

# Expressing Pipeline Parallelism Using TBB Constructs

## A Case Study on What Works and What Doesn't

Eric C. Reed

Rose-Hulman Institute of Technology  
reedec@rose-hulman.edu

Nicholas Chen    Ralph E. Johnson

University of Illinois at Urbana-Champaign  
{nchen,rjohnson}@illinois.edu

### Abstract

Task-based libraries such as Intel's Threading Building Blocks (TBB) provide higher levels of abstraction than threads for parallel programming. Work remains, however, to determine how straightforward it is to use these libraries to express various patterns of parallelism. This case study focuses on a particular pattern: *pipeline parallelism*. We attempted to transform three representative pipeline applications — content-based image retrieval, compression and video encoding — to use the pipeline constructs in TBB. We successfully converted two of the three applications. In the successful cases we discuss our transformation process and contrast the expressivity and performance of our implementations to existing Pthreads versions; in the unsuccessful case, we detail what the challenges were and propose possible solutions.

**Categories and Subject Descriptors** D.2.11 [Software Architectures]: Patterns; D.3.3 [Language Constructs and Features]: Patterns

**General Terms** Design, Measurement, Performance

**Keywords** Pipeline Parallelism, Threading Building Blocks, Pthreads, Pattern

### 1. Introduction

Pipeline parallelism [13] is an increasingly popular parallel programming pattern for emerging applications. Many stream applications in the domain of digital signal processing, graphics, and encryption are naturally expressed using the pipeline parallelism pattern.

Typically, these applications have been parallelized using the low-level constructs of a threading library, such as Pthreads, which could have deleterious consequences on un-

derstandability, maintainability and performance (scaling, load balancing). Researchers and practitioners agree that programming with threads is notoriously difficult and error prone [9]. There have been several attempts at providing higher levels of abstraction than threads for expressing pipeline parallelism. Some of these efforts rely on specialized programming languages: Brook [5], StreamIt [19] and StreamC/KernelC [8]. However, legacy code abounds making it challenging to adopt a new language. Thus, task-based libraries such as TBB [1] and Microsoft's Task Parallel Library (TPL) in .NET [10] offer attractive alternatives for legacy code. In addition to general task constructs, TBB also includes specialized pipeline constructs (Section 4) to make it easier to express pipeline parallelism. TPL Dataflow[2] (to be included in .NET 4.5) supports specialized dataflow constructs for expressing general producer/consumer relationships including pipeline parallelism<sup>1</sup>.

The question remains: can we use these libraries to express the kinds of pipeline parallelism that exist in programs parallelized using Pthreads? If so, then these libraries are attractive alternatives to Pthreads. On the other hand, if these libraries are insufficient, then we need to ask *what's missing?* This case study is a step in answering those questions. We document the challenges and lessons learned from converting three representative pipeline applications to use the pipeline constructs in TBB. Though we have used TBB, the general lessons learned are not limited to it; parts of the solutions are specific to TBB but the challenges documented serve as valuable lessons for both library developers who are considering what constructs to build and also application programmers who are deciding what constructs to use.

In our case study, each application had a corresponding version parallelized using Pthreads. We closely mimicked the parallelism strategies used in the Pthreads versions to provide a fair and useful comparison. We successfully transformed the content-based image retrieval (*ferret*) and compression (*dedup*) applications but had difficulties with

Copyright is held by the author/owner(s). This paper was published in the proceedings of the Workshop on Transitioning to MultiCore (TMC) at the ACM Systems, Programming, Languages and Applications: Software for Humanity (SPLASH) Conference, October, 2011, Portland, OR, USA.

<sup>1</sup>TBB 4.0 introduced *flow graph* constructs, similar to those of TPL Dataflow, that support general producer/consumer relationship. At the time of writing, the flow graph was still a community preview feature and not part of the official TBB release.

Application	Challenges	Solution
<i>Ferret</i>	Recursive pipeline stage	Replace recursion with a stack
<i>Dedup</i>	Single input, multiple outputs	Nested pipelines
	Stage bypassing	Enforce single path
<i>x264</i>	Backward and forward dependencies	<i>Not expressible using TBB pipeline</i>

**Table 1.** Main challenges in transformation process

the video encoder (*x264*). Below, we briefly summarize the lessons learned; Section 5 goes into greater details.

