Deterministic Scale-Free Pipeline Parallelism with Hyperqueues

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

Kallia Chronaki[*]
Barcelona Supercomputing Center, Spain
kallia.chronaki@bsc.es

Dimitrios S. Nikolopoulos
Queen's University Belfast
United Kingdom
d.nikolopoulos@qub.ac.uk

ABSTRACT

Ubiquitous parallel computing aims to make parallel programming accessible to a wide variety of programming areas using deterministic and scale-free programming models built on a task abstraction. However, it remains hard to reconcile these attributes with pipeline parallelism, where the number of pipeline stages is typically hard-coded in the program and defines the degree of parallelism.

This paper introduces hyperqueues, a programming abstraction that enables the construction of deterministic and scale-free pipeline parallel programs. Hyperqueues extend the concept of Cilk++ hyperobjects to provide thread-local views on a shared data structure. While hyperobjects are organized around private local views, hyperqueues require shared concurrent views on the underlying data structure. We define the semantics of hyperqueues and describe their implementation in a work-stealing scheduler. We demonstrate scalable performance on pipeline-parallel PARSEC benchmarks and find that hyperqueues provide comparable or up to 30% better performance than POSIX threads and Intel's Threading Building Blocks. The latter are highly tuned to the number of available processing cores, while programs using hyperqueues are scale-free.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Parallel programming; D.3.3 [**Language Constructs and Features**]: Concurrent programming structures; D.3.4 [**Programming Languages**]: Processors—*Runtime environments*

1. INTRODUCTION

Ubiquitous parallel computing aims to make parallelism accessible to a wide variety of programming areas without putting quality aspects of software at risk. It is understood

[*]Kallia Chronaki was with the Institute of Computer Science, Foundation for Research and Technology - Hellas, Greece, when this work was performed.

To appear in the 2013 International Conference for High-Performance Computing, Networking, Storage and Analysis. November 17 - 21 2013, Denver, Colorado USA.
Copyright 2013 ACM.
http://dx.doi.org/10.1145/2503210.2503233

```
struct data { ... };
void pipeline(int total) {
    versioned<data> value;
    versioned<int> fd = ...;
    for(int i=0; i < total; ++i) {
        spawn produce( (outdep<data>)value );
        spawn consume( (indep<data>)value,
                       (inoutdep<int>)fd );
    }
    sync;
}
```

Figure 1: A two-stage pipeline parallel program expressed with task dataflow.

that a *task abstraction*, where a task is a unit of computation, is a key element as it allows programmers to focus on the "What?" instead of the "How?". Moreover, determinism, or *determinism by default* [1], adds repeatability to parallel programs which is, among other things, critical for debugging and testing. Finally, *scale-free* parallel programs are performance-portable across architectures with different core counts. This is a necessity in light of the continuously growing number of cores on a chip, combined with decreasing reliability and dynamically managed power budgets.

Several research projects define deterministic and scale-free parallel programming languages [2, 3, 4, 5]. In these, most attention has gone to DOALL parallelism and fork-join parallelism. While these are important programming patterns, there is a large class of programs that contain pipeline parallelism and are within the remit of ubiquitous parallel computing. Models that are scale-free are, however, often not deterministic in the sense that the programming model neither defines nor respects a *serial elision* of the program.

Task dataflow is a deterministic and scale-free task-based parallel programming model. Programmers describe what variables are inputs and outputs to tasks, which essentially describes the task's side effects. The runtime system collects these side effects as tasks are spawned and computes the task dependence graph on the fly.

The task dataflow model is highly suitable for pipeline parallelism, as a pipeline is just one of the many patterns that the dependence graph may take [6]. Moreover, task dataflow systems provide a level of memory management that greatly simplifies writing pipeline parallel programs [7]. Figure 1 shows a simple pipeline parallel program in Swan [6], a task dataflow programming model designed as an extension to Cilk [8]. Variables that enforce dataflow dependences are defined with the **versioned** keyword which attaches facilities to them for tracking inter-task dependences. Also, automatic memory management is applied to versioned objects to break write-after-read dependences. Versioned variables may be used as procedure arguments provided they are cast to type **indep**, **outdep** or **inoutdep**, which describes side effects of reading, writing or both. The **spawn** keyword indicates that calling a task may occur in parallel with the continuation of the calling procedure, as in Cilk. The **sync** keyword blocks a procedure until all children have finished execution. The loop in Figure 1 corresponds to a two-stage pipeline where instances of the produce stage may execute in parallel as there are no dependences between those instances, while instances of the consume stage execute strictly in order due to the dependence on the **inoutdep** argument.

Task dataflow is an intuitive programming model where the pipeline pattern emerges on-the-fly as a side-effect of the code structure, rather than being designed-in. However, task dataflow has two limitations with respect to pipeline parallelism: (i) pipelines must be sufficiently coarse-grained as every stage invocation is modeled as a separately scheduled task, and (ii) each pipeline stage consumes a fixed number of elements from its predecessor and produces a fixed number of output elements [6]. This paper will address both shortcomings by introducing *hyperqueues*, a programming abstraction of queues for a task based programming language. Hyperqueues are deterministic and allow the construction of scale-free pipeline parallel programs.

Hyperqueues share commonalities with Cilk++ hyperobjects, specifically with reducers [9]. Reducers are special program variables that support *reduction* operations, i.e., they are identified by a type, an identity element and an associative reduction operation. A common example is addition over integers, but also appending to a list is an associative operation. The latter was, in fact, the main motivation for the development of reducers [9]. Reduction operations can be parallelized by creating duplicates of the reduction variable, called *views*, which are private to a task. As views are private, they are accessed without races. When tasks complete, the views are reduced to a single value in such a way that program order is respected. Moreover, Cilk++ uses a "special" optimization to reduce views only on task steals, as opposed to on all spawned tasks. Hyperqueues build on this property of reducers to perform push operations in parallel while retaining determinism.

However, hyperqueues also allow concurrent push and pop operations and are different in this respect from Cilk++ hyperobjects. To support this behavior, hyperqueues require a distinct implementation. Views are no longer private but are shared between a producing task and a consuming task. This paper shows how to design shared views that are data race free and how to ensure deterministic parallelism for programs utilizing hyperqueues.

Using hyperqueues, we parallelize several benchmarks with less programming effort than using POSIX threads or Threading Building Blocks (TBB) because synchronization is hidden in the runtime system and because the programming language does not impose a stringent format, as TBB does. Moreover, the hyperqueue version is scale-free and obtains the same or up to 30% better performance. It also outperforms task dataflow languages like [6] because the latter cannot capture varying numbers of inputs and outputs.

The remainder of this paper is organized as follows. Section 2 discusses the programming model. Section 3 discusses the internal representation of hyperqueues in the runtime system and views. Section 4 discusses how the runtime system merges views. Then, Section 5 presents programming idioms. We present an experimental evaluation in Section 6. Finally, Section 7 discusses related work and Section 8 concludes this paper.

```
 1 struct data { ... };
 2 void consumer(popdep<data> queue) {
 3     while( !queue.empty() ) {
 4         data d = queue.pop();
 5         // ... operate on data ...
 6     }
 7 }
 8 void producer(pushdep<data> queue, int start, int end) {
 9     if( end-start <= 10 ) {
10         for( int n=start; n < end; ++n ) {
11             data d = f(n);
12             queue.push(d);
13         }
14     } else {
15         spawn producer(queue, start, (start+end)/2);
16         spawn producer(queue, (start+end)/2, end);
17         sync;
18     }
19 }
20 void pipeline (int total) {
21     hyperqueue<data> queue;
22     spawn producer((pushdep<data>)queue, 0, total);
23     spawn consumer((popdep<data>)queue);
24     sync;
25 }
```

Figure 2: The simple pipeline-parallel program of Figure 1 expressed with the hyperqueue.

