Task-parallel reductions in OpenMP and OmpSs

Jan Ciesko[1] Sergi Mateo[1] Xavier Teruel[1]
Vicenç Beltran[1] Xavier Martorell[1,2]

[1] Barcelona Supercomputing Center [2] Universitat Politècnica de Catalunya
{jan.ciesko,sergi.mateo,xavier.teruel,
vicenc.beltran,xavier.martorell}@bsc.es

Abstract. The wide adoption of parallel processing hardware in mainstream computing as well as the raising interest for efficient parallel programming in the developer community increase the demand for parallel programming model support for common algorithmic patterns. In this paper we present an extension to the OpenMP task construct to add support for reductions in while-loops and general-recursive functions. Further we evaluate its implications on the OpenMP standard and present a prototype implementation in OmpSs. Application scalability is achieved through runtime support with static and on-demand thread-private storage allocation. Benchmark results confirm scalability on current SMP systems.

Keywords: OpenMP, Task, Reduction, Recursion, OmpSs

1 Introduction

Reductions are a reoccurring algorithmic pattern in many scientific, technical and mainstream applications. Their characteristic non-atomic update operation over arbitrary data types makes their execution computationally expensive and parallelization challenging.

In programming, a reduction occurs when a variable, *var*, is updated iteratively as

$$iter: \ var = op(var, expression),$$

where *op* is a commutative and associative operator performing an update on *var* and where *var* does not occur in *expression*. In case of parallel execution, mutual exclusive access is required to ensure data consistency.

Taking a broader look at usage patterns across applications reveals three common types of reductions: for-loop, while-loop and recursive. For-loop reductions enclose a reduction in a for-loop body. They are often used in scientific applications to update large arrays of simulation data in each simulation step (such as updating particle positions by a displacement corresponding to a time slice) or in numerical solvers where values are accumulated over a scalar to indicate convergence behavior and break conditions [5].

For-loops represent the class of primitive-recursive algorithms where the iteration space is computable and where control structures of greater generality are not allowed. The iterative formulation of primitive-recursive functions is currently supported in OpenMP [6].While-loop reductions represent another usage pattern and define the class of general-recursive functions. They appear in algorithms where the iteration space is unknown such as in graph search algorithms.

The last occurrence represents recursions. Recursive reductions can be found in backtracking algorithms used in combinatorial optimization. Even though one could argue that for each recursion an iterative formulation exists (either as a for-loop or a while-loop), recursions often allow very compact and readable formulations. Examples of a while-loop and recursive reduction are shown in Figure 1.

If you write these two lines in a single one, you will have both algorithms in 9 lines... Which will make the figure more symetric

```
1 int nqueens(...){
2   if (cond1(...))
3     return 1;
4   int count = 0;
5   for (int row = 0; row < n; row++){
6     if (cond2(...)))
7       count += nqueens(...);
8   }
9   return count;
10 }
```

(a)

```
1 ...
2 int red = 0;
3 int foo(node_t node,...){
4   while(node->next) {
5     red+=bar(node->value);
6     node=node->next;
7   }
8 }
9 ...
```

(b)

Fig. 1: The recursive, schematic implementation of n-Queens (a) and a graph algorithm (b) show the different occurrences of reductions in applications

In this work we propose an extension to the OpenMP standard by adding support for while-loop and recursive reductions through the *task reduction* pragma. Formally, this extends the existing support for primitive-recursive, iterative algorithms by the class of general-recursive algorithms for both, iterative and recursive formulations. In terms of parallel programming, the proposed task reduction allows the expression of so called task-parallel reductions. Further we propose an compliant integration into OpenMP and present a prototype implementation based on OmpSs.

The rest of the paper is structured as follows. Chapter 2 introduces the language construct. In Chapter 3 we introduce OmpSs, discuss compiler transformations and runtime implementation. Benchmark results are shown in Chapter 4. Finally we discuss related work in Chapter 5 and conclude this work in Chapter 6 with a summary and outlook on future work.

