Advanced Synchronization Techniques for Task-based Runtime Systems

David Álvarez
Barcelona Supercomputing Center
Barcelona, Spain
david.alvarez@bsc.es

Kevin Sala
Barcelona Supercomputing Center
Barcelona, Spain
kevin.sala@bsc.es

Marcos Maroñas
Barcelona Supercomputing Center
Barcelona, Spain
marcos.maronasbravo@bsc.es

Aleix Roca
Barcelona Supercomputing Center
Barcelona, Spain
arocanon@bsc.es

Vincenç Beltran
Barcelona Supercomputing Center
Barcelona, Spain
vbeltran@bsc.es

Abstract

Task-based programming models like OmpSs-2 and OpenMP provide a flexible data-flow execution model to exploit dynamic, irregular and nested parallelism. Providing an efficient implementation that scales well with small granularity tasks remains a challenge, and bottlenecks can manifest in several runtime components. In this paper, we analyze the limiting factors in the scalability of a task-based runtime system and propose individual solutions for each of the challenges, including a wait-free dependency system and a novel scalable scheduler design based on delegation. We evaluate how the optimizations impact the overall performance of the runtime, both individually and in combination. We also compare the resulting runtime against state of the art OpenMP implementations, showing equivalent or better performance, especially for fine-grained tasks.

CCS Concepts: • **Computing methodologies** → **Parallel programming languages**.

Keywords: parallel programming models, OmpSs-2, OpenMP, task-based runtimes, data dependencies, lock-free, wait-free

1 Introduction

Due to diminishing returns on modern CPUs' single-thread performance, the industry has shifted towards many-core and heterogeneous architectures [3, 11, 36]. The recent focus on energy efficiency has increased the interest in systems with numerous processing elements with lower frequencies. Those parallel systems can achieve huge performance figures, but their limiting factor is the scalability of the software.

One of the most widely used standards for programming shared-memory systems in both industry and academy is OpenMP[6]. OpenMP initially had only a *fork-join* execution model, where programmers explicitly define parallel regions. The fork-join model is an efficient way to exploit well-structured parallelism, but it is not well suited to exploit irregular, dynamic, or nested parallelism. In recent years, task-based parallelism has been introduced in OpenMP to overcome these limitations.

The task-based paradigm can exploit more fine-grained, dynamic, and irregular parallelism than the fork-join model. Additionally, it minimizes the need for global synchronization points, and it naturally copes with load-imbalance. These features make the paradigm especially promising to exploit modern many-core architectures [10, 22].

Moreover, the introduction of task data dependencies was the critical element to truly move forward to a *data-flow* execution model that relies on fine-grained synchronizations between tasks. This model reduces further the need for global synchronization points and allows the runtime to exploit data-locality between tasks. However, task management inside the runtime might incur some non-negligible overhead, especially for fine-grained tasks on large many-core systems.

This paper presents optimized designs for the main components of a task-based runtime system. We also present a lightweight and integrated instrumentation system, which we used to analyze in detail how the scalability of each component affects the global scalability of the runtime.

A task-based runtime system has three main components: the dependency system, the task scheduler, and the memory allocator, which tightly interact with each other. The first stage of a task's life cycle is its creation, which involves the memory allocator. The runtime then checks its data dependencies to determine if the task is ready or blocked based on the previous tasks' dependencies. Once all its dependencies are satisfied, the task becomes ready and is added to the

PPoPP '21, February 27-March 3, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8294-6/21/02…$15.00
https://doi.org/10.1145/3437801.3441601

scheduler, which will eventually schedule it on an available core. Once the task has executed, it releases its dependencies so that its successor tasks may become ready. Note that the three components require a synchronization mechanism as they have to deal with multiple requests simultaneously. Thus, the application developer has to strike a balance in task granularity: it has to be small enough to provide sufficient work for all available cores while being coarse enough to evade runtime system overheads [15]. However, as applications scale out to more cores (or nodes) and the problem size remains constant, task granularities naturally decrease. At some point in the scaling process, tasks can become so small that the application is overhead-bound, and the scalability depends on the ability of the parallel runtime to handle small tasks.

Our contributions are to (1) present a novel wait-free data structure and algorithm to support complex dependency models in a task-based runtime; (2) provide a scalable task scheduler that works well under high contention and uses delegation instead of work-stealing; and (3) analyze and create a detailed performance profile of the task-based runtime with a lightweight instrumentation framework.

2 Data dependency system

The data dependency system is one of the limiting factors in the scalability of task-based runtimes, especially when running programs with very fine-grained tasks. Such programs register large amounts of small tasks with dependencies that take a short time to execute. Thus the overhead in the dependency system can significantly impact the overall application performance.

In this paper, we apply our optimizations to the Nanos6 runtime [4] for the OmpSs-2 programming model. Compared to OpenMP, the model for data dependencies in OmpSs-2 is more complex [5, 13, 28]. The main complexity increase is because the dependency domains of tasks on different nesting levels can share dependencies, which complicates the locking scheme used to implement the model. Reductions also are treated as data dependencies on OmpSs-2 tasks, unlike OpenMP where they are defined at a task group level.

2.1 Dependencies in Nanos6

In Nanos6, a task is a *sibling* of another when they are at the same nesting level. Tasks declared inside another one (nested) are considered *child* tasks.

The dependencies of a task in Nanos6 are represented as accesses, which are composed of a memory address and an access type, e.g., read or write. Two accesses have a *successor* relation when they share the same memory address and their tasks have a *sibling* link. Similarly, two accesses have a *child* relation if they share the same memory address and their tasks have a *child* link. Task access relations on OmpSs-2 programs form binary trees between the linked tasks, as shown in Figure 1. Note that on OpenMP users cannot express dependencies crossing nesting levels, and the *child* relationship is not considered to determine the dependencies between tasks.

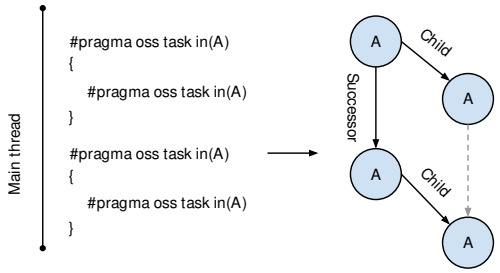

Figure 1. The graph of task dependency accesses on an OmpSs-2 program. The program (left) results in the dependency graph (right) between the accesses to location A.

During an OmpSs-2 program, several tasks can be created and finished concurrently. They have to propagate dependency information through the data structures to determine if the added tasks are ready and if the recently finished tasks allow any successor to become ready.

2.2 Wait-free data dependencies

The previous implementation of dependencies inside Nanos6 was based on fine-grained locking, but it was very complex to avoid possible deadlocks. Instead of protecting the data structures that hold the information about the dependencies through mutual exclusion, our alternative is to adapt some wait-free programming concepts to create a data structure capable of supporting the concurrency we need. Otherwise, if we had decided to build a data structure based on mutual exclusion, we would have to compromise either with the complexity of fine-grained locking or the performance degradation of coarse-grained locking. The main goal is not to have wait-freedom as a requirement but to provide fast and scalable dependency registration.

In this section, we describe the concept behind the dependency implementation we propose for the Nanos6 runtime. In the following section, we will formalize the approach and prove its wait-freedom property.

When a program creates a task, all the dependencies are registered inside the Nanos6 runtime using the *DataAccess* structure, shown in Listing 1. The *Task* structure stores all task-related information, including a pointer to the array of its accesses. Each access has an atomic *flags* field that stores its current state, indicating if the dependency is currently satisfied (not preventing the task execution) and whether the satisfiability information has propagated to its successor and child accesses. The access also stores a pointer to its *successor*, which is the next access (belonging to a successor task) to the same address in the current nesting level, and a

pointer to the *child*, which points to the first access to the same address that belongs to a child task.

