UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Final Master Thesis

MASTER IN INNOVATION AND RESEARCH IN INFORMATICS
High Performance Computing

Asynchronous Runtime for Task-Based Dataflow Programming Models

Course 2016/17 - Spring fall - July 2017

Author
Jaume Bosch Pons

Advisor
Dr. Carlos Alvarez Martínez
(Computer Architecture Department - DAC)

Co-Advisor
Dr. Daniel Jiménez González
(Computer Architecture Department - DAC)

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Abstract

The importance of parallel programming is increasing year after year since the power wall popularized multi-core processors, 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. They allow describing a set of data dependences per task that the runtime uses to order the execution of tasks. This order is calculated using shared graphs, which are updated by all threads but in exclusive access using synchronization mechanisms (locks) to ensure the dependences correctness. Although exclusive accesses are necessary to avoid data race conditions, those may imply contention that limits the application parallelism. This becomes critical in many-core systems because several threads may be wasting computation resources waiting to access the runtime structures.

This master thesis introduces the concept of an asynchronous runtime management suitable for task-based programming model runtimes. The runtime proposal is based on the asynchronous management of the runtime structures like task dependence graphs. Therefore, the application threads request actions to the runtime instead of directly executing the needed modifications. The requests are then handled by a runtime manager which can be implemented in different ways.

This master thesis presents an extension to a previously implemented centralized runtime manager and presents a novel implementation of a distributed runtime manager. On one hand, the runtime design based on a centralized manager [1] is extended to dynamically adapt the runtime behavior according to the manager load with the objective of being as fast as possible. On the other hand, a novel runtime design based on a distributed manager implementation is proposed to overcome the limitations observed in the centralized design. The distributed runtime implementation allows any thread to become a runtime manager thread if it helps to exploit the application parallelism. That is achieved using a new runtime feature, also implemented in this master thesis, for runtime functionality dispatching through a callback system.

The proposals are evaluated in different many-core architectures and their performance is compared against the baseline runtimes used to implement the asynchronous versions. Results show that the centralized manager extension can overcome the hard limitations of the initial basic implementation, that the distributed manager fixes the observed problems in previous implementation, and the proposed asynchronous organization significantly outperforms the speedup obtained by the original runtime for real benchmarks.

Acknowledgments

I would like to thank Dr. Carlos Álvarez and Dr. Daniel Jiménez-Gonazález for bringing me the possibility of working in this master thesis at Barcelona Supercomputing Center. This work is partially supported by the European Union H2020 Research and Innovation Action through the Mont-Blanc 3 project (GA 671697), AXIOM project (GA 645496) and HiPEAC (GA 687698), by the Spanish Government (projects SEV-2015-0493 and TIN2015-65316-P), and by the Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272).

I consider and important part of this master thesis the people that I meet meanwhile working at Barcelona Computer Science, specially all people working at the Programming Models Group that gave me support during my work. I also want to thank my family and my parents for its support last years from Menorca, without you this master thesis would not been possible. Finally, I special thanks to Susanna Coma who always helps me to get focused and makes me happy every day.

Contents

List of Figures

List of Tables

List of Listings

Introduction

Chapter 1

Parallel programming has become a common topic in computer science. Nowadays, almost any electronic device has a multicore processor and therefore, the importance of parallel programming models and runtime libraries to support the application's parallelism is growing, and its optimization is becoming crucial.

In this chapter, section 1.1 introduces the problem of current runtime systems for task-based parallel programming models that motivates this master thesis. Following, section 1.2 introduces the objectives of the master thesis and how they can help to improve the current runtime implementations. Section 1.3 summarizes the published articles that are related to the thesis. Finally, section 1.4 introduces the structure of the document.

1.1 | Motivation

The multicore processors popularization started due to the end of Dennard scaling law which states that the power density of an integrated circuit can stay constant meanwhile the transistors get smaller. Until 2006, Dennard's law and Moore's law have been the reference for Processor manufacturers to periodically reduce the transistors length and increase the clock frequency which also increases the processors' performance. However, the leakage current grows much faster at small transistor sizes; therefore the clock frequency cannot increase without impacting the overall power consumption. Since the transistor still reduces its size periodically as Moore's law states, processor manufacturers started to introduce multiple cores in their processors to keep the processors' performance increase.

As multicore processors have become popular, parallel programming has become a need to take advantage of these processors. Instead of dealing with complex applications programmed for one specific processor architecture, parallel programming models decouple applications from hardware. Their goal is to allow programmers to indicate the potential parallelism in the applications' source code without directly managing it. There are several examples like MapReduce [2], OpenMP [3], OpenCL [4], StarSs [5], etc. The exposed parallelism is then managed by a runtime library that coordinates the application execution transparently to the application programmer.

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, developers can provide additional task information like data requirements. This information defines the task execution order enforced by the runtime libraries at execution time. The OpenMP standard introduced task dependences in the 4.0 version greatly influenced by the OmpSs programming model which extends the standard syntax with additional features.

The runtimes of these models are responsible for guaranteeing the task execution order correctness defined by the task data requirements. One or more task graphs are maintained in execution time, so the runtime checks the required data of each created task against on-the-fly tasks and delays its execution until the data is available. After each task execution, the runtime checks the successor tasks based on the data that the current task generates and data that other tasks require. If a successor task was only waiting for the data of the finished task, the successor becomes ready and its execution can start immediately. Usually, these events (task creation and task finalization) require to read and write the information in the corresponding task graph atomically to ensure the order correctness.

In a processor with a lot of cores, the probability of collisions between threads trying to access the task dependence graph increases. Each collision implies that a thread is wasting its computation time waiting for another one modifications. This problem that has currently started arising is expected to be an important bottleneck as the number of cores in the future processors is expected to keep growing. Thereby the access contention on some runtime structures will kill the application performance if runtimes do not redesign its internals to tackle the problem.

1.2 | Objectives

The objective of this project is to improve the current task-based parallel programming runtimes to avoid the runtime's structures contention expected in the many-core processors. Therefore, the project tries to maximize the utilization of the processor cores to run application's code and avoid active waiting on the locks. To this end, an asynchronous runtime structure is proposed where the runtime threads do not update the runtime structures directly. Instead, the threads request the needed actions to the runtime and someone will handle them in the future. This asynchronous approach avoids the problem of actively waiting for the exclusive access and allows the thread to return immediately to the application's code.

The new runtime workflow must be transparent for application developers and must be general enough to be used in a wide range of task-based parallel programming models. The project aims to modify the runtime internals but not the runtime API. Thereby any working application must be able to switch between the original and the asynchronous runtimes out of the box. Moreover, the proposed modifications in the project should not be runtime dependent, and any task-based parallel programming runtime may introduce them.

The threads' requests to the runtime are handled by a runtime manager who updates the runtime structures. Two runtime manager implementations are proposed in this master thesis: a centralized and a distributed implementations. On one hand, the centralized runtime manager implementation is based on an extra thread (DAS Thread, DAST) together with a mechanism to avoid the manager saturation. The runtime uses the DAST manager to update the runtime structures but allows the other threads to update the runtime structures, which is the original approach, if it is considered better for the overall application performance. On the other hand, the distributed runtime manager implementation (Distributed DAST, DDAST) is based on a distributed mechanism were any thread may become a runtime manager thread. Therefore, the runtime tries to use all the available threads in a smart way to restrict the runtime structures accesses.

Finally, the project's objective is also to evaluate the performance of the newly proposed runtimes. The new asynchronous runtimes should provide similar performance to the original runtime for a reduced amount of threads or when the application has a small number of tasks. Obviously, when the number of tasks and/or the number of threads is large, the new runtimes should be able to achieve better performance due to the better thread utilization, data locality and contention reduction.

1.3 | Published Articles

The project development is done in the Computer Science (CS) department of the Barcelona Supercomputing Center (BSC) by the Programming Models (PM) group, and it is an ongoing work that does not finish with this Final Master Thesis. For instance, some of the works presented in this master thesis are already published. Moreover, the work and knowledge of this project have been used in other projects. The following list presents the published articles related to this master thesis and briefly describes its contribution:

- Reorganització del runtime Nanos++ [1].
  Final bachelor thesis that analyzes the runtime entry points that modify the task dependence graph. It also proposes an initial implementation of a centralized runtime manager that executes such runtime functions, and analyses the application's performance when using the centralized runtime manager. In this contribution, the implementation is based on using a unique extra thread (DAST) to perform the runtime functions.
- Characterizing and Improving the Performance of Many-Core Task-Based Parallel Programming Runtimes [6].
  Paper presented in *Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM), 2017 Annual Workshop*, that introduces a full implementation of a centralized runtime manager with automatic load balancing between the manager and the workers. The implementation allows the application threads to keep the original runtime structure if the pressure over the runtime manager is too high. The paper characterizes the runtime overheads of different implementations and analyzes the runtime performance running different benchmarks.

- Performance analysis of a hardware accelerator of dependence management for task-based dataflow programming models [7].
  Paper presented at *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium*, that presents the very first functional hardware prototype inspired by Picos (hardware accelerator for dependence management of fine grained tasks). The paper presents simulation results where the use of the hardware accelerator reduces the runtime overheads, the results are built considering the project's knowledge to characterize the runtime overheads.

- General Purpose Task-Dependence Management Hardware for Task-based Dataflow Programming Models [8].
  Paper presented at *International Parallel & Distributed Processing Symposium (IPDPS), 2017 IEEE International Symposium*, that presents Picos++ a general purpose hardware accelerator to manage the inter-task dependences efficiently in both time and energy. The paper presents a hardware/software co-design that integrates the hardware manager into a task-based programming model runtime. This integration is build considering the project's knowledge, substituting the centralized manager (DAST) by the hardware manager.

Also, we plan another article summarizing the unpublished work presented in this master thesis. The publication will contain the new distributed implementation of the runtime manager (DDAST) and the performance evaluation on different many-core processors.

1.4 | Document structure

The remainder of the document is structured as follows:

- Chapter 2 reviews the most relevant related work.
- Chapter 3 introduces the OmpSs parallel programming model and its runtime (Nanos++), that is used as a baseline to implement the asynchronous runtime.
- Chapter 4 explains the centralized manager implementation (DAST).
- Chapter 5 presents the runtime characterization and performance results of the runtime with the DAST manager.
- Chapter 6 explains the distributed manager implementation (DDAST) for the asynchronous runtime.
- Chapter 7 presents the manager tunning analysis and the performance results of the runtime with the DDAST manager.
- Finally, chapter 8 summarizes the work done, provides some concluding remarks and analyzes the possible future work in the project context.

Related Work

Chapter 2

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 open source and are the ones used to test our model, can execute inter-node and intra-node applications [9] 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.

Previous works discussed the task scheduling and dependences resolution overheads in data-driven task-based models like OpenMP and OmpSs. TurboBLYSK [10] is a framework which implements the OpenMP 4.0 with a custom compiler and a highly efficient runtime schedule 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, our proposal only uses the information provided by default to reduce the overall task management overhead, so our optimization is transparent to the programmers. However, in some sense, our works are complementary.

Other task-based programming models like Intel Threading Building Blocks [11] and Charm++ [12] use a execution model that is more pure dataflow than the OmpSs/OpenMP model which has a hybrid (control/dataflow) model [13]. 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 [14] 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 [14], but the dataflow model that they have moves the complexity of task-graph management into the application development process like in TBB.

Other works, which also try to accelerate current runtimes, propose moving part of the runtime into a specific hardware of FPGAs. Some examples are Nexus# [15] and Picos [7]. They present different hardware designs that can manage tasks dependences 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 aims 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 in this project 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. In contrast, this project improves the existing parallel programming model runtimes with software ideas that do not require additional hardware.

OmpSs: A Task-Based Parallel Programming Model

Chapter 3

The task-based parallel programming model used as a baseline in this project to develop the asynchronous version of the runtime is OmpSs. Currently, this programming model is supported by the Mercurium compiler and the Nanos++ runtime library. For this reason, the following sections explain:

- Section 3.1 explains the key features of the OmpSs Programming Model.
- Section 3.2 explains Mercurium: the source-to-source compiler that supports the OmpSs syntax.
- Section 3.3 explains Nanos++: the runtime library that handles the executions.

3.1 | The Programming Model

The OmpSs programming model is a task-based parallel programming model developed at BSC and composed by a set of directives and library routine. The name OmpSs comes from two others programming model names, OpenMP [3] and StarS [5]. The goal of the programming model was extend the OpenMP syntax with some of the StarSs to provide a productive environment for HPC applications development. Productive means that the applications developed in OmpSs achieve a reasonable performance compared to similar solutions for the same architectures and means that the development cost is small and does not require huge changes in the applications [17] [18].

On one hand, OmpSs takes from OpenMP the philosophy of providing a way to produce a parallel version of the application adding annotations that do not require modifications in the source code. These annotations allow the compiler to generate a parallel version of the application replacing the annotations by runtime API calls. This philosophy is intended to simplify the development process leading to a better productivity. On the other hand, OmpSs takes from StarSs the thread-pool model. In contrast to the fork-join model used by OpenMP, StarSs model has an implicit parallelism during all the execution so, programmers do not have to annotate the parallel regions [17] [18].

Task annotation

The main annotation in OmpSs is the task clause which defines a code region that will be asynchronously executed. The tasks created can be concurrently executed by any thread when they are ready. The task execution order can be defined by the programmers using the `in(...)`, `out(...)` and `inout(...)` clauses that extend the task annotation. These clauses define the data dependences for each task and implicitly define the dependences between tasks. The runtime is responsible for synchronizing task executions to guarantee the dependences.

The data dependence syntax in the OmpSs clauses supports variables and array regions. Moreover, the syntax for the array regions is rich and the same region can be annotated in different ways:

- Discrete elements in the array:
  `in( v[0], v[i], v[i*2+j] )`
- Region of N elements in the array:
  `out( [N]v, [8]v )`
- Region of elements in the array with initial and end positions (both included):
  `inout( v[first: last], v[0: 7], v[0: N-1] )`
- Region of elements in the array with initial position and number of elements:
  `inout( v[first; num_elems], v[0; 8], v[0; N] )`

In addition to the implicit synchronization created by the data dependences, the programmer can introduce explicit synchronization points using the taskwait annotation. This ensures that after the annotation all tasks created before it are executed and all data generated by tasks is available with the latest values.

```c
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]);
} }
```

Listing 3.1: OmpSs code annotation example

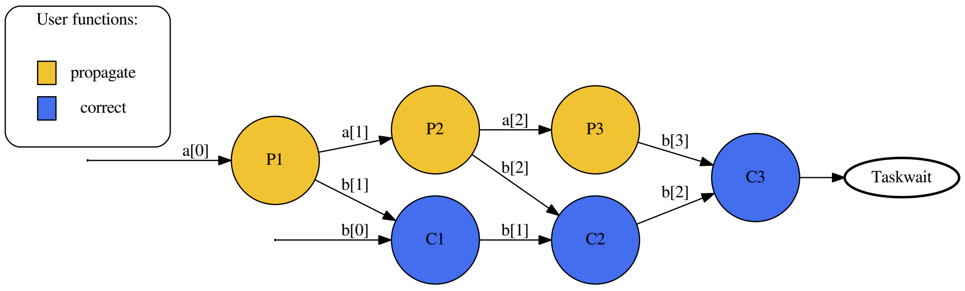


Figure 3.1: OmpSs task graph for listing 3.1 (N=3)

An example code [19] of a C function parallelized with OmpSs is shown on listing 3.1. The function contain two function calls that are annotated with the task directive: *propagate* and *correct*. Thus, the calls to those functions are asynchronously executed when the data dependences defined in the task annotation are satisfied. The resulting task dependence graph (for $N = 3$) is shown in figure 3.1 where the nodes are tasks and the edges true dependences among them. There is a true dependence between the i^{th} *propagate* and the i^{th} *correct* tasks due to the i^{th} element of *b*. There is a true dependence between the i^{th} *propagate* and the $i^{th} + 1$ *propagate* tasks due to the i^{th} element of *a*. Finally, there is a true dependence between the i^{th} *correct* and the $i^{th} + 1$ *correct* tasks due to the i^{th} element of *b*.

3.2 | Mercurium

Mercurium is a C/C++/Fortran source-to-source compilation infrastructure aimed at fast prototyping developed by the Programming Models group at the Barcelona Supercomputing Center [20].

Mercurium is mainly used together with the Nanos++ Runtime Library to implement the OmpSs programming model. Also, it implements the OpenMP 3.1 standard. Apart from that, since Mercurium is quite extensible it has been used to implement other programming models or compiler transformations, examples include Cell Superscalar, Software Transactional Memory, Distributed Shared Memory or the ACOTES project, just to name a few [20].

Figure 3.2 show a simplified view of an application compilation and linkage using Mercurium. The files are provided to the Mercurium profile which applies the source-to-source transformations needed (like annotation replacement by API calls to the Nanos++ runtime library). After that, the native compiler is used to generate the object files, which are linked against the Nanos++ library to generate the executable.

In the context of this project, there are not planned modifications in the compiler. This is because the changes are transparent to the users and this means that the runtime API is untouched. Otherwise, the runtime library API changes may require a recompilation of the

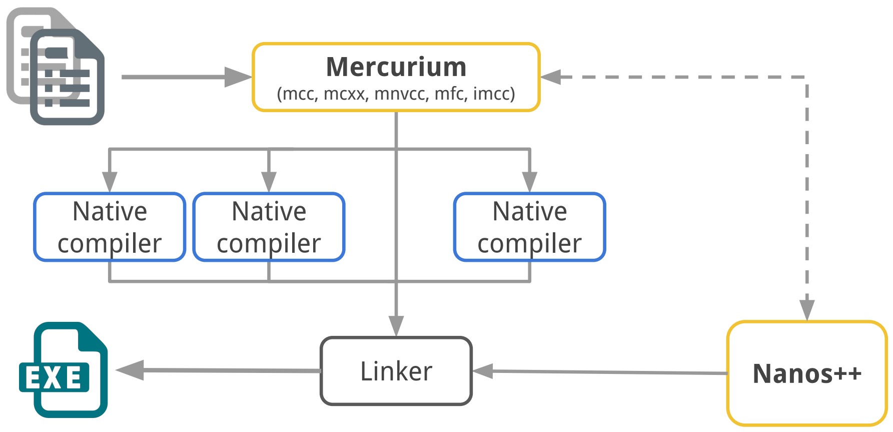


Figure 3.2: Mercurium compiler structure

applications or a new Mercurium instance conscious of new semantics.

3.3 | Nanos++

Nanos++ is a runtime library designed to serve as runtime support in parallel environments. The runtime is developed at the Barcelona Supercomputing Center within the Programming Models group, and its main use is support the OmpSs programming model. Appart from OmpSs Nanos++ also supports most of the OpenMP 3.1 features and includes some additional extensions (some of them also introduced in following OpenMP releases) [21].

The runtime provides the required services to support task parallelism based on data dependences. Data parallelism is also supported by means of services mapped on top of its task support. Task are implemented as user-level threads when possible (currently x86, x86-64, ia64, arm, ppc32 and ppc64 are supported). It also provides support for maintaining coherence across different address spaces (such as with GPUs or cluster nodes) by means of a directory/cache mechanism [21].

The main purpose of Nanos++ RTL is to be used in research of parallel programming environments. The runtime tries to enable easy development of different parts, so researchers have a platform that allows them to try different mechanisms. As such it is designed to be extensible by means of plugins. The scheduling policy, the throttling policy, the dependence approach, the barrier implementations, slicers and worksharing mechanisms, the instrumentation layer and the architectural dependant level are examples of plugins that developers may easily implement using Nanos++ [21]. Figure 3.3 shows a simplified vision of such runtime structure.

