Characterizing and improving the performance of many-core task-based parallel programming runtimes

ABSTRACT

The importance of parallel programming is increasing year after year since the power wall popularized multi-processors architectures, and with them, shared memory parallel programming models. In particular, task-based programming models like the standard OpenMP 4.0 have become more and more important.

OmpSs is a task-based data-flow parallel programming model, which clearly inspired the evolution of OpenMP 4.0, where the task execution is driven by the dependences among tasks. In particular, the dependences management is based on dependence graphs shared among threads, which have to be updated in exclusive access using synchronization mechanisms (locks). Although exclusive accesses are necessary to avoid data race conditions, those may imply contention that drags out the application execution by limiting its parallelism. This becomes critical in many-core systems because several threads may be wasting computation resources waiting for the shared region release.

This paper characterizes this behaviour, analyzing how it hinders performance under different execution conditions. Also, it presents an alternative organization for task-based programming models runtimes. This runtime organization allows to asynchronously manage the task dependences and consequently reduces potential synchronization contention. In addition, the new organization, based on the usage of a smart thread manager (DAST), is workload conscious and adapts itself, at runtime, according to the manager load with the objective of being as fast as possible. That helps to exploit the application parallelism to its full potential meanwhile improving the usage of computing resources.

Results show that the new runtime outperforms the peak speedup of the original runtime model when contention is huge and achieves similar or better performance results for real applications.

1. INTRODUCTION

The end of Dennard scaling [7] plus the continuity of Moore's law [10] resulted in a growing importance of multicore processors. The popularization of these chips increased the need to use all their power without having to deal with complex applications programmed for one specific architecture. Parallel programming models simplify application developing process by decoupling applications from hardware. The idea is to allow programmers to friendly indicate the potential parallelism in their source code without managing it. There are several examples like MapReduce [6], OpenMP [4], OpenCL [15] and StarSs [12].

The task oriented paradigm is one powerful way to define potential parallelism in one application. Programmers only have to annotate code regions called tasks that can run in parallel. Additionally, users can provide more information for each task, like data dependences that define the task execution order which is enforced by the runtime libraries in execution time. The OpenMP standard introduced task dependences in the 4.0 version greatly influenced by the OmpSs programming model that extends the standard syntax.

The runtimes of these models are responsible for guaranteeing the correctness of the task execution order defined by their dependences. To do it, a task graph is maintained in execution time, so the runtime checks the dependences of each created task against on-the-fly tasks and delays its execution if necessary. After each task, the runtime notifies successor tasks and wakes them up if they are ready to be executed. Usually, these actions are made when needed by any thread (worker) that is executing tasks. This distributed behaviour is the simplest design and the most used, but only one worker can modify the task graph at the same time due to coherence reasons. This bottleneck may significantly reduce applications performance when a high number of workers is used. The cause is simple: the more workers, the more tasks on-the-fly, the more workers competing for graph access, and consequently the more stalls waiting for the exclusive access.

As a motivation, Figure 1 shows OmpSs and OpenMP 4.0 speedup performance results for two parallel alternatives that solve a synthetic benchmark Those parallel alternatives use: task dependences synchronization (OmpSs and OpenMP taskdeps bars) or global task synchronization (OmpSs and OpenMP taskwait bars). All implementations solve the same problem resolving the synchronization among tasks in different ways. Results are from a MIC machine (explained in the experimental setup) running 1 thread per core and using the Nanos++ runtime for OmpSs or Intel OpenMP runtime for OpenMP. Although the implementations with task dependences synchronization have

CASES '16 October 2–7, 2016, Pittsburgh, PA, USA

ACM ISBN xxx-xxxx-xx-xxx/xx/xx.

DOI: xx.xxxx/xxxx

potentially more parallelism, Figure 1 shows a significant degradation when using this model, that is even more significant in the OpenMP version. Those results show how much the management of task dependences may make the runtimes limit the application performance.

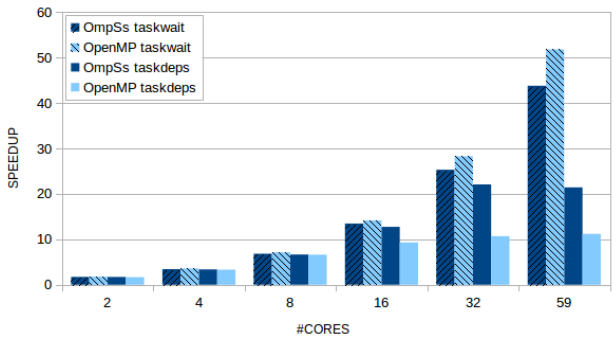


Figure 1: Comparison of speedup obtained with different programing models and application implementations

To avoid the explained problem, we present a new runtime structure useful for task-based parallel runtimes in many-core environments. The new structure is based on the usage of a manager that executes different runtime functionalities and takes care of task dependences management. The main goal is to keep the dependence graph access restricted as much as possible, doing the needed actions by the manager (DAST) instead of each worker thread. Also, the new runtime implementation (DAST runtime) is able to keep the management distributed among workers when the existence of a single centralized manager bounds the application parallelism, better balancing the work to be as fast as possible. In any case, this model may help to increase the data locality because the whole manager cache can be used to store the runtime structures that are distributed and moved among the workers in the original flow.

The remainder of the paper is ordered as follows. Section 2 describes the OmpSs task-based programming model, as an example of task-based programming model whose runtime takes care of the dependences management. Section 3 analyses the related work. Section 4 describes the proposed runtime structure for dependences management with the DAST. Section 5 presents the experimental setup. Section 6 characterizes the time cost of relevant runtime parts. Section 7 shows the performance that different runtimes obtain running a set of benchmarks. Finally, Section 8 presents the work conclusions.

