A case study of the task-based parallel wavefront pattern

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

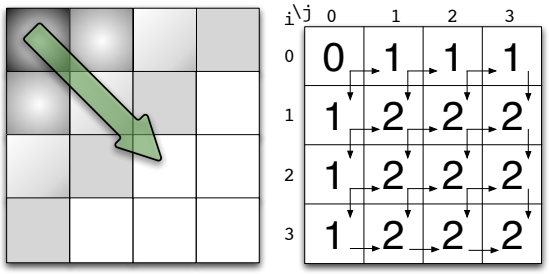
***Abstract*—This paper analyzes the applicability of the task programming model in the parallelization of the wavefront pattern. Computations on this type of problem are characterized by a data dependency pattern across a data space, which can produce a variable number of independent tasks through the traversal of such a space. We explore several implementations of this pattern, based on the current state-of-the-art threading libraries that support tasks. For each implementation, we discuss the particularities from a programmers point of view, highlighting the advantageous features in each case. We conduct several experiments to identify the factors that can limit the performance in each implementation. Moreover, we propose and evaluate some optimizations (task recycling, prioritization of tasks based on locality hints and tiling) that the programmer can exploit to reduce the overhead in some cases.**

Keywords:*Wavefront computation, Task programming model, Task recycling*

I. Introduction

Wavefront is a programming pattern that appears in scientific applications such as those based in dynamic programming [1] or sequence alignment [2]. In such a pattern, data elements are distributed on multidimensional grids representing a logical plane or space [3]. The elements must be computed in order because they have dependencies among them. One example is the 2D wavefront that we shown in Fig. 1(a). Here, computations start at a corner of the matrix and a sweep will progress across the plane to the opposite corner following a diagonal trajectory. Each diagonal represents the number of computations or elements that could be executed in parallel without dependencies among them.

Recently, there have been made available several parallelization frameworks [4], [5], [6], [7] that provide support for parallel tasks rather than parallel threads. We think task-based programming fits better than thread-based programming when addressing the implementation of parallel wavefront problems. Firstly, tasks are much lighter weight than logical threads, so in fine or medium computational workload wavefront applications (as in our case), tasks are a more scalable constructor. Secondly, task-based frameworks provide a programming model in which developers express the source of parallelism in their applications using tasks, while the burden of explicitly scheduling these tasks is managed by the library runtimes. Lastly, the task schedulers implemented in the library runtimes can get some high level information (provided by the user or a template), that can be used to dynamically redistribute the work across the available processors, even sacrificing fairness for efficiency, differently to how an O.S. thread scheduler works, offering that way improved scalability [5] .



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

Figure 1. Typical wavefront traversal and dependencies translated into counters

In this preliminary version of the paper we have focus our study on the frameworks that represent the state-of-the-art when programming using tasks: *OpenMP 3.0* [4], *Intel Threading Building Blocks* (TBB) [5], Cilk [8] and *Intel Concurrent Collections* (CnC) [6]. The goal of this paper is to explore the programmability and performance of different OpenMP, Cilk, TBB, and CnC implementations of a parallel task-based 2D wavefront pattern. We will start by highlighting the main features of the different implementations from a programmer's point of view (sec. II-B). Next, we conduct several experiments to identify the factors that can limit the performance for the different implementations (sec. II-C). From these experiments, we have found two important sources of overheads: synchronization and task creation/management. While overheads caused by locks or atomic operations within the runtime libraries are inevitable, unless specialized synchronization hardware becomes available, nevertheless task creation and task management overheads can be drastically reduced in our wavefront patterns, which can have a significant impact in the performance of fine task granularity codes. To reduce

these overheads, the user can guide the task scheduler by using the *task recycling* (or *task passing*) mechanism, as well as by prioritizing the execution of tasks to guarantee a cache-conscious traversal of the data structure, what might help to improve data locality. In particular, we have found that TBB provides enough flexibility to efficiently implement these optimizations, which we describe and evaluate in sec. III.

II. Implementations of a wavefront pattern based on tasks

A. The problem

As a case of study of the wavefront pattern, we select a simple 2D problem, that we show in Fig. 2. In this code, we compute a function for each cell of a $n \times n$ 2D grid [2]. Each cell has a data dependence with two elements of the adjacent cells. For example, in Fig. 1(a), we see that cell $(1,3)$ depends on the north $(0,3)$ and west $(1,2)$ ones, since 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])` (Fig. 2, line 3). Clearly, the diagonal cells are totally independent so they can be computed in parallel.

```
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 2. Code snippet for a 2D wavefront problem

In our task parallelization strategy, the basic unit of work is the computation performed by function `foo` at each (i,j) cell of the matrix. Thus, the intention is to parallelize the i and j loops (lines 1-2 in Fig. 2). Without loss of generality, we assume that there will be auxiliary work on each cell, and the computational load of this work will be controlled by the `gs` parameter of the `foo` function. That way, we can define the granularity of the tasks and therefore, we can study the performance of different implementations depending on the task granularity, as well as situations with homogeneous or heterogeneous task workloads, as we will see in section II-C.

B. Task-based approach

In Fig. 1(b), the arrows show the data dependence flow for our wavefront problem. For example, after the execution of the upper left task $(0,0)$, which does not depend on any other task, two new tasks can be dispatched (the one below $(1,0)$ and the one to the right $(0,1)$). This dependence information can be captured by a 2D matrix with counters, like the one we show in Fig. 1(b). The value of the counters points out to how many tasks you have to wait for. Only the tasks with the corresponding counter nullified can be dispatched.

Generalizing, each ready task first executes the task body and then it decrements 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. The pseudo code of this procedure is shown in Fig. 3. The `Task_Body()` function corresponds to the work that each task has to perform. It is important to note that the counters will be modified by different tasks that are running in parallel. Thus, the access to the counters must be protected in a critical section (lines 2–6 and 7-11). Inside each critical section, we decrement the counter and spawn the dependent task (lines 5 and 10) if the counter reaches a 0 value. In this particular 2D problem, each task decrements and checks two counters (lines 4 and 9).

