Resource-Aware Task Scheduling

MARTIN TILLENIUS and ELISABETH LARSSON, Uppsala University
ROSA M. BADIA and XAVIER MARTORELL, Barcelona Supercomputing Center

Dependency-aware task-based parallel programming models have proven to be successful for developing efficient application software for multicore-based computer architectures. The programming model is amenable to programmers, thereby supporting productivity, whereas hardware performance is achieved through a runtime system that dynamically schedules tasks onto cores in such a way that all dependencies are respected. However, even if the scheduling is completely successful with respect to load balancing, the scaling with the number of cores may be suboptimal due to resource contention. Here we consider the problem of scheduling tasks not only with respect to their interdependencies 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 runtime system, the annotations are translated into scheduling constraints. Experimental results for different hardware, demonstrating performance gains both for model examples and real applications, are presented. Furthermore, we provide a set of tools to detect resource sensitivity and predict the performance improvements that can be achieved by resource-aware scheduling. These tools are solely based on parallel execution traces and require no instrumentation or modification of the application code.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; B.8.0 [**Performance and Reliability**]: General; G.1.0 [**Numerical Analysis**]: General—*Parallel algorithms*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Dependency aware, dynamic scheduling, resource contention, task parallel

ACM Reference Format:
Martin Tillenius, Elisabeth Larsson, Rosa M. Badia, and Xavier Martorell. 2015. Resource-aware task scheduling. ACM Trans. Embedd. Comput. Syst. 14, 1, Article 5 (January 2015), 25 pages.
DOI: http://dx.doi.org/10.1145/2638554

1. INTRODUCTION

1.1. Motivation

Today, most computer systems—from embedded systems, via laptop and desktop computers, to high-performance computer systems—are based on multicore architectures. Efficient utilization of these systems (in terms of resources as well as power) requires application software to be parallel, unless it can be assured that enough processes to occupy all cores are normally being executed simultaneously.

This work was supported by the Swedish Research Council through the Linnaeus centre of excellence UPMARC, Uppsala Programming for Multicore Architectures; COST Action IC0805 "Open European Network for High Performance Computing on Complex Environments"; the Spanish Ministry of Education (CICYT TIN2011-23283); and the Generalitat de Catalunya (2009-SGR-980).
Authors' addresses: M. Tillenius and E. Larsson, Department of Information Technology, Uppsala University, Box 337, SE-751 05 Uppsala, Sweden; emails: martin.tillenius@gmail.com, elisabeth.larsson@it.uu.se; R. M. Badia and X. Martorell, Computer Sciences, Barcelona Supercomputing Center, Jordi Girona 1–3, Barcelona 08034, Spain; emails: {rosa.m.badia, xavier.martorell}@bsc.es.

ACM 1539-9087/2015/01-ART5 $15.00
DOI: http://dx.doi.org/10.1145/2638554

However, writing parallel software that performs well over a range of multicore architectures can be nontrivial, especially for applications with low numbers of computations per memory access (bandwidth limited applications). In Datta et al. [2008], where an optimized code for stencil computations is developed, it is observed that there is a "complete lack of multicore scalability without auto-tuning." The results in the paper are good but require hierarchical blocking, padding, thread and data co-location, prefetching, and use of SIMD instructions, combined with autotuning for each specific hardware system. Another bandwidth limited application—sparse matrix-vector multiplication—is studied in Williams et al. [2009]. Similar optimizations are employed, and it is reported that the resulting implementation is 13,000 lines long.

Performing this type of extensive optimizations can be feasible for library software that will be used in many different applications. However, in general, the cost in programmer hours for developing an application-specific software is high or even too high.

An approach that is becoming mainstream, especially for scientific applications, is task-based parallel programming. The idea is to let the programmer write a sequential program with the algorithm divided into tasks (chunks of work). Then the scheduling of the tasks onto the hardware is handled by a runtime system, relieving the programmer of the need to deal with the parallel programming aspects. The tasks must still be implemented efficiently with respect to block sizes and reuse of cached data, but the optimizations relating to locality and scheduling can be moved to the runtime level.

A typical task parallel framework for scientific applications handles data dependencies. The dependency information can be used by the runtime system to schedule for data locality [Pérez et al. 2008; Duran et al. 2011; Tillenius 2012b] or, going even further, to also include the estimated cost of data transfer and task execution to make scheduling decisions for heterogeneous architectures [Augonnet et al. 2011]. However, to our knowledge, the matter of scheduling to reduce resource contention has not been studied in the context of task parallel programming. The present work was motivated by observed performance degradation for particular tasks due to resource contention in real execution traces generated in OmpSs [Duran et al. 2011], the task parallel framework developed at Barcelona Supercomputing Center. We have also observed similar effects in execution traces from the SuperGlue framework [Tillenius 2012b] developed by the first author at Uppsala University.

1.2. Background

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 runtime system then handles the parallel execution and dynamic scheduling of tasks onto the available cores. The frameworks in which we are interested are dependency aware, where the data dependencies are deduced at runtime from user-supplied annotations of data accesses.

The OmpSs framework [Duran et al. 2011] provides an extension of the OpenMP API with clauses for task definitions and data accesses [Ayguadé et al. 2010]. The data dependencies are inferred from the data access clauses and translated into a directed acyclic graph (DAG) of the tasks, which is then used by the runtime scheduler. Load balancing is achieved by task stealing. The resource management techniques that we propose here are implemented as additional resource-related clauses in OmpSs. In the StarPU framework [Augonnet et al. 2011], tasks are instead expressed explicitly. Here, dependencies are expressed as DAGs as well, and the particular feature is that the time for data transfer and task execution is estimated at runtime to decide where it will be most time efficient to place a task. This is relevant for heterogeneous architectures combining multicore CPUs and GPUs. The widely used linear algebra library LAPACK is currently being ported to multicore architectures in the projects PLASMA and MAGMA [Agullo et al. 2009]. A tailored runtime system, QUARK [YarKhan et al. 2011], which also employs DAGs, has been developed for use in the new library versions.

Building a DAG that connects the tasks implies that dependencies are between tasks, where in fact they are data dependencies. In the SuperGlue framework for shared memory systems [Tillenius and Larsson 2010; Tillenius 2012a] and the DuctTEiP framework for distributed memory systems [Zafari et al. 2012], we have developed a dependency management approach based on data versions, which allow for more flexibility than a pure DAG. A related idea is implemented in Swan [Vandierendonck et al. 2011], which is a Cilk [Leiserson 2010] extension, combining the fork-join parallelism of Cilk with dependency-aware task parallelism. There, the data versions are handled by a ticket system. In all of these approaches, the focus is mainly on the tasks.

In the case of single-threaded applications running in parallel on a multicore system, extensive research has been performed on contention-aware thread-level scheduling in the operating system (OS). A very nice survey of different solutions is presented in Zhuravlev et al. [2012]. A basic assumption here is that the amount of resource sharing between different cores is uneven. Then, it makes sense to pay attention to which threads are scheduled close to each other and hence can interfere with each other's resource usage. The scheduling decisions are enforced by binding or migrating threads to the desired cores. Hence, the contention-aware OS scheduling provides a method of allocating the threads in space, but not time.

However, as mentioned already in Zhuravlev et al. [2012], the situation when running task parallel applications is different from the OS perspective in several ways. First, the OS sees a multithreaded application running on some or all of the cores instead of independent single-threaded jobs. Second, the task parallel application has its own runtime system, which handles work allocation to its allotted cores. Furthermore, in the task parallel runtime system, work allocation spans both space and time slots. Threads are typically associated with a specific core during the whole execution, and tasks can be assigned (dynamically) to any of the worker threads.

A situation that is more similar to the task parallel case is scheduling for multicore processors that employ simultaneous multithreading (SMT). Then the number of jobs to run (when all threads are occupied) is larger than the number of cores, and hence a decision needs to be made about which jobs to co-schedule in each time slice. Some examples will be discussed in Section 1.3.

A different context is real-time systems, where anything unpredictable like dynamic schedules or resource sensitivity can be damaging, as worst-case execution times must be guaranteed. There, a typical approach is to isolate threads by partitioning resources. For example, in Guan et al. [2009], strategies for handling cache contention by cache partitioning are analyzed.

