SWITCHES: A Lightweight Runtime for Dataflow Execution of Tasks on Many-Cores

ANDREAS DIAVASTOS, University of Cyprus
PEDRO TRANCOSO, University of Cyprus and Chalmers University of Technology

SWITCHES is a task-based dataflow runtime that implements a lightweight distributed triggering system for runtime dependence resolution and uses static scheduling and compile-time assignment policies to reduce runtime overheads. Unlike other systems, the granularity of loop-tasks can be increased to favor data-locality, even when having dependences across different loops. SWITCHES introduces explicit task resource allocation mechanisms for efficient allocation of resources and adopts the latest OpenMP Application Programming Interface (API), as to maintain high levels of programming productivity. It provides a source-to-source tool that automatically produces thread-based code. Performance on an Intel Xeon-Phi shows good scalability and surpasses OpenMP by an average of 32%.

CCS Concepts: • **Theory of computation** → **Parallel computing models**; • **Software and its engineering** → **Runtime environments**;

Additional Key Words and Phrases: Many-core, runtime system, dataflow, tasks, parallel programming, SWITCHES

ACM Reference format:
Andreas Diavastos and Pedro Trancoso. 2017. SWITCHES: A Lightweight Runtime for Dataflow Execution of Tasks on Many-Cores. *ACM Trans. Archit. Code Optim.* 14, 3, Article 31 (August 2017), 23 pages.
https://doi.org/10.1145/3127068

1 INTRODUCTION

Parallelism is currently the means to achieve high performance. But, scaling the number of threads and cores alone does not result in improved application performance. In order to expose maximum available parallelism and achieve increasing performance, scalability needs to be addressed at different levels and applications need to be developed using models that adapt better to their needs (e.g., the Task model and the Dataflow paradigm). Assuming that an application is expressed by an algorithm that exhibits enough parallelism, improving the performance is determined by a combination of the following factors: scalable architectures, low-overhead runtime systems, efficient use of the available resources, locality-aware execution, and productive programming tools.

Figure 1 presents how the state-of-the-art OpenMP runtime handles fine-grain task-parallelism on a many-core system. The graph (on the right) shows the speedup over the sequential execution of a synthetic application as we increase the number of tasks but maintain the total problem size constant. On the left of the figure is the dataflow graph of the synthetic application that is

New Paper, Not an Extension of a Conference Paper.
Authors' addresses: A. Diavastos and P. Trancoso; emails: {diavastos, pedro}@cs.ucy.ac.cy.

https://doi.org/10.1145/3127068

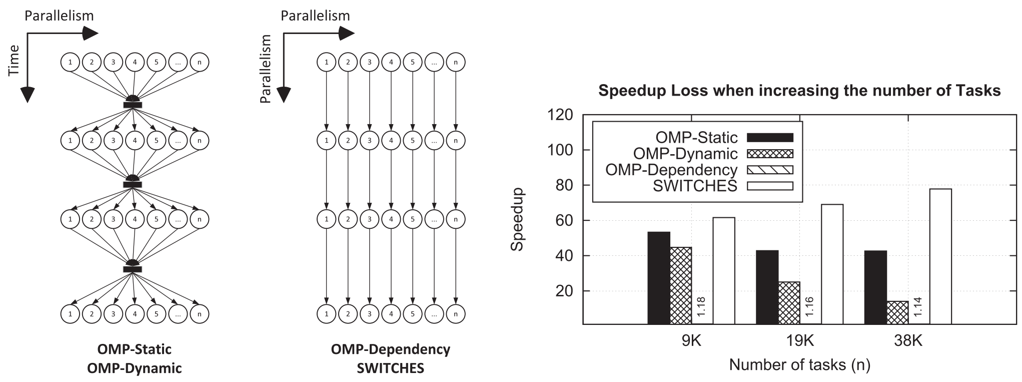


Fig. 1. Speedup on the Intel Xeon Phi with 240 threads, varying number of tasks, and constant input compared to the sequential execution of a synthetic application. On the left we show the dataflow graph two applications with four loops (the source code is presented later in Figure 4). The one on the left does not have dependences across the loops but instead uses synchronization barriers (*OMP-Static* and *OMP-Dynamic*). The one on the right has dependences across iterations of different loops (*OMP-Dependency* and *SWITCHES*). The graph on the right shows OpenMP losing performance as we divide the problem size into more tasks (*n*).

composed of four parallel loops with a synchronization point between each one and each loop is divided into *n* tasks. In theory, keeping the total work constant should keep the speedup unchanged. Nevertheless, results show that as we increase the number of tasks the speedup is reduced. This can be caused by two factors: (1) the amount of work per task and (2) the synchronization overhead of the runtime system. As long as the amount of work per task is large enough, the synchronization time can be hidden, but as we increase the number of tasks for a constant workload, the work per task is reduced while the synchronization primitives increase. This results in increased synchronization overhead and consequently a loss in performance.

With the application in Figure 1, we try to show the impact of the runtime overhead on performance using different OpenMP scheduling policies. In *OMP-Static* and *OMP-Dynamic* the iterations of each loop are executed in parallel but synchronization points (barriers) must be added between the loops (dataflow graph on the left). In the former, tasks are scheduled statically, while in the latter the runtime handles everything dynamically. The static implementation has much less performance loss as most scheduling operations are handled during compilation. The dynamic scheduler, on the other hand, has a lot more work to do synchronizing all the tasks during execution. *OMP-Dependency* and *SWITCHES* remove the barriers and add dependences across the iterations of the loops (dataflow graph on the right), but as the results show, the overhead of runtime dependence resolution in OpenMP dominates the execution time. The implementation of lightweight synchronization primitives in *SWITCHES*, in combination with a static runtime system reduces the overheads and achieves the highest performance. Its low-overhead design also results in scaling performance even when increasing the number of tasks. Although this is just a simple application, it presents a real problem of current task-based runtime systems with fine-grain parallelism, which can affect performance scalability in current and future many-core systems.

SWITCHES introduces a lightweight runtime system that supports the task-based dataflow model as a way to scale performance on many-core architectures. The dataflow paradigm has been proposed a long time ago [9, 10] but has only recently been widely adopted, as it is one of the most effective ways to exploit large-scale parallelism [4, 15, 19, 29–31, 35, 43, 46]. It is mostly used in the form of task-based parallel systems, which allow for efficient handling of synchronization, memory access, and communication latencies. This leads to better utilization of resources and increased performance in large-scale High-Performance Computing (HPC) systems with hundreds to thousands of cores [44].

SWITCHES implements a triggering system for dependence resolution, which distributes the runtime operations to all participating threads. It applies compile-time static scheduling and assignment policies in order to reduce runtime overheads. It improves locality, by providing simple mechanisms that allow flexible definition of task granularity. It provides constructs for declaring cross-loop dependences, consequently increasing the application coverage and improving the performance of applications with parallelism across different loops. It also provides for task resource allocation constructs that make efficient use of the hardware units and improve performance. *SWITCHES* maintains high levels of programming productivity by extending the latest standard of the widely used OpenMP Application Programming Interface (API) [8], while providing for a software tool that automatically produces parallel code.

SWITCHES is evaluated on a 61-core Intel Xeon Phi, using both task- and data-parallel applications from different benchmark suites. Results show good performance scalability for all applications tested and a considerable speedup increase compared to the best OpenMP results.

The main contributions of this work are the following:

- a lightweight, scalable runtime system for task-based dataflow execution on HPC many-cores, called *SWITCHES*;
- extensions to the OpenMP v4.5 API to support explicit resource allocation and cross-loop dependences with variable granularity on the `taskloop` directive;
- a source-to-source tool (*Translator*) that uses the source code with directives to produce parallel pthread code embedded with the runtime, which can be compiled with any commodity compiler;
- a comparison of *SWITCHES* performance with state-of-the-art OpenMP on a real HPC many-core using task- and data-parallel applications.

This article is organized as follows. In Section 2, we present the concepts of the dataflow model. In Section 3, we describe the *SWITCHES* execution model and runtime, while in Section 4 we present the extensions to the OpenMP API we propose. In Section 5, we describe *the Translator*. In Section 6, we describe our experimental setup and in Section 7 we present our results. In Section 8, we discuss the related work, and finally, in Section 9 we summarize our conclusions.

2 THE DATAFLOW MODEL

The original dataflow model was proposed by Jack Dennis in the early 1970s [9, 10] as an alternative to the control-flow (von Neumann) model. Instructions in a dataflow program are executed when all their input operands are available, creating an asynchronous (non-blocking) execution. The availability of the operands is expressed using data dependences, which define a dataflow graph representing the order of the execution. Using the dataflow graph and the input operands required by each instruction, one can expose parallelism in a program. Many systems today try to explore fine-grain parallelism by using dataflow-like models as a way to achieve high performance and utilization on large-scale many-core systems with hundreds to thousands of cores [15, 19, 29, 31].

