Hierarchically Tiled Array Vs. Intel Thread Building Blocks for Multicore Systems Programming

Diego Andrade
Universidade da Coruña

James Brodman
University of Illinois Urbana-Champaign

Basilio B. Fraguela
Universidade da Coruña

David Padua
University of Illinois Urbana-Champaign

Abstract

Multicore systems are becoming common, while programmers cannot rely on growing clock rate to speed up their application. Thus, software developers are increasingly exposed to the complexity associated with programming parallel shared memory environments. Intel Threading Building Blocks (TBBs) is a library which facilitates the programming of this kind of system. 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 block-recursive nature. The model underlying HTAs provides programmers with a 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 the ease of programming and the relative performance of both approaches.

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 tried 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

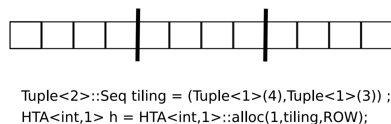

Figure 1: Creation of a HTA example

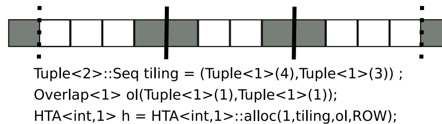

Figure 2: Overlapped tiling example

several tasks which are mapped automatically onto physical threads that are scheduled by the system. The Intel Thread Building Blocks (TBBs) library [5] 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 [3] which provide minimal functionality and for this reason some consider this approach the assembly language of parallelism. A third strategy to implement task-parallel programs is to use the `OpenMP` [4] set of compiler directives. `OpenMP`, however, is not as powerful as TBB, and it is mainly suitable for regular computations.

On the other hand, the Hierarchically Tiled Array (HTA) library [1, 2] enables the implementation of data parallel programs. An HTA is an array whose 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. Contrary to the approaches mentioned above, HTAs are suitable for shared, distributed and hybrid memory systems. The HTA library implementation for shared memory is implemented using 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 each implemented algorithm 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, as operations on them are defined to process so that their tiles may be processed concurrently.

Figure 1 shows the creation of an HTA with 3 tiles of 4 elements each. The 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 second line creates the HTA. The number of levels of tiling is passed as the first parameter, the tiling is specified by the second parameter,

and the third selects the data layout (`ROW` in this case). The data layout, which specifies how data will be stored in memory, can be row mayor (`ROW`), column major (`COLUMN`) or `TILE`. `TILE` specifies that the elements within a tile should be stored by rows and in consecutive memory locations. The data type and the number of dimensions of the HTA are template parameters of the HTA class.

HTA indices in the C++ library are zero-based. Individual tiles or scalars 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*: Values are assigned to different elements of one or more arrays, and each element is assigned at most once.
- *Reduction*: It applies reduction functions such as sum,maximum,minimum or logical and across all the members of an array.
- *Scan*: It computes a prefix operation across all the elements of an array.

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

Each of these 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 several additional HTAs parameters which must have the same tiling structure as the HTA instance on which the `hmap` is invoked. `Hmap` targets each tile separately so that the indexing of the elements inside the operation is relative to the first position of the processed tile. `Hmap` is executed concurrently across the tiles of the HTA to which it is applied.

2.1 Overlapped tiling

Stencil codes compute new values based on their neighbors. 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 achieved 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 overlapped regions. The overlapping is specified in the second statement. The `Tuple` passed as the first argument specifies the amount of overlap for each tile in each dimension in the *negative direction* (decreasing index value). The second `Tuple` specifies the same but in the *positive direction* (increasing index value). Thus these parameters define the size of the overlap and the size of the boundary region that is built around the array. The third argument specifies the nature of this boundary region and its value can be `zero` or `periodic`. In the first case, the boundary region is filled with constants, while in the second

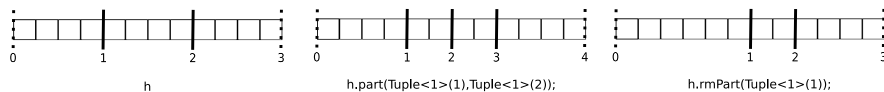

Figure 3: Dynamic partitioning example

case the boundary is periodic, i.e., it will replicate the values of the array on the opposite side. The HTA is created in the third statement, in this case we pass an additional argument the `ol` object, which specifies the desired overlapping. In this example, a shadow region of size one is created both in the positive and negative direction and the boundary is periodic. The shadow regions are highlighted in the figure.

