Analysis of Dependence Tracking Algorithms for Task Dataflow Execution[*]

Hans Vandierendonck
Queen's University Belfast
h.vandierendonck@qub.ac.uk

George Tzenakis
Queen's University Belfast
gtzenakis01@qub.ac.uk

Dimitrios S. Nikolopoulos
Queen's University Belfast
d.nikolopoulos@qubac.uk

Abstract

Processor architectures has taken a turn towards many-core processors, which integrate multiple processing cores on a single chip to increase overall performance, and there are no signs that this trend will stop in the near future. Many-core processors are harder to program than multi-core and single-core processors due to the need of writing parallel or concurrent programs with high degrees of parallelism. Moreover, many-cores have to operate in a mode of strong scaling because of memory bandwidth constraints. In strong scaling increasingly finer-grain parallelism must be extracted in order to keep all processing cores busy.

Task dataflow programming models have a high potential to simplify parallel programming because they alleviate the programmer from identifying precisely all inter-task dependences when writing programs. Instead, the task dataflow runtime system detects and enforces inter-task dependences during execution based on the description of memory each task accesses. The runtime constructs a task dataflow graph that captures all tasks and their dependences. Tasks are scheduled to execute in parallel taking into account dependences specified in the task graph.

Several papers report important overheads for task dataflow systems, which severely limits the scalability and usability of such systems. In this paper we study efficient schemes to manage task graphs and analyze their scalability. We assume a programming model that supports input, output and in/out annotations on task arguments, as well as commutative in/out and reductions. We analyze the structure of task graphs and identify *versions* and *generations* as key concepts for efficient management of task graphs. Then, we present three schemes to manage task graphs building on *graph representations*, *hypergraphs* and *lists*. We also consider a fourth edge-less scheme that synchronizes tasks using integers. Analysis using micro-benchmarks shows that the graph representation is not always scalable and that the edge-less scheme introduces least overhead in nearly all situations.

1 Introduction

The task dataflow model is gaining momentum to fill in an important gap in the parallel programming landscape. This model aims to simplify parallel programming by annotating tasks in

[*]This paper is accepted for publication in ACM Transactions on Architecture and Code Optimization.

a program with the side effects that they incur on memory. To this end, the programmer or compiler determines the memory footprint of a task and annotates it with the potential side-effect. Thus, it is made clear what memory a task may touch and whether it will be read or written. Next, the runtime system tracks dependences between tasks as they are spawned by analyzing overlap in memory footprints and read/write access modes. Using this information, it can schedule tasks with maximal parallelism while enforcing an execution order that yields results identical to serial execution.

The genericity of the task dataflow model has been demonstrated on scientific applications [8, 3, 21], but also on problems in bioinformatics [1] and H.264 video encoding [2] and decoding [11]. The task dataflow model is applied to generic applications such as image analysis, cryptography, etc. [19]. Moreover, it has been argued that task dataflow is an elegant way to express pipeline parallelism in general [25].

While the task dataflow model is very appealing in terms of programming simplicity, several recent research papers have reported significant overheads for dynamic dependence tracking. Best et al. [6] show up to 40% overhead compared to the task time. Perez et al. [22] report overheads per task spawn that are often around 100μs, but can be as large as 6.8ms. Such overheads are clearly limiting the applicability and scalability of the task dataflow model. More importantly, the number of cores on a chip is quickly outgrowing the available memory bandwidth. As such it is becoming increasingly important to strive for strong scaling [13]. Strong scaling occurs when a parallel program is executed on an increasing number of threads while the problem size is kept constant. Additional parallelism can be found in the program by performing less work per task, but this work is soon to be overshadowed by the sizeable overheads of the runtime systems cited above.

This paper contributes to task dataflow scheduling by presenting and analyzing techniques for dynamic dependence tracking that have overheads as low as 0.20μs per task spawn (about 400 cycles on our evaluation system) and are scalable to task graphs containing over 1 million outstanding tasks. We arrive at this state-of-the-art performance by carefully analyzing the algorithms for dynamic task graph management and by presenting a series of improvements on the basic algorithms.

The main contribution of this paper is to explicitly formulate and evaluate algorithms and accompanying data structures for dynamic dependence tracking. The literature is gravely failing to formulate the algorithms used. This is an important shortcoming in the literature as we have found that naive implementations (including our first attempts which are not reported here) perform orders of magnitude worse than the highly tuned algorithms presented in this paper.

This paper is structured as follows. Section 2 presents a task dataflow programming model that provides the typical annotations one would expect to see in these models: input, output, in/out, commutative in/out and reductions. Section 3 presents the execution model and analyzes the structure of the task graphs that may be built during execution. We identify two key concepts that characterize task graphs: *versions*, capturing renaming of arguments, and *generations*, capturing sets of parallel tasks.

Based on our analysis of task graphs, we present and analyze four schemes for dynamic task graph management in Section 4. The baseline scheme represents the task graph as a graph. An obvious approach, we show that the scheme is subject to runtime overheads of $O(N)$ *per task*, where N is the number of tasks in the task graph. Our second scheme exploits the structure of the task graph and represent it as a hypergraph, where edges connect sets of nodes instead of a pair of nodes. This representation is much more economical and scales well to large task graphs and to large numbers of task arguments. Our third scheme builds on observations made on the hypergraph scheme, and restyles the hypergraph to a list, using specific properties of task graphs. This reorganization yields the same performance scalability, but reduces the constant in the performance equation. Besides these edge-centric schemes, we also consider an edgeless tickets scheme that synchronizes tasks using integers [26]. Our experimental evaluation in Section 5 shows that, despite our efforts to optimize the edge-centric schemes, edge-centric schemes cannot match the performance of the tickets scheme in the common cases.

Next, Section 6 discusses related work and Section 7 concludes this paper.

2 Programming Model

We assume a task based programming model that extends the Cilk language with dataflow annotations and execution [25, 26]. In this language, the *spawn* keyword is inserted before a function call to indicate that the call may proceed in parallel with the continuation of the calling procedure. The *sync* keyword indicates that the execution of the procedure should be delayed until all spawned procedures have finished execution. Spawned functions may have arguments that are annotated with dataflow annotations. These annotations restrict the parallelism between spawned functions.

2.1 Objects

Objects are special program variables that can be used in task dataflow. Annotated task arguments can only accept objects as arguments (not constants or generic variable types). An object is an undivisible piece of memory for the purpose of dependence tracking. We assume that all objects are strictly non-overlapping.

An object may be renamed, which means that its address is changed by the runtime system. The runtime system performs renaming to increase parallelism. The runtime system also makes sure that latent pointers to renamed objects are properly translated to the appropriate version of the object before accessing memory.

The runtime system associates metadata to each object, e.g. to perform dependence analysis and to recover its most recent version after renaming. The runtime system stores this metadata side-by-side with the object in order to speedup the retrieval of metadata.

2.2 Memory Usage Annotations

The arguments of spawned procedures may be annotated with *memory usage information*, i.e. how the argument is accessed by the task. The memory usage may be *input*, *output*, *input/output*, *commutative in/out* or *reduction*. An input argument is read but not written to. An output argument is written and may be read, but it is always written before it is read. Consequently, its value upon initiation of the task is irrelevant. An input/output argument (or in/out for short) may be read and written and it may be read before it is written.

A commutative in/out annotation extends the in/out semantics with the notion that consecutively spawned tasks may be executed in any order, but not at the same time (mutual exclusion). Naturally, reordering is subject to the absence of other inter-task dependences.

A reduction is defined as a mathematical group consisting of an associative and commutative operator, together with a unit.[1] The runtime system automatically provides private storage for reduction arguments in case multiple reductions on the same variable are executing concurrently. The runtime system also reuses the private storage in order to improve memory locality and to minimize applications of the reduction operator. When the value of a reduced variable is

[1] Cilk++ supports reductions using non-commutative operators [15]. As our system is built on Cilk, we assume that such non-commutative reductions are present. These do not require the dataflow annotations. We provide the commutative reduction annotation for increased performance as it requires fewer reduction operations.

consumed by a task with a non-reduction annotation, then all private copies are automatically reduced to a single value. This reduction is performed serially for scalar reductions and it is performed using a parallel tree reduction for non-scalar (e.g. matrix) reductions.

We stipulate that all arguments passed to a task are unique objects. This is to avoid circular dependences of a task on itself.

