Design and Use of htalib – a Library for Hierarchically Tiled Arrays[⋆]

Ganesh Bikshandi[1], Jia Guo[1], Christoph von Praun[2], Gabriel Tanase[3], Basilio B. Fraguela[4], María J. Garzarán[1], David Padua[1] and Lawrence Rauchwerger[3]

[1]University of Illinois, Urbana-Champaign, IL
[2]IBM T. J. Watson Research Center, Yorktown Heights, NY
[3]Texas A&M University, College Station, TX
[4]Universidade da Coruña, Spain

Abstract. Hierarchically Tiled Arrays (HTAs) are data structures that facilitate locality and parallelism of array intensive computations with block-recursive nature. The model underlying HTAs provides programmers with a global view of distributed data as well as a single-threaded view of the execution. In this paper we present htalib, a C++ implementation of HTAs. This library provides several novel constructs: (i) *A map-reduce operator framework* that facilitates the implementation of distributed operations with HTAs. (ii) *Overlapped tiling* in support of tiling in stencil codes. (iii) *Data layering*, facilitating the use of HTAs in adaptive mesh refinement applications. We describe the interface and design of htalib and our experience with the new programming constructs.

1 Introduction

1.1 Hierarchically Tiled Arrays

A Hierarchically Tiled Array (HTA) [7, 4] is a recursive array data type where elements are either HTAs or scalars (at the bottom of the recursion). HTAs adopt tiling as a first class construct for array-based computations and empower programmers to control data distribution and the granularity of computation explicitly through the specification of tiling [2, 8, 9, 16, 18]. An HTA has the conventional array functionality: scalar access, pointwise operators, and assignment. The functionality of HTAs goes a significant step beyond mere arrays: HTAs provide a rich set of generic, block-recursive operations that execute with high efficiency in sequential or parallel manner. When programming with HTAs, a programmer seeks to harness the built-in operators or develop new operators by extending the generic framework and HTA-specific functionality.

[⋆] This work was supported by the National Science Foundation under the NGS program (Grant No. 0103610) and under the CSR-AES program (Grant No. 0509432). Basilio Fraguela was partially supported by the Ministry of Education and Science of Spain, FEDER funds of the European Union (Project TIN2004-07797-C02-02), and by the Galician Government (Project PGIDIT03-TIC10502PR). This work was also supported by the Defense Advanced Research Projects Agency (Contract NBCH30390004).

HTAs are not only a library construct but also a programming model: HTAs provide a global shared memory abstraction, and support (encourage) the programmer to structure algorithms in a block-recursive manner, yet following sequential program logic. This programming style can be efficiently mapped onto todays parallel and distributed computer architectures and memory hierarchies using standard compiler and communication systems. Unlike approaches that are entirely controlled by the compiler [2] or integrated into a specific programming language [9], HTAs offer an attractive programming model and performance while preserving the convenience of standard tools, and libraries (library-based approach).

In earlier work, we introduced the concepts of programming with HTAs [4] and an early prototype based on MATLAB [7]. This paper describes htalib, a portable C++ library and framework for HTAs, and three application-driven extensions to the original HTA proposal [4] that facilitate the use of HTAs in certain application domains and broaden the potential application scope. We report on our experience with HTAs on an IBM BlueGene/L system.

1.2 Contributions

We present the design and implementation of an operator framework for HTAs following the principles of *map-reduce* [12] that we illustrate with examples from the NAS kernel programs.

Overlapped tiling, a mechanism for implicit allocation and consistency of shadow regions and ghost cells. It provides flexible indexing scheme for HTA tiles and facilitates access to neighbor elements of adjacent tiles, a common access pattern in stencil computations. Overlapped tiling also provides a clean syntax.

Data layering, an extension of htalib where a hierarchy of scalar arrays (not just a single array) can be controlled and accessed through one HTA. Data layering makes HTAs a highly expressive and compact data structure for multigrid and AMR (Adaptive Mesh Refinement) [6, 17] applications.