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. We call the abstract lines that separate the tiles of an HTA *partition lines*. These lines include two implicit ones along each dimension, one before the first element and the other after the last element of the HTA, as well as the explicit partition lines specified by the user. Partition lines are numbered in ascending order starting at 0. A set of partition lines, containing no more than one line in each dimension, defines a *partition*. It is represented by a `Tuple` which for each dimension contains either partition number or the special symbol `NONE`. The dynamic partitioning feature allows the modification of the tiling structure of an HTA by removing partitions or adding new ones.

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 accepts two 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 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 was developed by Intel for the programming of multithreaded applications. As mentioned above, the TBB library enables the implementation of 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 has two additional interesting algorithm templates: `parallel_while`, which is used for unstructured workloads where the bounds are not clearly 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* and a function object. This object overloads the `operator()` and defines the operation to be performed on the range assigned.

The range identifies either (1) the set of index values to be used in the evaluation of each execution of the loop body, or (2) the number of times an the loop body must be executed. The range is split recursively into subranges by the task scheduler and mapped onto physical threads. A TBB `Range` is defined as a template, parameterized with a data type. The TBB library provides standard ranges, such as `blocked_range`, which expresses in terms of a lower bound, an upper bound, and optionally, a grain size, a linear range of values of the type specified by the template. The grain size is a guide for the workload size that the scheduler will use for the parallel tasks. The optimality of this value affects the performance and load balance of the parallel operation.

An interesting feature of the TBB library is the possibility of creating ad-hoc ranges. That is, the user can define its own range classes implementing specific policies to decide when and how to split them, 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 [5], which contains examples of parallel implementations of algorithms using TBBs[1]. 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 using 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. Lines 27-29 contain the initialization of the array with random values. In line 31 the

[1] 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];
23      raw_input[0] = 0;
24      raw_input[N+1] = 0;
25      float* padded_input = raw_input+1;
26
27      for ( size_t i = 0; i < N; ++i ) {
28        padded_input[i] = (float)(rand() % 1000);
29      }
30
31      task_scheduler_init init(nThreads);
32
33      Average avg(padded_input,output);
34      parallel_for ( blocked_range<int>( 0, n, 1000 ), avg );
35
36      return 0;
37 }
```

Figure 4: TBB implementation of the Average algorithm

task scheduler object is created and initialized with n threads, in the example 4. 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 any TBB library capability is used.

The first argument of the `parallel_for` in line 34 is a range which includes the whole vector. The TBB code selected a grain size of 1000. 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 indices for each subrange are directly extracted from the range parameter using the `begin`() and `end`() methods (line 12). In what follows, most of the lines of the initialization and of the declaration of the structures involved in the code will be omitted.

The HTA implementation of this algorithm is shown in Figure 5. The data structures are created and initialized between lines 18 and 22. `input` and `output` in lines 20 and 21 are one-dimensional HTAs of floats. Line 19 defines an object that describes an overlapping region. Shadows have size one in both the positive and negative direction and those in the external boundaries of the HTA are filled with zeros. In line 20 this overlapping specification is used to create an `input` HTA with `n` values distributed in `nT` tiles. The padding values are automatically generated and filled in this HTA thanks to overlapped tiling. Line 21 allocates the HTA where the result will be stored which has the same topology as those

```
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 }
```

Figure 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

Figure 6: Terrain initialization

used as an input but with no overlapped regions.

In line 24, the `hmap` method is invoked. 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 goes from 0 to `input.shape().size()[0]`, the size of the current tile. The HTA method `shape` returns a structure which defines the topology of an HTA. Its method `size` returns a `Tuple` with the size of each dimension of the tile. In the reminded of this paper, we will use (1,2,...) instead of `Tuple<n>(1,2,...)` to specify n-dimensional tuples. This notation cannot be used in our current HTA implementation, but we plan to add it. In this example the notation `Tuple<1>` was abbreviated using the `T1` constant.

4.2 Seismic

This code performs a simple seismic wave simulation (wave propagation). The main steps of the program correspond to the simulation of a seismic wave in a

