RC24732 (W0901-066) January 20, 2009
Computer Science

IBM Research Report

Load Balancing Using Work-stealing for Pipeline Parallelism in Emerging Applications

Angeles Navarro, Rafael Asenjo, Siham Tabik
Universidd de Málaga

Calin Cascaval
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Load balancing using Work-stealing for Pipeline Parallelism in Emerging Applications

Angeles Navarro[a] Rafael Asenjo[a] Siham Tabik[a] Călin Caşcaval[b]

[a]Universidad de Málaga [b]IBM Research

January 19, 2009

Abstract

Parallel programming is a requirement in the multi-core era. One of the most promising techniques to make parallel programming available for the general users is the use of parallel programming patterns. Functional pipeline parallelism is a pattern that is well suited for many emerging applications, such as streaming and "Recognition, Mining and Synthesis" (RMS) workloads. In this paper we develop an analytical model for pipeline parallelism based on queueing theory. The model is useful to both characterize the performance and efficiency of existing implementations and to guide the design of new pipeline algorithms.

We demonstrate how we used the model to characterize and optimize two of the PARSEC benchmarks, `ferret` and `dedup`. We identified two issues with these codes: load imbalance and I/O bottlenecks. We propose to address the I/O bottleneck using parallel I/O. We addressed load imbalance using two techniques: i) parallel pipeline stage collapsing; and ii) work-stealing. We implemented these optimizations using Pthreads and the Threading Building Blocks (TBB) libraries. We compare the performance of different alternatives and we note that the TBB implementation outperforms all other variants. However, the current pipeline template supported by TBB is restrictive and we discuss extensions that will allow more flexible pipeline designs using this library.

1 Introduction

As multi-core processors are becoming ubiquitous, parallel programming is seen as the technology that can make multi-cores succeed or not. There are two main directions that research has focused on: (i) language development, whether new parallel languages, such as X10 and Chapel, or parallel extensions to existing languages, such as UPC, Co-Array Fortran, Titanium; (ii) parallel libraries – developing highly optimized libraries that encapsulate data parallelism [9] or task parallelism [19]. Both these approaches aim at freeing the programmers from thinking about the complexities of the architecture by providing higher levels of abstractions which allow them to express the parallelism in the application and rely on the system to efficiently exploit parallelism.

To express concurrency, there are two main paradigms that can be used separately or in combination: data and task parallelism. With data parallelism, programmers define parallel data structures and essentially write a sequential program using operations on the parallel data structures. The operations themselves encapsulate the parallelism. Both language extensions and libraries can express data parallelism. For task parallelism, programmers mainly partition the work into tasks that are scheduled to execute in parallel. This can be expressed as parallel loops, parallel iterators, or constructing a task dependence graph and passing this graph to a runtime. For both data and task parallelism, the programmer must take care of the synchronization on data accesses, such that dependences between tasks are respected.

In this paper we shall focus on one type of task parallelism, pipeline parallelism. In this model, the application is partitioned in a sequence of *filters* or *stages*. The filters are code regions that may exhibit other kinds of parallelism (data- and/or task-parallelism). A number of workloads, including two benchmarks

from the PARSEC benchmarks suite [2], `ferret` and `dedup`, exhibit pipeline parallelism. Both are streaming applications: `ferret` implements image similarity searches and `dedup` compresses a data stream by using "de-duplication".

We consider pipeline parallelism as an emerging programming pattern, and we are interested in providing models, tools, and guidance to the programmers using this pattern with respect to the scalability and performance of codes using this pattern. Applications parallelized using the pipeline model are very sensitive to load balancing: for best efficiency, all the pipeline stages must be kept busy concurrently. This implies that the programmer must either partition the work into well balanced work items that flow through the pipeline or use a system that is able to dynamically allocate resources (without too much overhead) to the busiest pipeline stages. Another typical bottleneck encountered when using pipeline parallelism is the I/O, encountered usually at the ends of the pipeline. In our examples, `ferret` is bounded by the input I/O, while `dedup` is bounded by the output stage which needs to serialize the compressed output into a single file. Optimizing these stages typically require algorithmic knowledge, while load-balancing can be addressed using automatic methods, such as work stealing [4].

To summarize, this paper makes the following contributions:

- A detailed characterization of two benchmarks in the PARSEC suite that use the pipeline parallelism model. In our study, we identify the main issues that limit the scalability of the codes and propose some solutions (Section 2);
- New analytical models for pipeline parallelism based on queueing theory. The goal of the models is to help us better understand the resources interplay for each case (Section 3);
- A performance comparison of different implementations of these benchmarks, spanning the original versions, a collapsed pipeline version and a version using work stealing, both using Pthreads and TBB. We demonstrate that our analytical model can capture precisely the behavior of all these variants (Section 4);
- A rationale of the advantages of using task stealing and the Intel Threading Building Blocks (TBB) library in order to more productively implement pipeline parallel codes, but we also identify some aspects that can be improved in the TBB implementation (Section 6).

2 Background and Motivation

A number of emerging workloads, such as streaming and RMS applications employ pipeline parallelism as the main programming pattern. Pipeline parallelism, a form of task-parallelism, is a natural model for streaming applications, because they can easily be decomposed into stages. For example, in the PARSEC Benchmark suite [2], two of the applications, `ferret` and `dedup`, are implemented using the pipeline parallelism pattern. This pattern has several advantages: (i) parallelism can be exploited at multiple levels – within stages using either data parallelism or multiple worker threads, across stages using separate threads per stage, or by replicating the pipeline and using one separate threads for each pipeline. This allows the programmer to tolerate different dependence patterns; (ii) communication is deterministic, following a producer-consumer pattern between pipeline stages.

To better understand the usage of this programming pattern, we study the behavior of the two PARSEC codes, `ferret` and `dedup`. The base implementation uses Pthreads. We compiled and executed the codes on a HP9000 Superdome SMP with 64 dual-core Itanium2, running at 1.6GHz, with 380GB of main memory. Each core has the following cache hierarchy: 16KB L1I + 16KB L1D, 1MB L2I + 256KB L2D and an unified 9MB L3. The I/O consists of 14 SCSI buses to a 40TB RAID 5. The operating system is Linux SLES 10 SP2 2.6.16 kernel. We used both gcc 4.1.2 and Intel icc 10.1.

2.1 Similarity search: ferret

The Ferret toolkit is used for content-based search of audio, images, video, 3D shapes and genomic microarray data. The `ferret` application included in the PARSEC suite is an instantiation of the toolkit configured for image similarity search [2]. As shown in Figure 1, `ferret` is decomposed into six pipeline stages. The first and the last ones are stages for input and output. These are serial stages, that read a set of images to be analyzed against a database of images, and output the list of names of similar images, respectively. The four middle stages are parallel and configured with a c-size thread pool. Object data is passed between stages using software queues, configured to hold 20 entries (default $Queue\_size$). We ran our experiments using the largest input set (`native`). For this input set, the sequential version runs 760s when compiled with gcc and 437s when compiled with icc (both compilers using the same set of flags). For the reminder of the results, we used the Intel compiler.

