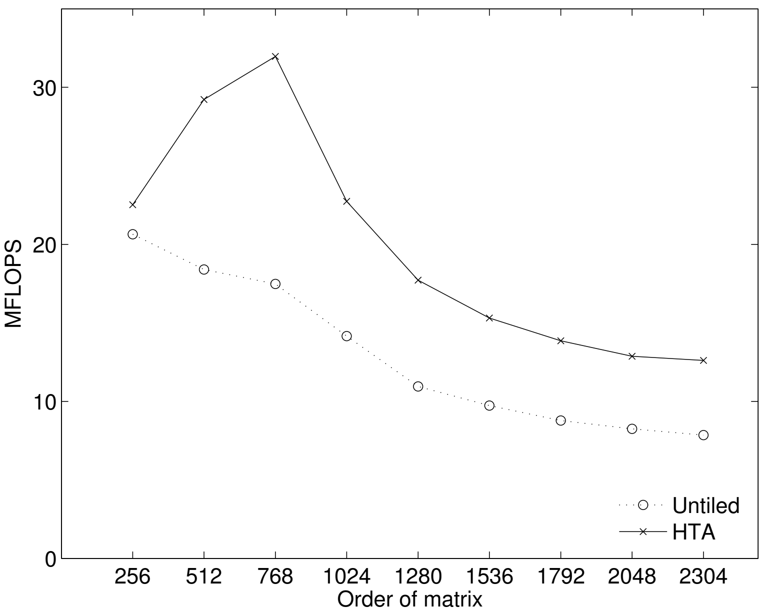


Figure 7.4: Performance of Vpenta in the MATLAB implementation

7.2 Sequential benchmarks using htalib

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.The tiling parameters for every sequential benchmark are summarized in Table7.2.

7.2.1 Matrix matrix multiplication

First, we show the performance of Matrix Matrix Multiplication (MMM). The HTA code in Figure 4.1 was used for the experiments. The recursive MMM code for leaf level HTAs has been implemented using the miniMMM code generated by ATLAS [46] We tried different configurations in HTA by changing the parameters at the construction time illustrated in Figure 4.1, line 3,4. The search space for performance tuning can be seen in Figure 7.5. Different HTA configurations include: different levels used in HTA `A`, `B` and `C` (denoted by `Lvl= n` in the legend); different data layout for HTA `A` and `B` (denoted by `ROW` or `TILE` in the legend). HTA `C` always uses `ROW` data layout. For both row major and tile layout, with one

Programs	Versions	Levels of tiling	Absolute tile sizes	Data layout	Comments
MMM	HTA	1 or 2	36×36, 144×144	Row or Tile	
	ATLAS	multiple	36×36 for L1 cache	Row	Version 3.6.0
CO MMM	HTA	≥ 4	base: 63×63	Row	
Transpose	HTA	1 or 2	60×60, 600×600	Row or Tile	
	Tiled C	1	60×60	Row	
staticLU	HTA	1	72×72	Column or Tile	
	LAPACK	1	72×72	Column	Version 3.0
dynamicLU	HTA	1	72×72	Column	
	FLAME	1	72×72	Column	Version 0.9
recLU	HTA	≥ 8	base: 4 columns	Row	
	ATLAS	≥ 8	base: 2 columns	Row	Version 3.6.0
Sylvester	HTA	4	40×40, 40×120, 120×120, $120 \times n$	Column	n:size of 2nd dimension.
	FLAME	4	same as HTA	Column	Version 0.9
Jacobi	HTA-OL	1	best value searched	Row	
	HTA-Orig	1	same as HTA-OL	Row	
	Tiled C	1	best value searched	Row	

Table 7.2: Summary of different tiling parameters used in sequential benchmarks

level of tiling (`ROW, Lvl=1`, and `ROW, Lvl=1`), the best tile size we obtain is 36 × 36. For two levels of tiling (`ROW, Lvl=2` and `TILE, Lvl=2`), the best absolute tile sizes we obtain are 36×36 for leaf level, and 144×144 for level 1. The best performance we obtain from the search matches the performance of ATLAS. It also shows with little effort, we are able to try different combinations of architecture related parameters to search for the best performance, thanks to a higher abstraction in the algorithm.

Second, the cache-oblivious algorithm of MMM (CO MMM) shown in Figure 5.6 was also implemented with dynamic partitioning. Although the algorithm requires the recursion go down to scalar level. We found the performance was very poorly, less then 1% of the peak performance on the target machine. It matches the observations in [49]. Yotov et al. [49] revealed two reasons of the poor performance of CO MMM: 1). The overhead of recursion. 2). The poor use of registers across procedure calls. Although the cache oblivious algorithm proves to be I/O optimal, it does not guarantee good overall performance. Therefore, we

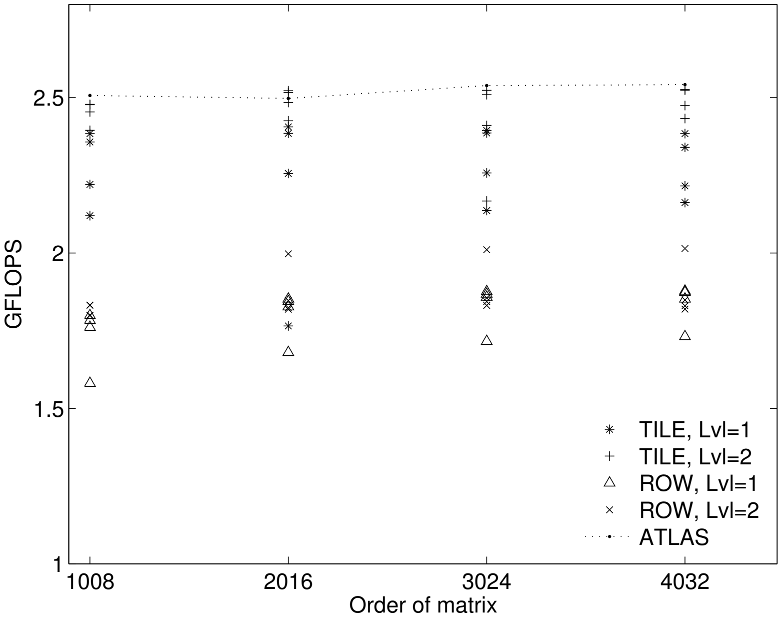


Figure 7.5: Search in MMM

reduce the levels of recursion so that it stops at tile size 63 × 63. The experimental data are shown in Figure 7.6.

Figure 7.6 gives a complete picture of the performance of HTA MMM codes compared to other versions. For ATLAS, we used the MMM code with the parameter values that ATLAS found to be optimal for tile size, register blocking parameters, among others. We also show the performance of Intel MKL library [1] version 8.0. However, since the miniMMM code inside HTA and the ATLAS library in this experiment do not use SSE multimedia extensions, there is a gap between MKL and HTA version. Nevertheless, the HTA version matches the trend presented in MKL.

