Task-parallel versus data-parallel library-based programming in multicore systems

Diego Andrade, Basilio B. Fraguela
University of A Coruña, Spain
{dcanosa,basilio}@udc.es

James Brodman and David Padua
University of Illinois at Urbana-Champaign, USA
{brodman2,padua}@uiuc.edu

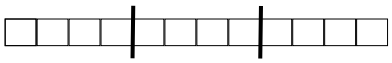
***Abstract*—Multicore machines are becoming common. There are many languages, language extensions and libraries devoted to improve the programmability and performance of these machines. In this paper we compare two libraries, that face the problem of programming multicores from two different perspectives, task parallelism and data parallelism. The Intel Threading Building Blocks (TBB) library separates logical task patterns, which are easy to understand, from physical threads, and delegates the scheduling of the tasks to the system. On the other hand, Hierarchically Tiled Arrays (HTAs) are data structures that facilitate locality and parallelism of array intensive computations with a block-recursive nature following a data-parallel paradigm. Our comparison considers both ease of programming and the performance obtained using both approaches. In our experience, HTA programs tend to be smaller or as long as TBB programs, while performance of both approaches is very similar.**

I. Introduction

Processor manufacturers are building systems with an increasing number of cores. These cores usually share the higher levels of the memory hierarchy. Many language extensions and libraries have been developed to ease the programming of this kind of systems. Some approach the problem from the point of view of task parallelism. The key notion is that the programmer has to divide the work into several tasks which are mapped automatically onto physical threads that are scheduled by the system. Task-parallelism can be implemented using libraries such as `POSIX` Threads [1] which provide minimal functionality and for this reason some consider this approach the assembly language of parallelism. Recently, the Intel Thread Building Blocks (TBB) library [2] has appeared, which allows expressing parallelism using a higher level of abstraction. A third task-parallel API is `OpenMP` [3] which is mainly based on simple compiler directives used to guide mostly the parallelization of regular loops although the recently proposed extension [4] with a task-enqueuing mechanism extends its scope of application. The task of writing parallel programs can also be faced from the point of view of data parallelism, where tasks perform the same operation on different pieces of data. Thus, data-based parallelism is a subset of control or task parallelism, as in task-based parallelism tasks can perform the same or different operations on the same or different pieces or data. Parallel array computation has been widely used since the appearance of the first array and vector computers [5] , playing a key role in the expression of parallelism in many languages such as High Performance Fortran [6] and its variants [7], ZPL [8], X10 [9], and others. The historical experience in the attempts to implant new languages with a focus on parallelism, coupled with the large base of existing legacy codes, has made many researchers think that macros, and more in general, libraries, are a better vehicle to bring parallelism to mainstream computing. This way, libraries such as POOMA [10] or POET [11] have explored this possibility. More recently, the Hierarchically Tiled Array (HTA) library [12], [13] facilitates the writing of data-parallel programs, putting a special emphasis on the concept of tiling [14] both to express locality and parallelism.

With the arrival of multicores to all computing systems from embedded ones to supercomputers, the relevance of parallel programming for these systems is enormous. As a result there are large efforts both from industry and academia to improve the productivity in the development of parallel codes as well as the resulting performance. In this context we have found of interest a comparison between task and data parallelism. Our aim is to explore in the scope of multicores the impact of following either approach on productivity and performance, and to which point the additional restrictions of data-parallelism may discourage its usage to parallelize certain problems. We have chosen as representatives of both approaches the Intel TBB library [2], which enables writing programs based on task parallelism, and the Hierarchically Tiled Array (HTA) library [12], [13], which facilitates the implementation of data parallel programs. As far as we know, the HTA library is the most up-to-date library specifically designed following a data-parallel paradigm with a good support for multicore systems. Among the large variety of approaches that allow task-parallel programming for these systems, the TBB library is the one that most closely resembles the HTA in terms of level of abstraction and interface provided to the programmer, thus enabling a fair comparison of both approaches. Another reason for choosing TBB is that, since the HTA library for shared memory is implemented on top of the TBB library, performance differences between both libraries can be more clearly attributed to the specific way in which they lead programmers to write their applications.

This paper is organized as follows. Sections II and III summarize the main features of the HTA and TBB libraries, respectively. In Section IV a high-level description of the algorithms used for the comparison is presented and the implementation both with TBB and HTAs is discussed briefly.



```
Tuple<2>::Seq tiling = (Tuple<1>(4),Tuple<1>(3)) ;
HTA<int,1> h = HTA<int,1>::alloc(1,tiling,ROW);
```

Fig. 1. Creation of a HTA

Section V describes the main differences between the TBB and HTAs libraries. We will illustrate how data and task parallelism face the same problems using different approaches. Section VI compares quantitatively the codes written using both libraries, and Section VII presents the conclusions.

II. The HTA library

The Hierarchically Tiled Array (HTA) [12], [13] is an array data type which can be partitioned into tiles. Each tile can be either a conventional array or a lower level HTA, thus HTA is a recursive data type. HTAs adopt tiling [14] as a first class construct for array-based computations. Tiles have been used to express data parallelism [8], [6], [15], [16] and to improve the locality of the accesses [17]. Thus HTAs empower programmers to control data distribution and the granularity of computation explicitly through the specification of tiling. This feature makes the library more versatile and provides maximal control to the programmer, allowing tuning the tiling to achieve the best performance, as we will see in Section VI. Of course, the tiling used for the HTAs can be actually obtained by an automatic method that tries to choose the best one; this is out of the scope of our work.

Figure 1 shows the operations needed to create an HTA with 3 tiles of 4 elements each. The variable `tiling`, defined in line 1, specifies the number of elements or tiles for each dimension and level of the HTA, from the bottom to the top of its hierarchy of tiles. The `alloc` operation in the second line creates the HTA. The number of levels of tiling is passed as the first parameter to `alloc`. The tiling structure is specified by the second parameter. The third parameter selects the data layout, which can be row mayor (`ROW`), column major (`COLUMN`) or `TILE`, in which the elements within a tile should be stored by rows and in consecutive memory locations. The layout across tiles is always row major. The data type and the number of dimensions are template parameters of the HTA class.

While the tiling structure of an HTA is specified at creation time, it can be modified dynamically by adding or removing *partition lines*, the abstract lines that separate the tiles in an HTA. This generates new tiles or merges existing ones, respectively, in a process known as *dynamic partitioning*.