The flags field represents a Finite State Machine in which the state diagram has no cycles, so there are starting and final states. Since we only modify this data structure with atomic operations, we have named it *Atomic State Machine* or ASM. Note that there is one instance of this state machine for each access.

```
struct Task {
  ...
  DataAccess *dataAccesses;
};

struct DataAccess {
  void *address;
  std::atomic<access_flags_t> flags;
  DataAccessType type;
  DataAccess *successor, *child;
};
```

Listing 1. Relevant fields in the Task and DataAccess structures

The only way for an ASM to transition from one state to another is through receiving a data access message. The structure of a message, shown in Listing 2, contains two flags fields. One field contains the flags to set in the target access. The other has flags that have to be set on the message's originator as a delivery notification. The ASM's transitions have to be done as a single atomic operation, optimally through a *fetch&or*. Based on the values before and after the transition, the ASM may generate additional messages to deliver to its child or successor accesses. All the messages that are still pending to deliver are stored in a simple per-thread queue called MailBox. We illustrate this process in Figure 2.

```
struct DataAccessMessage {
  access_flags_t flagsForNext, flagsAfterPropagation;
  DataAccess *from, *to;
};
```

Listing 2. DataAccessMessage structure

Figure 2 represents the basic structure of the algorithm. While the *MailBox* has undelivered messages, we pop one from the container and *deliver* it to the destination access. Upon receiving the message, the access atomically updates its *flags* field. The atomic update provides us with the flags' exact values before and after the message was received. As flags cannot be unset, we know each transition happens only once in the access lifetime. With this information, we decide if it is needed to generate more messages (to propagate information about satisfied accesses, for example). Finally, we atomically update the *flags* field of the originator access of the message to notify the delivery of the information. We use this last atomic update to determine we can safely delete an access.

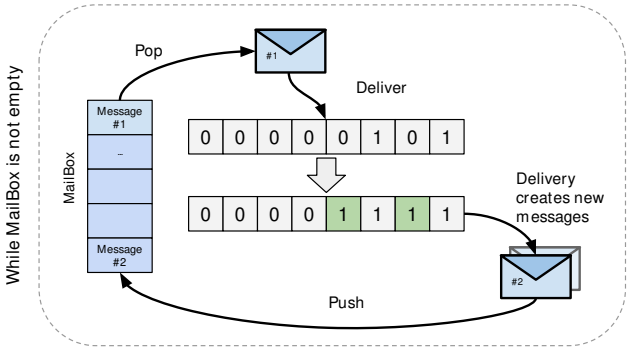

Figure 2. Atomic state machine dependency propagation

2.3 Formalization

In this section, we introduce several relevant definitions and finally prove wait-freedom.

Definition 2.1. *Access Flags.* We can define the set of all possible flags an access can have as set F. Then, the set F_a of flags that an access a has is defined as $F_a \subseteq F$. When a is created, flags are initialized as $F_a = \emptyset$.

Definition 2.2. *Delivery.* The only operation that can be done on the flags F_a of an access a is the delivery of a message $M \subseteq F$. A delivery operation is defined as:

$$F_{a,i+1} = M \cup F_{a,i}$$

Assuming the message M follows the two restrictions:

$$M \cap F_{a,i} = \emptyset$$

$$M \neq \emptyset$$

The restrictions on the content of the messages are part of what provides the wait-freedom assurance. Informally, bits in the flag field can only be set, and the field size is limited. Additionally, each message that an access receives has to contain at least one flag, and none of the flags can be already set in the access. By those properties, we can deduce that each access can receive only a limited number of messages, which in the worst case is $|F|$.

Lemma 2.3. *The delivery of a message M to an access a is non-blocking and wait-free.*

Proof. To establish wait-freedom, we need to prove that there is a bound on the time a delivery operation can take regardless of any other threads. Assuming that the operation is performed using a *CAS* primitive, we can assume constant time for the *CAS*, but it can fail in case of conflict with another thread. Hence, to prove wait-freedom, we need to bound the number of failures due to conflict a delivery operation can suffer.

A *CAS* can fail if and only if another thread modified the memory location during the delivery operation. However, in our system, the only way to change the flags F_a of an

access a is to deliver another message. As we established our restriction that a message cannot be empty, the maximum number of messages an access a can receive if each message only contained one flag would be:

$$M_{max} = \{\{x\} \mid x \in F\}$$

Then, the maximum number of *CAS* operations that have to be done until one succeeds, assuming a worst case scenario, is $|M_{max}|$, which trivially $|M_{max}| = |F|$ .We can establish that the maximum number of tries to deliver a message M is $T_m \leq |F|$. It is possible to get a closer bound on the retries if we consider the number of flags in the message to be delivered, but this is sufficient for us to prove Lemma 2.3. The time needed to deliver a message is clearly bounded to a constant number of *CAS* operations. □

Definition 2.4. *Unregister.* For a *Task* t that has a set of accesses A_t, the unregister operation on a Task is defined as delivering a specific message M to each access a so that $a \in A_t$.

A task is unregistered once it finishes its execution, and the message delivered to the access indicates this condition. An unregister operation will thus do $|A_t|$ delivery operations. As we have proved that a delivery is wait-free, the unregister operation will be wait-free because it does a finite and known number of delivery operations.

3 Task Scheduling System

The scheduling system orchestrates the execution of ready tasks on worker threads. Throughout this section, we assume that exactly one worker thread is bound to each CPU for simplicity but without loss of generality. When a task becomes ready, it is forwarded to the scheduling system. Then, when a core becomes idle, it calls the scheduler to ask for more work. If there are ready tasks, the scheduling system will determine the best task that can be executed on this specific core. It is worth noting that multiple ready tasks can be added to the scheduler concurrently and that several worker threads can simultaneously call the scheduler. Thus, it becomes mandatory to add some kind of synchronization on the scheduler to prevent data-races.

Most task-based runtime systems rely on multiple ready task queues combined with work-stealing to mitigate the above-mentioned problems. However, on the typical application design pattern in which a single thread creates all tasks, work-stealing behaves similarly to the global lock approach because most threads need to steal work from a single creator queue. In contrast, the approach described in this section adapts the global lock concept to handle both single creator and multiple creator cases efficiently.

Usi a global lock is the most straightforward approach to synchronize the scheduler. In this case, the lock is acquired to add ready tasks to the scheduler and to schedule tasks to worker threads. When task granularity is coarse enough, this approach works well and keeps the scheduling system's design simple and the scheduling policies accurate.

However, when task granularity is fine-grained and the system has many cores, the core that is creating new tasks might not be fast enough to feed all other cores. In this scenario, many worker threads will busy-wait on the global lock. This has two adverse effects. Firstly, it increases contention on the cache subsystem due to the additional cache coherence traffic. Secondly, it prevents ready tasks from entering the scheduler fast enough because the task creator has to fight with all of the worker threads to get the global lock.

A well-known technique to mitigate lock contention on the scheduler is to let the worker threads spin for a while, and if they do not get any ready task, block its thread using a mutex until a ready task becomes available. We avoid using this approach because it adds extra work to the thread that is creating tasks. When ready tasks are added to the scheduler, it has to check if there are blocked threads and wake them up with an expensive system call.

3.1 Optimizing task insertion

To avoid the stagnation of ready tasks in the scheduler, we have decoupled the actions of adding and scheduling ready tasks. When a task becomes ready, we do not directly add it to the scheduler but a bounded wait-free single-consumer single-producer (SPSC) queue working as a buffer.

The number of SPSC queues can be configured from a single one to one per core. In the first case, we would need a lock to synchronize all task additions, while in the latter, no locking is needed at all. We use the lock to synchronize between producers, but the synchronization between producer and consumers remains wait-free. In our experiments, we use one SPSC queue and lock per NUMA node.

When a worker threads enters the scheduler, it first drains all SPSC queues and inserts the ready tasks into the global ready queue. With this approach, we ensure that any contention generated by many worker threads calling the scheduler does not affect the performance of cores that are creating tasks. We can implement this optimization because the actual addition of tasks can be safely delayed until a core becomes idle and calls the scheduler. Notice that this delegation technique is compatible with any lock implementation.

3.2 Scalable lock designs

The scheduler system has to be extensible, and adding new scheduling policies should be easy. We have discarded a wait-free or lock-free scheduler because of its complexity and difficulty to maintain, as each scheduling policy would require a new ad-hoc design and implementation.

