Evaluating the Impact of OpenMP 4.0 Extensions on Relevant Parallel Workloads

Raul Vidal, Marc Casas(✉), Miquel Moretó, Dimitrios Chasapis, Roger Ferrer, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, and Mateo Valero

Barcelona Supercomputing Center (BSC),
Universitat Politècnica de Catalunya (UPC), Barcelona, Spain
marc.casas@bsc.es

Abstract. OpenMP has been for many years the most widely used programming model for shared memory architectures. Periodically, new features are proposed and some of them are finally selected for inclusion in the OpenMP standard. The OmpSs programming model developed at the Barcelona Supercomputing Center (BSC) aims to be an OpenMP forerunner that handles the main OpenMP constructs plus some extra features not included in the OpenMP standard. In this paper we show the usefulness of three OmpSs features not currently handled by OpenMP 4.0 by deploying them over three applications of the PARSEC benchmark suite and showing the performance benefits. This paper also shows performance trade-offs between the OmpSs/OpenMP tasking and loop parallelism constructs and shows how a hybrid implementation that combines both approaches is sometimes the best option.

1 Introduction and Motivation

OpenMP has been for many years the most popular programming model for shared memory architectures. The OmpSs programming model [5] developed at the Barcelona Supercomputing Center aims to be an OpenMP forerunner that handles the main OpenMP constructs plus other features not included in the OpenMP standard. OmpSs is based on `#pragma` annotations and its semantics are almost identical to the OpenMP standard. For these reasons, a code in OmpSs that uses only the features included in the OpenMP standard is equivalent to its OpenMP counterpart. It is not straightforward to make the choice on which OmpSs features should be adopted by the OpenMP standard and how these new features would interact with the already existing ones.

This paper brings some light to the above mentioned dilemmas by pursuing two goals: The first is to show the usefulness of three OmpSs features not currently handled by OpenMP 4.0 by using them to accelerate three well known applications of the PARSEC benchmark suite [3,4]. Secondly, this paper shows performance trade-offs between the OmpSs/OpenMP tasking and loop parallelism constructs (e.g. `#pragma omp for`) and proposes a hybrid implementation that combines both kinds of constructs to maximize performance. More precisely, this paper deploys the following OmpSs features:

© Springer International Publishing Switzerland 2015
C. Terboven et al. (Eds.): IWOMP 2015, LNCS 9342, pp. 60–72, 2015.
DOI: 10.1007/978-3-319-24595-9_5

- the multi-dependencies feature, which allows to specify different data-dependence scenarios in a single `#pragma` annotation, significantly increasing programmability.
- runtime support for NUMA-aware scheduling of tasks, which schedules them on the cores closest to the data the task accesses.
- the concurrent clause, which relaxes task synchronization and allows increased overlap of task creation with remaining computations.

Three applications of the PARSEC benchmark suite are considered in this paper: Facesim, Fluidanimate and Streamcluster. New OmpSs versions of these applications are used to show the potential of the new features. The concurrent clause is applied to Facesim and Fluidanimate to reduce synchronization penalties. The multi-dependencies clause is deployed in the Fluidanimate code to express complex data-dependencies that allow barrier removal without increasing the programming burden. Runtime support for NUMA-aware scheduling is deployed in the Streamcluster code. Finally, the performance trade-offs between tasking constructs and simpler forms of loop parallelism are analyzed in the Facesim code.

The rest of this paper is organized as follows: Sect. 2 describes the three applications studied in this paper and the proposed parallelization strategies, Sect. 3 presents the evaluation in terms of performance and programmability, while Sect. 4 describes the related work. Finally, Sect. 5 summarizes the main conclusions of this work.

2 Application Parallelization

2.1 Facesim

Description. Facesim animates a human face by simulating its movements. It employs a 3D model composed of a tetrahedral mesh representing the flesh of the face and two triangulated surfaces which model the bones of the head: the cranium and the jaw. The physical forces and motions in the model are computed frame by frame to produce the animation. Facesim uses the Newton's method for solving the system of equations that models the motion. The system is stored in a sparse matrix formed by two one-dimensional arrays: `dX_full` and `R_full`, defining the left-hand and right-hand sides of the equation system, respectively. The total number of nodes is equivalent to the arrays' size. The nodes are the vertices of the tetrahedrons the mesh is composed of. Each tetrahedron shares the nodes with its neighbors and for each node the force contributions are computed. A parallel conjugate gradient method is used in each step of the Newton's method to solve its associated linear system and find the displacement of the nodes in the current frame which is added to a separate array storing the current positions of the nodes.

