Hierarchically Tiled Arrays Vs. Intel Threading Building Blocks for Programming Multicore Systems *

Diego Andrade[1], James Brodman[2], Basilio B. Fraguela[1], and David Padua[2]

[1] Universidade da Coruña, Spain
{dcanosa,basilio}@udc.es
[2] University of Illinois at Urbana-Champaign
{brodman2,padua}@uiuc.edu

Abstract. Multicore systems are now the norm. Programmers can no longer rely on faster clock rates to speed up their applications. Thus, software developers are increasingly forced to face the complexities of parallel programming. The Intel Threading Building Blocks (TBBs) library was designed to facilitate parallel programming. The key notion is to separate logical task patterns, which are easy to understand, from physical threads, and delegate 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. The model underlying HTAs provides programmers with a data parallel, single-threaded view of the execution. The HTA implementation in C++ has been recently extended to support multicore machines. In this work we implement several algorithms using both libraries in order to compare ease of programming and performance.

1 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 system. 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. The Intel Thread Building Blocks (TBBs) library [1] enables the writing of programs that make use of this form of parallelism.

Task-parallelism can be implemented alternatively using libraries such as `POSIX` Threads [2] which provide minimal functionality and for this reason some consider this approach the assembly language of parallelism. A third task-parallel API is `OpenMP` [3] which, however, is not as powerful as TBB.

* 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 Ministry of Education and Science of Spain, FEDER funds of the European Union (Projects TIN2004-07797-C02-02 and TIN2007-67537-C03-02).

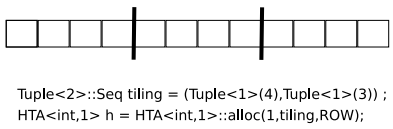

Fig. 1. Creation of a HTA example

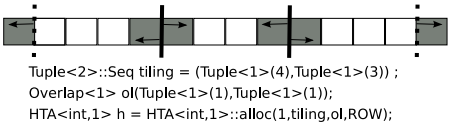

Fig. 2. Overlapped tiling example

On the other hand, the Hierarchically Tiled Array (HTA) library [4, 5] facilitates the implementation of data parallel programs. An HTA is a recursive array data type where elements are either HTAs or standard arrays. 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. In contrast to the approaches mentioned above, HTAs are good for shared, distributed and hybrid memory systems. The HTA library implementation for shared memory is implemented on top of the TBB library.

In this work, we compare the implementation of some algorithms using both the TBB and HTA libraries. Sections 2 and 3 summarize the main features of the HTA and TBB libraries, respectively. In Section 4 a high-level description of some implemented algorithms is presented and its implementation in both TBBs and HTAs is discussed briefly. Section 5 describes the main differences between the TBBs and HTAs libraries. We will illustrate how data and task parallelism face the same problems using different approaches. Section 6 discusses some validations results, and Section 7 presents the conclusions.

2 The HTA library

The Hierarchically Tiled Array (HTA) is an array data type which can be partitioned into tiles. Each tile can be either a conventional array or a lower level HTA. HTAs facilitate parallel programming by providing numerous methods that operate in parallel across tiles.

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, and the third parameter selects the data layout (`ROW` major in this case). The data type and the number of dimensions of the HTA are template parameters of the HTA class.

Tiles or scalars of HTAs can be selected using lists of integers and `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).

There are three main constructs in data-parallel computations:

– *Element-by-element operation*: A function is applied to each element of an array or corresponding elements of two or more conformable arrays.
– *Reductions*: These apply operations on an array to produce an array of lesser rank. For example, computing the sum of the elements of a one-dimensional array produces a scalar.
– *Scan*: It computes a prefix operation across all the elements of an array.

These operations take the form of three methods in the HTA library: `hmap` (which implements the element-by-element operation), `reduce` and `scan`.

The three constructs receive at least one argument, a function object whose `operator()` encapsulates the operation to be performed. In the case of `hmap`, the function may accept additional HTAs as parameters that must be comformable, that is, have the same tiling structure as the HTA instance on which the `hmap` is invoked. `Hmap` handles each tile separately so that the indexing of the elements inside the operation is relative to the first position of a tile. `Hmap` may be executed concurrently across the tiles of the HTAs to which it is applied.

2.1 Overlapped tiling

Stencil codes compute new values based on their neighbors as in the case of a(i) = a(i-1) + a(i+1). When this type of operation is applied to tiled arrays, elements of the neighboring tiles must be accessed during the processing of each tile. This can be done more easily and efficiently using shadow or ghost regions containing a copy of the elements of the neighboring tiles that are needed for the computation. The HTA library allows the automatic creation and update of these regions. This feature is called *overlapped tiling*.