Our scheduling system relies on a global lock to protect its internal data structures, making it easy to develop new scheduling policies. Ticket Locks [31] are fair and provide strict FIFO ordering, but they have contention problems under high-load conditions, so they are not suitable for our

centralized scheduler. Partitioned Ticket Locks [8] (PTLocks) extend Ticket Locks with a padded array used to do busy-waiting by idle threads. If the size of this array is equal to the number of CPUs, then each core will busy-wait in a different array slot, reducing the cache coherence traffic to the minimum. We use PTLocks as a building block of our optimized lock design presented in the next section.

```
1  struct PTLock {
2    // Can be a constructor parameter
3    const static int Size = 64;
4    std::atomic<uint64_t> _head = {Size};
5    uint64_t _tail = {Size + 1};
6    std::atomic<uint64_t> _waitq[Size] = {{Size}};
7
8    uint64_t _getTicket() {
9      return _head.fetch_add(1);
10   }
11   void _waitTurn(uint64_t ticket) {
12     while (_waitq[ticket % Size] < ticket) { spin(); }
13   }
14   void lock() {
15     _waitTurn(_getTicket());
16   }
17   void unlock() {
18     uint64_t idx = _tail % Size;
19     _waitq[idx] = _tail++;
20   }
21 };
```

Listing 3. Implementation of a PartitionedTicketLock

Listing 3 shows the implementation of a Partitioned Ticket Lock. For the sake of clarity, we have omitted the padding of the fields to prevent false sharing, and the memory order constraints of all atomic operations. The *_waitq* (line 6) is an array of unsigned 64-bit integers used to implement a circular buffer representing an infinite virtual waiting queue. The *_head* and *_tail* fields (lines 4 and 5) are used to index the *_waitq* array. The *_head* represents the index of the latest slot in the virtual waiting queue and the *_tail* is the index of the next slot that will be able to acquire the lock. When the lock is free and no thread is waiting to acquire it, *_tail == _head+1*.

We initialize the lock such that *_waitq[_head%Size] == _head*, guaranteeing that the first thread that arrives will be able to acquire it. The *lock()* operation (line 14) consists of just two calls. The first one is *_getTicket()* (line 8), which performs an atomic fetch and increment of the *_head* field to obtain the last ticket (line 9). The second call is *_waitTurn* (line 11) that receives the ticket as a parameter. The current thread busy-waits on the *_waitq[ticket % Size]* position until it matches (or exceeds) the ticket value. The *unlock* operation (line 17) calculates the next slot index that will be able to acquire the lock. Then it increments *_tail* and writes *_tail-1* in the computed slot to release the lock.

PTLocks perform as well as more complex designs such as MCS [25] or Ticket Locks Augmented with a Waiting Array (TWA) [9], however it requires more memory space.

3.3 Delegation Ticket Lock (DTLock)

Another well-known technique to improve the performance of data structures that are not amenable to fine-grained locking is delegation [32]. The main idea behind this method is that protected data structures are accessed only by one privileged thread, which executes all the operations on behalf of the other threads. Delegation relies on a lightweight and optimized communication protocol between the privileged thread and the rest of the threads to be able to delegate operations and forward results back. A drawback of this approach is that it requires a dedicated core for each independent set of data structures that has to be protected, making it unpractical in many situations.

There are variants of the delegation technique that use queues to delegate work to the threads currently inside the lock [21]. These are better suited to use in centralized schedulers as they do not require dedicated cores for each lock. In order to build our scalable centralized scheduler, we have developed a novel Delegation Ticket Lock (DTLock) that builds on state of the art delegation techniques and extends our implementation of the PTLock with support for fine-grained and dynamic delegation of operations. The DTLock supports the standard *lock*, *unlock* and *trylock* operations, as well as *lockOrDelegate*, *empty*, *front*, *popFront* and *setItem*. The *lockOrDelegate* operation either acquires the lock if it is free or delegates the operation to the current lock's owner. However, it is the current owner that will decide if it executes or not the delegated operations. Suppose the current owner releases the lock without performing a pending delegated operation. In that case, the thread that originally delegated that operation will eventually acquire the lock and execute it by itself. The *empty*, *front*, *popFront* and *setItem* operations can only be called by the thread that owns the lock and are used to manage the threads that are waiting outside the lock.

The main advantages of the DTLock are that it does not require a dedicated core and its additional operations can be freely combined with traditional *lock*, *unlock* and *trylock* operations. Additionally, the DTLock allows for threads to remain inside the critical section of the lock executing delegated operations. This can be leveraged to minimize operation latency when there is not enough work to keep all cores busy.

Listing 4 shows the implementation of a DTLock in C++, which inherits the PTLock's *_head*, *_tail* and *_waitq[]* members, as well as, *lock*, *unlock* and *tryLock* operations. The DTLock extends the PTLock with two additional arrays. The *_logq* array (line 3) is used to register waiting threads while the *_readyq* array (line 4) is used to store the result of delegated operations, i.e. ready tasks in our case.

The first parameter of the *lockOrDelegate* operation is a unique *id* that identifies the thread that is calling this function. This *id* should be in the range *0..Size-1* as it is used to index the *_readyq* array. Thus, we need to know in advance

```
1  template <typename T>
2  struct DTLock : public PTLock {
3    std::atomic<uint64_t> _logq[Size] = {};
4    struct { uint64_t ticket; T item; } _readyq[Size];
5
6    bool lockOrDelegate(uint64_t const id, T &item) {
7      uint64_t const ticket = _getTicket();
8      _logq[ticket % Size] = ticket + id;
9      _waitTurn(ticket);
10     if (_readyq[id].ticket != ticket) {
11       _tail++;
12       return true;
13     }
14     item = _readyq[id].item;
15     return false;
16   }
17   bool empty() const {
18     return _logq[_tail % Size] < _tail;
19   }
20   uint64_t front() {
21     return _logq[_tail % Size] - _tail;
22   }
23   void popFront() {
24     unlock();
25   }
26   void setItem(uint64_t const id, T item) {
27     _readyq[id].item   = item;
28     _readyq[id].ticket = _tail;
29   }
30 };
```

Listing 4. Implementation of a Delegation Ticket Lock

the maximum number of threads that can call the DTLock. If the *lockOrDelegate* operation is finally delegated, the second parameter is used to store the result.

The first thing done in *lockOrDelegate* is to obtain a ticket (line 7). Then, the thread is registered on the *_logq* (line 8) array with just one store operation that combines the ticket and calling thread's id. The values written on the *_logq* array cannot be overrun because it is guaranteed that there will be at most *Size* threads calling the *lockOrDelegate* operation, so that each thread will have their own position. Once the thread has been registered, it just busy-waits (line 9) until it acquires the lock (lines 11-12), or the operation has been delegated and the result is stored in the *&item* parameter (line 14).

The *empty* operation (line 17) returns *true* if there is no thread registered in the *_logq* array and *false* otherwise. We check the first position of the *_logq* array, and if the value is smaller than *_tail*, we know there is no thread waiting yet. Otherwise we would see the value written in line 8. Notice that this operation is intrinsically racy but harmless.

If a call to *empty* returns *false*, then the owner of the lock can call *front* (line 20) to obtain the id of the thread that is waiting. To that end, we only need to do the inverse of the operation done on line 8, subtracting the *_tail*'s value to obtain the thread id (line 21).

Then, the *setItem* operation assigns a result *T* to a registered thread using its id to index the *_readyq* array. First, it sets the *item* field (line 27) and then the *ticket* to the *_tail* value, marking the entry as valid. We use the *ticket* field in line 10 to determine if an operation was delegated or not. Finally, the *popFront* operation wakes up the first thread busy-waiting on the *_waitq* by executing the *unlock* operation.