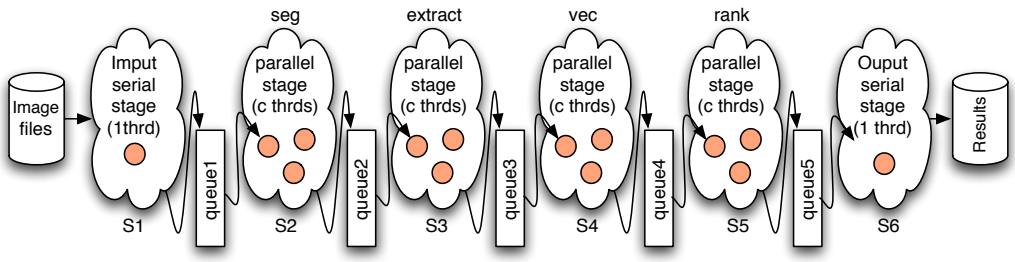
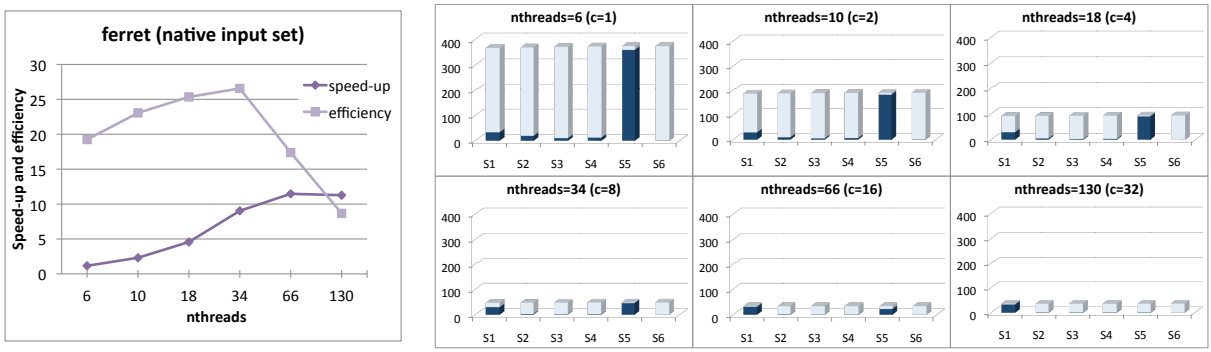


Figure 1: `ferret` pipeline configuration. Input and output stages are serial, but the middle stages are parallel, each one with c threads (3 in this example)

Figure 2(a) shows the speed-up and efficiency when using between 1 and 32 threads per stage. The maximum speed-up is 12.5 with a 27% efficiency. Analyzing the application, we found there are no data dependences between the items to be processed (the images to be checked for similarity), all the dependences are between stages. Therefore, the low efficiency pointed to an implementation bottleneck. In Figure 2(b) we show the breakdown of useful work and idle time for each stage. The figure clearly shows that the fifth stage (`rank`) is the bottleneck. Furthermore, as the number of threads per stage increases the parallel stages scale, until for c between 8 and 16, the serial input stage becomes the bottleneck.



(a) (b)

Figure 2: `ferret` original code: (a) Speedup and efficiency; (b) Breakdown of times (in seconds): for each stage, blue represents useful work whereas light-gray accounts for the waiting times of threads.

2.2 Data compression: dedup

The `dedup` kernel implements data stream compression with a high rate by combining global and local compression. The method implemented, called "de-duplication", uses five pipelines stages. Again, the first and last stages are sequential. The intermediate stages are parallel. The `DataProcess` stage generates independent coarse-grain data chunks which are enqueued for processing. The `FindAllAnchors` stage, partitions each coarse-grain data chunk into fine-grained blocks. The `ChunkProcess` stage computes a hash for each of the sub-blocks using a SHA1 checksum and then it checks for duplicates in a global hash table. If the block has not been seen it is sent to the `Compress` stage; otherwise it is classified as duplicate (already compressed) and sent to the output stage. The structure of the application is illustrated in Figure 3.

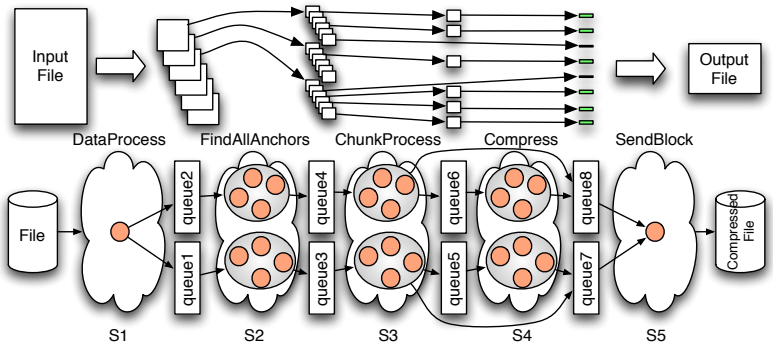


Figure 3: `dedup` pipeline configuration with $c = 8$ threads per parallel pipeline stage.

The largest inputs set available in PARSEC for the `dedup` benchmark, `native`, is an ISO file of 672MB. We choose not to preload the input file, to achieve the real streaming. The sequential code runs for 91.8s when compiled with gcc and 89.37s when compiled with icc.

The two main differences in the `dedup` pipeline compared to `ferret`, are: i) `dedup` has one stage that generates more output items than there were input (`FindAllAnchors`); and ii) some items may skip a pipeline stage (`Compress`). The implementation also limits the number of threads that share a queue in order to avoid contention, but adds additional queues between stages. Also, as opposed to `ferret`, the accesses to the shared hash table in `dedup` create dependences between work items, which are enforced by locks.

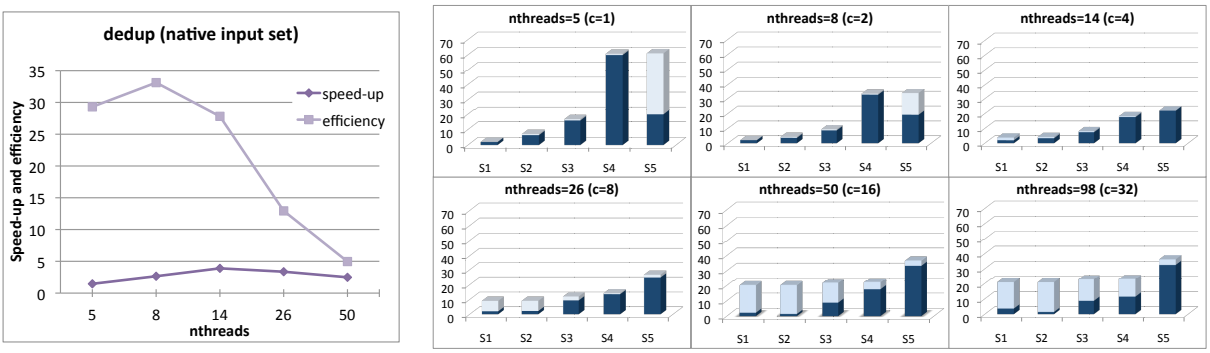
Figure 4(a) shows the speed-up and efficiency for different numbers of threads. Again, the speed-up and efficiency are low due to a high load imbalance between the stages, and the I/O bottleneck is reached on the output stage for c between 4 and 8.

2.3 Scalability Issues in Pipeline Parallelism

The scalability of both applications presented above is gated by load imbalance between stages for small number of threads and by the I/O stages for larger number of threads. We discuss solutions to address the load imbalance in this section, while I/O is discussed in Section 4.2.

Load imbalance is a result of the different amounts of work per item in each stage. If a programmer fixes the number of threads per stage arbitrarily (and it is common to use a unique number of threads for all stages), the stage with larger amounts of work per item becomes a bottleneck. To address load imbalance, we explore two orthogonal approaches: 1) collapsing all the parallel stages into one; 2) using dynamic scheduling to sharing the load among the different stages. Collapsing pipeline stages is applicable only if all the intermediate stages are parallel. Both `ferret` and `dedup` satisfy this condition, but this solution may not be generally applicable. Dynamic scheduling is a more general solution.

There are two ways in which dynamic scheduling can be implemented:



(a) (b)

Figure 4: `dedup` original code: (a) Speedups and efficiencies; (b) Breakdown of times (in seconds): for each stage, blue represents useful work whereas light-gray accounts for the waiting time.