3 Execution Model

The execution model of a task dataflow language assumes that a sequential thread of execution steps through the program and, in the process, encounters a sequence of tasks. This sequence, together with the memory usage annotations, defines dependences between tasks. A dependence states that a pair of tasks *must* execute in the order that they were spawned. These tasks are added one by one to the task graph, where nodes represent dynamic task instances and edges represent task dependences.

The task graph is a directed acyclic graph (DAG) because tasks can only depend on tasks that appear before them in (serial) program order. At any moment, the roots of the DAG are tasks that are either *executing* or that are *ready to execute.* We call the list of root tasks that are ready to execute the *ready list.* It provides direct access to the ready tasks when one is needed.

3.1 Task Graph Operations: Issue and Release

The two main book-keeping operations on a task are issue and release. Issue occurs when the task is spawned and implies that the task is linked to all tasks in the DAG on which it directly depends. If the task is a root of the DAG (no edges inserted for the task), then the runtime system may continue with the execution of the task; else it executes the continuation of the parent.

Release happens when the task has finished execution. At this moment, the task is removed from the DAG and from the ready list and all its dependents that become a root of the DAG are pushed on the ready list. At this moment, the runtime system selects a task for execution, which is either the parent procedure or a task on the ready list.

Note that the runtime system may be built such that multiple independent task graphs are operational for the same program, typically one task graph per procedure instance [19, 26]. As such, issue operations on the same task graph never occur concurrently as they are performed by the thread executing the function corresponding to that task graph. On the other hand, multiple release operations may occur concurrently and they may occur concurrently with an issue operation.

3.2 The Structure of a Task Graph

Tasks are ordered by *true*, *anti* and *output* dependences [18]. Table 1 shows the dependences between usage annotations. The entries “none$^{(x)}$” and “none$^{(r)}$” indicate that commutative and reduction annotations require special actions to ensure mutual exclusion and managing private copies, respectively. We consider these actions as separate steps from dependence tracking, so for the purpose of this paper we will treat these entries as meaning absence of dependence.

In this paper, we will reduce the overhead of dynamic task graph management by exploiting properties that follow from Table 1. Here we assume that the tasks have only a single argument, an assumption that we will make throughout this paper for reasons of simplicity. We claim that the presented reasoning applies equally well when aggregating our findings over multiple task arguments.

Table 1: Dependences arising between two tasks operating on a common argument.

		First task				
		input	output	in/out	commu-tative	reduction
Second task	input	none	true	true	true	true
	output	anti	output	output	output	output
	in/out	anti	true	true	true	true
	commutative	anti	true	true	none$^{(x)}$	true
	reduction	anti	true	true	true	none$^{(r)}$

First, we notice that tasks with an output annotation may be easily renamed. Renaming eliminates anti and output dependences by mapping an object to different storage space. We say that a new **version** of the object is created. A new version introduces new data storage for the object and has fresh metadata for dependence tracking that is initialized to an "unused" state. As such, distinct versions of the same object behave as if they were entirely distinct objects for the purpose of dependence tracking.

Figure 1 illustrates versions by means of a sequence of tasks in program order and the resulting task graph. Task T7 with output usage gives rise to renaming and the creation of version 1. Tasks younger than T7 operate on version 1 while older tasks operate on version 0.

Second, we find that parallelism between tasks exists only between tasks with *the same annotation* ("none" occurs only on the diagonal of Table 1). Distinct annotations imply dependences that must be enforced.

We exploit this property by introducing the notion of **generation** of an object. A generation captures a series of successively spawned tasks that apply the same annotation to the object. An exception are the output and in/out annotations, where only one task is allowed per generation. Generations are ordered from old to young, where the oldest generation contains tasks that spawned first.

Generations give us the following semantics for parallel execution of the program: **Serialization between generations:** generations must execute in program order, i.e. all tasks from the oldest generation must have finished before tasks from the next generation may issue. **Parallelism within generations:** the tasks within a generation may issue in any order.

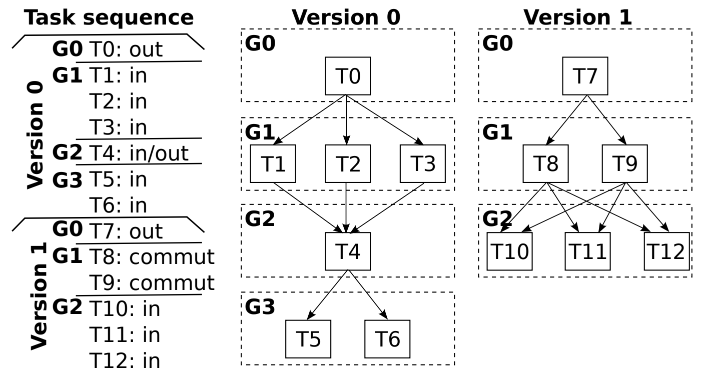


Figure 1: A task graph structured in versions and generations. All tasks access a single object.

Figure 1 illustrates the construction of generations: tasks in the same generation are independent; those in distinct generations are dependent.

Generations are a *property of objects*, so different objects may step through generations at a different rate. Real task graphs have multiple objects and are more complex than depicted in Figure 1, although we can always identify the structure presented here for each object in the program.

3.3 Renaming

We demonstrate that renaming of output annotation allows us to further optimize the task graph scheme by specializing code paths. This optimization performs renaming on output annotation, but not on in/out. If however, for some reason,[2] renaming is not possible, then the output annotation is modified to an in/out annotation. This retains correctness (an anti-dependence substitutes for an output dependence) but implies that we can always assume absence of older tasks accessing an object with output usage. We show in the evaluation that this improves the code paths measurably.

4 Task Graph Management

Preliminaries A task is stored in two separately allocated pieces of memory: the task structure and space for arguments and tags. The task structure is a fixed-size structure holding all task metadata, function pointer to call, pointer to the argument/tags block, etc. The argument/tags part stores the arguments as well as a variable amount of per-argument *tag space*. This tag space can hold any metadata specific to a scheme.

4.1 The Graph Scheme

The first task graph scheme represents the task graph as a graph by explicitly linking every task with all its direct dependents. As such, the task data structure is extended to contain a list of dependent tasks (deps), which is basically a list of outgoing edges from the task. The tag space is unused.

Figure 2 lists the issue and release actions for the graph scheme. The arguments to these actions are the object metadata and tags for the task argument, the argument annotation and the task data structure.

Upon task issue, we know that the new task is either added to the youngest generation, or a new generation is created and it is inserted there. Also, the new task must be linked to all tasks in the previous generation. To accomplish this, it suffices that this scheme stores the two most recent generations of an object in a directly accessible way (when adding a task to the youngest generation, we must know the previous generation to insert the links also). It models the two most recent generations with a list of tasks and the corresponding annotation.

When releasing a task, it is removed from the lists of tasks from each of the last two generations (lines 23–28). There is no guarantee that the task is stored on any of those lists, as it may belong to even older generations.

When all arguments have been released, we walk the list of dependent tasks (line 33) and decrement their incoming edge counter. If this counter drops to zero, the task becomes ready to execute and it is added to the ready list.

[2]SMPSS uses a fixed-size buffer to allocate renamed storage from [4]. Renaming is turned off when all rename storage has been used. In our runtime system, the programmer can prohibit renaming of specific objects.

```
issue(metadata md, tag_struct tags, annotation a, task_struct task)
    md.lock()
    // Start new generation if annotations mismatch
    if a = in/out or md.annot[md.cur_generation] != a then
        md.cur_generation := 1 − md.cur_generation
        md.tasks[md.cur_generation].clear()
        md.annot[md.cur_generation] := a
    endif
    // Add task to current generation
    md.tasks[md.cur_generation].push_back( task )
    // Link task with tasks in previous generation
    for task_struct prev_task in md.tasks[1−md.cur_generation] do
        prev_task.lock()
        ++task.incoming_count // atomic add
        prev_task.deps.push_back( task )
        prev_task.unlock()
    done
    md.unlock()
release(metadata md, tag_struct tags, annotation a, task_struct task)
    // Remove the task from the generation list that contains it
    // (if it is in any list). Avoid scanning lists where possible.
    md.lock()
    if md.annot[0] = a then
       md.tasks[0].delete_task( task )
    endif
    if md.annot[1] = a then
       md.tasks[1].delete_task( task )
    endif
    md.unlock()

    // Wakeup dependent tasks − this is executed once per task
    task.lock()
    for task_struct dep_task in task.deps do
        if −−dep_task.incoming_count = 0 then // atomic substract
            ready_list.push_back( dep_task )
        endif
    done
    task.unlock()
```

Figure 2: Actions on a task graph represented as a graph. Note that task issue for output dependences uses specialized code where the tasks and annot fields are updated unconditionally and the linking step is skipped.

