Wavefront template implementation based on the task programming model

Antonio J.Dios, Rafael Asenjo, Angeles Navarro, Francisco Corbera and Emilio L. Zapata
Dept. of Computer Architecture
University of Malaga
Malaga, Spain
{*antjgm, asenjo, angeles, corbera, ezapata*}*@ac.uma.es*

***Abstract*—A particular characteristic of the parallel wavefront pattern is the multi-dimensional streaming nature of the computations that must obey a dependence pattern. Modern task based programming libraries like TBB (*Threading Building Blocks*) provide interesting features to improve the scalability of this kind of codes but at a cost of leaving some low level task management details to the programmer. We discuss such low level task management issues and incorporate them into a high level TBB based template that we present in this paper. The goal of the template is to improve the programmer's productivity to allow a non expert user to easily code complex wavefront problems without worrying about task creation, synchronization or scheduling mechanisms. In our template, the user only has to specify a definition file with the wavefront dependence pattern and the function that each task has to execute. In addition, we describe our experience with the TBB based template when coding four complex real wavefront problems, finding that the programming effort of the user is reduced from 25% to 50% at a cost of increasing the overhead below 5% when compared with manual TBB implementations of the same problem.**

I. Introduction

Wavefront is a programming pattern that appears in important scientific applications such as those based in dynamic programming [1] or sequence alignment [2]. In this paradigm, data elements or cells are distributed on multidimensional grids representing a logical plane or space. Although these elements have to be computed in a given order due to dependencies among them, there is also plenty of parallelism to exploit. One interesting feature of this type of problems is that computations can produce a variable number of independent tasks when the data space is traversed.

In a previous research [3] we have explored the applicability of the task programming model in the parallelization of the wavefront pattern. We have found that this pattern fits well with the task programming paradigm for a number of reasons: i) The current state-of-the-art task based libraries (OpenMP 3.0 [4], TBB [5], CnC [6]) provide a programming model in which developers express the source of parallelism using tasks, leaving the burden of explicitly scheduling these tasks to the library runtime, offering this way a more productive programming environment; ii) Tasks are much lighter weight than threads, which allow a more scalable parallel implementation of real wavefront problems (the workload of a cell use to be small -some floating point operations-); iii) The task schedulers of those library runtimes are based on the work-stealing scheduling algorithm, that leads to better load balancing than an OS thread scheduler, offering that way improved scalability [5]. In our research, we have evaluated the functionalities that each task library provides to the user when programming the wavefront pattern, and we have concluded that TBB offers some advantageous features that allow more scalable and efficient implementations of our pattern, namely the atomic capture of shared variables to control synchronization and the task passing (or task recycling) mechanism which is essential to reduce the task creation and scheduling overheads.

However, for non expert parallel programmers, it may be difficult to implement a parallel wavefront algorithm in TBB taking care of task creation, synchronization and task recycling activities. To alleviate these difficulties, we propose a high level template in which the programmer only has to provide the dependence pattern and the actual task computation. The proposed template is built on top of TBB, in a similar style to other TBB templates, like the "parallel_for" or "pipeline" ones. These templates main goal is to help programmers to parallelize the algorithm by using very high level constructors, so that an user who is not an expert in a parallel language is able to easily exploit a parallel architecture without worrying about platform details or low level task management mechanisms.

So the main focus of this paper is to describe the wavefront template, its use and programmability advantages, as well as to present the specific TBB features that we have exploited in its internal design (Section II). In addition, we have conducted several experiments with real and complex wavefront problems (including an H.264 video decoder) to evaluate the abstraction penalty due to the use of the template, its performance and its "programmability" (Section V). Finally we present some related work (Section VI) and conclusions (Section VII).

II. Wavefront Template

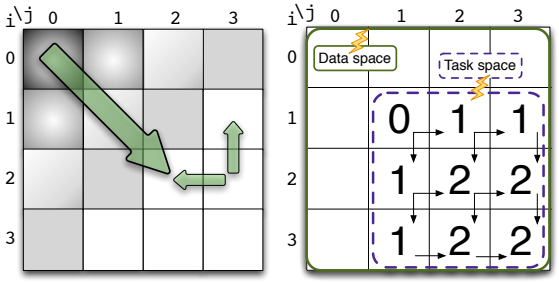
In order to illustrate a basic wavefront code implementation using TBB and how to rewrite it to take advantage of the

proposed wavefront template, we will first consider a simple 2D wavefront problem. It is a classic problem consisting in calculating a function, `foo`, for each cell of a $n \times n$ 2D grid [2], as we can see in the sequential code presented in Fig. 1. In that code, on each iteration of the i and j-loop, cells that were calculated in previous iterations are needed: A[i,j] depends on A[i-1,j] and A[i,j-1] (line 3). For that reason, although the data grid, A, is defined as $n \times n$, the iteration space is `[1:n-1, 1:n-1]`, since the first row and column of A are just initial values. The `gs` parameter of the `foo` function is used to tune the workload associate to each cell, such that we can analyze the performance for fine, medium and coarse grain tasks, as we will see in section V.

```
1 for (i=1; i<n; i++)
2   for (j=1; j<n; j++)
3     A[i,j] = foo(gs, A[i,j], A[i-1,j], A[i,j-1]);
```

Figure 1. Code snippet for a 2D wavefront problem

That way, each cell has a data dependence with two of the adjacent cells. For example, in Fig. 2(a), we see that cell $(2,3)$ depends on the north $(1,3)$, and west $(2,2)$ ones. Clearly, the cells in each anti-diagonal are totally independent so they can be computed in parallel. To exploit this parallelism (loops "*i*" and "*j*") a task will carry out the computations corresponding to each cell inside the iteration space (or task space from now on), and independent task will be executed in parallel.



(a) Sweep Diagonal (b) `counters` matrix

Figure 2. Typical wavefront traversal and dependencies translated into counters

In Fig. 2(b), the arrows in the task space show the data dependence flow for this wavefront problem. For example, after the execution of the upper left task $(1,1)$, which does not depend on any other task, two new tasks can be dispatched (the south $(2,1)$ and east $(1,2)$ ones). This dependence information can be captured by a 2D matrix with counters, like the one we show in Fig. 2(b). Note that, although the counter matrix has the same index domain as the data grid, tasks are assigned only to cells in the task space (region [1:n-1,1:n-1]), so the counters need to be initialized only for these coordinates. The value of the counters points out to how many tasks you have to wait for. That way, only a task with the corresponding counter nullified can be dispatched. Generalizing, each ready task first executes the task body and then, it will decrement the counters of the tasks depending on it. If this decrement operation ends up with a counter equal to 0, the task is also responsible of spawning the new independent task. In Fig. 3 we can see the code snippet for implementing this idea using the TBB library, as we describe in the next subsection.

A. TBB implementation

The matrix of counters, Fig. 2(b), has been implemented using the `atomic` type provided by TBB (`atomic<int>**counters`). TBB supports atomic captures, so that the expression "`if(--counters[i][j]==0) action()`" is safe and just one task will execute the "`action()`". In Fig. 3 we sketch the TBB code for the basic 2D wavefront example. In the main routine, we see the `counters` matrix initialization (lines 40–46), the initialization of the TBB scheduler (line 47) and the spawn of the $[1,1]$ task (line 48), which initially is the only one with the dependence counter equal to 0.

Following the TBB task based programming rules, we have to derive a class from the TBB task class, as we have done with class `Operation` in lines 1–7. Then, we have to override the virtual method `Operation::execute()` which does the actual task computation (lines 9–33). In this method, there are three key phases: i) computing the cell by calling `foo(gs,A[i][j],A[i-1][j],A[i][j-1])` (line 11); ii) decrementing and checking the neighbors counters (lines 14 and 18); and iii) spawning a new task or recycling the current one for the ready-to-run neighbors tasks (lines 22, 25, and 29).