2 Task-parallel reductions with OpenMP

The idea to support task-parallel reductions builds on top of the conceptual framework introduced with explicit tasking in OpenMP. Since tasking allows to

express concurrent while-loops and recursions, it represents a convenient mechanism to support task-parallel reductions as well. For its definition we use the current standard specification as a baseline and add a set of rules describing data consistency and nesting. While this work is written with a certain formalism in mind, it does not represent a language specification.

2.1 Definition

The task reduction construct is defined as:

```
1 #pragma omp task [clauses] reduction (identifier : list)
2 structured-block
```

The reduction clause in the task construct declares an asynchronous reduction over a list of items. Each item is considered as if declared *shared* and for each item a private copy is assigned. At implicit or explicit barriers or task synchronization, the original list item is updated with the values of the private copies by applying the combiner associated with the reduction-identifier. Consequently, the scope of a reduction over a list item begins at the first encounter of a reduction task and ends at an implicit or explicit barrier or task synchronization point. We call this region a reduction domain. Implications on synchronization in case of domain nesting is conforming to the OpenMP specification.

We would like to point out that the provided definition is generic and does not restrict the usage of task-parallel reductions to any particular enclosing construct. However, as in this case the scope of a task-parallel reduction is defined by both task synchronization as well as by barriers, its support would require to modify their current implementations. In particular they would need to check for outstanding private copies and reduce them. A solution to minimize the impact on unrelated programming constructs is to restrict the use of task-parallel reductions to the context of a *taskgroup*.

In the rest of this Chapter we discuss implications of this proposal on a the *taskwait* and taskgroup pragmas, reductions on data dependencies and nesting.

2.2 Reductions on taskwait

The *taskwait* construct specifies a wait on the completion of child tasks in the context of the current task and combines all privately allocated list items of all child tasks associated with the current reduction domain. A taskwait therefore represents the end of domain scope. The previous example shown in Figure 1 can be easily parallelized as shown in Figure 2.

Figure 1 (b)

2.3 Support in taskgroups

The *taskgroup* construct specifies a deep wait on all child tasks and their descendent tasks. After the end of the taskgroup construct, all enclosed reduction domains are ended and original list items are updated with the values of the private copies. Similarly to a taskwait construct, task-parallel reductions require to

```
1 ...
2 int red=0;
3 while(node->next) {
4   #pragma omp task reduction (+:red)
5   {
6     red+=bar(node->value);
7   }
8   node=node->next;
9 }
10#pragma omp taskwait
11return red;
```

Fig. 2: A concurrent reduction using a *taskwait* to ensure data consistency so a function would return a correct value of *red*

extend their role of task synchronization to activly perform a memory operation to restore consistency. Figure 4 shows a an example where a reduction domain is ended implicitly at the end of a taskgroup construct.

I think you want to refer figure 3, otherwise figure 3 is never referenced...

```
1 ...
2 int red=0;
3 #pragma omp taskgroup
4 {
5   while(condition(){
6     #pragma omp task reduction (+:red)
7     red += foo();
8   }
9 }
10return red;
```

You miss a ")"

The example does not show nesting tasks (which potencial they can be in foo()), btw I think it is not a big issue due you explain properly the semantic ok taskgroup, but this is not the best example.

Fig. 3: A concurrent reduction within the *taskgroup* performs a wait on all children and their descendant tasks (this is often referred to as deep wait)

2.4 Reductions on data dependencies

Data-flow based task execution allows a streamline work scheduling that in certain cases results in higher hardware utilization with relatively small development effort. Task-parallel reductions can be easily integrated into this execution model but require the following assumption. A list item declared in the task reduction pragma is considered as if declared *inout* by the *depend* clause. As this would effectively serialize task execution because of an "inout" operation over the same variable, this dependency needs to be relaxed.

