Resource-aware task scheduling

Martin Tillenius, Elisabeth Larsson
Dept. of Information Technology
Uppsala University
Box 337, SE-751 05 Uppsala, Sweden
Email: {martin.tillenius, elisabeth.larsson}@it.uu.se

Rosa M. Badia, Xavier Martorell
Computer Sciences
Barcelona Supercomputing Center
C/ Jordi Girona 1–3, Barcelona 08034, Spain
Email: {rosa.m.badia, xavier.martorell}@bsc.es

***Abstract*—Dependency-aware task-based parallel programming models have proven to be successful for developing application software for multicore-based computer architectures. Here we consider the problem of scheduling tasks not only with respect to their inter-dependencies, but also with respect to their usage of resources such as memory and bandwidth. At the software level, this is achieved by user annotations of the task resource consumption. In the run-time system, the annotations are translated into scheduling constraints. Experimental results demonstrating performance gains both for model examples and real applications are presented.**

I. Introduction

Dependency-aware task-based parallelization is emerging as an important programming model for extracting performance from multicore and manycore architectures, especially for scientific applications where dependencies typically are more complex than in e.g., streaming applications.

The key idea in a task parallel programming framework is to separate the expression of the potential parallelism from the implementation of the parallel execution, thereby reducing the effort of parallel programming. Application software is written in a sequential style in terms of tasks, possibly allowing for nesting of tasks and recursions. A run-time system then handles the parallel execution and dynamic scheduling of tasks onto the available cores. Dependencies are deduced at run-time from user supplied annotations of data accesses and are translated into a format that can be exploited by the run-time system for scheduling purposes. Some examples of successfully employed representations are directed acyclic graphs (DAG) used in OmpSs [1] developed at Barcelona Supercomputing Center, in StarPU [2] developed at INRIA–Bordeaux, and in QUARK [3] employed by the PLASMA and MAGMA projects at the Innovative Computing Laboratory, University of Tennessee; data versions used in SuperGlue [4], [5] and DuctTEiP [6] developed at UPMARC, Uppsala University, and related formats such as the ticket-based approach used in Swan [7], which is a Cilk extension developed by Vandierendonck et al. The first priority for the scheduler is to assure that all dependencies are respected. However, the knowledge of data dependencies can be further employed in order to schedule for locality by placing a task at the core where (parts of) the required data is already in cache.

This work was supported in part by the Swedish Research Council through the Linnaeus centre of excellence UPMARC, Uppsala Programming for Multicore Architectures, and in part by COST Action IC0805 "Open European Network for High Performance Computing on Complex Environments".

The StarPU run-time [2] also incorporates estimated or measured task execution and data transfer times in order to make scheduling decisions for heterogeneous architectures, where the placement in the system of the tasks can be crucial to the overall performance.

However, none of these frameworks take performance degradation due to oversubscription of shared resources into account. The most obvious resources to manage are the shared caches and the bandwidth to main memory. Consider for example a type of task that consumes a lot of bandwidth. If we keep scheduling an increasing number of these tasks to cores sharing the same memory bus, we reach a point where the bandwidth is saturated and it is not possible to get better performance by running more of these tasks in parallel. If there are several types of tasks in the application, a resource-aware scheduler can choose a mix of tasks that together puts less strain on the bandwidth.

Methods for measuring performance issues related to cache and memory bandwidth sharing between different types of processes have been developed in for instance [8], [9], and [10]. In this paper, we consider the related problem of taking shared resources into account when scheduling different types of tasks within a single process. For the time being, we take the pragmatic approach that the programmer specifies how the task uses resources, and leave the option to handle this automatically for future work.

Using a similar idea and implementation as for solving resource scheduling, we are also able to solve a problem where the data dependency model in OmpSs fails to detect parallelism in some cases. The situation arises when two or more tasks need to update the same shared memory address, and thus need exclusive access to this memory, but the order in which the updates occurs does not matter. There is currently no way to specify such dependencies in the StarSs programming model. There are two options in the current model; the tasks must either be executed in the order in which they were submitted, or the programmer needs to manage the exclusive accesses manually. Executing tasks in the order in which they were submitted can lead to lost opportunities for parallelism, and thus large performance penalties, as observed in for instance [11]. Managing the exclusive accesses manually is both an extra burden on the programmer, and a potential source of inefficiency.