2. BACKGROUND

OmpSs is a task-based programming model that allows to create parallel applications annotating a sequential code. The annotations, which follow OpenMP syntax (`#pragma omp ...`), are replaced to Nanos++ API calls at compiling time by Mercurium. The main annotation is `#pragma omp task` that specifies a code region which will be executed asynchronously and concurrently with other tasks. This pragma can be extended with some clauses (`in(...)`, `out(...)`, `inout(...)`) in order to specify data dependences between tasks. The runtime is responsible for synchronizing task executions to guarantee the dependences. Although this implicit synchronization, there are other annotations like `#pragma omp taskwait` to explicitly synchronize the execution at some point.

```
void foo (int *a, int *b) {
  for (int i = 1; i < N; i++) {
#pragma omp task in(a[i-1]) inout(a[i]) out(b[i])
    propagate(&a[i-1], &a[i], &b[i]);
#pragma omp task in(b[i-1]) inout(b[i])
    correct(&b[i-1], &b[i]);
  }
}
```

Figure 2: OmpSs code annotation example [14]

Figure 2 shows an example code of a parallelized function with two code regions converted into tasks (functions *Propagate* and *Correct*). In this example, the main worker (the one which starts program execution) will make two task creation calls to Nanos++ runtime for each loop iteration instead of the two function calls. For each API call, the system checks data dependences of the new task and calculates its predecessors, creating a dependency graph with the pattern shown on Figure 3. *Propagate* task is reading `a[i-1]` and `a[i]` values and writing to `a[i]` and `b[i]`, *Correct* task is reading `b[i-1]` and `b[i]` values and writing `b[i]`. So *Propagate* only depends on *Propagate* previous iteration task and *Correct* depends on *Correct* previous iteration task and same iteration *Propagate* task.

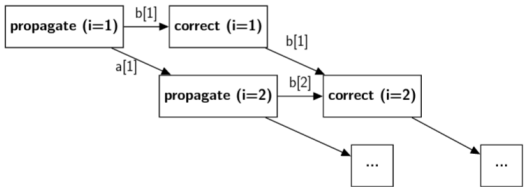


Figure 3: OmpSs graph example [14]

Task representation inside Nanos++ is made by one Work Descriptor (WD) for each task. These WDs contain all the information needed by the runtime to manage and schedule them. As tasks only can depend on sibling tasks, the dependences of each one are stored in a task graph on father's WD. Global order is guaranteed because father's dependences must be a super-set of its child tasks. Despite this distributed model, actions in each graph should be performed in mutual exclusion because different sibling tasks can finalize at the same time and/or collision with another sibling task creation. To ensure this mutual exclusion inside each graph, Nanos++ uses spin-locks implemented as a flag modified by an atomic compare and swap operation.

Figure 4 presents a simplification of the operational model for one task graph. Each thread sends information about the tasks (creation/finalization) through the shared lock between all workers. Ready tasks are moved to another structure after their dependences are satisfied and is used to obtain work by the workers. The Figure does not show the complexity of the *Ready Pool* since the implementation depends on the active scheduling policy that may change between executions.

At the beginning of any OmpSs application execution, only the master worker thread is running. It is started by

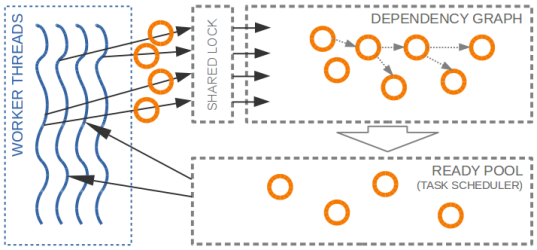


Figure 4: Nanos++ dependences operational flow

the operating system and is the only thread existing in the new process. It starts the other worker threads, initializes the Nanos++ structures to manage the execution and runs the main task that corresponds to the main function defined by the programmer. Moreover, it runs other common tasks during the execution and it is responsible to stop all the workers after all the tasks are executed, destroy runtime structures and perform last operations before exit.

Nanos++ runtime comes with several plugins that allow users to tune the default behaviour. It allows users to change the scheduling policy, barrier algorithms, dependency manager or throttle policy, among others. Scheduler policies define how ready tasks are managed. The default policy is Breadth First (BF) which implements a global ready FIFO queue for all the workers. Another available policy useful for many-core environments is Distributed Breadth First (DBF) where one FIFO queue for each worker is used.

3. RELATED WORK

Several works exist about parallel programming models characterization and improving. They are over different models working at different levels and with different approaches. OmpSs tools (Mercurium and Nanos++), which are the ones used to test our model, can execute inter-node and intra-node applications [2] and are under constant development introducing new features. Moreover, several people use this programming model as a base to develop different prototypes or extend its functionality.

One example of extending OmpSs functionality is the Energy Efficient Task Pool Scheduler in OmpSs [9]. This work proposes a new scheduling policy that tunes processors frequency in order to reduce power consumptions without decreasing performance. Our proposal is to restructure the dependency management system without extending the programming model functionality.

Previous works discussed about the task scheduling and dependences resolution overheads in data-driven task-based models like OpenMP and OmpSs. TurboBLYSK [13] is a framework which implements the OpenMP 4.0 with a custom compiler and a highly efficient run-time scheduler of tasks with explicit data-dependence annotations. Its objective is also to reduce the dependence management overheads of the runtime. However, TurboBLYSK approach requires extra information in the task dependences definition to allow the runtime to re-use previously resolved dependency patterns and to reduce the overall overhead. In contrast, we only use the default provided information to reduce the overall task management overhead, increasing the performance per resolved task dependence. In some sense, our works are complementary.

Other task-based programming models like Intel Threading Building Blocks [11] and Charm++ [8] use a execution model that is more pure dataflow than the OmpSs/OpenMP model which has a hybrid (control/dataflow) model [19]. This execution model usually allows to exploit better the parallelism of the applications but requires a specific structure and an application redesign. In the context of the Intel Threading Building Blocks, there is a previous work discussing the cost of the synchronization inside its runtime, but they focus the problem in the work distribution [3] instead of the task graph management that is implicitly done in their execution model. The Charm++ programming model is intended to provide some valuable features for executions in large computation systems like migratability, checkpoint application restarting, process failure tolerance, malleability, etc. They have previous work about optimizing the communications inside their runtime [3], but, as in the TBB, the dataflow model that they have moves the complexity of task-graph management into the application development process.

