Computer Science Department
University of Crete

Exploiting Pipelined Parallelism with Task Dataflow Programming Models

Master's Thesis

Kallia Chronaki

June 2013

Heraklion, Greece

University of Crete

Computer Science Department

Exploiting Pipelined Parallelism with

Task Dataflow Programming Models

Thesis submitted by

Kallia Chronaki

in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

Author: _____

Kallia Chronaki

THESIS APPROVAL

Committee approvals: _____

Angelos Bilas
Professor, Thesis Supervisor

_____

Dimitrios S. Nikolopoulos
Professor, Queen's University of Belfast

_____

Manolis G.H. Katevenis
Professor

_____

Hans Vandierendonck
Professor, Queen's University of Belfast

Department approval: _____

Angelos Bilas
Professor, Director of Graduate Studies

Heraklion, June 2013

Abstract

Task-based programming models are becoming the most efficient mechanism to achieve performance and programmability on parallel applications. However, the construction of parallel pipelined applications with the use of the state of the art task dataflow programming models still remains hard.

In this thesis we designed and implemented hyperqueues, a programming abstraction of queues that allows different pipeline stages of a parallel application to exchange data with flexibility. Hyperqueues are types of Cilk++ hyperobjects that allow different code paths in a multi-threaded program to maintain coherent local views of the same shared variable. Our design enables shared concurrent views among threads and guarantees the correct execution path by labeling the access type of each thread. We define the semantics of this programming abstraction and describe its implementation on a work stealing Cilk-like scheduler. The main contribution of hyperqueues is the abstraction they offer in the construction of parallel irregular pipelines. We performed an experimental evaluation on the PARSEC benchmarks that can be expressed with pipeline parallelism and we find that hyperqueues overcome the programmability limitations of the state-of-the-art task dataflow models while they achieve performance better than POSIX threads, by a factor of 1.85×, and same as Intel's Threading Building Blocks, with 50% and 10% less code effort (lines of code) respectively. The improvement of hyperqueues on Swan scheduler is demonstrated by a factor of 2.02× over the baseline.

Supervisor professor: Angelos Bilas

Περίληψη

Τα προγραμματιστικά μοντέλα δημιουργίας εργασιών τείνουν να είναι ο αποδοτικότερος μηχανισμός για την καλή επίδοση καθώς και για την προγραμματιστική ευκολία των εφαρμογών. Παρ'όλα αυτά, η κατασκευή παράλληλων εφαρμογών με ομοχειρία, χρησιμοποιώντας προγραμματιστικά μοντέλα δημιουργίας εργασιών, παραμένει μια δύσκολη διαδικασία.

Σε αυτή την εργασία σχεδιάσαμε και υλοποιήσαμε τα hyperqueues, έναν αφηρημένο προγραμματιστικό μηχανισμό, που επιτρέπει στα διαφορετικά στάδια της εφαρμογής να ανταλλάσουν μεταξύ τους δεδομένα με ευκολία. Τα hyperqueues είναι τύποι Cilk++ hyperobjects τα οποία επιτρέπουν σε διαφορετικά κομμάτια κώδικα ενός παράλληλου πρόγραμματος να διαχειρίζονται τοπικά τις κοινόχρηστές τους μεταβλητές. Ο σχεδιασμός μας, ενεργοποιεί κοινά, συνεπή και τοπικά δείγματα μεταξύ των νημάτων και εγκυάται τη σωστή σειρά εκτέλεσης του προγράμματος χρησιμοποιώντας ανάθεση ετικετών ανάλογα με τον τύπο πρόσβασης του κάθε νήματος. Ορίζουμε τη σημασιολογία αυτού του αφηρημένου μοντέλου και περιγράφουμε την υλοποίησή του σε ένα προγραμματιστικό μοντέλο τύπου Cilk++. Η κύρια συνεισφορά των hyperqueues είναι η αφηρημένη διεπαφή που προσφέρουν στην κατασκευή παράλληλων εφαρμογών με ομοχειρία. Η πειραματική αξιολόγηση έγινε στο υποσύνολο των προγραμμάτων από την σουίτα εφαρμογών PARSEC τα οποία μπορούν να εκφραστούν με παράλληλη ομοχειρία. Τα αποτελέσματα δείχουν ότι τα hyperqueues λύνουν τους προγραμματιστικούς περιορισμούς των εξελισσόμενων προγραμματιστικών μοντέλων δημιουργίας εργασιών και επιτυγχάνουν επίδοση καλύτερη από αυτή των POSIX threads κατά συντελεστή 1.85×, και ίδια με αυτή του προγραμματιστικού μοντέλου TBB με 50% και 10% λιγότερη προσπάθεια (σε γραμμές κώδικα) αντίστοιχα. Η βελτιωση στο προγραμματιστικό μοντέλο Swan με την προσθήκη των hyperqueues εκφράζεται κατά συντελεστή 2.02×.

Επόπτης καθηγητής: Άγγελος Μπίλας

Acknowledgements

First and foremost I would like to thank my advisor Dimitris Nikolopoulos and co-advisor Hans Vandierendonck for their patient guidance throughout this work. Their valuable insights and willingness to help made this work possible even from a long distance.

I would like to express my warmest thanks to the CARV Laboratory of the ICS-FORTH and all its members that made this work a great experience: Polyvios Pratikakis, Chrysti Symeonidoy, Apostolis Glenis, Alexandros Labrineas, Maria Chalkiadaki, Ioannis Manousakis, Michail Alvanos, Markos Fountoulakis, Vassilis Papaefstathiou and many more. I feel grateful to my friends Iraklis, Olga, Galateia, Amalia, Argiro, Viola for their support and to Giorgos for his patience and encouragement during the endless skype calls.

Last but not least, I wish to thank my family, my parents Ioanna and David and my sister Deppy for their support and strength they provided me with.

This work was carried with the financial and technical support from ICS-FORTH and the European Commission in the context of the TEXT project (FP7-261580).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis focuses on pipeline parallelism, which is one type of task parallelism. In this parallel programming pattern the application is divided into stages where each stage is a code region that contains a subset of the application's computations. Each stage may contain other kinds of parallelism, e.g. data, task parallelism or nested pipeline parallelism. Pipeline parallelism exploits both data and task parallelism and is prevalent in many emerging applications; for instance image processing applications, data compression etc. Using parallel pipelining is beneficial both for parallelism and program order. On the other hand, applications parallelized using the pipeline model are very sensitive to load balancing: for best efficiency pipelines must prevent resources from being idle, thus all stages must be processing at all times. To achieve this, the programmer must either partition the work into perfectly balanced work items that flow through the pipeline (using Pthreads) or use a programming model such as Cilk++ [12], [13], TBB [14], or StarPU [11] that dynamically shifts the busiest pipeline stages to the idle processors, through work stealing. Due to the structure of a pipelined application, shown in Figure 1, the parallelization of such a program introduces many data dependencies between stages. Thus, the dependency tracking mechanism of programming models is very important in pipeline parallelism. Swan programming model [7] is a new task dataflow model with work stealing that will be studied in this work, able to track dependencies and easy to use.

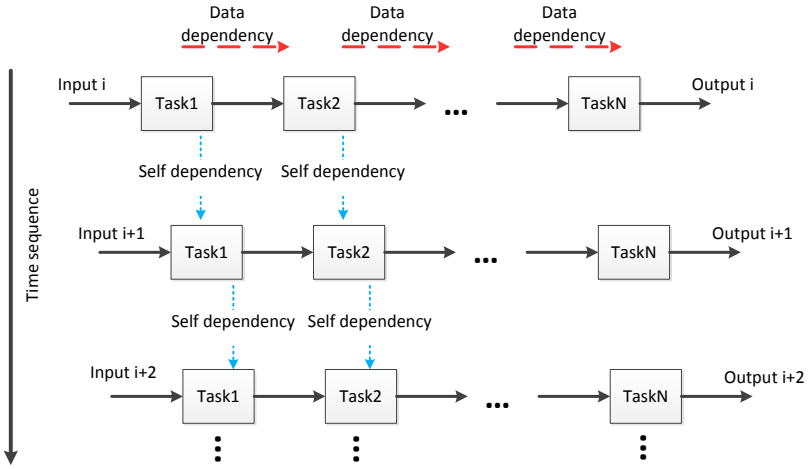


Figure 1: Structure of pipelined parallelism.

The typical way to parallelize pipelined applications is by using the low-level structures of a threading library like Pthreads. This could have bad consequences on understanding and maintain the code as well as on achieving efficiency in terms of load balancing. It is commonly accepted that programming with pthreads is a tedious chore and error prone. With pthreads, the programmer is responsible for detecting the data races, critical sections, dependencies between tasks, and concurrency of the program which makes it a very hazardous way of parallelization. Moreover, load balancing would be very hard to achieve as long as pthreads do not provide a work stealing mechanism so the threads executing the heavier-weight stages would be more loaded than the threads executing the lighter weight stages of the pipeline. Technically, it is almost impossible for most applications to create perfectly loaded pipeline stages that could be balanced through the pipeline without the need of work stealing.

Providing a high-level pipeline abstraction as in Intel's Threading Building Blocks (TBB) [14] programming model can assist programmers to safely develop parallel pipelines. TBB is a parallel programming library developed by Intel Corporation. It offers an abstract set of parallel primitives to parallelize the application by taking advantage of the available hardware resources. TBB supports the task based parallelism to parallelize applica-

tions; it replaces the low level threading libraries (pthreads), and simultaneously hides the implementation of the threading mechanisms for performance and scalability. Moreover, TBB maintains pipeline parallelism by providing a special pipeline class which the programmer has to use in order to create a parallel pipeline. Each stage of the pipeline has to be rewritten as a C++ class and then all stages are joined and the pipeline runs automatically. However, this indicates again a significant amount of effort from the programmer, compared to the way that Swan introduces pipeline parallelism. With TBB it is hard to express irregular pipelines or pipelines with variable input-output rates per stage. It is not applicable to bypass a pipeline stage with TBB and it can be hard to express nested pipelines in order to construct an irregular parallel pipeline. Another observed limitation of TBB is that it cannot support the construction of a recursive pipeline stage; each stage has to return one data item for the next stage in the pipeline.

Swan is a new task based programming model that also simplifies the development of parallel programs by using its dependency tracking mechanisms that dynamically detect and enforce dependencies between tasks. Swan offers a simplified API that allows programmer to freely select the appropriate programming pattern for parallelizing an application. Swan also provides easy ways to implement software of parallel pipelines. It introduces objects, a special data type of Swan that can handle dependencies between tasks. This data type is used as task argument, when an argument is dependent on a prior task or if it is the input of a future task. Objects support versions of themselves, meaning that they can keep all the data modifications and release the correct version to the next task when they are ready. The programmer has to define access mode labels (in, out, inout) for each argument of the pipeline stages in order to track their dependencies.

1.1 Thesis contributions

Swan and TBB programming models may have some limitations on the construction of irregular and variable rate parallel pipelines. For example there are cases where particular stages are not executed for each data item of

the pipeline but their execution may be under conditions, or in an irregular order. Second, there exists a serious limitation on having a stage that may produce more than one data per call in a recursive or iterative way. In Swan's pipeline, that uses versioned object special data structures, each spawned stage returns only one piece of data. Some of the above limitations could be bypassed in Swan and TBB implementations; nonetheless, we address these problems, and create a new special data structure for Swan, that can replace objects under certain circumstances, able to solve all of the above.

In this thesis we present the implementation of a new Swan's feature called hyperqueue. Hyperqueues are used like objects and they also keep versions of their stored data. One benefit from this feature is that programmer can define a pipeline with even higher abstraction and can be able to use special queues that are concurrent and can track dependencies between tasks. Moreover, hyperqueues are flexible when the application's pipeline is not regular as long as, the programmer can add as many elements as needed in each queue. This way data items can easily bypass a specific pipeline stage. Finally, hyperqueues can be used with recursive functions, where each function call produces one element in the queue. Queues also have access mode labels (push, pop) for each argument they appear (such as objects), which are defined by the programmer in order to define dependencies. The following sections analyze thoroughly how hyperqueues are implemented and the ways they can be used by the programmer with specific examples.

The next part of this work consists of the implementation of three of the PARSEC benchmarks [1], bodytrack, ferret and dedup, using Swan programming model and their experimental evaluation. These benchmarks are examples of applications that can use pipeline parallelism. The bodytrack computer vision application recognizes the pose of a human body from an image sequence of multiple cameras. This is a two-stage pipelined benchmark, where each stage contains multiple tasks from multiple parallelized

loops. Ferret is used for content-based similarity search in a set of images. Ferret consists of a six-stage fine grained pipeline with multiple dependencies. We parallelize ferret using objects and hyperqueues, and present how the new Swan feature can solve limitations on ferret benchmark. Finally dedup kernel compresses a data stream with a combination of global compression and local compression in order to achieve high compression ratios. Dedup is implemented using pthreads in a five-stage pipeline. In the transformation of these benchmarks, Swan's special data types were used in order to track dependencies.

The evaluation of the above implementations is performed on a 32-core AMD Opteron processor. Swan implementations are compared against the PARSEC built-in pthreads implementations for all benchmarks, as well as built-in TBB implementation for bodytrack and, ferret and Dedup, TBB implementations from [3].

1.2 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 refers to related work. Chapter 3 presents the background that this work is based on and includes a description of task based programming models, Swan programming model and how programmer can use it, and an Intel's Threading Building Blocks description. Chapter 4 presents the hyperqueue feature introduced in Swan, how it is implemented and its related mechanisms as well as example programs using hyperqueues. Chapter 5 contains detailed explanations of the applications that we parallelize with Swan, the parallelization strategies that we follow and their experimental evaluation in comparison with TBB and pthreads versions of the benchmarks. Finally, Chapter 6 summarizes the work of this thesis and draws conclusions and future work.

Chapter 2

Related Work

The work of this master thesis covers an amount of two topics in HPC. These contain software parallel pipelining and task dataflow parallel programming models. PARSEC Benchmark Suite is used for the experimental evaluation. Significant amount of research has been made on these fields.

Pipeline Parallelism: Pipelining is a parallel programming pattern that allows a program to execute in a decomposed fashion. A pipelined application splits its work into units of code (pipeline stages) and executes them concurrently on multiprocessors or multiple CPU cores. Each pipeline stage takes input from its prior pipeline stage, and produces data for the next pipeline stage.

The PARSEC pipelined benchmarks used in this work were first characterized by Bienia and Li [1], where they proposed bodytrack, ferret, dedup and x264 workloads and analyzed them in a matter of software pipelining.

Thies et al. [4] was the first that proposed an annotation-based method for automatically detecting and parallelizing pipeline parallelism in C programs. Rul et al. [5] improved Thies et al. [4] work by removing the annotations. Both tools are extremely time and memory consuming even on small programs. Their characteristic is to detect templates of pipeline parallelism based on data access. It is hard to examine the parallelized program because the transformed program is in binary format. Despite their drawbacks these tools are very useful starting points in order to detect the pipeline parallelism.

Parallel-Stage Decoupled Software Pipelining (PS-DSP) technique was exploited by Raman et al. [19]. This technique relies on identifying with the help of some small programmer interventions, the pipeline stages, and also on partitioning the threads between the different parallel stages. The thread binding in stages is static. The drawback of both approaches is that they suffer from load imbalance, making the scalability of the application poor.

Stream program characterization in terms of pipeline parallelism was also studied by Thies et al. [18]. Barriers to parallelization, scheduling characteristics and programming styles are the three main aspects that addresses in this work. Even though the language used is the StreamIt language the findings derived by this work have future importance on designing new languages and libraries.

Navarro et al. [2] also used the PARSEC suite and especially ferret and dedup in order to exploit their pipeline parallelism. This work has as main subject to model the performance differences of two dominant programming models, the Pthreads and TBB, by creating analytical models of parallel pipelines based on queuing theory. Furthermore, Reed et al. [3] used TBB to transform ferret, dedup and x264.

Parallel Programming models: Swan [7] is a task based language that detects inter-task dependencies and an extension to the Cilk scheduler. Other languages and schedulers that have been described in the literature are Supermatrix [12], StarPU, SMPSS [16] and CellSS. These languages are able to detect dependencies between tasks dynamically by tracking the memory accesses that each task makes. OpenCL and StarPU allow name based dependency tracking. By that, programmer is allowed to define dependencies between tasks and not care about their memory side-effects.

SMPSS stores object metadata in a hash table that is indexed with the address of the object. This results to an abstract way of metadata lookup and renaming but comes at a cost of additional overhead. Unlike this, programmer has to register the objects used in dependency tracking for StarPU. StarPU runtime system creates a descriptor with the object

metadata, which benefits in comparison to SMPSS approach because the hash table lookup overhead is removed. In Swan each object and its metadata are linked in one data structure retrieving the benefits of StarPU in a better abstraction. Neither SMPSS nor StarPU provide interface for irregular and variable rate pipeline parallelism.

SMPSS stores object metadata in a hash table that is indexed with the address of the object. This results to an abstract way of metadata lookup and renaming but comes at a cost of additional overhead. Unlike this, programmer has to register the objects used in dependency tracking for StarPU. StarPU runtime system creates a descriptor with the object metadata, which benefits in comparison to SMPSS approach because the hash table lookup overhead is removed. In Swan each object and its metadata are linked in one data structure retrieving the benefits of StarPU in a better abstraction. Neither SMPSS nor StarPU provide interface for irregular and variable rate pipeline parallelism.

Benchmark Suite: PARSEC [1] benchmark suite against other workloads are contains pipelined applications. There have been numerous benchmark suites developed, but none of them included parallel pipelined implementations.

SPLASH-2 is a suite composed for multi-threaded applications and hence seems to be an ideal candidate to measure performance of Chip Multiprocessors. However, its program collection is skewed towards HPC and graphics programs. It thus does not include parallelization models such as the pipeline model which are used in other application areas.

SPEC CPU2006 and SPEC OMP2001 are two of the largest and most significant collections of benchmarks. They provide a snapshot of current scientific and engineering applications. Computer architecture research, however, commonly focuses on the near future and should thus also consider emerging applications. Workloads such as systems programs and parallelization models which employ the producer-consumer model are not included. SPEC CPU2006 is furthermore a suite of serial programs that is not intended for studies of parallel machines.