```
1  struct UpdateStressBody {
2   void operator()( const tbb::blocked_range<int>& range ) const {
3    drawing_area drawing(0, range.begin(), UniverseWidth, range.end()−range.begin());
4    int i_end = range.end();
5    for( int y = 0, i=range.begin(); i!=i_end; ++i,y++ ) {
6     color_t * c = ColorMap[Material[i]];
7     drawing.set_pos(1, y);
8     for( int j=1; j<UniverseWidth−1; ++j ) {
9        S[i][j] += (V[i][j+1]−V[i][j]);
10       T[i][j] += (V[i+1][j]−V[i][j]);
11       int index = (int)(V[i][j]*(ColorMapSize/2)) + ColorMapSize/2;
12       if( index<0 ) index = 0;
13       if( index>=ColorMapSize ) index = ColorMapSize−1;
14      drawing.put_pixel(c[index]);
15     }
16    }
17   }
18  };
19   ...
20   tbb::parallel_for (
21        tbb::blocked_range<int>( 1, UniverseHeight−1, GrainSize ),
22        UpdateStressBody() );
23   ...
```

(a) TBB version

```
1  struct UpdateStressOp {
2  void operator() (HTA<value,2> S_tile, HTA<value,2> T_tile, HTA<value,2> V_tile){
3   int size_0 = S_tile.shape().size() [0];
4   int lower_bound_0=S_tile.memMap().leafPos()[0];
5   int index;
6
7   S_tile [0 :size_0 −1][1:UniverseWidth−1] += V_tile[0:size_0−1][2:UniverseWidth]−V_tile[0:size_0−1][1—
    :UniverseWidth−1];
8   T_tile [0 :size_0 −1][1:UniverseWidth−1] += V_tile[1:size_0][1:UniverseWidth−1]−V_tile[0:size_0−1][1—
    :UniverseWidth−1];
9
10  drawing_area drawing(0, lower_bound_0, UniverseWidth, size_0−1);
11
12  for(int i=0; (i!=(size_0)); ++i) {
13   color_t * c = ColorMap[Material[i+lower_bound_0]];
14   drawing.set_pos(1,i);
15   for( int j=1; j<(UniverseWidth−1); ++j ) {
16    index=(int)(V_tile[i][j]*(ColorMapSize/2)) + ColorMapSize/2;
17    if( index<0 ) index = 0;
18    if( index>=ColorMapSize ) index = ColorMapSize−1;
19    drawing.put_pixel(c[index]);
20   }
21  }
22 }
23 };
24  ...
25 S_inner=S[1:MAX_HEIGHT−2][0:MAX_WIDTH−1];
26  ...
27 S_inner.hmap(UpdateStressOp(),T_inner,V_inner);
28  ...
29 }
```

(b) HTA version

Figure 7: Update Stress

loop which sets the impulse from the source of the disturbance, does the two time consuming computations of update stress and velocity, and finally cleans up the edges of the simulation. The algorithm has two main parts: the initialization and the main loop, which is composed itself 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 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). In typical simulation the terrain characteristics would be passed as an input of the simulation. As this code is a benchmark they are initialized inside the code.

The function which updates the stress can be parallelized. It calculates the new values for the two matrices which contain the stress component of the simulation, `S` and `T`. The algorithm is an stencil computation which also used as an input the values of the matrix which contain the velocity, `V`. In this stage, it creates a pixel by pixel representation of the actual state of the wave simulation using the current values of matrix `V`. In the TBB version, shown in Figure 7(a), the outermost loop of the stencil is parallelized using a `parallel_for`. In the HTA version, shown in Figure 7(b), the same loop is parallelized using a `hmap` of the function defined between lines 2 and 22. Only the respective first dimensions of the involved HTAs are tiled. Lines 3 and 4 extract the size of the current tile and the absolute position of the lower element of the current tile, respectively. The stencil computation of the new values of matrices `S` and `T` is performed separately from the drawing and it has been rewritten using array notation. The drawing loop (lines 12-21), is similar to the TBB one, but the indexing of each tile considers relative positions inside that tile rather than absolute positions in the HTA, as it happened in the case of the average algorithm. As the stencil computation considers shifted values `V[i+1][j]` in the tiled dimension (first dimension), an overlapped region of size 1 in the positive direction of this dimension must be added when building this HTA (not shown in the figures). Besides, in three matrices `S`,`V` and `T`, dimension `i` will be only transversed from the second element to the penultimate element of that dimension. So we have to exclude the first position of the HTA in the first tile, and the last position in the last tile. This is achieved by applying the `hmap` operation in the area of interest of the HTAs. An example of how this is performed is in line 25. This problem is solved in the TBB version applying the operator on a `Range` of the indexes to be used, which excludes the first and the last point of the dimension (see line 21 in Figure 7(a)).

The function which updates the velocity is also an stencil computation, implemented in the TBB case using a `parallel_for` and a `hmap` in the HTA case. The implementation of both the TBB and HTA cases are very similar to those proposed for the implementation of the stencil computation inside the function

which updates the stress. The remaining parts of the code are sequential in both versions.

4.3 Parallel Merge

This code merges two sorted sequences. 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 $[a, b)$ indicates a partially opened interval), is 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. The two operations may be executed in parallel.

The TBB implementation of this algorithm, shown in Figure 8(a), implements the operation using a `parallel_for` (see lines 32-34). The subdivision of the sequences is implemented using an object of the ad-hoc range class `ParallelMergeRange`, defined in lines 1-23. 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 differentiate 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, lines 25-29, simply performs the merge sequentially by means of a `std :: merge`.

The HTA version of this algorithm, shown in Figure 8(b), 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, lines 6-27, is implemented using the dynamic partitioning feature. Lines 21-23 add new partitions to the two input HTAs and the output HTA in the points selected as described in the step 3 of the algorithm. Line 25 calls recursively `hmap` 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, see lines 27-35.