Another work about how to accelerate current runtimes consists on moving some parts of them into specific hardware or FPGAs. Some examples are Nexus# [5] and Picos [18]. They present different hardware designs that can manage tasks dependencies of task-based programming models. Besides these, there is active research in new computer architectures able to manage efficiently tasks in StarSs family. For example, some research aim to look for a new Runtime-Aware Architecture to overcome current multi-core restrictions like power, programmability and resilience [16]. The main difference between those works and the one proposed here is the way to improve the existing system. They proposed new hardware to work in harmony with the software in order to improve the performance while we propose how to improve the existing parallel programming model runtimes with software ideas that do not require additional hardware. Although, we claim that the ideas exposed here can be used to implement or integrate hardware acceleration support for task dependence management.

4. RUNTIME ORGANIZATION PROPOSAL

Our runtime execution model decouples the runtime functionality and the task execution by introducing a decoupled manager that does the main runtime functions. The aim is to avoid dependence graph access contention generated by the distributed management of the structure by all workers. Therefore, each runtime API call results in a unique request from the worker thread to the decoupled manager with all the needed information. This modification is transparent for the applications programmers and does not need an application recompilation, it just needs and update of runtime libraries.

The contention at dependence graph access comes from the time that each thread holds the lock and makes the others waste their computational capacity waiting for the lock release. That becomes more crucial in many-core architectures where the huge number of workers increases the probability of collisions at the locking step.

The objective of the novel model is to be conscious of the runtime load generated by the application tasks, and then, automatically adapt the runtime behaviour. Therefore, the new model centralizes the execution of runtime functionalities on a manager to erase the need of waiting for access the

dependence graph. Moreover, the decoupling of the runtime functionality execution parallelizes it with the task execution done by the workers, hiding the runtime functionalities costs and increasing the data locality of the dependence graph. However, the runtime functions that the decoupled manager executes are larger than the lock protected regions, they can be executed partially in parallel, and they can be done in parallel without synchronization if involve independent graphs. Consequently, the runtime functionality execution may be kept distributed if the number of requests to the decoupled manager is too high, trying to minimize the waiting time acquiring locks without losing parallelism. The overall contention should be much small than the existing in the Nanos++ runtime because usually the lock is acquired by decoupled manager and eventually by some worker.

The communication between the workers and the manager is done pushing messages, which contain the information to allow the later satisfaction of the request, inside a queue. This queue can be different for each worker thread because the messages that the workers will insert inside their queues are independent and can be executed out of order. The order that the manager must guarantee is the FIFO order of the messages in the same thread queue because the correctness of the dependencies calculation. The new runtime structure only requires a worker-manager synchronization, meanwhile the original runtime structure requires a global synchronization between workers at each API call that works with the task graph. The synchronization in new runtime is needed to ensure the correct insertion and deletion of the messages inside the same queue of the global distributed-queue system.

The manager executes the runtime functionality requests done by the workers using the default API of the runtime. At each API call, the load of the decoupled manager is analyzed and a unique request from the worker thread to the manager is generated if the decoupled behaviour is considered better, otherwise, the request is not generated and the worker does the needed actions. In any case, the message generation and delivery time are much smaller than the execution time of the runtime functionality.

The two main API calls that involve the task graph management are the task creation ("New task") and the task finalization ("Done task") functionalities. For both, a brief description and the main keys of each message/request are provided following:

New task: Sent when a new task is submitted into the runtime. This step of task creation does not include the WD creation and initialization for the new task that is still done by the workers. This request is satisfied comparing the task dependences against previous tasks in order to get the predecessor tasks that will trigger the new task readiness. All this information is obtained from the task dependency graph, which is updated with the new task information, and any predecessor task is notified about its new successor task.

Done task: Sent when a task finishes its execution. This request is satisfied notifying the task finalization to the successors tasks and setting them as a ready task if it does not have more predecessors that have not finished. Thus, this request has to update the dependency graph in order to extract the finished task information from it and update the successors information.

4.1 Implementation

The current implementation of our model uses the Nanos++ runtime and its WD data structures as a baseline. The new runtime includes the functionalities of Nanos++ runtime using the same API, meanwhile its behaviour may be decoupled or not depending on the task workload and synchronization contention. In the decoupled mode, the new organization includes the decoupled thread manager (DAST thread).

Figure 5 represents the new execution flow over the explained OmpSs structure for one dependence graph. In contrast to the original flow which has a shared lock for all workers, the new one allows having multiple locks that are only shared between one worker and the manager. These locks ensure the correctness of workers and DAST communication that is made inserting the requests inside the thread-distributed queue system. The minor number of threads able to take the lock, the minor time elapsed acquiring the lock and the minor runtime overhead.

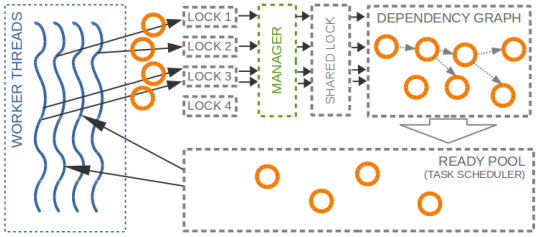


Figure 5: New lock organization for the proposed runtime.

The DAST thread executes an active-waiting loop which waits for requests inside the queue system and executes the needed actions to the runtime structures according to the message.

Algorithm 1 contains the pseudocode of the DAST thread loop. The loop goes until the system is not running (lines $1-11$ of algorithm 1), usually, because the execution ended. While the system is running, we select one worker following a round robin policy (line 2) and we set the DAST thread properties inside the runtime to execute the runtime code as it was done by the requester worker (line 3). After that, the DAST thread tries to get one request from the worker (line 5) and, if it gets one, the DAST thread executes the runtime functions needed to satisfy the request; if not, it iterates the main loop again. The process of obtaining and executing one request is repeated at most `MAX_REQ_COUNT` times to avoid processing requests from only one worker (line 4).

