Improving the scheduler of the OmpSs-2 runtime

Master thesis by:

Marc Marí Barceló

As part of the Master in Innovation and Research in Informatics
in the specialization of High Perfomance Computing

Advisors:

Vicenç Beltran Querol
Eduard Ayguadé Parra

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya - UPC BarcelonaTech

Barcelona, 27 June 2018

Abstract

Task parallelism is the new paradigm in parallel computing, where the programmer only has to annotate sections of the code as tasks and the data dependences between them. These new paradigms put a lot of pressure on the underlying runtime, in order to get the best performance from the application.

Nanos6, the reference implementation runtime for the task-based OmpSs-2 programming model has, therefore, to be as efficient as possible processing and distributing tasks across shared memory systems. A special challenge are those applications that create lots of very small tasks, which are currently bounded by the internal performance of Nanos6.

The aim of this thesis is to improve the performance of the Nanos6 runtime in the areas of task creation and task scheduling. Specifically, it aims to make task creation and scheduling more scalable for current and future large multi-core systems, and also make it more efficient, to improve the execution time of applications.

Acknowledgements

First, I would like to thank Eduard Ayguadé, who gave me the opportunity to work at BSC. This has been an incredible opportunity, which I have really enjoyed, and where I have learned lots of new things.

I would also like to thank Vicenç Beltran, my manager and mentor during my years at BSC. He has always been very helpful in finding solutions to new problems, and he has always been very understanding of the challenges of working and studying at the same time.

Thanks to Josep Maria Pérez, the lead developer of the Nanos6 project. He has always been open to questions about the runtime internals, and he has been very efficient correcting bugs. His comments on new developments were also very helpful.

My gratitude to the whole Nanos6 team, which are all great developers and great people. It has always been nice to chat with them while having coffee. And specially to Kevin Sala, with whom I shared the office space.

Last, but not least, thanks to all my family. My parents, my grandmother, my brothers and my uncle. All of them have been always very supportive of all my decisions.

Thanks to everybody. The journey has been great, and the future looks bright.

Contents

Abstract ii

Acknowledgements iii

Contents iv

List of Figures vi

List of Tables vii

List of Listings viii

1 The OmpSs-2 programming model 1
- 1.1 Introduction . . . 1
- 1.2 OmpSs and OmpSs-2 . . . 1
  - 1.2.1 Environment . . . 3
- 1.3 Nanos6 . . . 3
  - 1.3.1 Task flow . . . 3
  - 1.3.2 Threads and CPUs . . . 5

2 Task scheduling 6

3 Current Nanos6 scheduler 7
- 3.1 Design . . . 7
  - 3.1.1 Frontend . . . 7
    - Adding tasks . . . 7
    - Obtaining tasks and polling slots . . . 7
    - Other helper methods . . . 8
  - 3.1.2 Backend . . . 8
- 3.2 Design issues . . . 9
- 3.3 Performance results . . . 9
  - 3.3.1 Benchmarks . . . 9
  - 3.3.2 Results . . . 10
    - `multiaxpy-noop` . . . 10
    - `cholesky` . . . 11

4 New Nanos6 scheduler 14
- 4.1 Design . . . 14
  - 4.1.1 Adding tasks . . . 15

4.1.2 Getting tasks . . . 16
4.1.3 Queue threshold . . . 16
4.1.4 Local queue policy . . . 16
4.1.5 Resource management . . . 16
4.2 Improvements . . . 17
4.2.1 Dynamic threshold . . . 17
4.2.2 Load balancing . . . 17
4.3 Performance results . . . 18
4.3.1 Original design . . . 18
`multiaxpy-noop` . . . 18
`cholesky` . . . 19
4.3.2 Dynamic queue threshold . . . 19
`multiaxpy-noop` . . . 19
`cholesky` . . . 20
4.3.3 Load balancing . . . 20
`multiaxpy-noop` . . . 20
`cholesky` . . . 21

5 Memory allocator **23**
5.1 Introduction . . . 23
5.2 New design . . . 23
5.3 Performance results . . . 24
5.3.1 Original scheduler . . . 24
`multiaxpy-noop` . . . 24
`cholesky` . . . 25
5.3.2 New scheduler . . . 25
`multiaxpy-noop` . . . 25
`cholesky` . . . 26

6 Final performance evaluation **27**
6.1 `multiaxpy-noop` . . . 27
6.2 `cholesky` . . . 27
6.3 *multiaxpy* . . . 29
6.4 Other applications . . . 30

7 Conclusions **31**
7.1 Future Work . . . 31

Bibliography **32**

A Applications **33**
A.1 `multiaxpy-noop` . . . 33
A.2 `cholesky` . . . 33

List of Figures

1.1 Stages in the lifetime of a task in the Nanos6 runtime 4
1.2 Memory organization for Nanos6 tasks 4

3.1 Execution time for `multiaxpy-noop` for different block sizes 10
3.2 Profile for `multiaxpy-noop` for block size 1024 11
3.3 Execution time for `cholesky` for different block sizes 12
3.4 Profile for `cholesky` for block size 32×32 12

4.1 Conceptual diagram for the new scheduler design 14
4.2 Execution time for `multiaxpy-noop` using the new scheduler 18
4.3 Execution time for `cholesky` using the new scheduler 19
4.4 Execution time for `multiaxpy-noop` with dynamic queue threshold . . . 19
4.5 Execution time for `cholesky` with dynamic queue threshold 20
4.6 Execution time for `multiaxpy-noop` with load balancing 21
4.7 Profile for `multiaxpy-noop` for block size 1024 21
4.8 Execution time for `cholesky` with load balancing 22
4.9 Profile for `cholesky` for block size 32×32 22

5.1 Conceptual diagram for the new memory allocator design 24
5.2 Execution time for `multiaxpy-noop` with the memory allocator 25
5.3 Execution time for `cholesky` with the memory allocator 25
5.4 Execution time for `multiaxpy-noop` with the memory allocator 26
5.5 Execution time for `cholesky` with the memory allocator 26

6.1 Execution time for `multiaxpy-noop` with all improvements 27
6.2 Profile for `multiaxpy-noop` with all improvements for block size 1024 . . 28
6.3 Execution time for `cholesky` with all improvements 28
6.4 Profile for `cholesky` with all improvements for block size 32×32 29
6.5 Execution time for `multiaxpy-recursive` with all improvements 29
6.6 Execution time for `multiaxpy-recursive-10it` with all improvements . 30
6.7 Speedup for several applications with all improvements 30

A.1 Multiaxpy vector splitting pattern 33

List of Tables

A.1 Number of tasks created in the `multiaxpy-noop` application 34
A.2 Number of tasks created in the `cholesky` application 34

Listings

1.1 OmpSs programming model example 2

1. The OmpSs-2 programming model

1.1 Introduction

With current supercomputers, such as Marenostrum 4 [1], being composed of thousands of compute nodes, and each of them of tens of cores, it has become a necessity to make use of parallel programming models and runtimes such as OpenMP [2] or MPI [3] to ease the parallelization of applications and make a better use of the available resources. This also allows easier portability of applications between systems, as the architectural complexities of parallel programming are inside the runtimes.

One of these progamming models is OmpSs [4], a task-based programming model for shared-memory systems, developed at the Barcelona Supercomputing Center (BSC) [5].

1.2 OmpSs and OmpSs-2

The OmpSs [4] programming model is the result of combining ideas from the OpenMP and StarSs programming models. On one hand, it takes from OpenMP its viewpoint of providing a parallel version of the application by introducing annotations in the sequential source code. These annotations do not have an explicit effect in the semantics of the program, instead, they allow the compiler to produce a parallel version of it. This removes the need to redesign the application like in explicit programming models, which reduces the efforts required to debug and test.

On the other hand, it inherits from StarSs its thread-pool execution model, opposite to OpenMP's fork-join parallelism. It also includes features to target heterogeneous architectures through leveraging native kernels implementation. And finally, StarSs offers asynchronous parallelism as the main mechanism of expressing concurrency (OpenMP only started to implement it since its version 3) and sychronization by means of task dependences, enabling the look-ahead instantiation of tasks (OpenMP included it since version 4).

StarSs raises the bar on how much implicitness is offered by the programming model. In OpenMP, the developer has to find which regions of the program will be executed in parallel, and then he has to express how the code has to be executed by the threads forming the parallel region and, maybe, how to synchronize them. Conversely, StarSs provides an environment where parallelism is implicitly created from the beginning of the execution, and the definition of parallel code is done through the concept of tasks, pieces

of code that can be executed asynchronously in parallel, with the order of execution defined by a dependency mechanism.

The goal of OmpSs is to combine these ideas from OpenMP and StarSs into an environment to develop applications for modern High Performance Computing systems that is both easy to use and delivers high performance compared to other programming models targeting the same architectures.

In OmpSs, tasks are the elementary unit of work which represents a specific instance of an executable code. Dependences let the user annotate the data flow of the program and specify relationships between tasks. This way, at runtime this information can be used to determine if the parallel execution of two tasks may cause data races. Listing 1.1 shows a simple example of a code parallelized with the OmpSs programming model.

```
int compute() {
  #pragma oss task out(x)
  x = compute_x();

  #pragma oss task out(y, z)
  {
    y = compute_y();
    z = compute_z();
  }

  #pragma oss task in(x, y, x) out(result)
  result = compute_result(x, y, z);

  return result;
}
```

Listing 1.1: OmpSs programming model example

Other ways that the user can influence the order of execution of tasks are, for example, task priorities (assigning a higher priority to a task will indicate the scheduler that it should schedule that task sooner),

OmpSs-2 [6] is the second generation of the OmpSs programming model. It extends the tasking model of OmpSs to support both task nesting and fine-grained dependences across different nesting levels, which enables the effective parallelization of applications using a top-down methodology. Some of its most prominent features are:

- Nested dependency domain connection
- Early release of dependences
- Weak dependences
- Native offload API

- Task pause/resume API

1.2.1 Environment

The reference implementation of OmpSs-2 is based on the Mercurium source-to-source compiler and the Nanos6 runtime library.

The Mercurium source-to-source compiler [7] provides the necessary support for transforming the high-level directives into a parallelized version of the application.

The Nanos6 runtime [8], on the other hand, provides the services to manage all the parallelism in the user-application, including task creation, synchronization and data movement, and provide support for resource heterogeneity. This runtime and its performance will be the focus of this thesis.