**Expressivity** The initial transformation of the sequential code to TBB pipelines took the most time because we had to manually resolve and analyze the dependencies between stages. Because TBB’s pipeline also enforces certain restrictions on structure and control flow, we had to apply some non-obvious transformations. It is likely that developers would have to apply similar transformations while working on complex sequential code. Table 1 summarizes those transformations; Section 5 discusses them in detail.

After performing the transformations, we see a reduction in the amount of boilerplate code (setting up mutexes, semaphores, etc) that needs to be written compared to the Pthreads counterparts. Overall, the stages of the pipeline were made more explicit and it was easier to add new pipeline stages.

**Performance** The converted *ferret* program performed on par with its Pthreads counterpart in terms of running time. The converted *dedup* application ran up to 2.13 times faster than its counterpart after the conversion. In both converted applications, the memory requirements were comparable. Overall, when it was possible to successfully convert the applications, TBB presents an comparable alternative to Pthreads in terms of performance while also increasing maintainability.

## 2. Methodology

The three applications we studied were from the Princeton Application Repository for Shared-Memory Computers (PARSEC) [4], a benchmark suite for shared-memory multithreaded programs. PARSEC is unique because it is application-driven; it aims to capture *emerging workloads* that are missing from typical high-performance computing benchmarks. PARSEC already includes parallel versions of its applications parallelized using Pthreads. Thus, PARSEC is an ideal research testbed to answer the following research questions:

**Expressivity** Are the pipeline constructs in TBB 3.0 sufficient to *faithfully* express the patterns of pipeline parallelism present in the Pthreads versions of the applications? Does using these constructs improve the understandability and maintainability of the application?

**Performance** What are the performance impacts, if any, from transforming the applications to use the new TBB pipeline constructs? Are these performance impacts severe enough to deter developers from using TBB?

We measured the performance of each benchmark (original Pthreads version and TBB version) on a machine with four Intel Xeon L755 (1.87GHz) processors with 64GB of memory. Each processor has eight cores capable of simultaneous multi-threading with two threads each. In total, the machine is capable of up to 64 hardware threads. The operating system is CentOS release 5.5 running the 2.6.18-194.26 Linux Kernel. We used both GCC 4.1.2 and Intel ICC 11.1.

PARSEC provides the *parsecmgmt* tool for building and running its applications. Additionally, it also provides several input sets for each application. To make it possible for others to repeat our experiments, we made our modifications compliant with *parsecmgmt* and use it to run performance benchmarks using the *native* input set, the largest input set that closely resembles the typical input sets for each application. Our modifications are available from <http://vazexqi.github.com/ParsecPipelineParallelism>.

## 3. Related Work

Navarro et al. [14] also used *ferret* and *dedup* as examples of pipeline parallelism. Their work focused on modeling the performance differences between the Pthreads and TBB versions by creating analytical models of parallel pipelines based on queueing theory. They concluded that due to efficient work-stealing, the TBB versions outperform the Pthreads version even without optimizations such as *oversubscription* and *stage collapsing*. We, on the other hand, focused on the transformation process and tried to faithfully express the Pthreads versions in our TBB versions. When Navarro et al. encountered the problem of supporting multiple tokens in *dedup* (Section 5.2) they used a combination of Pthreads and TBB; we used nested pipelines instead and expressed everything cleanly with TBB.

Thies et al. [18] first proposed an annotation-based method for automatically detecting and parallelizing pipeline parallelism in C programs. Rul et al. [16] improved on their work and automatically detect and parallelize pipeline parallelism in applications without any annotations. Both tools take several hours to run and consume huge amounts of memory even on small programs They detect limited templates of pipeline parallelism based on data access. The transformed program is in binary form, making it hard to examine the parallelized program. Nonetheless, these tools are useful starting points for suggesting where pipeline parallelism exists;

ultimately the developer would use higher-level constructs such as TBB’s to make the parallelism explicit in source code.

Thies et al. [17] characterized the behavior of 65 stream programs; some of which employed pipeline parallelism. They addressed three main aspects: barriers to parallelization, scheduling characteristics and programming styles. While the programs were written using the StreamIt language, their findings are useful for designing future languages and libraries. It would be worthwhile to create a catalog and characterize the behaviors of stream programs written in general purpose languages such as C/C++ to see how they compare.

