This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

SAND2015-5576C

Towards task-parallel reductions in OpenMP

Jan Ciesko[1] Sergi Mateo[1,2] Xavier Teruel[1] Xavier Martorell[1,2]
Eduard Ayguadé[1,2] Jesus Labarta[1,2]
Alex Duran[3] Bronis R. de Supinski[4]
Stephen Olivier[5] Kelvin Li[6] Alexandre E. Eichenberger[6]

[1] Barcelona Supercomputing Center [2] Universitat Politècnica de Catalunya
[3] Intel Iberia Corporation [4] Lawrence Livermore National Laboratories
[5] Sandia National Laboratories [6] IBM Corporation
{jan.ciesko,sergi.mateo,xavier.teruel,xavier.martorell,
eduard.ayguade,jesus.labarta}@bsc.es
alejandro.duran@intel.com bronis@llnl.gov
slolivi@sandia.gov kli@ca.ibm.com alexe@us.ibm.com

Abstract. Reductions represent a common algorithmic pattern in many scientific applications. OpenMP* has always supported them on parallel and worksharing constructs. OpenMP 3.0's tasking constructs enable new parallelization opportunities through the annotation of irregular algorithms. Unfortunately the tasking model does not easily allow the expression of concurrent reductions, which limits the general applicability of the programming model to such algorithms. In this work, we present an extension to OpenMP that supports task-parallel reductions on task and taskgroup constructs to improve productivity and programmability. We present specification of the feature and explore issues for programmers and software vendors regarding programming transparency as well as the impact on the current standard with respect to nesting, untied task support and task data dependencies. Our performance evaluation demonstrates comparable results to hand coded task reductions.

Keywords: OpenMP, Task, Reduction, Recursion

1 Introduction

Migrating applications to multi-core and many-core architectures is a challenging but necessary step to achieve scalable performance on modern systems. Thus, parallel programming models such as *OpenMP* [7] have gained popularity through concepts and tools to introduce portable concurrency in a broad range of algorithms with relatively little programming effort. This work proposes a task redutcion extension to OpenMP that supports a yet wider class of algorithms.

A reduction is an iterative update of a variable *var*, defined as:

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

where *op* is an associative function and *var* does not occur in *expression*. Typically, a *for-loop* (bounded loop) or *while-loop* (unbounded loop) iteratively or recursively defines the iteration space.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

```
1 //Compute reduction by traversing nodes
2 float var = 0;
3
4     ...
5
6     while ( node ) {
7         var += node->value;
8         node = node->next;
9     }
10
11    ...
12
13
14 ...
```

(a) Original code (serial version)

```
1 //Compute reduction by traversing nodes
2 float var = 0;
3 #pragma omp parallel
4 {
5     #pragma omp single
6     while ( node ) {
7         #pragma omp task firstprivate(node)
8         {
9             #pragma omp atomic
10            var += node->value;
11        }
12        node = node->next;
13    }
14 }
```

(b) Parallel with atomics

```
1 //Compute reduction by traversing nodes
2 float var = 0;
3 float part[nthreads] = { 0 };
4
5 #pragma omp parallel reduction(+:var)
6 {
7     #pragma omp single
8     {
9         while ( node ) {
10            #pragma omp task \
11                    firstprivate(node)
12            {
13                part[thread_id] +=
14                        node->value;
15            }
16            node = node->next;
17        }
18    }
19    var += part[thread_id];
20 }
```

(c) Parallel with manual privatization

```
1 //Compute reduction by traversing nodes
2 float var = 0;
3 float part = 0;
4 #pragma omp threadprivate(part)
5
6 #pragma omp parallel reduction(+:var)
7 {
8     #pragma omp single
9     {
10        while ( node ) {
11            #pragma omp task \
12                    firstprivate(node)
13            {
14                part += node->value;
15            }
16            node = node->next;
17        }
18    }
19    var += part;
20 }
```

(d) Parallel with thread-privatization

Fig. 1: Different versions of a while-loop reduction over a linked list

For-loops have a constant iteration space. OpenMP supports their concurrent execution through worksharing constructs. The iterations space of while-loops and recursions is dynamic, which prohibits efficient use of worksharing constructs. OpenMP 3.0 added support for these irregular algorithms through the `task` directive. In this formulation, loop iterations and recursive calls create task instances of the enclosed code, typically the loop body.