HTA indices are zero-based. Tiles or scalars of HTAs can be selected using `n`−Tuples. In this explanation the `n`−Tuple notation is substituted by a list of integers $(x, y, \ldots)$. HTAs can also be indexed using `Ranges` of the form `low` : `step` : `high`. The list of integers and ranges can be enclosed by the $()$ operator, which selects tiles, or by the $[]$ operator, which selects scalar elements of the HTA. For example $h(1)[2]$ yields element $[2]$ within tile (1). If no tile is selected, the $[]$ operator is used to access the HTA as a conventional array.

HTAs facilitate parallel programming by providing numerous methods that operate in parallel across tiles. The main constructs are:

- `hmap`: It applies a function to each element of the HTA or corresponding elements of two or more conformable HTAs.
- `reduce`: It performs a reduction, that is, it applies operations on an HTA to produce an HTA of lesser rank.
- `scan`: It computes a prefix operation across all the elements of an array. Let `op` be an associative operation with left identity element id. The parallel prefix of `op` on a sequence $x_0, x_1, \ldots, x_{n-1}$ will be another sequence $y_0, y_1, \ldots, y_{n-1}$ where $y_0 = id$ `op` x_0 and $y_i = y_{i-1}$ `op` x_i.
- `mapReduce`: performs a reduction on the results of a given operation applied to each element of an statement. It is a mixture of an `hmap` and a `reduce` construct following the principles of the map-reduce [13]

The four constructs receive at least one argument, a function object (functor) whose `operator()` method encapsulates the operation to perform on a single tile of the object HTA. In the case of `hmap`, the function may accept additional HTAs as parameters that must be conformable, that is, have the same tiling structure as the HTA instance on which the `hmap` is invoked. The functor will handle in this case in each invocation a tile from each one of the HTAs involved. In any case, the functor handles a single tile per HTA in its `operator()`, so that the indexing of the elements inside the operation is relative to the first position of a tile. Also, the operation on each tile is performed in parallel, at least conceptually. Data-parallelism is enforced by the fact the functors can only write in the tiles received in each single invocation or the predefined outputs of the operation, such as in reductions. Thus, global data out of the specific tiles received in the invocation must be strictly read-only. Unfortunately, C++ does not provide mechanisms that allow the library to preclude programmers from writing to global data outside the tiles in the functors that are applied in parallel. If they do so, they are certainly not following the data-parallel structured approach promoted by HTAs.

An interesting feature of HTAs, called *overlapped tiling*, is the ability to create and manage automatically shadow or ghost regions around each tile that contain a copy of the elements of neighboring tiles. This is particularly useful for stencil codes, which compute new values based on their neighbors, as in the case of $\mathtt{a(i)} = \mathtt{a(i-1)} + \mathtt{a(i+1)}$.

Finally, it is interesting to notice that, in contrast to the Intel TBB library, which is restricted to shared memory, HTAs support shared, distributed and hybrid memory systems.

III. The Intel TBB library

The Intel Threading Building Blocks (TBB) [2] library was developed by Intel for the programming of multithreaded applications. In this case, it is not necessary to use a special type of data structure. However, the TBB library provides

of special types of containers which can be manipulated concurrently. But this is not the native method to specify parallelism, like in the HTA case. As mentioned above, the TBB library enables the implementation of multithreaded task-parallel programs. It does not base the specification of parallelism on data operations that are inherently parallel, which is the HTA approach. Rather, parallelism is achieved by defining tasks that can be performed concurrently. The task scheduler then maps tasks to available hardware threads. The task scheduler distributes the tasks between the available threads. When there are more threads available than tasks, it can split an existing task in several smaller tasks.

A. TBB operations

The element-by-element operation, reduction, and scan constructs are implemented in the TBB library using the `parallel_for`, `parallel_reduce` and `parallel_scan` *algorithm templates* respectively. The TBB library also includes the algorithm templates `parallel_do`, which supports unstructured workloads where the loop limits are not known at the beginning of the loop, and `pipeline`, which is used when there is a sequence of stages that can operate in parallel on a data stream.

The `parallel_for`, `parallel_reduce` and `parallel_scan` algorithm templates accept two basic parameters: a *range* defining loop limits, and a function object representing the body of the parallel loop. This object overloads the `operator()` method and defines the operation to be performed on the range assigned.

The range is split recursively into subranges by the task scheduler and mapped onto physical threads. The TBB library provides standard ranges, such as `blocked_range`, which expresses a linear range of values in terms of a lower bound, an upper bound, and optionally, a grain size. The grain size is a guide for the workload size per task. The value of granularity affects the performance and load balance of the parallel operation.

The TBB library allows creating ad-hoc ranges. That is, the user can define new range classes implementing specific policies to decide when and how to split, how to represent the range, etc. An example of usage of ad-hoc range will be shown in Section IV-C.

Some additional features present in the TBB library are a scalable and efficient memory allocator for multithreaded programs, mutual exclusion structures for explicit thread synchronization, support for atomic operations on primitive data types, and thread-aware timing utilities.

IV. Implementation of Some Algorithms

The codes used in this comparison were taken from the chapter 11 of [2], which contains examples of parallel implementations of algorithms using TBB[1]. We chose these codes because at the time the experiments were done there were very few codes available written using TBB, and they cover a diversity of parallel computation patterns. This section describes them and highlights the key differences between the TBB and HTA implementations using some snippets of code.

[1]These codes are in public domain and the can be downloaded from http://softwarecommunity.intel.com/articles/eng/1359.htm

```
1  class Average {
2  public:
3      float * input, output;
4      void operator ()( const blocked_range<int>& range ) const {
5          for( int i = range.begin(); i != range.end(); ++i )
6              output[i] = (input[i-1] + input[i] + input[i+1]) * (1/3.0f);
7      }
8      ...
9  };
10
11 const int N = 100000;
12 static int nThreads = 4;
13
14 int main( int argc, char* argv[] ) {
15     float raw_input[N+2], output[N];
16     raw_input[0] = 0;
17     raw_input[N+1] = 0;
18     float * padded_input = raw_input + 1;
19     ... /* Initialization not shown */
20     task_scheduler_init init (nThreads);
21
22     Average avg(padded_input, output);
23     parallel_for ( blocked_range<int>( 0, N, 1000 ), avg );
24
25     return 0;
26 }
```

Fig. 2. TBB implementation of the *Average* algorithm

A. Average

This algorithm calculates, for each element in a vector, the average of the previous element, the next element and itself and the result is the stored in an output vector. It can be parallelized by the TBB library using the `parallel_for` construct. The TBB code that implements this algorithm is shown in Figure 2. In this code, the first and the last elements of the array are special cases, since they don't have previous and next elements, respectively. This is solved by adding elements at the beginning and the end of the array which are filled with zeros as shown in lines 15-18 of the code. In line 20, the task scheduler object is created and initialized with 4 threads. The task scheduler is the engine in charge of mapping tasks to physical threads and of the thread scheduling. It must be initialized before executing any TBB parallel constructs.