PARSEC Pthreads Parallelization. In the Pthreads parallelization provided by the PARSEC benchmark suite, the mesh is split into a number of partitions

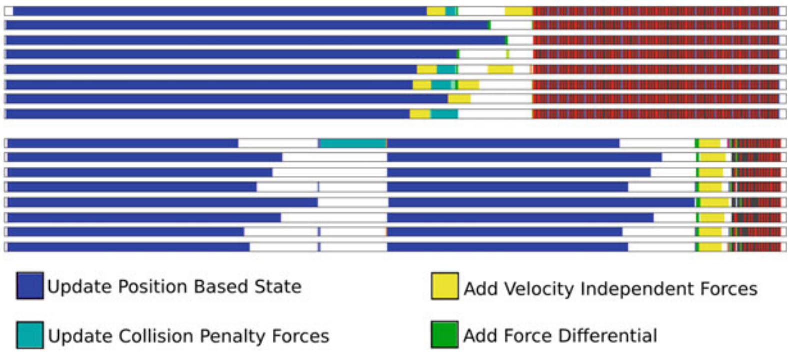

Fig. 1. Facesim UPBS, UCPF, AFD and AVIF parallel execution using the same time scale. Beginning of a frame. The OmpSs trace (top) exhibits no barriers. The original Pthreads trace (bottom) makes extensive use of barriers and UCPF routine is serialized.

equal to the number of threads available. It has a queuing system in which work units are queued to be processed by the team of threads the system spawns upon initialization. There is a master thread which executes the code of the application. When it reaches a parallel region, it calls the queuing system to create work units in a loop and waits in a barrier outside of the loop for the team of threads to finish. The work units are created by means of an ad-hoc scheduling library written in C which manages the team of threads.

Facesim's parallel computations are grouped in three major parallel kernels. Two of them generate the linear system associated with each iteration of Newton's method and the third one solves it.

- *Update State*: Updates velocities, force directions and material properties which depend on the current positions of the mesh. Update State is computed with two functions: Update Position Based State (UPBS) and Update Collision Penalty Forces (UCPF).
- *Add Forces*: Comes after Update State. Computes force contributions for each node. This kernel is actually computed with two functions: Add Velocity Independent Forces (AVIF) and Add Force Differential (AFD).
- *Conjugate Gradient (CG)*: This iterative method is set up to do a maximum of 200 iterations. The CG methods performs two reduction operation per iteration.

UPBS and CG are the most time consuming routines. There are several barriers in this application per iteration of Newton's method: One at the end of *Update State*, two from *Add Forces* and three within each *CG* iteration.

Taskification Strategy. With respect to Facesim we consider three different approaches. The first one exclusively uses tasking clauses with dependencies when necessary. The second one uses loop parallelism clauses, like the `omp for` construct. Finally, the third combines task and loop parallelism.

```
for each partition
#pragma omp task depend(in:variable)
    taskfunction1();

#pragma omp task depend(inout:variable)
    faketask();

for each partition
#pragma omp task depend(in:variable)
    taskfunction2();
```

Fig. 2. An additional task is used to create an anti-dependency. This is in fact a synchronization point since the `taskfunction2` tasks run after all the `taskfunction1` finish.

```
for each partition
#pragma omp task concurrent(variable)
    taskfunction1();

for each partition
#pragma omp task in(variable)
    taskfunction2();
```

Fig. 3. The `concurrent` clause is equivalent to an `inout` dependency on *variable*, but allows the tasks to operate concurrently on it.

The taskification concerning the first two phases of Facesim, *Update State* and *Add Forces*, is achieved by removing barriers and expressing control dependencies between the different subroutines. Such control dependencies are expressed by using a data dependency on a sentinel variable. As such, once the task that has the sentinel as an output parameter finishes, it passes the control flow to tasks that have the same sentinel as an input. In the *Update State* phase, UPBS and UCPF subroutines run concurrently and a task is generated per domain partition. With respect to the *Add Forces* phase, AFD and AVIF subroutines concerning a particular partition start right after the UPBS task operating over that same partition has finished. This is expressed by using task dependency semantics in OmpSs/OpenMP 4.0, removing a barrier synchronization from the original code. With respect to the implementation that uses the `#pragma omp for` construct, it mimics the Pthreads parallelization and uses barrier synchronization to handle parallelism. Figure 1 compares the parallel execution of these two phases in the original code (trace at the bottom) and the taskified code (top). All barriers are removed in the latter case, allowing subroutines to overlap and, as a consequence, the CG iteration starts much earlier. Also, thanks to specifying data dependencies, the UCPF routine is not serialized in the taskified version of the code.