The graph scheme shows inefficiency in a number of cases. Assume that a generation of M tasks is followed by a generation of N tasks. Issuing a task in the first generation is constant-time, but issuing a task in the second generation takes $O(M)$ time to insert outgoing edges in each task in the first generation. Thus task issue takes $O(MN)$ steps for $M + N$ tasks.

More importantly, releasing a task in the first generation takes $O(M+N)$ time as the previous generation list must be traversed to delete the task ($O(M)$), and because all outgoing edges must again be traversed ($O(N)$). Releasing a task in the second generation takes $O(N)$ time to delete the task from the generation lists. Thus, release takes $O((M+N)^2)$ time steps for $M+N$ tasks. Both issue and release introduce a superlinear overhead when $M > 1$ *or* $N > 1$.

It is important to keep in mind that many research works adopt the graph scheme or a variation of it [4, 3, 1] that is subject to the same flaws.

4.1.1 Optimization: Embedding Lists in Tags Storage

Task release time is influenced strongly by the deletion of a task from the "cur" and "next" generation lists. The graph scheme with embedded lists provides a $O(1)$ method to delete a task from the generation lists.

Assuming a circular doubly-linked list implementation, we provide direct access to the linked list node corresponding to a particular task. Note that a task is inserted on one generation list for every task argument. As such, we provision space for a doubly-linked list node in the tags space of the task, one node for each argument. The location of this node in memory is directly inferrable from its position in the argument list.

The list node contains previous/next link fields and a pointer to the task. It is initialized during issue (Figure 2, lines 6 and 10). Deleting a task from the list (lines 23– 28) now takes $O(1)$ steps as the linked list must no longer be traversed to find the node.

Reconsidering the example with a generation of M tasks followed by a generation of N tasks, we calculate that task release now takes $O(MN)$ time for $M + N$ tasks. There is still a superlinear overhead when $M > 1$ *and* $N > 1$. Clearly not a solved problem, but an improvement over the graph scheme.

4.2 The Hypergraph Scheme

Performance anomalies disappear when we exploit the generational structure of the task graph. Namely, all tasks in one generation have an outgoing edge to the tasks in the next younger generation. These edges between tasks can be summarized by edges between generations. Such a construction is a directed hypergraph [5], hence the name of this scheme.

The hypergraph scheme uses an explicit data structure to represent a generation. This data structure contains a list of tasks in the generation, a count of the number of tasks and a pointer to the next younger generation. The generation data structure also contains a mutex variable to control concurrent accesses.

The metadata for an object now consists of a pointer to the current and previous generations of that object. Older generations may exist, but are not referenced from the object metadata. The object metadata does not need mutually exclusive access because it is only accessed by the thread that issues tasks. Generations are protected by a lock and the locking order is from older to younger generations.

Figure 3 shows the issue and release algorithms. Where the graph schemes (Figure 2) store two generations and reuse this storage, the hypergraph scheme dynamically allocates new generations as they are created (line 5). Note that the incoming edge count for a task is incremented by one per incoming hyper-edge (line 13), reducing the book-keeping overhead.

```
issue(metadata md, tag_struct tags, annotation a, task_struct task)
    if ( a = in/out or md.cur.annot != a ) and md.cur.num_tasks > 0
    then
        md.prev := md.cur
        md.cur := new generation( annot := a, next := null )
        md.prev.next := md.cur
    else
        md.cur.annot := a
    endif
    if md.prev != null then
        md.prev.lock()
        md.cur.lock()
        task.incoming_count++ // atomic add
        if md.prev.num_tasks > 0 then
            md.cur.tasks.push_back( task )
        endif
        ++md.cur.num_tasks
        md.cur.unlock()
        md.prev.unlock()
    else
        // Don't need to build the list , it will not be traversed
        md.cur.lock()
        ++md.cur.num_tasks
        md.cur.unlock()
    endif
    tags.gen := md.cur
release(metadata md, tag_struct tags, annotation a, task_struct task)
    tags.gen.lock()
    --tags.gen.num_tasks
    // If there is a next generation, then wakeup all those tasks
    if tags.gen.num_tasks = 0 and tags.gen.next then
        tags.gen.next.lock()
        for task_struct t in tags.gen.next.tasks do
            if --t.incoming_count = 0 then // atomic substract
                ready_list .push_back( t )
            endif
        done
        // We do not need the list from now on
        tags.gen.next.tasks.clear()
        tags.gen.next.unlock()
    endif
    tags.gen.unlock()
```

Figure 3: Issue and release for the hypergraph.

This scheme uses the tag space to store a pointer to the generation of each argument that is used by this task (line 26).

Again, issue for an output annotation has a specialized code path that builds on the knowledge that both the previous and the current generation are empty. The only necessary actions are to set the annotation (line 8) and the task count (line 23) of the current generation.

During task release, we decrement the number of tasks in the generation. When the currently oldest generation becomes empty, then we try to wakeup each of the tasks in the next younger generation (lines 33-37). The scheme does not investigate the next younger generation as long as the oldest generation contains tasks. As such, it traverses the list of tasks in a generation only once per generation, as opposed to once per task in the graph scheme.

Furthermore, by careful structuring of the code, we obviate the need to delete a task from its generation during release. In fact, during release we erase the list of tasks at once when a generation is woken-up (line 39). After this, the list of tasks is no longer needed, neither for issue nor release. Corner cases require that a generation's task list is not updated when the previous generation is empty (line 14) or non-existent (line 20).

This design choice has a huge consequence on the time complexity of the issue and release steps. If we reconsider the example of the task graph with generations of M and N tasks, respectively, we find that the time complexity of issue and release is now $O(1)$ when amortized over all tasks in the task graph. Consequently, embedding the storage of linked list nodes in the tag space of the task arguments now does not improve the performance scalability of the scheme, although it will have some benefit due to a reduction in memory allocations.

4.3 The List Scheme

Optimizations to the generational scheme have demonstrated several principles that allow an organization around lists of tasks rather than generations. This results in a simpler organization of the task graph, but maintains the parallelism.

The algorithm is presented in Figure 4. The key insights behind the list scheme are:

1. At any one time, the task graph is operating on only two generations: the oldest generation, consisting of the tasks that are currently ready to execute (as far as this particular argument is concerned) and the youngest generation, where newly issued tasks are added.

2. For the oldest generation, we only need to know how many tasks are still executing (Figure 4, Line 23). When the last task in a generation has finished we need to wake up tasks in the next generation (Line 33).

3. For the youngest generation, we only need to know the annotation of the tasks, in order to switch to a new generation whenever the annotation changes or demands so (in case of in/out usage) (Line 10).

4. All tasks operating on an object may be kept on a single ordered list, provided that a marker is inserted on the last task in a each generation. Task issue appends tasks to the tail of this list. During task release, when moving to a new generation, we traverse the task list until the end-of-generation marker is reached, or the list runs empty. The task count in the oldest generation is updated with the number of tasks encountered.

5. Sometimes, the youngest and oldest generations are the same. This special situation must be properly accounted for by updating the task count in the oldest generation during task issue when there is one generation in the task graph (Line 23). It also requires resetting of the annotation in the youngest generation when the task graph runs empty (Line 48).

Accesses to the task graph metadata in the list scheme must be protected by mutual exclusion as in the other edge-centric schemes. We have experimented with a version of the scheme with a single lock and one with two locks. In the first version, the lock protects both the youngest and oldest generations. In the second version, distinct locks are used to provide access to the youngest and oldest generations. However, both locks must be acquired when there are two or fewer generations. Overall, we found the single-lock variation more performant. We attribute this to the fact that our scheduler executes most often in a scenario where only part of the task graph is instantiated at a time.

4.4 The Tickets Scheme

The ticket scheme was originally described for a model with input, output and in/out annotations [25, 26]. In this paper, we make a minor extension to the scheme to include commutative in/out and reductions. We provide a slightly different exposition of the algorithm which is rooted in our terminology of generations.

The scheme resembles a ticket-based queuing system such as operated in, e.g., a butcher's store. The ticket queuing system uses two counters: a global counter and a next counter. The next counter serves to serialize all clients: each client gets a successive value of the next counter, which is called a *ticket*. The global counter is incremented for every client that has been served. At any time, the client to be served is the one whose ticket equals the global counter.