```
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
25 template<typename Iterator> struct ParallelMergeBody {
26     void operator()( ParallelMergeRange<Iterator>& r ) const {
27          std::merge( r.begin1, r.end1, r.begin2, r.end2, r.out );
28     }
29 };
30
31  ...
32  parallel_for (
33    ParallelMergeRange<Iterator>(begin1,end1,begin2,end2,out),
34    ParallelMergeBody<Iterator>()
35    );
36  ...
```

(a) TBB version

```
1  struct Merging {
2    void operator() (HTA<float,1> output_, HTA<float,1> input1_, HTA<float,1> input2_) {
3     ...
4    size1=input1_.shape().size() [0];
5    size2=input2_.shape().size() [0];
6    if  (input1_size>GRAINSIZE) {
7
8     if ( input1_size  < input2_size ) {
9       h2=input1_;h1=input2_;
10      std::swap(size1, size2);
11    } else {
12      h1=input1_;h2=input2_;
13    }
14
15    begin2_ptr=h2.raw();
16    end2_ptr=begin2_ptr+size2;
17
18    float  *m2 = std::lower_bound( begin2_ptr, end2_ptr, h1[(size1−1)/2] );
19    int  pos=m2−begin2_ptr;
20
21    h1.part((0),(( size1−1)/2));
22    h2.part((0),(pos));
23    output_.part((0),(pos+((size1−1)/2)));
24
25    output_.hmap(Merging(),h1,h2,0);
26     ...
27  } else {
28    float  *begin1_ptr=input1_.raw();
29    float  *end1_ptr=begin1_ptr+size1;
30    begin2_ptr=input2_.raw();
31    end2_ptr=begin2_ptr+size2;
32    float  *begin3_ptr=output_.raw();
33
34   std::merge(begin1_ptr, end1_ptr, begin2_ptr, end2_ptr, begin3_ptr);
35  }//end of else
36 }
37 };
38  ...
39 output.hmap(Merging(),input1,input2);
40  ...
```

(b) HTA version

Figure 8: Parallel Merge

4.4 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 consits of 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 9(a) and 9(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 first position of the current tile.

4.5 Game of Life

The Game of Life 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 next to each 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 which 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 and since the computation depends on the value of an element in am array and its neighbors, it is an stencil computation.

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 10(a) and 10(b). Besides the differences in the implementation between a `parallel_for` and a `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 the new state of a cell in an edge of a tile is computed a shadow region of size 1 is required in order to access the state of the neighbors that belong to

