FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Erweiterung des Pheet Frameworks für Pipeline-Parallele Anwendungen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Bernhard Redl
Matrikelnummer 0828401

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Dr. Scient. Jesper Larsson Träff

Wien, 20. August 2016

Bernhard Redl Jesper Larsson Träff

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Extending the Pheet framework for Parallel Pipelined Applications

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Bernhard Redl
Registration Number 0828401

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Prof. Dr. Scient. Jesper Larsson Träff

Vienna, 20th August, 2016

Bernhard Redl	Jesper Larsson Träff

Erklärung zur Verfassung der Arbeit

Bernhard Redl
Mitteraustraße 3/31 3500 Krems

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 20. August 2016

Bernhard Redl

Danksagung

Zuallererst möchte ich mich bei Jesper Larsson Träff für die umfangreiche Betreuung dieser Arbeit bedanken. Ohne seine Hilfestellungen in technischen Belangen und sein Wissen bei Fragen, die das Schreiben von wissenschaftlichen Arbeiten betrafen, wäre diese Arbeit nicht möglich gewesen.

Ich möchte auch Martin Wimmer besonders erwähnen, da ohne seine Arbeit am Pheet-Framework und der Beantwortung zahlreicher Fragen zu diesem, diese Implementierung nicht möglich gewesen wäre.

Besonderen Verdienst an dieser Arbeit, die das Korrekturlesen und Unterstützung während des Studiums umfasst, kommt meiner Familie zu. Bei meinem Bruder bedanke ich mich für das Redigieren dieser Arbeit, welches die grammatikalische Qualität entscheidend verbessert hat.

Ich möchte meinen guten Freund Lukas besonders erwähnen, da er mir durchwegs mit fachlichem Feedback insbesondere zu C++ zur Seite stand.

Meiner Gruppe an Freunden, die mich durch die Studienzeit begleitet haben, ist geschuldet, dass ich dieses Studium abschließen kann. Das betrifft besonders Andreas, Martin, Christoph, Thomas, Roman, Sebastian, Maximilian, Claudio und alle anderen.

Last but not least, I would like to thank my group of Erasmus friends. They always provided me with the necessary distraction and cheered me up when things were not working as intended. Na zdrowie guys!

Kurzfassung

In dieser Arbeit wurde eine Implementierung des parallelen Pipeline Patterns für das Task-parallele Pheet framework erstellt. Das *Pipeline Pattern* kann in vielen Situationen in der Praxis angewandt werden. Es teilt Eingabedaten in unabhängige Teile (chunks), die durch eine Serie von Verarbeitungsschritten gehen. Diese Verarbeitungsschritte werden üblicherweise als *stages* bezeichnet. Manche dieser Schritte können parallel ausgeführt werden, während in anderen Schritten manche Teile auf vorangegangene Teile warten müssen.

Für die Entwicklung einer Pipeline Implementierung ist die Kenntnis von anderen existierenden Implementierungen notwendig. In dieser Arbeit wurden dafür die weitverbreitetsten Implementierungen analysiert. Für jede dieser Implementierungen wird eine technische Beschreibung und eine Analyse der Einschränkungen geliefert.

Unsere Implementierung ist *nicht-blockierend* (lock-free) and verwendet keine zentralisierten Datenstrukturen, um die Skalierbarkeit in Mehr-CPU-Systemen zu gewährleisten. Wir verwenden C++ atomics für die Synchronisierung in unserer Implementierung.

Um unsere Implementierung zu testen, wurden Benchmarks auf mehreren Systemen ausgeführt. Alle Benchmarks sind bekannte Probleme die sich gut auf eine Pipeline-Schnittstelle adaptieren lassen. Für jeden Benchmark wurde eine Referenz-Implementierung verwendet, um Eigenschaften der Benchmarks herauszufinden. Ein von uns erstellter synthetischer Benchmark misst den Mehraufwand für die Synchronisierung unserer Implementierung.

Das Ergebnis lautet, dass unsere Implementierung beim synthetischen Benchmark gut skaliert. Für die anderen Benchmarks ist das Ergebnis durchwachsen und hängt von den unterschiedlichen Systemen ab.

Abstract

In this thesis we provide a competitive implementation of the Parallel Pipeline pattern for the task parallel Pheet framework. The *Pipeline pattern* can be applied to many real world problems. The Pipeline pattern splits the input into independent chunks which go through a series of computation steps called stages. Some stages may run in parallel while others have to wait for preceding input chunks to finish.

To develop a new Pipeline implementation deep knowledge of existing implementations is required. Therefore this thesis analyzes common existing Pipeline implementations. A technical description of the implementations and their limitations is given.

Our implementation is designed to be lock-free and without central data structures to allow good scalability for many core systems. We use C++ atomics to perform non-blocking synchronization.

To evaluate our implementation, a set of benchmarks has been executed on different benchmark systems, each featuring a different architecture. The benchmarks contain popular problems which can be modeled efficiently with the Pipeline pattern. For each benchmark a reference implementation is used as baseline to compare our results. Additionally a synthetic benchmark is proposed to measure the synchronization overhead of our implementation.

We show that our implementation scales well using our synthetic benchmark. The other benchmarks yield very different results on our different test systems.

Contents

CHAPTER 1

Introduction

1.1 Introduction

This thesis describes an efficient, *lock-free* implementation of the *Pipeline Pattern* in the Pheet framework.

The Pheet framework [Wim13] developed at TU Vienna can be used to implement *task parallel programs* in *multiprogrammed environments.* See Section 1.2 for details about Pheet.

In *Task parallel programs* problems are decomposed into smaller independent units of work which are called *tasks.* Tasks are executed by a scheduler which distributes tasks to worker threads. Worker threads are typically bound to physical cores to increase the locality. Tasks may have dependencies to other tasks. Computations of such a program have to obey all dependencies and can be modeled as *directed acyclic graphs* (DAGs). *Tasks* are represented as nodes connected with edges representing dependencies. Dependency edges represent data dependencies between tasks. Task parallel systems are discussed in detail in Section 1.3.

Multiprogrammed environments [ABP98] name systems where parallel computations are executed on a growing or shrinking set of processors.

Lock-freedom is a progress property of parallel systems which requires that at least one thread makes progress in a bounded number of steps.

Patterns [Gam+95; MSM04] provide solutions for commonly occurring programming problems. Patterns in our context do not provide specific implementations, instead they provide a framework how to structure the specific problems to make them efficiently executable. Patterns for task parallel systems focus on decompositions of problems into smaller tasks and executions of those tasks in required order.

Task based systems are an extension to parallel models like fork-join parallelism. Often task based systems are implemented on top of fork-join models. There are multiple patterns for parallel programs extending the fork-join semantics. This thesis focuses on a specific pattern called the Pipeline pattern. Besides this pattern there are other related patterns which aim to solve similar problems on top of a parallel programming model. These related patterns like *DAG*, *Future* and the *Hyperobject* pattern are described and compared to the *Pipeline* pattern in this introduction.

- In the *DAG* pattern, an unrestricted directed acyclic graph (DAG) is specified and executed by a scheduler under a stricter computational model. Each node in the DAG is a task. Only DAGs where each task is executed once are considered. See Section 1.4 for details about the DAG pattern.
- The *Pipeline* [MSM04] pattern is a case of the DAG pattern where the input is divided into chunks which flow through a sequence of stages. Each stage can be seen as a task. It performs a computation and is executed once for each input chunk. Input chunks may have logical dependencies on previous chunks at certain stages. This is modeled with dependency edges. The Pipeline pattern is described in Section 2.1.
- A *Hyperobject* [Fri+09] is a high level construct which uses the underlying structure of the synchronization graph to merge *local views* of shared variables at specific points. It allows threads to maintain coordinated local views of the same shared variable. This allows efficient implementation of so-called Reducers and Finishers [Wim13]. Hyperobjects are also capable of implementing the Pipeline pattern using a special queue data structure. See Section 1.5 for details about the Hyperobject pattern.
- *Futures* are a parallel language construct which is often found in functional programming languages. Future values are computed in the background and execution is halted only when a read access is performed on the Future argument. The Future value acts as a placeholder of the value which is computed in the background. See Section 1.6 for details about Futures.

The remainder of this thesis is structured as follows: The following section introduces basic concepts of task parallelism. Additionally the three related patterns (DAG, Hyperobject and Futures) are described in more detail and their relation to the Pipeline pattern is shown.

The Pipeline pattern is explained in detail in Chapter 2 and properties of state of the art implementations like Intel TBB, Piper and Nabbit are analyzed. Chapter 3 introduces our Pipeline implementation in the Pheet framework. We introduce our C++ interface and show our *lock-free* data structures and argue that our implementation is correct. Our source code is available on request to the author.

Chapter 4 introduces the benchmarks we used and the necessary adaptions to convert them to our Pipeline interface. The benchmarks were used to analyze the scalability of our implementation for real world problems and synthetic workloads. We formulate expectations on the scalability based on previous work of others. Chapter 5 analyses our benchmark systems in detail. The results of the benchmarks can be seen in Chapter 6.

The last Chapter 7 gives a summary of our implementation and future work.

1.2 Pheet framework

The Pheet framework implements a task parallel model and was developed by Martin Wimmer at the TU Vienna during his PhD thesis [Wim14]. It was created with the goal to create a framework where every data structure and scheduler is plug-able and can be exchanged easily. It features multiple task schedulers for shared memory systems. Pheet has been released under an open source license[1].

Multiple concurrent and *wait-free* data structures are included in the framework. They can be easily exchanged using the highly flexible plug-in architecture which is very beneficial to integrate our Pipeline implementation. Implementation and interface are written entirely in C++ and heavily use C++ templates for customizations.

The included Pheet benchmark suite is used to benchmark Pheet components. We added new benchmarks and adapted the benchmark suite to work with our Pipeline implementation.

The Pheet framework is very portable and runs on many different system architectures. It only requires C++11 compiler support and the *hwloc* [Bro+10] library for thread pinning.

We used the Pheet framework to implement our Pipeline because it is very flexible and provides *wait-free* data structures out of the box (Finisher, Performance Counter). The Pheet task scheduler and the Finisher are important features which we rely on for our Pipeline implementation.

1.2.1 Pheet Data Types

Beside from multiple schedulers introduced in Section 1.2.2 Pheet contains important data types used in our implementation:

Reducer Hyperobjects allow different threads to maintain coordinated local views of shared variables. The reduce operation is performed on synchronization points like object destruction. We use Pheet Performance Counters based on Reducer Hyperobjects to measure performance across different threads without affecting the performance. For details about Hyperobjects and Reducers see Section 1.5.

[1]Martin Wimmer. *Pheet.* URL: `http://pheet.org/` (visited on 05/10/2016).

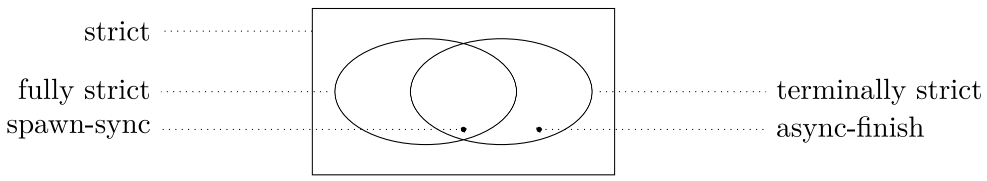


Figure 1.1: Expressiveness of different *Computation models* [Guo+09] and their relations are shown in this figure.

Finisher Hyperobjects uses a *wait-free* reference counting algorithm to ensure that all spawned threads complete. We use this Finisher to ensure that our Pipeline terminates.

1.2.2 Pheet Scheduler

Pheet provides multiple task schedulers which can be exchanged very easily using the plug-in architecture. We focus in our work on two schedulers:

BasicScheduler The *BasicScheduler* is a very simple work stealing scheduler. Aside from task scheduling it has no additional functionality and is very lightweight.

StrategyScheduler The *StrategyScheduler* allows the user to specify *strategies* which specify which tasks of the queue are executed first. Good strategies can increase the locality and increase performance. We used this scheduler as base for our Pipeline scheduler.

1.3 Task Parallelism

Task parallelism describes how to *"decompose a problem into a collection of tasks that can execute concurrently"* and how this *"concurrency can be exploited efficiently"* [MSM04].

We explain in this section how the executions of a task parallel program can be modeled. Also, we define certain characteristics which limit the expressiveness of the mentioned models. Further we explain how task parallel programs can be executed.

1.3.1 DAG - Structure

A DAG (directed acyclic graph) is a standard way to model an execution of a task parallel program. The execution of every task-parallel program can be modeled as a DAG [ALS10].

DAGs consists of nodes and directed dependency edges which are not allowed to contain cycles. Each task of the program is modeled as a node in the graph. Edges between nodes are dependencies.

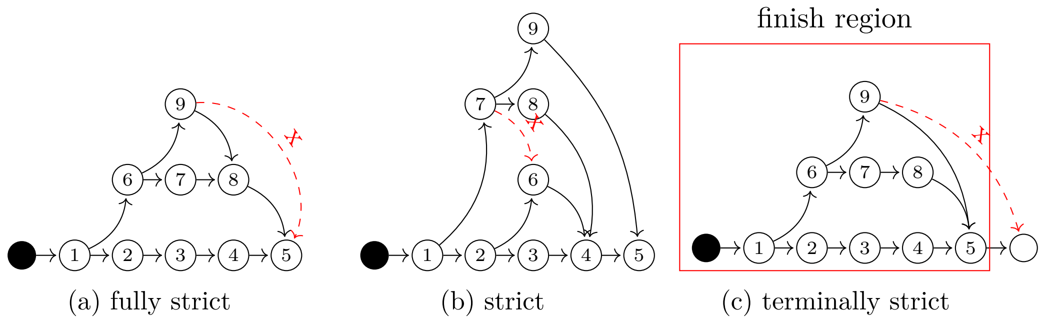


Figure 1.2: Expressiveness of different *Computation models* [Guo+09]. In the three shows DAGs, downward edges indicate *join* edges whereas edges going upward indicate *spawn* edges. Red crossed edges indicate forbidden *join* edges for this group. *Fully strict* computations force each child to join directly into the parent thread. *Strict* computations require each child to join into a spawn tree ancestor. In *Terminally strict* models children have to join inside their finish region. The corresponding implementation can be seen in Listing 1.1 (fully strict), Listing 1.2 (strict) and Listing 1.3 (terminally strict).

Fully strict Pheet example

```
void n1() {
  Pheet::Finisher f; //forces that node 6 join at the return of ↩
      the function
  Pheet::spawn( n6 );
  n2(); n3(); n4(); n5(); } // execute node 2-5

void n6() {
  Pheet::Finisher f;
  Pheet::spawn( n9 );        // execute node 9 in background
  n7(); n8(); }              // execute node 7,8
```

Listing 1.1: This code shows a *fully strict* example (see Figure 1.2a) implemented in Pheet. Each function which spawns nodes creates an own finish region using the Pheet Finisher object. Finisher objects force the join of all parallel computations at the end of the function (return). The function calls correspond to the task numbers in Figure 1.2a.

Strict C++ thread example

```
void n1() {
  thread7 = new std::thread( n7 );
  n1(); // n1 -> node 1
  n2(); // execute work of node 2
  n3(); // ..
  n4(); // ..
  n5(); // execute work of node 5
}
void n2() {
  // work of node 2 - not shown here
  thread6 = new std::thread( n6 ); } // start node 6 in the ↩
      background

void n4() {
  // work of node 4 - not shown here
  thread6.join(); thread7.join(); } //join node 6,7

void n5() {
  // work of node 5 - not shown here
  thread9.join(); }

void n7() {
  // work of node 7 - not shown here
  thread9 = new std::thread( n9 );
  n8(); }
```

Listing 1.2: This C++11 thread code implements the *strict* example in Figure 1.2b. Pheet does not give precise control for joining threads. Pheet is a task scheduler. It takes away the thread management from the user and thus does not provide fine control when to join subcomputations. For details about Pheet see Section 1.2.

Nodes may have multiple outgoing dependency edges. Such nodes enable multiple successor nodes to be executed concurrently.

In *spawn/sync* models the execution can be represented as a DAG. In these models nodes with multiple outgoing edges are called *spawn nodes*. Nodes with multiple incoming edges are called *sync nodes*.

Each DAG has a special start node which has no incoming dependency edges. The execution is started in this node.

Terminally strict Pheet example

```
void n1() {
  Pheet::Finisher f;
  Pheet::spawn( n6 );
  n2(); n3(); n4(); n5(); }

void n6() {
  Pheet::spawn( n9 );
  n7(); n8(); }
```

Listing 1.3: This Pheet code implements the *terminally strict* example in Figure 1.2c using a Pheet Finisher. The Finisher object creates a finish region. Note that node 6 (`n6`) does not create an own finish region here which is the main difference to the fully strict example in Figure 1.2a and Listing 1.1.

1.3.2 Computation models

The expressiveness of the DAGs can be classified into the following three *computation models* to obtain proveable bounds on space and time in specific implementations.

- Models with *fully strict* [Hal+14] semantics force a spawned child to join into the *parent thread* when it ends. This implies that a parent thread is active at least as long as its children threads are (this model is used by Cilk [Int13] and Cilk-P [Lee+13]). Figure 1.2a shows this. In this example node 9 has to join into node 8 because it is the last node of its parent thread. The parent thread was started with node 6. The red edge going directly from the top most node to the last node is forbidden because children are only allowed to join into their direct parent thread. Listing 1.1 gives a Pheet implementation of the example.

- In *Strict* computations [BL99] every join edge goes to a spawn tree ancestor. An example is given in Figure 1.2b. In this example the top most node 9 can join directly into the last node 5 because this node is an ancestor. In a fully strict computation node 9 would have to join into node 8. The red edge is forbidden because its target is a predecessor and therefore can not be a join target for the red edge. Listing 1.2 gives a C++11 threads example because Pheet does not offer fine control for thread joining. The Pheet framework is a *task scheduling* framework and takes away the handling of the threads from the user.

- *Terminally strict* [Aga+07; Guo+09] computations consist of *finish regions.* Threads which are spawned inside a finish region have to join in this finish region. The finish region only ends when all spawned threads have been joined. This method is used in X10, Habanero and OpenMP tasks. Pheet supports finish regions using the Finisher Hyperobject [Wim14]. In C++ there exists a draft for finish regions which are called *Task Blocks* [Hal+15; Hal+14].

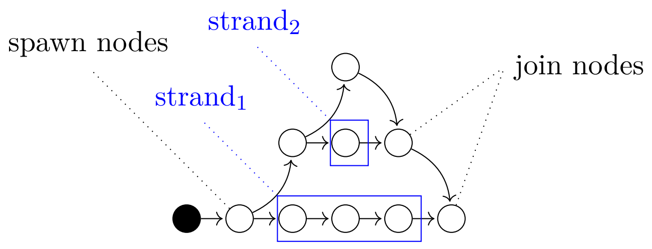


Figure 1.3: *Strands* [Fri+09] are sequences of nodes in DAGs without any parallel control nodes like *sync* or *spawn* nodes. The two strands in this example are marked with blue boxes.

> An example task graph for a terminally strict computation can be seen in Figure 1.2c. All edges joining inside their finish region are allowed. The red dashed edge joins a node outside the finish region and therefor is forbidden. Listing 1.3 gives a Pheet example using a Pheet Finisher object to create a finish region. In this example node 6 (`n6()`) does not create an own Finisher object in contrast to the *fully strict* example in Listing 1.1.

Figure 1.1 shows the relation of the *computation models* to each other. In Figure 1.2 examples for the three most important models are given. Forbidden join edges are shown in red. Edges going up represent *spawn edges*. Downward Edges represent *join edges*.

1.3.3 Programming models

There are multiple programming models for task parallelisms:

The *spawn/sync* model is applied by Cilk [ALS10] and provides the *spawn* and *sync* functions to manage parallelism. All threads spawned in a region have to join on the *sync* statement of the region. Also Pheet [Wim13] uses this model. The *sync* command is provided using a Finisher Hyperobject.

The *fork/join* model defines a *fork* method to spawn a new subcomputation and a *join* method to wait for a given subcomputation. An example can be seen in Figure 1.4. This gives a more precise control about the point where the subcomputation is joined. Pthreads in Linux are an example of this model.

Another related model is *Futures*. Futures provide an implicit form of parallelism and are common in functional programming languages. See Section 1.6 for details about Futures and their relation to Pipelines.

1.3.4 Strands

Strands [Fri+09] are sequences of nodes not containing any parallel control instructions. Parallel control instructions are *sync* or *spawn* nodes. A strand may also consist only of a single node. Figure 1.3 shows a DAG with two strands marked with blue boxes.

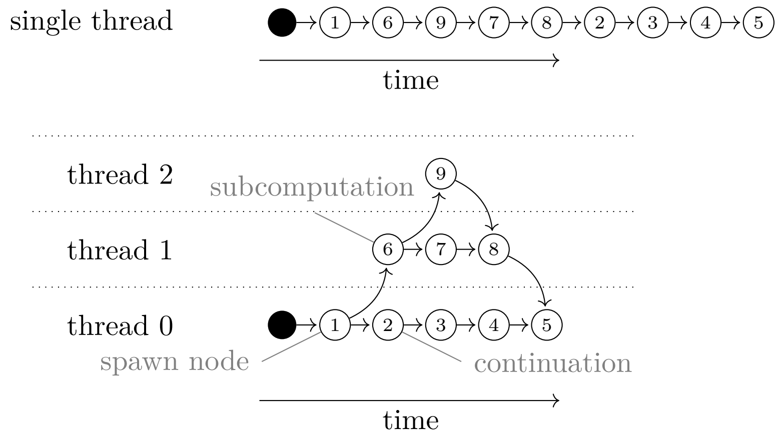


Figure 1.4: A possible *serial execution* of a DAG is shown in the upper part. Each spawn node is treated as a function call. The *parallel execution* of the DAG in the lower part uses multiple threads. The threads are shown on the y-axis whereas the time is shown on the x-axis.

1.3.5 Executions

Executions of DAGs can be done sequentially or in parallel.

In a *sequential execution* for spawn nodes the new subcomputation is executed first. Once it is finished the *continuation* is executed. A *spawn* node enables the *subcomputation* and continues to the *continuation.* Figure 1.4 shows the *spawn, continuation* and *subcomputation* nodes. In a sequential execution a *spawn* instruction is handled as a *blocking call.* Join instructions can be omitted because the subcomputation has completed before the continuation is executed.

In *parallel executions* DAGs can be dynamically executed by a scheduler. The scheduler distributes tasks to worker threads which execute tasks. A parallel execution using multiple threads can be see in Figure 1.4. One possible mechanism to coordinate worker threads is *work stealing* [BL99; ALS10; ABP98] which is used in *Pheet* [Wim13], *Cilk* [Fri+09], *X10* [Aga+07] and *Intel Threading Building Blocks* [Ale07]. In work stealing an empty worker thread picks a random victim and tries to steal work from the victim's queue.

Schedulers usually track the state of each node and its dependency edges. When all dependencies are satisfied a node is marked as *ready* and is scheduled for execution by the scheduler.

1.4 DAG - Pattern

One has to clearly distinguish between the DAG as structure of computation as described in Section 1.3 and the DAG as programming pattern explained in this section. The DAG programming pattern enables the programmer to execute an arbitrary, unrestricted DAG (structure) under a stricter programming model.