The ticket scheme that implements task graphs uses 4 ticket queues per version of an object called R (input), W (output and in/out), C (commutative in/out) and P (reductions). We use different ticket queues per annotation because we need to synchronize *between* generations but *not within* generations. When a task issues, it increments the next counter in the queue that corresponds to its annotation and it copies the next counter from each of the other queues (it takes 3 tickets). It becomes ready for execution when the global counters in the other queues match the corresponding tickets. This means that any task with a different annotation and that issued earlier must have finished execution before the task can execute. There is however no synchronization with other tasks with the same annotation, because we want these tasks to execute in parallel.[3] After serving a task, the global counter in the queue that matches the task's annotation is incremented.

Tasks with in/out annotations must also serialize with other tasks with in/out annotation. We enforce this synchronization by making these tasks wait also on the W queue, such that they take/return tickets from the same queue that they wait on.

Figure 5 shows the issue and release operations under the ticket algorithm. The object metadata consists of 8 counters (4 ticket queues of 2 counters each). The tag space for an argument is used to store ticket values that must be waited on. As such, the tags for an input, commutative and reduction annotation contain 3 values, the tags for an output annotation are empty and the tags for an in/out annotation contain 4 values.

The issue and release actions must not execute in exclusion from related actions on the same versions, provided that every increment in the release operations is performed atomically. Increments in the issue operations need not be atomic because the issue and release actions modify different variables and because the task graph is generated by a single thread of execution.

The ticket scheme has one peculiarity: as tasks do not contain pointers to the task that they may wake up, it is not possible to populate a ready list. Instead, the ticket scheme keeps all tasks in a pool and searches through this pool for ready tasks when a new task to execute is

[3]Note that tasks with the same annotation and that belong to different generations must be synchronized. This is accomplished by transitivity: there must be a generation in between with a different annotation which is synchronized with the other two generations.

```
issue(metadata md, tag_struct tags, annotation a, task_struct task)
    tags.task := task // tags are nodes in a linked list
    tags.last_in_generation := false
    tags.next := null
    md.lock()
    if md.num_gens = 0 then
        md.num_gens := 1
        md.oldest_num_tasks := 1
        md.youngest_annot := a
    else if ( a = in/out or md.youngest_annot != a ) and md.youngest_annot != empty then
        md.youngest_annot := a
        ++md.num_gens
        ++task.incoming_count // atomic add
        if md.tasks.tail != null then // append to linked list
            md.tasks.tail.last_in_generation := true
            md.tasks.tail.next := tags
        else
            md.tasks.head := tags
        endif
        md.tasks.tail := tags
    else
        if md.num_gens = 1 then
            ++md.oldest_num_tasks
        else
            ++task.incoming_count // atomic add
            md.tasks.tail.next := tags // append to linked list
            md.tasks.tail := tags
        endif
    endif
    md.unlock()
release(metadata md, tag_struct tags, annotation a, task_struct task)
    md.lock()
    if --md.oldest_num_tasks = 0 then
        if md.tasks.head != null then
            do
                tag_struct t := md.tasks.head
                if --t.task.incoming_count = 0 then // atomic substract
                    ready_list.push_back( t.task )
                endif
                ++md.oldest_num_tasks
                md.tasks.head := t.next
            while t.last_in_generation = false and md.tasks.head != null
        endif
        --md.num_gens // generation moved to ready list
        if md.tasks.head = null then
            md.tasks.tail := null
            if md.num_gens = 0 then
                md.youngest_annot := empty
            endif
        endif
    endif
    md.unlock()
```

Figure 4: Issue and release for the scheme organized around a task list.

```
issue input(metadata md, tag_struct tags, task_struct task)
    ++md.R.next // take ticket
    tags.w := md.W.next // copy ticket
    tags.c := md.C.next // copy ticket
    tags.p := md.P.next // copy ticket
ready check input(metadata md, tag_struct tags, task_struct task)
    return tags.w = md.W.global and tags.c = md.C.global
        and tags.p = md.P.global // wait (3 queues)
release input(metadata md, tag_struct tags, task_struct task)
    ++md.R.global // return ticket

issue output(metadata md, out_tags tags, task_struct task)
    ++md.W.next // take ticket
ready check output(metadata md, out_tags tags, task_struct task)
    return true // output never waits
release output(metadata md, out_tags tags, task_struct task)
    ++md.W.global // return ticket

issue in/out(metadata md, inout_tags tags, task_struct task)
    tags.r := md.R.next // copy ticket
    tags.w := md.W.next++ // copy and take ticket
    tags.c := md.C.next // copy ticket
    tags.p := md.P.next // copy ticket
ready check in/out(metadata md, inout_tags tags, task_struct task)
    return tags.r = md.R.global and tags.w = md.W.global
        and tags.c = md.C.global and tags.p = md.P.global // wait
release in/out(metadata md, inout_tags tags, task_struct task)
    ++md.W.global // return ticket
```

Figure 5: Actions for the tickets scheme. The commutative in/out and reduction annotations are similar to the case of input annotation.

demanded [26]. To this end, a ready check must be defined for every argument annotation. This ready check consists of comparing the appropriate global counters in the object metadata to the tickets stored in the tag space of an argument. The tasks in the pool are organized by their depth in the task graph in order to limit overhead when searching for a successor. The pool is organized as a resizable hash table with chaining where each chain stores the tasks with a particular depth.

5 Evaluation and Analysis

We evaluate the task graph schemes using micro-benchmarks. The evaluation platform is a 48-core AMD Opteron 6172 at 2.1 GHz. The codes are compiled using gcc 4.7.3 at optimization level -O4.

We measure time in the micro-benchmarks using the cpuid/rdtsc instruction sequence to measure delay in cycles. Although this instruction sequence disturbs execution time somewhat by serializing all instructions, it is an appropriate way to measure time delays on the order of 100s of cycles. Moreover, we measure multiple occurences of an event and present the average delay over those events to mitigate this source of imprecision.

The task graph schemes determine what tasks are ready to execute. The scheduler must choose which of these tasks to execute and on what core. These are two distinct problems. We assume a greedy scheduler in this work, such that the first task on the ready list is scheduled on the first core that becomes idle. The scheduler is a work-stealing task dataflow scheduler [26] that implements the work-first principle, as in [16].

5.1 Fast Execution Path: All Tasks are Ready

We constructed a micro-benchmark that measures the fastest path in our system to spawn a task. It contains a single thread of execution that spawns and immediately executes a number of one-argument tasks. All tasks are passed the same object. Task issue operates on an empty task graph and task release never wakes up dependent tasks. The micro-benchmark by-passes queueing of tasks as our system immediately executes a task when it is ready at the moment of the spawn. This behavior is similar to Cilk's work-first principle [16], which is implemented in our scheduler. Consequently, this fast execution path is an important and common behavior, also for dependence tracking.

Figure 6 shows the delay of the fast execution path measured in processor clock cycles, averaged over 10 million spawns. We show results for the task graph schemes discussed in this paper and all supported annotations. The spawn delays are broken down in three parts: the baseline spawn time equals the time to spawn and execute an empty task that has no arguments. The generic overhead is the overhead that is incurred due to task dependence tracking. These overheads are the largest in the graph schemes; they are smallest in the tickets scheme. Embedding generation lists hardly affects the delay of the fast execution path because the generation lists contain at most one task.

The annotation-specific overhead is the overhead that is introduced to track a particular dependence type on top of the generic infrastructure. Output dependences incur the least annotation-specific overhead because task issue is optimized to the situation where no prior tasks are operating on the same object. The commutative in/out annotation incurs somewhat more overhead than the in/out annotation because the runtime system implements exclusion by means of a mutex that is stored in the object metadata. The reduction annotations are also more expensive because of the management and reduction of private versions of the object.