Figure 2 shows the creation of an HTA similar to the one created in Figure 1 but with highlighted overlapped regions. Each of the shaded elements will be replicated to create the ghost region of the adjacent tile. The shape of the overlapping is determined by the optional `Overlap` object used in the creation of the HTA. Its constructor specifies, in this order, the overlap in the *negative* (decreasing index value) and *positive* (increasing index value) directions. In this example, a shadow region of size one is created both in the positive and negative directions and the boundary is periodic. Since we specified periodic boundaries, the last and first elements are replicated to create the ghost region of the first and last tiles, respectively.

2.2 Dynamic partitioning

The tiling structure of an HTA is specified at creation time. The dynamic partitioning feature enables the modification of the structure of an HTA after its creation 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.

Figure 3 shows an example of the use of dynamic partitioning. First, we add a new partition to the HTA created in Figure 1 using the `part` method which

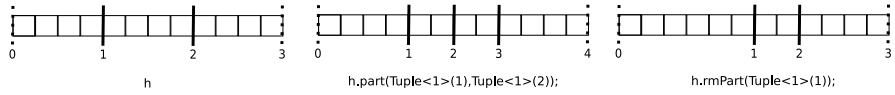

Fig. 3. Dynamic partitioning example

accepts two tuple parameters: the *source partition* and the *offset*. `part` inserts a new partition line along the ith dimension, $offset_i$ elements to the right of the location of $source\ partition_i$. In the example, a new partition is created with an offset of (2) from partition line (1).

In the second step, a partition is deleted using method `rmPart`. It receives as an argument the `Tuple` which specifies the partition to be deleted. In the case of the example of Figure 3, partition line (1) is removed.

3 The Intel TBB library

The Intel Threading Building Blocks (TBB) library enables the implementation of multithreaded task-parallel programs.

3.1 TBB operations

The element-by-element operation, reduction, and scan constructs are implemented in the TBB library using the `parallel_for`, `reduce` and `scan` *algorithm templates* respectively. The TBB library also includes the algorithm templates: `parallel_while`, which is used when loop limits are not pre-defined, and `pipeline` which is used when there is a sequence of stages that can operate in parallel on a data stream.

The `parallel_for`, `reduce` and `scan` algorithm templates accept two parameters: a *range* defining loop limits, and a function object representing the body of the parallel loop.

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 can create 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 4.3.

4 Implementation of Some Algorithms

The codes used in this comparison were taken from the chapter 11 of [1], which contains examples of parallel implementations of algorithms using TBBs[3]. This section describes some of them and it highlights the key differences between the TBB and HTA implementations using some snippets of code.

4.1 Average

This algorithm calculates, for each element in a vector, the average of the previous element, the next element and itself. It can be parallelized by the TBB library using the `parallel_for` construct. The TBB code that implements this algorithm is shown in Figure 4. In this code, the first and the last element 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 22-25 of the code. In line 27, the task scheduler object is created and initialized with 4 threads. The task scheduler is the engine in charge of the automatic mapping from 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 30 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 7 thru 16. 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 12).

The HTA implementation of this algorithm is shown in Figure 5. The data structures are created in lines 18-21. The padding values are automatically generated and filled in HTA `input` thanks to overlapped tiling. Line 19 defines an object that describes the overlapping of tiles in `input`. Shadows have size one in both the positive and negative direction and those in the boundaries are filled with zeros. In line 20 this overlapping specification is used to create an HTA with `N` values distributed in `nTiles`. Line 21 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 24. Its first argument is the operation to perform on each tile of the HTAs. This operation, `Average`, is defined as a `struct` in lines 5-10. `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.

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

```
1  #include "tbb/parallel_for.h"
2  #include "tbb/blocked_range.h"
3  #include "tbb/task_scheduler_init.h"
4
5  using namespace tbb;
6
7  class Average {
8  public:
9       float* input;
10      float* output;
11      void operator()( const blocked_range<int>& range ) const {
12           for( int i=range.begin(); i!=range.end(); ++i )
13                output[i] = (input[i-1]+input[i]+input[i+1])*(1/3.0f);
14      }
15      ...
16 };
17
18 const int N = 100000;
19 static int nThreads = 4;
20
21 int main( int argc, char* argv[] ) {
22      float raw_input[N+2], output[N];
23      raw_input[0] = 0;
24      raw_input[N+1] = 0;
25      float* padded_input = raw_input+1;
26      ... /* Initialization not shown */
27      task_scheduler_init init(nThreads);
28
29      Average avg(padded_input,output);
30      parallel_for ( blocked_range<int>( 0, N, 1000 ), avg );
31
32      return 0;
33 }
```

Fig. 4. TBB implementation of the *Average* algorithm

4.2 Seismic

This code performs a simple seismic wave simulation (wave propagation), using a few arrays.

