Evaluation of the Task Programming Model in the Parallelization of Wavefront Problems

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*

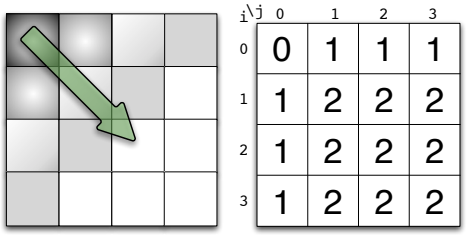
***Abstract*—This paper analyzes the applicability of the task programming model in the parallelization of generic wavefront problems. Computations on this type of problems are characterized by a data dependency pattern across a data space, which can produce a variable number of independent tasks through the traversal of such space. Precisely, we think that it is better to formulate the parallelization of this wavefront-based programs in terms of logical tasks, instead of threads for several reasons: more efficient matching of computations to available resources, faster start-up and creation task times, improved load balancing and higher level thinking. To implement the parallel wavefront we have used two state-of-the art task libraries: TBB and OpenMP 3.0. In this work, we highlight the differences between both implementations, from a programmer standpoint and from the performance point of view. For it, we conduct several experiments to identify the factors that can limit the performance on each case. Besides, we present in the paper a wavefront template based on tasks, template that makes easier the coding of parallel wavefront codes. We have validated this template with three real dynamic programming algorithms, finding that the TBB-coded template always outperforms the OpenMP based-one.**

***Keywords*-task programming; wavefront pattern; performance analysis;**

I. Introduction

Wavefront is a programming paradigm that appears in important scientific applications such as those based in dynamic programming [9] or sequence alignment [1] (Floyd algorithm [11], biological sequence, etc). In this paradigm, data elements are laid out as multidimensional grids representing a logical plane or space [10]. Typically, on each iteration over the grid, data elements have dependencies among them. It is well known that wavefront computation admit efficient, parallel implementation via pipelining [9].

Fig. 1(a) represents an example of a 2D wavefront, where the updating of a matrix element requires the updating of previous neighbor elements resulting in a computation that resembles a diagonal sweep across the elements in the logical plane. The computations start at a corner of the matrix and a sweep would move along the diagonal to the next corner. That diagonal represents the number of computations or elements that could be executed in parallel without dependencies among them. This number of independent elements will grow gradually to the maximum amplitude of the diagonal. Then it will decrease to end in the opposite corner of the grid. This diagonal sweep is the reason for the name wavefront.



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

Figure 1. Typical wavefront traversal, dependencies and a matrix of counters

The aim for this paper is twofold: first we want to validate the feasibility of the task programming model in order to implement parallel wavefront problems; and second, to provide a wavefront template to assist the programmer in the implementation of such codes without dealing directly with task spawns and low level parallel programming details. We think tasks based programming fits better than thread based programming on the implementation of parallel wavefront problems due to some different reasons:

1) Tasks are much lighter weight than logical threads, in fact the creation/termination of a task can be some orders of magnitude inferior that creating or terminating a thread at the OS level. We have noticed that in our wavefront applications, the computation over one element (our task) may suppose around 100-1,000 floating point operations, so we consider they represent a fine grain workload, best suited for a task model [13].
2) In task-based programming, the tasks scheduler does have some higher level information, and so can sacrifice fairness for efficiency: a thread scheduler typically distributes time slices in a round-robin fashion. However, when creating the threads, the programmer has to set the appropriate number of logical threads to avoid "under-subscription" (to have less logical threads than physical hardware threads) or "over-subscription" (to have more logical threads than physical hardware threads) leading both situations to some overheads. The sources of over-subscription overheads are cache cooling, context switching and lock preemption. A

task scheduler avoids under/over subscription by selecting the same number of logical threads as the number of physical hardware threads and then by allowing to map tasks in resulting logical threads. Specially, in presence of load unbalance, which is a scenario that we expect in wavefront codes, a task scheduler can outperform the default thread scheduler provided by the OS [13].