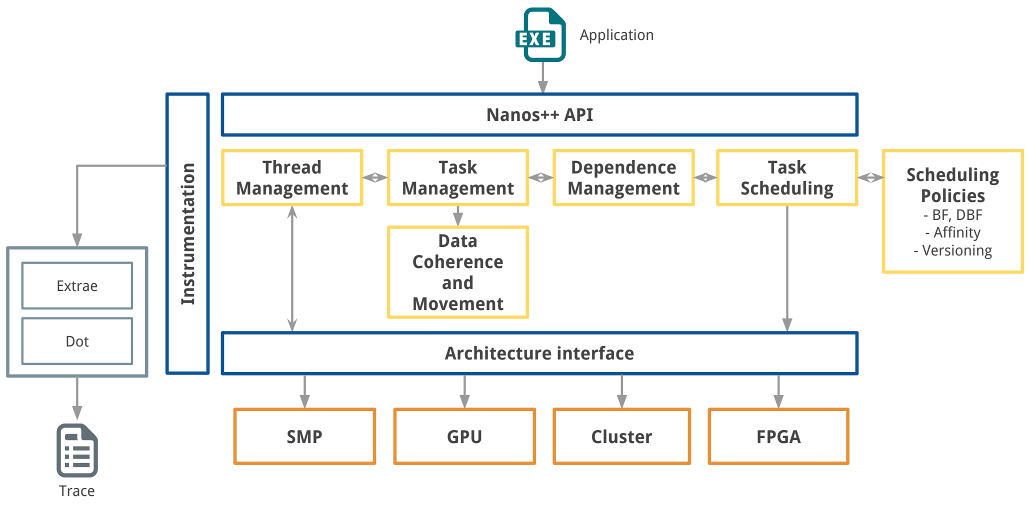


Figure 3.3: Nanos++ runtime structure

Task life cycle

Task representation inside Nanos++ is made by one Work Descriptor (WD) for each task. Each WD contain all needed information to manage the task during its life cycle. For instance, the WDs store the data dependences of each task. The parent task, which is the task being run when the child task is created, contains the task-graph with the relations of its children. This limits the tasks to depend on only sibling tasks, but the 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 are protected by spin-locks because different sibling tasks can finalize at the same time and/or collision with another sibling task creation.

The different steps in the task life cycle are summarized following and shown in figure 3.4 with the transitions between states.

1. Task creation.
   At this step, the WD structure is allocated and initialized with the information provided in the annotations related to the task. Moreover, the values of function arguments or local variables are stored in order to execute the code asynchronously.

2. Task submission.
   At this step, the data dependences of the task are stored in the WD and introduced in the task-graph to compute the predecessor WDs. If no predecessors are found, the task can immediately become ready. How the predecessors are computed depends on the used dependences plugin which can be changed in each execution.

3. Task becomes ready.
   At this step, task's data dependences have been satisfied or task's blocking condition had

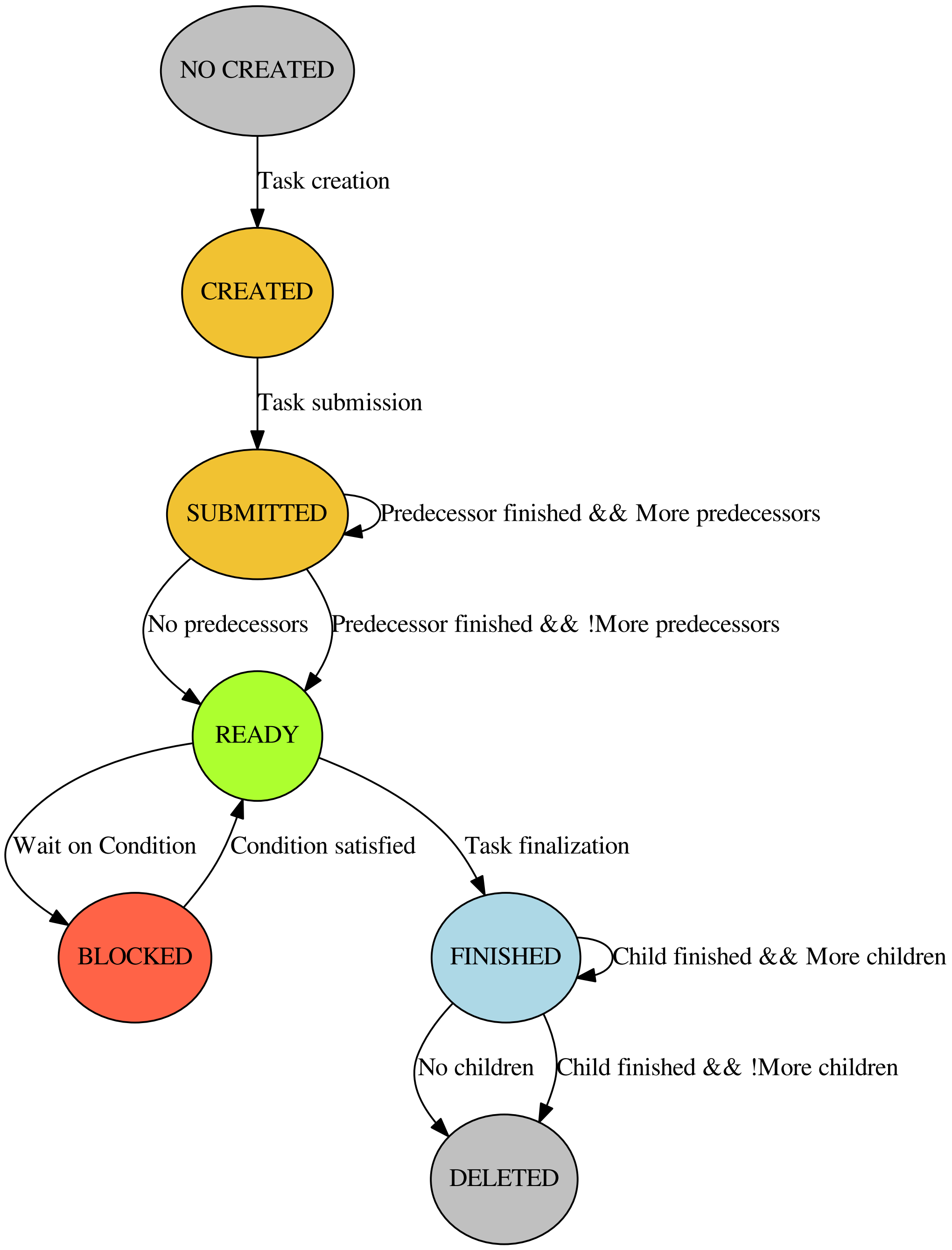


Figure 3.4: Nanos++ task life cycle diagram

become true. Consequently, the task execution can start. How the task will be executed depends on the used scheduling policy that can be changed in each execution.

4. Task becomes blocked.
   At this step, the task cannot proceed its execution until some condition becomes true. For example, when a task contains a `taskwait` annotation it becomes blocked until its children tasks finish.

5. Task finalization.
   At this step, the task has finished its execution and the successor WDs may become ready if they only depend on the finalized task. Therefore, the WD can be deleted if it does not have children tasks. Otherwise, the children tasks might reference the parent WD in its finalization to access the task-graph.

6. Task deletion.
   At this step, the WD can be safely deleted because no more references to it will be done.

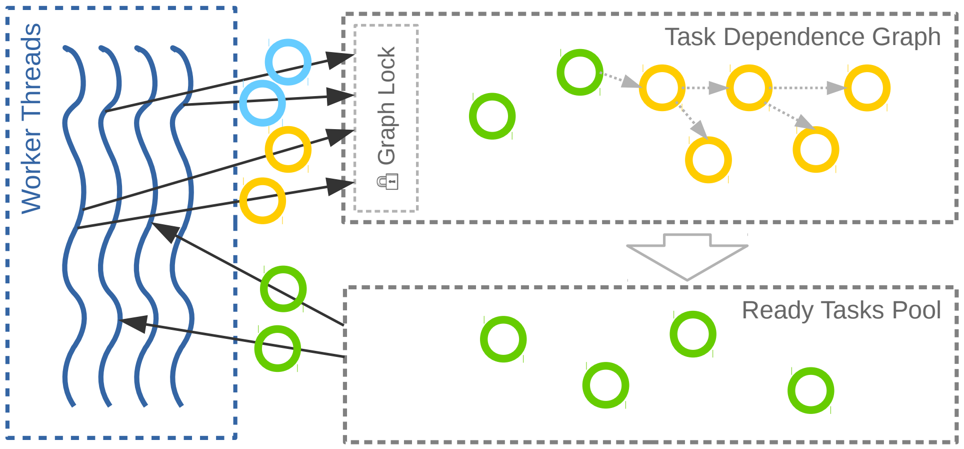


Figure 3.5: Nanos++ task flow over runtime structures

Each task state is mainly related with one runtime component, therefore the WDs flow over the different runtime structures during its life cycle. Figure 3.5 shows a simple representation of the tasks flow where each circle represents a task. Each circle color is associated with a task state like shown in figure 3.4: yellow for a task being created or a submitted task, green for a ready task and blue for a finished task. First, a thread pushes the created tasks into the task dependence graph to compute the task order. Moreover, the thread must acquire the graph lock before submitting the task and making it become submitted. Then, other threads "push" the finalized tasks into the task dependence graph through the same graph lock to notify the successor tasks. In addition, this action removes the finished task from the graph and adds the tasks that become ready into the ready tasks pool. Finally, the worker threads try to acquire ready tasks from the ready tasks pool to execute them. Also, the management (insertion, deletion, etc.) or tasks into the ready tasks pool may require acquiring some lock, but it is not shown on the figure 3.5 as the pool implementation depends on the scheduling policy.

Centralized Runtime Manager (DAST)

Chapter 4

The first design and implementation of an asynchronous runtime for task-based parallel programming models were proposed in the Final Bachelor Thesis: *Reorganització del Runtime Nanos++* [1], and extended in the paper: *Characterizing and improving the performance of many-core task-based parallel programming runtimes* [6]. The runtime implementation is based on a centralized manager thread (DAST) which manages the runtime structures according to the received requests. The requests are created by the worker threads during the application execution instead of directly modifying the runtime structures. This operational flow is further explained in:

- Section 4.1 which describes the initial basic implementation of the centralized runtime manager.
- Section 4.2 which presents the extension developed over the initial implementation to overcome the observed limitations.

4.1 | Forced Manager

The asynchronous runtime execution model decouples the runtime functionality and the task execution by introducing a decoupled manager that does the main runtime functions. At each runtime API call, the worker threads request an action to the manager instead of performing it directly. Therefore, the manager modifies the runtime structures according to the received requests. The aim is to optimize the productivity of all the threads, removing the contention created at some structures (like dependence graphs) when the threads are wasting time waiting for exclusive access.

The communication between the workers and the manager is done pushing requests, which contain the information to allow the later execution of the runtime function, inside a queue. This queue can be different for each worker thread because the messages/requests generated from different workers are independent and can be satisfied out of order. The order that the manager must guarantee is the FIFO order for the same thread messages to ensure the correctness when computing the dependences. Such organization only requires a worker-manager synchronization (when messages are inserted/removed). Meanwhile, the original runtime organization requires a global synchronization (when a runtime structure is modified).

The requests of runtime functions can be of three types:

- New task.
  Runtime code executed to compute the predecessor tasks for a new task. If no predecessors are found, this request also executes the runtime code to schedule the ready task.
- Done task.
  Runtime code executed to notify the finalization of a task to its successors. If the notified task does not have more predecessors, this request also executes the runtime code to schedule the ready task.
- Delete task.
  Runtime code executed to clean up and delete the WD associated to a task. It has to be a separate message to ensure that neither DAST nor the worker threads try to access a deleted work descriptor.

The asynchronous runtime with the centralized manager uses the Nanos++ runtime source code as a baseline, and the centralized manager implementation is based on an extra thread (DAST). The manager thread does not execute application code, only runtime code to handle the incoming requests from the other threads. Therefore, the DAST executes the runtime code regions described in the design.

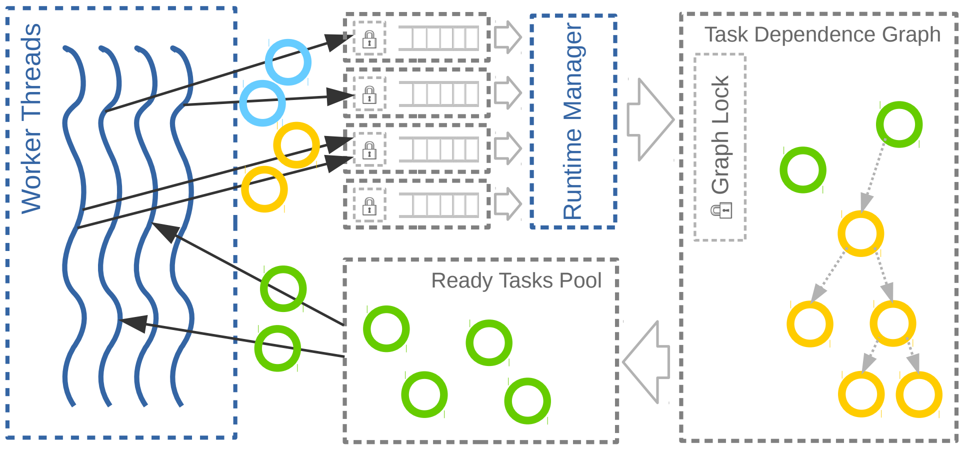


Figure 4.1: DAST runtime task flow over runtime structures

The manager integration and new task flow can be seen in figure 4.1 which extends the Nanos++ task flow figure (3.5). In the asynchronous runtime, the worker threads push the task events (task submission, task finalization and task deletion) into queues instead of directly going to the runtime structures. After that, the runtime manager updates the runtime structures according to the requests in the queues.

Limitations

After the implementation was presented as a Final Bachelor Thesis, several problems in the initial design arose. The most common problem observed was that, under some conditions, the worker threads might create requests faster than the manager satisfies them; and the worker threads become idle waiting for ready tasks. Summarizing, the centralized manager creates a bottleneck that may limit the application parallelism and the achieved performance. To overcome this bottleneck, the centralized manager might be implemented using more than one thread to overcome the saturation. However, this approach would lead to devote more threads to managing the runtime structures and fewer threads to executing the application tasks, so the performance may drop. To tackle the problem without increasing the number of manager threads an extension using a hybrid manager was proposed.

4.2 | Hybrid Manager

The proposed implementation is an improvement over the previous design and is based on a mechanism to keep some requests directly executed by the worker threads. The idea is to keep the original runtime organization when it is considered better than the asynchronous version. Consequently, the runtime decides before creating each request if it will be created and asynchronously executed or if it has to be immediately executed.

The decision of pushing a message in the manager queue, and execute it asynchronously, or execute it directly inside the API call, like in the original Nanos++, is made following different criteria depending on the message type.

```
if ( queueContainNewTaskRequest() ) {
  prev = getLastNewTaskRequest()
  if (not tasksAreSiblings(request, prev) ) {
    return FALSE
} }
return TRUE
```

Listing 4.1: Pseudo-code of new task request push policy

The *New task* messages are only sent from one WD at the same time to avoid the serialization of the dependence calculation for independent task graphs. In other words, this ensures that independent actions (dependence calculation of non-sibling tasks) are not serialized and can be executed in parallel. When one worker pushes one *New Task* message into the queue only itself will be able to push more *New task* messages until the DAST thread processes all requests of the type. Figure 4.1 shows a pseudo-code to decide whether making the request or not using that behavior. When the manager queue does not contain any *New task* message, the request can be pushed (line 6). Otherwise, the runtime checks if the tasks of current and existing requests are siblings.

```
1  nreq = numberRequestsWorker()
2  wlow = waitingForLowerBound()
3  nrdy = numberReadyTasks()
4  if ( wlow && nreq <= LOWER_BOUND ) {
5    unsetWaitingForLowerBound()
6  }
7  if ( nreq > UPPER_BOUND ) {
8    setWaitingForLowerBound()
9    return FALSE
10 } else if ( wlow && nreq > LOWER_BOUND ) {
11   return FALSE
12 }
13 return nrdy > MIN_TASKS
```

Listing 4.2: Pseudo-code of done task request push policy

Figure 4.2 shows the pseudocode of the *Done task* push policy that rejects push the message in three cases:

1. The number of pending requests from the worker ($nreq$) is bigger than an upper boundary (line 9). This case activates an hysteresis loop (line 8)
2. The policy has activated the hysteresis loop, and the number of pending requests from the worker ($nreq$) is bigger than the lower boundary (line 11).
3. The number of ready tasks to be executed by the worker ($nrdy$) is smaller than a threshold (line 13).

Otherwise, the request can be submitted (line 14). The hysteresis loop limits are dynamically tuned to optimize the load balancing and try to advance as much as possible the release of task dependences. For instance, the limits are doubled when the hysteresis loop is activated and is deactivated in the next call to the *Done task* push policy. Otherwise, the limits are halved after the hysteresis loop activation. After an empirical testing, the best initial values seem to be 1 for `MIN_TASKS` and 3-5 for `LOWER_BOUND-UPPER_BOUND`.

The *Delete task* requests are pushed only if the *Done task* message of the same WD is still inside the queue. In any other case, either the runtime decided to do not push the *Done task* message or the DAST thread already processed it, the worker will directly free the memory used to store the task information.

DAST Evaluation

Chapter 5

The evaluation of the asynchronous runtime based on the DAST manager implementation is explained in this chapter. The objective of the evaluation is to compare the behavior and performance of the DAST-based runtime against the original one. To this end, different benchmarks, well known in the High Performance Computing world, and different machines that have different architectures are used. The structure of the chapter is:

- Section 5.1 explains the software/hardware environment used to run the different benchmarks.
- Section 5.2 briefly describes the benchmarks used to characterize and test the performance of the new design.
- Section 5.3 characterizes the time cost of relevant runtime parts.
- Section 5.4 presents the scalability and performance results for each benchmark.
- Finally, section 5.5 explains the limitations that the centralized implementation has.

5.1 | Environment

The tests have been run on an Intel Xeon Phy [22] co-processor of Knights Corner (KNC) architecture, which is the first commercial generation of such processors. The used model is *Intel® Xeon® Phi(TM) CPU 7120*, and it has 61 x86 cores that can run up to 4 simultaneous threads with 16GB of shared RAM memory. The results gathered from this many-core architecture behave similarly to the ones obtained from other multi-core platforms like Intel Xeon, so the later ones are not presented.

In all executions, the first two cores are reserved for the OS thread (the co-processor runs a small Linux, and the thread provides some services) and master Nanos++ thread. Similarly, the third core is reserved for the manager thread (DAST) when needed. These reserved cores do not run more than one thread per core because the core-resource sharing will make the whole execution become slower. Also, the Nanos++ scheduling policy used is the Distributed Breadth First (DBF) but adapted to allow the manager dynamically distribute the work among all workers according to the loads.

The version of the Intel® Math Kernel Library (Intel(R) MKL) used by the applications is 2011.2.2. The compiler used to cross-compile the applications and the runtimes is the GNU C Compiler Collection (GCC) version 5.3.0.

Another conditions that have been considered to enhance the evaluation quality, avoid external inferences and facilitate the reproducibility of the results are:

1. All the machine's nodes used during all executions are exclusively reserved for the tests, despite the number of used cores is smaller than the available.
2. The applications are compiled with the `-O3` code optimization level.
3. The time measurements are repeated at least five times and the best execution time is kept for each configuration to do the comparisons. In addition, the executions have been repeated if the best timing only appears once.

The different runtime versions evaluated are the following:

- *Nanos++*.
  Baseline OmpSs runtime (version 0.7.2).
- *DAST Runtime*.
  Runtime with the centralized runtime manager implementation (DAST). This version is implemented on top of *Nanos++* runtime (version 0.7.2).
- *DAST (forced)*.
  DAST Runtime with the request push policies disabled (mandatory submission of requests). It is useful to validate the behavior of the push policies.
- *DAST (no locks)*.
  DAST (forced) without the dependence graphs locks that are not needed in the forced version. This version is useful to check the lock overheads.

5.2 | Benchmarks

The used benchmarks are explained in this section. For each one, its execution arguments are explained and provided with the number of created tasks in each configuration and any other remarks that may be valuable for reproducibility. In all of them, some timing instructions are added after the sequential initialization and after the final global taskwait. The wall-clock time elapsed between these two points is defined as execution time in the rest of the section.

For each benchmark, two different sets of execution parameters are used to create two tasks granularities: coarse grain tasks and fine grain tasks. In addition, the benchmark execution parameters are selected considering the following points:

1. Have a big enough problem size to gather significant results.
2. Have a reasonable performance in the coarse grain Nanos++ executions.
3. The fine grain is half of coarse grain (increasing the runtime load).