- **Oversubscription** – the application creates more threads than available processors and relies on the OS scheduling policies to keep all the processors busy. In our example, setting the number of threads c to the number of available cores creates 3-4 times more threads. In this case, the number of threads per stage is still fixed (statically) and there are different overheads introduced by time slicing: context switching, cache pollution, and lock preemption [19]. Section 4.1 discusses this approach.
- **Work stealing** – achieves load balancing by keeping one active thread per core and assigning several work units to each thread [3]. When a thread completes its assigned work, it queries the other threads for additional work.

The Thread Building Blocks (TBB [19]) provides support for work stealing [6] in its parallel pipeline template. We implemented `ferret` and `dedup` using TBB to quantify the performance of the work stealing parallel pipeline, and we introduce here some of the abstractions used in our implementation. In TBB, programmers express concurrency in terms of *tasks*. Tasks are lighter weight than threads, thus allowing fast work switching. The TBB pipeline template is illustrated in Fig. 5(a). The C++ classes `pipeline` and `filter` are the main constructors, that represent the pipeline container and its stages, respectively. In our example, lines 7(a)-12(a) create and append two stages in a `pipeline` object. The programer has the option of specifying whether a stage is serial or parallel.

A token in TBB represents an item that traverses all pipeline stages. A new token is created for each input item. The parameter *ntokens* to the method `pipeline.run` (line 15(a)) specifies the maximum number of tokens in flight, and it is used to throttle the arrival rate of items in the pipeline in order to manage resource contention.

As mentioned before, the pipeline implementation of TBB provides a dynamic scheduler that stores and distribute available parallelism in order to improve performance. This dynamic management is performed automatically, as follows: at initialization time (line 2(a)), a set of slave worker threads are created and each slave is associated with a software task queue. For each input token, a new task is spawned and enqueued into the queue of its corresponding worker thread. Task dequeuing is implicit and carried out by the runtime system. The main scheduling loop (`wait_for_all`) provides three options for a thread to obtain work [6]: explicit task passing, local task dequeue and random task stealing. When a task is obtained, its `execute` method is invoked. Fig. 5(b) shows the kernel of that method for the pipeline template. Each pipeline task starts spawning a sibling task if the number of in-flight tokens is less than *ntokens* (`input_tokens` > 0). Then the task executes the current filter function and check for the next filter in the pipeline filter list.

To preserve locality, the TBB designers rely on continuations: while a task has not reached the last pipeline stage, instead of terminating, it is actually recycled (line 11(b)) by returning its pointer to the TBB scheduler (line 12(b)). This effectively bypasses the scheduler, and allows the old task to execute a new stage.

```
1  // Start task scheduler
2  tbb::task_scheduler_init init (NTHREAD);
3
4  // Create the pipeline
5  tbb::pipeline pipeline;
6  // Create input stage
7  MyInputFilter input_filter(input_file);
8  pipeline.add_filter(input_filter);
9  // Create transform stage
10 MyTransformFilter transform_filter;
11 pipeline.add_filter(transform_filter);
12 ....
13
14 // Run the pipeline
15 pipeline.run(MyInputFilter::ntokens);
16
17 // Remove filters from pipeline
18 pipeline.clear;
```

(a)

```
1  task * stage_task::execute
2   if (first_entry && --input_tokens>0){
3    spawn_sibling(stage_task); ...}
4   //executes the stage
5   item=my_filter.function(/*previous*/item);
6   my_filter=my_filter->
        next_filter_in_pipeline;
7   //an intermediate filter in the pipeline
8   if(my_filter){
9    ...
10   //Scheduler bypass
11   recycle_as_continuation; next=this;
12   return next}
13  //last filter has completed an item
14  else {
15   // so start a new token (item)
16   input_tokens++;
17   ...
18   return NULL}
```

(b)

Figure 5: (a) Use of the pipeline template; (b) Sketch of the `execute` method.

When a token reaches the last stage in the pipeline (line 14(b)) a new token is released and the scheduler attempts to extract a task from the local task queue in FILO order. If unsuccessful, the outer loop of the scheduler attempts to steal a task from other thread (`steal_task`) in LIFO order. If the steal is unsuccessful the worker thread waits for a predetermined amount of time and then tries to steal again, giving up after a number of unsuccessful attempts.

The linear pipeline in `ferret` maps directly to the TBB pipeline template. On the other hand, `dedup`, with its item producing and bypassing stages, is more challenging. Because TBB requires an equal number of tokens entering and exiting a stage (and therefore the pipeline), we could not incorporate the `dedup` serial output stage as a TBB filter; instead, we created an individual queue and a dedicated thread, using Pthreads, to handle the output. The `FindAllAnchors` stage violates this constraint and thus the successive stages (`ChunkProcess` and `Compress`) can not be assigned to different TBB pipeline filters; instead, they have been collapsed into one parallel TBB pipeline filter. As a consequence, the `dedup` implementation is a hybrid, TBB and Pthreads, implementation.

We also explored other implementations of these codes, based on OpenMP [14] and MapReduce [17]. However, the streaming nature of these applications did not map well to these programming patterns.

3 Analytical Model of a Parallel Pipeline

The original Pthreads codes under study represent two cases of parallel pipeline. In this section we present an analytical model for each case. Besides, we incorporate the modeling and discussion of the collapsing of parallel stages. Furthermore, we study how to model the work-task stealing strategy proposed in TBB for the pipeline template.

The goal of our models is to understand the performance trade-offs involved in task stealing. The models can also serve as a system configuration tool to determine the size of the queues, the optimal number of stages, optimal number of tokens in the system, and the suitable number of parallel threads to scale to before the I/O bottleneck is reached.

As a primary performance metric, and to measure the validity of our models, we will focus in the computation of the execution times for the different implementations ($Time$). In all the cases, we compare

the analytical times obtained by our models with the times measured in an HP9000 Superdome, using the `native` input sets. Eqs. 1-3 are queueing theory equations, that we use to formulate the models.

$$\rho_i = \lambda_i \cdot \frac{Tser_i}{c_i} \tag{1}$$

$$\lambda_e = \begin{cases} min(\lambda_i) & \text{Pthreads (Case 1, Case 2)} \\ \sum_{i=1}^{npTh} \lambda_i & \text{TBB (Case 3)} \end{cases} \tag{2}$$

$$Time = \frac{1}{\lambda_e} \cdot nit \tag{3}$$

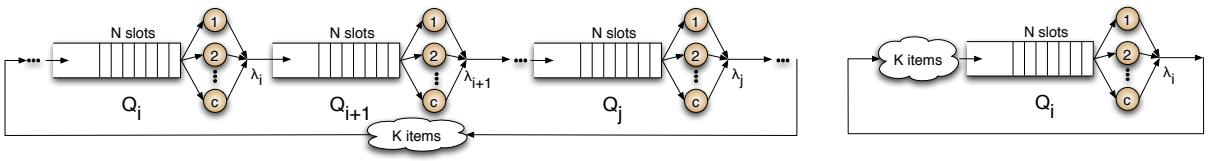
In our models, we use the concept of *logical queue* or *queue system* to explicitly represent: i) the buffer where the pending items are waiting to be processed, and ii) the servers (or threads) that process the items. In some cases (closed systems), the queueing system also includes a representation of the population that can ask for a service. The mean times used in our models ($Tser$ and $Tarr$) were obtained from the sequential codes. Due to space constraints we just sketch the main results of our models. The full model will be presented in a technical report associated with this paper.

3.1 Case 1. A parallel pipeline closed system

The parallel pipeline Pthreads implementations, where each stage S_i consists in an input queue buffer Q_i and a pool of dedicated working threads c_i, is naturally modeled as a network of single logical queues [1]. One particular feature of the Pthreads `ferret` implementations is that the queue buffers have limited buffer capacity which produces the stalling of items in the stages, as we see in Fig. 2(b). For that reason, we can see the *pipeline as a closed system*: since a new item can not enter until a previous one leaves the system, the items can be viewed as circulating continuously and never leaving the network of queues. Fig. 6(a) shows a scheme of that closed system. In this model, we can assume that there is a finite-calling population of K items.