There are two main libraries that represent the state-of-the-art when programming using tasks: *Intel Threading Building Blocks* (TBB) [13] and *OpenMP 3.0* [4]. TBB offers a rich and complete approach to express parallelism in a C++ program. TBB has some advantages: i) targets threading for performance; ii) is compatible with other threading packages; and iii) relies on generic programming. One of the big qualities of TBB is that provide some interesting high level predefined templates to help programmers to express the program in parallel using very high level constructors so that an user who is not an expert in a parallel language is able to parallelize a program in a simple way. For example, we can write a program based on the pipeline model adding stages (filters) with just a few lines of code. However, a high level wavefront template, which is one of the goals of this work, is lacking.

On the other hand, in the last decade, OpenMP has emerged as "the facto" standard for shared-memory parallel applications written in C/C++ or Fortran. OpenMP is based on an explicit programming model in which parallelism is specified through the use of compiler directives which are embedded in the source code. In OpenMP, when a thread encounters a task construct in a directive, then a task is generated for the code of the associated structured block. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply [4]. The encountering thread may immediately execute the task, or defer its execution. In the latter case, the task can be assigned to any thread in the team.

The goal of this paper is to explore the programmability and performance of a parallel task-based 2D wavefront problem. Firstly, in section II we describe a simple wavefront code and discuss its parallel task-based implementation using TBB and OpenMP, highlighting the differences between both implementations from a programmer point of view. In order to increase the abstraction level and simplify the parallel implementation of wavefront codes, in section III we present a wavefront template which, internally, can be built on top of TBB or OpenMP. Next, in section IV we conduct several experiments to identify the factors that can limit the performance on TBB and OpenMP implementations. In addition, we parallelize three real wavefront codes using the template, to validate its feasibility. In section V we present some related works. Finally, in section VI we summarize the conclusions.

II. The problem

In order to compare the TBB and OpenMP implementations of a wavefront problem, we have first selected a simple 2D wavefront problem. It is a classic problem consisting in calculating a function for each cell of a $n \times n$ 2D grid [1]. However, this operation have a data dependence with two elements of the adjacent cells as we show in Fig. 1(b), in which the $(1,1)$ cell depends on the north, $(0,1)$, and west, $(1,0)$, ones. Fig. 2 shows the code snippet for that problem: A[i,j] depends on A[i-1,j] and A[i,j-1]) (line 4).

```
1 for (i=1; i<n; i++)
2   for (j=1; j<n; j++)
3     for (k=0; k<index; k++)
4       A[i,j] = f(A[i,j], A[i-1,j], A[i,j-1]);
```

Figure 2. Code snippet for a 2D wavefront problem

The basic unit of work is a task which will take care of the computation performed at one (i,j) cell of the matrix. Thus, the intention is to parallelize the i and j loops. 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 k-loop. In fact, as we see in figure 2, the upper bound of this k-loop depends on the *index* variable, that we can set in our experiments to control the computational load for each cell. With this, we can define the granularity of the tasks and study the influence of this factor on performance. Similarly, we can set this upper bound to simulate a constant or a variable task load and therefore to analyze the impact of an homogeneous work distribution or, on the contrary, a highly load unbalance scenario.

A. Implementation

In Fig. 1(b) we can see the data dependence flow (the arrows) for this 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 have in the same figure. The value of the counters points out to how many tasks you have to wait for. That way, only task with the corresponding counter nullified can be dispatched.

Summarizing, each ready task can proceed to execute 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. The pseudo code of this procedure is shown in Fig. 3. The `Task_Body()` function (line 1) 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 by mutual exclusion, for instance inside a critical section (lines 4–8

and 11-15). Inside each critical section, we decrement the counter and spawn the dependent task (lines 7 and 14) if the counter reaches a 0 value.

```
1 Task_Body(); //Task's work
2
3 Critical Section
4 {
5   counter[i+1][j]--; //Dec. south neighbor counter
6   if (counter[i+1][j]==0)
7      Spawn();
8 }
9
10 Critical Section
11 {
12   counter[i][j+1]--; //Dec. east neighbor counter
13   if (counter[i][j+1]==0)
14      Spawn();
15 }
```

Figure 3. Pseudo code for each task

In the next subsections we will focus in the particular coding details regarding the TBB and OpenMP implementation of this problem.

B. TBB particularities

One interesting construct provided by TBB is the atomic template class. So we can declare the array of counters using `atomic<int>**counter`. There are several methods that will be carried out atomically for an atomic declared variable, for example, variable increment and decrement operations. For instance, the operation `--counter[i][j]`, atomically decrement and returns the new value of `counter[i][j]`. The expression "`if(--x==0) action()`" is safe and just one task will execute the "`action()`" [1]. Compared with locks, atomic operations are faster and do not suffer from deadlock and convoying. The code snippet for a task implementation using TBB is shown in Fig. 4 where the atomic operations are in lines 14 and 17.