Algorithm 1: DAST thread work-loop

```
1  while System is running do
2  |  worker ← selectWorkerRoundRobin()
3  |  simulateBeWorker( worker )
4  |  for reqCount ← 1 to MAX_REQ_COUNT do
5  |  |  if getRequest( worker ) then
6  |  |  |  satisfyRequest()
7  |  |  else
8  |  |  |  break
9  |  |  end
10 |  end
11 end
```

To keep the runtime system working as in the original Nanos++ runtime, three different messages, or requests, from workers to DAST must be sent from different runtime API calls. Two of them are the ones described in the general system description and the third one is the "Delete task". This message is sent when the WD memory space can be deallocated, usually, that is after the "Done task" message. This work has to be a separate request of the "Done task" to guarantee that DAST thread will not need the WD for any previous request and preserve data until the worker executing the task does not need it. This request is satisfied by calling the WD destructor.

The decision of pushing a message in the queue system and execute it asynchronously or execute it inlined on the worker, like in the original Nanos++, is made following different criteria depending on the message type. Algorithm 2, Algorithm 3 and Algorithm 4 contain the pseudocode that describes the followed criteria to push or not new, done and delete task messages, respectively.

Algorithm 2: Pseudocode of new task push policy

```
1  if queueContainNewTaskRequest() then
2  |  prev ← previousNewTaskRequest()
3  |  if tasksAreSiblings( request, prev ) then
4  |  |  pushNewTaskRequest( request )
5  |  else
6  |  |  executeInlined( request )
7  |  end
8  else
9  |  pushNewTaskRequest( request )
10 end
```

The "New task" requests are only sent from one WD at the same time to avoid the serialization of independent tasks management and reduce task readiness. This means that when one worker pushes one new task message into the queue system only itself will be able to push more new messages until DAST thread processes it. Algorithm 2 shows a pseudocode that implements this behaviour. When the queue system does not contain any "New task" request the message is directly pushed to be executed by the DAST thread (line 9). Otherwise, the runtime checks if the current request task and previous request task are siblings (they belong to the same dependence graph) and the request is pushed (line 4), or they are not siblings and it is executed inlined by the worker thread (line 6).

Algorithm 3 shows the pseudocode of the "Done task" push policy that executes the request inlined in three cases:

1. The number of pending requests from the worker (nr) is bigger than an upper boundary (lines $4-5$ of algorithm 3).

2. The number of requests in the previous try of pushing a "Done task" (nr_{prev}) was bigger than the upper boundary and now it (nr) is bigger than the lower boundary (lines $6-7$).

3. The number of ready tasks to be executed by the worker (rt) is smaller than a threshold (lines $8-9$).

Otherwise, the request is pushed in the queue system and executed by the DAST thread (line 11). In any case, if nr_{prev} was bigger than the upper boundary and nr is under the lower boundary, the runtime doubles the boundaries values because it means that the DAST thread can process a major number of requests (lines $13-15$).

Algorithm 3: Pseudocode of done task push policy

```
1  nr ← numberRequestsWorker()
2  nr_prev ← numberRequestsPreviousPush()
3  rt ← numberReadyTasks()
4  if nr > UPPER_BOUND then
5  |  executeInlined( request )
6  else if nr_prev > UPPER_BOUND & nr >
       LOWER_BOUND then
7  |  executeInlined( request )
8  else if rt < MIN_TASKS then
9  |  executeInlined( request )
10 else
11 |  pushDoneTaskRequest( request )
12 end
13 if nr_prev > UPPER_BOUND & nr <
       LOWER_BOUND then
14 |  doubleBoundaries()
15 end
```

The "Done task" push policy follows a hysteresis design that auto-tunes to its optimum values depending on the number of ready tasks and previous still-not-satisfied done messages. The reason to design such a complex policy for Done messages is because they are the most critical ones as finished tasks release dependences that liberate new ready tasks. After an empirical testing, the values for both policies are fixed to 1 for `MIN_TASKS` and 3-5 for the initial values of `LOWER_BOUND` and `UPPER_BOUND`.

Algorithm 4: Pseudocode of delete task push policy

```
1  if queueContainDoneRequestOf( request.wd ) then
2  |  pushDeleteTaskRequest( request )
3  else
4  |  executeInlined( request )
5  end
```

As pseudocode in Algorithm 4 shows, "Delete task" requests are pushed (line 2 of algorithm 4) only if the "Done task" request of the same WD is still inside the queue system (line 1) in order to avoid data races between runtime messages. In any other case, if the runtime decided to execute inlined the "Done task" message or the DAST thread just processed it, the worker will execute the new request inlined as this message is fully parallelizable (line 4).

5. EXPERIMENTAL SETUP

The development of the new runtime model is made in the top of OmpSs 14.09 release which is formed by Mercurium 1.99.4 and Nanos++ 0.7.2. The tools used to compile the different runtimes and the benchmarks are Autoreconf (version 2.69), Automake (version 1.14.1) and Intel C/C++ compiler (version 15.0.2).

5.1 Hardware and System Environment

To explore the potential of our proposal for many-core systems, benchmarks are run in an Intel Many Integrated

Core (MIC) [17] coprocessor. On one hand, this architecture brings the possibility to run the tests in a large number of small cores allowing us to highlight the trends expected in future processors. On the other hand, the results gathered from this architecture behave similar to the ones obtained from other multi-core plataforms like Intel Xeon.

The coprocessor is a C0PRQ-7120 model composed by 61 simple but fully-capable x86 cores that can run up to 4 threads simultaneously. Each core works at 1.238 Ghz and can access the 16GB of main memory. The coprocessor can be used as an independent mini-node, it runs a special Linux operating system which manages the processing units and other operating system services. Taking this into account, executions should be done using up to 60 cores, leaving core 0 free for the OS.

Following the same path as core 0 for the OS, the master thread and the decoupled thread manager (DAST) have a reserved core too. Thus, they run in the core 1 and core 2 respectively and these reservations are considered in the maximum number of extra workers calculation. This mapping of the OS, master and DAST threads is important because the sharing of core resources with other threads may delay the executions as those threads execute crucial runtime functions.