Through several simulations, we have found that in steady state, each stage S_i behaves like a $M/M/c_i/N/K$ queue system [1]. This represents a logical queue with exponential arrival and service time distributions, c_i servers (the working threads), system size N (approximately the queue buffer size, $N = Queue\_size$) and job population $K = N$ [18]. Fig. 6(b) shows the abstract view of one stage. For each stage S_i, the mean service time, $Tser_i$, is given by the time to process an item on that particular stage. One important observation is that the mean time for the arrival call for service on all stages, $Tarr$, is determined by the longer stage in the pipeline, i.e. $Tarr = max(Tser_i)$. In the closed model, the internal throughput for each stage S_i, λ_i, is obtained from eq. 4. It depends on P_n, the probability that there are n items in the stage, which is a function of c_i and $Tser_i$ (see [1]).

$$\lambda_i = \sum_{n=0}^{N} (N - n) \cdot \frac{P_n}{Tarr} \tag{4}$$



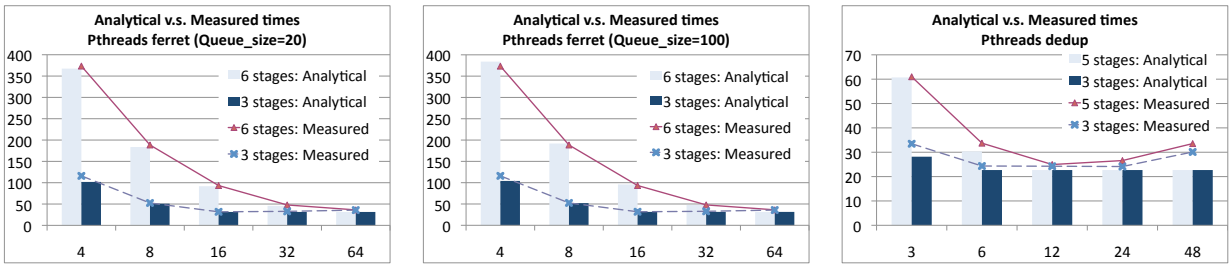
(a) (b)

Figure 6: (a) Model of a parallel pipeline closed system; (b) Model of a stage in a closed system.

The first implementation that we analyze is the original Pthreads version with 6 stages (see Fig. 1). In the model, one thread is assigned to the serial stages, S_1 and S_6, i.e. $c_1 = c_6 = 1$. For the parallel stages $i = 2:5$, the number of threads assigned to each stage is $c_i = npTh/npstages$, where $npTh$ and $npstages$ are the variables in our models and represent, respectively, the number of parallel threads, and the number of parallel pipeline stages. Using eq. 4, we compute the internal throughput, λ_i, for each stage. The effective throughput for the whole pipeline, which we name λ_e^{st6}, is computed by eq. 2 - Case 1. It depends on the slower stage. Once we know the effective throughput, λ_e^{st6}, and the number of items to be processed, nit, the execution time for a run, $Time^{st6}$, can be easily computed by eq. 3.

Next we analyze the Pthreads implementation that collapses the parallel stages. There are 3 stages: $S_{1'}$, $S_{2'}$ (which represents the collapsing of the parallel stages) and $S_{3'}$. Here, $c_{1'} = c_{3'} = 1$, and $c_{2'} = npTh$ is the number of parallel threads in the collapsed stage. Using eq. 4, we compute the internal throughput for each stage. The effective throughput for the whole pipeline, λ_e^{st3}, is determined by eq. 2 - Case 1. The running time for this implementation, $Time^{st3}$, is computed by eq. 3.

Fig. 7 (a)-(b) compares the analytical times obtained for $Time^{st6}$ and $Time^{st3}$ for different number of parallel threads (the $npTh$ parameter). We also represent the measured times for the two Pthreads implementations of `ferret`. We show the behavior of the system for two queue buffer sizes: (a) $Queue\_size = 20$ and (b) $Queue\_size = 100$.



(a) (b) (c)

Figure 7: Analytical vs. measured times (sec.) for different number of parallel threads: (a) `ferret` $Queue\_size = 20$ items; (b) `ferret` $Queue\_size = 100$ items; (c) `dedup`

The analytical times accurately predict the measured times. Observe that the impact of the queue size on the execution time is small for sizes above 20 items – the default value in the original implementation. Thus, the original implementation is correctly dimensioned. We will show in Section 3.4 how to use the model to find the optimal system capacity ($Queue\_size$) for each implementation. Another result is that each implementation presents a different scalability behavior and that the I/O bottleneck is found for different number of threads on each implementation. In fact, this bottleneck manifests sooner for the collapsed 3-stages version. In Section 3.4 we explain how to find the I/O bottleneck and how to estimate the optimal number of threads per stage ($c_{optimal}$) for which the effective throughput is the maximum.

The most important result from the model is that the 3-stages implementation outperforms the 6-stages one: it always reaches smaller running times for the same number of parallel threads and scales better (till the I/O bottleneck). The analytical model allows us to explain this effect: for all the cases, and before reaching the I/O bottleneck, $\lambda_e^{st6} < \lambda_e^{st3}$ due to the load imbalance in the service times.

3.2 Case 2. A parallel pipeline open system

The Pthreads implementations of the `dedup` code represents a different case of parallel pipeline. Although in this code, there can be more than one physical local queue buffer on each stage, without loss of generality, we can assume that they represent one logical queue storage per stage. In fact, the size of each local buffer is 1 million items. This can be modeled by considering that the system has infinite capacity ($N = \infty$), and thus, there is not stalling of items in the stages, as shown in Fig. 4(b). When the capacity is infinite, in the steady state, we can see the *pipeline as a open system*: the items flow among stages with different internal

throughput. Fig. 8 shows a view of this model. Assuming exponential arrival and service times, each stage S_i of this pipeline behaves likes a $M/M/c_i$ queue system. This is the model typically found in the literature for pipelines [15, 13].

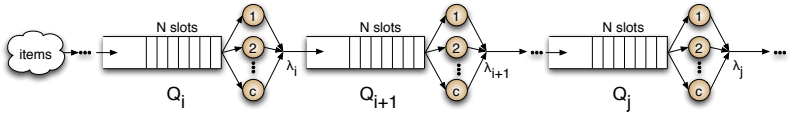


Figure 8: Model of a parallel pipeline open system.

In an open network, the arrival time, $Tarr_i$, is different for each stage S_i. It can be computed composing the throughput from the stages in the network that reach S_i. Let p_{ji} represent the fraction of items that outgoing from stage S_j, reach stage S_i. We can model the arrival time for each stage as,

$$Tarr_i = \frac{1}{\sum_{j\ reaches\ S_i}(p_{ji} \cdot \lambda_j)} \tag{5}$$

In this model, eq. 6 give us the internal throughput for each stage. It depends on the arrival time for the corresponding stage.

$$\lambda_i = \frac{1}{Tarr_i} \tag{6}$$

The first implementation that we analyze is the original Pthreads version with 5 stages. One thread is assigned to each serial stage, i.e. $c_1 = c_5 = 1$ and for the parallel stages $i = 2 : 4$, the number of threads per stage is $c_i = npTh/npstages$. The effective throughput for the whole pipeline, which we name λ_e^{st5}, is determined by the slower stage as eq. 2- Case 2 indicates, and the execution time for the run, $Time^{st5}$, is computed by eq. 3.

The second Pthreads implementation that we model is the 3-stage version, in which we collapse the parallel stages in one. We assign 1 thread to stages $S_{1'}$ and $S_{3'}$, respectively, and $c_{2'} = npTh$ parallel threads to the collapsed stage $S_{2'}$. We compute the internal throughput for each stage, the effective throughput for the whole pipeline, λ_e^{st3}, and the running time, $Time^{st3}$ (eq. 3).