2. PROGRAMMING MODEL

2.1 The Hyperqueue Abstraction of Queues

Hyperqueues are a programming abstraction for queues. A queue is an ordered sequence of values. Values are added to the tail of the sequence using a push method. Values are removed from the head of the sequence using a pop method.

We define a hyperqueue as a special object in our programming language that models a single-producer, single-consumer queue. Its implementation allows tasks to concurrently push and pop values without breaking the semantics of a single-producer, single-consumer queue, and without breaking the serializability of the parallel program.

Hyperqueues are defined as variables of type **hyperqueue**, which takes a type parameter to describe the type of the values stored in the queue. Hyperqueues may be passed to procedures provided they are cast to a type that describes the access mode of the procedure. This type can be **pushdep**, **popdep** or **pushpopdep**, to indicate that the spawned procedure may only push values on the queue, that it may only pop values from the queue, or that it may do both. A task with **push** access mode is not required to push any values, nor is a task with **pop** access mode required to pop all values from the queue. A hyperqueue may be destroyed with values still inside.

A simple 2-stage pipeline using the hyperqueue is shown in Figure 2. The procedure *pipeline* at line 20 creates a hyper-

```
void producer(pushdep<data> queue, int start, int end) {
    if( end−start <= 10 ) {
        for( int n=start; n < end; ++n ) {
            data d = f(n);
            queue.push(d);
        }
    } else {
        for( int n=start; n < end; n += 10 )
            spawn producer(queue, n, min(n+10,end));
        sync;
    }
}
```

Figure 3: A parallel producer program with improved locality.

queue object where elements of the queue are of type *struct data*. It then spawns a procedure *producer* with **pushdep** access mode which will produce data in the queue using the *push* method. The procedure *consumer* is spawned with **popdep** access mode and will consume the data. It may utilize the method *empty* to check whether any data on the queue is pending and the method *pop* to remove data from the head of the queue.

The *empty* method checks if more values are pending in the queue. It is designed such that it mimics the result of sequential execution: the *empty* method returns false only if it is certain that no more values will be added to the queue. If there is a possibility that values will be added that are visible to the task executing the *empty* method, then the *emtpy* call will block until a definite decision can be made.

Pop must only be called on non-empty queues, as popping elements from an empty queue is an error.

Executing the producer will grow the queue as big as it needs. When a program, or part of it, is executed sequentially, Swan's depth-first execution order will make the queue grow and store all data that is produced, before any of the data is consumed. This may have an adverse effect on memory locality. We show how to avoid unbounded queue growth in Section 5.

2.2 Parallel Execution and Memory Locality

Multiple producers may be active simultaneously on a hyperqueue, each producing a different range of values. The *producer* procedure in Figure 2, line 8, is recursively divided to produce subranges of the values, following Cilk best practice. The Swan runtime system ensures that the consumer sees all values in correct (serial) program order, i.e., $f(0)$, $f(1)$, $f(2)$, etc., no matter what.

In the case of hyperqueues, it may be more appropriate to write the producer as in Figure 3. In this case, values are produced with better locality as all active threads are concentrating on the head of the queue. However, there will be more frequent work stealing activity as the program's spawn tree is shallow.

2.3 Task Scheduling

The Swan runtime system utilizes the queue access modes **pushdep**, **popdep** and **pushpopdep** to decide when a spawned procedure may start execution. This process is similar to how **indep**, **outdep** and **inoutdep** define an execution order between tasks operating on versioned objects [6].

The task scheduler enforces the following constraints due to queue access modes (the access modes on all arguments are taken into account when scheduling tasks, including the versioned object access modes):

1. Tasks with **pushdep** access mode on the same queue may execute concurrently. The runtime system will use the concept of reduction to manage concurrent pushes and expose the pushed values in serial program order to any consumer.

2. A task with **popdep** access mode may execute concurrently with the preceeding tasks with **pushdep** access mode on the same queue. This enables concurrent pushes and pops on the queue. The runtime system ensures that pops do not run ahead of pushes.

3. A task with **popdep** access mode may initiate execution only when all older tasks with **popdep** access mode on the same queue have completed execution. The rationale is that values are exposed in program order, so the oldest task must perform all its pops before a younger task may perform its pops.

4. A task P with **pushdep** access mode may execute concurrently with an older task C with **popdep** access mode. The rationale is that P will create a sequence of values, but this sequence of values is independent of the actual pops performed by C. Moreover, C is not allowed to see any of the values pushed by P because this would violate the serializability of the program. The runtime system will ensure that any values left in the queue when C completes execution will be merged with the values produced by P in program order.

Tasks with **pushpopdep** access mode are scheduled by taking restrictions of both **pushdep** and **popdep** modes into account. The Swan runtime system uses the same machinery to enforce the execution order of tasks with queue dependences as it does for versioned objects [6].

In recursive programs, tasks can only spawn child tasks with a subset of the privileges that they hold, i.e., tasks with **pushpopdep** access on a hyperqueue can pass both privileges on that hyperqueue, while tasks with either **pushdep** or **popdep** access mode can pass only the named privilege on the corresponding hyperqueue. This restriction makes it safe to apply the above rules for task scheduling separately to each procedure instance [10].

Consider the following program to illustrate these rules:

```
hyperqueue<T> queue;
spawn A( (pushdep<T>)queue );
spawn B( (pushdep<T>)queue );
spawn C( (popdep<T>)queue );
spawn D( (pushpopdep<T>)queue );
spawn E( (pushdep<T>)queue );
spawn F( (popdep<T>)queue );
sync;
```

Procedure A is the oldest procedure and is immediately ready to execute. B may execute concurrently with A due to case 1. C may execute concurrently with A and B due to case 2. D must wait until C completes due to case 3. E may execute concurrently with A, B and C and prior to D following case 4. Finally, procedure F must wait until D completes due to case 3. F will never start execution prior to E due to the work-first principle (spawned tasks are executed immediately by the spawning thread).

3. INTERNAL DATA STRUCTURES

3.1 Requirements

Let us first consider push operations. The push operation is, like list concatenation, a reduction operation. Reduction operations are essentially *associative* operations, meaning that the operations may be reassociated to their operands provided that the operands remain in order. For instance, pushing the elements a, b and c on a queue Q may be obtained by pushing the elements one by one, which can be written as $((Q + a) + b) + c$. Associativity implies that the operations may be reordered, for instance as $((Q + a) + (\varepsilon + b)) + c$ where ε is an empty queue. Consequently, pushing a and b may occur concurrently on distinct queues, which are subsequently merged.

Reductions are implemented in Cilk++ reducer hyperobjects [9], which form the basis of our hyperqueue implementation. Reducers are variables that provide a local view on the variable for each task that accesses it. In a way, this view bears similarity to thread-local storage. Contrary to thread-local storage, reducers' views are task-local and views are reduced as tasks complete, or they are handed over unmodified from one task to the next. Moreover, reducers retain the relative ordering of tasks, implying that only associativity of the reduction operation is required and not commutativity. The latter is essential to build list reducers.

From a bird's eye view, hyperqueues operate as follows: (i) parallel producers operate on distinct queues, (ii) queues are merged as tasks complete, (iii) consumers operate on the head of the hyperqueue and (iv) consumers can observe only values that were pushed by tasks that precede them in serial program order. Hyperqueues, however, require a significant functional extension to Cilk++'s list reducer as pop operations may occur concurrently with pushes, popping values from the queue before it has been fully constructed. In hyperqueues, views are not strictly private, but they can be shared by at most one producer and at most one consumer.

In the remainder of this section, we will discuss the design of the underlying data structure in detail, as well as the algorithms for sharing head and tail pointers with the appropriate tasks, and how we succeed in giving a correct view on the queue for all tasks involved.