While for-loops and while-loops can be efficiently parallelized through worksharing constructs or tasks, reductions within them require special attention. A closer look reveals that the reduction operation represents a read-modify-write sequence that is not atomic so that its parallel execution introduces data races.

Figure 1 shows while-loop reductions over a linked list that avoid data races by introducing locks or by applying techniques like thread-privatization. Programming model support would eliminate the required boilerplate code. Even though manual implementations are viable solutions, they are error prone and require the programmer to select a specific implementation, which may be inefficient on a given architecture or incur unnecessary memory overheads.

OpenMP needs a solution that supports task reductions and minimizes the effect on unrelated constructs. It should comprehensively define the scope of the reduction and a data context for the private reduction variable.

2 Related Work

OpenMP supports reductions on parallel and worksharing constructs through the reduction clause. It implies data privatization of the reduction variable that removes race conditions by replacing accesses to the original variable with accesses to per-thread private copies. Each copy is initialized with the operation's identity and is reduced to the original variable at the end of the construct.

While the specification does not yet support task reductions, prior work has explored them for OpenMP [3] and OmpSs [1]. These papers discussed different scenarios in which to use task reductions and compared the results with manual transformations that use atomics. This general approach could specify the task reduction scope through the *taskgroup*, *taskwait* or *barrier* constructs or task dependences on the reduction variable. This paper extends that work.

Intel(R) Cilk(TM) [5] coordinates the view of a variable of a task and its descendants through *hyperobjects* [4]. A reduction operation can combine these views when a descendant task finishes execution. This mechanism targets a multilevel task hierarchy. We target the task hierarchy within a taskgroup region.

The X10 [2] programming model supports task reductions through phaser-accumulators [8, 9]. Focused on the Partitioned Global Address Space environment, X10's *phaser-accumulators* can send and receive results from different activities and combine them in a point-to-point pattern.

3 Discussion

We propose to extend the *taskgroup* and *task* constructs to support task reductions. Prior work identified *taskgroup* construct as a possible scope of the reduction [3]. We prefer this choice since it does not affect other OpenMP mechanisms (e.g., barriers) and the taskgroup structured block defines a clear reduction scope.

We extend the *taskgroup* and *task* construct with the clauses *reduction* and *in_reduction* respectively. The *in_reduction* clause declares a task as a participant in the computation of *var* that was previously declared in an enclosing taskgroup *reduction* clause with the same *reduction-identifier*. We deliberately use the *in_reduction* clause instead of reusing the *reduction* clause in order to stress the differences in behavior to the programmer. The *reduction* clause in the *taskgroup* construct follows its current specification for other constructs. Alternatively, the *in_reduction* clause on a task construct defines an access pattern (an update operation) to one of those copies. Figure 2(a) illustrates our proposal for the previous example.

3.1 Updates of a reduction variable outside a reduction context

Programmers must consider that an update of the original reduction variable occurs just after the *taskgroup* region and that accesses to that outside of the taskgroup may create a race condition. Figure 2(b) shows code that updates the

```
1 //Compute reduction by traversing nodes
2 float var = 0;
3
4 #pragma omp parallel
5 {
6   #pragma omp single
7   #pragma omp taskgroup \
8             reduction(+:var)
9   while ( node ) {
10     #pragma omp task \
11             firstprivate(node) \
12             in_reduction(+:var)
13     {
14       var += node->value;
15     }
16     node = node->next;
17   }
18 }
19
20
21
22
23 ...
```

(a) While-loop reduction (tentative)

```
1 //Compute reduction by traversing nodes
2 float var = 0;
3
4 #pragma omp parallel
5 {
6   #pragma omp single
7   {
8     #pragma omp task
9     var++;
10
11    #pragma omp taskgroup \
12            reduction(+:var)
13    while ( node ) {
14      #pragma omp task \
15              firstprivate(node) \
16              in_reduction(+:var)
17      {
18        var += node->value;
19      }
20      node = node->next;
21    }
22  }
23 }
```

(b) While-loop reduction (race condition)

Fig. 2: Examples of our proposal

reduction variable both inside and outside a taskgroup reduction. The task created in line 8 can be executed concurrently with the taskgroup reduction update occurring at the end of the taskgroup created in lines 11 − 12. This situation may also occur when multiple taskgroup reductions are working with the same variable simultaneously. The programmer must provide proper synchronization to avoid this situation. This requirement is analogous to existing restrictions on reductions:

> To avoid race conditions, concurrent reads or updates of the original list item must be synchronized with the update of the original list item that occurs as a result of the reduction computation (line 20, p. 170 [7]).