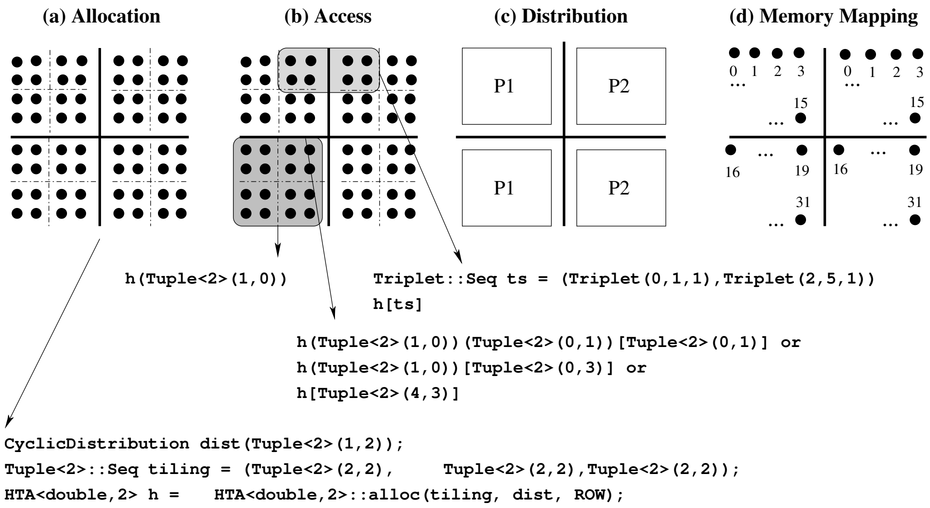
2 Design and Use of htalib

2.1 Overview

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

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.

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 [7]. An HTA is part of a hierarchical structure of recursively composed HTAs. The *root* of the hierarchy is designated as level 0, with its tiles at level 1, etc..



```
h(Tuple<2>(1,0))
```

```
Triplet::Seq ts = (Triplet(0,1,1),Triplet(2,5,1))
h[ts]
```

```
h(Tuple<2>(1,0))(Tuple<2>(0,1))[Tuple<2>(0,1)] or
h(Tuple<2>(1,0))[Tuple<2>(0,3)] or
h[Tuple<2>(4,3)]
```

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

Fig. 1: HTA allocation, access, distribution and memory mapping.

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.

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 Section 3.

2.2 Example

In Figure 1, the distribution of the HTA occurs at the root of the tiling hierarchy in a cyclic manner over 2 processors along the second dimension (columns). At each processor, the scalar array corresponding to the tiles is allocated in row-major layout (spawning sub-tiles). The memory mapping in Figure 1 illustrates the logical index of the scalar variables in each processor.

To simplify the presentation in subsequent examples, the C++ code is slightly modified: Instead of the instances `Tuple<DIM>(x,y,...)` and `Triplet(low,high,step)`, we use a comma separated list of integers of the form

```
HTA<double,1> A, B;
double S = 0.125;
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]);
  ...
}
```

(a)

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

(b)

Fig. 2: HTA examples (a) HTA code for 1D Jacobi with one level of tiling. (b) Relaxing sequential evaluation order to facilitate overlap of communication and computation.

`(x,y,...)` to represent coordinate points, and the `low:high:step` triplet notation. Regions are constructed from sequences of triplets as in Figure 1. Template arguments are elided when their value is apparent from the context.

Figure 2(a) illustrates a Jacobi computation with HTAs. A and B are HTAs with one level of tiling; there are n tiles at the root of the tiling hierarchy (level 0), each tile holding d+2 variables (level 1). Variables at index 0 and d+1 in each tile are ghost cells. The boundary exchange first updates the ghost cells at index 0, then at index d+1. The iteration across tiles is implicit in all assignments. In the stencil computation, the region is not specified at tile access and thus all tiles at level 0 are considered in the operation. The example illustrates that scalars and arrays can be combined as operator arguments; htalib follows Fortran90 [3] conventions for conformability and automatically promotes scalars to arrays in expressions.

2.3 Programming Model

The programming model supported by htalib has the following key properties:

Global view. Remote and local data are accessed though the same syntax and address space. In the Jacobi computation in Figure 2(a), the tiles of the HTA could be distributed, yet scalars and tiles are seamlessly accessed within one global logical index space.

Implicit communication. Data is communicated when necessary as part of the evaluation of one of the following constructs: array assignment, certain array transformations, and when materializing temporary arrays during the evaluation of complex array expressions (spilling).

Serial program logic. Statements and expressions that involve arrays are evaluated according to serial semantics, in particular array assignment follows Fortran90 conventions [3]. Parallelism is achieved when data-parallel array operators are evaluated in a block-recursive manner following the tile distribution.

Automated memory management. The implementation maintains reference counts for HTAs and associated structures, facilitating automatic de-allocation. Tuples, triplets and sequences thereof have a very light-weight implementations and are, by convention, allocated on the stack or inlined in objects.

2.4 Implementation

This section briefly describes four key implementation aspects of htalib:

```
struct plus {
  double operator() (const double a,
                     const double b) {
    return a+b;
   } }
```

```
struct ft {
 void operator() (Array* x) {
     //...
     mkl.dftForward(x);
     //...
  } }
```

Fig. 3: Primitive Operators

Owner computes. At allocation time, the top-level tiling of an HTA determines the data distribution, i.e., each tile is assigned a *home* location where the master copy of the scalar data is allocated. The computation of an array expression is distributed among the owners of tiles that receive the result of the expression. Arguments data is communicated when necessary.

