A case study of the task-based parallel wavefront pattern[1]

Antonio J. DIOS [a], Angeles NAVARRO [a] Rafael ASENJO [a] Francisco CORBERA [a] and Emilio L. ZAPATA [a]

[a] *Dept. of Compt. Architect. Univ. of Malaga, Spain.*
{antjgm, angeles, asenjo, corbera, ezapata}@ac.uma.es

Abstract. This paper analyzes the applicability of the task-programming model to the parallelization of the wavefront pattern. Computations for this type of problem are characterized by a data dependency pattern across a data space. This pattern can produce a variable number of independent tasks through traversing this space. Different implementations of this pattern are studied based on the current state-of-the-art threading frameworks that support tasks. For each implementation, the specific issues are discussed from a programmer's point of view, highlighting any advantageous features in each case. In addition, several experiments are carried out, and the factors that can limit performance in each implementation are identified. Moreover, some optimizations that the programmer can exploit to reduce overheads (task recycling, prioritization of tasks based on locality hints and tiling) are proposed and assessed.

Keywords. Wavefront computation, Task programming model, Task recycling

1. Introduction

Wavefront is a programming pattern that appears in scientific applications such as those based on dynamic programming [1] or sequence alignment [2]. In this type of pattern, data elements are distributed on multidimensional grids representing a logical plane or space [3]. The elements must be computed in order because there are dependencies among them. One example is the 2D wavefront where computations start at a corner of a matrix and progress across the plane following a diagonal trajectory. Each anti-diagonal represents the number of computations or elements that could be executed in parallel without dependencies among them.

There have been several research studies that have targeted the problem of parallelizing wavefront patterns, either on distributed memory architectures [4] or on heterogeneous architectures such as SIMD-enabled accelerators [5], the Cell/BE architecture [6] or GPUs [7]. In all these studies, the authors have focused on vectorization optimizations and on appropriate work distribution on each architecture, and thus the programmer had to deal with several low-level programming details. The present study differs in that the focus is on higher-level programming approaches that release the user from low-level architectural details.

[1]This material is based on work supported by Spanish projects: TIN2006-01078 from the Ministerio de Ciencia e Innovación, and P08-TIC-3500 from the Junta de Andalucía.

Recently, several parallelization frameworks that provide support for parallel tasks rather than parallel threads [8,9,10] have become available. Task-based programming seems more appropriate than thread-based programming when addressing the implementation of parallel wavefront problems. First, tasks are much lighter than logical threads, so in wavefront applications with light or medium computational workload (as in our case), tasks are a more scalable constructor. Second, 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. Finally, the runtime task schedulers can obtain 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, what offers improved scalability [9]. This differs from how an O.S. thread scheduler works.

The aim of this paper is to explore the programmability and performance of different *OpenMP 3.0* [8], *Intel Threading Building Blocks* (TBB) [9], Cilk [11] and *Intel Concurrent Collections* (CnC) [10] implementations of a parallel task-based 2D wavefront pattern. We begin by highlighting the main features of the different implementations from a programmer's point of view (Sec. 2.1). Next, we conduct several experiments to identify the factors that can limit the performance for the different implementations (Sec. 2.2). These experiments demonstrate two important sources of overheads: synchronization and task creation/management. Whereas the former are inevitable, unless specialized synchronization hardware becomes available, the latter can be drastically reduced in the wavefront patterns. To this end, the user can guide the task scheduler by using the *task recycling* (or *task passing*) mechanism, in addition to prioritizing the execution of tasks to guarantee a cache-conscious traversal of the data structure. In particular, TBB provides sufficient flexibility to efficiently implement these optimizations, which we describe and assess in Sec. 3.

2. Implementations of a wavefront pattern based on tasks

As a case-study of the wavefront pattern, we select a simple 2D problem in which we compute the function `A[i,j]=foo(gs, A[i,j], A[i-1,j], A[i,j-1])` for each cell of a $n \times n$ 2D grid, A. Clearly, each cell has a data dependence with two elements of the adjacent cells; however, the anti-diagonal cells are totally independent and thus can be computed in parallel.

