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 the usefulness of the model by characterizing and optimizing two of the PARSEC benchmarks, ferret and dedup. We identified two issues with these codes: load imbalance and I/O bottlenecks. We addressed load imbalance using two techniques: i) parallel pipeline stage collapsing; and ii) dynamic scheduling. 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 based on work stealing outperforms all other variants.

1 Introduction

As multi-core processors are becoming ubiquitous, parallel programming is the technology that can make them succeed or not. Research in parallel programming has focused on two main directions: (i) new parallel languages; and (ii) parallel libraries – developing highly optimized libraries that encapsulate data parallelism [9] or task parallelism [17]. Both approaches aim at providing higher levels of abstraction to allow programmers to focus on algorithms and data structures rather than the complexity of the architecture.

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”.

Pipeline parallelism is an important programming pattern and we are interested in providing models, tools, and guidance to the programmers for tuning the scalability and the performance of codes using this pattern. Applications parallelized using the pipeline model are very sensitive to load balancing: for best efficiency, pipelines must avoid bubbles, thus all stages must be processing at all times. Thus, the programmer must either partition the work into well balanced work items that flow through the pipeline or use a system that dynamically shifts resources from idle stages to the busiest pipeline stages. One such example is a system supporting work stealing [4]. Another bottleneck of the pipeline parallelism pattern is I/O, typically encountered at the ends of the pipeline. In the examples we studied, at large number of threads ferret is bounded by the input I/O and dedup is bounded by the output I/O stage. Optimizing I/O stages typically requires algorithmic knowledge and the use of a parallel I/O library.

Given this sensitivity, we explore analytical modeling of pipeline parallelism using queueing theory. We consider several pipeline configurations that capture the most common patterns. Such models can be used to assist programmers in tuning the parameters of their pipeline implementations without resorting to large numbers of simulations. We validate our models against measurements on a large parallel system.

To summarize, this paper makes the following contributions: i) 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 solutions to address them (Section 2); ii) New analytical models for pipeline parallelism based on queueing theory. The goal of the models is to help us better understand the use of resources
for different pipeline configurations (Section 3); iii) A performance comparison of several implementations of these benchmarks, including the original versions, a collapsed pipeline version and a version using work stealing, using Pthreads and Intel Threading Building Blocks (TBB) [17]. We demonstrate that our analytical model can capture precisely the behavior of all these variants (Section 4); iv) A discussion of the advantages of using work stealing and the TBB library for enhanced productivity in coding pipeline parallel programs. We also identify several aspects that can be improved in the TBB implementation of the pipeline template (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 which allows the programmer to tolerate different dependence patterns; (ii) communication is deterministic, following a producer-consumer pattern between pipeline stages. Note that pipeline parallelism is most appropriate implementation for the intended design of these applications [2], although if considered just in the benchmarking setup they could be rewritten using a different, more efficient, parallelism structure.

First we study the behavior of ferret and dedup benchmarks, 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 the Intel icc 10.1 compiler. The base implementation uses the Pthreads library. In the following discussion, we use c to denote the number of threads used per parallel stage, and nThreads for the total number of threads in the application.

2.1 Similarity search: ferret

The ferret application included in the PARSEC suite is an instantiation of the Ferret toolkit configured for image similarity search [2]. As shown in Fig. 1, ferret is decomposed into six pipeline stages. The input and output 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 parallel middle stages are configured with a c-size thread pool each. Object data is passed between stages using software queues, configured for 20 entries (default Queue size). We ran our experiments using the largest input set (native), comprising of 59,695 images. The sequential execution time for this input is 437s. The maximum speed-up of the parallel version is 11.38 (66 threads) and maximum efficiency of 27% (34 threads). Fig. 2 shows the execution time breakdown of useful work (dark-gray) and idle time (light-gray) for each stage Si. These results clearly point to the fifth stage (rank) as an implementation bottleneck. As the number of threads is increased, the performance of the parallel stages scales. However, for c between 8 and 16, the serial I/O input stage becomes the bottleneck.