We have taken advantage of the task recycling (or task passing) feature provided in TBB in order to reduce the number of spawns and scheduling overheads. The idea is that we set the flag `recycle_into_east` (line 15) if there is a ready to dispatch task to the left of the executing task. Otherwise, we set the flag `recycle_into_south` (line 20) if the south task is ready to dispatch. Later, according to these flags, we recycle the current task into the east (line 25) or south tasks (line 29). Note that, since in this example the data structure is stored by rows, if both, east and south, task are ready, the cache can be better exploited by recycling into the east task. That way, the same thread/core executing the current task is going to take care of the task traversing the neighbor data, so we make the most of the spatial locality. So in that case we recycle into the east task and spawn a new south task that would be executed later.

We performed some preliminary experiments to characterize the TBB runtime overheads in a basic 2D wavefront implementation in which only the `spawn()` function is invoked to create the ready tasks, without exploiting the task recycling mechanism. We found that the `spawn()`

```
Class Operation: public TBB::task
{
  int i, j;
  public:
    Operation(int i_ , int j_) :  i(i_), j(j_) {}
    task * execute();
};

TBB::task * Operation::execute()
{  //Task's work
   A[i][j] = foo(gs,A[i][j],A[i-1][j],A[i][j-1]);

  if (j<n-1){   // There is east neighbor
    if (--counters[i][j+1]==0)
      recycle_into_east = true;
  }
  if (i<n-1){  // There is south neighbor
    if (--counters[i+1][j]==0)
          if (!recycle_into_east){
            recycle_into_south = true;
          else
            spawn(i+1,j));
  }
  if (recycle_into_east){
          recycle_as_child_of();
          j = j+1;
          return this;
  }else if(recycle_into_south){
          recycle_as_child_of();
          i=i+1;
          return this;
  }else
          return NULL; //No ready neighbor
}

int main()
{
  atomic<int>**counters;
  ....
  for(i=1; i<n; i++)
    for (j=1;j<n;j++){
       if (i ==  1) counters[i][j] = 1;
       else if (j == 1) counters[i][j] = 1;
       else counters[i][j] = 2;
    }
  counters[1][1] = 0;
  task_scheduler_init init();
  spawn_root_and_wait  ( Operation(1,1) );
  ....
}
```

Figure 3. Coding details for an optimized and manually implemented wavefront algorithm using TBB

method, and some of the TBB task management methods (`get_task()` and `allocate()`) can become an important source of overhead when the number of cores increases and the granularity of a task is not sufficiently large. For instance, for task granularities of around 100 floating point operations, the above mentioned methods can suppose more than 35% of the execution time in 8 cores. When exploiting the task recycling technique, the basic 2D wavefront pattern has the opportunity of avoiding one call to `spawn()` by returning a pointer to the next task, thus the current task recycles into the new one. That way we achieve two goals: reducing the number of calls to `spawn()` (and `allocate()`) as well as saving the time for getting new tasks from the local queue (reducing the number of calls to `get_task`). Using this mechanism, we reduced the overheads of the mentioned methods by an order of magnitude, in fact they dropped by 3.5% of the execution time in 8 cores for our basic 2D code. When adding the cache concious functionality that prioritizes the recycling of the east ready task over the south ready task (that we incorporated in our optimized manual TBB version as we saw in Fig. 3) we improved the speedup of our basic 2D code in 45% (8 cores). In this case we observed a significant reduction of the L1D cache miss ratio.

In the next subsection we present an alternative implementation of the same code that take advantage of the high-level template we propose, in which the programmer does not have to deal with such low level details.

B. The Wavefront Template

The goal of the template is to minimize the time and effort the programmer will need to invest in the development of a wavefront algorithm. In order to do that, the template requires from the programmer the minimum amount of information to automatically generate a code performing similarly to the one shown in Fig. 3:

- A definition file with the wavefront dependence pattern that defines data and task domains as well as the dependences among tasks and the counter matrix values (see Fig. 4).
- The function that has to execute each task (see ExecuteTask method in Fig. 5).

```
1     //Section 1: Data grid
2     [0:n-1,0:n-1]
3     //Section 2: Task grid
4     [1:n-1, 1:n-1]
5     //Section 3: Indices
6    <i, j>
7     //Section 4: Dependency vectors
8    [1:n-2 , 1:n-2 ] -> (0,1); (1, 0)
9     [n-1, 1:n-2] -> (0, 1)
10    [1:n-2 , n-1] -> (1, 0)
11   //Section 5 (Optional): counter values
12    [1,1] = 0
13    [1, 2:n-1 ] = 1
14    [2:n-1 , 1] = 1
15    [2:n-1, 2:n-1 ] = 2
```

Figure 4. Definition file for the basic 2D wavefront algorithm

In Fig. 4 we can see the definition file for the running wavefront example. This file has five sections, although the last one can be omitted. Sections 1, 2, 4 and 5 use domains or regions to specify the index space in which some computations are valid. A region is a rectangular index set of arbitrary rank and stride. In particular, we use F90-Matlab notation to define a dimension of a region: it is a 3-tuple sequence of the form *l:h:s*, where *l* and *h* represent the low

and high bound of the sequence and s is the stride. A d-dimensional region is defined as a d-ary sequence of tuplas of the form $[l_1 : h_1 : s_1, \ldots l_d : h_d : s_d]$. When the stride in a sequence is 1, it can be omited from the corresponding tupla. A degenerate dimension (one with just a single index) can be declared by specifying the index value (e.g. $[1, 1 : n-1]$). When the dimension is indicated with "`:`", it represents the whole rank of the corresponding dimension in the data domain.

The first section of the definition file, line 2, points out the domain of the data grid while the second, line 4, indicates the domain of the task space. Then in line 6, we associate the `i` variable to the first dimension of the task space and `j` to the second one. These variables are not needed in this particular definition file, but they can be used to parametrize the dependence and counter information for more complex cases, as we will see later. The fourth section, lines 8 – 10, represents the dependence information. It is specified by a region in the LHS and a list of dependence vectors separated by ";" in the RHS. For example, line 8 captures the horizontal and vertical dependences of Fig. 2(b): in the region `[1:n-2,1:n-2]` the vectors `(0,1)` and `(1,0)` apply. In line 9 only the horizontal dependence applies to the last row; and in line 10 only the vertical dependence applies to the last column. Please, note that regions can not overlap and that a dependence vector represents relative directions or displacement from each cell in the corresponding region. Moreover, the programmer can provide locality hints when there is a list of vectors, like in line 8, because the order is significant: vectors defined first have higher priority. This is, as in the manual TBB implementation, in case that there are more than one ready to dispatch tasks, our template will choose to recycle the current one into the one pointed to by the higher priority vector. Therefore, if data is stored by rows, it is better to order the vectors as in Fig. 4 to prioritize the horizontal traversal of the data.

From the dependence information it is possible to automatically initialize the counter matrix (see Fig. 2(b)), but as we will see later, in some cases we can produce a faster initialization code if a definition for the counter values is provided as we do in the last section of the definition file (lines 12–15). As in the previous section, the left hand side identifies a region, but now, the right hand side, contains the value for each entry in that region (instead of a dependence vector).

With this dependence file, a code generator is able to produce a "wavefront.h" header file, comprising all the clases and methods to easily implement the wavefront code. So, the programer should invoque the code generator and include the resulting header file in his main module, as we see in Fig. 5, line 1.