```
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;
13        max_pos = j;
14       }
15      }
16     }
17
18    max_array[i] = max_size;
19    pos_array[i] = max_pos;
20   }
21  }
22   ...
23 };
24  ...
25  parallel_for (blocked_range<size_t>(0, to_scan.size(), 100),
26                   SubStringFinder( to_scan, max, pos ) );
27  ...
```

(a) TBB version

```
1  struct SubStringFinderOp {
2  void operator() (HTA<int,1> max_, HTA<int,1> pos_) {
3       ...
4       init_i = max_.memMap().leafPos()[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;
17          max_pos = j;
18         }
19        }
20       }
21      }
22      max_[pos] = max_size;
23      pos_[pos] = max_pos;
24      pos++;
25     }
26 }
27 };
28  ...
29 max.hmap(SubStringFinderOp(),pos);
30  ...
```

(b) HTA version

Figure 9: Substring Finder

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

(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 }
13 };
14  ...
15 Overlap<2> * ol= new Overlap<2>(Tuple<2>(1,1),Tuple<2>(1,1),PERIODIC);
16 data= HTA<int,2>::alloc(1,((SIZEX/NTILESX,SIZEY/NTILESY),(NTILESX,NTILESY)),ol,NULL,ROW);
17  ...
18 for(int counter=1;counter<NSTAGES;counter++)
19   data.hmap(EvolutionOp(),data);
20  ...
```

(b) HTA version

Figure 10: Game of Life

another tile. The shadow region is created in lines 15 and 16, 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 15, 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 opposite 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 special need of the HTA library 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. The function `calculateState` of the class `Cell` has to compute the new state for each cell. In the TBB version, most of the time, the state of cell (i, i) depends on the state of its neighbors located in positions: $(i + 1, i)$,$(i + 1, i)$,$(i, i - 1)$,$(i, i - 1)$,$(i - 1, i - 1)$,$(i+1, i+1)$,$(i+1, i-1)$ and $(i-1, i+1)$. But, as we said before, in the edge region, the neighbor values must be searched in the opposite side of the matrix because is handled by means of a series of conditionals that choose the data to read in each direction from the cell of interest depending on its location. This complicates the implementation of the `calculateState` function. But in the case of the HTA version, as we have shadow regions around each tile as well as around the whole matrix filled using the PERIODIC criteria, the indexing of the neighbors can be always be performed using standard HTA indexing. Both versions of the `CalculateState` function are included in the A

5 Comparison of TBB and HTA capabilities

Both Hierarchically Tiled Arrays (HTAs) and Threading Building Blocks (TBBs) are libraries devoted to facilitate the expression of parallelism.

HTAs are a special type of arrays which may be organized into one ore more levels of tiles. When an operation is applied to this data structure, the different 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 tiled structure can be used for locality enhancement. More importantly, HTAs can be equally well executed 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 is to parallelize loops by specifying tasks using ranges which will be recursively subdivided. The distribution of the work is performed automatically by the task scheduler.

Much parallelism found in programs can be expressed as one of these three types of operations: element-by-element operation, reduction, and scan, already described in Section 2. The TBB library implements these operations using `parallel_for`, `reduce`, and `scan` operations respectively. The HTA library

uses alternatively `hmap`, `reduce`, and `scan` operations, respectively.

The manipulation of HTAs benefits from array-oriented notation. Some computations can be expressed in a more readable form using this notation instead of the alternative implementations using nested loops (see Figure 6) thi argument is supported by the measurement of the number of lines of code presented in next section. However, the advantage of the array notation goes beyond the lines of code. Array notation is intrinsically deterministic when only pure function are used, and should for all practical purposes completely avoid the possibly 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. Similar capabilities are supported in the HTA library by means of dynamic partitioning. One interesting property of the TBBs which could also be implemented for the HTA library is the ability to subdivide the range to process depending on the number of available processors. Besides, if one of the processors finishes very soon, the amount of remaining work in another processor can be recursively divided to generate a new subrange assigned to the idle processor.