Interesting scheduling issues arise in the intersection of these different domains. Multithreaded task parallel applications can be co-scheduled with either a combination of OS scheduling on the application level and internal scheduling by the respective runtime systems or by combining different applications in the same runtime system. Furthermore, in the real-time case, mixed-criticality systems arise in many applications (see Ekberg and Yi [2012] for a study of mixed-criticality scheduling). Perhaps resource partitioning for the hard real-time tasks could be combined with contention-aware scheduling of less critical parts.

A resource as discussed previously is typically a shared hardware component such as a shared cache or the off-chip memory bus. In this article, we use the resource concept broadly to represent anything that may cause contention between tasks. We have developed two types of resource constructs. The *commutative* clause is used for tasks that require exclusive access to a resource. Examples of resources that fit in this category are (commutative) write accesses to a specific memory location and I/O. The general resource construct can be used for resources that can be shared by more than one thread, such as cache and bandwidth. Locality can also be considered as a resource, which we implement here by managing the available cores and sockets in the resource framework.

1.3. Related Work

Choosing which tasks to run in parallel is closely related to choosing which (single or multithreaded) jobs to co-schedule in an SMT context. There, all jobs have to be given a fair share of the execution time to make progress, which means that the execution is time sliced and different jobs are running together in different slices. An SMT schedule should be interpreted as a set of job selections that are rotated over time. In Snavely and Tullsen [2000], the performance of different job schedules is predicted in a sampling phase. Several measures based on performance counters are used in the prediction. Then the best candidate is selected and used for the remaining execution time. Improvements of 17% between the worst and best schedules are reported. In El-Moursy et al. [2006], the execution is divided into time intervals, performance counters are examined at the end of each interval, and decisions are made about migrating threads. Hence, the schedule can change dynamically over time. Improvements of 7% compared with the best static schedule were observed in the simulations. A scheduling approach with focus on finding groups of threads that result in low L2 cache miss ratios was simulated in Fedorova et al. [2005] with performance improvements of 27% to 45%. In Bhadauria and McKee [2010], co-scheduling is performed with respect to power and performance. Here the jobs are supposed to be multithreaded and malleable. The number of cores for each job is chosen by the scheduler as well as which jobs to co-schedule. The energy-delay product (EDP) is improved by a factor 1.5 over standard job rotation and time multiplexing. The energy was reduced by 26%, whereas the performance increased by 19%.

The combined results of these papers show that there really is a case for resource-aware or contention-aware scheduling. However, there is no unique answer to the question of which type of contention is most important. Each strategy that was developed considers a variety of measures and counters; furthermore, the paper of Bitirgen et al. [2008], which takes a machine learning approach to determine the best scheduling approach, states that multiple contention sources must be considered, otherwise "resource interactions render individual adaptive management policies largely ineffective."

The present article is an extension of the conference paper [Tillenius et al. 2013] presented at the PARMA 2013 workshop. For the workshop, we had developed a resource model for task parallel programming and implemented it in OmpSs; we had shown that we could get performance improvements for several benchmarks; and we had provided a theoretical analysis for best case speedup. The new technical contributions in the present article are as follows:

—An extension of the resource model to include a core resource
—A simple diagnostic measure to estimate task sensitivity to resource sharing
—A method to predict the performance improvement when resource-aware scheduling replaces resource-oblivious scheduling
—Further experiments and analysis for multiple resource contention and co-scheduling.

Our estimates of task sensitivity are based solely on execution times from an unconstrained schedule. We are not trying to determine the exact cause of the performance degradation, but rather to see the compound effect of contention and sharing. The approach is simple and does not require any instrumentation of the code. An alternative approach to measure sensitivity is described in Ceballos and Black-Schaffer [2013], where a Cache Pirate [Eklov et al. 2011] application is run in parallel either with a single task or with the whole application to determine its sensitivity to losing access to parts of the shared cache. The observed sensitivities are quite small. A possible reason could be that the actual cache contention is not the dominant cause for performance degradation when threads share the cache. In Blagodurov et al. [2010], it is determined that memory controller contention, memory bus contention, and prefetch hardware contention all contribute in a complex way. Furthermore, again referring to Bitirgen et al. [2008], there may be effects of other types of multiple resource contention as well.

A problem noted in Ceballos and Black-Schaffer [2013] is that many applications have phases where most tasks are of the same type. We concur with this observation. It will not always result in performance gains to schedule with respect to resources for task parallel applications. Instead, the potential gain highly depends on the mix of tasks in the application. However, there can clearly be benefits from co-scheduling different types of (task parallel) applications when possible.

A specific type of contention problem is considered in Niethammer et al. [2012] in relation to task parallel programming and the use of DAGs to represent dependencies. The situation arises when two or more tasks need to update the same shared memory address and thus need exclusive access to this memory; however, the order in which the updates occur does not matter. Currently, there is no way to specify such dependencies in the StarSs programming model used by OmpSs. The options are to either specify that the tasks *update* the memory address, in which case the tasks will be scheduled in the order in which they were submitted, or to manage the dependencies manually. Executing tasks in the order in which they were submitted can lead to lost opportunities for parallelism and thus large performance penalties. Managing the exclusive accesses manually is both an extra burden on the programmer and a potential source of inefficiency. In this article we show how the new commutative dependency clause leads to an efficient and user-friendly solution to the problem and renders the work-around strategies suggested in Niethammer et al. [2012] unnecessary.

2. MODELING RESOURCES AND ACCESSES

Both for the modeling and the analysis performed later, we assume that the resource consumption of a task is uniform during its execution—that is, we consider tasks as homogeneous units. We also assume that an application has a finite number of task types that recur during the application execution. As mentioned previously, from a practical point of view, the effect of the contention is more interesting than its cause. This means that we can be quite free in choosing what kind of resources we model, instead of breaking them done into exact components. For example, a general memory resource may in fact refer to all combined contention sources in the shared parts of the memory system.

2.1. The Resource Model

The resources to be managed are specified in a configuration file. In this file, the name and the available quantity of each resource is specified. As part of the task specification, it is then possible to declare that a task requires a certain amount of a certain resource.

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 declared available quantity of a resource provides an upper limit for how many tasks (that require the 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, GPU, or accelerator. 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.

2.2. The Core Resource

As described in [Al-Omary et al. 2013], nonuniform memory access (NUMA) effects may result in performance losses similar to those incurred due to resource contention. The key issue is data locality, which cannot be directly tied to a resource. However, by introducing a core resource, we can control where tasks are scheduled, which does affect data locality. A set of cores can be reserved for execution of a certain group of tasks by requesting the associated core resources.

We use hierarchical (or nested) tasks [Planas et al. 2009] as an example. A parent task generates a number of subtasks that all work on related data. The subtasks are distributed over the available cores. Several of the parent tasks can run at the same time, and their respective subtasks all compete for the different cores.

By introducing a core resource and letting the parent tasks require a certain quantity of the core resource, we can provide exclusive access, meaning that all subtasks working on related data can be scheduled onto these reserved cores. Data locality is improved in two ways. Subtasks from the same parent are scheduled together, and there is no interference from other tasks. However, the group of cores may still be distributed over the NUMA nodes (sockets), resulting in variable performance. For example, it takes longer for a core in one socket to access the data of a core in another socket than to access the data of the neighboring core in the same socket. We refer to this variability in terms of NUMA distance. To go further, we can group the resources according to locality in such a way that the reserved cores have the smallest possible NUMA distance from each other.

2.3. Commutative Tasks

It is possible to use the resource constraints to enforce that only a single task of a certain task type can execute at a time. This can also be useful for correctness, not only for performance. For instance, if several tasks accumulate their subresults into the same memory location, the order in which these tasks are executed does not matter as long as they do not run concurrently.

However, applying a resource model to a commutative access is unnecessarily complicated. In situations such as when 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 runtime system. As has been noted in [Tillenius and Larsson 2010; Niethammer et al. 2012], 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* 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-hand image of the figure. However, the right-hand image of the figure

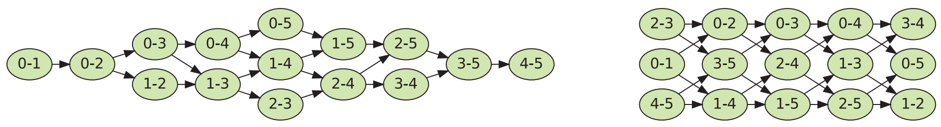


Fig. 1. The image on the left shows the DAG resulting from declaring all dependencies as *inout*. The image on the right shows an optimal scheme for the actual *commutative* accesses.