3.4 Synchronized Scheduler

This section presents a synchronized scheduler that leverages the SPSC queues and the DTLock described in sections 3.1 and 3.3, respectively.

```
1  struct SyncScheduler {
2    DTLock<Task *> _lock;
3    UnsyncScheduler _sched;
4    PTLock _addQueueLock;
5    boost::lockfree::spsc_queue<Task *> _addQueue = {100};
6
7    void processReadyTasks() {
8      _addQueue.consume_all(
9          [&](Task *t){ _sched.addReadyTask(t); });
10   }
11   void addReadyTask(Task *task) {
12     while (1) {
13       _addQueueLock.lock();
14       bool added = _addQueue.push(task);
15       _addQueueLock.unlock();
16       if (added) break;
17       if (_lock.tryLock()) {
18         processReadyTasks();
19         _lock.unlock();
20       }
21     }
22   }
23   Task *getReadyTask(uint64_t const id) {
24     Task *task;
25     if (!_lock.lockOrDelegate(id, task))
26       return task;
27     processReadyTasks();
28     while (!_lock.empty()) {
29       uint64_t waitingId = _lock.front();
30       task = _sched.getReadyTask(waitingId);
31       if (task == nullptr) break;
32       _lock.setItem(waitingId, task);
33       _lock.popFront();
34     }
35     task = _sched.getReadyTask(id);
36     _lock.unlock();
37     return task;
38   }
39 };
```

Listing 5. Implementation of the Synchronized Scheduler using a Delegation Ticket Lock

Listing 5 shows the implementation of a synchronized scheduler using a DTLock (line 2) and a SPSC wait-free queue (line 5) to synchronize the *getReadyTask* and *addReadyTask* operations, respectively. The SyncScheduler is a wrapper of the unsynchronized scheduler (line 3), which implements the actual scheduling policy.

The PTLock (line 4) protects the producer side of the SPSC queue in the *addReadyTask* function (line 14), while the DTLock protects the consumer side in the *processReadyTasks* function. In the *addReadyTask* function, if there is no free space on the SPSC queue, the thread will try to acquire the

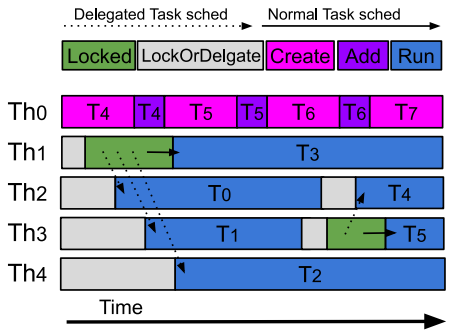

Figure 3. Timeline of five threads using a DelegationLock to add and get ready task into the scheduler.

DTLock with a *tryLock* operation (line 17). If it succeeds, it will call the *processReadyTasks* function (line 18), which removes all tasks from the SPSC queue and inserts them to the unsynchronized scheduler (line 8).

We use the *lockOrDelegate* operation (line 25) of the DTLock to synchronize *getReadyTask* operations. If the operation is delegated because another thread owns the DTLock, the calling thread will busy-wait until it gets a task (lines 25-26). Otherwise, it will eventually acquire the lock and call the *processReadyTasks* (line 27) to add the tasks that are waiting on the SPSC queue into the unsynchronized scheduler. Then, it will try to schedule a task for each of the threads that are waiting on the DTLock (lines 28-33) until there are no more waiting threads (line 28) or no ready tasks are left (line 31). At that point, it will try to get a task for him (line 35) and then release the DTLock (line 36). In our simplified design, the thread inside the scheduler leaves as soon as there are no more tasks to schedule. However, it is easy to extend our design in a way that *processReadyTasks* is called when no tasks are left inside the scheduler.

Figure 3 shows a timeline of five threads creating and scheduling tasks using the SyncScheduler and a simple FIFO scheduling policy. Th_0 has already created and inserted three tasks (T_0 - T_3) that are inside the SPSC queue before creating and inserting four additional tasks (T_4 - T_7). Th_1 to Th_4 call the *getReadyTask* function, one after the other, to obtain a ready task. The call to *lockOrDelegate* of Th_1 acquires the lock, while the other threads delegate and busy-wait. Once Th_1 is inside the lock, it calls *processReadyTasks* and inserts tasks T_0 to T_3 into the actual scheduler. Then, Th_1 schedules one ready task for each of the waiting threads, and finally, it gets a ready task for itself. When Th_3 finishes the execution of T_1, it calls again to *getReadyTask*, and just after that, Th_2 does the same. Th_3 acquires the lock and calls *processReadyTasks*, moving the tasks from the SPSC queue (T_4 and T_5) to the actual scheduler. Finally, Th_3 schedules T_4 for Th_2, and then, it executes T_5.

In microbenchmarks, we found a fourfold speedup on task scheduling using a DTLock compared to a PTLock, and a twelvefold speedup compared to serial task insertion thanks to the SPSC queues.

4 Memory management

When optimizing a runtime to achieve the lowest overhead, every operation that requires synchronization between threads quickly becomes a bottleneck. This is the case for memory allocation. Some general-purpose allocators are not well suited to handle a high volume of memory requests in many-core systems. Many implementations require the serialization of every allocation in the system. Additionally, the operating system may introduce even more overhead when allocators request more memory areas through system calls.

In the Nanos6 runtime case, removing contention from the scheduler and dependencies caused an even more significant bottleneck on memory allocation. However, the current state of the art techniques for scalable memory allocation can be applied to any software [2, 12], solving most of the contention problems. To solve this bottleneck and achieve the performance presented in this article, we had to substitute the default allocator in Nanos6 for Jemalloc, a widely used scalable memory allocator.

5 Instrumentation

Analyzing runtime performance and finding problems or bottlenecks in the different component requires a mechanism to collect fine-grained instrumentation data. This instrumentation must have a very low overhead, which is difficult to achieve on a very optimized runtime. Additionally, runtime systems are sensitive to OS noise (such as thread preemptions), making exploiting kernel internals particularly useful when evaluating latency-critical features, such as those presented in this article. For this reason, we have developed a new tracing backend aiming at minimum overhead and with both runtime and kernel tracing capabilities.

The backend generates traces in the Common Trace Format (CTF) [7], which strives for fast data writes. Instrumentation overhead is minimized by storing events on lock-free NUMA-aware per-core circular buffers. Each buffer is divided into page-aligned sub-buffers that, when full, are periodically flushed to a tmpfs backed file by Nanos6 threads between tasks execution. Each file contains a time-ordered event subset of the final trace, with either kernel or user events.

Nanos6 threads write events on the lock-free per-core buffer they are pinned to. User-selected Kernel events are obtained from a per-core memory-mapped circular buffer exported by the Linux Kernel through the *perf_event_open()* system call. Between task executions, Nanos6 threads read and format the kernel events according to the CTF specification and move them to an exclusive kernel-events Nanos6 per-core circular buffer.

6 Evaluation

In this section, we evaluate the effects of the different optimizations that we present in this paper. To evaluate their impact, we have prepared different versions of the Nanos6 runtime system, where each one removes one of the three optimizations. This methodology allows for a better understanding of which optimizations have the most significant impact on runtime performance. To prove that these optimizations make Nanos6 one of the lowest-overhead task runtimes, we also compare our most optimized version with the most relevant OpenMP implementations. We conduct our experiments on three HPC machines.

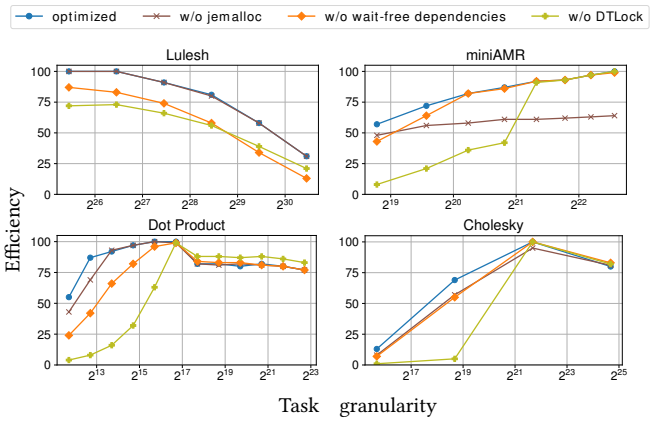

Figure 4. Efficiency vs task granularity of the Nanos6 runtime with and without the described optimizations on Intel Xeon (higher is better)

6.1 Methodology

To evaluate the task-based runtimes and check the capability of scaling to more finely partitioned work, we will use the following benchmarks, running constant problem sizes and varying the task granularity. These are (1) a **Dot product** between two arrays that uses a task reduction to aggregate the results from each block, (2) an iterative **Gauss-Seidel** method solving the heat equation of a 2-D matrix in blocks and task reductions to calculate the residual of each time step, (3) a taskified **HPCCG** with with several kernels using task reductions and multi-dependencies, (4) a taskified version of **Lulesh** 2.0 [20], (5) a taskified **miniAMR** that mimics the different patterns of Adaptive Mesh Refinement applications [33, 34], (6) a classic parallel blocked **Matmul**, (7) an **NBody** benchmark that mimics dynamic particle system simulations, and (8) a blocked **Cholesky** decomposition that is generally compute bound.