The main advantage of the dataflow model is the ability to exploit maximum parallelism from an application by exposing fine-grain tasks. This large degree of parallelism can be exploited to hide the latency of memory accesses. Unlike other models, dataflow does not require synchronization mechanisms as the correctness of the execution is assured by enforcing the data dependences. Nevertheless, exploiting fine-grain parallelism was also the limiting factor in the success of this model in past implementations, mostly due to the overheads in enforcing the data dependences at

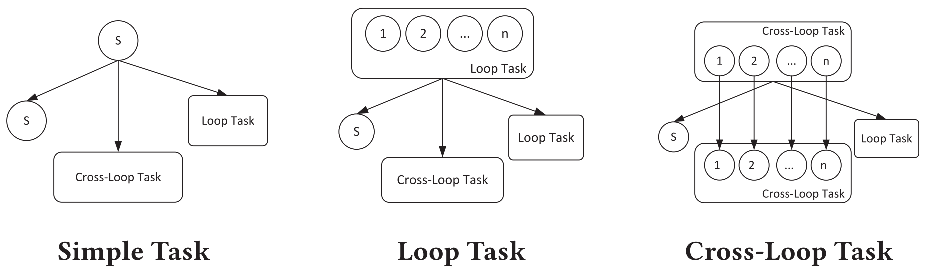


Fig. 2. Type of tasks in the *SWITCHES* execution model.

the instruction level. More recent attempts managed to overcome these overheads by adopting the model at a coarser granularity (e.g., tasks), consequently achieving high performance [15, 29–31].

Another relevant factor toward using dataflow for increasingly large systems is its disciplined access to shared data. Assuming a task-based implementation of dataflow, it is ensured by the model that no concurrent tasks will be modifying the same data, as this would result in a data dependence violation [9, 10, 32]. Therefore, in a shared-memory system the dataflow model does not require a hardware implementation of a cache-coherence protocol as access on shared data may be coordinated by the model itself. Correctness of the application will be assured simply by updating cached data to and from main memory on completion of a task, i.e., flushing updated values to memory and invalidating cached copies in other cores. The fact that hardware cache-coherence is not required by the dataflow model, allows for increasing performance scalability on many-cores as shown in [13], reduces hardware costs, and improves energy-efficiency [48].

3 THE SWITCHES SYSTEM

SWITCHES is built to satisfy two major requirements: (1) the scalability of application performance, and (2) the reduction runtime overheads. To achieve scalability, it implements the dataflow model as a fully de-centralized runtime by evenly distributing the scheduling operations to all software threads. To reduce runtime overheads, tasks are assigned to threads statically at compile-time. In addition, each task holds its own scheduling structures that will be loaded in the scheduler during execution. This design requires minimum support for dynamic scheduling of tasks, consequently reducing runtime overheads. *SWITCHES* is publicly available for download in [11].

3.1 The Execution Model

SWITCHES is a task-based model that allows the definition of dependences on every task in an application. The dependences form a producer/consumer relationship between tasks, where one task produces data and others consume it. The dependence itself shows the role of each task, i.e., if there is a dependence from *taskA* to *taskB*, then *taskA* is the producer and *taskB* is the consumer. The dependences also define when a task can be executed, thus additional synchronization primitives are not required. Dependences are resolved only after all producers have completed execution, and only then can a consumer task execute.

There are three types of tasks in a *SWITCHES* program (Figure 2): (1) `Simple-Tasks`, which represent a non-iterative structured block, (2) `Loop-Tasks`, which represent iterations of a *for* loop, and (3) `Cross-Loop-Tasks`, which represent iterations of a *for* loop with dependences on iterations of other loops. The difference between (2) and (3) is the format of their dependences. `Loop-Tasks` are iterations of a loop that will execute in parallel but are isolated from the rest of the tasks. That is, if another task (of type (1) or (2)) depends on a `Loop-Task`, then it will wait for the entire loop to finish before starting execution. `Cross-Loop-Tasks` are iterations that have direct

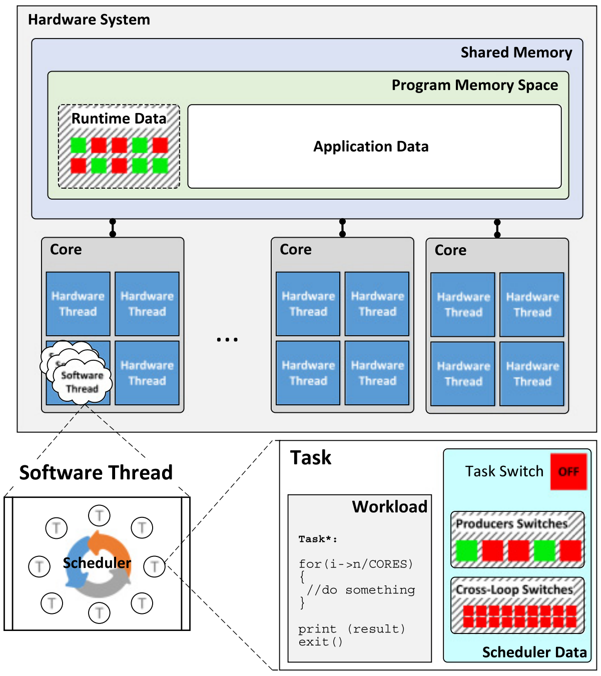


Fig. 3. The architectural design of *SWITCHES*.

dependences on iterations of other loops. Assuming a granularity of one, the iterations of a loop have a one-to-one dependence with the iterations of another loop, therefore providing cross-loop parallelism.

The level of granularity in such a scenario can be defined by the programmer. The granularity is increased by packing consecutive iterations into a single task and is particularly efficient as it favors spatial locality. A smaller granularity, on the other hand, improves the parallelism by increasing the number of tasks (fine-grain parallelism). It is important to notice that, although cross-loop iteration dependences and task-loop granularity are supported by the latest release of the OpenMP standard (v4.5), the combination of the two clauses is not.

3.2 The Runtime

The scheduling structures of the runtime are called *switches* and are implemented using simple memory constructs, stored in shared-memory and cross-referenced by the tasks using them. Switches are Boolean variables that denote whether a task has executed (ON) or not (OFF). Each task is assigned its own unique switch that can only be updated by the thread that executes the task (*single-writer*). A task is ready to execute only when all its producers' switches are set to ON. Each thread checks the producers' switches of a task to be executed (*multiple-readers*) and if all are set to ON, the task is executed.

Since *SWITCHES* is following a *single-writer/multiple-readers* model for all runtime data, a protection mechanism for simultaneous access (e.g., locking) on switches is not required. Because it is built on top of a shared-memory system, switches are manually updated (from producers caches) to main memory and self-invalidated in consumers caches. This process adds little overhead to the execution as the information accessed by each thread is statically assigned at compile-time.

The architectural design of *SWITCHES* in Figure 3 shows that the data of both the runtime and the application are stored in the shared-memory of the system. One or more software threads can

ALGORITHM 1: The *SWITCHES* Scheduler

```
1:  procedure SCHEDULE
2:      while !empty(ownTaskQueue) do
3:          task ← getNextTask(ownTaskQueue)
4:          ready ← checkSwitches(task)
5:          if ready = TRUE then
6:              execute(task)
7:              turnSwitchON(task)
8:              removeTaskFromQueue(task, ownTaskQueue)
9:          end if
10:     end while
11:     resetSwitches(ownTaskSwitches)
12: end procedure
```

be assigned to each core (or hardware thread), depending on the total number of threads defined by the user. A software thread consists of the *SWITCHES Scheduler* and the tasks it will execute. Each task is a combination of the application source code, its own switch, and remote references to all switches of its producers.

The *SWITCHES* runtime system is implemented in the form of a software unit called *Scheduler*. The *Scheduler* is responsible for monitoring producers' switches, updating task switches, and triggering ready tasks for execution.

3.3 The Scheduler

Most task-based parallel systems that exist today provide for a single, centralized scheduler that monitors shared-data during execution and resolves dependences based on reads and writes on those data [8, 15]. Depending on the number of tasks, and the dependences defined, this can lead to an overload of the scheduling unit which can be expensive (as results show in Figure 1).

To solve this, in *SWITCHES* each thread is equipped with its own *Scheduler*. Every *Scheduler* is statically assigned a number of tasks to execute and only has knowledge of their incoming dependences, based on the cross-references that each task holds on its producers' switches (Figure 3). It does not require global information of the application, making it simple and scalable regardless of the number of threads, as the more threads used in the execution the less information each one will hold.

Algorithm 1 shows the pseudo-code of the *Scheduler*. The *Scheduler* is assigned a number of tasks at compile-time in its `ownTaskQueue` and executes a busy-wait loop (line 2) until all assigned tasks are finished. The *Scheduler* first gets a waiting task from its `ownTaskQueue` (line 3). It checks the switches of its producers and if all are set to ON, it triggers the task for execution (line 4). If at least one producer has not yet finished, the task is not ready for execution and the *Scheduler* gets the next task from its `ownTaskQueue`. To minimize long waits of tasks that are ready to execute, the `checkSwitches()` operation (in line 4) stops immediately when it finds the first producer switch that is not set to ON. Also, as soon as a task finishes execution, the *Scheduler* sets its switch to ON (line 7) and removes it from the `ownTaskQueue` (line 8), so it will not be checked again.