In more detail, in lines 1–7 of Fig. 4, we declare the `Operation` class that inherits from the TBB task class. Then, following the TBB task based programming rules, it is necessary to define the method `execute` to override the virtual method `task::execute` (lines 9–20). This methods does the actual task computation. This method precisely follows the scheme of the pseudo-code provided in Fig. 3.

C. OpenMP particularities

From the programmer point of view, the first difference between TBB and OpenMP relies in that OpenMP does not provided an atomic type comparable to the one available in TBB. Although there is an `atomic` directive in OpenMP, it has a lot of constraints, and constructs like:

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

```
1 Class Operation: public TBB::task
2 {
3   int i, j;
4   public:
5     Operation(int i_ , int j_) :  i(i_), j(j_) {}
6     task * execute();
7 };
8
9 TBB::task * Operation::execute()
10 {
11    for (int k=0; k<index; k++)  //Task's work
12       A[i][j] = f(A[i][j] , A[i-1][j] , A[i][j-1]);
13    if (i<n-1) //There is south neighbor
14       if (--counter[i+1][j]==0)
15          spawn( * new(parent()->
                allocate_additional_child_of(* parent()))
                Operation(i+1,j) );
16    if (j<n-1) //There is east neighbor
17       if (--counter[i][j+1]==0)
18          spawn( * new(parent()->
                allocate_additional_child_of(* parent()))
                Operation(i,j+1) );
19    return NULL;
20 }
```

Figure 4. C++ implementation details using TBB

or

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

are NOT valid (actually they result in compilation errors). That way, we have relayed in critical sections to deal with the mutual exclusion. Thus, the OpenMP directive "`#pragma omp critical`" has been used.

The second difference is that, contrary to what we did in TBB to define the task function, in OpenMP we just need to use the "`#pragma omp task`" directive in a recursive way, as we see in Fig. 5. So now, instead of the `Operation` class, we have the `Operation` recursive function. First, it takes care of the work (lines 5–6) and then spawn a neighbor task (lines 15-16 and 26-27) if the corresponding counter reaches a 0 value (lines 9-13 and 20-24). Please note that the two previously mentioned "unnamed" critical sections are considered to have the same unspecified name so they are mutually exclusive: "a thread waits at the beginning of a critical region until no thread is executing a critical region with the same name" [4]. Obviously, this lead to a coarse grain locking approach, that would be avoided by declaring a matrix of `omp_lock_t` data types, and by using the OpenMP runtime functions `omp_set_lock` and `omp_unset_lock` to get a finer grain locking implementation. However, this approach just emulates a Pthread programming model and diverges from our goal of providing a high productive programming environment and of rising the language abstraction level.

III. Wavefront template

For non expert parallel programmers, it may be difficult to implement a parallel wavefront algorithm, taking care of task creation and synchronization. To alleviate this difficulties,

```
1 void Operation(int i, int j)
2 {
3    int k;
4    bool ready;
5    for (int k=0;k<index;k++)
6       A[i][j] = f(A[i][j], A[i-1][j], A[i][j-1]);
7
8    if (j<n-1) {
9 #pragma omp critical
10       {
11          --counter[i][j+1];
12          ready = counter[i][j+1]==0;
13       }
14       if (ready){
15 #pragma omp task
16             Operation(i,j+1);
17       }
18    }
19    if (i<n-1){
20 #pragma omp critical
21       {
22          --counter[i+1][j];
23           ready = counter[i+1][j]==0;
24       }
25       if (ready){
26 #pragma omp task
27              Operation(i+1,j);
28       }
29    }
30 }
```

Figure 5. C++ implementation details using OpenMP

we propose a high level template in which the programmer only has to provide the dependence pattern and the actual task computation.

In order to do that we provide a new `Wave` class, that contains a list data structure to hold the dependence information, and a `WaveTask` class that has the virtual method `ExecuteTask`. This method has to be defined by the programmer in order to code the actual computation of each task. Without loss of generality, the programmer needs to identify each task with a unique `id` identifier, which can be used to parametrize the computation that each task has to perform. In our example, task are identified with an `id` in the range $[0..(n \times n-1)]$ traversing the 2D grid in a row order. The described classes, `Wave` and `WaveTask`, have been build on top of TBB and OpenMP. Experimental results with both implementations of the template are provided in the next section.