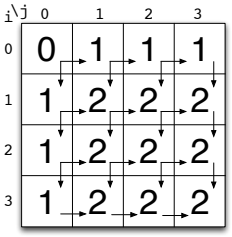
In this task parallelization strategy, the basic unit of work is the computation performed by function `foo` at each (i,j) cell of the matrix. Without loss of generality, we assume that there will be auxiliary work on each cell, and the computational load of this work will be controlled by the `gs` parameter of the `foo` function. In this way the granularity of the tasks can be defined and therefore it is possible to study the performance of different implementations depending on task granularity, as well as situations with homogeneous or heterogeneous task workloads, as shown in section 2.2.

The pseudo-code of each task in the wavefront problem is shown in Fig. 1(a). In Fig. 1(b), the arrows show the data dependence flow and the same dependence information captured by a 2D matrix with counters. The value of the counters shows how many tasks are pending. Only the tasks with the corresponding counter nullified can be dispatched. Generalizing, each ready task first executes the task body, `Task_Body()`, and

then decrements the counters of the tasks depending on it. If this decrement operation ends with a counter equal to 0, the task is also responsible for spawning the new independent task. It is important to note that the counters will be modified by different tasks that are running in parallel. Thus, access to the counters must be protected in a critical section (lines 2-6 and 7-11).

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

(a) Pseudo-code for each task



(b) `counter` matrix

Figure 1. Typical wavefront traversal with dependencies translated into counters

2.1. OpenMP, Cilk, TBB and CnC specific issues

The coding of the wavefront problem using OpenMP is quite straightforward. We first define an `Operation` recursive function, in which task work is performed, `Task_Body()`, and then the OpenMP directive "`#pragma omp critical`" is used to access and update the dependence counters individually. Then, if the corresponding counter reaches 0, a neighbour task is recursively spawned using the directive "`#pragma omp task`" preceding the recursive call to the `Operation` function. This implementation is named OpenMP_v1.

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. atomic decrement and compare); however, this is not the case. Another way to obtain a finer-grain locking implementation 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 access to the shared counters. This alternative implementation is named OpenMP_v2. Clearly, this approach is less productive than the previous OpenMP_v1 version from a programmer's point of view, because the programmer now has to ensure that the code is deadlock free. Another important issue in this approach is the waste of memory and time due to the locks matrix storage and initialization.

Regarding the Cilk implementation, the language extensions provided by the Intel Cilk Plus framework [11] were used. There are two main constructors from Cilk Plus required by the 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 following it from executing until all the previously spawned children of the parent have run to completion. Cilk does not support atomic captures, so locks must be used to control access to the shared counters. The coding details differ mainly in the use of the `pthreads_lock()` functions (Cilk does not provide constructors for locks), and in that `cilk_sync` statements are required.

With respect to the TBB implementation, the atomic template class provided is a construct of interest. Thus, the matrix of counters can be declared by using `atomic<int>**counter`. For example, the expression "`if(--counter==0) action()`" is safe and just one task will execute the "`action()`". Compared to locks, atomic operations are faster and do not suffer from deadlock and convoying; this is distinguishing feature that OpenMP or Cilk does not currently support. This implementation is named TBB_v1.

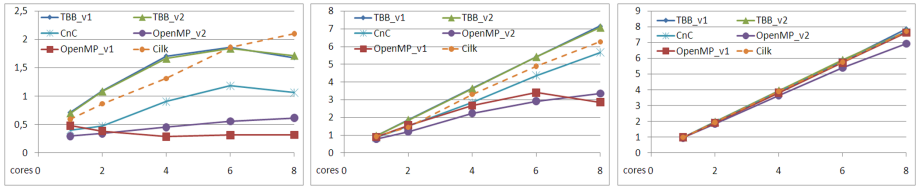
There is also a higher-level programming approach to code wavefront codes in TBB [9]. The idea is to use the TBB `parallel_do_feeder` class template. Basically, this class implements a work-list algorithm in such a way that new tasks can be added dynamically to the work-list by invoking the `parallel_do_feeder::add()` method. Thus, the `spawn()` invocations would be replaced by `feeder.add()` invocations. This implementation is called TBB_v2.