MacDonald et al. [12] proposed the master-slave scheduling technique that TBB uses as its execution model for pipelines. They implemented their technique in the pattern-based parallel programming system *CO<sub>2</sub>P<sub>3</sub>S*. The *CO<sub>2</sub>P<sub>3</sub>S* system enabled a top-down approach to developing parallel programs: a developer selected the parallel programming patterns (pipeline parallelism being one of the selections), configured the parameters and the system generates the skeleton code (Java) for the different components. Because this system requires programs to be designed from scratch, it is not possible to use it on legacy code.

#### 4. The TBB Pipeline Construct



Figure 1. An example of a three-stage pipeline

In TBB, a pipeline is composed of a series of filters. Each filter takes an input token, processes it and produces an output token. The first filter in the pipeline does not require an input token; similarly the last filter does not produce an output token. To implement the pipeline shown in Figure 1, requires creating three `Filter` objects and composing them together in a `Pipeline` object as shown in Figure 2. `Filter` objects can be *serial* or *parallel*. Only one token can be working at a time in a serial filter – this enforces a way to process tokens in order. Multiple tokens can be working at a time in a parallel filter – this provides a way to execute tokens in an out-of-order manner in parallel to improve throughput. Tokens in different filters may run simultaneously.

To transform an existing program to use pipeline parallelism in TBB requires three steps: (i) identify the stages of a pipeline and convert them into serial or parallel `Filter` objects; (ii) identify the tokens that pass through each `Filter` object and override `operator()` to process them; (iii) construct a `Pipeline` object and call its `run()` method. The pipeline constructs take care of most of the bookkeeping that happens underneath. In contrast, implementing a Pthreads version would require two *additional* boilerplate steps: (i)

```

1 #include "tbb/pipeline.h"
2
3 class Filter1: public tbb::filter {
4     // generate tokens
5     void* operator()(void* token);
6 };
7 class Filter2: public tbb::filter {
8     // process tokens and output tokens
9     void* operator()(void* token);
10 };
11 class Filter3: public tbb::filter {
12     // process tokens
13     void* operator()(void* token);
14 };
15
16 // Create the pipeline
17 tbb::pipeline ThreeStagePipeline;
18 ThreeStagePipeline.add_filter(new Filter1());
19 ThreeStagePipeline.add_filter(new Filter2());
20 ThreeStagePipeline.add_filter(new Filter3());
21 // Run the pipeline
22 ThreeStagePipeline.run();

```

Figure 2. Expressing the pipeline in Figure 1 using TBB

create a `BlockingQueue` in between each filter to hold tokens that might arrive earlier due to load imbalance between stages (ii) wrap each token with a sequence number in the event to identify which tokens to process in order when necessary.

TBB’s pipeline execution model is based on MacDonald’s work [12]. This tasking model alleviates the ramp-up/ramp-down problem as the pipelines starts/ends while also providing better load-balancing by allocating more tasks to different filters dynamically. TBB’s pipeline also preferentially *carries* a token as deep into the pipeline as possible before switching to a different task; this improves memory performance as the token is more likely to remain in cache for each filter. Since these optimizations are built into the constructs, the developer is freed from having to manage any of these issues. In contrast, developers using Pthreads might have to implement these optimizations by hand to improve performance.

This section described the advantages that TBB offer in terms of reducing boilerplate code and its built-in mechanisms for improving performance. The next section examines whether these built-in constructs and mechanisms actually help or hinder developers as they try to express parallelism in the PARSEC applications.

#### 5. The Applications

*Ferret*, *dedup* and *x264* exhibit pipeline parallelism in different forms and provide a good sample of the cases that developers might encounter. We describe the applications in increasing order of complexity.

##### 5.1 Ferret



Figure 3. Six-stage pipeline of *ferret*

*Ferret* (10,765 SLOC) is a content-based image search application [11]. Given an input image, it segments the image, extracts relevant features, queries the database for candidate images, ranks the candidates based on similarity and outputs the results. These six stages are shown in Figure 3. The input and output stages are serial; the four middle stages can run in parallel.

The Pthreads version uses *oversubscription*: specifying the program to run with  $x$  threads would create  $x$  threads for each of the parallel stages. *BlockingQueues* configured for a maximum of 20 items were used to pass tokens between stages. The files *ferret-parallel.c* (437 SLOC) and *tpool.c* (92 SLOC) set up and coordinated the parallelism using Pthreads.