An example of the main function using our template is shown in Fig. 6. In line 2, we create the Wave object, `control`, which will store the dependence relations for all the tasks. Then, we create the first task, `initial`, in line 5, which does not depend on any other task and will trigger the wavefront execution. Now, we have to define the dependence information for this task. This is done with the method `AddDep`, which relates the current id with the dependent task id's (the east one in line 7, and the south one in line 8). The lines 10–20 in Fig. 6 are just the same lines that a programmer would write to define a dependence matrix like the one in Fig. 1(b), but instead of initializing a dependence counter, line 12 creates a new id, whereas lines 15–16 should be used to establish the corresponding dependencies. Finally, the method `run()`, line 21, should be instantiated in order to start the wavefront execution. The internal machinery will take care of dispatching the necessary tasks when appropriate.

```
1 int main() {
2   Wave<Operation>  * control = new Wave<Operation>;
3
4   int id=0; //Id of the first task
5   Operation * initial = new Operation(control, id);
6
7   control->AddDep(id, id+1); // add j+1
8   control->AddDep(id, id+n);  // add i+1
9
10  for (int i = 0; i<n; i++){
11    for (int j=0; j<n; j++){
12       id++;  //New id
13       if (i!=0 || j!=0){
14         .... // create the list of task with dependence
                 information
15         control->AddDep(id, id+1); // add j+1
16         control->AddDep(id, id+n)  // add i+1
17         ....
18       }
19    }
20  }
21  control->run(); // execute the wavefront code
22 }
```

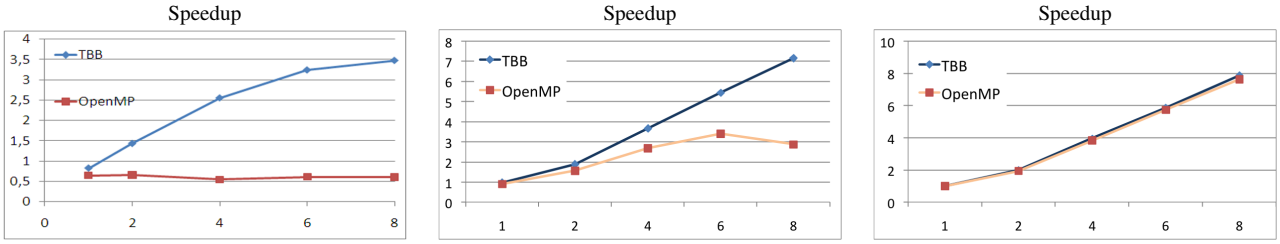
Figure 6. Wavefront dependence structure initialization and execution

IV. Experimental Results

We conduct two set of experiments to evaluate the suitability of the task programming model in the parallelization of wavefront problems. In all the 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 OS in the target platform. The codes were compiled with icc 11.1 -O3. The TBB version is 2.1. We executed each code among 3 to 10 times and computed the average execution time of the runs to get the times that we present in the next sections. Typically we performed the experiments using different numbers of cores (1, 2, 4, 6 and 8), in order to see how codes scale as we increase the number of cores. When computing speedups, they are calculated respect to the sequential code.

A. Simple problem case study: comparison of TBB vs. OpenMP implementations

1) Load balanced workload: We conduct a first set of experiments using the simple wavefront problem shown in Fig. 2 as case of study. We aim to compare the performance of the TBB vs. OpenMP task based implementations of this problem (described in Fig. 4 and Fig. 5, respectively). In these experiments, we vary the problem input size (i.e. the size of the 2D matrix) and the granularity of a task (i.e. the amount of computational load in a task). To control the computational workload of a task we just vary the number of iterations of the inner k-loop by setting the $index$ (the upper bound) value. We evaluate three task



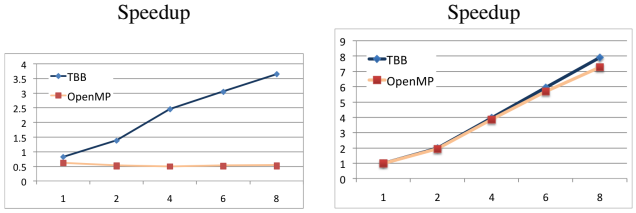
Case a: $index$=200 Case b: $index$=1,000 Case c: $index$=10,000