```
1 Task_Body(); //Task's work
2 Critical Section {
3   counter[i+1][j]--; //Dec. south neighbor counter
4   if (counter[i+1][j]==0)
5       Spawn();
6 }
7 Critical Section {
8   counter[i][j+1]--; //Dec. east neighbor counter
9   if (counter[i][j+1]==0)
10      Spawn();
11 }
```

Figure 3. Pseudo code for each task

As we mentioned in the introduction, we want to explore the possibilities that the current state-of-the-art task parallelization frameworks (OpenMP, Cilk, TBB and CnC) provide us to express this pattern. First, we will focus in the particular coding details regarding the different implementations of this problem focusing in the distinguish features for each case. Next, we will discuss the performance results.

1) OpenMP particularities: The coding of our wavefront problem using OpenMP is quite straightforward, as we see in Fig. 4. Here, we have defined the `Operation` recursive function, in which firstly we take care of the task work (line 6), and next we use the OpenMP directive "`#pragma omp critical`" (lines 8 and 15) to deal with the access and the update of the dependence counters in mutual exclusion. Then, if the corresponding counter reaches a 0 value, we recursively spawn a neighbor task using the directive"`#pragma omp task`" (lines 12 and 19). Please note that the two previously mentioned "unnamed" critical sections are considered to have the same unspecified name so they are mutually exclusive [4]. We have named OpenMP_v1 to this implementation.

Obviously, this implementation based on the `omp critical` pragma leads to a coarse grain locking approach. This would be avoided if OpenMP could support the *atomic caption* operation, i.e. the atomic modification and comparison of a variable. Although there is an `atomic` directive in OpenMP, it has a lot of constraints, and constructs like:

```
#pragma omp atomic
if(--counter==0) action();
```

```
#pragma omp atomic
ready=--counter;
if(ready==0) action();
```

are NOT valid (actually they result in compilation errors). Another way to get a finer grain locking imple-

```
1 void Operation(int i, int j)
2 {
3    int gs;
4    bool ready;
5
6    A[i][j] = foo(gs, A[i][j], A[i-1][j], A[i][j-1]);
7    if (j<n-1) {
8        #pragma omp critical{
9           --counter[i][j+1];
10          ready = counter[i][j+1]==0;}
11       if (ready){
12            #pragma omp task
13               Operation(i,j+1);}}
14   if (i<n-1){
15       #pragma omp critical{
16          --counter[i+1][j];
17           ready = counter[i+1][j]==0;}
18       if (ready){
19            #pragma omp task
20               Operation(i+1,j);}}
21 }
```

Figure 4. Coding details for the OpenMP implementation based on the critical pragma (OpenMP_v1 version)

mentation in OpenMP, consists in declaring an $n \times n$ matrix of `omp_lock_t` data types (`**locks`), and by using the OpenMP runtime functions `omp_set_lock()` and `omp_unset_lock()` to control the access and decrement of the shared counters. For instance, the `omp critical` directive of lines 8–10 in Fig. 4, could be replaced by the `omp_set_lock(&locks[i][j+1])` and `omp_unset_lock(&locks[i][j+1])` calls. We named OpenMP_v2 to this alternative implementation. Obviously, this approach is more complex (or less productive from a programmer's point of view) than the previous OpenMP_v1 version, because the programmer now has to ensure that the code is deadlock free or does not suffer from blocking. Other important limitations in this approach are the wastage of memory due to the locks matrix, as well as some time overhead due to the initialization of locks.

2) Cilk particularities: We have also incorporated Cilk into our study, due to the position of that framework as one of the primary instances of modern task-based parallelism. In particular we have used the language extensions provided by the intel Cilk Plus framework [8]. There are two main constructors from Cilk Plus, that we need in our pattern: `cilk_spawn` and `cilk_sync`. In Cilk, the `cilk_spawn` keyword before a function invocation specifies that this child function can potentially execute in parallel with the continuation of the parent (caller). The `cilk_sync` keyword precludes any code after it from executing until all previously spawned children of the parent have completed. Cilk does not support atomic captures, so to control the access and decrement of the shared counters we have to use locks.

The coding details of our 2D wavefront problem in Cilk Plus are shown in Fig. 5, where we see that this implementation is close to the OpenMP version based on locks. It differs mainly in the use of the `pthreads_lock()` functions (Cilk does not provide constructors for locks), and the `cilk_sync` statement (line 24). This statement is required because the `cilk_spawn` constructor does not wait on all the spawned tasks, a implicit feature provided in OpenMP (or TBB) by the `omp task` (or `spawn()`) constructor.