1.3 Nanos6

The goal of this project is to improve the internal performance of the Nanos6 runtime. In specific, this project aims to evaluate and improve the performance of the Nanos6 runtime in the areas of task creation and task scheduling. These areas are part of the critical path of the task execution lifetime, and, therefore, it is important to be as efficient and as scalable as possible. These areas have been chosen because, as will be seen through this document, there is room for improvement, and they also are self-contained areas that can be completely redesigned without modifying significantly the rest of the runtime.

Nonetheless, it is critical to understand the internal organization of Nanos6 before approaching the problems on those specific areas. In particular, it is important to take into account the different stages in the lifetime of a task, and how it relates with the final execution components: threads and CPUs.

1.3.1 Task flow

As mentioned, the task is the basic unit of execution in the OmpSs and OmpSs-2 programming models. It is then important to understand the different stages of a task through its lifetime to understand how the Nanos6 runtime works and where its performance can be improved. Figure 1.1 is a diagram of the possible stages in which a task can be.

Before the runtime knows about the existence of a task, it is necessary to create and fill all its information and data structures. This is handled by the compiler, who is responsible of translating the `#pragma oss` statements into data structures and runtime API calls.

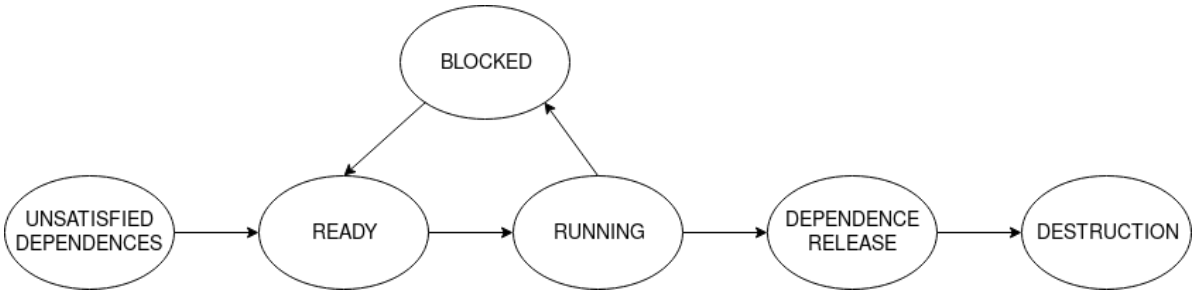


Figure 1.1: Stages in the lifetime of a task in the Nanos6 runtime

The two main Nanos6 API calls related to task creation tasks are:

- `nanos_create_task`: allocates the necessary memory in the runtime address space for both the task and its arguments. This is a continuous region aligned to 128 bytes, as shown in Figure 1.2.
- `nanos_submit_task`: submits a task to the runtime to be ran.

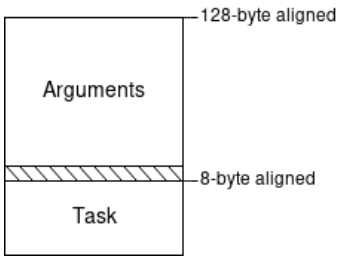


Figure 1.2: Memory organization for Nanos6 tasks

After the task is submitted to the runtime, its dependences with other tasks are checked. If these dependences are not satisfied, this task is left pending until it can run.

Once a task is ready to run, this is, all task dependences are satisfied, this task is submitted to the scheduler. The scheduler is the responsible of assigning tasks to available resources (idle CPUs). The first time that a task is scheduled, a thread is assigned to it, which will not change for the lifetime of the task.

If the task is blocked while running (for example, in a `taskwait`), the CPU in which it is running will be freed so other tasks can run on it. When the blocking condition is released (when all children tasks finish, in the case of a `taskwait`), the task will be sent to the scheduler again so that it can obtain a new CPU to run on. Nonetheless, as mentioned before, the thread assigned to the task will not change, it will only migrate CPUs.

When the task finishes, it will free its dependences (which, in turn, may make new tasks ready), and its data structures will be deleted.

1.3.2 Threads and CPUs

As mentioned in the previous section, a task is assigned to one and only one thread during its lifetime. Nonetheless, a thread may have more than one task assigned to it. All the tasks that are assigned to the same thread are always descendants of one another, to avoid deadlocks. These limitations are because these threads are standard Linux *pthreads*. Therefore, it is not possible to reorder the stack of the thread, and it is necessary to release the stacked tasks in the proper order.

When a task is scheduled, a free CPU is assigned to it. Therefore, the thread that is running that task must be pinned to that specific CPU and, if the task is rescheduled after a blocking condition (a `taskwait` for example), the thread must migrate and pin itself to a new CPU. If a thread was to fetch a task from the scheduler that already had a thread assigned, the fetching thread would release the CPU to allow the new thread to run.

When all tasks assigned to one thread finish and there are no more tasks pending in the scheduler, that thread is marked as idle and it is stored in an idle thread pool. When a new task requires a thread, this idle thread will be reused, instead of creating a new one.

2. Task scheduling

The problem of scheduling in task-based parallel runtimes has been widely discussed, specially since the introduction of tasks in OpenMP. As mentioned in [9], an efficient task scheduler must meet challenging and sometimes conflicting goals: exploit cache and memory locality, maintain load balance and minimize overhead.

In [10] and [11] multiple scheduling policies are discussed, each with its pros and cons:

- Predetermined scheduling: for very regular applications, it is possible obtain an excellent performance with an static schedule. But for most applications it will not be possible to precalculate a good schedule.
- Opportunistic scheduling: uses a centralized or distributed list of ready tasks associated to load balancing mechanisms. This approach is the most natural but does not use affinity information from the compiler or the programmer, and thus cannot achieve the best performance.
- Negotiated scheduling: intermediate solutions between predetermined and opportunist. This includes programming languages like OpenMP where the programmer just has to give suggestions and clues, sometimes indirectly, and the compiler and runtime adapt the scheduling to different parallel architectures. This is also the case in OmpSs and OmpSs-2, where, for example, dependences can indicate the relationships between data used by two tasks.

According to [12], a hierarchy of task lists generally brings better performance than simple per-processor lists. And it also makes task binding to processor sets easier.

Authors in [9] tested several scheduling policies for OpenMP, and compared them. Their results show that hierarchical approaches that respect data locality in the underlying architecture give the best performance. In specific, they implemented a scheduler with one shared LIFO queue per NUMA node, with work stealing that obtained better performance than other implementations such as single queue schedulers or schedulers with a queue per core with and without work stealing.

Other authors such as [10] have also developed a hierarchical scheduler, in this case aimed to better data distribution. This approach also obtained a significant improvement.

3. Current Nanos6 scheduler

3.1 Design

In Nanos6, the scheduler is the responsible of assigning resources (CPUs) to the tasks that are ready to run. As mentioned previously, a task is ready to run (and therefore, queued on the scheduler) when all its dependences are satisfied.

The current implementation of the Nanos6 scheduler is based on a single-queue model, protected by a lock, which allows much more control, but becomes a bottleneck in systems with a high number of CPUs or with applications that create lots of small tasks, as already studied by other authors, and mentioned in Chapter 2.

This scheduler is composed of two sections clearly defined: its frontend (how it interacts with the rest of the runtime), and its backend (how it manages the tasks internally, and how this behaviour can be changed).

3.1.1 Frontend

Adding tasks

There are two different ways to add tasks to the scheduler. One one side, the method `addReadyTask` adds a newly created task to the scheduler and returns an idle CPU if there is one. If an idle CPU is returned, an idle thread needs to be woken up on that CPU, so that it can get the newly added task from the scheduler.

On the other side, the method `taskGetsUnblocked` adds a recently unblocked task to the scheduler. It does not return anything.

Obtaining tasks and polling slots

The method `getReadyTask` returns a task from the scheduler, if there are any available.

It is also possible to obtain tasks by means of *polling slots*. A polling slot is basically an atomic variable that can store a task. A thread that wants to actively wait for tasks for a period of time can set up a polling slot, and as soon as a task becomes ready to be scheduled, it will be set on the polling slot so that the thread can start running it immediately.

These polling slots reduce the overhead on the scheduler, as polling threads will not be continuously requesting the lock of the scheduler queue (which would be the case if `getReadyTask` was called repeatedly). They also reduce overhead on the threading system, as suspending and resuming a thread is an expensive operation (as mentioned previously, threads are Linux *pthreads*, therefore suspending means locking on a condition variable and switching context on the OS, and resuming is the opposite operation).

To work with polling slots the methods `requestPolling` and `releasePolling` are available, which set and free the internal scheduler's polling slot, respectively. These methods return a boolean value indicating if the operation was or not successful.

Other helper methods

Generally, the resources assigned to the runtime are set in the initialization phase and are not modified throughout the execution. But, in cases where Nanos6 needs to share its resources with other runtimes or applications running in the system, it is possible to increase or reduce the number of CPUs assigned to it. To notify the scheduler of these changes, the methods `disableComputePlace` and `enableComputePlace` are available.

It is also possible to obtain a CPU that is currently idle through the `getIdleComputePlace` method. This is useful, for example, in cases where a task is unblocked and it is necessary to run it as soon as possible.

3.1.2 Backend

This generic interface with the rest of the runtime explained in the previous section allows for different scheduling policies to be implemented and tested. The main scheduler implementations currently available are:

- LIFO and FIFO schedulers: a simple single queue LIFO or FIFO scheduler. These basic implementations don't have polling slots available. These are implemented with a call to `getReadyTask`.
- LIFO and FIFO schedulers with polling slots: similar to the previous ones, but with polling slots available.
- LIFO and FIFO schedulers with polling slots and immediate successor awareness: similar to the previous ones, but these also try to schedule the first successor of the currently running task in the same CPU to increase data locality and reduce scheduling time.
- Priority scheduler with polling slots and immediate successor awareness: like the previous ones, but with a priority queue instead of a FIFO or LIFO, for those cases where tasks have priorities. This is currently the default scheduler in Nanos6.