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.

3. 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. The description of the implementation here refers to the OmpSs programming model and runtime system. However, the presented constructs are general and could as easily be implemented in another task parallel framework.

3.1. Scheduling in OMPSs

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. The scheduling policy determines how ready tasks (tasks whose dependencies are already satisfied) are executed. Nanos++ supports different task scheduling policies (extensible through a plugin interface) with centralized and distributed queues, in the latter case optionally with work stealing. Which policy to use can be configured when the application is executed through environment variables.

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 located.

The *socket-aware* scheduling policy keeps one ready queue per NUMA node, and the queues are sorted by task priority. When tasks are submitted, they are assigned to a NUMA node, and nested tasks will run on the same node as their parents.

All mechanisms for resource-awareness introduced in this article are oblivious to the actual scheduling policy in use, except that some additional support is implemented in the socket-aware scheduling policy.

3.2. 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, as well as 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 a resource clause. 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.

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

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

The actual implementation is straightforward. 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 are 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, as 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. The current implementation does not allow resources to be oversubscribed, so a task that requires more resources than are available is prevented from ever being executed. This is detected at task submission and generates a runtime error. Whether to accept oversubscription of resources or not is a property of the resource and could be added as a parameter in the configuration file. By guaranteeing that the resource constraints are respected, the resource concept can also be used for correctness, typically for guaranteeing that at most one task uses an otherwise unprotected shared resource.

3.3. Core Resources

Resources representing computational cores are handled as a special case. In addition to behaving like other resources, there is a special procedure to allocate several worker threads to work on a single task. The first worker thread that finds a task requiring several cores puts a request in for other worker threads to join in a queue, then waits until the requested number of worker threads have joined. All worker threads check this queue for requests after each executed task, thus giving such tasks higher priority than tasks in the ready queue. If a request is found, the worker thread joins and is locked to only execute the task that initiated the request and its descendants. Task stealing is also limited so that worker threads locked for a certain task can only steal from other worker threads locked for the same task, and external workers cannot steal tasks from locked threads. The *core* resources also work as ordinary resources that are checked out before the request for other worker threads is put in the queue to make sure that there are not requests for more cores than are available in the system.

3.4. Commutative Accesses

Figure 3 shows an example of how commutative accesses are declared. The actual implementation of commutative accesses is similar to that of resources. When a task in

```
#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.

the ready queue is checked for execution and a commutative access has been flagged, the runtime system tries to acquire a lock on the memory address to which it needs exclusive access. 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 locks are released upon completion of the task execution.

4. SCHEDULE ANALYSIS

In this section, we consider how to extract simple diagnostic statistics and accurate predictive information from recorded task execution times. We also discuss the theoretical implications for the potential gain that resource-aware scheduling can provide.

4.1. Simple Schedule Diagnostics

Assume that we have recorded execution times for the n_t different task types in an application during parallel execution with resource-oblivious scheduling. To define a measure of the sensitivity to contention of the different tasks, we start from a statistical measure—the interquartile range (IQR) of a dataset. It is a measure that is robust with respect to outliers of the scale in a distribution. The IQR is measured as $Q_3 - Q_1$, where Q_1 is the first quartile and Q_3 the third quartile. In execution time measurements, it is natural to have some slow outliers due to exceptional conditions. However, it is less likely to find times that are unreasonably fast. Therefore, we have modified the measure by replacing Q_1 with the minimum execution time m to not lose information about short times. Let a subscript i indicate that a measure is computed with respect to the dataset corresponding to task type i. To compare the sensitivity of different tasks, we normalize the results with respect to m_i. Then we get the sensitivity of task type i from

$$\sigma(i) = \frac{(Q_3)_i - m_i}{m_i}, \quad i = 1, \ldots, n_t. \tag{1}$$

Figure 4 shows measured execution times for two types of tasks in the same application. For task type 1, there is little variation with a few slow outliers, as well as some shorter times. The computed sensitivity is $\sigma(1) \approx 0.25$. For task type 2, the variation is large, and the resulting sensitivity is $\sigma(2) \approx 1.99$. Clearly, task type 2 is much more sensitive than task type 1. If the sensitivity $\sigma > 1$, then 25% of the tasks are more than twice as slow as the fastest task.

It is also possible to get a rough estimate of the potential reduction in execution time R of the application if the resources can be managed perfectly by comparing the measured execution time T_0 with the resulting time if all tasks are executed as fast as the fastest-measured task execution time m_i. Assuming that we have n_i tasks of type i, the estimated reduction is

$$R = \frac{T_0 - \sum_{i=1}^{n_t} n_i m_i}{T_0}. \tag{2}$$

Relating this to Figure 4, the estimate tells us how a large part of the total execution time is above the lines, representing the fastest task executions. For the example, the

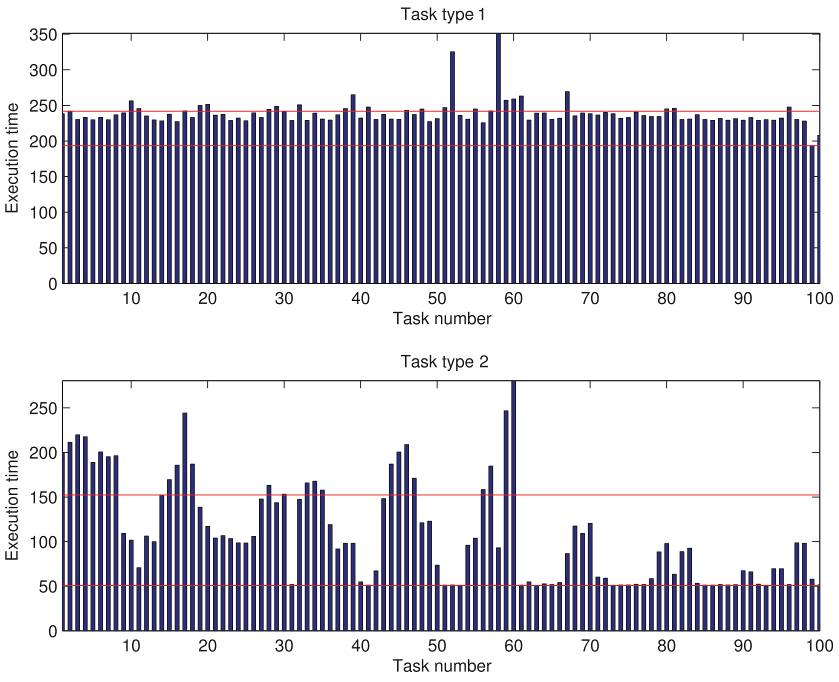


Fig. 4. Execution time variation for two task types. The solid lines show the minimum execution time and the third quartile time. The first type of tasks is computationally heavy, whereas the second consumes a lot of memory bandwidth.

resulting number is $R \approx 30\%$. However, a large part of this reduction is for the first tasks that were not so sensitive. If we exclude the two exceptionally short times at the end that may be uncertain, we get $R \approx 21\%$. If m_i is a reasonable estimate, R provides an upper limit for the performance benefits that can be achieved with resource-aware scheduling.

Both measures perform well as long as the estimates for the shortest execution times $m(i)$ are close to the real values. This may not be the case if there is interference between all types of tasks. However, if serial execution traces are available, $m(i)$ can be found from these instead. The serial results can then be used as input to the sensitivity and time reduction measures for parallel execution.

4.2. Performance Prediction

By studying the parallel execution traces in more detail, we aim to determine a priori how the resource parameters should be chosen to get the best possible performance. We start with a basic model where we assume contention (or constructive sharing) only between tasks of the same type. The execution time prediction contains several subproblems that are discussed in detail in the following subsections.

4.2.1. Inferring Task Execution Times under Contention. Based on the data, we are going to determine the speed of each task type when competing with different numbers of tasks of the same type. The total execution time of each task can be divided into intervals with different levels of competition. Figure 5 illustrates a part of an execution trace. The execution time of one of the tasks is divided into intervals with the numbers of tasks of the same type indicated. Let the amount work in a task of type i be denoted by w_i. Let $c_i(r)$ be the speed of a task of type i in terms of work per time unit when r

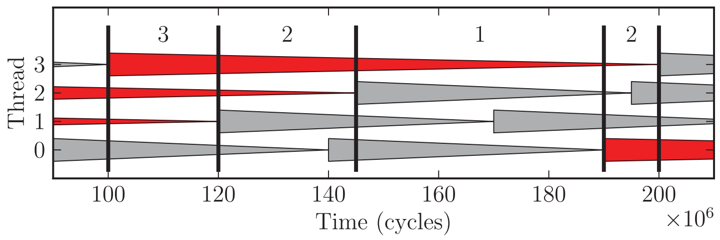


Fig. 5. Extract from an execution trace. Each triangle represents a task with the start time at the base and the finish time at the tip. The two colors represent different task types. The vertical lines indicate all events that involve the red task type and divide the execution of the top-most task into four different intervals with the number of concurrent tasks of the same type indicated.

instances of the same type are running simultaneously. If we denote the time that the jth task of type i is competing with r instances of its own type by $\delta t_{ij}(r)$, we can build an equation for each task in the following way:

$$\sum_{r=1}^{p} c_i(r)\delta t_{ij}(r) = w_i, \quad j = 1, \ldots, n_i, \quad i = 0, \ldots, n_t - 1. \tag{3}$$

For the example in Figure 5, with a task of type $i = 1$, the time intervals are of lengths 20, 25, 45, and 10 Mcycles, resulting in the following equation:

$$45\, c_1(1) + 35\, c_1(2) + 20\, c_1(3) = w_1.$$

Gathering the equations for all of the tasks, we get a linear system of equations $A_i \underline{c_i} = \underline{w_i}$ for each task type, where A_i is a rectangular matrix of size $(n_i \times p)$, $\underline{c_i} = (c_i(1), \ldots, c_i(p))^T$, and $\underline{w_i}$ is a vector of length n_i with all elements equal to w_i. We need to normalize the equations, and we choose to let the work in each task $w_i = 1$. This system of equations can then be solved in the least squares sense to get the speed estimates for each task. This works well if all levels of contention are well represented. However, this is not necessarily the case. Therefore, we have developed a three-step algorithm that can obtain good results even with very limited datasets. The steps in the algorithm can be described in the following way:

```
while unresolved speeds
  Solve a reduced system for the variables that are well represented.
  if successful solve
    Update the matrix and right hand side with the new results.
  else
    Replace bad variable with linear interpolation from nearest neighbors.
  end
end while
```

Remark 1. To define the reduced system in the first step, we first look at the sums of the elements in each column of A_i. Columns that have sums that are less than a tolerance (100 was used) times the largest column sum are considered to be too small. Then we find the equations (rows) where the elements in the "small" columns are negligible. Together, these form a new system for the well-represented columns.

Remark 2. If an equation has a negative right-hand side after the update, it is discarded because it no longer can provide information about the remaining variables that need to be positive.

Remark 3. If a computed speed is negative, we know that this variable cannot be determined directly from the data. There can also be cases where the number of remaining equations is too small in relation to the number of variables left to determine. In this case, because of lack of information, we assume a linear model for the speed of the failing variable in relation to the speeds of the closest neighbors. This gives us a relation

$$c_i(r-1) - 2c_i(r) + c_i(r+1) = 0,$$

which can be used for interpolation of the middle variable or extrapolation of either of the other two variables.

When all speeds have been computed, they can be converted into time per task under contention at a level r through

$$t_i(r) = \frac{1}{c_i(r)}.$$

4.2.2. The Resource Model and Allocations. The available resources in a system with k different resources are defined as $\pi = (p_1, \dots, p_k)$, where p_i is the available amount of resource i. We can express the resources in any unit that we find practical. A typical choice is to let a task of type i require one unit of resource i. Then the available resource p_i tells us how many tasks of type i are allowed to execute concurrently. In this way, the resource definition becomes equivalent with the allocation of tasks to worker threads, and here we will refer to π as an allocation. However, we only use this for the numbers of threads, not for the actual location. Note that when we talk about allocations, we allow p_i to be real numbers, not only integers. As an example, $p_i = 1/2$ indicates that task i can use one worker thread half the time. For the resource construct, this does not make sense directly but can be implemented such as by connecting two available resources so that they are exclusively available.

We require that each $p_i > 0$; otherwise, tasks of type i will never be allowed to run. Furthermore, we require that $\sum_{j=1}^{k} p_i \leq p$, where p is the total number of threads. We will examine all possible allocations of the k resources. If we require p_i to be integers, the total number of unique allocations is

$$N_k^p = \binom{p}{k} = \frac{p!}{k!(p-k)!}. \tag{4}$$

As an example, for $k = 2$ constrained resources and $p = 4$ cores, we get $\pi_1 = (1, 1)$, $\pi_2 = (1, 2)$, $\pi_3 = (1, 3)$, $\pi_4 = (2, 1)$, $\pi_5 = (2, 2)$, $\pi_6 = (3, 1)$. If we instead let the smallest unit that we can assign be $1/s$ and let m of the resources be exclusive and assigned to a whole thread each, we get

$$\hat{N}_k^p = \binom{(p-m)s}{k-m}. \tag{5}$$

For large p, these numbers can become large. However, it is not likely that we want to manage more than two or three independent resources. Furthermore, if p is large, it is likely that we only set aside a smaller number of cores $\hat{p}$ for the resource constrained tasks. Another aspect to consider is that large multicore systems typically have a substructure, for example, in terms of sockets. Some resources will then be managed per socket, breaking down the problem into smaller subsets.

The actual occupation of the threads (in the theoretical schedule) at any given time will depend on which tasks types remain to be executed. At the beginning of the execution, we have $n_i \equiv n_i(0)$ tasks of each type. These numbers will change with time as tasks are executed. We define the conditional allocation $\tilde{\pi} = (q_1, \dots, q_{n_t})$, given a model allocation π and the current task numbers. Note that the conditional allocation also involves the unconstrained task types.

The resource constrained tasks are scheduled according to the resource model as long as there are unconstrained tasks to occupy the other threads. This yields

$$q_i = p_i, \quad i \leq k, \quad n_i > 0, \quad \sum_{j=k+1}^{n_t} n_j > 0. \tag{6}$$

When there are no more unconstrained tasks, we have two options. The additional threads can be left idle. This is the only option for exclusive resources. Alternatively, the resource constraint can be allowed to be overridden by task stealing. Then we assume that the resulting distribution is proportional to the constrained allocation. This is reasonable if stealing has equal chance of being successful at each occupied core:

$$q_i = \begin{cases} p_i, \; i \leq k, \; n_i > 0, \; n_j = 0, \; j > k, & \text{no task stealing}, \\ \alpha p_i, \; i \leq k, \; n_i > 0 \;\; n_j = 0, \; j > k & \text{with task stealing}, \end{cases} \tag{7}$$

where $\alpha = p/p_r$ and $p_r = \sum_{\{i|n_i>0\}} p_i$. The unconstrained tasks are scheduled with an equal share of the time for each type. This assumption is not crucial, as these tasks are expected to execute at approximately the same speed with any configuration. Let n_z be the number of unconstrained task types with $n_j \neq 0$, $j > k$. Then

$$q_i = \frac{p - p_r}{n_z}, \quad i > k, \quad n_i > 0. \tag{8}$$

4.2.3. Estimating the Total Execution Time. To determine which task type we will run out of first, we need to find the time required to execute all remaining tasks of each type under the current allocation. The time to execute n_i tasks of type i with allocation q_i is given by

$$T_i(q_i) = \frac{t_i(q_i)\, n_i}{q_i},$$

where the effective time per task in the case of noninteger q_i is computed as

$$t_i(q_i) = \beta\, t_i(\lceil q_i \rceil) + (1-\beta)\, t_i(\lfloor q_i \rfloor),$$

and where the fraction of time that we can run on one extra core is $\beta = (q_i - \lfloor q_i \rfloor)$. To handle the case when $q_i < 1$, we also define $t_i(0) = t_i(1)$. By inverting the relation, we get the number of tasks of type i that can be executed in a time interval of length T:

$$\eta_i(T, q_i) = \frac{T}{t_i(q_i)} q_i.$$

Algorithmically, the expected execution time can be computed in the following greedy-like way, where as many tasks as possible are "scheduled" in each step:

```
for j=1 to $\hat{N}_k^p$ // Loop over all possible allocations
  $T_j = 0$, $n_i$ is initialized to total task number of type $i$
  while any $n_i > 0$ // There are remaining tasks
    Compute $\tilde{\pi}_j$ from (6)–(8) using $\pi_j$ and $n_i$, $i = 1, \ldots, n_t$
    // Use this allocation until one task type runs out
    $\Delta T = \min_i T_i(q_i)$
    $n_i = n_i - \eta_i(\Delta T, q_i)$, $i = 1, \ldots, n_t$
    $T_j = T_j + \Delta T$
  end while
end
```

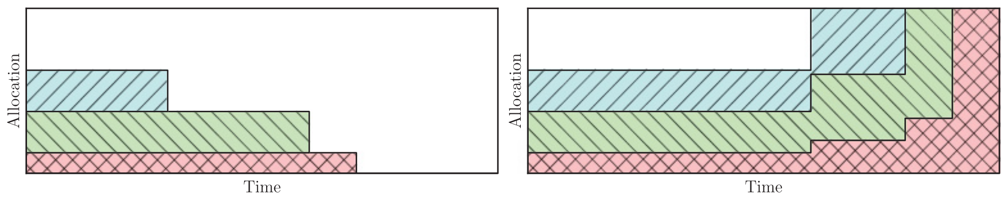


Fig. 6. Schematic of the conditional allocation in a resource-constrained theoretical schedule. White represents the unconstrained task types. The resource allocation is in both cases $\pi = (2, 2, 1)$. In the left-hand image, there are more than enough unconstrained tasks, whereas in the right-hand image, they run out and the resource constrained tasks are distributed over all cores.

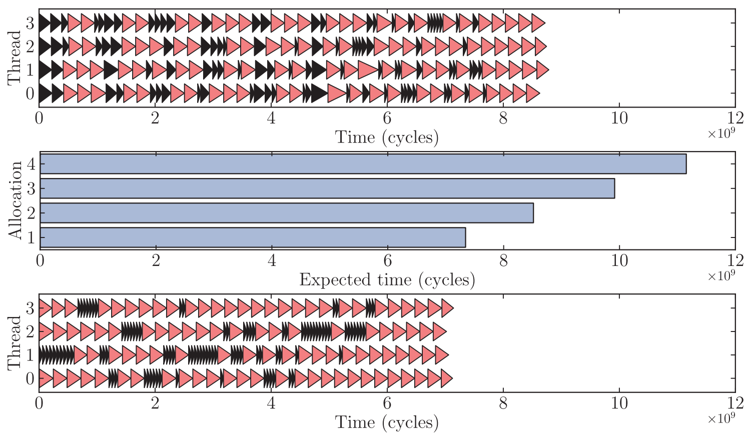


Fig. 7. Top: An unconstrained execution trace with four threads and two task types. The execution times of the black tasks vary a lot due to contention. Middle: The predicted total execution times with different resource allocations. Bottom: The resulting execution trace when only one resource-constrained task can run at a time.

Figure 6 shows a schematic picture of how the conditional allocation may change over time depending on the mix of tasks. To test the prediction method, we go back to the example in Figure 4 with one resource-constrained task type. Figure 7 shows the unconstrained schedule that was used for the prediction of the task execution times, the predicted total execution times when one to four threads are allocated to the resource constrained tasks, and the resulting schedule when the resource-constrained tasks are limited to run one at a time. The total time for the original schedule falls between the two and three resource cases, and in fact, if the average concurrency over time of task type 2 is computed, it is around 2.8. The predicted time for one resource and the total time for the resulting execution trace are almost identical, at 0.0073 and 0.0071, respectively. The time reduction between the two execution traces is 19%. This is lower than the highest rough estimate of 30%, but very close to the more conservative estimate of 21%, where the two exceptionally short times were excluded. More experimental results are provided in Section 5.

4.3. Multiple Resources

In the previous subsection, we made the simplified assumption that only contention with tasks of the same type affect the task execution times. If we instead assume that there is multiple resource contention and that contention on one resource affects performance for tasks that need another type of resource as well, then we need to modify the way in which we estimate task execution speed.

First of all, we need to estimate a different speed variable for each possible unconstrained allocation π_i. This means that we can allow both $p_i = 0$ and $p_i = p$ as possible allocations for a task type. We choose to neglect cases when one or more threads are unoccupied, which for an efficient scheduler should be a very small portion of the execution time. This then gives us N_k^{p+2} speed variables for each task type. As is done in Equation (3), we can divide each task of type i into intervals with different allocations and collect them into an equation for the different speeds.

For the jth task of type i, we get

$$\sum_{q=1}^{N_k^{p+2}} c_i(\pi_q)\delta t_{ij}(\pi_q) = w_i. \tag{9}$$

In the same way as before, we collect the equations for all tasks into a linear system of equations for each task type. In the multiple resource case, we have a much larger number of variables. Therefore, to get good-quality estimates, we either need more data than for the independent resource case or need to reduce the number of variables. Typically, all allocation states are not present in the execution trace, so we can drop all columns that correspond to empty states. This reduces the system to a more manageable size.

After estimating the performance of all available allocation states, we can then proceed to estimate the total execution time for the states for which we have information in the same way as we did previously.

We tested this on a constructed example with three task types. The first task type is computational with few memory accesses, the second task type acquires a lock (mutex) 5 million times, and the third type reads approximately 200MB of memory with an unstructured access pattern. Figure 8 shows the task execution times resulting from an unconstrained execution. The sensitivity test gives $\sigma(1) \approx 0.02$, $\sigma(2) \approx 19.8$, and $\sigma(3) \approx 0.1549$, and the rough time reduction estimate gives $R \approx 63\%$. Clearly, the mutex task is extremely sensitive to contention. We use the full analysis method to estimate the variation in task execution time depending on contention for all three task types. If we use only the data in Figure 8, the best estimated resource allocation is $\pi = (1, 1, 2)$ with an execution time of 73Gcycles. The second best allocation becomes $\pi = (2, 1, 1)$ with a time of 113Gcycles. The real execution times for these allocations are 57 and 112 Gcycles. The data that we get from a schedule for this problem is difficult to analyze because the mutex task dominates the picture and because some task combinations are overrepresented (lack of variation). To improve the approximation, we added three more execution traces with even less variation in the data. The estimated task execution times are displayed in Table I. The resulting estimated total times for the same allocations become 64 and 112. The best result is closer to the real value. However, in the latter case, the estimated task execution times are not reliable even if the total time is accurate.

4.4. Theoretical Best-Case 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 only independent resources so that the execution time of a task depends on the task type and how many other tasks of the same type that are executing concurrently.

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 of a given type are executed across all threads until there are none left, whereas in the best-case scenario, tasks that require a resource are combined with other tasks that are

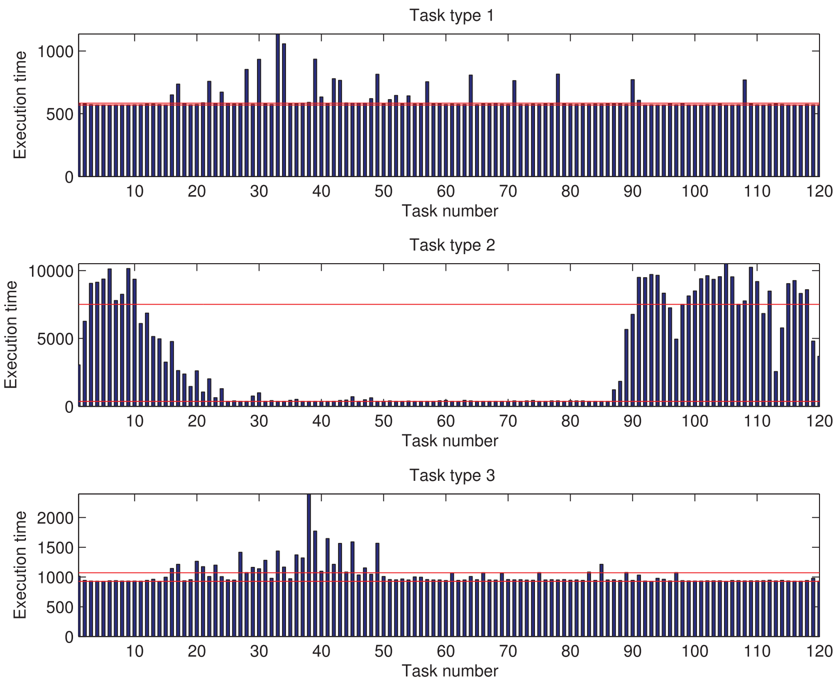


Fig. 8. Execution time variation for the computational task (top), mutex task (middle), and bandwidth-dependent task (bottom). The solid lines show the minimum execution time and the third quartile time.

Table I. Estimated Task Execution Times in Cycles for the Multiple Resource Example

	Task type 1				Task type 2				Task type 3			
p_2 \ p_3	0	1	2	3	0	1	2	3	1	2	3	4
0	–	(196)	831	700					–	1,327	3,050	1,011
1	(226)	904	523		–	(113)	366	–	(177)	888	(484)	
2	571	595			3,931	1,673	(1)		966	951		
3	573				3,436	16,559			933			
4					13,913							

Note: A dash indicates missing data, and a number in a parenthesis is not reliable because it is lower than the single-thread task execution time.

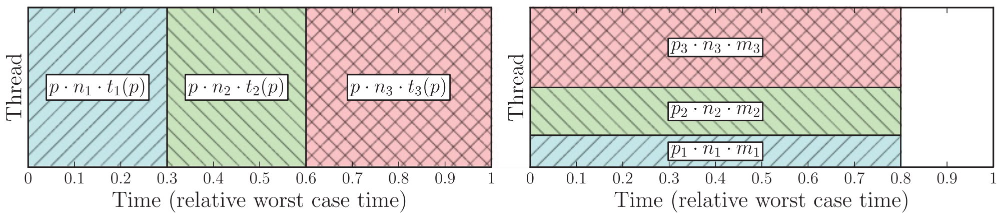

Thread
$p \cdot n_1 \cdot t_1(p)$
$p \cdot n_2 \cdot t_2(p)$
$p \cdot n_3 \cdot t_3(p)$
Time (relative worst case time)
$p_3 \cdot n_3 \cdot m_3$
$p_2 \cdot n_2 \cdot m_2$
$p_1 \cdot n_1 \cdot m_1$

Fig. 9. The worst and best cases for running tasks that share resources. In the worst case, each task type is run across all p threads, and there is a performance penalty due to resource contention. In the best case, at most q_i tasks of type i tasks run concurrently, and all tasks run at their optimal speed.

independent of this resource. In addition, in the best case, the mix of tasks should be ideal so that the total execution time for each task type, when running at optimal speed, is equal. The time to execute each task type i across p_i threads is then expected to be the smallest possible total execution time, assuming that each task is executing at the measured minimum time per task m_i. That is,

$$t_{\text{best}} = \frac{n_1 m_1}{p_1} = \frac{n_2 m_2}{p_2} = \cdots = \frac{n_{n_t} m_{n_t}}{p_{n_t}}.$$

If the times are not equal, there are either too many resource-constrained tasks or too many resource-independent tasks. If there are too many resource-independent tasks, a smaller fraction of the total runtime will be affected by taking resources into account when scheduling, and the gain will be proportional to how large this fraction is. If there are too many resource-constrained tasks, the schedule must either contain idle threads or resource-constrained tasks that oversubscribe the resource. Again, the total gain will at least be proportional to the fraction of the total execution time where there are enough tasks to fill all threads without oversubscribing any resources. The total gain might also be higher if oversubscribing a resource is disallowed, and leaving some threads idle is more efficient than oversubscribing the resource.

In the worst-case scenario, all resource-constrained tasks are executed at the same time on all p threads, each with an execution time of $t_i(p)$ so that

$$t_{\text{worst}} = \sum_{i=1}^{n_t} \frac{n_i t_i(p)}{p}.$$

Under the optimal conditions, where $n_i m_i / p_i = n_j m_j / p_j$ for all i, j, the maximal speedup is

$$\frac{t_{\text{worst}}}{t_{\text{best}}} = \sum_{i=1}^{n_t} \frac{p_i}{p} \frac{t_i(p)}{m_i}. \tag{10}$$

In the next section, we will compare our experimental results against this upper bound.

5. EXPERIMENTAL RESULTS

The experimental results with four threads were run on a single-socket machine with a four-core Intel 2600K processor with an 8MB shared cache and hyperthreading disabled. It should be noted that many resources, such as bandwidth, are connected with one socket. Therefore, using a multisocket machine would give similar per-socket performance. The rest of the experiments were conducted on a two-socket system with two quad-core Intel Xeon 5520 (Nehalem 2.26GHz, 8MB cache) processors.

5.1. 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 100MB of memory (more than fits in the cache) from one location to another and a *computation* task that performs dummy computations and has very few memory accesses.

Since the memory bandwidth is shared between the cores, the copy tasks are likely to be sensitive with respect to bandwidth. By letting a single or few such tasks run at the same time, and schedule tasks whose performance does not depend on memory bandwidth on the other threads, we can reduce contention and increase efficiency.

To enforce such 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

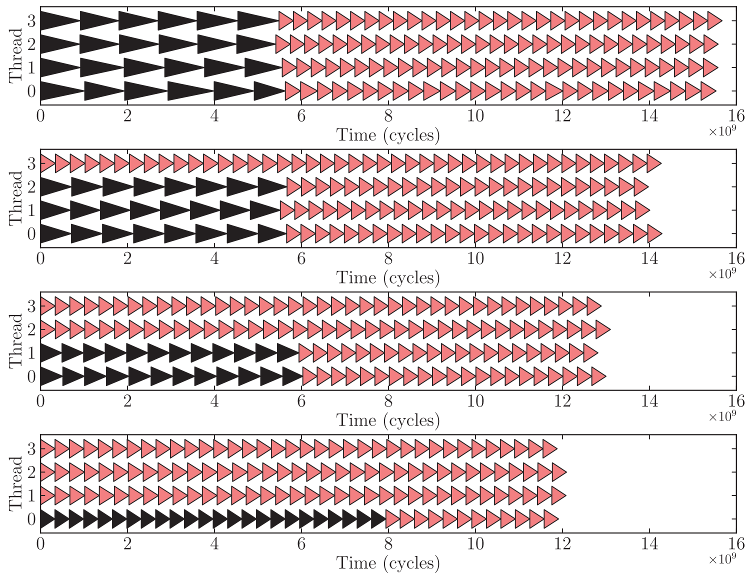


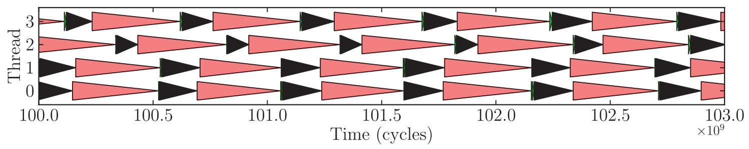
Fig. 10. Memory bandwidth benchmark. Execution traces with different limits on the number of copy tasks that may execute at a time, starting with allowing at most four concurrent copy tasks in the first trace, down to at most a single copy task in the last row. Black triangles represent copy tasks, and red triangles represent computation tasks.

this resource is then varied to investigate how the runtime of the application behaves. Figure 10 shows execution traces of these runs. Here we selected several copy tasks that are large enough to show the behavior clearly, then we selected the number of computation tasks to be large enough to occupy all threads even if only a single copy task is allowed to run at once. All executions in Figure 10 perform the same amount of work and have the same number of tasks; however, 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 were run concurrently on all four threads. If we instead measure the performance in terms of speedup over serial execution, we go from a speedup of 2.3 (58%) to 3.7 (93%) when limiting the copy tasks to run one at a time.

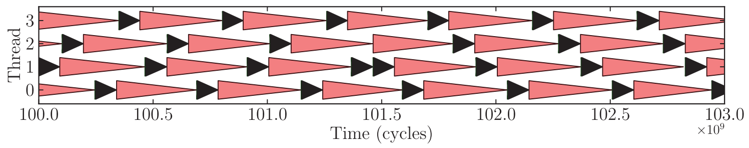
With the parameters for this problem, we can compute the best theoretical speedup using Equation (10), resulting in approximately 47%. However, for about 35% of the execution time, there are only calc tasks to execute, which will be unaffected by the resource constraints. By taking this into account, we should expect a reduction in execution time that is lower than 30.6%, which is just above the 30% that we achieved. The rough estimate of 2 yields 28%.

The second example presented that uses the new resource-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 11 shows small parts of execution traces from the application when resources are used to limit the file accesses, and for comparison as well when tasks are scheduled freely.



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



(b) Using resources to constrain file-accessing tasks (black) to at most one at a time

Fig. 11. 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.