Synthetic Benchmark

The synthetic benchmark creates two waves of `NUM_TASKS` tasks with a duration defined by the `ITERS` parameter. Any pair of tasks created from the same wave are independent between them and the $ith-$task of the second wave depends on the $ith-$task of the first wave. This benchmark is useful to evaluate the adaptability and performance of *DAST Runtime* when the runtime has a huge load. The pseudo-code that implements the described pattern can be seen in listing 5.1 and the resulting task dependence graph in figure 5.1. Each node of figure 5.1 represent a task and the edges between them the data dependences detected. Finally, the used values for the arguments are summarized in the table 5.1 with the number of tasks created in each configuration.

```
for (int i = 0; i < NUM_TASKS; i++) {
 #pragma omp task inout(a[i])
 foo1(a[i]);
}
for (int i = 0; i < NUM_TASKS; i++) {
 #pragma omp task in(a[i]) out(b[i])
 foo2(a[i], b[i]);
}
#pragma omp taskwait
```

Listing 5.1: Pseudo-Code of Synthetic benchmark

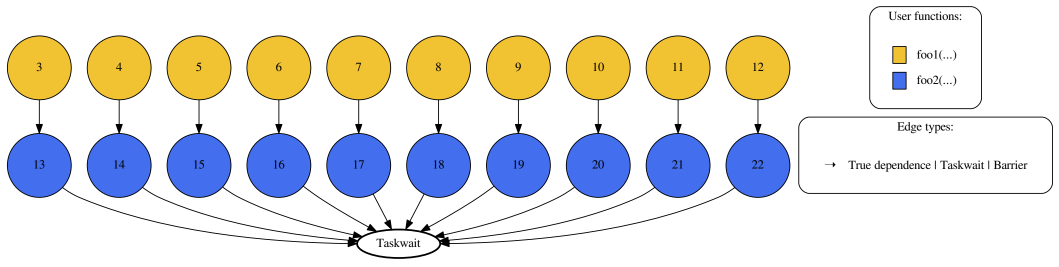


Figure 5.1: Synthetic Benchmark task graph showing the data dependences between tasks

NUM_TASKS	**Coarse Grain**		**Fine Grain**	
	ITERS	**Num. Tasks**	ITERS	**Num. tasks**
10000	50000	20000	25000	20000

Table 5.1: Synthetic Benchmark execution arguments

Cholesky

The Cholesky benchmark [23] computes the cholesky decomposition of a matrix in parallel. The application takes two arguments: the matrix dimension (MATRIX_SIZE) and the block dimension (BLOCK_SIZE). Therefore, the matrix with MATRIX_SIZE∗MATRIX_SIZE elements is divided into sub-matrices with BLOCK_SIZE∗BLOCK_SIZE elements. After that, each task deals with some of these sub-blocks. The used values for the MATRIX_SIZE and BLOCK_SIZE arguments are summarized in the table 5.2.

Matrix Size	**Coarse Grain**		**Fine Grain**	
	Block Size	**Num. Tasks**	**Block Size**	**Num. tasks**
8192	256	5984	128	45760

Table 5.2: Cholesky execution arguments

Figure 5.2 shows the data dependences between the tasks created in a small execution of the benchmark (MATRIX_SIZE is 1024 and BLOCK_SIZE is 128). The nodes represent the task instances (each color represents one task type) and the edges between them the data dependences detected. As can be seen, the benchmark has an irregular pattern with a variable parallelism among the execution.

Matrix Multiply

The Matrix Multiply (Matmul) benchmark [23] computes the product of two matrices in parallel. The application takes three main arguments: the matrix dimension (MATRIX_SIZE), the block dimension (BLOCK_SIZE) and the number of repetitions. Therefore, the matrices with MATRIX_SIZE∗MATRIX_SIZE elements are divided into sub-matrices with BLOCK_SIZE∗BLOCK_SIZE elements. Consequently, each task uses three of these sub-matrices to compute the corresponding multiplication.

The values for the MATRIX_SIZE and BLOCK_SIZE arguments used are summarized in the table 5.3. In all executions, the number of repetitions is set to 1.

Figure 5.3 shows the data dependences between the tasks created in a small execution of the benchmark (MATRIX_SIZE is 256 and BLOCK_SIZE is 64). The nodes represent the task instances and the edges between them the data dependences detected. As can be seen, the benchmark has a regular pattern with several independent chains that group all tasks working with the same

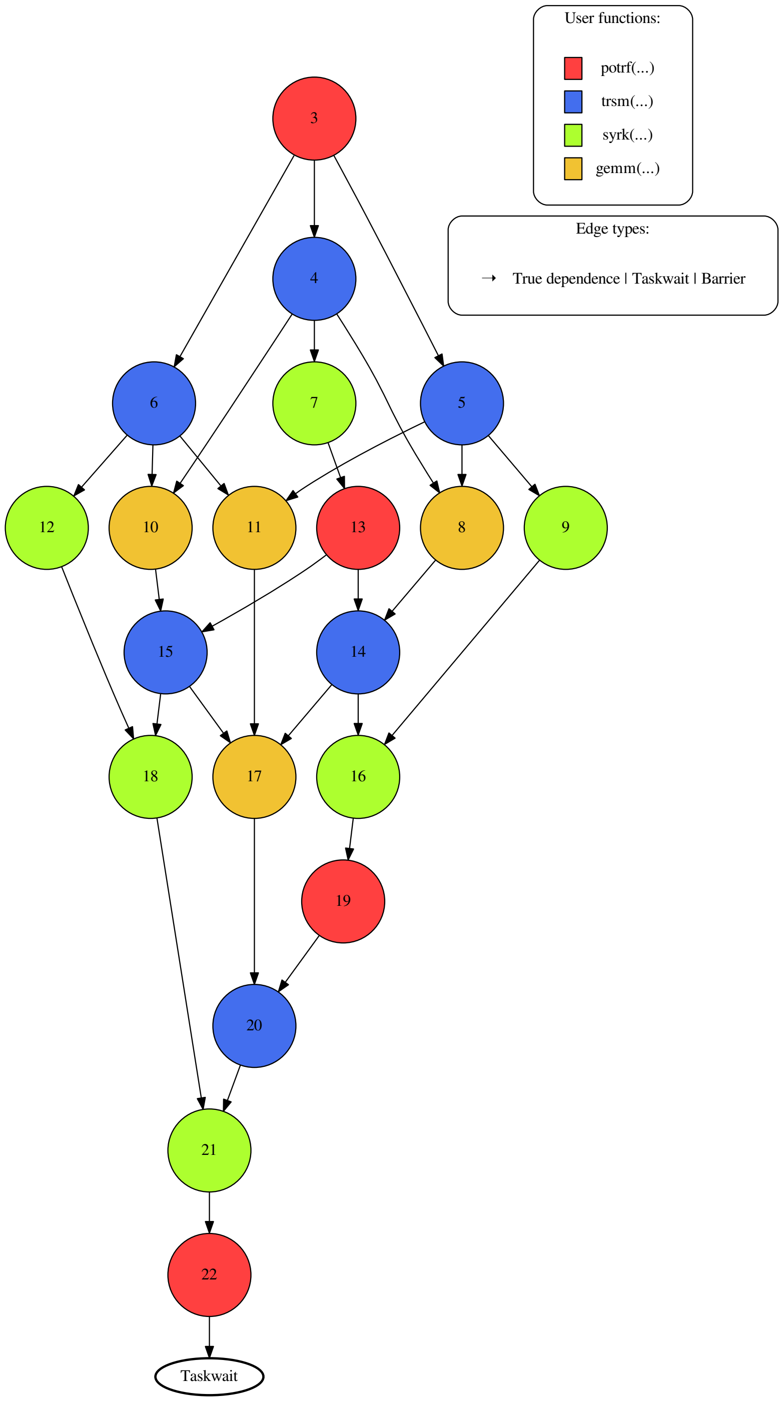


Figure 5.2: Cholesky task graph showing the data dependences between tasks

Matrix Size	Coarse Grain		Fine Grain	
	Block Size	Num. Tasks	Block Size	Num. tasks
4096	256	4096	128	32768

Table 5.3: Matmul execution arguments

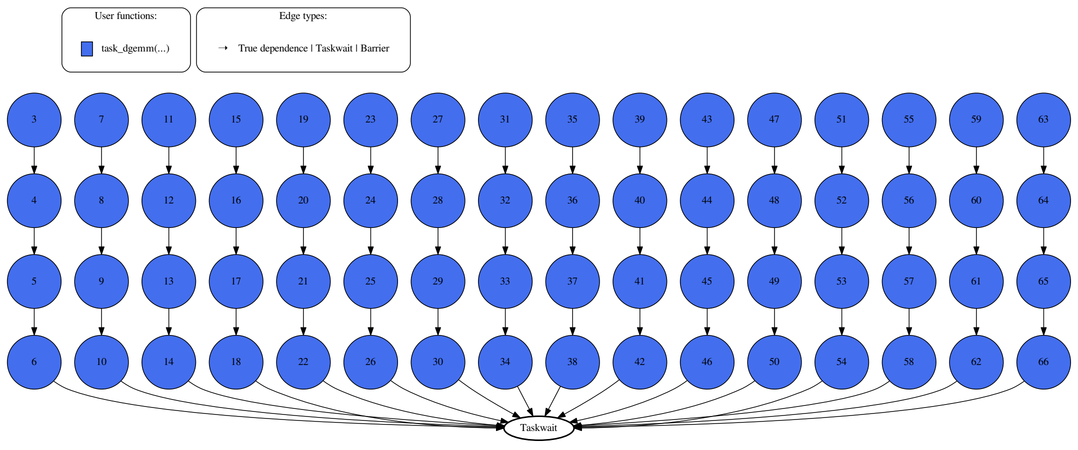


Figure 5.3: Matmul task graph showing the data dependences between tasks

output block.

Sparse LU

The Sparse LU benchmark [23] computes the Lower Upper (LU) decomposition of a matrix in parallel. The application takes two arguments: the matrix dimension (`MATRIX_SIZE`) and the block dimension (`BLOCK_SIZE`). Therefore, the matrix with `MATRIX_SIZE`$*$`MATRIX_SIZE` elements is divided into sub-matrices with `BLOCK_SIZE`$*$`BLOCK_SIZE` elements. Moreover, each task deals with some of these sub-blocks. The used values for the `MATRIX_SIZE` and `BLOCK_SIZE` arguments are summarized in the table 5.4.

Matrix Size	Coarse Grain		Fine Grain	
	Block Size	Num. Tasks	Block Size	Num. tasks
2048	64	1512	32	11472

Table 5.4: Sparse LU execution arguments

Figure 5.4 shows the data dependences between the tasks created in a small execution of the benchmark (`MATRIX_SIZE` is 1024 and `BLOCK_SIZE` is 256). The nodes represent the task instances and the edges between them the data dependences detected. As can be seen, the benchmark has a much more complex and irregular pattern than the Matmul benchmark.

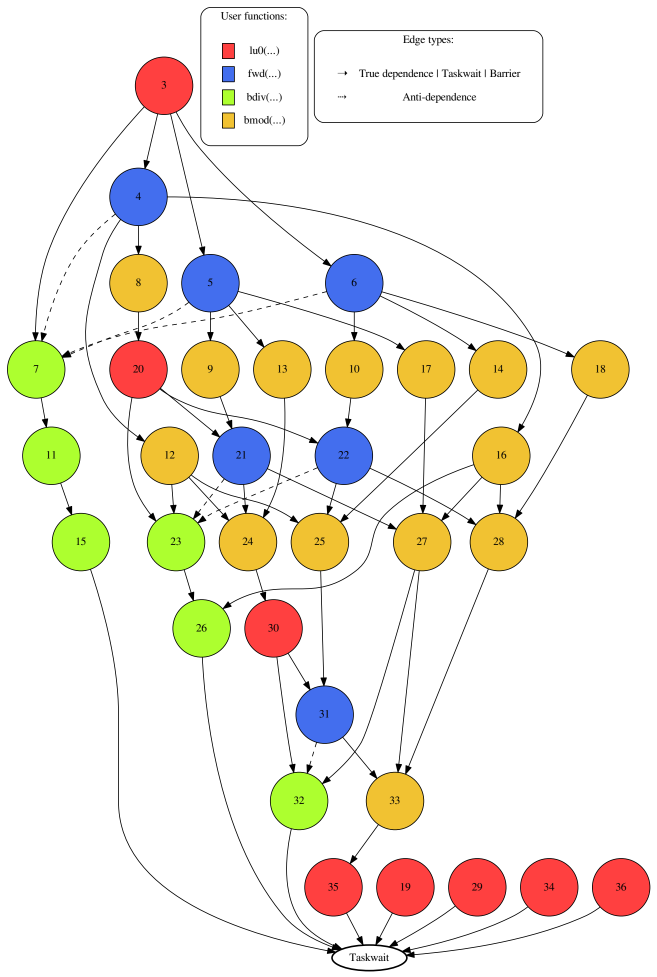


Figure 5.4: Sparse LU task graph showing the data dependences between tasks

5.3 | Runtime Characterization

This section analyzes the cost of the three runtime functionalities executed by the runtime manager (new, done and delete task) to show how it behaves when the number of worker threads increases. Also, the same analysis is showed for the cost of lock acquiring in the different runtime versions.

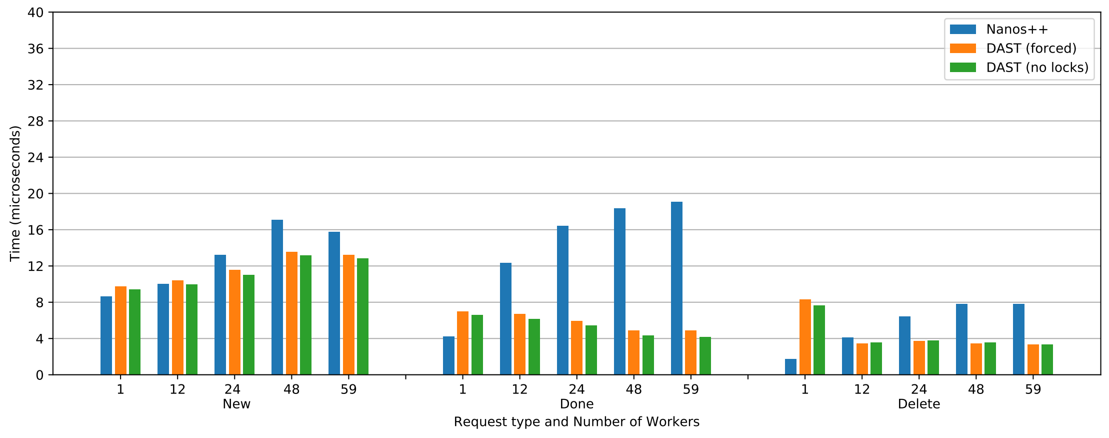
Requests execution cost

Figure 5.5 shows the average elapsed time in microseconds (y-axis) needed to execute the functions related to each message. These times include the period when the workers are trying to acquire the lock and maybe waiting for its release. And besides, the results consider different amounts of workers (x-axis) for each runtime version (Nanos++ baseline, *DAST (forced)* and *DAST (no locks)*). Figure 5.5a shows the elapsed time for the synthetic benchmark implemented with coarse grain tasks and figure 5.5b for the Matmul with 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. This metric is used instead an arithmetic mean to discard outliers and show the trends more clearly in each case.

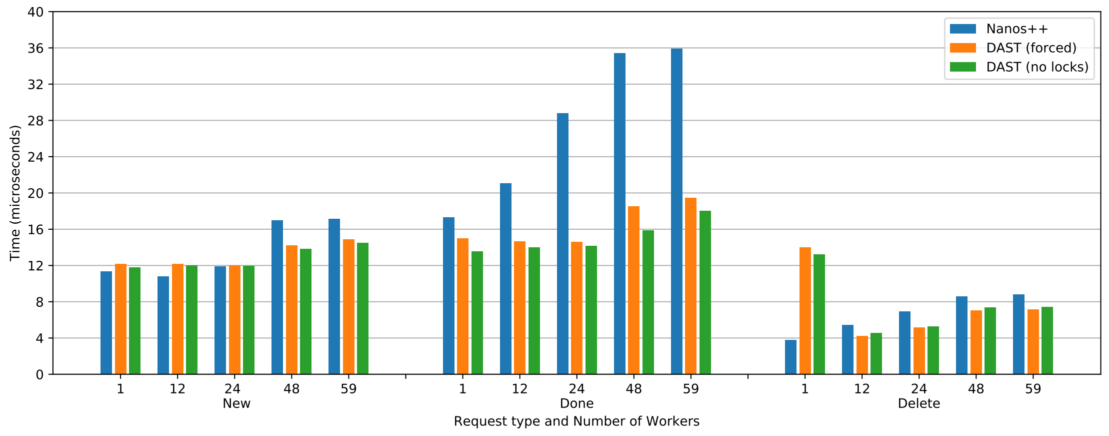
The *New task* request code takes similar time between the three runtime versions for small amounts of threads. This is because all of them behave similarly and one thread creates all the tasks at the beginning. However, as can be seen in figure 5.5, the Nanos++ version increases the time more than the 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 5.5a 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 3.5) and this increases the waiting time to get access, preventing the performance. The times in figure 5.5b 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, two different behaviors can be observed depending on the number of workers. On one hand, with Nanos++ runtime, Delete task functions costs is very small when running in one worker as can be seen in figure 5.5. 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 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.



(a) Synthetic Benchmark

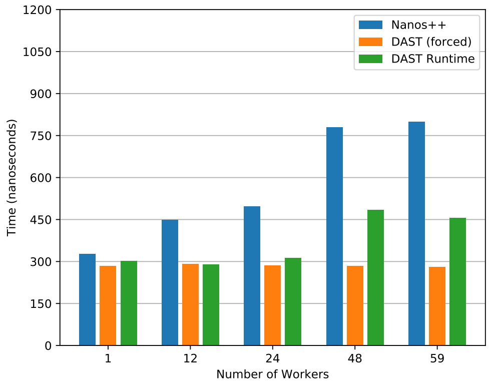


(b) Matmul

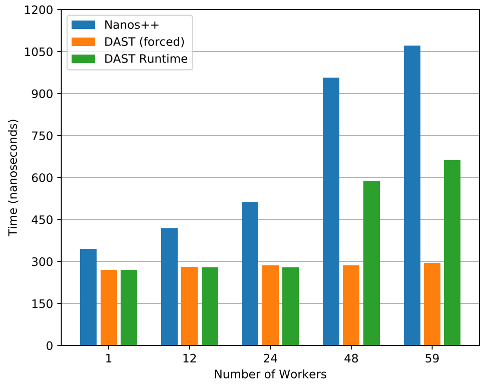
Figure 5.5: Execution cost of runtime functions for each DAST request type

Lock acquisition cost

To better understand the differences between runtimes, figure 5.6 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 *DAST Runtime*. As in figure 5.5, figure 5.6 shows the harmonic mean of times for the synthetic benchmark (figure 5.6a) and the Matmul (figure 5.6a) with the same previous granularities.



(a) Synthetic Benchmark



(b) Matmul

Figure 5.6: Acquisition cost of dependences graph lock

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. On the other hand, the elapsed time for the Nanos++ runtime increases significantly with the number of workers, due to the major number of collisions between the threads. In the *DAST Runtime* results, the elapsed time is equal to the forced version until the number of workers is large enough and makes the push policies reject some messages. For that reason, some stalls between the workers and the manager appear increasing the elapsed time.

The dependence graph lock is acquired several times in each application execution and the differences shown in the figure 5.6 may become crucial. Moreover, the difference in time between the both runtime approaches is larger in the figure 5.5 than in figure 5.6 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*.

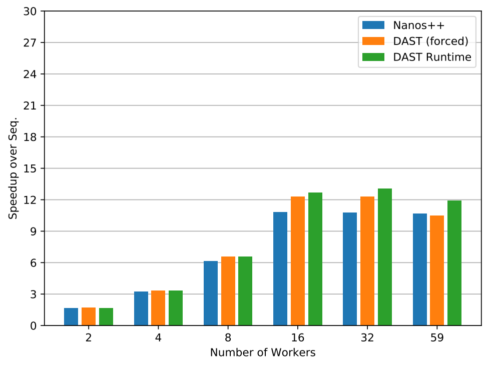
5.4 | Performance Results

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 all figures, the y-axis represents speedup values and the x-axis shows the number of workers for each execution including the master worker thread but not the extra DAST thread. This comparison is intended to see if the manager can speedup the runtime with the same amount of resources to execute tasks, it is like the manager is an external element. Moreover, the results show that running with a larger number of workers does not mean increasing the speedup. Therefore, the comparison is still fair although *DAST Runtime* uses one more thread.