3.2 Design issues

Aside from the obvious performance and scalability issues of using a single queue in systems with a high number of cores (analyzed in Section 3.3), there are also design problems with the frontend of the scheduler that need to be taken into account for a redesign:

- Adding unblocked tasks: there is no real need of having a specific method for adding tasks that are unblocked. The `addReadyTask` method already provides a parameter to give a "hint" to the scheduler, which is used to indicate, for example, if the task currently being added to the scheduler is a child or a sibling of the currently running task. This hint could be used to also indicate if the task is an unblocked task.
- Polling slots: these were introduced to reduce the overheads caused by the internal scheduler lock and the thread suspension and resumption process. But polling slots also expose part of the internal scheduler workings to the rest of the runtime, which is not desirable. They could be scrapped completely, and its functionality could be integrated in the `getReadyTask` method: if there are no tasks available, instead of returning immediately, poll for a certain period of time.
- Thread resumption: it is also important to address who should be the responsible of waking up threads when tasks are added to the scheduler. The current implementation returns an idle CPU, and the calling code is responsible of waking up a thread on that CPU. This means that all calls to `addReadyTask` across the runtime are followed by a call to wake up a thread on a certain CPU if it is non-null. These code snippets could be easily integrated inside the scheduler, if desired.

3.3 Performance results

3.3.1 Benchmarks

Before analyzing the current performance of the Nanos6 scheduler, its shortcomings and possible improvements, it is very important to decide on a set of benchmarks that will serve as a basic metric. For this purpose, two benchmarks have been selected: `multiaxpy-noop` and `cholesky`.

The application `multiaxpy-noop`, described in Appendix A.1, is a very synthetic application, purely aimed to stress as much as possible the task creation and scheduling subsystems of Nanos6. Very few, if any, real-world applications will create as many tasks as fast as this example. But this extreme example will serve as a good metric of how efficient and scalable task creation is on Nanos6.

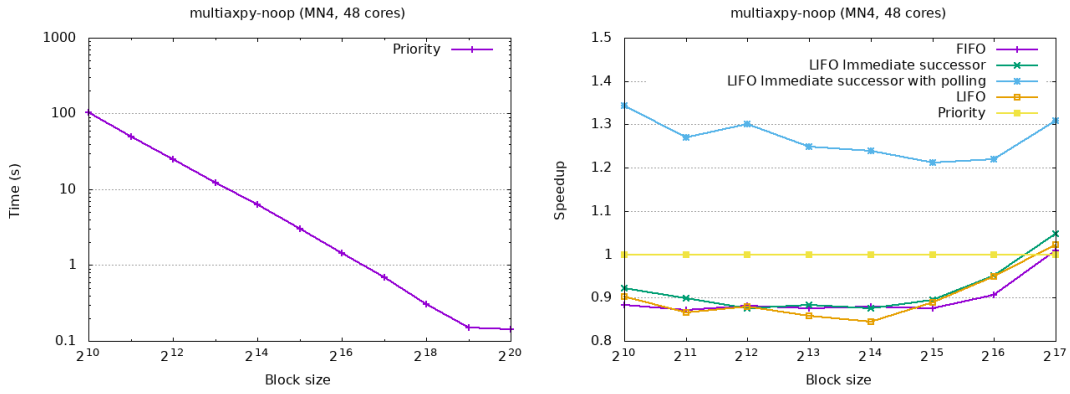
The application `cholesky`, described in Appendix A.2, on the other hand, is a very classic benchmark. It has only one task creator, it has dependences, and creates much fewer tasks than the previous example. Therefore, it has been chosen as a complete opposite of the previous application, in order to avoid designing and implementing solutions based on very specific scenarios.

Apart from these basic benchmarks used through the design and development of new solutions, several more applications have been tested against the proposed final solution, to verify that it is a valid solution that improves or, at least, does not hinder the performance with respect to the original implementation.

All these applications have been run in multiple machines at BSC, but this thesis focuses on Marenostrum 4, as results were similar across machines. Marenostrum 4 [1] is a supercomputer composed of 3456 nodes, each of them with 2 24-core Intel Xeon Platinum 8160 at 2.1 GHz and 96 GB of RAM.

3.3.2 Results

`multiaxpy-noop`



(a) (b)

Figure 3.1: Execution time for `multiaxpy-noop` for different block sizes

Figure 3.1a shows the results of running the benchmark `multiaxpy-noop` with a vector size of 1G elements and 10 iterations for different block sizes, and the default Nanos6 scheduler, the priority scheduler.

From these results, it is clear that, for block sizes bigger than 2^{19}, the execution time is bounded by the application itself. But for smaller block sizes, the execution time is bounded by the scalability and the load of the runtime. In specific, it is bounded by the runtime's task creation and scheduling times, as this application is not accessing data (so there can be no effect from caches or NUMA nodes) and it has no dependences or other constructs (so there can be no effect from other parts of the Nanos6 runtime). For

this reason, only the lower part of the plot (more interesting) is shown in the rest of the analysis.

Figure 3.1b, on the other hand, shows the speedup of the different backends implemented in Nanos6 with respect to the priority scheduler shown before. From this plot, it is clear that, for this benchmark, the scheduler with both polling slots and immediate successor slots achieves the best performance.

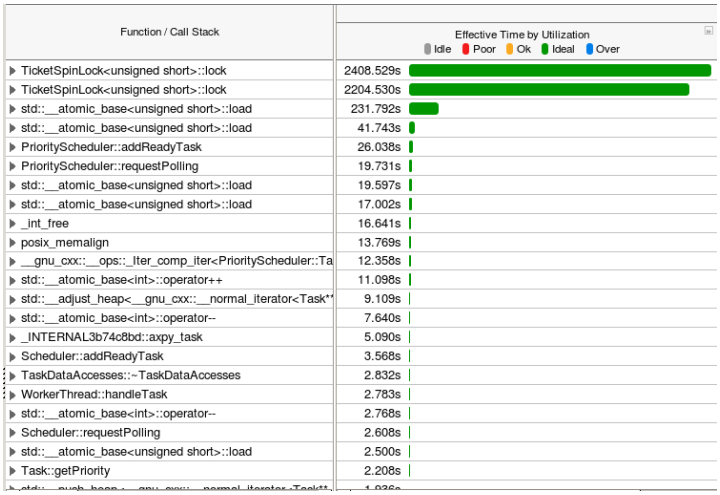


Figure 3.2: Profile for `multiaxpy-noop` for block size 1024

A detailed profile of the execution for the smallest block size, in Figure 3.2, shows that in this application most of the execution time is spent waiting for permission to access the scheduler's lock. Specifically, when accessed from the functions `requestPolling` (first) and `addReadyTask` (second). This is mainly because of the centralized nature of the current scheduler implementations.

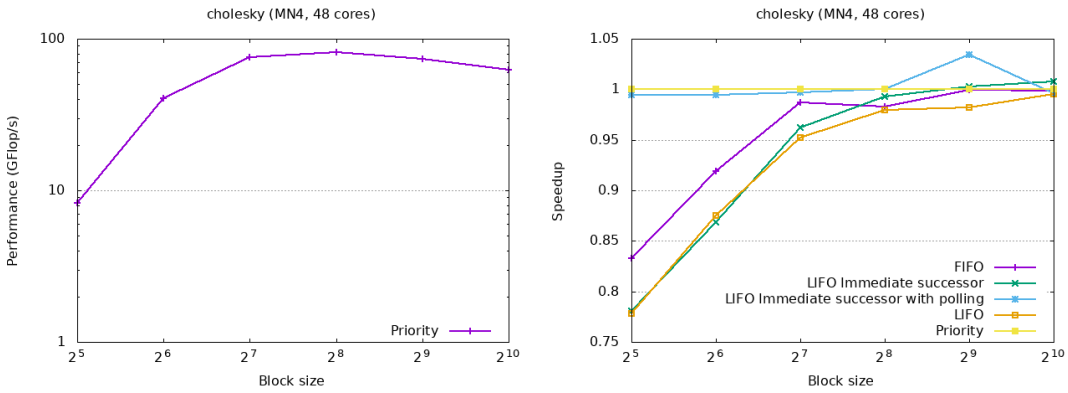
Other calls that take an important amount of time are the atomic loads of the polling slots.

In conclusion, this benchmark shows that there is a lot of margin in the Nanos6 scheduler to improve its throughput and scalability.

`cholesky`

Figure 3.3a shows the perfomance of the priority scheduler to solve a *Cholesky* factorization on a 16K $\times$ 16K element matrix for different block sizes. Figure 3.3b, on the other hand, shows the relative performance of different schedulers with respect to the priority scheduler.

These figures show that this application also looses performance with very small sizes. But, in this case, due to the characteristics of the application, it may be because of multiple factors, not only lack of performance in task creation.



(a) (b)

Figure 3.3: Execution time for `cholesky` for different block sizes

With respect of the different scheduler backends, the highest performance is achieved by the scheduler with immediate successor slots and polling slots, independent of its internal queue (LIFO or priority).

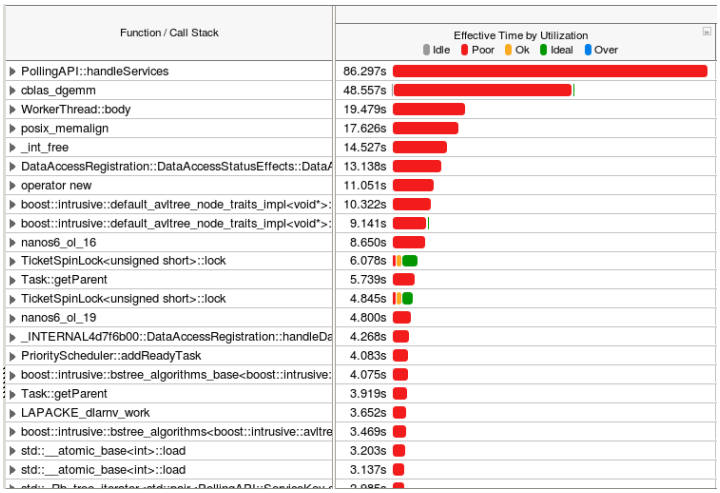


Figure 3.4: Profile for `cholesky` for block size 32×32

The profile of the execution of this benchmark for the smallest size (Figure 3.4) shows a much better picture than in the case of `multiapxy-noop`.

Most of the time is spent in the function `handleServices`, part of the polling services API subsystem. This function is called periodically from the threads that are idle, which means that there are no ready tasks available in the scheduler. If the thread that is creating tasks was able to create and schedule tasks faster, the rest of the threads may be able to obtain tasks faster, which would improve overall performance.

The next most time consuming section is the actual body of the benchmark, followed by the Nanos6 thread body function. After that come functions related to task memory allocation and release, and dependence management.

In conclusion, this benchmark shows that, for complex applications, the time spent in the runtime is distributed in many different areas. In particular, threads polling the scheduler for tasks represent an important amount of time (as polling services are called when the scheduler returns no tasks) which could be reduced by making the task creation and scheduling process faster.