7.2.2 Matrix transposition

The performance of matrix transposition in Figure 4.2 is reported in Figure 7.7. Pseudo MFLOPS which measures one million assignment per second since the code executes no floating point operations. We tried different configurations in HTA by changing the parameters at construction time in lines 2-5 in Figure 4.2. It includes different levels of tiling in

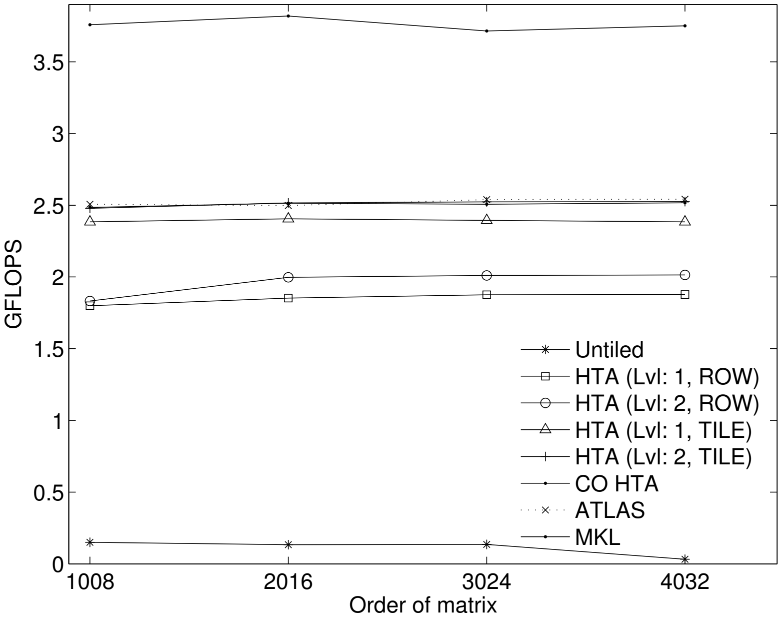


Figure 7.6: Performance of MMM

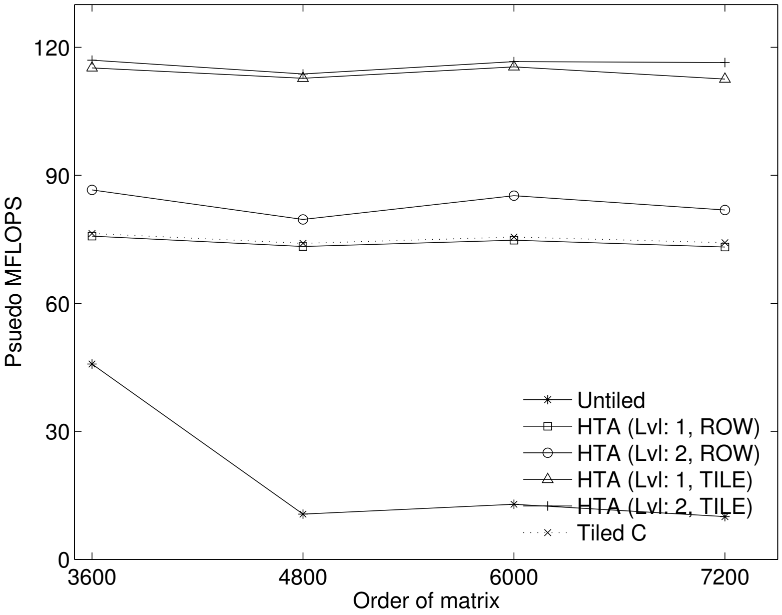


Figure 7.7: Performance of Transpose

HTA `A`, `B` (denoted by `Lvl:  n` in the legend); different data layout for `A` and `B` (denoted by `ROW` or `TILE` in the legend). And the tile size for the HTA leaves is 60 × 60, the tile size searched to achieve best performance. Compared with the tiled C code where tiling is represented by 4 nested loops `Tiled C`, the HTA code with one level of tiling in Row major layout `Lvl:1, ROW` exhibits little overhead. Further, the HTA with more levels of tiling in Row major (`Lvl:2, ROW`) and the HTA with Tile layout (`Lvl:1, TILE` and `Lvl:2, TILE`) show shorter execution time because of the increased locality.

7.2.3 LU factorization

We show the performance of the iterative static LU algorithm, iterative dynamic LU algorithm, and the recursive LU algorithm in Figures 7.8, 7.9 and 7.10 respectively. The tile size used was 72×72 for the iterative static and dynamic LU algorithms. For the recursive algorithm, the recursion stopped when A had 4 columns instead of 1 column in our experiment to reduce the recursive function call overhead. The HTA codes for the three algorithms have already been discussed and shown in Figures 4.6, 5.3 and 5.10, respectively.

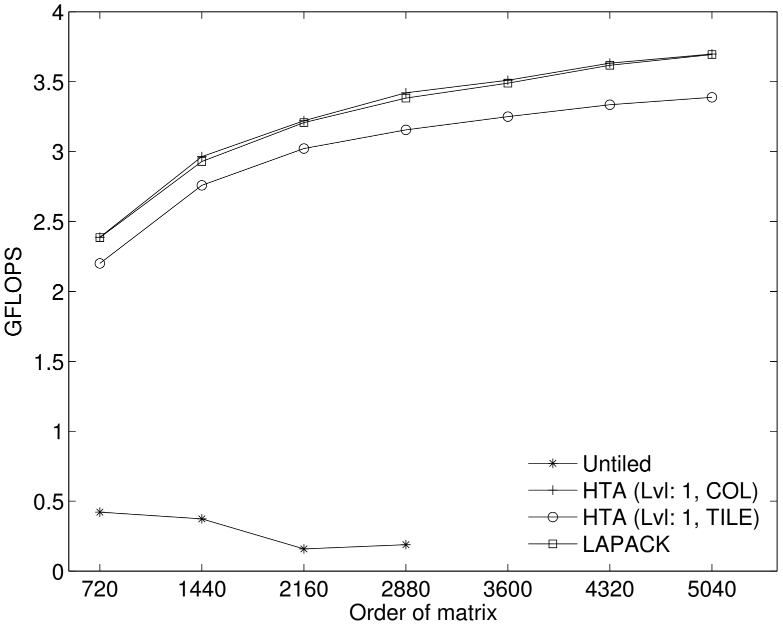


Figure 7.8: Performance of iterative static LU algorithm

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 [5]. LAPACK implements the same iterative LU factorization, with column major layout for the matrix A. We see that version `HTA COL` has a negligible performance difference from the LAPACK version.