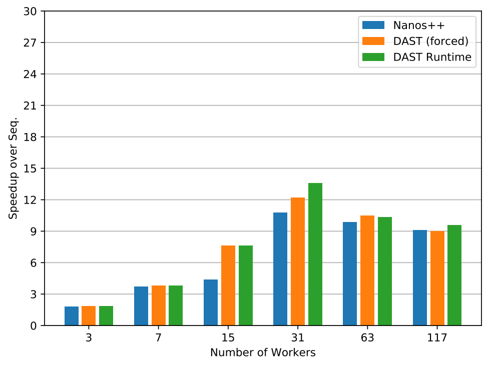
Due to the core reservations explained in section 5.1, experiments can be run up to 58 cores and 1, 2, 3 or 4 worker threads per core. Considering it and depending on the number of threads per core, x-axis values go from 2 (1 master worker + 1 core with one worker) to 59 (1 master worker + 58 cores with 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.

5.4.1 | Synthetic benchmark

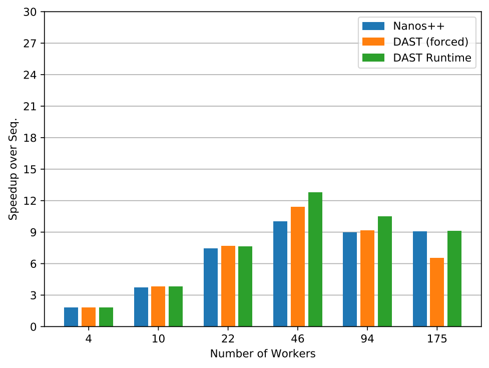
Figure 5.7 and figure 5.8 show a comparison of the speedups obtained for the synthetic benchmark with fine grain and coarse grain granularities, respectively, and using the Nanos++, *DAST (forced)* and DAST runtimes.



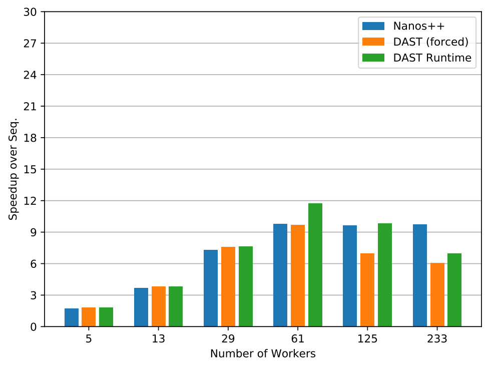
(a) 1 thread per Core



(b) 2 Threads per Core



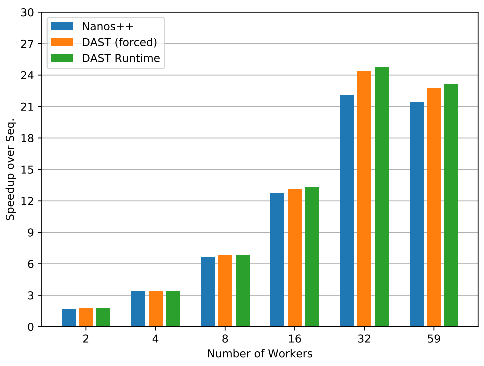
(c) 3 threads per Core



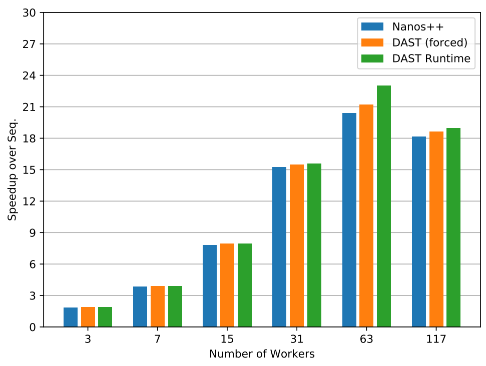
(d) 4 threads per Core

Figure 5.7: Speedup running Synthetic Benchmark (fine grain tasks)

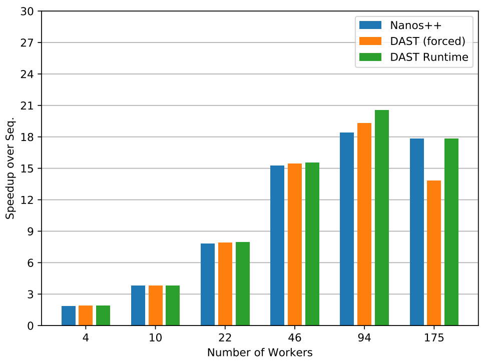
On one hand, the fine grain results in figure 5.7 show that using more than 16 cores (and not workers, since using 2, 3, or 4 threads per core does not guarantee the parallel execution of the



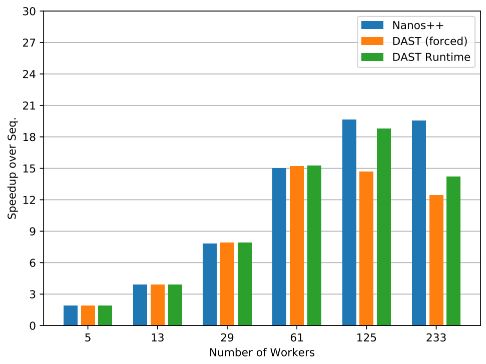
(a) 1 thread per Core



(b) 2 Threads per Core



(c) 3 threads per Core



(d) 4 threads per Core

Figure 5.8: Speedup running Synthetic Benchmark (coarse grain tasks)

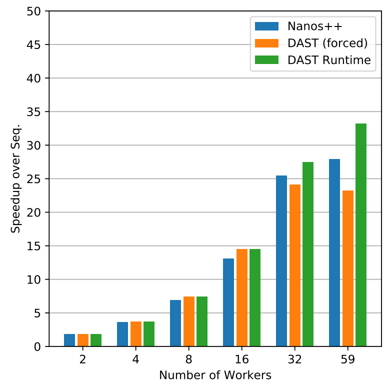
workers) does not improve the speedup. Indeed, it reduces the overall performance. The reason is that fine granularity makes runtime overheads critical, limiting the speedup. On the other hand, moving to coarse grain tasks (see figure 5.8) allows improving the speedup when increasing the number of cores 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 5.7 and 5.8 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.

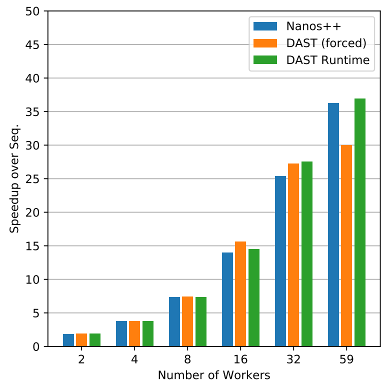
Independently of the runtime version, there is a common trend in figure 5.7 and figure 5.8 which is that the best speedups are obtained using only one thread per core. The increment in the number of threads per core is not reflected in a performance increase. Instead, the more threads per core used, the less speedup is obtained. The same standstill has been found in most of the application executions for this architecture because the available resources in each core are shared between threads cutting 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 behavior are shown only with one thread per core. On the other hand, the figure 5.7 and figure 5.8 show that *DAST runtime* overcomes *DAST (forced)* for almost all the cases, so the real benchmarks comparison is focused on *DAST Runtime* and the baseline runtime implementation (Nanos++ runtime).

5.4.2 | Real benchmarks

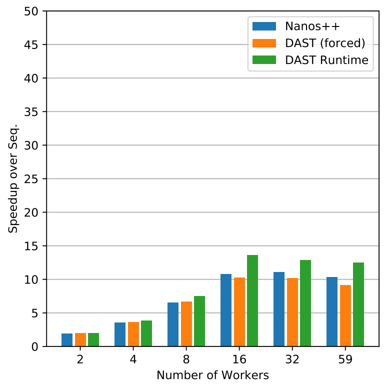
Figure 5.9 shows the speedups in three real benchmarks executions of OmpSs applications running over Nanos++ and *DAST Runtime* with fine grained (figure 5.9a, figure 5.9b and figure 5.9c) and coarse grained (figure 5.9d, figure 5.9e and figure 5.9f) tasks.



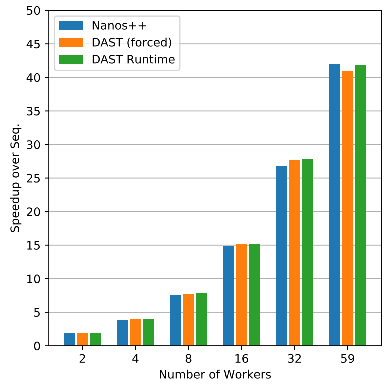
(a) Cholesky (FG)



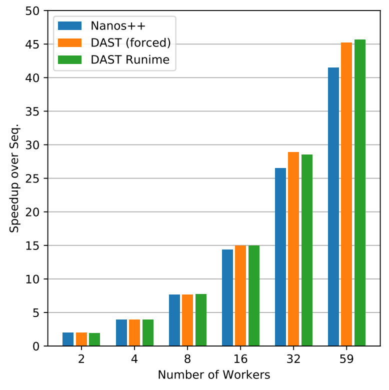
(b) Matmul (FG)



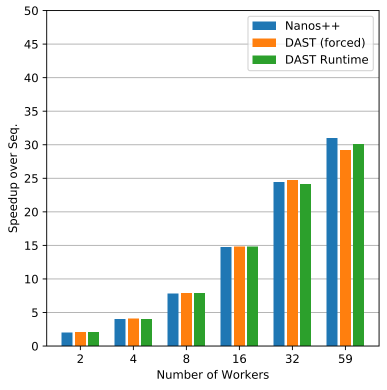
(c) Sparse LU (FG)



(d) Cholesky (CG)



(e) Matmul (CG)



(f) Sparse LU (CG)

Figure 5.9: Speedup running real benchmarks with fine grain (FG) and coarse grain (CG) 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 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 Sparse LU. That shows how the gains increase with the complexity of the dependence graph. Indeed, figure 5.9 shows that *DAST Runtime* overcomes *DAST (forced)* for almost all the cases. *DAST Runtime* balances the task workload and the task dependency contention as expected.

Finally, with the only purpose of showing that *DAST Runtime* is able to achieve the same performance or better for coarse grain applications, figure 5.9 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, the *DAST Runtime* helps to achieve the same or better performance than Nanos++ runtime for any task granularity.

5.5 | Limitations

Although the mechanism to avoid some requests increases the performance over the forced runtime, the performance may still be degraded due to the runtime manager thread underutilization. As one core of the processor is used to execute the DAST thread instead of a worker thread, the computational power capacity devoted to the application is reduced. Moreover, the best speedups over the sequential are achieved with configurations where the centralized manager behaves similarly to the baseline implementation. So, the *DAST Runtime* specially behaves better when fine grain tasks are used but such tasks granularity has a worst speedup over the sequential.

To tackle the problems that the centralized runtime manager has, the design of a new distributed implementation became the best option. In such design, the need to avoid some requests to the runtime should disappear as the manager capacity can increase to process any peak of requests. Moreover, all threads can proceed as usual and execute applications tasks when there is a marginal amount of requests to the manager, thus better adapting to the load of the application.

Distributed Runtime Manager (DDAST)

Chapter 6

A new asynchronous runtime implementation is proposed based on a distributed manager (Distributed DAST, DDAST) to overcome the problems of the centralized runtime manager (DAST), which are explained in section 5.5. The idea is that any worker thread can become a DAST thread and start executing only runtime code. With this approach, all threads can cooperate to satisfy the pending requests when there are several, and all of them can execute application tasks when the number of pending requests is small.

The design and the implementation of the asynchronous runtime with the distributed manager are completely developed from scratch in a newer runtime version. However, the knowledge acquired in the previous design and implementation has been used to isolate the runtime regions that must become runtime messages and be asynchronously executed. In addition, the implementation is based on general modules that can be extended to support other runtime functionalities than the DDAST runtime manager.

The main components of the new asynchronous runtime design are explained as follows:

- Section 6.1 explains the messages (requests) that the worker threads send to the DDAST and the queues used to transmit/store them.
- Section 6.2 explains the Functionality Dispatcher module introduced in the runtime to mediate between different components and used by the DDAST.
- Section 6.3 explains the module that implements the DDAST and is registered in the Functionality Dispatcher.

6.1 | Messages and Queues

In the distributed implementation, the messages sent by the worker threads to the runtime manager can be of two types: the *Submit Task Message* and the *Done Task Message*. The first one, the *Submit Task Message* is sent when a worker thread wants to submit a new task into the runtime structures to compute its predecessor tasks. The second one, the *Done Task Message* is sent when a worker thread finishes the execution of a task and wants to notify the successor tasks, scheduling them if they become ready.

In contrast with the centralized manager implementation, the Task Deletion request is avoided now because a new step in the task life cycle is added. This new step is a "Zombie State" that is used to coordinate the runtime manager and the worker thread. It works as follows: The first one, either worker or manager that reaches an execution point where it will not reference the task anymore marks the task as a zombie. Thus, the next worker or manager, which reaches the same execution point, will also try to mark the task as a zombie but, as it already is a zombie, instead will delete the task. So, this optimization synchronizes the worker and the manager reducing the amount of requests compared to *DAST Runtime* implementation. In fact, the addition of this new state has a similar effect to the push policy implemented for the *Delete Task* request with the additional benefit that the new state avoids the creation of a message to the manager.

Regarding the messages used in this distributed implementation, they are stored in a queue meanwhile the runtime manager does not satisfy them. As in the baseline implementation, each worker thread has its queue where only itself can insert messages, and only the DDAST can pop messages. It is important to stress that the queue respects the insertion order for the *Submit Task Message*s. The order must be satisfied to create the right task dependence graph. The practical implication is that only one DDAST thread can pop and process *Submit Task Messages* for each queue at the same time, otherwise, a newer message could enter in the task dependence graph before an older one and the task dependences will be wrongly computed. In contrast, the *Done Task Message*s can be processed by any DDAST thread concurrently without any restriction. Therefore, two independent queues are created for each worker thread, one for each message type.

The *Submit Task Message* queue requires DDAST threads to acquire the queue before getting messages from it. If the queue is already acquired by one thread, the acquisition will fail, and the thread cannot start satisfying the *Submit Task Message*s in the queue. When a DDAST thread ends processing the messages, it releases the queue to allow other threads to acquire the queue and start satisfying messages.

The queue with the Done messages allows any DDAST thread to get a message from it. In fact, to further increase the parallelism, several threads can try to get messages from the queue concurrently because the implementation supports it, the pop call will fail only if the queue becomes empty. Finally, to ensure correctness, DDAST threads check if they had successfully obtained a message from the queue or if the operation failed because the queue is empty and then process the message.

6.2 | Functionality Dispatcher

The Functionality Dispatcher is a new module introduced in the runtime core by this project that mediates between different runtime parts. This module easily allows both using the idle resources to execute any runtime operation and implementing some runtime functionalities without having dedicated computational resources to them.

Any runtime module can register a callback function in the Functionality Dispatcher during the runtime initialization or the application execution. Those callbacks are listed into the new module, which is also notified by the worker threads when they become idle. In current Nanos++ implementation, the worker threads make a busy waiting loop until they obtain tasks to execute. Therefore, the Functionality Dispatcher tries to take advantage of those idle resources and uses them to execute the different registered callbacks.

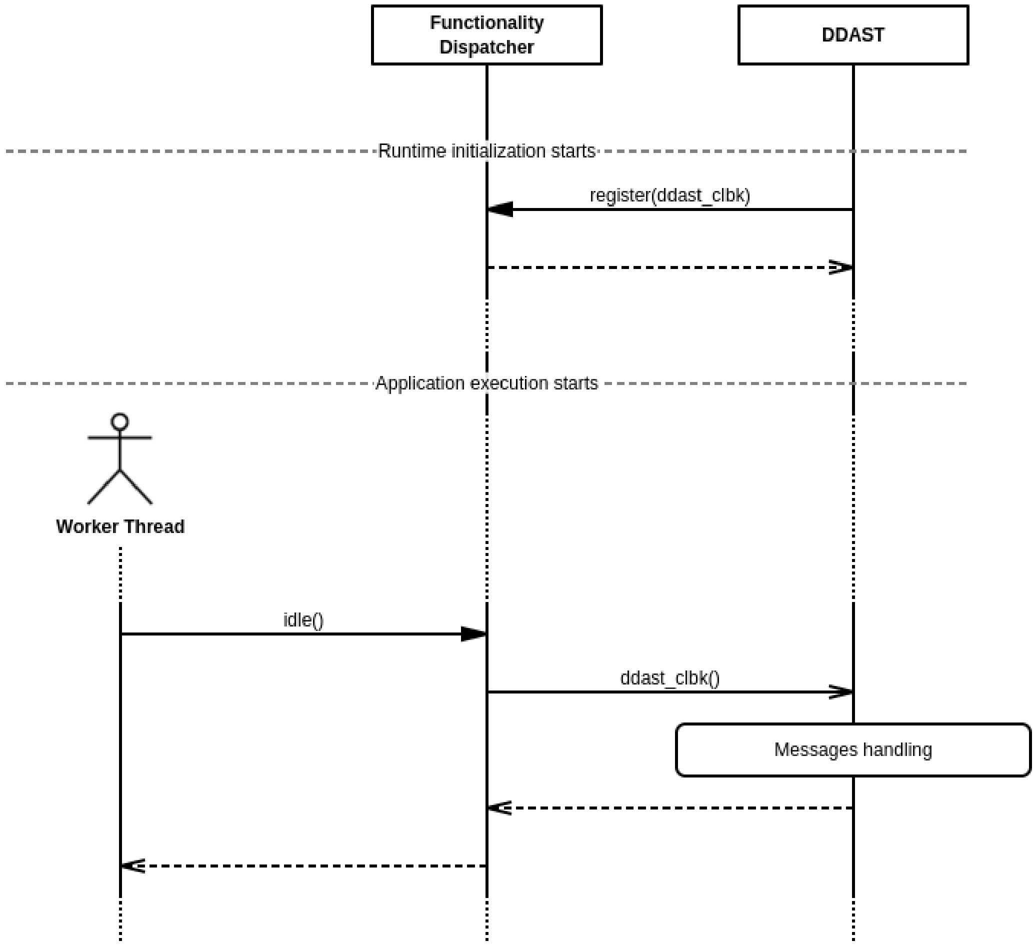


Figure 6.1: Functionality Dispatcher Sequence Diagram Example

Figure 6.1 shows the sequence diagram for the implemented Functionality Dispatcher with the DDAST runtime manager. During the runtime initialization, the DDAST module registers a callback function into the Functionality Dispatcher. Therefore, during the application execution, a worker thread that becomes idle notifies the Functionality Dispatcher, which instructs the worker to execute the DDAST callback function that starts handling the pending messages.

6.3 | DDAST Callback

The distributed runtime manager is implemented in a callback function registered in the Functionality Dispatcher. Therefore, the callback is executed when a SMP worker thread becomes idle and the Functionality Dispatcher calls the registered function. That a worker thread becomes idle usually means that the pending messages in the queues must be processed to submit more tasks into the task graph or trigger the scheduling of some new ready tasks.

The behavior of the callback is parametrized by different constants defined at the beginning of the application execution. The performance impact and the default values for these variables are analyzed in section 7.3. Here follows a brief list and explanation of these variables::

- `MAX_DDAST_THREADS`.
  Maximum number of threads allowed to execute the callback concurrently.
- `MAX_SPINS`.
  Number of times that the thread will try to get messages without success before leaving the callback.
- `MAX_OPS_THREAD`.
  Maximum number of messages satisfied from the same queue before changing to another worker thread queue.
- `MIN_READY_TASKS`.
  Minimum number of ready tasks available to exit the callback.

Listing 6.1 shows the pseudo-code of the callback function. First, the number of threads inside the DDAST is checked and the function returns if the maximum number is reached (listing 6.1 line 1). After that, the idle thread tries to retrieve messages and satisfy them. This is done until the minimum number of ready tasks is reached or the thread iterates `MAX_SPINS` times without finding any message (listing 6.1 line 24). The way to retrieve messages is to iterate through all worker queues and try to satisfy up to `MAX_OPS_THREAD` requests combining requests from the two possible queues. Note that not all worker thread queues are iterated if the number of ready tasks becomes higher than `MIN_READY_TASKS` (listing 6.1 line 7).

```
if (numThreads >= MAX_DDAST_THREADS) return
++numThreads

do {
  totalCnt = 0
  forEach(worker: workers) {
    if (readyTasks >= MIN_READY_TASKS) break

    cnt = 0
    if (worker.queueSubmit.acquire()) {
      while (cnt < MAX_OPS_THREAD && (msg = worker.queueSubmit.pop())) {
        msg.satisfy()
        ++cnt
      }
      worker.queueSubmit.release()
    }
    while (cnt < MAX_OPS_THREAD && (msg = worker.queueOthers.pop())) {
      msg.satisfy()
      ++cnt
    }
    totalCnt += cnt
  }
  spins = totalCnt == 0 ? (spins - 1) : MAX_SPINS
} while (spins != 0 && readyTasks < MIN_READY_TASKS)
--numThreads
```

Listing 6.1: DDAST callback pseudo-code

DDAST Evaluation

Chapter 7

The evaluation of the asynchronous runtime implementation is explained in this chapter. The objective of the evaluation is to compare the behavior and performance of the new runtime structure with the original one. To this end, different benchmarks, well known in the High Performance Computing world, and different machines that have different architectures are used. The structure of the following sections is:

- Section 7.1 explains the software and hardware environments used to run the different benchmarks.
- Section 7.2 explains the benchmarks used to test the behavior and performance of the new design.
- Section 7.3 explains the results of the exploration to tune the DDAST callback parameters.
- Section 7.4 explains the scalability and performance results for each benchmark comparing the different runtime versions.