SPMD computation and communication. The execution and communication mechanisms inside htalib follows the SPMD principle. The communication of tiles or part of tiles is based on two-sided message passing (MPI).

Dynamic optimizations. htalib implements *lazy evaluation* to reduce or avoid the overhead due to temporary arrays. At an array assignment, the evaluation of the rhs is delayed until the target of the assignment is determined. If lhs and rhs have no data dependence, the assignment is directly evaluated into the lhs.

Relaxation of serial evaluation semantics. htalib provides a mechanism to temporarily relax the serial evaluation ordering and overlap of communication with computation. The example in Figure 2(b) shows the boundary exchange in the Jacobi example in Figure 2(a). As there is no data dependence among the assignments, both statements can proceed concurrently. This is achieved through the runtime calls to `asyc` and `sync`. Similarly, the runtime system permits to selectively disable strict evaluation order for array assignment and resort to split-phase semantics [11].

3 Operator Framework

3.1 Primitive Operators

htalib defines several primitive operators over the scalar and array domain. To provide an uniform interface, all these operators are wrapped in STL like functor objects that define method `operator()` with appropriate arguments. Primitive operators are applied to HTAs directly or through *high-order* operators (Section 3.2).

htalib includes primitive operators for standard arithmetic, logical, relational, vector and matrix operations. For example, in the Figure 3, `plus` is the addition operation defined on scalars, while `ft` is *Fourier Transform* defined on an array.

3.2 Higher-order Operators

htalib provides the following higher-order operators: `map`, `reduce`, `scan`. Higher-order operators are parametrized with primitive operators and define the strategy and result format resulting form the application of primitive operators to the tiles or scalar values in an HTA.

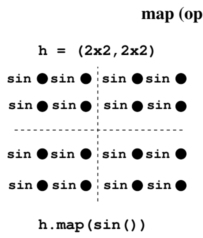


Fig. 4: HTA `map` operator

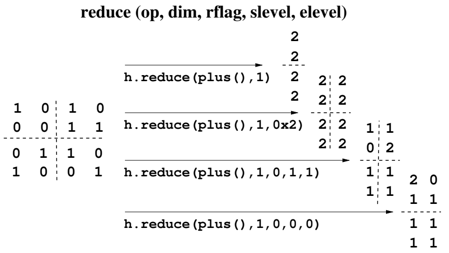


Fig. 5: HTA `reduce` operator

Map `map` applies a function *f* to each one of the elements of the input HTA at a given level. The syntax of `map` is shown in Figure 4, along with examples. In the figure, `h = (n` × `m ,..., x` × `y)` indicates the region of each level of the HTA `h` from leaf to root. `map` takes as input a primitive operator *op*, and a level *mlevel*. The *mlevel* specifies the level of the HTA at which *op* will be invoked. The default value of *mlevel* is $LEAF\_LEVEL$. Invocation of `map` on HTA `h`, results in the mapping function being applied to the top level of `h`. If its elements are tiles themselves, then the function is invoked recursively on each of their elements until $level = LEAF\_LEVEL$ or $level = mlevel$. The `op` is invoked over the elements at that level.

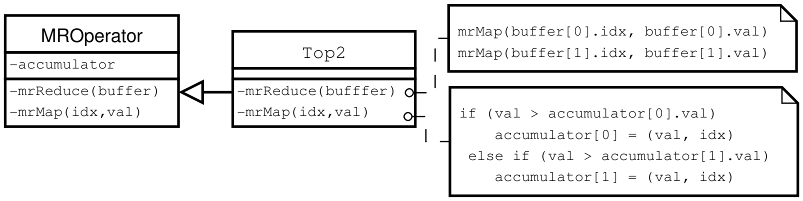
The method *op* is required to be a *PURE* function. The result of `map` is an HTA with identical index space and tiling. Though not shown in the figure, `map` can also take one or more HTAs as its argument, along with *op*. The argument HTAs should be conformable with the receiver of the method. `map` forms the basis for several point-wise HTA operations. For example, scalar addition of two HTAs, `h1` and `h2`, is implemented using `map` as follows: `r = h1.map(plus(), h2)`;

The order of iteration in the `map` does not affect the result of the computation. During its application, there is no communication between the sibling elements (i.e. elements at the same level) of an HTA. Thus, `map` is a useful abstraction for *do-all* parallelism. A `map` invocation on a distributed HTA is an equivalent of a *do-all* parallel loop.