This requires that the *task graph* is converted to a structure supported by the underlying scheduler.

1.4.1 Task graphs

The model of the DAG pattern is a *task graph* which models the dependencies of subcomputations (tasks). Every ready task may be scheduled for execution. A ready task is a task where all dependencies to other tasks are satisfied. This means that the depending tasks already have been executed.

Task graphs can be modeled as DAGs. An example of a specific DAG state is given in Figure 1.5. Blocked nodes with unsatisfied dependencies are drawn with dashed lines.

Formally a task graph is a tuple $D = (V, E)$ where E is a set of dependency edges and V is a set of nodes.

Pipelines can be modeled using multiple task graphs. Each Pipeline iteration can be modeled as an own small task graph. This way a Pipeline model can be transformed into a DAG. For more details about modeling a Pipeline using a DAG see Section 1.4.2.

In theory a task graph gives the following lower bound: The *completion time* [Cor+09, p. 780] on P processors is at least $max\,(T_1/P, T_\infty)$ where T_1 is the sum of all node completion times in the DAG and T_∞ is the completion time of the nodes on the longest directed path in the DAG. A graphical representation is given in Figure 1.6 P is the number of computation cores.

T_1/T_∞ is called the *parallelism* [ABP98]. It restricts the maximum possible *speedup*. Even if more computation resources are added to the execution, the speedup can not be

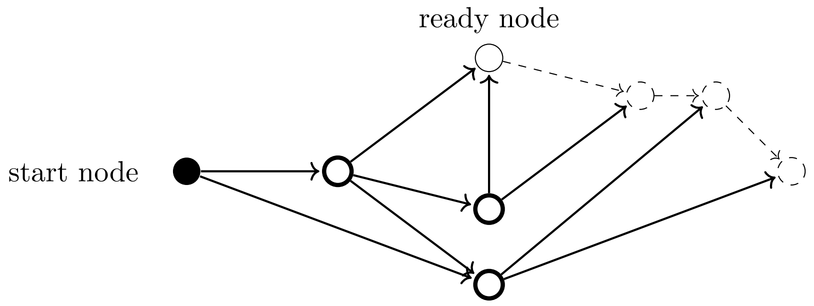


Figure 1.5: Example of a specific *DAG* state. Edges represent dependencies where $\longrightarrow$ mark satisfied dependencies, $\dashrightarrow$ mark currently unsatisfied dependencies, dashed circles ◌ mark tasks with *unsatisfied dependencies* and bold circles ○ mark *completed tasks*. The circle ○ indicates a *ready node* where all dependencies are satisfied but which has not been executed yet.

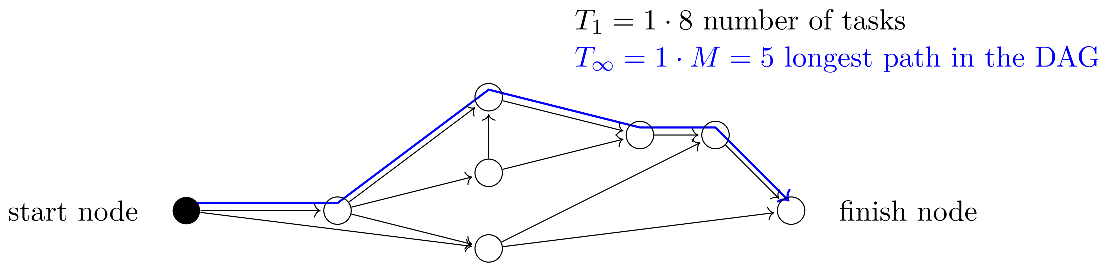

$T_1 = 1 \cdot 8$ number of tasks
$T_\infty = 1 \cdot M = 5$ longest path in the DAG
start node
finish node

Figure 1.6: The blue line shows the *longest path* T_∞ in the DAG. For the longest path T_∞ only the completion time of the nodes is relevant. M is the length of T_∞. T_1 is the sum of the completion times of all nodes in the DAG. For easier understanding in this example, all nodes have a *completion time* of 1.

increased further. See Section 4.2 for more details about the speedup.

Task graphs can be distinguished into two types regarding the flexibility:

- In *Static Task Graphs* [ALS10] all nodes and dependencies between them are known a priori. Additionally the computation time for each node is known. An optimal schedule can be created beforehand. The generation of such a minimal-time execution schedule is NP-complete but heuristics are available.

  To show that the schedule generation is NP-complete Ullman [Ull75] showed that the well known NP-complete 3-SAT problem can be *reduced* in polynomial time to a scheduling problem. The proof is rather complex and is not given here.

- In *Dynamic Task Graphs* [ALS10] new nodes and dependency edges are created on-the-fly. A *dynamic scheduler* is needed to maintain a task pool of *ready nodes*. Scheduling decisions are made during runtime by the scheduler. Several methods for work distribution between the workers exist. One method of distributing work in a task pool is work stealing (used in Cilk++ and Intel TBB).

1.4.2 Relation to Pipelines

This pattern is very related to the Pipeline pattern because every Pipeline execution can be modeled as a task graph. Because task graphs are represented using DAGs they can not contain loops. Therefore each Pipeline iteration has to be modeled as a separate DAG. An artificial start node connects all DAGs.

A DAG scheduler can thus execute a converted Pipeline, but it can not take advantage of restrictions imposed by the Pipeline pattern. These restrictions, like no forward edges, can be used to reduce the memory footprint and reduce the amount of needed synchronization in the implementation.

1.4.3 Nabbit

As an example of a general DAG scheduler *Nabbit* [ALS10] is given. Nabbit is an extension to the multithreaded programming language Cilk++. It executes dynamic task graphs inside the fully strict Cilk programming model. Nabbit could be implemented in any fork-join library which supports *work-stealing* [ALS10].

Nabbit integrates well with Cilk++ *fork-join* constructs. Inside Nabbit nodes, these Cilk++ constructs can be used to exploit additional parallelism.

Nabbit is oblivious to changes of processing resources in multiprogrammed environments. When a worker thread is descheduled by the operating system, its nodes are stolen by other active worker threads (*work stealing*). This is considered an important property in *multiprogrammed environments* [ABP98].

Nabbit supports *static* and *dynamic task graphs.* In *static task graphs* all dependencies and nodes are known a prior. The execution time of each node is usually not known.

In *dynamic task graphs* dependencies and nodes are created on-the-fly during runtime. Nabbit offers an interface for the programmer to add new nodes and dependencies during runtime. Dependencies are added during a *discover* phase before executing a node. This discovery follows the notion that each node knows its dependencies.

As an example we use a dynamic program to compute a $M(n, n)$ matrix with an input matrix $s(n, n)$. Each cell is calculated by checking the cell above and the cell left of the current position. From the input matrix `s` a factor is added to each of these neighbor cells. The bigger value is used as a result for the current cell. The equation used is:

$$M(i,j) = max \begin{cases} M(i-1,j) + s(i-1,j) \\ M(i,j-1) + s(i,j-1) \end{cases} \tag{1.1}$$

In Listing 1.4, a Nabbit pseudo code example is given which solves this problem with Nabbit. First, for every cell the neighbor above and left are added as dependencies. Then the execution is started and Nabbit schedules the executions according to the dependencies.

Implementation

To track ready nodes Nabbit intuitively counts the number of immediate (not yet completed) predecessors for each node in the graph. Additionally Nabbit has to track the status of each node in the dynamic task graph.

For *static task graphs* Nabbit has to maintain certain information for every node in the graph. This is mainly a *successor array* which stores the reverse of the dependency edges and a *join counter* which is used to track yet unfinished predecessors. Once a node is finished Nabbit decrements the join counters of the successors in parallel and schedules a successor if its counter reached 0.

Nabbit static task graph example

```
g = new Node[n*n]
for i=0 ; i < n ; i++:
  for j=0 ; j < n ; j++:
    k = n*i+j
    g[k].pos = k
    if i > 0: g[k].AddDep(g[k-n]) //cell above
    if j > 0: g[k].AddDep(g[k-1]) //cell left
    g[0].execute() // start task graph execution
    // this is a nabbit system method

class Node
  int res;
  void Compute():
    // user defined function performing the work
    res = 0;
    for i=0 ; i < predecessors.size() ; i++:
      Node pred = predecessors.get(i)
      int pred_val = pred->res + s[pred->pos]
      res = MAX(pred_val,res)
```

Listing 1.4: Pseudo Nabbit code to solve Equation 1.1 with a static task graph [ALS10]. The equation defines 2 dependent cells for each cell in the matrix. After the values of the dependent cells have been computed, a value from an input matrix is added. The bigger value is stored in the current cell. The `AddDep` method is used to define dependencies. The `execute` method is provided by Nabbit and starts the computation. This function has to be called on the start node of the DAG. In contrast the `Compute` function has to be defined by the user. In this function the work for the node has to be performed. It is called by Nabbit when all dependencies are met.

For *dynamic task graphs* dependencies of each node have to be *discovered* by Nabbit before executing the node. To accomplish this, the user has to provide a method to Nabbit which returns the direct predecessors for a given node. This method is guaranteed to be called once per node by Nabbit. A new *status field* for each node ensures this.

Performance

To analyze the performance of Nabbit a *work/span analysis* [Cor+09, Chapter 27] is used.

It is known that the Cilk work-stealing scheduler [BL99] has the following time bounds with probability of $1 - \epsilon$, where ϵ can be chosen freely.

$$\mathcal{O}\left(T_1/P + T_\infty + \lg\left(P/\epsilon\right)\right) \tag{1.2}$$

Where

T_1 is the total work of all nodes,

P is the amount of used computation cores and

T_∞ is the span which is also called *critical path length*

T_1 is the time it takes to execute the computation on a single computation core. The *critical path length* T_∞ is the time it would take on ∞ computation cores. This is visualized in Figure 1.6.

Agrawal, Leiserson, and Sukha [ALS10] have shown the computation time bounds for Nabbit to be (with freely chosen probability $1 - \epsilon$):

$$\mathcal{O}(\underbrace{T_1/P + T_\infty}_{\text{lower bound}} + \lg(P/\epsilon) + \underbrace{M \lg \Delta_o}_{\substack{\text{visit successors}\\ \text{in parallel}}}) + \mathcal{O}(\underbrace{(|E|/P + M)}_{\substack{\text{graph traversal}\\ \text{in parallel}}} \cdot \underbrace{\min\{\Delta_i, P\}}_{\substack{\text{worst case}\\ \text{contention}}}) \tag{1.3}$$

Where

$\mathbf{\Delta_o}$ is the maximum *out degree* of nodes in the DAG. out-going edge represents a dependency for the target node.

$\mathbf{\Delta_i}$ is the maximum *in degree* of nodes in the DAG,

M is the number of nodes on the longest path.

$M \lg \Delta_o$ represents the work to visit successor nodes in parallel, which is dominated by the T_∞ term in graphs with reasonable work in the nodes.

$(|E|/P + M) \cdot \min\{\Delta_i, P\}$ models the worst-case contention (waiting for join counters) where,

$(|E|/P + M)$ is a bound for the parallel traversal of the DAG.

$\min\{\Delta_i, P\}$ term indicates the worst case where P computation cores try to decrement the join counter of a single node. The join counter is decremented using an atomic decrement operation. The DAG is traversed in parallel by P threads. So up to P threads may decrement the join counter of a single node concurrently. If the node has less input edges Δ_i than threads, at most Δ_i threads can reach this node at the same time.

For *dynamic task graphs* the time bounds of the static version have to be slightly modified:

$$\begin{aligned} \min\{\Delta_i, P\} &\quad\rightarrow\quad \min\{\Delta\ , P\} \\ M \lg \Delta_O &\quad\rightarrow\quad M\Delta \end{aligned} \tag{1.4}$$

Where Δ is the maximum degree of any node in the graph. This term arises because every node is visited twice in the dynamic version: Once during the `Init` phase and once when the `Compute` function is called by Nabbit. The new $M\Delta$ term arises because successors may be discovered and added serially to the *successor array.* The *static version* of Nabbit visits successors in parallel ($\lg \Delta_O$).

According to the authors of Nabbit the *dynamic version* suffers a slowdown of factor 5 in practice compared to the *static version.* The benchmark was done with a constructed medium size task graph where the work done inside the nodes is small compared to the scheduler overhead. In graphs where the work inside the nodes dominates the bookkeeping, the static and the dynamic version perform nearly the same.

Agrawal, Leiserson, and Sukha [ALS10] claim that the overhead for a single core execution of a problem implemented in *static Nabbit* is about 16 % compared to a *serial implementation* of the same problem.

1.5 Hyperobjects

Hyperobjects [Fri+09] are a high level construct which allows different threads “to maintain coordinated local views of the same shared variable” [Fri+09]. They use the structure of the underlying synchronization graph to merge local views at defined points. The goal is to reduce the synchronization overhead for shared variables in parallel environments.

The Hyperobject Pattern can be used to implement a Pipeline, therefore it is explained here.

For each strand a *local view* of the shared variable exists. As explained in Section 1.3.4 a *strand* is an instruction sequence without concurrency instructions (join, spawn). Inside a strand no synchronization is needed to access the shared variable because the local view is exclusively owned by the strand’s thread. On synchronization points where two strands are joined, the local views are combined. New local views are created in *spawn* nodes in the computation DAG.

A Hyperobject also provides an abstraction to coordinate different strands that depend on the same variables. Generally, Hyperobjects are used to separate dependent strands and perform synchronization. [Fri+09]

An alternative to Hyperobjects for synchronization of shared variables would be a *centralized lock* to protect the shared variable which may increase contention as the worker count increases.

The main usages of Hyperobjects are:

Reducer [Fri+09] which provide a way to merge partial results computed in parallel. Preferably the order is preserved as in a serial execution. The reduction operation can be user-defined and applied in the background once intermediate results are available.

Finisher [Wim13] can be used to implement the basic synchronization primitive *finish* in *async finish* models. Async finish models are more flexible than *fully strict* models like Cilk (see Section 1.3.2). A *finish region* [Wim13] forms a boundary for all threads spawned inside the region. An example of a finish region can be seen in Figure 1.2c.

Hyperqueues [VCN13] are special queues which allow program construction in a *Pipeline fashion.* They extend the local views of Hyperobjects by introducing *shared views* between a single producer and a consumer. Each stage is represented as a thread which takes a Hyperqueue object parameter. This parameter is annotated to define if the Hyperqueue object is used as producer (output) or consumer (input). All stage threads are operating on the same Hyperqueue object instance. The synchronization is done using the Hyperobject Pattern. Hyperqueues are discussed in detail in Section 1.5.1.

Wimmer showed that Hyperobjects can be implemented *wait-free* [Wim13].

1.5.1 Hyperqueues

Hyperqueues [VCN13] are special queues which are an implementation of the Pipeline pattern. They extend Hyperobjects with the addition that two threads can share a common view, in contrast to exclusive local views in the above mentioned Hyperobjects.

Hyperqueues implement the Pipeline pattern using techniques from Hyperobjects and from *task dataflow* models. In *task dataflow* models programmers describe what variables are inputs to tasks and what variables are used as output. These annotations model task dependencies.

Other approaches which provide the Pipeline pattern are often tuned to a specific number of processing cores. Hyperqueues instead are *scale-free* [VCN13] which means that they are oblivious to the specific number of computation cores.

Hyperqueue Pipeline example

```
struct data { ... };
void consumer( popdep<data> queue ) {
  while( !queue.empty() ) {
    data d = queue.pop();
    // ... operate on data ...
  }
}
void producer( pushdep<data> queue, int start, int end ) {
  if ( end-start <= 10 ) {
    for ( int n=start; n < end; ++n ) {
      data d = f( n ); // create chunk n
      queue.push( d ) ;
    }
  } else {
    spawn producer( queue, start , ( start +end )/2 );
    spawn producer( queue, ( start +end )/2, end );
    sync;
  }
}
void pipeline ( int total ) {
  hyperqueue<data> queue;
  spawn producer( (pushdep<data>) queue, 0, total );
  spawn consumer( (popdep<data>) queue );
  sync;
}
```

Listing 1.5: This example shows a simple Pipeline which consists of a single producer stage and a consumer stage. Both stages are spawned as threads and operate on the same Hyperqueue Hyperobject. The arguments are annotated with `pushdep` for producer and `popdep` for consumers [VCN13].

They are executed in a predictable, deterministic way which can reduce debugging effort. Deterministic execution is considered an important property [Boc+09].

Implementation

As a specific implementation the only to our knowledge known implementation of Hyperqueues [VCN13] will be discussed. In this implementation threads are classified as *producers*, *consumers* or both. The Hyperqueue is a special *single-producer*, *single-consumer* queue and each local view is shared between one producer and one consumer thread.

Each task is represented by a *procedure* which takes arguments. Annotations of these arguments decide if the task is a producer or a consumer, which models the dependencies.

Possible annotations for task arguments are:

Consumer Threads executing tasks with this annotation can only run when older *consumer* threads are already completed. This ensures a deterministic order which is considered an important property.

Producer Multiple producer threads may push created values concurrently into the same queue. In this case every producer creates a different range of data. *Reductions* are used to merge the different ranges on synchronization points (like Hyperobjects). Producer threads can also execute concurrently with earlier started consumer threads, with the limitation that older consumer threads can not see newly created data.

A combination of consumer and producer is also possible. In this case the executing thread inherits the restrictions from both types.

It is important for the program order that consumers can only read values written by preceding producers (the data of a later started producer is not visible to earlier consumers).

An example of a simple Pipeline program implemented with Hyperqueues is given in Listing 1.5. In this example the Pipeline consists of two stages. A producer stage which creates data in parallel and a consumer stage which consumes the data. All stages are modeled as threads which take the same instance of the Hyperqueue object. This argument is annotated to specify if it is used to consume (input) or to produce data (output).

1.6 Futures

Futures are *parallel language constructs* which can be commonly found in functional orientated programming languages [BR97].

When a Future value is declared the computation of the value is spawned in a new thread but a pointer to the result location is returned immediately. This pointer can be used like the normal value and can be passed to functions or added to lists. When a thread actually tries to read the value of the pointer it is checked if the value has already been computed. In case the value is ready, the thread can continue immediately and in case it is not ready it has to wait.

Futures exploit the fact that often the results of a computation are not needed immediately. Often results are just passed to functions or are added to lists without reading / processing them instantly. Later when the actual value is read the other thread may already have finished the computation.

Futures are a very easy to use concept of parallelism because the user does not have to care about synchronization issues and can phrase his algorithm naturally.

```
fun produce(n) =
  if (n < 0 ) then nil
  else n::?produce(n-1);

fun consume(sum,nil) = sum
  | consume( sum, h::t) = consume(h+sum,t);

consume(0,?produce(n));
```

Listing 1.6: This example shows a Pipeline consisting of a producer and a consumer using Futures implemented in the functional programming language ML [BR97]. In ML syntax the `?` operator defines a Future argument. The `::` operation performs list concatenation. In this example the producer creates numbers which are concatenated into an output list. The consumer splits the list into a head and a tail list (`h::t`) and adds the head element to the calculated sum. The functions call themselves recursively, which is very common for functional programming. An execution diagram of this listing is shown in Figure 1.7.

1.6.1 Pipelines

Blelloch and Reid-Miller first used Futures [BR97] to model Pipelines. It allows much simpler Pipelines by not coding the Pipeline explicitly. A consumer gets a *Future* list where the data elements are constructed on-the-fly in background. This method works very well in functional languages without side effects. Although the approach is very versatile it suffers from possible unbounded memory usage.

Executions of such Pipelines implemented with Futures in a functional programming language can be modeled using a DAG. The code of a Pipeline modeled in ML using Futures can be seen in Listing 1.6. The Pipeline consists of a single producer and a consumer. A list of elements is created by the producer and passed to the consumer. The consumer splits the list into a head and a tail list and adds the read head element to the computed sum. The corresponding DAG is shown in Figure 1.7.

1.7 Summary

We have shown three commonly used patterns for task parallelism beside of the Pipeline Pattern.

The *DAG pattern*, where a generic DAG (structure) is scheduled efficiently by a stricter scheduler. An open scalability problem is the limited memory bandwidth. A future research topic is the combination of multiple nodes into a single node in regular graphs to improve locality. Also the *garbage collection* is a further research topic for DAG schedulers.

The *Hyperobject* pattern uses local views of shared variables which are reduced on specific synchronization points. The local views allow concurrent computations without syn-

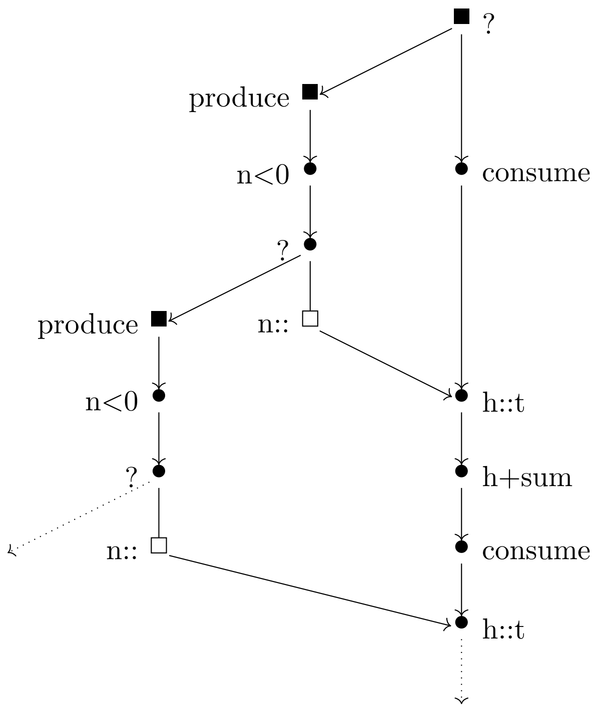


Figure 1.7: This DAG shows the execution of a Future Pipeline [BR97] from Listing 1.6 containing a single producer and a single consumer. Edges going left indicate spawns of new threads, whereas edges going right indicate join edges. The `?` operator denotes a Future argument. `n::` denotes adding element `n` to the beginning of the list.

chronization during instruction sequences without parallel control instructions (strands). They provide an efficient method to implement Reducer and Finish regions. Also the synchronization burden of shared variables is moved away from the user into the library or runtime system. We showed that *Hyperqueues* can implement Pipelines by annotating arguments as producer or consumer. A topic for further research could be *hierarchical tree-based* reductions. *Concurrent consumer threads* for producer-consumer systems would be another topic for further research. The same applies to *Garbage collection*.

Futures are a parallel language construct which is very common in functional programming languages. Future arguments are computed by spawned threads in the background. A thread only blocks when it tries to read values which are not yet ready. Pipelines can be modeled with Futures very compact and the user does not have to take care about synchronization.

1.8 Related work

A related pattern not described in detail is *Stream Programming* [UGT09]. Stream Programming is a programming model where programs consist of *filters* that are applied to independent data in parallel. This model can be used to program GPU architectures or Stream CPUs efficiently because they consist of a large number of independent computation cores. These cores execute the same code in parallel for independent input data.

Gordon, Thies, and Amarasinghe created the StreamIt library to exploit parallelism with coarse-grain pipeline tasks [GTA06].