```
for (int i = 0; i < N; i++) {
  potrf(i); // $A_{ii}$ = Cholesky($A_{ii}$)
  for (int j = i+1; j < N; j++)
    trsm(i,j); // $A_{ji} = A_{ji}A_{ii}^{-T}$
  for (int j = i+1; j < N; j++)
    for (int k = i+1; k <= j; k++)
      gemm(i,j,k); // $A_{jk} = A_{jk} - A_{ji}A_{ki}^T$
}
```

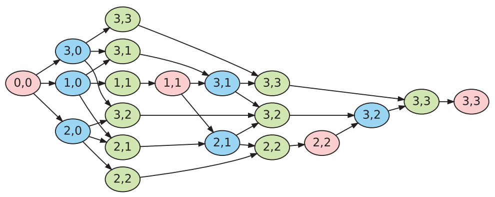


Fig. 12. Left: The Cholesky factorization algorithm for a block matrix. Right: The corresponding DAG for 4×4 blocks.

As can be seen in Figure 11(a), 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 threads 0 and 1.

We then introduce a resource for file accesses required by both the read and the write task, and part of an execution trace from running with these constraints is shown in Figure 11(b). In this trace, the read tasks are always fast and notably more uniform in length than in Figure 11(a).

By using resources to constrain the read and write tasks, reads became 30% faster and writes 40% faster, whereas the compression tasks were unaffected. Since most of the total runtime is spent on compression, speeding up the read and write tasks have a smaller impact but still give a total speedup of 6%. The constrained execution time was predicted with an accuracy of 0.4% from an analysis of the unconstrained schedule.

5.2. Scheduling Using the Core Resource

To test the core resource, we use Cholesky factorization with hierarchical tasks as an example. The algorithm, its (parent) tasks, and their dependencies are described in Figure 12. Inside the parent tasks, the matrix blocks are themselves treated as block matrices and the algorithm is divided into subtasks of a finer granularity.

In the experiments, we use a matrix of size $10{,}000 \times 10{,}000$ elements divided into 5×5 blocks a the highest level, then each of these blocks is subdivided into 4×4 smaller blocks of size 500×500 elements for the subtasks. The experiment is run on a compute node with two quad-core Intel Xeon 5520 (Nehalem 2.26GHz, 8MB cache)

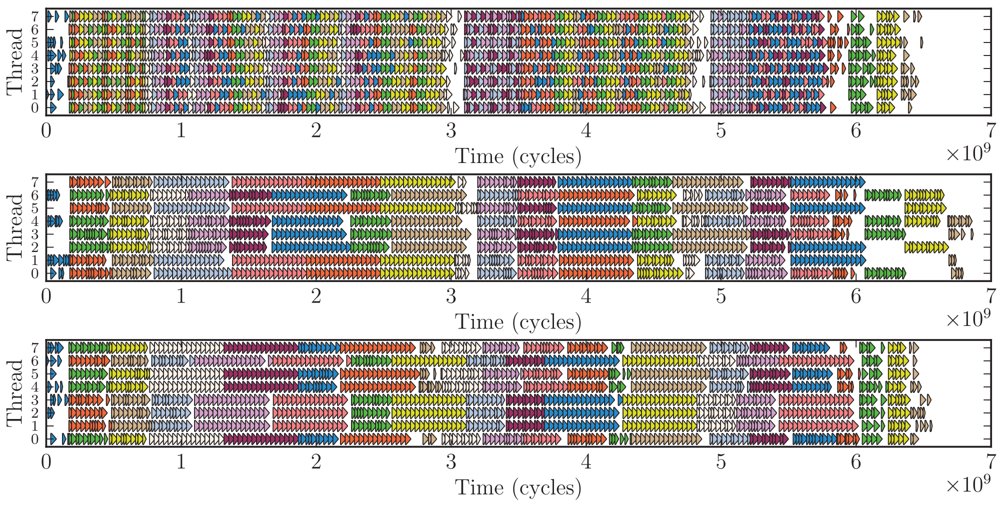


Fig. 13. Execution traces for the Cholesky factorization without using resources (top), with each parent task requiring four cores (middle), and with relaxation of the resource requirement for the first and last few tasks (bottom). Only subtasks are displayed. Subtasks of the same color belong to the same type of parent task.

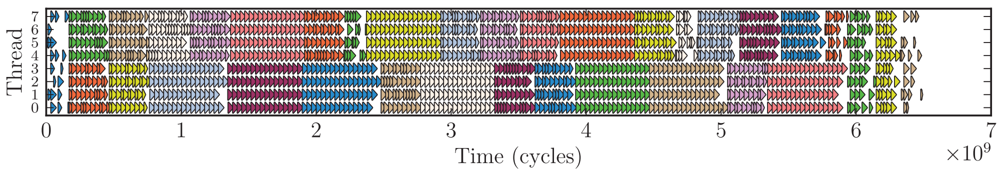


Fig. 14. Execution trace for the Cholesky factorization, where all worker threads that work on the same parent task are restricted to run on the same socket.

processors. The NUMA distance of cores in different sockets is twice that of cores in the same socket.

Figure 13 shows results for the Cholesky example with and without using resources. The top trace shows what happens when all tasks are scheduled without constraints. Subtasks from different parent tasks are completely mixed on all threads. For the middle trace, we have introduced the core resource, and each parent task requires four cores. Task stealing is allowed within the group of reserved cores. As can be seen in the figure, two parent tasks at a time are allowed to run and the subtasks are not mixed. At the end of the trace, the level of parallelism is too low (cf. Figure 12) and only one parent task can run at a time. In the bottom trace, we have removed the resource constraint for the first and last few tasks, allowing them to use all eight cores. This indicates that it may be important to have a mechanism for relaxing resource constraints in parts of the execution.

When parent tasks are allowed to compete with each other, they can take longer to complete due to interference, and since the tasks further down in the DAG may depend on these, it is possible that the whole execution becomes delayed due to an unresolved dependency. With resources, each of the parent tasks are fast. Without resources, there are some delays in this example. However, they do not affect the total execution time.

In Figure 14, all worker threads that work on subtasks belonging to the same parent task are required to be located on the same socket. The reason for this is to increase locality and minimize penalties due to NUMA effects. On the particular hardware used,