Chapter 3

Background

3.1 Task dataflow programming models

In this thesis we investigate how task dataflow programming models enable accessible and efficient parallelization of applications with mixed pipeline and task/data parallelism. Task based languages facilitate the construction of parallel programs by offering a flexible interface that lets the programmer to freely define code regions as tasks that will execute in parallel. Moreover, task based languages develop a runtime scheduler that is aware of dependencies between tasks. The way this is handled, is by labeling each argument with a memory access mode that describes the side-effects of the present task in its arguments (input, output or input/output labels). Input access mode label indicates a read-only argument, output is for the write-only arguments and input/output is used for read and write access. The scheduler, according to the access mode labels, then can track dependencies between tasks and also change the execution order of the tasks by respecting program order and their dependencies.

The dynamic dependence analysis mechanism that task dataflow languages provide, in combination with the renaming of memory objects can increase parallelism. State of the art task dataflow languages are investigated mostly in the area of high performance computing. Task dataflow languages are beneficial for benchmarks that have irregular parallelism such as Cholesky decomposition or applications with many iterative dependencies such as h264, ferret and dedup that can be parallelized using the pipeline model.

On the other hand, task dataflow languages often exploit a single level

of parallelism meaning that the master thread spawns tasks but the generated tasks cannot set up the execution of new tasks. As a result, these task dataflow languages are not compatible with Cilk-like languages that support recursive fork/join. However the efficient parallelization of many algorithms is well known and in such an algorithm, dependency tracking would be pure overhead.

3.2 Threading Building Blocks

A recently introduced parallelization library is the Intel Threading Building Blocks (TBB) runtime library [14], [17]. TBB is a task dataflow parallel language based on the C++ language and provides an API for expressing parallelism in an abstract way. TBB uses work stealing which helps on improving load balance between cores, thus improving performance scalability. Parallel runtime libraries like TBB make it easier for the programmer to produce parallel software but, on the other hand, they introduce overheads from the dynamic management of parallelism [17]. These parallelization overheads are often implied by the instructions' increment and by the memory latency costs. With the increasing improvement of CMPs' resources, their efficient utilization becomes more and more challenging. In most cases this can be addressed by fine-grained parallelism which takes advantage of higher amounts of resources. However, fine-grained parallelism may introduce higher parallelization overheads.

3.2.1 Runtime

Being a task based library, TBB benefits from the use of tasks for two reasons: first is that the creation and destruction of the tasks is easier than threads, thus tasks can have shorter execution bodies, and second is that tasks reduce load imbalance by being dynamically assigned to the available resources. In applications written with the use of TBB tasks are declared through C++ classes that inherit the attributes of the `tbb:task class`. The special `tbb:task` class provides the virtual method execute(), which the programmer has to overload with the body of the task. Inside TBB runtime the declared task is ready to be launched for execution. The

common way for spawning a task in TBB is by using the method `spawn`(task *t) and provide as argument the new task to be launched. The runtime library schedules the task and executes it by calling the corresponding `execute`() method of the task. Each created task is allowed to create and spawn new tasks being aware of the hierarchical dependencies that may occur.

```
class stage1 : public tbb::filter {
    public:
    void* operator() (void* a) const {
        if(i<N)     return (void*)&a;
        else        return NULL;
    }
}
class stage2 : public tbb::filter {
    public:
    void* operator() (void* a) const {
        int b = (int(float)a);
        return (void*)b;
    }
}
class stage3 : public tbb::filter {
    public:
    void* operator() (void* c) const {
        int* temp = (int*)c;
        *temp++;
        *c = *temp;
        return (void*)c;
    }
}
void pipeline() {
    stage1 s1;
    stage2 s2;
    stage3 s3;
    tbb::pipeline run_pipe;
    run_pipe.add_filter(s1);
    run_pipe.add_filter(s2);
    run_pipe.add_filter(s3);
    run_pipe.run();
    run_pipe.clear();
}
```

Figure 2: TBB pipeline construction

3.2.2 Programming Model and Pipeline Parallelism

Using TBB, the programmer is allowed to generate complex execution task graphs that support their mutual dependencies. The definition and management of dependencies can become a tedious process, and for that reason TBB provides an API through C++ template classes that describe a set of common parallel patterns such as parallel loops and reductions. Such template classes are the `tbb:pipeline` and `tbb:filter` templates that offer to the programmer the ability to construct parallel pipelines. By defining a class that inherits from the `tbb:filter` template class and overloading the `operator()` programmer can declare one pipeline stage. The method `tbb:pipeline:add_filter()` adds the stage to the pipeline, and by executing `pipeline.run()` the parallel pipeline runs while the scheduler is aware of dependencies between the pipeline stages. Figure 2 shows an example of a parallel pipeline with three pipeline stages, executing in N iterations using TBB. Pipeline stops execution at the time that one of the stages returns a NULL token.

3.3 Swan Programming Model

Swan [7] is a parallel Cilk-like programming language that supports task dependency tracking and recursive fork/join. Using Swan the programmer can easily express parallelism using the appropriate parallelization pattern, depending on the algorithm of the application. Furthermore, Swan supports parallel pipeline construction, a programming pattern that is present in many emerging workloads [1] and is studied in this thesis. Cilk partially supports parallel pipelines construction, while TBB offers the programmer an API for easiness in parallel pipelines.

3.3.1 Runtime

Swan's scheduling policy performs work-first scheduling and also supports dependency tracking. The provably-good properties of the Cilk scheduler are violated in Swan's scheduler. Despite this, Swan mimics the good behavior of Cilk scheduler for algorithms with independent tasks or for serial execution of tasks [7].

Versioned object (`object_t`) is a special Swan data structure, which is a type of hyperobject [9], and is used from Swan to track dependencies between tasks. Versioned objects keep track of the necessary meta-data in order to investigate if dependencies are met. Moreover, versioned objects are being renamed so that WAW and WAR hazards are being resolved and as a result parallelism is increased. Finally, versioned objects can allow versioning and enable dependency tracking of every possible user-defined data structure.

The mechanism used in versioned objects for checking the readiness of data uses tickets. The idea is similar to ticket locks. The so called ticket metadata are able to enforce the program order of the serial version of the application.

Performance of Swan has been investigated in [7] where is proven that Swan is as efficient as Cilk and more efficient than SMPSs [16]. In this thesis Swan will be compared to TBB.

3.3.2 Programming Model

Swan contains parallel control statements as in Cilk: `run` initializes the scheduler and starts the parallel execution; `spawn` statement allows a procedure call to proceed in parallel with the caller. Statement `ssync` stalls the execution of a procedure until the completion of all spawned procedures [7]. Statements `call` and `leaf_call` express that a procedure call proceeds sequentially with the caller, making a simple function call. The difference between these two statements is that when using `leaf_call` the scheduler considers this function call as a leaf function call so it stops tracking the program, thus inside a leaf function the programmer cannot use spawn, call or other Swan features. Otherwise, inside a `call`-ed function, programmer can `spawn`, `call`, `leaf_call`, use dependency tracking etc. Figure 3 illustrates programming in Swan.

The special Swan data types indep, outdep and inoutdep enable dependency tracking on tasks. Tasks that take arguments of these types are forced to wait for their preceding tasks to finish or start execution if their dependencies are met. The inner structure of these types contains the

aforementioned versioned objects and gives them the appropriate memory access mode label (in, out, in/out). These types are only allowed as task arguments.

```
typedef float (*block_t)[16];   //16x16 tile
typedef object_t<floag[16][16]> object_block_t;
typedef indep<float[16][16]> in_block_t;
typedef inoutdep<float[16][16]> inout_block_t;

void mul_add(in_block_t A, in_block_t B, inout_block_t C) {
    block_t a = (block_t)A;   // Recover pointers
    block_t b = (block_t)B;   // to the raw data
    block_t c = (block_t)C;   // from the versioned objects
    // ... serial implementation on a 16x16 tile ...
}
void matmul(object_block_t * A, object_block_t * B,
            object_block_t * C, unsigned n) {
    for( unsigned i = 0; i<n; ++i ) {
        for( unsigned j = 0; j<n ++j ) {
            for( unsigned k = 0; k<n; ++k ) {
                spawn( mul_add, (in_block_t)A[i*n+j],
                                (in_block_t)B[j*n+k],
                                (inout_block_t)C[i*n+k] );
            }
        }
    }
    ssync();
}
```

Figure 3: Square matrix multiplication in Swan with enforcement of task dependencies.

In a task spawn, Swan's scheduler recognizes the task dependencies by the arguments with a memory access mode label. If no such label is defined, scheduler spawns the task unconditionally; otherwise it postpones the execution of tasks that have arguments with input mode and forces the execution of tasks that have arguments with output mode. The postponed tasks are stored in a queue with pending tasks. Swan's `ssync` statement mimics the behavior of Cilk's sync, meaning that the execution of the procedure is postponed until all prior tasks have finished execution.

3.3.3 Pipeline Parallelism

The use of Swan programming language can facilitate the construction of a parallel pipelined benchmark. A pipelined benchmark can easily be expressed with Swan's statements and data types. Programmer has to consider the tasks of the workload, what each task produces and what each task consumes, and accordingly define the proper access mode labels. Then

dependencies between tasks are automatically identified by the runtime system. After this step, the parallelization of a simple pipelined benchmark is a straightforward procedure with Swan.

```
void stage1(int i, outdep<float> a) {
    a = (float)i;
}
void stage2(indep<float> a, outdep<int> b) {
    b = (int(float)a);
}
void stage3(inoutdep<int> c) {
    int temp = (int)c;
    temp++;
    c = temp;
}
void pipeline() {
    object_t<float> a;
    object_t<int> b;
    for(int i = 0; i<N; i++) {
        spawn(stage1, i, (outdep<float>)a);
        spawn(stage2, (indep<float>)a,
                      (outdep<int>)b);
        spawn(stage3, (inoutdep<int>)b);
    }
    ssync();
}
```

Figure 4: Pipelined program with 3 stages in Swan

Figure 4 demonstrates a simple example of a pipelined program parallelized with Swan that has 3 pipeline stages, indicating how pipeline stages are spawned with Swan. `stage1` in Figure 3 has an out dependency for object `a` while `stage2` has an input dependency of the same object. This means that `stage2` will postpone until variable `a` has been processed by the first stage. The same happens for variable `b` which has output dependency for `stage2` and input/output dependency for `stage3`. `stage3` will be postponed as well until variable `b` is produced from `stage2`. The program in Figure 2 is a simplified pipeline with light-weight stages but, depending on the algorithm, each stage may contain heavy computations on input or output data.

The code format of Swan's pipeline parallelism is generic and flexible compared to TBB's code format which requires the declaration of template classes that override TBB classes. However, both approaches lack in flexibility in terms of data exchange and stage execution. Figure 5 shows

the same example as in Figure 4 slightly transformed so that it contains one producer stage (`stage1`) which, in one function call, is generating all its data. In this example instead of repeatedly spawning this stage, there is a for-loop

```
void stage1(int i, outdep<float> a) {
    for(int i = 0; i<N; i++) {
       a = (float)i;
    }
}
void stage2(indep<float> a, outdep<int> b) {
    b = (int(float)a);
}
void stage3(inoutdep<int> c) {
    int temp = (int)c;
    temp++;
    c = temp;
}
void pipeline() {
    object_t<float> a;
    object_t<int> b;
    spawn(stage1, i, (outdep<float>)a);
    for(int i = 0; i<N; i++) {
        spawn(stage2, (indep<float>)a,
                      (outdep<int>)b);
        spawn(stage3, (inoutdep<int>)b);
    }
    ssync();
}
```

Figure 5: Pipelined program with a producer stage.

inside the stage which controls the number of data items of production. This example is not applicable in Swan and will not have the desired results. The `outdep` will not keep the versions of the data that will be written in it during the execution of `stage1`, but instead, only one value will be returned as input for `stage2`, and this will be the last value. This problem can appear also in the case where `stage1` is a recursive function; a recursive function would change the content of the Swan object and when `stage1` finishes, the last value will be kept in the `outdep` object.

```
void stage1(int i, outdep<float> a) {
    a = (float)i;
}
void stage2(indep<float> a, outdep<int> b) {
    b = (int(float)a);
}
void stage3(inoutdep<int> c) {
    int temp = (int)c;
    temp++;
    c = temp;
}
void pipeline() {
    object_t<float> a;
    object_t<int> b;
    for(int i = 0; i<N; i++) {
        spawn(stage1, i, (outdep<float>)a);
        if(i!=5)
          spawn(stage2, (indep<float>)a,
                        (outdep<int>)b);
        spawn(stage3, (inoutdep<int>)b);
    }
    ssync();
}
```

Figure 6: Pipelined program with a conditional executed stage.

Figure 6 shows the same pipeline of Figure 4 with a condition in the execution of one pipeline stage. This is also not applicable in Swan, because in the case that `stage2` is not executed due to the condition, the dependencies will be harmed. `stage3` will not start execution at that point as long as the dependency on variable `b` will not be fulfilled.

Chapter 4

Hyperqueue Extension for Irregular Pipeline Parallelism

4.1 Motivation

Swan is a viable approach for parallelizing pipelined workloads like PARSEC [1], however it has limitations when the workloads need variable rate or irregular pipeline parallelism. The use of Swan's objects for dependency tracking requires the spawning of the tasks inside a loop, where each task can produce or consume object data. Figure 2 is an example of the loop that spawns the pipeline stages using objects. Each `spawn` in this loop can produce or consume as many data items, as the number of object arguments of this stage. For example if there is one `outdep` declared, then the spawn of this stage, produces one data unit for this `outdep`; if there is one `indep`, then the corresponding stage consumes one data unit. Inside the loop, there is plurality of this mechanism. Through this usage, one stage cannot produce many data units. Under certain circumstances in an irregular pipeline a stage may need to produce many data units instead of only one. This is the first limitation that objects insert to the parallelization of pipelines. A recursive pipeline stage that reads a directory (which is a procedure that cannot be split through iterative function calls) will never end up to be a Swan's pipeline stage with object-dependencies. In general a stage that produces or consumes data inside a loop cannot be a part of a pipeline with objects. All stages have to contain code that affects or prepares one data unit instead of contain a loop that can affect many data units. Figure 7 shows the

described situation. In this example the first stage (Task1) is producing all the data need to be processed and can be either iterative producer or recursive producer. Then all tasks that are executing the second stage of the pipeline (Task2) are retrieving information from the producer stage. By producing one data item per call this structure is not possible.

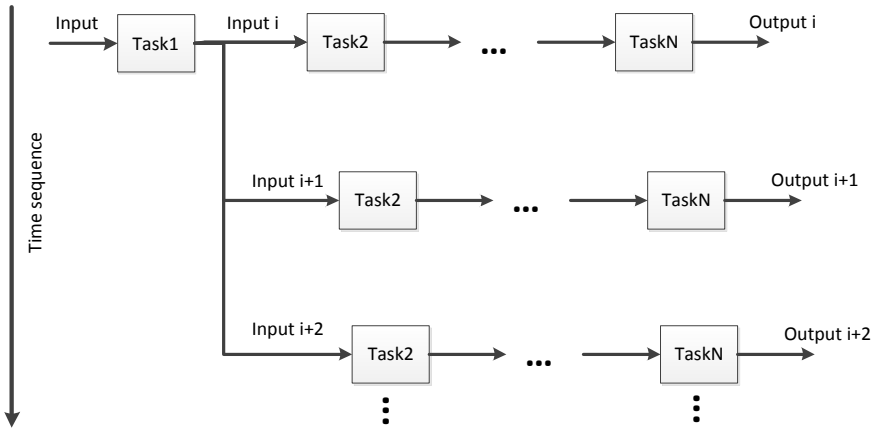


Figure 7: Pipeline structure with one producer task.

This programmability issue by extension can affect the pipeline structure; the computations of a specific pipeline stage may need to be performed in a subset of the entire pipeline's data. This could not be possible if the programmer had to define a loop like Figure 2 demonstrates; within this loop every stage is spawned as many times as the remaining stages. It is not possible to have a condition for the spawning of a stage, because this would harm dependency tracking definitions (`outdep/inoutdep/indep` would not match). Figure 8 shows the structure of the pipeline that contains one conditional stage. This structure would ideally be expressed with code like in Figure 6; this is not possible for Swan objects, as the dependencies would be harmed and introduced a runtime error that would freeze execution of the program.

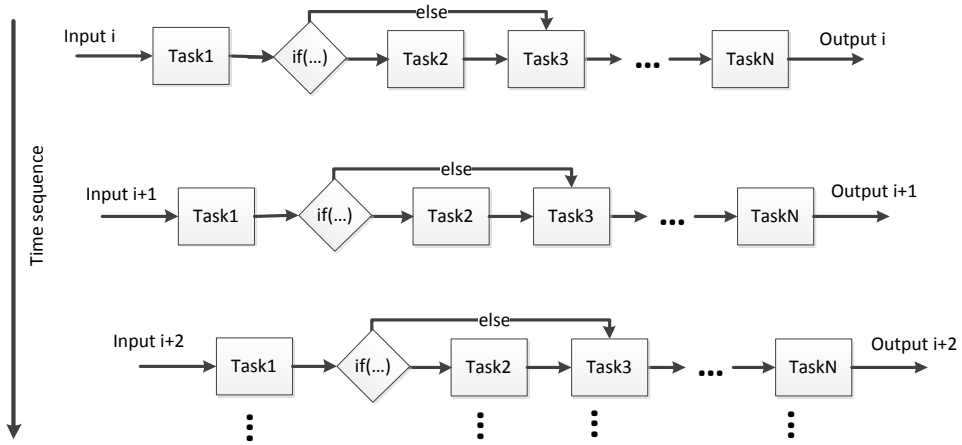


Figure 8: Pipeline structure with one conditional stage (Task2).