Reduce An operation which is applied to all components of a vector to produce a scalar is called a *reduction*. For example, $reduce(+, x)$ is *sum* and $reduce(\times, x)$ is *product* of a vector x. The reduction operation can be generalized for n-dimensions, resulting in an (n-1)-dimensional space.

The reduction operation is extended to HTAs in the form of the `reduce` operator. Like `map`, `reduce` is also a recursive operation. The dimension of the elements at each level will be reduced by one. However, it is possible to control the starting and ending level of the reduction operation. The HTA reductions are parameterized by the following arguments:

- *op*: This is any associative operation from the primitive operator set.
- *dim*: This is the dimension of reduction.



```
mrMap(buffer[0].idx, buffer[0].val)
mrMap(buffer[1].idx, buffer[1].val)
```

```
if (val > accumulator[0].val)
    accumulator[0] = (val, idx)
 else if (val > accumulator[1].val)
    accumulator[1] = (val, idx)
```

Fig. 6: `MROperator` and `Top2`

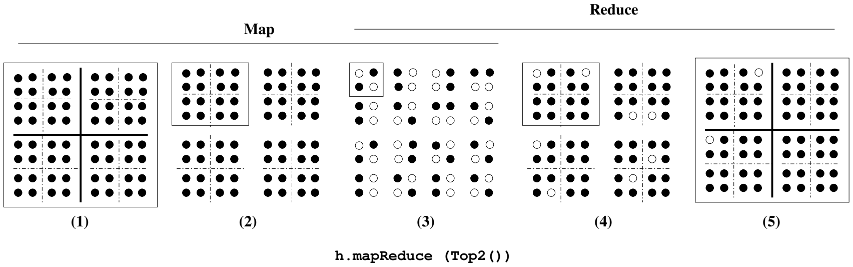


Fig. 7: `mapReduce` operation

- *rflag*: This is a bit vector, whose i^{th} bit, if set to true, will imply replication of the result along `dim`, for level i. The resulting HTA will have the same number of dimensions as the input HTA, at the level i. This is the equivalent of an *all-to-all* reduction. Default value is set to 0 for all dimensions.
- *slevel*: This is the starting level of the reduction. Default value is *LEAF_LEVEL*.
- *elevel*: This is the ending level of the reduction. Default value is *ROOT_LEVEL*. For example, if *elevel* = *LEAF_LEVEL* and the HTA has one-level of tiling, then the reduction is applied over each leaf tiles of the HTA. Also, *elevel* ≤ *slevel*.

For example, if the HTA has a single level of tiling and *slevel* = *elevel* = 1, then the scalar elements of the leaf tiles are reduced along *dim*. If *slevel* = *elevel* = 0, the tiles of the top level HTA are reduced along *dim*. These two cases occur in the IS program of NAS benchmark suite, and an example is shown in the third and fourth row in Figure 5.

Scan `scan` computes the reductions of all prefixes of a vector. For example, if the vector $v = (1, 2, 3, 4, 5)$, then $\texttt{scan}(+, v) = (1, 3, 6, 10, 14)$. htalib provides a similar `scan` operation for HTAs. The syntax is very similar to that of `reduce`. However, unlike `map` and `reduce`, `scan` is an ordered computation; the result is dependent on the order of iteration over the elements. `scan` represents *do-across* parallelism. Owing to space restrictions, `scan` is not explained in detail.

MapReduce Certain computations require composition of `map` and `reduce`. `mapReduce` is a higher-order operator that combines the two steps in an efficient manner. `mapReduce` takes as input instances of a map-reduce operator.

The implementation of this map-reduce operator is based on class MROperator with two key abstract methods, mrMap and mrReduce, and a generic variable accumulator that holds at first intermediate data and finally the results of the operator evaluation.

To implement an application specific map-reduce operator, programmers must extend MROperator and provide implementations for mrMap and mrReduce. Figure 6 shows the MRoperator and a subclass, Top2, which finds the top 2 values and their indices from a numerical matrix. A programmer only implements the core sequential logic: the code for accumulating the results in the accumulator in mrMap, and the code for combining two accumulators in mrReduce. The invocation of mrMap with arguments, and the parallel reduction strategy and data exchange are the responsibility of htalib.