```
// 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. 15. Code for generating the three tasks used to process an element.

and with this application, no significant difference in execution time can be observed. However, based on Al-Omary et al. [2013], we believe that on a more nonuniform multisocket machine, the difference would be larger.

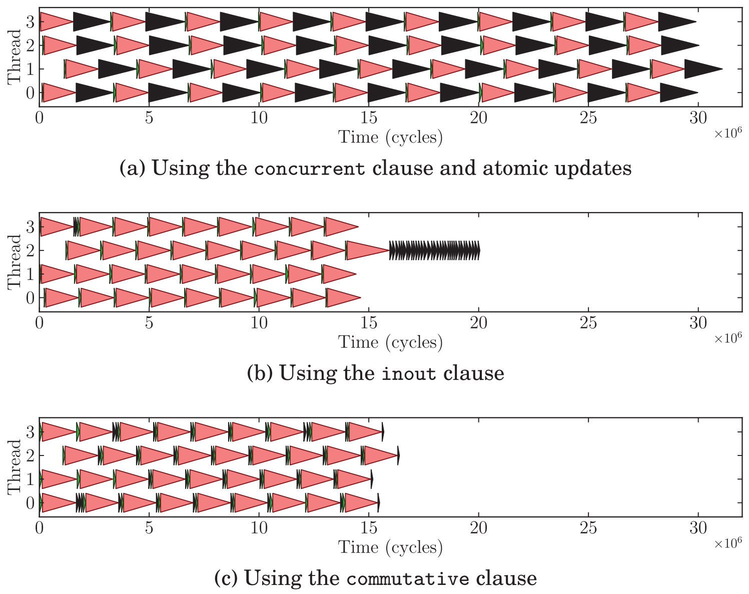
5.3. The Commutative Clause

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 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—that is, the resource is write access to a data location. If this accumulation is performed by a task that uses the `inout` clause, the runtime system must accumulate the forces in the order the tasks were created. This introduces false dependencies and is the situation illustrated on the left-hand side of Figure 1. Using the `commutative` clause instead, the runtime system is free to schedule the accumulation of the forces in any order and can run the tasks as on the right-hand side of Figure 1 instead.