7.1 | Environment

The tests have been run in different processor architectures explained in the following sections. Each section includes the specific software environment used for the architecture. However, a common criteria for all the architectures has been followed to enhance the evaluation quality, avoid external inferences and facilitate the reproducibility of the results.

1. The machine's nodes used during all executions are exclusively reserved for the tests, despite the number of used cores is smaller than the available.
2. The applications are compiled with the code optimizations enabled. At least, the `-O3` flag is used. If some architecture flags have been used, they are listed in the corresponding machine section.

3. The time measurements are repeated at least 5 times and the best execution time is kept for each configuration to do the comparisons. Also, the executions have been repeated if the best timing appears only once.

4. The scheduling policy used in the OmpSs executions is the Distributed Breadth First (DBF). The DBF policy uses a queue of ready tasks for each thread with an stealing mechanism [19]. The change is intended to avoid having a huge contention in the global ready tasks queue.

Intel Xeon Phi (KNL)

The Intel Xeon Phi is a series of processors manufactured by Intel and characterized by its high parallelism and vectorization capacity. The Knights Landing (KNL) is the second generation of such processors. Moreover, they are designed to provide a high energy efficiency and binary compatibility with the legacy x86 compiled applications [24].

The KNL model used in our evaluation is the *Intel® Xeon® Phi(TM) CPU 7230* which has 64 cores working at 1.30GHz. In addition, the machine has 96GB of main memory distributed in 6 DIMMs. The 16GB of high bandwidth memory integrated in the package are used as a cache. Therefore, the total amount of available memory is 96GB and is the OS which manages the 16GB memory as another cache level. Finally, the machine is a self-boot socket version configured in Quadrant mode [24].

The executions use up to 64 worker threads, which is the number of cores. Although the processor has hyper-threading and each core can run up to 4 threads per core, the used machine has the hyper-threading disabled. In the executions with less than 64 cores, the use of the processor's resources is maximized. For instance, the cores 0 and 2 are used in a 2-cores execution because cores 0 and 1 share part of the cache.

The version of the Intel® Math Kernel Library (Intel(R) MKL) used by the applications is 2017.0.2. The compiler used to compile the applications and the runtimes is the GNU C Compiler Collection (GCC) version 6.3.0. Moreover, the following set of compiler flags has been used to optimize the binaries for the KNL architecture:

```
-march=knl -O3 -mavx512f -mavx512pf -mavx512er -mavx512cd -mfma -malign-data=cacheline
```

ThunderX (ARM)

The ThunderX is a family of processors developed by Cavium based on the 64-bit ARMv8 architecture. They directly integrate the support for several interfaces, like 10x10Gbps Ethernet ports and SATA connections, in the SoC. Therefore, the energy efficiency delivered by the processor is very promising without limiting the computational capacity [25].

The ThunderX model used in our evaluation is the 48-core variant with 1 thread per core. The machine has 64GB of main memory. Although the ThunderX processor is intended for the server market, the high number of available cores with a limited power consumption makes it attractive to the HPC market. The compiler used to natively compile the applications and the runtimes is the GNU C Compiler Collection (GCC) version 5.3.0. The version of the ARM Performance Libraries (ARM PL) used by the applications is 2.0.0.

7.2 | Benchmarks

The used benchmarks are explained in the following sections. For each one, its execution arguments are explained and provided with the number of created tasks in each configuration and any other remarks that may be valuable for reproducibility. In all of them, some timing instructions are added after the sequential initialization and after the final global taskwait. The wall-clock time elapsed between these two points is defined as execution time in the rest of the section.

For each benchmark, two different sets of execution parameters are used to create two tasks granularities: coarse grain tasks and fine grain tasks. In addition, the benchmark execution parameters are selected considering the following points:

1. Have a big enough problem size to gather significant results. The fastest executions take around 1.5 seconds.
2. Have a reasonable performance in the coarse grain Nanos++ executions, that means, have large enough tasks to hide the runtime overheads.
3. The fine grain executions solve the same problem using tasks half the size of the tasks of the coarse grain executions.

Matrix Multiply

The Matrix Multiply (Matmul) benchmark [23] follows the same implementation described in section 5.2, and only the argument values have changed. The used values for the `MATRIX_SIZE` and `BLOCK_SIZE` arguments in each machine are summarized in the table 7.1. In all executions, the number of repetitions is set to 1.

Machine	Matrix Size	Coarse Grain		Fine Grain	
		Block Size	Num. Tasks	Block Size	Num. tasks
KNL	8192	512	4096	256	32768
ThunderX	4096	128	32768	64	262144

Table 7.1: Matmul execution arguments

N-Body

The N-Body benchmark [26] simulates particles' movements under some physic forces, such as gravity. The application takes three arguments: the number of particles (`NUM_PARTICLES`), the number of time steps (`NUM_TIMESTEPS`) to be simulated and the number of particles per block (`BLOCK_SIZE`). Therefore, the particles are spread into blocks with `BLOCK_SIZE` particles, which are used as task input/output. The values for the arguments used in each machine are summarized in the table 7.2 with the number of tasks created in each configuration.

Machine	Num. Particles	Num. Timesteps	Coarse Grain Block Size	Coarse Grain Num. Tasks	Fine Grain Block Size	Fine Grain Num. tasks
KNL	16384	16	128	262176	64	1048608
ThunderX	16384	16	128	262176	64	1048608

Table 7.2: N-Body execution arguments

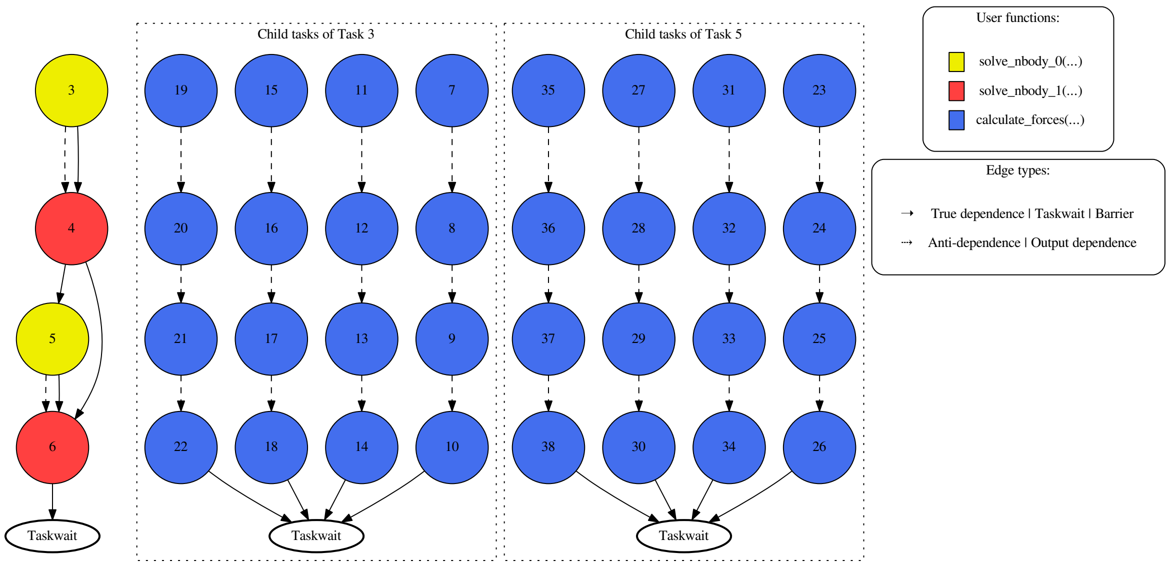


Figure 7.1: N-Body task graph showing the data dependences between tasks

Figure 7.1 shows the data dependences between the tasks created in a small execution of the benchmark (`NUM_PARTICLES` is 4096, `NUM_TIMESTEPS` is 2 and `BLOCK_SIZE` is 512). The nodes represent the task instances (each color represents one task type) and the edges between them the data dependences detected. In addition, the dotted squares cluster all tasks created inside other tasks. The dependence graph is quite similar to the Matmul, with independent chains of dependent tasks. However, the nested tasks make more critical some of the requests to the DDAST manager because they may block the application parallelism until they are processed.

Sparse LU

The Sparse LU benchmark [23] uses the same implementation described in section 5.2, and only the argument values have changed. The used values for the `MATRIX_SIZE` and `BLOCK_SIZE` arguments in each machine are summarized in the table 7.3.

Machine	Matrix Size	Coarse Grain		Fine Grain	
		Block Size	Num. Tasks	Block Size	Num. tasks
KNL	8192	128	11472	64	89504
ThunderX	8192	128	11472	64	89504

Table 7.3: Sparse LU execution arguments

7.3 | DDAST Tunning

The initial executions with the new runtime structure were intended to find good default values for the callback parameters explained in section 6.3. To this end, some initial values are predefined, based on a reasonable approximation, and the same execution is repeated changing only one parameter value. The executions for each parameter are done with all benchmarks and architectures, doubling the parameter's value from 1 up to 128. Also, different amounts of threads are considered: doubling from 2 up to the machine limit. However, the results are only shown for the two larger amounts of threads because they are the most interesting as with the maximum number of threads is when modifications to the parameters most influence the execution time.

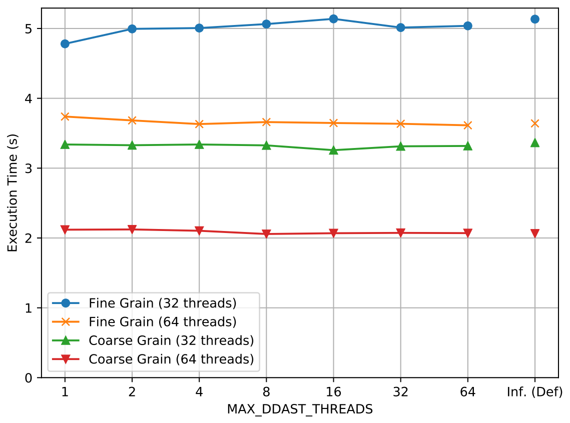
Parameter	Initial Value	Tunned Value
`MAX_DDAST_THREADS`	∞	∞
`MAX_SPINS`	20	4
`MAX_OPS_THREAD`	6	8
`MIN_READY_TASKS`	4	4

Table 7.4: DDAST parameters values

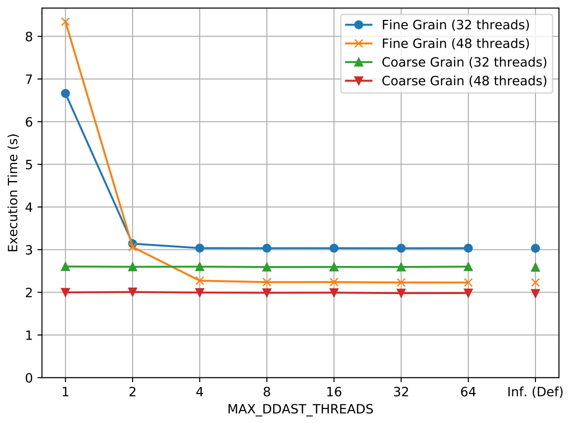
The predefined values for each parameter before (*Initial value*) and after (*Tunned Value*) the tuning are shown in table 7.4. Using the initial values as default, the following sections present the results obtained when one of the parameters is modified.

7.3.1 | Maximum number of DDAST threads

The execution time for each benchmark when changing the value of `MAX_DDAST_THREADS` parameter is shown in the plots of figure 7.2, figure 7.3 and figure 7.4. Figure 7.2a, figure 7.3a and figure 7.4a show the execution time in the KNL machine for Matmul, N-Body and Sparse LU applications respectively. Figure 7.2b, figure 7.3b and figure 7.4b show the time consumed by the same applications (Matmul, N-Body and Sparse LU respectively) when executed on the ThunderX machine.

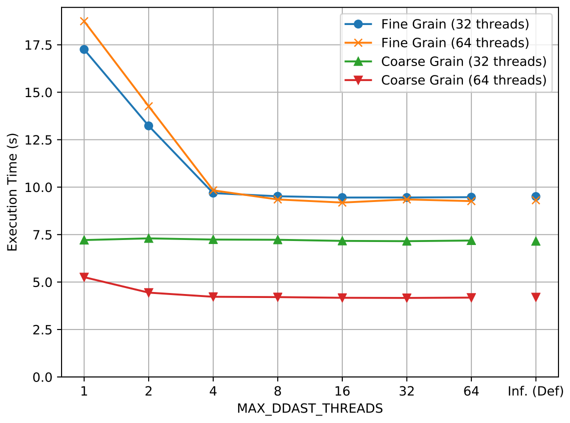


(a) KNL

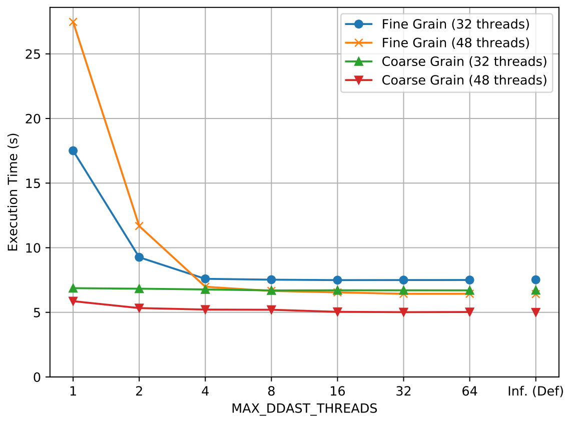


(b) ThunderX

Figure 7.2: Matmul execution time changing the `MAX_DDAST_THREADS`



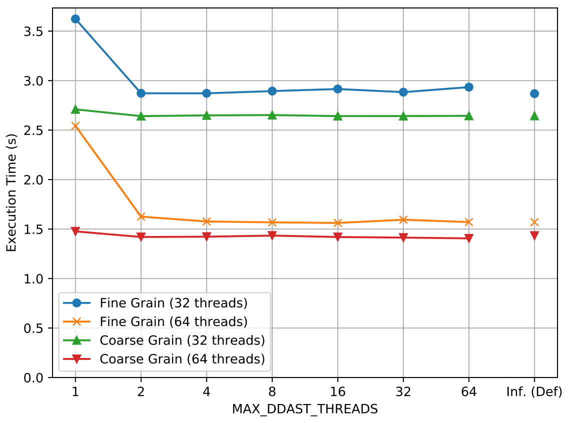
(a) KNL



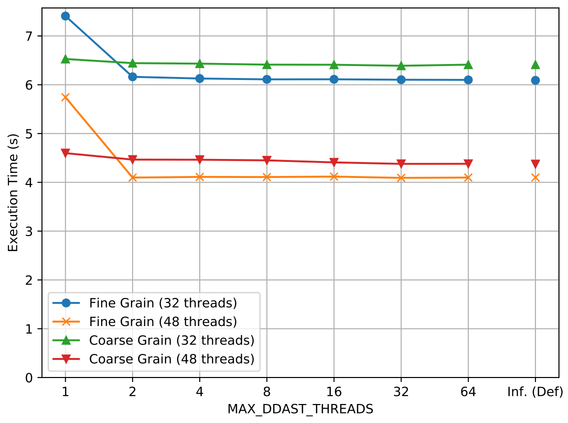
(b) ThunderX

Figure 7.3: N-Body execution time changing the `MAX_DDAST_THREADS`

The results show that the smaller execution times are obtained when the value is at least 4. Moreover, increasing the number of allowed threads in the DDAST callback does not increment the execution time, thereby limiting the number of threads that can execute runtime functions makes no sense. As can be seen in the time results with less than four allowed threads, limiting the number of threads in the DDAST callback can lead to increase the application execution time for nearly all the benchmarked applications. As a result, the predefined value, which is infinite, is assumed as the correct default value of the `MAX_DDAST_THREADS` parameter.



(a) KNL

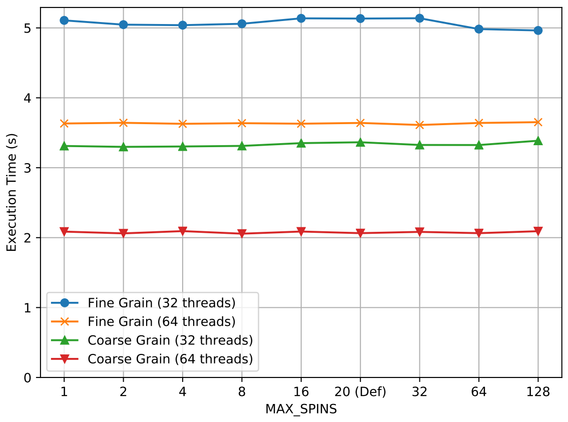


(b) ThunderX

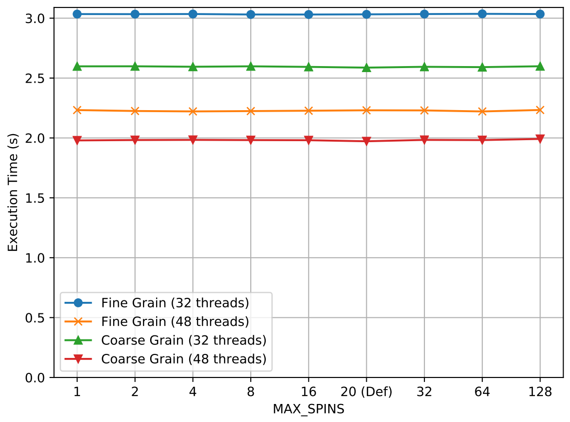
Figure 7.4: Sparse LU execution time changing the `MAX_DDAST_THREADS`

7.3.2 | Maximum number of spins

The execution time for each benchmark when changing the value of `MAX_SPINS` parameter is shown in the plots of figure 7.5, figure 7.6 and figure 7.7. Figure 7.5a, figure 7.6a and figure 7.7a show the execution time in the KNL machine for Matmul, N-Body and Sparse LU applications respectively. Figure 7.5b, figure 7.6b and figure 7.7b show the execution times obtained in the ThunderX machine for Matmul, N-Body and Sparse LU applications respectively.