![Figure 2. ferret: Execution time (seconds) breakdown](image)

2.2 Data compression: dedup

The dedup kernel implements data stream compression using the “de-duplication” method. The kernel is decomposed in five pipelines stages (see Fig. 3). The first and last stages are sequential and handle I/O. The intermediate
stages are parallel. 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 real streaming. The sequential code runs for 89.37s.

The two main differences in the dedup pipeline compared to ferret, are: i) one stage that generates work (more output items than input items) – FindAllAnchors; and ii) stage bypassing – not all stages process all items, e.g. Compress. There are additional implementation artifacts, such as a limit on the number of threads that share a queue in order to avoid contention, and additional queues between stages to buffer items. For dedup, the maximum speed-up is close to 5 for 14 threads. Fig. 4 shows that there is a high load imbalance between the stages, and the I/O bottleneck is reached on the output stage for \( c = 4 \).

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 [17]. 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 [17]) library parallel pipeline template provides support for work stealing [6]. We re-implemented ferret and dedup using TBB to quantify the performance of the work stealing parallel pipeline. We introduce next some of the abstractions used in this implementation. In TBB, programmers express concurrency in terms of tasks. Tasks are lighter weight than threads, thus allowing fast work switching. In the TBB pipeline template, the C++ classes pipeline and filter represent the pipeline container and its stages, respectively. 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 specifies the maximum number of tokens in flight and it is used to manage resource contention by throttling the arrival rate of items in the pipeline.

The linear pipeline in ferret maps directly to the TBB pipeline template. On the other hand, dedup, with its item producing and bypassing stages, can not be directly mapped. Therefore, we selected a combined Pthreads and TBB implementation. To work around the TBB constraint that the number of tokens entering/leaving the stage is constant, we created a separate queue for the output stage and assigned a dedicated pthread to handle the output. To handle the bypassing limitation of the Compress stage as well as the equal number of tokens limitation of the FindAllAnchors stage, we collapsed them into one parallel TBB pipeline filter.

2.3 Scalability in Pipeline Parallelism

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

Load imbalance is a result of different amounts of work per item in each stage. When a programmer fixes the number of threads per stage arbitrarily (it is a common practice to use the same number of threads for all stages), the stage with largest 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 share 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:

- **Oversubscription** – the application creates more threads than available processors and relies on the OS scheduling policies to keep all the processors busy.

2.3 Scalability in Pipeline Parallelism

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

Load imbalance is a result of different amounts of work per item in each stage. When a programmer fixes the number of threads per stage arbitrarily (it is a common practice to use the same number of threads for all stages), the stage with largest 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 share 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:

- **Oversubscription** – the application creates more threads than available processors and relies on the OS scheduling policies to keep all the processors busy.
to determine pipeline queue sizes, the optimal number of stages, optimal number of tokens in the system, and the optimal number of parallel threads to achieve high efficiency and scalability. Manually setting these parameters typically requires a multitude of runs in different configurations to understand all the trade-offs and interactions between them. Due to space constraints we just sketch the description of our models. The interested reader can find more details in [14].

We use the concept of logical queue to explicitly represent: i) the buffer where pending items are waiting to be processed, and ii) the servers (or threads) that process the items. In some cases (e.g., closed systems), the queueing system also includes a representation of the population that can ask for a service. Some parameters, such as service time $T_{ser}$ and inter-arrival time $T_{arr}$, are obtained from the sequential codes.

\[
\lambda_e = \begin{cases} 
\min(\lambda_i) & \text{Pthreads (Cases 1 & 2)} \\
\sum_{i \in TBB} \lambda_i & \text{TBB (Case 3)} 
\end{cases}
\]  
\[
\text{Time} = \frac{1}{\lambda_e} \cdot \text{nit}
\]  

Eqs. 1 and 2 are the main metrics computed for our models. $\lambda_e$ is the effective throughput of the whole pipeline: it depends on the throughput $\lambda_i$ of each internal queue, and it is particular to the modeled template. $\text{Time}$ – the estimated execution time, is our primary performance metric, and it is computed using Little’s law, based on the effective throughput $\lambda_e$ and the total number of processed items $\text{nit}$. We use measurements on the HP9000 Superdome machine configuration described above to verify that the analytical models correctly predict the execution time.