Figure 7 illustrates the semantics of mapReduce invoked with an instance of Top2. In the figure, the mapReduce is applied to an HTA with two levels of tiling. A solid rectangle is used to show the scope of mapReduce at each stage. mapReduce is evaluated recursively across the tile hierarchy down to a specified level, i.e. method mapReduce is recursively invoked on each tile (step 1-2). When the bottom or the recursion is reached, the method mrMap is applied to each scalar of the tiles at this level (step 3). This is followed by calls to mrReduce as the operator evaluation ascends in the tile hierarchy (steps 3-5). In the figure, the top 2 elements chosen at each level are shown with white dots.

mapReduce offers better abstraction and performance benefits over applying map and reduce separately. Since the reductions involve an associative operation, potentially large intermediate buffers can be replaced by much smaller accumulators. This saves both memory usage and network bandwidth. Furthermore, several maps or reduces can be combined in to a single operation, eliminating several unwanted buffers.

4 Overlapped Tiling

Many scientific codes contain operations on several neighboring points within an array. A typical example is iterative PDE solvers based on finite difference techniques, also known as *stencil* codes. Jacobi is a simple example of an iterative PDE solver, which is similar to the stencil computation in NAS MG benchmark. These codes benefit from tiling, and thus from HTAs, both because it improves their locality [15] and because it can be used to distribute their data in order to parallelize them. However, the processing of the data located in the borders of the tiles requires accesses to neighboring data located in other tiles. The usual approach to deal with this problem in parallel codes is to resort to shadow regions or ghost cells that the programmer is responsible for keeping updated. Such situation is shown in Figure 2(a) for a 1D Jacobi code. There are two statements to update shadow regions of HTA B before the stencil computation. In htalib, we have opted for a cleaner approach: 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 neighbor tile. The HTA becomes responsible for creating shadow regions of storage if they are needed, and updating them as necessary.

4.1 Syntax

Creation When a tile T is defined, we say that the region it comprises is owned by T. The size of this owned region for each tile is given by the tiling parameter, the first parameter in HTA constructor in Figure 8(a). The regions that can be accessed across tile boundaries are defined as *shadow* regions. In particular, outside the region owned by T, are the extended regions that T can access. We call them *shadow-out* regions. Inside T, there are regions that other tiles are allowed to access. We call those *shadow-in* regions. For example, in Figure 8(b), we show the shadow-in and shadow-out regions for the first tile of A defined in Figure 8(a). The sizes of the shadow-in and shadow-out regions are given by a parameter of type Overlap. First, we show how to construct an object of type Overlap. The general form is,

```
Overlap<DIM> ol = Overlap<DIM>( T<DIM> negativeDir, T<DIM> positiveDir, boundaryMode mode);
```

The negativeDir specifies the amount of the overlap for each tile in the *negative direction* (decreasing index value) in each dimension. The positiveDir specifies the amount of the overlap for each tile in the *positive direction* (increasing index values). The mode parameter specifies the nature of the boundary regions in the original array with three options: zero, preset, periodic. The zero mode means shadow regions filled with all zeros will be allocated to the boundary of the array by htalib. The preset mode means the shadow regions for the array elements in the boundary have been allocated and preset by the programmer. The periodic mode means the shadow regions for the array elements satisfy the periodic boundary conditions. In the 1D array case, for instance, the last element of the array treats the first element as its neighbor. The Overlap object is used as the last parameter in the HTA constructor to create the HTA with overlapped tiling.

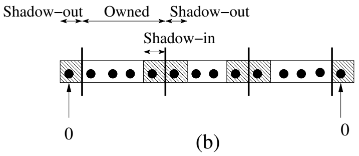
Figure 8(a) shows an example code that creates an 1×3 HTA A with 4 elements per tile. Each tile in the HTA overlaps one element in both directions. Around the boundaries of the array, the shadow regions are allocated by htalib with zeros. The pictorial view is shown in Figure 8(b). As a result of the zero mode overlap, the two shadow-out regions for the third tile are the last element of the second tile and the last zero element added to A's boundary by htalib.

Indexing The 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 range for the owned data starts from 0 to the maximum size of the tile in each dimension. The indexing for the shadow regions 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 direction with the length 2. Then, the region owned by T can be indexed as T[0:3]. The left shadow region can be indexed as T[-2:-1] and the right shadow region can be indexed as T[4:5]. We define the symbolic shape All to index all elements in the region owned by

```
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)