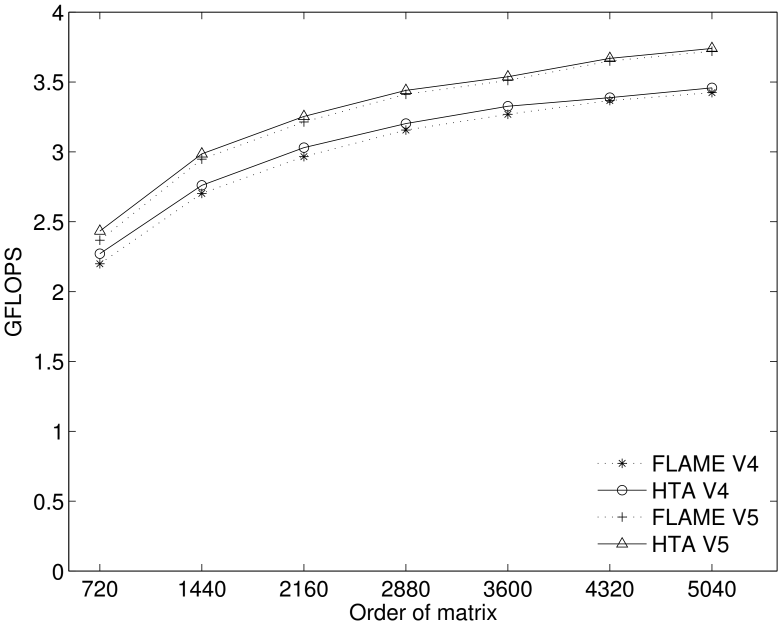


Figure 7.9: Performance of iterative dynamic LU algorithm

The iterative dynamic LU algorithm, was implemented using both HTAs and the FLAME API [8] in C. Since FLAME derives a family of variations for LU factorization, we chose version `v4` and `v5` to compare, which are tested to achieve best performance in the family of variations. Figure 5.2 is the HTA implementation of version 5. The FLAME implementation of version 5 is in Figure A.1 in Appendix. The algorithm for version 4 shares similar structure of the first and third part. It uses different formulas in the second part. The performances of HTA and FLAME API are very close. In average, HTA performs 1.2% and 1.1% better than FLAME on version 4 and 5 respectively. Further, HTA's performance is within 1.6% difference in average from that of LAPACK.

Figure 7.10 shows two implementations of the recursive cache oblivious LU algorithms:

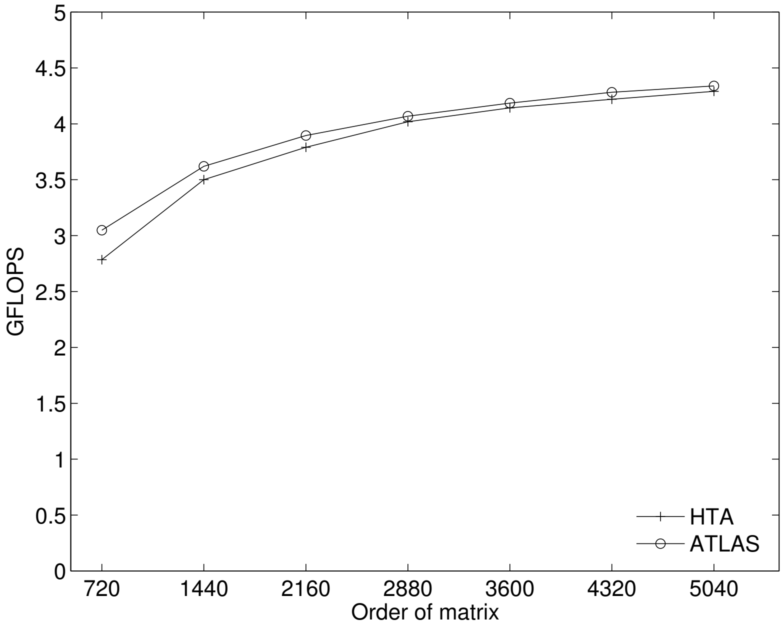


Figure 7.10: Performance of recursive LU algorithm

our HTA version from Figure 5.10 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 2.7%.

To summarize, Figures 7.8, 7.9 and 7.10 show little performance degradation using HTA's static partitioning, and dynamic partitioning.

7.2.4 Triangular Sylvester equation

The triangular Sylvester equation was implemented using both HTAs and the FLAME API [8] in C. From a family of variations for triangular Sylvester equation, we choose the best version, variant 3, reported in [39]. The absolute tile sizes for each level are $120 \times n$, 120×120, 40×120, 40×40. Figure 7.11 shows the performance for both implementations. The difference between HTA and FLAME API is small. For example, for 1800×1800 matrix, the difference is less than 2.6%

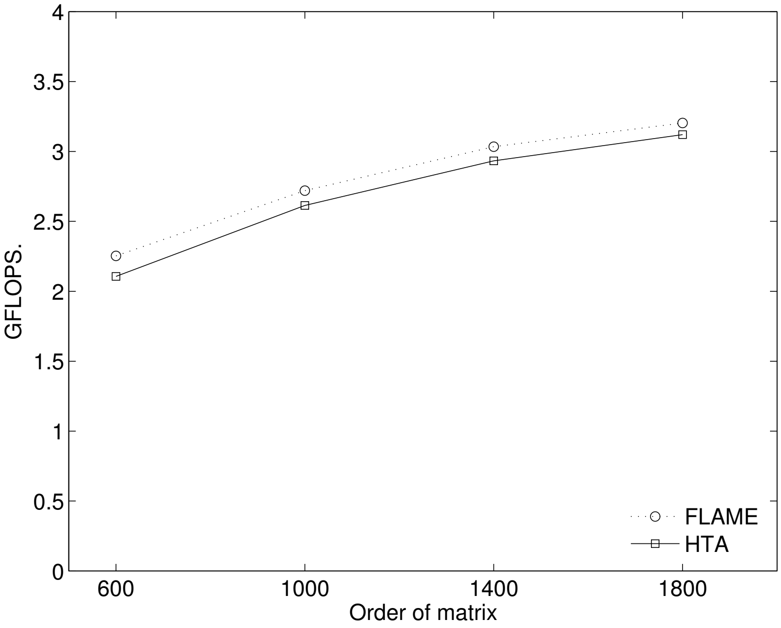


Figure 7.11: Performance of triangular Sylvester equation

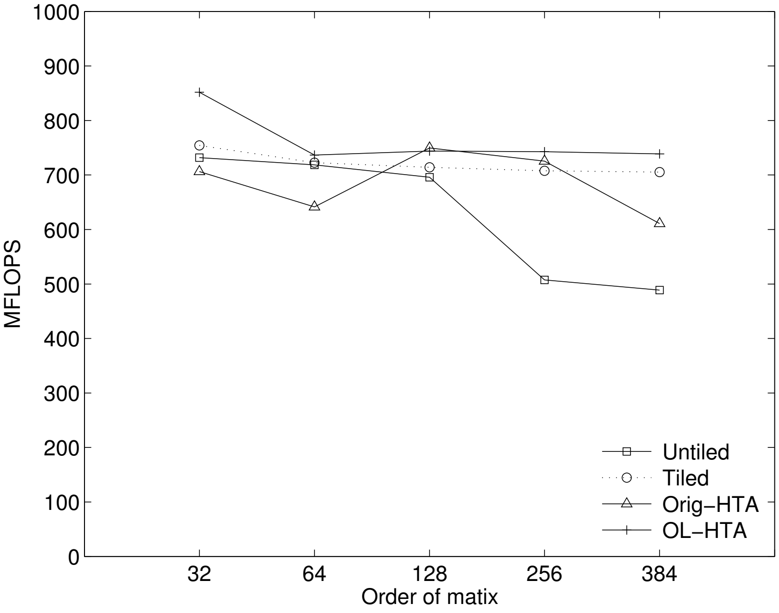


Figure 7.12: Performance of sequential 3D Jacobi computation

7.2.5 3D Jacobi

Figure 7.12 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 tiling sizes for the three tiled versions and present the best performance found. 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` follows the same tiling scheme as `OL-HTA` for all cases. It performs worse than `OL-HTA` because it has to pay the price of managing shadow regions for not being able to access elements across neighbors. The more tiles in HTA, the more overhead incurred in `Orig-HTA`. For matrix order 128 and 256, `Orig-HTA` performance is close to `OL-HTA` because the optimal tiling at those points generates fewer tiles. For example, for matrix order 256, the optimal HTA found is $1 \times 8 \times 1$ with the leaf tile size $256 \times 32 \times 256$

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, htalib uses a single loop to traverse the tiles.