The HTA library can define overlapped regions during the definition 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 of the facilities implemented in the TBB library are not implemented by any HTA construct such as *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. Still, the TBB library can be used in codes which use the HTA library, since both libraries can be used in the same program.

6 Evaluation

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	428	+77%
Substring finder	49	49	0%
Average value	109	177	+26.85

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	4665	3365	11823	5543	5144	3968	3793
Game of life	18761	9357	6785	5193	3915	63304	32491	22546	17740	12763
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 measurement of the impact of a library on the ease of programming is difficult to quantify. There is no formula to calculate exactly the readability of a program although experienced programmers can usually easy determine which implementation and notation are 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 for the five algorithms covered by this work for both the TBB and HTA version in Table 1. The fourth column stands for the percentage decrease of the source number of lines of code. As can be seen from the table, in some cases HTA codes are sinificantly shorter that the corresponding TBB codes and never meaningfully larger. This supports our claim that HTAs provide typically an easier implementation and better readability than the TBB library without extensions. The codes used in this comparison are those introduced in Section 4.

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 core 2.66 Ghz Xeon processors. The experiments were run using 1,2,3,4 and 8 of the processors available on this machine. For each case, the average time of 5 different executions is shown. The results show that the average times obtained using the TBB version of the code is slightly lower for the Seismic code. In this code there much computation devoted to adapt the HTA structures layout involved to the code to its special requirements, however, this is solved more efficiently in the TBB version. However, the execution of the HTA version performance improves in the case of the Average, Substring Finder, Parallel Merge and Game of Life. It seems that the dynamic partitioning is a more efficient way to express the Parallel Merge code than the ad-hoc TBB `Ranges`. The Game of Life HTA version is better because this code takes a great advantage of using an overlapped region with `PERIODIC` boundaries, which allows it to remove a costly `case` statement in the main core of the computation. Another reason why the HTA version is faster is the statical partitioning of task before the parallel work begins (except in Parallel Merge), while the TBB version cannot do this; it is forced to do the partitioning dynamically only once the parallel task execution has been requested, and each time the execution is requested. One

can observe that both approaches take advantage of an increasing number of cores. The times for the sequential versions of the codes are not shown since the serial implementation of some of them was not available in the TBB repository of codes. However, both the TBB and HTA versions of the codes obtain big speedups with respect to the serial implementations available. For example both the HTA and TBB parallel versions of the Substring finder code ran ≈ 8 times faster than the serial version using the 8 processors available in the machine.

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 usually shorter than the TBB ones. This is because array notation of some computations simplifies TBB codes with loops and conditional statements, dynamic partitioning is easier to use than ad-hoc TBB `Ranges` and overlapped regions avoids the programmer managing of padding regions. The performance results shows that the times obtained for the HTA version are smaller or slightly bigger than those obtained with the TBB one. Dynamic partitioning seems to be more efficient than ad-hoc TBB `Ranges` and sometimes we can take a big performance improving of using the HTA overlapped regions feature, like 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.

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

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

[3] David R. Butenhof. *Programming with POSIX Threads.* Addison Wesley, 1997.

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

[5] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism.* O'Reilly, 1 edition, July 2007.