```
HTA<double, 1> A, B;
while (!converged){
    A()[All] = S* ( B()[All-1]+ B()[All+1]);
     ... ...
}
```

(c)

Fig. 8: Example of overlapped tiling. (a) The constructor for an HTA with tiles overlap to both directions and shadow regions in the boundary are inserted.(b) The pictorial view. (c) HTA code for 1D Jacobi with overlapped tiling

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].

Figure 8(c) shows the same code as Figure 2(a), but uses overlapped tiling. Compared with the original code, it helps simplify the program with only one statement in computation. Furthermore, the simpler indexing scheme makes the code more readable.

4.2 Shadow Region Consistency

When there is a write to the shadow region, every tile that has access to this region should be consistent with this change. The shadow region consistency problem is handled by htalib. Consistency is trivial for some data layouts such as row or column major. However, for some special data layouts such as distributed HTAs, proper updates should be performed by htalib in order to keep all copies of the overlapped region consistent.

We use the update-on-write policy to keep the shadow region consistent. Since every element only belongs to one owned region, we only allow that owner tile to perform the write to its data. Once the owner tile modifies the data, it updates the set of tiles whose shadow-out regions contain the data.

5 Data Layering

So far the hierarchical nature of HTAs serves to achieve data distribution and access locality. We now investigate HTAs for applications where the hierarchy reflects a property of the application domain. The key extension we propose for HTAs is to associate scalar data with different layers of an HTA (not just with the lowest layer in the hierarchy) to facilitate applications requiring mesh refinement.

When HTAs are used for locality or to control data distribution, data is stored only in the leaf tiles, while tiles above the leaves store only meta data about their children in the tiling hierarchy (see Figure 9(a)). We extend HTAs to provide support for refinement, by allowing any level in the hierarchy to store both data and tiling information about refined levels (see Figure 9(b)). For example, the

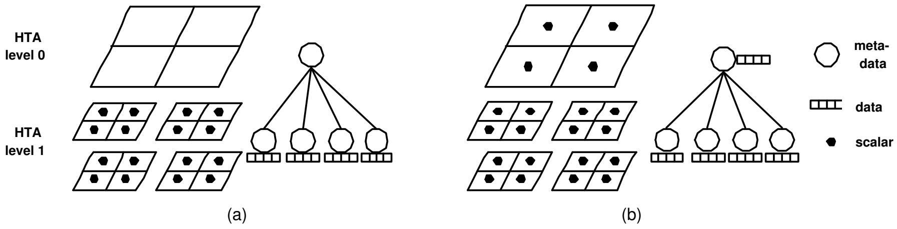


Fig. 9: (a) HTA with no data layering. (b) Multi layered HTA stores data at different levels of the hierarchy.

NAS benchmark MG [1] defines a set of grids that are successive refinements of an original grid, as it can be seen in Figure 10(a), where levels 0, 1 and 2 are shown. Every cell in a grid is expanded by a factor of two along all dimensions, at the next level. The set of grids defined by MG can be naturally represented as a hierarchy of tiled arrays.

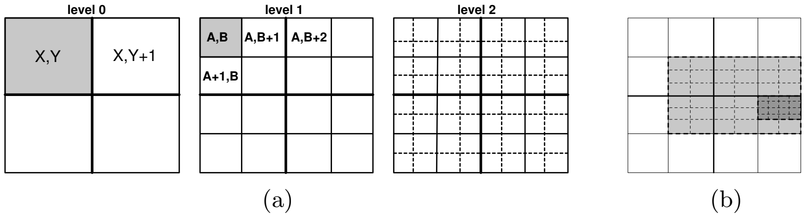


Fig. 10: Mesh Refinement (a) MG (b) Adaptive Mesh Refinement

5.1 HTA Support for Mesh Refinement in MG

Figure 11(a) shows code for a subset of functions in the NAS MG benchmark, based on HTAs without layering support. In this code, the user has to explicitly allocate space for the HTAs corresponding to different levels of refinement. Function `setup()`, in Figure 11(a), allocates the HTAs and stores them in an array. The position in the array corresponds to different refinement levels. The users will also have to explicitly store information on how data at a certain level relates to data in the refined and coarse grids. This information is used to communicate data between different levels. In Figure 11(a), function `interp()`, data from the coarse grid `z` is used to initialize data in the refined one `u`. The `(A,B)` and `(X,Y)` tuples specify how data elements between two levels correspond and this information has to be provided by the user. The explicit specification of the coordinate mapping is error prone and a tedious task for a programmer. We extend the HTA library to handle the refinement, and to automatically provide these mappings.

For the NAS MG benchmark, allocating the grids for the refined levels is relatively simple. Every cell is refined uniformly, in all directions by a constant factor. In a general adaptive mesh refinement problem there are extra complications that have to be addressed: When the refinement is not uniform, as shown in

```
#define Grid HTA
void setup(...) {
 for i = 0..num_levels {
  compute tiling level i
  array[i] = allocate HTA level i
 }
}

void interp (Grid z, Grid u ) {
  //get info about tiles
  Weight w;
  int iSize = ..
  int jSize = ..
  int kSize = ..
  Triplet X(0, iSize-2);
  Triplet Y(0, jSize-2);
  Triplet A(0, 2 * (iSize-2), 2);
  Triplet B(0, 2 * (jSize-2), 2);

  u()((A,B)) += w * z()((X,Y));
  u()((A+1,B)) += w * z()((X,Y));
  ...
}
```

(a)