Figure 7. Constant task workloads: results for the TBB and OpenMP codes, with input matrix size M=1,000 x 1,000 and different task grain ($index$)

granularities: $index$=200 (which approximately corresponds to 400 floating point operations and that we define as fine granularity task size), $index$=1,000 (which is around 2,000 floating point operations or medium granularity task size), and $index$=10,000 (which corresponds to 20,000 floating point operations or coarse granularity task size). In any case, as $index$ remains constant for all the tasks, the computational load of each task is the same. For this reason, except for the initial and last computations, eventually the workload will be evenly balanced among the threads. The speedups for the experiments with an input matrix size of 1,000 x 1,000 elements are presented in Fig. 7. Similar speedups were obtained for other input matrix sizes.

The first conclusion that we can extract from these results is that the granularity of a task (and not the problem input size) is a very important factor that will determine the scalability of the code. Other result is that the TBB implementation consistently outperforms the OpenMP one, for different input sizes of the problem and for different grain size of the task workload. We notice that the performance in OpenMP tends to degrade faster when the number of cores increases, specially when the task grain size is fine or medium (Cases a, b). However, when the task granularity is coarse both implementations exhibit similar behavior (Case c).

2) Load unbalanced workload: Next, we conduct another set of experiment to study the effect of an unbalanced workload. For it, we allow that different cells in our matrix (in other words different tasks) have different grain size load. Our goal is to probe if the task stealing distinguish feature of the TBB scheduler can give some advantage over the task tied scheduler of OpenMP [7]. Again, in these experiments we vary the input problem size and the granularity of a task. Now, we select two task granularities ranges: $index$=$[100..100 + i * j/(m * 10)]$ and $index$=$[1,000..1,000 + i * j * 10/m]$, where m is the size of a matrix row, and i and j are the indices of the external loops (see the code sample in Fig. 2). In these experiments, the computational load of each task will be variable, and the workload could be quite unbalanced among the threads. The speedups for the experiments with an input matrix size of 3,000 x 3,000 are presented in Fig. 8. Similar speedups were obtained for other input matrix sizes.



Case A: $index$=[100.. 400] Case B: $index$=[1,000 .. 31,000]

Figure 8. Variable task workloads: results for the TBB and OpenMP codes, with input matrix size M=3,000 x 3,000 and different task grain ($index$)

From the results in this set of experiments with unbalanced workload, we conclude that the performance of TBB is always above OpenMP for every matrix input size and every task grain load, although the difference is small for coarse task grain (Case B). OpenMP fits badly in problems with fine or medium task grain (Case A). One interesting result that we notice is that in TBB there is a reduction of the profit from 6 cores onwards when the grain size of the task is fine (Case A). The same happens for the Case a in Fig. 7.

3) Sources of overhead: In order to understand the sources of overhead, we planned to conduct a new set of experiments. We employed Vtune (the performance analyzer of Intel [12]) to analyze the extreme situations: Cases a and c for the balanced workload and Cases A and B for the unbalanced experiments.

We decided to employ the *call-graph activity* of Vtune. This technique works by analyzing the points of entry and exit of all program and library functions. It can detect program modules when they are loaded at runtime. The call-graph activity provides important information about all the functions analyzed: self time, waiting time, and calling features such as number of calls, callers and callee functions, etc. We used this activity to collect information about the library functions that consumed more time in our cases of study, both in the TBB and the OpenMP codes. That way, we