4. New Nanos6 scheduler

4.1 Design

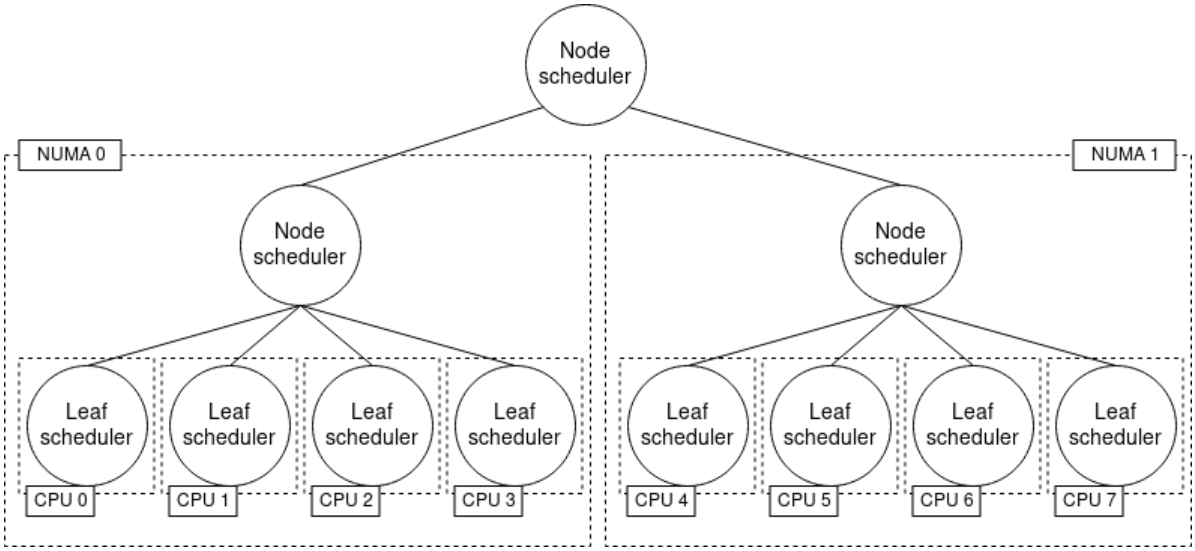


Figure 4.1: Conceptual diagram for the new scheduler design

With the ideas from literature from Chapter 2, and the ideas from the current implementation, a new scheduler was designed, trying to modify as little as possible the interface with the rest of the runtime. This new scheduler is a distributed scheduler in the shape of a tree, as shown in Figure 4.1, where each of the nodes has a direct correspondence with a hardware component in the underlying hardware architecture.

The two components of this tree structure are:

- Leaf nodes: there is one of these nodes for each CPU in the system. They are the entry point to the scheduler (so several operations can take place in parallel in different CPUs), and each of them has a local queue of pending tasks.
- Internal nodes: there is one of these nodes for each NUMA node in the system (with the CPU nodes as children), and one node as the the root (with the NUMA nodes as children). Each of them has a local queue of pending tasks, and a local queue of idle children.

This design allows for more scheduling work done locally in each CPU, without the need to interfere with the execution of other CPUs, which reduces locking and contention. However, a design based on local queues must also enable mechanisms of work stealing

or, at least, load balancing, so that no CPUs are idle while others have big backlogs of tasks.

For this reason, a queue threshold has been introduced, which is the responsible of keeping all scheduler nodes with a similar load (a similar number of tasks enqueued). This way, it is not necessary for CPUs to steal work from other nodes, as the nodes themselves send their extra backlog of tasks to their nearest idle neighbor.

In the future, if a task creation cutoff mechanism was introduced in Nanos6, the number of tasks in the scheduler can be estimated from this queue threshold value, giving a reasonable estimate, at the cheap cost of checking one atomic variable.

This tree-shaped distributed scheduler also improves data locality, as it is prioritized that tasks are scheduled in the same node where they are created, or, if not possible, the same NUMA node.

In the following sections, it is explained how these nodes interact with each other in the different scenarios they can encounter.

4.1.1 Adding tasks

When the method `addReadyTask` of the scheduler is called, the task is added to the local task queue of leaf node that corresponds to the calling thread's CPU. It is directly added to the queue because it is impossible for that CPU to be creating tasks and be idle (and requesting tasks) at the same time.

The only two exceptions to this are the main task, and tasks that are unblocked from an external thread. In these special cases the task is sent to a random leaf node which may be idle (then, the task must be stored in the polling slot) or may not be idle (the task is added to the local task queue of the node).

When a task is pushed to the local task queue, it might be the case that there are too many tasks in that queue (the queue size is over the threshold). In this case half of the queue tasks are submitted to the parent node. The parent node checks if there are any children in its idle children queue to submit the set of tasks that it received, and if there were none, it enqueues the set of tasks locally. If the queue size of this node is also over the threshold, the process is repeated recursively.

If, on the other hand, there were idle children available, the set of tasks is sent to one of those children, who stores them locally. If the child is a leaf node, it will store one task in the polling slot and, if there is no thread polling, start a thread on its corresponding CPU to start running the newly added tasks. Therefore, with this design there is no need for `addReadyTask` to return an idle CPU as it was the case in the previous design.

4.1.2 Getting tasks

When the method `getReadyTask` of the scheduler is called, a task is fetched from the leaf node that corresponds to the calling thread's CPU. First, the polling slot is checked. If no tasks are pending on the polling slot, a task is extracted from the queue. If there are no tasks in the queue, tasks are requested from the parent node and polling starts. If after some time polling no tasks are obtained, the corresponding CPU is marked as idle, and the thread can be suspended.

When tasks are requested from the parent node, first the parent's local queue is checked. If there are tasks in the local queue, some of them are returned to the child. Otherwise, the child is pushed to the idle children queue, and tasks are requested to the parent recursively.

4.1.3 Queue threshold

A very important concept for the correct load balancing of this scheduler design is the queue threshold. This threshold determines the maximum number of tasks that can be stored in the local queue of a node. It also determines how many tasks will be sent to other nodes when requested or when there is an overflow, which is defined at half of the threshold value.

Therefore, it is important to set properly this value to obtain good performance results.

4.1.4 Local queue policy

One aspect not mentioned about this design is the policy of the internal local queue. This is on purpose, because there are no requirements on the internal policy of this queue. It can be implemented as a FIFO, LIFO or any other policy that may seem fit. The only requirement is the interface, which must provide *get* and *add* methods for both one task and a set of tasks.

The queue policies that were implemented in this initial version are:

- FIFO: when only one task is requested, the least recently added tasks (oldest) are returned. But when a set of tasks is requested, the youngest are returned.
- LIFO: when one task is requested, the youngest is returned. But when a set of tasks is requested, the oldest are returned.

4.1.5 Resource management

Enabling and disabling resources is also possible in this scheduler. When a CPU is disabled, the leaf node corresponding to that CPU will push all its locally queued tasks

(if any) to its parent, and request its parent to unqueue it from the idle children queue (if it is idle).

To enable CPUs, no further work is necessary, as the responsible of enabling the CPU will take care of resuming a thread to start getting tasks.

4.2 Improvements

Once the basic design was implemented and tested, several improvements were proposed and implemented. These improvements addressed several shortcomings of the initial design. These are detailed in the sections following.

4.2.1 Dynamic threshold

The first improvement was to make the threshold independent of the application or the stage the application is in. When the threshold was static, for cases with only one task creating tasks and not many tasks created in total (like the `choleksy` benchmark with a big block size), a lot of work was left pending in one CPU's local scheduler, while the rest of CPUs were idle.

For this reason, a dynamic queue threshold was implemented. The threshold starts with the value 0, which means that no tasks can be stored in the local queues. Therefore, at the beginning of the application, all CPUs will receive tasks irregardless of its creator. When overflowing tasks arrive to the topmost scheduler node, the threshold is doubled (or set to 2) for all nodes. This way, all scheduler nodes will have a similar number of tasks pending.

On the other hand, when tasks are requested from the root node, which means that some leaf node is idle, the threshold of the system is halved.

4.2.2 Load balancing

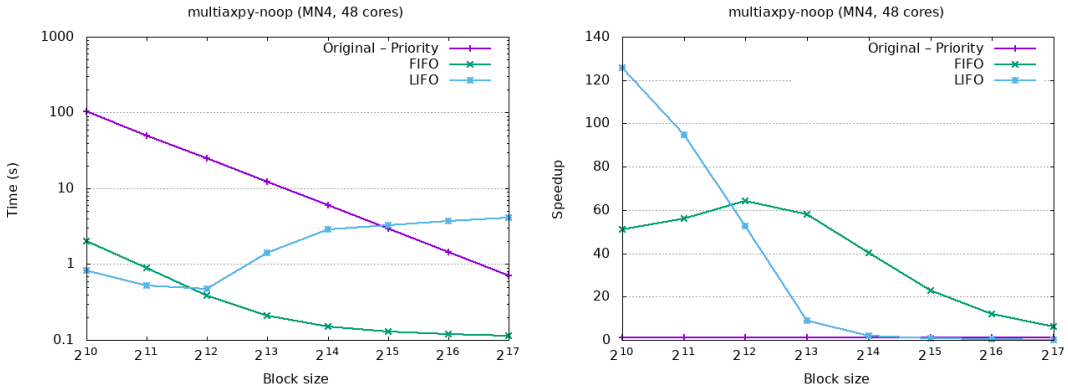
The dynamic threshold solved the problem of load balancing during task creation. But it did not solve the problem of load balancing at the end of the application. In this case, there could be lots of (presumably long) tasks queued on a CPU, while the rest of CPUs are idle.

This problem is solved by rebalancing tasks when the threshold is reduced. If the node has tasks over the threshold when it is reduced, these extra tasks are pushed to the parent node following the same algorithm as the other cases.