```
#define Grid HTA
void setup(...) {
 allocate H, HTA level 0;
 for i = 1..num_levels {
  H.refine(level_i, refinement_factor)
 }
}

void interp(Grid u, level lev) {
  Weight w;
  Region r_coarse(X,Y), r_fine(A,B);
  <r_coarse, r_fine> = u.project(lev);
  //project level lev of u down; return mapping
  u.get_level(lev+1)(r_fine) += w *
              u.level(lev)(r_coarse);
  r_fine.X += 1;
  u.get_level(lev+1)(r_fine) += w *
              u.level(lev)(r_coarse);
  ...
}
```

(b)

Fig. 11: MG `setup()` and `interp()` functions adapted for 2D Grids: (a) without refinement support and (b) with refinement support

Figure 10(b), the user will have to allocate explicit space for all refined sections and to store, separated from the grid data structure, more complex information on how grids at different levels are correlated. Both aspects can be naturally expressed with our extensions to HTA, such that to simplify the programmer's effort in maintaining refinement information.

5.2 The Extensions for Multiple Levels of Data

Layered HTAs extend the interface of HTA with a set of primitives that facilitate computation requiring refinement as follows:

void refine(level lev, region, refinement_factor): at level `lev`, refine the specified `region` (see Section 2.1), using the specified `refinement_factor`. The refinement factor is a tuple specifying the refinement in each dimension. In Figure 10(a), e.g., level 1 is a refinement of the original grid, created through `refine(0, All, Tuple(2,2))`.

<region_coarse, region_fine> = project (level): computes how elements from two adjacent levels correspond to each other. `region_coarse` and `region_fine` are initialized by this method. Both regions will have the same size (iteration space) and they encapsulate the mapping information that is used to perform operations between elements on two adjacent levels. Figure 10(a), illustrates an HTA with two levels of refinement; the call `<region_coarse, region_fine> = HTA.project (0)`, initializes the two regions such that element (X,Y) from `region_coarse` will be associated with element (A,B) from `region_fine`, (X,Y+1) with (A,B+2), and similarly for all elements at the coarse level. The corresponding regions initialized with `project()` are used to perform assignments and different operations between elements of HTAs at adjacent levels. To exemplify the use of the `project()` method, we show in Figure 11(b),

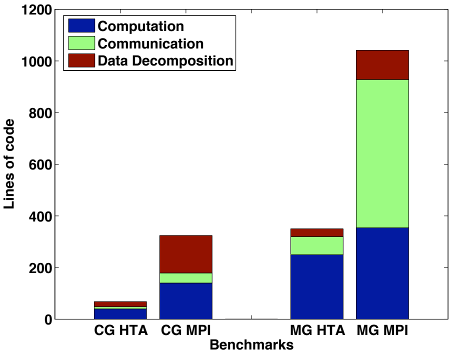


Fig. 12: Linecount of key sections of HTA and MPI programs.

Fig. 13: Execution times of the NAS HTA codes on BlueGene/L.

Number of	Benchmark			
Processors	CG A	CG B	MG A	MG B
1	68.63	2631.39	-	-
2	35.27	1335.76	-	-
4	17.94	671.96	-	-
8	12.49	451.84	9.57	38.03
16	6.47	227.78	5.06	20.37
32	6.94	186.08	2.84	11.7
64	4.19	93.02	1.91	8.27

function `interp()` from the NAS MG kernel. The function has the same functionality as the `interp()` method in Figure 11(a), the mapping corresponding to the refinement are however not computed explicitly, but provided by the layered HTA.

HTA get_level(level lev): To simplify the design, layered HTAs can't be used in expressions with regular HTAs. Layered HTAs provide `get_level()`, a cast type of method that returns an HTA representing the tiling and data at a specific level. This mechanism allows to apply regular HTA operations to a specific layer of a multi layered HTA.

The extensions described in this section, provide an interface that facilitates the development of applications requiring mesh refinement. The hierarchy of refinement levels existent in the application is naturally mapped onto the hierarchy of an HTA. We are currently exploring an adaptive mesh refinement application[17] to analyze the impact of the proposed HTA extensions on productivity and performance.

6 Experimental Results

HTA programs benefit from a high level notation, much more expressive than that of other approaches to implement parallel programs. Thus it is expected that they will boost programmer's productivity. Figure 12 measures this property by comparing the number of lines of code used by the htalib implementation of the CG and MG NAS benchmarks with that of their corresponding MPI counterparts. As we can see, the HTA codes have substantially fewer lines of code in each one the three categories in which we have classified them.

Figure 13 shows the execution times for CG and MG for classes A and B for different numbers of processors on the BlueGene/L. Some experiments could not be run due to the limited amount of main memory available in a single compute node. We observe good scalability and our current efforts are focused on identifying and optimizing the primitives that will further improve this to match the speedup numbers achieved by the FORTRAN + MPI versions. In terms of absolute performance, the FORTRAN + MPI versions of the benchmarks included in Figure 13 run on average 2.2 times faster than the htalib version.

7 Related Work