Mapping *ferret* to TBB’s pipeline was relatively straightforward: each stage was transformed into a `Filter` object and marked as *serial* or *parallel*. The main challenge was the input stage: it used recursion to obtain a list of images from a root directory; TBB `Filter` objects are not permitted to recursively call themselves. We solved this by replacing recursion with a stack object. This problem with recursive calls seems common and would need to be handled in a similar manner by other developers for their own projects.

Our TBB implementation, *ferret-tbb.cpp* reduced the lines of code to 376 SLOC by eliminating the boilerplate code that needs to be written to set up the blocking queues and thread pools. It made each stage of the pipeline more explicit and facilitated adding new stages, as necessary. Figure 4 shows that the TBB versions performed on par with the Pthreads version for the *native* test input of 3,500 image queries. The scalability of pipelines is limited by the serial I/O stages; in *ferret*, performance does not scale beyond 20 threads.

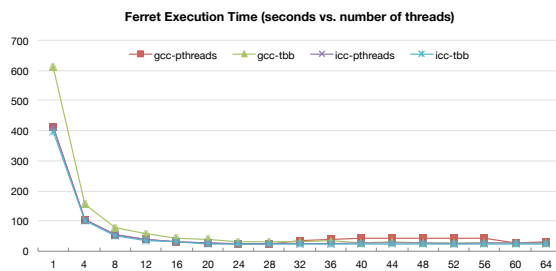


Figure 4. Execution time for *ferret*

## 5.2 Dedup

*Dedup* (5,968 SLOC) is a compression kernel that uses the “de-duplication” method [4]. Given a data stream to compress, it splits the data into smaller *blocks*; splits the blocks into smaller *segments*; computes and checks the hash for each segment; compresses the segments, if necessary; organizes the segments and blocks in their proper order; and,

finally, writes the compressed stream. Figure 5(a) shows the configuration for the Pthreads implementation.

Two artifacts make *dedup* challenging to parallelize:

**Single input, multiple outputs** The `SplitBlocks` stage takes a block and splits it into smaller segments; it takes a *single* input token and produces *multiple* output tokens. However, `Filter` objects in TBB can only take a *single* input and produce a *single* output. To mimic the Pthreads version, we had to resort to *nested pipelines* as shown in Figure 5(b). The inner pipeline deals with segments while the outer pipeline deals with blocks. We needed to add the `ReassembleBlocks` stage to reassemble segments into blocks before passing the tokens to the outer pipeline as both pipelines operate on different types of data granularity. Figure 6 shows the code snippet for implementing nested pipelines. Lines 19-37 show the construction and execution of the inner pipeline.

```

1  /* THE OUTER PIPELINE CLASSES */
2  class SplitData : public tbb::filter {...};
3
4  class ProcessBlocks : public tbb::filter {
5
6  ...
7
8  protected:
9  /* THE INNER PIPELINE CLASSES */
10 class SplitBlocks : public tbb::filter {...};
11
12 class CheckHash : public tbb::filter {...};
13
14 class Compress : public tbb::filter {...};
15
16 class ReassembleBlocks : public tbb::filter {...};
17
18 public:
19 void* operator()(void* token) {
20
21     tbb::pipeline pipeline;
22     // Splits token into blocks
23     SplitBlocks split(token);
24     CheckHash check;
25     Compress compress;
26     ReassembleBlocks reassemble();
27
28     pipeline.add_filter(split);
29     pipeline.add_filter(check);
30     pipeline.add_filter(compress);
31     pipeline.add_filter(reassemble);
32
33     // Run the inner pipeline
34     pipeline.run();
35     pipeline.clear();
36
37     ...
38 };
39 };
40
41 class WriteOutput : public tbb::filter {...};

```

Figure 6. Nested pipelines in TBB

**Stage bypassing** The `CheckHash` stage can either proceed to the `Compress` or `WriteOutput` stage depending on its result. In TBB, it is not possible to bypass a stage; instead, all tokens will proceed through `Compress` stage. An additional flag would need to be added in the token to signal whether it needs to be compressed.