3.2 Over-specifying the reduction identifier

The declaration of the reduction identifier in the *in_reduction* clause could be inferred from the *taskgroup* context and thus could be omitted to minimize the potential for programming errors. However, vendor feedback indicates that omitting the identifier could limit compiler optimizations, or at least introduce some additional overhead (i.e., registering the reduction inside the runtime) to perform these optimizations. OpenMP vendors may use the identifier to combine a local-copy of a reduction variable with the original/thread-copy (depending on the implementation approach), which specification of the identifier in the *in_reduction* clause would facilitate. Thus, we choose to require it.

3.3 Supporting untied tasks

Untied tasks can be suspended at a task scheduling point and later resumed on a different thread. Without proper handling, a task might resume execution on a different thread but still continue using the thread-private copy of the thread that started its execution, which could create a race condition. Tied tasks do

not encounter this issue since they execute entirely on one thread even if they are suspended at some point. Thus, they can safely use that thread's copy as they will not be suspended while accessing it.

Several solutions could support untied reduction tasks. First, an implementation could not migrate any task (e.g., treat it as tied) if it is involved in a reduction even though it is declared as *untied*. This approach is simple but eliminates the potential benefit of untied task migration.

Alternatively, an implementation could introduce an additional local variable for each untied reduction task. This task-local variable must be initialized to the identity. A reference to the local variable would replace all references to the reduction variable inside the untied task. Finally, at the end of the task, the partial result stored in the task-local variable would be combined with the thread-private copy of the thread that finalizes the task. This approach supports tasks that migrate among threads at the cost of an additional task-local copy that must be initialized and an additional partial reduction per untied task.

Finally, the compiler could generate a request for the thread-private copy after each possible task scheduling point, thus supporting the use of the thread-private copy. The reduction task would then always access the thread-private copy of the thread that is executing it. This approach supports tasks that migrate among threads at the cost of repeatedly obtaining the thread-private location.

We recommend that the following be implementation defined:

- Whether untied tasks involved in reductions can migrate;
- The number of private copies that are created for a task reduction.

The number of private copies could be defined as the number of tasks that participate in the reduction. Our recommendation thus allows an implementation to choose any of the above solutions (or a hybrid of them). Untied tasks could migrate and the number of private copies could be anything between the number of threads to the number of tasks.

Evaluating support for untied tasks: We use two benchmarks to evaluate the choice of supporting untied tasks by not migrating them or by introducing a new local copy per task. The first performs a reduction over a scalar. The performance of both versions is equivalent since the extra overhead introduced in the task-local approach is small in scalar reductions and the benchmark is well-tuned to obtain good performance using tasks so the extra overhead of the task-local version is insignificant compared to the task granularity.

Our second benchmark, Array Sum UDR (since it has a *User Defined Reduction*) reduces an array of structs to a unique struct. This struct has a static array of TS integers. The UDR's initializer sets every element of the struct to zero and its combiner adds the values of the two arrays. We choose this benchmark since it increases the cost to allocate and to initialize the extra copy and to perform its associated reduction.