In Fig 7 (c) we compare the analytical $Time^{st5}$ and $Time^{st3}$ times vs. the measured times for the two Pthreads implementations of `dedup`, for different number of parallel threads. In this case, the analytical times just represent an optimistic lower bound of the pipeline behavior. One particular issue, non captured in the model, is the non-deterministic behavior of this code, particularly the increment in the size of the output file when the number of threads increases. This is the main cause of the divergence between the analytical and measured times for both Pthreads implementations, especially when the number of parallel threads is greater than 12.

We find similar results as in the `ferret` case: the scalability behavior of each Pthreads version is different, in fact the 3-stages version scales faster; for this reason the I/O bottleneck appears sooner in the collapsed version. As in `ferret`, the load imbalance of service times explains why the 3-stages version always outperforms the original 5-stages code.

3.3 Case 3. Model for work stealing

Deriving an analytical model for the parallel pipeline paradigm which incorporates work stealing is difficult because stealing hinges on temporary imbalances among the working threads, making the application of steady state modeling techniques non-trivial. While the analytical model can give us some hints about how the parameters and resources interact, we will capture the dynamic behavior of stealing and the associated overheads with the aid of real-system measures of the different implementations, whose results we show and discuss in section 4.

Let's recall that in the pipeline TBB implementation there is a logical queue of tasks per thread (differently to the Pthreads implementations with a logical queue per stage). In this new model, we do not see a network of logical queues, but rather a set of loosely coupled threads working in parallel. Each thread is the server of his local queue, where the items (the tasks) to be processed arrive. Eventually the other remote threads in the system can steal work from the local queues. Each time that a task is processed by one thread, a slot becomes available in his local queue and a new task can enter in the system. So, each thread can be viewed as a queue in a closed model: each thread can be modeled as a $M/M/c/N/N$ queue system (see Fig. 9(a)).

Each task carries on the work from the input filter to the output filter through the intermediate stages of the pipeline. The service time, $Tser_{TBB}$, depends on the whole processing of an item. $Tarr_{TBB}$, or the time for the arrival call for service, also depends on the processing of one item too, $Tarr_{TBB} = Tser_{TBB}$. The number of servers (c) per queue is initially 1 thread ($c = 1$). The total system capacity is finite and is given by the number of tasks that can exist simultaneously in the system: *ntokens* (a parameter described in 2.3). Initially we assume that the logical queues for all the threads have equal capacity and that the items are evenly distributed, i.e. $\forall\ i = 1 : npTh, N = ntokens/npTh$. Thus, N represents the number of tokens per thread. The population that can call for a service is finite and equal to that capacity.

The concept of work stealing is similar to that of *load sharing*. An analytical model of load sharing for a distributed closed system was proposed in [21]. It uses a three steps framework to model the load sharing. We can exploit their observations for our model as follows: the effect of load sharing (task stealing) on the performance of a queue based distributed system, is similar to that of adding more working resources to each queue. The model for work stealing can be derived following a three steps framework that we sketch next.

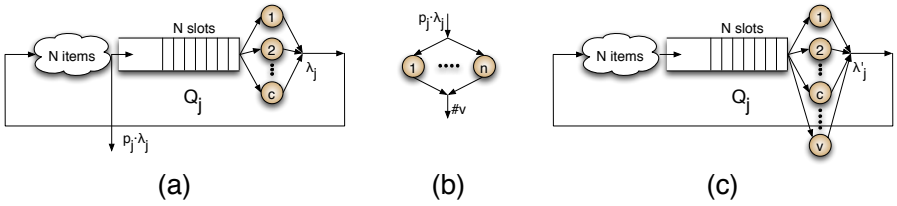


Figure 9: Model for work stealing: (a) Step 1: A local queue; (b) Step 2: Stealing of items; (c) Effect of the stealing in the local queue.

Step 1. Each stage in the pipeline is an isolated local $M/M/c/N/N$ queue system (see Fig. 9(a)). Initially, $c = 1$. From this step we get λ_j (the internal throughput for j-th thread) and p_j (the probability that the j-th thread is busy [21]).

Step 2. The stealing of tasks from other threads can be modeled as a $M/M/n_j/n_j$ system. For each thread, we use $p_j \cdot \lambda_j$ as the offered load (see Fig. 9(b)) for the stealing of tasks and n_j is the remote available capacity to perform the stolen work. That n_j is determined by eq. 7. Next, we derive v, which is the average number of active services in this system, that represents the stolen tasks.

$$n_j = nidle_j = \sum_{\substack{k=1:npTh \\ j \neq k}} (1 - \rho_k) \cdot c_k \tag{7}$$

Step 3. When we incorporate work stealing in our model, each thread is now modeled as a $M/M/c+v/N/N$ queue system, i.e. each thread behaves as if a (virtual) working capacity of v additional threads are added. See Fig. 9(c). Now, we recompute the internal throughput λ'_j (see eq. 4), taking into account the new working capacity. We refer to this model as the Dynamic Scheduling (DS) model.

We compute the effective throughput of the whole system, λ_e^{TBB}, by aggregating λ'_j from all the threads, as indicated by eq. 2- Case 3.

One important issue that is not directly captured in this model, is that the serial filters eventually become a bottleneck. This limitation can be easily modeled by adding eq. 8 as a new constraint in our model:

$$\lambda_e^{TBB} = \frac{1}{max(Tin, Tout)} \quad , \quad If \quad max(Tin, Tout) > \frac{1}{\sum_{j=1}^{npTh} \lambda'_j} \tag{8}$$

where Tin and $Tout$ are the times to process a item in the input and output filters, respectively. Finally, the execution time, $Time^{TBB}$, is computed from eq. 3.

In Fig 10 we compare the analytical $Time^{TBB}$ time (named "DS-Analytical") vs. the measured times for the TBB `ferret` implementation on different numbers of threads. The TBB `ferret` code has 6 pipeline filters: one input serial filter, four parallel filters, and one output serial filter. We compare the times for different number of tokens in the system: 20, 100 and $6 \times npTh$. Recall that the *ntokens* variable controls the system capacity. In fact, they represent the number of task that exists concurrently at any given point, and it allows us to measure whether it helps to take advantage of TBB's dynamic load balancing feature.

For comparison, we also include the times obtained for an ideal centralized system based on a global $M/M/npTh/ntokens/ntokens$ logical queue, which we name "Ideal". This is a boundary case that represents a centralized queue system with the same aggregate number of threads and task capacity that our queue distributed implementation in TBB. This centralized system represents the optimal case [21] in queueing theory, and serve as a baseline to measure the optimality of the task stealing implementation.

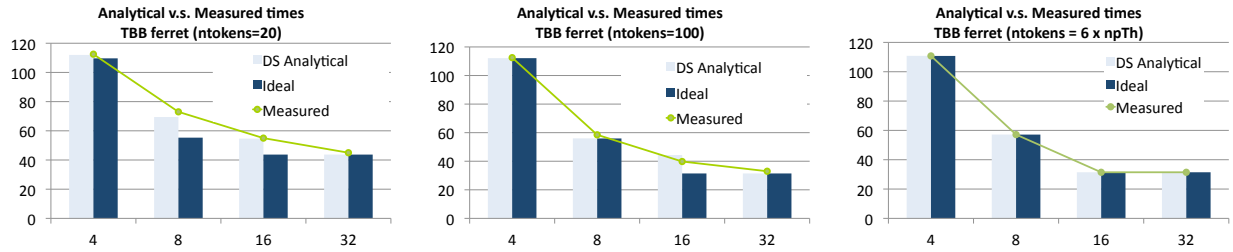


Figure 10: Analytical vs. measured times (sec.) for TBB and different number of parallel threads: (a) $ntokens = 20$; (b) $ntokens = 100$; (c) $ntokens = 6 \times npTh$.