An example, where a reduction domain begins with the first occurrence of a participating task and is ended implicitly by a dependency introduced by a successor task, is shown is Figure 4. In this example the actual reduction of private copies can by overlapped by the asynchronous execution of *bar* which again might improve hardware utilization.

```
1 ...
2 int red=0;
3 #pragma omp taskgroup
4 {
5   for(int i=0; i<SIZE; i+=BLOCK){
6     #pragma omp task shared(array) reduction (+:red)
7     for(int j=i; j< i+BLOCK; ++j){
8       red += array[j];
9     }
10  }
11  #pragma omp task
12  bar();
13  #pragma omp task shared(red) depend(in:red)
14  printf("%i\n",red);
15 }
16 return red;
```

Fig. 4: The reduction domain over the variable *red* is ended by a task dependency

2.5 Nesting support

Nesting of tasks Nested task constructs typically occur in two cases. In the first, each task at each nesting level declares a reduction over the same variable. This is called multi-level reduction. In this case, a taskwait at each nesting level is not mandatory as long as a deep wait ensures proper synchronization later on. It is important to point out that only task synchronization that occurs at the same nesting level at which a reduction scope was created (that is the nesting level that first encounter a reduction task for a list item), ends the scope and reduces private copies. An example for a multi-level domain reduction is shown in Figure 5.

```
1 ...
2 int red = 0;
3 #pragma omp taskgroup
4 {
5   for(int i=0; i<SIZE; i+=BLOCK){
6     #pragma omp task shared(array) reduction (+:red)
7       for(int j=i; j< i+BLOCK; ++j){
8         #pragma omp task shared(array) reduction (+:red)
9           red += array[j] + bar(/*long_computation*/);
10      }
11    }
12 }
13 return red;
```

Fig. 5: A multi-level domain reduction is computed over the same variable by tasks participating at different nesting levels

One of them is misspelled

In the second occurrence each nesting level reduces over a different reduction variable. This happens for example if a nested task performs a reduction on task-local data. In this case a taskwait at the end of each nesting level is required. We call this occurence a nested-domain reduction. Figure 6 shows an example of an element-wise matrix summation, where inner tasks iterate over rows and

compute partial results that are then reduced by outer tasks to compute the final value.

```
1 ...
2 int red = 0;
3 for(int i = 0; i < SIZE_Y; i++{
4   #pragma omp task shared(array) reduction(+:red)
5   {
6     int red_local = 0;
7     for(int j = 0; j < SIZE_X; j+=BLOCK_X) {
8       #pragma omp task reduction(+:red_local)
9       for (int k = j; k < j + BLOCK_X; ++k){
10        red_local += array[i][k];
11      }
12      #pragma omp taskwait   Taskwait out
13    }                        of the loop
14    red += red_local;
15  }
16 }
17 #pragma omp taskwait
18 return red;
```

Fig. 6: Element-wise matrix sum implemented as a nested domain reduction, where each dimension is processes in different nesting levels over different variables

Interleaving reduction tasks with regular tasks in a nested scenario is not permitted, since the regular task would modify data of an ongoing reduction. By definition, within a reduction domain, the reduction variable can be only modified by operations declared as a reduction and are of the same operator.

General case The general support for nesting would allow scenarios where *worksharing* construct enclose the aforementioned constructs or where tasks and taskgroups enclose worksharing constructs. If this general support of task-parallel reductions is desirable depends on its necessity. As currently task-parallel reductions enclosed in the taskgroup construct represent a satisfactory approach, we defer the evaluation of the general case to future work. Figure 7 shows a concurrent version of the n-Queens application shown in Chapter 1.

3 Implementation in OmpSs

To evaluate requirements for front-end compilers as well as for runtime support we implemented the presented proposal in the OmpSs programming model[1]. OmpSs is a high-level, task-based, parallel programming model supporting SMPs, heterogeneous systems (like GPGPU systems) and clusters.