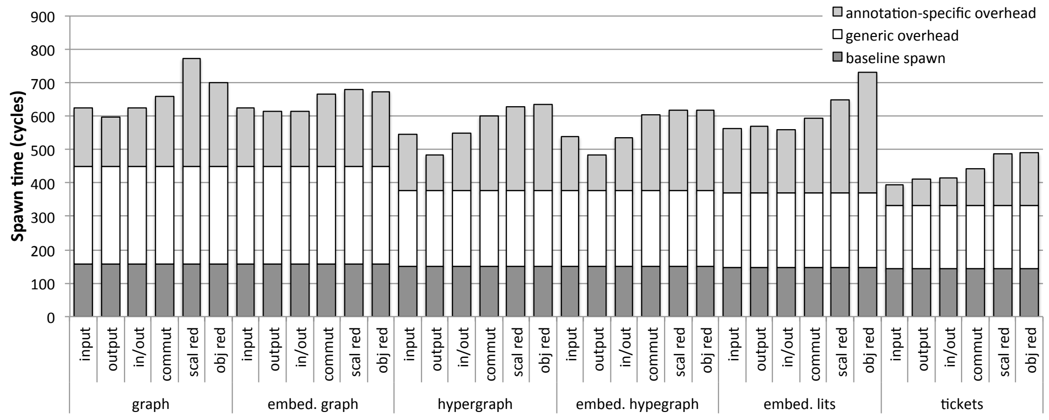


Figure 6: Delay of fast execution path when spawning and immediately executing a task with one ready argument. The abbreviations 'commut' stands for the commutative annotations, 'scal red' is a reduction using serial accumulation of results while 'obj red' is a reduction using parallel tree reduction.

Overall, the tickets scheme introduces the least generic overhead, and the least annotation-specific overhead. The task spawn delay in the tickets scheme can be as low as 400 cycles or 0.20μs on our evaluation machine.

5.2 Slow Execution Path: All Tasks Require Wakeup

The slow path for executing a task involves an initial ready check, issue, execute and release. We measure the slow path with a micro-benchmark that artificially inserts an active task in the metadata of an object and then spawns N tasks that access the same object. These tasks have a variable annotation (output, in/out, etc.) but perform no work. Next, the micro-benchmark releases the artificial task and then waits for all issued tasks to wakeup and complete. This process is repeated several times to capture variance on the measurements. The micro-benchmark uses one thread of execution to factor out multi-threading overheads.

Figure 7 shows the spawn overhead in this micro-benchmark for a representative set of the annotations. Other annotations lead to similar results. The number of tasks in a generation is varied on the horizontal axis. The vertical axis shows the average delay per task spawn. These results confirm that the graph scheme introduces a super-linear overhead when $N > 1$ (see Section 4.1). Moreover, the embedded graph scheme does not suffer super-linear overheads because the first generation consists of just one task ($M = 1$). The hypergraph schemes have no scalability problems.

Note that $N = 1$ for output and in/out annotations because a new generation is constructed for every task. Thus, for this micro-benchmark, all graph scheme are scalable. Also, renaming is applied for the output annotations, so the tasks are spawned at a speed close to the fast-path speed.

Three Generation Variant of the Micro-Benchmark We also constructed a micro-benchmark with 3 generations: one in/out task that is artificially delayed, a sequence (generation) of N tasks with input annotation and a sequence of N tasks with variable annotation. Figure 8 illustrates the shape of the task graph of this micro-benchmark. Figure 9 shows the performance of the

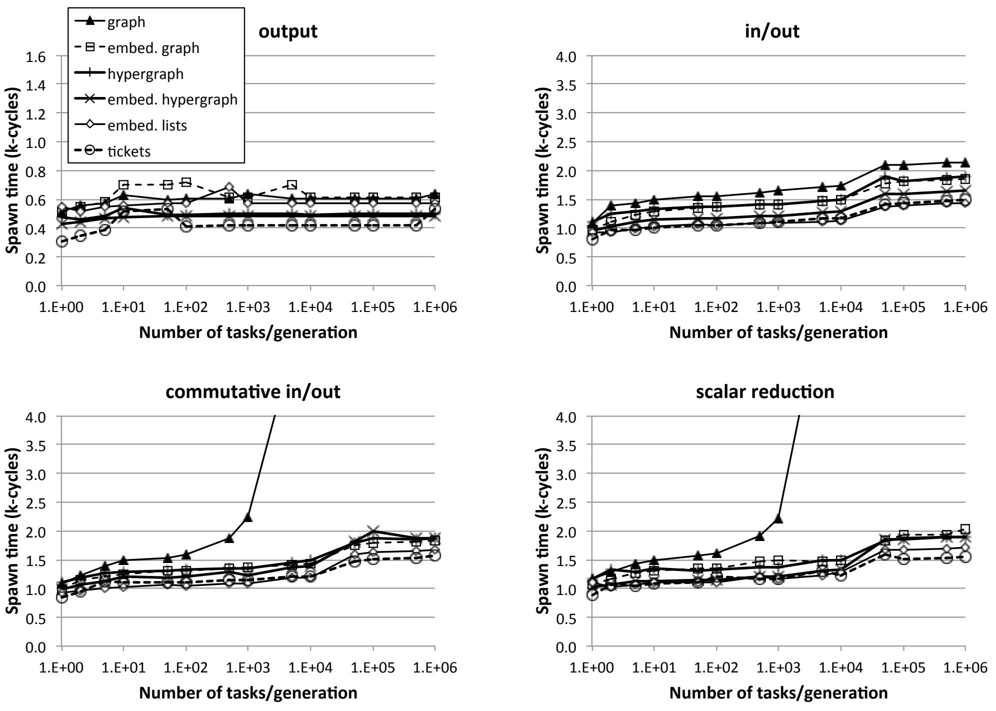


Figure 7: Spawn time for the slow-path microbenchmark with two annotations.

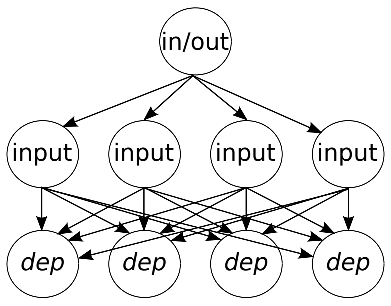


Figure 8: Task graph constructed in the three-generation slow path micro-benchmark for $N = 4$.

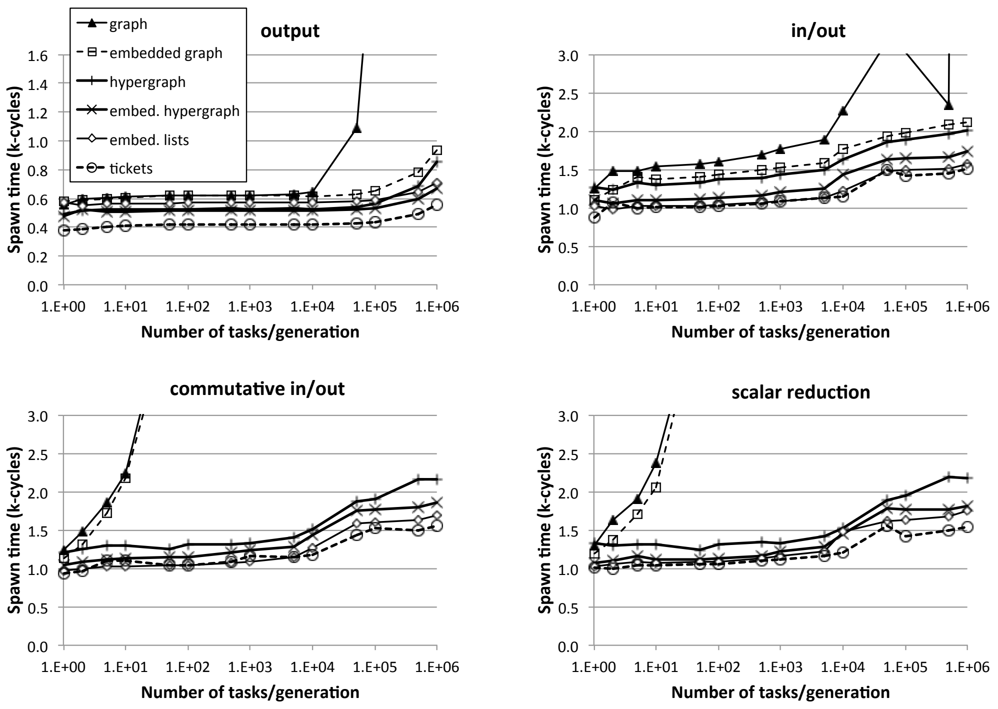


Figure 9: Spawn time for the slow-path microbenchmark with three generations.