7.3 Parallel benchmarks using htalib

We rewrote the NAS [2] MG and LU solver benchmarks using HTAs with overlapped tiling in order to evaluate the performance of this extension in parallel computation. The experiments were made in *Turing Cluster*, 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.

```
//NX, NY, NZ are the number of tiles in dimensions x,y,z
//nx, ny, nz are the tile sizes in dimensions x,y,z
if (NX > 0)
   u(0:NX-1, 0:NY, 0:NZ)[nx, 1:ny-1,1:nz-1] =
          u(1:NX, 0:NY, 0:NZ)[1, 1:ny-1, 1:nz-1];
u(NX, 0:NY, 0:NZ)[nx, 1:ny-1, 1:nz-1] =
             u(0, 0:NY, 0:NZ)[1, 1:ny-1, 1:nz-1];
if (NX > 0)
   u(1:NX, 0:NY, 0:NZ)[0, 1:ny-1, 1:nz-1] =
    u(0:NX-1, 0:NY, 0: NZ)[nx-1, 1:ny-1, 1:nz-1];
u(0, 0:NY, 0:NZ)[0, 1:ny-1, 1:nz-1] =
         u(NX, 0:NY, 0:NZ)[nx-1, 1:ny-1, 1:nz-1];
```

Figure 7.13: Code for explicit shadow region exchange in dimension x in the original HTA program for MG

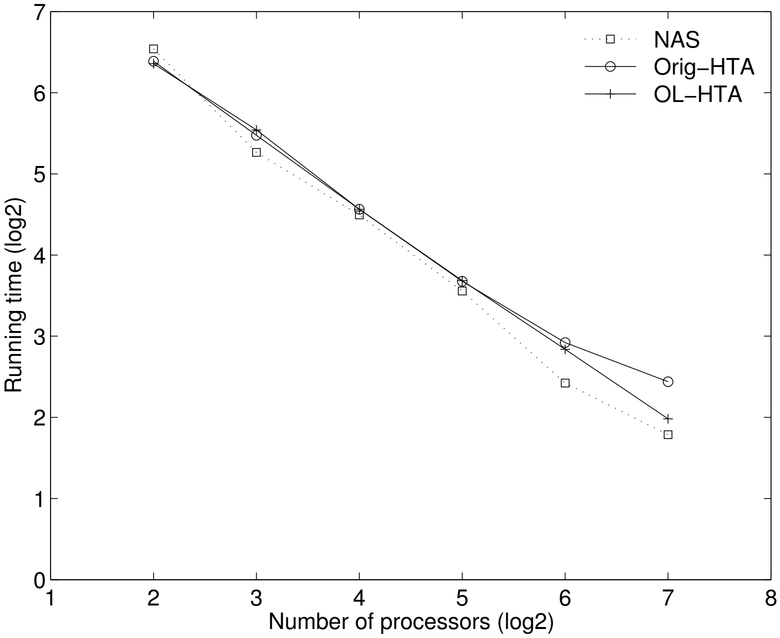


Figure 7.14: Performance of parallel MG benchmark class C

7.3.1 NAS MG

The NAS MG benchmark, as well as the parallel Jacobi program, represent examples of loosely synchronous computations, in which all processors alternate between phases of local computation and synchronous global communications [19]. In the NAS MG program even 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 are greatly reduced with respect to other programming approaches, but are still complex. Figure 7.13 shows the communication code incurred in dimension x in the original HTA code. The counterpart is the Fortran and MPI routine `comm3` in Figure 6.4. Although the source lines of codes are greatly reduced, 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 the shadow regions are completely removed from the program.

Figure 7.14 shows the performance for three versions of the NAS MG benchmark written in Fortran and MPI in class C (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 adds little overhead when compared with the original HTA version. It is even faster as the number of processors grows since it avoids the indexing operations shown in Figure 7.13. The performance of OL-HTA is also close to that of the NAS benchmark.

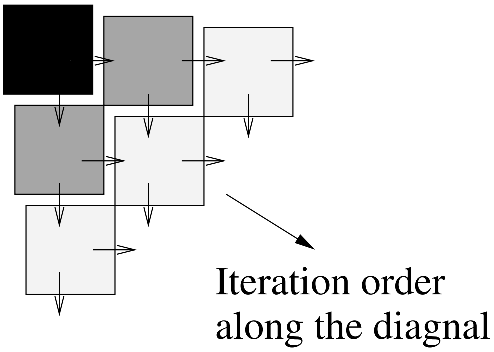


Figure 7.15: Illustration of diagonal iteration over tiles in the wavefront algorithm

```
for (int k=0; k<nPlanes; k++){
  fill (idx0, k);
  fill (idx1, k+1);
  A(idx0)[1:tx,1:ty] = A(idx0)[0:tx-1, 1:ty] +
                       A(idx0)[1:tx, 0:ty-1];
  //shdow region update
  A(idx1)(1:m-1, :)[0, :]= A(idx0)(0:m-2, :)[tx-1, :];
  A(idx1)(:, 1:n-1)[:, 0]= A(idx0)(:, 0:n-2)[:, ty-1];
}
```

Figure 7.16: HTA code for 2D wavefront algorithm

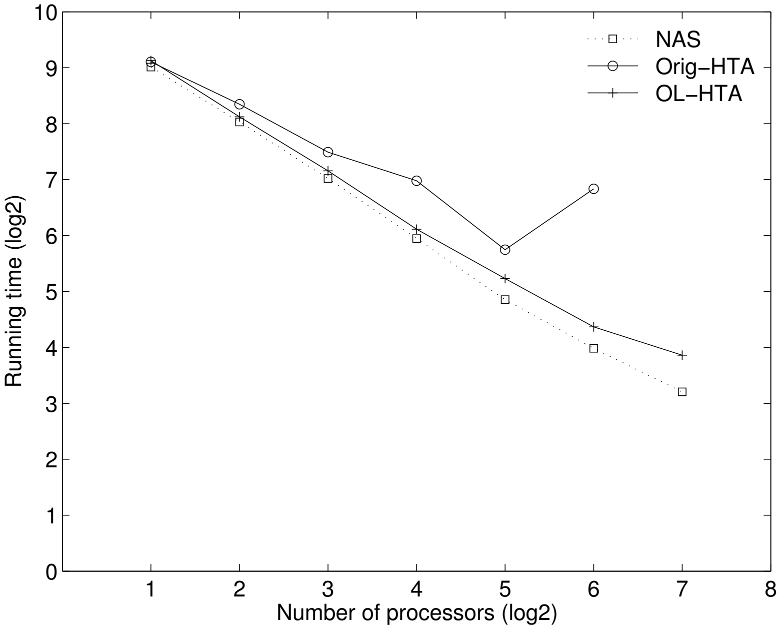


Figure 7.17: Performance of parallel LU benchmark class B

7.3.2 NAS LU Solver

The NAS LU solver solves the Navier-Stroke's equation for 3D regular grid. The Symmetric Successive Over Relaxation (SSOR) algorithm used by LU solver represents a different class of stencil computation which contains loop-carried cross-processor data dependences that sequentialize computations over distributed array dimensions. Such computations are called *pipelined* [29]. We use a wavefront algorithm [9] which can iterate the tiles in a special order