In the main function, the programer should just call to `wavefront_init()` and `wavefront->run()` (lines 12 and 13). Basically, the first function calls the TBB initialization routine, initializes some template variables, allocates a new wavefront object and initializes the matrix of counters. Then, the `run()` method, defined in the `wavefront.h`, is invoked so that all the initial tasks are dispatched (those with the counter equal to zero).

```
1  #include "wavefront.h"
2
3  void Operation::ExecuteTask()
4  {
5    int i = GetFirst();
6    int j = GetSecond();
7    A[i][j]=foo(gs, A[i][j], A[i-1][j], A[i][j-1]));
8  }
9
10 int main(){
11   ....
12   wavefront_init(); // Initialize TBB and vars
13   wavefront->run(); // execute the wavefront code
14   ....
15 }
```

Figure 5. Task computation and main function

Besides, in the main module, the programmer needs to override the `Operation::ExecuteTask()` method with the corresponding operation that has to be carried out for each cell. Depending on the cell a given task is processing, `GetFirst()` (line 5) and `GetSecond` (line 6) template functions can be used to identify the cell coordinates, which are usually necessary to carry out the computation (line 7). This method will be further explained in section III.

C. Using the template

To further illustrate the use of the template we briefly show here how it should be used to implement four real wavefront problems. These are the Checkerboard [1], Financial [7], Floyd [8] and H.264 algorithms [9]. A short description of each algorithms follows.

The Checkerboard code simulates a board with $m \times n$ squares, where a cost function $c(i, j)$ returns the cost associated with square (i, j) (being i the row and j the column). The goal of the code is to find the shortest path (a path is the sum of the costs of the visited squares) to get to the last rank, assuming the checker can move only diagonally left forward, diagonally right forward, or straight forward. To compute the solution, we define the function $q(i, j)$, eq. 1, as the minimum cost to reach square (i, j):

$$q(i,j) = \begin{cases} \infty & j < 0 \; or \; j > n-1 \\ c(i,j) & i = 0 \\ f(i,j) & otherwise \end{cases} \quad (1)$$

where function $c(i, j)$ returns the cost associated with cells $[i, j]$ and $f(i, j)$ is computed as:

$$f(i,j) = min(q(i-1,j-1), q(i-1,j), q(i-1,j+1)) + c(i,j)$$

The Financial problem assumes that given m functions $f_1, f_2, ..., f_m$ (each one represents a financial interest function of a bank i) and a positive integer n (the budget), we want to maximize the function $f_1(x_1)+f_2(x_2)+...+f_m(x_m)$ with the restriction $x_1 + x_2 + x_3 + ... + x_m = n$ where $f_i(0) = 0(i = 1, .., m)$. Thus the goal, is to maximize the total financial interest of investing n euros in different banks. The x_i values represent the quantity of n to invest in the bank i. Now, to solve the problem we define a $m \times n$ matrix I where we keep partial values. The value $I(i, j)$ is computed by eq. 2, getting the solution to the problem in $I(m - 1, n - 1)$. Here, we will identify a task with the computation of $I(i, j)$:

$$I(i,j) = \begin{cases} f_1(j) & if\ i = 1 \\ \max\limits_{0 \le t \le j} \{I(i-1, j-t) + f_i(t)\} & otherwise \end{cases} \quad (2)$$

The Floyd's algorithm [8] uses the dynamic programming method to solve the all-pairs shortest-path problem on a dense graph. The method makes efficient use of an adjacency matrix $D(i, j)$ to solve the problem. The new values of that matrix are computed by eq. 3.

$$D_k(i,j) = \min_{k \ge 1} \{D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j)\} \quad (3)$$

Finally, we have also implement the H.264 decoder. H.264 or AVC (Advanced Video Coding) is a standard for video compression. In H.264, a video sequence have multiples video pictures called frames. A frame has several slices, which are self contained partitions of a frame that contain some number of MacroBlocks (MBs). MBs are blocks of 16×16 pixels and are the basics unit for coding a decoding. MBs in a frame are usually processed in scan order, which means starting from the top left corner of the frame and moving to the right, row after row. However, processing MBs in a diagonal wavefront manner satisfies all the dependencies and at the same time allows to exploit parallelism between MBs. In [9] a 2D wavefront version of the H.264 decoder is implemented using Pthreads. In particular, we focus on the `tf_decode_mb_control_distributed` function where the MBs are processed following the wavefront pattern. With this implementation as a starting point we have recoded this H.264 procedure taking advantage of our wavefront template.

In Fig. 6 we show the dependence pattern, counter matrix and corresponding definition file for these four algorithms. Although these definition files are self-explanatory if thoroughly analyzed together with the provided dependence pattern, we will pay attention to some remarkable details. For instance, in the Financial problem, although the data grid is $m \times n$, tasks will be dispatched only for the subregion `[1:m-1,1:n-1]` (all the cells but those in the first row and column). Also, dependence vectors are described with `[1:m-2,1:n-1]->(1,0:n-j-1)`, which means two things:

- The vectors are associated to cells with coordinates `(i,j)` that belong to the described region: `i` has to be in the range `[1:m-2]` and `j` in `[1:n-1]`, and
- For each one of the cells in the described region, there are $n - j$ dependent cells, identified by vectors `(1,0:n-j-1)`. So, for example, for cell `[2,1]`, the dependent cells are those in the next row (row $2 + 1$), and in columns within the range $1 + (0 : n - 1 - 1)$, so the dependent cells are (3,1), (3,2) and (3,3), if $n = 4$.

As we said, from that dependence vector it is possible to generate the counter matrix, but with an extra initialization time because, for each cell, a loop has to increment the counter of each dependent cell. On the other hand, if the programmer provides the counter information (`[2:m-1,1:n-1]=j`) a single traverse of the matrix will initialize each counter to the column index value, which is actually the number of arrows reaching each cell (see Fig. 6).

In the case of the Floyd algorithm, the sequential algorithm has a k-i-j triple nested loop in which D(i,j) is written in the inner loop body. For the wavefront approach we have removed the "j" dimension from the task space, so each task has to compute the $D(i, :)$ row for each `k` and `i` iterations, as we show in Fig. 7. That way, the indices in the task region has been defined as `(k,i)`. In that region, the subregion `[0:m-2,k+1]` identifies the cells in the superdiagonal due to these cells `i=k+1` for `k=0:m-2`. Besides, the subregion `[0:m-2,!(k+1)]` represents all the other cells (removing those in the last row), due to in these cases `i!=k+1`. Note that for this last region, the dependence vector `(1,0)` applies, since $D_{k-1}(i, :)$ has to be ready in order to compute $D_k(i, :)$ according to eq. 3. In addition, vectors `(1,-i:m-i-1)` associated to the superdiagonal cells, capture the fact that, as indicated in eq. 3, $D_{k-1}(k, :)$ is needed for all the `i` iterations of the iteration `k`. For example, if k=1, $D_0(1, :)$ is read in all the `i` iterations. Also note that, as in the Floyd's serial implementation, the wavfront algorithm can also be computed in place with a two dimensional D array due to D_k values can safely overwrite D_{k-1} ones.

```
1 void Operation::ExecuteTask()
2 {
3   int k = GetFirst();
4   int i = GetSecond();
5   for (int j=0; j<m; j++)
6     D[i][j] = Min(D[i][j], D[i][k]+D[k][j]));
7 }
```

Figure 7. Task body for the Floyd program.