4.3 Performance results

4.3.1 Original design

`multiaxpy-noop`



(a) (b)

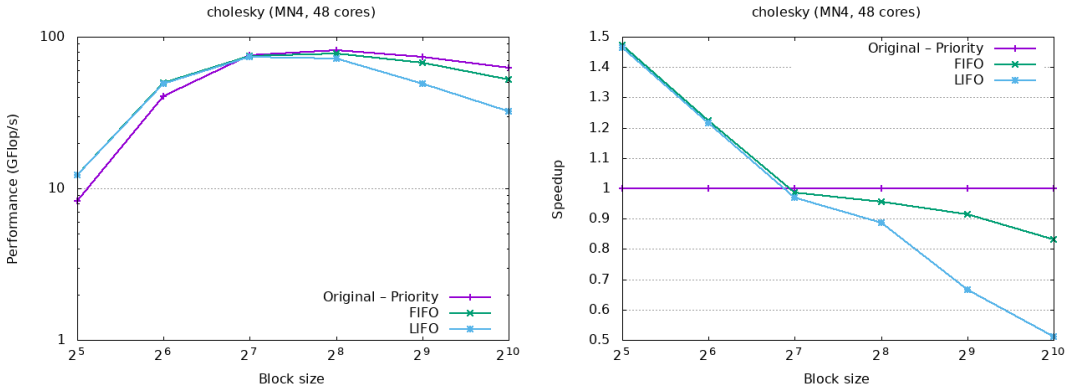
Figure 4.2: Execution time for `multiaxpy-noop` using the new scheduler

Figure 4.2 shows the absolute execution time and the speedup with respect to the original scheduler for two different policies of the internal local queues with a threshold of 20 tasks.

It is clear from these plots that there is a significant performance improvement, specially in those cases with a very small block size (and therefore, a high number of tasks). With this new design, the cases where execution time is practically constant extend up to block size 2^{14} for the scheduling policy FIFO.

It is also interesting to see the influence of the internal scheduling policy on the execution time. This influence is mainly due to the lack of load balancing mechanisms in the current design. For this benchmark in particular, the thread that runs the main task will enqueue to its local queue the highermost nodes in the recursion tree of the application. If these highermost tasks are executed as soon as possible (FIFO policy), more parallelism will be enabled for the rest of the threads. If, on the other hand, the highermost tasks are delayed (LIFO policy), there will not be enough parallelism and threads will have to wait more.

With these results in mind, it is clear that it is necessary to introduce load balancing mechanisms in this new design.



(a) (b)

Figure 4.3: Execution time for `cholesky` using the new scheduler

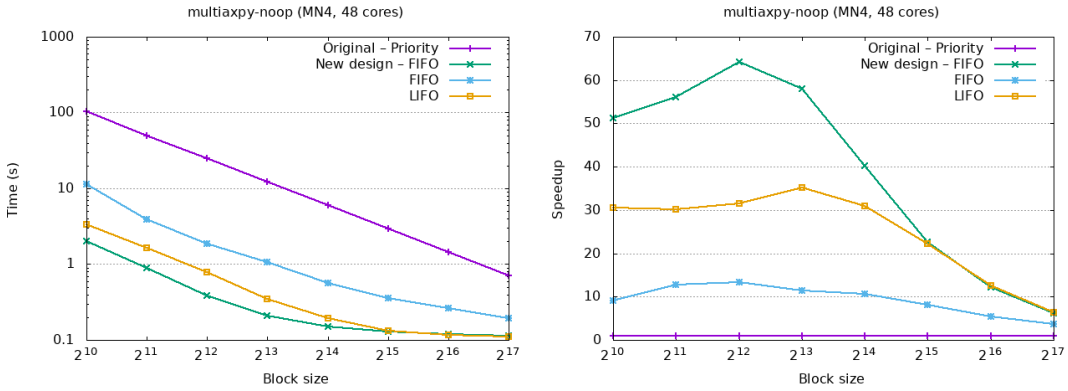
`cholesky`

The `cholesky` benchmark (Figure 4.3), also executed with a queue threshold of 20, does not show an improvement as big as the `multiaxpy-noop` benchmark. But, for the cases with a small block size, it shows an improvement of up to 50%.

However, what this benchmark also shows is that for big block sizes there is a very significant drop in performance. This is also due to lack of load balancing, because there is only one task creator, which will enqueue tasks locally and keep the rest of CPUs idle until enough tasks are created to produce an overflow.

4.3.2 Dynamic queue threshold

`multiaxpy-noop`



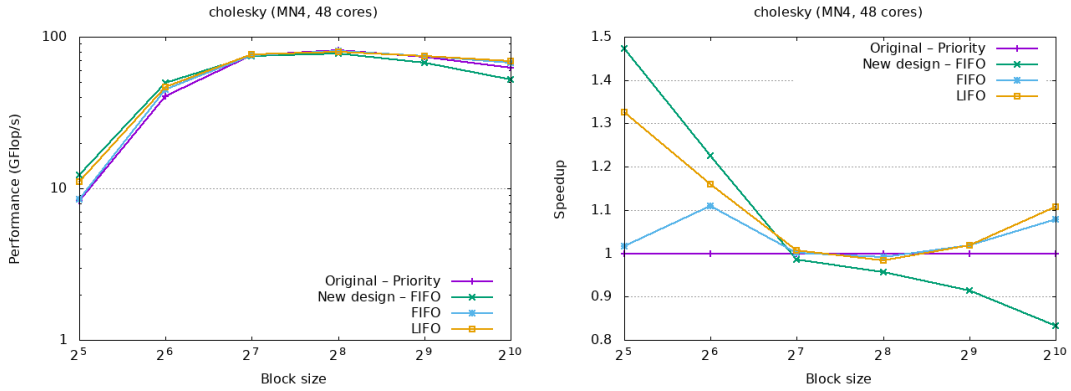
(a) (b)

Figure 4.4: Execution time for `multiaxpy-noop` with dynamic queue threshold

Figure 4.4 shows that introducing a dynamic queue threshold makes the execution with the LIFO scheduling policy much faster than the FIFO scheduling policy. But, in this case, both are slower than the initial implementation.

Nonetheless, even with this performance drop with respect to the first implementation, the performance with respect to the original scheduler is still significantly improved (30 times faster for small block sizes).

`cholesky`



(a) (b)

Figure 4.5: Execution time for `cholesky` with dynamic queue threshold

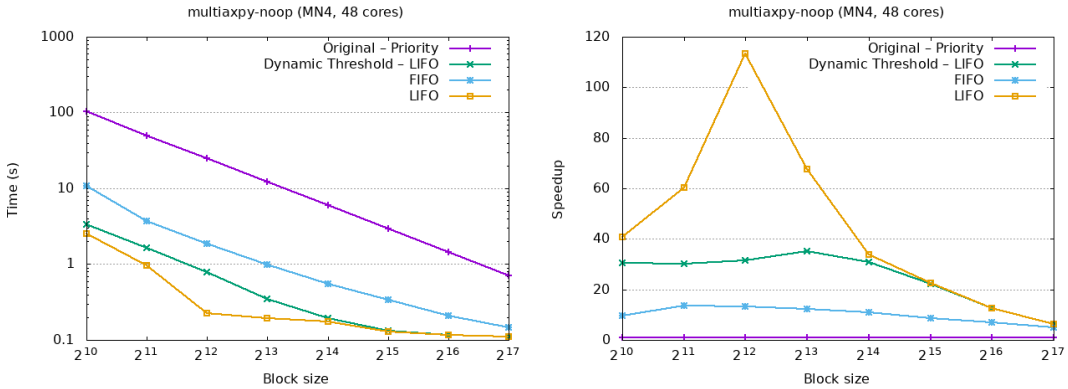
In the case of this benchmark, Figure 4.5 indicates that the dynamic queue threshold does indeed help with load balancing, as now there is no performance drop even for small block sizes.

Like in the case with the `multiaxpy-noop` benchmark, performance drops for very small block sizes with respect to the initial implementation, but it still provides a 30% improvement with the LIFO policy. And, for big block sizes, performance is up to 10% better than the original scheduler.

4.3.3 Load balancing

`multiaxpy-noop`

The introduction of a load balancing mechanism (Figure 4.6) introduces a significant speedup on the LIFO scheduling policy for very small block sizes with respect to only having a dynamic threshold. More importantly, execution time stays nearly constant up to a block size of 2^{12}, a dramatic improvement with respect to the original case. There is also a very slight improvement of the FIFO scheduling policy, but not as important as in the LIFO scheduling policy.



(a) (b)

Figure 4.6: Execution time for `multiaxpy-noop` with load balancing

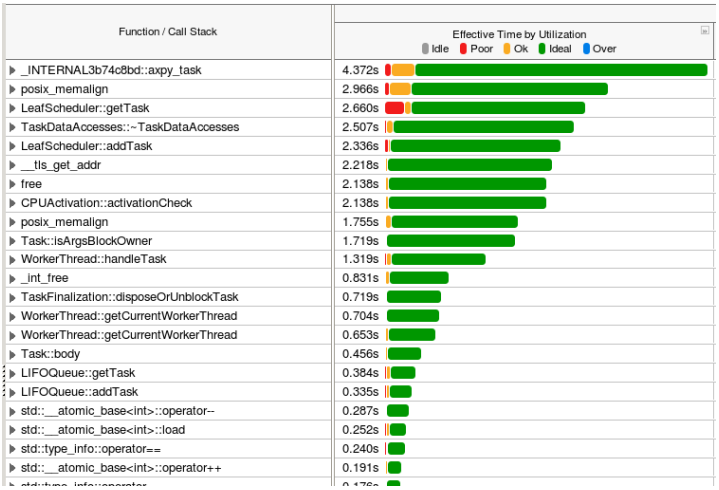


Figure 4.7: Profile for `multiaxpy-noop` for block size 1024

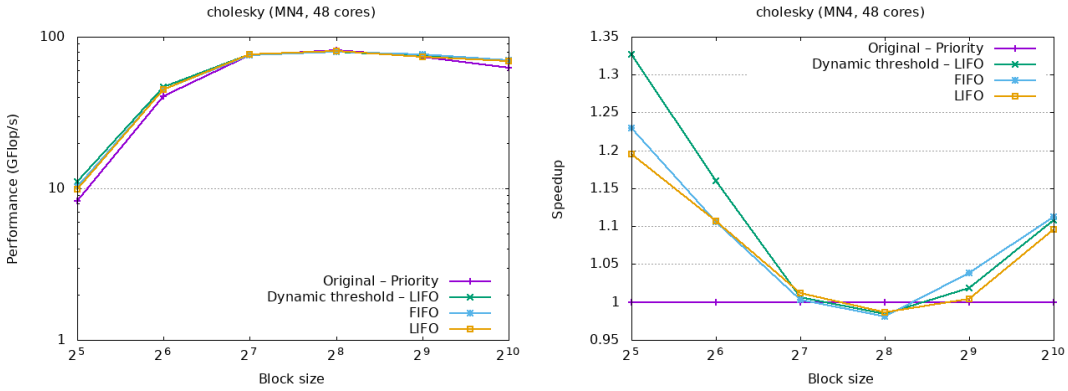
With this final version, it can be seen in Figure 4.7 that now the most time consuming section for the case with a very small block size is the task itself. This is a dramatic improvement over the original version.