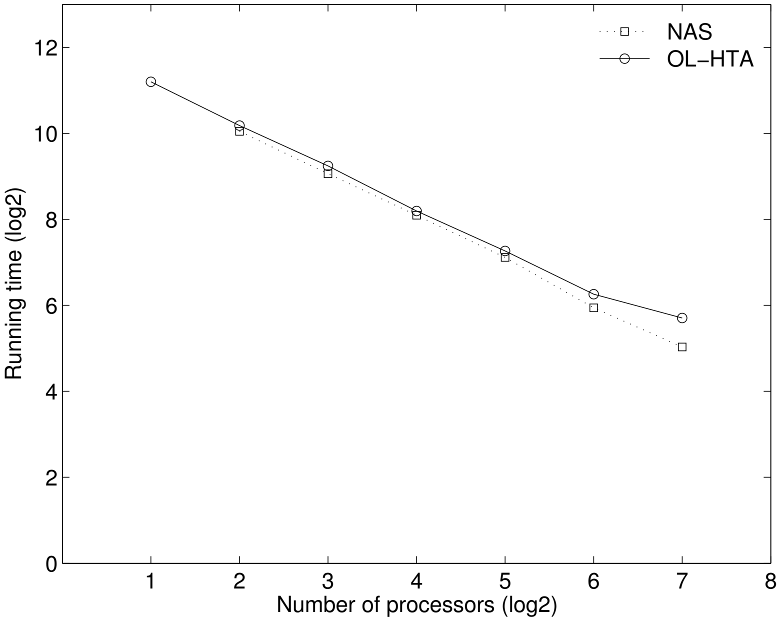


Figure 7.18: Performance of parallel LU benchmark class C

along the diagonal (or hyperplane for high dimensions).

Figure 7.15 shows the wavefront algorithm for 2D case. Tiles in the same color are selected and executed in parallel. The iteration moves toward the southeast direction along the diagonal. Then, shadow regions of the next band of tiles should be updated from their north and/or west neighbors. The HTA code is shown in Figure 7.16, where logical indexing is used to determine the tiles that can operate in each iteration of the `k` loop. `A` is a $m \times n$ HTA, distributed on a $m \times 1$ processor mesh, so that rows of tiles are mapped to the same processor. The logical indexing is created through function `fill(k)` in line 2 or 3. It selects the tiles, the addition of which indexes in all dimensions equals to `k`. For example, `fill(2)` selects `A(1,0)` and `A(0,1)` in Figure 7.15. The selected tiles will locally compute the 2D wavefront computation in lines 4-5. The next two statements copy the last row and column of tiles that finished the computation in iteration `k` to the first row and column of the tiles that are going to start the computation in the iteration $k+1$. Notice in htalib, the tile access operator "()" is used twice to select the tiles which both satisfy the logical indexing matrix and fall into the range in the second "()". Consequently, multiple conditions are used in tiling indexing to complete the complicated shadow region exchanges.

In wavefront algorithms, the explicit shadow region updates in different positions need updates from different neighbors. Our original HTA implementation of NAS LU Solver used a set of 54 indexing operations to perform this update in one iteration. On the other hand, with overlapped tiling, the complex indexing can be dropped. 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 Solver in is presented in Figure 7.17 for class B and Figure 7.18 for class C. 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.

7.4 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 to indicate the degree of readability for sequential programs. The three metrics are the programming effort [26], the cyclomatic number [36], and the source line of codes (SLOCs).

In software science [26], a program is considered a string of tokens, which can be divided into two groups: *operators* and *operands*. The *operators* are defined as variables or constants that the implementation employs. The *operators* are symbols or combinations of symbols that affect the value or ordering of operands. We denote η_1, the different number of operators; η_2, the different number of operands; N_1, the total number of occurrences of operators; and N_2, the total number of occurrences of operands. The program volume V is defined as

$$V = (N_1 + N_2) log_2(\eta_1 + \eta_2) \tag{7.1}$$

The potential volume V^* describes the shortest possible or most succinct form of an algorithm.

$$V^* = (N_1^* + N_2^*) log_2(\eta_1^* + \eta_2^*) \tag{7.2}$$

Now in the minimal form, neither operators nor operands could require repetition, thus

$$V^* = (\eta_1^* + \eta_2^*) log_2(\eta_1^* + \eta_2^*) \tag{7.3}$$

Furthermore, the minimum possible number of operators η^* for any algorithm is known. It must consist of one distinct operator for the name of the function and another to serve as an assignment or grouping symbol. Therefore,

$$\eta_1^* = 2$$

Equation 7.3 then becomes

$$V^* = (2 + \eta_2^*) log_2(2 + \eta_2^*) \tag{7.4}$$

where η_2^* should represent the number of different input/output parameters. Program level Lvl is defined as

$$Lvl = \frac{V^*}{V} \tag{7.5}$$

It follows that only the most succinct expression possible for an algorithm can have a level of unity. Programming effort E required to generate a given program should be

$$E = \frac{V}{Lvl} \tag{7.6}$$

A further implication of the effort equation can be shown by recalling equation 7.5 and substituting in equation 7.6

$$E = \frac{V^2}{V^*} \tag{7.7}$$

Equation 7.7 indicates that the mental effort required to implement any algorithm with a given potential volume should vary with the square of its volume in any language, rather linearly.

The second metric, the cyclomatic number [36], 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 C is given by

$$C = P + 1 \tag{7.8}$$

Again, the smaller the C, 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 7.3 shows the measurements for the three metrics for the three LU algorithms and the algorithm to solve triangular Sylvester equation. We show that the HTA programs achieve the least programming effort, the lowest cyclomatic number and the least number of lines. One of the interesting observations is all the non-HTA programs have a 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, "," are 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 the number of indexing operations. For example, arithmetic operations and assignments apply to the owned region of each tile. Using overlapped tiling, the refer-

Programs	η_1	N_1	η_2	N_2	E	C	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
Sylvester HTA	23	323	41	272	423,404	6	47
Sylvester FLAME	25	375	82	270	700,629	6	95

Table 7.3: Summary of three metrics: programming effort (E), the cyclomatic number (C) 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 7.4: Code difference between original (Orig) and overlapped (OL) HTA programs

ence 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 7.4 illustrates the reduction of communication statements and indexing operations for NAS LU and MG.

Chapter 8

Related Work

Tiling [32] has been extensively studied and used as a parallel execution enabler/optimizer [40] [48] and as a mechanism to improve locality [47]. 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 some compilers can apply tiling themselves, the analysis required are non-trivial, particularly when dealing with parallel programs and they are mostly limited to the exploitation of locality in affine codes [35]. Besides, it is often the case that the user needs to address the tiles directly to express an algorithm [7]. The result of the relevance of tiling and the lack of straightforward support for it in these languages has been the generation of programs that trade efficiency for code bloating and lack of readability to several degrees.

In this chapter, we introduce several research efforts using programming language approach or library approach to exploit the parallelism and locality in the program.