Sanchez et al. implemented a *GRAMPS* runtime which bounds the memory footprint of a Pipeline program [San+11]. The GRAMPS programming model supports irregular Pipelines. They use per-stage queues and *backpressure* to limit the memory usage.

Pop and Cohen proposed an extention to the widely known OpenMP framework to support stream computing and Pipelines [PC11]. Pipelines are defined using input and output queues but support for dependency edges and non-linear Pipelines is limited.

Macdonald, Szafron, and Schaeffer showed that it is possible to model Pipelines as *state transformation* in an object orientated context [MSS04]. They use a *master/slave* approach to mitigate startup effects and allow better dynamic scalability. The *state design pattern*, where the behavior of an object changes depending on the state, is used to implement the Pipeline pattern. Worker threads pick arbitrary ready objects and transform them to their next stage.

CHAPTER 2

Pipelines

This chapter describes the Pipeline pattern in detail and shows different classifications for Pipelines. Additionally three state of the art implementations of the Pipeline pattern are analyzed. Advantages and disadvantages are given.

2.1 Pipeline Pattern

The Pipeline pattern [MSM04] targets problems where input can be split into partially independent *chunks*. These chunks all have to go through a series of processing steps often called *stages*.

An example of the Pipeline pattern is shown in Figure 2.1. It shows the production of cars C_n in a factory. Cars have to go through four stages. After a start-up phase every stage is filled with work and the computation runs in parallel [MSM04, p. 86]. A specific state of the Pipeline execution is shaded in orange. It is the first state after the start-up phase is completed. In the upper part of Figure 2.1 the corresponding serial execution is shown. Pipelines do not reduce the processing time for specific elements, instead they reduce the total processing time of all inputs.

The Pipeline pattern provides a natural way to express a lot of relevant real world problems like deduplication, video encoding and similarity search. For example many *server software* [BL12] related problems can be addressed with this pattern. Server software often has to perform a series of tasks per client request. These tasks can be modeled as Pipeline stages.

Because of the frequent use of this pattern, three Pipeline problems *Ferret* [Lv+07a], *Dedup* [Bie11] and *x264* [Vid] have been included in *PARSEC benchmark suite* [BL12]. *Ferret* searches for similarities in data sequences, *Dedup* compresses files by deduplicating blocks of data (Listing 2.3) and *x264* is the popular H.264 video encoder.

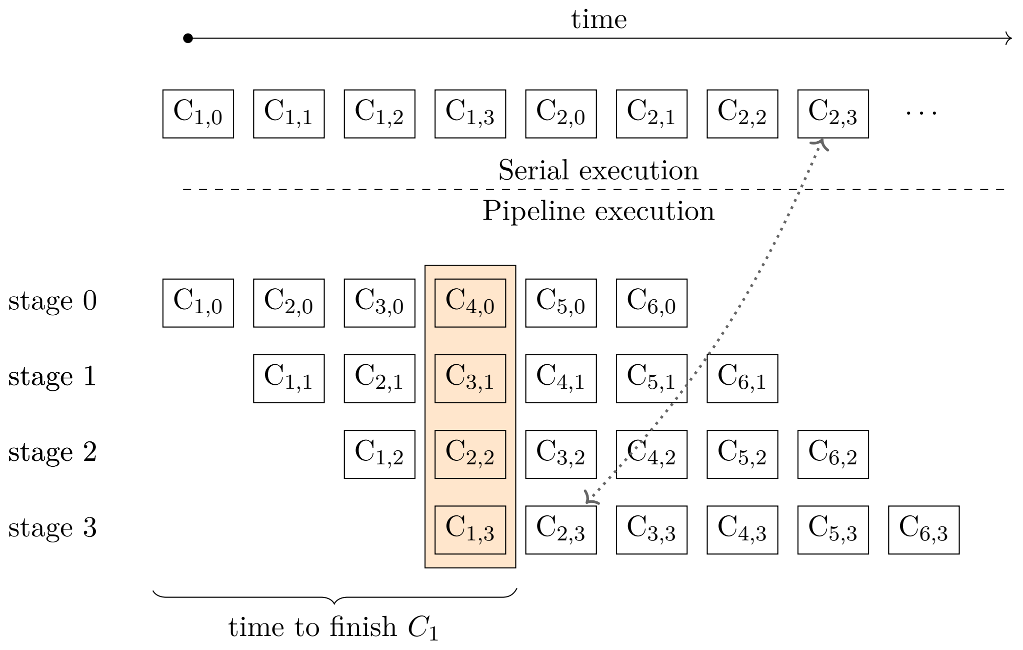

time
$C_{1,0}$ $C_{1,1}$ $C_{1,2}$ $C_{1,3}$ $C_{2,0}$ $C_{2,1}$ $C_{2,2}$ $C_{2,3}$ ...
Serial execution
Pipeline execution
stage 0 $C_{1,0}$ $C_{2,0}$ $C_{3,0}$ $C_{4,0}$ $C_{5,0}$ $C_{6,0}$
stage 1 $C_{1,1}$ $C_{2,1}$ $C_{3,1}$ $C_{4,1}$ $C_{5,1}$ $C_{6,1}$
stage 2 $C_{1,2}$ $C_{2,2}$ $C_{3,2}$ $C_{4,2}$ $C_{5,2}$ $C_{6,2}$
stage 3 $C_{1,3}$ $C_{2,3}$ $C_{3,3}$ $C_{4,3}$ $C_{5,3}$ $C_{6,3}$
time to finish C_1

Figure 2.1: Visualization of the Pipeline Pattern [MSM04]. The stages are numbered starting from 0. The used example shows the production of a car C_n. Each car requires four processing steps called $C_{n,0}, \cdots, C_{n,3}$. In the Pipeline model each processing step is modeled as a stage. The area shaded in orange marks a specific state of the entire Pipeline. At that moment the startup of the Pipeline is completed and every stage is processing work.

2.1.1 Definition

Formally Pipelines consist of abstract functions which are called stages $S = S_0, \ldots S_{s-1}$ and independent input chunks named $I = I_0, I_1, \ldots, I_{n-1}$. Every input chunk element must sequentially go through these stages.

Pipelines can be more complex than the previous car production example. In the previous example dependencies between iterations have been omitted. More complex Pipelines can have dependencies between adjacent iterations.

A linear Pipeline with four stages and n input chunks can be seen in Figure 2.2. This figure shows dashed nodes (input chunks at a specific stage) which are ready to be executed and nodes which have yet unresolved dependencies to other iterations. Two of the four stages are executed in parallel (filter$_1$, filter$_2$) resulting in no dependency edges between adjacent iterations. The first and the last stage do have dependencies between adjacent iterations. This means that an iteration has to wait for its preceding iteration until it reaches the required stage.

A Pipeline computation can be represented as a DAG (see Section 1.3.1). For each input

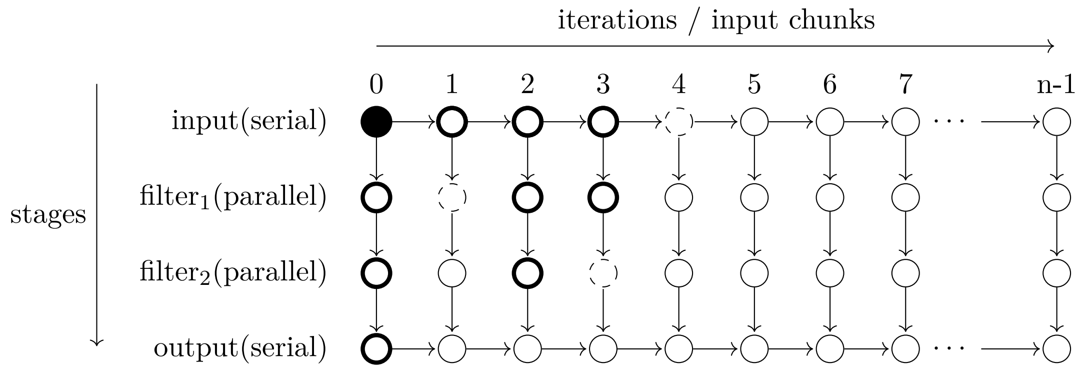


Figure 2.2: Specific execution state of a linear Pipeline with four stages. The first stage is a *serial stage* input stage, the second and the third stage are parallel stages (no dependencies between adjacent iterations). The last stage is a serial stage again ($\approx$ output stage). This represents a typical Pipeline in Intel TBB. Bold circles ○ mark *completed tasks*. The filled node indicates the start node. Dashed circles ◌ mark ready tasks. Circles ○ mark tasks which are not ready yet because of unfulfilled dependencies.

chunk there is a separate DAG because every node is executed exactly once in a DAG. We name the node representing chunk i at stage j (i, j). These DAGs are merged into one big DAG connected by the following edge types (see Figure 2.4):

stage edges $(i, j) \to (i, j') \mid j < j', j \in S, i \in I$ where $j \in S$ denotes the stage and $i \in I$ denotes the input data. This edge means that stage$_{j'}$ depends on the previous stage$_j$, which is usually the case in most problem instances. It is possible to skip stages with these edges which is useful in *on-the-fly* Pipelines.

cross edges $(i - 1, j) \to (i, j) \mid j < j', j \in S, i \in I$ means that the next input item at stage$_j$ can only be processed after the input chunk at iteration $i - 1$ has completed stage$_j$ (blue edge in Figure 2.4). Stages not containing any cross edges are called *parallel stages*. In *serial stages* all iterations of the stage are connected with cross edges. Mixed stages are called *hybrid stages*.

throttle edges In the Pipeline pattern *stack space usage* is an important consideration. Stack space may grow to an unlimited size because started, yet unfinished iterations have to be kept on the stack. To ensure proveable bounds, the maximum number of running iterations must be limited. Therefore, logical *throttle edges* are added in implementations supporting arbitrary dependency edges (represented as dashed blue lines in Figure 2.4). For further information see Section 2.1.2.

2.1.2 Throttling

For large problem instances not every input chunk can be held in memory. Each started input chunk has to be kept in memory until it is finished. If the scheduler does not take

Cilk-P Pipeline example

```
#define STAGE1 1
#define STAGE2 2
#define STAGE3 3

int fd_out = open_output_file();
bool done = false;
pipe_while (! done ) {
  chunk_t *chunk = get_next_chunk();
  if ( chunk == NULL ) {
    done = true;
  } else {
    pipe_wait( STAGE1 );      //Stage 1 is a serial input stage
    bool isDuplicate = deduplicate( chunk );
    pipe_continue( STAGE2 ); //Stage 2 (compress) runs in ↩
        parallel
    if (! isDuplicate )
      compress( chunk );
    pipe_wait( STAGE3 );      //Stage 3 is a serial output stage
    write_to_file( fd_out, chunk );
  }
}
```

Figure 2.3: Pipeline code example with two serial stages (`pipe_wait(STAGE1 ↩ )`, `pipe_wait(STAGE3)`) and one parallel stage (`pipe_continue(STAGE2)`). This pseudo code implements the *Dedup* benchmark [BL12] using the *Cilk-P* Pipeline interface [Lee+13]. The Dedup benchmark deduplicates duplicate file chunks to reduce the file size.

care of the amount of running iterations this may lead to a *runaway* Pipeline [Lee+13].

Multiple different approaches exists to limit the amount of running iterations. In implementations supporting arbitrary dependency edges, *throttle edges* can be added by the implementation to limit the amount of running iterations. A Pipeline example with throttle edges can be seen in Figure 2.4.

Another possibility is to count the amount of running iterations and block the spawning of new iterations if the counter exceeds the limit.

2.1.3 Result Passing

Every Pipeline stage has input and output data. We consider output data of stages as their result. Stage results may be passed in two dimensions during a Pipeline execution:

- Results have to be passed in an iteration from *stage to stage*. This is required because a specific input chunk has to go through the series of stages.

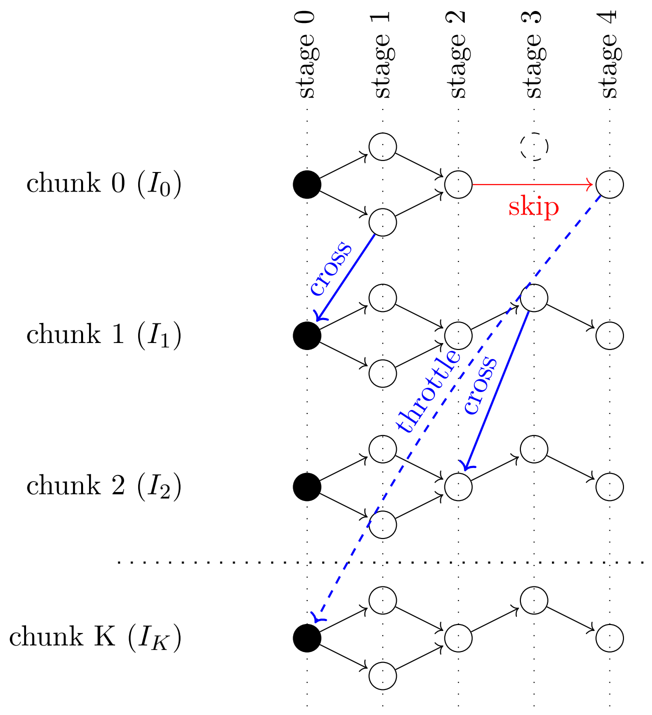


Figure 2.4: In theory the computation of every chunk may be modeled with an individual DAG. In practice this leads to hard to maintain code. In this example only the DAG for the first chunk is different compared to the other chunks. The DAGs are connected with *cross edges* and *throttle edges*. Filled circles indicate start nodes. Blue arrows indicate *cross edges* which are used to model dependencies between different input chunks. A black edge indicates a *stage edge*. The red stage edge is used to skip stage 3. To prevent unbounded stack space usage *throttle edges* (dashed blue edges) are added to restrict the amount of concurrently running iterations.

- In case an iteration depends on a preceding iteration, results may be passed from *iteration to iteration*. Whether this is necessary depends on the specific algorithm. Some algorithms only use dependencies for synchronization while others need result values from dependent iterations.

2.1.4 Execution of Pipelines

Pipelines can be distinguished according their type into two major groups. These two groups require different execution methods.

Construct-and-Run Pipelines require a static structure before the Pipeline is executed. The structure can not be changed during runtime. Every input chunk has to go through the same series of stages. Systems like TBB (Intel Threading Building Blocks) use this approach.

The dependent stages can be connected with queues. Worker threads use these queues to retrieve ready tasks and execute them. The queue data type has to be thread safe to allow a parallel execution of the Pipeline.

With this restricted Pipeline pattern some algorithms which require dynamic runtime decisions (like x264) can not be easily implemented [RCJ11].

On-the-Fly [Lee+13] Pipelines are types where the structure emerges during runtime. Dependencies between the stages can be different for each input chunk I_i. This type of Pipelines can not be implemented using static queues between stages.

It can be implemented on top of a scheduler which allows arbitrary dependencies between nodes. Also the DAG pattern can be used to create the structure during runtime and execute it on a more strict scheduler.

There are two possible scheduling approaches. Firstly, *bind-to-element* [Lee+13] where a worker executes tasks from multiple stages and only suspends a task if it hits an unresolved dependency. In this method, *work stealing* is applied to synchronize workers. Secondly, with *bind-to-stage* [Lee+13] every worker is bound to a stage and only executes ready nodes of this stage.

One possibility to coordinate stages in On-the-Fly Pipelines is to use *Futures* [BR97]. Futures are a construct used in functional languages to enable parallelism. This allows even more expressive definitions but may lead to *unbounded space* [Lee+13; BL93]. See Section 1.6 for details about Futures.

2.1.5 Flexibility of Pipelines

Pipelines can be distinguished according their flexibility into two groups:

Linear Pipelines have a simple structure. Each stage consumes one input element and outputs one element. All input chunks have to go through the same series of stages although stage skipping is allowed. Examples of implementations supporting linear Pipelines are Intel TBB (see Section 2.2) and Cilk-P (see Section 2.3).

Non-linear Pipelines allow that each stage outputs more than one element. This allows to model *nested* Pipelines which are required by some benchmark instances like the *Dedup* [Bie11] benchmark of the Parsec suite. It gives more flexibility to express parallelism. At the moment only DAG scheduler like Nabbit and the Intel TBB *flow* interface support this.

The following sections provide in depth analysis of three well known Pipeline implementations.

2.2 Intel Threading Building Blocks

Intel TBB [Nav+09] provide an implementation supporting *linear* Pipelines.

A restriction in TBB before version 4 was that it only supported *linear*, *construct-and-run* (see Section 2.1.4) Pipelines where every stage had to produce the same number of output elements. Because of this limitation it is “hard but not impossible” [RCJ11] to model *on-the-fly* algorithms like *x264* with TBB. Additionally there was no way to model *cross edges* which made it hard to express *non-linear* Pipelines. This limitation has been lifted partially with the new *flow* construct introduced in TBB 4 [MRR12]. To limit the stack space usage, TBB provides a tunable argument limiting the amount of concurrent iterations.

To coordinate the workers which use a *bind-to-element* approach [Lee+13], TBB uses *work stealing* [Nav+09]. In a bind-to-element approach a worker tries to execute all stages for a single input chunk. It only switches to another input chunk when there are unsatisfied dependencies. A typical execution diagram can be seen in Figure 2.2.

2.2.1 Interface

Intel Threading Build Blocks offers three interfaces to create a Pipeline application [Ale07]. The oldest is C++ class focused whereas the newer interfaces take advantage of the *Lambda construct* [JF10] (see Section 3.3.1) introduced in the C++11 ISO standard. The newest interface is the *flow* interface which allows dynamic construction of task graphs with nodes and dependency edges during runtime. This interface can also be used to model Pipelines. Examples and detailed descriptions are given later in this section.

Pipeline stages are called *filters* in TBB and can be combined into chains. Each filter can either run in *parallel* or *serial*. A linear input stage would be modeled as a *serial filter*. For each filter the *input* and the *output* data type have to be specified and can not be changed during runtime.

During construction a throttle limit can be specified. This value caps the number of concurrently active iterations to limit the memory usage. It is *mandatory* and effects performance in case it is set too low.

C++ Class Interface

Each filter is a C++ class which has to inherit from the `tbb::filter` class. The concurrency is controlled by calling the parent class constructor with `parallel` or `serial_in_order` enum values. An example is given in Listing 2.1. The filters are added to and the execution is controlled by an instance of the `tbb::pipeline` class. Filters can not be modified or added once the execution of the Pipeline has started.

Lambda based Interface

This interface uses the construction method `make_filter` which specifies the concurrency of the stage with the first argument and accepts a *Lambda function expression* (see Section 3.3.1) as second argument. The concurrency is specified by two enum values (`filter::serial`, `filter::parallel`). The construction function makes use of C++

Intel TBB Pipeline class interface

```
#include "tbb/pipeline.h"
class Filter1 : public tbb::filter {
  Filter1() : tbb::filter( serial_in_order ) {} // serial stage
  // generatetokens
  void* operator()( void* token );
};
class Filter2 : public tbb::filter {
  Filter2() : tbb::filter( parallel ) {} // parallel stage
  // process tokens and output tokens
  void* operator()( void* token );
};
class Filter3 : public tbb::filter {
  Filter3() : tbb::filter( serial_in_order ) {} // serial stage
  // process tokens
  void* operator()( void* token );
};
tbb::pipeline ThreeStagePipeline; //Create the pipeline
ThreeStagePipeline.addfilter( new Filter1() );
ThreeStagePipeline.addfilter( new Filter2() );
ThreeStagePipeline.addfilter( new Filter3() );
ThreeStagePipeline.run(); //Run the pipeline
```

Listing 2.1: Code example showing the Intel TBB C++ class interface by Reed, Chen, and Johnson [RCJ11] with small modifications (also showing the parent constructor call which specifies the concurrency for the filter).

templates to specify the input and output data types of the filter. Listing 2.2 shows the definition and a Pipeline consisting of three filters (serial input, parallel, serial output). The number of concurrently active iterations is specified in the first argument to the `parallel_pipe` loop. In this example float numbers are passed between stages as result data. The actual work in the stages has been omitted in this example.

Flow interface

The new *flow graph* [Vos11] interface which is fully supported since TBB 4 can be used to model Pipelines more dynamically. It allows to model nodes and dependency edges between them. The chunks are passed from node to node with a message interface. The work can be expressed with Lambda functions as shown in Listing 2.3.

A major advantage over TBB Pipelines is that the flow graph interface allows multiple successor nodes. This allows the creation of *non-linear* Pipelines (see Section 2.1.5). This means that a stage with a single input can split the input into multiple output chunks which flow through the following stages. The *dedup* [Bie11] benchmark requires non-linear Pipelines for full parallelism.

TBB Pipeline Lambda interface - Definition

```
parallel_pipeline( max_number_of_live_tokens,
                   make_filter<void,I1>(mode0,g0) &
                   make_filter<I1,I2>(mode1,g1) &
                   make_filter<I2,I3>(mode2,g2) &
                   ...
                   make_filter<In,void>(moden,gn) );
```

```
parallel_pipeline( /*max_number_of_live_token=*/16,
     make_filter<void,float*>( /* serial stage void => float */
          filter::serial,
          [&](flow_control& fc)->float*{     |   float* - return type
               if( first<last ) {           |   (flow_control& fc) - parameter
                    return first++;         |   [&] - capture
                } else {                    |
                    fc.stop();              |  λ function
                    return NULL;            |
               }                            |
          }                                 |
     ) &
     make_filter<float*,float>( /* parallel computation stage */
          filter::parallel,
          [](float* p){return (*p)*(*p);} /* Lambda function */
     ) &
     make_filter<float,void>( /* serial stage float => void */
          filter::serial,
          [&](float x) {sum+=x;} /* Lambda function */
     )
);

```

Listing 2.2: Code example showing the construction of a TBB Pipeline with three filters (serial input, parallel processing, serial output). The Pipeline is constructed using the `make_filter` function which takes an input and an output type and a *Lambda expression* (see Section 3.3.1). The Lambda expression represents the work done in the filter. The capture `[&]` specifies that all variables in the calling scope can be accessed by reference in the Lambda function. It takes a pointer to a flow control object as parameter and returns a pointer of type **`float`**. The input stage generates numbers from one to a parameter value and the computation stage calculates n^2 for each input. The last stage sums up all the computed square values. Input and output data types for each filter are specified using C++ template arguments [Int15].

```
tbb::flow::graph g;
int sum=0;
function_node< int, int > input( g, 1 /* serial */, []( int v ) -> int {
  return v;
} );
function_node< int, int > parallel_compute( g, tbb::flow::unlimited, []( int v ) -> int {
  return v * v;
} );
function_node< int, int > stage_sum( g, 1 /* serial */, []( int v ) -> int {
  sum +=v;
} );

make_edge(input,parallel_compute);
make_edge(parallel_compute,stage_sum);
for(int i = 1 ; i < param ; i++) {
  input.try_put(i); //send message to the node
}
g.wait_for_all();
```

Listing 2.3: Intel TBB flow example creating a Pipeline consisting of two stages (serial, parallel) using the TBB 4 flow interface [Int11]. Concurrency for each node is specified in the constructor with a numeric argument between 1 and ∞ (`tbb::flow::unlimited`). Iterations are messages which are sent to the first node and make their way through the Pipeline. This example models the same Pipeline as in Listing 2.2 but using the flow interface. It calculates the sum of square values using three stages.