In this paper we demonstrate how resource constraints can be incorporated into the existing schedulers in the OmpSs framework and that resource scheduling can lead to significant performance gains. We also introduce a new dependency clause, `commutative`, for denoting that tasks can execute in any order but not concurrently. This leads to an efficient and user-friendly solution to the problem, and renders the work-around strategies suggested in [11] unnecessary.

II. The resource model

The resource model has two sides, the *available resources*, which are the actual resources provided by the computer system that is used, and the *required resources*, which are the resources requested by the tasks. The resources can be of different types. There are *physical resources*, which are provided by the hardware such as cache sizes, memory bandwidth, and memory size. Some of these can be detected automatically by the run-time system, while others need to be manually provided. The physical resources can be connected with a hardware location, for example a bandwidth resource is typically per socket. We also consider *logical resources*, which can be set up by the programmer to mean for instance accessing the hard drive or causing network traffic.

The declared amount of an available resource provides an upper limit for how many of the tasks consuming that resource can run at the same time. This requires some pre-knowledge of the task behavior or tuning of the required resource parameter. Typically, this kind of information can be obtained from execution traces by observing how task execution times vary depending on what the other threads are doing at the same time. These kinds of observations were the motivation behind this work.

Resources can also be used to bind a task to a certain node, socket, core, or GPU. The ability to enforce that only a single task can execute at a time can also be used for correctness, not only for performance. For instance, if the order in which tasks are executed does not matter but they modify the same data, the tasks can be submitted without dependencies but require exclusive access to a resource that represents write access to this data.

III. Commutative accesses

As mentioned above, resources can be used for handling *commutative accesses*, i.e, accesses that modify the same data, but that can be executed in any order. However, applying a resource model to a commutative access is unnecessarily complicated. Since situations where for example several tasks add results to the same variable are commonly occurring, we simply include commutative among the access types that can be annotated by the programmer and interpreted by the run-time system. As has been noted for example in [4], [11], commutative accesses cannot be efficiently represented in a DAG. In Figure 1, we consider an application where computations are performed for all pairs of data. The order does not matter. If the commutative accesses are declared as *inout* in order to create a DAG for the tasks, false dependencies are created between tasks that access the same data. The resulting DAG is shown in the left part of the figure. However, the right part of the figure shows an optimal graph where a unique combination of pairs is accessed at each level. As the figure illustrates, failure to handle commutative accesses properly can result in significant performance losses and reduced levels of parallelism in an application.

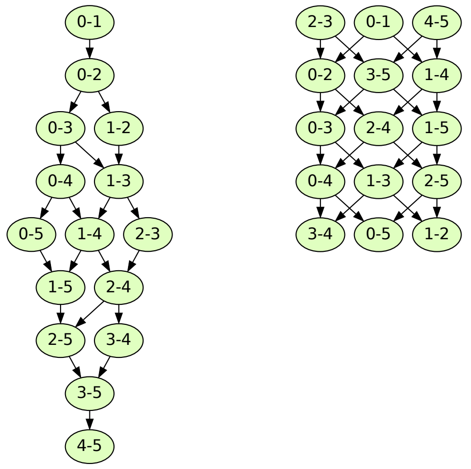


Fig. 1. The left subfigure shows the DAG resulting from declaring all dependencies as *inout*. The right subfigure shows an optimal scheme for the actual *commutative* accesses.

IV. Implementation

We have developed prototype implementations of both the resource-aware scheduling and the commutative accesses in OmpSs. The fact that OmpSs employs DAGs for representing dependencies does not preclude an elegant and efficient implementation.

A. Resources

The *available* resources are defined in a configuration file with name and amount (integer). This file may be partly automatically generated by probing the hardware and partly supplied by the programmer. Resources can currently be declared both globally and per socket. Since resources can be defined per socket, this feature can also be used for affinity. By declaring that a resource is only available on a certain socket, all tasks requiring this resource will be pinned to that socket. This feature could be expanded to include not only sockets but also graphics cards and other accelerators, where pinning certain tasks to certain accelerators could be very useful.