We ran our experiments on various HPC platforms featuring very different architectures: (1) the **Intel Xeon** with 2x Intel Xeon Platinum 8160 (Sky-lake) for a total of 48 cores, (2) the **ARM Graviton2** with 64 ARM Neoverse N1 cores, and (3) the **AMD Rome** with 2x AMD EPYC 7H12 processors for a total of 128 cores and 256 hardware threads.

For all benchmarks, the parallelization is implemented using tasks with OpenMP and OmpSs-2 versions that feature the same amount of parallelism. However, the kernels used inside each task are sourced from the best available vendor library for each machine, to guarantee competitive performance. In the AMD and Intel machines this library was Intel MKL, and on the ARM Graviton2 we used the ARM Performance Libraries. We ran each benchmark a minimum of five times to extract each measurement.

Following this evaluation, we will compare our optimized Nanos6 runtime with the most relevant OpenMP implementations for each machine, including GOMP 9.2.0 [17], LLVM 10 [23], Intel OpenMP and the AMD AOCC depending on availability for each platform. It is worth noting that both the LLVM, AMD AOCC and Intel OpenMP runtime are based on a work-stealing scheduler, which will allow us to determine if our centralized delegation-based implementation can outperform work-stealing runtimes.

Note that some combinations might not be available depending on the platform because of incompatibilities, non-implemented OpenMP features or compiler bugs. For brevity, we only show the four most relevant benchmarks for each machine.

6.2 Results

The best way we found to evaluate objectively how each component of the runtime affects the overall performance is to remove the optimizations we have described selectively. To present the results, we use a metric [35] we will refer to as *efficiency*. It is calculated by dividing the performance of a specific run of a benchmark by the peak performance obtained across all executions. This *efficiency* provides a view of how close to peak performance is a specific run while being agnostic to benchmark specific units. Combining this metric with varying task granularity [15, 24] gives a good view of each runtime version's scalability. The granularity is expressed in instructions executed per task, which gives an approximation of the task's size. We chose this unit instead of using time or cycles because the scheduling policies used by each of the runtimes can affect the execution time of a task, and thus it could result in unfair comparisons.

The runtime version without the wait-free dependencies uses the previous dependency implementation based on fine-grained locking. The variant without the DTLock has a simple mutual exclusion mechanism (based on the PTLock) protecting the scheduler. Finally, the version without jemalloc uses the standard system allocator for each of the machines.

Figure 4 displays our benchmarks running in the Intel Xeon platform. The different versions allow us to explore precisely how each optimization affects the scalability for

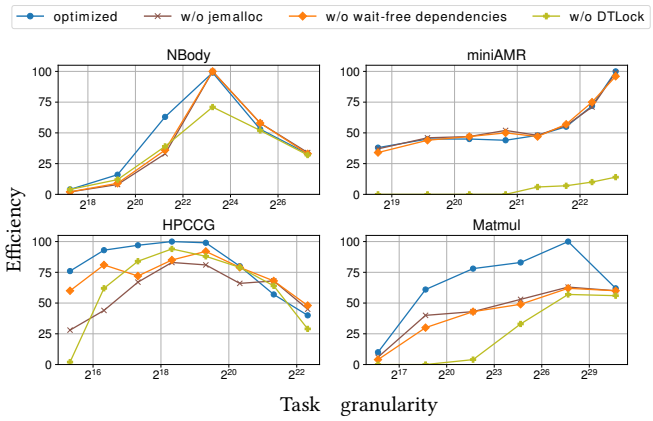

Figure 5. Efficiency vs task granularity of the Nanos6 runtime with and without the described optimizations on AMD Rome (higher is better)

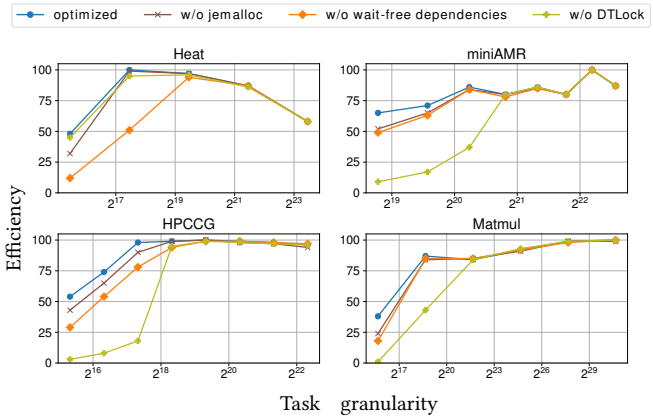

Figure 6. Efficiency vs task granularity of the Nanos6 runtime with and without the described optimizations on ARM Graviton2 (higher is better)

different benchmarks. The results also confirm that for every benchmark at least one optimization greatly increases the performance for fine-grained tasks.

Figure 5 shows a similar picture on the AMD Rome system. This system has a much larger number of CPUs, which can increase the performance degradation caused by heavily contended locks. Illustrating this point, we see that the scheduler optimization is much more relevant than in the Intel Xeon. The clearest example is seen on the *miniAMR* benchmark, which we analyze with detailed traces in subsection 6.4.

Finally, Figure 6 shows the same benchmarks running on an ARM Graviton2 system. Results are similar to our Intel Xeon evaluation, although some benchmarks have different behaviors due to the lack of NUMA effects on this platform.

Overall, we have seen that our optimizations achieve significant performance gains, especially on small task granularities. Depending on the benchmark, the wait-free dependencies or the scheduler are the most important optimizations.

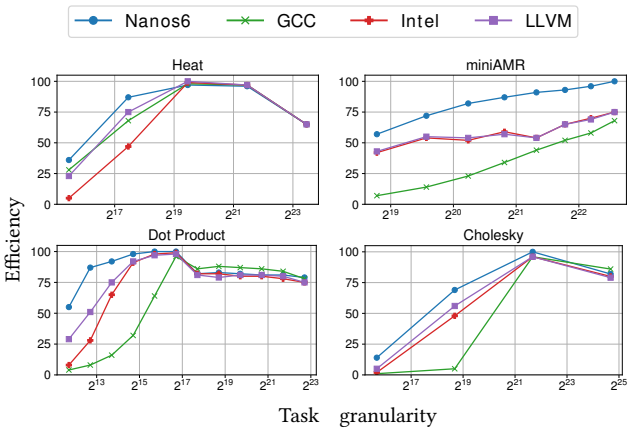

Figure 7. Comparison of performance between the current Nanos6 version and the main OpenMP runtimes on Intel Xeon (higher is better)

However, the scalable memory allocator also delivers some notable performance improvements, especially on the Intel Xeon and AMD Rome machines.

6.3 Comparison versus OpenMP

To give context to the results, we compare the optimized Nanos6 runtime to current state-of-the-art OpenMP runtimes on the same three systems. Our baseline for every benchmark will be the GOMP runtime, distributed with the GCC Compiler, and the LLVM OpenMP Runtime. However, on Intel Xeon we use the Intel OpenMP runtime, and on AMD Rome the runtime provided by the AMD AOCC, as long as they implement all the features needed by the benchmark. To ensure a fair comparison, we expressed the same parallelism in the OmpSs-2 and OpenMP versions of the benchmarks. We also used available kernels in Intel MKL or the ARM Performace Libraries, to prevent noise introduced by the compilers to alter the results.

Figures 7, 8 and 9 feature the results of comparing the current OpenMP implementation versus the optimized variant of the Nanos6 runtime. The results are really positive, as in all of the machines, and all of the benchmarks, the best performance in small granularity tasks is provided by the Nanos6 runtime. In some cases, a higher peak performance is also achieved. This happens when the ideal block size for a specific benchmark is small enough that performs better in one runtime than another.