It allows dynamic edge creation during runtime which is not possible with the static TBB Pipeline interface mentioned before. This is an advantage which opens the possibility to port the x264 video encoder to the flow interface [RCJ11].

2.2.2 Implementation

In this section the implementation of the Intel TBB Pipeline *class interface* will be described. Our analysis is based on the source of the latest Intel TBB version 4.4 [Ale07].

Each TBB filter object has an input buffer (`input_buffer`) which is used to store tasks in an array. This array is of dynamic size and can grow during runtime. In case there is no free slot in the array, the grow operation doubles the size of the array until the space requirements are fulfilled. In the grow operation all task pointers in the array are copied to a newly allocated array.

The write access to the array of task objects is protected with a spin mutex. Also read access to the array is protected using a mutex.

Intel TBB supports user defined threads which execute Pipeline stages. This construct is called *bound threads*. These user defined threads us a semaphore in case the buffer is empty.

The throttle limit (see Section 2.1.2) is controlled by the `max_number_of_live_tokens ↩ variable`. It gets decremented when a new task is started and incremented again when the task finishes. If the counter reaches 0 no new threads are spawned. The computations are executed using normal blocking calls.

2.3 Piper

A further implementation of the Pipeline pattern is *Cilk-P* [Lee+13] with the *Piper* scheduler. Piper [Lee+13] is implemented using the *fork-join* semantics of Cilk. It uses a *bind-to-element* approach (see Section 2.1.4) combined with *work stealing* for communication between different workers. As of this writing Piper is not included in the official Cilk package but the source is publicly available[1].

The execution is controlled by a custom while loop which replaces the normal while loop. The Pipeline loop is related to a serial loop but executes the body in parallel. We discuss the necessary manual transformation of this loop in Section 2.3.3.

Cilk-P does automatic throttling in the scheduler to prevent uncontained memory growth caused by *runaway* Pipelines [Lee+13] (see Section 2.1.2).

2.3.1 Interface

The Pipeline interface of Piper is written in C. This makes the adaption of legacy application written in C easier. On the other hand an object orientated interface creates a much more readable structure. The main parts of the interface are:

Loop The interface consists of a new loop function called `pipe_while(bool ↩ running)`. This function executes stage$_0$ serially but spawns new threads for each iteration. This is necessary to not block the spawn of new iterations in case an older iteration blocks.

Wait Inside the loop the `pipe_wait(int stage)` function is used to create a cross dependency to the preceding iteration. This creates a serial stage. It is only possible to create wait dependencies between adjacent (directly preceeding) iterations. See Section 2.3.2 about limitations of Piper. Therefore the interface does not offer a parameter to specify the iteration in the wait method. Implicitly the previous iteration is taken as dependency target.

[1]Intel. *Piper: Experimental Language Support for Pipeline Parallelism in Intel® Cilk^{TM} Plus.* URL: https://www.cilkplus.org/piper-experimental-language-support-pipeline-parallelism-intel-cilk-plus (visited on 05/10/2016).

The dependencies between iterations are specified dynamically during runtime. E.g. Iteration I_i at stage S_n has to wait for its predecessor whereas Iteration I_j at the same stage S_n may continue without waiting. This enables the implementation of algorithms which behave differently in each iterations. One such example is the video encoder x264[2].

Continue The `pipe_continue(int stage)` function advances the current iteration to the given stage without waiting for predecessors. This creates a parallel stage. It is possible to skip stages dynamically.

As example a Cilk-P Pipeline program consisting of three stages is given in Listing 2.4. In this example a simple non-linear Pipeline is shown. During runtime for each input chunk at stage 1 it is decided whether it should be processed in parallel or if it should wait for the preceding input chunk. The first and last stage are serial stages.

2.3.2 Limitations

Piper allows only dependencies to directly preceding iterations. Dependencies like $I_i \rightarrow I_{i+n} | n > 1$ are not possible. This limitation is useful to hold memory guarantees. For some benchmarks like x264 this limitation requires special workarounds to transform the benchmark into a Pipeline. Lee et al. [Lee+13] used a Pipeline model with a huge number of stages to compensate this.

Another limitation is the lack of support for *non-linear* Pipelines (see Section 2.1.5). Complex non-linear Pipelines are required to model the Dedup benchmark. Lee et al. [Lee+13] used only a single level of input granularity in their Piper Dedup implementation. They skipped the stage from the Parsec implementation which splits chunks into course grain chunks. Their Pipeline implementation does not support nested Pipelines, which makes this modification necessary. The pseudocode for their Dedup implementation can be seen in Listing 2.3.

The interface does not provide a method to pass results between iterations. The user has to pass stage results by other means, like shared memory resources or broadcasts. This is a disadvantage because thread safe access to these results may require locks or additional logic. The result type between stages is not explicitly specified.

2.3.3 Implementation

An adapted Cilk scheduler and worker threads are used in Piper to execute Pipeline programs. The custom scheduler is necessary to handle throttle edges correctly.

As of this writing, the Cilk-P functions are not fully integrated into the Cilk language and therefore have to be *hand compiled*. This means that the new `pipe_while` loop

[2]Videolan. *x264 the best H.264/AVC encoder.* URL: https://www.videolan.org/developers/x264.html (visited on 10/13/2015).

```
#define STAGE1 1
#define STAGE2 2

// actual work is done in here
void work_stage0(int); void work_stage1(int); void work_stage2( ↩
    int);

pipe_while( !done ) {
  // Each iteration starts executing in Stage 0.
  iterarion++;
  done = work_stage0( iteration );
  if(!done) {
    if(decide( iteration )) { // on-the-fly dynamic Pipeline
      pipe_continue( STAGE1 ); // Advance to Stage 1 (parallel)
    } else {
      pipe_wait( STAGE1 );     // Advance to Stage 1 (serial)
    }
    work_stage1( iteration );
    pipe_wait( STAGE2 ); // Advance to Stage 2 (serial)
    work_stage2( iteration );
  }
}
```

Listing 2.4: Example of a dynamic (on-the-fly) Pipeline program written in Cilk-P [Lee+13] adapted from [Suk13]. The example consists of an input stage (stage 0) which is always serial in Cilk-P. It is followed by a stage which decides during runtime for each input chunk if it processed in parallel or has to wait for its predecessor. The last stage is a serial output stage (stage 2). This example only shows the creation of the Pipeline structure. The actual work is hidden in the `work_stage0-2` functions.

has to be hand replaced in the source with a Lambda function and an ordinary while loop which spawns a new thread for each iteration [Lee+13]. The result of this manual replacement is shown in Listing 2.5. Because of this modification Stage 0 (input stage) is always a serial stage.

During execution the program is transformed into a fork-join computation DAG. The last step adds cross and throttle edges to the computation DAG which can be executed using the Cilk scheduler.

Finally, the resulting DAG is a normal Cilk *fork-join DAG* with the following edge types:

serial edge from one instruction to the next instruction inside a strand (instruction sequence without parallel control instructions).

spawn edge after a *spawn* node to the first node of the newly spawned function.

continuation after a *spawn* to the first node after the *spawn* instruction.

Cilk-P Pipeline example

```
 1 #define STAGE1 1
 2 #define STAGE2 2
 3 #define STAGE3 3
 4
 5 // actual work is done in here
 6 void work_stage0(int); void work_stage1(int); void work_stage2( ↩
       int);
 7
 8 bool done = false;
 9 [&]() { // lambda function
10   while ( !done ) { // pipe_while
11     iteration++;
12     done = work_stage0( iteration );
13     if( !done ) {
14       cilk_sync; // ensure stage 0 is completed
15       cilk_spawn [ iteration ]() { //spawn
16         if(decide( iteration )) {
17           pipe_continue( STAGE1 );
18         } else {
19           pipe_wait( STAGE1 );
20         }
21         work_stage1( iteration );
22         pipe_wait( STAGE2 );
23         work_stage2( iteration );
24         cilk_sync; // wait for iteration to finish
25       }();
26     }
27   }
28   cilk_sync; // wait for Pipeline to finish
29 }();
```

Listing 2.5: Piper example after the `pipe_while` loop was manually transformed into a Lambda function and an ordinary while loop which spawns a new task for every iteration. Only the first stage has to be a serial stage. The original code can be seen in Listing 2.4. `cilk_sync` waits for subcomputations spawned in the enclosing function. The sync in line 14 ensures that stage 0 has finished computing. Line 24 ensures that the Pipeline iteration is finished. The last sync in line 28 ensures that all iterations of the Pipeline are finished.

return edge from the last node of a spawned thread to the node directly after the *sync* instruction.

The Piper scheduler adds virtual *throttle edges* [Lee+13] to prevent *runaway* Pipelines (see Section 2.1.2). Internally this is implemented using counters tracking the number of currently running iterations for each `pipe_while` loop. If a worker tries to start a new iteration and the throttle limit is already reached the iteration is suspended.

2.3.4 Performance

Lee et al. shows that the implementation is *asymptotically efficient* with an expected runtime of (with high probability $1 - \epsilon$, where ϵ can freely be chosen):

$$T_1/P + \mathcal{O}(T_\infty + \underbrace{\lg P}_{\text{work-steal contention}} + \lg(1/\epsilon)) \tag{2.1}$$

Where T_1 is the runtime of all nodes in the Pipeline DAG, P is the number of used computation cores and T_∞ is the computation time of the longest path. The additional $\lg P$ term arises because multiple processors may try to steal work from the victim's queue concurrently.

Lee et al. have shown space bounds for Piper to be:

$$S_P \leq \underbrace{PS_1}_{\text{P times serial space}} + \underbrace{PfDK}_{\text{nested factor}} \tag{2.2}$$

Where

S_P is the stack space of a parallel execution,

S_1 is the stack space of a serial execution (executed with only one computational core). This means that only one input instance has to be kept on the stack at any time,

P is the number of used computation cores,

f is the frame size which represents the size of the *activation record* and *local variables* of a worker,

D is the number of *nested Pipelines* and

K is the *throttling limit* restricting the amount of concurrent iterations.

The proof uses *trees of contours* where a contour is a path in the DAG which only contains serial and continue edges. The contours are organized in a tree, representing a spawn hierarchy. Space usage is represented by the frame size of all contours where a node is currently in a workers queue or where the earliest not yet executed node is not ready (suspended). The proof is given in detail in [Lee+13].

2.4 Nabbit

Nabbit [ALS10] provides a library for executing task graphs with arbitrary dependency edges. It implements the DAG pattern which enables the execution of arbitrary task graphs on a stricter scheduler.

For details about the pattern and the implementation see Section 1.4.

Task graphs can be used to model Pipeline programs therefore the interface is discussed here.

2.4.1 Interface

The interface models each task as a C++ class. This means for a Pipeline program every stage has to be modeled as a dedicated class. A special `init` method is called by the library on every class instance. In this method new dependencies can be added. This method is called during runtime and allows the implementation of *on-the-fly* Pipelines (see Section 2.1.4).

The execution of the DAG is started by calling `Execute()` on the first task. The first task has to be an artificial start node enabling all iterations at stage 0. This function blocks the execution flow until the entire DAG has been executed.

An example of a Pipeline program implemented in Nabbit is given in Listing 2.4. The program consists of three stages (serial, parallel, serial). The example shows only the creation of the Pipeline structure. The actual work for each Pipeline stage is hidden in the `workStage0`, `workStage1` and `workStage2` functions. In our example the nodes are identified using a string of the schema `iteration@stage`.

2.4.2 Limitations

Nabbit is capable of executing arbitrary DAGs which makes it difficult to implement effective throttling. Because arbitrary DAGs allow a more complex non-linear structure than the Pipeline Pattern allows, delaying a task may lead to delayed executions. Particularly the limitations of only backward dependencies in the Pipeline pattern makes the throttling easier.

The class interface requires a lot of *boilerplate code* which makes the source code hard to read. The example in Listing 2.6 which models a Pipeline with only three stages requires 46 lines of source code.

```
class Start : public DAGNode { ... }
class Stage0 : public DAGNode {
  int iteration;
  Stage0(int i_) : iteration(i_) { }
  void Init() {
    this->AddDep("start");
    if(iteration>0) {
      //Add Dependecy to stage 0 of the preceeding iteration
      this->AddDep(iteration-1 +"@" + 0);
    }
  }
  void Compute() {
    workStage0(); //serial input stage
    new Stage1(iteration);
} } };

class Stage1 : public DAGNode {
  int iteration;
  Stage1(int i_) : iteration(i_) { }
  void Init() {
    this->AddDep(iteration +"@" + 0); //Dependecy to stage 0
  }
  void Compute() {
    workStage1(); //parallel stage 1
    new Stage2(iteration);
} } };

class Stage2 : public DAGNode {
  int iteration;
  Stage2(int i_) : iteration(i_) { }
  void Init() {
    this->AddDep(iteration-1 +"@" + 2); //Dependecy to prev. ↩
       iteration
  }
  void Compute() {
    workStage2(); //serial output stage
} } };

int main() {
  int iter++;
  Start start;
  while(!stop) {
    Stage0 *iter = new Stage0(iteration);
  }
  start->Execute(); //blocks
}
```

Listing 2.6: Pipeline program created using the task graph library Nabbit [ALS10]. The example consists of a serial input stage followed by a parallel processing stage. The last stage is a serial output stage. This example only shows the construction of the Pipeline structure. The actual work is hidden in the `workStage0()`, `workStage1()` and `workStage2()` functions.

CHAPTER 3

Pheet Pipelines

3.1 Introduction

The Pheet framework [Wim14] currently lacks an implementation of the Pipeline pattern. In this thesis we created a competitive implementation. This chapter gives an overview of design decisions, interface and synchronization details. Also data structures created for our implementation are explained in detail. We also argue that our implementation is lock-free.

3.2 Goals and Design decisions

Based on the analyzed related work we propose the following Pipeline design:

- The design should allow *on-the-fly* and *non-linear* nested Pipelines (see Section 2.1.5). Both features are necessary to implement all benchmarks in a Pipelined version.
- An interface which allows *short* Pipeline definitions. No boilerplate code should be contained in Pipeline definitions. Adapting legacy applications should only require few changes.

  Support for passing *results* between dependent iterations should be provided by our design. Having a tested and fast way to query and store results removes complexity from user-code. The interface should be type safe to reduce the likelihood of programming errors.
- Our design should be scalable even on systems with more than 16 cores. Therefore centralized locks are avoided to reduce contention on many-core systems. *Lock-free* data structures and synchronization is used to make our implementation scale well.

In case *active waiting* is not avoidable it should be reduced to a minimum extent. The scalability will be measured and compared on our test systems.

- Even for very *lightweight tasks* the overhead of the Pipeline creation should not be dramatically high. It should feature a comparable performance to a Pthread implementation of a given problem.
- The *memory system* should not be stressed even on many-core systems. Centralized data-structures should only be used where unavoidable. Memory should be freed as soon as possible to support long running Pipelines.

  The amount of running iterations should be controlled by the user to limit memory usage. This *throttle limit* should be tunable during runtime.
- No additional requirements other than Pheet support should be necessary to compile Pheet Pipelines. It should be compilable on every platform supporting the hwloc library [Bro+10] and providing a C++11 compiler.
- *Performance characteristics* about the execution should be recorded by our framework to allow optimization of programs using our Pipeline implementation.

3.3 State of the Art Interfaces

The programming interface for Pipeline modelling is a key factor for readability and maintainability of source code. The new Pipeline interface should be easy to use and allow good adaption of legacy applications. Additionally it should provide a built-in mechanism to pass result data between stages and dependent iterations.

Most of the adapted benchmarks are legacy software written in C. Nevertheless it was chosen not to provide a C interface to the Pheet Pipeline because the whole Pheet framework is written in C++. Therefore all benchmarks have to be compiled with a C++ compiler to use the new Pheet Pipeline interface. For some benchmarks, particular those with a good modular design, linking of existing object files was possible.

Another important consideration is control of the number of concurrently running iterations. This topic has been in discussed in Section 2.1.2. Intel TBB [Ale07] refers to this issue with the term *live tokens* whereas Piper [Lee+13] uses the term *throttling*. Too many uncompleted but started iterations may waste memory space. The design goal is to allow dynamic modifications of the upper limit based on the maschine's current resource allocation.

The interfaces of Intel TBB (Section 2.2.1), Cilk-P (Section 2.3.1) and Nabbit (Section 2.4.1) have been discussed in previous sections. In this section we give a short summary of advantages and disadvantages of these interfaces. Then we present our Pheet Pipeline interface mitigating some of the mentioned disadvantages.

```
flowcontrol fc; // outer scope
auto f = [&](param& p)->float*{
            if( first<last ) {
                return first++;
            } else {
                fc.stop();
                return NULL;
            }
};
f(p1);
```

```
  float* - return type
  (param& p) - parameter
  [&] - capture
λ function
```

Figure 3.1: This example shows the definition of a named C++ Lambda function. The capture `[&]` specifies that all variables in the calling scope can be accessed by reference in the Lambda function.

3.3.1 Lambda Functions

Lambda functions are a C++11 construct for "local unnamed functions which can be defined on-the-fly" [JF10]. Lambda function can also be assigned a name and can be used similar to a function pointer.

The main difference to a function pointer is that the Lambda function allows to use variables from the enclosing scope. Variables in the enclosing scope which are used inside the Lambda body have to be listed in the *capture list* in the Lambda declaration. Variables must be locally defined to the current scope and can either be passed *by value* or *by reference.*

An example with a named Lambda function can be seen in Listing 3.1. This Lambda function takes a single parameter of type `param` and returns a **`float`** pointer.

These functions can be used to create very short (no boilerplate code) Pipeline interfaces, because they can effectively use variables of the enclosing scope and can be defined locally inline.

3.3.2 Intel TBB

All the mentioned Intel TBB interfaces are written in C++ and force the specification of an input and an output data type for each stage. The specification of input and output types can be considered as advantage because it takes the burden from the user to pass results from stage to stage. One major drawback of the interface is that there is no interface to pass results from iteration to iteration.

The first two mentioned TBB interfaces (Pipeline and Lambda based) only support *construct-and-run* Pipelines where the structure can not change during runtime. Every input chunk has to follow the exact same path from the first filter to the last filter which implies that *no filter can be skipped* dynamically.

Compared to the traditional TBB Pipeline interface the new flow interface allows more than one output value per stage. This is a major practical advantage for some benchmark problems which require non-linear Pipelines, most noteable the *Dedup* Benchmark [Bie11; RCJ11].

The upper limit for concurrently running iterations can only be set during startup and can not be changed dynamically. This makes dynamic user defined adaptions during runtime impossible [Int15]. For more details about the interface see Section 2.2.1.

3.3.3 Cilk-P (Piper)

The Cilk-P C interface supports on-the-fly Pipelines. Throttling is implemented in the Piper scheduler. The decision between serial and parallel stage can be done per iteration during runtime.

The interface does not provide a mechanism to pass results between iterations. There is no clear specification of input and output types for stages.

There is no support for non-linear Pipelines. Non-linear Pipelines like nested Pipelines are important to port the Dedup benchmark to a Pipeline model. Lee et al. [Lee+13] ported the Dedup benchmark by omitting one nested Pipeline stage.

Only adjacent (directly preceding) iterations can form dependencies. This is a limitation for benchmarks like x264 but allows to infer strong memory-bounds.

Several manual transformation steps have to be conducted to make the while loop compilable. Further details can be seen in Section 2.3.1.

3.3.4 Nabbit

The Nabbit interface encapsulates each task into a class. Because the interface is built-in Cilk++, C++ is required as programming language. This may pose more work on porting legacy applications to Nabbit.

Tasks and dependencies can be created during runtime on-the-fly. With this dynamic approach also benchmarks like x264 can be implemented. The interface was designed for DAG execution and is not ideally fitted for Pipelines.

The interface requires a lot of boilerplate code because of the C++ class definitions. It requires about 60 lines of code even for a simple three stage Pipeline.

For further details see Section 2.4.1.

3.4 Pipeline Interface Design

This section shows how the Pipeline design decisions formed our C++ Pipeline interface.

3.4.1 Overview

The Pheet Pipeline interface is object orientated and realized in C++. It consists of two important classes:

- The *Pipeline Environment* object is used to control the Pipeline. It keeps track of all the running instances and only exists once for each Pipeline. Also the *throttling* is performed by this class.
- For each iteration a *Pipeline Iteration* object is created. This object controls the stages. It is used to advance the stages and takes care about stage results.

  For nested Pipelines this class takes care about nested children and ensures that all objects are freed when they are not needed any more.

Listing 3.1 shows a basic Pheet Pipeline example. In this example a Pipeline with two stages is created. The first stage is a parallel stage and the second stage is a serial stage. This example only models the structure of the Pipeline, no work is performed in the stages. Every stage is identified by an integer number.

In our implementation the work of each stage has to be modeled as a dedicated Lambda expression. This is necessary because the Pheet scheduler does not allow tasks to wait or use locks. A blocking task blocks the whole worker thread and slows down the whole Pipeline computation.

To ensure termination of all Pipeline iterations the actual Pheet Pipeline loop is wrapped inside a block. In this block a Pheet Finisher Hyperobject ensures that all spawned tasks have finished. Our classes require the C++ **`typename`** prefix, but for better readability it is omitted in all examples. This prefix is required because of the customizable template architecture of the Pheet framework.

3.4.2 Pipeline Iteration

Each iteration is represented with an object instance of type `PipelineIteration`. It embeds the state and all stage result values. A unique numeric id is used to identify iterations. This id has to be created by the user. In Listing 3.1 line 13 a `counter` is used to create unique ids in the Pipeline loop.

Two special functions are used to advance the current stage. A `pipe_continue` call is used to advance in parallel while `pipe_wait` waits for the preceding iteration to finish the required stage. We named our functions the same way as Lee et al. named their Cilk-P interface [Lee+13].

Parallel Stage

A parallel stage is created using the `pipe_continue` method. Figure 3.2 gives a graphical overview of the method call. The `pipe_continue` call takes the following three arguments.

1. The *stage number* to proceed to. Stages have to be monotonic growing, but stage skipping is possible. *Stage skipping* is an important property for on-the-fly Pipelines (see Section 2.1.4).
2. A *result value* of the completed stage. The value type is specified using template arguments. The programmer has to provide a special *null value* called *null trait* in case he does not want to pass results from iteration to iteration. Null traits are explained in detail in Section 3.5.7.
3. A *Lambda* expression representing the code of the next stage. Lambda expressions have been chosen because they allow easy *capturing* of variables but can also be used as tasks in the Pheet scheduler. Section 3.3.1 gives details about C++ Lambda functions.

Serial Stage

The `pipe_wait` method creates a serial stage where the current iteration has to wait for a given preceding iteration. Compared to the continue call `pipe_wait` takes an additional optimal argument:

4. An *iteration id* specifies the iteration to wait for. In combination with the *stage* parameter the iteration has to wait until the specified iteration finishes the specified *stage*. In case the parameter is omitted, the previous iteration is used.

   The Pipeline pattern restricts wait dependencies to go only backward. Therefore only ids of preceding iterations can be passed to this function.