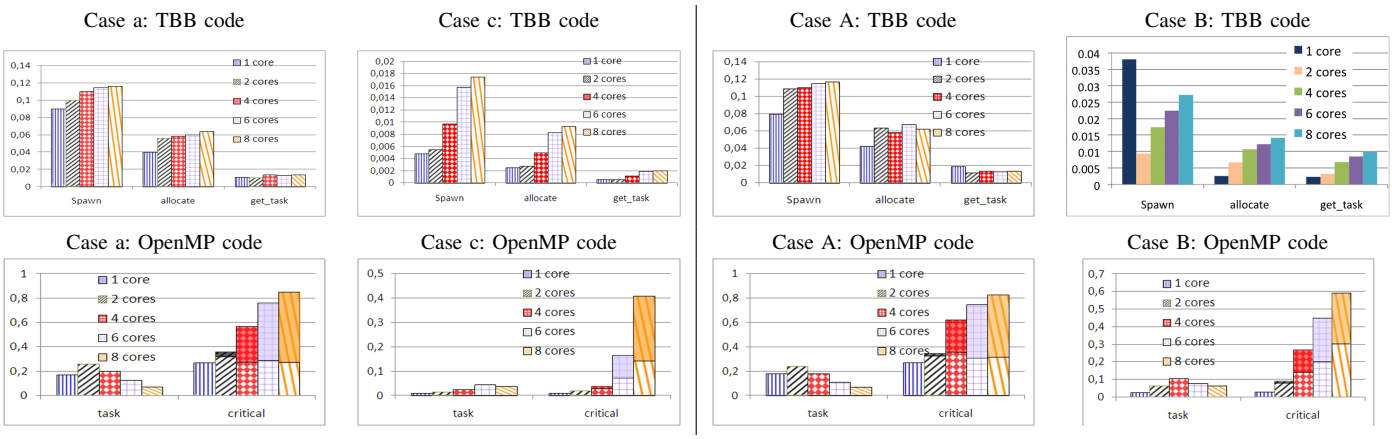


Figure 9. Call graph results for the most time consuming library functions, for Cases a and c (balanced workload), and Cases A and B (unbalanced workload), for TBB and OpenMP codes in 1,2,4,6 and 8 cores. The y-axes present the ratio between the self time of each library function and the total execution time

can study the overhead produced by TBB library functions an the OpenMP directive calls. The results for the most time consuming library functions are shown in Fig. 9. The y-axes represents the ratio between the self time of each function and the total time.

From Fig. 9, we see that for the TBB codes, the library functions that consume more time are: `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). When the task grain is fine (Cases a and A), the overheads due to `spawn`, `allocate` and `get_task` will be higher than when the task grain is coarse (Cases c and B). In particular, `spawn` is the main source of overhead for any number of cores, specially in Cases a and A (fine task granularity), where it can reach a 12% of execution time, while this function barely reachs the 3% of overhead in Cases c and B. This result corroborates the TBB yield loss that we noticed in Figs. 7 and 8 for Cases a and A. Interestingly, the overheads tend to increase with the number of cores, so a promising research line could be to limit or control the overheads of TBB (mainly the `spawn` method) for a high core count configuration.

Similarly, in Fig. 9 we see the library functions that consume more time for the OpenMP codes: `task` (it accounts for the `omp_task` and `omp_task_alloc` internal functions which are associated to the directive spawn a new task), and `critical` (it accounts for the `omp_critical` and `omp_end_critical` internal functions, that are associated to the critical directive -used to access in mutual exclusion to the critical sections). From these results, we see that the overhead due to task creation in OpenMP codes is significant in Cases a and A (fine granularity), where it can reach around 20% of execution time, although differently to TBB implementations, it does not tend to increase with the number of cores. However, the overhead due to the critical section management is the main source of inefficiency. We have shadowed in the top side of the critical bar, the contribution due to a waiting time produced by an internal `omp_lock_acquire` function that is called by `omp_critical`. Precisely, that locking is a contributing factor that significantly increases the waiting time with the number of cores in all the cases, more specially in the fine grain ones where it can account for near the 60% of execution time. This explains (added to the task creation overhead) the poor scalability results we saw in the fine grain task OpenMP codes in Fig. 7-Case a and Fig. 8-Case A. The main problem is that the waiting time due to the locking can become in a serialization bottleneck even in the coarse grain OpenMP codes when the number of cores increases (see the trend in Fig.9 for Cases c and B).