The *required* resources are implemented as an additional clause when defining tasks in OmpSs. This clause defines which resources the task requires, and the amount of resources. The resources are specified by their name, which must also appear in the configuration file, and the required amount must be an integer. Figure 2 shows an example of how a resource clause could appear. A task can require any number of resources.

```
#pragma omp task input( [n]src ) output( [n]dst ) \
                 resource( bandwidth, 2 )
void copydata( int n, double *dst, double *src );
```

Fig. 2. A prototype syntax example for the resource clause in a task pragma. The input and output clauses are used for detecting the data dependencies between tasks.

```
#pragma omp task input( [size][NDIM]local_vec, [size]idx ) \
                 commutative( [NGLOB][NDIM]global_vec )
void scatter( int size, int idx[size],
              float local_vec[size][NDIM],
              float global_vec[NGLOB][NDIM] );
```

Fig. 3. The declaration of the scatter task in the SpecFEM3D application using the `commutative` clause.

By using an external configuration file, the same compiled binary can be executed on several different systems, while the scheduling constraints are adjusted in the configuration file according to the available resources on the different systems.

In OmpSs, the Nanos++ runtime system is responsible for different mechanisms such as context switching, idling, blocking, and queuing, as well as scheduling tasks according to the scheduling policy. All mechanisms are oblivious of the actual policy in use, which can be configured at execution.

Nanos++ supports different task scheduling policies (extensible through a plugin interface) with centralized and distributed queues, in the latter case optionally with work stealing. The *default* scheduling policy uses a locality-aware scheduling algorithm. It uses the data directionality hints provided in the task directive and favors execution at the thread/GPU or node where most of the referenced data is.

The *affinity* scheduling policy favors execution of tasks where most of the accessed data currently resides. A global directory stores the location of all data in the system. When a new task is submitted, the scheduler computes an affinity score for each logical device, and selects the one with the highest affinity. In case of a tie, the task is placed in a global queue which is accessed by all devices and the main host.

The global resource scheduling feature can be used by all the OmpSs schedulers, while the per socket resource declarations only apply for socket-aware schedulers.

The actual implementation is straight-forward. When a task is checked for execution, a new test marks the task as not ready if not all requested resources are available. Otherwise, the resources are acquired during the resource check, and released again when the task completes its execution. The resource awareness is thus limited to knowing whether a task is prevented from running, and is independent of which scheduling strategy is selected in OmpSs.

Tasks in OmpSs can be suspended. If the task is using some resources, it may in some cases, but not always, be desirable to release these resources while the task is suspended. A resource representing memory use should not be released, since the memory will still be held during the suspension. Conversely, a resource referring to cache use should be released, as the task cannot hold on to the cache during suspension. Hence, whether a resource is to be released or not when a task is suspended is a property of the resource. This behavior can be specified per resource in the configuration file.

Since the number of available resources is defined in an external file, and varies with different computer systems, it is possible for a task to require a larger quantity of a resource than is available, preventing it from ever being executed. This is detected at task submission, and generates a run-time error.

B. Commutative accesses

Figure 3 shows an example of how commutative accesses are declared. The actual implementation of commutative accesses is very similar to that of resources. When a task in the ready-queue is checked for execution and a commutative access has been flagged, the run-time system tries to acquire a lock on the memory address it needs exclusive access to. There may be several such accesses for a single task, in which case a lock is acquired for each. If there is a failure along the line of lock acquisition, the already acquired locks (if any) have to be released and the execution check moves to the next task in the ready-queue. If successful, the task is started, and the locks are released upon completion of the task execution.

V. Experimental results

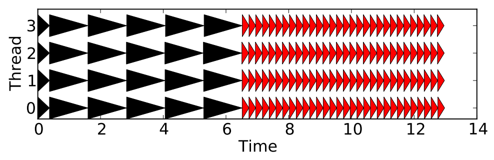
In this section we present experimental results from using the two new constructs; resource-constrained scheduling and the `commutative` clause. All the tests were run on a 4 core Intel 2600K processor, with 8 MB shared cache and hyper-threading disabled, unless otherwise stated.