The addressing of the aforementioned limitations of Swan's objects led to the implementation of a collection of data where dependencies could apply to one item at a time. FIFO queue is an appropriate data structure for this usage as long as it can assist on preserving the program order in the parallelized program. The idea mainly came from PARSEC built in implementations [1] that use POSIX threads co-operating through blocking queues in data exchange between stages. The main problem is to address all possible consumer-producer cases that a program may contain and how a parallel data-collection should behave in order to be concurrent and coherent.

Hyperqueue is a special Swan data structure that behaves like a queue and can retain all the benefits of the aforementioned versioned-object data structure in plurality. Considering the difficulties faced by the use of Swan programming model we decided to address Swan's pipeline programmability issues and create a unique structure that can handle them. Hyperqueues provide an N-N relationship between producers and consumers. One consumer is able to have as input as many data units as needed and can produce a different amount of data units, on another hyperqueue, for the next consumer. Also, inner task dependencies can be formulated a more flexible way than just any collection of tasks. The task producing the collection does not have to finish execution for the consuming task to start executing. Moreover, by using hyperqueues, scheduler can track dependencies

to higher levels of calling procedures. The pipeline can contain stages that use hyperqueues through nested function calls while in these calls the hyperqueue is updated. Hence, we can benefit from dependence analysis at multiple levels.

Hyperqueues are advantageous by providing:

- **Nested or Recursive calling of tasks:** Swan objects were restricted as they can return just one resulting data unit to the caller level. Hyperqueues are a collection of data where new items can be produced and consumed at will, including at multiple levels of nesting.
- **Collaborative produce and consume:** The hyperqueue's data structure does not contain a fixed number of data items. Each producer can decide the amount of data that the next stage will need, so the programmer does not need to know the number of times that a stage has to be spawned; one is enough to produce the appropriate amount of data. Moreover, a producer-stage can generate data for two or more consumer-stages.
- **Increased concurrency and reduced scheduler overhead:** Consumer tasks do not postpone in the case of using hyperqueues; they are immediately spawned and when data is ready can be consumed. Moreover, a program can have multiple producers performing on the same data for increasing the speed of production.
- **Retaining program order:** Dependency resolution between producer and consumer tasks guarantees that data is produced and consumed in program order without requiring additional effort from the programmer.

4.2 Design

This section presents a detailed description of the design of the hyperqueue mechanism. Choices made and decisions taken about the

programming interface, the implementation of the hyperqueue storage structure, the dependence analysis and the data versioning are presented and discussed.

4.2.1 Interface

Figure 9 shows a simple producer-consumer example using hyperqueues. First comes the definition of the hyperqueue (`queue_t` type), where the data element type is a template parameter to `queue_t`. The hyperqueue special variable is then passed to the tasks using the appropriate access mode labels. Access mode labels in the case of hyperqueues are `pushdep`, for write only memory access and `popdep` for read only. Tasks that contain `pushdep` arguments are considered as producer tasks while tasks with `popdep` arguments are considered consumer tasks. In the example above, the `queue_t` structure is transformed to `pushdep`, and passed to the producer, while it is transformed to a `popdep` to be passed to the consumer. There is only one consumer in this example, so it consumes all the data from the queue. The program may have multiple consumers for one hyperqueue instance or multiple producers. The restriction is, that in order to retain program order, the second consumer will block its execution until the first one has finished. In the case of multiple consumers the programmer is res-

```
void producer(pushdep<int> q) {
    for (int i = 0; i<N; i++) {
        q.push(i);
    }
}
void consumer(popdep<int> q) {
    int j;
    while(!q.empty()) {
        j = q.pop();
        output[j] = j;
    }
}
void caller() {
    queue_t<int> q;
    spawn(producer, (pushdep<int>)q);
    spawn(consumer, (popdep<int>)q);
    ssync();
}
```

Figure 9: Single producer - single consumer example with queues.

ponsible for the amount of data units that each consumer will consume. The following sections contain examples of every possible usage of hyperqueues.

4.2.2 Internal data structures

Figure 10 demonstrates an abstract organization of one `queue_t` instance's storage structure. This structure consists of an ordered list that is organized by queue segments in order to create a high degree of concurrency with little synchronization overhead. A task declaration that contains a `popdep` in its argument list is a consumer task, while a task that contains a `pushdep` in its argument list is a producer task. A consumer task pops data only from the head of the ordered list while each producer (each `pushdep`) is assigned one of the queue segments and pushes data exclusively on its private queue segment. This supports the concurrent execution of multiple producers, as it allows each producer to write data in a different queue and doesn't get influenced from the remaining producing tasks.

The queue segment structures contain fixed size queues for easier allocation/deallocation as well as for performance reasons. A queue segment can be empty, full or busy. When a fixed size queue is full, the producer that owns this queue is responsible to allocate a new queue. Then inserts the new queue right after the one that was full, (list-manipulation) and continues pushing to his new queue. For instance, in Figure 10 the producer that is pushing data on queue segment number 4 has already filled his previous queue (number 3), and has inserted the last segment right after the full one. Also, the queue segment number 1 from which consumer pops data belongs to a producer task that has finished execution and that explains why the producing task is not present in the figure. A busy queue segment means that this queue segment is currently in progress of data production. Before a producer task starts pushing data in its queue segment, it marks this queue segment as busy, so that possible next producers will allocate a new queue segment for pushing their data. In Figure 10, the queue segment number 1 is not busy, because the producer task that was pushing at this queue segment has finished pushing in this segment so has marked the queue segment as

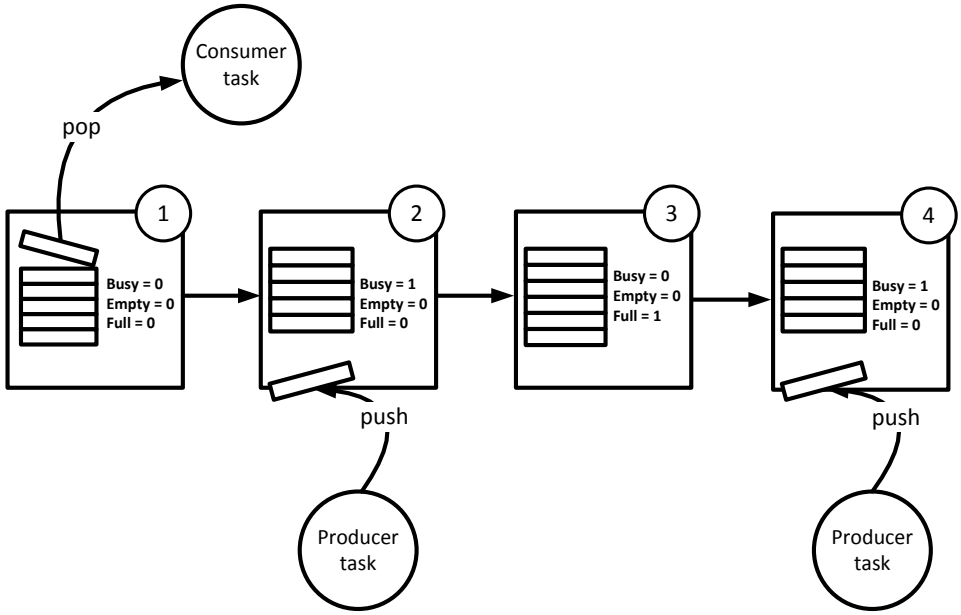


Figure 10: Internal hyperqueue structure design. Shapes numbered 1, 2, 3 and 4 are queue segments containing one fix sized queue each.

not-busy. In accordance, queue segment number 2 is busy because it currently has one active producer task.

4.2.3 Use of the internal data structures

Figure 11 addresses four simple possible use cases and explains how program order is retained; 11(a) is the simple example that Figure 9 shows. In this case, the hyperqueue structure behaves like a simple concurrent queue where one producer writes data and one consumer reads and pops them from the queue. Concurrent non-blocking queues were used so that writing and reading can be performed simultaneously with the added constraint that the consumer should block when it catches up with the producer. To achieve simultaneous reads and writes in the lower level concurrent queue, readers read only from the head of the queue while writers write only in the tail of the queue. The head and tail of the queue are in different cache lines to avoid conflicts. If the queue is empty, the read fails and in the next queue-level the reader attempts to read again. This is repeated until the read is successful. If the queue is full, the producer task allocates a new fixed-size concurrent queue thus write never fails.

```
void caller() {
    queue_t<int> q;
    spawn(producer, (pushdep<int>)q);
    spawn(consumer, (popdep<int>)q);
    ssync();
}
```

(a) One producer and one consumer: behavior is like a concurrent queue; one producer writes data and one consumer reads and pops them.

```
void caller() {
    queue_t<int> q;
    for( int i = 0; i<NUM_PRODUCERS; i++ ) {
        spawn(producer,(pushdep<int>)q);
    }
    spawn(consumer, (popdep<int>)q);
    ssync();
}
```

(b) Multiple producers and one consumer: producers execute in parallel; each producer task pushes in its own privatized queue segment. The consumer checks the queue segments one at a time as they are produced; if a queue segment contains data, the consumer reads and pops the data.

```
void caller() {
    queue_t<int> q;
    spawn(producer, (pushdep<int>)q);
    for( int i = 0; i<NUM_CONSUMERS; i++ ){
        spawn(consumer, (popdep<int>)q);
    }
    ssync();
}
```

(c) One producer and multiple consumers: structure will contain one queue segment; block the new consumers' tasks until their elder have finished reading; only one consumer reads each time.

Figure 11: Use cases of hyperqueues, assuming one queue_t in use.

The program in Figure 11(b) spawns many producer tasks and has only one task to consume the data. The problem here is how to retain the correct order of the data, so that producers enqueue their data in the right order and consumer reads them in the same order they were written. All producers will execute in parallel hyperqueues have to obtain the correct order in writes in the queue. The runtime system was extended to achieve this conceptual serialization of producers without executing them serially. This is possible by utilizing the data structure presented above and by as-

signing a new private segment of the queue to each producer. Before each producer task starts executing in parallel, it is privatizing one queue segment from the queue. Each producer privatizes one segment at the tail of the ordered list and makes it its own queue segment. Then the producer task updates the tail of the list and marks his private queue segment as busy, meaning that this queue segment is currently in use by a producer task so a new producer task is now allowed to perform writes in this segment. The next producer task will check if the tail of the list is busy, and if this is true, it adds a new queue segment at the tail of the list. These actions are performed before parallel execution starts from inside the runtime's mechanisms for recognizing queues as Swan's special objects. All the above result to a structure as the one of Figure 10 where each producer task adds elements in its private queue segment, which is placed in the correct position in the list, so that data are in order.

During the pop operation the consumer is not aware of the number of queue segments in the list or the number of producers. Data sent to the consumer comes from the oldest producer and are always indicated by the first queue segment in the list. The head of this queue segment is the data returned to the consumer. If the first queue segment is empty, but it is marked as busy, the consumer waits for the producer owning this segment to add more data. Otherwise, if the queue segment on the head is empty and is marked as non-busy, the consumer removes the queue segment from the list, and tries reading from the next segment with the same procedure.

Code in Figure 11(c) indicates one producer and multiple consumers. In this example, the queue structure will contain one queue segment (except if the producer needs to add more data than one queue segment can store) and the consumers will all try to read from the same queue segment. The consumers are forced to execute serially. The motivation for the serialization of consumers is that the runtime is not aware of how many items each will consume. It is thus impossible to let the consumers execute in parallel and enforce that they consume data in logical program order. This ensures

that all the pop operations are serialized and each consumer reads the data in order. This is the only case of inter-task dependence on queues that prohibits the task from starting execution immediately at the point of spawn. This solution indicates flexibility on the reads but the programmer should be careful on how many data items tasks will pop.

Figure 12 illustrates an example of a recursive producer and one consumer. The scenario in Figure 12 is the allocation of multiple queues each one containing one data item. First the spawning of `rec_producer` will privatize one segment and push in this segment the first value which is 0. After that, `rec_producer` will call itself to push the incremented value (1) and by this call a new queue segment will be created and inserted after the existing queue segment of the caller (`rec_producer`). This case differs from the default segment privatization, as long as segment is not allocated every time at the tail of the list, but it is allocated in the position right after the caller's (`rec_producer`) queue segment. The reason for this is to preserve program order even if the program spawns more than one producer tasks.

```
void rec_producer(pushdep<int> q, int i) {
    if(i<N) {
        q.push(i);
        call(rec_producer, (pushdep<int>)q, pos, ++i);
    }
    return;
}
void caller() {
    queue_t<int> q;
    spawn(rec_producer, (pushdep<int>)q, 0);
    spawn(consumer, (popdep<int>)q);
    ssync();
}
```

Figure 12: Use case of hyperqueues with one recursive producer

Figure 13 shows the code for one recursive and one normal producer and one consumer while Figure 14 illustrates how queue structure will be transformed when this code executes. First, two segments will be privatized, the first one will be segment number 1 from Figure 14 and the second will be segment number 4. Segment 1 is the segment where `rec_producer` will push its first item. When the first item is pushed, `rec_producer` will call itself, and by that a new queue segment, segment number 2, will be created

and inserted right after segment number 1. `rec_producer` will call itself again and the same will happen, creating segment number 3 after segment number 2. Segment number 4 remains to the correct position, as it was privatized in the tail of the list. This mechanism results to data ordering in the queue structure so it is assured that consumer will read them in order.

```
void rec_producer(pushdep<int> q, int i) {
    if(i<3) {
        q.push(i);
        call(rec_producer, (pushdep<int>)q, pos, ++i);
    }
    return;
}
void caller() {
    queue_t<int> q;
    spawn(rec_producer, (pushdep<int>)q, 0);
    spawn(producer, (pushdep<int>)q, 0);
    spawn(consumer, (popdep<int>)q);
    ssync();
}
```

Figure 13: Example - code of one recursive and one normal producer with one consumer

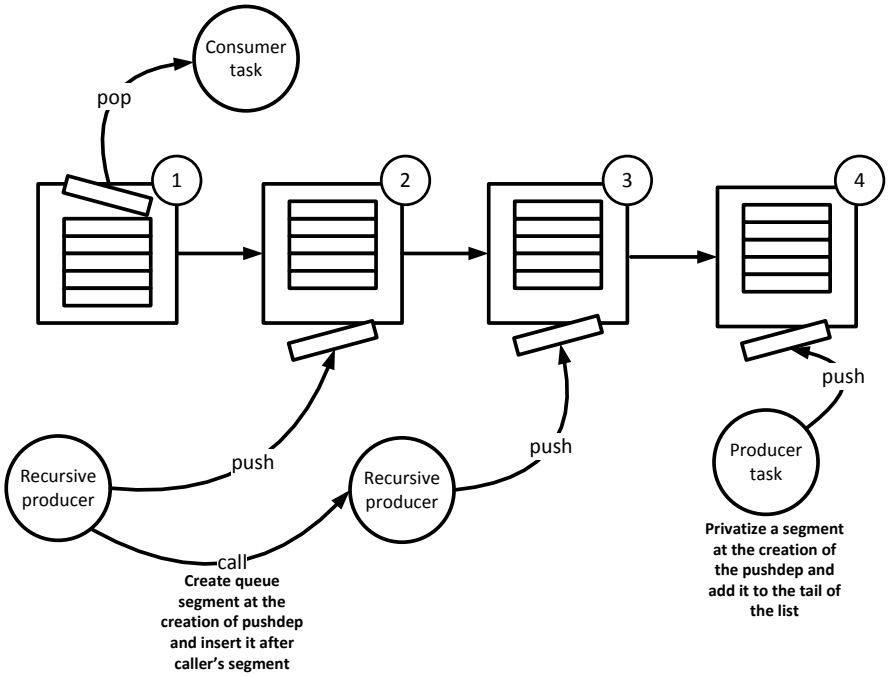


Figure 14: Two unrolled recursions of the code in Figure 13 and how this works in queue structure.

4.2.4 Versioning

Hyperqueues are also versioned objects like Swan objects. Those are types of hyperobjects which allow different code paths in a multi-threaded program to maintain coherent local views of the same shared variable [9]. The corresponding shared variable in the case of hyperqueues is the hyperqueue structure described above and the local view of this is the private queue segment that each producer holds for pushing data. Hyperqueues are Swan data structures, so they are recognized by the scheduler when they appear in the argument list of `spawn` and `call`. This is forcing the scheduler to control their dependencies, spawn them if they are ready or postpone the task and issue it when all its hyperqueue arguments are ready to use.

Each time a producer task is spawned (indicated by the `pushdep` argument) a new queue version is allocated. This queue version contains:

a. Pointer to the segmented queue structure
b. One private queue segment
c. Ticket metadata for this version
d. One reference counter

Consumer uses the pointer to the segmented queue structure in order to retrieve data from it. On the other hand, producer needs a new private queue segment along with the segmented queue pointer. Each producer task needs a queue version in order to keep the pointer of its private queue segment and to be up to date regarding the segmented queue. When a new queue version is created, the caller procedure (the parent of the producer task) is accessing the same queue version as the producer task. This is necessary so that this version is visible when a consumer is spawned in the parent procedure and needs to consume data from the producer spawned earlier. This renaming mechanism allows all tasks operating in a queue to be up to date. Ticket metadata are used for dependence analysis. The reference counter helps on the effective deallocation of the queue structures; all queue structures are reference counted and automatically deallocated when the last reference to it is dropped.

4.2.5 Dependence Analysis

The way queues perform dependency tracking is by using Swan's ticket metadata. Each allocated queue version contains its metadata. The metadata structure holds the number of readers and the number of writers that the instance of the queue it belongs has. In the case of hyperqueues, the number of writers is needless because a producer or a consumer can be issued regardless of the fact of having a writer or not. What is used is the number of readers because if the program has multiple consumers, the younger consumers need to be postponed. Every time a queue argument is issued, if it is a reader, the number of readers in the metadata structure is increased. Then, before scheduler issues the next consumer argument, checks if its metadata contains prior readers. If it has, argument is not issued; hence scheduler postpones the whole task from execution.