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 6(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 6(b).

4.3 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, $[begin1, end1)$ (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.

```
1  #include " htalib_serial.h"
2  typedef HTA<float,1> HTA_1;
3  #define T1(i) Tuple<1>(i);
4
5  struct Average {
6      void operator()(HTA_1 input_, HTA_1 output_) const {
7          for( int i=0; i!=input_.shape().size() [0]; ++i )
8              output_[i] = (input_[i-1]+input_[i]+input_[i+1])*(1/3.0f);
9      }
10 };
11
12 const int N = 100000;
13 static int nTiles = 4;
14
15 int main( int argc, char* argv[] ) {
16      Traits::Default::init (argc,argv);
17
18     Seq< Tuple<1> > tiling(T1(N/nTiles),T1(nTiles));
19     Overlap ol(T1(1),T1(1));
20     HTA_1 input=HTA_1::alloc(1,tiling,ol,NULL,ROW);
21     HTA_1 output=HTA_1::alloc(1,tiling,NULL,ROW);
22      ... /* Initialization not shown */
23
24     input.hmap(Average(),output);
25
26     return 0;
27 }
```

Fig. 5. 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  M[0.3*UH+1: 0.5*UH] = 1.0/8;
4  Material[0.3*UH+1: 0.5*UH] = SANDSTONE;
5  M[0.5*UH+1: 0.7*UH] = 1.0/2;
6  Material[0.5*UH+1:0.7*UH] = SHALE;
7  M[0.7*UH+1:UH-1] = 1.0/8;
8  Material[0.7*UH+1:UH-1] = SANDSTONE;
```

(b) HTA version

Fig. 6. Terrain initialization

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.

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 7(a). The predicate `is_divisible` performs the test in step 1. The `ParallelMergeRange` class has two constructors. The first one, shown in lines 7-21, contains the dummy 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`.

```
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;
15         begin2 = m2;
16         end1 = r.end1;
17         end2 = r.end2;
18         out = r.out + (m1-r.begin1) + (m2-r.begin2);
19         r.end1 = m1;
20         r.end2 = m2;
21     }
22     ...
23 };
24 ...
```

(a) TBB version