Even the reservations, the huge number of workers available in MIC architecture makes no sense with Nanos++ default scheduling policy (BF), which has a global shared pool for ready tasks. In order to tune the runtime according to this architecture, the scheduling policy used in all MIC's executions is Distributed Breadth First (DBF) as explained in section 2. The reason behind this decision is that this scheduling policy reduces the access contention that the huge number of workers will generate in the global ready pool. In order to work with DAST thread, DBF has been adapted without changing its behaviour. The push of ready tasks is done in the queue of one worker thread instead of the ready queue of DAST that does not execute tasks, where the victim worker is selected according to their loads.

5.2 Benchmarks

The performance comparison uses four benchmarks to compare the performance of the different runtimes, one synthetic benchmark and three real benchmarks from Barcelona Application Repository (BAR) on BSC website [1]. All of them are listed in the Table 1 and the Table 2 with their execution parameters for the fine-grain and the coarse-grain tasks. The used parameters are intended to have a big enough problem size to gather significative results and an acceptable scalability with coarse-grain task size, the fine-grain task size is defined as a half of coarse-grain. The compilation flags used for each real benchmark are the default ones provided by the BAR repository and the optimization level for the synthetic benchmark is set to `-O3`.

Name	Parameters		
	`NUM_TASKS`	`TASK_SIZE`	
		Fine	Coarse
Synthetic	10000	25000	50000

Table 1: Synthetic Benchmark information

The synthetic benchmark is a simple application that creates two waves of `NUM_TASKS` tasks with a duration defined by `TASK_SIZE`. Any pair of tasks created from the same wave are independent between them and the $ith-$task of second wave depends on the $ith-$task of the first wave. Figure 6 presents the pseudocode of this benchmark with the described dependence pattern. This benchmark is useful to evaluate the adaptability and performance of DAST runtime, changing the number of parallel tasks and their granularity in the different benchmarks.

```
1  for (int i = 0; i < NUM_TASKS; i++) {
2   #pragma omp task inout(a[i])
3   foo1(a[i]);
4  }
5  for (int i = 0; i < NUM_TASKS; i++) {
6   #pragma omp task in(a[i]) out(b[i])
7   foo2(a[i], b[i]);
8  }
9  #pragma omp taskwait
```

Figure 6: Synthetic benchmark using tasks

The same benchmark can be implemented as two waves of tasks without dependencies and a taskwait in the middle of the waves (between lines 4 and 5 of Figure 6). The computational time required by the application is the same in both approaches but this second one removes the runtime overhead associated to dependencies calculation and management. However, the implementation based on tasks does not have the synchronization overhead of the taskwait and some some tasks from different loops can be executed at the same time.

Name	Parameters		
	`MATRIX_SIZE`	`BLOCK_SIZE` (#Tasks)	
		Fine	Coarse
Cholesky	8192	128 (45760)	256 (5984)
Matrix Multyply	4096	128 (32768)	256 (4096)
SparseLU	2048	32 (11472)	64 (1512)

Table 2: Real benchmarks information

Cholesky application computes the parallel Cholesky decomposition of one matrix. The `MATRIX_SIZE` parameter defines the matrix dimension and the `BLOCK_SIZE` parameter defines the block dimension for each sub-matrix used during the calculation. BAR offers multiple versions of this application, we chose `dchol_ll` that is a double-precision version.

Matrix Multiply application computes the parallel matrix multiply of two dense matrices. Its parameters have the same meaning as Cholesky ones. The chosen version from all the available ones is `dmm` that is the double-precision version.

SparseLU application computes the parallel Sparse LU decomposition of one matrix. Both parameters to define its execution characteristics are the same as in Cholesky and Matrix Multiply benchmarks.

5.3 Runtime versions

The main different runtime versions evaluated are the following:

- ***Nanos++***. One of the current OmpSs runtime implementations. Nanos++ runtime, among other tasks, supports the task dependence management of the task-based parallel programming model. This runtime dis-

tributes the runtime functionality execution among workers as commented above.

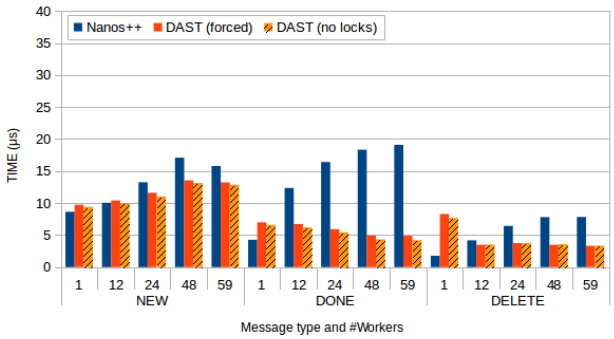
- ***OpenMP***. Intel OpenMP runtime implements the OpenMP 4.0 task dependences similar to OmpSs dependences. This runtime is a manufacture software with the only availability of the binary.

- ***DAST Runtime***. New runtime organization that has been implemented to support OmpSs parallel programming model task dependence using a self-adaptable execution model. The new runtime implementation over Nanos++ runtime and with the DAST thread.

- ***DAST (forced)***. DAST runtime derived version that, in execution time, deactivates the mechanism to dynamically decide if the runtime functionality is executed by the DAST thread or directly by the worker thread. This means that the runtime functionalities are always made by the DAST thread, so the worker threads are forbidden to execute any runtime code. It is useful to validate the behaviour of the mechanism that decides where the API calls are executed.

- ***DAST (no locks)***. Version that follows the same *DAST (forced)* idea but removing the unnecessary locks and forcing the submission when the runtime is compiled. This version is useful to check the overhead that the removed locks imply inside the runtime.

(a) Synthetic Benchmark



(b) DMM

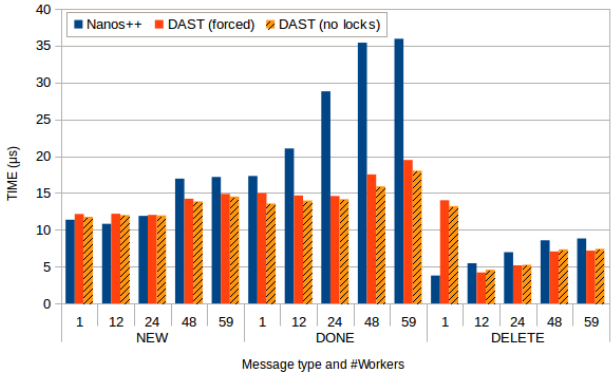


Figure 7: Execution cost of runtime functions for each DAST message type

6. RUNTIME CHARACTERIZATION

In this section, the cost of the three decoupled main runtime functionalities (new, done and delete task) and the acquision cost of a shared dependences graph lock are analyzed. That will show the main weaknesses of Nanos++ baseline and the strengths of the new runtime.