The first argument of the `parallel_for` in line 23 is a range which includes the whole vector. A grain size of 1000 is advised in this case. The second argument is an object of the class `Average` which encapsulates the operation to be executed by the `parallel_for`. This class is defined in lines 1 thru 9. The `operator()` method in this class defines the operation that will be applied on each subrange. The low and high values of the indexes for each subrange are directly extracted from the range parameter using the `begin()` and `end()` methods (see line 5).

The HTA implementation of this algorithm is shown in Figure 3. The data structures are created in lines 17-20. The padding values are automatically generated and filled in HTA `input` thanks to overlapped tiling. Line 18 defines an object that describes the overlapping of tiles in `input`. The first two arguments of the constructor specify that shadow regions have size one in both the positive and negative direction. This constructor allows a third optional argument to specify whether the boundary region built around the array is filled with zeros, which is default behavior when nothing is specified, or it

```
1  typedef HTA<float,1> HTA_1;
2  #define T1(i) Tuple<1>(i);
3
4  struct Average {
5      void operator()(HTA_1 input_, HTA_1 output_) const {
6          for( int i = 0; i != input_.shape().size()[0]; ++i )
7              output_[i] = (input_[i-1] + input_[i] + input_[i+1]) * (1/3.0f);
8      }
9  };
10
11 const int N = 100000;
12 static int nTiles = 4;
13
14 int main( int argc, char* argv[] ) {
15     Traits::Default::init (argc,argv);
16
17     Seq< Tuple<1> > tiling( T1(N / nTiles), T1(nTiles) );
18     Overlap ol( T1(1), T1(1) );
19     HTA_1 input = HTA_1::alloc(1, tiling, ol, NULL, ROW);
20     HTA_1 output = HTA_1::alloc(1, tiling, NULL, ROW);
21     ... /* Initialization not shown */
22
23     input.hmap(Average(), output);
24
25     return 0;
26 }
```

Fig. 3. HTA implementation of the *Average* algorithm

```
1  for( int i=1; i<UH-1; ++i ) {
2    value t = (value)i / UH;
3    Material Type m = SANDSTONE;
4    M[i] = 1.0/8;
5    if( t < 0.3f ) {
6      m = WATER;
7      M[i] = 1.0/32;
8    } else if( (0.5 <= t) && (t <= 0.7) ) {
9      m = SHALE;
10     M[i] = 1.0/2;
11   }
12   Material[i] = m;
13 }
```

(a) TBB version

```
1  M[1:0.3*UH] = 1.0/32;
2  Material[1: 0.3*UH] = WATER;
3
4  M[0.3*UH+1: 0.5*UH] = 1.0/8;
5  Material[0.3*UH+1: 0.5*UH] = SANDSTONE;
6
7  M[0.5*UH+1: 0.7*UH] = 1.0/2;
8  Material[0.5*UH+1:0.7*UH] = SHALE;
9
10 M[0.7*UH+1:UH-1] = 1.0/8;
11 Material[0.7*UH+1:UH-1] = SANDSTONE;
```

(b) HTA version

Fig. 4. Terrain initialization in the Seismic program

is periodic, i.e., it replicates the values of the array on the opposite side. In line 19 this overlapping specification is used to create an HTA with `N` values distributed in `nTiles`. Line 20 allocates the HTA where the result will be stored, which has the same topology as the one used as input but with no overlapped regions.

The `hmap` method is invoked in line 23. Its first argument is the operation to perform on each tile of the HTAs. This operation, `Average`, is defined as a `struct` in lines 4-9. `Hmap` calls this operation for each tile of the HTA. The `for` loop of line 7 iterates on the indexes of the elements in each tile.

B. Seismic

This code performs a simple seismic wave simulation (wave propagation) which sets the impulse from the source of the disturbance, does the two most time consuming computations of update stress and velocity, and finally cleans up the edges of the simulation. This way, the algorithm has two main parts: the initialization and the main loop, which consists of four steps: set impulse, update stress, update velocity and clean the edges.

The initialization of the data structures involved in the code is sequential both in the TBB and the HTA versions, but in the HTA version it has been rewritten using array notation, which allows to remove some loops and conditional statements. Figure 4(a) shows this initialization in the TBB version. Arrays `Material` and `M` contain the characteristics and composition of each band of the terrain. This code fills one band of the terrain with WATER, two with SANDSTONE and another one with SHALE. The HTA implementation is shown in Figure 4(b).

The updates of the stress and the velocity are stencil computations, implemented using a `parallel_for` for TBB and a `hmap` for HTA. The implementation of both is very similar to the implementation of the Average code in Section IV-A. The only difference is that in these computations we have to exclude the first position and the last position of the array. This is achieved in the HTA version by applying the `hmap` operation on a new HTA which does not include those elements. The creation and processing of that HTA is very costly in the current implementation of the library and it will have a negative impact on the performance, as we will see in the Evaluation, Section VI. The same problem is solved more efficiently in the TBB version by applying the operator using a range of the indexes to be used, which excludes the first and the last point of the dimension. The remaining parts of the code are sequential in both versions.

C. Parallel Merge

This code merges two sorted sequences into an output sorted sequence. The algorithm operates recursively as follows:

1) If the sequences are shorter than a given threshold, they are merged sequentially. Otherwise, Steps 2-5 are performed.
2) The sequences are swapped if necessary so that the first sequence, $[begin1, end1)$ (the notation $[$ indicates that the first value of the interval is included but not the last one), must be at least as long as the second sequence $[begin2, end2)$.
3) m1 is set to the middle point in the first sequence. The item at that location is called *key*.
4) m2 is set to the point where *key* would fall in the second sequence.
5) Subsequences $[begin1, m1)$ and $[begin2, m2)$ are merged to create the first part of the merged sequence and subsequences $[m1, end1)$ and $[m2, end2)$ are merged to create the second part. Both operations take place in parallel with each other.