When a thread finishes all its assigned tasks (its `ownTaskQueue` is empty), the *Scheduler* breaks the busy-wait loop, resets all its tasks' switches (line 11), exits the current parallel section, and returns to the main program. Switches are reset to avoid creating multiple instances of the same code and the same switches in case there is a repetitive execution of the same tasks later in the program (e.g., a function that is called multiple times). To avoid resetting switches that are still

Table 1. The Basic *SWITCHES* Programming API. In Bold We Emphasize Our Extensions Compared to OpenMP[a]

Pragma Directives	*Supported Clauses (Optional)*	*Description*
`#pragma omp parallel`	`num_threads(NUMBER)` `private(list)`	Defines a parallel function to be executed by `NUMBER` threads and `private` is used to declare variables as private to each thread.
`#pragma omp parallel for`	`private(list)` `num_threads(NUMBER)` `schedule(type:[,CHUNK])` `reduction(OPERATION:list)`	Defines a parallel loop with each iteration considered a task. All clauses have the same functionality as in OpenMP.
`#pragma omp task` **`#pragma omp master`**	`private(list)` `firstprivate(list)` `depend(type:list)`	Defines a task and its dependences using the `depend()` clause. `private` and `firstprivate` have the same functionality as in OpenMP. The `master` clause in *SWITCHES* extends a normal `task` that will be executed by the master thread.
`#pragma omp taskloop` `#pragma omp for`	`private(list)` `firstprivate(list)` `grainsize(CHUNK)` **depend**`(type:list)` **num_threads**`(NUMBER)` **schedule**`(type:[,CHUNK])` **reduction**`(OPERATION:list)`	Defines a loop task with `grainsize` defining the number of consecutive iterations assigned to each task. The depend clause is used to apply dependences to the iterations, while the `num_threads` clause explicitly allocates resources for a loop task. The `schedule` clause defines the scheduling policy of the loop (`static` or `cross`). A `taskloop` directive can also support a reduction function using the `reduction` clause.

[a]More directives are implemented but in this table we present the most relevant ones to the applications tested.

in use by other threads, resetting takes places only after all tasks of a parallel section have been completed.

4 THE SWITCHES API

The API of *SWITCHES* is an extension to the latest OpenMP v4.5 [36]. The applications are written in C/C++ with tasks and dependences declared using compiler directives. We chose the API of OpenMP because it is widely used for parallel programming in shared-memory environments and since v4.0 provides a complete set of directives for declaring tasks and dependences. Consequently, it satisfies the productivity goal of quickly porting applications for execution with *SWITCHES*. The *SWITCHES* API implements all task directives from OpenMP v4.5 and extends them by using existing clauses from non-task directives such as `num_threads`, `schedule`, and `reduction`. Table 1 summarizes the basic directives used for writing a *SWITCHES* program and highlights in bold the extensions implemented. It is also important to notice that even though at this time we only implement a portion of the OpenMP API, the *SWITCHES* runtime will not become heavier if we implement the entire API as we follow a static runtime approach. All necessary scheduling information is produced during the compilation of each application keeping the runtime system simple and light. For conventional OpenMP implementations (that use a dynamic runtime system) to be more flexible with different applications most scheduling data is produced at runtime and require more information to be kept by the runtime system.

4.1 Compiler Directives

Any tasks declared in a *SWITCHES* program must be enclosed in a parallel section using a `#pragma omp parallel` directive. A parallel section works in the same way as in OpenMP and denotes that all enclosed tasks are to be executed in parallel according to their dependences. The use of `single` directive is not required as in OpenMP, since tasks will only be created once statically at compile time and started when the master thread reaches the specific parallel section. Any code written in a parallel section that is not enclosed in a `task` directive will be executed by all participating threads as in OpenMP. Different parallel sections are executed sequentially in the order found in the program, just like in OpenMP. Also, all data are considered shared across the entire program unless declared otherwise using the optional `private` or `firstprivate` clauses. The former creates a new empty variable private for every task, while the latter will also initialize it with the value of the corresponding global variable at the time the task is called.

The `#pragma omp task` directive defines a structured block as a task of type `Simple-Task`. The `depend(type:list)` clause defines the data processed by the task. The list argument holds the name of the data (variables, arrays, etc.). The `type` argument indicates whether the data will be read (`in`), written (`out`), or both (`inout`).

The `#pragma omp taskloop` directive declares iterations of a loop as tasks of type `Loop-Task` or `Cross-Loop-Task`. Similarly to OpenMP, the `grainsize` clause defines the number of consecutive iterations to be packed for each task. In contrast to OpenMP, in *SWITCHES* the user can explicitly define the number of threads to use for the execution of a taskloop with the `num_threads` clause. This can be beneficial for applications with limited parallelism where loops with little work do not occupy all available resources. By using this clause it is possible to utilize cores, that would otherwise be idle or free cores that do not do any work at all. *SWITCHES* also supports the definition of dependences on a `taskloop` directive. The `depend` clause is used as described earlier for the `task` directive. The *SWITCHES* `taskloop` directive also implements the `schedule` clause (from the `#pragma omp for` directive). This clause is used to define a policy for scheduling loop tasks to threads. Two policies are supported at the moment: (1) `static`, which is similar to OpenMP and (2) `cross`, which declares the iterations of the loop as tasks with dependences. With `static`, iterations of the loop are declared as tasks of type `Loop-Tasks`, while with `cross` iterations are declared as `Cross-Loop-Tasks`. Note that, in such case all associated loops must have the `cross` policy.

If the `static` scheduling policy is used along with the `depend` statement, a single dependence will be declared on the entire loop. If the `cross` policy is used, dependences will be applied on individual iterations of the associated loops and create a scenario with *Cross-Loop Iteration Dependences*. The `CHUNK` parameter is used to define the number of consecutive iterations to be packed in a single task (similarly to the `grainsize` clause). The final extension of *SWITCHES* compared to OpenMP is the `reduction` clause where a loop of tasks can be declared as having a reduction operation after all iterations are completed. *SWITCHES* supports all OpenMP standard reduction operations with any number of reduction variables declared in the `list` option.

4.2 Cross-Loop Iteration Dependences

As explained earlier, OpenMP does not allow the definition of dependences on a `taskloop` directive. It supports dependences on loop iterations only by using the `task` directive within the body of a `for` loop. This limits the level of granularity of tasks with dependences to one iteration per task, affecting the locality of the data and limiting the performance in certain applications. To increase the level of granularity of a taskloop, OpenMP offers the `grainsize` clause but since dependences cannot be defined on a `taskloop` directive, a synchronization barrier will be inserted at the end of the loop. This adds additional synchronization overheads and ignores cross-loop parallelism that may exist between two different loops.

With the extensions on the `taskloop` directive described earlier, *SWITCHES* allows the packing of consecutive iterations into a single task and can then define dependences on these tasks (`Cross-Loop-Tasks`). Thus, *SWITCHES* provides locality for tasks of the same loop and at the same time offers asynchronous execution of tasks from different loops (cross-loop parallelism) by removing all barrier synchronization. Note that this is applied to different loops, in contrast to OpenMPs' *doacross* technique that uses the `ordered` directive to serialize iterations in nested loops [36].

4.3 Task Resource Allocation

The distribution of resources within a parallel region in OpenMP is a responsibility of the runtime. OpenMP only allows explicit definition of resources in a `parallel` construct, which denote how many threads will participate in the specified parallel region. But, the increased parallelism and dependences in task-based models provide the user with information that can be vital to the performance of an application. Dependences can be a natural limitation in the scalability of an algorithm, and using the entire pool of execution units leads to wasting resources. A possible solution in a dynamic runtime system is the use of a work-stealing approach that reassigns tasks to different threads during execution. Such a technique tries to balance the workload in the available resources but it also increases the work of the scheduler and depending on the application it may increase runtime overheads. Such a solution cannot be used in a static runtime system as it would diminish all the benefits of a static implementation. To address this problem, in *SWITCHES* we employ the explicit allocation of resources per task. *SWITCHES* extends the `taskloop` directive to use the `num_threads` clause to allow for explicit definition of threads to be used for the execution of `Loop-Tasks` and `Cross-Loop-Tasks`. Therefore, *SWITCHES* allows for finer-grain allocation of resources that overcomes the scalability boundary in algorithms with data dependences and efficiently uses the available hardware resources.

4.4 Example

Figure 4 shows an example that makes use of the extensions proposed. It depicts a kernel with two loops that have cross-loop iteration dependences and split the execution resources. Figure 4 also presents a graphical diagram of the same example. *Loop A* is producing data in array `k1`, while *Loop B* consumes data from `k1`, thus defining a dependence. Analyzing the algorithm, we find that the iterations of the two loops have a one-to-one dependence on array `k1` only, thus only `k1` needs to be declared in the `depend` statements. All other data used by the two loops could have been declared as well but the system would have ignored them, as there are no true dependences on any other data. To define the cross-loop dependence, we declare the scheduling policy of the two loops as `cross` and in the `depend` clause we add the indexes of the continuous dependent iterations. Note that the `CHUNK` in the `schedule` clause must be the same number as the end-index of the dependences, so that consecutive dependent iterations are packed to the same task. Summarizing, the example shows that each loop consists of tasks with chunk size of two iterations per task, and every task of

```
#pragma omp parallel num_threads(10)
{
   // Loop A
   #pragma omp taskloop schedule(cross, 2)
                        \ depend(out: k1[0:2])
                        \ num_threads(5)
   {
      for (i = 0; i < SIZE; i++)
      {
         yt[i] = 0.0;
         for (j = 0; j < SIZE; j++)
            yt[i] += c[i][j]*y[j];
         k1[i] = h*(power[i]-yt[i]);
      }
   }

   // Loop B
   #pragma omp taskloop schedule(cross, 2)
                        \ depend(in: k1[0:2])
                        \ depend(out: k2[0:2])
                        \ num_threads(5)
   {
      for (j = 0; j < SIZE; j++)
      {
         yt[j] = 0.0;
         for (k = 0; k < SIZE; k++)
            yt[j] += c[j][k]*(y[k]+0.5*k1[j]);
         k2[j] = h*(power[j]-yt[j]);
      }
   }
   ...
}
```

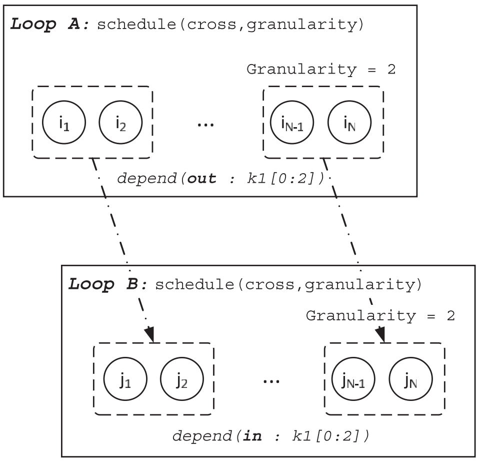


Fig. 4. Cross-loop Iteration Dependences example and diagram. This example is part of a larger synthetic application that is based on a differential equation kernel (RK4, presented in Section 6). The Data-flow graph of the entire application is shown in Figure 1.