3.2 Queue Segments

We select an internal data structure for the hyperqueue that consists of a singly-linked list of queue segments. Each queue segment is a fixed-size linearly stored list (array). A queue segment may also act as a queue in its own right and is utilized as a circular buffer under those circumstances.

The queue segment is a fixed-size single-producer, single-consumer queue. As such it has a data buffer to store values, the buffer's allocated size, a head and tail index, a pointer to the next queue segment and a producing flag.

Each queue segment has a producing flag that indicates whether additional values may be pushed onto it. The producing flag is used by the *empty()* call to check whether a queue is permanently empty, or just temporarily empty.

The producing flag is initially set to *true* when a segment is created. It is set to *false* when a task with push privileges terminates and (i) there are no younger tasks outstanding on the hyperqueue (meaning that all data has been produced), or (ii) when the next younger task has pop privileges (meaning that no more data may be produced that is visible for the next task). The producing flag is turned on again when spawning a task with push privileges for the tail of the segment that is currently visible. Moreover, the producing flag is ignored when the pointer to the next queue segment is non-null, as the subsequent data is trivially accessible.

The hyperqueue uses a mixed design of single-producer single-consumer queues based on arrays [11] and on dynamically linked lists [12]. The head of the queue is reachable only by the single consumer task that may pop values from the queue. From the head of the queue stretches a linked list of segments that hold produced data. Additionally, the hyperqueue may hold linked lists of segments that are not (yet) accessible from the head. These lists are simultaneously under production by parallel tasks. It is only when tasks complete that these lists can be merged and possibly be linked to the head of the queue in order to guarantee determinism.

The internal hyperqueue data structure was selected for its performance in common circumstances. A buffer-based implementation amortizes the overhead of memory allocation per buffer, while a linked-list implementation allows concatenation of lists (reduction operation) in $O(1)$ steps. Moreover, a concurrent producer and consumer may continuously reuse a queue segment, realizing a queue implementation with zero allocation cost in steady state.

Race-free queue implementations require hardware synchronization operations, which have varying performance cost on different architectures [13]. Several papers discuss how to build correct and high-performant single-producer single-consumer queues using arrays [11, 14, 15] and linked lists [12, 16, 17].

The hyperqueue, however, is a simplified case as the queue holds at least one segment and the head and tail pointers in the linked list representation are each accessed by a single task. Moreover, there can be at most one consumer active on the hyperqueue and this consumer operates on the head segment of the hyperqueue. Thus, all but one queue segment may be viewed as write-only buffer during the production of data, or for a part of that. Making this optimization race-free requires careful design to detect when a segment switches to concurrent usage. We have not pursued this optimization.

3.3 Views

A local view of the queue, created and owned by a single task, is represented by a linked list of queue segments. As such, two pointers to queue segments are used, namely to the head and tail of the linked list. Thus, in the view (h, t), h points to the head of a linked list of queue segments, and t points to the last segment in the list. We say that h and t are local pointers when they point to a queue segment.

Shared views give a task access to queue segments that may be operated on by distinct tasks, in particular a concurrent producer and consumer. The producer holds a tail-only view (p_{NL}, t), consisting of a non-local head pointer p_{NL} and a tail pointer t, and pushes values on the segment t. The consumer holds a head-only view (h, p_{NL}), consisting of a head pointer h and a non-local tail pointer p_{NL}, and pops values from the segment h. Non-local pointers indicate that the queue segment is shared with another view and should not be (and cannot be) accessed from the view. Non-local pointers always occur in pairs and must match between successive views in program order. (In practice, all non-local pointers are represented by a null pointer.)

Two operations are defined on views: *split* makes a view shared by splitting it in two views. *Reduce* takes two views and returns two views that define the new values for both arguments. The split operation is defined as:

$$\text{split}((s, s)) = ((s, p_{NL}), (p_{NL}, s))$$

where (s, s) is the local view on the queue segment s, and p_{NL} is a unique "non-local" pointer.

The split operation is unique to hyperqueues and does not appear with Cilk++ hyperobjects. The split operation is required to make the head of a section of the queue accessible to the consumer task by attempting to attach it to the immediate and logically preceeding view. In particular, if all earlier tasks have completed, then the head view on the new queue segment will be accessible by the consumer.

When tasks complete, views are reduced by the reduction operation, defined as:

$$\text{reduce}((h_1, t_1), (h_2, t_2)) = ((h_1, t_2), \epsilon)$$

where ϵ is the empty view. There are two cases to consider:

1. The pointers t_1 and h_2 are local pointers. t_1 is a local tail pointer to a queue segment s_1 and h_2 is a local head pointer to a distinct queue segment s_2. Reduce also concatenates the segments s_1 and s_2 by setting the next pointer in s_1 to point to s_2.
2. The pointers t_1 and h_2 are non-local pointers. These non-local pointers must match: $t_1 = h_2 = p_{NL}$, a condition that is guaranteed true in our system. This case is the inverse of *split.* As such, the queue segments pointed to by the views are already linked and further concatenation is not required.

The cases above express constraints on the tail pointer in the left-hand view and on the head pointer in the right-hand view. The remaining pointers may be either local or non-local. E.g., if h_1 equals q_{NL}, a non-local pointer distinct from p_{NL}, then the view (h_1, t_2) becomes (q_{NL}, t_2), again a shared view. If t is also a non-local pointer, say r_{NL}, then the result is the view (q_{NL}, r_{NL}), again holding non-local pointers. Note that such a shared view is distinct from the empty view.

The reduction is defined also if any of the arguments is the empty view:

$$\begin{aligned}\text{reduce}((h, t), \epsilon) &= ((h, t), \epsilon)\\ \text{reduce}(\epsilon, (h, t)) &= ((h, t), \epsilon)\\ \text{reduce}(\epsilon, \epsilon) &= (\epsilon, \epsilon)\end{aligned}$$

Other cases for the reduction operator cannot occur during execution due to the properties of the system.

4. HYPERQUEUE MANAGEMENT

The runtime system maintains the logical ordering of the partial lists of pushed values using up to 4 views on the queue per task. Each view is a shared queue segment as discussed above and may contain a head and tail pointer. Every task has the views *user* and *right.* Tasks with push privileges also have the view *children*, while tasks with pop privileges have the view *queue.* The top-level task always has both push and pop privileges and thus maintains 4 views.

4.1 Updating Views with New Segments

Push operations work on the *user* view, which represents the slice of the queue viewable to the currently executing task. If that task has spawned other tasks, then those tasks' pushed values will be collected in the *children* view. The *right* view represents the values pushed by the tasks' right siblings (tasks later in program order).

The *queue* view gives access to the end of the queue where values can currently be popped from. Initially, an empty queue segment is generated when the hyperqueue is created. The head pointer of the *queue* view and the tail pointer of the *user* view are set to point to the segment s_{new}:

$$(queue, user) \leftarrow \text{split}((s_{new}, s_{new}))$$

The *push* operation appends a value to the queue segment identified by the tail pointer of the *user* view. If the queue segment is full, a new segment is created and appended to the *user* view. This updates only the tail of the user view.

During parallel execution, the *push* operation may also find an empty *user* view. In this case, a new segment is created and linked to the logically preceeding segment. Once the segment is linked in place, the consumer task is able to reach it as it holds a pointer to the first segment in the hyperqueue in its *queue* view, and it can follow the linked list of segments to reach all segments linked to it.

However, care must be taken to respect the program order wherein values are created. Depending on how the parallel program is executed, it may be premature to link a new segment to the segments accessible by the consumer. Indeed, tasks earlier in program order may not have completed yet. This problem is solved by the *children*, *user* and *right* views in the task and by linking a queue segment only to the immediate logically preceeding task.

Formally, a new view is created pointing to the new segment s_{new} and is split into a temporary view:

$$(tmp, user) \leftarrow \text{split}((s_{new}, s_{new}))$$

The temporary view is then merged with a view in the immediate logically preceeding task.