Finally, to recode the original Pthread H.264 implementation using our wavefront template was not a big problem. The definition file presented in Fig. 6 capture the dependence

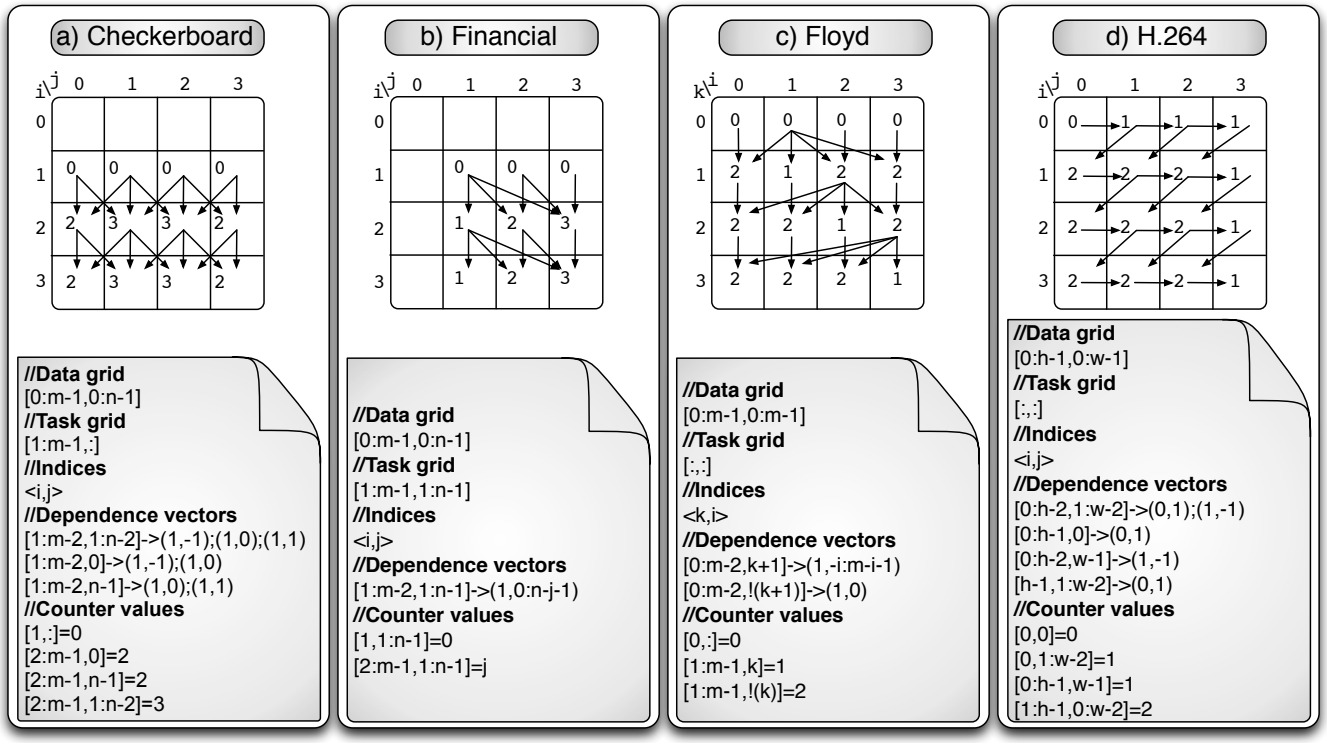


Figure 6. Dependence pattern, `counters` matrix and definition files for four wavefront algorithms

patter described in [9] and [10]. A data grid of $h \times w$ macroblocks, MBs, is defined, and a task takes care of each MB. Task dependences are described just with vectors (0,1) and (1,-1). Note that, although de (1,0) and (1,1) dependences also exist, these vectors are not really needed since they are implicit in the sequence (0,1)–(1,-1).

The original wavefront code in [9] is a Pthread code with a master thread and a pool of worker threads that wait for work on a task queue. The master thread is responsible for handling the initialization and finalization operations. The dependencies of each MB are expressed by a dependence table. Each time that the dependencies for a MB are satisfied a new task is inserted in the task queue. Synchronization among threads and the accesses to the task pool were implemented using semaphores. However, in our implementation, besides the already discussed definition file, we only need to specify the task body, as shown in Fig. 8. This highly simplify the task of porting the code to TBB, since thanks to the template, we are released of many responsibilities: taking care of pthread/task creation, managing the task queue, synchronizing tasks, to name a few of the them.

The H.264 task body of Fig. 8 uses a privatized work space (`local_WS`) per thread. This means that, although there may be thousand of tasks, in TBB these tasks will be carried out by the running threads, so the maximum number of simultaneous running tasks is equal to the number of threads. Since this number of threads is much smaller (in TBB it is usually equal to the number of cores to avoid oversubscription overheads), using a privatized work space per thread instead of per task is much more efficient. Thus, the function `Get_Local_Workspace` (line 4 in Fig. 8) identifies the thread in wich the task is running and the corresponding local work space. Then, some information of the current MB, identified by the `GetFirst()` and `GetSecond()` template methods, is copied to the local work space (line 7) and later decoded (line 8).

```
1 void Operation::ExecuteTask()
2 {
3     H264mb * curr_MB, * local_WS;
4     local_WS = Get_Local_WorkSpace();
5     curr_MB = Get_Curr_MB(GetFirst(),GetSecond());
6 // decode current MB
7     copy_MB_to_WS(local_WS, curr_MB);
8     decode_MB(local_WS);
9 }
```

Figure 8. `ExecuteTask()` method for the H.264 template implementation

III. Implementation details

The template has been implemented taking into account that it should support 2D and 3D wavefront

problems. In this scenario and without loss of generality, all tasks are internally identified with an integer, ID, starting at 0. The functions GetFirst(), GetSecond() and GetThird() can be used from a given task to convert its ID in the corresponding task coordinates (typically needed to access the data space). There are also functions GetFirstFromID(ID), GetSecondFromID(ID) and GetThirdFromID(ID), that are used internally if coordinates of task ID are needed. The inverse functions CoordinateToID(i,j) and CoordinateToID(i,j,k) are also available.

The internal machinery of the wavefront template is based on the Wave class, the TaskWave template class and the Operation class, derived from the latter and that can be used by the user. This Operation class has three public methods: ExecuteTask() that has to be overridden, as we have seen in figures 5, 7 and 8; GetCounter(ID) and GetDependency(o). The last two are automatically generated from the definition file and included in the wavefront.h, although they may be later modified if the programer realizes that they can be optimized somehow. In the wavefront.h it is also created the wavefront_init() function, which initializes some variables (data grid dimensions, task space frontiers, etc) and initializes a Wave<Operation> *wavefront object. In the constructor of this object the matrix of counters is allocated and all the counters are initialized inside a parallel loop that traverse all the tasks IDs. In each iteration of this parallel loop, the automatically generated method GetCounter(ID) is called. The GetCounter method generated from the 2D wavefront definition file (Fig. 4), is shown in Fig. 9. As we can see in this figure, the method just return a counter value depending on the coordinates of the corresponding task ID. At the time we are initializing the counter matrix, we also collect in a list the IDs of all the tasks with that counter equal to 0.

```
1  int Operation::GetCounter(int ID){
2    int i= GetFirstFromID(ID);
3    int j= GetSecondFromID(ID);
4    int counter = 0;
5    if((i==1) && (j==1))
6       counter= 0;
7    if((i==1) && (j>=2 && j<=m-1))
8       counter= 1;
9    if((i>=2 && i<=m-1) && (j==1))
10      counter= 1;
11   if((i>=2 && i<=m-1) && (j>=2 && j<=m-1))
12      counter= 2;
13   return counter;
14 }
```

Figure 9. GetCounter(ID) method generated from the basic 2D wavefront definition file

Once overridden the ExecuteTask() method and called the wavefrom_init() one, we can call the run() method of the just created wavefront object, as we saw in Fig. 5. This run() methods only spawn the ready to dispatch tasks which are in the list we gathered during the initialization. Each running task first executes the ExecuteTask() method and then the internally defined launch one. This launch method is the one that identifies all the dependent tasks, decrement their counters and spawn or recycle into them if they are ready. This process iterates until all the counters are nullified so there are no more tasks to spawn.