```
1 void Operation(int i,int j)
2 {
3    int gs;
4    bool ready_e, ready_s;
5
6    A[i][j] = foo(gs, A[i][j] + A[i-1][j] + A[i][j-1]));
7    if (j<m-1){
8       pthread_mutex_lock(&locks[i][j+1]);
9       counter[i][j+1]--;
10      ready_e = (counter[i][j+1]==0);
11      pthread_mutex_unlock(&locks[i][j+1]);}
12   if (ready_e)
13      cilk_spawn Operation(i, j+1);
14
15   if (i<m-1){
16      pthread_mutex_lock(&locks[i+1][j]);
17      counter[i+1][j]--;
18      ready_s = (counter[i+1][j]==0);
19      pthread_mutex_unlock(&locks[i+1][j]);}
20   if (ready_s)
21      cilk_spawn Operation(i, j+1);
22
23   if (ready_e || ready_s)
24      cilk_sync;
25 }
```

Figure 5. Coding details for the Cilk implementation (Cilk version)

3) TBB particularities: One interesting construct provided by TBB is the atomic template class. So we can declare the matrix of counters using `atomic<int>**counter`. There are several methods available for an atomic declared variable. For instance, the operation `--counter[i][j]`, atomically decrements and returns the new value of `counter[i][j]`. The expression "`if(--counter==0) action()`" is safe and just one task will execute the "`action()`". Compared with locks, atomic operations are faster and do not suffer from deadlock and convoying, what adds a distinguish feature that OpenMP or Cilk does not support at the moment. The code snippet for a task implementation of our problem using TBB and the atomic feature is shown in Fig. 6, where the atomic operations are in lines 12 and 15. We name TBB_v1 to this implementation.

Contrary to the simpler OpenMP approach that defines tasks using directives, in TBB we have to declare a class and to define an execute method for our tasks. For instance, in lines 1–7 of Fig. 6, we declare the `Operation` class that inherits from the TBB task class. Then, in lines 8–17 we define the method `execute()` which does the actual task computation. Now, as we see in lines 13 and 16, the spawn of ready-to-run neighbors tasks is performed directly by invoking the `spawn()` function.

There is a higher level programming approach to code wavefront codes in TBB [5]. The idea is to use the TBB `parallel_do_feeder` class template. This class essentially implements a work-list algorithm, in such a way that

```
1 Class Operation: public TBB::task
2 {
3   int i, j, gs;
4   public:
5     Operation(int i_ , int j_) :  i(i_), j(j_) {}
6     task * execute();
7 };
8 TBB::task * Operation::execute()
9 {
10    A[i][j] = foo(gs, A[i][j] , A[i-1][j] , A[i][j-1]);
11    if (i<n-1) //There is south neighbor
12       if (--counter[i+1][j]==0)
13          spawn( ..... Operation(i+1,j) );
14    if (j<n-1) //There is east neighbor
15       if (--counter[i][j+1]==0)
16          spawn( ..... Operation(i,j+1) );
17 }
```

Figure 6. Coding details for the TBB implementation based on atomic operations (TBB_v1 version)