The above metadata checks are performed through functions that control the dependencies of task arguments. Functions like these were implemented for objects as well but with different mechanisms. Scheduler sees an abstraction of these functions as long as they are overloaded. When a task is created in Swan, the runtime initializes the argument and immediately checks if it is ready by checking its metadata. If it's not, at a later time, scheduler performs the check of the metadata readiness again. When it is proved to the scheduler that the argument is ready, the runtime updates the argument's metadata and then it depends on the remaining task arguments if the task is going to be postponed or if it is ready to execute. Finally, at the end of a task execution, scheduler updates the metadata (decrease readers) so that other possible blocked arguments and tasks can be issued. Figure 15 represents how the above checks work when a queue argument is encountered.

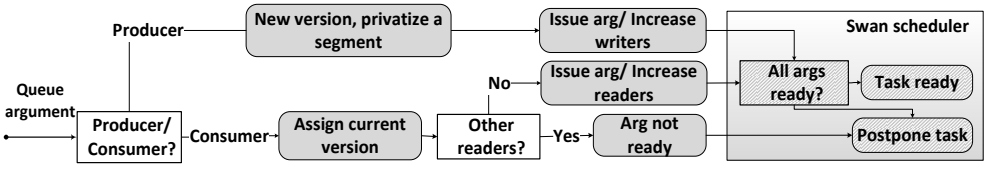


Figure 15: Queue arguments recognition and effect on Swan scheduler

Chapter 5

Experimental Evaluation

This chapter describes how Swan is used in order to parallelize pipelined benchmarks. We parallelize bodytrack, ferret and dedup from the PARSEC benchmark suite [1] which are characterized of their pipelined parallelism. Through these experiments Swan programming model can be evaluated regarding its correctness, ease of use and performance. We compare Swan with TBB and pthreads implementations of these benchmarks. We compiled all versions of benchmarks using g++ 4.6.3 on Ubuntu 12.04 operating system. The experimentation machine was a dual-socket AMD Opteron processor (6272) with 8 cores (2GHz) with 2 hardware threads per core in each socket, counting in total 32 threads. The threads in each socket share a 16MB L3 cache. Speedups are computed relative to the serial elision of the benchmarks, which we compile using gcc 4.6.3. In each case the optimization level is set to –O4. For POSIX implementations, where the number of software threads being created overcomes the number of cores, we use the command `numactl` to specify the number of cores that the application will use. Moreover, this command is used for all the experiments for specifying the most efficient core binding in terms of socket and cache usage. AMD CodeAnalyst is used for the profiling of the load balance between cores.

5.1 Bodytrack

The bodytrack application from PARSEC benchmark suite [1], [20] is an Intel RMS computer vision workload that recognizes the pose of a human body from an image sequence of multiple cameras. The tracking of the human body is performed through an annealed particle filter that tracks the pose by using edges and the foreground silhouette as image features. The input given to bodytrack application consists of sets of frames from different placed cameras, where each set contains the same number of distinct time frames. Bodytrack tracks the human body for each distinct frame in all frame-sets. The output of the application is a set of images with highlighted parts of the body. Figure 16 shows the output of one input frame; body parts are highlighted in each camera's frame with the same color according to each specific body part. The application uses parallel pipelines to perform I/O asynchronously. The bodytrack parallelization exploits mostly data-parallelism because its computationally intensive parts use at most parallel loops rather than parallel pipelines.

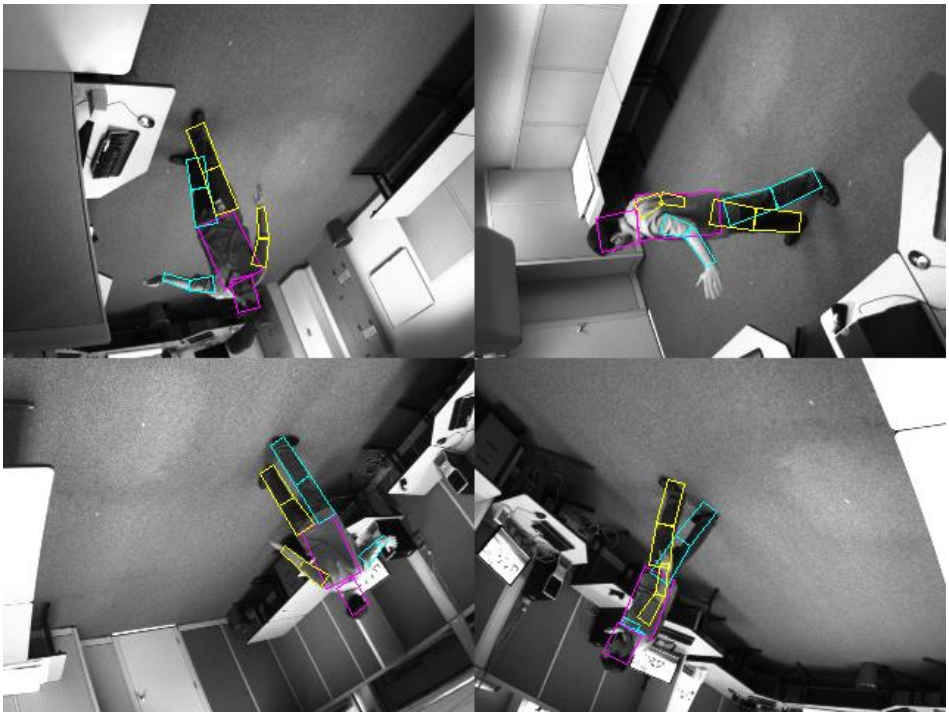


Figure 16: Output of the bodytrack benchmark.

Bodytrack is parallelized using pthreads by implementing a persistent thread pool. Images from different cameras are loaded asynchronously from the disk in order to overlap with the computational part. One main thread sends tasks to the thread pool when it reaches a parallel region of the code. When the main thread receives the result from the worker threads in the thread pool, it resumes the execution of the program [20].

5.1.1 Parallelization

Our implementation uses TBB version as a starting point following the same pipeline stages and parallel loops. The implementation has two major stages which have been implemented using the pipeline model of Swan runtime. Table 1 shows the time breakdown of the two bodytrack pipeline stages; both stages are executed in the same number of iterations, once for each input frame.

Bodytrack	Num iterations	Time (sec)	Time %
Tracking Model	261	20.196	7
Particle Filter	261	267.233	93

Table 1: Bodytrack number of iterations and stages time breakdown

1st stage: TrackingModel – Edge detection and Edge smoothing: In this stage bodytrack employs a gradient based edge detection mask to find edges. After the edge detection bodytrack performs edge smoothing by using a Gaussian filter of size 7 x 7. The edge smoothing result is then used to produce a map of pixels flagged with 0 or 1 each. The value of each pixel is relative to its distance from the edge. Those steps are executed in parallel for each distinct camera frame using the Swan `parallel_for`, and also constitute a pipeline stage.

2nd stage: ParticleFilter: This stage computes weights for the particles by evaluating the foreground silhouette and the image edges produced in `stage1`. This is the most computationally intensive part of the application

```
void stage1(TrackingModel *model, inoutdep<ImageSetToken*> token,
            int currFrame);

void stage2(ParticleFilter<TrackingModel>* pf,
            indep<ImageSetToken*> token,
            string *output,
            int currFrame);

for(int i = 0; i < frames; i++) {
    spawn(stage1, &model, (inoutdep<ImageSetToken*>) token, i);
    spawn(stage2, &pf, (indep<ImageSetToken*>) token, &output, i);
}
ssync();
```

Figure 17: Iterations of two pipeline stages of bodytrack

while it computes the weights once for every annealing layer of each time step. Furthermore, this stage resamples particles and creates a new set of particles. At this point Swan's `parallel_for` was used in order to distribute the work though cores. Also, the next step of this stage is fully loop parallelized, through the number of particles. After the particle resampling is completed, the task that was assigned to execute this pipeline stage writes the result of the above computations to the output file `output.txt` and produces the corresponding to this time step output image.

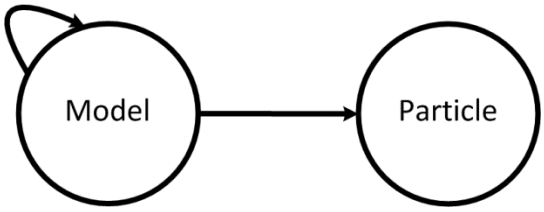


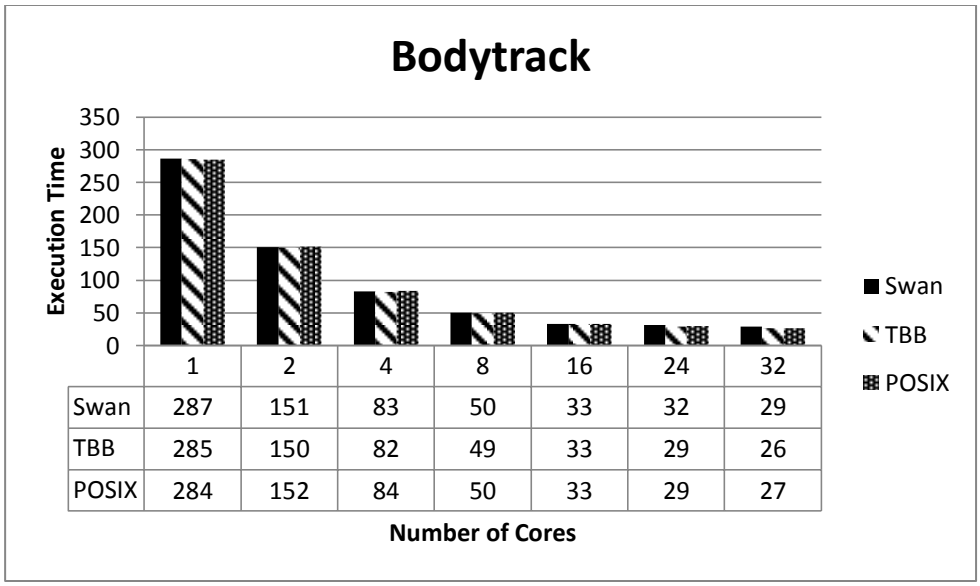
Figure 18: Bodytrack stages. Arrows show dependencies.

Figure 17 shows the code for the spawning of the above two stages of bodytrack. Figure 18 shows the task-graph for this code; there is one input token in both stages which is updated and used in all iterations. This is flagged by the `inoutdep` and `indep` in `stage1` and `stage2` respectively.

5.1.2 Evaluation

Figure 19 illustrates the execution time for Swan, TBB and POSIX bodytrack implementations. The difference in the execution times of the

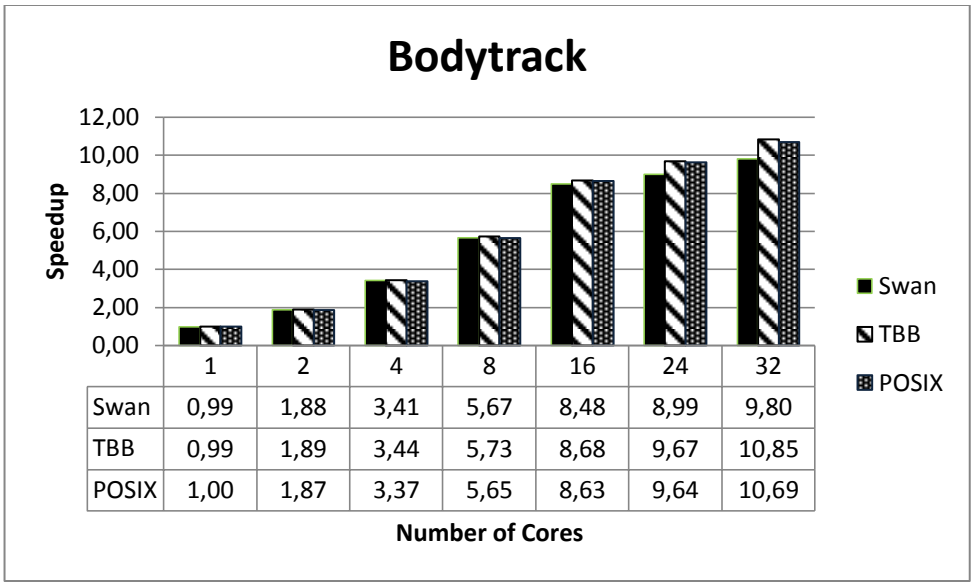
three programming models is not significant. Swan has an additional overhead than the other two implementations which is approximately 3 seconds. We attribute this to the need of the compilation flag `-fno-omit-leaf-frame-pointer` for Swan. This flag, as experiments confirm, imports overhead to the execution time. Multiple experiments of the same code could verify this statement. The original serial bodytrack version compiled without `-fno-omit-leaf-frame-pointer` takes 283 seconds, while the same version compiled using `-fno-omit-leaf-frame-pointer` takes 287 seconds. This flag is blocking some optimizations that the -O4 introduces. It is a mandatory flag for Swan compilation because of the way that Swan calls (spawns) functions; by using this flag, it is easier to spawn because the `rbp` register is used to point to the function stack pointer. If it doesn't, then the compiler knows how to compute it from `rsp` (stack pointer), but Swan cannot be aware of this in library code.



	1	2	4	8	16	24	32
Swan	287	151	83	50	33	32	29
TBB	285	150	82	49	33	29	26
POSIX	284	152	84	50	33	29	27

Figure 19: Execution time of bodytrack, TBB, pthreads and Swan implementations.

Figures 20 and 21 represent the speedup of the above execution times. Because of the limitation of the compilation flag mentioned before, Swan is slightly less efficient.



	1	2	4	8	16	24	32
Swan	0,99	1,88	3,41	5,67	8,48	8,99	9,80
TBB	0,99	1,89	3,44	5,73	8,68	9,67	10,85
POSIX	1,00	1,87	3,37	5,65	8,63	9,64	10,69

Figure 20: Speedup of Swan, TBB and POSIX for bodytrack. Specific core counts.

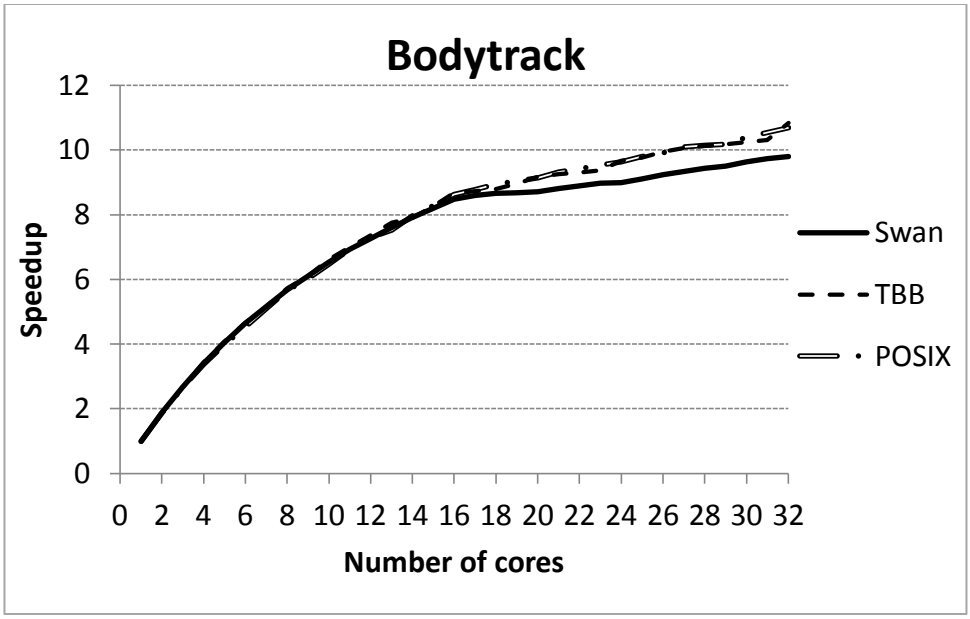


Figure 21: Speedup of Swan TBB and POSIX.

5.2 Ferret

Ferret is a content-based similarity search application based on the Ferret toolkit. In PARSEC benchmark suite ferret compares a set of images and returns which images contain the same kind of object. The pipeline

implementation of this benchmark contains six stages of processing. First and last stages are input (`load` stage) and output (`out` stage) of the algorithm. The middle four stages implement image segmentation (`seg` stage), feature extraction (`extr` stage), indexing of candidate sets (`vec` stage) and ranking (`rank` stage). Segmentation is a process whereby an image is divided into smaller areas that display different objects. A high or low weight of interest is assigned in each such area. An image segment that belongs to the background of the image usually has a low weight of interest, while a segment that belongs to the foreground has the highest weight of interest. After this computation stage, extract stage, creates a feature vector, which is a multi-dimensional mathematical description of the segment contents that encodes the fundamental image properties (color, shape, area). Thereafter, the `vec` stage queries the image database to obtain a candidate set of images. The `rank` stage then determines a rank for each image and sorts the images in descending order according to their calculated rank.

The PARSEC pthreads implementation of ferret is a six-stage pipeline with all the aforementioned stages, where input and output stages are executed by one thread each. The remaining middle stages use oversubscription: specifying the program to run with x threads would create x threads for each of these stages. Blocking queues configured for a maximum of 20 items were used to pass tokens between stages.

Figure 22 illustrates the task-graph of ferret. Each stage communicates with the next one through blocking synchronization queues. The `out` stage writes out the results of the whole benchmark in the output file.

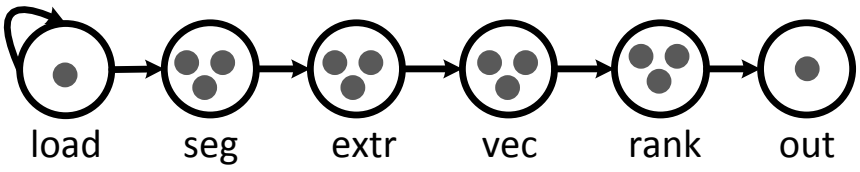


Figure 22: Task-graph for POSIX ferret implementation. Arrows show dependencies. Gray dots indicate the number of sw threads that execute each stage.