The TBB implementation of this algorithm is based on a `parallel_for`. The subdivision of the sequences is implemented using an object of the ad-hoc range class `ParallelMergeRange` whose definition is shown in Figure 5(a). The predicate `is_divisible` performs the test in step 1. The `ParallelMergeRange` class has two constructors. The first one, shown in lines 7-18, contains the dummy

```
1  template<typename Iterator> struct ParallelMergeRange {
2      ...
3      bool empty()    const {return (end1 - begin1) + (end2 - begin2) == 0;}
4      bool  is_divisible () const  {
5           return  std::min( end1 - begin1, end2 - begin2 ) > grainsize;
6      }
7      ParallelMergeRange( ParallelMergeRange& r,  split  ) {
8           if( (r.end1 - r.begin1) < (r.end2 - r.begin2) ) {
9                std::swap(r.begin1, r.begin2);
10               std::swap(r.end1, r.end2);
11          }
12           Iterator  m1 = r.begin1 + (r.end1 - r.begin1) / 2;
13           Iterator  m2 = std::lower_bound( r.begin2, r.end2, *m1 );
14          begin1 = m1; begin2 = m2;
15          end1 = r.end1; end2 = r.end2;
16          out = r.out + (m1 - r.begin1) + (m2 - r.begin2);
17          r.end1 = m1; r.end2 = m2;
18     }
19     ...
20 };
21 ...
```

(a) TBB version

```
1  ...
2  if ( input1_size > GRAINSIZE) {
3   size1 = input1_.shape().size() [0];
4   size2 = input2_.shape().size() [0];
5
6    if( size1 < size2 ) {
7      h2=input1_; h1=input2_;
8      std::swap( size1 , size2 );
9    } else {
10     h1=input1_; h2=input2_;
11   }
12
13   int pos = h2.lower_bound(h1[(size1 - 1) / 2]);
14
15   h1.part( (size1 - 1) / 2 );
16   h2.part( pos );
17   output_.part( pos + (size1 - 1) / 2 );
18
19   output_.hmap(Merging(), h1, h2, 0);
20   ...
21 } else {
22   ...
```

(b) HTA version

Fig. 5. Parallel Merge

```
1  class SubStringFinder {
2   ...
3   void operator() ( const blocked_range<size_t>& r ) const {
4    for ( size_t i = r.begin(); i != r.end(); ++i ) {
5     size_t max_size = 0, max_pos = 0;
6     for (size_t j = 0; j < str.size(); ++j)
7      if (j != i) {
8       size_t limit = str.size()-( i > j ? i : j );
9       for (size_t k = 0; k < limit; ++k) {
10       if (str[i + k] != str[j + k]) break;
11       if (k > max_size) {
12        max_size = k; max_pos = j;
13      } } }
14
15    max_array[i] = max_size;
16    pos_array[i] = max_pos;
17   } }
18   ...
19 };
20 ...
21 parallel_for (blocked_range<size_t>(0, to_scan.size(), 100),
22                SubStringFinder( to_scan, max, pos ) );
23 ...
```

(a) TBB version

```
1  struct SubStringFinderOp {
2  void operator() (HTA<int,1> max_, HTA<int,1> pos_) {
3      ...
4      init_j =lower_bound_0;
5     end_i= init_i +max_.shape().size() [0];
6
7     int pos=0;
8     for (size_t i = init_i ; i != end_i; ++i) {
9      int max_size = 0, max_pos = 0;
10     for (size_t j = 0; j < str.size(); j++) {
11      if (j != i) {
12       int limit = str.size()-( i > j ? i : j );
13       for (int k = 0; k < limit; ++k) {
14        if (str[i + k] != str[j + k]) break;
15        if (k > max_size) {
16         max_size = k; max_pos = j;
17        } } } }
18     max_[pos] = max_size;
19     pos_[pos] = max_pos;
20     pos++;
21    } } };
22  ...
23 max.hmap(SubStringFinderOp(),pos);
24  ...
```

(b) HTA version

Fig. 6. Substring Finder

variable `split`. This argument is used by the TBB library to flag a `Range` constructor that is used to split an input `Range` in two. The constructor builds a new range that stores one of the halves of the original `Range` and modifies the original `Range`, received as first parameter, to hold the other half. This constructor performs the steps described in steps 2-5 of the algorithm. The other constructor is a conventional constructor. The basic operation simply performs the merge sequentially by means of a `std :: merge`.

The HTA version is based on `hmap`. In the function applied by `hmap`, if the sequences are bigger than a given threshold, steps 2-5 are implemented. This part of the algorithm, shown in Figure 5(b), is implemented using the dynamic partitioning feature. Lines 15-17 add new partitions to the two input HTAs and, the output HTA in the points selected as described in step 3 of the algorithm. This is performed using the `part` method, which accepts the position in which a new partition line is to be added, giving place to two separate tiles. The position where key would fall in the second sequence, mentioned in step 4 of the algorithm, is calculated in line 13 using the HTA function $lower\_bound$, which returns the index of the first element of the HTA that is equal or larger than its argument.

Line 19 calls `hmap` recursively with the repartitioned structures. In this call, `hmap` applies its functor argument on each chunk in parallel. After this, these partitions are removed using a method called `rmPart`. The recursion finishes when the sequences to merge are smaller than a given threshold, then step 1 is performed.

D. Substring Finder

In this code, given a string, for each position in the string, the program finds the length and location of the largest matching substring elsewhere in the string. For instance, take the string `flowersflows`. Starting the scan at the first character at position 0, the largest match is `flow` at position 7 with a length of 4 characters. The position and length of those matches are stored for each position of the string.

The parallelization strategy consists in searching the largest matching string for each position of the scanned string in parallel. The TBB version uses a `parallel_for`, while the HTA version uses a `hmap`.