If the task performing the *push* has a left sibling in the spawn tree, then the temporary view is reduced with the left sibling's *right* view:

$$(left.right, tmp) \leftarrow \text{reduce}(left.right, tmp)$$

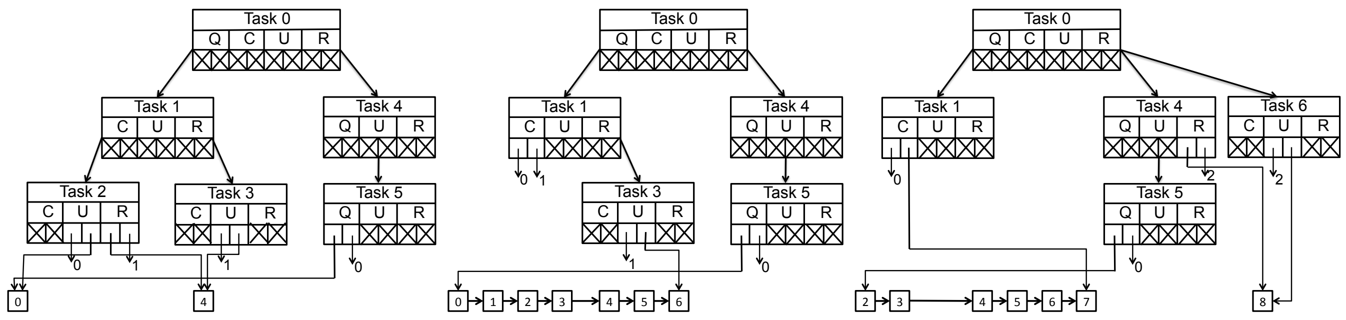
If the creating task does not have a left sibling, then the head pointer is reduced with the parent task's *children* view. If this view is empty before the reduction, then the algorithm to share the queue head is executed recursively from the parent, until the top-level task is encountered, where it is merged with the *children* view.

4.2 Updating Views at Scheduling Points

We have discussed above how views are updated with new segments. This section describes how views are updated at spawn and sync statements, and when tasks complete.

Spawn with push privileges. The *user* view, if any, is passed from the parent frame to the child frame. The parent's *user* view is cleared. This behavior is the common path when executing code sequentially.

Return from spawn with push privileges. Let us assume that a child frame C has finished execution, and that C was originally spawned by its parent frame P. As the child frame C has finished execution, its *user* view can no longer grow. The *right* view of C is reduced with its *user* view, linking it to the data produced by C's right sibling $(C.user, C.right) \leftarrow \text{reduce}(C.user, C.right)$.



(a) Tasks 2 and 3 concurrently push elements. (b) Task 2 has completed. Task 5 can pop from queue head. (c) Tasks 2 and 3 have completed. Task 5 has popped values from the queue. Task 6 pushes more values that are invisble to tasks 4 and 5.

Figure 4: Illustration of the concurrent execution of producing and consuming tasks and their effect on the construction and destruction of the hyperqueue. Legend: Q is queue view, C is children view, U is user view and R is right view. Arrows with a numeric idea represent non-local pointers, which occur in pairs.

If C has a left sibling L, then C's values are merged with L: $(L.right, C.user) \leftarrow \text{reduce}(L.right, C.user)$. If C has no left sibling, then it must be the oldest child of P. Thus, we perform: $(P.children, C.user) \leftarrow \text{reduce}(P.children, C.user)$.

Call and return from call with push privileges. For reasons of simplicity, we treat calls in the same way as spawns for the purpose of hyperqueues. We do not anticipate that call statements would be a common idiom on hyperqueues because calls forego concurrency with consumers.

Spawn with pop privileges. When a parent frame P spawns a child frame C with pop privileges, then P's *user* view is copied to C's *user* view and P's *user* view is cleared. The *user* view is passed to C to hide it from subsequent tasks with push privileges. The *user* view will be merged back in in correct program order when the current task completes. Similarly, P's *queue* view is passed over to C.

Return from spawn with pop privileges. When returning from a frame that was spawned with pop privileges, it is not necessarily the case that all elements have been consumed from the queue. The remaining view is passed back to the parent procedure.

First, the same actions are taken as in the case of "return from spawn with push privileges". Second, C's *queue* view, which is a head-only view, is returned to its parent P.

Sync. A frame P that executes a sync statement waits until all children have completed execution. As such, all spawned children have completed and they have reduced their local views with P's *children* view. P's *user* view is updated with the reduction of P's *children* and *user* views.

4.3 Example

Figure 4 presents an example of view creation and reduction. The top-level task (Task 0) spawns Task 1 with push privileges, followed by Task 4 with pop privileges, followed by Task 6 with push privileges. Determinism requires that the effects of the tasks must be observed in this order. Task 1 in turn spawns Tasks 2 and 3 with push privileges. Task 2 pushes values 0–3 on the queue, while Task 3 pushes 4–7. Task 4 spawns Task 5 which pops values from the queue. Finally, Task 6 pushes the value 8 on the queue, which may not be observed by Tasks 4 and 5 in light of determinism.

Tasks 2 and 3 are spawned first and generate a partial list of values (Figure 4 (a)). Task 2 inherits access to the initial queue segment through its *user* view and pushes values on that segment. Task 3 creates a new queue segment which it splits and then merges the head with Task 2's *right* view. The split creates a new non-local pointer with unique ID (1). As discussed above, it is too early to link this segment to the segment operated on by Task 2 as Task 2 may perform more pushes and may require additional segments.

When Task 2 completes, its *user* and *right* views are reduced, together with Task 1's *children* view. The *user* view is a tail-only view (due to the push), while the *right* view is a head-only view (due to the *split* and propagation of the head performed by Task 3's push). These views merge, leaving Task 1's *children* view with non-local pointers. This case shows the utility of splitting the view on new segments and reducing the head-only view ahead of the reduction of the tail. Even though Task 3 is still executing, the consumer is able to pop values produced by Task 3.

Tasks 4 and 5 are created concurrently while Tasks 1 and 2 execute. Task 5 inherits the *queue* view through Task 4 and pops the values 0 and 1 from the queue (Figure 4 (b)).

Finally, Task 6 is created and pushes values onto a new queue segment (Figure 4 (c)). Similar to Task 3, Task 6 shares the head of this queue segment with its left sibling (Task 4). By consequence, this segment is not linked with its predecessor and remains inaccessible to Tasks 4 and 5. This is, again, a requirement for deterministic execution.

4.4 Hyperqueue Invariants

At any moment, hyperqueues respect the following invariants which we state without proof:

1. Every hyperqueue holds **at least one segment**. An initial segment is created when the hyperqueue is constructed. The last segment is not deleted when it is empty.

2. At any one time, for a given hyperqueue, there is **exactly one *queue* view with a local head pointer**. This view is accessible by the single task with **pop** privileges that is allowed to consume data.

3. The tail pointer in the *queue* view and the head pointer in the *user* view are **always** non-local unless if these views are empty. Space may be saved by not storing these pointers.

4. Every segment in a hyperqueue is pointed to by either **one** *next-segment* pointer, or by **one** view's *head* pointer.

5. Every segment stored in a hyperqueue is pointed to by **at most one** view's *tail* pointer. Every segment stored in a hyperqueue is pointed to by **exactly one** view's *tail* pointer if and only if the segment's *next-segment* pointer is null.

6. A consequence of invariants 4 and 5 is that **any segment may be shared by at most two tasks**, of which one is a consumer and one is a producer, as a consumer requires access through the head pointer and a producer requires access through the tail pointer.