Loop A has a direct dependence on the corresponding task of *Loop B*. Finally, we explicitly specify that only five threads will be used to execute the tasks of each loop. Thus, the two loops split the available resources of the parallel region (10 threads) to execute their tasks in parallel.

5 THE TRANSLATOR

The translation of a directive-based application to a *SWITCHES* parallel program is automatically done by a source-to-source tool called the *Translator*. The *Translator* is a software tool built using Lex and Yacc that parses the C/C++ directive-based code and produces threaded parallel code. Pragma directives can be inserted anywhere in the code and also in multiple files. The *Translator* also produces error and warning messages when directives or clauses are not used correctly. Such messages include directive syntax errors, unknown parameters in clauses, mismatching resource allocation values, among others.

The *Translator* takes four major inputs from the programmer: (1) *the source code* files, embedded with pragma directives, (2) *the scheduling policy*, which defines how tasks are divided to participating software threads, (3) *the assignment policy*, which denotes how software threads are assigned to hardware resources, and (4) *the number of threads*, which execute the application. The number of software threads can be as many the Operating System (OS) allows. After parsing the directives in the source code, it automatically extracts the tasks and their dependences and produces the synchronization graph of the application. It then executes a *transitive reduction* operation on the

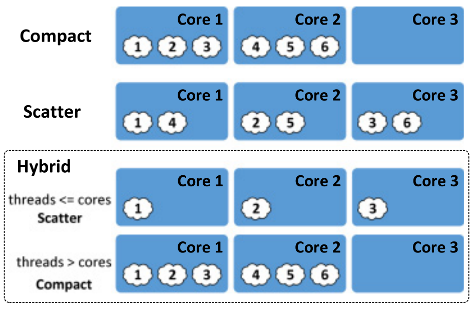
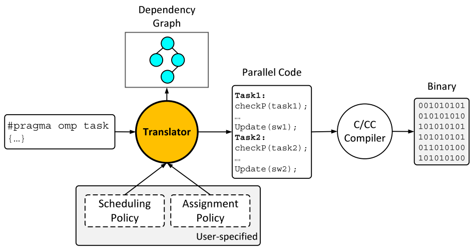


Fig. 5. *SWITCHES* assignment policies.



```
#pragma omp task
{…}
```

```
Task1:
checkP(task1);
…
Update(sw1);
Task2:
checkP(task2);
…
Update(sw2);
```

```
001010101
010101010
101010101
101010101
011010100
101010100
```

Fig. 6. The process of translating an application.

graph to remove redundant dependences that might have been implicitly or explicitly declared [2]. This optimization reduces the size of the graph and the runtime data structures that are produced, thus reducing the workload of the scheduler and minimizing runtime overheads.

To further reduce runtime overheads compared to other dynamic parallel systems, many of the scheduling operations in *SWITCHES* are moved to compile-time and executed by the *Translator*. The *Translator* will use the final graph to impose the scheduling and assignment policies by statically mapping tasks to threads and threads to cores, respectively. At the moment, *SWITCHES* schedules tasks based on the availability of the threads using a *round-robin* scheme. During the program translation the user can choose from three predefined assignment policies: `Compact` and `Scatter` that can also be found in the *icc* compiler as affinity policies (`KMP_AFFINITY`) or the *gcc* compiler with the names `close` and `spread`. The third and new assignment policy is called `Hybrid` as it implements a combination of the previous two. We present each policy in the example of Figure 5 where we assume a system with three cores and three hardware threads per core. The `Compact` policy assigns software threads close to each other occupying the hardware thread units of a core before moving to the next one. This policy does not utilize all cores when the software threads are less than the hardware threads of a system. To evenly utilize all available cores, the `Scatter` policy assigns the software threads to the cores in a round-robin way. The `Hybrid` policy is a combination of the two previous, where `Scatter` is used when the software threads are fewer than the number of the available cores to increase processor utilization, while `Compact` is used when the number of threads is more than the available cores to favor locality of shared data in the caches. The `Hybrid` policy is implemented for simplicity of the execution of the *Translator*. Depending on the number of threads defined by the user at the translation stage, the system automatically determines the appropriate policy to use. All policies use the software *thread-id* to assign the threads to the cores.

After invoking the *Scheduler* it creates the output parallel source code as shown in Figure 6. It can also generate a fully detailed synchronization graph for possible debugging of the application. The produced source files consist of the tasks code, the creation of the software threads (pthreads), and the *SWITCHES* runtime system. The output source code can be compiled with any commodity C/C++ compiler.

6 EXPERIMENTAL SETUP

SWITCHES is evaluated on a set of seven data- and task-parallel applications. Applications were chosen based on references from other evaluations of task-based runtime systems and many-core processors ([12, 16, 38, 42, 44]). Details for all the applications and their input sizes are shown in

Table 2. Experimental Workloads Description and Dataset Sizes

Benchmark	Description	Dataset Sizes (Computation Iterations) *DS1*	*DS2*	*DS3*
Q12	Nested-Loop Join from TPC-H [7]	$60K \cdot 1.5K$	$60K \cdot 15K$	$600K \cdot 150K$
MMULT	Matrix multiply [42]	$256 \cdot 256$	$512 \cdot 512$	$1{,}024 \cdot 1{,}024$
RK4	Differential equation [42]	4,800	9,600	19,200
SU3	Wilson Dirac equation [12]	$1{,}920K$	$3{,}840K$	$7{,}680K$
Poisson2D	Five-point 2D stencil computation [44]	4,096	8,192	16,384
SparseLU	LU factorization of sparse matrices [16]	$120 \cdot 32$	$240 \cdot 32$	$480 \cdot 32$
OCEAN	Red-Black solution (Gauss-Seidel [47])	$4{,}096 \cdot 4{,}096$	$8{,}192 \cdot 8{,}192$	$16{,}384 \cdot 16{,}384$

Table 2. The Dataset Sizes represent the total number of computation iterations each application executes on its data.

As representatives of data-parallel applications, we use (1) Q12, a C-code version of Query 12 from the TPC-H Benchmark suite [7] that emulates the Scan and Join operations on the data from two tables representing a memory-bound application, (2) MMULT, which implements a matrix multiplication algorithm [42], (3) RK4, which solves a differential equation [42], and (4) SU3, which is a component of the Wilson Dirac equation that involves the multiplication with the gauge-links (vector multiplication of complex C99 numbers) [12].

Task-parallel applications include (1) Poisson2D, a five-point 2D stencil computational kernel of the Poisson equation from the KASTORS Benchmark suite [44], (2) SparseLU, from the BOTS Benchmark suite [16] that computes an LU matrix factorization using sparse matrices creating an imbalance workload, and (3) OCEAN, representing a Red-Black solution of the Gauss-Seidel method [47].

Our main evaluation platform is an Intel Xeon Phi 7120P with 61 cores and 4 threads per core (totaling 244 hardware threads). Note that we only used 60 cores so as to avoid any interference by the OS that always uses the last core of the system. This board has a total of 16GB of main memory and runs at 1.238GHz. To cross-compile applications for the Xeon Phi we use the Intel icc v.17.0.2 compiler (and the corresponding libiomp5 library) with the *-mmic* flag indicating the Many Integrated Architecture (MIC) target. We also tested *SWITCHES* on a smaller system, a 12-core machine with two 6-core (12 hardware threads) AMD Opteron 2427 running at 2.2GHz with an available main memory of 31GB. This system is running an Ubuntu SMP ×86-64 OS with the gcc v.5.4 compiler (with libgomp1 v.6.2). For both compilers, we use the *-O3* optimization flag. The results are presented as *Speedup*, calculated by dividing the execution time of the best sequential implementation of each application with the time of the parallel execution. The execution times are collected using the `gettimeofday` system call that provides a resolution of microseconds. The time is measured from the start of the first parallel function until the last, including all runtime costs (such as thread creation and scheduler initialization). Execution time is determined by the arithmetic average of five consecutive execution runs after removing the outliers (for all experiments, the standard deviation is within 5%).

7 EXPERIMENTAL EVALUATION

The main objectives of the evaluation are (1) to show the performance scalability achieved by *SWITCHES*, and (2) to compare it with the state-of-the-art OpenMP. The OpenMP applications used in the evaluation are the original source codes taken from the suites described in Table 2. In BOTS suite, SparseLU is implemented with both the `parallel_for` and the `omp_single` directives for creating tasks. In our tests, we used the `omp_single` version because it makes the implementation purely task-based and identical to the scenarios tested in KASTORS [44]. In order to provide a fair comparison of the two runtime systems, OpenMP source codes were used as is for the *SWITCHES* evaluation without extra optimizations. Only the syntax of the directives for the scenarios with cross-loop dependences were modified since OpenMP currently does not support it.