3.4.3 Nested Pipelines

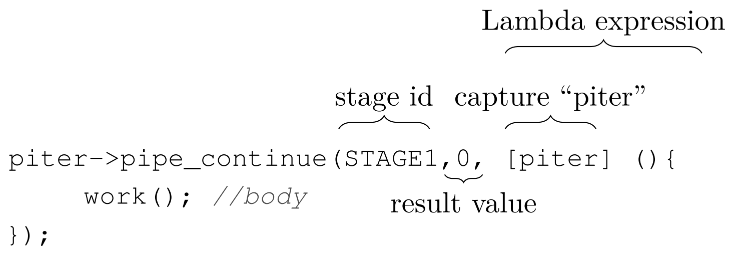
To allow *non-linear* Pipelines we added the possibility of *nesting* to our interface. With *nested Pipelines* a single iteration can have multiple outputs at a stage, which is equal to creating multiple child iterations. Nevertheless we do not weaken the other restrictions of the Pipeline DAG, like only backward edges.

This is necessary to fully exploit parallelism for problems like Dedup (see Section 4.5). The interfaces of Cilk-P (Piper) and Intel TBB do not support nested Pipelines.

To create a child iteration a call to `gen_nest_iteration(child_id)` on the current iteration object is required. Usually this method is called in a loop to create all the needed child iteration objects. The numbering schema of the subcomputations follows

```
 1 #define STAGE1 1
 2 #define STAGE2 2
 3 // result datatype:int bock-size: 1024
 4 typedef PheetPipelineEnvironment<Pheet,int,1024> PipelineEnv;
 5
 6 PipelineEnv penv { ppc };
 7 {
 8   //Finisher Hyperobject to ensure termination
 9   typename Pheet::Finish f;
10   size_t counter = 0;
11
12   while( ... ) {
13     PipelineEnv::PipelineIteration* piter =
14       new PipelineEnv::PipelineIteration(&penv, counter++ );
15
16     piter->pipe_continue( STAGE1,0, [piter] () {
17       // STAGE1 is executed in parallel
18       piter->log( "stage 1" ); //create a debug message
19
20       piter->pipe_wait( STAGE2,0, [piter] () {
21         // STAGE2 waits for the preceeding iteratoin
22         piter->finished( 0 ); //iteration completed
23       });
24     });
25   }
26 }
```

Listing 3.1: A simple Pheet Pipeline example with two stages. The *Finisher* Hyperobject in line 8 is required to ensure termination of all spawned iterations. New `PipelineIterations` have to be created in the main thread with a monotonic growing user supplied id. The stage is advanced by calling `pipe_continue` with the next stage id. `pipe_continue` starts a parallel stage where the work is executed concurrently, whereas `pipe_wait` creates a serial stage which maintains serial iteration order. An iteration is marked as finished by calling the `finished` method (line 21). **`typedef`** in line 3 is recommended to define template arguments for the environment. See Section 3.5.2 for details about the Pipeline Environment class.



```
piter->pipe_continue(STAGE1,0, [piter] (){
     work(); //body
});
```

Figure 3.2: The *continue* call takes three arguments: The numeric id of the *next stage*, the *result value* with the data type depending on template arguments and finally the *code* for the next stage in a Lambda expression. In the Lambda expression a copy of the Pipeline iteration pointer (`piter`) is made available using the *capture*. See Section 3.3.1 for an introduction to the C++ lambda construct. The arguments are passed using the capture list, no parameters `()` are accepted by the Pheet Pipeline system for the Lambda function. Code or function calls are contained in the Lambda expression body.

the same system: For each iteration the sub iteration ids start with zero and need to grow monotonically.

This leads to a situation where not all iterations of a given stage follow a global monotonic numbering schema. To address this issue a new hierarchical addressing scheme for iterations is proposed. This proposal is shown with an example in Figure 3.5. The address of the second child of iteration number 3 would be $3 \to 2$. The third sub-iteration of this iteration would be identified by $3 \to 2 \to 3$. Figure 3.5 visualizes this. Iteration $3 \to 1 \to 5$ may be the predecessor of $3 \to 2 \to 1$. The implementation ensures that wait calls wait for the right preceding iteration.

3.4.4 Throttling

Throttling is used to limit the amount of concurrent active iterations. Our Pheet Pipeline implementation relies on the user to specify the desired iteration limit. The limit is not static and may be changed by the user during runtime.

To apply a throttle limit, a call to the Pipeline Environment function `throttle( ↩ limit)` must be inserted in the main Pipeline loop. In case the limit has been reached this function call executes other ready iterations until the number of running iterations decreases. This can not deadlock the Pipeline because iterations are only allowed to have backward dependencies and only newly created iterations are throttled.

In nested iterations obeying a global throttle limit may be suboptimal. One solution would be to define the throttle limit per stage. This topic is left open for further research.

3.4.5 Logging

Log messages can be printed using two methods:

The first method is to invoke a log function in the Pipeline Iteration class. `piter ↩ ->log("output ",4711)` prints "`iter 55 stage 3:output 4711`" when executed in iteration 55 in stage 3. This method accepts a variable number of arguments.

The second method is a log function in the Pipeline Environment class. `penv->log( ↩ "string")` directly prints text and ensures thread safety. This method only accepts a single `std::string` as argument.

3.4.6 Disadvantages of the Interface

Our interface features several disadvantages which could be addressed by future work:

- Nested *Lambda* functions lead to convoluted code. Each Pipeline stage has to be defined in an own indented block to allow *on-the-fly* Pipelines.
- Our implementation only supports a *single results data type* for all stages. As a workaround we proposed the use of C++ union data types to pass results, which on the downside are not type safe.
- *Null traits* have to be defined manually by the user to allow our current implementation to track null values.
- The *throttle* functionality is not included directly in the scheduler. This requires the user to manually insert a call to the `throttle` function at the end of the Pipeline while loop.

3.5 Pipeline Implementation

This section explains the implementation of the three main data structures used in our Pheet Pipeline implementation to model and control Pipeline executions.

- *Pipeline Environment* object is used to control the execution of the Pipeline. It performs cleanup tasks after execution and has some sanity checks built-in to check if the Pipeline has been executed successfully. Additionally it provides logic for thread-save debug output and keeps track of performance data.
- A *Pipeline Iteration* object represents a single iteration in the Pipeline. It keeps track of its own state and checks the state of adjacent iterations in case it has to wait for dependent iterations. The interface of the Pipeline Iteration object has been discussed in Section 3.4.2. This class contains the synchronization logic of the Pipeline.
- *Blocks* are used in multiple places in our implementation. They store data in a given (gap) free order and provide a combination of very fast and lightweight access and dynamic growth.

These classes can be found under the *primitives* folder of the Pheet framework. The source is available on request by contacting the author under bernhard.redl@vishap.at. Our implementation is part of the Pheet framework and is built with the Pheet framework by executing the `make` command.

3.5.1 C++11 atomics

We used the atomic data type of C++11 to handle synchronization. The advantage is that atomics have less overhead than mutex locks and allow wait free data structures.

The most important two operations on the atomic data type are *load* and *store* which both allow to specify the memory order using an optional argument. In our context the most important memory orders are [ISO12]:

sequential consistency which is the default if no memory order is specified. This memory order creates a single total order of all threads observing the value. No reordering with other atomic instructions is allowed. We use this memory order to formulate all our critical sections and to perform synchronization.

relaxed is the most loose memory order. It only guarantees atomicity, but does not guarantee any ordering constraints.

Additionally atomics support an atomic *compare and swap* operation (short CAS) which replaces a given value only if the old value matched the expected value. A compare and swap operation is a very expensive operation and may produce a *cache miss*.

3.5.2 Pipeline Environment

Iterations are registered and managed by the central `PipelineEnvironment` entity. After Pipeline Environment runs out of scope, memory is freed. In case the mandatory Finisher Hyperobject had been omitted by the user, sanity checks raise a fault. Additionally it provides a mutex lock to serialize *debug log messages* generated by concurrently running iterations.

The Pipeline Environment is also used to perform throttling (see Section 2.1.2). Because our interface does not feature a special *loop* function the throttle function has to be called manually by the user during each loop iteration. The throttle logic uses an user supplied parameter to limit the number of running iterations. If more iterations are active, ready stages of existing iterations are executed to delay the execution of the main loop until the throttle limit is ensured again. To prevent fast flipping of the state the main loop can only continue when the current iteration count is 20 % below the throttle limit.

The number of currently active iterations is measured using the atomic `running` member variable. It is incremented by Pipeline iterations during startup and decremented when iterations finish. To prevent negative scalability effects of this centralized variable, it is fetched with *relaxed memory order* (see Section 3.5.1). It is possible that there are more active iterations than the throttle function observes, because of the relaxed read operation. Therefore the throttle limit is not a hard limit.

3.5.3 Template arguments

C++ template arguments were used to allow the user to specify data types of result values and performance values during runtime. Template arguments allow type safe code which is easier to maintain.

To simplify the usage of the environment class and to specify template arguments a **`typedef`** as shown in Listing 3.1 line 3 is recommended. The Pipeline Environment takes three template arguments.

```
PheetPipelineEnvironment<Pheet,int,1024>
```

1. The first arguments specifies the *Pheet scheduler*. Pheet supports several schedulers which can be plugged in (see Section 1.2). Only schedulers which have been slightly modified to allow direct task execution are supported by our Pipelines. This modification is necessary to allow the execution of ready tasks during the *throttling phase*. See Section 2.1.2 for more details about throttling.
2. The *data type* of Pipeline stages result values. Each iteration can store results for every completed stage. Results of stages in any iteration can be queried with this interface. See Section 3.5.7 for details.
3. The *block-size* specifies the size of internally used blocks which store the iteration objects and stage result values. See Section 3.5.4 for details about our block data structure.

3.5.4 Blocks

The number of iterations and stages is not known a priori and is only limited by the amount of available memory. All iterations have to be stored in a central data structure to allow wait dependencies between iterations. The interface to retrieve results of dependent iterations benefits from a central data structure containing iterations.

This data structure requires the following properties:

- Access to list elements should not get slower when the amount of elements increases.
- Read operations to neighbor elements should be possible in constant time. The key for a given element is a strictly increasing number. No gaps are allowed in this sequence. Read operations are performed concurrently.
- Write operations to a given list element happen only once and are performed only by a single thread. No incomplete value is ever read because only backward dependencies are allowed.

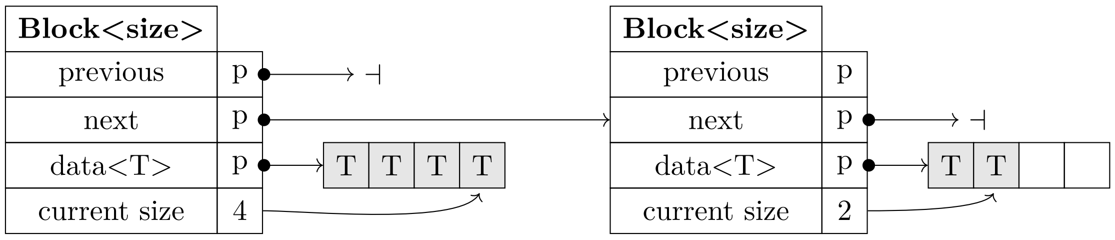


Figure 3.3: Shows the *block chain* used to store iteration instances and stage results. The *size* and the content data type are defined by template parameters. The keys to access elements are numeric values which have to be a gap-free sequence starting from zero. The *data* is stored in constant size plain C++ arrays. The pointer to the last written position is stored in *current size*. The blocks are connected in both directions with pointers (*previous*, *next*).

To meet these requirements a doubly linked list consisting of blocks was implemented. We call this list *block chain*. A single block is show in Figure 3.3. The block size is defined by a template parameter and has to be consistent for a given block chain. The data type for elements is specified using template arguments. The block chain is visualized in Figure 3.3. In this example the block-size is four and two blocks store in total six elements in this block chain.

The Pipeline Environment keeps all iterations in a block chain. Each iteration knows the parent block it is contained in. In most Pipeline programs iterations only wait for the *directly preceding* iteration. Therefore a direct array access is performed to find adjacent iterations taking $\mathcal{O}(1)$ time on average.

Access across blocks requires reading the whole block chain. To access an element by id without a direct block reference, the block chain has to be searched from the start. Only the next pointer of every block has to be read. Once the right block reference has been obtained a direct array access reads the element.

Access to iterations far away is uncommon and therefore no time was spend to optimize these lookups. This operation takes $\mathcal{O}(n)$ time where n is the number of blocks in the block chain. The number of blocks and iterations have a linear relation of $n \approx |I|$.

For future work new blocks could be allocated with the double size of the previous block to achieve asymptotic time $\mathcal{O}(\log_2(|I|))$ where $|I|$ is the number of iterations.

This optimization is not realized in the current implementation because random access is very uncommon for our benchmark instances. In the most case (read of adjacent iterations) the block reference is already known resulting in $\mathcal{O}(1)$ time to read an element.

The block-size is specified using template arguments to the Pipeline Environment class. This value depends highly on the problem size and can be optimized by the user. Although in practice we did not experience big performance differences for different block-sizes values between $1024 - 4096$. In the current implementation the block-size of the stage result block is set to block-size/8. This initial value fits well for most problem instances which have more iterations than stages. In case memory overhead should be limited a lower value may be considered. It is trivial to implement the block-size of the stage results as template argument. This feature is not implemented in the current version.

3.5.5 Pipeline Iteration

In Pheet Pipelines a dedicated object is used to represent an iteration. All iteration objects have to be initialized in a single thread with strictly increasing, gap free, iteration ids.

The interface of the Pipeline Iteration was explained in Section 3.4.2.

In the following a detailed explanation of each field will be given. The fields can be seen in Figure 3.4.

id Identifies the iteration using a numeric id. Because the iteration number is defined by the user and can not be changed after object initialization, it is stored as a **`const`** ↩ **`int`**. No concurrent reads can occur during object creation because dependencies can only go backwards and iteration objects creation is done in a single thread.

stage The stage number represents the current stage of the iteration. Each stage is identified by an integer number. It is monotonically growing during execution. The stage number is read by multiple threads concurrently. Only the thread executing the iteration is allowed to update the stage variable. Therefore a simple `std::atomic<size_t>` is used to store the current stage.

results The stage results are stored in a block chain. The result data type is specified using template arguments (denoted `T` in Figure 3.4). Result values are only written by the executing thread and never change afterwards. Therefore our block chain with atomic members is sufficient. The implementation does not check if for a given stage actual data exists. In case a stage is skipped, positions in the result array are filled with null values.

As an optimization the result block chain is not allocated at the iteration object construction. It is only created when the first stage has completed and a *non-null* result value is saved. If the user does not need result passing he can pass a null value and the block chain is never allocated. Null traits allow the user to specify type safe null values. See Section 3.5.7 for details.

parent block Each iteration has a reference to the *parent block* of the iteration block chain in the containing Pipeline Environment. This allows the iteration to have direct access to neighbor iterations. In most benchmarks, iterations only have to wait for direct predecessors, making this an important optimization. Access to direct predecessors is a simple array access for all but one iterations of a block. Only the first iteration of a block has to perform an additional lockup to get the address of the preceding block to find its predecessor.

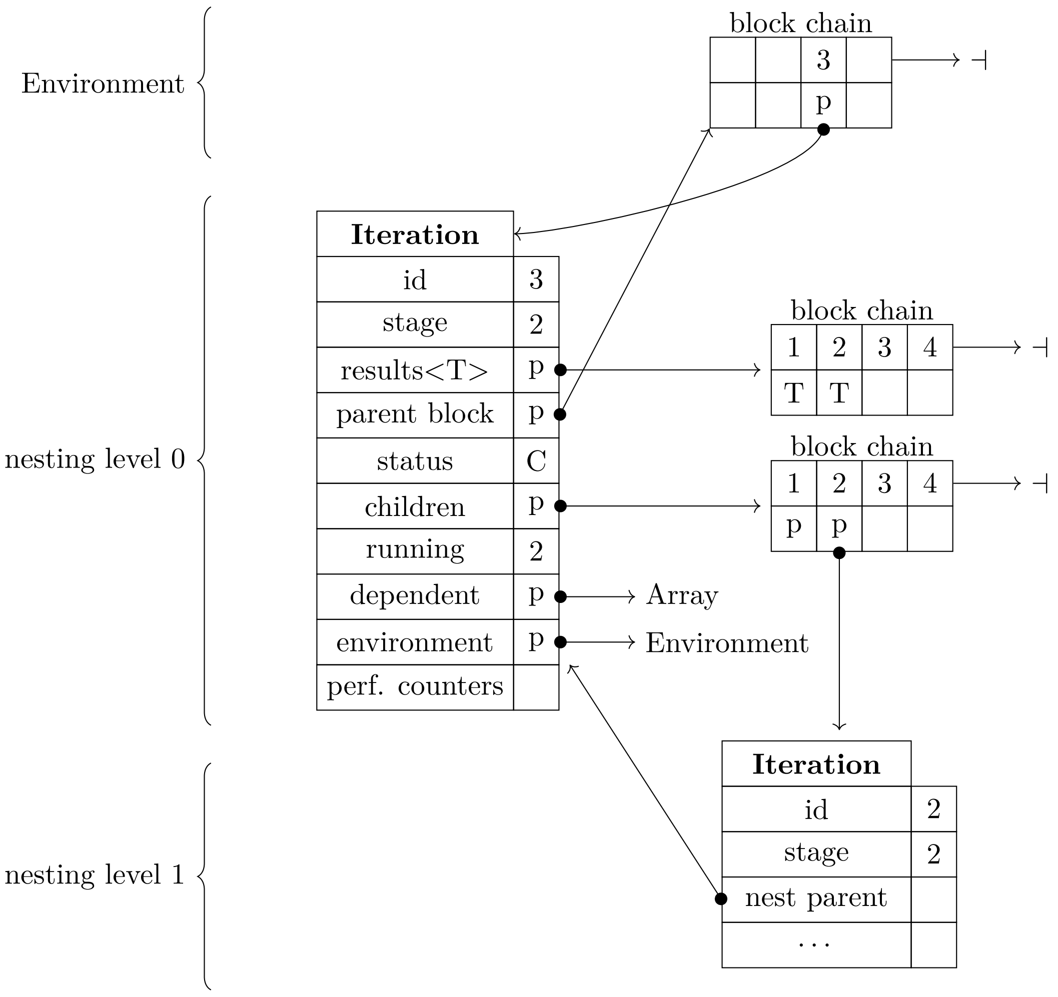


Figure 3.4: Shows the fields of a Pipeline iteration object. For each iteration in the Pipeline an object instance is created. The *id* and current stage of the iteration are stored in the object. Other iterations have to check this *atomic* field to obtain the current stage. *Results* are managed by iterations and stored in blocks. The type of results is decided by the user (type T). Each iteration has a direct reference to the *parent block* it is contained in. This allows fast access to direct predecessors. The *status* of iterations specifies if the iteration is already finished, if it is active or if nested sub-iterations are running. These nested sub-iterations are stored in the *children* block list. The *dependent* array is used by other iterations to get notified when the iteration advances to the next stage. Various *performance counters* are used to record performance characteristics.

status The status tracks the state of the current iteration. It can have the following states:

- `ACTIVE` iterations have been initialized and started. They can advance to following stages using either wait or continue calls.
- `CHILDREN` is the state which denotes that the iteration has spawned child iterations at the next nest level. This creates a non-linear Pipeline (see Section 2.1.5). Iterations in this state can not advance their stage, they can only create child iterations. They stay in this state until all nested children have been created. Then the iteration is changed to the `CHILDREN_WAIT` state with a `finished()` call.
- `CHILDREN_WAIT` is the state for iterations which have finished creating nested children. In this state the iteration waits till all its children have finished.
- `FINISHED` is the state reached by an iteration without children after terminiation (`finished()` call). In case an iteration has children, the last child iteration changes the parent's state to `FINISHED`. The children use the *running* counter of their parent iteration to check if other children are still running.

running is used only in nested Pipelines. It tracks the number of currently running child iterations. This counter is used to check when the last children finishes. The last children marks the parent iteration as finished. Because the value is updated by multiple threads a `std::atomic` is used.

dependent array tracks dependent iterations which are waiting for this iteration to proceed. Iterations which depend on this iteration and can not proceed until it reaches the required stage, can put their reference into the *dependent* array. Once the iteration advances, it checks its dependent array for waiting iterations. If the iteration reached the stage the dependent iteration was waiting for, it enables the waiting iteration.

environment points to the `PipelineEnvironment` object. As explained in Section 3.5.2 the environment is used to synchronize debug output and keeps a list of all iterations of nesting level 0. Iterations at nesting level 0 which try to access other iterations outside of their own block have to use the block chain of the environment to get the reference.

performance counters are the last member of the Pipeline Iteration class. They are used to gather multiple performance values of the Pipeline execution. See Section 3.9 for details.

3.5.6 Nested Pipelines

To support nested Pipelines iterations are organized in several layers. Figure 3.5 gives a graphical representation of a nested Pipeline. Iterations of nesting level 0 are stored in

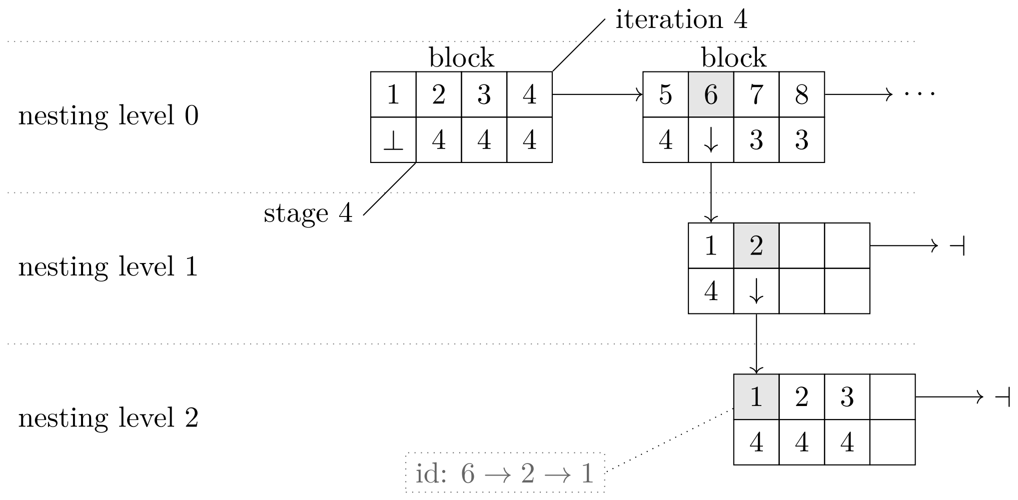


Figure 3.5: Example of the Pheet block data structure to manage nested iterations for non-linear Pipelines. The first row shows the iteration number and the second row shows the current stage for the iteration. For example in the first block in the top level (nesting level 0) iteration 2, 3 and 4 are all at stage 4. ⊥ denotes iterations which have finished already. ↓ indicates that this iteration has created nested sub-iterations. The numbering schema for sub-iterations starts again with iteration 1. The combined unique iteration id is shown in grey (e.g id: 6 → 2 → 1).

the block chain of the Pipeline Environment object. Iterations of nesting level 1+ are stored in block chains of their parent iterations.

Each block chain starts the numbering at one. To allow unique identification of iterations a hierarchical iteration id is proposed. Details of this id can be seen in Figure 3.5.