The codes, shown in Figures 6(a) and 6(b) are very similar. The operation performed in parallel is the same in both cases, the only difference is the indexing of the data structures, as it happened in previous codes. In the HTA version, the `max` and `pos` arrays, where the result will be stored, are divided in tiles, and the `hmap` operation is applied separately on each tile, so the indexing will be relative to the start position of the current tile.

```
1  ...
2  class  tbb_parallel_task
3  {
4   ...
5   static  void  set_values (Matrix* source,  char* dest)
6   {
7    ...
8    m_source = source;  m_dest = dest;
9    ...
10  }
11
12  ...
13  void  operator ()(  const  blocked_range<size_t>& r ) const
14  {
15   ....
16   begin=(int)r.begin();
17   end=(int)r.end();
18   Cell  cell;
19
20   for  ( int  i=begin;  i<=end; i++)
21   {
22   *(m_dest+i) =  cell . CalculateState (
23                        m_source->data, m_source->width,
24                            m_source->height,i);
25    }}
26  ...
27 };
28  ...
29 for( int  counter=1;  counter<NSTAGES; counter++)
30    parallel_for  (blocked_range<size_t> (begin, end, grainSize),  tbb_parallel_task ());
31  ...
```

(a) TBB version

```
1  struct  EvolutionOp {
2 void  operator ()  (HTA<int,2> data_source, HTA<int,2> data_dest) {
3    ...
4   CellHTA cell;
5   size=data_dest.shape().size();
6
7   for( int  i=0;  i<size[0];  i++) {
8     for( int  j=0;  j<size[1];  j++) {
9          data_dest[i][j]= cell . CalculateState (data_source,  (i,  j));
10   }}}};
11  ...
12 Overlap<2>  * ol= new Overlap<2>(Tuple<2>(1,1), Tuple<2>(1,1), PERIODIC);
13 data= HTA<int,2>::alloc(1, ((SIZEX/NTILESX, SIZEY/NTILESY), (NTILESX,NTILESY)), ol, NULL, —
   ROW);
14  ...
15 for( int  counter=1;  counter<NSTAGES; counter++)
16   data.hmap(EvolutionOp(), data,  0);
17  ...
```

(b) HTA version

Fig. 7. Game of Life

E. Game of Life

The ”Game of Life” is a problem which opened the mathematical research field of *cellular automata*. The game is played in a two-dimensional orthogonal grid of square cells, each of which is in one of two possible states: *live* or *dead*. Every cell interacts with its eight *neighbors*, which are the cells that touch the cell horizontally, vertically or diagonally. In every step of this evolution, each cell lives, dies, stays empty or is born based on a simple decision depending on the surrounding population (number of neighbors). The rules that determine the evolution of life are:

1) Life persists in any cell where it is also present in two or three of their eight neighboring cells and otherwise disappears (from loneliness or overcrowding).
2) Life is born in any empty cell for which there is life in exactly three of the eight neighboring cells.

The decisions about each generation are taken based on the state of the cells in the previous generation, so the problem is fully parallel.

The parallel version decomposes the two-dimnesional space of cells in a number of regions, and the decisions for the next generation are taken in parallel in the different regions. This is implemented in the TBB and HTA versions using a `parallel_for` and a `hmap` respectively. Both implementations can be seen in Figures 7(a) and 7(b). Besides the differences in the implementation between `parallel_for` and `hmap` that we have seen in previous examples, in this code, as the decisions for each cell depend on the state of its eight neighbors, when we are computing the state of a cell in an edge of a tile we will need to have a shadow region of size 1, in order to access the state of the neighbors that belong to another tile. This way in lines 12 and 13 of Figure 7(b), the HTA which represents the board of cells is created with a shadow region of size one in both positive and negative direction of each dimension of the board. The last argument of the constructor of the overlap region in line 13, PERIODIC, determines which values will contain the shadow cells in the edge regions of the board. PERIODIC means that they contain the value located in the other side of the matrix. For example, the upper cell of position $(0,0)$ would be $(N-1,0)$ where $N-1$ is the size of the first dimension.

The need of an overlapped region in the HTA implementation can be seen as an overhead but it greatly eases the implementation of another part of the code with respect to the TBB version. The class `Cell` is used to model the behavior of an isolated cell of the board. Method `calculateState` of this class has to compute the new state for each cell. In the TBB version, most of the time, the state of cell (i,j) depends on the state of its neighbors located in positions: $(i-1,j-1)$,$(i-1,j)$,$(i-1,j+1)$,$(i,j-1)$,$(i,j+1)$,$(i+1,j-1)$,$(i+1,j)$ and $(i+1,j+1)$. But in the edge region, the neighbor values must be searched in the other side of the matrix. This complicates the implementation of `calculateState`. In the case of the HTA version, as we have shadow regions around each tile, and around the whole matrix filled using the PERIODIC criteria, all the indexing of the neighbors can be always be performed using standard HTA indexing. This will lead to a much better overall performance of the HTA version as we will see in Section VI.

Code	Arithmetic intensity (Flops)	Problem size (kB)	Parallelism pattern
Average	1.5	23906	regular
Seismic	330	24584	regular
Parallel merge	1	5859	irregular
Game of life	210	19531	regular
Substring finder	1397	88	regular

TABLE I
CHARACTERISTIC OF THE CODES PARALLELIZED

F. Characteristics of the code

Table I summarizes the main characteristics of the codes used in this comparison. The arithmetic intensity column contains the ratio of floating point operations (Flops) per word (32 bytes) of data processed. The problem size column contains the size in kilobytes of the data structures involved in each code. The seismic and game of life codes have very high arithmetic intensity because they perform 100 stages of

the simulation and evolution respectively. The substring finder has the highest arithmetic intensity as the inner loop of this algorithm compares several times the same positions of its array of strings. The parallelism pattern column is devoted to the regularity of the distribution of the parallel work. It is regular when the parallel work is divided into chunks of the same size and it is irregular if they use chunks of different sizes defined by the algorithm applied. Only the parallel merge code uses an irregular distribution of the work.

Code	Lines (HTA)	Lines (TBB)	HTA reduction
Average	28	39	+28.0%
Seismic	304	295	-3.0%
Parallel merge	70	74	+5.4%
Game of life	97	309	+69.0%
Substring finder	49	49	0.0%

TABLE II
NUMBER OF LINES FOR THE FIVE CODES PARALLELIZED IN THE HTA AND TBB VERSION

V. QUALITATIVE COMPARISON

Both Hierarchically Tiled Arrays (HTAs) and Threading Building Blocks (TBB) are libraries devoted to facilitating the expression of parallelism. HTAs are arrays which may be organized into one or more levels of tiles. When an operation is applied to an HTA its tiles can be processed concurrently. An interesting characteristic of the HTA library is that its programming model is useful both in serial or parallel scenarios. In the serial case, the array notation usually improves readability and the tiling structure can be used for locality enhancement. More importantly, HTAs can be used in both shared and distributed memory environments, although some operations such as dynamic partitioning are more costly in the distributed memory environment.