As far as Xeon Phi-specific optimizations, only SU(3) uses vector intrinsics for both implementations as it was the only application that already supported it. Because we are studying performance scalability of the runtimes on a large-scale many-core and not the capabilities of the underlying hardware, we chose not to alter the original source codes with hardware-specific optimizations. Scalability results for each application are for the largest dataset. For granularity, we used the value that produces the highest speedup for each application. The input size is defined in the title and the granularity for each runtime tested is shown in parentheses in the key-legend of each chart. In Section 7.4, we show how each system performs for each dataset by presenting results of a weak scaling test. *SWITCHES* results are taken using the *hybrid* assignment policy to increase resource utilization when the number of threads is less than the number of available cores (60).

7.1 Data-Parallel Application

Data-parallel applications (Q12, MMULT, RK4, and SU3) are compared against the OpenMP `parallel-for` using both `static` and `dynamic` scheduling policies (*Static-For* and *Dynamic-For*, respectively). For the scenarios using the `dynamic` policy, we let the runtime system dynamically decide all scheduling options, therefore granularity is not statically changed. The applications are also implemented using OpenMP tasks (*Task*). Note that the version of the OpenMP library used at the time of this work still does not support the OpenMP v4.5 (that includes the `taskloop` directive), therefore to support higher granularities in OpenMP we manually implemented the taskloop scenarios (*Taskloop*) using the `taskgroup` directive, as suggested by the Red Hat Developer Program in [40]. Next to the name of each system in the charts we show in parentheses the granularity level used to achieve the highest performance for the specific system.

Results in Figure 7 show Q12 as an application that benefits more from a task-based runtime system. A task-based implementation provides isolation to the execution of each task, and therefore allows better scheduling of the Q12 workload as the tasks execute independently. Both *SWITCHES* and OpenMP (*Taskloop*) achieve the highest speedup for 240 threads (72× and 67×, respectively) but only in the scenarios where the granularity of tasks is increased. The rest of the OpenMP implementations achieve a maximum speedup at only 180 threads. The work distribution at 240 threads is too fine-grain to hide the runtime overheads of these implementations, while the lightweight runtime of *SWITCHES* achieves the highest performance at 240 threads. Oversubscribing the Xeon Phi to 300 and 360 threads results in degraded performance as it can cause higher resource contention and pipeline latencies [33].

MMULT in Figure 8 shows that different OpenMP implementations achieve the same performance. Also, increasing the number of threads from 180 to 240 results in little benefit for OpenMP. The size of the work assigned to each thread decreases as we increase the number of threads, creating fine-grain parallelism, with the overhead of the OpenMP runtime dominating the execution. The low overhead imposed by *SWITCHES* allows scalability regardless of the number of

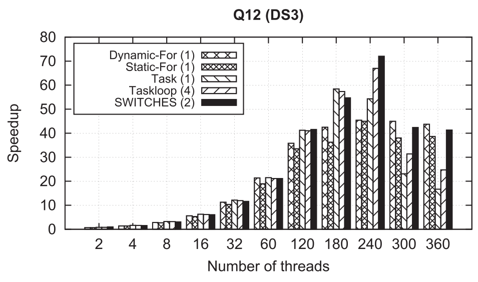


Fig. 7. Q12 speedup on the Xeon Phi.

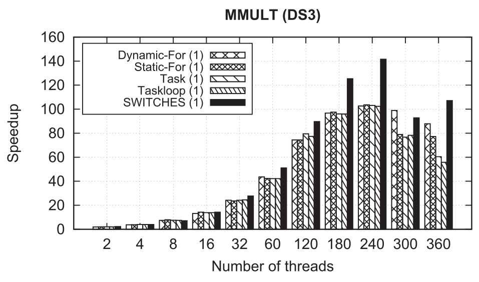


Fig. 8. MMULT speedup on the Xeon Phi.

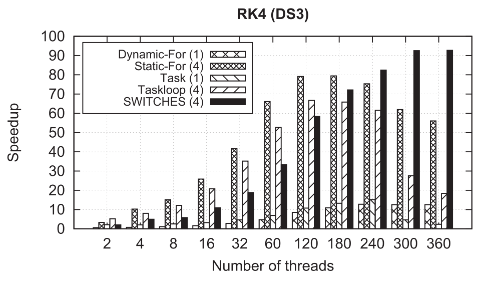


Fig. 9. RK4 speedup on the Xeon Phi.

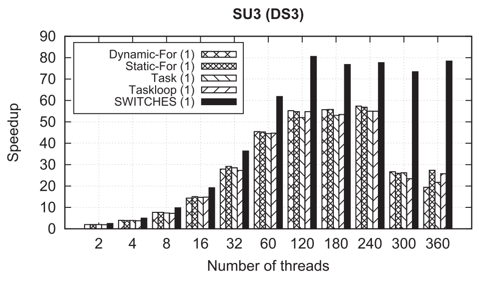


Fig. 10. SU3 speedup on the Xeon Phi.

threads and benefits compared to OpenMP as the number of threads increases. *SWITCHES* achieves a speedup of 141×, while the best OpenMP results is 103× for *Static-For*. Again, oversubscribing the cores reduces the speedup due to overheads imposed due to resource contention.

RK4 (results in Figure 9) is a load-balanced application where each iteration of the containing loops produces the same amount of work. Its algorithm suggests that consecutive iterations use data from consecutive memory locations. Therefore, the best results are produced with the implementations that increase the granularity to favor locality (*Static-For* (79×), *Taskloop* (66×), and *SWITCHES* (92×)). The *Task* and *Dynamic-For* implementations are limited to a maximum of 15× and 10× speedup, respectively, because the runtime does not take into account the data-locality of adjacent tasks. Because each task in RK4 uses different data for its computations, oversubscribing of the cores could hide memory latencies and increase the performance and that is why we see the *SWITCHES* performance improve after 240 threads.

SU3 is another application where we observe the impact of the runtime system on its execution (Figure 10). Although the algorithm implemented limits the scalability of all systems to 120 threads, the low overhead of *SWITCHES* to the execution allows it to achieve a speedup of 81× compared to the best OpenMP result of 57× (*Dynamic-For*). In SU3, we also see a large drop of performance when we increase the number of threads to more than 240, in contrast to *SWITCHES* that maintains steady performance.

7.2 Task-Parallel Applications

SparseLU, OCEAN, and Poisson2D are compared against OpenMP task-based implementations with dependences (*Task-Dep*), and without dependences (*Task* and *Taskloop*). The latter require explicit declaration of synchronization between parallel loops as dependences are not declared.

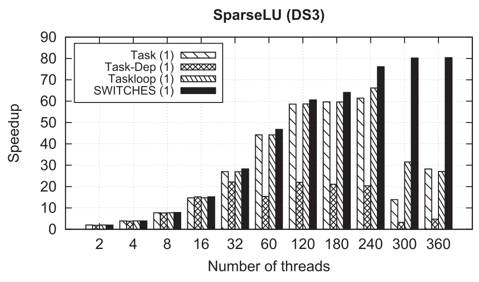


Fig. 11. SparseLU speedup on the Xeon Phi.

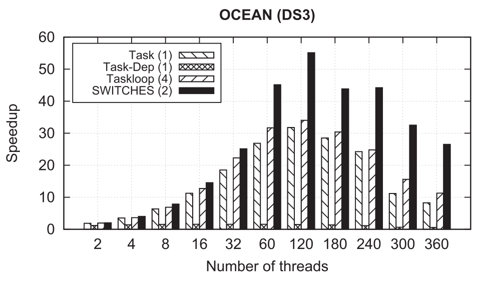


Fig. 12. OCEAN speedup on the Xeon Phi.

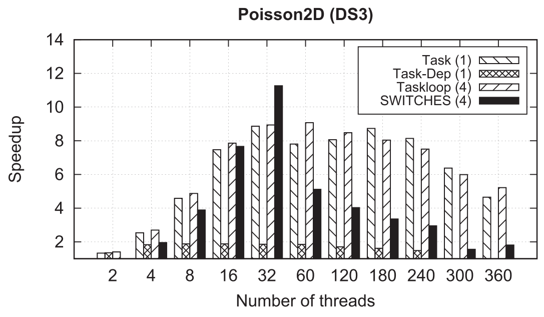


Fig. 13. Poisson2D speedup on the Xeon Phi.

SparseLU is an application that provides cross-loop parallelism between three loops that can be expressed using tasks with dependences. But, results in Figure 11 show that if we apply dependences on OpenMP tasks (*Task-Dep*), the parallelism offered cannot produce enough performance to overcome the overhead of runtime dependence resolution. When dependences are declared in OpenMP, shared data are packed in a single dependence graph and marked for monitoring during execution. This monitoring is managed by a centralized runtime system with its workload increasing as the amount of data to monitor is increased. If we remove the dependences, this overhead is removed but parallelism across loops is not exposed and potential performance is lost. Because *SWITCHES* takes care of the dependences and the scheduling during compilation, it is possible to expose parallelism across loops without incurring additional runtime overheads and increase the performance over all OpenMP implementations, achieving a speedup of 80× compared to the 66× of *Taskloop*. Oversubscribing cores in SparseLU helps *SWITCHES* to improve performance beyond 240 threads, while for OpenMP the performance degrades as it happens in RK4.