```
1 // Declarations. Tag collection to control execution
2 < par ElementTag >
3 // Step prescription: for each ElementTag instance
4 // we control an step exec.
5 <ElementTag>:: (Operation)
6 // Step execution:  a step may produce a new ElementTag
7 (Compute) -> <ElementTag>
8 int Operation::execute(const par & t, wave & c ) const
9 {
10    int i = t.first;
11    int j = t.second;
12    int gs;
13
14    A[i][j] = foo(gs, A[i][j] + A[i-1][j] + A[i][j-1]);
15    if (i < n-1)
16        if (--counter[i+1][j] == 0)
17            c.ElementTag.put(par(i+1,j));
18    if (j < n-1)
19       if (--counter[i][j+1] == 0)
20            c.ElementTag.put(par(i, j+1));
21    return CnC::CNC_Success;
22 }
```

Figure 7. Coding details for the CnC implementation (CnC version)

new tasks can be added dynamically to the work-list by invoking the `parallel_do_feeder::add()` method. For instance, the `spawn()` invocations in lines 13 and 16 of Fig. 6 would be replaced by `feeder.add()` invocations. Each one of these calls will implicitly spawn a new task to execute. We call TBB_v2 to this implementation.

4) CnC particularities: Although CnC represents a framework oriented to improve the programmer productivity through the support and combination of parallel patterns [6], it also provides a runtime library based on TBB and the task programming model. This approach allows to increase the level of abstraction when coding a parallel code, because the user just focus on expressing semantic constraints when programming, and do not have to worried about expressing the parallelism or handling concurrent data structures. Thus we think of interest to study an implementation of our wavefront problem using CnC. We will name CnC to this new implementation, for which some coding details are shown in Fig. 7.

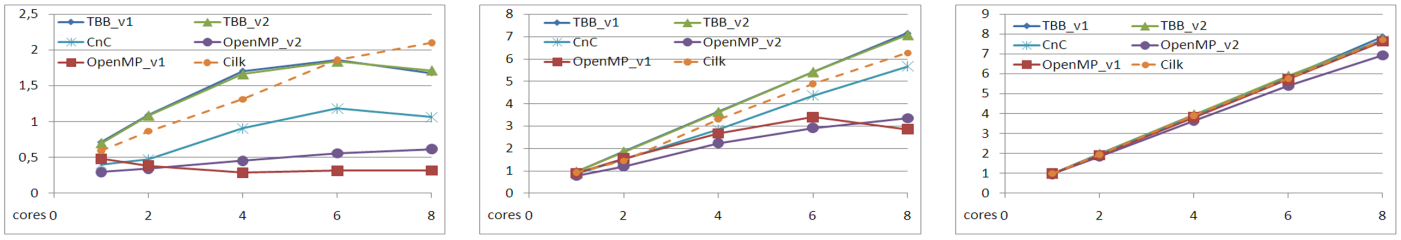
CnC provides three types of static collections: i) the computation steps which are the high-level operations, in our case the `Operation` collection, for which the `execute()` method is defined in lines 8–22; ii) the data items collections; and iii) the control tags collections that prescribe the steps, in our case the `ElementTag` collection, declared at line 2. In CnC, the collections are connected via data and control dependencies that specify the program's ordering constraints: in our example the control constrains are defined in lines 5 and 7. Regarding the `execute()` method, we see that precisely follows the scheme of Fig. 3, being the basic difference with the TBB code of Fig. 6 that instead of explicit calls to `spawn()`, we have to generate the control tags for the ready-to-run neighbors invoking the `c.ElementTag.put()` method (see lines 17 and 20).

C. Experimental evaluation

Next, we conduct several experiments to evaluate the performance of the previously described implementations (OpenMP_v1, OpenMP_v2, Cilk, TBB_v1, TBB_v2 and CnC). In particular, the library versions are Intel Open_MP v. 3.0, Cilk Plus, TBB v. 3.0 and CnC v. 0.4. For all the experiments, we have used a multicore machine with 2 quad-cores Intel(R) Xeon(R) CPU X5355 at 2.66GHz, SUSE LINUX 10.1. 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.

We evaluate two different scenarios: in the first case of study, we fixed the workload of each task to a constant size (subsec. II-C1), whereas in the second case we allowed variable task workload sizes (subsec. II-C2). Our goal is to evaluate the effect of different task granularities in both cases of study, and to find where the sources of overhead are in each implementation.

1) Constant task granularity: In this case of study, we fixed the task workload to a constant grain size in all the cells. For it, we set to a constant value the `gs` parameter of the `foo` function (line 3 in Fig. 2). That parameter sets the number of floating point operations per task. In our experiments we evaluate three task granularities: i) fine granularity case (which approximately corresponds to 200 floating point operations); ii) medium granularity case (around 2,000 floating point operations); and iii) coarse granularity case (i.e. 20,000 floating point operations). The speedups for each case are shown in Figs. 8(a), (b), and (c) respectively. For all the cases, the matrix size was 1,000 $\times$ 1,000. We should remark that sometimes we got aborted executions for the Cilk version. In fact, it was impossible to run the Cilk version for a matrix size of 1,500 $\times$ 1,500 elements (or for larger matrix sizes). This is due to limitations on the number of nested tasks that supports the current runtime implementation of Cilk Plus. It is expected that in future releases, the number of possible nested tasks



(a) Fine grain task (b) Medium grain task (c) Coarse grain task

Figure 8. Speedup results for constant task granularities. The x-axes represent the number of cores

increases, what will solve this problem. For the rest of the versions, we performed the same experiments with other matrix sizes without problems, and we obtained similar results. Let's note that in this experiment, except for the initial and last computations, eventually the workload will be evenly balanced among the threads.