With respect to the third phase of Facesim, *CG*, the tasking OmpSs/OpenMP versions contain specific code to relax the synchronization points and allow some degree of overlap between task creation and computation. In case of OpenMP, we add an additional task to create an anti-dependency to make sure the synchronization is respected while task creation is overlapped with it. In Fig. 2 we show how this approach is implemented. Although there are features in OpenMP 4.0

that allow alternative implementations, like the `taskgroup` construct, they can be used to implement a synchronization point but not to overlap task creation with synchronization. In the case of OmpSs we use the `concurrent` clause which is equivalent to an `inout` dependency, but allows tasks to operate concurrently on this data dependency. Figure 3 shows how the `concurrent` clause is used. Tasks that have an input or output dependency on *variable* respect it and do not overlap their execution with the concurrent tasks.

The implementation that uses loop parallelism adds the corresponding `#pragma omp parallel for` construct and uses `static` scheduling. A global parallel region for the CG iterations wraps the external loop. Inside of it, a `single` construct is used to update variables after the three parallel loops of each CG iteration.

Finally, in the hybrid approach, loop parallelism is used to handle the fine grain parallelism required by the *CG* phase, while the parallelism required by other routines is expressed in terms of tasks, as this combination showed the best performance results. Each one of these three approaches is implemented using OpenMP 4.0 and OmpSs, which means that we have 6 different version of Facesim in addition to the baseline Pthreads code.

2.2 Fluidanimate

Description. This application simulates incompressible fluid interactive animation, using the Smoothed Particle Hydrodynamics (SPH) method [11]. Each iteration of Fluidanimate involves running 8 different routines which are responsible for actions like rebuilding the spatial index, computing fluid densities and forces at given points, handling fluid collisions or updating particle locations.

Original Parallelization. The fluid surface is partitioned into N segments and there is one thread per segment. N is equal to the number of cores the application runs on. The kernels are parallelized and separated by barriers. When a particular thread runs a particular kernel, it takes care of all the computations involving its grid segment. For each iteration of the algorithm, the Pthreads implementation requires 8 barriers to make sure the execution of each kernel starts once the previous kernel computations have finished. That is required because each thread needs the previous kernels' computations on its grid segment and its neighbors to be finished once the execution of the new kernel finishes. Threads may have to update values belonging to neighbor segments, which requires the use of locks to avoid data races.

Taskification Strategy. Several different taskification strategies are considered: *OmpSs Trivial*, *OmpSs Finer Task*, *OmpSs Multi-Dependencies* and *OmpSs without Barriers*.

The *OmpSs Trivial* task-based implementation follows the same approach as Pthreads. Every time the application starts a new iteration, a task is created for each kernel and segment. Since the kernels are separated by barriers, only tasks related to the same kernel are allowed to run concurrently. Accesses to foreign grid segments are controlled by locks.

OmpSs Finer Tasks: The main difference between this strategy and the *OmpSs Trivial* consists in the number of tasks created. In the trivial version, a single task is created for each kernel and segment, meaning that a maximum of N tasks, N being the number of partitions, can run concurrently. For the *OmpSs trivial* version, N is equal to the number of cores the application runs on. In case of the *OmpSs Finer Tasks* implementation, we increase the number of segments to four times the number of cores. By doing this, we split the work into four times more pieces than the previously presented versions, which implies that the OmpSs runtime has more flexibility to balance the load between two barriers.

```
if (segment in corner)
#pragma omp task in( neighborhood[0], ..., neighborhood[3] )
//Task Code
else if (segment in boundary)
#pragma omp task in( neighborhood[0], ..., neighborhood[5] )
//Task Code
else if (internal segment)
#pragma omp task in( neighborhood[0], ..., neighborhood[8] )
//Task Code
```

Fig. 4. Fluidanimate code handling multiple dependency scenarios by using one #pragma per scenario.

```
#pragma omp task in( { neighborhood[j] , j=0:neighborhood.size() } )
//Task Code
```

Fig. 5. Fluidanimate code where multiple dependency scenarios are handled by a single #pragma annotation.

```
#pragma omp task dependence_type ( { item_list[j], j=0:item_list.size() } )
//Task Code
```

Fig. 6. Generic #pragma annotation with multi-dependencies. The dependencies are defined over a list of items, which has a dynamically defined size.