Nested iterations use the special states `CHILDREN` and `CHILDREN_WAIT` which are explained in Section 3.5.5.

3.5.7 Result types

Each iteration can store a result value for every completed stage. The type of result values is specified with template arguments to the Pipeline Environment class. Only a single result type for all Pipeline stages can be specified.

In case different stages have different output types, a C++ **`union`** data type is recommended. Union is a data type which holds only one of its members at a time. One member for every stage data type should be created in the union. The disadvantage of this method is that it is not type safe.

To query results of an arbitrary iteration the iteration reference has to be obtained by querying the Pipeline Environment using the `find` method. With the correct iteration reference the `get_result(stage)` method returns result values.

```
template <>
class nullable_traits<int > {
public:
  static int const null_value = 0;
};
```

Listing 3.2: This example shows the *null trait* definition for the **`int`** data type. Null traits are template constructs allowing type safe *null* values. They are specified by the user. The Pheet Pipeline implementation uses `nullable_traits` from the Pheet framework [Wim13].

There are no sanity checks in the implementation to ensure that the iteration already has finished the stage. Although it is safe to call the `get_result` method on iterations after waiting for them with `pipe_wait`.

For the specified result data type a *null trait* has to be created by the user. This trait allows Pheet Pipelines to detect *null* values in a type safe manner. The Pheet framework provides default null traits for all C++ pointer data types. An example of the null trait for the data type **`int`** can be seen in Listing 3.2.

No memory is allocated for results in case the user stores only *null values*.

3.6 Memory Management

Iterations are stored in *blocks* of fixed size. Once a block only contains iterations in a *finished* state it is ready to be freed. Iteration objects store the results of completed stages. Once a block is freed these results are not available any more.

To circumvent this issue the user can specify a delayed deallocation of blocks. The minimum delay is one block to correctly handle wait calls from the begin of the following block.

Two dedicated variables are required in the block data-structure to handle the memory management correctly.

- `finished` is an atomic counter which is incremented by every finished iteration. Once an iteration reads `finished` equal to the block-size it calls to free the block.
- Multiple threads may experience the finished variable equal to the block-size simultaneously. It is critical that each block is only freed once. To ensure mutual exclusivity a compare and swap operation is performed on the atomic `freed` boolean variable. Only one thread succeeds and frees the content of the block.

Every block chain is exclusively owned by a single thread. There are no race-conditions possible during block creation because iterations are only added serially by the owning thread. Blocks are never removed from the block chain. Once a block is *freed* all iteration objects are freed and the block is marked as finished.

3.7 Synchronization

Synchronization in Pheet Pipelines is done without locks except for the optional debug logging interface.

In all other cases `std::atomics` (see Section 3.5.1) are used to perform synchronization.

Theorem 3.1 *We show that our implementation is* lock-free *and* correct*.*

In the following sections the critical points in a Pipeline execution which need synchronization are described and shown to be *lock-free*.

3.7.1 Iteration Object creation

Lemma 3.2 *The Iteration object creation is lock-free and correct.*

Pheet Pipelines benefit from the restricted Pipeline task graph. In our implementation only backward dependencies are allowed. *Iteration* objects of a nesting level must be created serially by a single thread. This ensures that the iteration id sequence is *gap free and monotonically growing.* It also ensures that when iteration n is constructed, all previous iterations $0, \ldots, n-1$ already have been fully initialized. The copy constructor of Pheet performance counters is not thread safe and needs to be executed in the parent thread. Because iteration object creation can become a serial bottleneck, several members of the object are initialized later in parallel.

During object creation all iterations increment an atomic `running` counter in their parent objects. This is done by calling the `thread_register` parent method which also stores the iteration pointer in the block chain. For objects at nesting level 0 the parent is the Pipeline Environment class. For objects at deeper nesting levels the counter of the parent iteration is incremented.

This `running` counter is crucial to check if all iterations have correctly terminated. Each iteration only decrements the parent running counter when all of its own children have finished. ■

3.7.2 Iteration termination

Lemma 3.3 *Every finished iteration is freed and enables all dependent iterations.*

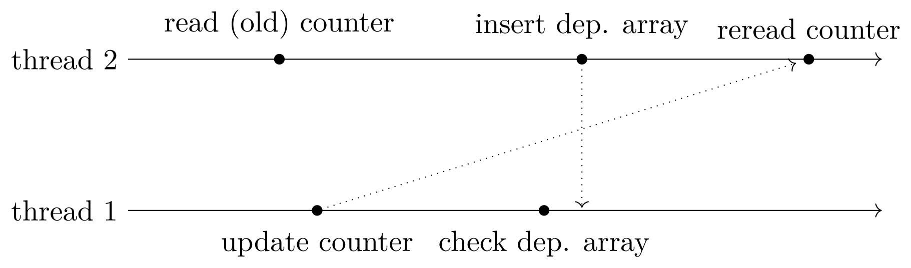


Figure 3.6: Critical case during Iteration object destruction where the waiting iteration reads an old stage counter value and then inserts its reference in the dependent array. But at a position already checked by the finishing iteration. It is guaranteed to be correct because thread 2 rechecks the stage counter after putting the reference into the dependent array.

Iterations track their state using the `status` field. An iteration is considered as `FINISHED` when it has been correctly marked as finished by calling the finished function of our interface. Iterations with nested children change their state to `CHILDREN_WAIT` at the finish call. The last child changes the parent's state to `FINISHED` during termination. An atomic compare and swap operation ensures that exactly one child thread sets its parent to finished.

Only iterations in `FINISHED` state are allowed to be freed. This is ensured using asserts. During object destruction all nested child iterations are destructed too. All other operations (wait, continue) on a finished iteration are prohibited.

During the finish call all waiting iterations are enabled, after increasing the stage counter atomically.

No iteration can be forgotten in the dependent array. The critical case (Figure 3.6) is when an iteration observed the old stage counter and inserted itself at an empty position at the beginning of the dependent array. Even when the finishing iteration already has advanced further in the dependent array, the waiting iteration will reread the stage counter and continue execution.

The Finisher Hyperobject guarantees that all spawned Pheet tasks are executed. Therefore all dependent iterations of the Pipeline are executed and the iteration object is freed. ■

3.7.3 Block destruction

Lemma 3.4 *A block only containing finished iterations gets freed by exactly one thread.*

Iterations increment the `finished` counter in their block during termination. If an iteration experiences `finished` equal to block-size it requests the block to be freed. This is done by calling the Pipeline Environment function `free_block` with the block reference.

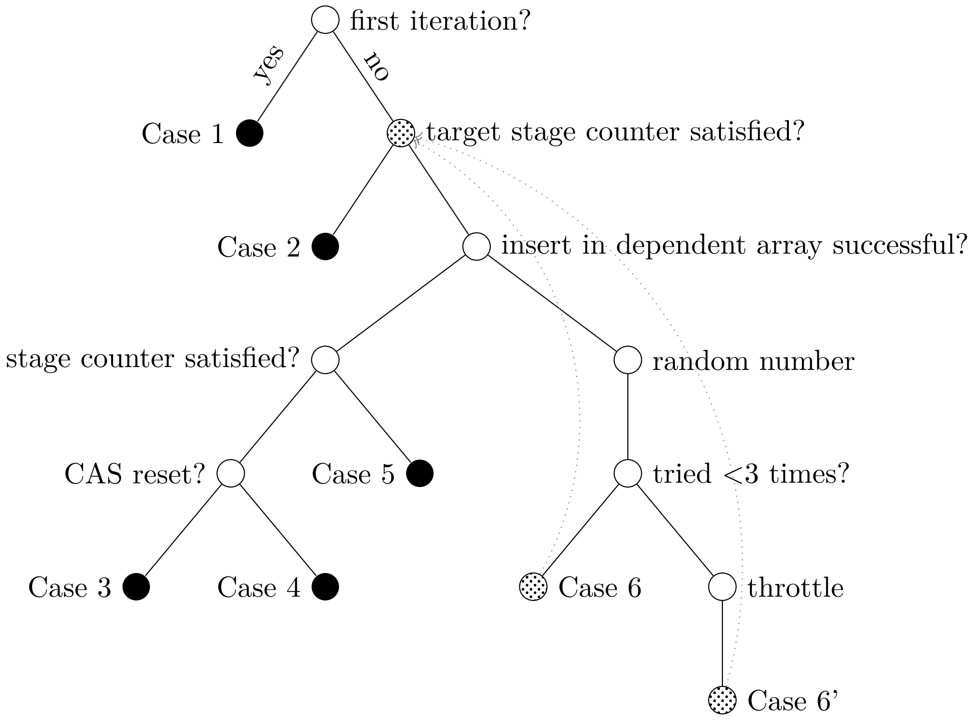


Figure 3.7: Decision diagram of the Pheet Pipeline wait function. Left edges mean yes or success and right edges mean no or failure. Filled black circles mark end stages. The circle with the pattern is used as a jump target.

Multiple iterations may experience finished = block-size because of threading issues. A *compare and swap operation* on a boolean variable of the block instance is used to ensure that blocks are only freed by one thread. ■

3.7.4 Pipeline Wait

Lemma 3.5 *The Pipeline wait function is lock-free and correct.*

The most critical part is to synchronize iterations with dependencies. The control flow diagram in Figure 3.7 shows the critical steps. The control flow is shown from the iteration calling the `wait` function to create a dependency to a preceding iteration.

If the iteration is not the first iteration, the stage counter of the dependent iteration is compared with the requirement. In case the stage requirement is not satisfied, a `PheetPipelineTask` object is created which contains the stage. It is then tried to insert this task object into the fixed size `dependent` array of the iteration we are depending. To ensure no previous element is overwritten an atomic *compare and swap operation* is performed. In case this atomic insert fails it is tried again at another random position until it succeeds or the requirements are met. After a successfully insert the stage counter is rechecked to cover a corner case. In case the iteration already has advanced the inserted task object is removed again from the `dependent` array.

Case 1 In the simple case the first iteration (I_0) tries to wait for a preceding iteration. In this case no actual wait has to occur and the call is treated internally as a continue call. This allows the user to consistently use `pipe_wait` and does not require a dedicated if expression for the first iteration. ■

Case 2 In case this iteration is not the first one the atomic stage counter of the depending iteration is loaded. In this case the other stage already satisfies the required stage, no wait has to occur. And the iteration can directly proceed to the next stage. ■

Case 3 In this case the *insert* into the dependent array was successful but the *reread* of the stage counter resulted in a satisfied dependency. The inserted element was removed using a *compare and swap* operation again because waiting is not necessary any more. In this case the CAS operation was successful. The waiting iteration can now be executed directly because the dependencies are already fulfilled. ■

Case 4 In case after a successful insertion into the dependent array the *stage counter* is reread. If the new stage counter satisfies the requirement and the CAS operation failed, the dependent iteration already has scheduled (and removed) the waiting task and no further action has to be performed. ■

Case 5 In case after a successful insert into the dependent array, the *stage counter* is reread. If the stage counter of the dependent iteration is still not satisfying this iteration, it has to wait until it is enabled by the other iteration. ■

Case 6 and Case 6' In case the stage counter was not satisfied and the insertion into the dependent array failed, a new random position in the array is diced. This random number is used as index in the `dependent` array in the next loop iteration. The insert into the dependent array fails when the chosen slot is already taken by another waiting iteration.

In case the insert failed three times in a row, the iteration is throttled and other ready work is executed. Then in both cases the control flow goes back up and reads again the stage counter. ■

3.7.5 Pipeline Continue

Lemma 3.6 *The Pipeline continue function is lock-free and correct.*

The Pipeline continue operation increases the atomic stage counter of the iteration. Then it is checked if the increased stage counter enabled dependent iterations. The control flow diagram of the continue path is shown in Figure 3.8.

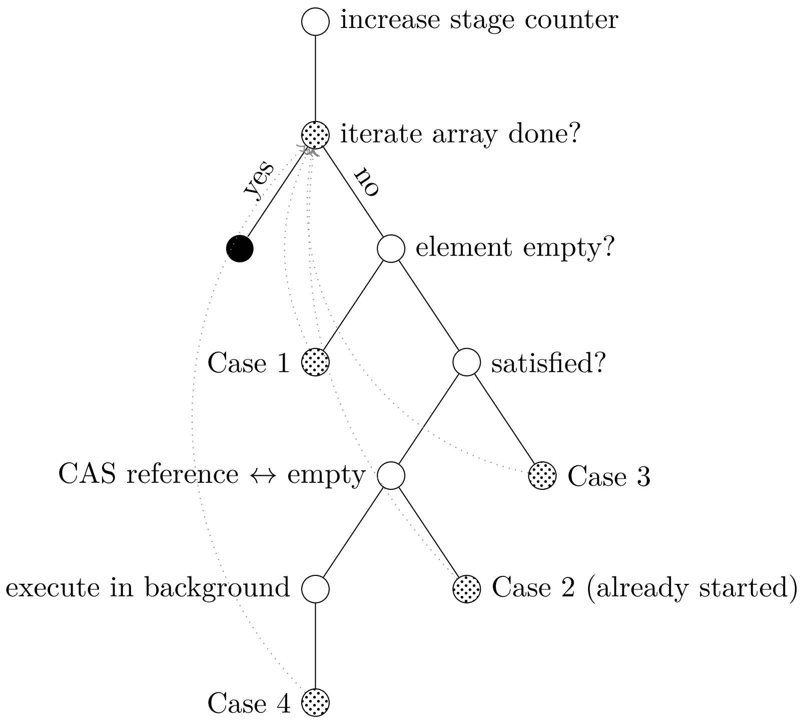


Figure 3.8: Decision diagram of the Pheet Pipeline continue function. Left edges mean yes or success and right edges mean no or failure. Filled black circles mark end stages. The circle with the pattern is used as a jump target.

To check if the new stage counter enables other dependent iterations the `dependent` array is checked. This is done by *iterating* over all array elements.

Case 1 In this case the slot is empty we proceed to the next slot. ■

Case 2 In case a *non-empty element* is found, the stage counter of the found iteration is checked. In case the required stage dependency is now *satisfied* with the new stage, an atomic *compare and swap* operation is executed to reset the array element in order to schedule it. In case the value could *not be swapped* no operation is performed. If the element can not be swapped it has already been reset by the dependent iteration itself because it already noticed the updated stage counter. ■

Case 3 In case a *non-empty element* is found, the stage counter of the found iteration is checked. In case the stage counter of the dependent iteration is still not satisfied, no operation is performed. Latest when the dependent iteration is finished using the `finished` method, all depending iterations are enabled. ■

Case 4 In case a *non-empty element* is found, the stage counter of the found iteration is checked. In case the required stage dependency is now *satisfied* with the new stage, an atomic *compare and swap* operation is executed to reset the array element. After *successfully resetting* the array element, the corresponding iteration object is loaded and the element's stage counter is incremented. Then the task is spawned in background. ■

After possible dependent iterations have been enabled, the Lambda body is executed. In the first stage the continue call is *spawned* in background while in later stages a direct `Pheet::call` is executed. This corresponds to a *work-first* strategy which finishes running iterations faster. This reduces the memory usage because only finished iteration can be freed.

3.7.6 Stage results

Lemma 3.7 *Stage results are available after waiting for a dependent iteration.*

Stage results are only added by the iteration itself. The result block is only allocated when a non-null result is added. In case the result block has not been allocated the `get_result` method always returns a null value.

Result blocks store results in an atomic list. To store the values the memory order *relaxed* is used (see Section 3.5.1). The later stage counter increase is done using a *sequential consistent* order making the result value visible for other threads. Reading threads are only allowed to access the result after they waited for the iteration creating the result. To ensure correct order, results are always stored *before* the stage counter is incremented. ■

3.7.7 Pipeline termination

Lemma 3.8 *The Pipeline executes all iterations correctly.*

The Pipeline must be enclosed in a `Pheet::Finisher` Hyperobject to ensure termination of all spawned tasks. After all children have finished, the running counter in the Environment is zero. This is ensured with a check during Pipeline destruction. In further version the Pipeline Environment instance may be used directly as Finisher object.

As we have shown in Lemma 3.5, all stages of all iterations are spawned as Pheet tasks during the Pipeline execution. In combination with the Finisher Hyperobject this ensures that the Pipeline is executed correctly.

The Environment object has to be freed after all iterations have been freed. Only after all iterations and the Environment class were freed the performance counters are accessible. Performance counters in Pheet are based on Hyperobject Reducers which merge their separate values during destruction. ■

3.8 Debug Logging

Debug output is a complicated issue in multithreaded programming. The output from concurrent threads has to be synchronized before written to the system output. Otherwise different lines may be mangled and displayed incorrectly.

To solve this issue and to make development of Pipeline programs easier a debug output interface was integrated into Pheet Pipelines. This integration makes it possible to display information about the current iteration like stage and iteration number. The logging interface is discussed in Section 3.4.5.

To coordinate the different worker threads a mutex (`std::mutex`) is used in combination with a `std::lock_guard` to protect the system output. Because this reduces the scalability of programs, the debug output can be disabled using compiler options. When debug logging is disabled the compiler removes all the log calls during optimization.

This function is *deadlock-free* because only a single lock is used. In the critical section only the output to the terminal is performed.

The logging function is defined in the `PipelineIteration` object and therefore always accessible for the user. It is implemented using *Variadic templates* [ISO12] which have been included in the C++11 standard.

Variadic templates accept a variable number of template arguments. The templates are not available directly to the function, so a recursive approach is used to process the arguments one by one and then call the function again with the tail of the template list. This construct can be used to create a type safe print function using recursion.

The new construct make the usage of the `log` function very straight forward. The user can simply put all the information he wants to display as arguments to the function.

3.9 Performance Counting

Performance values are important to investigate problems with scalability on many-core systems. The Pheet Pipeline implementation uses various performance counters provided by the Pheet framework to measure important characteristic values during execution.

Pheet Pipelines provide the following performance counters:

- The number of *created blocks* to store Pipeline iterations. When this value is very high, the overhead of the block creation can be reduced by increasing the block-size parameter.
- The total number of *wait*, *continue* calls. This number is mainly interesting for non-linear programs, where the number of continue calls is data dependent.
- The number of how many *wait* calls had to actually *block* because the dependent iteration had not yet completed the required stage.

- The total number of created *iteration* objects on all nesting levels. This value may be useful to determine the best block-size. It may also give some insight about the best throttle limit.

`BasicPerformanceCounter` of the Pheet framework were used to implement the counters. They use a *Sum-Reducer* [Wim13] internally to prevent contention induced by a single counter variable. Reducers provide per thread local views which are merged on defined synchronization points. Reducers in Pheet are implemented as Hyperobjects and are wait-free [Wim13]. In our implementation synchronization points are the terminations of iterations. Performance values are collected by all iteration objects independently and are moved to their parents during destruction.

Because of the way how Reducers work, performance values can not be inspected during runtime. The values are available after the central Pipeline Environment object has been destructed.

For most of our benchmark instances the overhead of performance counting was neglectable, although they can be disabled easily using compile time flags.

3.10 Implementation Summary

We have shown our interface and design decisions of our C++ Lambda based Pheet Pipeline interface. Our interface allows short Pipeline definitions without boilerplate code.

Additionally we have shown how our implementation supports *non-linear nested* Pipelines which are important for some benchmark instances like Dedup.

Our interface supports the passing of results between iterations which removes synchronization burden from the user. This is not supported by most other analyzed Pipeline implementations. Nabbit supports result passing between dependent iterations.

To avoid central contention points our implementation is *lock-free* and *deadlock-free* which we also argued in the previous sections. We extensively use the C++11 atomic data type.

Our interface supports a dynamic user editable throttle limit to limit the amount of memory used during the Pipeline execution. Other implementations only support static throttle limits which we consider not flexible enough in multiprogrammed environments.

Future work may address the following open issues: Only a single data type can be specified as result data type for stages. This is not very flexible and leads to code which is not type safe.

The throttle function has to be manually called by the user to take effect. This should be moved directly into the scheduler.

The user has to manually create a Finisher Hyperobject which ensures termination of all Pipeline iterations. This should be moved into our Pipeline Environment object.

CHAPTER 4

Benchmarks

This section describes the benchmarks used and how the benchmark values have been measured. Where possible we provide an expectation for each of our benchmarks based on previous work of others. We also formulate the goals of our benchmarks in this chapter.

To show the scalability of our Pipeline implementation following benchmarks were implemented with our Pheet Pipeline interface:

- *Dedup* [Bie11] is a block level data deduplication and compression algorithm. The input is two times fragmented into smaller chunks. These chunks are compressed and duplicates are eliminated based on a fingerprint. See Section 4.5 for details.
- *PrefixSum* [Ble89] sums all previous items in an array for each position. See Section 4.6 for details.
- *X264* [Vid] is the popular H.264 video encoder. See Section 4.7 for details.
- *Ferret* [Lv+07a] is a similarity framework developed at Princeton University. See Section 4.8 for details.
- *Synthetic* is a new benchmark to test the scalability of our implementation itself. It contains a parametrized workload of matrix rotations. See Section 4.9 for details.

4.1 Goals - Comparison

The goal of our benchmarks is to investigate the scalability of our Pheet Pipeline implementation. To achieve this goal multiple known Pipeline benchmarks have been ported to our Pipeline interface. The benchmarks are described in detail in Chapter 4.

We executed the transformed benchmarks on our test systems with varying cores to measure the speedup and scalability. We tried to predict the scalability for executions utilizing many threads using information about the hardware architecture of the benchmark systems. The scalability characteristics mentioned in other papers were used to formulate our scalability expectations for our implementation.

To our knowledge no work had been published comparing the Pheet framework with other task parallel frameworks. Because our implementation is implemented on top of Pheet we can not compare our Pipeline performance to other Pipeline implementations because we can not rule out effects of the Pheet scheduler itself. Therefore no comparison tests with other Pipeline implementations (like Intel TBB, Piper) have been conducted. Comparing performance with other Pipeline implementations would require a detailed comparison of the Pheet framework first, which would go beyond the scope of this thesis.

The different benchmarks have different requirements to the memory subsystem. Some work on independent data, some require access to results of adjacent iterations. We generally want to see if our implementation scales well for real world problems even beyond 16 cores.

To give a reference value most of our benchmarks have been benchmarked using a Pthreaded version as well. All of the Pthreaded implementations can be considered as reference implementation by the creators of the benchmarks. We used these Pthread versions to investigate characteristics of the benchmarks itself.

4.2 Speedup

The speedup gives the speed improvement of an execution with multiple threads compared to single threaded execution.

The speedup of an actual execution is calculated as follows:

$$\text{Speedup}_n = \frac{T_1}{T_n} \tag{4.1}$$

Where T_1 denotes the execution time for a single threaded execution and T_n denotes the execution time with n threads.

A speedup is *linear* when it satisfies the condition $\frac{T_1}{T_n} = \Omega(P)$ [ALS10].

Amdahl's Law

Amdahl's Law [Amd67] can be used to hint the speedup of parallel application.

$$\text{Speedup} = \frac{1}{r_s + \frac{r_p}{n}} \tag{4.2}$$

Where r_s defines the sequential portion and r_p the parallel fraction of a program. Together $r_s + r_p = 1$. n names the number of computation cores.