The analytical times accurately predict the measured times, which give us a proof of their validity. One important result is that the selection of *ntokens* has impact on the execution time, especially when the *ntokens* is a small value and the number of threads increases. This indicates that load balancing opportunities are lost. When the number of tokens is large enough, the TBB implementation behaves optimally. The *ntokens* parameter is in fact a trade-off parameter. The appropriate selection of this variable give us the key to get the whole pipeline system behaves optimally. The analytical model provides some insight: when *ntokens* is sufficiently large, the task stealing in TBB emulates a global queue that is able to feed all the threads. An additional advantage of the TBB implementation is that the distributed solution with one queue per thread avoids the contention that a global single queue would exhibit. However the value of *ntokens* should be selected carefully, since high values introduce overhead (non-captured in the model) because of excessive overheads due to task management, synchronization and contention. These can eliminate the benefits of the dynamic scheduling [6]. In Section 3.4 we demonstrate how to use the model to compute the optimal *ntokens* for the system.

Another important result is that the analytical times obtained for the TBB `ferret` code, do not depend on the number of pipeline stages (as opposed to the Pthreads implementation). In other words, from the TBB model point of view, selecting the appropriate number of stages in the parallel pipeline is not an issue. This result will be corroborated later in the experimental section, where we measure and compare the execution times for some other parallel TBB-pipeline implementations of `ferret` in which we choose different number of stages.

We also analyzed the TBB pipeline port of the `dedup`, The results and conclusions are similar to the `ferret` case.

3.4 Sizing the System

In this section we use our model to find out i) the optimum values for the capacity of the closed model, N ($Queue\_size$ in Pthreads and $ntokens$ in TBB) so we can minimize the storage requirements of each implementation; and ii) the optimum number of threads, $c_{optimal}$ that with a high utilization achieves the maximum effective throughput.

In the closed models, N, can be optimized from eq. 4 where we see that increasing N increases the internal throughput until we reach an upper bound. Ideally, assuming a high utilization $\rho_i \approx 1$, and from eq. 1, we find that the upper bound λ_{upper_i} for the internal throughputs is,

$$\lambda_{upper_i} \approx \frac{c_i}{Tser_i} \tag{9}$$

The other factor that limits the effective throughput is the critical serial stage, that is, the serial stage with longer service time. This stage ends up being the I/O bottleneck of our implementations. Let λ_{max} be the throughput due to the critical serial stage. Let Tin and $Tout$ be the service times to process an item in the serial input and output stages. From eq. 1, as $\rho_i < 1$ and as the serial stages have $c_i = 1$ thread, we can find an optimistic upper bound for λ_{max},

$$\lambda_{max} = min\left(\frac{1}{Tin}, \frac{1}{Tout}\right) \tag{10}$$

Therefore, an upper bound of the effective throughput is $\lambda_{eu} = min(\lambda_e, \lambda_{max})$. For the Pthreads implementations (see eq. 2- Case 1) $\lambda_{eu} = min(min(\lambda_{upper_i}), \lambda_{max})$ and for the TBB ones (see eq. 2- Case 3) $\lambda_{eu} = min(\sum(\lambda_{upper_i}), \lambda_{max})$. From eq. 4 we can find the optimal value of $N_{optimal}$ that reaches that upper bound, as we express in eq. 11:

$$\sum_{n=0}^{N_{optimal}} (N_{optimal} - n) \cdot \frac{P_n}{Tarr} = \lambda_{eu} \tag{11}$$

In Fig. 11 we show the term $\lambda_e = min(\lambda_{upper_i})$ for the Pthreads codes and $\lambda_e^{TBB} = \sum(\lambda_{upper_i})$ for the TBB codes. They are represented for different number of parallel threads ($npTh$) and values of N. We study our closed systems: the 6 stages and the collapsed 3 stages Pthreads implementations of `ferret`, as well as the TBB implementations of `ferret` and `dedup`. In each case, we also represent λ_{max}. As shown in Figure 11, the throughput eventually reaches the upper bound value. Applying eq. 11, we found that for the `ferret` code, $N_{optimal}$ is around 21 items per queue in the 6 stages Pthreads, and 24 items per queue in the 3 stages Pthreads one. In these cases, the queue buffers will consume a total storage requirement of $N_{optimal} \times (npstages + 1)$ items.

Regarding the TBB cases, the optimal for the `ferret` case is around 5 tokens per thread and in the `dedup` case is around 3 tokens per thread. In other words, the optimal capacity is $ntokens = 5 \times npTh$ and $ntokens = 3 \times npTh$, respectively, which are in fact the storage requirements for these implementations.

The implementations will scale with the number of threads, until the effective throughput reaches λ_{max}, i.e. until the I/O bottleneck. Another interesting application of the model is to find the value of c_i once the system capacity is selected, which is the intersection between the effective throughput and λ_{max}. This value is the optimal number of threads, $c_{optimal}$ for which we expect the maximum performance, i.e. the maximum effective throughput. So, once selected the system capacity, from equations 4, 2- Case 1, and 2- Case 3, we can find the value of $c_{optimal}$ which verifies $\lambda_e = \lambda_{max}$. Fig. 11 can help us again. For the `ferret` Pthreads implementations with $N = 20$, we have found that, in the collapsed 3 stages implementation $c_{optimal} = 15$ parallel threads, while in the 6 stages code $c_{optimal} = 12$ threads per parallel stage, it is a total of $npTh = c_i \times npstages = 12 \times 4 = 48$ parallel threads. In the `ferret` TBB implementation we have

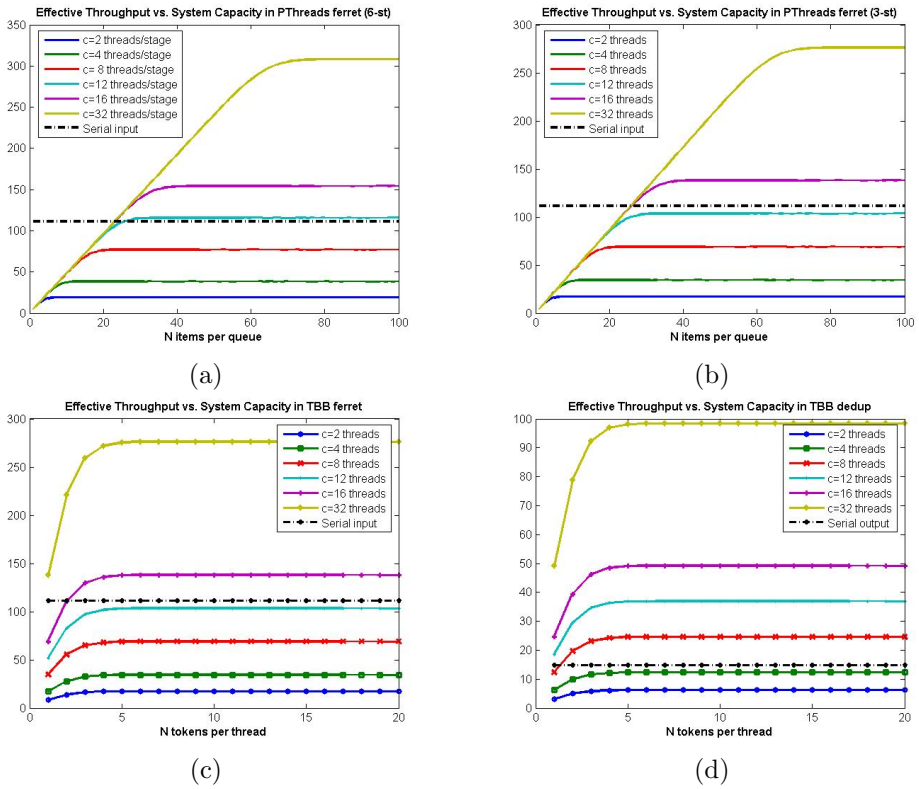


Figure 11: Effective throughput vs. System capacity (N): (a) 6 stages Pthreads `ferret`; (b) 3 stages Pthreads `ferret`; (c) TBB `ferret`; (d) TBB `dedup`.

found that for $N = 5$, $c_{optimal} = 15$ parallel threads, while for TBB `dedup` with $N = 3$, $c_{optimal} = 5$ parallel threads.