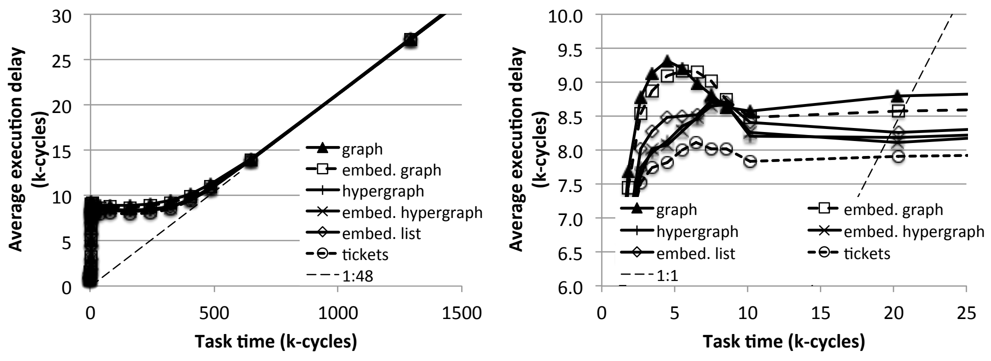


Figure 10: The impact of task size on scalability when executing with 48 threads. The graph on the right zooms-in on the graph on the left.

task graph schemes for a representative set of the annotations.

The graph scheme shows a scalability problem on output and in/out annotations when the number of tasks approaches 10,000. This is due to deleting the input tasks from the generation lists.

With the commutative in/out and scalar reduction annotations, the task graph is structured as two generations of N tasks each. In this case, our analysis (Section 4.1.1) has already shown that the graph and embedded graph schemes are not scalable.

Overall, the tickets scheme introduces least overhead. The hypergraph schemes perform well also and introduce up to around 30% and 15% slowdown over the tickets scheme for the hypergraph and embedded hypergraph schemes, respectively, when $N \leq 1000$. The embedded lists scheme has an average overhead of 5% compared to the tickets scheme.

5.3 Slow Execution Path: Work Stealing

We measure the runtime system overhead in the presence of work stealing using a micro-benchmark that issues several thousand tasks with an input annotation. We pass the same object to each task's argument. We execute this benchmark using 48 cores. Figure 10 shows the observed delay of a task spawn in the micro-benchmark. We vary the size of the tasks on the horizontal axis. In each case, the number of tasks is chosen such that the total execution time remains in the 1-10s range.

The results show that the scheduler obtains linear scaling at a task size of about 190–230μs (around 400K cycles). The task graph scheme used has little impact hereon. This is expected, as the task graph schemes show performance differences in the sub-microsecond range.

The task graph scheme has a noticeable impact when the scheduler does not achieve linear scaling. In this case, the tickets scheme is 10-20% faster than the graph scheme and up to 8% faster than the embedded lists scheme. Note that the majority of the 8000-cycle task execution delay measured here is due to the scheduler and the way it performs task stealing. This is not affected by the task graph scheme.

We have performed this experiment also when using up to twice as many objects as there are threads. Objects are accessed in round-robin manner such that subsequently spawned tasks operate on distinct objects. The results are similar to Figure 10, showing that the objects and the taskgraph data structures are not a point of contention.

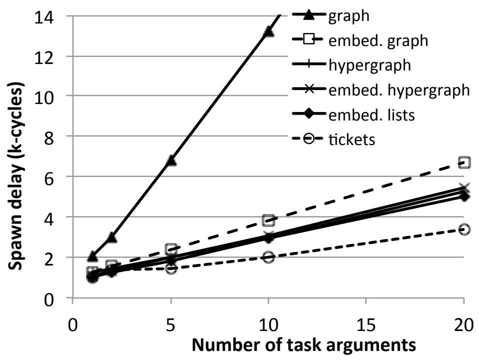


Figure 11: Varying number of task arguments.

5.4 Number of Task Arguments

Figure 11 shows the spawn delay of the slow-path micro-benchmark where the number of arguments per task is varied. A different object is passed to each argument. The overhead of the edge-centric schemes becomes more pronounced as tasks have more arguments. At 20 arguments, the best edge-centric scheme is over 30% slower than the tickets scheme. The graph scheme shows unacceptable slowdowns.

While 20 arguments for a task seems high, it is not unreasonable as the programming model discourages the use of global variables. Global variables cannot participate in dataflow dependences. Our implementation of the PARSEC Blackscholes benchmark, e.g., has a task with 8 arguments of which 7 are annotated. Here, the tickets scheme already outperforms the other schemes by 25%.

5.5 Micro-Benchmarks with Realistic Task Graphs

We evaluate the task graph schemes on micro-benchmarks that exhibit the same task graph as real codes: matrix multiply and cholesky factorization. The actual tasks in these micro-benchmarks are configurable, as in the experiment of Section 5.3.

5.5.1 Task Graph: Matrix Multiply

The matrix multiply mimicked is a block matrix multiply where the result matrix is divided in DxD blocks of a fixed size, in this case 16x16 blocks (Figure 12). For matrices of DxD blocks, the program generates D^3 tasks, where disjoint groups of D tasks update the same block of the C matrix, which causes serialization of these tasks. As there are D^2 distinct blocks in the C matrix, the task graph shows D^2 strands of D tasks each. In the experiments presented here, $D = 128$. Due to the regularity of the taskgraph, matrix multiply is not a key motivator for task dataflow languages. This benchmark is, however, helpful for performance analysis.

Figure 13 shows the average task execution time for varying amounts of work per task and also for two loop orderings of matrix multiply. On the left hand side the 'ijk' loop order is used. Here, the inner-most k-loop computes the product for one block of the result matrix. Tasks are traversed one strand of the task graph at a time. Consequently, each task in the inner loop is dependent on the previous one and the scheduler must scan ahead in the program to the next iteration of the j-loop to expose parallelism. Based on the properties of the scheduler, we can calculate that execution of this program uses $O(D^2)$ work stealing events, i.e., one per strand

```
typedef float (*block_t )[16]; // 16x16 tile
typedef versioned<float[16][16]> vers_block_t ;

void matmul(vers_block_t A [], vers_block_t B [],
            vers_block_t C [], unsigned D) {
    for( unsigned i=0; i < D; ++i ) {
        for( unsigned j=0; j < D; ++j ) {
            for( unsigned k=0; k < D; ++k ) {
                spawn mul_add( indep A[i*D+k],
                               indep B[k*D+j],
                               inoutdep C[i*D+j] );
            }
        }
    }
    sync;
}
```

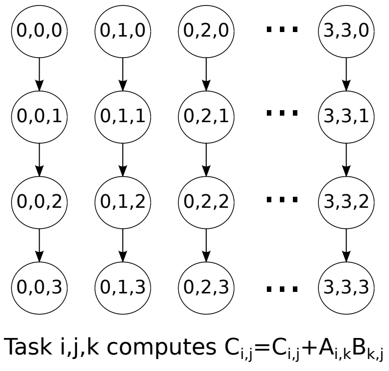


Figure 12: Left: Task dataflow program for matrix multiply over DxD matrices of 16x16 blocks using an 'ijk' loop order. Adapted from [26]. Right: The task graph of matrix multiply when D=4.

in the task graph. Note that work stealing is significantly more expensive than creating and initiating a pending task.

With the 'ijk' loop order, the graph scheme shows a performance anomaly where tasks shorter than 150 k-cycles incur increased delay. The tickets scheme performs worse than the other schemes. As the structure of the program forces construction of nearly the full task graph, additional pressure is put on the pool to locate the next ready task.

On the right hand side in Figure 13, a 'kij' loop order is used. This loop order launches one task for every block in the result matrix before launching the second task on the first block. As such, tasks are traversed by visiting each strand in the task graph in round-robin fashion and the program has a great deal of readily exploitable parallelism. However, this task traversal order also implies that $O(D^3)$ work stealing events are used to execute the program, assuming that $D^2 \gg P$, where P is the number of processors.

The tickets scheme outperforms the other task graph schemes, typically by about 400 cycles or 3.5% of the total task delay. Note that the average task delay is in the best case about 4600 cycles larger for 'kij' loop order than for the 'ijk' loop order. This is a side effect of the design of the scheduler and the cost of work stealing, in combination with an increase in work stealing events by $O(D)$.

We have not shown the embedded graph and hypergraph schemes as they perform nearly as well as the embedded lists scheme.