OCEAN (Figure 12) has a very balanced workload and using dependences exposes even more parallelism. But the large number of dependences in the algorithm produce too much overhead for the OpenMP runtime to handle efficiently. The static and distributed nature of *SWITCHES* dependence resolution avoids the extra overhead and produces a speedup of 55× for 120 threads. Increasing the granularity of tasks benefits the execution even more and the speedup of *Taskloop* increases to 34× compared to *Tasks'* 31×. The algorithm of Poisson2D (Figure 13) shows a scalability limitation and the maximum speedup is achieved at 32 threads (11× for *SWITCHES* and 9× for *Taskloop*). *SWITCHES* also seems to be losing performance as the number of threads increase beyond that point, while OpenMP maintains steady results. The reason for this is that the workload of Poisson2D is not well balanced by the algorithm, therefore a static implementation such as *SWITCHES* is bound to lose performance. On the other hand, the dynamic scheduler of OpenMP

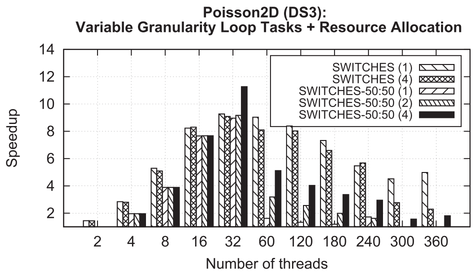


Fig. 14. Performance when using the cross-loop dependence support in combination with increasing granularity and allocation of resources (50:50 means the threads are equally divided between the two loops).

can handle the load imbalance at runtime and maintain the higher performance. To address the issue of load balance in *SWITCHES* we used explicit allocation of resources for the two parallel taskloops of Poisson2D as shown in Section 7.3.

7.3 Explicit Task Resource Allocation

As explained in Section 4.3, one way to solve the low-utilization problem of an application when using a static scheduling runtime system is to explicitly allocate resources for tasks. Figure 14 highlights the benefit this technique in combination with the cross-loop dependences and variable granularity introduced by *SWITCHES* (results in Figure 13 also make use of all these techniques). When we increase the granularity to four iterations per task and at the same time split the resources (threads) among the two loops of the application, the maximum speedup of Poisson2D is increased by almost 24%. *SWITCHES* unveils the cross-loop parallelism and the splitting of the resources allows for better allocation of the hardware resources as the two loops execute simultaneously. This technique also helps to use the resources more efficiently during the execution, especially in applications with scaling limitations such as Poisson2D that do not scale beyond 32 threads.

7.4 Discussion

Figure 15 summarizes all results on the Intel Xeon Phi by showing the highest achieved speedup for each application for all OpenMP policies and *SWITCHES*. In Table 3, we show the execution times of each of these scenarios. To examine the behavior of *SWITCHES* on a smaller system we run the same workloads on an AMD 12-core Opteron processor and present the highest achieved speedups in Figure 16. Overall, the results show that *SWITCHES* is on par with OpenMP for most applications on the AMD system with a small loss of around 10% on two of them but an average increase of 11%. When the applications are scaled to a larger system, *SWITCHES* surpasses OpenMP for all applications, achieving an average of 32% performance increase compared to the best OpenMP results. Results for SU3 on the AMD system are not presented as the implementation used had Intel-specific intrinsics for the Xeon Phi processor. Even though we do not show in these charts, for SparseLU, which is an application from the BOTS benchmark suite [16], we also compared *SWITCHES* against OmpSs. OmpSs showed performance that is on par with OpenMP on both hardware systems and thus the same conclusions drawn for OpenMP are also valid for OmpSs.

In Figures 17 and 18, we present results from a weak scaling analysis of *SWITCHES* compared to all other OpenMP implementations described earlier. In a weak scaling scenario, we test how the runtime systems behave when varying the data input with a fixed number of resources. In these tests, we used the maximum hardware available resources (240 threads) for all applications except Poisson2D where we used the number of threads that achieves the highest performance (which is

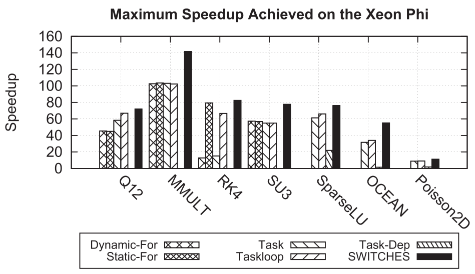


Fig. 15. Best speedup achieved by each system in the Intel Xeon Phi.

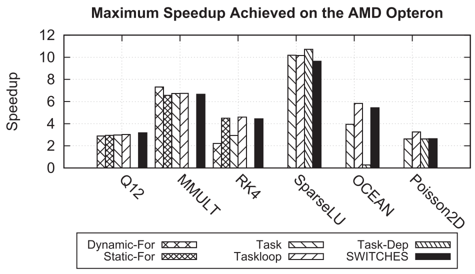


Fig. 16. Best speedup achieved by each system on the AMD Opteron.

Table 3. The Execution Time of Each Scenario that Achieves the Highest Speedup Result. The Input Sizes for All Applications are Those of Previous Scenarios Except for SparseLU on the AMD Which We Used a Size of (96 · 64)

System	Application	Serial	SWITCHES	System	Application	Serial	SWITCHES
PHI	Q12	*373.35s*	*5.18s*	AMD	Q12	*398.15s*	*125.21s*
	MMULT	*78.17s*	*0.55s*		MMULT	*10.74s*	*1.61s*
	RK4	*32.27s*	*0.34s*		RK4	*5.78s*	*1.30s*
	SU3	*3.59s*	*0.048s*		-	-	-
	SparseLU	*6798.37s*	*84.53s*		SparseLU	*40.12s*	*4.16s*
	OCEAN	*14.57s*	*0.26s*		OCEAN	*3.88s*	*0.71s*
	Poisson2D	*12.14s*	*1.07s*		Poisson2D	*4.36s*	*1.65s*

32 threads). We used three different data sizes to monitor the behavior. In general, we notice that for most applications OpenMP closes the performance gap with *SWITCHES* as the data size increases. This happens because with larger data sizes the work per task increases compared to the work of the runtime system. Consequently, OpenMP manages to hide its runtime overheads within the application computation. Looking at the results from the opposite perspective, reducing the data size should expose the runtime system operations and make them more prone to overheads. This happens to OpenMP but not to *SWITCHES* due to its low-overhead runtime implementation. Another important outcome of this study is that for Task-parallel applications with dependences between the executing tasks, OpenMP never surpasses the performance of *SWITCHES* (Figure 18) as the dependence resolution mechanism of *SWITCHES* performs better than that of OpenMP (*Task-Dep*) and at the same time the extra parallelism produced by using task dependences keeps the performance of *SWITCHES* at high levels compared to the OpenMP scenarios that do not make use of them (*Task* and *Taskloop*).

Taking all results into account, we see that a static implementation of a Task dataflow model produces less overheads and has little impact on the parallel execution of the tested application. The de-centralized architecture of *SWITCHES* allows for performance scalability regardless of the number of threads, while its lightweight implementation of handling dependences minimizes the runtime overhead of resolving dependences and minimizes the negative effect on the application execution. This is obvious in many of our results as we see applications performance when using OpenMP in many cases to stop scaling at 180 threads as the work per task becomes too little to hide the scheduling overheads. On the other hand, for some applications performance increases

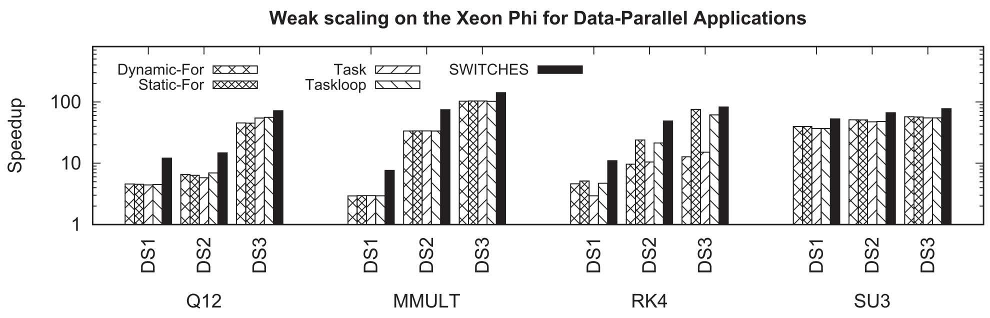


Fig. 17. Speedup achieved with the best configuration for each system when scaling the input size of the *Data-parallel* applications with fixed resources (240 threads).

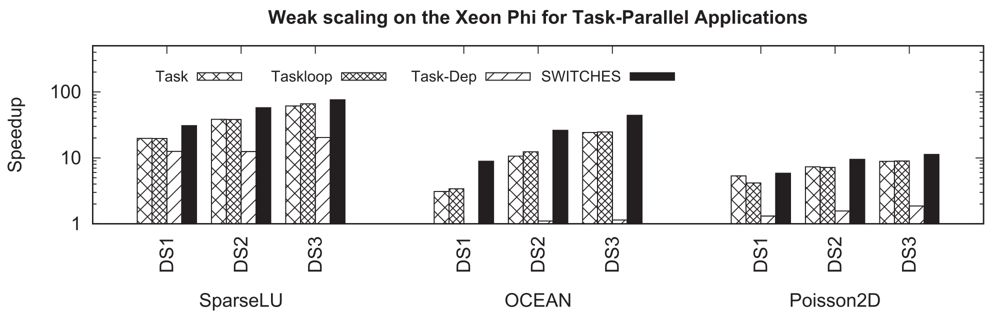


Fig. 18. Speedup achieved with the best configuration for each system when scaling the input size of the *Task-parallel* applications with fixed resources (240 threads for all, except Poisson2D that achieves best speedup results at 32 threads).

when using *SWITCHES* even when oversubscribing the system with more software threads than the available hardware resources as it reduces runtime overheads.