One general first conclusion that we get is that task grain size heavily affects scalability, specially in the fine and medium grain cases. Moreover, the scalability results for the fine granularity case (a) are disappointing. In the fine and medium cases, the TBB versions outperform all the other implementations, except in the fine grain case with 8 cores, where Cilk tends to scale better (although let's not forget that this latter version was unable to run to completion in some executions). On the other hand, in the coarse granularity case (c) all the implementations exhibit a similar behavior. Both TBB_v1 based on explicit spawns and TBB_v2 based on the `parallel_do_feeder()` template, present similar speedups, although in the fine granularity case (a) TBB_v1 exhibits better results from 1 to 6 cores. The OpenMP_v1 implementation based on the critical pragma, is the one with the poorest performance. On the contrary, the OpenMP_v2 implementation based on locks, although also exhibits bad results in the fine task granularity case (a) for small number of cores, it keeps scaling when the number of cores increases. On the other hand, the CnC implementation keeps between the TBB and the OpenMP ones.

In order to understand where the sources of overhead for each implementation are, we decided to profile our codes by employing the *call-graph activity* of Vtune [9]. This tool works by analyzing the entry and exit points of all user and library functions. The call-graph activity provides important information about all the functions analyzed: self time, waiting time, number of calls, etc. We focus our discussion in the results of the versions that ran to completion, in particular in the fine granularity case (a). These results are shown in Figs. 9 and 10.

For the OpenMP results shown in Fig. 9, `task` accounts for the contribution of the `omp_task()` and `omp_task_alloc()` internal functions which are associated to the `omp task` directive, `critical` accounts for the `omp_critical()` and `omp_end_critical()`

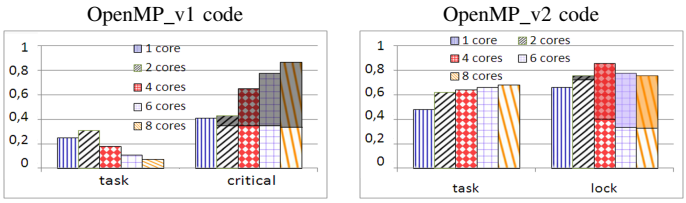


Figure 9. The most time consuming library functions in OpenMP implementations for the fine granularity case (a). The y-axes present the ratio between the self time of each function and the total execution time for 1, 2, 4, 6 and 8 cores

internal functions that are associated to the `omp critical` directive in the OpenMP_v1 code (see Fig. 4), whereas `lock` accounts for the `omp_set_lock()` and `omp_unset_lock()` functions, that are directly invoked by the user in the OpenMP_v2 code to control the access to the shared counters. From these results we see that the overhead of the task creation in OpenMP is important (around 30% of the execution time). However, the main source of inefficiency is due to the critical and lock functions. In the figures for the OpenMP_v1 and OpenMP_v2 codes, we have shadowed in the upper side of the `critical` and `lock` bars, the contribution due to the waiting time in the corresponding functions. Precisely, this waiting time represents the contention among the threads to get a lock. In particular, the contention to capture the global lock used by the internal functions of critical in the OpenMP_v1 code is the main contributing factor of overhead. This contention significantly increases with the number of cores, and the resulting waiting time it accounts for near the 60% of the execution time on 8 cores. This explains (added to the task creation overhead) the poor scalability results that we saw for the OpenMP_v1 code in Fig. 8-case (a). Regarding the waiting time due to the explicit `omp_lock()` calls in the OpenMP_v2 code, we see again that there is contention, although to a lesser extent. For instance, the waiting time is around 20% on 8 cores. Although with the approach of explicit management of locks, the overhead due to contention is now much smaller than the overhead measured for the critical section implementation, still we see that this waiting time increases with the number of cores.

From the TBB results shown in Fig. 10, we see that

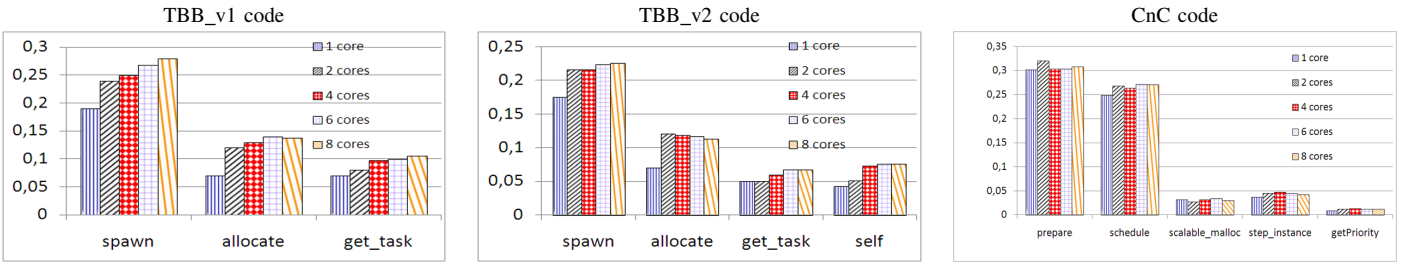


Figure 10. The most time consuming library functions in TBB and CnC implementations for the fine granularity case (a). The y-axes present the ratio between the self time of each function and the total execution time for 1, 2, 4, 6 and 8 cores

the functions that more time consume are: `spawn()` (this method is invoked each time a new task is spawned), `allocate()` (it selects the best memory allocation mechanism available each time a task is created) and `get_task()` (this method is called by the internal scheduler after completing the execution of a task). The overheads due to these functions, are very similar in the TBB_v1 and TBB_v2 implementations. However in TBB_v2 there is an additional internal function, `self()` (that is invoked by the `parallel_do_feeder()` template) whose contribution is noticeable, what explains the slight difference in performance that we saw in Fig. 8-case (a) for the TBB codes. In any case, we should remark that the main source of overhead is due to the `spawn()` method, and that it grows when the number of cores increases. This is more noticeable in the TBB_v1 code, where `spawn()` may consume near the 28% of the execution time for 8 cores. For other task granularities, we also noticed that `spawn()` is the most contributing factor to the overhead, and that similarly its contribution increases with the number of cores.

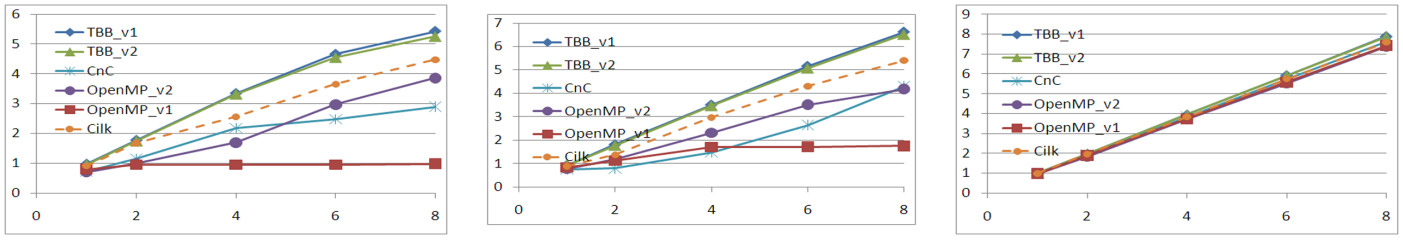