Finally, CnC [10] also provides a runtime library based on TBB and so it can also be considered a task programming framework. CnC provides three types of static collections: i) the computation steps which are the high-level operations. In this case these are the `Operation` collection; ii) the data items collections; and iii) the control tags collections that prescribe the steps. In this case they are the `ElementTag` collection. In CnC, the collections are connected via data and control dependencies that specify the program's ordering constraints. The basic difference with the TBB code is that instead of explicit calls to `spawn()`, we have to generate the control tags for the ready-to-run neighbours invoking the `ElementTag.put()` CnC method. This new implementation is named CnC to.

2.2. Experimental assessment

Next, we present several experiments that assess the performance of the previously described implementations (OpenMP_v1, OpenMP_v2, Cilk, TBB_v1, TBB_v2 and CnC). Specifically, the library versions are Intel Open_MP 3.0, Cilk Plus, TBB 3.0 and CnC 0.4. For all the experiments, a multicore machine with 2 quad-cores Intel(R) Xeon(R) CPU X5355 at 2.66GHz, SUSE LINUX 10.1 was used. The codes were compiled with icc 11.1 -O3. Each code was executed 5 times and the average execution time of the runs was computed to obtain the execution times. Next, the speedups were calculated in relation to the sequential code time.

In the first case-study, we fixed the task workload to a constant grain size in all the cells. Therefore, the workload will be evenly balanced among the threads (except for the initial and final computations). To do this, the `gs` parameter of the `foo` function was set to a constant value. This parameter sets the number of floating point operations per task. In our experiments three task granularities were assessed: i) the fine granularity case (which corresponds to approximately 200 floating point operations); ii) the medium granularity case (around 2,000 FLOP); and iii) the coarse granularity case (i.e. 20,000 FLOP). These granularities were selected in accordance with our observation of real wavefront codes [12]. The speedups for each case are shown in Figs. 2(a), (b), and (c), respectively. For all the cases, the matrix size was 1,000 $\times$ 1,000. Note that aborted executions sometimes occurred 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.



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

Figure 2. Speedup results for constant task granularities. The x-axes represent the number of cores

In the fine- and medium-grain cases, the TBB versions outperformed all the other implementations, except in the fine-grain case with 8 cores, where Cilk scaled better (although recall that the Cilk version was sometimes unable to run to completion). On the other hand, in the coarse granularity case (c) all the implementations exhibited similar behaviour. Both TBB_v1 and TBB_v2 presented similar speedups, although in the fine-granularity case (a) TBB_v1 exhibited slightly better results from 1 to 6 cores. The OpenMP_v1 provided the poorest performance. In contrast, the OpenMP_v2 implementation based on locks scaled better. On the other hand, the performance of the CnC implementation was between those of the TBB and OpenMP implementations.

To identify the sources of overhead for each implementation, we decided to profile our codes by employing the *call-graph activity* of Vtune [13]. Here, we focus our discussion on the results of the version that ran to completion with the worst performance: the fine-granularity case (a). It was observed that the overhead of task creation in OpenMP is high (around 30% of the execution time). However, the main source of inefficiency is due to the critical and lock functions. In particular, the main contributing factor of overhead in the OpenMP_v1 code is the contention due to acquiring the global lock used by the internal functions in the critical directive. This contention significantly increases with the number of cores, so the resulting waiting time accounts for nearly 60% of the total time on 8 cores. In addition to the task creation overhead, this explains the poor scalability results shown for the OpenMP_v1 code in Fig. 2-case (a). Compared to this, the waiting time due to the explicit `omp_lock()` calls in the OpenMP_v2 code is around 20% on 8 cores. According to the TBB profiling results, we observed that the functions that consume more time are as follows: `spawn()`, `allocate()` and `get_task()` (that obtains the next task to dispatch). 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 notable; this explains the slight difference in performance shown in Fig. 2-case (a) for the TBB codes. In any case, we saw that the main source of overhead is due to the `spawn()` method, and that it increases when the number of cores increases. This is more noticeable in the TBB_v1 code, where `spawn()` may consume nearly 28% of the execution time for 8 cores. According to the CnC profiling results, the most time-consuming functions are `prepare()` and `schedule()` (helper functions called by the `ElementTag.put()` method, see [14]), which account for more than 50% of execution time. This explains the poor performance shown in Fig. 2-case (a) for the CnC code. The overhead due to the helper functions tends to decrease for coarser task granularities, which explains the superior scalability observed for the medium- and coarse-granularity cases.