As for the other runtimes, the LLVM OpenMP implementation comes second in most benchmarks in terms of scalability, and ties on AMD Rome with the runtime provided with the AOCC compiler. However, this is expected because the AOCC compiler is based on LLVM 10.

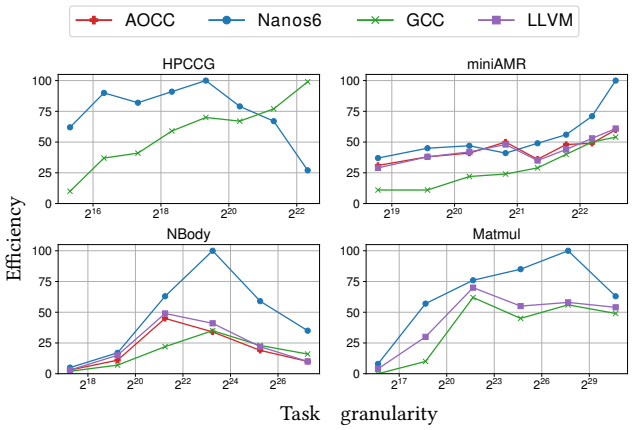

Figure 8. Comparison of performance between the current Nanos6 version and the main OpenMP runtimes on AMD Rome (higher is better)

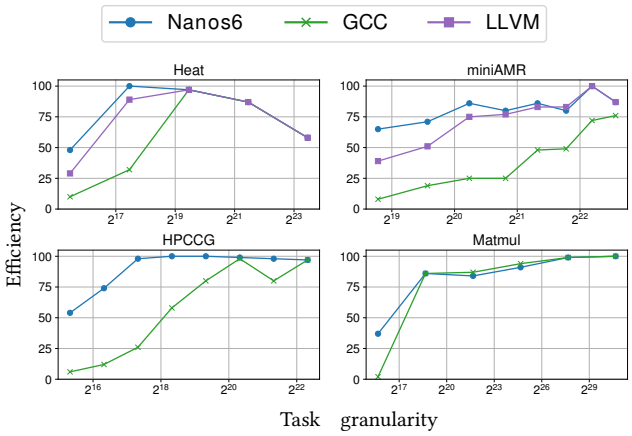

Figure 9. Comparison of performance between the current Nanos6 version and the main OpenMP runtimes on ARM Graviton2 (higher is better)

6.4 Detailed traces

Figure 10 shows two miniAMR traces obtained with the new Nanos6 CTF instrumentation backend comparing the Partitioned Ticket Lock (PTlock) and the combination of wait-free queues with the Delegation Ticket Lock (DTLock). The view displays running tasks (in red), specific runtime subsystems such as task creation (in cyan), or other generic runtime parts (in deep blue) along time (X axis) for a number of cores (Y axis). The total length is 500us and the zoomed areas (black rectangles) are 10us long, approximately. Hint *A* points to cores running a "task creation" task. The wait-free version (above) allows created tasks to be queued independently of other cores requesting ready tasks to run. Instead, in the PTLock version (below), adding and getting a ready task requires obtaining a shared lock which leads the "task creation" task to undergo heavy pressure as other cores requesting a ready task also attempt to acquire the same lock. Consequently, the number of available ready tasks cannot match the task completion rate and most cores starve (in khaki green).

Hint *B* points to DTLock task serving periods. Yellow arrows depict single tasks being served by the delegation lock owner to waiting cores. When no more ready tasks left, the lock owner moves ready tasks from the wait-free queues (in green) and continues serving tasks.

Figure 11 exemplifies the effect that the operating system noise can incur on the runtime system. The upper trace displays runtime threads (in deep blue) and hardware interruptions (purple). The trace below shows a view similar to Figure 10, where a considerable delay is introduced in the server which causes all cores to stall but the "task creator" core. Note the yellow lines pattern difference before (irregular) and after (regular) the interrupt. While the serving thread was stalled in the interrupt, a provision of ready tasks was accumulated. The surplus of tasks is enough to feed all cores leading to long periods of red (tasks) without yellow lines. As the extra reserve of tasks lowers, the chances of at least two idle cores requesting a task simultaneously increase, and extra yellow lines appear. In conclusion, combining OS events with runtime events allows us to complete the whole picture and to identify the source of problems better.

7 Related work

Other dependency system implementations have been described in previous literature, such as the implementation for the OpenUH compiler [16]. The GOMP library and LLVM's OpenMP runtime are also available online as Open Source software [17][23]. The topic of overhead in dependency resolution has also been tackled from other angles, such as the TurboBLYSK framework, to create dependency patterns [29].

Previous research has also aimed at applying a lock-free approach to dependency resolution, with [37] analyzing several generic dependency resolution schemes and concluding that a ticket-based lockless scheme provided the best performance in their benchmarks. On the same line, another lock-free dependency system was implemented for the OMPi OpenMP/C Compiler [1], which supported OpenMP 4.0 and was based on the same patterns as lock-free lists. However, our implementation offers a stronger wait-free guarantee.

Regarding task scheduling, there are several studies on alternatives to centralized lock-based scheduling. Many have studied work-stealing techniques for hierarchical and partitioned schedulers in shared-memory systems [18, 30, 39]. Olivier et al. [27] proposed a hierarchical scheduler featuring a lock-free ready task queue per socket. Once the socket's queue is empty, only one of the threads in that queue can try to steal tasks from other sockets, while the others wait. Similarly, Muddukrishna et al. [26] proposed a lock-based queue per NUMA node, but in this case, workers from the same NUMA can steal at the same time. In contrast, Vikranth

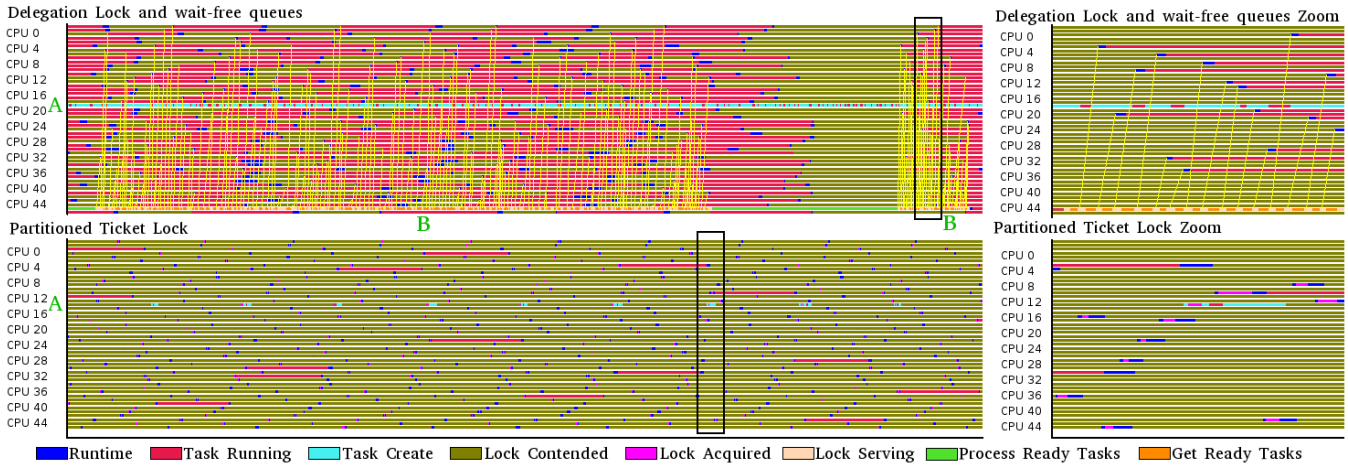

Figure 10. Scheduler lock comparison

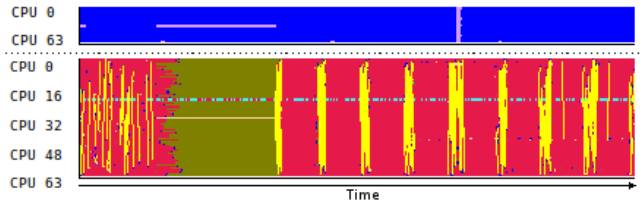

Figure 11. Operating System noise effect on Scheduler

et al. [38] implemented a task queue per thread, grouped into stealing domains (e.g., one per socket). Workers always first try to steal from queues in the local domain before trying in the rest. However, all these approaches can suffer the same bottlenecks as centralized schedulers when running in modern many-core systems. The ready task queues of consumer threads tend to be empty, so they steal tasks from the creator's queues (usually few), producing contention in those sections. Also, hierarchical schedulers often have complicated implementations, so developing new scheduling policies becomes an arduous task.