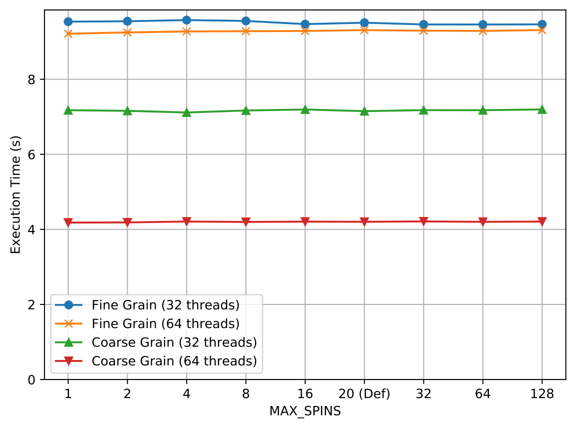
(a) KNL



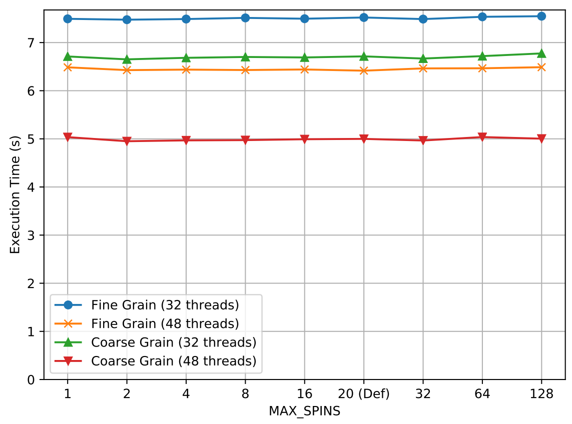
(b) ThunderX

Figure 7.5: Matmul execution time changing the `MAX_SPINS`

The results show that the execution time is not affected by the value of `MAX_SPINS`. Regardless the values, the execution time is almost constant for each benchmark and task granularity. However, the parameter can increase the runtime overheads because a large value will retain the worker threads in the DDAST callback until any other break condition is satisfied. Thus, the threads are prevented from doing other useful runtime work meanwhile they are idle. But, if no other callback functions are registered, the best approach may be retain the threads in

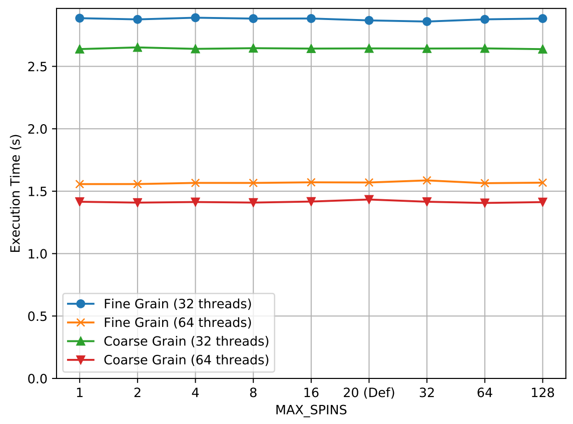


(a) KNL

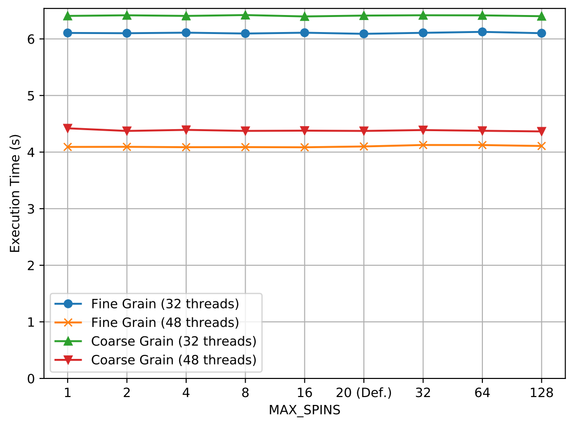


(b) ThunderX

Figure 7.6: N-Body execution time changing the `MAX_SPINS`



(a) KNL



(b) ThunderX

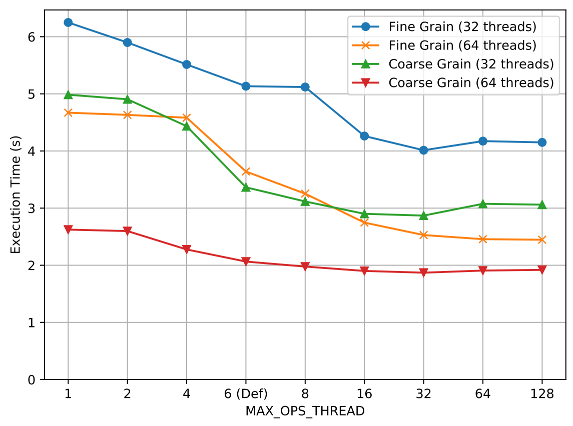
Figure 7.7: Sparse LU execution time changing the `MAX_SPINS`

the DDAST callback until there are some ready tasks. Considering a future scenario where the Functionality Dispatcher is used to manage more runtime functionalities, the predefined default value for the `MAX_SPINS` parameter is set to *4*.

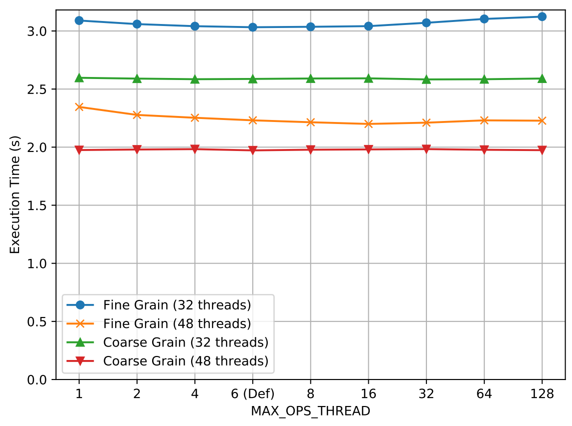
7.3.3 | Maximum operations per thread

The execution time for each benchmark when changing the value of `MAX_OPS_THREAD` parameter is shown in the plots of figure 7.8, figure 7.9 and figure 7.10. Figure 7.8a, figure 7.9a and figure 7.10a show the execution time measured in the KNL machine for Matmul, N-Body and Sparse LU applications respectively. Figure 7.8b, figure 7.9b and figure 7.10b show the execution times measured in the ThunderX machine for Matmul, N-Body and Sparse LU applications respectively.

The results show that this parameter is application and machine dependent. One one

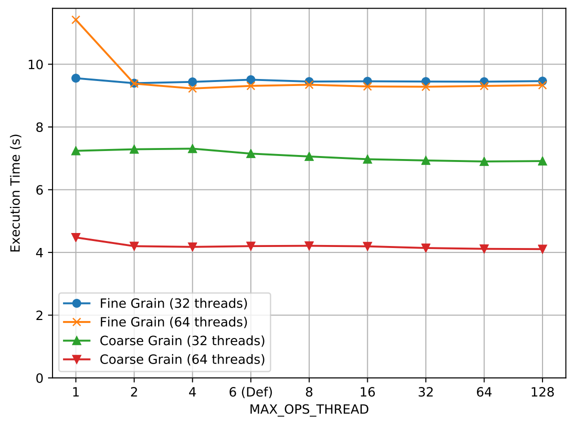


(a) KNL

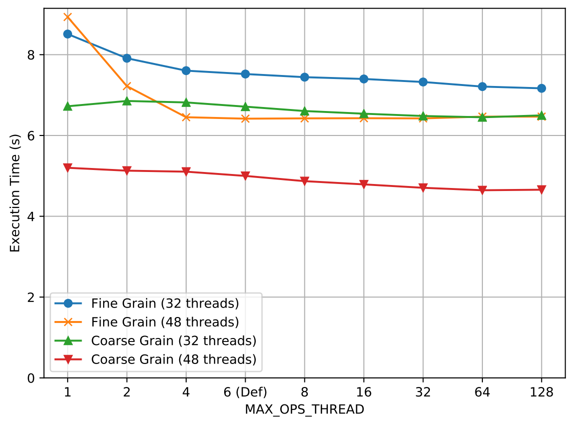


(b) ThunderX

Figure 7.8: Matmul execution time changing the `MAX_OPS_THREAD`



(a) KNL



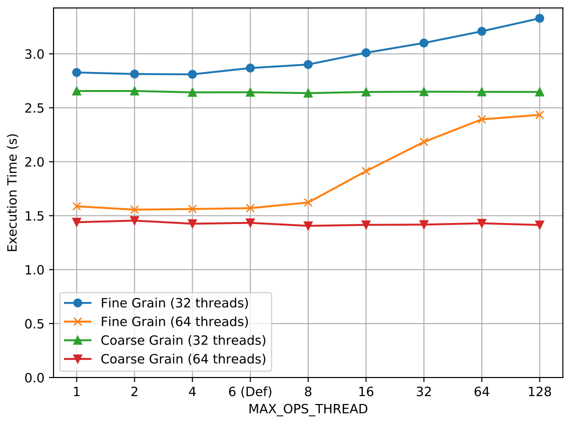
(b) ThunderX

Figure 7.9: N-Body execution time changing the `MAX_OPS_THREAD`

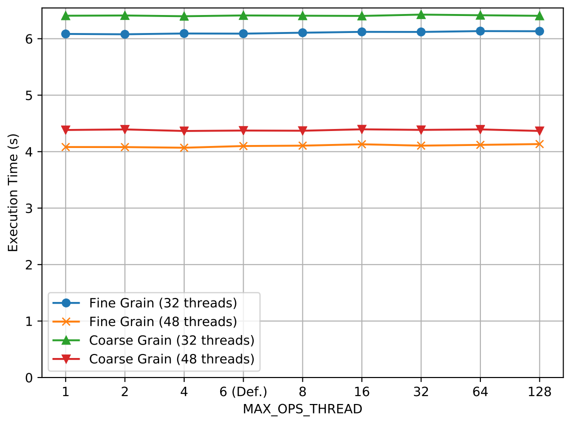
hand, the ThunderX results show times much more stables than the KNL results for different `MAX_OPS_THREAD` values. On the other hand, the global execution time decreases in figure 7.8a with larger values and it increases in figure 7.10a with the same values.

The opposed behaviors shown in KNL are due to the different dependence patterns that each application has. The execution time may increase with a large value of `MAX_OPS_THREAD` if the ready tasks depend on a *Done Request*. In this case, a critical request processing may be delayed because the DDAST threads will process before a large number of requests from different worker threads. However, the execution time may decrease with large values of `MAX_OPS_THREAD` if most of the requests schedule a new ready task. In this case, the DDAST threads may benefit from the data locality when processing several requests from the same queue.

Considering the results, a reasonable value for the default value of the `MAX_OPS_THREAD` parameter is 8. In all benchmarks, machines and task granularities, a value of 8 obtains the same



(a) KNL



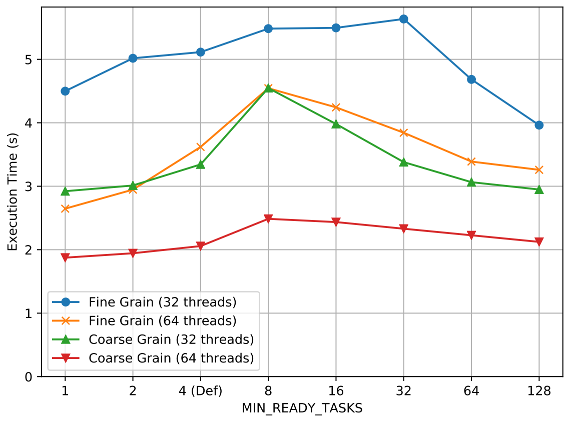
(b) ThunderX

Figure 7.10: Sparse LU execution time changing the `MAX_OPS_THREAD`

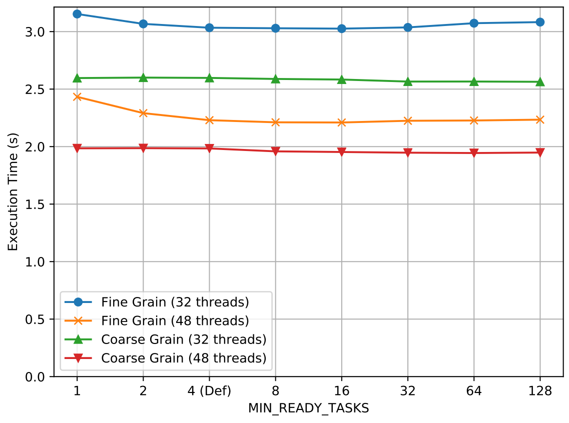
or a smaller execution time than the predefined value (6). Even there are better parameter values for some specific cases, the new default value is the best considering all configurations.

7.3.4 | Minimum number of ready tasks

The execution time for each benchmark when changing the value of `MIN_READY_TASKS` parameter is shown in the plots of figure 7.11, figure 7.12 and figure 7.13. Figure 7.11a, figure 7.12a and figure 7.13a show the execution time obtained in the KNL machine for Matmul, N-Body and Sparse LU applications respectively. Figure 7.11b, figure 7.12b and figure 7.13b show the times measured in the ThunderX machine for Matmul, N-Body and Sparse LU applications respectively.



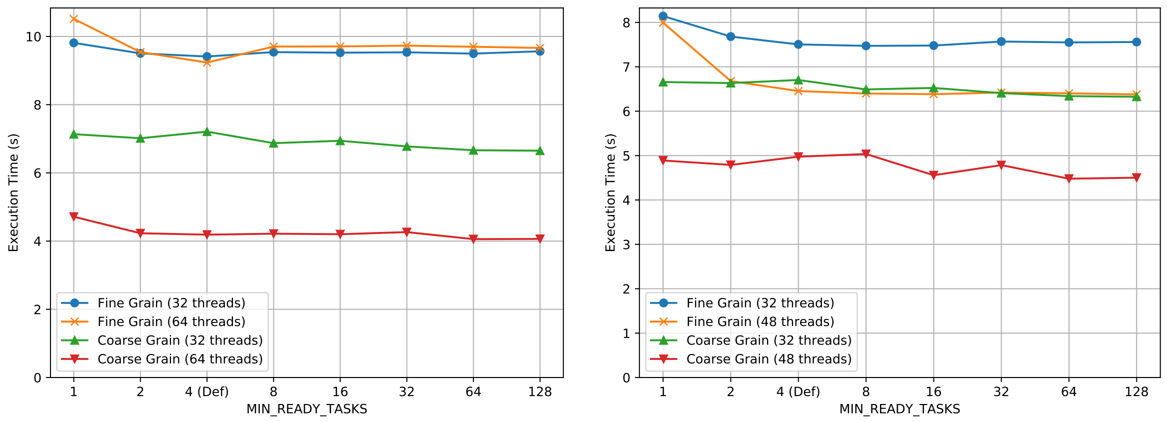
(a) KNL



(b) ThunderX

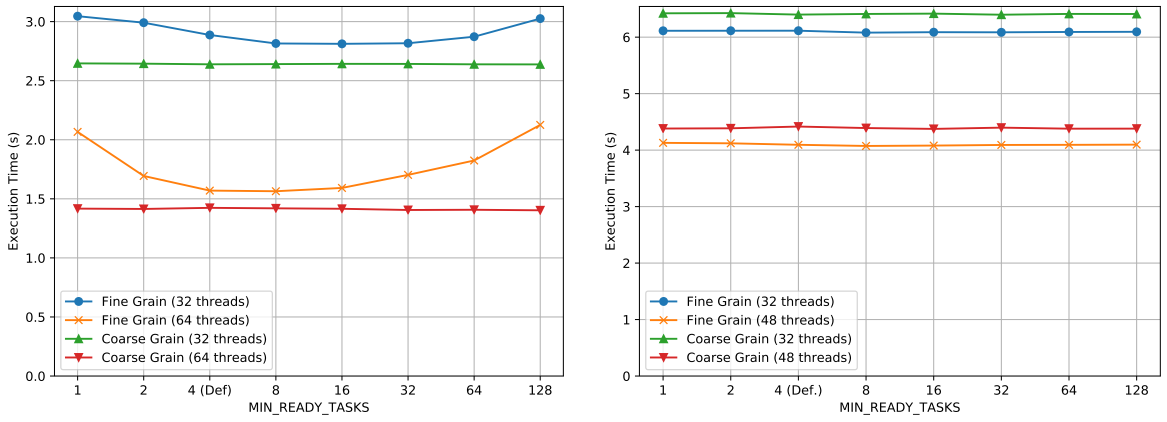
Figure 7.11: Matmul execution time changing the `MIN_READY_TASKS`

The results show that this parameter behaves differently in each benchmark and machine. As for the `MAX_OPS_THREAD` parameter, the different task dependence patterns result in a different



(a) KNL (b) ThunderX

Figure 7.12: N-Body execution time changing the `MIN_READY_TASKS`



(a) KNL (b) ThunderX

Figure 7.13: Sparse LU execution time changing the `MIN_READY_TASKS`

behavior depending on the benchmark. Moreover, the times are similar regardless the parameter value in the ThunderX machine, and they change in the KNL machine. The results show that the predefined default value for the `MIN_READY_TASKS` parameter, which is 4, delivers a good execution time overall configurations.

7.4 | Performance Comparison

The obtained scalability/performance results, for each benchmark, are explained in the following sections. The DDAST parameters values used in all executions are the *Tunned Values* summarized in table 7.4. The results are shown for 4 different runtime versions/configurations:

- *Nanos++*.

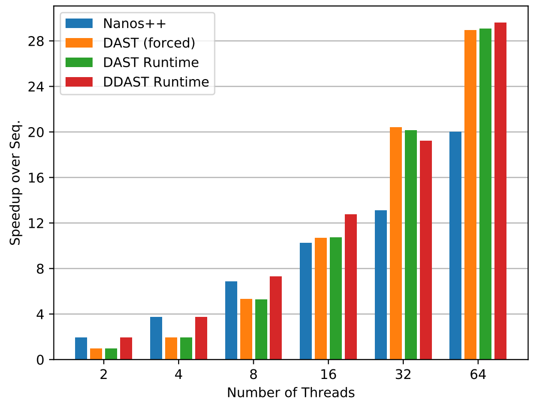
Baseline OmpSs runtime (version 0.11a).

- *DAST (forced)*.
  DAST Runtime with the request push policies disabled (mandatory submission of requests).
- *DAST Runtime*.
  Runtime with the centralized runtime manager implementation (DAST). This version is implemented on top of *Nanos++* runtime (version 0.7.2).
- *DDAST Runtime*.
  Runtime with the distributed runtime manager implementation (DDAST). This version is implemented on top of *Nanos++* runtime (version 0.11a).

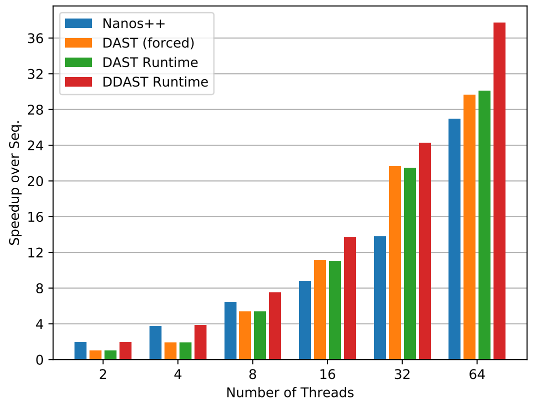
All execution traces shown in the following sections contain the initial and end time in the x-axes (time) labels. Despite the initial and end timestamps may not match between executions with different runtimes, any pair of traces from different runtimes intended to be compared have the same time duration. That different initial/final times are due to the variable startup overheads that may change between executions, thereby the initial time is adjusted to match the point where the execution of the first task starts. Also, the end time is defined by the initial time plus the required time to show all task executions in all traces of the different runtimes.

7.4.1 | Matrix Multiply

The speedup over the sequential version for the Matmul benchmark is shown in figure 7.14 and figure 7.15. One one hand, figure 7.14a and figure 7.15a show the fine grain results for KNL and ThunderX respectively. On the other hand, figure 7.14b and figure 7.15b show the coarse grain results for KNL and ThunderX respectively. In any case, all of them show the scalability when increasing the number of threads with the same benchmark input (strong scaling).

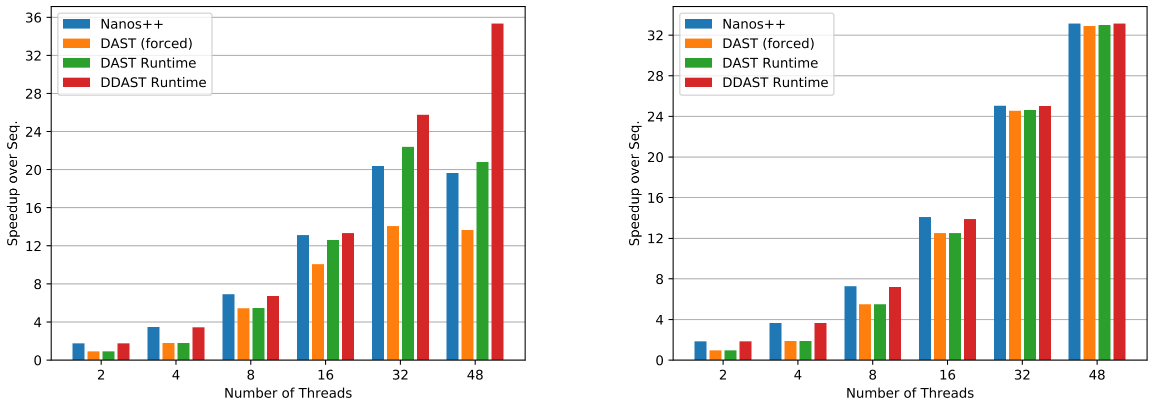


(a) Fine grain



(b) Coarse Grain

Figure 7.14: Matmul scalability in KNL



(a) Fine grain (b) Coarse Grain

Figure 7.15: Matmul scalability in ThunderX

Figure 7.14 shows a significant performance improvement (∼40% for fine grain and ∼30% for coarse grain) when using the *DDAST Runtime* in comparison to the Nanos++ baseline. Moreover, the *DDAST Runtime* outperforms the DAST variants for both benchmark configurations. The huge difference between the Nanos++ and *DDAST Runtime* is not only due to the reduction of runtime structures contention. This can be understood analyzing the execution trace of those runs. Figure 7.16 and figure 7.17 show an execution trace of 50ms duration for the fine grain Matmul benchmark in KNL. In these figures, each row presents the state of one thread where pink means running a task, red means running the main task and sky-blue means other actions. Comparing both figures, the execution of the tasks takes less time in the *DDAST Runtime* than in the Nanos++ runtime (∼9 ms vs. ∼6 ms, in average). The difference is due to better data locality that can be explained by the way each approximation accesses the data: Nanos++ runtime is accessing the runtime data structures between two task executions and therefore the runtime is "polluting" the thread caches with its data. In contrast, the DAST runtimes avoid those accesses by their asynchronous approach.