As an additional case-study, we changed the task workload on each cell to a variable grain size. Variable values were assigned to the `gs` parameter of the `foo` function. There

could be some load imbalance among the threads in this scenario. However, we observed that the variable workload was not a factor that affected performance, because load imbalance is successfully managed by the libraries task-schedulers. In particular, Cilk, TBB (CnC) and the Intel OpenMP runtimes manage task scheduling through the work-stealing strategy, which in these experiments has proved to be an effective mechanism for load-balancing in the wavefront pattern. Particular attention was paid to the overheads due to the work stealing functions and it was found that they always represented less than 1% of execution time.

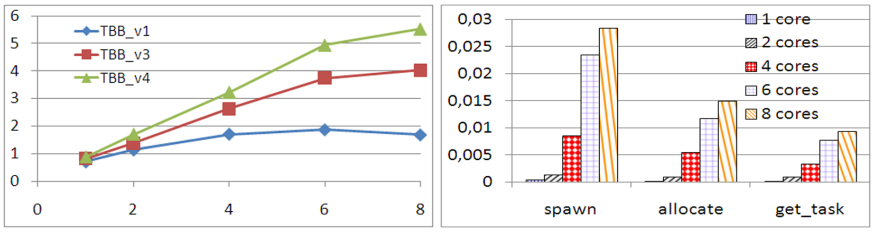
3. Further Optimizations

The sources of inefficiency due to the lack of atomic capture in the OpenMP 3.0 framework could be overcome by directly using compare-and-swap (CAS) instructions. This alternative OpenMP implementation (named OpenMP_v1.2) was also assessed. The results for the constant fine-task granularity case (200 FLOP) show that synchronization based on the CAS instructions mechanism significantly reduces the contention problem in the OpenMP codes, making the OpenMP optimized version the best one on 6 and 8 cores. For the medium-grain case, the TBB_v1 code shows slightly better performance than the optimized OpenMP code, whereas for the coarse-grain case, both OpenMP_v1.2 and TBB_v1 codes behave similarly. In any case, these results demonstrate that 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).

In addition, the cost of task creation and task management methods can be significant if task granularity is not sufficiently coarse. The programmer can reduce these costs by using the explicit task passing (or *task recycling*) mechanism [9], which is available in TBB. In the wavefront pattern, each task has the opportunity to spawn two new tasks (east and south neighbours). One of them can be prevented from spawning by returning a pointer to the next task and instead of spawning the new task, the current task recycles into the new one. Two aims are achieved: the number of calls to `spawn()` is reduced and time is saved for obtaining new tasks from the local queue (reducing the number of calls to `get_task()`). In OpenMP or Cilk codes, this recycling mechanism could in some way be emulated by recursively calling the task body of a ready-to-dispatch new task, thus avoiding spawning. However, note that this emulation strategy could exhaust the stack when the number of nested tasks is too large, a problem that the TBB recycling mechanism avoids.