Even though, notice that these hierarchical approaches with work-stealing could use our DTLock (Section 3.3) to protect the access to task queues.

The Linux Kernel static tracing infrastructure is used by several backends such as LTTng, ftrace, perf, SystemTap or eBPF. Yet, only LTTng focuses on efficient user and kernel correlated static tracing [14]. When tracing on the context of runtime systems for HPC, LTTng has two main drawbacks. On the one hand, tracing the kernel requires installing an out-of-tree Linux Kernel module that usually clashes with operating policies of data centers and supercomputing facilities. On the other hand, LTTng relies on server daemons to collect and write tracepoints from both user and kernel space. Such daemons might oversubscribe runtime threads leading to undesired noise. Therefore, we concluded that our particular case needed an ad-hoc tracing solution.

8 Conclusion and Future Work

The proliferation of many-core architectures and workloads with irregular parallelism and load imbalance have shifted the focus from traditional fork-join parallelism to task-based parallelism. Nevertheless, task management costs are still an important source of overhead, especially when using fine granularities. Throughout this paper, we enhance two critical components that bound the ability of runtime systems to manage fine-grained tasks: the dependency system and the scheduler. We combine both with a state of the art memory allocator to achieve very competitive performance.

We have introduced a novel wait-free approach to implementing dependency management inside a parallel runtime. We have also defined the *Atomic State Machine* concept and its restrictions and formalized its wait-freedom. We believe the ASM concept is applicable to similar models and runtimes that use a data-flow execution model.

Additionally, we proposed a novel *Delegation Ticket Lock* that delivers very good performance compared to other state-of-the-art locks, while keeping the simplicity in the development of scheduling internals and policies.

We also identified the critical contention bottleneck caused by memory management and tackled the problem by leveraging the *jemalloc* state-of-the-art scalable memory allocator.

Finally, we implemented a highly-detailed instrumentation to provide information from both application and kernel level, while introducing minimal overhead. Such a tool is crucial to identify and analyze bottlenecks in modern runtime systems.

Our evaluation assesses the performance of the different components separately and together, showing important performance improvements compared to (1) the previous

version of the runtime system, and (2) state-of-the-art runtime systems such as Intel OpenMP, GNU GOMP and LLVM OpenMP.

As future work, we plan to investigate extensions of the DTLock interface to support flat combining [19]. This interface will require the ability to access and unblock several waiting threads simultaneously to be able to combine their operations.

Acknowledgments

This project is supported by the European Union's Horizon 2020 Research and Innovation programme under grant agreement No.s 754304 (DEEP-EST), by the Spanish Ministry of Science and Innovation (contract PID2019-107255GB and TIN2015-65316P) and by the Generalitat de Catalunya (2017-SGR-1414). We acknowledge PRACE for awarding us access to Joliot-Curie at GENCI@CEA, France.

References

[1] Anastasios Souris. 2015. Design and implementation of the OpenMP 4.0 task dataflow model for cache-coherent shared-memory parallel systems in the runtime of the OMPi OpenMP/C compiler. https://doi.org/10.26233/HEALLINK.TUC.30331

[2] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS IX)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/378993.379232

[3] Shekhar Borkar and Andrew A. Chien. 2011. The Future of Microprocessors. *Commun. ACM* 54, 5 (May 2011), 67–77. https://doi.org/10.1145/1941487.1941507

[4] BSC. 2020. Nanos6 GitHub. https://github.com/bsc-pm/nanos6

[5] BSC. 2020. OmpSs-2 Specification. https://pm.bsc.es/ftp/ompss-2/doc/spec/OmpSs-2-Specification.pdf

[6] L. Dagum and R. Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (Jan 1998), 46–55. https://doi.org/10.1109/99.660313

[7] Mathieu Desnoyers. 2020. The Common Trace Format. https://diamon.org/ctf/v1.8.3

[8] David Dice. 2011. Brief Announcement: A Partitioned Ticket Lock. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 309–310. https://doi.org/10.1145/1989493.1989543

[9] Dave Dice and Alex Kogan. 2019. TWA – Ticket Locks Augmented with a Waiting Array. In *Euro-Par 2019: Parallel Processing*, Ramin Yahyapour (Ed.). Springer International Publishing, Cham, 334–345.

[10] Alejandro Duran, Josep M. Perez, Eduard Ayguadé, Rosa M. Badia, and Jesus Labarta. 2008. Extending the OpenMP Tasking Model to Allow Dependent Tasks. In *OpenMP in a New Era of Parallelism*, Rudolf Eigenmann and Bronis R. de Supinski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 111–122.

[11] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. 2011. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 365–376.

[12] Jason Evans. 2020. jemalloc. http://jemalloc.net/

[13] Ferran Pallarès Roca. 2017. *Extending OmpSs programming model with task reductions: A compiler and runtime approach*. Bachelor's Thesis. Barcelona School of Informatics, Universitat Politècnica de Catalunya.

[14] Pierre-Marc Fournier, Mathieu Desnoyers, and Michel R Dagenais. 2009. Combined tracing of the kernel and applications with LTTng. In *Proceedings of the 2009 linux symposium*. Citeseer, 87–93.

[15] Thierry Gautier, Christian Perez, and Jérôme Richard. 2018. On the Impact of OpenMP Task Granularity. In *Evolving OpenMP for Evolving Architectures*, Bronis R. de Supinski, Pedro Valero-Lara, Xavier Martorell, Sergi Mateo Bellido, and Jesus Labarta (Eds.). Springer International Publishing, Cham, 205–221.

[16] Priyanka Ghosh, Yonghong Yan, Deepak Eachempati, and Barbara Chapman. 2013. A Prototype Implementation of OpenMP Task Dependency Support. In *OpenMP in the Era of Low Power Devices and Accelerators*, Alistair P. Rendell, Barbara M. Chapman, and Matthias S. Müller (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 128–140.

[17] GNU Project. 2020. GOMP Source Code. https://github.com/gcc-mirror/gcc/tree/master/libgomp Accessed: 2020-02-01.

[18] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. 2010. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 1–12.

[19] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Thira, Santorini, Greece) *(SPAA '10)*. Association for Computing Machinery, New York, NY, USA, 355–364. https://doi.org/10.1145/1810479.1810540

[20] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes*. Technical Report LLNL-TR-641973. 1–9 pages.

[21] D. Klaftenegger, K. Sagonas, and K. Winblad. 2018. Queue Delegation Locking. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (2018), 687–704. https://doi.org/10.1109/TPDS.2017.2767046

[22] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M. Badia. 2010. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice and Experience* 22, 1 (2010), 15–44. https://doi.org/10.1002/cpe.1467 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1467

[23] LLVM Project. 2020. LLVM OpenMP Library Source. https://github.com/llvm/llvm-project/tree/master/openmp

[24] M. Maroñas, K. Sala, S. Mateo, E. Ayguadé, and V. Beltran. 2019. Worksharing Tasks: An Efficient Way to Exploit Irregular and Fine-Grained Loop Parallelism. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 383–394. https://doi.org/10.1109/HiPC.2019.00053

[25] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (Feb. 1991), 21–65. https://doi.org/10.1145/103727.103729

[26] Ananya Muddukrishna, Peter A Jonsson, and Mats Brorsson. 2015. Locality-aware task scheduling and data distribution for OpenMP programs on NUMA systems and manycore processors. *Scientific Programming* 2015 (2015).

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

[28] J. M. Perez, V. Beltran, J. Labarta, and E. Ayguadé. 2017. Improving the Integration of Task Nesting and Dependencies in OpenMP. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 809–818. https://doi.org/10.1109/IPDPS.2017.69

[29] Artur Podobas, Mats Brorsson, and Vladimir Vlassov. 2014. TurboBŁYSK: Scheduling for Improved Data-Driven Task Performance with Fast Dependency Resolution. In *Using and Improving OpenMP for*

Devices, Tasks, and More, Luiz DeRose, Bronis R. de Supinski, Stephen L. Olivier, Barbara M. Chapman, and Matthias S. Müller (Eds.). Springer International Publishing, Cham, 45–57.