Figure 3 shows the relative performance of the task-local version compared against the untied-as-tied version, with different number of threads and fixing

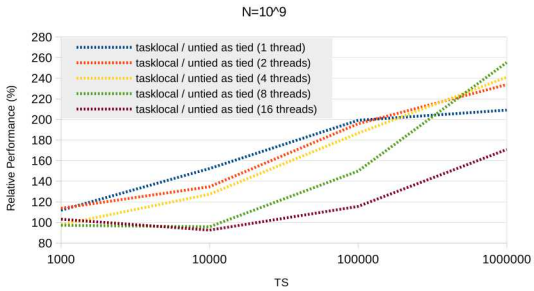


Fig. 3: Array Sum UDR benchmark results

```
1 int a = 0;
2 #pragma omp taskgroup reduction(+:a)
3 {
4     ...
5     int b = 0;
6     #pragma omp taskgroup reduction(+:b)
7     {
8         ...
9     }
10    ...
11    a += b;
12 }
```

(a) Nesting over two different variables

```
1 int a = 0;
2 #pragma omp taskgroup reduction(+:a)
3 {
4     ...
5
6     #pragma omp taskgroup reduction(+:a)
7     {
8         ...
9     }
10    ...
11
12 }
```

(b) Nesting over the same variable

Fig. 4: Nested taskgroup reduction scenarios

the total number of integers to $N = 10^9$. The relative performance is computed dividing the execution time of an approach by the execution time of another.

The overhead of the task-local version increases with TS, the size of the static array. The differences among the different relative perfomances require further analysis and it's deferred to future work. Thus, the task-local approach is reasonable for scalar reductions but may incur excessive overhead for array reductions or UDRs; implementations could define values based on the type of the reduction.

3.4 Supporting nested taskgroups

Nested taskgroup reductions can be defined either over different list items or the same ones, as Figure 4 shows. If the nested taskgroup defines a reduction over a different list item (Figure 4(a)), the runtime registers a new reduction that is independent of the ongoing outermost taskgroup reduction. Thus, the runtime creates a new set of thread-private copies to compute the reduction.

Two alternatives exist if the nested taskgroup reduction is over the same list item (Figure 4(b)). The first uses the same approach as when the list item is different: register a new reduction. The second alternative reuses the same set of private copies for both reductions. With this approach, we cannot reduce the private copies at the end of the nested taskgroups reductions: the final reduction must be computed at the end of the outer *taskgroup* region, counter to current reductions semantics that compute the reduction at the end of the construct that has the reduction clause.

3.5 Cancellation, dependencies and merged tasks

Cancellation implies the value of the reduction variable is unspecified since we cannot guarantee how far the computation of the reduction has progressed. The programmer must anticipate this behavior.

The specification of a dependency (using the task *depend* clause) over a reduction variable might introduce a conceptually misleading situation. The programmer might intend a dependency over the original variable or the private copy in the data context of the taskgroup reduction. We could explicitly restrict the use of the *in_reduction* clause and depend clause over the same variable. However the current OpenMP specification does not restrict similar cases. A dependency over a private variable produces a similar situation where the OpenMP specification does not provide clarification about the interaction between data-sharing attributes and dependencies.

A merged task that participates in a reduction does not have a data environment. Thus, it must use the parent's data environment that includes the private copy of the reduction variable. Since the parent environment for a reduction task can only be either a taskgroup reduction or another reduction task environment, the use of the corresponding private copy[1] in the parent region is always guaranteed. Thus, this case also does not require additional specification.

4 Syntax Additions

This section describes the syntax of our proposal. We update the syntax of the taskgroup construct to:

`#pragma omp taskgroup` *[clause[[,] clause]...] new-line*
structured-block

where clause is:

`reduction(`*reduction-identifier: list*`)`

We also modify the *reduction* clause description to cover taskgroup regions. Once the scope of a reduction is defined, we must identify tasks within the taskgroup that participate in the computation. Thus, we extend the clauses allowed on a task construct to include:

`in_reduction(`*reduction-identifier* `:` *list*`)`

We add a section for the *in_reduction* clause and modify the description of the *reduction* clause to specify the semantics of references to the list items that we discussed in the previous section. The section on the *in_reduction* clause includes this restriction:

[1] This case may involve multiple private copies due to support for untied tasks.

– The task to which the *in_reduction* clause is applied on a list-item must be closely nested in a *taskgroup* region to which a *reduction* clause is applied on the same list-item.

5 Evaluation

This section compares the performance of our prototype implementation of our proposed taskgroup reduction with manual implementations that Figure 1 shows.