OmpSs consists of a language specification, a source-to-source compiler for C, C++ and Fortran [2] and a runtime [3]. The language defines a set of pragma annotations that allow a descriptive expression of tasks. With this information the runtime is capable of dependency-aware task scheduling. While this is similar

```
1 int nqueens(...){
2   if (cond1(...))
3     return 1;
4   int count = 0;
5   for (int row = 0; row < n; row++){
6     if (cond2(...)))
7       #pragma omp task reduction(+:count)
8       count += nqueens(...);
9   }
10  #pragma omp taskwait
11  return count;
12 }
```

Fig. 7: A concurrent implementation of N-Queens using the task reduction pragma over the reduction variable *nqueens*

to OpenMP, the OmpSs runtime implements a different execution model. In OmpSs, an application is launched as a single implicit task in an implicit parallel region that lasts throughout the execution of the application. Therefore the parallel construct nor barriers are supported and memory consistency is ensured through data dependencies and task synchronization directives. Consequently the rest of this work discusses compiler and runtime support for task and task synchronization constructs as supported in OmpSs. However we believe that similar implementations are possible for other OpenMP implementations.

3.1 Compiler support

The goal of the Mercurium compiler is to generate code transformations according to pragma annotations provided by the programmer. In case of encountering a reduction task, the compiler replaces all occurrences of the original reduction variable within the task by a reference to a previously requested thread-private reduction store, called TPRS. This transformation includes the following steps.

- Generate call to the runtime to obtain a TPRS. The runtime serves a TPRS corresponding to the current thread that is executing the task
- Replace all references to the original reduction variable within the task by a reference to the TPRS

The compiler transformation for this implementation applied to Figure 5 is shown in Figure 8.

3.2 Runtime support

The runtime implementation is based on the idea of privatization. In order to avoid the need for mutual exclusive access to the reduction variable, a thread-private copy (TPRS) is created and used as a temporal reduction target. Since its creation, initialization and processing later on are expensive operations, it is important to maximize the life span and reuse of a TPRS.

Therefore we introduce a thread-team private reduction manager object that tracks privatized memories and assigns them to requesting tasks. Consequently

```
1 ...
2 for(int i=0; i<SIZE; i+=BLOCK){
3 rt_create_task("task 1", args = {array, &red}) {
4   int *tp_red = rt_get_thread_storage(red);
5     for(int j=i; j< i+BLOCK; ++j) {
6       rt_create_task("task 2", args = {array, tp_red, j}) {
7         int *tp_tp_red = rt_get_thread_storage(tp_red);
8         (*tp_tp_red) += array[j] + bar(/*long_computation*/);
9       }
10    }
11    rt_taskwait();
12  }
13 rt_taskwait();
14 return red;
```

Fig. 8: Transformations applied by the compiler that redirect accesses to a redirecting thread-private reduction store

all tasks that are executed on the same thread and belong to the same reduction domain always receive the same allocated thread-private memory. Once the domain ends, one of the participating threads reduces all corresponding TPRSs.

Allocation strategies To evaluate memory allocation in more detail, we implemented two strategies called static and dynamic allocation. An execution diagram of an application where a parent task P creates four reduction tasks R running on two threads (*TID 1, 2*) is shown in Figure 10.

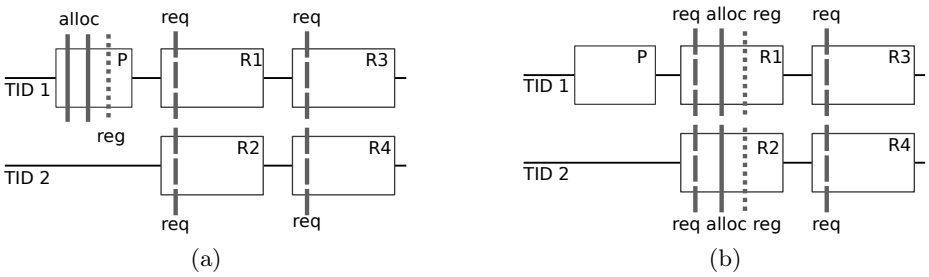


Fig. 9: Static (a) and dynamic (b) allocation differs in when and where thread-private memory is requested (req), allocated (alloc) and registered (reg) in the reduction manager object