[30] Aleksandar Prokopec and Martin Odersky. 2013. Near optimal work-stealing tree scheduler for highly irregular data-parallel workloads. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 55–86.

[31] David P. Reed and Rajendra K. Kanodia. 1979. Synchronization with Eventcounts and Sequencers. *Commun. ACM* 22, 2 (Feb. 1979), 115–123. https://doi.org/10.1145/359060.359076

[32] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. 2017. Ffwd: Delegation is (Much) Faster than You Think. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 342–358. https://doi.org/10.1145/3132747.3132771

[33] Kevin Sala, Alejandro Rico, and Vicenç Beltran. 2020. Towards Data-Flow Parallelization for Adaptive Mesh Refinement Applications. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 314–325.

[34] Aparna Sasidharan and Marc Snir. 2016. *MiniAMR - A miniapp for Adaptive Mesh Refinement*. Technical Report. University of Illinois. 1–21 pages. http://hdl.handle.net/2142/91046

[35] Elliott Slaughter, Wei Wu, Yuankun Fu, Legends Brandenburg, Nicolai Garcia, Wilhem Kautz, Emily Marx, Kaleb S. Morris, Qinglei Cao, George Bosilca, Seema Mirchandaney, Wonchan Lee, Sean Teichler, Patrick McCormick, and Alex Aiken. 2020. Task Bench: A Parameterized Benchmark for Evaluating Parallel Runtime Performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. Association for Computing Machinery.

[36] T. N. Theis and H. . P. Wong. 2017. The End of Moore's Law: A New Beginning for Information Technology. *Computing in Science Engineering* 19, 2 (2017), 41–50.

[37] Hans Vandierendonck, George Tzenakis, and Dimitrios S. Nikolopoulos. 2013. Analysis of Dependence Tracking Algorithms for Task Dataflow Execution. *ACM Trans. Archit. Code Optim.* 10, 4, Article 61 (Dec. 2013), 24 pages. https://doi.org/10.1145/2541228.2555316

[38] B Vikranth, Rajeev Wankar, and C Raghavendra Rao. 2013. Topology aware task stealing for on-chip NUMA multi-core processors. *Procedia Computer Science* 18 (2013), 379–388.

[39] Yizhuo Wang, Yang Zhang, Yan Su, Xiaojun Wang, Xu Chen, Weixing Ji, and Feng Shi. 2014. An adaptive and hierarchical task scheduling scheme for multi-core clusters. *Parallel computing* 40, 10 (2014), 611–627.

A Artifacts Appendix

A.1 Getting Started

The artifacts of this paper are provided as a docker image and can be found at the Zenodo archive:
https://doi.org/10.5281/zenodo.4290558

To run the image, the only prerequisite is to have docker installed in your local machine. To download and run the image, the following commands can be used:

```
$ wget \
  https://zenodo.org/record/4290558/files/ppopp.tar
$ docker load --input ppopp.tar
$ docker run -h debian --name artifact \
  -it artifacts/ppopp:1.0.0
```

After running the earlier commands, you will be presented with an interactive prompt in an image with all the prerequisites to run the benchmarks installed.

The full suite of benchmarks can take several hours to run completely, and requires a system with a large amount of main memory (+ 32 GB), as some problem sizes are big. To test the functionality of the artifacts, a small suite of benchmarks is provided, which will generate the granularity scaling plots with smaller problem sizes, and can be run in a few minutes. The small suite also does only one execution of each benchmark, providing no standard deviation information.

To run the reduced set of benchmarks, the following script is provided which can be executed from `/home/user`:

```
user@debian:~$ ./run-small-suite.sh
```

After running the benchmarks, the results corresponding to the comparison between the optimized Nanos6 runtime compared with GNU OpenMP and LLVM OpenMP will be stored in the `/home/user/output/` folder. To retrieve the results back to the local machine, the following command can be used **outside the container**:

```
$ docker cp artifact:/home/user/output .
```

Which will create a folder named `output` in the current path and retrieve the plots in pdf format.

A.2 Step by step

The artifact is prepared to be flexible and all the sources as well as scripts to re-build all of the software are included. In this section we will explain the structure of the image, how to run the full benchmark suite, and how to change and re-build the experiments and software.

In case it is needed to install more software on the docker image, the password for the user of the machine is `user`.

A.2.1 Full benchmark suite. It is possible to run the full benchmark suite with the same parameters that were used on the original paper. However, be warned that the expected runtime is several hours, and that not all machines may be able to handle the input sizes due to lack of memory. To do so, run the following command:

```
user@debian:~$ ./run-full-suite.sh
```

A.2.2 Directory structure. The image has the following structure on which the relevant files can be found:

- `Sources`: Contains the source code for the Nanos6 runtime as well as its dependencies (Mercurium, Jemalloc and TAMPI), and GCC 9.3.0 for the benchmarks.
- `Benchmarks`: Contains the source code and binaries for the benchmarks, each one in a separate folder with a standalone (and working) Makefile.
- `Install`: Contains the built binaries for Nanos6 and its dependencies.
- `Automate`: Contains the Pyhton scripts and JSON configuration files that are used to execute the benchmarks.

- `output`: Output directory, where the plots of granularity and efficiency for each of the benchmarks are saved after running the benchmark suite.

A.2.3 Nanos6 Sources. Although the sources used in the article evaluation are included in the docker image, Nanos6 is free software and the most up-to-date version of the sources is publicly available on the following GitHub repository: https://github.com/bsc-pm/nanos6

We suggest using the last version from git if you want use Nanos6 in your research.

A.2.4 Rebuilding the software. In the root folder, an `install-all.sh` script is provided which will re-extract all the sources and re-build Nanos6, its dependencies, and the GCC 9.3.0 toolchain. This is provided as a way to make the image re-usable, as it is possible to change the sources and re-build the whole stack.

In case you want to install Nanos6 bare-metal in a machine, we suggest to refer to the official Nanos6 documentation which can be found on the GitHub repository or inside the included source tarball, which will guide you through all the configuration and building process.

A.2.5 The OmpSs-2 Programming Model. The BSC Programming Models research group routinely supports other researchers that want to use or improve the OmpSs-2 programming model, the Nanos6 runtime or any of our tools. The specification for the OmpSs-2 programming model can be found online at https://pm.bsc.es/ompss-2, and you can reach us by email at `pm-tools@bsc.es`.

A.3 Supported claims

This artifact is designed to support the claims done in Section 6 of the paper, specifically the performance comparison between the Nanos6 runtime, containing all of the novelties presented on the paper, and other state of the art OpenMP runtimes.

The goal of the artifacts is to facilitate the reuse of our research by others, and allow access to a functional and reusable version of the sources and benchmarks referenced in the paper. Reproducing exactly the results obtained in the paper would require access to the exact same machines and software that was used, which is not in the scope of the artifact.

Running the benchmark suite will generate the performance plots based on task granularity, using the same method that was used to generate the original plots. Raw results are also extractable, and are available on the `Automate/scaling` folder (or the `Automate/scaling_small/` for the small suite). However, there are some caveats and claims that are not supported by the artifacts:

- The artifacts do not include non-free software that was used during the evaluation. Specifically, the following software is not included and thus not evaluated in the comparison:
  - The Intel MKL library. Instead, the BLAS and LAPACK kernels used in the benchmarks have been linked against the OpenBLAS library, which may affect the results. If desired, the user can download the Intel MKL libraries from debian's non-free repositories and link the benchmarks against them.
  - The Intel Compiler and OpenMP runtimes, which require Intel licenses to use.
  - The AMD Optimizing C/C++ Compiler and OpenMP runtime, which is available on AMD's website.
  - The ARM Performance Libraries, which were used on the Graviton2 machine.
- Performance evaluation was done bare-metal in all the machines, without container overhead and always on exclusive nodes. Caution is advised when drawing conclusions from the results obtained running the artifacts, as similar conditions need to be achieved for the results to be valid.