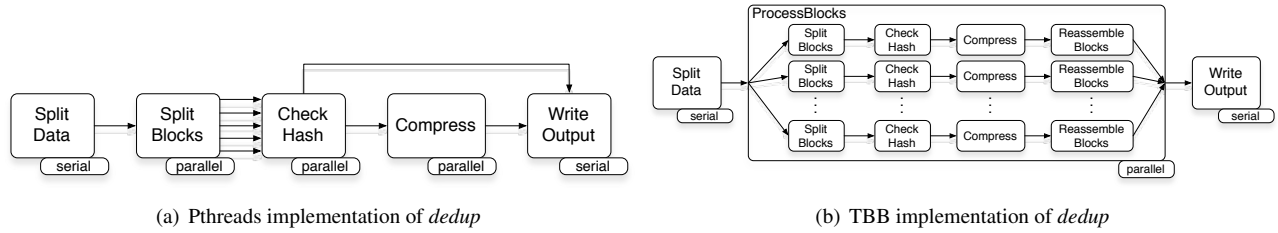


Figure 5. Dedup pipeline configurations

At first glance, having to use nested pipelines and not being able to bypass stages seem counterintuitive and detrimental to performance. Nested pipelines creates many more temporary objects in memory; not being able to by-pass stages requires redundant processing. However, our experiment shows that the increased parallelism more than compensates for the overhead. Figure 7 shows the execution times of both versions with the *native* test input of compressing an ISO file of 672 MB. The TBB version compiled using GCC consistently outperforms the other versions. The scalability of pipelines is limited by the serial I/O stages; in *dedup*, performance does not scale beyond 32 threads.

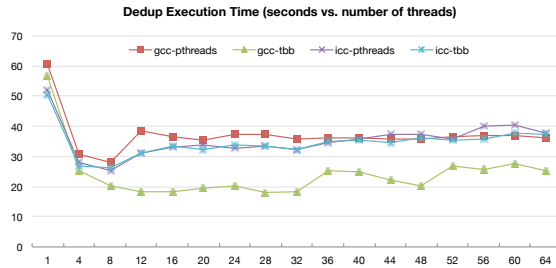


Figure 7. Execution time for *dedup*

Nested pipelines, however, required more code to express and could be more difficult to understand. In the Pthread version, the files *encoder.c* (148 SLOC) and *queue.c* (81 SLOC) set up and coordinated the parallelism. In the TBB version, a single file, *encoder-tbb* (418 SLOC) was used.

### 5.3 x264

*x264* (29,324 SLOC) is an HD video encoder for the H.264/MPEG-4 standard [3]. The encoder predicts the contents of a *frame* from previously encoded *reference frames*. A frame can reference frames that occur before or after itself in play order. Parts of frames, called *macroblocks*, do not necessarily use the same reference frames.

There are three types of macroblock: intra blocks (I-blocks), predicted blocks (P-blocks), and bipredicted blocked (B-blocks). *I-blocks* do not reference other frames. *P-blocks* reference only one frame. *B-blocks* reference a frame before and a frame after itself [15]. An *I-frame* consists entirely of I-blocks. A *P-frame* contains as least one P-block, but no

B-blocks. A *B-frame* contains at least one B-block. Circular frame dependencies are not allowed, so dependencies define a partial ordering on frames. Once a frame has been encoded, it is available for its dependents in a *global buffer*. Before a frame can be encoded, all of its dependencies must be in the buffer. Figure 8 shows a valid configuration of frames and the dependencies between frames.

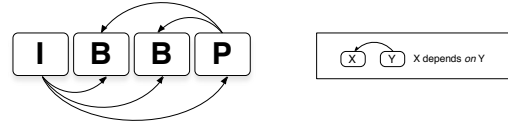


Figure 8. Dependencies between I, B and P frames in H.264

The *x264* implementation in PARSEC assigns a Pthread to each frame. Each frame has a *condition variable* associated with it, which is used to broadcast readiness to dependents. By waiting on the condition variables of all its dependencies, a frame ensures it will block until its dependencies are ready. Just before a frame enters the encoding process, its type (I, P, or B) and dependencies are decided in a way that avoids potential deadlocks. In effect, the pipeline is a dynamically constructed directed acyclic graph where each frame is a stage [4]. *x264* was originally categorized as pipeline parallelism in [4]; after examining the code, we believe that the categorization is *inaccurate*. *x264*, as implemented, exemplifies the wavefront pattern.

The difficulty in constructing a TBB pipeline implementation of *x264* is enforcing frame dependencies. TBB pipelines structures cannot be changed while running, so stages must be constructed from the tasks involved in encoding, unlike the Pthreads implementation. In a pure TBB implementation, if multiple frames are in the pipeline then there is no guarantee that frames near the end of the pipeline will complete before frames near the start require them. Restricting the pipeline so that frames are processed one at a time forces the guarantee, but prevents any parallelism.