5.2.1 Parallelization

Ferret introduces an important challenge in Swan's parallelization: the presence of a recursive producer pipeline stage. Table 2 shows how this issue was skipped in order to implement a working parallelization using Swan programming model. The recursive stage was kept serial outside the parallel pipeline in the two parallel versions of ferret using Swan. There is an important constraint in Swan programming model: the pipeline of a benchmark cannot contain a stage that is a producer of multiple data items. Each call of a stage produces only one item, so the program needs to iterate through the number of data items that the stages have to produce. Ferret benchmark contains a producer-stage that is parsing a directory. This operation is not possible with multiple calls to this stage, because the function needs to be aware of the current position inside the directory. TBB programming model also cannot support the presence of a recursive producer pipeline stage. The TBB implementation of ferret uses a stack for storing the current position in the directory for each stage call so that the next task that loads data retrieves the correct copy of the directory pointer. Swan is intended to implement pipelined parallel applications with low effort and the minimum modifications necessary to the algorithm. Table 3 shows the time breakdown of each pipeline stage in addition to the number of iterations that each stage is executed.

Challenge	Solutions	Appears in
• Recursive pipeline stage	• Serialize stage	• Ferret 5-stage, Ferret 2-stage

Table 2: The challenge faced in Ferret's parallelization was the recursive pipeline stage.

Ferret	Num iterations	Time (sec)	Time %
Input (load)	1	34.000	4.480
Segmentation (seg)	3500	26.800	3.570
Extraction (extr)	3500	2.773	0.350
Vectorizing (vec)	3500	133.939	16.200
Ranking (rank)	3500	603.286	75.300
Output (out)	3500	2.000	0.100

Table 3: Ferret number of iterations and stages time breakdown.

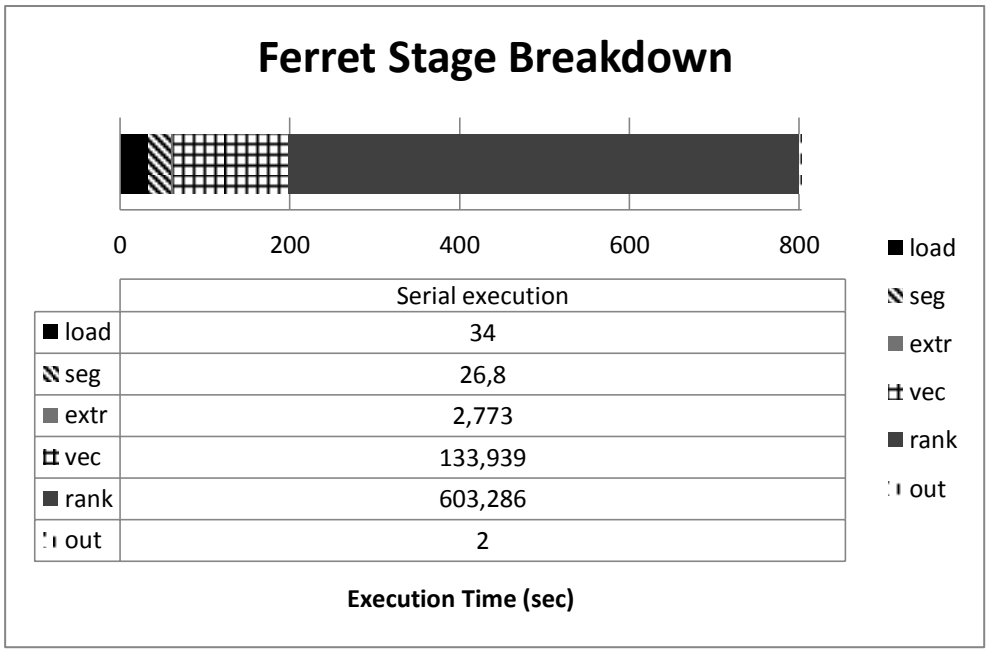


Figure 23: Stage breakdown of ferret

- **Ferret 5-stage pipeline with Swan**

In the first parallel version of ferret using Swan programming model, the pipeline stages are kept the same as POSIX implementation. The main challenge is the first stage, (input stage) which loads the input data into

data structures. This is a recursive pipeline stage, and that fact restricts Swan to analyze dependencies. Hence, this stage is removed from the parallel pipeline (Figure 24) for this implementation, fact that reduces parallelism. Also this gives an important increase in memory consumption and degrades memory locality as all data have to be stored at the same time, which would not happen in the pthreads/TBB implementations. Another difference between Swan and POSIX implementations is that with Swan last stage is executed by multiple threads whereby each thread outputs a subset of the data in the right order. To achieve the serial output of results, an `inout` dependence (self-dependence) was added in the `out` stage. With this dependence, each `out` task that is spawned has to wait for the previous `out` task to finish execution in order to start. The queues used in POSIX implementation (except for the one that sends the data from `load` stage to `seg` stage) have been replaced by the versioned objects that Swan programming model offers, for activating dependence analysis between pipelined tasks.

Figure 25 shows the code for this implementation. Tasks are defined as functions that have as arguments `indep/outdep/inoutdep` objects. Then, by the time they are spawned, dependence analysis is turned on for these tasks.

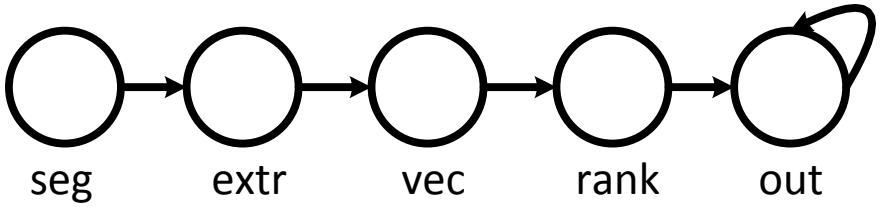


Figure 24: Ferret 5-stage pipeline. Omitting input stage. Arrows show dependencies.

```
void load(load_t *load_array);
void seg(load_t load_data, outdep<seg_t> seg_data);
void extr(indep<seg_t> seg_data, out-dep<extr_t>extr_data);
void vec(indep<extr_t>extr_data, out-dep<vec_t>vec_data);
void rank(indep<vec_t>vec_data, out-dep<rank_t>rank_data);
void out(indep<rank_t>rank_data, inoutdep<int>wait);

int main(){
    int array_size = leaf_call(load, load_array);
    for(int i = 0; i<array_size; i++){
      spawn(seg, load_array[i], (outdep)seg_data);
      spawn(extr, (indep)seg_data, (outdep)extr_data);
      spawn(vec, (indep)extr_data, (outdep)vec_data);
      spawn(rank, (indep)vec_data, (outdep)rank_data);
      spawn(out, (indep)rank_data, (inoutdep)wait);
  }
}
```

Figure 25: Code for ferret 5-stage pipeline with Swan

- **Ferret 2-stage pipeline with Swan**

A second approach to ferret parallelization was to merge four of the stages into one single stage. This could be beneficial because, as Table 3 indicates, stages `seg`, `extr` and `vec` have minimal overhead, comparing to stages `rank` and `out`. With this pipeline these two overhead-adding stages could be better overlapped. We develop a new task function in order to serialize the four merged tasks. The output function remains unchanged with the same dependencies. However, in this implementation input is provided from the new task function. Figure 26 shows the task-graph of this implementation. `load` stage remains serial, outside the pipeline as before. `out` stage is aligned by using a self-dependence for writing the file in the right order. Figure 27 shows the code for what is described here.

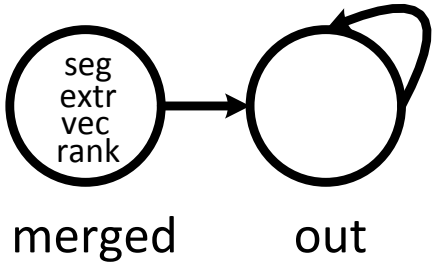


Figure 26: Taskgraph of 2-stage pipeline of ferret

```
void load(load_t *load_array);
void seg(load_t load_data, seg_t seg_data);
void extr(seg_t seg_data, extr_t extr_data);
void vec(extr_t extr_data, vec_t vec_data);
void rank(vec_t vec_data, rank_t rank_data);
void merged(outdep<rank_t> rank_data);
void out(indep<rank_t>rank_data,
         inoutdep<int>wait);

void merged(outdep<rank_t> rank_data_out){
    call(seg, load_data, seg_data);
    call(extr, seg_data, extr_data);
    call(vec, extr_data, vec_data);
    call(rank, vec_data, rank_data);
    rank_data_out = rank_data;
}
int main(){
  int array_size = leaf_call(load, load_array);
    for(int i = 0; i<array_size; i++){
     spawn(merged, (outdep<rank_t>)rank_data);
     spawn(out, (indep<rank_t>)rank_data,
                (inoutdep)wait);
  }
  ssync();
}
```

Figure 27: Code for ferret 2-stage pipeline

- **Ferret 6-stage pipeline with Swan using hyperqueues**

By introducing the hyperqueue in Swan we could solve the main challenge of the ferret implementation. Using hyperqueues in combination with objects can assist Swan to handle the parallelization of the first stage, which introduces overhead of 34 seconds, along with the remaining parallel pipeline stages. Figure 23 shows the ferret stages time breakdown. We overlap the first 34 seconds of its execution time by spawning the first stage and inserting the load data in the queue.

The second stage of ferret is now able to start immediately right after load stage produces the first data unit to consume. This overhead is marginal keeping in mind that the production of 3500 data items is done in 34 seconds. This way we can achieve better load balancing of cores and also overlap the first 34 seconds overhead of the serialization of the first stage.

```
void load(pushdep<load_t>load_queue);
void seg(popdep<load_t>load_queue, outdep<seg_t> seg_data);
void extr(indep<seg_t> seg_data, outdep<extr_t>extr_data);
void vec(indep<extr_t>extr_data, outdep<vec_t>vec_data);
void rank(indep<vec_t>vec_data, outdep<rank_t>rank_data);
void out(indep<rank_t>rank_data, inoutdep<int>wait);

void pipeline() {

    queue_t<load_t> load_queue;
    spawn(t_load, (pushdep)load_queue);

    for(int i = 0; i<array_size; i++){

        spawn(seg, (popdep)load_queue, (outdep)seg_data);
        spawn(extr, (indep)seg_data, (outdep)extr_data);
        spawn(vec, (indep)extr_data, (outdep)vec_data);
        spawn(rank, (indep)vec_data, (outdep)rank_data);
        spawn(out, (indep)rank_data, (inoutdep)wait);
    }
    ssync();
}
```

Figure 28: Implementation of Ferret using hyperqueues.

Figure 28 shows the implementation of ferret with queues. The first stage is spawned and executed by one core, simultaneously with the remaining stages. In this case the program has one producer and multiple consumers so we know that tasks of `seg` stage will be postponed in a way that one `seg` task runs each time, as long as there will be more than one readers of the queue.

Figure 29 shows the task-graph of this implementation. Stages `seg`, `extr`, `vec`, `rank` and `out` iterate through the number of queries, while stage `load` is spawned only once. This is indicated in the figure by the dashed arrow showing that this dependence exists for queue objects and not for objects inside the loop of spawning pipeline stages.

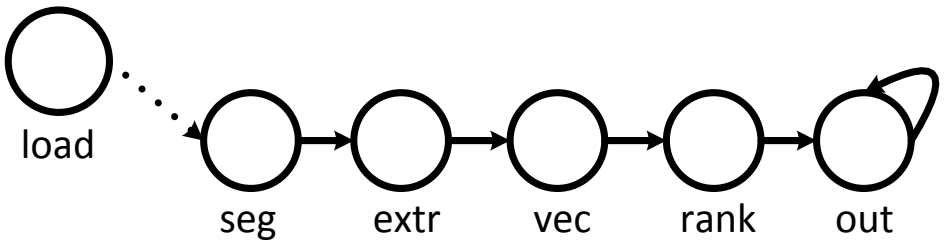


Figure 29: Dashed arrow: hyperqueue; solid arrows: dependencies tracked with objects.

5.2.2 Evaluation

The ferret benchmark contains a built in PARSEC implementation of POSIX threads that uses cores in a different way than the other programming models. It uses multiple light-loaded software threads per pipeline stage so software threads in use can overcome the number of hardware threads of the CPU, contrary to Swan and TBB that use as many software threads as are the hardware threads. Figure 30 demonstrates the execution time of Ferret with multiple software threads per pool, on multiple core counts. 10 threads per pool for the POSIX implementation of ferret means 2+10*4 = 42 concurrent software threads. Number of cores on *x* axis, indicates the number of cores those software threads are running onto, with the help of `numactl`. Speedup is obtained while the number of cores increases and there are available software threads to take advantage of those resources. We compare the best POSIX configuration, which uses 10 threads per pool as Figure 30 implies with Swan and TBB implementations. The TBB implementation we use is from [3]. It should be reported that POSIX configurations that use more than 10 threads per pool introduce overheads that lower performance.

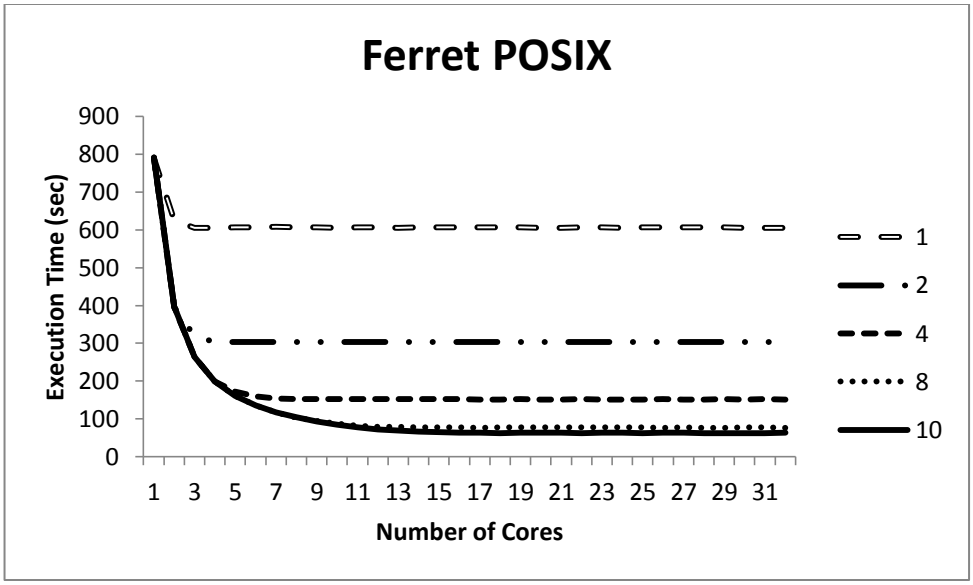


Figure 30: Ferret POSIX execution time with multiple numbers of threads per pool.

For Swan, two implementations using objects were described: Swan 2-stage pipeline and Swan 5-stage pipeline. Those two implementations have very close execution times, and indicate the same speedup in ferret. We will refer to them both as Swan implementation or as Swan – object implementation, as long as their graphs completely overlap. Figures 31 and 32 show the execution time and speedup respectively, of Swan, TBB and POSIX ferret implementations. Swan's performance falls short due to the additional overhead that the serialized `load` stage introduces, which is approximately 34 seconds for each execution, regardless the number of cores it is running onto.

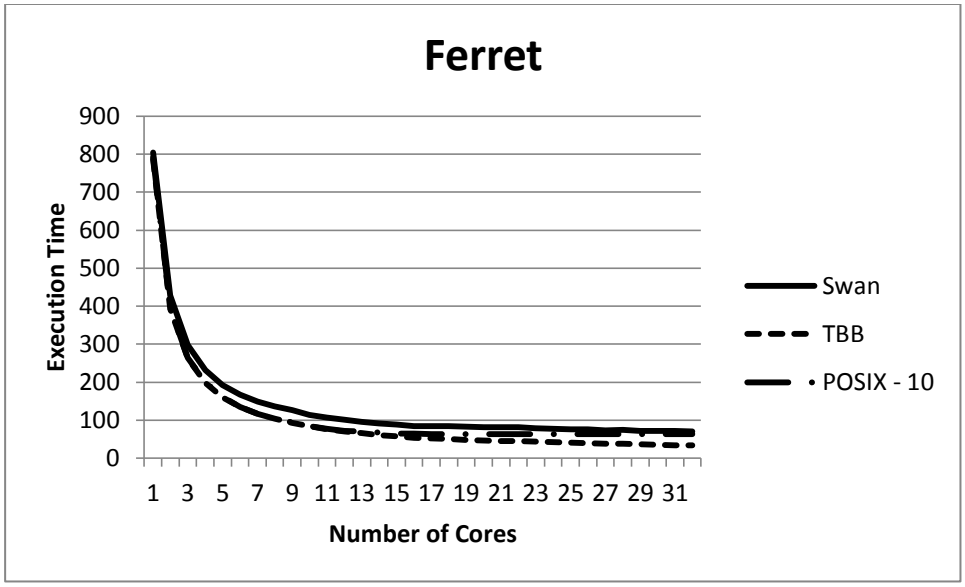


Figure 31: Ferret execution time of Swan, TBB and POSIX implementations.

Figure 33 shows the execution time for 1, 2, 4, 8, 16 and 32 core counts. Comparing the execution times from Figure 33 leads to the conclusion that TBB performs better than Swan because of the load stage serialization while POSIX performs better than Swan for the same reason.