Although a static implementation of a runtime system may not be able to efficiently handle applications that dynamically change their load, we showed alternative approaches that can be beneficial for certain applications. One such approach is the explicit task resource allocation construct that in combination with the cross-loop dependences proposed helps in hiding load imbalances or low resource utilization.

8 RELATED WORK

The processor industry is moving closer to the development of many-cores with Intel recently proposing the MIC Architecture [24] and launching the Xeon Phi [25], its first many-core co-processor product for HPC systems. The Xeon Phi offers a fully cache-coherent environment across all cores, and thus supports the most commonly used programming models for developing HPC applications like OpenMP [8], POSIX threads [41], and Message Passing Interface (MPI) [20] without the immediate need to rewrite an application. The Graphics Processing Unit (GPU) [23] is another many-core architecture, which has been around for some time and supports parallel execution with hundreds of cores. GPUs offer large amounts of computing power with low cost but the design of a GPU only favors regular applications with streaming computation that have little sharing. Although today more programming models support GPU execution, sufficient skills and knowledge of the underlying hardware are needed in order to achieve good performance. GPUs are special purpose hardware units that offer significant performance increase mostly to data-parallel applications.

Several software solutions exist today that target the wide exploitation of parallelism on commodity general-purpose many-cores. Many of them show a renewed interest in the dataflow computation approach that was pioneered in the late 1970s and 1980s [6, 21, 26, 27, 37, 45]. These past projects demonstrated that it was feasible to express large amounts of parallelism but a significant problem was how to throttle and schedule it efficiently. Subsequent projects show that coarsening the granularity of parallelism (from instructions to tasks) results in efficient execution. Many hardware solutions have been proposed as well but in this related work we only focus on software systems that use unmodified existing hardware and can be directly compared to what we propose. In the next paragraphs we describe the basic concepts of such systems and outline their differences compared to *SWITCHES*.

OmpSs [15] is a software shared-memory task-based programming model based on OpenMP [8] and StarSs [39], which targets heterogeneous multi-cores. In contrast to *SWITCHES*, OmpSs is an all-dynamic system as it resolves task dependences at runtime and at the same time dynamically assigns tasks to cores. Since version 3.0, OpenMP also provides support for task-based execution, and in version 4.0 complex data dependences [44] were introduced. Some of these applications were also used for the evaluation of *SWITCHES* and show a significant increase in performance compared to OpenMP. Intel TBB [29] is another dynamic system that implements the task-based model and distributes tasks to cores at runtime. TBB's scheduler uses a task-stealing approach to dynamically re-distribute tasks to available cores in order to achieve better load balancing. Although in some applications task stealing will improve the utilization of the hardware, it will also add runtime overheads. In *SWITCHES* we use an explicit resource allocation technique (described in Section 4.3) as a static solution that provides a mechanism for efficient allocation of resources and increases performance without incurring runtime overheads as shown in our results (Figure 14).

Swan [43] implements a unified scheduler on top of the Cilk [17] language to support recursive and Task dataflow with more than one level of parallelism. Like the previous systems, it features dynamic scheduling and assignment of tasks and implements shared *tickets* for resolving task dependences. To avoid simultaneous access on shared *tickets*, the runtime requires atomic access and locking protection that can become a bottleneck when scaling to larger systems. In *SWITCHES* we eliminate locks by implementing a de-centralized, single-writer/multiple-readers runtime system that provides each scheduler with its own private scheduling structures. SWARM [31] is another dataflow system implemented for both shared and distributed memory systems. Its runtime system is built in such a way that no hardware cache-coherence mechanism is needed but in contrast to *SWITCHES* it uses a dynamic scheduling policy to assign tasks to cores.

Qthreads [46] was developed to provide a portable abstraction for lightweight thread control and synchronization primitives. Although Qthreads is not a dataflow implementation, it has many similarities. Qthreads is based on the Full/Empty Bits (FEBs) technique developed by the Denelcor HEP [18] and used by the Cray XMT [1] and PIM [5, 22, 28] designs. This technique marks each word in memory with a *full* or *empty* state, allows programs to wait for either state, and makes the state change atomically with the word's contents. Although this technique can be emulated in software, it is clearly stated by the authors in [46] that without hardware support for lightweight synchronization, FEB locks will be a bottleneck.

The Legion [4] runtime system attempts to express locality and independence of program data and tasks using logical regions that name a set of objects (data). The programmer is responsible for explicitly grouping these objects to regions and this requires significant source code modifications, in contrast to *SWITCHES* where we employ the OpenMP directives that can be used on top of the original sequential source code. Similarly to OmpSs, Legion also uses a dynamic detection and enforcement of dependences on parallel tasks that increases the work of the runtime system and potentially increases overheads.

The Open Community Runtime (OCR) [35] is a recent effort that aims to provide a runtime system for extreme-scale computing. Although OCR resembles some similarities with *SWITCHES* (such as the shared globally unique name space without hardware cache-coherence support), it uses a dynamically generated task graph to express parallelism that could potentially produce significant runtime overheads (depending on the application and the hardware used). An implementation of OCR using the TBB task scheduler for the Intel Xeon Phi showed that it does not scale that well to such a high number of threads, and significant optimizations are required in order to achieve performance comparable to OpenMP as shown in [14].

The Codelet [49] execution model aims to develop a methodology for exploiting parallelism in future exascale machines. A codelet is a collection of instructions that can be scheduled atomically as a unit of computation. Codelets are more fine-grained than traditional threads and can be seen as a small chunk of work belonging to a larger task. The authors in [49] believe that the overhead of context switching (used in traditional task-based models) is too large, therefore system software and hardware will require significant optimizations in order to produce performance improvements.

DDM [30] is another model that implements the dataflow concepts on tasks instead of instructions in order to minimize runtime overheads. To determine when a task is ready for execution, DDM uses a globally shared structure called *Readycount* that holds the number of dependences remaining unresolved from each task. A *Readycount* value needs to be explicitly protected for simultaneous access using locking primitives and it requires more complex and larger data structures to implement as it grows as a value, depending on the number of tasks and their dependences. In contrast, the ON/OFF structures used in *SWITCHES* follow the *single-writer/multiple-readers* model that requires no locking and the smallest data structure in a system is sufficient for its implementation. Different software and hardware implementations have been proposed in [3, 34, 42] that target multi-core systems but all use a centralized scheduling unit that dynamically schedules parallel tasks while burdening the programmer with the responsibility of assigning tasks to cores manually.

All these solutions are software implementations for shared-memory systems with each one providing a task-based runtime that dynamically schedules parallel tasks to processor cores based on data availability. Each system also comes with its own programming interface that requires extra effort from the programmer and all except for SWARM and OCR require the support of hardware cache-coherence in order to provide correct execution. With *SWITCHES* we propose a new Task dataflow runtime system that reduces overheads during execution and removes all bottlenecks that can limit its scalability to hundreds or more cores. More specifically, our work differs from other solutions in the following ways: (1) we implement a distributed triggering system that avoids all synchronization primitives, such as locks and barriers; (2) we implement *static* scheduler that minimizes runtime overheads; (3) we require no hardware support for cache-coherence that provides for portability of the implementation in future processors that may not support hardware cache-coherence (also shown in a previous work [13]); and (4) we maintain programming productivity by extending a widely used programming API (OpenMP v4.5 [36]).

9 CONCLUSIONS

In this work, we present *SWITCHES*, a lightweight scalable runtime system that uses the dataflow paradigm to schedule parallel tasks in many-core systems. In large-scale systems where the number of cores keeps increasing, it is important to improve performance that comes from parallelism. To achieve this, *SWITCHES* implements a fully distributed scheduler that uses static scheduling policies and produces smaller runtime overheads. It therefore allows for finer-grain tasks to be implemented as a means to increase the parallelism exposed without losing performance.

SWITCHES also incorporates a source-to-source tool that produces parallel code from a sequential application embedded with OpenMP v4.5 API directives. It automatically produces the dataflow graph with the tasks and their dependences and statically schedules the tasks to the available execution units. Its execution model supports variable-granularity loop tasks that, combined with cross-loop iterations dependences and explicit resource allocation techniques, increases the exploitable parallelism, takes advantage of the data-locality in loop tasks and efficiently allocates hardware resources. Without affecting programming productivity, *SWITCHES* achieves significant performance benefits (an average of 32%) on the many-core system tested, compared to the state-of-the-art OpenMP implementation.

ACKNOWLEDGMENTS

The authors would like to thank The Cyprus Institute and the Cy-Tera HPC Facility for providing the hardware resources (Intel Xeon Phi accelerators) used in the evaluation of this work.

REFERENCES

[1] 2007. Cray XMT platforrm. (2007). http://www.cray.com/products/xmt/index.html. [Online].