5.1 System environment

We obtained our results on MareNostrum III and the Knight system located at the Barcelona Supercomputing Center. Each Marenostrum III node contains two 8-core Intel Xeon E5-2670 CPUs running at 2.6 GHz with 20MB L3 cache and 32GB of main memory organized as two NUMA nodes. Each Knight node includes an Intel Xeon Phi coprocessor with C0 silicon and board version C0PRQ-7120 (61 cores at 1238095 Khz, 16 GB of GDDR Memory at 5.5 GT/sec, 300W TDP), driver v3.4-1, MPSS v3.4 and flash v2.1.02.0390).

Applications on Marenostrum and Knight were compiled using the Mercurium source-to-source compiler v1.99.8[2] (using GCC v4.7.2 and Intel(R) C Compiler 15.0.2 as the back-end/native compiler respectively). In both cases the compiler optimization level was `-O3`, and the parallel runtime used in all experiments was based on the Nanos++ RTL v0.9a[3].

5.2 Benchmark descriptions

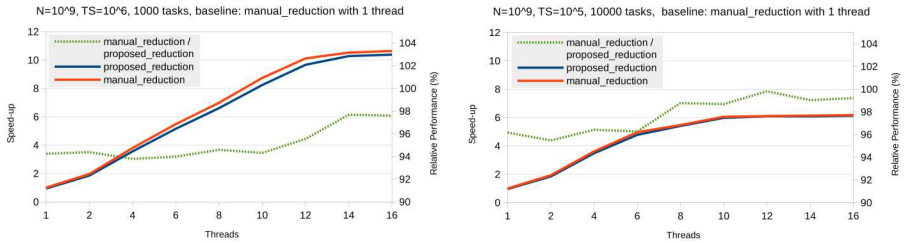
`Array Sum:` This algorithm takes a single array of N integers as an operand and computes the sum of its elements. We create a task for each TS elements.

`Dot Product:` The dot product algorithm is a simple operation on two vector operands of N elements. The result is the sum of the products of their components. We create a task for each TS elements.

`NQueens:` This application computes the number of placements of N chess queens on a NxN chessboard such that none of them can attack any other. This implementation uses a Branch and Bound algorithm following a recursive pattern, taskified and using the final clause to control task granularity.

`Unbalanced Tree Search (UTS):` This benchmark computes the number of nodes in an implicitly defined unbalanced tree [6]. The program begins with a single tree node and an initial seed that is used to generate a sequence of

[2] mcxx 1.99.8 (git 538d492)
[3] nanox 0.9a (git master 10f6134)



(a) Array Sum (b) Dot Product

Fig. 5: Array Sum and Dot Product benchmarks results

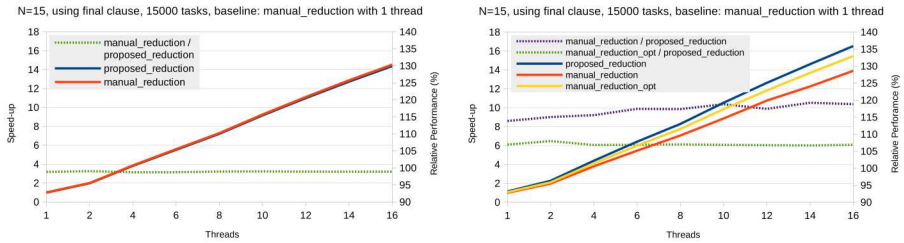
pseudo-random numbers. For each node, the next value in the sequence is used to sample a parameterized probablity distribution to determine the number of children for a given node. This algorithm creates an unpredictably unbalanced workload that makes the use of a cut-off value in the final clause difficult.

5.3 Performance results on Intel Xeon processors

In this section we evaluate the performance of our proposal against the performance of manual versions of the benchmarks on Intel Xeon processors.