A. Resource-constrained scheduling

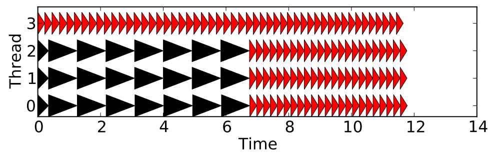
We present two examples that use the resource-constrained scheduling feature to illustrate the possible performance benefits. The first example contains two types of tasks; a *copy* task that copies 100 MB of memory (more than fits in the cache) from one location to another, and a *computation* task that performs dummy computations and have very few memory accesses.

Since the memory bandwidth is shared between the cores, it is expected that the copy tasks will take much longer time when several threads compete for the bandwidth. Instead of running several tasks that use a lot of bandwidth at the same time, we expect better efficiency if a single or few such tasks run at the same time, and tasks whose performance does not depend on memory bandwidth are scheduled on the other threads.

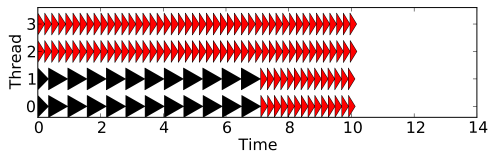
To enforce such a scheduling, a resource called `bandwidth` is defined, and the copy task is declared to require one such resource, as in Figure 2. The available quantity of these resources is then varied, to investigate how the run-time of the application behaves. Figure 4 shows execution traces of these runs. Here we have selected a number of copy tasks that is large enough to show the behavior clearly, and then we selected the number of computation tasks to be large enough to occupy all threads even if only a single copy



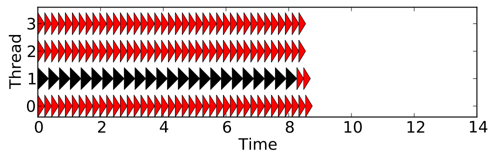
(a) No limitation on the number of copy tasks (black).



(b) Only 3 copy tasks (black) can run at a time.

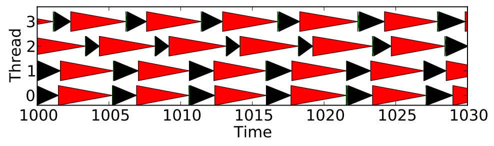


(c) Only 2 copy tasks (black) can run at a time.

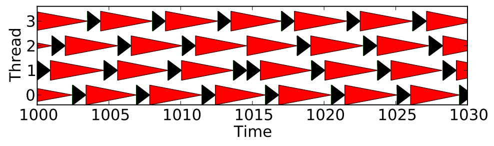


(d) Only 1 copy task (black) can run at a time.

Fig. 4. Memory bandwidth benchmark: Execution traces with different limits on the number of copy tasks that may execute at a time. Black triangles represent copy tasks and red triangles represent computation tasks.



(a) Without constraints, allowing several file accesses (black) concurrently.



(b) Using resources to constrain file accessing tasks (black) to at most 1 at a time.

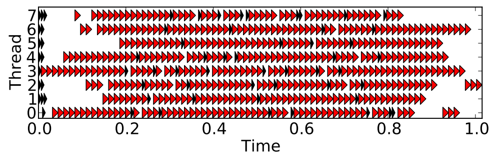
Fig. 5. JPEG compression application: Execution traces for an application that reads images from file and compresses them. There are three task types; read (black), compress (red), and write (green). The writes are very quick and barely visible here.

task is allowed to run at once. All the executions in Figure 4 perform the same amount of work and has the same number of tasks, but due to bandwidth limitations, the execution time varies with how many copy tasks are allowed to run at a time. In this example, the execution time was reduced by about 30% when the bandwidth demanding tasks were run sequentially, as compared to when they run concurrently on all four threads.