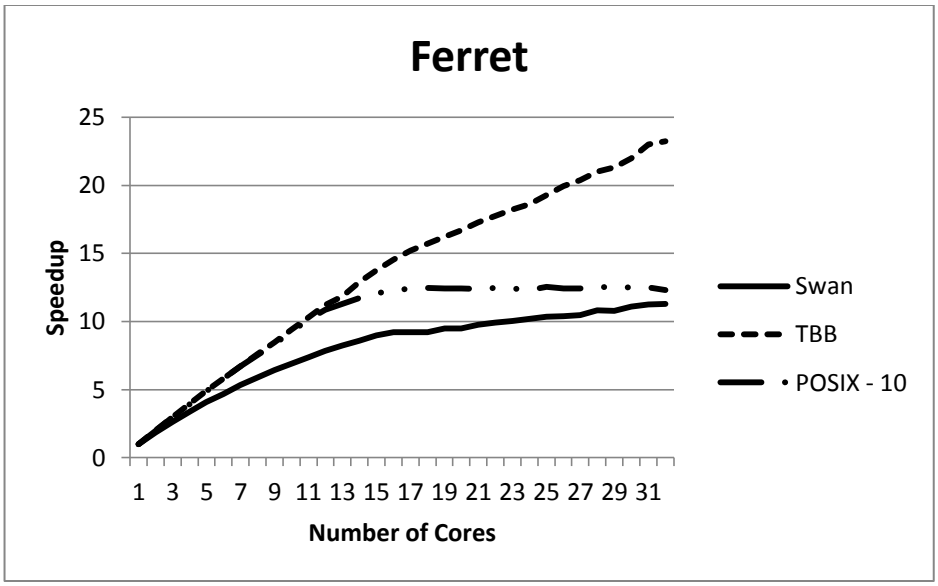
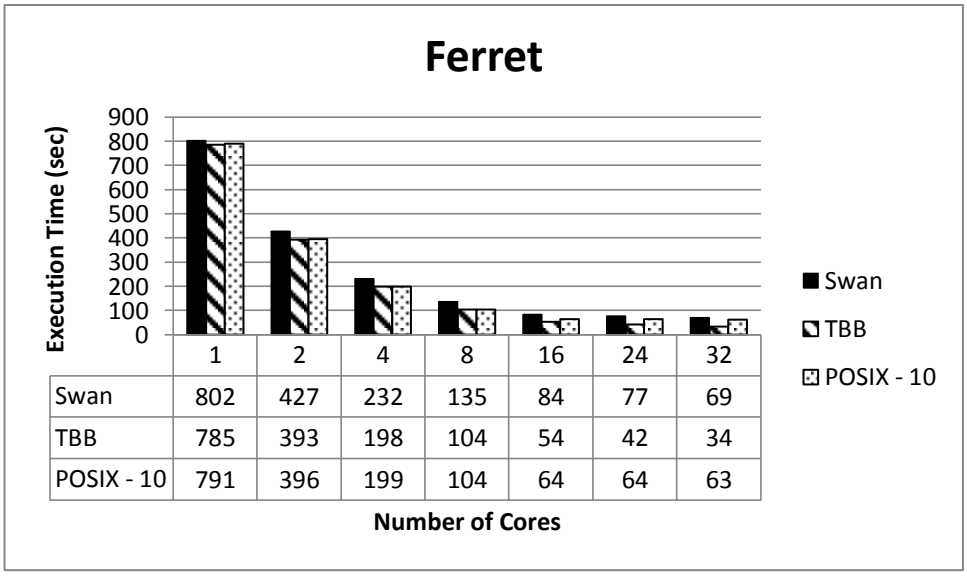


Figure 32: Ferret speedup of Swan, TBB and POSIX implementations.



	1	2	4	8	16	24	32
Swan	802	427	232	135	84	77	69
TBB	785	393	198	104	54	42	34
POSIX - 10	791	396	199	104	64	64	63

Figure 33: Execution times of Swan vs TBB vs POSIX. Specific core counts.

Num Cores	Swan (sec)	TBB (sec)	POSIX (sec)	Swan-TBB (sec)	Swan-POSIX (sec)
1	802	785	791	17	11
2	427	393	396	34	31
4	232	198	199	34	33
8	135	104	104	31	31
16	84	54	64	30	20
24	77	42	64	35	13
32	69	34	63	35	6

Table 4: Comparison of Swan, POSIX and TBB execution times in seconds. Last two columns show the relative overhead of Swan (in seconds) compared to the other models.

In a more detailed examination of the execution time between the implementations, it is observed that for core counts greater than 1 the difference between Swan and TBB in execution time is approximately 30-35 seconds, while the difference between Swan and POSIX is reducing while core count increases. Table 4 shows this comparison for each core count. POSIX implementation is less scalable than TBB and this is due to the blocking queues that is using for the data transfers between stages which prevents the parallel use of the data. Moreover, core to core communication increases as the number of cores increases while the number of software threads remains stable (42 software threads). Unlike this, TBB gains scalability and the overhead of Swan over TBB is the serialization of the `load` stage.

Figure 34 shows the load balance comparison between Swan and TBB implementations. The performance loss observed in Swan is a result of the serialization of the `load` stage. Core 1 is the most loaded one due to the fact that it is executing the first stage that loads the input data.

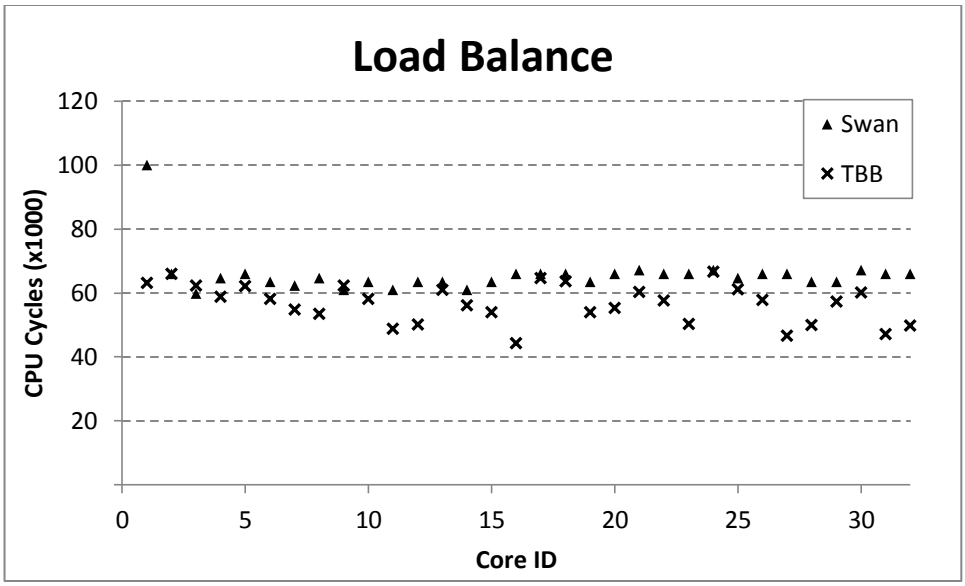
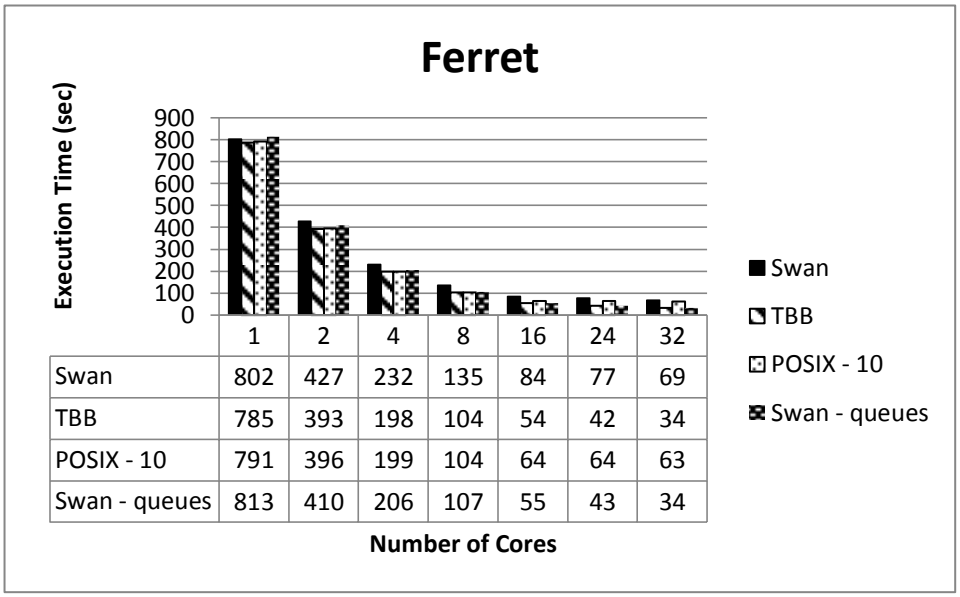


Figure 34: Load balance comparison between TBB and Swan with objects for Ferret benchmark



	1	2	4	8	16	24	32
Swan	802	427	232	135	84	77	69
TBB	785	393	198	104	54	42	34
POSIX - 10	791	396	199	104	64	64	63
Swan - queues	813	410	206	107	55	43	34

Figure 35: Ferret execution time of Swan implementations (objects only or with hyperqueues), TBB and POSIX.

We present the evaluation of ferret parallelized with Swan using hyperqueues. Figure 35 illustrates the execution times of TBB and POSIX implementations presented before in comparison with ferret Swan implementation using hyperqueues. Comparing Swan with TBB now

substantiates that the previously observed overhead was due to the serialization of first stage. The addition of the `load` stage in the pipeline could overlap approximately 34 seconds of the benchmark resulting in better performance. Swan's performance is very close to TBB while it now has overcome POSIX implementation's performance which adds multiple overheads by using blocking queues. For 1 core, the overhead of the Swan-hyperqueues is clear, but running on many cores it is entirely overlapped.

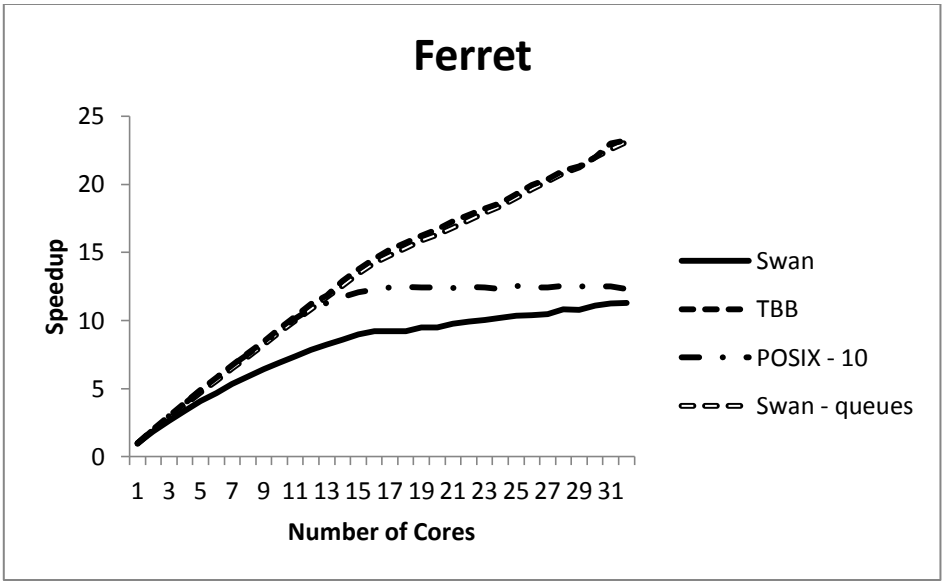


Figure 36: Speedup comparison Swan vs TBB vs POSIX vs Swan with queues

Figure 36 shows the speedup comparison between Swan, TBB and POSIX ferret versions. The implementation of ferret using POSIX threads consists of 955 lines of code (including queues for communication between stages and thread pool implementation). TBB implementation reduced the lines of code to 512, and Swan implementation to 445.

5.3 Dedup

The dedup kernel was developed by Princeton University. Dedup is a file compression algorithm that uses a combination of global compression and local compression for achieving high compression ratios in less

computation time. The type of this compression is called 'deduplication'.

Dedup consists of five pipeline stages: `Fragment` stage segments the data stream into smaller data segments; `FragmentRefine` splits the data segments into smaller data chunks; The `Deduplicate` stage looks up a hash table for each chunk and determines if the chunk is new chunk of if it is a duplicate of a previous one; if the chunk is not a duplicate, it is passed to the `Compress` stage where it is compressed. Otherwise, compression is skipped and the compressed data of the original chunk are used for the compression of the duplicate chunk. Finally, the `Output` stage reorders the data chunks into the original order, if necessary, and writes the compressed stream creating the compressed file.

Figure 37 shows the pipeline structure used in pthreads implementation of PARSEC. `FragmentRefine` stage generates work, which means that it has a single input and produces multiple outputs, which are inputs to the next stage. Moreover, `Deduplicate` stage performs stage bypassing; Compress stage is executed in certain circumstances as described above. Each one of the three intermediate stages is executed by multiple threads, similar to ferret implementation, and the first and last stages by a single thread each.

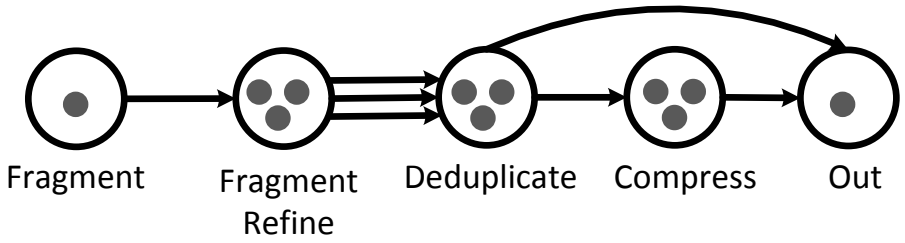


Figure 37: Dedup task-graph of pthreads implementation. Gray dots are threads per pool. Arrows show dependence/data exchange.

5.3.1 Parallelization

The task graph of Figure 37, visualizes a number of challenges in the parallelization of this benchmark. This workload's pipeline structure is not as straightforward as the structure presented in ferret. This pipeline's parallelization is challenging for Swan, but in the same time, easier than

POSIX implementation. The first challenge was the stage bypassing of `Compress` stage (see Figure 37). Second, and most important, was the `FragmentRefine` stage behavior, that produces multiple outputs from one input. Another challenge was the requirement to enforce ordering of tasks that execute the `Deduplicate` stage. In this stage, chunks are determined to be duplicates of another prior chunk or not. If a chunk that should be considered as duplicate of a prior chunk, reaches this stage first, then it is not marked as duplicate and decompression of the file fails; the reason is that `Compress` stage recognizes the original chunk as duplicate and thus cannot find the appropriate data to decompress it. We provide a number of solutions that address the aforementioned challenges.

Challenge	**Solutions**	**Appears in**
• Single input, multiple outputs	• Nested pipelines	• Nested Pipeline 3-stage, nested pipeline 2-stage
	• Remove stage Fragment Refine	• Coarse-grained 4-stage
	• Add to lists the multiple outputs	• Simple 5-stage, nested pipeline 3-stage, nested pipeline 2-stage
	• Serialize until multiple outputs are produced	• Nested pipeline 2-stage
• Stage bypassing	• Call stage and return	• All
• Deduplicate align-ment	• Additional in-outdep dependence	• All

Table 5: Challenges faced in dedup, their solutions, and the implementations each solution appears

Table 5 summarizes the above challenges and their solutions in specific implementations. The TBB implementation used in this work is the one in [3]. This version of dedup uses nested pipelines in order to solve the challenge that the `FragmentRefine` stage introduces. The outer pipeline has to make the appropriate number of calls to the nested pipeline, which is determined by the number of coarse chunks that `Fragment` stage produced. This means that for each call of `FragmentRefine`, there is a nested pipeline that calls its stages as many times as are the data items that `FragmentRefine` produced. Swan can handle this and parallelize dedup in the same way. Regarding the stage-bypassing challenge, neither Swan nor TBB can skip the execution of a pipeline stage, so both approaches are spawning the task to call this stage, and if data do not need to be compressed, stage immediately returns.

Dedup	Num iterations	Time (sec)	Time %
Fragment	336	1.85	3.5
Fragment Refine	336	3.45	6.5
Deduplicate	336×*N*	4.08	7.7
Compress	336×*N*	39.13	73.7
Output	336×*N*	4.56	8.6

Table 6: Dedup number of iterations and stages time breakdown. N depends on each of the 336 coarse-grained chunks and varies between 473 and 65537. Compress is called 168364 times in total and, Deduplicate and Output, 369950 times.

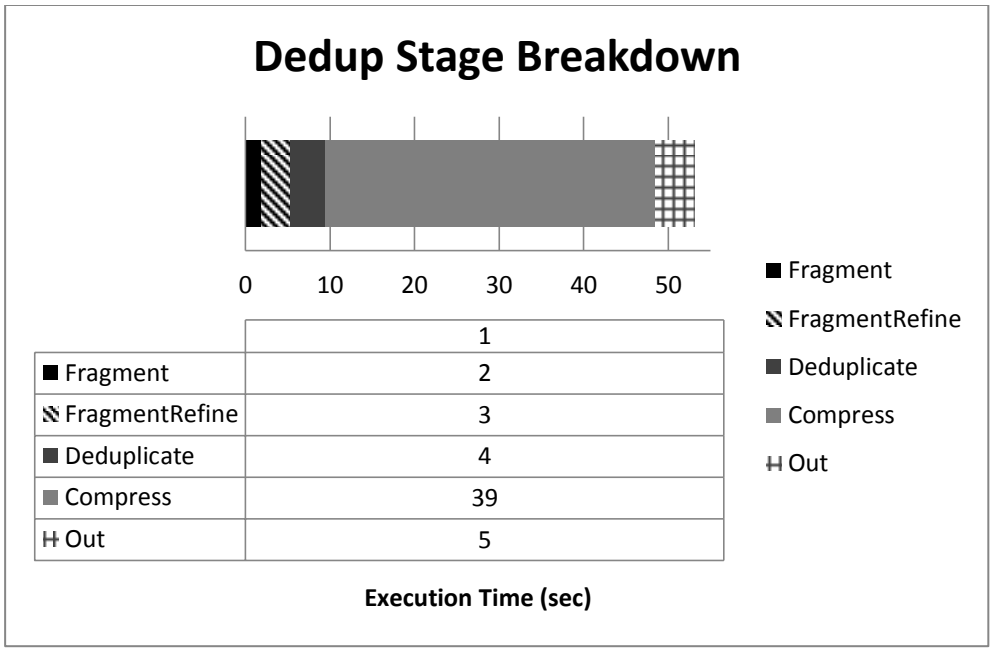


Figure 38: Dedup graph for stage breakdown in seconds

- **Dedup 5 stage pipeline with lists**

In this implementation we provide a 5-stage pipeline with the same stages as in POSIX implementation. What constraints Swan implementation is that each stage has to be spawned as many times as the data items that it has to process are. This is how objects in Swan work, and for that reason `FragmentRefine` stage that takes one input and produces multiple outputs was a significant challenge on the parallelization. The way to handle this issue is to generate a list of the data that have been produced. This list can be marked with Swan's access mode labels in order to be passed to the next stages. The task-graph on Figure 39 shows the dependencies between stages. When `FragmentRefine` has finished the production of one list, then the next stage, `Deduplicate`, starts consuming this list and reproduces it with the appropriate changes to `Compress` stage. The number of lists that are created during this procedure are equal to the number of chunks that `Fragment` stage produces. In each iteration there is one list, produced by Fragment, passed through all the stages and is consumed and/or updated.