The programming for distributed memory systems has traditionally followed a SPMD, message-passing model. Programmers specify the path of execution for each processor, and messages are exchanged either using standard libraries like MPI [14] or higher level constructs provided by languages like Co-Array Fortran (CAF) [16] and UPC [8]. This paradigm produces very efficient programs at the cost of high development and maintenance costs, as the programmer has to distribute and communicate data, synchronize processes and choose execution paths for each processor manually.

There are three principal approaches to implement global view programming models for distributed memory systems: (i) extensions of existing languages with directives such as HPF [2], (ii) novel languages like ZPL [9], and (iii) libraries like htalib or STAPL [5]. We find the latter strategy most attractive as it facilitates the reuse of existing codes. Moreover, the library-based approach allows the gradual migration of sequential codes to a parallel form without relying on complex compiler technology that is not always effective in optimizing the overheads associated with the global view model.

What differentiates the HTA from other approaches is (a) its unique treatment of the tile as a first-class object that is explicitly addressed and manipulated and (b) its emphasis in the recursive subdivision of the tiles in order to adapt the data storage and computation structure of the codes to the underlying machine. This latter characteristic is shared with Sequoia [13], although its principal construct is procedural (the *task*), while our approach is data centric. Also, Sequoia tasks are not associated to particular processors; instead they can run in any node in which their working set fits. HTAs provide on the contrary a clear model of data and task placement and communications. Finally, Matrix+$^+$ [10] also allows the construction of hierarchical matrices and implements recursive operations on them, but it lacks the flexible notation of the HTAs to access and manipulate the tiles. Rather, it is focused on computations involving the whole resulting matrices.

8 Concluding Remarks

In this paper we describe three extensions to HTAs that facilitate their use in a wide variety of application contexts. Our extensions factor out functionality that is commonly encountered in array-based codes. Using the *operator framework*, programmers can specify powerful, block-recursive array operations in a sequential logic, while the skeleton of reduction and iteration is provided by htalib. *Overlapped tiling* relieves the programmer from explicitly allocating and maintaining ghost cells in stencil computations. Support for *multi-layering* helps users express refinement and computations that involve arrays at adjacent layers of the hierarchy. Our experience with a portable C++ library shows that the new HTA features permit to implement the NAS kernels in a structured and concise manner without compromising scalability or performance on a BlueGene/L system.

Acknowledgment We thank George Almási, Nancy Amato, and Calin Cascaval for their advice and comments on our work.

References

1. NAS Parallel Benchmarks. Website. http://www.nas.nasa.gov/Software/NPB/.
2. High Performance Fortran Forum. *HPF Specification Version 2.0*, January 1997.
3. J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 90 Handbook*. McGraw-Hill, 1992.
4. G. Almási, L. D. Rose, B. B. Fraguela, J. Moreira, and D. Padua. Programming for Locality and Parallelism with Hierarchically Tiled Arrays. In *Proc. of LCPC*, pages 162–176, Oct 2003.
5. P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An Adaptive, Generic Parallel Programming Library for C++. In *Proc. of LCPC*, pages 193–208, August 2001.
6. M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82(1):64–84, 1989.
7. G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, pages 48–57, 2006.
8. W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.
9. B. L. Chamberlain, S. Choi, E. Lewis, C. Lin, S. Snyder, and W. Weathersby. The Case for High Level Parallel Programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–86, July–September 1998.
10. T. Collins and J. C. Browne. Matrix+$^+$: An object-oriented environment for parallel high-performance matrix computations. In *Proc. of the 28th Annual Hawaii Intl. Conf. on System Sciences (HICSS)*, page 202, 1995.
11. D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split-c. In *Conference on Supercomputing (SC)*, pages 262–273, 1993.
12. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation (OSDI)*, 2004.
13. K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *To appear in Proc. of Supercomputing 2006*, Nov 2006.
14. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI (2nd ed.): Portable Parallel Programming with the Message-Passing Interface"*. MIT Press, 1999.
15. 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.
16. R. W. Numrich and J. Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.
17. T. Wen and P. Colella. Adaptive mesh refinement in Titanium. *IPDPS*, 2005.
18. M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI*, pages 30–44, 1991.