OmpSs multi-dependencies: This strategy consists of removing all barriers between the 8 different routines of each iteration. For each routine and partition we generate a set of tasks and we specify dependencies between them to make sure the previous routine has finished its pass over a segment and its neighbors when a task starts operating over this particular segment. The number of task dependencies is defined by its segment's position over the grid. If the segment is located on one of the four corners of the square grid, the total number of task input dependencies is 4. If the segment is located at the border, the dependencies are 6 and if it is an internal segment, its corresponding task has 9 input dependencies. Figure 4 shows the code required in OpenMP 4.0 to handle this scenario where the number of dependencies is variable. Of course, a `#pragma omp task` annotation is required in each case, implying that 3 different annotations are required for each of the 8 different routines each iteration of Fluidanimate is composed of, which ends up increasing the number of pragma annotations to 24.

To avoid such programming hardship, OmpSs has support to handle this complexity using a single high-level pragma annotation. In Fig. 6 there is generic #pragma annotation with multi-dependencies in OmpSs. The dependencies are defined over a list of items, which has a dynamically defined size. Figure 5 illustrates how the multi-dependency feature is used in the Fluidanimate source code. The only requirement is to generate a data-structure for each segment that lists all the neighbors. The size of this data structure changes depending on the number of neighbors and it is used to figure out the number of dependencies at runtime. The number of tasks considered by the *OmpSs multi-dependencies* strategy is the same as *OmpSs Finer Tasks*.

OmpSs without Barriers: This strategy includes all the improvements of the *OmpSs Finer Tasks* and the *OmpSs multi-dependencies* techniques plus the removal of the barrier between different iterations. Since computations of different iterations cannot be overlapped, the barrier between iterations is replaced by a `concurrent` clause, as is done in Facesim between the different CG iterations.

2.3 Streamcluster

Description. Streamcluster solves an online clustering problem. It takes a stream of points and then groups them in a predetermined number of centers. The program spends up to 90 % of the time in a function called `Pgain`, where points are assigned to existing centers using the Euclidean distance. Also `Pgain` calculates whether opening a new center is advantageous or not. If opening the new center lowers the cost of the current clustering, then the center is opened and points that are closer to this center than to previously created centers are reassigned to the new center. `Pgain` is executed a predefined number of iterations, obtaining new centers.

Original Parallelization. The Pthreads parallelization is very simple: the large array containing all the points to cluster is broken into chunks of constant size (200,000 points in our experiments). Each chunk is then processed in parallel in a number of partitions equal to the number of threads. A barrier synchronization is added to make sure that all threads finished processing all the points before a new chunk is processed. Streamcluster provides its own barrier implementation to synchronize threads. Once all the chunks of the stream of points are processed, a final pass to cluster the centers found on the different chunks is done. Streamcluster is a memory intensive application as it continuously reads data from memory. In the original parallelization, the data structures that store these points are allocated before creating the different threads and reused in each chunk processing. As a consequence, this application suffers scalability difficulties in NUMA machines.

Taskification Strategy. In the case of Streamcluster we develop two tasking versions, one in OmpSs and the other in OpenMP. We focus on the function

`Pgain` of this code as the program spends the majority of its execution time in it. While the Pthreads version of `Pgain` makes use of a dynamically allocated array per thread to store the partial cost computations and performs a reduction of all these costs over a global array after the parallel work, the tasking implementation does not need the global array and uses a local one per task. With `atomic` synchronization the local arrays of costs are updated. These changes simplify the code and minimize the time spent in index table computations.

Also, additional changes are made in the OmpSs code to taskify memory allocation and exploit the NUMA aware scheduling that the OmpSs runtime system performs for systems with multiple sockets. This scheduler tries to ensure that tasks execute in the sockets where their data structures have been allocated, reducing the cost of accessing memory. To do so, a few API calls to schedule tasks in specified NUMA spaces are added to the code. Figure 7 depicts how to use this API. OpenMP 4.0 has some environment variables to specify either on which cores the threads should be placed (`OMP_PLACES`) or whether threads can be moved between cores (`OMP_PROC_BIND`), however it does not have the feature of doing that in a per task basis.

```
for each partition p
{
nanos_current_socket(socket of partition p) ; //API call
#pragma omp task out(dest[k1:k2],points[k1:k2])
//initialization task
nanos_current_socket(socket of partition p); //API call
#pragma omp task in(points[k1:k2])
//task accessing a Point
}
```

Fig. 7. The NUMA-aware scheduling API specifies the socket where tasks run. In this way, the programmer can force tasks to run in the socket where the data they access is allocated.