8.1 Language approaches

8.1.1 HPF and Co-Array Fortran

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 [28] data mapping directives for block and block-cyclic distribution or UPC shared arrays [13]. It constrains the programmer from exercising the fine-grained control over the code generated once the source of the performance bottleneck was identified [33].

Others, like Co-Array Fortran [38] allow to 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.

8.1.2 Sequoia

Sequoia [18] is a programming language designed to facilitate the development of memory hierarchy aware parallel programs that remain portable across modern machines with different memory hierarchy configurations.

It deserves special mention, as it is the only approach whose tiling is not data-centric. Instead its main focus is on task decomposition; thus Sequoia works on a hierarchy of tasks rather than a hierarchy of arrays. This way concepts such as the overlapped tiling are foreign to Sequoia.

8.2 Library approaches

8.2.1 POOMA

Libraries like POOMA [41] integrate their classes in a host OO language, and exploit operator overloading and polymorphism in order to provide a global view of the data and a single-threaded view of the computation, as our HTA library. However, POOMA differs in fundamental ways from our approach. For example, while POOMA's arrays can be distributed in tiles, the library provides no easy means to explicitly refer to those tiles. Also, hierarchical decomposition is not natural to POOMA's arrays, while it is a defining property of HTAs.

POOMA also allows to define internal guard layers (shadow regions) between the patches (tiles) that are updated automatically, i.e, their automatic update cannot be disabled. Separate external guard layers (boundaries) can be defined too, although the availability of automatically updated periodic boundaries depends on the kind of container used, as POOMA provides several containers with different properties. Also, no asynchronous update functions are provided. Still, POOMA's most important difference with respect to htalib is that its tiles and shadow regions are largely hidden from the user once they are created. This is because POOMA's patches are not used to express computation and communication as HTA tiles do. Instead, POOMA mostly accesses and uses its arrays as a whole unit. It would be difficult to write *pipelined* stencil computations as those shown in LU.

8.2.2 FLAME

FLAME [8] provides APIs to help express its tile-based algorithms in different languages. Two examples of FLAME API are shown in Figure A.1 and Figure A.2 in the Appendix. In the iterative dynamic LU code in Figure A.1 for example, the dynamic partitioning functions includes six function calls for different tiling structures, while in HTA code in Figure 5.3, two functions are used for dynamic partitioning. Further, in FLAME API code, the merge function `FLA_Merge_...()` has to be invoked before an operation is performed on several tiles. In HTA code, since the indexing allows the selection of a range of tiles, no merge function is needed. Finally, The indexing of HTA codes in dynamic partitioning is usually simple: it consists of several constants such as 0, 1, or 2. By contrast, FLAME API uses large number of arguments of some FLAME functions, which lead to large invocations and the usage of many variables. It implies a cost for the programmer to maintain various variables, some of which, although passed to functions, are not actually used in the function.

Our experiments suggest that HTAs are a more powerful way to express this kind of algorithms. The dynamic partitioning interface in HTA code is more general and concise. HTA also provides more features for tiled algorithms.

Chapter 9

Conclusion

9.1 Conclusion

In conclusion, tiled algorithms have received considerable attention motivated by optimizing for data locality as well as by exploiting parallelism. Many programming languages, such as Fortran or C, lack language constructs related to tiling and place the burden of manipulating tiles on the programmer side. Some compilers like HPF compiler try to apply tiling transformations themselves especially for parallelism, but fail to achieve high performance due to the complexity of the code [33]. Furthermore, programmers lose control over the the code so that it is difficult to identify and correct performance problems.

Our approach, extending the current OO language with HTA data type, provides a natural notation for tiles. We have observed that operations for tiles may lead to very regular code. It is ideal to be delegated to a library or compiler. The HTA data type enables a high level abstraction of tile operations and encapsulates the mechanic tile management such as shadow region manipulation, communications inside. Therefore, programmers can develop their code at tile level which matches the tiled algorithms. Table 9.1 summarizes the useful language constructs and their contributions for simplifying the implementation of tiled algorithms.

On the other hand, the library approach places the control at the programmer's hand. With well defined syntax and semantics in HTA operations, programmer can exercise fine grained control over the code. For instance, changing data layout for a specific HTA, disabling automatic update for shadow regions, or switching between static partitioning and

Language constructs	Encapsulation	Applications
tile index	index calculation	all tiled algorithms
data layout	index calculation, data movement	all tiled algorithms
assignment	communications	parallel programs
dynamic partitioning	tile topology reconstruction	FLAME, Cache-oblivious algorithms
overlapped tiling	shadow region management	stencil computations, sorting verification

Table 9.1: Summary of language constructs in HTA

dynamic partitioning.

The contributions of the thesis are as follows.

- *The support for tiling as a first class object.* Operators and basic functions are overloaded to support tiles. The operations between HTAs and other data types are formally defined. HTA interfaces to commonly encountered dense linear algebra routines in ATLAS and LAPACK are provided.

- *The efficient implementation for HTA runtime system.* An SPMD execution model is used to efficiently build the HTA runtime system in both MATLAB and C++. Many optimization approaches, such as template classes, specialized methods, inlining, lazy evaluation, relaxation of serial evaluation semantics are used in htalib to enhance the performance.

- *The strengths of HTAs to express the high degree of locality.* The tile indexing of a single or a range of tiles and the specification of data layout for tiles are considered useful weapons to express locality efficiently.

- *The design of two new language constructs for tiling.* Tiling is generalized with two new language constructs: the dynamic partitioning and overlapped tiling. The dynamic partitioning aims to match the actions described in certain algorithms while the overlapped tiling encapsulates the management of shadow regions in order to highlight the algorithm itself. The syntax and semantics are clearly defined and both of them have been used for a wide range of algorithms.

- *The study of productivity in HTA programs.* Three key metrics: the programming effort, the complexity of the code and the lines of codes, indicate the increased productivity for programs written in HTA. The performance of various benchmarks also demonstrated matched performance achieved using our approach.