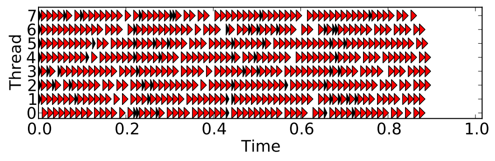
The second example we present that use the new resource-

```
// Gather nodes from displ to local
gather( size, idx, displ, elemvec );

// Perform computations on the local vector
process( elemvec );

// Add results into global vector accel
scatter( size, idx, elemvec, accel );
```

Fig. 6. Code for generating the three tasks used to process an element.



(a) Using the `inout` clause.



(b) Using the `commutative` clause.

Fig. 7. Execution traces for an n-body simulation illustrating the benefit of using the information that certain accesses are commutative.

constrained scheduling is an application for converting raw images to JPEG files. A large number (about a thousand) of uncompressed images are read from file, compressed, and written back to disk. This is performed by three different kind of tasks called *read*, *compress*, and *write*. In this case we wish to limit the number of tasks that access the disk at the same time, that is, the read and write tasks.

Figure 5 shows small parts of execution traces from the application, when resources are used to limit the file accesses, and for comparison also when tasks are scheduled freely.

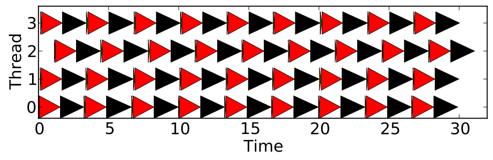
As can be seen in Figure 5a, the read tasks are wider when several of them are executed at the same time. Comparing the black triangles on thread 2 in the first part of the trace, these are distinguishably shorter than the black triangles on thread 0 and 1.

We then introduce a resource for file accesses that both the read and the write task requires, and a part of an execution trace from running with these constraints is shown in Figure 5b. In this trace, the read tasks are always fast and notably more uniform in length than in Figure 5a.

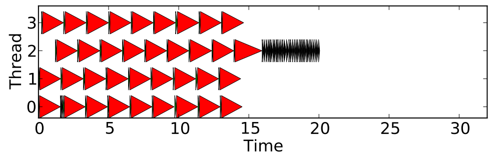
By using resources to constrain the read and write tasks, reads became 30% faster and writes 40% faster, while the compression tasks were unaffected. Since most of the total run-time is spent on compression, speeding up the read and write tasks have a smaller impact, but still gave a total speedup of 6%.

B. The `commutative` clause

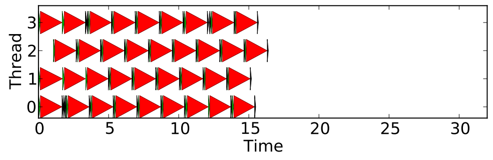
For evaluating the `commutative` clause, we also have two different applications. The first application is an n-body simulation. The application uses a time-stepping algorithm where



(a) Using the `concurrent` clause and atomic updates.



(b) Using the `inout` clause.



(c) Using the `commutative` clause.

Fig. 8. Execution traces for the SpecFEM3D benchmark, using three different methods for writing the results into the shared vector.

each time step consists of calculating the force interactions between all pairs of particles, and then moving the particles according to these forces. For each particle, the forces from all other particles acting upon this particle are accumulated. If this accumulation is performed by a task that uses the `inout` clause, the run-time system must accumulate the forces in the order the tasks were created. This introduces false dependencies, and is the situation illustrated in the left side of Figure 1. Using the `commutative` clause instead, the run-time system is free to schedule the accumulation of the forces in any order, and can run the tasks as in the right-hand side of Figure 1 instead.

Figure 7 shows execution traces of an simulation of 8192 particles divided up in blocks of 512 particles each, run for 4 time steps. In Figure 7a, the lack of parallelism in the beginning and in the end of the run is evident. This corresponds to the narrow parts in the beginning and the end of the DAG in the left-hand side of Figure 1. When the `commutative` clause is used instead, the tasks are much more densely packed, as shown in Figure 7b. In this case, using the `commutative` clause instead of the `inout` clause gave a speedup of 13%.