3 Evaluation

The evaluation is performed on an IBM System X server iDataPlex dx360 M4, composed of two 8-core Intel Xeon E5-2670 processors at 2.6 GHz with 20 MB of shared last-level cache and with hyperthreading disabled. There is 32 GB of DDR3 RAM at 1.6 GHz.

The OpenMP implementation used is the GNU OpenMP (GOMP) included with gcc 4.9.1. We also used the OmpSs programming model [5] and its associated toolflow: Nanos++ runtime system (version 0.7.5), Mercurium source-to-source compiler (version 1.99.6), and gcc 4.9.1 as the back-end compiler. To analyze the behavior of the benchmarks, we used the Extrae instrumentation package (version 2.5) and the Paraver trace viewer (version 4.5) [10].

3.1 Performance Evaluation

While the PARSEC benchmark suite provides different input sets, the experiments shown in this paper make use of the largest set, the 'native' input. All the benchmarks are executed with 1, 2, 4, 8 and 16 threads, mapping one thread per core. Figure 8 shows the measured speedups of all the applications and strategies considered. The speedups are computed taking the execution time of the Pthreads implementations of Facesim, Fluidanimate and Streamcluster available in the PARSEC benchmark suite running on 1 thread. The left hand side of Fig. 8 shows the speedups of each one of the 7 parallelization strategies considered for Facesim: Pthreads, loop parallelism using OpenMP and OmpSs, tasking using OpenMP and OmpSs and the hybrid approaches combining tasking and loop parallelism. The best performing version is OmpSs Hybrid, which shows a speedup of 11.4x when run on 16 cores, closely followed by OpenMP Hybrid and the OmpSs and OpenMP loop parallelism strategies which have a speedup of 10.7x, 10.7x and 10.9x. The two parallel strategies that exclusively use a tasking approach show a speedup of 9.8x and 9.4x when run on 16 cores, significantly less than the hybrid and loop parallelism approaches. The hybrid approaches are the most well suited as they combine the benefits of barrier substitution by task dependencies, the low overheads of loop parallelism when tasking provides no benefit and the locality of the static scheduling performed by the CG routine.

In case of Fluidanimate, results are shown at the center of Fig. 8. The Pthreads and the *OmpSs trivial* versions have identically poor performance, achieving speedups below 8x when they run on 16 cores. If the granularity of the tasks is reduced, the speedup reaches 8x when run on 16 cores. The *OmpSs Multi-dependencies* strategy of removing all the barriers that separate the 8 internal routines of each iteration and replacing them by task dependencies provides significant benefits and allows the speedup to be slightly above 9x when 16 cores are used. Finally, if the barrier that separates the different iterations is removed, the application scales up to 10.1x on 16 cores.

Streamcluster performs similarly on all of its versions when 1, 2, 4 and 8 cores are considered. When the two 8-cores sockets are used, NUMA effects bring load