Assume a total order $<$ of views that reflects the program order (following serial elision) in which the data stored in those views has been produced. We say that for views v_1 and v_2, $v_1 < v_2$ when the following holds:

(i) For a task T, $T.queue < T.children < T.user < T.right$. If a task does not have a particular view, the relation for that view is irrelevant.

(ii) For sibling tasks T_1 and T_2 where T_2 is later in program order, all views of T_1 are ordered before T_2's views.

(iii) For tasks P and C where P is the parent of C, and for any view v of C, $P.children < C.v < P.user$.

7. If a linked list of segments is pointed to by the *head* pointer of view $T_1.v_1$ and by the *tail* pointer of view $T_2.v_2$, then $T_1.v_1 < T_2.v_2$ provided that v_1 is not a queue view. An interpretation of this invariant is that values are stored in an order that corresponds to program order.

8. For views $T_1.v_1$ and $T_2.v_2$ as in invariant 7, it holds that $T_2.v_2 < T_1.v_1$ provided that v_1 is a queue view and T_1 does not have both push and pop privileges. This invariant shows that a consumer task can only observe values that have been pushed by tasks preceeding it in program order.

9. For views $T_1.v_1$ and $T_2.v_2$ as in invariant 7, if v_1 is not a queue view, then for any non-queue view v held by any task T, if $T_1.v_1 < T.v < T_2.v_2$, then v is a non-local view or ϵ.

4.5 Discussion

Double reduction. The hyperqueue assumes two reduction steps: first, when a new segment is created in an empty *user* view, the head is reduced with the immediate logically preceeding view. Second, when a task has completed, its views are reduced as in the case of hyperobjects. The early head reduction is required to make partial queue segments discoverable as soon as possible. This reduction has limited overhead as (i) it occurs only on empty *user* views and (ii) it terminates after 1 step (in case a left sibling task exists) or in at most d steps for a d-deep spawn tree.

Hyperqueues are free of deadlock. To demonstrate this, we need to show that there cannot be dependence cycles between tasks [18]. To be more precise, we will demonstrate that there cannot be dependence cycles between *strands*, where a strand is a maximum sequence of instructions without spawn and sync statements [9]. The Cilk programming model defines dependences between strands such that the parallelism defined by *spawn* statements is exposed, and the serialization of *sync* statements is enforced. The dependences between strands are a partial ordering of strands that respects the total order of strands defined by sequential program order, i.e., the serial elision of the program.

On top of these dependences, we introduce additional producer-consumer dependences for the hyperqueues. These producer-consumer dependences also respect program order: only strands containing an *empty()* or *pop()* call depend on other strands and they can only depend on strands earlier in program order. As such, neither the Cilk-defined dependences nor the hyperqueue dependences introduce a dependence between strands that does not exist in the serial elision, which is a total order. It follows that the total set of dependences cannot contain cycles. As such, there always exists at least one strand that the scheduler can execute. This guarantees forward progress.

Scalability. Consuming tasks (with **popdep** arguments) may block on the *empty()* call. There are two design choices to deal with blocking: (i) the executing task and worker may block until the blocking condition is resolved, or (ii) the executing task may be suspended and the worker may continue operating on a distinct task. Either approach is possible and can be implemented without changes to the programming interface. We have opted to block the worker in this work for pragmatic reasons. The blocking delays are short in practice in our benchmarks, so the overhead of de-scheduling the task would not be justifiable. In other circumstances, suspending the task may be a better choice. A possible extension to the programming interface is to allow the programmer to express whether the task should block or be suspended.

Blocking tasks has the potential downside that in the worst case all but one of the worker threads may be blocked. Defensive programming is possible whereby a good mix of consumers and producers is simultaneously active. However, even without this, blocking occurs rarely in our benchmarks.

Special Optimization. Cilk++ hyperobjects are efficient even on deep spawn trees. The reason is that hypermaps are created and reduced only when tasks are stolen, but not every time a task is created. This is referred to as the *special* optimization [9]. The special optimization is also applicable to hyperqueues. The *children*, *user* and *right* hypermaps may be handled as specified by the special optimization (our discussion in Section 4.2 only discusses the cases of stolen tasks). The *queue* hypermap is distinct, as only one task has a non-empty *queue* view for a given hyperqueue. As such, it is preferrable to store the *queue* view in the hyperqueue variable and attach an ownership label to it (e.g., the stack frame pointer) such that access to it can be arbitrated.

Our experimental evaluation does not include the special optimization. We expect that it would have a negligible impact on our benchmarks as they have shallow spawn trees.

5. PROGRAMMING IDIOMS

5.1 Queue Segment Length Tuning

The programmer often knows the best queue segment size for a program. E.g., a program performing producing or consuming data in parallel may generate the same number of values in each leaf task. It is beneficial to set the queue segment length equal to this number. Alternatively, the programmer may know that the total queue size is often around a particular size, or that the consumer and producer require a particular queue buffer length to remain in balanced execution without blocking. The queue segment length may be set at queue initialization time as a parameter to the constructor of the hyperqueue class.

5.2 Queue Slices

Queue slices provide direct access to a queue segment, which is as fast as an array access. Instead of performing push, empty and pop operations on the queue, the programmer first requests a slice and then performs the operations on the slice. It is guaranteed that the storage space for a slice is available and that all data is ready.

```
bool producer( pushdep<int> queue, int block ) {
    for( int i=0; i < block; ++i )
        queue.push( ... );
    return more_work_to_do() ? true : false;
}
void consumer( popdep<int> queue ) {
    while( !queue.empty() )
        ... = queue.pop();
}
void pipeline () {
    hyperqueue<int> queue;
    while( producer( (pushdep<int>)queue, 10 ) ) {
        spawn consumer( (popdep<int>)queue );
    }
    sync;
}
```

Figure 5: Taking the main queue iteration loop outside the tasks.

Read slices can be requested from tasks with pop privileges. The system returns the slice starting at the current head of the queue up to the requested length under the constraints that (i) the data must have been pushed and (ii) the slice must fit inside a single segment. If not, a shorter slice will be returned.

Write slices can be requested from tasks with push privileges. A new queue segment may be created to accommodate the requested slice length.

5.3 Selectively Enabling Pipelining

When executed sequentially, the hyperqueue will grow as large as necessary to accommodate all data sent through the queue. It is possible to avoid this behavior by providing both sequential and pipeline-parallel implementations of the code. Then, a runtime check can be made to check if the code is executing in parallel or not and an appropriate version of the code can be selected.

There are several ways to detect whether a Cilk program is executing sequentially or in parallel. Cilk provides a direct way by checking the variable SYNCHED [19]. Alternatively, Cilk++ hyperobjects can also be designed in order to give away this information [20]. These features must be used with care because they can violate determinism.

5.4 Queue Loop Split and Interchange

Another potential protection against unbounded queue growth is to split each stage's main loop over queue values and bring the outer loop outwards of the queue. This technique is illustrated in Figure 5. Instead of calling the *producer* function once, it is now called once for every 10 elements. The total degree of parallelism is equal to that of a solution with a single call to *producer* and *consumer*, except that memory usage is limited to grow by a factor 10 when the program is executed serially.

5.5 Selective Sync

The procedure in Figure 6 spawns a consumer task and performs *empty()* and *pop()* calls itself. The procedure will block on *empty()* until completion of the consumer as it has an empty *queue* view while the consumer executes. It is, however, preferrable to suspend the task, freeing the worker to execute other tasks. The following syntax suspends a task until all children with a particular access mode on a particular object have completed: **sync (popdep<int>)queue;** suspends the procedure until all child tasks with **popdep** access mode on *queue* have completed. Adding this statement before *empty()* changes blocking to suspension. This is an extension of the syntax "**sync queue;**" supported by Swan to suspend a task until all children operating on the *queue* object have completed.

```
hyperqueue<int> queue;
spawn producer( (pushdep<int>)queue );
spawn consumer( (popdep<int>)queue );
spawn producer( (pushdep<int>)queue );
if( !queue.empty() ) // block until consumer() done
    queue.pop();
```

Figure 6: A case for selective sync.