```
1  ...
2  if (input1_size>GRAINSIZE) {
3   size1=input1_.shape().size() [0];
4   size2=input2_.shape().size() [0];
5
6    if( input1_size < input2_size ) {
7      h2=input1_;h1=input2_;
8      std::swap(size1,size2);
9    } else {
10     h1=input1_;h2=input2_;
11   }
12
13   begin2_ptr=h2.raw();
14   end2_ptr=begin2_ptr+size2;
15
16   float *m2 = std::lower_bound( begin2_ptr, end2_ptr, h1[(size1-1)/2] );
17   int pos=m2-begin2_ptr;
18
19   h1.part(Tuple<1>(0),Tuple<1>((size1-1)/2));
20   h2.part(Tuple<1>(0),Tuple<1>(pos));
21   output_.part(Tuple<1>(0),Tuple<1>(pos+((size1-1)/2)));
22
23   output_.hmap(Merging(),h1,h2,0);
24   ...
25 } else {
26 ...
```

(b) HTA version

Fig. 7. Parallel 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 7(b), is implemented using the dynamic partitioning feature. Lines 19-21 add new partitions to the two input HTAs and, the output HTA in the points selected as described in step 3 of the algorithm. Line 23 calls `hmap` recursively with the repartitioned structures. In this call, `hmap` applies its functor argument on each chunk in parallel. After this call these partitions are removed using `rmPart`. The recursion finishes when the sequences to merge are smaller than a given threshold, then step 1 is performed.

5 Qualitative Comparison

Both Hierarchically Tiled Arrays (HTAs) and Threading Building Blocks (TBBs) 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 can be more costly in the distributed memory environment.

The approach of TBBs, which are 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. 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 2. The TBB library implements these operations using a `parallel_for`, a `reduce`, and a `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 6). 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.

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 Section 4.1

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 such primitives are needed, they can be used in codes which use the HTA library, since both libraries can be used in the same program. Nothing special is needed to make any of them aware of the usage of the other one.

One interesting property of the TBBs which is not available in today's implemented HTA library is the ability to subdivide the range to process depending on the number of available processors. If one of the processors finishes very soon,

Code	Lines (HTA)	Lines (TBB)	HTA reduction
Average	28	39	+28%
Seismic	304	295	-3%
Parallel merge	70	74	+5.4%
Game of life	97	309	+69%
Substring finder	49	49	0%

Table 1. Number of lines for the five codes parallelized in the HTA and TBB version

Code	HTA					TBB				
	1	2	3	4	8	1	2	3	4	8
Average	490	403	381	260	253	536	193	189	190	196
Seismic	1993	1060	1010	778	503	1500	802	832	670	483
Parallel merge	8783	4704	4591	3885	3365	11823	5543	5144	3968	3793
Game of Life	21472	115731	8568	6802	5182	19788	11685	8976	7593	5520
Substring finder	6180	3130	2350	1570	810	6413	3200	2130	1605	810

Table 2. Times, measured in milliseconds, for both the TBB and HTA versions using 1,2,3,4 and 8 processors respectively

the amount of remaining work in another processor can be recursively divided to generate a new subrange assigned to the idle processor. This feature can be implemented of the HTA library.

6 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 1 for both the TBB and HTA version of the codes introduced in Section 4 and two other ones implemented also using the features covered in that section: The Game of Life and the Substring Finder. All these codes are also included in Chapter 11 of [1]. 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 from 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 2 shows the times in milliseconds for the execution of both the HTA and TBB versions of the codes. The machine used for the tests had two Quad

Tiles	4	8	16	20	40	50	80	100	200	250	400	500
Times	6953	5182	4341	4274	4130	4219	4399	4929	5585	6240	9254	12135

Table 3. Times, measured in milliseconds, for different number of tiles per dimension for the HTA version of the Game of Life on a 2000x2000 grid on 8 processors

core 2.66 Ghz Xeon processors and used version 4.2.1 of the GCC compiler with optimization level three. Several measurements were taken using 1,2,3,4 and 8 of the processors available in this machine. The results show that the times obtained using the HTA versions are approximately on par with those of the TBB versions. In these experiments, one tile per processor 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 processor per dimension was used. This does not imply a dependence on the number of processors as HTAs are objects created at runtime whose tiling structure is computed dynamically. Thus the number of processors can be obtained dynamically and used in a general computation of the desired tiling structure.

In Table 3, one can see the results of experiments where we used significantly more tiles than threads for the HTA Game of Life. We observed that the performance of the HTA version can improve almost 25% on what is shown in Table 2 by increasing the number of tiles. The top performance can be seen when the tiles under computation fit into L1 cache. Additional benefit comes from the dynamic distribution of work on the available threads as the parallel computations in our implementation inherit from TBBs. This is possible due to the overdecomposition of the problem.

Parallel computations in the HTA library are implemented using TBB parallels construct 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 possibly changing 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.

7 Conclusions

We have compared Intel TBBs and HTAs, two libraries devoted to facilitating the programming of multicore machines. For this purpose several algorithms were implemented using both libraries. The evaluation shows that the HTAs codes are shorter or on par with the length of the TBB ones. However, 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 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. Dynamic partitioning seems to be more efficient than ad-hoc TBB `Ranges` and sometimes both coding and performance improvements can be observed due to features like overlapped tiling, as in the case of the Game of Life code.

These two libraries can coexist in the same program. The HTA library seems a more natural way to express data-parallelism, which arises frequently in real programs, while the TBB offers more flexibility and can be used to solve other situations for which HTAs may not be suitable.

An interesting property of TBBs not yet implemented in the HTA is the ability to repartition the work in an automatic way according to the number of idle processors. Thus our future work involves enabling the automatic repartitioning of HTAs dynamically according to the number of idle processors in a similar way to the behavior of ranges in the TBB library.

References

1. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. 1 edn. O'Reilly (July 2007)
2. Butenhof, D.R.: Programming with POSIX Threads. Addison Wesley (1997)
3. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R.: Parallel programming in OpenMP. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)
4. Bikshandi, G., Guo, J., Hoeflinger, D., Almasi, G., Fraguela, B.B., Garzarán, M.J., Padua, D., von Praun, C.: Programming for parallelism and locality with hierarchically tiled arrays. In: Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'06). (March 2006) 48–57
5. Bikshandi, G., Guo, J., von Praun, C., Tanase, G., Fraguela, B.B., Garzarán, M.J., Padua, D., Rauchwerger, L.: Design and Use of htalib - a Library for Hierarchically Tiled Arrays. In: Proc. of LCPC 2006. Volume 4382 of LCNS., Springer-Verlag (Nov 2006)