The identification of the dependen tasks is carried out by iteratively calling the GetDependency(o++) method. In Fig. 10 we show the automatically generated GetDependency(o) method for the 2D wavefront example. As we see, this method first identify the (i,j) coordinates of the calling task. Then, depending on the region in which this task is located, the corresponding vectors specified in the definition file will apply one by one. For example, in the basic 2D wavefront we defined three regions (see Fig. 4) and in the first region there are two vectors. Now, assuming a task is in the first region, each time this task calls the function GetDependency(o++), a dependence vector is applied: (0,1) the first time (o==1) and (1,0) the second one (o==2). For each vector, the resulting task coordinates are checked to validate if they are inside the task space and in that case the ID of the resulting task is returned. Otherwise, INVALID_POSITION is returned pointing out that the task is in one of the task space frontiers and that there are no more tasks in that vector direction. When there is no more vectors to follow in the region, NO_MORE_DEPENDENCIES is returned, so the launch() method realizes that all dependent task have been considered and dispatched when ready.

```
1  int Operation::GetDependency(int o){
2  int IDdepTask = NO_MORE_DEPENDENCIES;
3  int i, j, i1, j1;
4  i = getFirst();
5  j = getSecond();
6  if (Region1(i,j)){
7    if (o==1){
8        i1 = i + 0;  j1 = j + 1;
9        bool ok = CheckTaskCoordinates(i1,j1);
10       if (!ok) IDdepTask = INVALID_POSITION;
11       else IDdepTask = CoordinateToID(i1,j1);
12     }
13   if (o==2){
14       i1 = i + 1; j1 = j + 0;
15       bool ok = CheckTaskCoordinates(i1,j1);
16       if (!ok) IDdepTask = INVALID_POSITION;
17       else IDdepTask = CoordinateToID(i1,j1);
18     } }
19 else if (Region2(i,j){....}
20 else if (Region3(i,j){....}
21 return IDdepTask;
22 }
```

Figure 10. GetDependency() method generated from the basic 2D wavefront definition file

Now, in Fig. 11 we show the kernel of the tem-

plate machinery that is implemented in the `launch` method. Note that this method is called by a task just after executing the task body, `ExecuteTask()`. So the `lunch()` method has to decrement the counters of all the dependent tasks and dispatch them if they are ready. In line 5 we first take the ID of the first dependent task, `IDdepTask`, from the `getDependency()` method. Then, the `while` loop of line 6 will be visiting all the dependent tasks till `NO_MORE_DEPENDENCIES` is returned from `getDependency()`. For each one of these dependent tasks, first in line 7 we check if the ID of the dependent task is valid (`IDdepTask>=0`, due to `INVALID_POSITION` is a negative number) and in that cases we decrement the counter value of the task and check if it has been nullified. In that case, if we have not decided to recycle into a previous ready task (line 8), we spawn a new one, or otherwise (line 10) we annotate that we are going to recycle the current task into the new one. Note, that a task is going to recycle into the first ready to dispatch dependent task, so the order in which `GetDependency()` returns the task is relevant. This is, the dispatching mechanism is biased towards recycling into the first tasks returned by `GetDependency` and spawning the other ones for later execution. Therefore, dependent tasks that can better exploit the data in the cache of the current task should be returned first by `GetDependency()` and this can be achieved by adequately ordering the dependency vectors in the definition file.

Finally, in line 18, in case that the current task is going to recycle into a new one, we update the ID of the task by the one in which is going to be converted.

```
1 template<class TaskWave>  int Wave<TaskWave>::
      launch(task *t1){
2 TaskWave * t = (TaskWave*) t1;
3 int IDRecycledTask = -1;
4 int o=1;
5 int IDdepTask= t->getDependency(o++);
6 while(IDdepTask>=0 || (IDdepTask==INVALID_POSITION
      )){
7  if ((IDdepTask>0) &&(--counters[IDdepTask]==0)){
8     if (IDRecycledTask>=0){
9       t->spawn( *new( this->allocate(t)  )
            TaskWave(this,IDdepTask) );
10    }else{  // no recycled task yet
11      t->recycle_as_child_of(*t->parent());
12      IDRecycledTask = IDdepTask;
13    }
14  }
15  IDdepTask = t->getDependency(o++);
16 }
17 if (IDRecycledTask>0)
18   t->ID = IDRecycledTask;
19 return IDRecycledTask;
20 }
```

Figure 11. The launch() method checks the counters and spawns or recicles into new tasks.

That explained, if the programmer does not like to start from the definition file or it is difficult to express the wavefront dependences with the provided syntax, it is always possible to write the `GetDependency` and `GetCounter` methods from scratch.

In the next section we describe the BNF notation of the definition file grammar, and the code generator that translate a definition file into the corresponding `wavefront.h` header file.

IV. Definition file syntax

In Fig. 12 we see the BNF for the definition file syntax. The first rule in line 1 points out that the description file has a "Data Grid" line, a "Task Grid" one, an "Index" line, one or more "Dependency vectors" and zero or more "Counter values" definitions.

```
1 <Configurator> :: = <Datagrid> \n <TaskGrid>  \n <
      Index> \n +(<Dependencies> \n) *(<Counters> \n
      )
2 <Datagrid> :: = <Region>
3 <TaskGrid> :: = <Region>
4 <Region> ::= [<Vector>, <Vector>]
5 <Vector> ::= <exp> | <exp>  :  <exp>|<exp> : <exp>
       : <exp> | :
6 <Index> ::= < +<Letter> ,  +<Letter> >
7 <Dependencies> ::= <Pair> | <Vector_dep> | <
      RegionesDep>
8 <exp> ::=+<Digit>|+<Letter>|+<exp> <sign> <exp>
9 <Pair> ::= (<exp>, <exp>)
10 <Vector_dep> ::= (<Vector>,<exp>) | (<exp>,<Vector
      >)
11 <RegionDep> ::= <Region> -> <Dep>
12 <Dep> ::= <Dep>; <Dep>;| <Pair>;<Vector_dep>
13 <Counters> ::=<Region> =<exp>
14 <Letter> ::=a|b..|y|z
15 <Digit> ::=  0|1..|8|9
16 <Sign> ::= *|+|-|/|%
```

Figure 12. Syntaxis of the definition file for 2D wavefronts in BNF

V. Experimental Results

We conduct two main set of experiments, the first one to validate the efficiency of our template implementation and the second one to evaluate the template programmability. We have used as benchmarks the basic 2D, Checkerboard, Financial, Floyd and H.264 previously described codes.

A. Template Overhead

For these experiments, we have used a multicore machine with 8 cores, where each core is an Intel(R) Xeon(R) CPU X5355 at 2.66GHz, being SUSE LINUX 10.1 the Operating system in the target platform. The codes were compiled with icc 11.1 -O3. We executed each code 5 times and computed the average execution time of the runs to get the execution times. Then, we computed the speedups, that are calculated with respect to the sequential code time.