### 3.1 Case 1. Parallel pipeline closed system

In the original PARSEC implementations (using Pthreads), each stage $S_i$ in the parallel pipeline consists of an input queue buffer $Q_i$ and a pool of dedicated working threads $c_i$. We model this pipeline as a network of single logical queues [1]. The Pthreads implementations of ferret have queue buffers with limited capacity, causing items to stall in the pipeline stages, as shown in Fig. 2(b). Therefore we model it as a closed system: items can be viewed as circulating continuously and never leaving the network of queues because a new item can not enter until a previous one leaves. Fig. 5(a) shows a graphical representation of a closed system, depicting all the stages in the pipeline. In this model, we assume that there is a finite-calling population of $K$ items.

In steady state, each stage $S_i$ behaves like a $M/M/c_i/N/K$ queue system. This represents a logical queue with exponential inter-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 $K = N$ items [16]. Fig. 5(b) shows the abstract view of one stage. For each stage $S_i$, the mean service time, $T_{ser}$, is the time to process an item on that particular stage in the sequential execution. The mean inter-arrival time is the same for all stages: $T_{arr}$, and it is determined by the longest stage in the pipeline, i.e. $T_{arr} = \max(T_{ser})$.

In the closed model, Eq. 3, represents another statement of Little’s law: it computes the internal throughput for each stage, $\lambda_i$, based on i) the number of items processed on the stage per unit time, where $P_n(i)$ is the probability that there are $n$ items in the stage $S_i$; and ii) the time $T_{arr}$ (see [1]).

\[
\lambda_i = \sum_{n=0}^{N} (N - n) \cdot \frac{P_n(i)}{T_{arr}}
\]

**Figure 5.** (a) Closed system; (b) Closed system stage.

To apply this model to ferret, we start with the 6 stages Pthreads pipeline (see Fig. 1). We model one thread per each serial stage, $S_1$ and $S_6$: $c_1 = c_6 = 1$. For the parallel stages $S_{i=2..5}$, the number of threads assigned to each stage is $c_i = npTh/npstages$, where $npTh$ and $npstages$ represent, the number of parallel threads, and the number of parallel pipeline stages, respectively. In this version, $npstages = 4$. Using Eq. 3, we compute the internal throughput $\lambda_i$ for each stage. The effective throughput for the whole pipeline, $\lambda_e^{6T}$, and the execution time for a run $Time^{6T}$, are computed using Eq. 1-Case 1 and Eq. 2, where $\text{nit}$ is the number of items to be processed.

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, $c_1' = c_2' = 1$, whereas $c_2' = npTh$ is the number of parallel threads in the collapsed stage, because now $npstages = 1$. Using Eq. 3 we compute the internal throughput for each stage, then the effective throughput for the whole pipeline, $\lambda_e^{4T}$ (Eq. 1-Case 1), and the running time for the implementation $Time^{4T}$ using Eq. 2.

Fig. 6(a) compares the analytical times obtained for $Time^{6T}$ and $Time^{4T}$ for different number of parallel threads (the $npTh$ parameter) and for $N = Queue\_size = 20$ (default value in the original implementation). We also represent the measured times for the two Pthreads implementations of ferret.

We observe that the analytical times accurately predict the measured times. In Section 3.4 we discuss how to use the model to predict queues sizes and the optimum number of threads to achieve good scalability. As shown by
3.2 Case 2. Parallel pipeline open system

Another type of pipeline is the open system: the items flow among stages with different internal throughputs. Fig. 7 shows a view of this model. Assuming exponential inter-arrival and service times, each stage $S_i$ of this pipeline behaves like an $M/M/c_i$ queue system. This is the model typically found in the literature for pipelines [15, 13]. The Pthreads PARSEC implementation of dedup matches this model. The code has logical queues per stage, but these are sized to 1 million entries, providing the system with essentially infinite capacity ($N = \infty$). Indeed, we did not observe any stalling in the pipeline.