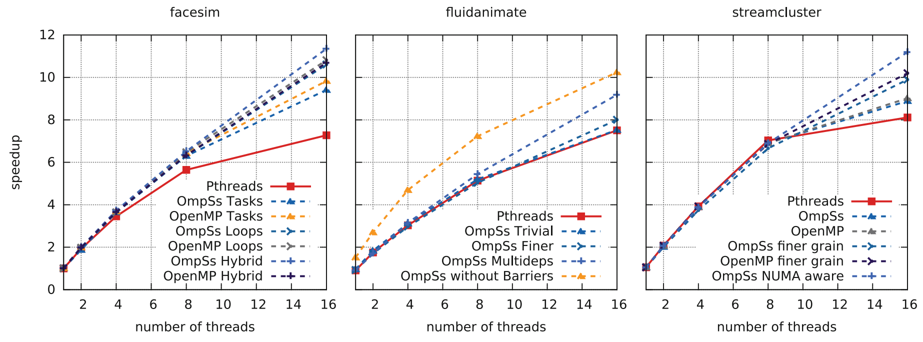

Fig. 8. Speedups of the different benchmarks and their tested versions

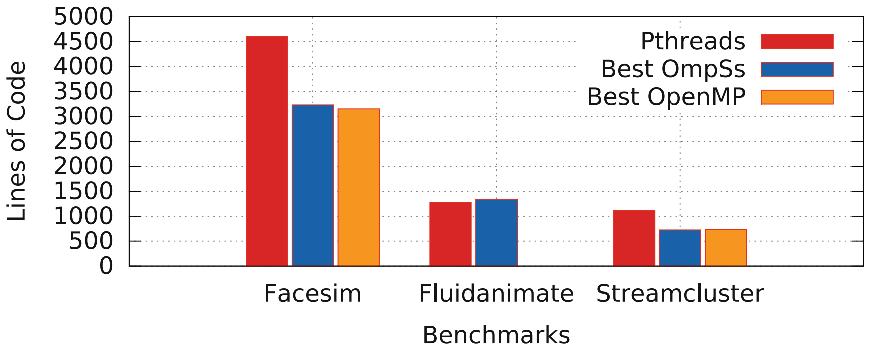

Fig. 9. Lines of code of the different benchmarks and their different versions

imbalance, which undermines the performance of the Pthreads implementation. The OpenMP and OmpSs implementations partially correct this load imbalance and achieve a speedup close to 9x on 16 cores. These load balancing benefits increase if finer grain tasks are considered, achieving scalabilities close to 10x on 16 cores. The fine grain versions make use of 5 tasks per thread, while the original OpenMP/OmpSs version use just 1 task per thread. Finally, the NUMA aware scheduling feature of the OmpSs runtime system provides further improvements reaching a speedup of 11.1x.

3.2 Programmability

Ease of use, portability and versatility are of paramount importance when deciding whether to use a programming model or not. It is difficult to quantify the above statement, but we can provide some insight on how easy it is to use such task-based models compared to Pthreads in terms of lines of code (LOC). LOC for the selected benchmarks is as follows: Facesim has 35,000 LOC, Fluidanimate 3,000 LOC, and Streamcluster 1,500 LOC.

Figure 9 shows the LOC of our task-based implementations compared to the original Pthreads implementations considering only files that are relevant to the parallel implementation, i.e. files that contain calls to Pthreads or task invocations, atomic primitives, etc. In this case, we only show the LOC of the best performing version of our OmpSs and OpenMP codes. The other versions have very similar number of LOC, with less than 3.5 % variation with respect to the best performing one.

On one hand, using OpenMP/OmpSs to parallelize applications allowed to reduce the size of the original code base in the case of Facesim (25 % less LOC) and Streamcluster (20 % less LOC). This is achieved by means of removing unnecessary barrier implementations and thread scheduling facilities. It also allowed to express more parallelism in all applications, whether allowing to parallelize originally sequential sections or by allowing more tasks to run concurrently.

This is the case with Fluidanimate, where a more advanced parallelization strategy is performed without significantly increasing the number of LOC (less than 4 %).

On the other hand, sometimes specifying dependencies might not be easy depending on the accessed data structure. For example, irregular and dynamic data structures are difficult to handle with current data dependencies. Also, very fine-grain tasks and an excess of dependency annotations can cause performance degradation due to runtime overheads. Designing future architectures driven by the runtime of the target parallel applications can be a suitable solution to reduce some of these overheads [12].

4 Related Work

In this paper we apply several parallelization strategies available in OpenMP 4.0 and OmpSs to three applications of the PARSEC benchmark suite. Similarly, the KASTORS suite [13] uses the OpenMP 4.0 task dependency constructs to extend the Cholesky and QR decompositions from the PLASMA library [9]. Also, the KASTORS suite provides a parallelized Poisson equation based kernel and extends the SparseLU and Strassen benchmarks from the Barcelona OpenMP Tasks Suite [6]. The main improvement of the work presented in this paper is that we do not only use the tasking features available in OpenMP 4.0 but also suggest and evaluate new ones. In contrast, the mentioned KASTORS approach [13] suggests new features, different from the ones proposed in this paper, but does not evaluate them.