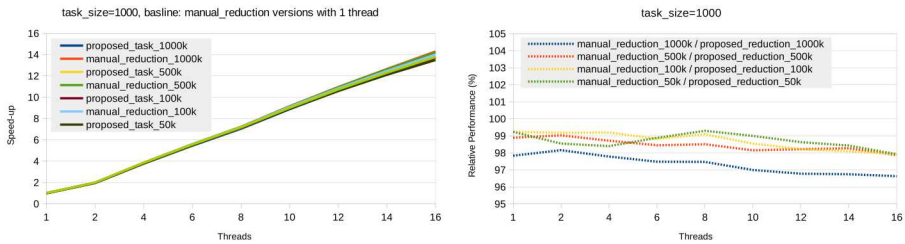
Figure 5 shows the performance results of the Array Sum and Dot Product benchmarks. Both benchmarks exhibit similar behavior in which performance drops levels off with higher thread counts. In this case, scalability is limited by memory bandwidth. In Array Sum, bandwidth saturation starts with 12 threads (with a 10x speed-up), while for Dot Product this effect becomes visible with 6 threads (reaching a speedup of 5x). These two different phases (scale and saturate) have a counterpart in the relative performance (the green dashed line in the figure). For all thread counts with Array Sum, the performance reaches at least 94% of the performance of the manual version. For larger thread counts, the differences between the implementations become smaller because task execution time shifts towards the computation as the algorithm saturates the memory bandwidth and reduces the importance of reduction performance. For the Dot Product benchmark, the relative speedup is between 95% and 100%. For both benchmarks, gains in maintainability and portability easily compensate for the slight differences in relative performance.

Figure 6 shows the results for the NQueens benchmark. For this application we have implemented two versions: one that reduces over a global variable (subfigure a) and another that reduces over a local variable (subfigure b). We explore these two versions primarily because the global version only registers one reduction in the whole program while the local version registers a new reduction at each recursive level. When reducing over a global variable, speedup is essentially linear and relative performance is close to 100%. When the reduction is performed over a local variable, we compare our proposal against two different manual versions. The first one is the regular transformation presented



(a) NQueens Global (b) NQueens Local

Fig. 6: NQueens benchmark results



(a) UTS, Scalability (b) UTS, Relative Performance (%)

Fig. 7: Unbalance Tree Search benchmark results

previously whereas the second version optimizes the code when in a final task. The problem with the regular transformation is that we are still allocating, intializing and reducing an array of $NUM_THREADS$ elements even if we are going to use just one element. Thus, the optimized versions makes use of the *omp_in_final()* runtime service to avoid this extra overhead. Despite comparing our proposal against the manual optimized version, the scalability and the relative performance of our version is still better.

Figure 7 shows the results of executing the UTS benchmark with configurations that vary the number of created tasks from 50k to 1M tasks. All configurations achieve essentially linear speedup (subfigure a), and relative performance is between 96% and 99% for programmability issues again more than compensate.

5.4 Performance results on Intel Xeon Phi coprocessors

In this section we evaluate the performance of our proposal against the performance of manual versions on a Intel Xeon Phi coprocessor.

Figure 8 shows that our approach scales slightly better than the manual version of Array Sum. The relative performance line shows that the performance of our proposal is at least 10% better than the performance of the manual version in almost all cases. While not shown in the figure, the exceptions is when we use

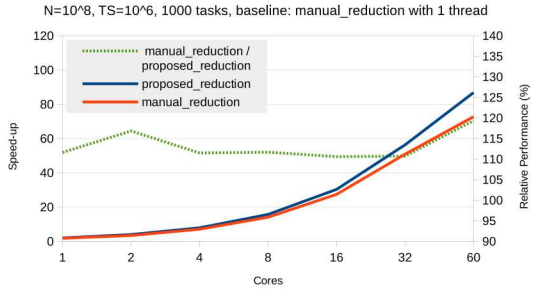
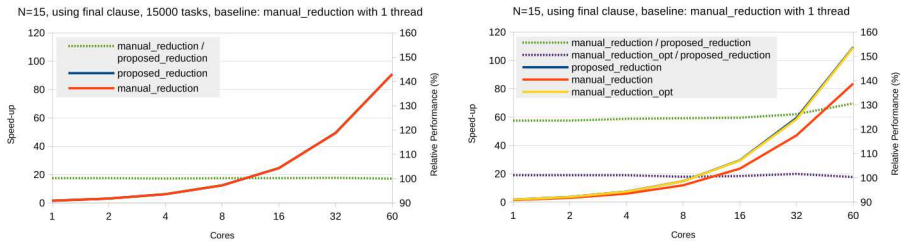


Fig. 8: Array Sum benchmark results on Xeon Phi



(a) NQueens Global (b) NQueens Local

Fig. 9: NQueens benchmark results on Xeon Phi

all 60 cores and more than 2 threads per core, in which case our approach underperforms and does not scale well due to cache problems and more contention when we increase the number of threads.