![Figure 7. Model of a parallel pipeline open system.](image)

In an open network, the inter-arrival time, $T_{arr_i}$, is different for each stage $S_i$. We compute it by composing the throughput from the stages in the network that reach $S_i$, Eq. (4) gives us the internal throughput for each stage. It depends on the inter-arrival time for the corresponding stage.

$$\lambda_i = \frac{1}{T_{arr_i}}$$  \hspace{1cm} (4)

Now, we analyze the original dedup Pthreads version with 5 stages. Again, one thread is assigned to each serial stage, i.e. $c_1 = c_5 = 1$, whereas for the parallel stages $S_{i, i=2-4}$, the number of threads per stage is $c_i = npTh/npstages$. In this case, $npstages = 3$. The effective throughput for the whole pipeline $\lambda_{eff}^{5st}$, is determined by the slower stage as Eq. 1-Case 2 indicates, and the execution time for the run $Time^{5st}$, is computed by Eq. 2. We also model the 3 stages Pthreads version, in which we collapse the parallel stages. We assign one thread to each of the serial $S_1$ and $S_5$ stages, and $c_2' = npTh$ parallel threads to the collapsed stage $S_2$, because $npstages = 1$. We compute the internal throughput for each stage, the effective throughput for the whole pipeline $\lambda_{eff}^{5st}$ (by Eq. 1-Case 2), and the running time $Time^{5st}$ (by Eq. 2).

In Fig. 6(b) we compare the analytical $Time^{5st}$ and $Time^{3st}$ times vs. the measured times for the two Pthreads implementations of dedup, for different number of parallel threads. In this case, the analytical times 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: the size of the output file may be larger as 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. The analytical model will give hints about how the parameters and resources interact.

We model the TBB implementation as a set of loosely coupled systems working in parallel, rather than a network of logical queues, because there is a logical queue of tasks per each thread, as opposed to the Pthreads implementations with a logical queue per pipeline stage. Each thread is the server of its local queue, where the items (the tasks) to be processed arrive. All 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 its 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 an $M/M/c/N/K$ queue system, where the population that can call for a service is finite and equal to the queue capacity, i.e. $K = N$ items (see Fig. 8(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 $T_{serv_{TBB}}$, depends on the whole processing
of an item. The inter-arrival time $T_{arr_{TBB}}$, also depends on
the processing of one item, i.e. $T_{arr_{TBB}} = T_{arr_{TBB}}$. The
number of servers per queue is initially $c = 1$ (1 thread).
The total system capacity is finite and is given by the num-
ber of tasks that can exist simultaneously in the system:

$ntokens$ (a parameter of the TBB pipeline template that
is 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. $N = ntokens/npTh$, be-
ing $npTh$ the number of parallel threads. Thus, $N$ repre-
sents the number of tasks (or tokens) per thread.

The concept of work stealing is similar to that of load
sharing. An analytical model of load sharing for a dis-
tributed closed system was proposed in [19]. 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 (work stealing) on the performance of a queue
is modeled as an open queue system with the same aggregate
number of tasks. Thus, $N$ represents the number of tasks (or
tokens) per thread.

Step 2. The stealing of tasks from other threads can be
modeled as a closed $M/M/c/N/K$ queue, i.e. each thread behaves as if a (virtual)
working capacity of $v$ additional threads are added, as
shown in Fig. 8(c). Now, we re-compute the internal
throughput $\lambda^*_j$ (see Eq. 3), taking into account these
new working capacity.

We compute the effective throughput of the whole sys-
tem $\lambda^T_{c,TBB}$, by aggregating $\lambda^*_j$ from all the threads, as indicated
by Eq. 1-Case 3.

As we saw, the serial stages eventually become a bottle-
neck. Therefore, we add Eq. 7 as a new constraint to model
this behavior:

$$\lambda^T_{c,TBB} = \frac{1}{\max(T_{in}, T_{out})}. \text{If } \max(T_{in}, T_{out}) > \frac{1}{\sum_{j=1}^{K} \lambda^*_j}$$

where $T_{in}$ and $T_{out}$ are the times to process a item in
the input and output filters, respectively. Finally, the execution
time $Time_{TBB}$, is computed by Eq. 2.