The approach of TBB, which is restricted to shared memory environments, is to parallelize loops by specifying tasks using ranges which will be recursively subdivided. Since TBB does not have the notion of tiling like HTA, it must rely on loop structure to improve locality, although a partitioner that tries to promote locality has been added to the newest version of the library. The distribution of the work is performed automatically by the task scheduler.

Much parallelism found in programs is data parallel and can be expressed as an element-by-element operation, a reduction, or a scan, as described in Section II. The TBB library implements these operations using a `parallel_for`, a `parallel_reduce`, and a `parallel_scan` operation respectively. The HTA library uses alternatively an `hmap`, a `reduce`, and a `scan` operation.

The manipulation of HTAs benefits from array-oriented notation, which allows expressing some computations in a more readable form than using nested loops (see Figure 4). This tends to reduce the number of lines of code as discussed in the next section. However, the advantage of array notation goes beyond the lines of code. Array notation is intrinsically deterministic and should for all practical purposes completely avoid the possibility of race conditions.

One important feature of the TBB library is the ability to create ad-hoc ranges which divide the iteration space using special rules. This feature is supported in the HTA library by means of dynamic partitioning as exemplified in IV-C.

The HTA library can define overlapped regions during the creation of an HTA. However, programs based on the TBB library have to resort to the use of padding regions managed by the programmer, or to implement special treatment for the edge regions of the array, which complicates the programming. An example of this can be seen in Sections IV-A and IV-E.

Some TBB library primitives are not implemented by any HTA construct. Examples of such primitives include *software pipeline*, some STL-like concurrent containers, mutual exclusion structures for explicit thread synchronization, support for atomic operations on primitive data types, and thread-aware timing utilities. When needed, TBB can provide those primitives to programs based on HTAs, since both libraries can be used in the same program. This is trivially proved by the fact that the HTA library implementation for shared memory is built on top of the TBB library. Nothing special is needed to make any of them aware of the usage of the other one. The programmer must only take into account that since the HTA library initializes the TBB scheduler because it uses it to schedule its tasks internally, s/he should not initialize it again.

One interesting property of TBB which is not available in the current implementation of the HTA library is the ability to subdivide the range to process depending on the number of cores available. If one of the cores finishes very soon, the amount of remaining work in another core can be recursively divided to generate a new subrange assigned to the idle core. This feature can be implemented in the HTA library. In the meantime, HTAs can compensate not having this feature by allowing the programmer to subdivide the work in much more tasks than threads so that they can used for load balancing. We evaluate this possibility in the next section, finding that it can indeed improve performance up to 25%.

VI. EVALUATION

The measurement of the impact of a library on the ease of programming is quite subjective. There is no formula to calculate exactly the readability of a program although experienced programmers can usually easily determine which implementation and notation is easier for development and maintenance. We have chosen the source lines of code as an objective method to compare the implementation of the algorithms using the TBB and HTA libraries. This metric counts all the source lines in the code ignoring the comments and empty lines. This metric has been measured in Table II for both the TBB and HTA version of the codes introduced in Section IV. The fourth column stands for the reduction of the source number of lines of code obtained in the HTA version with respect to the TBB one expressed as a percentage of the source number of lines of code of the TBB version. As can

be seen in the table, the HTA codes are either virtually on par or shorter than their TBB equivalents. The Game of Life sees significant improvements that can be attributed to overlapped tiling.

Table III shows the times in milliseconds for the execution of the HTA and the TBB versions of the codes in a system with two Quad core 2.66 Ghz Xeon processors using gcc 4.2.1 with optimization level three. Several measurements were taken using 1, 2, 3, 4 and 8 of the cores available in this machine. The execution times reflected in this table are the minimum of 10 runs. Table IV shows the results of the same experiments in an HP Integrity rx7640 server with eight dual-core 1.6 GHz Itanium Montvale processors with the same compiler and optimization flags. Several measurements were taken using 1, 2, 3, 4, 8, 12 and 16 cores. Again, each execution time is the minimum of 10 runs.

Table III shows that the HTA versions of each code achieve a performance similar to that of the TBB versions when the maximum number of cores (8) are used in the Quad core system. However, in Table IV the HTA versions of Average and Game of Life outperform by a large margin the TBB versions when the maximum number of cores are used in the Itanium-based system (16). In fact the HTA versions of these two programs scale much better than the TBB ones, as in both computers the TBB version of Average and Game of Life is the fastest for one core, while for the maximum number of cores the HTA is quite better. It is interesting to notice that these ones are also the algorithms in which the HTA code had the biggest reduction in number of lines of code with respect to the TBB version. Both the programmability and the performance improvement are mainly due to the overlapped tiling feature of the HTA, thus arising as a very useful property of this library. With respect to the other three programs, Substring finder scales well for both libraries, tends to be always faster in the HTA implementation. Parallel Merge, on the contray, works reasonably for few cores, but does not scale to a large number of them. Also, we notice that the dynamic partitioning feature of the HTA is somewhat slower than the ad-hoc ranges of TBB. This is not surprising given the need to modify the internal structure of an HTA each time it is repartitioned, and again when the newly introduced new partition lines are removed once the work is completed. Finally, Seismic scales similarly for both libraries, the TBB version being systematically the fastest. That is because the HTA version has to perform a costly operation to extract the first and the last element of each HTA, as we said in Section IV-B. This problem is solved more efficiently in the TBB version. The differences in the performance obtained in both architectures are attributable to the peculiarities of the Xeon and Itanium 2 architectures, since they differ largely in clock rate, method to exploit ILP, and memory hierarchy structure and size.