A CalculateState function of the Game of Life code

A.1 TBB version

```
char Cell::GetAdjacentCellState(
                                        char* source,   // pointer to source data —
                                        block
                                        int x,              // logical width of —
                                        field
                                        int y,              // logical height of —
                                        field
                                        int cellNumber, // number of cell position to —
                                        examine
                                        cellPosition cp // which adjacent position
                                     )
{
        char cellState = 0;             // return value

        // set up boundary flags to trigger field-wrap logic
        bool onTopRow = false;
        bool onBottomRow = false;
        bool onLeftColumn = false;
        bool onRightColumn = false;

        // check to see if cell is on top row
        if (cellNumber < x)
        {
                onTopRow = true;
        }

        // check to see if cell is on bottom row
        if ((x*y)-cellNumber <= x)
        {
                onBottomRow = true;
        }

        // check to see if cell is on left column
        if (cellNumber%x == 0)
        {
                onLeftColumn = true;
        }

        // check to see if cell is on right column
        if ((cellNumber+1)%x == 0)
        {
                onRightColumn = true;
        }

        switch (cp)
        {
                case upperLeft:
                        if (onTopRow && onLeftColumn)
                        {
                                return *(source+((x*y)-1));
                        }
                        if (onTopRow && !onLeftColumn)
                        {
                                return *(source+(((x*y)-x)+(cellNumber-1)));
                        }
                        if (onLeftColumn && !onTopRow)
                        {
                                return *(source+(cellNumber-1));
                        }
                        return *((source+cellNumber)-(x+1));
                        break;

                case upper:
                        if (onTopRow)
                        {
                                return *(source+(((x*y)-x)+cellNumber));
                        }
                        return *((source+cellNumber)-x);
                        break;

                  ...
                  // code for upperRight, left, right, bottomLeft and bottomRight cases
                  ...
        return cellState;
}


char Cell::CalculateState(      char* source,   // pointer to source data block
                                        int x,                  // logical width of field
                                        int y,                  // logical height of field
```

```
                                    int cellNumber // number of cell position to examine
)
{
        char total = 0;

        total += GetAdjacentCellState(source, x, y, cellNumber, upperLeft);
        total += GetAdjacentCellState(source, x, y, cellNumber, upper);
        total += GetAdjacentCellState(source, x, y, cellNumber, upperRight);
        total += GetAdjacentCellState(source, x, y, cellNumber, right);
        total += GetAdjacentCellState(source, x, y, cellNumber, lowerRight);
        total += GetAdjacentCellState(source, x, y, cellNumber, lower);
        total += GetAdjacentCellState(source, x, y, cellNumber, lowerLeft);
        total += GetAdjacentCellState(source, x, y, cellNumber, left);

        // if the number of adjacent live cells is < 2 or > 3, the result is a dead
        // cell regardless of its current state. (A live cell dies of loneliness if it
        // has less than 2 neighbors, and of overcrowding if it has more than 3; a new
        // cell is born in an empty spot only if it has exactly 3 neighbors.
        if (total < 2 || total > 3)
        {
                return 0;
        }

        // if we get here and the cell position holds a living cell, it stays alive
        if (*(source+cellNumber))
        {
                return 1;
        }

        // we have an empty position. If there are only 2 neighbors, the position stays
        // empty.
        if (total == 2)
        {
                return 0;
        }

        // we have an empty position and exactly 3 neighbors. A cell is born.
        return 1;
}
```

A.2 HTA version

```
int CalculateState(HTA<int,2> data,   // pointer to source data block
                              Tuple<2> cellCoordinates // coordinates of cell position to examine
)
{
        int total = 0;

        total += data[Tuple<2>(cellCoordinates[0]-1,cellCoordinates[1]-1)];
        total += data[Tuple<2>(cellCoordinates[0],cellCoordinates[1]-1)];
        total += data[Tuple<2>(cellCoordinates[0]+1,cellCoordinates[1]-1)];
        total += data[Tuple<2>(cellCoordinates[0]+1,cellCoordinates[1])];
        total += data[Tuple<2>(cellCoordinates[0]+1,cellCoordinates[1]+1)];
        total += data[Tuple<2>(cellCoordinates[0],cellCoordinates[1]+1)];
        total += data[Tuple<2>(cellCoordinates[0]-1,cellCoordinates[1]+1)];
        total += data[Tuple<2>(cellCoordinates[0]-1,cellCoordinates[1])];

        // if the number of adjacent live cells is < 2 or > 3, the result is a dead
        // cell regardless of its current state. (A live cell dies of loneliness if it
        // has less than 2 neighbors, and of overcrowding if it has more than 3; a new
        // cell is born in an empty spot only if it has exactly 3 neighbors.
        if (total < 2 || total > 3)
        {
                return 0;
        }

        // if we get here and the cell position holds a living cell, it stays alive
        if (data[cellCoordinates])
        {
                return 1;
        }

        // we have an empty position. If there are only 2 neighbors, the position stays
        // empty.
        if (total == 2)
        {
                return 0;
        }

        // we have an empty position and exactly 3 neighbors. A cell is born.
        return 1;
}
```