In Fig. 9 we compare the analytical $Time_{TBB}$ (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. We compare
the times for different values of $ntokens$: 20 and $6 \times npTh$.
Recall that the $ntokens$ variable controls the number of
tasks that exists concurrently at any given point, represent-
ing the total system capacity.

For comparison, we also include the times obtained for
an ideal centralized system based on a global
$M/M/npTh/ntokens$ logical queue, which we
name “Ideal”. This is a boundary case that represents a cen-
tralized queuing system with the same aggregate number of
threads and task capacity that our queue distributed imple-
mentation in TBB. This centralized system represents the
optimal case in queuing theory [19], and serve as a base-
line to measure the optimality of the work stealing
implementation.

Next, we derive $v$ which is the average number of ac-
tive services in the $M/M/n_j/n_j$ remote queue that
represents the average number of effectively stolen
tasks.

Step 3. When we incorporate work stealing in our model,
each thread is now modeled as a closed $M/M/c +
v/N/N$ queue, i.e. each thread behaves as if (a virtual)
working capacity of $v$ additional threads are added, as
shown in Fig. 8(c). Now, we re-compute the internal
throughput $\lambda^*_j$ (see Eq. 3), taking into account these
new working capacity.

We compute the effective throughput of the whole sys-
tem $\lambda^T_{c,TBB}$, by aggregating $\lambda^*_j$ from all the threads, as indicated
by Eq. 1-Case 3.

As we saw, the serial stages eventually become a bottle-
neck. Therefore, we add Eq. 7 as a new constraint to model
this behavior:

$$\lambda^T_{c,TBB} = \frac{1}{\max(T_{in}, T_{out})}. \text{If } \max(T_{in}, T_{out}) > \frac{1}{\sum_{j=1}^{K} \lambda^*_j}$$

where $T_{in}$ and $T_{out}$ are the times to process a item in
the input and output filters, respectively. Finally, the execution
time $Time_{TBB}$, is computed by Eq. 2.

In Fig. 9 we compare the analytical $Time_{TBB}$ (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. We compare
the times for different values of $ntokens$: 20 and $6 \times npTh$.
Recall that the $ntokens$ variable controls the number of
tasks that exists concurrently at any given point, represent-
ing the total system capacity.

For comparison, we also include the times obtained for
an ideal centralized system based on a global
$M/M/npTh/ntokens$ logical queue, which we
name “Ideal”. This is a boundary case that represents a cen-
tralized queuing system with the same aggregate number of
threads and task capacity that our queue distributed imple-
mentation in TBB. This centralized system represents the
optimal case in queuing theory [19], and serve as a base-
line to measure the optimality of the work stealing
implementation.
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 is a trade-off parameter that has impact on the execution time, especially for a large number of threads when the ntokens is small. This indicates an underutilized system where there are no load balancing opportunities. For large number of tokens, the TBB implementation behaves optimally. The analytical model provides some insight: when ntokens is sufficiently large, the work 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 there is a trade-off in selecting the value of ntokens: while large values improve thread utilization by exposing work-stealing opportunities, they also may incur additional overheads due to task management, synchronization and contention. These can potentially eliminate the benefits of the dynamic scheduling [6]. In Section 3.4 we demonstrate how to use the model to compute the optimal ntokens to get the whole pipeline system behaves optimally.

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 code. The results and conclusions are similar to the ferret case.

### 3.4 Sizing the System

In this section we use the analytical model to determine: i) the optimum values for the capacity of the closed model, N (Queue size in Pthreads and ntokens in TBB) that minimizes the storage requirements and overheads of each implementation; and ii) the optimum number of threads, cope, that achieves the maximum effective throughput.