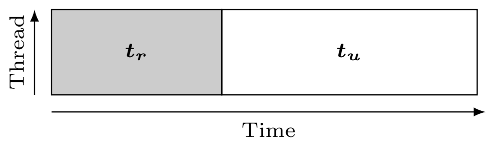
We have also used the `commutative` clause in an application software for simulating earthquakes called SpecFEM3D. This software uses a finite element method with large elements, consisting of $5 \times 5 \times 5$ nodes each. The part of the application that is interesting for our case is the local computations on each element, shown as pseudo-code in Figure 6. The calculations on each node are performed by first gathering the nodes that belong to the element from a vector with global indexing of all the nodes, into a local vector. The element is then processed by calculations on this local vector, and when the computations are finished, the results are accumulated in another large vector using the global indexing. This is performed in three different tasks; *gather*, *process*, and *scatter*. Several different scatter tasks write to the same elements in the global output vector, as nodes are shared between several elements. The order in which the results are accumulated does not matter, but two tasks must not write to the same element at the same time.

This mutual exclusion can be managed in different ways in the current version of OmpSs. We will compare our solution using the new `commutative` clause to two other options; (i) using the `concurrent` clause, or (ii) using the `inout` clause. In the first case, the scatter tasks can run in any order, and at the same time, but all updates need to be performed atomically. The drawback of this is that atomic updates are more expensive than direct writes, especially when there is contention. In the other case, the scatter tasks are executed in the order they were submitted, as the run-time system cannot assume that it is safe to reorder the tasks. This may cause bad scheduling.

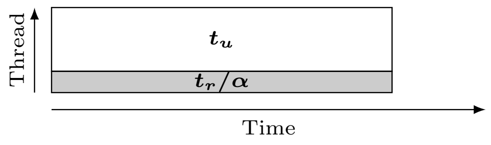
An alternative solution is to introduce a resource of which there is only one available, and use the `concurrent` clause on the output vector but at the same time require this resource. This would cause the tasks to execute one at a time, and in any order. However, by using the new `commutative` clause instead, the mutual exclusion will automatically be associated with the memory address, and it is not necessary to introduce resources and specify which tasks requires them. This is more elegant, requires less writing, and does not rely on using resources for correctness.

Figure 8 shows small parts of execution traces for the three methods. A single gather-compute-scatter cycle is shown, while the full execution contains many such steps, together with other tasks. Here, the black triangles represent the scatter tasks, red tasks are computation tasks, and the barely visible green tasks are gather tasks. In 8a, the scatter tasks are much slower than in the other methods, because of the additional costs from using the compare-and-swap instruction. Figure 8b shows the behavior when the scatter tasks have an `inout` access to the output vector. Here the scatter tasks are much faster, but since they must execute in a predefined order, almost all of them are executed first after all the other tasks have finished. A few scatter tasks are actually executed on thread 0, after the first computation task (red triangle), but since the scatter tasks must execute in a given order, and they have dependencies on the computation tasks, the scatter task that depend on the last computation task is encountered, and the remaining scatter tasks must wait to the end. In average it is expected that about half of the scatter tasks must execute at the end, but in practice computation tasks that are added early are executed late by the default scheduler, causing most scatter tasks to be pushed to the end. Hence, the execution trace in this figure does not show a particularly unlucky scheduling, but a representative for the common case.

In this application, there are no other tasks that run between the computation tasks and the next step, so this causes a section where a single thread executes scatter tasks, and the



(a) Worst case: First run all tasks that shares resources, then run the remaining, resource-independent, tasks.



(b) Best case: Constrain the tasks that share resources, and fill remaining threads with the remaining tasks.

Fig. 9. The worst and best case for running tasks that shares resources. The total time spent on resource-independent tasks (t_u) is the same in both figures, while the time for running resource-sharing tasks (t_r) is divided by a speedup factor α when the scheduling is resource constrained.

other threads are idle.

Using the `commutative` clause reduced the time by 40% compared to atomic updates and by 10% compared to `inout` dependencies.

VI. Analysis

Here we analyze and predict what speedup can be achieved by taking resource sharing into account when scheduling, by finding the upper limit in a best-case scenario. We consider a single resource, which is used to decide how many tasks of a certain type that may run concurrently.