Moreover, as listed in the challenges of this benchmark, Deduplicate stage has to be sequential, meaning that only one task of this stage is execut-

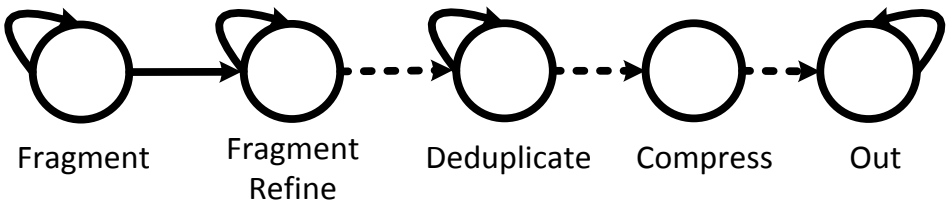


Figure 39: Task graph for Dedup simple 5-stage pipeline. Dashed arrows indicate dependency on list-objects

ing at each period of time. Swan could offer an easy and not time consuming solution to this problem by inserting a self-dependence on that stage, which serializes the calls of it.

Regarding stage bypassing, Swan is not properly applicable to irregular pipelines, so we need to spawn this stage and for each element of the list, the stage either performed compression, or skipped the element. The same solution is applied in the TBB implementation.

This solution introduced parallelization between stages for 336 iterations, (for native input) which is the number of chunks that the first stage creates. This is a coarse-grained parallelization, as long as each stage gener-

```
int Fragment (outdep<chunk_t> chunk);
void FragmentRefine(indep<chunk_t> chunk,
                    inoutdep<u32int> self_data,
                    outdep<std::list<chunk_t>> chunkslist,);
void Deduplicate(inoudep<list<chunk_t>> chunklist,
                 inoutdep<int> wait);
void Compress(inoudep<list<chunk_t>> chunklist);
void Out(inoudep<list<chunk_t>> chunklist, inoutdep<int>wait);

int main(){
  read_done = 0;
  while(!read_done){
     read_done = call(Fragment, (outdep<chunk_t>) chunk);
     spawn(FragmentRefine, (indep<chunk_t>)chunk,
                           (inoutdep<u32int>)self_data,
                           (outdep<list<chunk_t>>)chunklist);

     spawn(Deduplicate, (inoutdep<list<chunk_t>>)chunklist,
                        (inoutdep<int>)wait);

     spawn(Compress, (inoutdep<list<chunk_t>>)chunklist);
     spawn(Out, (indep<list<chunk_t>>)chunklist,
                (inoutdep<int>)wait2);
   }
}
```

Figure 40: Code of dedup simple 5 stage pipeline.

ates only 336 tasks that iterate over a list instead of as many as the number of outputs that `FragmentRefine` stage produces (fine-grained chunks). Figure 40 shows the code for this implementation; note that the entire lists are the arguments with analyzed dependencies.

- **Dedup 3-stage nested pipeline**

We tried to provide a better solution than the above to the Dedup parallelization. In [3], Reed et.al, present the implementation of dedup using nested pipelines, with the use of TBB programming model. We borrow this structure to parallelize dedup with Swan. Figure 41 illustrates the task-graphs of the two pipelines used. Outer pipeline (Figure 41a) contains the stages of `Fragment`, `FragmentRefine` and `NestedPipeline`. First, `Fragment` is called (not spawned) and `FragmentRefine` waits for `Fragment` to finish in order to get the first-level-split chunk so that it can split it more. When `FragmentRefine` divides a chunk, it creates a list for this chunk, and adds in this list the smaller chunks produced, as discussed in the simple 5-stage pipeline parallelization above. This list is the input of the `NestedPipeline` stage. In this stage, (Figure 41b) we have `Deduplicate`, `Compress` and `Out` stages which are spawned for every chunk in the list. As discussed before, `Compress` stage is executed under conditions. Swan cannot handle this due to the static definition of the dependencies. In Swan approach, `Compress` stage is spawned for all chunks and, if the condition is met, chunk gets compressed; otherwise `Compress` stage takes no action. In order to solve the `Deduplicate` stage's serialization of tasks, we insert an `inout` (self) dependence to this stage, so that it waits every time for itself to finish the execution of the previous iteration. Figure 42 shows the pseudo code for this implementation, omitting the dependencies described in the task-graph.

This implementation can be characterized by its fine-grained, tasks opposite to the previous one which was coarse-grained. It is consisted of 336 iterations spawning three stages and the one of these stages generates tasks inside a loop that executes for a very large number of iterations (be-

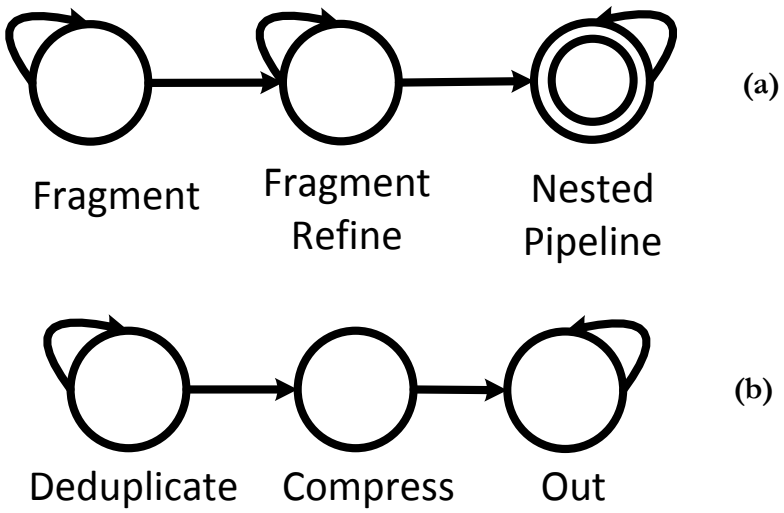


Figure 41: (a): Outer dedup pipeline: same number of iterations as the number of chunks after Fragment (336 iterations). **(b):** Nested pipeline: number of iterations equals to the number of chunks that FragmentRefine produces (473 – 65537 iterations).

tween 473 and 65537 iterations). This is the most fine-grained solution that we develop in Dedup.

```
int Fragment (outdep<chunk_t> chunk);
void FragmentRefine(indep<chunk_t> chunk,
                    outdep<list<chunk_t>> chunkslist);
void NestedPipline(indep<list<chunk_t>chunkslist,
                   inoutdep<int>align);
void Deduplicate(inoutdep<chunk_t> chunk,
                 inoutdep<int> wait);
void Compress(inoutdep<chunk_t> chunk);
void Out(indep<chunk_t> chunk,
         inoutdep<int> wait);

void NestedPipeline(indep<list<chunk_t>chunkslist,
                    inoutdep<int>align)
{

   iterator iter = chunklist.iterator();
   while(!chunklist.empty()) {

      object_t<chunk_t> chunk = iter.next();
      object_t<int> wait;

      spawn(Deduplicate, (inoutdep<chunk_t>)chunk,
                         (inoutdep<int>)wait );
      spawn(Compress, (inoutdep<chunk_t>)chunk);
      spawn(Out, (inoutdep<chunk_t>)chunk,
                 (inoutdep<int>)wait);
   }
}
ssync();
int main() {

  read_done = 0;
  object_t<int> wait;
  while(!read_done) {

     object_t<chunk_t> chunk;
     read_done = call(Fragment, (outdep<chunk_t>)chunk);

     spawn(FragmentRefine, (indep<chunk_t>)chunk,
                           (outdep<list<chunk_t>>)chunkslist);
     spawn(NestedPipeline, (indep<list<chunk_t>>)chunkslist,
                           (inoutdep<int>)wait);
   }
   ssync();
}
```

Figure 42: Code for dedup nested pipeline implementation.

- **Dedup coarse-grained 4-stage pipeline**

FragmentRefine stage exists in order to create smaller chunks for more fine grained parallelism. This results to finer granularity of tasks after the execution of FragmentRefine. Considering all the effort needed to solve the problem with the work generation of FragmentRefine, we thought that skipping this stage would, give better results. Compression remains the same as long as the same data are compressed. This imple-

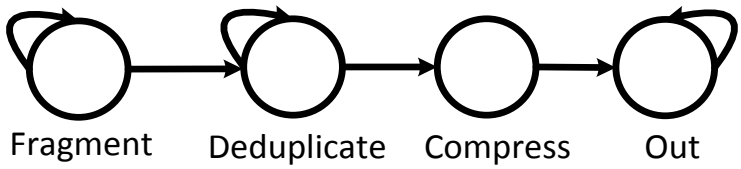


Figure 43: Task-graph of dedup 4-stage coarse-grained implementation.

mentation's task-graph is shown in Figure 43. First, `Fragment` is called and produces the first-level split chunks, which are then passed to `Deduplicate`, in-order, later in `Compress`, where if the condition is met, data get compressed and finally are written in the output file. This implementation can eliminate parallelization overheads from the reduced task creation but can have drawbacks regarding the load balance, as long as synchronized threads should wait longer for a larger task to finish. This implementation has the same task granularity as the one with the simple 5-stage pipeline. The difference is that it is skipping the computations of the `FragmentRefine` stage, as long as they don't introduce more parallelism in the 5-stage pipeline implementation. This results to a 4-stage pipeline iterating for 336 times for native input.

```
int Fragment (outdep<chunk_t> chunk);
void Deduplicate(inoutdep<chunk_t> chunk, inoutdep<int> wait);
void Compress(inoutdep<chunk_t> chunk);
void Out(indep<chunk_t> chunk, inoutdep<int>wait);

int main(){
  read_done = 0;
  object_t<int> wait1;
  object_t<int> wait2;
  while(!read_done){
     object_t<chunk_t> chunk;
     read_done = call(Fragment, (outdep<chunk_t>)chunk);
     spawn(Deduplicate, (inoutdep<chunk_t>)chunk,
                        (inoutdep<int>) wait1);
     spawn(Compress, (inoutdep<chunk_t>)chunk);
     spawn(Out, (indep<chunk_t>)chunk,
                (inoutdep<int>)wait2);
   }
   ssync();
}
```

Figure 44: Code for dedup 4-stage coarse-grained implementation.

- **Dedup 2-stage nested pipeline**

Table 6 and Figure 38 show the time breakdown of the dedup pipeline stages; it is observed that the most time-consuming stages are `Compress` and `Out` stages. For that reason it would be beneficial if the light-weight stages (`Fragment`, `FragmentRefine`, `Deduplicate`) were merged into one stage and gain parallelism from the two heavy-weight stages (`Compress`, `Out`). This was the motivation for the 2-stage nested pipeline implementation, whose task-graph is shown in Figure 45.

This implementation introduces fine-grained parallelism, but in less amount than the one of the 3-stage nested pipeline. Task creation is reduced by merging three of Dedup's stages into one. Figure 46 shows how this solution can be implemented with Swan.

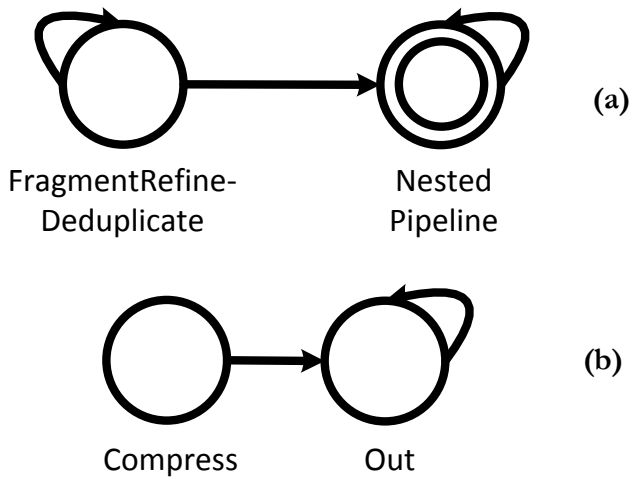


Figure 45: 2-stage nested Dedup pipeline. (a): Outer pipeline, first stage includes Fragment, FragmentRefine, and Deduplicate stages merged in one. Second is the nested pipeline shown in b. (b): Nested 2-stage pipeline with Compress and Out stages.

```
void FragmentRefine_Deduplicate(outdep<list<chunk_t>> chunkslist);
void NestedPipline(indep<list<chunk_t*>chunkslist, inoutdep<int>align);
void Compress(inoutdep<chunk_t> chunk);
void Out(indep<chunk_t> chunk, inoutdep<int> wait);


void FragmentRefine_Deduplicate(outdep<list<chunk_t>> chunkslist) {
    call(Fragment);
    call(FragmentRefine, chunkslist);
    call(Deduplicate, chunkslist);
}
void NestedPipeline(indep<list<chunk_t*>chunkslist, inoutdep<int>align) {

   iterator iter = chunklist.iterator();
   while(!chunklist.empty()) {

      object_t<chunk_t> chunk = iter.next();
      object_t<int> wait;

      spawn(Compress, (inoutdep<chunk_t>)chunk);
      spawn(Out, (inoutdep<chunk_t>)chunk,
                 (inoutdep<int>)wait);
   }
   ssync();
}
int main() {
  read_done = 0;
  while(!read_done) {
     read_done = call(FragmentRefine_Deduplicate,
                      (outdep<list<chunk_t>>)chunkslist);
     spawn(NestedPipeline, (indep<list<chunk_t*>)chunkslist,
                           (inoutdep<int>)align);
   }
   ssync();
}
```

Figure 46: Code for Dedup 2-stage nested pipeline implementation

5.3.2 Evaluation

As in ferret, dedup's POSIX implementation can be configured with various multitudes of threads per pool. The total number of software threads running is indicated by the equation:

$$software\_threads = threads\_per\_pool \times 3 + 2$$

The first and last stages are executed by one thread each and the remaining three middle stages by *threads_per_pool* threads. Figure 47 demonstrates the execution time for POSIX dedup implementation, with different amounts of threads per pool showing the best option that is 8 threads per pool. Using 8 threads per pool, the total number of software threads is, according to the above equation, 28. This is the best configuration for dedup running with POSIX implementation because for more software threads

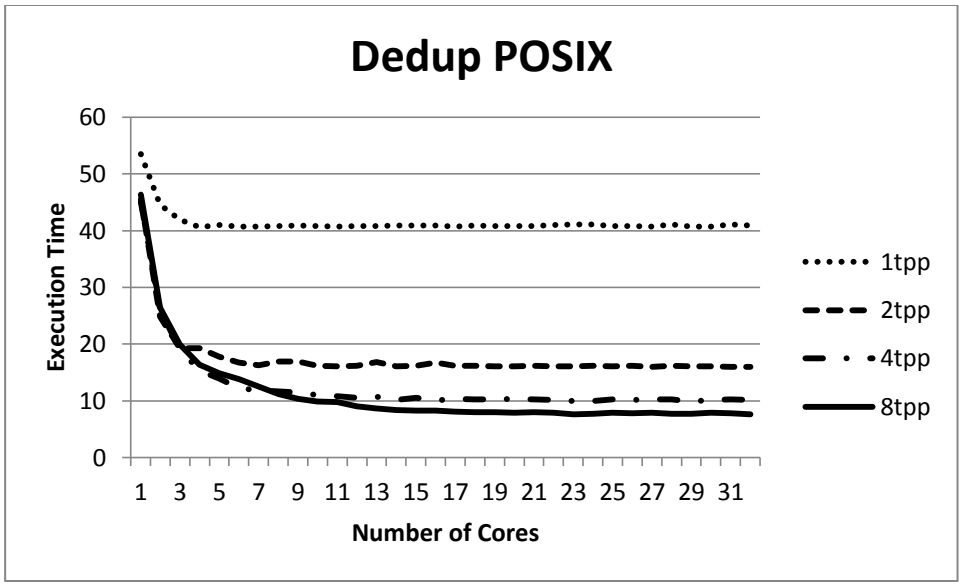


Figure 47: Dedup POSIX execution time with multiple number of threads per pool.

I/O bottleneck is reached on the output stage. We compare the rest implementations with the POSIX using 8 threads per pool.

Figure 48 presents the execution times for the explained Swan implementations. Using the coarse grained implementation of dedup delivers the best performance. In that implementation, `FragmentRefine` stage was skipped, in order to have coarse grained tasks and also skip the procedure of the nested pipelines that increases task creation as well as data dependencies. Coarse grained implementation is beneficial for Swan, as long as there are fewer dependencies to be analyzed, fewer tasks, and contains the execution of a simple 4-stage pipeline. The next best performance from those implementations is the one with the simple 5-stage pipeline using lists. As explained before, this implementation has the same dependencies as the coarse-grained one. The difference is that instead of having a coarse chunk to process and compress, dedup has a list of split chunks that all of them together form one coarse chunk. The overhead of this implementation, compared to the coarse-grained one, results from the list creation and the iterations on the list.

The remaining two implementations consist of nested pipelines.