Static allocation preallocates an array of thread-private reduction storages for all threads of a team (as defined by *omp_get_num_threads()*) at the moment when the first reduction task is created. This marks the beginning of a reduction domain. During execution, the runtime provides previously allocated TPRS objects according to a domain and thread identifier to requesting child tasks. The allocation in this implementation is performed by the encountering thread serially.

With dynamic allocation, memory is allocated on demand and at task execution. Once a task requests a thread-private storage, the runtime performs an allocation, registers the storage in the reduction manager and returns a TPRS. An allocation is performed for each first execution of a reduction task of a domain on a participating thread. In this case a reduction domain begins at the execution of the first reduction task. This allocation strategy does not create any work for the encountering thread as the allocation is called at execution time of child tasks in parallel.

The advantage of static allocation is that it allows to allocate memory in a single call (to malloc for example) and its implementation is lock free once all TPRSs have been allocated. On the other hand, allocation is in the critical path and potentially can result in allocating unused storage in case not all threads participate in the computation. Further this approach does not adapt to changing numbers of participating threads.

Dynamic allocation allocates memory in parallel and avoids unnecessary allocation for busy threads that will not participate in the reduction computation. Since the number of registered TPRS storages changes over time, this implementation requires a lock in the global manager which can potentially introduce lock contention for fine grained tasks. This approach corresponds to the idea of dynamic parallelism where problem size nor thread counts are unknown.

Nesting support In case of nesting, synchronization constructs might occur at any nesting level. In this case the runtime must be able identify storages that correspond to an ending domain.

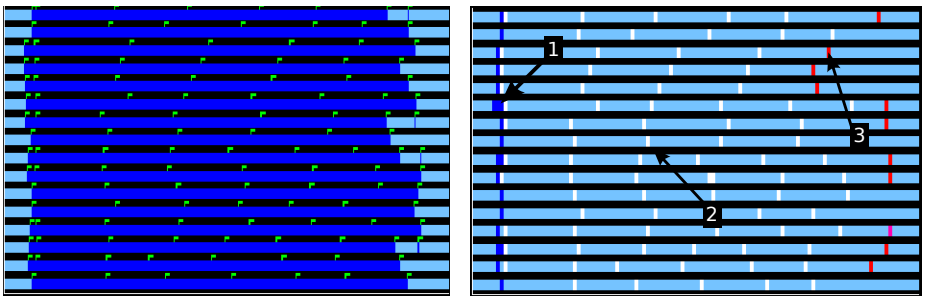
For this purpose the reduction manager object implements a list of TPRSs and two maps that point to individual items in that list. One map uses task identifiers while the other one uses target addresses (pointers to the original reduction variable) as primary keys.

At the first execution of a task of new reduction domain, a new TPRS is allocated for the current thread and the parent work descriptor identifier as well as the address of the reduction variable are stored in the corresponding maps. Each successive task running on that thread and reducing over same variable will receive the same TPRS because of matching addresses. Once tasks finish, and the recursion starts to collapse, only those tasks that have a matching task identifier stored in the map are allowed to reduce TPRS storages. This corresponds exactly to those tasks that created a new reduction domain.