In a first experiment, we analyzed in detail the behavior of the template for our basic 2D wavefront running example. In particular, the goal of this experiment was to evaluate the scalability of our template for different task

granularities. In Fig. 13 we can see the speedups for two implementations of the 2D code and three task granularities: i) `TBB-Fine`, `TBB-Medium` and `TBB-Coarse` represent the speedups for the manual TBB implementation (see Fig. 3) that was used as baseline; ii) `Template-Fine`, `Template-Medium` and `Template-Coarse` represent the speedups for our TBB Template implementation (details are shown in Fig. 5. In the experiments, the granularity of a task is controled by the `gs` parameter of the `foo` function (line 3 in Fig. 1). That parameter set the number of floating point operations per task: Fine granularity represents 200 FLOP, Medium 2,000 FLOP and Coarse 20,000 FLOP. In all the cases, the matrix data has 10,000 $\times$ 10,000 cells. From the results of Fig. 13, we clearly see that the greater the granularity, the better the scalability of the codes (both manual and Template versions). Besides, we can see that the differences of performance in both implementations are small. Similar results were obtained for other matrix data sizes.

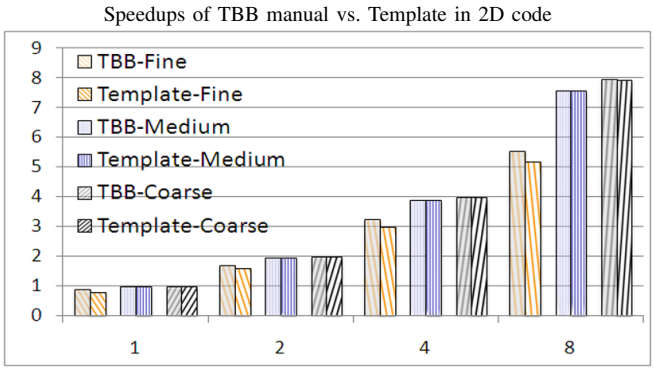


Figure 13. Speedup for the basic 2D wavefront codes for different task granularities (200, 2,000 and 20,000 floating point operations). The x-axis represents the number of cores

Fig. 14 represents now the overhead due to the Template for our 2D code and the different granularities. The overhead is computed as the ratio between the difference of times of the template and manual version vs. the time of the manual version. In other words, the increasing of time regarding the optimized manual version. As we can see in the figure, the overhead is not significant for the coarse, even the medium granularity cases (less than 2%). However, in the fine granularity case, the overhead due to the Template may increment the execution times from 6% to 9%. More in detail, in Fig. 15 we show the contribution to this overhead, due to the different stages of our template (see Fig. 5): the initialization (`wavefront_init` method, line 12) and the computation (`wavefront_run` method, line 13) stages. Clearly, the computation stage represents the main source of overhead on any granularity case. This overhead is mainly due to the method call costs on that stage.

In the next experiment, we focus in the Checkerboard,

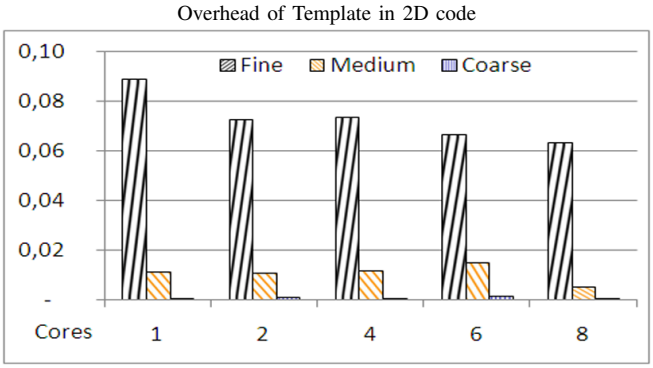


Figure 14. Overhead of the Template for the basic 2D wavefront code for different task granularities (Fine=200, Medium=2,000 and Coarse=20,000 Floating Point Operations). The x-axis represents the number of cores

Financial and Floyd codes. We selected a data matrix of 1,500 $\times$ 1,500 cells for Checkerboard, 300 $\times$ 300 elements for Financial and 5,000 $\times$ 5,000 elements for Floyd. In the Fig. 16 we show the speedups for two implementations of each code: `TBB XXX` represents the speedups for the manual TBB implementations whereas `Template XXX` represent the speedups for the Template versions. Now, from these results we see again that the differences of both implementations are small, although the TBB versions tend to scale better. Again similar results were obtained for other matrix input sizes.

We could mention that in all these real problems we found fine to medium grain size tasks (around 100-1,000 FLOP), with load unbalance among the tasks in the Financial and H.264 cases. As pointed in the basic 2D wavefront experiment, the finer the granularity of a task, the poorer the performance of the code. In particular, the Checkerboard corresponds to the case of finer granularity task, what explains the lower scalability. The reason for the poor performance in problems with fine grain was found in the next experiment. We used Vtune (the performance analyzer of Intel [12]) to analyze which functions of the TBB manual version of the problems consumed more time. We got that the degradation of performance in problems with fine grain size (Checkerboard) was because the overhead introduces by the TBB functions `spawn` (this method is invoked each time a new task is spawned), `allocate` (it selects the best memory allocation mechanism available) and `get_task` (this method is called after completing the execution of a previous task). In fact, these methods accounted for more than 6% of the total execution time. In problems with medium or big grain size (Floyd code) the overhead of those functions was smaller and it was far lower than 1% of the total execution time.

Now, Fig. 17 represents now the overhead due to the Template for our Checkerboard, Financial and Floyd codes. Again, the overhead is computed as the ratio between the

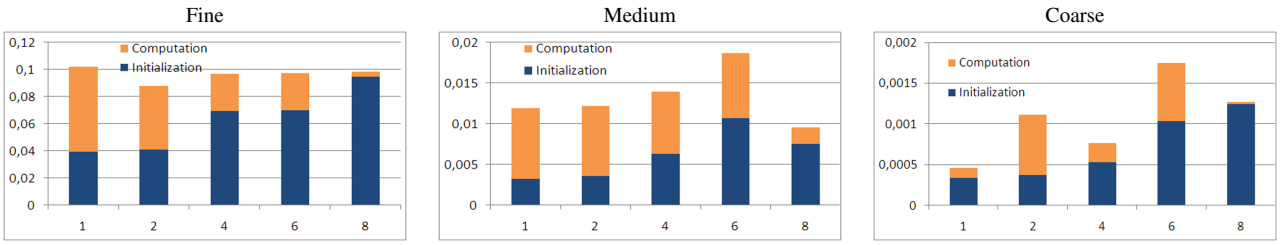


Figure 15. Contributions to the template overhead: the initialization stage and the computation stage. The x-axes represent the number of cores

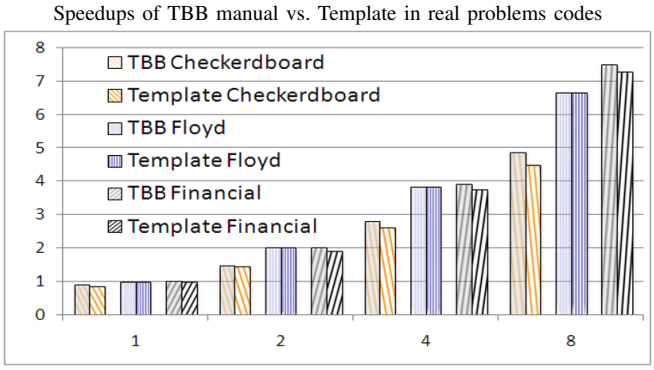


Figure 16. Speedup for Checkerboard, Floyd and Financial codes for the manual and Template implementations. The x-axis represents the number of cores

difference of times of the template and manual version vs. the time of the optimized manual version. As we can see in the figure, the overhead of the Template is small, ranging from 5% in the Financial code, to less than 0.5% in the Floyd code. In detail, Fig. 18 illustrates the contributions to the overhead, due to the initialization and computation stages of our template in these codes. Now, the initialization overhead has a very low impact specially for the Checkerboard and Financial codes, because the matrix data for these experiments, and therefore the counters matrices are quite small.