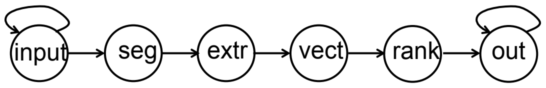


Figure 7: Schematic of ferret's pipeline.

6. EVALUATION

We evaluate the performance of pipeline parallel benchmarks implemented with POSIX threads, Intel's Threading Building Blocks and Swan, a task dataflow system [6]. Moreover, the hyperqueues are also implemented in Swan in order to leverage the dataflow ordering functionality required to sequence tasks with pop privileges. Our implementation is published at `http://github.com/hvdieren/swan`.

The experimental system is a multi-core node with 2 AMD Opteron 6272 (Bulldozer) processors. On this processor, pairs of cores share a floating-point unit (FPU). The processors have 6144 KB L3 cache shared per 8 cores. Main memory is distributed over 4 NUMA nodes. The system runs the Ubuntu OS version 12.04.1 LTS and gcc version 4.6.3. We use Intel Threading Building Blocks (TBB) version 4.1 20130314oss.

We evaluate the hyperqueue on 3 pipeline parallel benchmarks: ferret and dedup taken from the PARSEC suite [21], and the bzip2 compression utility. Our codes are available from `http://github.com/hvdieren/parsec-swan`.

6.1 Ferret

Ferret performs content-based similarity search, determining for a set of images which images contain the same kind of object. The required computation is spread over a 6-stage pipeline (Figure 7) consisting of, respectively, input (loading images from disk), segmentation, feature extraction, vectorizing, ranking and output. The first (input) and last (output) stages are serial stages, implying that these stages must operate on all images strictly in their original order. The stages in between have no permanent state. As such, multiple instances of these stages may be executing in parallel on distinct images.

We have measured the amount of time taken by each stage when executing the serial version of the benchmark on the PARSEC 'native' input (Table 1). This table shows that the majority of execution time is taken by the *ranking* stage (75.3%), while the *vectorizing* stage also takes a sizable fraction of execution time (16.2%). The segmentation and extraction stages are less time consuming.

Serial stages can pose major limitations to scalability. In

Table 1: Characterization of ferret's pipeline.

	Iterations	Time (s)	Time (%)
Input	1	34.000	4.48
Segmentation	3500	26.800	3.57
Extraction	3500	2.773	0.35
Vectorizing	3500	133.939	16.20
Ranking	3500	603.286	75.30
Output	3500	2.000	0.10

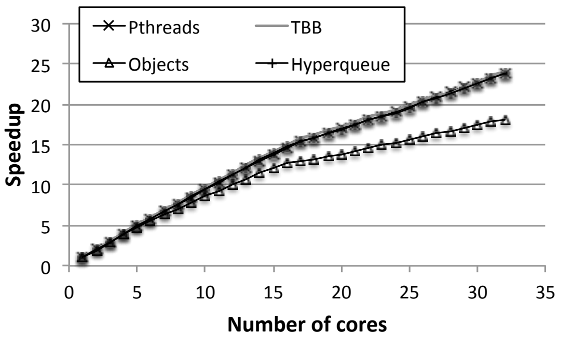


Figure 8: Ferret speedup by using various programming models.

ferret, the input stage takes about 4.5% of execution time. As such, scalability is limited to roughly 22 if we fail to overlap its execution with other work.

While the structure of the computation of ferret does not pose any problems toward parallelization (it is a common pipeline pattern), the code exposes a generic programmability issue. The input stage is a recursive directory traversal that collects image files in a directory tree. Written in Pthreads, files are pushed on a queue as they are discovered.

Turning ferret into a pipeline structure using programming models such as TBB or Swan is not impossible. However, it requires thoroughly restructuring the input stage in such a way that it can be called repeatedly to produce the next file [22]. To this end, its internal state must be made explicit (i.e., its current position in the traversal of the directory tree) and passed as an argument to the first stage. This is all but rocket science. But it is tedious and error-prone.

Hyperqueues avoid restructuring the program, thereby making it much easier to extract the latent parallelism in the program. With hyperqueues, the directory traversal pushes discovered image files on the queue, as in the pthreads version. These images can be concurrently consumed by the next pipeline stage.

We measured the performance of ferret using Pthreads, TBB and Swan. We show two versions of the code for Swan. The "objects" version uses the baseline task dataflow model. In this case, we did not implement the code restructuring of the input stage as with the TBB code in order to demonstrate the importance of overlapping the execution of the input stage with the remainder of the pipeline. The "hyperqueue" version uses a hyperqueue to communicate data between the input stage and the segmentation stage, and also to communicate between ranking and output. The latter hyperqueue was inserted because of the fine granularity of the output stage. As such, we avoid spawning many small tasks. Instead a single large task is spawned for this stage which iterates over all elements in the queue.

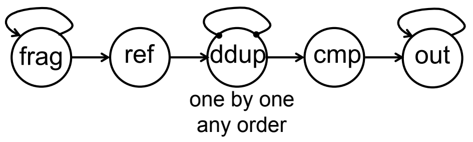


Figure 9: Schematic of the dedup pipeline.

Table 2: Characterization of the dedup pipeline.

	Iterations	Time (s)	Time (%)
Fragment	336	1.900	3.08
FragmentRefine	336	3.916	6.35
Deduplicate	369950	4.854	7.90
Compress	168364	45.881	74.48
Output	369950	5.049	8.19

Figure 8 shows the speedup of the pthreads, TBB, objects and hyperqueue implementations relative to the serial implementation. Performance of the objects version is clearly limited by not overlapping the input stage with the remainder of the pipeline. The remaining implementations show nearly the same performance.

Note a slight decrease of scalability when the number of cores exceeds 16. This is due to the sharing of FPUs between pairs of cores in the Bulldozer architecture.

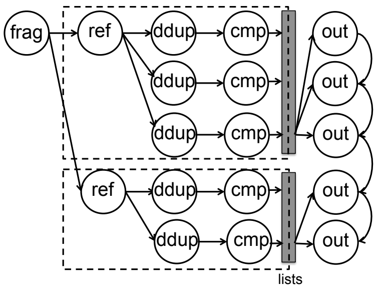
The pthreads version uses massive core oversubscription. It starts 28 threads for each of the parallel stages. Launching the same number of threads is clearly not justified by the breakdown in Table 1. For best performance, the number of threads per stage needs to be tuned individually. The number 28 was experimentally determined and is likely a result of the maximum number of cores we used (32) and the fact that one stage dominates the execution time. As such, it is important to assign many threads to this stage. The hyperqueue implementation obtains the same performance as pthreads and does not require core-count dependent tuning.

6.2 Dedup

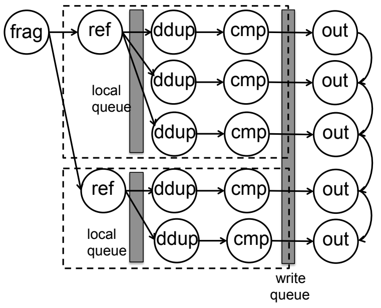
Dedup performs file compression through deduplication (eliminating duplicate data blocks) and compression. Dedup has a 5-stage pipeline that is tricky to implement efficiently using structured programming models such as TBB and Swan. The dedup pipeline stages consist of fragmentation (dividing the input file in large chunks), refining (splitting large chunks in small chunks), deduplication (finding equal chunks), compression of chunks and output (Figure 9). This pipeline poses implementation problems because of the variable number of input and output items in several stages. In particular, the fragment refining stage produces a variable number of small chunks per large chunk and the compression stage is skipped for duplicate chunks.