For the open pipeline system, i.e. the Pthreads `dedup` codes, the upper bound of the internal throughput is given by eqs. 1 and 6, $\lambda_{upper_i} \approx c_i/Tarr_i$. In this case, the I/O bottleneck will be found when $\lambda_e = min(\lambda_{upper_i}) = \lambda_{max}$. The value of c_i that verifies that expression is $c_{optimal}$.

4 Evaluation

So far we have shown that our analytical model faithfully tracks the performance of the `ferret` and `dedup` benchmarks, even though we do not model all overheads. In this section, we shall focus on the measured performance of the different optimized implementations as discussed in Section 2.3. Since we have already proved that collapsing parallel stages of a pipeline is profitable, we address here the other proposed solutions. As an experimental platform we use the same methodology and target platform that we described in section 2.

4.1 Static scheduling vs. work-stealing

As discussed before, the first bottleneck for scalability is load imbalance. To tackle this issue, we propose two solutions: oversubscription and dynamic scheduling.

In [2], the authors recommend a configuration in which the number of logical threads per parallel pipeline stage, c, is set to the total number of available cores. Our target platform, the HP Superdome, has a total 128 cores which will result in a total of 514 and 386 threads for the `ferret` and `dedup` pipelines, respectively.

Threads	6-stages Pthreads not-oversubscribed				6-stages Pthreads oversubscribed			
	# cores	Time (sec.)	Speed-up	Efficiency	# cores	Time (sec.)	Speed-up	Efficiency
6	6	370,68	1,18	19,65%	1	438,45	1,00	99,67%
10	10	186,69	2,34	23,41%	2	220,87	1,98	98,93%
18	18	94,87	4,61	25,59%	4	111,88	3,91	97,65%
34	34	47,95	9,11	26,80%	8	69,21	6,31	78,93%
66	66	38,41	11,38	17,24%	16	48,86	8,94	55,90%

Table 1: Not-oversubscribed vs. oversubscribed comparison for the 6-stages Pthreads `ferret` code.

	6-stages TBB			3-stages TBB		
# cores	Time (sec.)	Speed-up	Efficiency	Time (sec.)	Speed-up	Efficiency
1	437,65	0,77	77,00%	442,88	0,76	76,09%
2	221,13	1,52	76,20%	223,5	1,51	75,39%
4	112,59	2,99	74,83%	113,58	2,97	74,18%
8	58,5	5,76	72,01%	59,56	5,66	70,73%
16	35,87	9,40	58,72%	36,98	9,11	56,96%
32	33,14	10,17	31,78%	33,42	10,08	31,51%
64	35,66	9,45	14,77%	36,44	9,25	14,45%

Table 2: 6-stages vs. 3-stages comparison of the TBB implementations of `ferret`.

Since we reach the serial I/O bottleneck imposed by the input and output stages at about 16 threads, such a configuration is not practical. Therefore we chose to achieve oversubscription using a parametrized number of cores and binding processes (and all their threads) to cores using the `taskset` UNIX command. To achieve high utilization, our Pthreads implementation relies on the OS scheduling policy to assign ready threads to the hardware threads, while the TBB implementation relies on the library-level tasks.

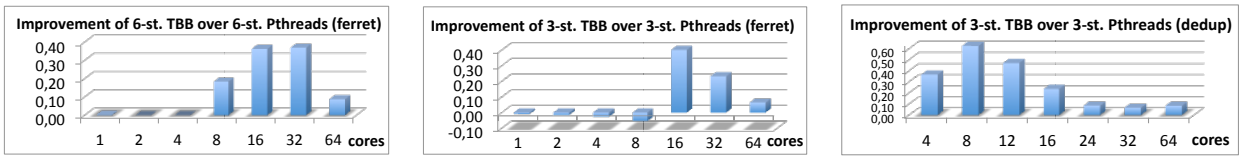
Table 1 shows the execution time, speed-up relative baseline sequential time, and efficiency for the non-oversubscribed and the oversubscribed 6-stage Pthreads versions. The oversubscribed version clearly outperforms the non-oversubscribed version running with the same number of threads on a smaller number of cores by achieving better hardware utilization.

The second solution we evaluate is dynamic scheduling using work-stealing. We coded `ferret` and `dedup` using the pipeline template from the TBB library. We have used the *ntokens* values suggested by our model, that we have empirically validated. For `ferret`, we implemented two TBB versions: 6-stages and 3-stages, both with $ntokens = 5 \times npTh$. Table 2 shows the execution time, speed-up and efficiency for these two versions. As predicted by the analytical model, the number of stages in the TBB pipeline implementation does not affect the execution time. The load imbalance present in the original 6-stage pipeline is essentially removed by dynamic scheduling, and the two versions have similar behavior profiles across all thread configurations.

In Figure 12 we compare the performance of the TBB implementation using work stealing with the Pthreads version using oversubscription. Again, the measurements are in accordance with the analytical model prediction: the TBB version is faster when the number of cores is larger than 8 since there are more opportunities for task stealing. The largest improvement for the 6-stages version is for 16 and 32 cores where the TBB version is 37% faster. For the 3-stages version, the largest difference is for 16 cores where the TBB version is 42% faster. These improvements degrade when approaching the input bottleneck which is reached faster in the oversubscribed Pthreads 3-stages implementation (around 16 cores).

For the `dedup` kernel, we implemented only a 3-stages TBB pipeline version due to the restrictions in the TBB template explained in Section 2.2, and $ntokens = 3 \times npTh$. Figure 12 (c) compares the 3-stages oversubscribed Pthreads and the 3-stages TBB implementations. The TBB version shows up to 63% of improvement (8 cores). As we approach to the output bottleneck, which happens for more than 8 cores, the advantages of dynamic scheduling start to diminish.

We also quantified the overhead of the TBB pipeline template using the Intel VTune 9.1 profiler. Contrary to what it was reported in [6] for the TBB `parallel_for` template, the overhead introduced by the TBB scheduler is very small for the pipeline template and the contribution of the `steal_task` function (the main



(a) (b) (c)

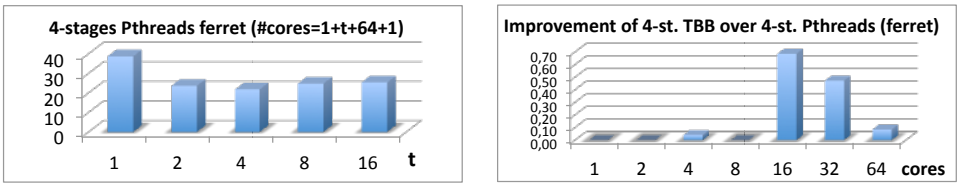
Figure 12: (a) 6-stages and (b) 3-stages oversubscribed Pthreads versus TBB implementations of `ferret`; (c) 3-stages oversubscribed Phreads versus TBB implementations of `dedup`.

function that implements work stealing) is negligible at less than 0.05% of execution time in all the cases. While a naive measurement will assign a significant part of the execution time to the TBB library, our measurements indicate that the serial bottleneck on the input stage causes the other stages to wait in the TBB scheduling loop.

4.2 I/O Optimization

It is quite common that the stages at the ends of the pipeline become a bottleneck. Both the input and the output stages may be gated by either the serial requirements to read or write data from/to the I/O stream, or simply by the I/O bandwidth of the machine. We measured both causes on our benchmarks when we removed the load imbalance in the pipeline.