This law restricts the possible parallelism to the serial part of the algorithm. In case we assume a 10 % serial portion of a problem and let the $\lim_{n\to\infty} = \frac{1}{r_s}$ we are limited to a speedup of 10.

Gustafson's Law

Gustafson's law [Gus88] is an alternative to Amdahl's law. It says that "the problem size scales with the number of processors" [Gus88]. This idea can be used to explain good speedup on modern multi core systems. Whereas Amdahl's law assumes that the problem size is constant, Gustafson's law makes the total execution time constant [JM12].

Some of our benchmark systems have a theoretical speedup limit about 160. We run all our benchmarks with two different instance sizes to check if a bigger instance also yields a better speedup.

4.3 Measurement

To remove the possibility of one time effects, all benchmark instances were run $n = 50$ times on each system. Based on the results the *standard error* of the mean with a 95 % *confidence interval* was computed.

The 95 % confidence interval ($ci_{0.95}$) was calculated assuming a normal distribution Θ. In our benchmark plots the confidence interval is shown using a vertical stroke for every data point.

We calculated the confidence interval as follows:

$$1 - \alpha = 0.95 \rightarrow \alpha = 0.05 \tag{4.3}$$

$$\Theta^{-1}\left(1 - \frac{\alpha}{2}\right) = \Theta^{-1}(0.975) = 1.96 \tag{4.4}$$

$$se = \frac{\sigma}{\sqrt{n}} \tag{4.5}$$

$$ci_{0.95} = 1.96 \cdot \frac{\sigma}{\sqrt{n}} \tag{4.6}$$

$$P(\bar{X} - 1.96 \cdot \frac{\sigma}{\sqrt{n}} \leq \mu \leq \bar{X} + 1.96 \cdot \frac{\sigma}{\sqrt{n}}) = 0.95 \tag{4.7}$$

Where σ denotes the standard derivation, n denotes the number of repeated test runs and $\bar{X}$ is the *sample mean*. *se* denotes the *standard error*.

The tests are run with a set of CPUs cores to investigate scalability. To remove statistical errors the benchmarks are run in a randomized order.

Time is measured using two time stamps obtained with `high_resolution_clock` from the `std::chrono` namespace. Time needed by benchmarks is calculated in *microseconds* and then converted to seconds with six fractional digits.

4.4 Benchmark Systems

The benchmarks have been executed on four different systems. Each system features different architectural and performance properties.

- *Mars* is an Intel Xeon system with 8 CPUs each having 10 cores. Each core can execute two threads (Hyper Threading) to hide memory based latency. Each CPU has a shared 24 MB L3 cache. See Section 5.1 for details.
- *Ceres* is an Oracle Spark T5-4 system with 4 CPUs each having 16 cores. The CPU features a high number of 8 threads per core (similar to Hyper Threading) and makes use of out-of-order executions to speedup single thread performance. See Section 5.3 for details.
- *Pluto* is a system equipped with 2 Intel Xeon Phi coprocessor cards each having 61 cores. The coprocessor cards run a customized Linux but require binaries to be compiled for the special MIC architecture. See Section 5.4 for details.
- *Saturn* is a AMD Opteron based system with 4 CPUs each having 12 cores. Each CPU has a shared 12 MB cache. See Section 5.2 for details.

4.5 Dedup

Dedup [Bie11] is a data deduplication and compression kernel developed at Princeton University. It is included in the Parsec [Bie11] suite because of its combination of global and local compression resulting in high compression rates. Furthermore it is a mainstream method for backup storage systems to reduce disk space usage.

The input data is first split into fixed size blocks which are processed in parallel afterwards. Each block is split into further coarse grained *fragments* using *Rabin-Karp fingerprints* [KR87]. For each fragment a hash value is computed. Afterwards new fragments are compressed using the `GZIP` compression which is based on the *Ziv-Lempel algorithm* [ZL77]. The compress stage is skipped when the same fragment already occurred before. This requires *stage skipping* support in the Pipeline. Hash values are stored in a global hash table. For duplicate fragments the compressed data is written only once to the output file.

4.5.1 Aim of this benchmark

The benchmark was included in the Parsec [Bie11] suite because the method is getting attention in enterprise backup storage and network solutions [ZLP08]. The algorithm offers a good speedup on multi-core systems and has a very good data locality. Because of its non-linear structure it requires support for *non-linear* (nested) Pipelines (see Section 2.1.5).

The benchmark is heavily data dependent and thus can reveal problems with non local code execution and cache flushes. We used this benchmark to find the memory limits of our different test systems.

4.5.2 Testdata

We used the *simlarge* test instance of the Parsec [Bie11] suite for our benchmarks. The input is 184 MB of data containing literature text from Project Gutenberg[1]. The ratio of uncompressed to compressed data is about 2.2 : 1. The instance creates two iterations at the first nesting level and ≈93 000 iterations at the second.

Additionally the *native* test instance of Parsec was used as benchmark. The input of this instance is a 704 MB FedoraCore[2] 6 disc image (`FC-6-x86_64-disc1.iso`). The file has a compression ration of 1.07 : 1, which is not surprising because the disk image mostly contains already compressed software archives. The instance creates six iterations at the first nesting level and ≈370 000 iterations at the second.

4.5.3 Expectations

With Piper [Lee+13] Lee et al. showed that speedup comparable to the Pthread version can be achieved using their Pipeline implementation. They showed that a speedup of 6.7 is possible for a Pipeline implementation with 16 threads on a system similar to our Saturn system. Because the number of physical cores of their AMD based system was limited to 16 threads, no tests with more threads had been published.

The Pthreaded Dedup version contained in the Parsec [Bie11] suite was used as reference implementation. We expect a similar or slightly worse performance compared to the Pthreads implementation because it is carefully optimized and the *bind-to-element* approach is beneficial for this benchmark [Lee+13].

We do not expect much speedup after the chip boundaries (16 for Ceres, 10 for Mars, 24 for Saturn). After this thread counts, cross chip memory latency is expected to be a limiting factor.

Better speedup of the bigger *native* test instance is expected because it yields more parallelism. The *simlarge* instance yields only two iterations at the first level while the larger *native* instance yields six iterations. According to *Gustafson's law* [JM12] the bigger instance should scale better.

4.5.4 Implementation

The algorithm can be modeled with the following five stages in a Pipeline fashion. Only the first (input) and the last (output) stage have to be serial. The Pipeline schema can

[1]Project Gutenberg. URL: `http://www.gutenberg.org/wiki/Main_Page` (visited on 01/10/2016).

[2]Fedora Project. URL: `https://getfedora.org/de/` (visited on 01/10/2016).

be seen in Figure 4.1 and the stages are:

1. *Coarse-grained fragmentation* is a *serial* stage which splits the input data into fixed size chunks. The file is read from disk. To eliminate effects of the I/O subsystem and the caching system the benchmark was run multiple times and the worst execution was dropped from the results.

2. *Fine-grained fragmentation* is a *parallel* stage which processes chunks generated in the previous stage. This stage uses Rabin-Karp fingerprints to further refine chunks into smaller fragments. This stage is a *non-linear* stage and creates for one input element n output elements.

3. *Hash computation* is a *parallel* stage which calculates a SHA1 sum of each fragment. Then the hash is looked up in the global hash table to check if the same fragment already occurred. If the hash is not present it is added and the compression stage is not omitted. The hash table is protected using a dedicated lock for each bucket. See below for details.

4. *Compression* is a *parallel* stage which compresses duplicate fragments using a `GZIP` or `BZIP2` compression based on compiler flags. In our benchmarks we used the fast `GZIP` compression.

5. *Assemble output stream* stage is the last serial stage which creates the compressed output file. The function either writes a compressed fragment or a SHA1 hash in case it already occurred earlier.

   The Parsec reference implementation [Bie11] needs extra logic to reconstruct the original order of fragments in the output stage.

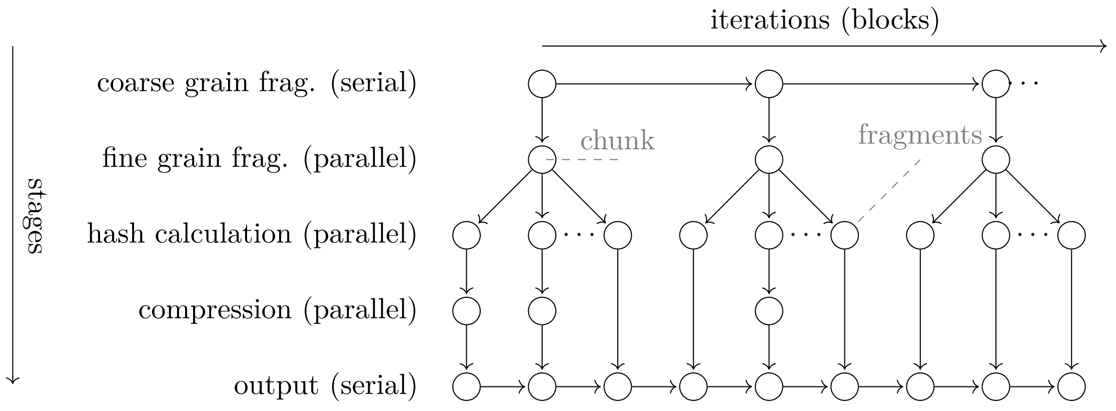
Locks

Because the global hash table is accessed in a parallel stage by multiple threads, locks are required for synchronization. The hash table is split into a large number of buckets ($\approx 100\,000$ for our benchmark). Each bucket has its own lock, reducing the chance of lock contention.

Non-linear Pipelines

The algorithm requires a *non-linear* Pipeline because stage three further coarsens the chunks. This stage takes a single input chunk as input but outputs multiple smaller fragments. The algorithm requires *stage skipping* (see Section 2.1.4) because duplicate segments are not compressed and therefore the compression stage is omitted (see Figure 4.1).

Piper lacks support for non-linear Pipelines therefore they omitted stage three resulting in less parallelism.



iterations (blocks)
stages
coarse grain frag. (serial)
fine grain frag. (parallel)
hash calculation (parallel)
compression (parallel)
output (serial)
chunk
fragments

Figure 4.1: The Dedup Pipeline schema shows the non-linearity of the parallel algorithm. Input is divided into fixed size chunks in the first serial stage. Then each chunk is further divided into smaller fragments in a parallel stage. This stage takes one input element and outputs multiple elements. The number of smaller fragments depends on the algorithm and varies from iteration to iteration. For each of these small fragments a hash is calculated and stored in a global hash table. Duplicate fragments are not compressed resulting in *stage skips*. The output stage writes either the compressed fragment or a placeholder hash into the output stream.

Adaptions

In the Parsec reference implementation not the whole chunk is consumed in a single iteration. Remaining parts are added to the next chunk read from the input file. In a parallel computation this would create a serial stage and chunks could not be processed in parallel. To avoid this issue the remaining parts of a chunk are processed in an extra sub iteration.

Because chunks are processed in parallel a duplicated fragment may be first discovered in a later chunk. Only the first occurrence of a chunk is compressed and written to the file in the final stage. For all further occurrences only a *SHA1* hash is written to the file instead. Because of this possible miss order, the hash would be written before the actual compressed data.

To circumvent this, for every fragment already existing in the global hash table, which is in the wrong order, the fragment is compressed too. No additional hash table lookup is necessary. This modification may increase the output file size by a small amount. Correctness of the algorithm is untouched by this modification.

To make this fix possible the `sequence` of the Pthread Dedup implementation is used to store the original fragment position in the input stream. This sequence contains the id of the chunk (1$^{\text{st}}$ level) and the id of the number of the fragment (2$^{\text{nd}}$ level). Is is stored for each fragment and uniquely identifies it.

In our test instance from the Parsec suite reordering affected only 4 out of ≈100 000 fragments.

To run the benchmarks on the Xeon Phi architecture it was necessary to compile *libressl*[3] `2.2.5` and *zlib*[4] `1.2.8` for the MIC architecture.

Reference Implementation

As reference the *Pthreaded* version of the original Dedup variant was benchmarked on all systems with the same input instances. The Pthreaded version is more complex because an additional reordering stage is needed to output output chunks in order. Output chunks are cached in this stage until all preceding chunks have been written.

The additional Pthreads code results in 1600 additional lines of source. Pthreads parts are enabled by a compile-time switch (`ENABLE_PTHREADS`).

Our *non-linear Pipeline* version of Dedup is only about 20 lines of additional code. Most of this code is used to initialize the Pipeline Iterations correctly.

The Pthreaded version starts dedicated threads for each stage. This results in a total of at least five threads. Our implementation uses the Pheet task scheduler which can have an arbitrary amount of worker threads. To make the results comparable with our task scheduler the number of operating system threads had been limited for the Pthreads version using the *hwloc* library [Bro+10].

4.6 PrefixSum

PrefixSum is a common problem and the bases for many parallel algorithms [Ble89]. It calculates the sum of all previous items in an array for each position. The single threaded implementation is trivial - a single pass is sufficient. For the list `1 3 5` the PrefixSum would be `1 4 9`.

A parallel implementation requires a different algorithm with multiple passes. In the first pass the input array is split into blocks. Then for each block the sum is computed and stored in an auxiliary array. The content of the newly created auxiliary array is replaced by the computed PrefixSum of the array itself. In the final pass the values of the auxiliary array are used as offset to compute the PrefixSum of he original blocks in parallel. Figure 4.3 shows an example of a parallel PrefixSum computation.

4.6.1 Aim of this benchmark

PrefixSum is commonly used to implement other parallel algorithms [Wim14]. Although it reads data sequentially, it is memory-bound because every list element is read only once.

[3]LibreSSL. URL: `http://www.libressl.org/` (visited on 01/10/2016).
[4]zlib. URL: `http://www.zlib.net/` (visited on 01/10/2016).

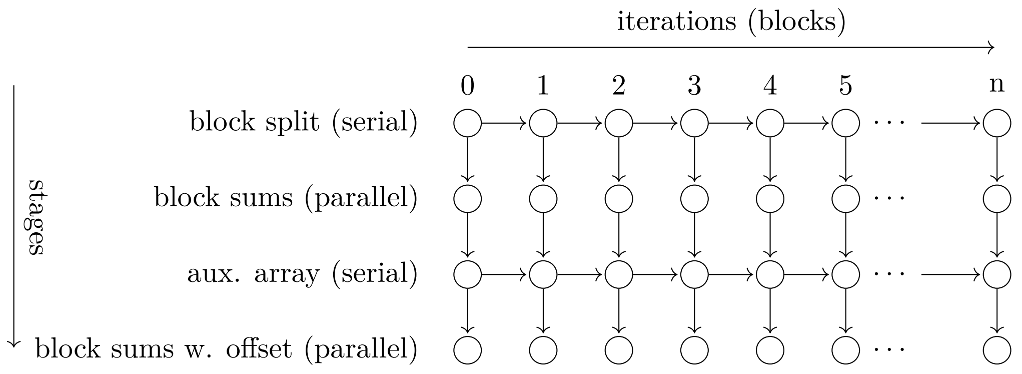


Figure 4.2: PrefixSum Pipeline schema showing the four stages. The first stage splits data into blocks. In the second stage the sum for each block is calculated in parallel. Results are stored in an auxiliary array. In stage three the PrefixSum for the auxiliary array is computed. In the last parallel stage the PrefixSum for each block is computed using the offset from the auxiliary array.

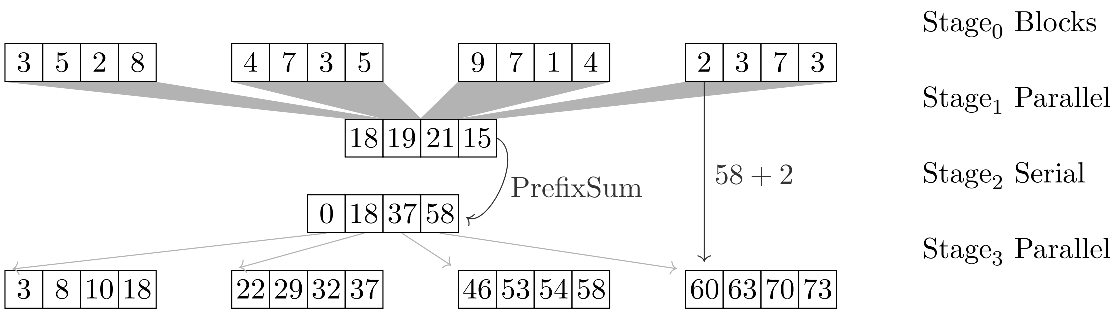


Figure 4.3: Example of a parallel PrefixSum computation [Wim14]. The input array is split into blocks. The sum of each block is calculated in parallel and stored in an auxiliary array. For the auxiliary array itself the PrefixSum is computed. In the final stage the computed values are used as offset to compute the PrefixSums in parallel per block.

It was used as benchmark for our Pipeline implementation because there exist various implementations in the Pheet [Wim14] test suite. This benchmark measures the overhead of our Pipeline implementation and allows comparison of the scalability with the version implemented with Pheet tasks.

4.6.2 Testdata

The test instances used to benchmark our Pipelined implementation of PrefixSum was derived from existing testdata of the Pheet framework.

We benchmarked with two instances with an array size of 2^{25} and 2^{27}. The data type of all array values is a 32 bit integer. All array elements are set to `1` which makes the correctness check very easy.

We used a *block-size* (*Cutoff-size*) of 4096 for our Pipeline implementation and for the Pheet Threads reference implementation.

4.6.3 Expectations

The PrefixSum computation yields only very little work for each Pipeline iteration. To improve the performance the block-size is set to 4096. This block-size results in ≈8000 iterations for the first test instance and ≈32 000 for the second instance. The first serial stage only calculates the boundaries of the blocks which are processed in parallel in the second stage. Therefore we expect that there is enough possible parallelism for our test systems.

Previous benchmarks of the Parallel PrefixSums implementation showed that speedup up to 48 cores is possible on the Saturn system [Wim14]. We used this parallel Pheet implementation as the *first reference* for our Pipeline version. Wimmer used an array of size $10^8 \approx 2^{26}$. We also expect a similar speedup of 2.8 at 48 cores on the Saturn system.

Because there exists only one array instance on which all threads operate, we expect some NUMA effects at around 24 threads because of cross chip traffic on the Saturn system. The Saturn system features 12 cores per CPU with two CPUs on one die. The testdata array is initialized with multiple threads. This may lead to bad locality on Ceres because of its *first touch* policy. See Section 5.3 for details about Ceres.

The last stage replaces the values of the input array, thus we expect some cache misses there for executions using many threads.

Our *second reference* implementation is a primitive *sequential* implementation. We expect this implementation to outperform our implementation for low core counts (1-4) because of our Pipeline overhead.

4.6.4 Implementation

We did not put any effort into macro optimizations like vectorization and stick with a basic implementation of the parallel PrefixSum algorithm. Listing 4.4 shows the *continue* and *wait* calls to create a parallel and a serial stage. The first stage in our implementation is always a *serial* stage, which calculates only the block boundaries.

The results of each stage are passed to the *continue*, *wait* and *finished* calls. Stages without results like the first stage (stage_0) pass a null value as result.

On Ceres we experienced problems with the Pheet StrategyScheduler which also forms the base for our *PipelineScheduler*. In particular we ran into *segmentation faults* at executions using a high thread count. We asume that there is a problem with the reuse of datastructures in the Pheet scheduler on Ceres but it was not possible to investigate this issue into more detail. We switched to the Pheet *BasicScheduler* (see Section 1.2.2) to mitigate this issue.

4.6.5 Reference Implementation

As reference implementation an existing Pheet task version was used. This PrefixSums implementation uses a *divide and conquer* approach and splits the input array in the middle. Then for both parts of the split array a new task is started in a recursive fashion. A Finisher Hyperobject is used to wait for both child computations to be finished.

The parallel recursion is stopped at the *Cutoff*-Limit which is also set to 4096. This Cutoff-Limit is also the block size in our Pipeline implementation. Below the Cutoff-Limit the PrefixSum is calculated using a single pass over the array.

Each iteration tracks its stage using a numeric value. A *memory fence* is used to ensure synchronization before the stage is increased.

As second reference version an existing sequential version of the sequential algorithm was used. This algorithm just performs one pass over the input array.

4.7 X264

The x264 encoder[5] is a popular open source encoder for the widely used H.264 [Ric11] video compression. H.264 is a highly space efficient video codec. X264 is capable of converting a range of video formats into the H.264 codec which is widely used in the internet.

X264 uses Pthread mutexes for thread synchronization and is considered as one of the fastest H.264 encoders.

4.7.1 Aim of this benchmark

X264 is included in the Parsec [Bie11] benchmark suite because of its sophisticated Pthreads implementation featuring almost linear speedup [Lee+13]. The H.264 video

[5]Videolan. *x264 the best H.264/AVC encoder.* URL: `https://www.videolan.org/developers/x264.html` (visited on 10/13/2015).

PrefixSum Pipeline implementation

```
#define STAGE1 1
#define STAGE2 2
#define STAGE3 3

Pheet::Environment env {cpus}; // Pheet Environment
Pheet::Pipeline penv { ppc }; // Pipeline Environment
int counter = 0; /* iteration counter */
size_t Cutoff = 4096; //block size
while( Cutoff*counter <= length) {
  Pipeline::PipelineIteration* piter =
    new Pipeline::PipelineIteration(&penv, counter);
  //stage0 - calculate block-size
  size_t blength = length-counter*Cutoff;
  if(blength > Cutoff) {
    blength = Cutoff;
  }
  auto bdata = data+counter*Cutoff; // block start
  piter->pipe_continue(STAGE1,0, [piter,bdata,blength,&penv] () ↩
      {
    //stage1 parallel computation of block sums
    int blocksum = 0;
    for(size_t i = 0; i < blength; ++i) {
      blocksum += bdata[i];
    }
    piter->pipe_wait(STAGE2,blocksum, [piter,bdata,blength, ↩
        blocksum,&penv] () {
        //stage2 add sum prev block
        int prevBlockSum = 0;
        if(piter->iteration > 0) {
          prevBlockSum = penv.find(piter->iteration)-> ↩
              get_result(STAGE2);
        }
        int STAGE2_RESULT = prevBlockSum +blocksum
        piter->pipe_continue(STAGE3, STAGE2_RESULT, [piter, ↩
            bdata,blength,prevBlockSum] () {
          //stage3 PrefixSum of block with offset
          bdata[0] += blockResult;
          for(size_t i = 1; i < blength; ++i) {
            bdata[i] += bdata[i-1];
          }
          piter->finished(0);
        });
      });
    });
  counter++;
}
```

Figure 4.4: Implementation of the parallel PrefixSum algorithm using our Pipeline interface. For easier readability some necessary C++ modifiers (**`typename`**) have been omitted.

codec is used widely for internet video applications. X264 is one of the most popular open source encoders for H.264.

Lee et al. [Lee+13] showed that the algorithm can be modeled in a Pipeline fashion with a near linear speedup up to 16 cores. The original algorithm was adapted to use their *Cilk-P* framework running on their *Piper* scheduler [Lee+13].

In this benchmark we use an uncompressed video in *raw yuv* format which is converted into the H.264 format by x264. The *yuv* format is an uncompressed video format consisting only of still images.

4.7.2 H.264

A H.264 video sequence consists of frames which contain the actual image. To achieve a small file size, not all frames contain a full image. Some information may be referred from another frames to save space like file compression algorithms do.