[2] A. V. Aho, M. R. Garey, and J. D. Ullman. 1972. The transitive reduction of a directed graph. *SIAM J. Comput.* 1, 2 (1972), 131–137. DOI: http://dx.doi.org/10.1137/0201008

[3] S. Arandi and P. Evripidou. 2010. Programming multi-core architectures using Data-Flow techniques. In *International Conference on Embedded Computer Systems (SAMOS)*. 152–161.

[4] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society Press, Article 66, 11 pages.

[5] Jay B. Brockman, Shyamkumar Thoziyoor, Shannon K. Kuntz, and Peter M. Kogge. 2004. A low cost, multithreaded processing-in-memory system. In *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture (WMPI'04)*. ACM, New York, 16–22. DOI: http://dx.doi.org/10.1145/1054943.1054946

[6] D. Cann. 1991. Retire Fortran? A debate rekindled. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing'91)*. ACM, New York, 264–272.

[7] Transaction Processing Council. 2006. TPC Benchmark H (Decision Support). Standard Specification Revision 2.6.1. (2006).

[8] L. Dagum and R. Menon. 1998. OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (Jan. 1998), 46–55.

[9] J. B. Dennis. 1974. First version of a data flow procedure language. In *Programming Symposium*. Springer, 362–376.

[10] J. B. Dennis and D. P. Misunas. 1975. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture (ISCA'75)*. ACM, New York, 126–132.

[11] Andreas Diavastos. 2017. SWITCHES Platform. Retrieved from https://github.com/diavastos/SWITCHES. [Online].

[12] A. Diavastos, G. Stylianou, and G. Koutsou. 2016. Exploiting very-wide vectors on Intel Xeon Phi with lattice-QCD kernels. In *Proceedings of the 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. 296–300.

[13] A. Diavastos, G. Stylianou, and P. Trancoso. 2015. TFluxSCC: Exploiting performance on future many-core systems through data-flow. In *Proceedings of the 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 190–198. DOI: http://dx.doi.org/10.1109/PDP.2015.69

[14] Jiri Dokulil and Siegfried Benkner. 2015. Retargeting of the open community runtime to Intel Xeon Phi. *Procedia Computer Science* 51 (2015), 1453–1462. DOI: http://dx.doi.org/10.1016/j.procs.2015.05.335

[15] A. Duran, E. Ayguad, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. 2011. OmpSs: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters* 21, 2 (2011), 173–193.

[16] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. 2009. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *Proceedings of the International Conference on Parallel Processing (ICPP'09)*. IEEE Computer Society, 124–131.

[17] M. Frigo, C. E. Leiserson, and K. H. Randall. 1998. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98)*. ACM, New York, 212–223.

[18] M. C. Gilliland, B. J. Smith, and W. Calvert. 1976. Hep—A semaphore-synchronized multiprocessor with central control (heterogeneous element processor). In *1976 Summer Computer Simulation Conference*. 57–62.

[19] Roberto Giorgi, Rosa M. Badia, Franois Bodin, Albert Cohen, Paraskevas Evripidou, Paolo Faraboschi, Bernhard Fechner, Guang R. Gao, Arne Garbade, Rahul Gayatri, Sylvain Girbal, Daniel Goodman, Behran Khan, Souad Kolia, Joshua Landwehr, Nhat Minh L, Feng Li, Mikel Lujn, Avi Mendelson, Laurent Morin, Nacho Navarro, Tomasz Patejko, Antoniu Pop, Pedro Trancoso, Theo Ungerer, Ian Watson, Sebastian Weis, Stphane Zuckerman, and Mateo Valero. 2014. TERAFLUX: Harnessing dataflow in next generation teradevices. *Microprocessors and Microsystems* 38, 8, Part B (2014), 976–990. DOI: http://dx.doi.org/10.1016/j.micpro.2014.04.001

[20] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing* 22, 6 (1996), 789–828.

[21] J. R. Gurd, C. C. Kirkham, and I. Watson. 1985. The Manchester prototype dataflow computer. *Communications of the ACM* 28, 1 (Jan. 1985), 34–52.

[22] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Jay Brockman, Apoorv Srivastava, William Athas, Vincent Freeh, Jaewook Shin, and Joonseok Park. 1999. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (SC'99)*. ACM, New York, Article 57. DOI: http://dx.doi.org/10.1145/331532.331589

[23] Q. Huang, Z. Huang, P. Werstein, and M. Purvis. 2008. GPU as a general purpose computing resource. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing, Applications and Technologies*. 151–158.

[24] Intel. 2016. Intel Many Integrated Core Architecture. Retrieved from http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core.

[25] Intel. 2016. The Intel Xeon Phi coprocessor. Retrieved from http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[26] W. M. Johnston, J. R. Paul Hanna, and R. J. Millar. 2004. Advances in dataflow programming languages. *ACM Computing Surveys* 36, 1 (March 2004), 1–34.

[27] Richard M. Karp and Rayamond E. Miller. 1966. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics* 14, 6 (1966), 1390–1411.

[28] P. M. Kogge, S. C. Bass, J. B. Brockman, D. Z. Chen, and E. Sha. 1996. Pursuing a petaflop: Point designs for 100 TF computers using PIM technologies. In *Frontiers'96 Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computing*. 88–97. DOI: http://dx.doi.org/10.1109/FMPC.1996.558065

[29] A. Kukanov and M. J. Voss. 2007. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal* 11, 4 (2007), 309–322.

[30] C. Kyriacou, P. Evripidou, and P. Trancoso. 2006. Data-driven multithreading using conventional microprocessors. *IEEE Transactions on Parallel and Distributed Systems* 17, 10 (2006), 1176–1188.

[31] C. Lauderdale, M. Glines, J. Zhao, A. Spiotta, and R. Khan. 2013. SWARM: A unified framework for parallel-for, task dataflow, and distributed graph traversal. ET International Inc., Newark (2013).

[32] E. A. Lee and D. G. Messerschmitt. 1987. Synchronous data flow. *Proceedings of the IEEE* 75, 9 (Sept. 1987), 1235–1245. DOI: http://dx.doi.org/10.1109/PROC.1987.13876

[33] B. Li, H. C. Chang, S. Song, C. Y. Su, T. Meyer, J. Mooring, and K. W. Cameron. 2014. The power-performance tradeoffs of the Intel Xeon Phi on HPC applications. In *Proceedings of the 2014 IEEE International Parallel Distributed Processing Symposium Workshops*. 1448–1456. DOI: http://dx.doi.org/10.1109/IPDPSW.2014.162

[34] George Matheou and Paraskevas Evripidou. 2015. Architectural support for data-driven execution. *ACM Transactions on Architecture and Code Optimization* 11, 4 (Jan. 2015), Article 52, 25 pages. DOI: http://dx.doi.org/10.1145/2686874

[35] T. G. Mattson, R. Cledat, V. Cav, V. Sarkar, Z. Budimli, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, Min Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo. 2016. The open community runtime: A runtime system for extreme scale computing. In *Proceedings of the 2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. DOI: http://dx.doi.org/10.1109/HPEC.2016.7761580

[36] OpenMP Architecture Review Board. 2015. OpenMP 4.5 API C/C++ Syntax Reference Guide. Retrieved from http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[37] G. M. Papadopoulos and D. E. Culler. 1990. Monsoon: An explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)*. ACM, New York, 82–91.

[38] P. Petrides, A. Diavastos, C. Christofi, and P. Trancoso. 2013. Scalability and efficiency of database queries on future many-core systems. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 24–28. DOI: http://dx.doi.org/10.1109/PDP.2013.14

[39] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. 2009. Hierarchical task-based programming with StarSs. *International Journal of High Performance Computing Applications* 23, 3 (2009), 284–299.

[40] Red Hat Developer Program. 2016. What is new in OpenMP 4.5. Retrieved from https://developers.redhat.com/blog/2016/03/22/what-is-new-in-openmp-4-5-3.

[41] Red Hat Inc. 2003. *The Native POSIX Thread Library for Linux*. Red Hat Inc.

[42] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso. 2008. TFlux: A portable platform for data-driven multithreading on commodity multicore systems. In *Proceedings of the 37th International Conference on Parallel Processing*. 25–34.

[43] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. 2011. A unified scheduler for recursive and task dataflow parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT 2011)*. 1–11.

[44] P. Virouleau, P. Brunet, F. Broquedis, N. Furmento, S. Thibault, O. Aumage, and T. Gautier. 2014. Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In *Proceedings of the 10th International Workshop on OpenMP (IWOMP'14)*. 16–29.

[45] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, and J. Sargeant. 1988. Flagship: A parallel architecture for declarative programming. In *Proceedings of the 15th Annual International Symposium on Computer Architecture (ISCA'88)*. IEEE Computer Society Press, 124–130.

[46] K. B. Wheeler, R. C. Murphy, and D. Thain. 2008. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–8.

[47] S. Yoon and A. Jameson. 1988. Lower-upper symmetric-Gauss-Seidel method for the Euler and Navier-Stokes equations. *AIAA Journal* 26, 9 (1988), 1025–1026.

[48] X. Zhou, H. Chen, S. Luo, Y. Gao, S. Yan, W. Liu, B. Lewis, and B. Saha. 2010. A case for software managed coherence in many-core processors. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism (Poster Paper)*.

[49] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. 2011. Using a "Codelet" program execution model for exascale machines: Position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT'11)*. ACM, New York, 64–69. DOI:http://dx.doi.org/10.1145/2000417.2000424

Received April 2017; revised June 2017; accepted July 2017