Figure 7 shows the average elapsed time in microseconds (y-axis) needed to execute the functions related to each message (explained in section 4.1) with different amount of workers (x-axis) for Nanos++ baseline, DAST runtime with the decoupled mode forced (labelled as DAST (forced)) and DAST runtime with the decoupled mode forced and the global shared lock removed (DAST (no locks)). Figure 7a shows the elapsed time for the synthetic benchmark implemented with coarse-grain tasks and Figure 7b for the Dense Matrix Multiply (DMM) running fine-grain tasks. Both benchmarks have a different task dependence pattern and task creation order that influence the cost of the studied functions. Each value in both plots is the harmonic mean of all times that each function is executed in one benchmark run for an amount of worker threads. We have used this metric instead an arithmetic mean to discard outliers and show the trends more clearly in each case.

The new task code takes similar time between the three runtime versions for small amounts of threads because all of them behave similarly, one thread creates all the applications tasks at the begining. However, as can be seen in Figure 7, the Nanos++ version increses the time more than the DAST-based versions with larger numbers of workers because the contention at shared lock starts to be huge.

The "Done task" functionality costs clearly show the contention problem. Nanos++ done task functionality takes between 2 and 4 times the DAST-based runtimes time. Figure 7a shows how with more than one worker Nanos++ runtime version doubles the cost of this functionality in comparison to the execution time using just one thread. The reason is that several workers try to access the task dependency graph, using the shared lock, to release the tasks (see Figure 4) and this increases the waiting time to get access, preventing the performance. The times in Figure 7b show an analog increasing scenario but with smaller relative increments. In any case, Nanos++ functionalities costs are still twice the execution time of DAST-based runtimes.

For Delete task functions, we can observe two different behaviours depending on the number of workers. On one hand, for Nanos++ runtime, Delete task functions costs is very small when running in one worker, as it can be seen in Figure 7. The reason is that it exploits much better the cache hierarchy than DAST scheme because the worker that executes the deletion of the task is the same that runs it and manages it, so, it should have all the information in its cache. On the other hand, DAST runtime versions present better performance results for larger numbers of workers since there is not data locality to be exploited.

To better understand the differences between runtimes, Figure 8 shows the average elapsed time in nanoseconds (y-axis) needed to acquire the shared lock and get access to one dependence graph with different amount of workers (x-axis). The figure shows the results for Nanos++ baseline, DAST (forced) and both runtimes using a MCS locks (instead of the

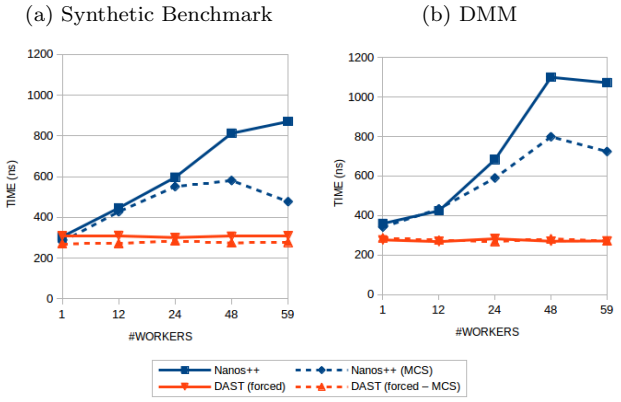


Figure 8: Acquision cost of dependences graph lock

default lock implementation) to ensure the mutual exclusion during the dependence graph modifications. As in Figure 7, Figure 8 shows the harmonic mean of times for the synthetic benchmark (Figure 8a) and the DMM (Figure 8a) with the same previous granularities.

One one hand, the elapsed time for the DAST (forced) runtime is always the same, without matter the number of workers, because the lock is only accessed by DAST thread and there is not a possible waiting time to acquire the shared lock. The MCS locks do not impact significantly on the required time for the same reason, there is not a contention problem in that implementation. On the other hand, the elapsed time for the Nanos++ runtime increases significantly with the number of workers. That increment is smaller in the MCS implementation, so a part of the increase is driven by the lock implementation. However, as also can see in the Figure 8, the Nanos++ with MCS locks still increases the elapsed time to higher values than the DAST runtime meaning that the whole contention is not due to the lock but to the time needed to process the shared region.

The dependence graph lock is acquired several times in each application execution and the small difference shown in the Figure 8 may become crucial. For example, the Nanos++ time is 2.5-3.5 times the DAST runtime time when many workers are used (48/59 workers). Independently on that time increase, the difference in time between the both runtime approaches is larger in the Figure 7 than in Figure 8 meaning that part of the improvements on the runtime functionalities execution cost must come from the dependence graph data locality that is better in the DAST runtime.

7. PERFORMANCE RESULTS

In this section, the overall application performance is analyzed when using DAST runtime. The speedup results are shown compared to the sequential execution of the same application with the same parameters.

In Figures 9 to 11, the y-axis represents speedup values and x-axis shows the number of workers for each execution including the master thread. The total number of workers does not include the extra DAST thread for both DAST series, but the results show that running with a larger number of workers does not mean increasing the speedup. Therefore, although DAST runtime uses one more thread, the comparison is still fair. Due to the core reservations explained above, experiments can be run up to 58 cores, with 1 to 4 worker threads per core. Considering the previous facts and depending on the number of threads per core, x-axis values go from 2 (1 master worker + 1 core with one worker) to 59 (1 master worker + 58 cores with one worker), 3 (1 master worker + 1 core with two workers) to 117 (1 master worker + 58 cores with two workers), 4 to 175 or 5 to 233 workers.