In the closed models, the system capacity N (Queue size in Pthreads and ntokens in TBB), is optimized using Eq. 3: as N increases, the internal throughput increases until it reaches an upper bound. Ideally, assuming a high utilization $\rho_i \approx 1$, from Eq. 5, we determine the upper bound $\lambda_{upper,i}$ for the internal throughputs as:

$$\lambda_{upper,i} \approx \frac{c_i}{T_{ser,i}}$$

(8)

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 $\lambda_{max}$ be the throughput due to the critical serial stage. Let $T_{in}$ and $T_{out}$ be the service times to process an item in the serial input and output stages. From Eq. 5, assuming $\rho_i \approx 1$, and as the serial stages have $c_i = 1$ thread, we can find an optimistic upper bound for $\lambda_{max}$,

$$\lambda_{max} = \min\left(\frac{1}{T_{in}}, \frac{1}{T_{out}}\right)$$

(9)

Therefore, an upper bound of the effective throughput is

$$\lambda_{eu} = \min(\lambda_c, \lambda_{max})$$

(10)

Once computed $\lambda_{eu}$, we can use Eqs. 3, 1-Case 1, and 1-Case 3, to find the value of $N_{optimal}$ which verifies $\lambda_c(N_{optimal}) = \lambda_{eu}$. For the Pthreads implementation of the ferret code, $N_{optimal}$ is around 21 items per queue in the 6 stages case, and 24 items per queue in the 3 stages one. Thus, the queue buffers will consume a total storage requirement of $N_{optimal} \times (npstages + 1)$ items. In the TBB implementations, the $N_{optimal}$ value for ferret is around 5 tokens per thread and around 3 tokens per thread for dedup. We deduce that the optimal total capacity is $ntokens = 5 \times npTh$ and $ntokens = 3 \times npTh$, respectively, which represent the storage requirements for these implementations.

The implementations will scale with the number of threads, until the effective throughput reaches $\lambda_{max}$, i.e. until they reach the I/O bottleneck. Another interesting application of the model is, given a certain system capacity, to find the optimal number of threads $c_{optimal}$ for which the effective throughput reaches $\lambda_{max}$. From Eqs. 3, 1-Case 1, and 1-Case 3, we compute the value of $c_{optimal}$ which verifies $\lambda_c(c_{optimal}) = \lambda_{max}$. For the ferret Pthreads implementations with $N = 20$, we find $c_{optimal} = 15$ parallel threads for the collapsed 3 stages implementation, and $c_{optimal} = 12$ threads per parallel stage for the 6 stages code, which give us a total of $12 \times 4 = 48$ parallel threads for this last code. In the ferret TBB implementation we have 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 (Case 2), i.e. the Pthreads dedup codes, the upper bound of the internal throughput is given by Eqs. 5 and 4: $\lambda_{upper,i} \approx c_i/T_{arr,i}$. In this case, the I/O bottleneck will be found when $\lambda_c = \min(\lambda_{upper,i}) = \lambda_{max}$.
Table 1. Non-oversubscribed vs. oversubscribed comparison for the 6 stages Pthreads ferret code.

<table>
<thead>
<tr>
<th># PE/Th</th>
<th>Time</th>
<th>Sp.</th>
<th>Eff. (%)</th>
<th># PE/Th</th>
<th>Time</th>
<th>Sp.</th>
<th>Eff. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6/6</td>
<td>390.6</td>
<td>1.2</td>
<td>19.7%</td>
<td>1/6</td>
<td>438.4</td>
<td>1.0</td>
<td>99.7%</td>
</tr>
<tr>
<td>10/10</td>
<td>186.6</td>
<td>2.3</td>
<td>23.9%</td>
<td>2/10</td>
<td>220.8</td>
<td>1.9</td>
<td>98.9%</td>
</tr>
<tr>
<td>18/18</td>
<td>94.8</td>
<td>4.6</td>
<td>25.5%</td>
<td>3/18</td>
<td>111.8</td>
<td>3.9</td>
<td>97.7%</td>
</tr>
<tr>
<td>32/32</td>
<td>41.9</td>
<td>9.1</td>
<td>25.6%</td>
<td>4/32</td>
<td>62.5</td>
<td>6.3</td>
<td>99.9%</td>
</tr>
<tr>
<td>64/64</td>
<td>38.4</td>
<td>11.3</td>
<td>17.2%</td>
<td>16/64</td>
<td>48.8</td>
<td>8.9</td>
<td>55.9%</td>
</tr>
</tbody>
</table>

Table 2. 6 stages vs. 3 stages comparison for ferret-TBB. Time is in seconds.