A mixed TBB/Pthreads implementation could use Pthread condition variables in the same manner as the existing Pthread implementation. However, when a task in the pipeline waits on a condition variable the entire thread will block. This prevents the usual TBB automatic load balancing between threads and requires oversubscription, which TBB

tries to avoid, to achieve significant parallelism. In this implementation, TBB is little more than a wrapper around Pthreads that provides automatic queue management, but at a high runtime overhead due to the task scheduler.

Implementing *x264* in TBB is not impossible, but the TBB pipeline structure is not suitable. An implementation using the newly introduced TBB 4.0 flow graph interface [1], a wavefront [6], or pure TBB tasks would be more feasible. *x264* is a complex application with many approaches for parallelization; for a survey of the different partitioning strategies for parallelizing *x264*, refer to [7].

## 6. Discussion

Though we have focused on just three applications, the lessons learned have implications on the design of future automated software engineering tools. In particular, the conversion process of transforming sequential C++ code to pipeline stages is a necessary step when trying to use a task-based library. The conversion process is also the most tedious — dependencies have to be manually analyzed and the program has to be transformed in a way that is *maintainable*. However, the process of transforming sequential code to use a TBB construct is simpler than a transformation to use a Pthreads construct and presents an opportunity for tool automation. Tools that help identify these dependencies in an intuitive manner and tools that help with stepwise transformation of existing code would assist greatly in the parallelization process.

Given the peculiarities of each application that we have encountered, it seems unlikely that a single tool could ever fully automatically transform a complex sequential program to use pipeline parallelism. Pipeline parallelism is much more advanced than loop transformation and would require deeper analysis and more sophisticated tools. A more pragmatic solution, instead, would be to create a series of tools e.g. *interactive* refactoring tools that can assist the developer through the smaller steps of *preparing* a program for pipeline parallelism. Our future work would distill those smaller refactorings for pipeline parallelism and implement them in an IDE. Such an IDE could also provide additional refactorings that programmers might otherwise perform by hand such as *collapse stage* or *replicate stage* for improving the performance of pipelines. Work remains to mine a catalog of such refactorings that developers perform frequently.

## Acknowledgments

Eric Reed was supported by National Science Foundation Grant No. A2133. Nicholas Chen was supported by US Department of Energy Grant No. DOE DE-FG02-06ER25752. The computing resources used to generate the results in this paper were funded by the Universal Parallel Computing Research Center at the University of Illinois. The Center is sponsored by Intel Corporation and Microsoft Corporation. SLOC data was generated using David A. Wheeler's 'SLOCCount'.

## References

- [1] Intel Threading Building Blocks. <http://www.threadingbuildingblocks.org/>.
- [2] TPL Dataflow. <http://msdn.microsoft.com/en-us/devlabs/gg585582>.
- [3] *x264*. <http://www.videolan.org/developers/x264.html>.
- [4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, January 2008.
- [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*.
- [6] A. J. Dios, R. Asenjo, A. Navarro, F. Corbera, and E. L. Zapata. Wavefront template implementation based on the task programming model. Technical report, University of Malaga, 2011.
- [7] H. Hoffman, A. Agarwal, and S. Devadas. Partitioning Strategies: Spatiotemporal Patterns of Program Decomposition. In *ICPADS '10*.
- [8] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *Computer*, 36:54–62, August 2003.
- [9] E. A. Lee. The Problem with Threads. *Computer*, 39:33–42, May 2006.
- [10] D. Leijen, W. Schulte, and S. Burckhardt. The Design of a Task Parallel Library. In *OOPSLA '09*.
- [11] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Ferret: a toolkit for content-based similarity search of feature-rich data. In *EuroSys '06*.
- [12] S. MacDonald, D. Szafron, and J. Schaeffer. Rethinking the Pipeline as Object-oriented States with Transformations. In *HIPS '04*.
- [13] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004.
- [14] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval. Analytical Modeling of Pipeline Parallelism. In *PACT '09*.
- [15] I. E. Richardson. *The H.264 Advanced Video Compression Standard*. Wiley, 2010.
- [16] S. Rul, H. Vandierendonck, and K. De Bosschere. A profile-based tool for finding pipeline parallelism in sequential programs. *Parallel Comput.*, 36:531–551, September 2010.
- [17] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *PACT '10*.
- [18] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. In *MICRO '07*, .
- [19] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC '02*. Springer-Verlag, .