7.1 Synthetic benchmark

Figure 9 and Figure 10 show a comparison of the speedups obtained for the synthetic benchmark with task dependences, fine-grain and coarse-grain granularities, respectively, and using the Nanos++, Intel OpenMP, DAST (forced) and DAST runtimes.

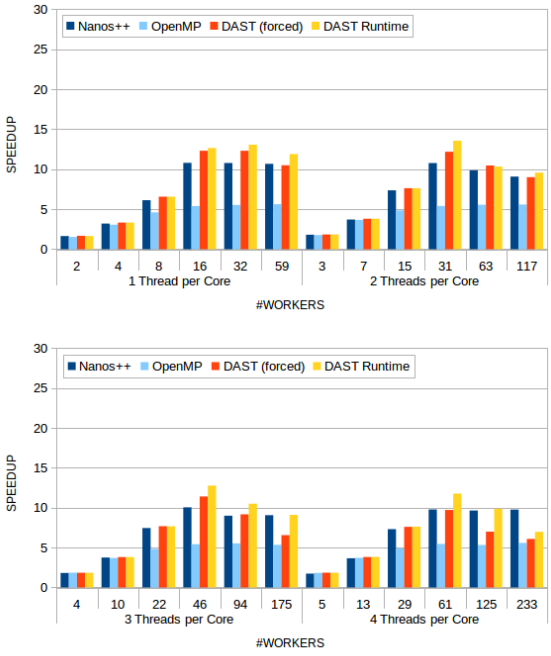


Figure 9: Speedup running Synthetic Benchmark (fine-grain tasks)

On one hand, fine-grain results in Figure 9 show that using more than 16 cores (and not workers, since using 2, 3, or 4 threads per core does not guarantee the parallel execution of the workers) does not improve the speedup, and indeed, reduces the overall performance. The reason is that fine granularity makes runtime overheads critical, limiting the speedup. On the other hand, moving to coarse grain tasks (see Figure 10) allows improving the speedup when increasing the number of cores up to 32, independently of the number of threads per core. In this case, the task size doubles the fine grain task sizes. Bigger task sizes allow overlapping the task execution with the runtime overheads, hiding them, and improving the overall application speedup.

Both figures 9 and 10 show the benefits that the application execution takes using the DAST runtime from the execution time point of view. Comparing the best speedups of each version for the same problem size, the same synthetic OmpSs application running over DAST runtime has a 26% higher speedup than running over Nanos++ runtime for fine-grain tasks, and a 12% for coarse-grain tasks.

The version using the Intel OpenMP runtime obtains the

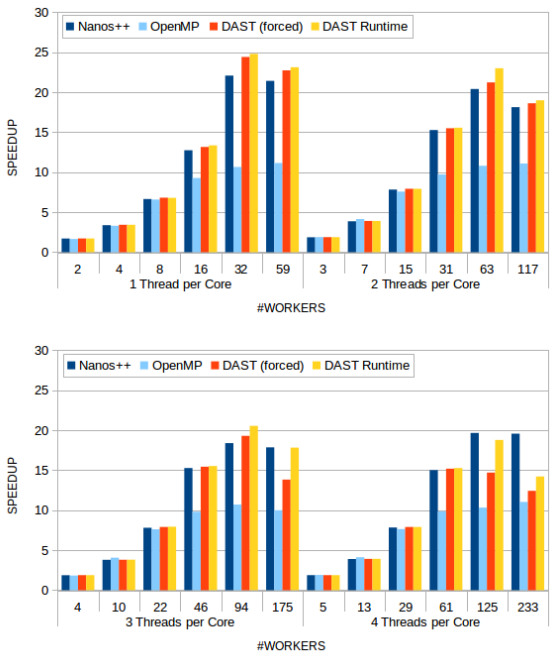


Figure 10: Speedup running Synthetic Benchmark (coarse-grain tasks)

worst performance in Figure 9 and Figure 10 despite the fact that benchmarks are run in the same manufacturer architecture. As we have seen in Figure 1, the OpenMP executions, that use the Intel OpenMP runtime, have a poor performance when thousands of tasks with thousands of different dependences must be managed.

Independently of the runtime version, there is a common trend in Figure 9 and Figure 10 which is that the best speedups are obtained using only one thread per core. The increment in the number of threads per core is not reflected in a performance increment, instead, the more threads per core used, the less speedup is obtained. We found the same standstill in most of the application executions for this architecture because the available resources in each core are shared between threads and can cut the maximum performance. Independently of that, the runtime management overhead increases with each extra worker and reduces the global performance. For this reason, the real benchmark results that present the same behaviour are shown only with one thread per core. On the other hand, the Figure 9 and Figure 10 show that DAST runtime overcomes DAST (forced) for almost all the cases, so we focused the real benchmarks comparison to DAST runtime and the baseline of our implementation (Nanos++).

7.2 Real benchmarks

Figure 11 shows the speedups in three real benchmarks executions of OmpSs applications running over Nanos++ and DAST runtime with fine-grained (Figure 11a, Figure 11c and Figure 11e) and coarse-grained (Figure 11b, Figure 11d and Figure 11f) tasks. Fine-grain results show that the Nanos++ runtime and the DAST runtime implementation achieves similar speedups with a few workers and that our structure increases the peak-speedups of the three applications. These