Let all tasks be divided into two categories, one for tasks that are limited by the resource, and one for the other tasks. The run-time for tasks not limited by a resource is assumed to be independent of which other tasks are running. Tasks that are limited by a resource are assumed to run faster when they are not all running at the same time. We let α denote the mean speedup of the resource-limited tasks when they are running on n_r threads instead of on all available threads n.

To find the upper limit of the possible speedup, we consider the best and worst case scenarios, as illustrated in Figure 9. In the worst case scenario all tasks that share a resource run at the same time. In the best case scenario, tasks that require a resource are mixed with other tasks. Also, the relation between time spent in resource constrained tasks t_r and time spent in unconstrained tasks t_u should be such that the execution of both task types finish at the same time when all the resource constrained tasks are run on n_r processing units while the remaining $n-n_r$ threads execute the unconstrained tasks. That is, the relation should be

$$\frac{t_r}{\alpha n_r} = \frac{t_u}{n - n_r}.$$

If there are more resource constrained tasks than this, there are no unconstrained tasks to fill the remaining threads with, and there will be idle threads. On the other hand, if the time spent on resource constrained tasks is smaller, it is a smaller fraction of the total run-time, and speeding them up will affect the total run-time less.

Under these conditions, the expected speedup from resource scheduling is

$$1 + \frac{n_r}{n}\left(\alpha - 1\right).$$

Applying this for the JPEG compression application, we have $n_r = 1$, $n = 4$, $\alpha = 1.30$, and the best possible speedup is found to be 7.5 %, quite close to the speedup of 6 % we achieved in the experiment.

VII. Conclusions

We have proposed two simple additions to the OmpSs programming model. First we have introduced the concept of `resources` into OmpsSs, to provide a method for constraining too many tasks that share common resources from being scheduled at the same time. We have made a prototype implementation of this feature in OmpSs, and present two example applications where the total run-time was shortened by 30% by avoiding depletion of the memory bandwidth in the first case and by 6% in the second, by serializing IO accesses.

Secondly, we have added a new dependency clause to OmpSs; `commutative`. This allows tasks to be executed in any order but not at the same time. By simply changing the directive `inout` to `commutative` for tasks that performs commutative accesses, we were able to achieve a speedup of 13% in a n-body simulation application, and 10% in SpecFEM3D, a software for earthquake simulations.

References

[1] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, 2011.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[3] A. YarKhan, J. Kurzak, and J. Dongarra, "QUARK Users' Guide: QUeueing And Runtime for Kernels," ICL, University of Tennessee, Tech. Rep. ICL-UT-11-02, 2011.

[4] M. Tillenius and E. Larsson, "An efficient task-based approach for solving the n-body problem on multicore architectures," in *PARA 2010: State of the Art in Scientific and Parallel Computing*. University of Iceland, Reykjavík, 2010, 4 pp.

[5] M. Tillenius, "Leveraging multicore processors for scientific computing," Licentiate thesis, Department of Information Technology, Uppsala University, Sep. 2012.

[6] A. Zafari, M. Tillenius, and E. Larsson, "Programming models based on data versioning for dependency-aware task-based parallelisation," in *CSE 2012: The 15th IEEE International Conference on Computational Science and Engineering*, Paphos, Cyprus, 2012, 6 pp.

[7] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos, "A unified scheduler for recursive and task dataflow parallelism," in *PACT*, 2011, pp. 1–11.

[8] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao, "Cache contention and application performance prediction for multi-core systems," in *ISPASS*, 2010, pp. 76–86.

[9] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten, "Cache pirating: Measuring the curse of the shared cache," in *ICPP*, 2011, pp. 165–175.

[10] ——, "Bandwidth bandit: Understanding memory contention," in *ISPASS*, 2012, pp. 116–117.

[11] C. Niethammer, C. Glass, and J. Gracia, "Avoiding serialization effects in data / dependency aware task parallel algorithms for spatial decomposition," in *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, july 2012, pp. 743–748.