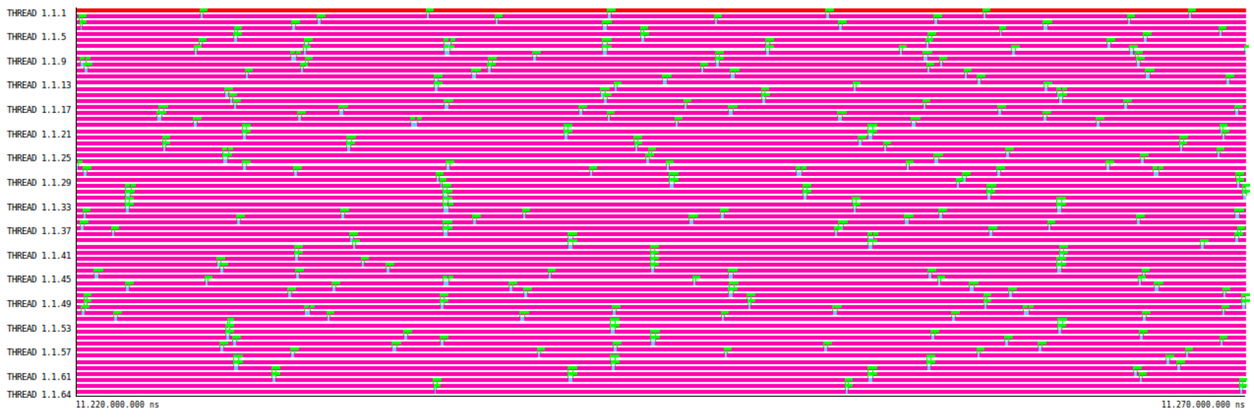


Figure 7.16: Fine grain Matmul execution trace on KNL with 64 threads and Nanos++

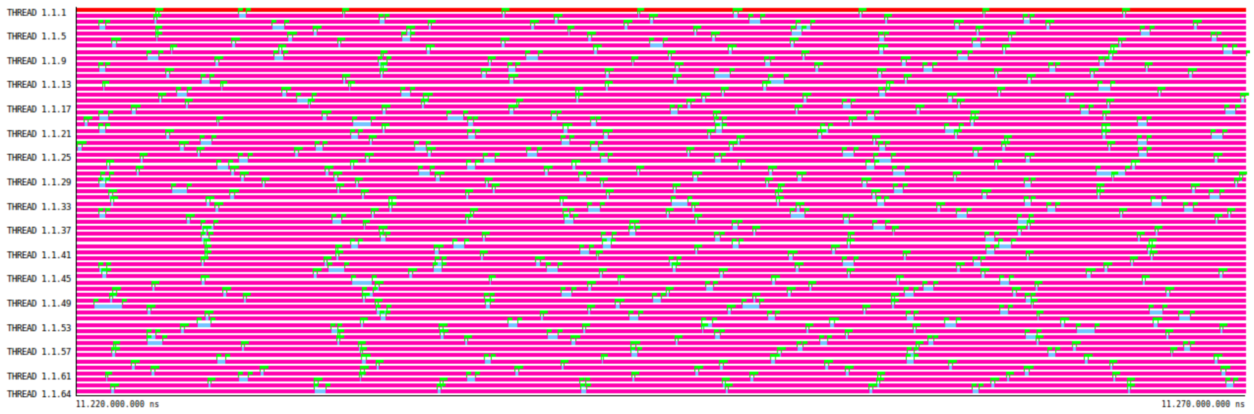
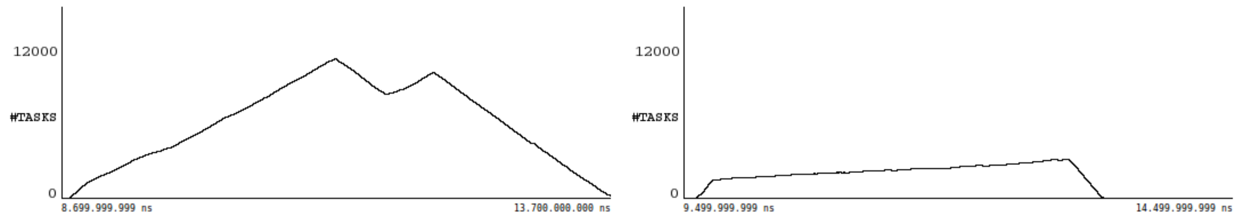


Figure 7.17: Fine grain Matmul execution trace on KNL with 64 threads and DDAST

Figure 7.15a shows even a larger performance increase between Nanos++ and *DDAST Runtime* for the fine grain executions (∼70%). Nanos++ reduces the performance when the number of used threads is increased from 32 to 48. In contrast, the DDAST is able to take benefit of the extra resources and increase the application performance. On the other side, the *DAST (forced)* runtime cannot handle all requests limiting the application parallelism and the *DAST Runtime* is only able to achieve a Nanos++ similar performance.

Nanos++ and DAST runtimes have a similar performance in figure 7.15b where the coarse grain tasks are used. The DAST versions have the worst performance in all thread amounts because the implementation uses only one thread as runtime manager. Therefore, the reduction of available computational resources may become a problem if the manager is not occupied. This behavior can be seen in the two thread results of figure 7.14 and figure 7.15 where the DAST runtime versions only have half of the Nanos++ runtime performance.

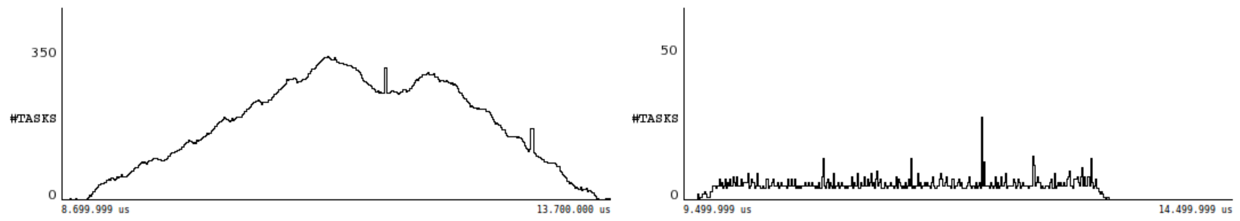


(a) Nanos++

(b) DDAST

Figure 7.18: Evolution of the number of tasks in-graph on KNL with 64 threads for fine grain Matmul

Despite the explicit differences between Nanos++ and *DDAST Runtime* in performance terms, they have other implicit (or internal) behavior differences. Figure 7.18 shows the evolution during the execution (x-axis) of the number of tasks in the dependence task graph (y-axis) and the figure 7.19 shows the number of ready tasks at each moment for the same period. All of them have the same duration for the x-axes (5 seconds) but different scales in the y-axes due to the huge value differences. Therefore, figure 7.18a and figure 7.18b use the range [0, 12000] for the number of tasks. Figure 7.19a uses the range [0, 350] and figure 7.19b [0, 50], both axes



(a) Nanos++ (b) DDAST

Figure 7.19: Evolution of the number of ready tasks on KNL with 64 threads for fine grain Matmul

are number of tasks. One one hand, Nanos++ runtime has almost a pyramid shaped evolution where a huge amount of tasks are concurrently managed in the task graph (figure 7.18a) and ready queues (figure 7.19a). In fact, the evolution is not a perfect pyramid due to a trace flush to disk as figure 7.18 and figure 7.19 are extracted from execution traces. The flush caused that the main thread temporally interrupted the task creation. On the other hand, *DDAST Runtime* has a roof shaped evolution (figure 7.18b and figure 7.19b) where only the minimum amount of tasks needed to discover some parallelism are used; the rest are kept in the manager queues. This difference may greatly influence the runtime overheads which are related to the number of elements that should be managed in the runtime structures.

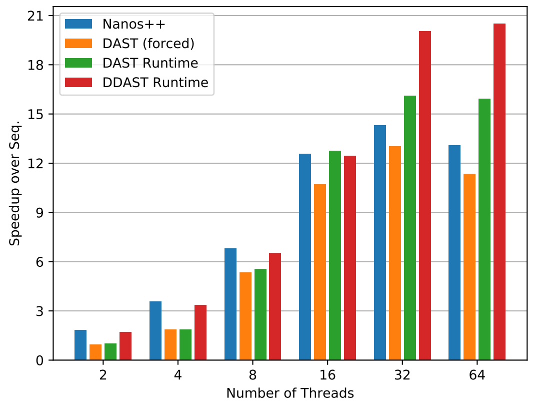
7.4.2 | N-Body

The speedup of the different runtimes over the sequential version for the N-Body benchmark is shown in figure 7.20 and figure 7.21. Figure 7.20a and figure 7.21a show the fine grain results for KNL and ThunderX respectively. Figure 7.20b and figure 7.21b show the coarse grain results for KNL and ThunderX respectively. All of them show the scalability when increasing the number of threads while keeping the size of the input (strong scaling).

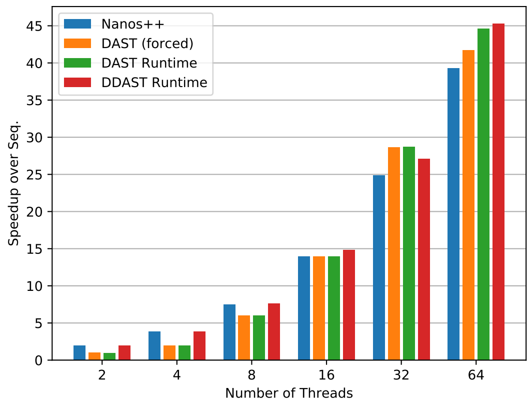
The fine grain results show a behavior similar to the Matmul results explained in section 7.4.1. In both figures (7.20a and 7.21a), Nanos++ runtime results show a performance slowdown when more than 32 worker threads are used. In contrast, the *DDAST Runtime* increases the overall performance or, at least, keeps the same one.

The coarse grain results show a different scenario with similar performances between the runtimes. Up to 16 threads, Nanos++ and DDAST behave similar and DAST versions fail due to the manager underutilization. With 32 threads, the manager thread in DAST runtimes is fully utilized, thereby the low processing latency of a dedicated manager thread provides the best performance. Finally, with all cores is use (64), the throughput of requests exceeds the *DAST (forced)* capacity but the *DAST Runtime* and *DDAST Runtime* can cope with the requests increasing the overall performance.

Figure 7.22 and figure 7.23 show the execution trace of N-Body with coarse grain tasks in ThunderX for Nanos++ and DDAST respectively. However, only two timesteps are simulated instead of the value provided in table 7.2. The traces show executions with 48 threads and the

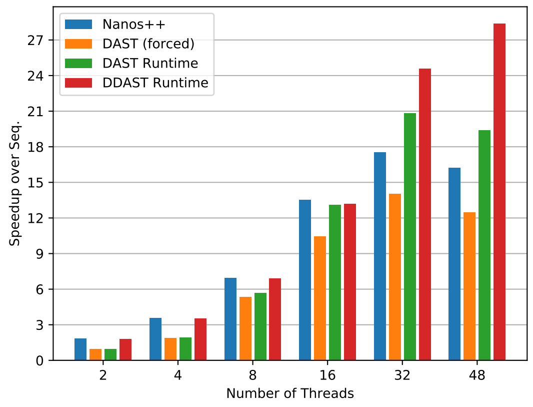


(a) Fine grain

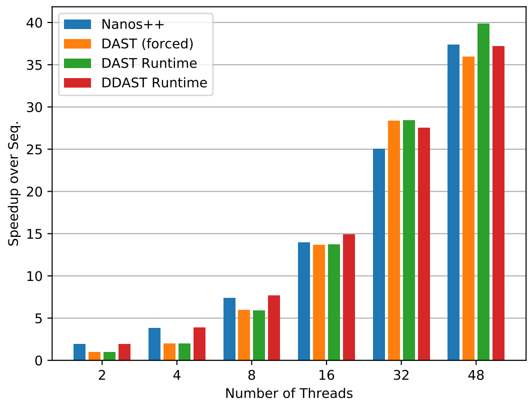


(b) Coarse Grain

Figure 7.20: N-Body scalability in KNL



(a) Fine grain



(b) Coarse Grain

Figure 7.21: N-Body scalability in ThunderX

duration (time, x-axes) is adjusted to be the same in both figures. Each line represents a worker thread and the different colors represent the thread state: sky-blue for IDLE state and other colors for the different task types execution. For instance, the brown regions represent the execution of `solve_nbody_0` top level tasks which execution create the `calculate_forces` children tasks (both shown in figure 7.1) that are executed in the pink regions.

The execution traces show that the threads execute the tasks at the same throughput that are created. During each timestep, the execution of the brown task ends almost at the same time than pink tasks, which are created by the first one. In this case, the number of ready tasks is near to zero all the time and the maximum peaks go up to 5 ready tasks as can be seen in figure 7.24, which shows the number of ready tasks (y-axes) for an small execution region of 30 miliseconds (x-axes) for both runtimes. This is a different behavior than the shown in the Matmul traces, and in this scenario the DDAST is able to reduce the timestep execution time as traces show.

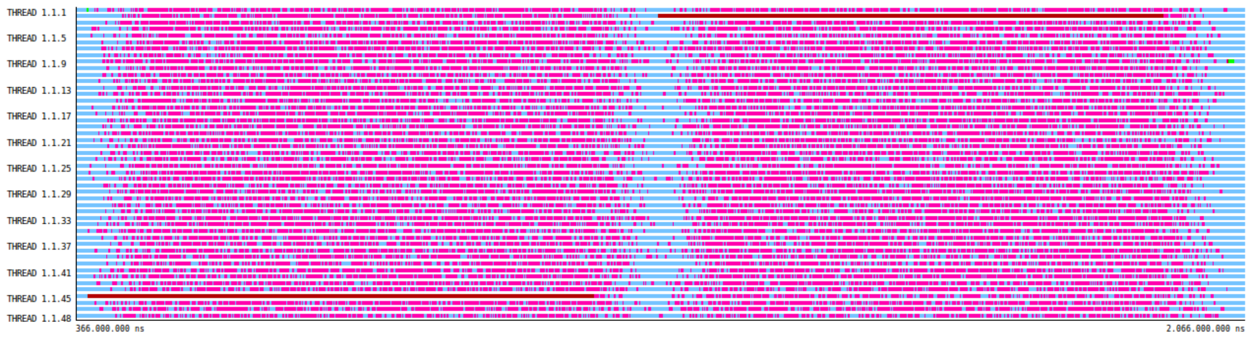


Figure 7.22: Coarse grain N-Body execution trace on ThunderX with 48 threads and Nanos++

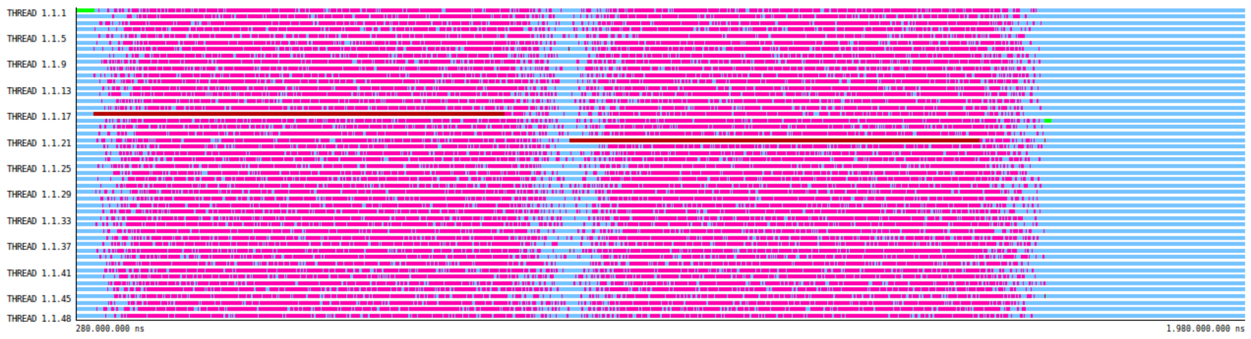
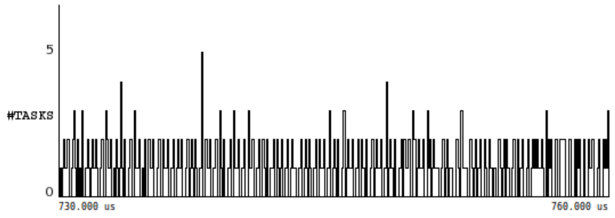


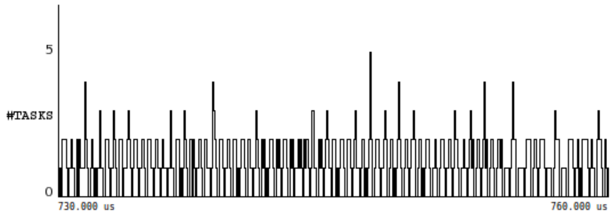
Figure 7.23: Coarse grain N-Body execution trace on ThunderX with 48 threads and DDAST

However, the smaller benchmark execution time is not due to a fast task execution, it is due to the faster execution of the brown tasks than before.

Finally, figure 7.25 shows the number of tasks in graph (y-axes) evolution during the entire two timesteps executions for Nanos++ and *DDAST Runtime*. These traces use the same time scale as figure 7.22 and figure 7.23. The requests to the runtime manager are quickly processed because the worker threads become idle very frequently. This, allows the *DDAST Runtime* increase the throughput of task submission and the number of tasks in the dependence graph increases respect to the Nanos++ baseline. Moreover, the two brown execution regions in figure 7.23 are shorter than the ones in figure 7.22, thereby the creation and submission of children tasks is faster in the *DDAST Runtime* than in the Nanos++ runtime. In addition, the same behavior has been observed in the fine grain executions of the same benchmark explaining the better performance results of DDAST.

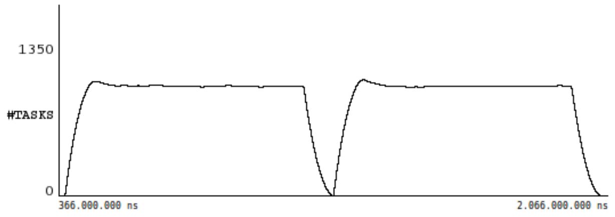


(a) Nanos++

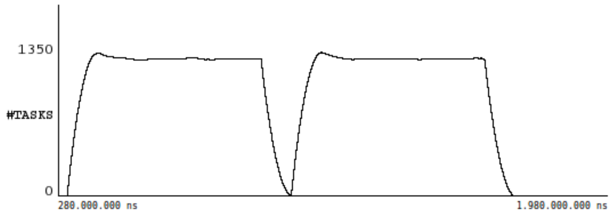


(b) DDAST

Figure 7.24: Evolution of the number of ready tasks on ThunderX with 48 threads for coarse grain N-Body



(a) Nanos++



(b) DDAST

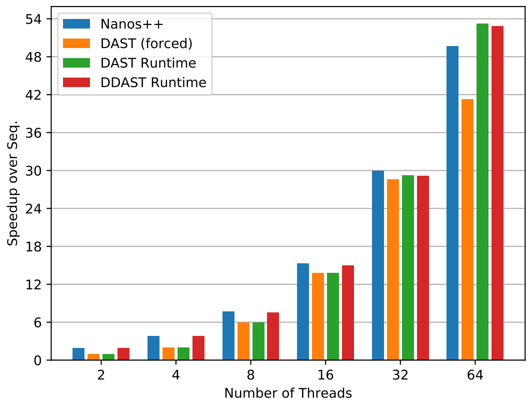
Figure 7.25: Evolution of the number of tasks in-graph on ThunderX with 48 threads for coarse grain N-Body

7.4.3 | Sparse LU

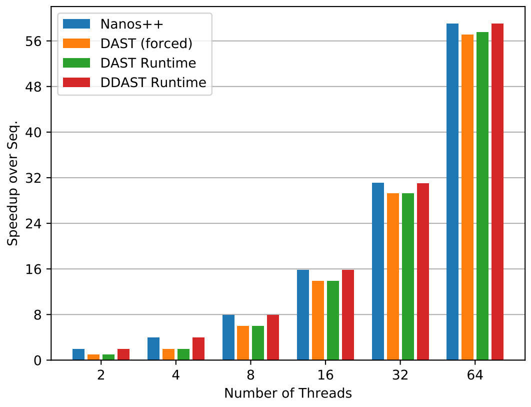
The speedup over the sequential version for the Sparse LU benchmark is shown in figure 7.26 and figure 7.27. Figure 7.26a and figure 7.27a show speedup obtained by the different runtimes against the sequential version of the fine grain configuration of SparseLU in the KNL and ThunderX machines respectively. Figure 7.26b and figure 7.27b show speedup obtained by the different runtimes against the sequential version of the coarse grain configuration of SparseLU for KNL and ThunderX respectively. All of them show the scalability obtained when increasing the number of threads while keeping the benchmark input size (strong scaling).