Appendix A

FLAME API Implementations

```
FLA_Error FLA_LU_piv_blk_var5( FLA_Obj A, FLA_Obj p, FLA_LU_t* cntl )
{
  FLA_Obj ATL,   ATR,      A00, A01, A02,
          ABL,   ABR,      A10, A11, A12,
                           A20, A21, A22;

  FLA_Obj pT,              p0,
          pB,              p1,
                           p2;

  FLA_Obj AB0, AB1, AB2;

  int b;

  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,     0, 0, FLA_TL );

  FLA_Part_2x1( p,    &pT,
                      &pB,            0, FLA_TOP );

  while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) &&
          FLA_Obj_width( ATL ) < FLA_Obj_width( A )){

    b = FLA_Determine_blocksize( A, ATL, FLA_TL, FLA_Cntl_blocksize( cntl ) );

    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
                        /* ************* */   /* ******************** */
                                                &A10, /**/ &A11, &A12,
                           ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
                           b, b, FLA_BR );

    FLA_Repart_2x1_to_3x1( pT,                &p0,
                        /* ** */            /* ** */
                                              &p1,
                           pB,                &p2,        b, FLA_BOTTOM );

    /* *********************************************************************** */
    /* AB1 = / A11 \
             \ A21 / */
    FLA_Merge_2x1( A11,
                   A21,      &AB1 );

    /* AB1, p1 = LU_piv( AB1 ) */
    FLA_LU_piv_internal( AB1, p1,
                         FLA_Cntl_sub_lu( cntl ) );

    /* AB0 = / A10 \
             \ A20 / */
    FLA_Merge_2x1( A10,
                   A20,      &AB0 );

    /* Apply computed pivots to AB0 */
    FLA_Apply_pivots( FLA_LEFT, FLA_NO_TRANSPOSE, p1, AB0 );

    /* AB2 = / A12 \
             \ A22 / */
    FLA_Merge_2x1( A12,
                   A22,      &AB2 );

    /* Apply computed pivots to AB2 */
    FLA_Apply_pivots( FLA_LEFT, FLA_NO_TRANSPOSE, p1, AB2 );

    /* A12 = trilu( A11 ) \ A12 */
    FLA_Trsm_internal( FLA_LEFT, FLA_LOWER_TRIANGULAR,
                       FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
                       FLA_ONE, A11, A12,
                       FLA_Cntl_sub_trsm1( cntl ) );

    /* A22 = A22 - A21 * A12 */
    FLA_Gemm_internal( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
                       FLA_MINUS_ONE, A21, A12, FLA_ONE, A22,
                       FLA_Cntl_sub_gemm1( cntl ) );

    /* *********************************************************************** */

    FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,       A00, A01, /**/ A02,
                                                     A10, A11, /**/ A12,
                            /* ************** */  /* ****************** */
                              &ABL, /**/ &ABR,       A20, A21, /**/ A22,
                              FLA_TL );

    FLA_Cont_with_3x1_to_2x1( &pT,                p0,
                                                  p1,
                            /* ** */           /* ** */
                              &pB,                p2,     FLA_TOP );

  }

  return FLA_SUCCESS;
}
```

Figure A.1: The FLAME API implementation for the iterative dynamic LU algorithm.

```
int FLA_Syl_C3( FLA_Obj A, FLA_Obj B, FLA_Obj C, int nblks, int *bm_algs, int *bn_algs )
{
  FLA_Obj
    ATL, ATR,   A00,  A01,     A02,
    ABL, ABR,   A10,  A11,     A12,
                A20,  A21,     A22,
    BTL, BTR,   B00,  B01,     B02,
    BBL, BBR,   B10,  B11,     B12,
                B20,  B21,     B22,
    CTL, CTR,   C00,  C01,     C02,
    CBL, CBR,   C10,  C11,     C12,
                C20,  C21,     C22;

  int value=FLA_SUCCESS, bm, bn, bm_alg, bn_alg;

  if( ( FLA_Obj_width( C ) ==  0 )||( FLA_Obj_length( C ) == 0 ))
    return value;

  bm_alg = bm_algs[ 0 ];
  bn_alg = bn_algs[ 0 ];

  FLA_Part_2x2( A,  &ATL, /**/ &ATR,
             /* ************** */
    &ABL, /**/ &ABR,
            /* with */ 0, /* by */ 0, /* submatrix */ FLA_BR );

  FLA_Part_2x2( B,  &BTL, /**/ &BTR,
             /* ************** */
    &BBL, /**/ &BBR,
            /* with */ 0, /* by */ 0, /* submatrix */ FLA_TL );

  FLA_Part_2x2( C,  &CTL, /**/ &CTR,
             /* ************** */
    &CBL, /**/ &CBR,
            /* with */ 0, /* by */ 0, /* submatrix */ FLA_BL );

  while ( ( FLA_Obj_length( BBR ) != 0 )||( FLA_Obj_length( ATL ) != 0 ) ){
    bm = min( bm_alg, FLA_Obj_length( ATL ) );
    bn = min( bn_alg, FLA_Obj_length( BBR ) );

    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,       &A00,  &A01,    /**/  &A02,
                /**/            &A10,  &A11,    /**/  &A12,
                        /* ************* */   /* ************************* */
                           ABL, /**/ ABR,       &A20,  &A21,    /**/  &A22,
            /* with */ bm, /* by */ bm, /* A11 split from */ FLA_TL );

    FLA_Repart_2x2_to_3x3( BTL, /**/ BTR,       &B00,  /**/ &B01,     &B02,
                        /* ************* */   /* ************************* */
                /**/            &B10,  /**/ &B11,     &B12,
                           BBL, /**/ BBR,       &B20,  /**/ &B21,     &B22,
            /* with */ bn, /* by */ bn, /* B11 split from */ FLA_BR );

    FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,       &C00,  /**/ &C01,     &C02,
                /**/            &C10,  /**/ &C11,     &C12,
                        /* ************* */   /* ************************* */
                           CBL, /**/ CBR,       &C20,  /**/ &C21,     &C22,
            /* with */ bm, /* by */ bn, /* B11 split from */ FLA_TR );

    /* ********************************************************************* */
/* C10 <- X10 where X10 solves A11 X10 + X10 B00 = C10 */

 if ( nblks == 1 ){
   //      FLA_Obj_show("C10: ", C10, "%f", "---");
   FLA_Syl_level2( A11, B00, C10 );

 }
 else
   FLA_Syl_C3( A11, B00, C10, nblks-1, &bm_algs[ 1 ], &bn_algs[ 1 ] );


 /* C21 <- X21 where X21 solves A22 X21 + X21 B11 = C21 */
 if ( nblks == 1 )
   FLA_Syl_level2( A22, B11, C21 );
 else
   FLA_Syl_C3( A22, B11, C21, nblks-1, &bm_algs[ 1 ], &bn_algs[ 1 ] );

 /* C11 <- - A12 X21 + C11 */
 FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
   FLA_MINUS_ONE, A12, C21, FLA_ONE, C11 );

 /* C11 <- - X10 B01 + C11 */
 FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
   FLA_MINUS_ONE, C10, B01, FLA_ONE, C11 );

 /* C11 <- X11 where X11 solves A11 X11 + X11 B11 = C11 */
 if ( nblks == 1 )
   FLA_Syl_level2( A11, B11, C11 );
 else
   FLA_Syl_C3( A11, B11, C11, nblks-1, &bm_algs[ 1 ], &bn_algs[ 1 ] );


 /* C00 <- - A01 X10 + C00 */
 FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
   FLA_MINUS_ONE, A01, C10, FLA_ONE, C00 );

 /* C01 <- - A01 X11 + C01 */
 FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
   FLA_MINUS_ONE, A01, C11, FLA_ONE, C01 );

 /* C01 <- - A02 X21 + C01 */
 FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
   FLA_MINUS_ONE, A02, C21, FLA_ONE, C01 );

 /* C22 <- - X21 B12 + C22 */
 FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
   FLA_MINUS_ONE, C21, B12, FLA_ONE, C22 );

 /* C12 <- - X10 B02 + C12 */
 FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
   FLA_MINUS_ONE, C10, B02, FLA_ONE, C12 );

 /* C12 <- - X11 B12 + C12 */
 FLA_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
   FLA_MINUS_ONE, C11, B12, FLA_ONE, C12 );

 /* ********************************************************************* */
```

```
    FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,      A00, /**/ A01,      A02,
                            /* ************** */ /* ************************ */
                                    /**/            A10, /**/ A11,      A12,
                              &ABL, /**/ &ABR,      A20, /**/ A21,      A22,
            /* with A11 added to submatrix */ FLA_BR );

    FLA_Cont_with_3x3_to_2x2( &BTL, /**/ &BTR,      B00,  B01,     /**/ B02,
                                    /**/            B10,  B11,     /**/ B12,
                            /* ************** */ /* ************************ */
                              &BBL, /**/ &BBR,      B20,  B21,     /**/ B22,
            /* with B11 added to submatrix */ FLA_TL );

    FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,      C00,  C01,     /**/ C02,
                            /* ************** */ /* ************************ */
                                    /**/            C10,  C11,     /**/ C12,
                              &CBL, /**/ &CBR,      C20,  C21,     /**/ C22,
            /* with C11 added to submatrix */ FLA_BL );
  }

  return value;
}
```

Figure A.2: The FLAME API implementation for the triangular Sylvester equation.