In these experiments, one tile per used core was created in each tiled HTA, except in the cases of Parallel Merge, where the HTA was tiled recursively using dynamic partitioning until the threshold tile size was reached, and the Game of Life, where one tile per core per dimension was used. This does not imply a dependence on the number of cores as HTAs are objects created at runtime whose tiling structure is computed dynamically. Thus the number of cores can be obtained dynamically and used in a general computation of the desired tiling structure. Parallel computations inside the HTA library are implemented using TBB parallel constructs and consequently make use of TBB's scheduler. The HTA library allows these algorithms to be expressed differently and often more clearly as well as respecting the minimum task granularity specified by the programmer when s/he chose the tile sizes; as opposite to the TBB which can choose the granularity with little or no control of the user. As a result, this possibly changes the number and order of operations. If an HTA and a TBB program performed the same operations in the same order, one would expect no difference in performance as the programs would essentially be syntactically as well as semantically identical. This is evidenced by the Substring Finder example.

Figures 8 and 9 show the speedup obtained when significantly more tiles than threads are used for the HTA version of each code in both the two quad-core Xeon server and the eight dual-core Itanium 2 server, respectively. The base of each bar (in black) represents the speedup obtained when just one tile per core is used, for a varying number of cores, which corresponds to the times in the tables II and III. We also run these codes creating up to 10 tiles per thread. The white bar on top of the black one marks the highest speedup achieved. The number of tiles per thread used to get this best performance is above the bar. We observed that the performance of the HTA version can be improved by a modest increase of the number of tiles. The effectivity of the overpartitioning depends on the code: some will not improve, while others get great benefits. The codes that benefit the most from overdecomposition are Game of Life, Parallel Merge, to a lesser extent, Average, and sporadically, Seismic. Substring finder seems not to get any benefit from overdecomposition as the performance with only one tile is optimal. Additional benefits come from the dynamic distribution of work on the available threads as the parallel computations in the implementation of the HTA library inherit from TBB. This is possible due to the overdecomposition of the problem, which can compensate for the inexistence in HTAs of an automatic dynamic partitioning feature of the work as the one available in TBB.

VII. Conclusions

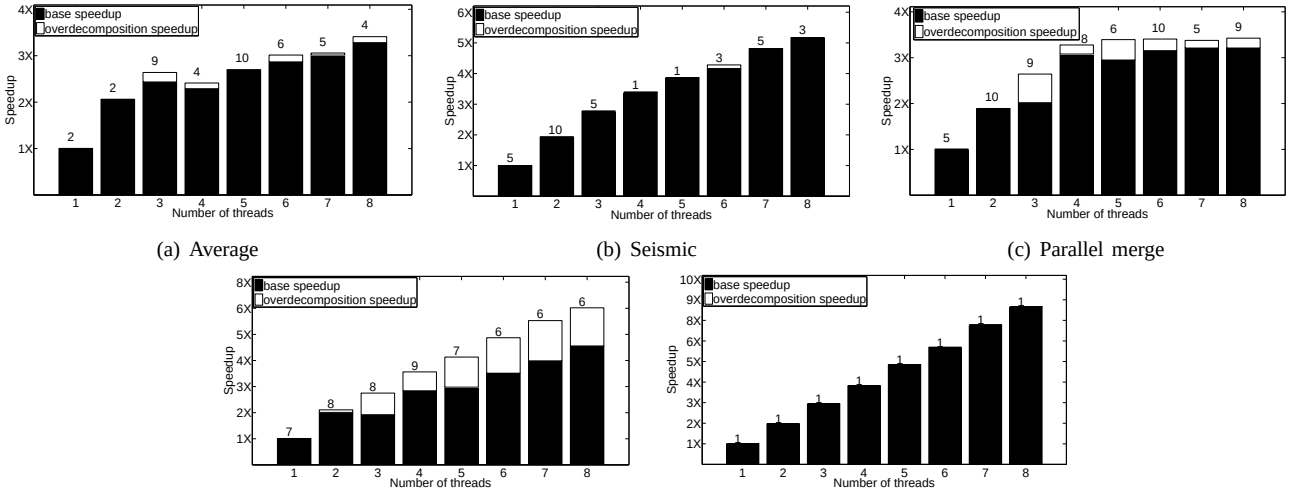
We have compared Intel TBB and HTAs, two libraries devoted to facilitating the programming of parallel machines following two very different approaches, since their parallelism is task and data based, respectively. For this purpose several algorithms were implemented using both libraries. The evaluation shows that the HTA codes are shorter or on par with the length of the TBB ones. This is because, array notation of some computations simplifies the HTA implementation of the TBB codes with loops and conditional statements, dynamic partitioning is easier to use than ad-hoc TBB `Ranges`, and overlapped regions hide the details of management of shadow

Code	HTA					TBB				
	1	2	3	4	8	1	2	3	4	8
Average	5.2	2.5	2.1	2.2	1.5	3.1	2.5	2.2	2.4	2.5
Seismic	8133.8	4234.1	2937.6	2399.8	1577.9	5975.2	3117.8	2328.4	1853.2	1458.7
Parallel merge	68.6	36.4	34.0	22.2	21.3	73.0	36.1	26.0	20.7	19.5
Game of life	4957.0	2465.0	2577.4	1745.7	1088.1	4473.9	2745.5	2130.2	1813.3	1381.3
Substring finder	5885.9	2992.0	2003.7	1541.6	768.9	6380.2	3203.8	2132.1	1610	820.3

TABLE III
TIMES IN MILLISECONDS, FOR THE TBB AND HTA VERSIONS IN THE TWO QUAD-CORE XEON SERVER USING 1,2,3,4 AND 8 CORES RESPECTIVELY

Code	HTA						TBB					
	1	2	4	8	12	16	1	2	4	8	12	16
Average	25.1	11.2	7.2	4.5	3.8	3.5	23.8	13.1	11.4	11.2	11.7	11.3
Seismic	19359.5	11201.7	7503.8	4916.8	3712.2	4129.8	15552.0	8824.3	6124.7	4000.6	3748.6	3215.6
Parallel merge	199.2	128.3	79.6	52.1	44.8	44.5	202.4	116.7	66.9	44.3	38.1	35.0
Game of life	19396.7	9486.7	6953.0	3478.9	2109.4	1690.8	16483.5	9623.1	6147.6	4386	3654.7	3409.9
Substring finder	9510.4	4895.6	2455.3	1256.8	791.5	689.9	10689.4	5361.9	2692.9	1366.2	924.4	717.0

TABLE IV
TIMES IN MILLISECONDS, FOR THE TBB AND HTA VERSIONS IN THE EIGHT DUAL-CORE ITANIUM 2 SERVER USING 1,2,4,8,12 AND 16 CORES RESPECTIVELY