Besides OpenMP 4.0 and OmpSs, other programming models and runtime system handle task-based parallelism. For example, the StarPU task programming library [2] provides a runtime system and an API to handle task-level parallelism. StarPU has been successfully used to implement important numerical routines [1] on heterogeneous environments, although its capabilites do not outperform OpenMP 4.0. Other approaches reproduce the OmpSs vision to target specific research issues, like the Distributed asyncHronous Adaptive Resilient Management of Applications (DHARMA) [8]. DHARMA is a task programming model designed with resilience as a primary focus. It is a data-flow approach that uses work-over-decomposition. Also, the Open Community Runtime (OCR) [7] initiative aims at creating a standard task-based runtime system. Very simple micro-kernels are publicly available to validate this approach.

5 Conclusions

In this paper we demonstrate the usefulness of three OmpSs features not currently available in the OpenMP 4.0 specification. The first one is the concurrent clause, which can be used to relax synchronization by overlapping task creation with computation. The second is the possibility to handle multiple dependency scenarios in a single `#pragma` annotation and the third one is the NUMA-aware scheduling feature available in the OmpSs runtime system. Each one of these three features provides significant improvements in terms of scalability and programmability. Additionally, this paper provides a comparison in terms of performance of task parallelism against loop parallelism and shows how combining them is sometimes the best option. We expect to provide more examples in the future to further motivate the need for OpenMP extensions and to strengthen the position of OmpSs as an OpenMP forerunner.

The importance of features like the ones discussed in this paper and, in general, of the task parallelism provided by OpenMP and OmpSs is increasing with the emergence of massivelly parallel and heterogeneous hardware, which will certanly require task clauses to allow programmers to handle large amounts of concurrency.

Acknowledgments. This work has been partially supported by the European Research Council under the European Union's 7th FP, ERC Grant Agreement number 321253, by the Spanish Ministry of Science and Innovation under grant TIN2012-34557 and by the HiPEAC Network of Excellence. It has been also supported by the Severo Ochoa Program awarded by the Spanish Government (grant SEV-2011-00067) M. Moreto has been partially supported by the Ministry of Economy and Competitiveness under Juan de la Cierva postdoctoral fellowship number JCI-2012-15047. M. Casas is supported by the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the Co-fund programme of the Marie Curie Actions of the 7th R&D Framework Programme of the European Union (Contract 2013 BP_B 00243).

References

1. Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Roman, J., Thibault, S., Tomov, S.: Dynamically scheduled Cholesky factorization on multicore architectures with GPU accelerators. In: Symposium on Application Accelerators in High Performance Computing (SAAHPC), Knoxville, USA (2010)
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 863–874. Springer, Heidelberg (2009)
3. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: characterization and architectural implications. In: The 17th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 72–81 (2008)
4. Bienia, C., Li, K.: Parsec 2.0: a new benchmark suite for chip-multiprocessors. In: Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation, June 2009
5. Duran, A., Ayguad, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. **21**(02), 173–193 (2011)
6. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP tasks suite: a set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: International Conference on Parallel Processing (ICPP), pp. 124–131 (2009)

7. Knauerhase, R., Sarkar, V.: The open community runtime and its use in systems research. In: Tutorial: International Conference on Architectural Support for Programming Languagues and Operating Systems (ASPLOS) (2013)
8. Kolla, H., et al.: DHARMA: distributed asynchronous adaptive resilient management of applications. In: Minisymposia on Resilience in Numerical Simulations and Algorithms at Extreme Scale. SIAM Conference on Computational Science and Engineering (2015)
9. Kurzak, J., Luszczek, P., YarKhan, A., Faverge, M., Langou, J., Bouwmeester, H., Dongarra, J.: Multithreading in the plasma library. In: Multicore Computing: Algorithms, Architectures, and Applications, p. 119 (2013)
10. Labarta, J., Gimenez, J.: Performance analysis: from art to science. In: Parallel Processing for Scientific Computing, Chap. 2, pp. 9–32. SIAM (2006)
11. Müller, M., Charypar, D., Gross, M.: Particle-based fluid simulation for interactive applications. In: Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA), pp. 154–159 (2003)
12. Valero, M., Moreto, M., Casas, M., Ayguade, E., Labarta, J.: Runtime-aware architectures: a first approach. Int. J. Supercomput. Frontiers Innovations **1**(1), 29–44 (2014)
13. Virouleau, P., Brunet, P., Broquedis, F., Furmento, N., Thibault, S., Aumage, O., Gautier, T.: Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In: DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2014. LNCS, vol. 8766, pp. 16–29. Springer, Heidelberg (2014)