Regardless the task granularity, all runtimes provide a reasonable scalability in this benchmark. The worst one is the *DAST (forced)* runtime due to its saturation problem, specifically in the fine grain configuration. The data dependences in this benchmark create an irregular task graph that enables the runtime to hide its overhead much better than the other benchmarks. Therefore, the improvement margin for our asynchronous runtime mechanism is smaller. However, the results show that even in this situation the *DDAST Runtime* is able to achieve a performance similar to Nanos++ because the request handling overheads are also hidden.

Figure 7.28 and figure 7.29 show the execution trace of Sparse LU with coarse grain tasks in ThunderX for Nanos++ and DDAST respectively. Each line represents a worker thread (thread 1 to thread 48) and the different colors represent the thread state: sky-blue for IDLE state and other colors for the different task types execution. For instance, the yellow regions represent the execution of main task whose execution creates the other children tasks that are executed in the brown, pink and green regions.

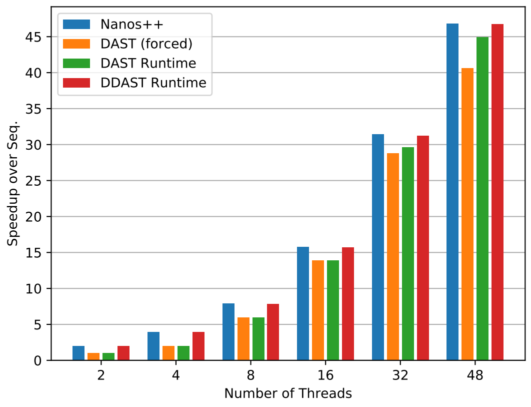


(a) Fine grain

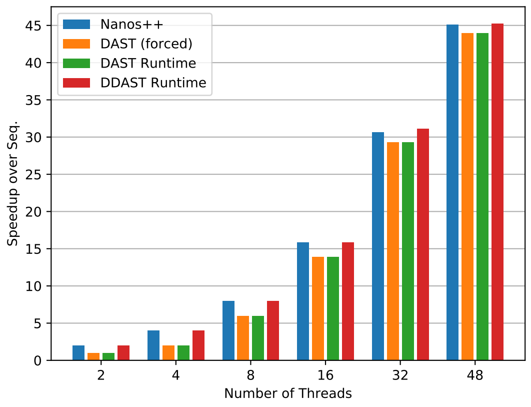


(b) Coarse Grain

Figure 7.26: Sparse LU scalability in KNL



(a) Fine grain



(b) Coarse Grain

Figure 7.27: Sparse LU scalability in ThunderX

The execution traces show that both runtimes follow similar patterns. The major differences are: First, the execution of the main task (yellow regions) is shorter in the DDAST trace, it goes from ∼760ms in Nanos++ to ∼510ms in *DDAST Runtime*. Secondly, in the Nanos++ trace, the execution of green and pink tasks are joined in big chunks. However, in the *DDAST Runtime* they are disperse along all the execution and threads. Finally, in the DDAST execution, there is a point when almost all threads become idle.

The latter behavior can be explained by the fastest task managing of DDAST. Figure 7.30 shows three execution traces for the small portion of the DDAST execution when almost all threads become idle. Figure 7.30a shows the tasks that are executed in each threads, figure 7.30b shows the number of in-graph tasks and 7.30c shows the number of ready tasks. At this point, the explained problem that involves tasks openning parallelism and processing requests at the same time can be seen. Figure 7.30c shows that the number of ready tasks becomes nearly

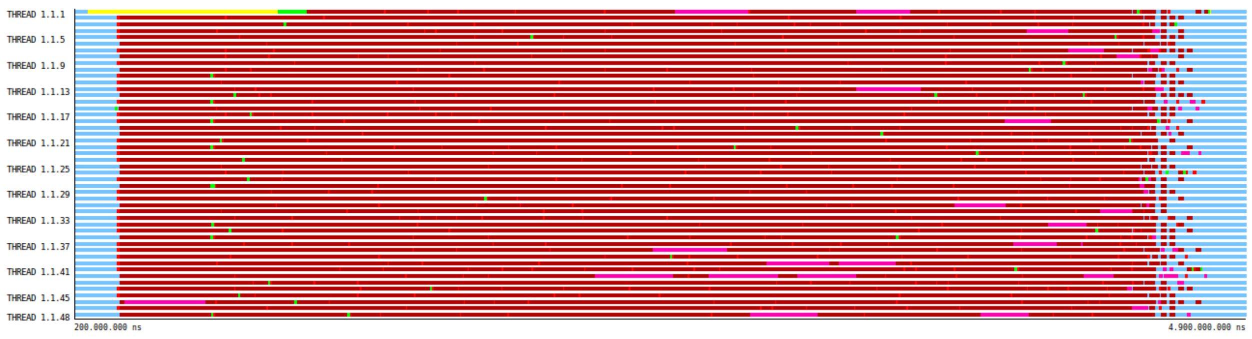


Figure 7.28: Coarse grain Sparse LU execution trace on ThunderX with 48 threads and Nanos++

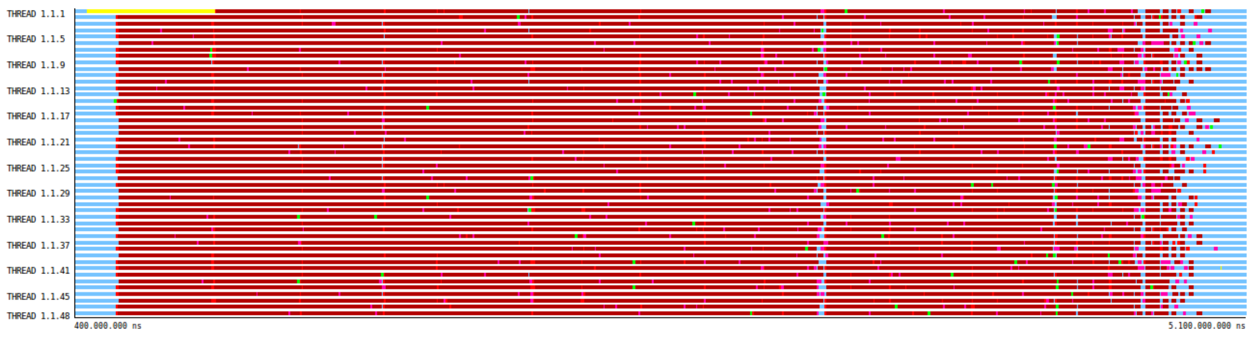
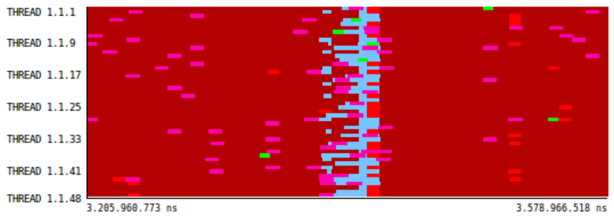


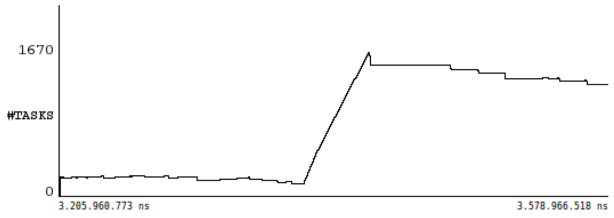
Figure 7.29: Coarse grain Sparse LU execution trace on ThunderX with 48 threads and DDAST

zero for a relative long portion of time. Therefore, almost all threads are idle as can be seen in figure 7.30a, so they start to process the pending requests to the runtime manager. In addition, several tasks are added to the task dependence graph as can be seen in figure 7.30b but their data dependences are not satisfied so the tasks do not become ready. They do so when the *Task Finalization* requests of the critical tasks are processed, at this point the number of ready tasks suddenly increases from zero to more than 100 as can be seen in the ready tasks trace.

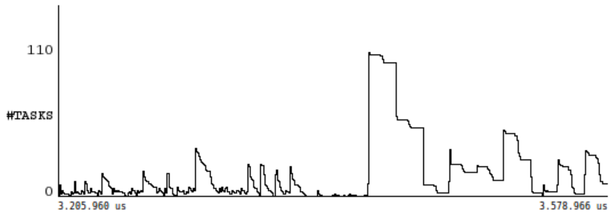
The evolution along all the execution time for the number of tasks in-graph and ready tasks is shown in figure 7.31 and figure 7.32 respectively. Each figure contains a trace for the Nanos++ and the *DDAST Runtime* using the same elapsed time of figure 7.28 and figure 7.29. The evolution of those parameters is equivalent to the observed in the Matmul benchmark. The number of in-graph tasks has a pyramid shaped evolution in the Nanos++ runtime and a plain shape with small peaks in the *DDAST Runtime*.



(a) Tasks execution

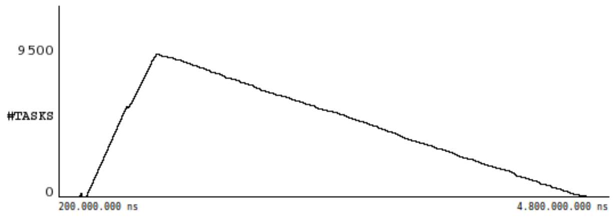


(b) Number of in-graph tasks

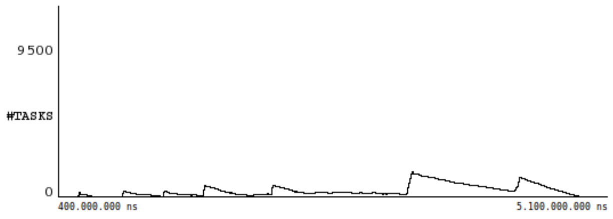


(c) Number of ready tasks

Figure 7.30: Coarse grain Sparse LU partial execution traces on ThunderX with 48 threads and DDAST

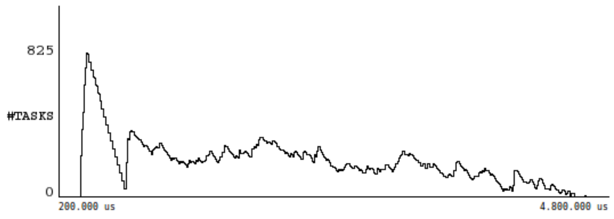


(a) Nanos++

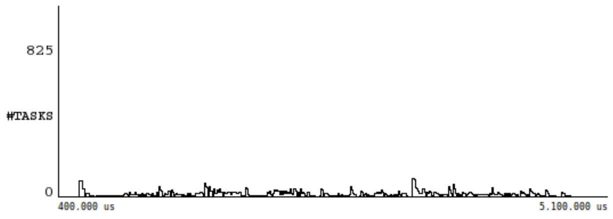


(b) DDAST

Figure 7.31: Evolution of the number of tasks in-graph on ThunderX with 48 threads for coarse grain N-Body



(a) Nanos++



(b) DDAST

Figure 7.32: Evolution of the number of ready tasks on ThunderX with 48 threads for coarse grain Sparse LU

Conclusion

Chapter 8

This master thesis results form the fact that multicore processors have become popular and are present in almost any electronic device nowadays. Task-based parallel programming models, like OmpSs, are one easy way to use such processor architectures by simply annotating the sequential applications' source code. However, the runtime libraries that support such models present a contention problem when the number of threads grows to some tens. As current many-core processors, the future processors are expected to have several cores; thereby the runtimes may become a bottleneck to exploit the applications' performance.

This master thesis uses the OmpSs runtime (Nanos++) as a baseline to implement the master thesis proposals. The first contribution is the extension and implementation of an asynchronous runtime structure based on a centralized runtime manager. In particular, this centralized runtime manager has a new organization to avoid the contention problem in the runtime structure accesses. In this organization, the threads request the runtime structures modification instead of directly doing the changes. The requests are satisfied by a centralized runtime manager that is implemented using an extra thread which only executes runtime code. This thesis contribution allows the runtime to keep its original structure (where the worker threads directly modify the runtime structures) when it is considered better than the asynchronous version. This idea arose from the limitations seen in the initial implementation in [1] where the centralized runtime manager cannot satisfy the requests fast enough and becomes a bottleneck that limits the application's performance. However, the characterization and performance evaluation of this first contribution showed performance issues depending on the target architecture and the application.

The analysis of the first contribution and its performance issues leads to the second contribution of the master thesis, which is a new design and implementation of the asynchronous runtime structure based on a distributed runtime manager. The philosophy is the same as in the previous implementation. It is based on requests from the worker threads to the runtime manager, which modifies the runtime structures handling the requests. In contrast to the previous implementation based on a reserved thread for the manager, this new design in based on the idea that any thread should be allowed to become a manager thread if needed. To implement the distributed runtime manager, the runtime core is extended with new generic modules that provide the possibility of using the idle worker threads to implement runtime services. Moreover, the new runtime manager can be parametrized and outperforms the previous contribution thanks to its parallel and distributed characteristics.

Future Work

This master thesis is part of an ongoing project, so the work does not end with this thesis. As it can be seen in the performance evaluation, the distributed runtime manager implementation outperforms the centralized runtime manager one. Therefore, the future development of the asynchronous runtime model will be focused on the novel distributed design. The two main future work lines are: extend and improve the current manager implementation and reuse the new modules added to the runtime core in other runtime functionalities.

The manager implementation can be extended by allowing only a subset of the worker threads to become manager threads. This approach may be interesting in a heterogeneous many-core architecture, similar to current big.LITTLE ARM processors [27]. Besides, the execution of the runtime manager may be interesting in some other points rather than when a worker thread becomes idle. This extension will require new points where the threads notify the Functionality Dispatcher about their state or execution point. For example, the Functionality Dispatcher may be called when a worker thread reaches a taskwait to execute a reduced subset of callbacks with a high priority.

The new modules introduced in the runtime core are only used to implement the DDAST runtime manager for now. However, its usage to implement other runtime services will be evaluated. For example, the new callback could be utilized to offload tasks to external accelerators by the corresponding architecture plugin. The current Nanos++ implementation is based on a particular helper thread for each accelerator that handles the needed data copies into the accelerator address space and launches the kernel execution. An implementation based on a callback registration into the Functionality Dispatcher would allow any thread to take care of the tasks associated with the accelerators and also to execute tasks providing a better load balance between all the threads. This behavior would be specially beneficient for small systems with specific accelerators (like the ones used in low-power embedded systems) where it would allow them to use all its resources while keeping their power consumption at bay.

Bibliography

[1] Jaume Bosch Pons. Reorganització del runtime Nanos++. B.S. thesis, Universitat Politècnica de Catalunya, 2015. URL `http://hdl.handle.net/2117/77250`.

[2] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492.

[3] L. Dagum and R. Menon. OpenMP: an Industry Standard API for Shared-Memory Programming. *Computational Science Engineering, IEEE*, 5(1):46–55, Jan 1998. ISSN 1070-9924. doi: 10.1109/99.660313.

[4] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12(3):66–73, 2010.

[5] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. Hierarchical Task-Based Programming With StarSs. *Int. J. High Perform. Comput. Appl.*, 23(3):284–299, August 2009. ISSN 1094-3420. doi: 10.1177/1094342009106195.

[6] Jaume Bosch, Xubin Tan, Carlos Álvarez, Daniel Jimínez-Gonzílez, Xavier Martorell, and Eduard Ayguadé. Characterizing and Improving the Performance of Many-Core Task-Based Parallel Programming Runtimes. In *Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM) Annual Workshop*, page to appear. IEEE, 2017.

[7] Xubin Tan, Jaume Bosch, Daniel Jimínez-Gonzílez, Carlos Álvarez, Eduard Ayguadé, and Mateo Valero. Performance analysis of a hardware accelerator of dependence management for task-based dataflow programming models. In *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*, pages 225–234. IEEE, 2016.

[8] Xubin Tan, Jaume Bosch, Miquel Vidal, Carlos Álvarez, Daniel Jimínez-Gonzílez, Eduard Ayguadé, and Mateo Valero. General Purpose Task-Dependence Management Hardware for Task-based Dataflow Programming Models. In *International Parallel and Distributed Processing Symposium (IPDPS)*, page to appear. IEEE, 2017.

[9] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M Badia, Eduard Ayguade, and Jesús Labarta. Productive cluster programming with ompss. In *Euro-Par 2011 Parallel Processing*, pages 555–566. Springer, 2011.

[10] Artur Podobas, Mats Brorsson, and Vladimir 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.

[11] Chuck Pheatt. Intel® Threading Building Blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, April 2008. ISSN 1937-4771. URL `http://dl.acm.org/citation.cfm?id=1352079.1352134`.

[12] Laxmikant V. Kale and Sanjeev 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. ISBN 0-89791-587-9. doi: 10.1145/165854.165874. URL `http://doi.acm.org/10.1145/165854.165874`.

[13] Fahimeh Yazdanpanah, Carlos Álvarez, Daniel Jiménez-González, and Yoav Etsion. Hybrid dataflow/von-Neumann architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 25(6):1489–1509, 2014.

[14] Gilberto Contreras and Margaret 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.

[15] T. Dallou, A. Elhossini, B. Juurlink, and N. Engelhardt. Nexus#: A Distributed Hardware Task Manager for Task-Based Programming Models. In *International Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1129–1138, May 2015. doi: 10.1109/IPDPS.2015.79.

[16] Mateo Valero, Miquel Moreto, Marc Casas, Eduard Ayguadé, and Jesus Labarta. Runtime-Aware Architectures: A First Approach. *Supercomputing frontiers and innovations*, 1(1), 2014. ISSN 2313-8734.

[17] Alejandro Duran, Edurad Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: A PROPOSAL FOR PROGRAMMING HETEROGENEOUS MULTI-CORE ARCHITECTURES. *Parallel Processing Letters*, 21(02):173–193, 2011. doi: 10.1142/S0129626411000151.

[18] Javier Bueno Hedo. Run-time support for multi-level disjoint memory address spaces. 2015.

[19] Programming Models Group BSC. OmpSs User Guide. `https://pm.bsc.es/ompss-docs/user-guide/`, May 2017.

[20] Programming Models Group BSC. Mercurium C/C++/Fortran source-to-source compiler. `https://github.com/bsc-pm/mcxx`, May 2017.

[21] Programming Models Group BSC. Nanos++ Runtime Library. `https://github.com/bsc-pm/nanox`, May 2017.

[22] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. MIC Hardware and Software Architecture. In *High-Performance Computing on the Intel® Xeon Phi*, pages 13–56. Springer, 2014.

[23] Computer Science Department BSC. BSC Application Repository. `https://pm.bsc.es/projects/bar/wiki/Applications`, April 2017.

[24] A. Sodani. Knights landing (KNL): 2nd Generation Intel® Xeon Phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–24, Aug 2015. doi: 10.1109/HOTCHIPS.2015.7477467.

[25] Linley Gwennap. Thunderx Rattles Server Market. *Microprocessor Report*, 29(6):1–4, 2014.

[26] Programming Models Group BSC. BAR-Benchmarks [at] INTERTWinE. `https://pm.bsc.es/gitlab/ompss/bar-benchmarks/`, April 2017.

[27] Peter Greenhalgh. big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. *ARM White paper*, 17, 2011.