(a) Average (b) Seismic (c) Parallel merge

(d) Game of life (e) Substring finder

Fig. 8. Speedup obtained considering different number of threads and maximum speedup obtained due to decomposition when creating up to ten tiles per thread in the two quad-core Xeon server

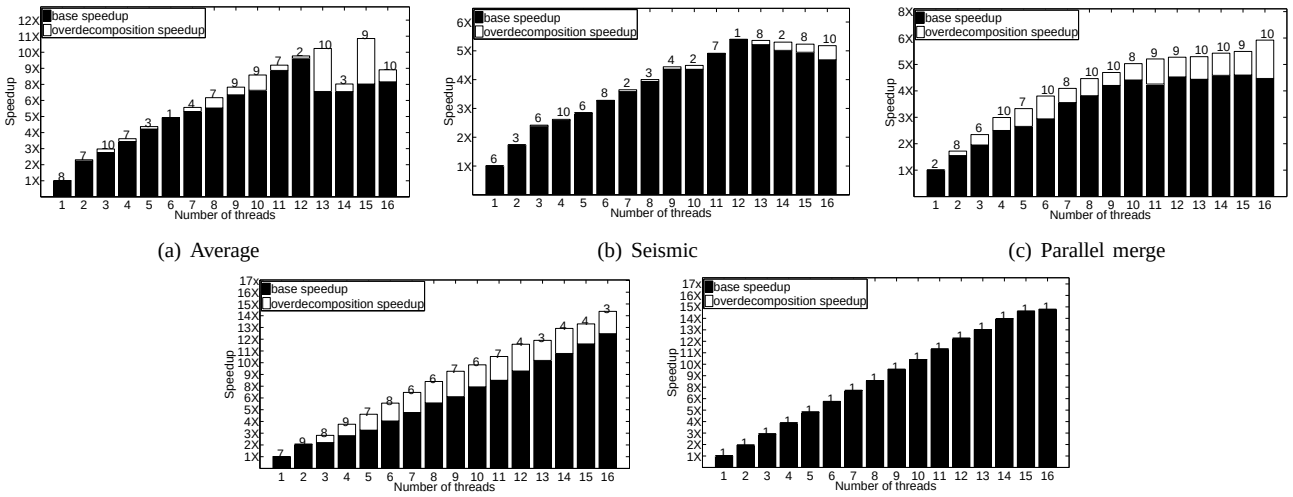
and padding regions from the programmer. The performance results show that the times obtained for the HTA versions are comparable to those obtained with the TBB ones. Sometimes both coding and performance improvements can be observed due to features like overlapped tiling, as in the case of the Game of Life and Average programs.

The HTA library seems a more natural way to express data-parallelism, which arises frequently in real programs, while TBB offers more flexibility and can be used to solve other situations for which HTAs may not be suitable. However, while HTA codes can be run in hybrid and distributed memory systems, TBB codes can only be run in shared memory environments.

An interesting property of TBB not yet implemented in the HTA library is the ability to repartition the work in an automatic way according to the number of idle cores. Thus, enabling the automatic repartitioning of HTAs dynamically according to the number of idle cores in a similar way to the behavior of ranges in the TBB library is an interesting extension for this library. Still, we have tested the possibility that HTAs provide of manually overdecomposing the work domain to facilitate balancing, achieving noticeable performance improvements in several codes.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Awards CCF 0702260 and CNS 0509432. Diego Andrade and Basilio B. Fraguela were partially supported by the Xunta de Galicia under project INCITE08PXIB105161PR and the Ministry of Education and



(a) Average (b) Seismic (c) Parallel merge

(d) Game of life (e) Substring finder

Fig. 9. Speedup obtained considering different number of threads and maximum speedup obtained due to decomposition when creating up to ten tiles per thread in the eight dual-core Itanium 2 server

Science of Spain, FEDER funds of the European Union (Project TIN2007-67537-C03-02). We also want to acknowledge the Centro de Supercomputación de Galicia (CESGA) for the usage of its supercomputers for this paper.

REFERENCES

[1] D. R. Butenhof, *Programming with POSIX Threads*. Addison Wesley, 1997.

[2] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*, 1st ed. O'Reilly, July 2007.

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

[4] OpenMP Architecture Review Board, "OpenMP Program Interface Version 3.0," May 2008.

[5] G. Barnes, R. Brown, M. Kato, D. Kuck, D. Slotnick, and R. Stokes, "The ILLIAC IV Computer," *IEEE Trans. on Comp.*, vol. C-17, no. 8, pp. 746–757, Aug. 1968.

[6] High Performance Fortran Forum, "High Performance Fortran language specification, version 2.0," Tech. Rep., January 1997.

[7] B. M. Chapman, P. Mehrotra, and H. P. Zima, "Vienna fortran—a fortran language extension for distributed memory multiprocessors," pp. 39–62, 1992.

[8] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, L. Snyder, W. D. Weathersby, and C. Lin, "The case for high-level parallel programming in zpl," *IEEE Computational Science and Engineering*, vol. 05, no. 3, pp. 76–86, 1998.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an Object-oriented Approach to Non-uniform Cluster Computing," in *ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*. New York, USA: ACM, 2005, pp. 519–538.

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

[11] R. Armstrong, "Poet (parallel object-oriented environment and toolkit) and frameworks for scientific distributed computing," in *30th Hawaii Int. Conf. on System Sciences*. Washington, DC, USA: IEEE Computer Society, 1997, p. 54.

[12] 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 Symp. on Principles and Practice of Parallel Programming (PPoPP'06)*, 2006, pp. 48–57.

[13] G. Bikshandi, J. Guo, C. von Praun, G. Tanase, B. B. Fraguela, M. J. Garzarán, D. Padua, and L. Rauchwerger, "Design and Use of htalib - a Library for Hierarchically Tiled Arrays," in *Proc. of LCPC 2006*, ser. LCNS, vol. 4382. Springer-Verlag, Nov 2006.

[14] A. C. McKellar and J. E. G. Coffman, "Organizing matrices and matrix operations for paged memory systems," *Commun. ACM*, vol. 12, no. 3, pp. 153–165, 1969.

[15] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[16] 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, Tech. Rep., January 1999.

[17] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *PLDI '91: Proc. of the ACM SIGPLAN 1991 conf. on Programming language design and implementation*. New York, NY, USA: ACM, 1991, pp. 30–44.