5.5.2 Task Graph: Cholesky Factorization

Figure 14 shows the performance of the task graph schemes on the task graph for Cholesky factorization of a matrix divided in 128x128 blocks. The results confirm previous findings: the tickets scheme is inherently faster than the edge-centric task graph schemes at very small task sizes. It outperforms the embedded lists scheme by 10-15% for task times of 250k cycles and less. However, Cholesky factorization has a complex task graph and the pool fails to locate the next ready task sufficiently fast. For 500k cycle tasks, the tickets scheme is 3% slower than the

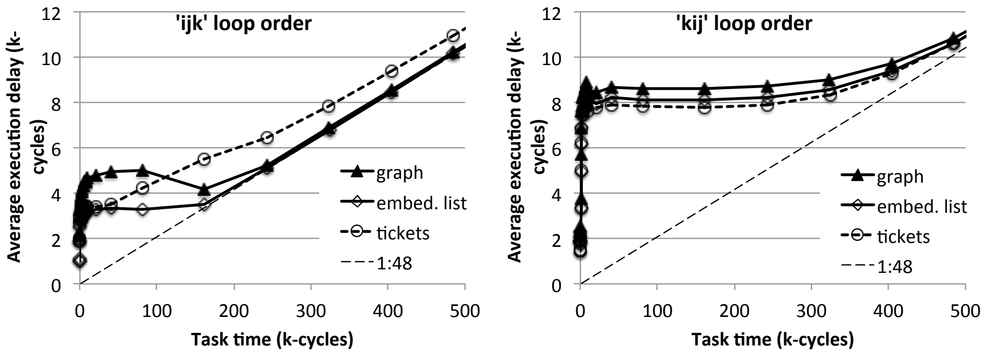


Figure 13: Impact of the task graph schemes on the performance of blocked matrix multiply.

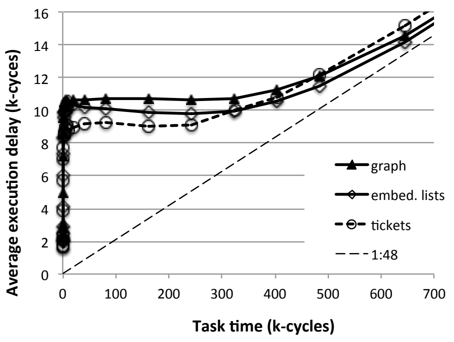


Figure 14: Impact of the task graph schemes on the performance of the cholesky facctorization skeleton.

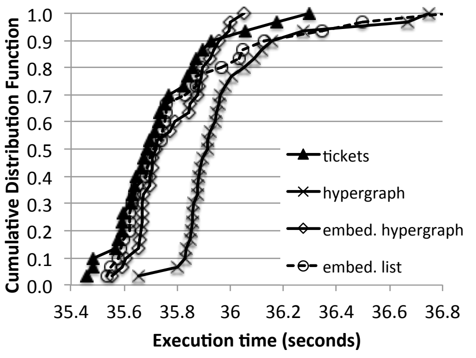
task graph schemes. The performance of all schemes converges as task time increases.

5.6 Benchmarks

We study the impact of performance on two benchmarks with higher complexity, namely ferret and dedup, taken from the PARSEC benchmark suite [7]. We have analysed the performance of these benchmarks using our runtime system in other work [24], where we have also proposed new programming language concepts to accelerate these applications and improve their programmability. In this work, we evaluate the baseline versions using input, output and in/out annotations.

5.6.1 Ferret

The parallelism exploited in ferret consists of a 5-stage pipeline where images are processed and compared in successive steps. Ferret's performance scales nearly linearly with an increasing number of cores [24]. Ferret has fine-grain tasks, which causes differences between the dependence tracking schemes to show up in the overall execution time. Figure 15 shows the cumulative



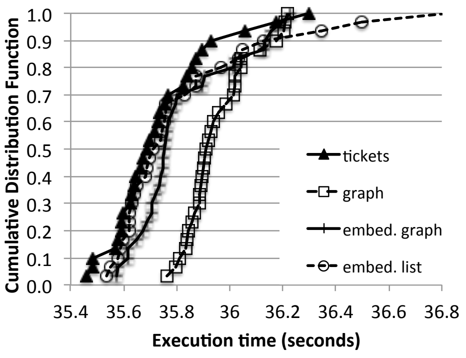


Figure 15: Cumulative distribution function of the execution times of ferret over 30 executions, when using each of the dependence tracking schemes. The left and right graph show different subsets of the results. The graph and hypergraph schemes incur similar execution times, causing their curves (shown in distinct graphs for clarity) to overlap.

distribution function of overall execution time of ferret on 48 cores over 30 runs, when executing the native input set supplied with PARSEC.

The tickets scheme results in lower execution times than the other schemes, although the difference with the lists scheme is negligible. The main result of this experiment is that the graph and hypergraph schemes are clearly slower than their embedded graph and embedded hypergraph counterparts. Given the overlap of measured execution times between different schemes, we validate this result statistically.

We use the two-sided Wilcoxon rank sum test [12], a non-parametric test, to establish equality of means. We test the null hypothesis that two schemes yield the same mean execution time with a significance level of 5%. The difference between the tickets scheme and the graph scheme is statistically significant ($\mu = 35.74s$ and $\sigma = 0.19s$ for tickets vs. $\mu = 35.96s$ and $\sigma = 0.13s$ for the graph scheme) with a Wilcoxon value of $W(30) = 132$ and $p = 2.58e - 6$, with failure to accept the null hypothesis. A similar statistically significant difference can be measured between the tickets and hypergraph schemes.

The tickets scheme does not differ from the remaining schemes in a statistically significant way. E.g., comparing to the embedded graph scheme ($\mu = 35.82s$, $\sigma = 0.19s$) we find $W(30) = 321$ and $p = 5.75\%$, so we cannot reject the null hypothesis.

In summary, we find that the graph and hypergraph schemes are slower than the other schemes with statistical significance, but there is no statistically significant difference in average performance between the tickes scheme and the embedded variants. These results demonstrate that the embedding optimisation is crucial as it minimizes the operations in the dependence tracking scheme, especially those involving extra memory alloations and de-allocations.

5.6.2 Dedup

Dedup is an application with nested pipelines, i.e., one pipeline stage in the outer pipeline is decomposed as another pipeline with finer-grain parallelism [7]. As such, dedup is an example of a program with *nested task graphs*. The version we test here is slightly different from the baseline version in [24], as we have collapsed three stages in the inner pipeline (hash computation, hash table access and compression) as this gives better performance.

Table 2: Range of execution times measured for 30 runs of dedup when executing on 16 cores, using each of the dependence tracking schemes. Threads are packed and pinned on the cores of 3 processors.

	mean (s)	std.-dev. (s)	min (s)	max (s)
tickets	6.60	0.27	6.11	7.23
graph	6.65	0.23	6.33	7.23
embedded graph	6.71	0.29	6.13	7.45
hypergraph	6.72	0.34	6.25	7.75
embedded hypergraph	6.60	0.42	5.97	7.47
embedded lists	7.22	1.42	6.24	10.85

The performance ranges of dedup are summarised in Table 2 when 16 cores are utilized. Dedup quickly runs out of parallelism. The measurements on 16 cores reflect a situation where threads are essentially idle and engaging in work stealing. We observe that the performance of dedup is independent of the dependence tracking scheme. Although some difference on the average execution time can be observed, these are much less than the standard deviations. As such, the Wilcoxon test does not validate these differences as statistically significant.

All schemes have fairly heavy tails which is probably typical for programs engaging frequently in locking. Some executions with the embedded lists scheme are much slower (up to 73% larger than the shortest execution time). We do not have enough data to conclude whether this is an artefact of the implementation or whether it is typical for the embedded lists scheme, as it only shows up with this severity in dedup.

6 Related Work

6.1 Programming model

The literature presents many variations of the task dataflow model. Several papers use the basic model with input and output annotations [1, 17], supplemented with the in/out annotation [10]. CnC considers only inputs and outputs and calls them producers and consumers [9]. What SMPSS calls a reduction [4] is in fact commutativity in our model, except that SMPSS requires the programmer to lock the reduction variable when it is accessed. The annotation is renamed to 'concurrent' in OmpSs [14], the successor to SMPSS. In this work it is the runtime that ensures exclusive access. This work also implements a real reduction annotation that automatically manages multiple copies of the reduction variable. In this respect, the annotation is as strong as a Cilk++ reducer [15]. Cilk++ reducers allow non-commutative operators, but our system assumes commutativity for practical reasons.

6.2 Task Graph Management

Most papers that discuss runtime systems for task dataflow languages are vague about the implementation of the dependence tracking algorithms. The underlying rationale being that there is no risk of scalability problems, or the assumption that the advantages of exposing additional parallelism will always outweigh its cost. This paper has shown that these assumptions are false: there is a risk of lack of scalability if the task dependence tracking is implemented using a graph representation of the task graph.

We were able to determine through analysis of source code that some systems construct a graph that explicitly contains edges for every pair of dependent tasks [4, 3, 1], as in the graph scheme. As such, they are prone to the pitfalls of this scheme.