References

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

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

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

[4] Vikram Adve and John Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. In *Proc. of the ACM SIGPLAN 1998 conf. on Programming language design and implementation*, pages 186–198. ACM Press, 1998.

[5] 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.

[6] R. H. Bartels and G. W. Stewart. Solution of the matrix equation ax + xb = c [f4]. *Commun. ACM*, 15(9):820–826, 1972.

[7] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[8] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: the flame application program interfaces. *ACM Trans. Math. Softw.*, 31(1):27–59, 2005.

[9] Ganesh Bikshandi. *Parallel Programming with Hierarchically Tiled Arrays*. PhD thesis, University of Illinois at Urbana-Champaign, 2007.

[10] Ganesh Bikshandi, Jia Guo, Dan Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua, and Christoph von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, pages 48–57, 2006.

[11] Ganesh Bikshandi, Jia Guo, Christoph von Praun, Gabriel Tanase, Basilio B. Fraguela, María J. Garzarán, David Padua, and Lawrence Rauchwerger. Design and use of htalib - a library for hierarchically tiled arrays. In *Proc. of the International Workshop on Languages and Compilers for Parallel Computing*, November 2006.

[12] Gianfranco Bilardi, Paolo D'Alberto, and Alexandru Nicolau. Fractal matrix multiplication: A case study on portability of cache performance. In *WAE '01: Proceedings of the 5th International Workshop on Algorithm Engineering*, pages 26–38, London, UK, 2001. Springer-Verlag.

[13] 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.

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

[15] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[16] Siddhartha Chatterjee, Vibhor V. Jain, Alvin R. Lebeck, Shyam Mundhra, and Mithuna Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *International Conference on Supercomputing*, pages 444–453, 1999.

[17] B. Chamberlain et al. ZPL's WYSIWYG Performance Model. In *Procs. of the High-Level Parallel Programming Models and Supportive Environments*, pages 50–61. IEEE Computer Society, 1998.

[18] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: programming the memory hierarchy. In *Proc. of Supercomputing'06*, page 83, 2006.

[19] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving problems on concurrent processors. Vol. 1: General techniques and regular problems.* Prentice-Hall, Inc., 1988.

[20] B. B. Fraguela, M. G. Carmueja, and D. Andrade. Optimal tile size selection guided by analytical models. In *Procs. of Parallel Computing 2005 (ParCo 2005).*, pages 565–572, Malaga, Spain, Sept 2005.

[21] B.B. Fraguela, R. Doallo, and E.L. Zapata. Memory hierarchy performance prediction for blocked sparse algorithms. *Parallel Processing Letters*, 9(3):347–360, September 1999.

[22] Matteo Frigo and et al. Cache-oblivious algorithms (extended abstract).

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

[24] Manish Gupta, Sam Midkiff, Edith Schonberg, Ven Seshadri, David Shields, Ko-Yang Wang, Wai-Mee Ching, and Ton Ngo. An hpf compiler for the ibm sp2. In *Proc. of Supercomputing '95*, page 71. ACM Press, 1995.

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

[26] Maurice H. Halstead. *Elements of Software Science.* Elsevier, New York, 1977.

[27] H. Han, G. Rivera, and C. Tseng. Software support for improving locality in scientific codes. In *Proc. of the Eighth International Workshop on Compilers for Parallel Computers (CPC'2000)*, Aussois, France, Jan 2000.

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

[29] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for fortran d on mimd distributed-memory machines. In *Proc. of Supercomputing '91*, pages 86–100. ACM Press, 1991.

[30] P. Husbands and C. Isbell. Matlab*p: A Tool for Interactive Supercomputing. In *Procs. of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[31] Eun-Jin Im. *Optimizing the performance of sparse matrix-vector multiplication.* PhD thesis, 2000. Chair-Katherine A. Yelick.

[32] 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.

[33] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of high performance fortran: an historical object lesson. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 7–1–7–22, New York, NY, USA, 2007. ACM Press.

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

[35] Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proc. of the 8th ACM SIGPLAN Symp. on Principles and Practice of parallel Programming (PPoPP '01)*, pages 103–112, 2001.

[36] McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[37] A. C. McKellar and Jr. E. G. Coffman. Organizing matrices and matrix operations for paged memory systems. *Commun. ACM*, 12(3):153–165, 1969.

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

[39] Enrique S. Quintana-Ortí and Robert A. van de Geijn. Formal derivation of algorithms: The triangular sylvester equation. *ACM Trans. Math. Softw.*, 29(2):218–243, 2003.

[40] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for non-shared memory machines. In *Proc. of Supercomputing'91*, pages 111–120, 1991.

[41] John V. W. Reynders, Paul J. Hinker, Julian C. Cummings, Susan R. Atlas, Subhankar Banerjee, William F. Humphrey, Steve R. Karmesin, Katarzyna Keahey, Marikani Srikant, and Mary Dell Tholburn. POOMA: A Framework for Scientific Simulations of Paralllel Architectures. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming in C++*, pages 547–588. MIT Press, 1996.

[42] Hanan Samet. *The design and analysis of spatial data structures.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[43] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[44] Sivan Toledo. Locality of reference in *LU* decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[45] A. E. Trefethen, V. S. Menon, C. Chang, G. Czajkowski, C. Myers, and L. N. Trefethen. MultiMATLAB: MATLAB on Multiple Processors. Technical Report TR96-1586, May 1996.

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

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

[48] Jingling Xue. *Loop tiling for parallelism.* Kluwer Academic Publishers, 2000.

[49] Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnels, and Fred Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 93–104, New York, NY, USA, 2007. ACM Press.

Author's Biography

Jia Guo was born in Shashi, Hubei province, China but moved to Beijing when she was eleven months old. She developed her interests in literature and mathematics in high school and won many national prizes in writing and math competitions. She was ranked 9th among female students in the entrance examination in Beijing.

In 1997, she joined Tsinghua University in Beijing and was awarded the first class of outstanding student scholarship for four consecutive years. In 2001, she received a B.E. degree in Computer Science from Tsinghua University.

She continued her graduate study at the University of Illinois at Urbana-Champaign. In 2004, she earned an M.S. degree in Computer Science. Her master thesis is on the Effectiveness of Register Allocation on Long Basic Blocks. Then she and her colleagues built the HTA runtime system. She began to investigate the useful language constructs to facilitate the programming for locality and parallelism, which became her Ph.D. thesis topic. She also worked for IBM T.J. Watson Research Center as a summer intern and won an award for excellent student internship.