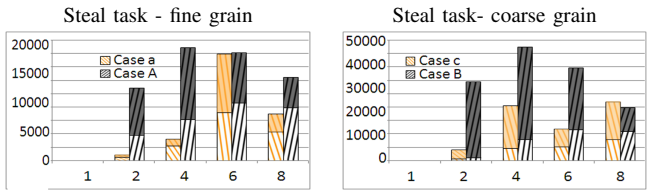


Figure 10. Number of calls for the `steal_task` and `sched_yield` TBB functions

Other interesting issue we wanted to analyze was the number of stealings in TBB and the overhead associated, in order to understand if the task stealing distinguish feature of TBB is relevant in the wavefront problems. For it, we focused on the `steal_task` function, which is the responsible of such feature, and the `sched_yield` which is called by the TBB scheduler when, after a certain number of attempts, the stealing fails (what usually happens when there is not enough work at the initial and last computations). We collected the time consumed by these functions and the number of calls to them, using the Vtune call-graph activity.

Regarding the times consumed by the two functions, we found that, in all the cases, they were very small (they spend less than 0.1% of execution time, respectively). In Fig. 10 we show the number of calls to the `steal_task` function and, shadowed on the top side of each bar, the number of calls to the `sched_yield` function, both for the fine granularity task cases (a and A) and for the coarse granularity ones (c and B). Obviously, the non shadowed part represents the successful steals. From the figure we see that, in general, the number of stealing attempts is higher in the coarse task granularity cases, and it tends to increase with the number of cores. This is reasonable, because with a coarse grain task workload it is necessary more time to process the initial and last elements, and with more cores there will be more idle cores trying to steal work. In addition we note that there is always much more attempts, as well as successful steals in the load unbalance scenarios for each task granularity situation. On the other hand, the difference between the number of successful steals on an unbalanced case and the number of successful steals on the corresponding balanced case, seems a little more significant for the fine grain task scenario (left figure) than for the coarse one (right figure), what hint us that the workload in Case A is quite unbalanced. Therefore the work stealing feature has been another contributing factor that explains the non degradation in performance for this fine task granularity case (compared with the balanced Case a), is spite of the high load unbalance situation. However we think that the current work stealing approach is perhaps too much aggressive, specially in the coarse task granularity cases, where there are a very high number of steal attempts but a small ratio of success. Although the times consumed by this function are small, it consumes energy, which can be saved if the core is halted. An interesting open research could be to dynamically adjust and reduce the steal attempts when the scheduler notes a low ratio of successful steals and let the core halts in these cases.

B. Real problems case study: comparison of TBB vs. OpenMP templates

Now, we conduct a new set of experiments to evaluate the performance of our TBB and OpenMP wavefront templates, that we presented in section III. We use three wavefront benchmark codes, that represent real problems: Checkerboard [6], Financial [2] and Floyd [11] algorithms.

We present the speedups for our TBB and OpenMP wavefront templates in Fig. 11. We repeat the experiments for different problem matrix sizes, obtaining similar results. From the figures, we clearly see that our TBB wavefront implementation always outperforms the OpenMP one, for any number of cores. We should mention that in all these real problems we have found fine to medium grain size tasks, with high load unbalance among them, as in Case A. Therefore we get a behavior and performance behavior similar to what we described for that case in the previous subsection.

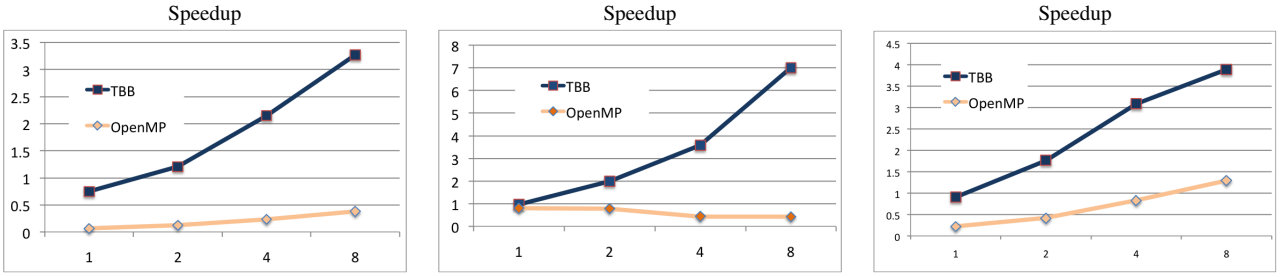
V. Related works

There is an interesting project [1] based on a pattern-based parallel programming framework (implemented in Java) that generates parallel programs from predefined design patterns, being one of those the wavefront pattern. However, it just allows the specification of some predefined dependencies patterns, so is not as general as our template.