SuperMatrix implements Tomasulo's algorithm in software [10]. The presented scheme is limited to one output or one in/out argument per task and does not support commutativity or reductions. As such, the programmability of the system is reduced. More importantly, Tomasulo's algorithm as designed with a hardware implementation in mind. This imposes certain restrictions on the algorithm that are irrelevant for a software implementation.

OoOJava [19] uses heap dependence queues. These queues operate using generations as in this work. It is however more complex as generations may be organized as hash tables, where each bucket in the hash table holds a chain of dependent tasks. Chains in different buckets may execute concurrently.

Best et al. [6] present a generation-based scheme with one program-wide queue. Tasks are added to a generation when they are proven to have non-overlapping memory footprints; otherwise a new generation is started. The scheduler processes one generation at a time. This, however, limits parallelism when generations are small and is prone to unbalance within generations.

Gupta and Sohi [17] present a scheme where each object has a wait-list of tasks that will operate on that object. This is a variation on the list scheme. However, their scheme requires that tasks with an input dependence issue in program order (although they may complete in any order). This may reduce parallelism measurably. This restriction is necessary because they do not move tasks in the oldest (ready) generation to a ready list.

Instead of following edges in a graph, some schemes scan the list of all spawned tasks to find common arguments [20]. This consumes time proportional to the size of the task graph and appears necessary only for handling aliased task arguments [22].

7 Conclusion

This paper studies data structures and algorithms for the efficient implementation of task dataflow scheduling. Efficiency is key in the context of strong scaling, where scheduler overheads must be minimized. Our algorithms are based on the notions of *versions* and *generations* of objects. We defined schemes based on graphs, hypergraphs and lists. We also presented an optimization to embed a linked list of tasks in the task representation which dramatically reduces runtime system overhead. Besides the three edge-centric schemes, we analyzed an edge-less scheme based on ticket locks.

An unanticipated result of our evaluation is that a straightforward implementation of the task graph as a graph is susceptible to bad scaling behavior. This is true if the language supports annotations such as commutative or reduction operands.

Evaluation with micro-benchmarks consistently showed that the edge-less tickets scheme outperforms the edge-centric schemes. The tickets scheme allows single-argument task spawns in as little as 0.20μs and is at least 5-10% faster than the best edge-centric scheme. Some experiments reveal substantially larger improvements.

The tickets scheme has one shortcoming, which is in the design of the task pool. The task pool holds all pending tasks and is searched for ready tasks by idle processors. It is organized as a hash table where the depth of a task in the task graph is used as a key to speed up searches. In contrast, edge-centric schemes are able to locate ready tasks more efficiently. The task pool however only adds latency to the scheduler when it holds a large task graph. This is not always the case in our scheduler.

It is an open question whether the edge-less ticket scheme can be extended to store successor tasks efficiently, without incurring the overhead of the edge-centric schemes. Such a scheme could avoid the need for a task pool.

In future work we will study dependence tracking in the context of array sections, which would lead to partially overlapping address ranges. There exist schemes that allow this [23, 22], or where uncertainty in the address ranges plays a role [6, 19]. The overheads for these schemes has been reported as substantial, up to milliseconds per task spawn.

Acknowledgments

The authors which to express their gratitude towards Giorgis Giorgakoudis and Babis Chalios for their technical support towards this research. The research leading to these results has received funding from The Research Foundation Flanders (FWO) and from the European Communitys Seventh Framework Programme [FP7/2007-2013] under the People Programme (Marie Curie Actions), grant agreement no 327744, the ENCORE Project (http://www.encore-project.eu), grant agreement no 248647, the TEXT Project (http://www.project-text.eu/), grant agreement no 261580, the SCoRPiO Project (http://www.scorpio-project.eu/), grant agreement no 323872 and under the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC, http://www.hipeac.net), grant agreement no 217068.

References

[1] Agrawal, K., Leiserson, C. E., and Sukha, J. Executing task graphs using work-stealing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (Atlanta, GA, USA, Apr. 2010), pp. 1–12.

[2] Alvanos, M., G., T., Bilas, A., , and Nikolopoulos, D. S. Design and Evaluation of a Task-based Parallel H.264 Video Encoder for Heterogeneous Processors. In *Proceedings of SAMOS XI: International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)* (July 2011), pp. 217–224.

[3] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience 23*, 2 (2010), 187–198.

[4] Barcelona Supercomputing Center. *SMP Superscalar (SMPSS) User's Manual*, 2.2 ed., Sept. 2008.

[5] Berge, C. *Graphs and Hypergraphs.* North-Holland, 1973.

[6] Best, M. J., Mottishaw, S., Mustard, C., Roth, M., Fedorova, A., and Brownsword, A. Synchronization via scheduling: techniques for efficiently managing shared state. In *PLDI '11* (2011), pp. 640–652.

[7] Bienia, C. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, Jan. 2011.

[8] Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., and Dongarra, J. Dague: A generic distributed dag engine for high performance computing,. Tech. rep., Innovative Computing Laboratory, 2010.

[9] Budimlić, Z., Burke, M., Cavé, V., Knobe, K., Lowney, G., Newton, R., Palsberg, J., Peixotto, D., Sarkar, V., Schlimbach, F., and Taşirlar, S. Concurrent collections. *Sci. Program. 18* (August 2010), 203–217.

[10] Chan, E., Quintana-Orti, E. S., Quintana-Orti, G., and van de Geijn, R. Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures. In *SPAA '07* (2007), pp. 116–125.

[11] Chi, C. C., and Juurlink, B. A QHD-capable parallel H.264 decoder. In *Proceedings of the international conference on Supercomputing* (New York, NY, USA, 2011), ICS '11, ACM, pp. 317–326.

[12] Conover, W. J., and Iman, R. L. Rank transformations as a bridge between parametric and nonparametric statistics. *The American Statistician 35*, 3 (1981), 124–129.

[13] Dongarra, J., Beckman, P., and et al. The international exascale software project roadmap. *International Journal of High Performance Computer Applications 25*, 1 (2011), 3–60.

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

[15] Frigo, M., Halpern, P., Leiserson, C. E., and Lewin-Berlin, S. Reducers and other Cilk++ hyperobjects. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures* (2009), pp. 79–90.

[16] Frigo, M., Leiserson, C. E., and Randall, K. H. The implementation of the Cilk-5 multi-threaded language. In *PLDI '98: Proceedings of the 1998 ACM SIGPLAN conference on Programming language design and implementation* (1998), pp. 212–223.

[17] Gupta, G., and Sohi, G. S. Dataflow execution of sequential imperative programs on multicore architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (2011), MICRO-44 '11, pp. 59–70.

[18] Hennessy, J. L., and Patterson, D. A. *Computer architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann, 2003.

[19] Jenista, J. C., Eom, Y. h., and Demsky, B. C. Ooojava: software out-of-order execution. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming* (New York, NY, USA, 2011), PPoPP '11, ACM, pp. 57–68.

[20] Kurzak, J., and Dongarra, J. Fully dynamic scheduler for numerical computing on multicore processors. Tech. Rep. UT-CS-09-643, University of Tennessee, June 2009. LAPACK Working Note 220.

[21] Perez, J. M., Badia, R. M., and Labarta, J. A dependency-aware task-based programming environment for multicore architectures. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)* (Sept. 2008), pp. 142–151.

[22] Perez, J. M., Badia, R. M., and Labarta, J. Handling task dependencies under strided and aliased references. In *ICS '10* (2010), pp. 263–274.

[23] Tzenakis, G., Papatriantafyllou, A., Kesapides, J., Pratikakis, P., Vandierendonck, H., and Nikolopoulos, D. S. BDDT: block-level dynamic dependence analysisfor deterministic task-based parallelism. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2012), PPoPP '12, ACM, pp. 301–302.

[24] Vandierendonck, H., Chronaki, K., and Nikolopoulos, D. S. Deterministic scale-free pipeline parallelism with hyperqueues. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2013), SC '13, ACM, pp. 32:1–32:12.

[25] Vandierendonck, H., Pratikakis, P., and Nikolopoulos, D. S. Parallel programming of general-purpose programs using task-based programming models. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Parallelism (HotPar)* (May 2011), p. 6.

[26] Vandierendonck, H., Tzenakis, G., and Nikolopoulos, D. S. A unified scheduler for recursive and task dataflow parallelism. In *PACT '11: Proceedings of the 20th international conference on Parallel architectures and compilation techniques* (Oct. 2011), pp. 1–11.