We performed a simulation of 8,192 particles divided into blocks of 512 particles each, run for four time steps. This experiment was run on a system with two four-core Intel Xeon 5520 processors. 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 SpecFEM3D, an application software for simulating earthquakes. 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 pseudocode in Figure 15. 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 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: (1) using the `concurrent` clause or (2) 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 runtime system cannot assume that it is safe to reorder the tasks. This may cause bad scheduling.



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

(b) Using the `inout` clause

(c) Using the `commutative` clause

Fig. 16. Execution traces for the SpecFEM3D benchmark using three different methods for writing the results into the shared vector.

An alternative solution is to introduce a resource of which there is only one available and to 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 require them. This is more elegant, requires less writing, and does not rely on using resources for correctness.

Figure 16 shows small parts of execution traces for the three methods. A single gather-compute-scatter cycle is shown, although the full execution contains many such steps together with other tasks. Here, the black triangles represent the scatter tasks, the red tasks are computation tasks, and the barely visible green tasks are gather tasks. In Figure 16(a), the scatter tasks are much slower than in the other methods because of the additional costs from using the compare-and-swap instruction. Figure 16(b) 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 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 have dependencies on the computation tasks, the scatter task that depends on the last computation task is encountered, and the remaining scatter tasks must wait to the end. On average, it is expected that about half of the scatter tasks must execute at the end; however, 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 one that is 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 other threads are idle.

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

6. CONCLUSIONS

We have proposed a simple model for managing resources in task parallel programming environments. The `resource` construct provides a method for constraining tasks that share common resources from being scheduled too many at the same time. By declaring tasks as `commutative`, we provide exclusive access to a resource while allowing the tasks to be executed in any order.

Both resource-aware scheduling and commutative tasks have been implemented in the OmpSs programming model and tested on different applications and for different types of resources. For a bandwidth-limited example, we could reduce the execution time by 30%, whereas serializing I/O accesses in a JPEG application resulted in a 6% reduction. For commutative updates, we could reduce the execution times by 13% in an *n*-body simulation application and 10% in SpecFEM3D, a software for earthquake simulations. We have also tested the concept of a core resource that provides exclusive access to a group of cores, which may be used for improving data locality and reducing NUMA effects.

The improvements in performance that we have observed are comparable to those obtained in research on OS scheduling in the SMT context as reported in Section 1.3. It is clear from our results the the effects of contention can depend very much on the application and the type of contention. To determine the best scheduling approach, profiling and prediction tools are needed.

We have shown that rough estimates and quite sophisticated predictions can be obtained simply by analyzing execution traces from unconstrained scheduling of the application. The methods require the scheduling to be somewhat irregular to allow sampling of all scheduling constellations. However, even with little variation in the data, we were able to find resource allocations that improved performance.

ACKNOWLEDGMENTS

Parts of the experiments were performed on the Kalkyl cluster provided by SNIC through UPPMAX under project p2009014.

REFERENCES

Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series* 180, 1, 012037. http://stacks.iop.org/1742-6596/180/i=1/a=012037

R. Al-Omary, Guillermo Miranda, Xavier Martorell, Jesus Labarta, Rosa M. Badia, D. Keyes, and Hatem Ltaief. 2013. Dense Cholesky factorization on NUMA architectures with socket-aware work stealing. Submitted.

Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2, 187–198.

Eduard Ayguadé, Rosa M. Badia, Pieter Bellens, Daniel Cabrera, Alejandro Duran, Roger Ferrer, Marc González, Francisco D. Igual, Daniel Jiménez-González, and Jesús Labarta. 2010. Extending OpenMP to survive the heterogeneous multi-core era. *International Journal of Parallel Programming* 38, 5–6, 440–459.

Major Bhadauria and Sally A. McKee. 2010. An approach to resource-aware co-scheduling for CMPs. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS'10)*. ACM, New York, NY, 189–199. DOI:http://dx.doi.org/10.1145/1810085.1810113

Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. 2008. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st Annual*

IEEE/ACM International Symposium on Microarchitecture (MICRO 41). IEEE, Los Alamitos, CA, 318–329. DOI:http://dx.doi.org/10.1109/MICRO.2008.4771801

Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. 2010. Contention-aware scheduling on multicore systems. *ACM Transactions on Computer Systems* 28, 4, 8.

Gérman Ceballos and David Black-Schaffer. 2013. Shared resource sensitivity in task-based runtime systems. In *Proceedings of the 6th Swedish Workshop on Multicore Computing*. 61–64.

Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'08)*. IEEE, Los Alamitos, CA, Article No. 4. http://dl.acm.org/citation.cfm?id=1413370.1413375

Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. OmpSs: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters* 21, 2, 173–193.

Pontus Ekberg and Wang Yi. 2012. Outstanding paper award: Bounding and shaping the demand of mixed-criticality sporadic tasks. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS'12)*. 135–144. DOI:http://dx.doi.org/10.1109/ECRTS.2012.24

David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. 2011. Cache pirating: Measuring the curse of the shared cache. In *Proceedings of the International Conference on Parallel Processing (ICPP'11)*. 165–175.

Ali El-Moursy, Rajeev Garg, David H. Albonesi, and Sandhya Dwarkadas. 2006. Compatible phase co-scheduling on a CMP of multi-threaded processors. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*. DOI:http://dx.doi.org/10.1109/IPDPS.2006.1639376

Alexandra Fedorova, Margo Seltzer, Christoper Small, and Daniel Nussbaum. 2005. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC'05)*. 26. http://dl.acm.org/citation.cfm?id=1247360.1247386

Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. 2009. Cache-aware scheduling and analysis for multicores. In *Proceedings of the 7th ACM International Conference on Embedded Software (EMSOFT'09)*. ACM, New York, NY, 245–254. DOI:http://dx.doi.org/10.1145/1629335.1629369

Charles E. Leiserson. 2010. The Cilk++ concurrency platform. *Journal of Supercomputing* 51, 3, 244–257.

Cristoph Niethammer, Colin W. Glass, and José Gracia. 2012. Avoiding serialization effects in data/dependency aware task parallel algorithms for spatial decomposition. In *Proceedings of the IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*. 743–748. DOI:http://dx.doi.org/10.1109/ISPA.2012.109

Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. 2008. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the IEEE International Conference on Cluster Computing*. 142–151.

Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. 2009. Hierarchical task-based programming with StarSs. *International Journal of High Performance Computing Applications* 23, 3, 284–299.

Allan Snavely and Dean M. Tullsen. 2000. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. ACM, New York, NY, 234–244. DOI:http://dx.doi.org/10.1145/378993.379244

Martin Tillenius. 2012a. *Leveraging Multicore Processors for Scientific Computing*. Licentiate thesis. Department of Information Technology, Uppsala University.

Martin Tillenius. 2012b. SuperGlue Project. Retrieved October 28, 2014, from http://www.it.uu.se/research/scicomp/software/superglue

Martin Tillenius and Elisabeth Larsson. 2010. 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*.

Martin Tillenius, Elisabeth Larsson, Rosa M. Badia, and Xavier Martorell. 2013. Resource aware task scheduling. In *Proceedings of the 8th International Conference on High-Performance and Embedded Architectures and Compilers (Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures)*. ACM, New York, NY.

Hans Vandierendonck, George Tzenakis, and Dimitrios S. Nikolopoulos. 2011. A unified scheduler for recursive and task dataflow parallelism. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 1–11.

Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2009. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. *Parallel Computing* 35, 3, 178–194. DOI:http://dx.doi.org/10.1016/j.parco.2008.12.006

Asim YarKhan, Jakub Kurzak, and Jack Dongarra. 2011. *QUARK Users' Guide: QUeueing and Runtime for Kernels*. Technical Report ICL-UT-11-02. ICL, University of Tennessee, Knoxville, TN.

Afshin Zafari, Martin Tillenius, and Elisabeth Larsson. 2012. 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*.

Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. 2012. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys* 45, 1, Article No. 4. DOI:http://dx.doi.org/10.1145/2379776.2379780

Received May 2013; revised April 2014; accepted June 2014