Table 2 shows the number of chunks processed and the time spent per pipeline stage. Execution time is biased towards *Compress*. Instances of this stage can execute in parallel so there is ample parallelism. The *Output* stage is the most limiting serial stage. Taking 8.2% of the execution time, it limits overall application speedup to 12.7.

Reed *et al* observed that dedup exhibits a nested pipeline [22]. The outer pipeline, handling large chunks, consists of three stages: *Fragment*, *InnerPipeline* and *Output*. The inner pipeline consists of *FragmentRefine*, *Deduplicate* and *Compress*. A new instance of the inner pipeline is created for every large chunk and produces a list of small chunks that makes up the corresponding large chunk.



(a) Nested pipelines



(b) Positioning of hyperqueues

```
void Fragment( pushdep<chunk_t *>write_queue ) {
    while( more coarse fragments ) {
        chunk_t * chunk = ...;
        { // Set up inner pipeline with local queue
            hyperqueue<chunk_t*> * q
                = new hyperqueue<chunk_t *>;
            spawn FragmentRefine(
                chunk, (pushdep<chunk_t *>)*q );
            spawn DeduplicateAndCompress(
                (popdep<chunk_t *>)*q,
                (pushdep<chunk_t *>)write_queue );
        }
    }
    sync;
}
int main() {
    hyperqueue<chunk_t*> write_queue;
    spawn Fragment( (pushdep<chunk_t*>)write_queue );
    spawn Output( (popdep<chunk_t*>)write_queue );
    sync;
}
```

(c) Hyperqueue implementation of dedup.

Figure 10: Alternative implementation choices for dedup. The graphics (a) and (b) show dynamic instantiations of each pipeline stage, how they are grouped and where collections of data elements are used. Dashed lines indicate instances of the inner pipeline. (c) Sketch of hyperqueue code according to (b).

Figure 10 (a) shows the dynamic instantiations of all pipeline stages. Two large chunks have been found, where the first is further split in three small chunks and the latter is split two-ways. This graphic demonstrates a shortcoming of the nested pipeline approach: all the small chunks for a large chunk must be completed and gathered on a list before the output stage can proceed. This puts an important limit to scalability, as the number of small chunks per inner pipeline is typically 500-600 and may run up to 65537, potentially resulting in long and skewed delays.

Hyperqueues allow consuming elements concurrently to pushes, removing the wait times of the output stage until large chunks have been fully processed as in the case of nested pipelines. Moreover, like Cilk++ list reducers, hyperqueues allow us to construct parts of the list concurrently and merge list segments as appropriate. This way, all nested pipelines can push elements on the same hyperqueue and the write actions become synchronized and ordered between invocations of the nested pipeline. Finally, hyperqueues can be used directly as a drop-in replacement for lists, as they support the required push and pop operations (Figure 10 (b)).

Our hyperqueue implementation inserts a local hyperqueue between the *FragmentRefine* stage and the *Deduplication* stage. Also, all instances of the *Deduplication* and *Compress* stages that correspond to the same nested pipeline (large chunk) are merged into a single sequential task. This design was chosen to coarsen the tasks and reduce dynamic scheduling overhead (which is absent in the pthreads implementation). Ample parallelism remains in the program.

Our formulation of dedup follows the original sequential algorithm, which greatly affects programmer productivity. Figure 10 (c) shows a sketch, where the *main* procedure spawns two tasks *Fragment* and *Output*. *Fragment* calls all but the output stage in a recursive manner: whenever a large chunk is constructed, a nested pipeline is created using two tasks that communicate through a local hyperqueue. Completed small chunks are produced on the *write_queue*. In contrast, the TBB version of dedup requires significant restructuring of the code in order to match the structure imposed by TBB.

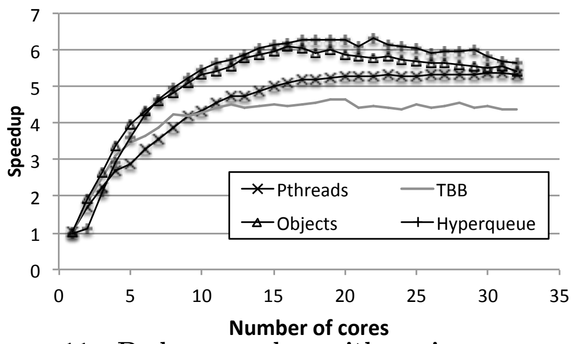


Figure 11: Dedup speedup with various programming models.

Note that the hyperqueue enforces dependences across procedure boundaries. This is an effect that is hard to achieve in Swan, where dataflow dependences can exist only within the scope of a procedure.

Figure 11 shows speedup for dedup in the pthreads, TBB and Swan programming models. While Reed *et al* demonstrated improved performance of their TBB implementation relative to the pthreads implementation in PARSEC 2.1 [22], our evaluation using PARSEC 3.0 shows that the TBB implementation is slower than the pthreads implementation. The Swan implementation with hyperqueues outperforms the pthread version by at least 12% and up to 30% in the region of 6-8 threads. The hyperqueue implementation looses some of its advantage for 22 threads and higher due to task granularity and locality issues.

6.3 Bzip2

We only report the main results on bzip2 in the interest of brevity. Prior work shows that the baseline task dataflow model is well-suited to execute bzip2's pipeline in parallel [7]. We compared a hyperqueue implementation against the task dataflow implementation to verify the performance of the hyperqueue. bzip2 has a 3-stage pipeline where the first and last pipeline stages must execute serially.

Our first implementation assigns one task to each pipeline stage, connected through two hyperqueues. The second stage's task performs a spawn for every element popped from the input queue to exploit parallelism in the second stage. Passing the output hyperqueue of stage2 to each of these spawned functions allows them to execute in parallel while retaining the order of the elements through the reduction properties. This implementation scales well, however, it suffers from bad memory locality when executed serially. Thus, we applied the technique of Section 5.4 to improve memory locality and obtained performance equivalent to that of the baseline task dataflow implementation.

7. RELATED WORK

We describe related work concerning the properties of the programming model and also the runtime scheduler.

7.1 Programming Model

The Threading Building Blocks (TBB) [23] provide parallel skeletons that allow programmers to express parallel code structures in a generic way. TBB, however, does not define a serialization of the program and does not guarantee determinism, even in the case of specially crafted functionality [24]. TBB programs tend to be free of thread-count dependent parameters.

StreamIt [25] defines a language and compiler for streaming programs, which are closely related to pipelined programs. StreamIt programs are scale-free. However, the StreamIt compiler statically schedules the computations to cores, at which point this property is lost. StreamIt programs may be non-deterministic in which case there exists no unique serialization.

A fine-grain scheduler for GRAMPS graphics pipelines is described in [26]. The paper does not discuss aspects of determinism nor the existence of a serialization of GRAMPS programs. It does not provide examples to demonstrate that the system encourages scale-free programs.

Phasers are a multi-purpose synchronization construct applicable also to pipelines [27]. Programs constructed with phasers are not serializable and are not scale-free, although they are deterministic [4].

OpenSTREAM is a system for stream- and task-based programming [28]. OpenSTREAM programs are deterministic provided that producing and consuming tasks are created in a fixed order. Removing parallel constructs from OpenSTREAM programs does not deliver a workable serialization. OpenSTREAM does, however, provide compiler support to optimize the execution of stream-based programs.

Concurrent data structures [29, 30] can be used in conjunction with thread-oriented parallel programming abstractions such as POSIX threads and Java threads. Concurrent data structures allow multiple threads to access the data structure concurrently with a guarantee that each thread's effects occur in some perceived order, as in the case of the linearizability condition [31]. Concurrent data structures are not deterministic (in the sense used in this paper) and they do not provide a serialization of the program.

7.2 Scheduling

It has been shown that pipeline parallelism is best scheduled dynamically in order to cope with imbalanced pipeline stages [32]. The baseline Swan runtime system performs such dynamic load balancing very effectively, also for pipeline parallel programs [6].