4 Evaluation

The evaluation of the presented runtime support is based on three application kernels that include while-loop and recursive reductions. The first application, n-Queens, represents the satisfiability problem in numerical combinatorics. It computes the maximum number of different configurations of n queens on a chess board of size n. This application is implemented as a recursive backtracking algorithm in two versions. Once the reduction is performed over a global and

once over a task-local variable. This allows to demonstrate the reuse of thread-private memory across nesting levels in case of a global variable. The schematic concurrent code of this benchmark is shown in Figure 7. Execution traces obtained from n-Queens with a global reduction variable running on 16 threads, shown in Figure 10, illustrate task execution and different stages in the lifetime of TPRSs. In this execution the task granularity was set by using the *final* clause in the task pragma. This clause defines a cut-off value for task generation. Figure 11a shows the importance of task granularity where different final values substantially change execution times of the application.



(a) (b)

Fig. 10: Execution trace for the n-Queens application (n=11) showing tasks (a) and allocation (1), reuse (2) and reduction (3) of thread-private reduction storages (b)

For each benchmark

Figure 11b shows application scalability for the n-Queens application as well as for max-height and powerset. For benchmarks we have selected the best task granularity. Max-height computes the longest path over a directed, unbalanced graph. This application represent a while-loop reduction. Due to its frequent, irregular memory accesses, its scalability is limited by memory bandwidth. Powerset, computes the number of all possible sets over a given number of elements. This application is implemented recursively where unlike the n-Queens application, each recursive branch is of the same length. It scalability is currently limited as the baseline represents an optimized serial code whereas code optimization of the final block is currently not supported by the OmpSs compiler. Figure 11c shows speed-ups relative to code implementations using atomics or critical sections. While all application were executed with both allocation strategies and exhibited performance differences, a detailed analysis would exceed the scope of this paper. For this reason we defer further analysis of allocation strategies to future work.

Have the impression we need to extend evaluation... But I know we have no space...

4.1 Environment

All benchmark results presented in this work were obtained from the MareNostrum 3 supercomputer located at the Barcelona Supercomputing Center. Each

Also we need to talk about computation weight and runtime overhead, due we cannot avoid to ask for private storage at each recursive level, which is comparable (in this case) with th task computation.

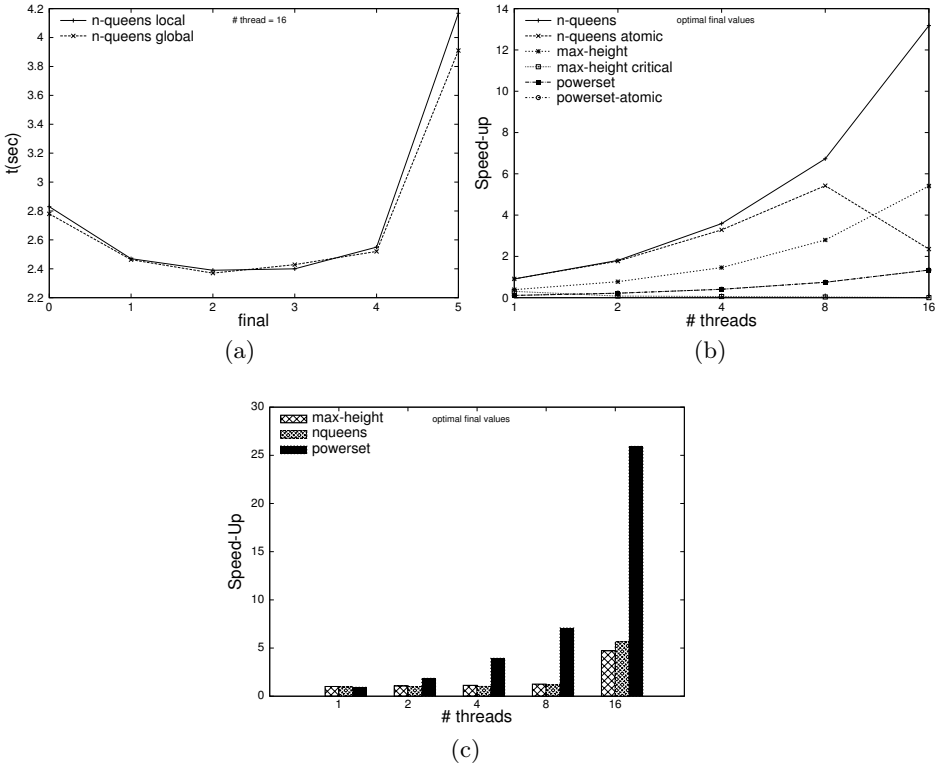


Fig. 11: Application performance depends on an task granularity which can be tuned through the use of the final clause (a) resulting in near-linear speed-up over serial execution (b) and a substantial speed-up over implementations with atomics or critical sections(c)