While the `ferret` benchmark input stage is serialized in the original 6-stage implementation (i.e., `load` serially scans the input directory to find new images), there is no dependence between different images, therefore this stage can be parallelized. One optimization scenario is to divide the first stage into two stages, a serial one that scans the input directory and enqueues all the image names in an intermediate queue, and a second parallel stage that extracts names from the queue and reads the images. We implemented this optimization in a 4-stage Pthreads `ferret` version: two input stages as described above, all processing stages collapsed into one stage, and an output stage. Our goal is to find the maximum number of threads that can read the image files in parallel. Figure 13(a) shows the total execution time when running `ferret` with one thread for stage 1, t threads for the parallel input stage 2, 64 threads for the working stage 3 (ensuring that way that total execution times are practically bounded by the input time), and one thread for the output stage. By increasing the number of dedicated input threads to 4 in the second stage, we see a 2x performance improvement. As the number of threads increases above 4, threads start contending on the I/O bandwidth of the machine. Although other parallel input solutions are possible, we have chosen to keep the streaming philosophy of the original `ferret` code in which the first stage of the pipeline is serial.



(a) (b)

Figure 13: (a) Total execution time in 4-stages `ferret` for different number, t, of threads collaborating in reading files from the hard disk; and (b) Improvement of 4-stages `ferret` TBB over Pthreads.

We also implemented a 4-stages TBB implementation of `ferret`, splitting the first stage in a similar manner. As opposed to the 4-stages Pthreads implementation where the number of threads can be set by the user, in the TBB implementation with work-stealing the number of task working in each pipeline stage is dynamic. As shown in Figure 13 (b), the TBB pipeline outpeforms the Pthreads pipeline up to 69%, again showing the advantage of dynamic load balancing.

For `dedup` the bottleneck is in the output stage, but since it is writing a compressed stream in a single file, it can not be parallelized without significantly changing the original algorithm. A possibility we have not yet explored would be to allow multiple threads to write different output streams and combine these into a single file in a reduction stage. However, since this is compressed data, the combining algorithm is non-trivial and it is outside of the scope of this paper.

5 Related Work

The emergence of new workloads that are streaming data has brought attention to the parallel pipeline programming paradigm, since these workloads can be parallelized naturally using this model. The work of Raman et al. [16] exploits a technique called Parallel-Stage Decoupled Software Pipelining (PS-DSP). The technique allows to identify, with some small interventions of the programmer, the pipeline stages as well as to partition the threads between the parallel stages. However, their thread partitioning algorithm uses a static (compile-time) partitioning of threads for the stages. Thies et al. propose in [22] a set of simple annotations to let the programmer mark the pipeline stages and use dynamic analysis to track the data communications between stages. They also present the dynamic data in a stream graph representation of the application to help the programmer improve parallelism and load balancing. In this work, the selection of threads is also static. Both approaches suffer from load imbalance, thus degrading the scalability of the applications.

Other approaches for coding streaming applications is to use programming language with support for streams. Examples include: StreamC [8], Brook [5], and StreamIt [20]. The main target of this kind of languages are stream processors [7], or graphics processors [5], although there have been some works that have studied the mapping to general purpose multiprocessors [10]. In all the cases, a static scheduling of threads per stage of the pipeline is always implemented, and degradation of the speedups due to runtime load imbalance is generally reported.

A model for study the performance of a parallel pipeline system is proposed by Liao et al. in [13]. This paper models only the open parallel pipeline, and although the collapsing of parallel stages is considered, strategies for solving the load balancing problem are not studied. The model of the parallel pipeline closed system, as well as the analytical model for work stealing in the context of the parallel pipeline paradigm are two important contributions of our paper.

An important body of work, based on dynamic scheduling, has been developed for solving the problem of load imbalance, especially in the context of irregular applications. For instance, load balancing schemes using tasks queues on shared-memory architectures are described in [11] and [12]. Languages such as Cilk [3], and X10 [23] have proposed efficient implementations of dynamic scheduling based on work stealing [4]. In these studies, the parallel pipeline paradigm has not been considered – a distinguishing feature of our work.

6 Conclusions

In this paper we studied the pipeline parallelism programming pattern. We identified two workloads that exhibit pipeline parallelism in the PARSEC benchmark suite and characterized their behavior on a large scale SMP machine. We identified two issues that limit scalability: load imbalance and I/O bottlenecks. We developed a queueing model for the behavior of pipeline parallelism that can be used to understand and tune the behavior of applications using this programming patterns. We propose two techniques to address load imbalance in the pipeline programming model, namely, parallel stage collapsing and dynamic scheduling based on work-stealing. We implemented these techniques and evaluated them both by applying our pipeline parallelism model and by running on a real system. We observe that the model faithfully captures the behavior.

For our implementation of dynamic scheduling we used the TBB library, which provides a pipeline template and supports work-stealing. There are a number of advantages of using a library that supports work-stealing: i) The programmer can build his algorithm from a high level point of view, without worrying

about the number of stages, load balancing between stages or the number of threads allocated to each stage; ii) When selected the appropriate number of tokens, the TBB behaves near optimally as we have demonstrated with our analytical model; and iii) The TBB library overheads for the pipeline template are negligible for the number of cores we have used in our experiments (up to 64). We have also identified limitations in the TBB pipeline template, in particular the ability of specifying non-linear pipelines, specifying stages that produce a different number of outputs, and setting the maximum degree of parallelism independently for each pipeline stage.

Finally, regarding the I/O bottleneck we explored the possibility of improving the performances by allowing parallel I/O. While in many cases this is algorithm dependent, we point out that once load balancing is resolved, feeding and clearing the pipeline becomes the next bottleneck. This will become a cornerstone to achieve scalability on multi-core systems for RMS and streaming applications.

References

[1] A. O. Allen. *Probability, Statistics, and Queueing Theory - With Computer Science Applications*. Academic Press; 2 edition, 1990.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[3] 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 *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[4] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM.

[6] G. Contreras and M. Martonosi. Characterizing and improving the performance of intel threading building blocks. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 57–66, Sept. 2008.

[7] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2003. IEEE Computer Society.

[8] A. Das, W. J. Dally, and P. Mattson. Compiling for stream processing. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 33–42, New York, NY, USA, 2006. ACM.

[9] B. B. Fraguela, J. Guo, G. Bikshandi, M. J. Garzarán, G. Almási, J. Moreira, and D. Padua. The hierarchically tiled arrays programming approach. In *LCR '04: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, pages 1–12, New York, NY, USA, 2004. ACM.

[10] J. Gummaraju, J. Coburn, Y. Turner, and M. Rosenblum. Streamware: programming general-purpose multicore processors using streams. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 297–307, New York, NY, USA, 2008. ACM.

[11] J. Hippold and G. Rnger. Task pool teams: a hybrid programming environment for irregular algorithms on smp clusters. *Concurrency and Computation: Practice and Experience*, 18(12):1575–1594, 2006.

[12] M. Korch and T. Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms. *Concurrency and Computation: Practice and Experience*, 16(1):1–47, 2004.

[13] W.-K. Liao, A. Choudhary, D. Weiner, and P. Varshney. Performance evaluation of a parallel pipeline computational model for space-time adaptive processing. *J. Supercomput.*, 31(2):137–160, 2004.

[14] The OpenMP API specification for parallel programming. `http://www.openmp.org/`.

[15] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 114–123, New York, NY, USA, 2008. ACM.

[16] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 114–123, New York, NY, USA, 2008. ACM.

[17] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[18] G. Redinbo. Queueing systems, volume i: Theory–leonard kleinrock. *Communications, IEEE Transactions on*, 25(1):178–179, Jan 1977.

[19] J. Reinders. *Intel Threading Building Blocks: Multi-core parallelism for C++ programming*. O'Reilly, 2007.

[20] The StreamIt Programming Language. `http://www.cag.lcs.mit.edu/streamit`.

[21] Y. Tay and H. Pang. Load sharing in distributed multimedia-on-demand systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):410–428, 2000.

[22] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[23] The X10 Programming Language. `http://www.x10-lang.org`.