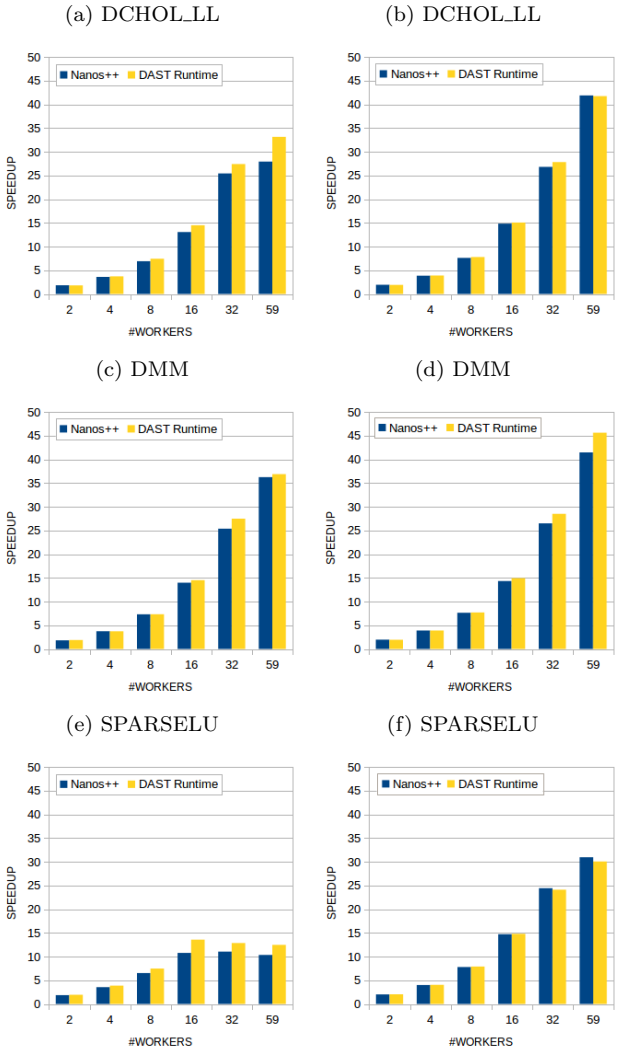


Figure 11: Speedup running real benchmarks (fine-grain and coarse-grain tasks)

maximum speedup values are obtained with 59 workers and the increases are: 19% for the Cholesky, 2% for the Matrix Multiply and 23% for the SparseLU, showing that the gains increase with the complexity of the dependence graph.

Finally, with the only purpose of showing that DAST runtime is able to achieve the same performance or better for coarse-grain applications, Figure 11 also shows results for the same real benchmarks with coarse-grain tasks. In particular, for those applications, the chosen coarse-grain tasks help to achieve better performance than for fine-grain tasks and the Nanos++ runtime does not have problems to hide the management of the dependences. Therefore, DAST runtime helps to achieve the same or better performance than Nanos++ runtime for any task granularity.

8. CONCLUSIONS

This paper characterizes the performance of dataflow task-based runtimes when executing applications in many-core systems. Results show how resource contention in the runtime shared data structures hinders the application perfor-

mance, resulting in underexploited parallelism in many-core systems.

With the objective to solve this problem, a new runtime organization for task-based parallel programming models is proposed. The proposed implementation, called DAST runtime, automatically adapts the dependencies management system to the task workload and the runtime internal lock contention. DAST runtime is based on a dedicated manager thread, called DAST, that is activated to perform runtime functionalities over the task dependency graph when the synchronization contention among worker threads is high, and the task workload is medium. Therefore, DAST runtime model fits on the big.LITTLE philosophy where the DAST thread may be run in small cores (or hardware accelerators) to help the big cores running the workers without impacting the performance. This actually will improve the overall performance and exploitation of those systems and the parallel applications.

The presented results for stressing synthetic and real benchmarks show that there is a trade-off between parallel execution of runtime functionalities (current runtimes with worker threads) using a unique shared lock to access task dependency graph and a centralized execution of runtime functionalities (DAST runtime with a smart thread manager). In particular, those results show that for more than 16-32 worker threads, the DAST runtime helps to improve the overall performance of the OmpSs applications; being more significant for fine-grain tasks.

Finally, the DAST runtime model can be scaled using a distributed DAST runtime version. This distributed version would be used either to manage dependences on disconnected sub-graphs or use a DAST/worker mutant thread to help leverage the task management and execution resources. As this paper shows, this approaches are going to be necessarily taken into account in next-generation task-based runtimes in order to allow them to cope with the constant growth of execution units in current hardware platforms.

9. ACKNOWLEDGEMENTS

Section empty for blind review.

10. REFERENCES

[1] BSC - Computer Science Department. BSC - Application Repository, 2015.

[2] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta. Productive cluster programming with ompss. In *Euro-Par 2011 Parallel Processing*, pages 555–566. Springer, 2011.

[3] G. Contreras and M. Martonosi. Characterizing and improving the performance of intel threading building blocks. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 57–66. IEEE, 2008.

[4] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, Jan 1998.

[5] T. Dallou, A. Elhossini, B. Juurlink, and N. Engelhardt. Nexus#: A distributed hardware task manager for task-based programming models. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1129–1138, May 2015.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[7] R. H. Dennard, F. H. Gaensslen, H. nien Yu, V. L. Rideout, E. Bassous, Andre, and R. Leblanc. Design of ion-implanted mosfets with very small physical dimensions. *IEEE J. Solid-State Circuits*, page 256, 1974.

[8] L. V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[9] T. B. Martinsen. Energy efficient task pool scheduler in ompss. 2013.

[10] G. E. Moore. Cramming more components onto integrated circuits. 1965.

[11] C. Pheatt. Intel® threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, Apr. 2008.

[12] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with starss. *Int. J. High Perform. Comput. Appl.*, 23(3):284–299, Aug. 2009.

[13] A. Podobas, M. Brorsson, and V. Vlassov. Turbobłysk: scheduling for improved data-driven task performance with fast dependency resolution. In *Using and Improving OpenMP for Devices, Tasks, and More*, pages 45–57. Springer, 2014.

[14] Programming Models Group BSC. Ompss user guide. May 2015.

[15] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, 2010.

[16] M. Valero, M. Moreto, M. Casas, E. Ayguade, and J. Labarta. Runtime-aware architectures: A first approach. *Supercomputing frontiers and innovations*, 1(1), 2014.

[17] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang. Mic hardware and software architecture. In *High-Performance Computing on the Intel® Xeon Phi*, pages 13–56. Springer, 2014.

[18] F. Yazdanpanah, C. Álvarez, D. Jiménez-González, R. M. Badia, and M. Valero. Picos: a hardware runtime architecture support for ompss. *Future Generation Computer Systems*, 53:130–139, 2015.

[19] F. Yazdanpanah, C. Álvarez, D. Jiménez-González, and Y. Etsion. Hybrid dataflow/von-neumann architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 25(6):1489–1509, 2014.