system node contains two 6-core Intel Xeon E5-2670 CPUs running at 2.6 GHz with 20MB L3 cache and 32GB of main memory. Applications were compiled using Mercurium C/C++ compiler v1.99 and GCC v4.8.2 back-end/native compiler with -O3. The runtime is based on the Nanos++ RTL v0.7a.

5 Related Work

OpenMP allows concurrent reductions in work-sharing constructs since its first specification (OpenMP 1.0) in 1997. The supported clauses allow the use of a reduction operator and a list of scalar, shared locations. During parallel execution, the runtime creates private copies for each list item and thread in the team. The result variable is initialized with the identity value according to the operator that is declared in the clause. In successive versions of the OpenMP specification, additional features have been added to the standard. These include min- and max-operators for C/C++ in version OpenMP 3.1, extended reduction sup-

port to Fortran Allocatable Arrays (OpenMP 3.0) and User Defined Reduction (OpenMP 4.0). Aside from small incremental updates, the OpenMP specification has never allowed OpenMP tasking in reductions. In fact it explicitly forbids the use of reduction symbols in combination with tasks: "A list item that appears in a reduction clause of the innermost enclosing work-sharing or parallel construct may not be accessed in an explicit task." (OpenMP 4.0, page 170, lines 28-29). This restriction reduces flexibilities achieved by dynamic parallelism in many algorithms.

See that you remove lot of RW due space limitation,...

6 Conclusions and Future Work

In this paper we presented an extension to OpenMP tasking to support reductions in while-loops and general-recursive functions. In turned out that the OpenMP taskgroup is well suited to support task-parallel reductions as it minimizes implications on unrelated constructs. A general support in OpenMP is possible but requires further analysis on induced overheads on barriers. This effort should be made in the future if applications exist that render the taskgroup construct insufficient. The presented runtime implementation offers two different allocation strategies and maximizes storage reuse. Dynamic allocation follows the idea of dynamic parallelism where neither the amount of work nor the number of participating threads is known beforehand. Results show that task granularity is important and in the case of recursive algorithms can be efficiently controlled by the final clause. In the case of while-loops, task granularity needs to be taken into account by the programmer through appropriate application design. Performance results obtained on a MareNostrum 3 system node show a near-linear speed-up for test cases with optimal granularity. Future work includes advanced runtime features to support array reductions through software caching [4], user-defined reductions and efficient execution on accelerators.

References

1. Barcelona Supercomputing Center: OmpSs Specification (April, 25th 2014), `http://pm.bsc.es/ompss-docs/specs`
2. BSC - Parallel Programming Models group: Mercurium c/c++ source-to-source compiler. (May 2014), `http://pm.bsc.es/projects/mcxx`
3. BSC - Parallel Programming Models group: Nanos++ runtime library (May 2014), `http://pm.bsc.es/projects/nanox`
4. Ciesko, J., Bueno-Hedo, J., Puzovic, N., Ramirez, A., Badia, R.M., Labarta, J.: Programmable and scalable reductions on clusters. p. 560–568. IEEE, Boston, United States (May 2013)
5. Komatitsch, D., Tromp, J.: Introduction to the spectral-element method for 3-D seismic wave propagation 139(3), 806–822 (1999)
6. OpenMP Architecture Review Board: OpenMP application program interface version 4.0 (July 2013)