Figure 9 shows the results of the NQueens benchmark on the Xeon Phi. For the global version of the NQueens, the scalability and the relative performance between our approach and the manual version are identical. For the local version, the scalability and the relative performance of our proposal is equivalent to the manual optimized version and far better than the nonoptimized one.

6 Conclusions and Future Work

In this paper we have presented a proposal to support task-parallel reductions in OpenMP that extends the *taskgroup* and *task* constructs with *reduction* and *in_reduction* clauses. We find that the *taskgroup* construct provides a convenient data environment for reductions and the scope of the reduction is clearly defined by the deep synchronization at the end of the *taskgroup* region. The *in_reduction* clause for the task construct associates tasks with a reduction declared in a *taskgroup* construct. This approach does not impact barriers or other task synchronization constructs. We explored implementation options to support nested taskgroups and untied tasks, which demonstrate that implementors

can explore a range of implementations and optimizations. Our performance results demonstrate that the approach incurs little overhead compared to manual versions currently required and it may provide small performance benefits in some specific cases like recursive benchmarks. Most importantly, it significantly reduces boilerplate code that programmers must currently use to implement reductions manually.

In the future, we continue our work in this area by conducting more analysis and evaluation. Apart from that, we plan to provide a draft of the OpenMP specification to the OpenMP committee.

7 Acknowledgments

This work has been developed with the support of the grant SEV-2011-00067 of Severo Ochoa Program, awarded by the Spanish Government and by the Spanish Ministry of Science and Innovation (contracts TIN2012-34557, and CAC2007-00052) by the Generalitat de Catalunya (contract 2009-SGR-980) and the Intel-BSC Exascale Lab collaboration project.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Also the authors would like to thank the OpenMP community for their substantial contribution to this work.

Intel, Xeon, Xeon Phi and Many Integrated Core are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
*Other brands and names are the property of their respective owners.

References

1. Barcelona Supercomputing Center: OmpSs Specification (April, 25th 2014), `http://pm.bsc.es/ompss-docs/specs`
2. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An Object-oriented Approach to Non-uniform Cluster Computing. SIGPLAN Not. 40(10), 519–538 (October 2005)
3. Ciesko, J., Mateo, S., Teruel, X., Beltran, V., Martorell, X., Badia, R.M., Ayguadé, E., Labarta, J.: Task-Parallel Reductions in OpenMP and OmpSs. In: 10th International Workshop on OpenMP, IWOMP 2014. p. 1–15. Springer, Springer, Salvador de Bahia, Brazil (Sep 2014)
4. Frigo, M., Halpern, P., Leiserson, C.E., Lewin-Berlin, S.: Reducers and Other Cilk++ Hyperobjects. In: Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures. pp. 79–90. SPAA'09, ACM, New York, NY, USA (2009)

5. Leiserson, C.E.: The Cilk++ Concurrency Platform. In: Proceedings of the 46th Annual Design Automation Conference. pp. 522–527. DAC '09, ACM, New York, NY, USA (2009)
6. Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., Tseng, C.W.: UTS: An Unbalanced Tree Search Benchmark. In: Almási, G., Cascaval, C., Wu, P. (eds.) LCPC 2006: Proceedings of the 19th International Workshop on Languages and Compilers for Parallel Computing. LNCS, vol. 4382, pp. 235–250. Springer (2007)
7. OpenMP Architecture Review Board: OpenMP Application Program Interface Version 4.0 (July 2013)
8. Shirako, J., Peixotto, D.M., Sarkar, V., Scherer, W.N.: Phasers: A Unified Deadlock-Free Construct for Collective and Point-to-Point Synchronization. In: ICS 08: Proceedings of the 22nd annual international conference on Supercomputing. pp. 277–288. ACM, New York, NY, USA (2008)
9. Shirako, J., Peixotto, D.M., Sarkar, V., Scherer, W.N.: Phaser Accumulators: A New Reduction Construct for Dynamic Parallelism. In: Parallel and Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on. pp. 1–12. IEEE, Rome, Italy (May 2009)