Pipeline stages may be seen as transformations on work items [33]. Threads pick work items from queues holding work items from various stages in the pipeline. Threads advance the work items to the next stage and return them to the queues until processing is completed. This model is scalable as more threads are easily added to execute the pipeline. It also closely corresponds to the way the baseline Swan system executes pipelines, except that Swan retains program order and gives preferences to complete older work items before generating new ones.

DoPE [34] adapts the degree of parallelism in statically scheduled programs by switching dynamically between static schedules. DoPE introduces some opportunity to change the scale, but switching between versions is costly as it requires to drain the pipeline.

Others have devised specific strategies to identify performance limiting stages [35]. Additional threads are assigned to the limiting stages and taken away from the others. Swan achieves this effect automatically, without analyzing per-thread performance.

GRAMPS [26] employs optimization to bias the scheduler towards limiting memory footprint, to optimize the usage of intermediate buffer space and to recycle thread state for serial pipeline stages. Overall, the Swan scheduler executes a comparable schedule, but its genericity foregoes optimization specific to pipeline parallelism.

7.3 Hyperqueue Implementation in Cilk

Our implementation of hyperqueues builds on the task dataflow runtime of Swan. With a few modifications, however, they may also be implemented in Cilk. Hyperqueues use two features that are not available in Cilk: (i) The possibility to postpone tasks with **popdep** arguments in case an older task with a **popdep** argument is executing. (ii) Differentiating the actions on a hyperqueue depending on the branch in the spawn tree. Hyperobjects behave the same throughout a Cilk program, i.e., a reducer is always a reducer and a holder is always a holder [9]. Hyperqueues, however, show a produce interface with the push method on some branches of the spawn tree. They show a consume interface with the pop and empty methods on other branches.

It is possible to overcome these limitations. For instance, one may require that only a single consuming task may be spawned per parallel region, i.e., between any two sync statements. To resolve the multi-faceted aspects of hyperqueues, statements may be added just before spawn statements that declare that the hyperqueue will specialize to a produce (pushdep) or consume (popdep) interface. This information is then visible to the subsequently spawned tasks.

Finally, it is necessary to construct a coordinated view between all tasks operating on a hyperqueue. We believe that this can be achieved with appropriate definition of the hyperobject and with a few modifications to the generic reducer mechanism in Cilk++.

8. CONCLUSIONS

Determinism and scale-free parallelism are key characteristics of ubiquitous parallel programming models that improve programmer productivity and code quality. This paper presents hyperqueues, a programming abstraction of queues that allows to specify deterministic and scale-free programs with pipeline parallelism.

We explain the semantics and an implementation of hyperqueues in the context of a task dataflow programming language and runtime system. Application to several irregular pipeline parallel programs shows that the same performance (for ferret and bzip2) or up to 30% better performance can be obtained (for dedup) on a 32-core shared memory node.

Most importantly, hyperqueues simplify the construction of highly parallel pipelined programs. Programs written in our task dataflow language extended with hyperqueues are serializable, deterministic and scale-free. This improves programmer productivity and aids performance portability.

In future work we aim to extend the semantics of the hyperqueue to allow concurrent pop operations, while retaining the programming productivity properties of the hyperqueue.

9. ACKNOWLEDGMENT

We are grateful to the reviewers and the shepherd for their constructive comments on this paper. This work is partly supported by the European Community's Seventh Framework Programme (FP7/2007-2013) under the NovoSoft project (Marie Curie Actions, grant agreement 327744), under the TEXT project (grant agreement 261580) and by the United Kingdom EPSRC GEMSCLAIM project (grant agreement EP/K017594/1).

10. REFERENCES

[1] R. Bocchino, V. Adve, S. Adve, and M. Snir, "Parallel programming must be deterministic by default," in *HotPar*, 2009.

[2] R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman, "Safe nondeterminism in a deterministic-by-default parallel language," in *POPL*, 2011.

[3] J. C. Jenista, Y. h. Eom, and B. C. Demsky, "OoOJava: software out-of-order execution," in *PPoPP*, 2011.

[4] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-java: The new adventures of old X11," in *Principles and Practice of Programming in Java*, 2011.

[5] M. Bauer, S. Treichler, E. Slaughter, and A. Aitken, "Legion: Expressing locality and independence with logical regions," in *SC*, 2012.

[6] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos, "A unified scheduler for recursive and task dataflow parallelism," in *PACT*, 2011.

[7] H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos, "Parallel programming of general-purpose programs using task-based programming models," in *HotPar*, 2011.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multi-threaded language," in *PLDI*, 1998.

[9] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin, "Reducers and other Cilk++ hyperobjects," in *SPAA*, 2009.

[10] P. Pratikakis, H. Vandierendonck, S. Lyberis, and D. S. Nikolopoulos, "A programming model for deterministic task parallelism," in *Workshop on Memory Systems Performance and Correctness*, 2011.

[11] L. Lamport, "Specifying concurrent program modules," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 2, pp. 190–222, Apr. 1983.

[12] J. Valois, "Implementing lock-free queues," in *Proc. of the 7th Intl. Conf. on Parallel and Distributed Computing Systems*, 1994.

[13] D. Lea, "The JSR-133 cookbook for compiler writers," 2011.

[14] J. Giacomoni, T. Moseley, and M. Vachharajani, "Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue," in *PPoPP*, 2008.

[15] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev, "Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated," in *POPL*, 2011.

[16] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *PODC*, 1996.

[17] P. Fatourou and N. D. Kallimanis, "A highly-efficient wait-free universal construction," in *SPAA*, 2011.

[18] E. G. Coffman, M. Elphick, and A. Shoshani, "System deadlocks," *ACM Comput. Surv.*, vol. 3, no. 2, pp. 67–78, 1971.

[19] "Cilk 5.4.6 reference manual," http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf, 1998.

[20] A. Robinson, "Detecting theft by hyperobject abuse," http://software.intel.com/en-us/blogs/2010/11/22/detecting-theft-by-hyperobject-abuse/, 2010.

[21] C. Biena, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, Jan. 2011.

[22] E. C. Reed, N. Chen, and R. E. Johnson, "Expressing pipeline parallelism using TBB constructs," in *Workshop on Transitioning to Multicore*, 2011.

[23] *Intel Threading Building Blocks*, Intel, Sep. 2010, document Number 319872-006US.

[24] A. Katranov, "Deterministic reduction: a new community preview feature in Intel threading building blocks," http://software.intel.com/en-us/blogs/2012/05/11/deterministic-reduction-a-new-community-preview-feature-in-intel-threading-building-blocks, 2012.

[25] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *CC*, 2002.

[26] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis, "Dynamic fine-grain scheduling of pipeline parallelism," in *PACT*, 2011.

[27] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, "Phasers: a unified deadlock-free construct for collective and point-to-point synchronization," in *ICS*, 2008.

[28] A. Pop and A. Cohen, "Openstream: Expressiveness and data-flow compilation of openmp streaming programs," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 53:1–53:25, 2013.

[29] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger, "STAPL: an adaptive, generic parallel C++ library," in *LCPC*, 2003.

[30] D. Lea, "Concurrency JSR-166 interest site," http://gee.cs.oswego.edu/dl/concurrency-interest/.

[31] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[32] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval, "Analytical modeling of pipeline parallelism," in *PACT*, 2009.

[33] S. Macdonald, D. Szafron, and J. Schaeffer, "Rethinking the pipeline as object-oriented states with transformations," in *Intl. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS) at IPDPS*, 2004.

[34] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August, "Parallelism orchestration using DoPE: the degree of parallelism executive," in *PLDI*, 2011.

[35] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt, "Feedback-directed pipeline parallelism," in *PACT*, 2010.