Nonetheless, a significant amount of time is spent on the allocation and release of memory for tasks (`posix_memalign` and `free`). This may also be an area where improvements can be made.

`cholesky`

For this benchmark in particular, as seen in Figure 4.8, the introduction of a load balancing mechanism reduced slightly the performance of this new scheduler design, specially with small block sizes.

In this final version, profiling (Figure 4.9) shows that most of the time is now spent in



(a) (b)

Figure 4.8: Execution time for `cholesky` with load balancing

Function / Call Stack	Effective Time by Utilization (Idle Poor Ok Ideal Over)
_int_free	739.674s
cblas_dgemm	77.243s
Task::getParent	74.070s
LeafScheduler::getTask	39.462s
_INTERNAL1c151d45::DataAccessRegistration::processD	33.111s
DataAccessRegistration::DataAccessStatusEffects::DataA	19.568s
boost::intrusive::default_avltree_node_traits_impl<void*>:	19.218s
operator new	18.502s
boost::intrusive::bstree_algorithms<boost::intrusive::avltre	18.332s
posix_memalign	12.371s
boost::intrusive::default_avltree_node_traits_impl<void*>:	9.928s
nanos6_ol_16	8.960s
std::__atomic_base<int>::load	7.640s
SpinLock::lock	6.511s
std::__atomic_base<int>::operator--	6.271s
Task::getParent	5.545s
LAPACKE_dlarnv_work	5.300s
nanos6_ol_19	4.792s
boost::intrusive::default_avltree_node_traits_impl<void*>:	4.212s
_INTERNAL1c151d45::DataAccessRegistration::handleDa	4.197s
SpinLock::lock	4.106s
boost::intrusive::bstree_algorithms_base<boost::intrusive:	3.809s

Figure 4.9: Profile for `cholesky` for block size 32×32

releasing tasks' memory. This is also an area where improvements can be made.

5. Memory allocator

5.1 Introduction

Currently, Nanos6 does not have any mechanism of data structure reusal. When tasks are created, all its memory and data structures are allocated as explained in Section 1.3.1, and when they finish, their memory is freed.

As was seen in the profiles in Figures 3.2 and 3.4, and more significantly in Figures 4.7 and 4.9, calls to `posix_memalign` and `free` take a significant percentage of the execution time. Although Linux *libc* allocation functions try to manage memory very efficiently, it may be possible to improve the perfomance of Nanos6 in the area of task creation and destruction by implementing a memory allocator and data structure reusal system specifically designed for the needs of Nanos6.

5.2 New design

The design parameters for this memory management mechanism are very similar to those of the new scheduler: fast and scalable. Therefore, some of the design principles used in the scheduler may be used in this subsystem. In particular, this new memory management system is also distributed mimicking the underlying hardware architecture. This, again, allows for an implementation with less locks, as most of the time work can be done locally.

In order to make this component as fast as possible, two simplifications are made:

- It is only used in those data structures that are allocated and freed repeatedly inside the runtime. This is, it is only used for tasks and for dependences.
- All block sizes are rounded to a number of full cache lines.

The final design of this allocator is depicted in Figure 5.1. At the CPU level, it is composed of several stacks of memory chunks of the same size (multiples of the cache line size, as mentioned) per CPU. At the NUMA node level, it is composed of a pool of available memory aligned to the page size.

In order to reduce the memory required to store a local stack of chunks, and to reduce the number of data structures used, the links between nodes are stored in the memory chunks themselves.

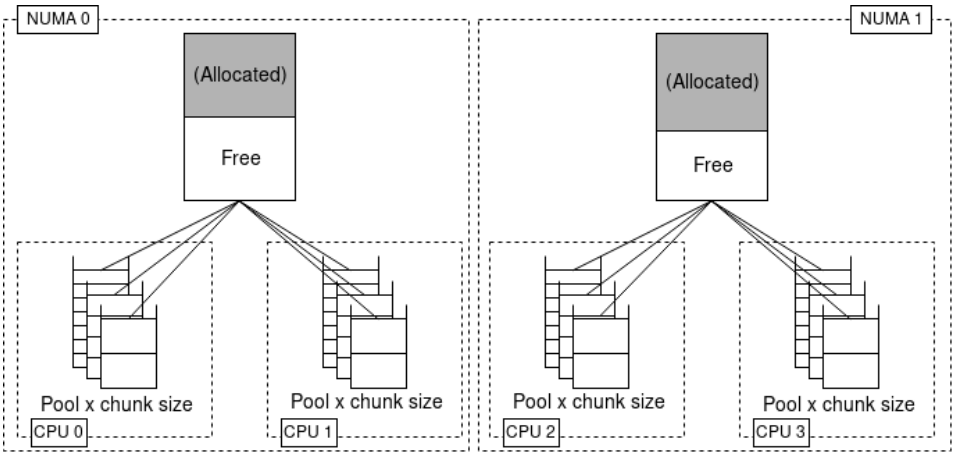


Figure 5.1: Conceptual diagram for the new memory allocator design

When a CPU requests memory, a chunk of the requested size (rounded to a number of cache lines) is obtained from its local chunk stack. If there are no chunks available (or the stack doesn't exist yet), the stack is filled from the pool of available memory at the NUMA node level. If there is no memory available at the NUMA node level pool, it is refilled by calling `posix_memalign`.

When a CPU frees memory, the chunk is pushed to the local stack of the corresponding size. Chunks are not returned to the higher level (NUMA node level) to reduce complexity and execution time.

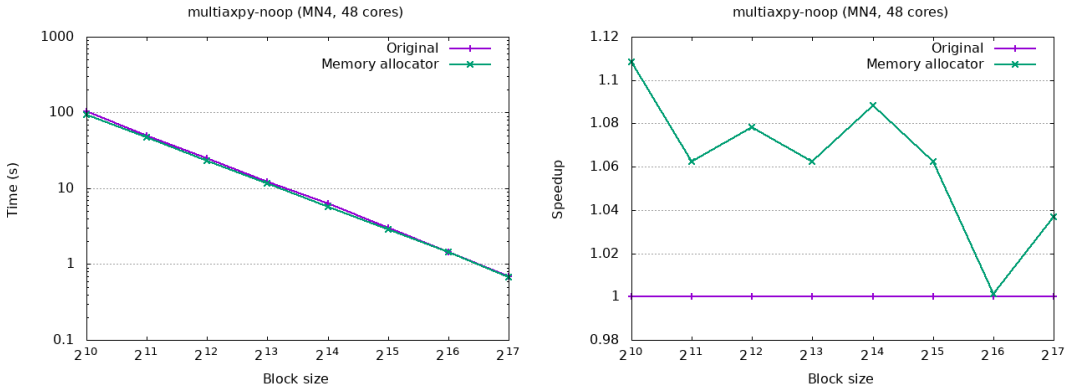
The allocation size for the NUMA level pool can be configured, but it is set to 8MB by default (2048 memory pages of 4KB). The size of the memory retrieved from the NUMA level pool when a local stack is empty can also be configured, but it is set to 128KB by default.

5.3 Performance results

5.3.1 Original scheduler

`multiaxpy-noop`

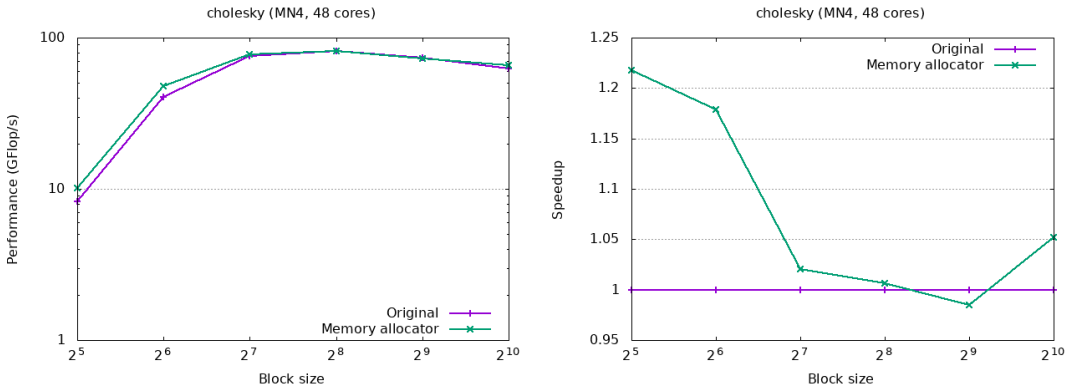
The results of this allocator combined with the original Nanos6 scheduler show a very small improvement for the application `multiaxpy-noop` (Figure 5.2). This is because of two reasons: on one hand, time spent in memory allocation and release relative to the whole execution is small. On the other hand, an improvement in memory allocation does not need to be directly correlated with an improvement in time, as faster task creation may contribute to even slower scheduling times.



(a) (b)

Figure 5.2: Execution time for `multiaxpy-noop` with the memory allocator

`cholesky`



(a) (b)

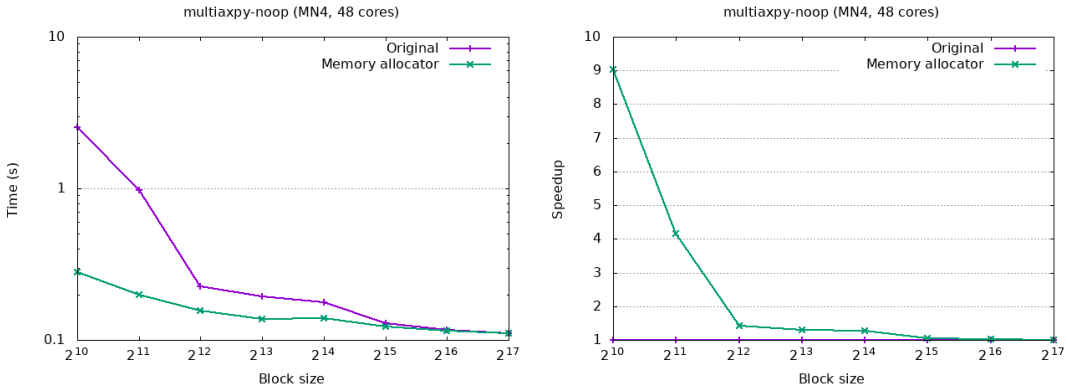
Figure 5.3: Execution time for `cholesky` with the memory allocator

For this application, as seen in Figure 5.3, speedup is slightly better than with the previous application with a very small block size. But for the rest of block sizes, the improvement is really small. The causes are similar to the ones discussed earlier.