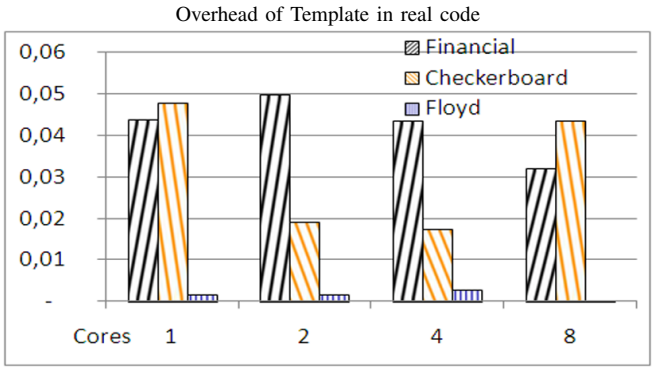


Figure 17. Overhead of the Template for the Checkerboard, Financial and Floyd codes. The x-axis represents the number of cores

Summarizing, these experiments have shown us that our Template implementation is quite efficient for a variety of wavefront codes. The abstraction penalty of our Template due to the higher level problem management supposes an increasing of the execution times, ranging for 8% of increment for codes with fine grain tasks, to less than 1% of increment of time for codes with a coarse grain tasks. We think these are quite small contributions, provided the Template improves the programmer productivity. This issue will be discussed in Section V-B.

In order to evaluate our template with even a more complex and large problem, we conduct a new experiment where we compare two parallel implementations of the H.264 decoder benchmark: the original Pthreads implementation [9] against the version based in our TBB Template. Both H.264 implementations were run in four eight-core Intel(R) Xeon(R) CPU X7550 2.00GHz (32 cores) running SUSE 11.1. The codes were compiled with g++ 4.1. We executed each code version 5 times and computed the average execution time of the runs to get the execution times. In particular, we measured the running times of the MBs decoding section (without CABAC). We executed each code version for different frame resolutions of the same video. The goal was now to evaluate the impact of different problem input sizes on the scalability of each version. In Fig. 19 we see the times (in ms.) for the two versions, in the low resolution (352 $\times$ 288 pixels=396 MBs), medium resolution (704 $\times$ 576 pixels= 1,584 MBs), and high resolution (1,280 $\times$ 720 pixels= 3,600 MBs) input video data.

From the results, we see that the overhead incurred due to our template is negligible. Besides, our TBB template implementation has a better scalability behavior for any video resolution. Although for low core counts (1, 2 cores) both implementations have similar times, we see that for higher core counts (12, 16, 32 cores), the performance of the Pthreads implementation tends to degrade faster than our TBB template implementation, specially for the medium and high resolution cases. In these cases, the overhead due to contention on the global Pthreads task queue is one of the reasons of the degradation in the Pthreads implementation, a problem that does not arise in the distributed task queue

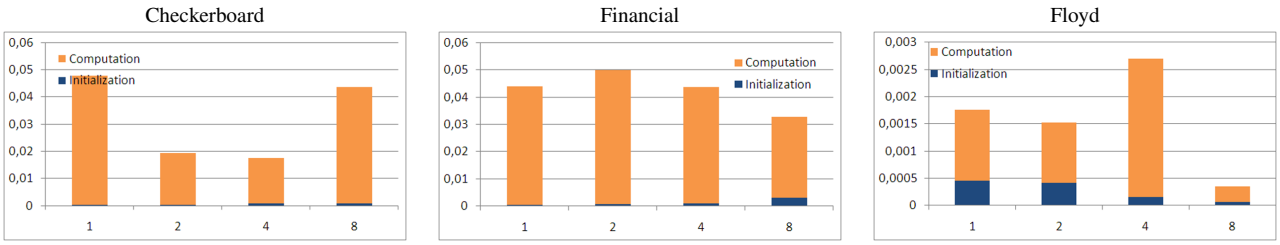


Figure 18. Contributions to the template overhead: the initialization stage and the computation stage in our real problems codes. The x-axes represent the number of cores

model of TBB. Anyway, the lack of scalability, specially for the low and medium resolution cases, is explained by the insuficient workload to keep busy all the cores.

B. Template Programmability

In this section we discuss our findings regarding our second experimental goal: to evaluate the template *Programmability*. In other words, how productive from a programmer point of view, is to use our Template to code complex wavefront codes. It is difficult to measure the ease of programming. We follow the methodology proposed in [13], where the authors suggest three quantitative metrics to measure the easy of programming of a code. These metrics are: the SLOC (*Source Lines Of Code*), the CC (*Cyclomatic Complexity*) and the PE (*Programming effort*).

When computing the SLOC, comments and empty lines are excluded. This metric is perhaps the more dependent on the user programming style than the other two metrics. In general, we can assume that higher values for this metric can originate more error prone and difficult to maintain code. Regarding the CC, the authors in [14] define this metric as the number of predicates plus one. In a well structured program, higher values for this parameter uses to mean a more complex code. Respecting the PE parameter, it is defined in [15] as a function of the number of unique operands, unique operators, total operands and total operators found in a code. The operands correspond to constants and identifiers, while the symbols or combinations of symbols that affect the value of operands constitute the operators. This metric can be representative of the programming effort required to implement an algorithm. So, a higher value of PE means that it is more difficult for a programmer to code the algorithm.

Fig. 20 shows the results of the Programmability metrics for our real wavefront programs: Checkerboard, Financial, Floyd and H.264. We compare the metrics on two implementations of each code: the manual TBB version (see Fig. 3) for the Checkerboard, Financial and Floyd codes or the original Pthreads implementation for the H.264 (see [9]), vs. the version using our Template. The SLOC, CC and PE values are normalized with respect to the corresponding values of the manual implementations. For all the codes, the metrics corresponding to our Template versions, have always smaller values. For instance, the SLOC, the CC and the PE for the Template codes are typically about 50% below of the corresponding manual TBB implementations, except in the H.264 case, where the PE metric for our Template implementation is about 25% below of the Pthreads version. In any case, these results seem to indicate that we gain some easy of programming when using our Template.

In summary, the experimental evaluation performed in this section has show us that the proposed TBB based Template for the wavefront pattern, suppose a productive tool with a low overhead cost, when coding complex applications.

VI. Related works

There have been several research works that have targeted the problem of parallelizing wavefront problems. In [16] the authors propose the concept of region-based programming and describe its benefits for expressing high level array computations in the context of the parallel language ZPL, being the wavefront pattern a particular case of array computation. Although the definition of the regions in that work is similar to the one we propose in this paper, we differ in the goal: the authors in [16] used the information provided by the regions to identify the communication patterns of the array computations in a distributed memory architecture, whereas in our work we just want to simplify the programmer effort when expressing the data dependence information of the problem.

Other authors have addressed the implementation of parallel wavefront problems on heterogeneous architectures such as SIMD-enabled accelerators [17], or the Cell/BE architecture [18]. In these works, the authors have focused on vectorization optimizations and on studying the appropriate work distribution on each architecture, forcing the programmer to deal with several low level programming details. We depart from these works in that we rather focus on higher level programming approaches that release the user from the low level architectural details.