Other research lines have focused on allowing the programmer or the compiler to specify high-level data and control flow information in the form of a graph, as the stream-graph [15] in the *StreamIt* project [14], or the prescriptions graph in the *Concurrent Collections for C/C++* (CnC) project [5]. The StreamIt language is aimed at streaming applications for stream processors or graphic processors [3], although there have been some works that have studied the mapping to general purpose multicores [8]. However, in this context wavefront problems have not been explicitly addressed. Other difference from our work, is that in the StreamIt language, although the user can initially use a set of simple annotations to mark the stages of computations and use a dynamic analysis tool to build the stream graph representation of the application, finally the programmer is responsible for improving parallelism and load balancing. In our approach, the programmer just specify the tasks and the dependencies, leaving the runtime of the system to deal with the parallelism and load balancing issues. On the other hand, in the CnC language, programs are written in terms of high-level application specific operations, being this language targeted to general purpose multiprocessors. The user just must specify the semantic constraints and indicate how data and control flow among the steps of the algorithm. Although CnC allows to specify several programming paradigms, it is not trivial to specify the wavefront paradigm. Anyway, as the CnC runtime is based in the TBB library it is more related to our work. In fact we plan as a future work to study the design of wavefront templates in CnC and evaluate its performance.

VI. Conclusions

In our implementation of the wavefront paradigm we have found that tasks are a promising programming model. In such implementation, we have required a mutual exclusion to control the concurrent access to shared counters, which are necessary to check the dependencies and to spawn the new tasks. We have implemented this mutual exclusion through the use of the *atomic* capture feature in TBB, and the *critical* pragma in OpenMP. Through our experimental evaluation we have found that *atomic* capture allows a more efficient implementation, because an important source of overhead and serialization bottleneck in the OpenMP codes is due precisely to the critical section. Other important source of



(a) (b) (c)

Figure 11. Template results for different real applications: (a) Checkerboard algorithm with matrix size 1,500 x 1,500; (b) Financial problem with matrix size 300 x 300; (c) Floyd code with matrix size 5,000 x 5,000

overhead in the OpenMP implementation, specially in the context of fine grain size tasks, is due to task creation. On the contrary, TBB implementation offers low overhead for task creation and management, even for very fine grain task granularities problems what explains its more competitive performance, both in load balanced and unbalanced scenarios. To democratize the use of tasks for the development of wavefront codes we have implemented a high level template that frees the programmer from the burden of dealing with low level task programming details. We have validated this template by using it to parallelize three real dynamic programming algorithms, finding that TBB-based template always outperforms OpenMP-based one.

ACKNOWLEDGMENTS

This work was supported in part by the following Spanish projects: TIN2006-01078 from Ministerio de Ciencia e Innovación, and P08-TIC-3500 from Junta de Andalucía.

REFERENCES

[1] John Anvik, Steve MacDonald, Duane Szafron, Jonathan Schaeffer, Steven Bromling, and Kai Tan. Generating parallel programs from the wavefront design pattern. *Parallel and Distributed Processing Symposium, International*, 2:0104, 2002.

[2] Gilles Brassard and Paul Bratley. *Fundamentals of algorithmics*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[3] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.

[4] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

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

[6] Vazirani Umesh Dasgupta Sanjoy, Papadimitriou Christos. *Algorithms*. McGraw-Hill Higher Education, 2007.

[7] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. Evaluation of OpenMP Task Scheduling Strategies. In Rudolf Eigenmann and Bronis R. de Supinski, editors, *Lecture Notes in Computer Science: Proceedings of the 4th International Workshop on OpenMP*, volume 5004, pages 100–110. Springer, Springer, May 2008.

[8] Jayanth Gummaraju, Joel Coburn, Yoshio Turner, and Mendel Rosenblum. Streamware: programming general-purpose multicore processors using streams. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 297–307, New York, NY, USA, 2008. ACM.

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

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

[11] Bruno Richard Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. wiley, 2000. 635 pp. ISBN 0-471-34613-6.

[12] Software Engineering Support Programme. Intel(r) vtune(tm) performance analyzer 8.0.2 for linux. Technical report, Mathematical Software Group Rutherford Appleton Laboratory Chilton, 2006. http://www.sesp.cse.clrc.ac.uk/.

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

[14] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *CC '02: Proceedings of the Compiler Construction*, France, 2002.

[15] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.