<table>
<thead>
<tr>
<th># PE</th>
<th>Time</th>
<th>Sdup</th>
<th>Eff. (%)</th>
<th>Time</th>
<th>Sdup</th>
<th>Eff. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>437.65</td>
<td>1.00</td>
<td>99.85%</td>
<td>442.88</td>
<td>0.99</td>
<td>98.67%</td>
</tr>
<tr>
<td>2</td>
<td>221.13</td>
<td>1.98</td>
<td>98.81%</td>
<td>223.5</td>
<td>1.96</td>
<td>97.76%</td>
</tr>
<tr>
<td>4</td>
<td>112.59</td>
<td>3.88</td>
<td>97.03%</td>
<td>113.58</td>
<td>3.85</td>
<td>96.19%</td>
</tr>
<tr>
<td>8</td>
<td>58.5</td>
<td>7.47</td>
<td>93.38%</td>
<td>59.56</td>
<td>7.34</td>
<td>91.71%</td>
</tr>
<tr>
<td>16</td>
<td>35.87</td>
<td>12.18</td>
<td>76.14%</td>
<td>36.98</td>
<td>11.82</td>
<td>73.86%</td>
</tr>
<tr>
<td>32</td>
<td>33.14</td>
<td>13.19</td>
<td>41.21%</td>
<td>33.42</td>
<td>13.08</td>
<td>40.86%</td>
</tr>
<tr>
<td>64</td>
<td>35.66</td>
<td>12.75</td>
<td>19.15%</td>
<td>36.44</td>
<td>11.99</td>
<td>18.74%</td>
</tr>
</tbody>
</table>

4 Evaluation

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. We use the same methodology and target platform that we described in section 2 for our experiments.

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. 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 (Time), speed-up (Sp.), relative to the baseline sequential time, and efficiency (Eff.) for the non-oversubscribed and the oversubscribed 6 stages Pthreads versions. The oversubscribed version clearly outperforms the non-oversubscribed version, given that the same number of cores (PE) can run a much higher number of threads (Th) by better utilizing the hardware resources.

The second optimization 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 determined previously by our analytical model. For ferret, we implemented two TBB versions: 6 stages and 3 stages, both with ntokens = 5 × 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 issue of the original 6 stages pipeline is essentially removed by dynamic scheduling, and the two versions have similar behavior profiles across all threads configurations.

In Fig. 10 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.

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 × npTh. Fig. 10(b) 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 function that implements work stealing) is negligible and 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
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 stages implementation (i.e. the load stage 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 stages 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. Fig. 11(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.

![Figure 11.](image)

Figure 11. (a) Total execution time in 4 stages ferret; and (b) Improvement of 4 stages ferret / TBB.

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 Fig. 11(b), the TBB pipeline outperforms the Pthreads pipeline by 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. [15] exploits a technique called Parallel-Stage Decoupled Software Pipelining (PS-DSP). This technique identifies the pipeline stages (with small programmer interventions), as well as partitions the threads between the parallel stages. Thies et al. propose in [20] a set of simple annotations to let the programmer mark the pipeline stages and use dynamic analysis to track the data communications between stages. In these works, the selection of threads for the stages is static. Both approaches suffer from load imbalance, thus degrading the scalability of the applications. Other approaches for coding streaming applications is to use programming languages with support for streams. Examples include: StreamC [8], Brook [5], and StreamIt [18]. 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.

Liao et al. [13] propose a model to study the performance of a parallel pipeline system. They model 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 [21] 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 measured 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) Programmer can code their algorithms 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 or specifying stages that produce a different number of outputs. Finally, regarding the I/O bottleneck we explored the possibility of improving the performances by allowing parallel I/O. We point out that once load balancing is resolved, feeding and clearing the pipeline becomes the next bottleneck and will become a cornerstone to achieve scalability on multi-core systems for RMS and streaming applications.

Acknowledgments

This work was supported by Grants JC2008-00178 and PR2008-0194 by the MCIN of Spain, by contracts TIN2006-01078 and Juan de la Cierva by the MEC of Spain and partially supported by TIC-3500 (J. de Andalucía). We also thank Rafael Larrosa, system administrator of the Bioinformatic and Supercomputing Center of Málaga, for his help with the HP9000 Superdome SMP.

References