Precisely, to free the user from dealing with those low level details, there has been substantial effort invested in characterizing parallel patterns. Patterns may also go by the name “algorithm skeleton”[19], being the wavefront one of these patterns [2]. In this research line, there have

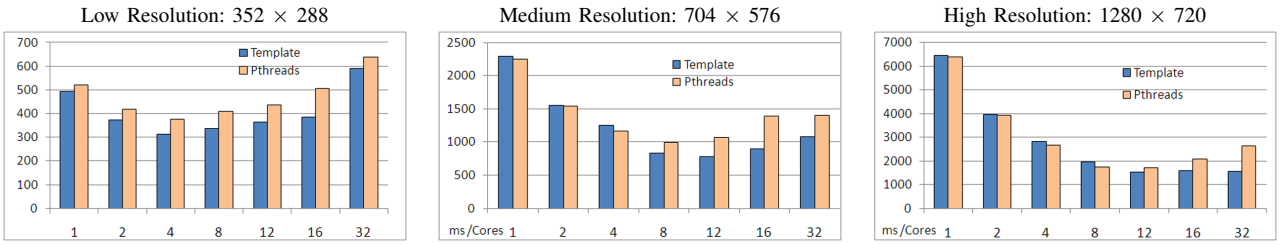


Figure 19. Time in seconds for the Pthreads and the TBB Template versions of H.264 and different frame resolutions. The x-axes represent the number of cores

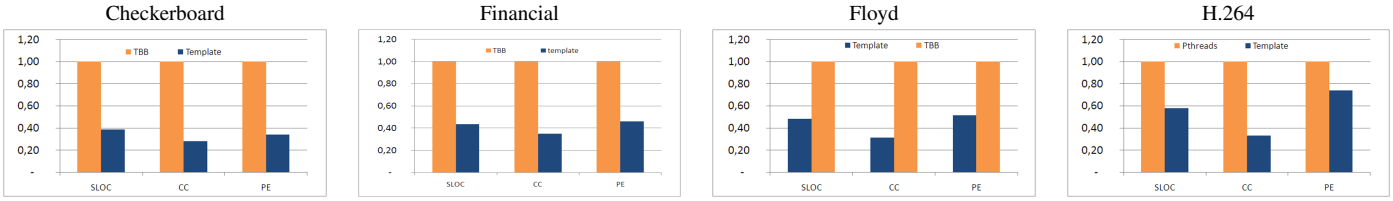


Figure 20. Program-ability metrics for the Checkerboard, Financial, Floyd and H.264 codes, comparing the manual (TBB) and Template implementations

been a recent proposal [20] in which the authors propose a "wavefront" abstraction for multicore clusters. We differ from this work in that they address specific regular wavefront problems, where the granularity of a computation is very coarse (one cell needs around 117 sec. in a 1Ghz CPU). They use Pthreads and rely on the O.S. scheduler to process the work. On the contrary, our study focus on much more fine task granularity workload, both in regular and irregular problems.

A distinguish feature of our work from all the previous related work, is that we propose a wavefront Template in the context of a modern task programming-based library. We identify the low level features of the task programming model needed in the wavefront pattern, and through a high level Template we hide them from the programmer.

VII. Conclusions

The parallel wavefront pattern is an interesting paradigm for which the task based TBB library has no template. Precisely in this paper we propose a TBB based template for this pattern that helps the programmer by hiding the low level task management mechanisms such as: i) the task synchronization through the use of the atomic capture; ii) the task recycling or spawning when a new computation has to be performed; and iii) the task priorization that can exploit the spatial locality. When using our template, the programmer only has to specify a configuration file with the dependence pattern information, and the function that each task has to perform (the ExecuteTask method). Using four complex benchmarks we have found that the abstraction penalty due to the template only supposes at most a 5% of additional overhead when compared to a manual TBB implementation of the same code. Even in the case of the H.264 code, the TBB-based template version outperforms the original Pthreads manual version, although in this case thanks to the work-stealing scheduler of the TBB library runtime. Additionaly, we have evaluated the programmability of our template in our four complex codes, using three quantitative metrics that characterize the effort of programming, finding that the template based codes reduce the effort programming metrics from 25% to 50% when compared to the manual versions. Therefore, for the evaluated codes, we conclude that our template is a productive tool with a low overhead cost.

References

[1] V. U. Dasgupta Sanjoy, Papadimitriou Christos, *Algorithms*. McGraw-Hill Higher Education, 2007.

[2] J. Anvik, S. MacDonald, D. Szafron, J. Schaeffer, S. Bromling, and K. Tan, "Generating parallel programs from the wavefront design pattern," *Parallel and Distributed Processing Symposium, International*, vol. 2, p. 0104, 2002.

[3] A. J. Dios, R. Asenjo, A. Navarro, F. Corbera, and E. L. Zapata, "Evaluation of the task programming model in the parallelization of wavefront problems," *High Performance Computing and Communications, 10th IEEE International Conference on*, vol. 0, pp. 257–264, 2010.

[4] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2009. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2008.105

[5] J. Reinders, *Intel Threading Building Blocks (Scientific and Engineering Computation)*. O'Reilly, 2007, http://www.threadingbuildingblocks.org/.

[6] "Intel concurrent collections for c/c++," http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc.

[7] G. Brassard and P. Bratley, *Fundamentals of algorithmics*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[8] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, pp. 345–, June 1962. [Online]. Available: http://doi.acm.org/10.1145/367766.368168

[9] M. A. Mesa, A. Ramirez, A. Azevedo, C. Meenderinck, B. Juurlink, and M. Valero, "Scalability of macroblock-level parallelism for h.264 decoding," *Parallel and Distributed Systems, International Conference on*, vol. 0, pp. 236–243, 2009.

[10] E. B. V. D. Tol, E. G. T. Jaspers, and R. H. Gelderblom, "Mapping of h.264 decoding on a multiprocessor architecture," 2003, pp. 707–718.

[11] A. Dios, R. Asenjo, A. Navarro, F. Corbera, and E. L. Zapata, "Wavefront template implementations based on the task programming model," in *Technical Report at http://www.ac.uma.es/∼asenjo/research/*, February 2011.

[12] S. E. S. Programme, "Intel(r) vtune(tm) performance analyzer 8.0.2 for linux," Mathematical Software Group Rutherford Appleton Laboratory Chilton, Tech. Rep., 2006, http://www.sesp.cse.clrc.ac.uk/.

[13] C. H. Gonzalez and B. B. Fraguela, "A generic algorithm template for divide-and-conquer in multicore systems," *High Performance Computing and Communications, 10th IEEE International Conference on*, vol. 0, pp. 79–88, 2010.

[14] T. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 4, pp. 308 – 320, dec. 1976.

[15] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. New York, NY, USA: Elsevier Science Inc., 1977.

[16] E. C. Lewis and L. Snyder, "Pipelining wavefront computations: Experiences and performance," in *In Fifth IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS*, 1999.

[17] O. Storaasli and D. Strenski, "Exploring accelerating science applications with fpgas," in *Proc. of the Reconfigurable Systems Summer Institute*, July 2077.

[18] A. M. Aji, W.-c. Feng, F. Blagojevic, and D. S. Nikolopoulos, "Cell-swat: modeling and scheduling wavefront computations on the cell broadband engine," in *CF '08: Proceedings of the 5th conference on Computing frontiers*. New York, NY, USA: ACM, 2008, pp. 13–22.

[19] J. Falcou, J. Srot, T. Chateau, and J. Laprest, "Quaff: Efficient c++ design for parallel skeletons," *Parallel Computing*, vol. 32, no. 7–8, pp. 604–615, 2006, algorithmic Skeletons.

[20] L. Yi, C. Moretti, S. Emrich, K. Judd, and D. Thain, "Harnessing parallelism in multicore clusters with the all-pairs and wavefront abstractions," in *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*. New York, NY, USA: ACM, 2009, pp. 1–10.