From the CnC results in Fig. 10, we see that the more time consuming functions are `prepare()`, `schedule()`, `step_instance()`, `scallable_malloc()` and `get_priority()`. All of them are internal helper functions called by the `c.ElementTag.put()` method (see Fig. 7), invoked in our program by the user to add a new tag in order to prescribe the computational steps. The most time contributing functions are `prepare()` and `schedule()` which account for more than a 50% of execution time, what explain the mediocre performance we saw in Fig. 8-case (a) for the CnC code. Interestingly, around the 50% of the time of `schedule()` is due to TBB overhead (`spawn()`, `allocate()`, ...). Therefore, the rest of overhead is due to the abstraction penalty introduced by the helper functions. We also noticed that the overhead due to the helper functions tends to decrease for coarser task granularities, what explain the better scalability we observed for the medium and coarse granularity cases in the CnC codes.

2) Variable grain size: In the next case of study, we changed the task workload on each cell, to a variable grain size. For it, we assigned variable values to the `gs` parameter of the `foo` function (line 3 in Fig. 2). Now, in this scenario, it could be some load imbalance among the threads, so we would like to study how this new factor affects the performance of our implementations, as well as to identify where the sources of overhead are.

We evaluated three grain ranges: i) a variable fine granularity case, for which `gs`=$[200..800]$ FLOP; ii) a variable medium granularity case for which `gs`=$[800..2,000]$ FLOP; and iii) a variable coarse granularity case (now `gs`=$[2,000..20,000]$ FLOP). The speedups for these cases are shown in Fig.11(a), (b) and (c) respectively. Again, for all the cases the matrix input size was 1,000 $\times$ 1,000. Similar results were obtained for other matrix sizes. As mentioned before, sometimes we got aborted executions for the Cilk version.

From the results in Fig. 11, we see that, in spite of the variable task granularity, the behavior of each implementation is consistent with the behavior for the constant task granularity. In any case, again the task grain size is the main factor that affects scalability. In fact, the results that we obtain for the variable fine granularity case (a) are better to what we obtained for the constant fine granularity case (Fig. 8(a)). The reason is that now, the average workload of a task is coarser than in the constant case, therefore the overheads will have less weight.

Again, we used the Vtune call-graph activity to profile the most time consuming functions for each implementation that ran to completion, getting similar results to those discussed in the previous subsection. We could mention that the momentary load imbalance is not a factor affecting the performance, because it is successfully managed by the task-schedulers implemented in the libraries. In particular, Cilk, TBB (CnC) and the Intel OpenMP runtimes manage task scheduling through the work-stealing strategy, that has proved to be an effective mechanism for balancing the load in our wavefront codes. In particular, we studied the overheads due to the work stealing functions, finding that they always represented less than a 1% of execution time.



(a) Fine grain task range (b) Medium grain task range (c) Coarse grain task range

Figure 11. Speedup results for variable task granularities. The x-axes represent the number of cores

III. Further Optimizations

A. Atomic Capture

One way to overcome the source of inefficiency due to the lack of atomic capture in the OpenMP 3.0 framework, could be by directly using compare-and-swap (CAS) instructions. We show in Fig. 12 a new OpenMP version based on this optimization, where function `CompareAtomic()` (lines 3-6) is now responsible of performing the atomic accesses to the counters matrix (lines 11 and 17).

We evaluate this new OpenMP version (named OpenMP_v1.2) following the same methodology explained in the previous section. The results for the constant fine task granularity case (200 FLOP) are shown in Fig. 13, where we keep the OpenMP_v1 and OpenMP_v2 (OpenMP versions based on critical and locks, respectively), as well as the TBB_v1 (TBB version with the atomic template class) for comparison. As we can see, clearly the synchronization based on CAS instructions mechanism significantly reduces the contention problem in the OpenMP codes, making the OpenMP optimized version the best one in 6 and 8 cores. For the medium grain size, the TBB_v1 code exhibits slightly better performance than the optimized OpenMP one, whereas for the coarse grain size, both OpenMP_v1.2 and TBB_v1 codes behave similarly.

We used the Vtune event-based sampling tool for collecting the *Locked Operations Impact* ratio [9] in order to measure the percentage of cycles spent by locks (or atomic) operations on each code. We found that, in the fine grain case, the contention due to atomic operations arose to near the 30% of the CPU cycles for 6 and 8 cores in the TBB_v1 code, whereas in the OpenMP_v1.2 code that impact ratio was around 20%, what explains the better performance in the OpenMP optimized code. However, in the medium grain case, the ratio was always below 5% for the TBB_v1 version, whereas it was higher than 6% for the OpenMP_v1.2 one, explaining the slight differences in scalability. In the coarse grain case, the ratios were negligible for both versions. In any case, these results demonstrate us that the atomic capture is an important feature that should be available as a high level construct. In fact, this atomic capture facility will be implemented in the next version of OpenMP (v. 3.1).