In addition, by recycling, we provide the scheduler with hints on how to prioritize the execution of tasks to guarantee a cache-conscious traversal of the data structure. The main idea is to set two flags when there is a ready-to-dispatch task to the east or south of the executing task. Subsequently, according to these flags, we recycle the current task into the east or south tasks. Since in this example the data structure is stored by rows, note that if the east and south tasks are both ready, the data cache can be better exploited by recycling into the east task. Thus, the same thread/core executing the current task will manage the task traversing the neighbour data, and thus we take full advantage of the spatial locality. In this case, we recycle into the east task and spawn a new south task that would be subsequently executed. 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 fully assess the impact of each mechanism (recycling and locality), the performance of two versions were studied: TBB_v3, a version that prioritizes the south task instead of the east task. Its aim is to isolate the advantages of recycling (without exploiting cache); and TBB_v4, the version that implements both recycling and locality. Fig. 3(a) shows the speedups for the TBB_v1 (shown here as 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, when speedups were measured for other fine-grain sizes it was found that the finer the granularity, the better the improvement. Furthermore, it is of interest that a great deal of the improvement is due to the recycling optimization, as indicated by the TBB_v3 enhancement over the TBB_v1 version.



(a) Speedups. Num. of cores in the x-axis (b) Overhead in TBB_v3 and TBB_v4

Figure 3. 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, the call-graph activity of Vtune is used again to collect information on the most time-consuming functions in the optimized implementations. Fig. 3(b) shows 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. The figure shows that the percentage of time wasted by these functions is less than 6% of execution time. In fact, for each internal function there is a reduction of around one order of magnitude in the recycling versions when compared to the TBB_v1 version. On the other hand, thanks to the Vtune event-based sampling tool, we obtained the L1D and L2 miss ratios for the TBB_v3 and TBB_v4 implementations, and for the fine-granularity case again. The measurements confirm that cache miss ratios are much higher in TBB_v3 than in TBB_v4.

As described above, the performance of the wavefront algorithms consistently decreases as the task workload grain becomes finer. To counteract this trend, the tiling technique is a well-known solution to obtain coarser grain size and it can also be applied to wavefront problems. By tiling we achieve two goals: to reduce the number of tasks (and therefore, the number of spawns); and to save some overhead in wavefront bookkeeping (memory space and the initialization time of the counter/dependence matrix, which is now smaller due to it requiring a counter per block-tile, and not one per matrix element). Another possibility could be to select rectangular tiles (instead of square tiles) to take better 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. The assessment of rectangular tiles has

been left to future studies. In this experiment, an increment in speedup for 8 cores was found of around 9% when $BS = 10$, which was the optimal block size. For larger block sizes, speedups start to decrease again, because the degree of parallelism is lower when we increase the size of the block while keeping the problem size constant (the matrix size). The interested reader may wish to refer to [14], where the different versions, code snippets, experimental results and graphs are presented and discussed in more detail.

4. Conclusions

Assessment of the different task-based implementations of a wavefront pattern shows that TBB provides some distinguishing features that allow more efficient implementations, particularly for the fine-grain case. This is frequently found in real wavefront problems and is more challenging. Features such as atomic capture and the task recycling mechanism, coupled with prioritizing tasks to exploit data locality, have been shown strongly improve performance, especially 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 help less experienced users code wavefront codes. Future work includes implementing a wavefront template for the TBB library (such as `parallel_do` or `pipeline`) that encapsulates the mentioned optimizations and allows the developer to express the wavefront dependencies and the data locality hints without having to take into account 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 Distr. Processing Symp.*, 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] E. C. Lewis and L. Snyder, "Pipelining wavefront computations: Experiences and performance," in *HIPS'99: Workshop on High-Level Parallel Programming Models and Supportive Environments*, 1999.

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

[6] 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*, 2008, pp. 13–22.

[7] B. Liu, G. Clapworthy, and F. Dong, "Wavefront raycasting using larger filter kernels for on-the-fly GPU gradient reconstruction," *The Visual Computer*, vol. 26, no. 6-8, pp. 1079–1089, 2010.

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

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

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

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

[12] A. Dios, R. Asenjo, A. Navarro, F. Corbera, and E. Zapata, "High-level template for the task-based parallel wavefront pattern," in *IEEE Intl. Conf. on High Perf. Comp. (HiPC)*, Bengaluru, Dec. 2011.

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

[14] A. Dios, R. Asenjo, A. Navarro, F. Corbera, and E. L. Zapata, "A case study of the task-based parallel wavefront pattern," in *Technical Report at http://www.ac.uma.es/∼asenjo/research/*, June 2011.