5.3.2 New scheduler

`multiaxpy-noop`

In the case of the new scheduler designed in Chapter 4, the relative speedup when introducing the memory allocator is much more significant (Figure 5.4). This is because the percentage of execution time spent in memory allocation and release functions for this case is much bigger.

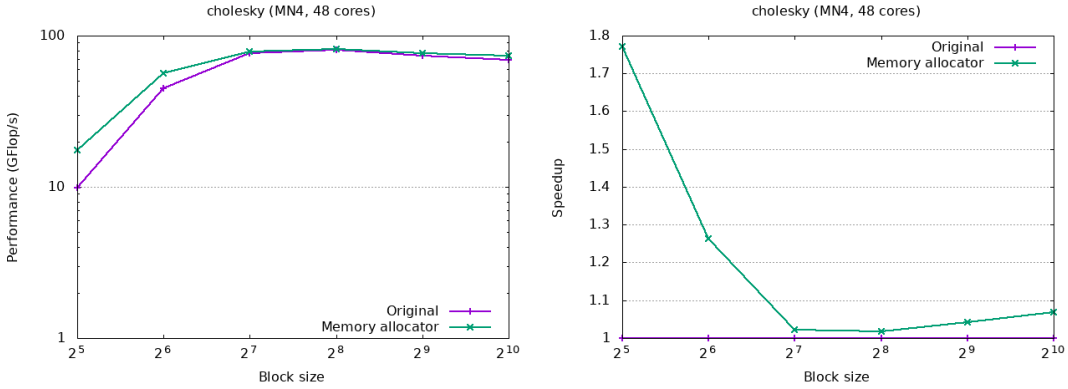


(a) (b)

Figure 5.4: Execution time for `multiaxpy-noop` with the memory allocator

The introduction of the memory allocator also makes the execution time of `multiaxpy-noop` much closer to a constant value independent of the number of tasks created. This, as mentioned before, means that task creation and scheduling in Nanos6 is close to a constant time, independent of the load of the system.

`cholesky`



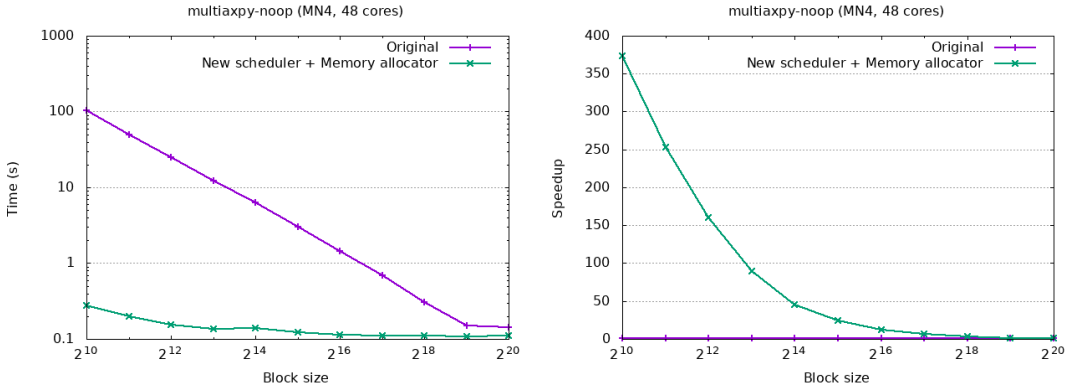
(a) (b)

Figure 5.5: Execution time for `cholesky` with the memory allocator

For `cholesky` (Figure 5.5), there is also a significant improvement when introducing the memory allocator. In this case, it is not as dramatic as in the previous application, due to the many more complexities of this application. But it is still a remarkable improvement.

6. Final performance evaluation

6.1 `multiaxpy-noop`



(a) (b)

Figure 6.1: Execution time for `multiaxpy-noop` with all improvements

As mentioned through this document, this application was chosen as a synthetic benchmark for extensive task creation. Final results in Figure 6.1 show that the optimizations designed and developed really increase task creation and scheduling scalability and concurrency, even for bigger block sizes not shown in previous results.

For Marenostrum 4, these results show more than a 350x improvement with very small sizes, and a smaller 50% improvement for the biggest block sizes tested. Results also show a very small increase in execution time related to the number of tasks created, varying between 0.1s and less than 0.3s, a significant improvement when compared with the 0.15s to 100s range of the original version.

A final profile of this application with the smallest block size, shown in Figure 6.2, indicates that most of the execution time is now spent in the task itself, followed by calls to the scheduler.

6.2 `cholesky`

A less synthethic application used throughout this document is the *Cholesky* factorization algorithm. Figure 6.3 shows the final results with all the improvements previously

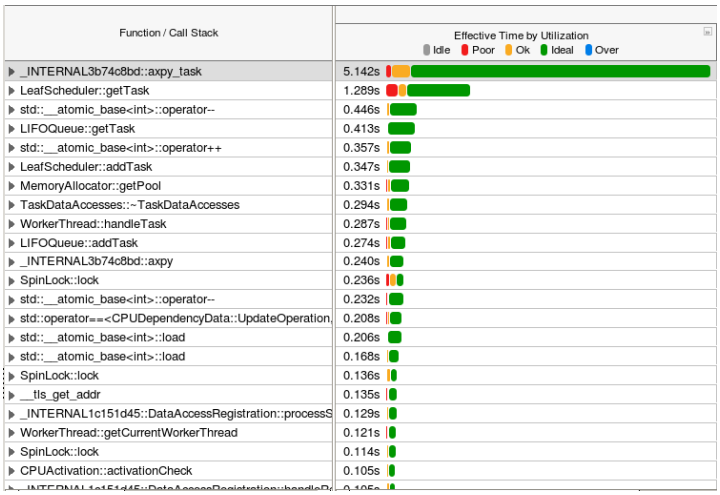
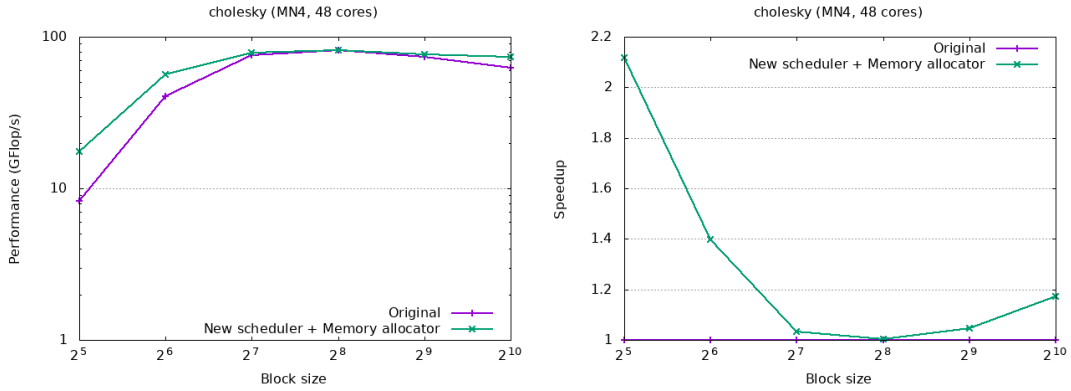


Figure 6.2: Profile for `multiaxpy-noop` with all improvements for block size 1024



(a) (b)

Figure 6.3: Execution time for `cholesky` with all improvements

described. For a small block size (32 × 32), performance has doubled. And for commonly used block sizes, such as 512 × 512 or 1024 × 1024, performance is up to 20% better.

These are still significant improvements, taking into account that the application has not been modified in any way, only the internal runtime has been modified.

Figure 6.4 shows a profile of this final version for the smallest block size tested. The highest percentage of the execution time is now spent in the task itself, followed by dependence management. This indicates that the next bottleneck that could be studied in Nanos6 is the dependence subsystem.

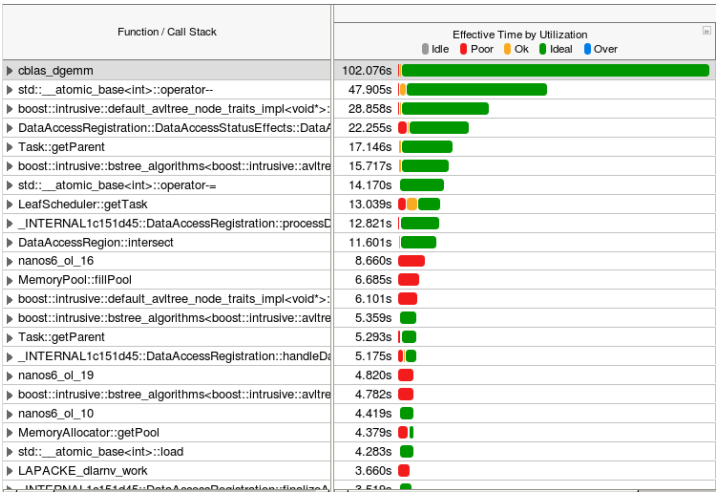
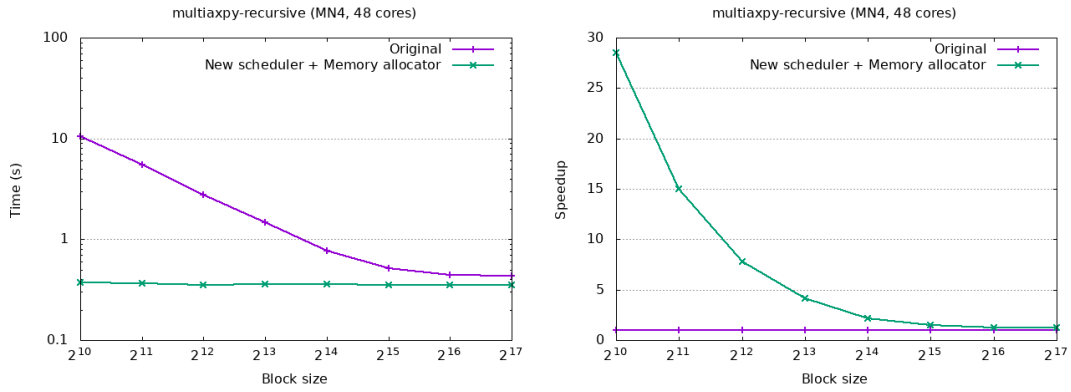


Figure 6.4: Profile for `cholesky` with all improvements for block size 32×32

6.3 *multiaxpy*

As mentioned in appendix A.1, the benchmark `multiaxpy-noop` is based on performing an *axpy* between two vectors by splitting them into smaller chunks recursively. Two versions of this original code have also been tested against these improvements.