```
1  void Operation(int i,int j)
2  {
3     int CompareAtomic(int i, int j){
4        int x = __sync_sub_and_fetch((volatile int *)&
               counter[i][j], 1);
5        return x==0;
6     }
7     int gs;
8
9     A[i][j] = foo(gs, A[i][j] + A[i-1][j] + A[i][j-1]));
10    if (j<m-1) {
11      if (CompareAtomic(i,j+1)) {
12           #pragma omp task {
13                 Operation(i,j+1);}
14      }
15    }
16    if (i<m-1) {
17      if (CompareAtomic(i,j+1)) {
18           #pragma omp task {
19                 Operation(i+1,j);}
20      }
21    }
22 }
```

Figure 12. Coding details for the OpenMP implementation based on CAS instructions (OpenMP_v1.2 version)

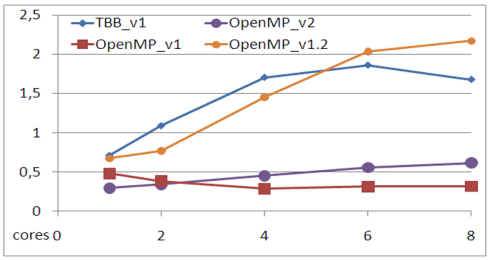


Figure 13. Results of the optimized version of OpenMP based on CAS operations (OpenMP_v1.2), for the constant fine task granularity case

B. Guiding the scheduler

While it might be difficult to reduce synchronization overheads caused by the atomic operations within the OpenMP or TBB runtime library, unless specialized synchronization hardware becomes available, nevertheless the results of the previous section have shown us that library developers of both OpenMP and TBB should try to reduce the other sources of overhead: the cost of task creation and task management methods, which is significant if task granularity is not sufficiently coarse.

One way the programmer can go to reduce the cost of task

creation and task management consist in the use of explicit task passing (or *task recycling*) mechanism [5], which is available in TBB. In our wavefront pattern, each task has the opportunity to spawn two new tasks (east and south neighbors). We can avoid the spawn of one of them by returning a pointer to the next task, so instead of spawning a new task, the current task recycles into the new one. This way, we achieve two goals: reducing the number of calls to `spawn()`, as well as to save the time for geting new tasks from the local queue (reducing the number of calls to `get_task()`). In OpenMP or Cilk codes, this recycling mechanism could be somehow emulated by recursively calling the task body of a ready to dispatch new task, avoiding the spawning. However, note this emulation strategy could exhaust the stack when the number of nested tasks is too large, a problem that the TBB recycling mechanism avoid. In addition, when recycling we can provide hints to the scheduler about how to prioritize the execution of tasks to guarantee a cache-conscious traversal of the data structure, what might help to improve data locality.

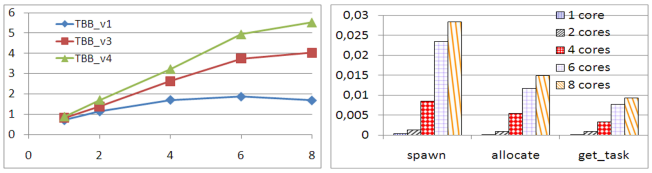
In Fig. 14 we see a code snippet with the optimized TBB implementation. In line 5, we set the flag `recycle_into_east` if there is a ready to dispatch task to the east of the executing task. Otherwise, we set the flag `recycle_into_south`, in line 11, if the south task is ready to dispatch. Later, according to these flags, we recycle the current task into the east, line 16, or south tasks, line 20. Note that, since in this example the data structure is stored by rows, if both east and south tasks are ready, the data 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. In any case, the number of spawns in this version is reduced from $n \times n - 2n$ (the number of spawns in TBB_v1 and TBB_v2) to $n - 2$ (approximately the size of a column).

In order to properly evaluate the impact of each mechanism (recycling and locality), we have studied the performance of two versions: TBB_v3, a version that prioritizes the south task instead of the east one. Its goal is to isolate the advantages of recycling (without exploiting cache); And TBB_v4, the version that implements the algorithm described in Fig 14 (recycling and locality). In Fig. 15(a) we can see the speedups for the TBB_v1 (shown here as a baseline), TBB_v3 and TBB_v4 implementations for the case of constant fine task granularity (`gs`=200 FLOP).

It is clear that TBB_v4 is the best solution. In fact, we measured speedups for other fine grain sizes finding that the finer the granularity, the better the improvement. Besides, it is interesting to see that a great deal of the improvement contribution is due to the recycling optimization, pointed out by the TBB_v3 enhancement over the TBB_v1 version.