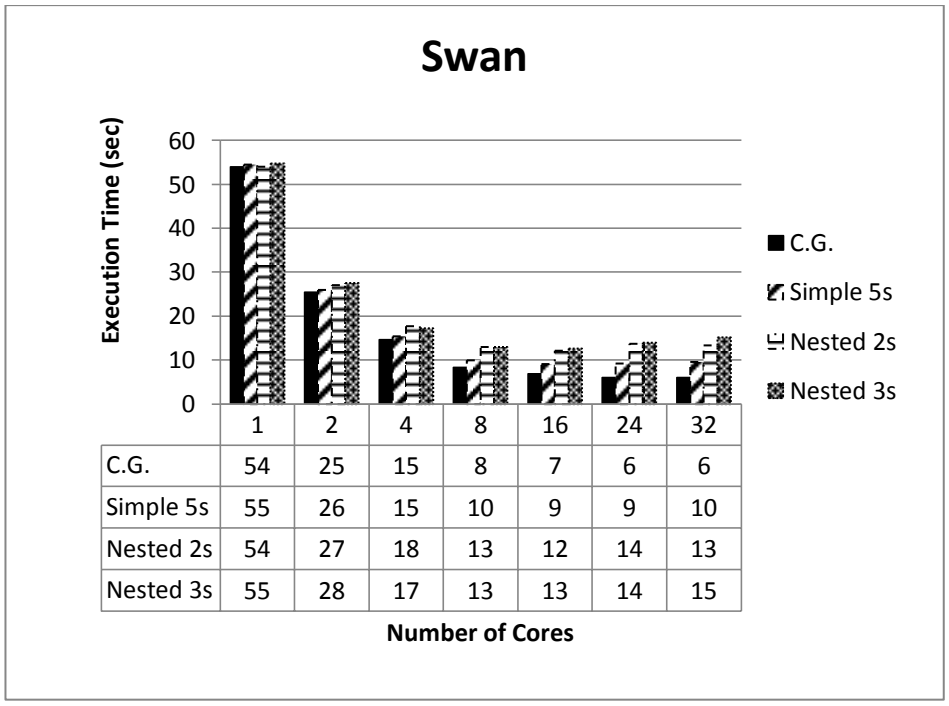
This mechanism creates finer-grained tasks, while the outer pipeline is performing 336 iterations and for each of these iterations another pipeline is executing which performs a number of iterations between 473 and 65537. As a result the total number of tasks for the 3-stage nested pipeline implementation, if N is the number of the nested pipeline's iterations, would be:

$$336 \times (outer\_pipeline\_num\_stages + inner\_pipeline\_num\_stages \times N) = 336 \times (3 + 3 \times N)$$

For the 2-stage nested pipeline implementation the number of tasks would accordingly be:

$$336 \times (outer\_pipeline\_num\_stages + inner\_pipeline\_num\_stages \times N) = 336 \times (2 + 2 \times N)$$

This is because in the second implementation we have 2 stages in the outer pipeline, as well as two stages in the inner pipeline.



	1	2	4	8	16	24	32
C.G.	54	25	15	8	7	6	6
Simple 5s	55	26	15	10	9	9	10
Nested 2s	54	27	18	13	12	14	13
Nested 3s	55	28	17	13	13	14	15

Figure 48: Execution time of Swan implementations of Dedup. C.G. is the coarse grained implementation.

These computations imply that the implementation of the nested pipeline that contains two stages instead of three has less task creation so the time consumption for dependency tracking is less.

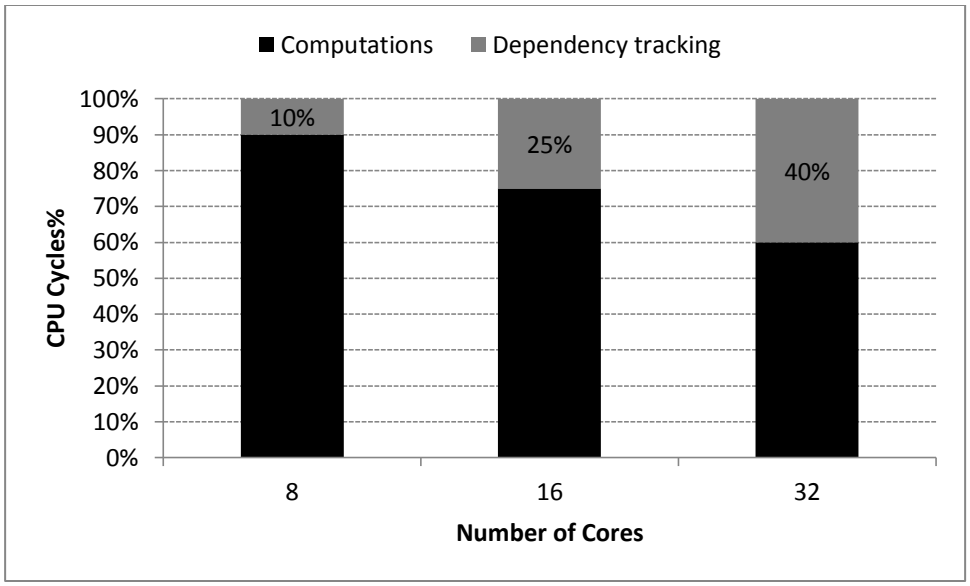
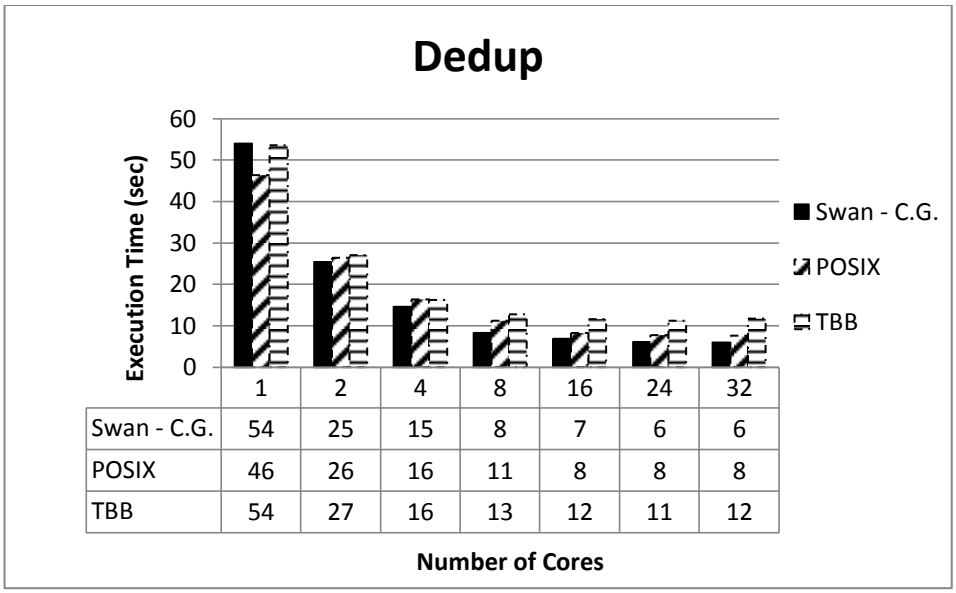


Figure 49: Dependency tracking CPU cycles vs computation CPU cycles breakdown

Figure 49 shows the computation-dependency tracking CPU cycles breakdown for 8, 16 and 32 cores, measured for the dedup 2-stage nested pipeline implementation (approximately same behavior as the 3-stage pipeline implementation). It is observed that as the core count increases the time spent in dependency tracking increases. This explains the saturation in performance. In contrast, for the coarse grained implementation, saturation point is reached in higher number of cores as long as the elimination of dependencies improves the efficiency of the parallelization.

Figure 50 shows the execution time of the best approaches of Swan and POSIX, and the TBB version of dedup. TBB lacks performance because of the nested pipeline that is using, which adds more overhead. POSIX pipeline is a simple 5-stage pipeline, which creates fine grained chunks of data but even this implementation cannot run efficiently with more than 28 concurrent threads on 32 cores. TBB implementation is the one with the highest number of tasks, and as we can see this can also be a bottleneck comparing to the coarse grained implementation.

Figure 51 shows the speedup occurred from the above implementations. Swan coarse grained implementation is very sensitive to



	1	2	4	8	16	24	32
Swan - C.G.	54	25	15	8	7	6	6
POSIX	46	26	16	11	8	8	8
TBB	54	27	16	13	12	11	12

Figure 50: Comparison of execution time for the best approaches of POSIX and Swan, and TBB implementation

load balance as long as the tasks are coarse grained resulting longer time for synchronization or wait conditions between threads. Nevertheless, tasks have been reduced so Swan can obtain better performance. Moreover performance saturation reaches higher number of cores because of the less dependency tracking overhead of this implementation.

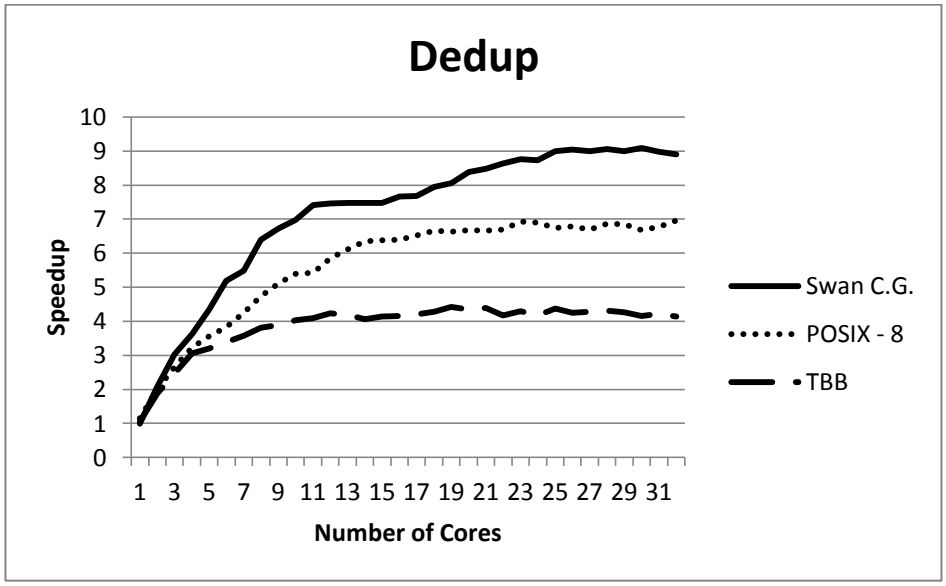


Figure 51: Speedup comparison of Swan coarse grained POSIX and TBB versions.

Swan coarse grained implementation achieves the best performance compared to the rest versions but the difference between the implementations is important. Skipping one stage of the benchmark can lead to an unfair comparison as long as tasks are of different granularity and computations are reduced by one pipeline stage. Still this comparison can lead to the conclusion that dedup performs better without FragmentRefine stage for the native PARSEC input. This behavior may be input dependent or the splitting of the chunks does not help parallelism as long as it introduces large synchronization overheads from the runtime.

Figure 52 shows the speedup occurred from POSIX, TBB and Swan 2-stage nested pipeline implementations. This implementation of Swan has been chosen for comparison because it is similar to the TBB implementation and through this graph TBB and Swan can be compared fairly. Swan performs as well as TBB, while both saturate for larger than 16 core counts. This is due to the fact of the additional overhead of both runtimes for dependency tracking as has been illustrated previously. POSIX implementation also saturates for higher core counts but performs better as long as the irregular pipeline can be handled through queues in a more flexible way. `Compress` stage is called only for the chunks that need to be

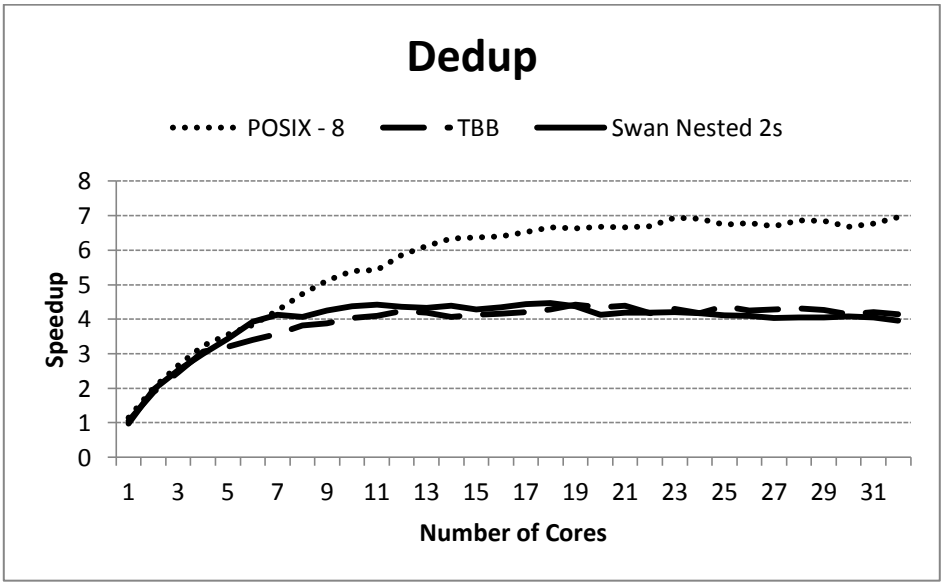


Figure 52: Speedup comparison between implementations of same granularity

compressed despite of TBB and Swan where the program cannot have a conditional pipeline stage. Moreover, there are less synchronization overheads due to the fact that in POSIX implementation a nested pipeline is not used as `FragmentRefine` producer stage can be handled with queues.

Dedup implementation using POSIX threads is written in 2032 lines of code (including the queue and tree implementations for communication and reordering of chunks). TBB version has a reduced number of lines of code to 1264 lines and Swan implementations need approximately 900 lines of code for dedup.

Chapter 6

Conclusions

6.1 Summary

In this thesis we studied how task-dataflow models exploit parallelism in pipelined applications. We showed that fine-grained parallelism is not always the best option; however it is very important for load balancing. In addition, we were confronted to many parallelization overheads such as dependence analysis overhead when the program has many dependent tasks and we tuned the performance through different implementations and parallelization strategies. Through this study we concluded that state of the art task dataflow models do not provide the appropriate flexibility for parallelizing irregular pipelined workloads. We addressed the limitations in programming flexibility and we implemented hyperqueues, a programming abstraction of queues.

The hyperqueue programming abstraction overcomes the limitations of the state of the art task dataflow programming models. Such limitations are the presence of a recursive pipeline stage or the presence of a conditional pipeline stage. A pipelined application can be implemented with the simple interface that hyperqueues provide which assures concurrency and prevents common errors that programming with pthreads can cause. Furthermore, hyperqueues indicate potential optimization as long as fewer tasks are postponed by the use of queues. Hyperqueues can be used as the only dependency tracking mechanism in a pipelined application, or better, with the coordination of special object data structure. We presented the results of this effort in ferret, a benchmark with irregular pipeline parallelism.

6.2 Future Work

Hyperqueues help the programmer overcome many limitations that current programming models have in the construction of irregular pipeline parallel applications. One limitation of hyperqueues is that they are not applicable through nested parallelism. Although hyperqueues are implemented for working in plurality, the task that uses them is spawned once and inside its body it cannot fetch new tasks that use a mutual hyperqueue. This results to reduced parallelism and load imbalance; as so, hyperqueues are better used in combination with objects. Hyperqueues abstraction aims to be extended in order to be applicable with nested parallelism.

Moreover, hyperqueues postpone the execution of new consumer tasks which leads to serialization of the consumers of the same queue. We aim to revise this and extend the semantics of the hyperqueue to allow the concurrent execution of consumers of the same queue without harming the correctness of consumption order.

Bibliography

[1] Christian Bienia and Kai Li. "Characteristics of Workloads Using the Pipeline Programming Model", in Proc of the 3rd Workshop on Emerging Applications and Many-core Architecture, June 2010.

[2] Navarro, R. Asenjo, S. Tabik, and C. Cascaval. Analytical Modeling of Pipeline Parallelism. In PACT '09.

[3] Eric C. Reed, Nicholas Chen, Ralph E. Johnson. Expressing Pipeline Parallelism Using TBB Constructs.

[4] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In PACT '10.

[5] S. Rul, H. Vandierendonck, and K. De Bosschere. A profile-based tool for finding pipeline parallelism in sequential programs. Parallel Comput., 36:531–551, September 2010.

[6] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. In MICRO '07.

[7] Hans Vandierendonck, George Tzenakis and Dimitrios S. Nikolopoulos. A Unified scheduler for Recursive and Task Dataflow Parallelism, in the proceedings of the Conference on Parallel Architectures and Compilation Techniques, 2011, pp. 1-11.

[8] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, "Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures", in SPAA '07, 2007, pp. 116-125.

[9] M. Frigo, P. Halpern, C. E. Leiserson and S. Lewin-Berlin, "Reducers and other Cilk++ hyperobjects", in SPAA'09, 2009, pp. 79-90.

[10] P. Pratikakis, H. Vandierendonck, S. Lyberis, and D. S. Nikolopoulos, "A programming model for deterministic task parallelism", in Proc. Of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness, 2011, pp. 7-12.

[11] "StarPU: a Runtime System of Scheduling Tasks over Accelerator-Based Mul-

ticore Machines", INRIA, Research Report RR-7240, 03 2010. [Online]. Available: http://hal.inria.fr/docs/00/46/76/77/PDF/RR-7240.pdf

[12] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system", in PPoPP'95, pp. 207-216.

[13] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multi-threaded language", in PLDI'98, 1998, pp. 212-223.

[14] "Intel Threading Building Blocks (Intel TBB)" Available: http://software.intel.com/en-us/intel-tbb

[15] OpenCL Specification, 1sted., Khronos OpenCL Working Group, Sep. 2010.

[16] J. M. Perez, R. M. Badia, and J. Labarta, "A dependency aware task-based programming environment for multicore architectures", in CLUSTER'08, Sep. 2008, pp. 142-151.

[17] G. Contreras, M. Martonosi "Characterizing and Improving the Performance of Intel Threading Building Blocks", in International Symposium on Workload Characterization, September 2008

[18] W. Thies and S. Amarasinghe. "An empricial characterization of stream programs and its implications for language and compiler design", in PACT '10.

[19] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. "Parallel-stage decoupled software pipelining", in CGO '08: 6th Intl Symp on Code generation and optimization, 2008.

[20] C. Bienia "Benchmarking Modern Multiprocessors" A dissertation presented to the Faculty of Princeton University in candidacy for the degree of Doctor of Philosophy, January 2011