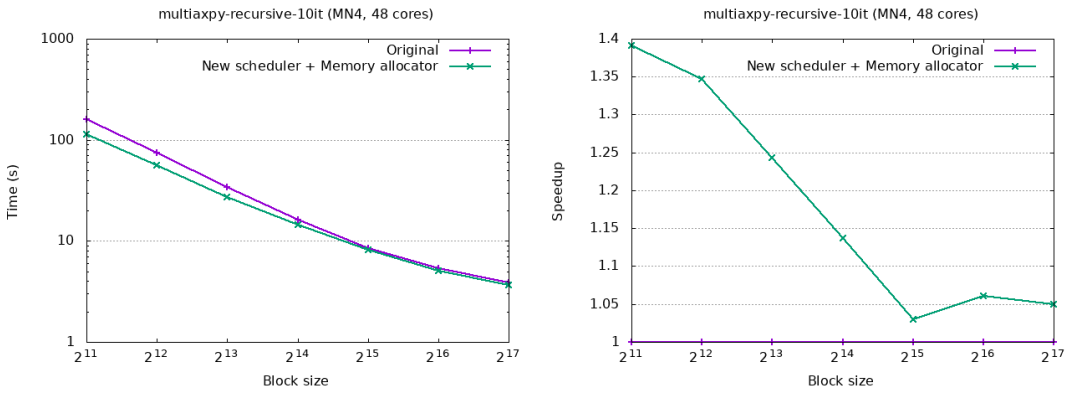


(a) (b)

Figure 6.5: Execution time for `multiaxpy-recursive` with all improvements

The first one, seen in Figure 6.5, is a simple *axpy*, without any dependence (and, therefore, only one iteration). It behaves very similarly to the `multiaxpy-noop` application shown before, and it also benefits from this scheduler, as its execution time becomes quasi constant.

The second version, in contrast, runs the *axpy* 10 times, with dependences between iterations. In this case (Figure 6.6), the improvements in Nanos6 only obtain up to 40% speedup with very small block sizes, and also a very poor absolute execution time. This is because of dependences: as there are two vectors with 1G elements each, with



(a) (b)

Figure 6.6: Execution time for `multiaxpy-recursive-10it` with all improvements

very small block sizes there will be a lot of small dependence regions, and dependence registration and checking will take more time.

6.4 Other applications

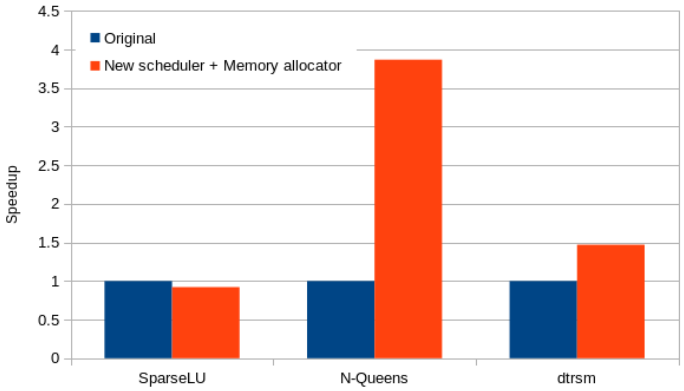


Figure 6.7: Speedup for several applications with all improvements

Other applications have been also tested with this new scheduler and memory allocator, as seen in Figure 6.7. Like with the rest of results, some applications show some big improvements, like *N-Queens*, and others, like *SparseLU* show a slight slowdown, of less than 10%.

7. Conclusions

This project has consisted in the analysis of the performance of the Nanos6 runtime, and its improvement in the areas of task creation and scheduling. In specific, a new scheduler and a new memory allocator have been designed, implemented and tested. These new components have been designed with the purpose of being fast, scalable, and also flexible for the future needs of the runtime.

As seen in Chapter 6, results are very satisfactory, achieving very good performance in applications bounded by task creation and scheduling time, while keeping the same performance for other applications with other requirements.

These new features will allow some OmpSs-2 developers to run their applications faster without the need to modify their applications, and will allow Nanos6 to obtain good performance in a wider range of applications, specially those with very fine grained tasks with very few dependences.

7.1 Future Work

Even if good results have been achieved, it is still necessary to test both the new scheduler and the memory allocator against a wider range of applications to ensure that they behave as expected in all of them.

It would also be possible to improve even more the results of this new scheduler design by taking into account data distribution of the tasks, expressed by their dependences. With this information it could be possible to better select which node of the scheduler hierarchy is the most suitable to run a specific task.

Support for heterogeneous architectures in Nanos6 is currently under development. This scheduler design can also be extended for these architectures just by using a leaf scheduler on each of the resources available, and making them aware of which type of tasks they can run.

Finally, task creation and scheduling are not the only points of the runtime that introduce significant overhead in applications. Subsystems like dependences need to be studied carefully in order to obtain the best performance possible, while keeping all the range of features. This is a continuous improvement work that is as important as introducing new useful features to OmpSs-2.

Bibliography

[1] Barcelona Supercomputing Center. Marenostrum IV. `https://www.bsc.es/marenostrum/marenostrum/technical-information`. Accessed: 2018-06-02.

[2] OpenMP. OpenMP. `https://www.openmp.org/`. Accessed: 2018-06-02.

[3] MPI Forum. MPI. `https://www.mpi-forum.org/`. Accessed: 2018-06-02.

[4] Programming Models @ Barcelona Supercomputing Center. The OmpSs programming model. `https://pm.bsc.es/ompss`. Accessed: 2018-06-02.

[5] Barcelona Supercomputing Center. Barcelona Supercomputing Center. `https://www.bsc.es/`. Accessed: 2018-06-02.

[6] Programming Models @ Barcelona Supercomputing Center. The OmpSs-2 programming model. `https://pm.bsc.es/ompss-2`. Accessed: 2018-06-02.

[7] Programming Models @ Barcelona Supercomputing Center. Mercurium. `https://pm.bsc.es/mcxx`. Accessed: 2018-06-02.

[8] Programming Models @ Barcelona Supercomputing Center. Nanos6. `https://github.com/bsc-pm/nanos6`. Accessed: 2018-06-02.

[9] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. OpenMP task scheduling strategies for multicore NUMA systems. *The International Journal of High Performance Computing Applications*, 26(2):110–124, 2012.

[10] Samuel Thibault. A flexible thread scheduler for hierarchical multiprocessor machines. *arXiv preprint cs/0506097*, 2005.

[11] Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Building portable thread schedulers for hierarchical multiprocessors: The bubblesched framework. In *European Conference on Parallel Processing*, pages 42–51. Springer, 2007.

[12] Sivarama P Dandamudi and SP Cheng. Performance impact of run queue organization and synchronization on large-scale NUMA multiprocessor systems. *Journal of systems architecture*, 43(6-7):491–511, 1997.

A. Applications

A.1 `multiaxpy-noop`

Originally, this application performed a vector *axpy* ($a * x + y$) by means of recursively splitting the vector until a minimum block size is obtained, which is then calculated sequentially. This can be seen in Figure A.1.

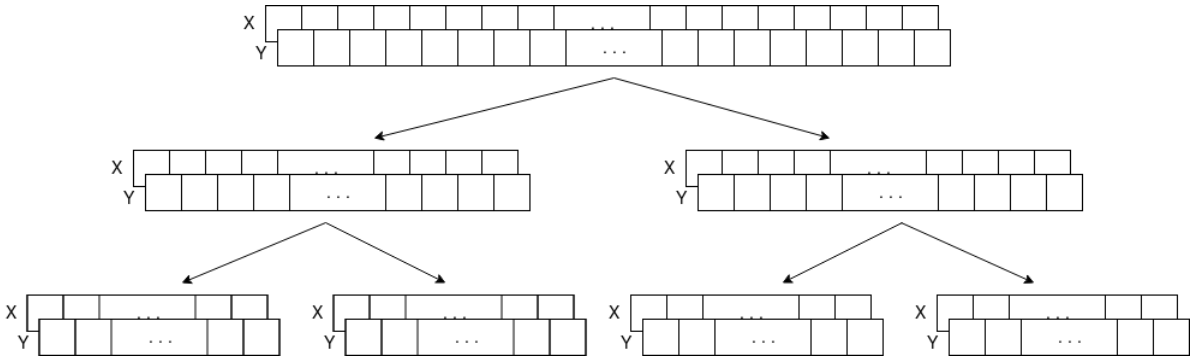


Figure A.1: Multiaxpy vector splitting pattern

The implementation used to test the performance of Nanos6 is a modified version of this code where the *axpy* is executed several times iteratively, there are no dependences between tasks, and the innermost task (which would be the responsible of calculating the vector *axpy* sequentially) does not really access or modify any data, it just spins for the same number of iterations as the vector size.

This implementation is focused on stressing the creation and scheduling of tasks in Nanos6, as tasks will be created recursively from all CPUs at the same time. For example, running this application for a vector size of 1G elements, a block size of 1024 and 10 iterations, will create around 2M tasks per iteration as fast as it can. The number of tasks created for a vector size of 1G and 10 iterations for different block sizes can be seen in Table A.1

A.2 `cholesky`

This application is a traditional implementation of an iterative *Cholesky* factorization. Therefore, this application creates tasks from only one thread, with dependences between them. Furthermore, in cases where the block size is big with relation to the matrix size,

Block size	Tasks created
1024	20971511
2048	10485751
4096	5242871
8192	2621431
16384	1310711
32768	655351
65536	327671
131072	163831

Table A.1: Number of tasks created in the `multiaxpy-noop` application

the number of tasks created will be small. Table A.2 shows the number of tasks generated for a matrix of 16384 × 16384 elements, for different block sizes.

Block size	Tasks created
64 × 64	2829057
128 × 128	357761
256 × 256	45761
512 × 512	5985
1024 × 1024	817
2048 × 2048	121
4096 × 4096	21
8192 × 8192	5

Table A.2: Number of tasks created in the `cholesky` application