```
1 Task_Body();    // Task's work
2 if (j<n-1){    // There is east neighbor
3 {
4   if (--counters[i][j+1]==0)
5     recycle_into_east = true;
6 }
7 if (i<n-1){   // There is south neighbor
8 {
9   if (--counters[i+1][j]==0)
10        if (!recycle_into_east){
11          recycle_into_south = true;
12        else
13          spawn(i+1,j));
14 }
15 if (recycle_into_east){ //Recycle this into east
16        recycle_as_child_of();
17        j = j+1;
18        return this;
19 }else if(recycle_into_south){ //Recycle this into south
20        recycle_as_child_of();
21        i=i+1;
22        return this;
23 }else
24        return NULL; // There is no neighbor task ready
```

Figure 14. Coding details for the optimized TBB implementation: recycling and locality (TBB_v4 version)



(a) Speedups. The x-axis represents the number of cores

(b) Ratio for library functions in TBB_v3 and TBB_v4

Figure 15. Results of the optimized versions of TBB in the constant fine task granularity case

To go deeper into the comparison of these versions and better understand the sources of improvement, we have used again the call-graph activity of Vtune to collect information of the most time consuming functions in the optimized implementations. In Fig. 15(b) we show the ratio of time consumed by the task creation and management functions (`spawn()`, `allocate()`, and `get_task`) relative to the total execution time. The profiling times for these functions are the same for TBB_v3 and TBB_v4. In that figure we see that the percentage of time wasted by these functions is less than 6% of execution time. In fact we see that, for each internal function there is a reduction of around one order of magnitude in the recycling versions when compared with the TBB_v1 version (see Fig. 10). For instance, one of the reasons of this reduction is that the number of calls to `spawn()` in TBB_v1 is 998,000 (clearly one call per matrix element, but for the last column and row), while this number is of 998 calls in TBB_v3 and TBB_v4.

On the other hand, thanks to the Vtune event-based sampling tool, we have collected the L1D and L2 miss ratios for our TBB_v3 and TBB_v4 implementations, and for the fine granularity case again. The ratios are shown in Fig. 16, that confirms that cache performance is much

worse in TBB_v3 than in TBB_v4. We have measured the miss ratios with other granularities, and we have noticed that improvement of the cache-conscious version is larger when we move from coarse grain to fine grain and is slightly better appreciated in L1D than in L2.

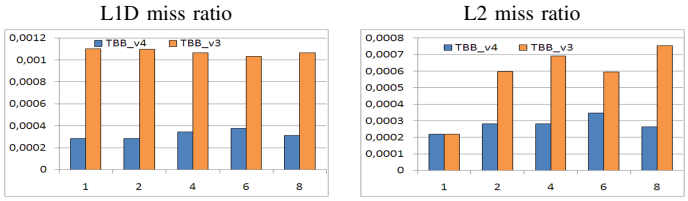


Figure 16. L1D and L2 miss ratio for TBB_v3 and TBB_v4 for the fine granularity case in 1, 2, 4, 6 and 8 cores

C. Tiling: increasing the granularity

As we have seen, the performance of the wavefront algorithms consistently decreases as the task workload grain gets finer. The tiling technique is a well known solution to get coarser grain size and can also be applied to wavefront problems. The idea is to assign an equal number of adjacent columns and rows to each processor (square tiles). This technique achieves two goals: to reduce the number of tasks (and therefore, the number of spawns); and to save some overhead of the wavefront bookkeeping (the memory space and the initialization time of the counter/dependence matrix, which is now smaller due to it needs a counter per block - tile, not per matrix element). Another possibility could be to select rectangular tiles (instead of square tiles) to better take advantage of cache. A rectangular tile with more columns than rows could lead to a smaller number of cache misses if prefetching strategies are available in the cache architecture.

We implemented square tiling in the TBB_v4 version, and measured the speedups for different block sizes (BS) in the fine grain case. We have left the evaluation of rectangular tiles for a future work. In our experiment, we found an increment of speedup for 8 cores of around a 9% when $BS = 10$, which was the optimal block size. For larger block sizes, speedups start to diminish again, because the degree of parallelism is lower when we increase the size of the block while keeping a constant problem input size (the matrix size). Basically, if we have a larger block size, there are less independent blocks to process concurrently. We confirmed this result using profiling information about the number of calls to the `sched_yield()` TBB internal function, which is called by the scheduler when there is not enough parallel work, finding an important increment in the number of calls when BS is higher than 10.

IV. Related works

There have been several research works that have targeted the problem of parallelizing wavefront problems, well on distributed memory architectures [10], or on heterogeneous architectures such as SIMD-enabled accelerators [11], or the Cell/BE architecture [12]. In all 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"[13], being the wavefront one of these patterns [2]. In this research line, there have been a recent proposal [14] 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 threads and rely on the O.S. scheduler to process the work. On the contrary, our study focus on much more fine task granularity regular and irregular problems, as well as in the study of the programming support and evaluation of the performance that frameworks based on task-schedulers provide.

V. Conclusions

Through our study of different task-based implementations of a wavefront pattern, we have found that TBB provides some distinguishing features that allow the more efficient implementations. Features such as the atomic capture, and the task recycling mechanism, coupled with prioritization of task to exploit data locality, are demonstrated to lead to important performance improvements, specially when the granularity of the task is fine. We believe they should be available as user level constructors (as TBB does) to allow high level optimizations guided by the programmer.

In any case, these optimizations could be wrapped in a higher level template to facilitate the coding of wavefront codes to less experienced users. As a future work, we plan to implement a wavefront template for the TBB library (as the `parallel_do` or the `pipeline` ones) that encapsulates the mentioned optimizations and allow to the developer to express the wavefront dependencies and the data locality hints without worrying about atomic counters or task recycling functionalities.

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] *Wavefront Pattern*, University of Illinois at Urbana-Champaign. College of Engineering Department of Computer Science. [Online]. Available: http://www.cs.uiuc.edu/homes/snir/PPP/patterns/wavefront.pdf

[4] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[5] J. Reinders, *Intel Threading Building Blocks*. O'Reilly, 2007. [Online]. Available: http://www.threadingbuildingblocks.org/

[6] *Intel Concurrent Collections for C/C++*, Intel Corp. [Online]. Available: http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc

[7] P. Kambadur, A. Gupta, A. Ghoting, H. Avron, and A. Lumsdaine, "Pfunc: Modern task parallelism for modern high performance computing," in *SC'09: Conference on High Performance Computing, Networking, Storage and Analysis*. New York, USA: ACM, 2009, pp. 1–11.

[8] *Intel Cilk++ SDK*, Intel Corp. [Online]. Available: http://software.intel.com/en-us/articles/intel-cilk

[9] J. Reinders, *VTune Performance Analyzer Essentials*. Intel Press, 2005.

[10] E. C. Lewis and L. Snyder, "Pipelining wavefront computations: Experiences and performance," in *HIPS'99: IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 1999.

[11] O. Storaasli and D. Strenski, "Exploring accelerating science applications with FPGAs," in *Proc. of the Reconfigurable Systems Summer Institute*, July 2007.

[12] 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: Conference on Computing Frontiers*. New York, NY, USA: ACM, 2008, pp. 13–22.

[13] 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.

[14] 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: ACM International symposium on High performance distributed computing*. New York, NY, USA: ACM, 2009, pp. 1–10.