Therefore each frame is divided into a two dimensional array of *macroblocks* which can be referenced by other frames if they contain similar content. This is beneficial because often only parts of a picture change between near frames (e.g static scene and camera with only one moving object).

To achieve a balance between coding efficency and computational effort during encoding and decoding the range of reference frames is limited. The H.264 standard [Ric11] describes three different frame types:

I/IDR-Frames contain a full picture. Macroblocks of these frames can only reference preceding macroblocks of the same frame. IDR-Frames are a reference barrier and are important to enable quick seeking in video files. They also support fast error recovery in case some frames are dropped in video streams. No frame after an IDR-Frame can reference frames before.

P-Frames Macroblocks in P-Frames can only reference macroblocks in preceding P-Frames and I-Frames. In practice the number of reference frames is limited to the two closest preceding P or I-Frames.

B/BREF-Frames Additionally to the previously mentioned P-Frames macroblocks in these frames may also reference macroblocks in future frames. B-Frames are the most space efficient frames. B-Frames can not be referenced by other frames. Only B_{REF}-Frames can be used as reference by other frames.

The x264 encoder generates a list of `I P B P B P ... I P B P B ..` frames [CJ11]. About every 70 frames an IDR-Frame is inserted to enable good seeking performance. Because the H.264 video codec was designed also for streaming, regular IDR-Frames are beneficial to restore the full picture in case of dropped frames. The exact amount of P and B-Frames is not specified by the H.264 specification [Ric11]. For our tests only the x264 implementation was investigated.

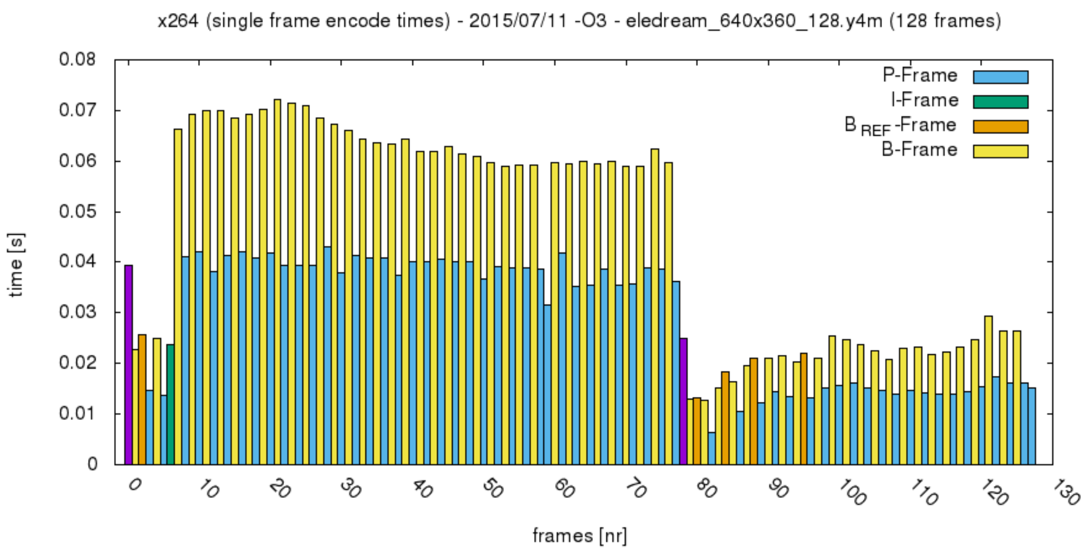


Figure 4.5: Frame encoding times and types of a sample input video (Elephants Dream [Roo+06; Bie11] 640x480 128 frames). In frame 10-73 the camera is moving and the frame content is changing a lot. From frame 74-128 the camera is steady and only parts of the scene are changing. The data was measured using debug output of the original x264 implementation encoding the Elephants Dream video [Roo+06].

Encoding times and decided frametypes of a sample input video (Elephants Dream [Roo+06; Bie11]) are shown in Figure 4.5. Because B-Frames have more possible reference frames the encoding of these frames takes longer by a factor of two. IDR-Frames are shown in pink and can be seen at frame number 0 and 77.

4.7.3 Testdata

We used the Parsec [Bie11] input sequences of the open source movie Elephants Dream [Roo+06]. The inputs are stored as uncompressed *yuv* image sequences.

In particular we used the small instance containing only 32 frames (`eledream_640x360_32.y4m`). As large instance we used the 480 frames sample (`eledream_640x360_480.y4m`).

4.7.4 Implementation

To our knowledge, there exists no up-to-date documentation about the x264 code base. The x264 code is partly documented using C source file comments. Because of the very specific complexity of video encoders a lot of effort has been put into understanding the way x264 works. In this short overview of the x264 architecture important functions are explained to make x264 Pipeline modifications more understandable for the reader.

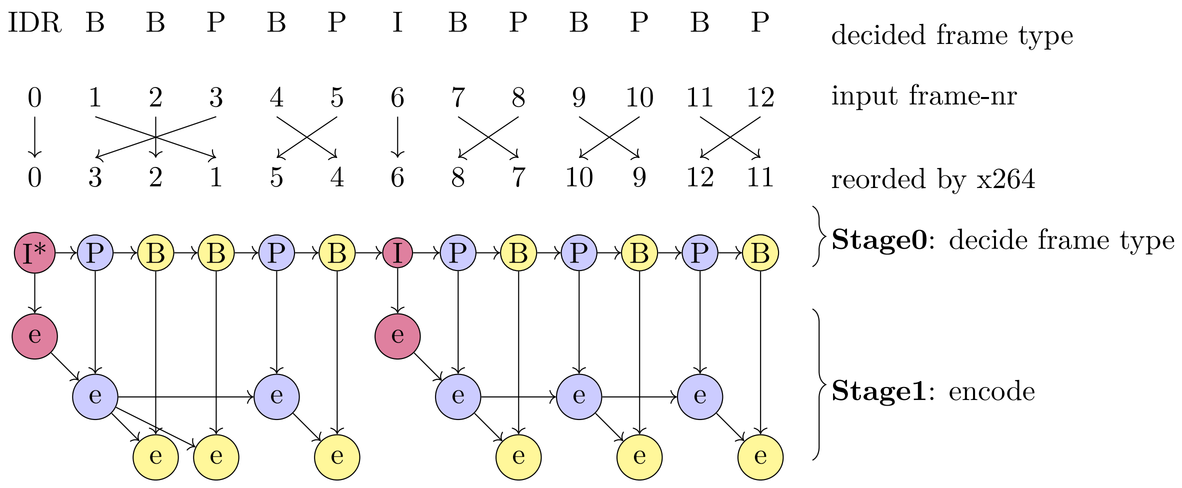


Figure 4.6: This figure shows the Pheet Pipeline x264 version. It shows how the first twelve input frames are re-ordered by x264 and how the frame type is decided. *I-Frames* can only be self-referencing on a macroblock level. *P-Frames* may refer I or P-Frames up to the next preceding I-Frame, whereas *B-Frames* may also refer frames ahead. This diagram shows a Pipeline schema applied on a frame level. Lee et al. used a macroblock-level Pipeline [Lee+13] which can achieve more parallelism but is beyond the scope of this thesis.

In general x264 uses Pthreads for parallelism. The threads are grouped into multiple pools executing tasks from central queues. The synchronization between the threads is done using Pthread mutexes. X264 uses the following threads in the default configuration:

Main thread This thread is used to parse command line arguments and does all data structure initialization. It then waits for frames read by a lookahead thread. The frame type is decided and frames are passed to the encoding thread pool. Synchronization between the encoder threads is done in the main thread.

Lookahead threadpool This pool reads frames from the input file and analyses them. In case parallelism is enabled using command line options, multiple lookahead threads are started. The actual number is based on the used encoding settings.

Lookahead manager thread This manager thread orchestrates the lookahead pool threads.

Encoder threadpool The threads of this thread pool are used to encode frames. The corresponding function is named `x264_slices_write`. Each thread encoding a frame with dependencies waits on a macroblock level for the depending thread to proceed. The number of encoder threads is set with the command line argument `threads`.

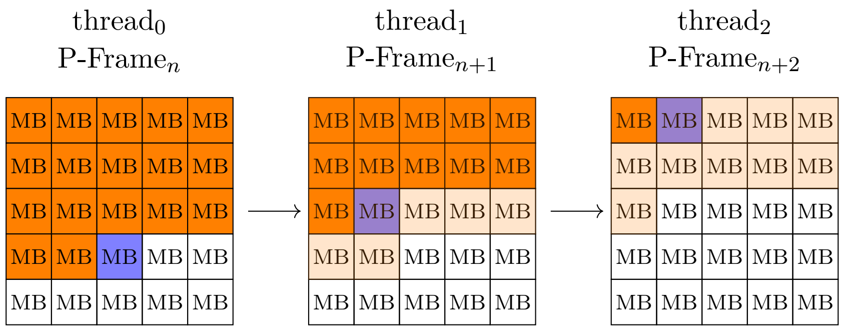


Figure 4.7: X246 P-Frame macroblock (MB) encoding dependencies [CJ11]. A P-Frame macroblock can refer to macroblocks in the preceding P-Frame up to his own position. Macroblocks are encoded row-wise. Once a macroblock is encoded the next thread encoding the following P-Frame can continue up to the just completed position. This interleaving is the main source of parallelism in x264. The current macroblock is colored blue. The already completed macroblocks are colored orange. Macroblocks ready for encoding are shown in a light orange.

The general state is stored in one big data structure (**`struct`** `x264_t`) which exists for each encoder thread. After an encoder thread has finished its work, the data structure is cleaned and reused for the next thread.

Data from the analyze threads is also passed using the global data structure. The fact that so much data is passed using *global variables* and the high amount of data structure *reuse* makes it very challenging to adapt the algorithm to a Pipeline schema.

The main parallelism in x264 arises because of the interleaving of macroblock encoding. As shown in Figure 4.7 the encoding of P-Frame$_{n+1}$ can start even before P-Frame$_n$ is complete. Each P-Frame macroblock can reference preceding macroblocks in previous frames up to his own position. A dedicated thread is running for every P-Frame in Figure 4.7. The already encoded macroblocks are colored in orange. The macroblock currently encoded by threads are colored in blue. Macroblocks which are ready to be encoded because of their satisfied dependencies are colored in a light orange.

Cilk-P Implementation

Lee et al. [Lee+13] already ported the algorithm to a Pipeline version using the Cilk-P [Lee+13] interface. We use some of their results with their kind permission.

Their version used Pipeline paralellsim on a macroblock level which yields good speedup.

Pheet Pipeline Implementation

Our Pheet implementation only relies on *IDR-frames* for parallelism. All frames between two IDR-frames only depend on the preceding IDR-Frame. This schema can be seen in

Figure 4.6. As a simplification we treated B-Frames as P-Frames.

Pthread Reference Implementation

The Pthread reference implementation is the unmodified x264 version from the Parsec suite [Bie11]. We restricted the number of concurrently running threads using the hwloc [Bro+10] library.

This version uses a lot of data structure reuse and features macroblock level parallelism. It is carefully optimized and considered as very fast H.264 encoder.

4.8 Ferret

The Ferret [Lv+07a] benchmark of the Parsec [Bie11] suite is used to perform content-based similarity search and was developed at Princeton University. Ferret just provides base functionality every similarity search needs. Specific implementations for images, audio data and 3D data are added with data type plugins. Ferret features a comparable performance to domain specific algorithms [Lv+07a].

The Parsec Ferret benchmark provides different input instances which consist of a database containing reference images and a number of query image files. During the benchmark the closest k matches of each query file in the reference database are computeted. The results are written into the output file including their similarity score.

4.8.1 Aim of this benchmark

The benchmark was included into the Parsec suite because it covers the emerging field of “next-generation desktop and internet search engines for non-text document data types” [Bie11]. It was included in this thesis because the problem can be easily addressed with the Pipeline pattern. In addition the near ideal speedup of the Pipelined version observed by the Parsec authors is promising.

4.8.2 Testdata

The *simlarge* benchmark instance of the Parsec suite [Bie11] was used for benchmarking. This instance contains 256 queries (images) and a database with 34 973 images. The goal is to extract the top $k = 10$ closest images for each query [Bie11].

As second instance the *native* Parsec instance was used. This instance contains 3500 queries and a database with 59 695 images. The closest $k = 50$ images are extracted [Bie11].

4.8.3 Expectations

Each query image is processed in a single iteration. The first stage is a serial stage although it only reads the filenames of the query images. After this stage all query images are processed in parallel. The last serial stage writes the result values in the output file.

We expect good scalability because even the small instance contains 256 query images which are processed in parallel. Bienia [Bie11] calculated a theoretical possible speedup of 16 for 16 threads.

Because of the 10 cores per chip on Mars and 16 on Ceres, we expect a drop in speedup afterwards because of cross chip memory access. For the Saturn system we expect the drop to be at about 24 threads because two CPUs with 12 cores each are on one die. After these thread counts cross chip memory access is expected to slow down the computation. The memory architecture of the Xeon Phi card in the Pluto system is quite different, therefore we can not give a prediction on the speedup for this system.

We also expect the runtime of the bigger *native* instance to be about 10× longer than the *simlarge* instance because of a 10× bigger query count.

Because the Pthreads version uses per stage central queues we expect that our Pipeline version, which uses *work-stealing*, has a better scalability for higher thread counts. We expect a better performance for the Pthread version for low thread counts because of the Pipeline overhead.

4.8.4 Implementation

The Ferret algorithm can be modeled with six Pipeline stages (schema and stages are shown in Figure 4.8):

1. *Load* is the *serial* input stage. It loads a single query from the queries directory. Each query is represented as a single file. In this benchmark the queries are JPEG image files which are read into memory in this stage.
2. *Segment* is a *parallel* stage which performs segmentation of the image into similar regions. A similar region may have the same color or same contrast information.
3. *Extract* is a *parallel* stage and extracts feature vectors from found segments. Feature vectors pose a dimension reduction and make it possible to compute the distance between two segments.
4. *Vector / Filter* is a *parallel* stage which uses *Locality Sensitive Hashing* [Lv+07b] to index candidate sets [Bie11]. The distance between vectors of the query image and database candidates are first computed using a fast algorithm. Results are used to filter out candidate images which are not similar at all.
5. *Rank* is a *parallel* stage which uses *Locality Sensitive Hashing* [Lv+07b] to rank the different candidate sets [Bie11]. Because the exact distance calculation between two feature vectors takes more time, only filtered candidates are considered. After ranking the best k reference images are saved.
6. *Output* is a *serial* output stage. For each query the k best matching candidates from the database are written into the output file. Also the similarity score is saved.

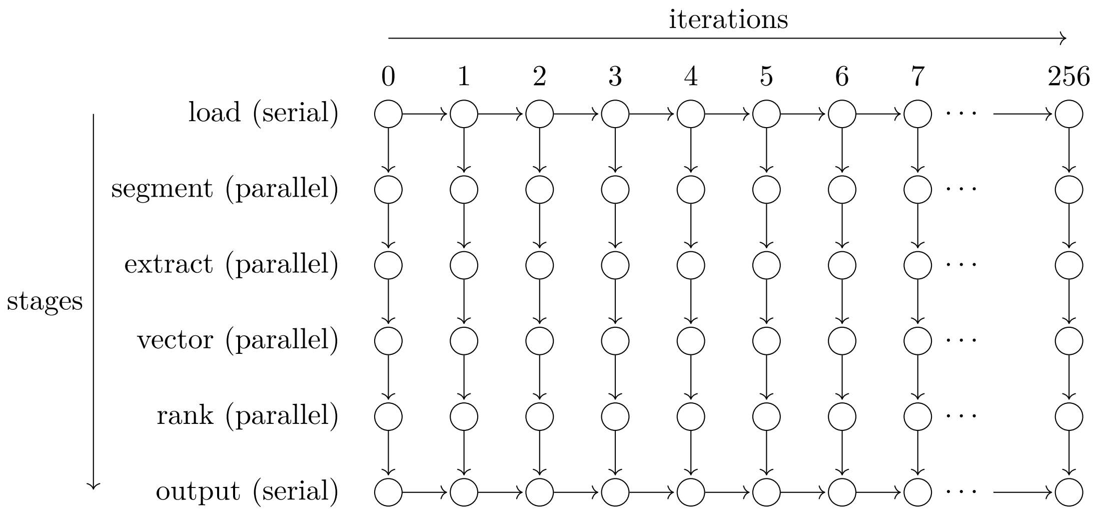


Figure 4.8: The Ferret Pipeline schema showing six work stages and 256 iterations. Each query can be executed in parallel. Results of each query are stored in a single file and require a serial output stage.

There exists an Intel TBB Pipeline version of Ferret in the Parsec suite. The Intel TBB Pipeline interface is similar to our Pipeline interface. Therefore it was easy to port the Intel TBB Ferret version to our Pipeline interface.

Listing 4.1 shows our Pheet Pipeline Ferret implementation. Mainly the nested Lambda expressions which are necessary to provide dynamic Pipelines are shown. Because all operations perform their changes in-place, no result values are passed between iterations. The current work item is passed via the capture list of the closure (see Section 3.3.1 about the C++ Lambda expression).

4.8.5 Reference Implementation

We used the existing Ferret Pthreads implementation included in the Parsec suite [Bie11] for reference.

For each stage a number of worker threads is started. Because we limit the number of threads globally we have to reduce the threads per stage to obey our global limit.

Each stage has an own queue which is used by the worker threads to obtain work. The queue length was set to 100 elements for our benchmarks. Access to the queue is protected using Pthread *mutex locks*. We expect this central lock structure to be a bottleneck for executions with a high number of threads.

```
#define STAGE_SEQUENCE 1
#define STAGE_EXTRACT 2
#define STAGE_VECTOR 3
#define STAGE_RANK 4
#define STAGE_OUTPUT 5

Pipeline penv { ppc }; // initialize the Pipeline Environment
int counter = 0; /* iteration counter */

while( (item= this->filter_load(nullptr,ref_m_single_file, ↩
    ref_m_path,m_path_stack,m_dir_stack)) != nullptr) {
  Pipeline::PipelineIteration* piter =
    new Pipeline::PipelineIteration(&penv, counter);
  counter++;
  piter->pipe_continue(STAGE_SEQUENCE,0, [piter,this,item] () {
    this->filter_seg(item);
    piter->pipe_continue(STAGE_EXTRACT,0, [piter,this,item] ()  ↩
        {
      this->filter_extract(item);
      piter->pipe_continue(STAGE_VECTOR,0, [piter,this,item] () ↩
          {
        this->filter_vec(item);
        piter->pipe_continue(STAGE_RANK,0, [piter,this,item] () ↩
            {
          this->filter_rank(item);
          piter->pipe_wait(STAGE_OUTPUT,0, [piter,this,item] () ↩
              {
            this->filter_out(item);
            piter->finished(0);
          });
        });
      });
    });
  });
}
```

Listing 4.1: This listing shows Ferret implemented in Pheet Pipelines. No results are passed between iterations. The Lambda capture list is used to pass iteration items from stage to stage.

4.9 Synthetic

A synthetic benchmark was created to measure the overhead of our Pipeline Scheduler and the scalability on many core systems. This synthetic Pipeline benchmark simulates work by performing *matrix rotations* in multiple Pipeline stages and iterations.

To be able to control the amount of *work* in the Pipeline, different numbers of iterations have been benchmarked. A 128×128 matrix was rotate 256 times in 1000 and 10 000 iterations.

The matrix rotation was picked because it needs n^2 work for a $n \times n$ matrix and can be performed *in place* which is very memory efficient. Each iteration has a dedicated matrix instance which is rotated 90° counterclockwise in each stage.

We implemented this benchmark in the following versions:

Pipeline - Wait This version has dependencies between all adjacent iterations. It is called *wait* version from now on. The Pipeline schema of the wait version can be seen in Figure 4.11. The Pipeline with 256 stages was benchmarked with 1000 iterations for the small instance and with 10 000 iterations for the large instance. The huge Pipeline depth of 256 stages should enable enough parallelism in the *wait* version. It is used to measure the overhead of the dependency waiting logic of our Pipeline implementation.

Pipeline - Continue This version only consists of parallel stages. The structure of the Pipeline is the same as in the wait version. It has no dependencies between iterations. It is called *continue* version from now on. The schema of this dependency-free version can be seen in Figure 4.10. The first serial stage only starts iterations and does not perform work on its own. Therefore it is not a limiting factor. This version should measure the overhead of the Pipeline object creation.

Pheet Tasks This version calculates the matrix rotations using a dedicated *Pheet task* for each iteration. This results in 10 000 tasks performing 256 rotation operations in a row for the large instance. For the small instance only 1000 tasks were spawned. The tasks are executed by Pheet worker threads. The number of worker threads is controlled by the Pheet Environment to allow comparisons of different worker counts.

C++11 Threads This version uses *C++11 Threads* to start a dedicated thread for each iteration. The number of concurrently running threads is limited with the hwloc [Bro+10] library for comparable results. This version is used to measure the overhead of the Pheet scheduler.

On the Ceres system the hwloc library does not allow arbitrary thread limitations. Therefore this benchmark starts a number of worker threads which rotate more than one matrix instance. This keeps the overall number of matrix rotations the same.

The above mentioned Pipeline versions can be executed with or without a *throttle* limit (see Section 2.1.2). This limit reduced the amount of concurrently active iterations.

4.9.1 Aim of this benchmark

This benchmark should measure multiple characteristics of our Pipeline implementation in a controlled environment.

Wait vs Continue It should measure the scalability of the *wait* version compared to the *continue* version. A *wave front* execution of the *wait* Pipeline schema should yield enough parallelism because of the huge Pipeline depth of 256 stages. It should show how the additional dependency bookkeeping slows down the speedup of the wait version.

Continue vs Pheet task It should measure the overhead of a Pipeline execution without dependencies (continue-version) compared to a reference Pheet task version. This will give us the overhead of the Pipeline bookkeeping which mainly consists of creating iteration object instances.

Pheet tasks vs C++ threads Additionally we check the overhead of the Pheet Scheduler compared to an execution with operating system threads. This will show how much the Pheet scheduler overhead effects our implementation in general.

Throtteling It should show the effects of *throtteling* concurrently running iterations on the performance. Throtteling should increase the locality and create a lower memory usage in general. The memory usage is lower because the number of currently running, yet unfinished, iterations is constrained.

4.9.2 Testdata

The testdata consists of $n = 10\,000$ or $n = 1000$ iterations and 256 stages. The matrix size is always 128×128 and contains values of type `int` with a size of 4 byte on our test systems. The huge number of iterations should mitigate startup effects of the Pipeline.

The small instance runs 1000 iteration which perform 256 rotations each. This leads to a total memory requirement of 62.5 MB for the matrices.

The large instance uses 10 000 iterations and the same number of stages. This instance requires 625 MB of main memory for the matrix instances.

4.9.3 Expectations

Overall we expect that the *continue* version outperforms the *wait* version and that the *C++11 Thread* version outperforms all other versions because of less overhead.

In detail we expect:

Wait vs Continue We expect better scalability for the *continue* version than for the *wait* version because it features more parallelism. We also expect a faster total run time for the *continue* version on executions with more than one thread.

The difference between wait and continue version for an execution performed with only one thread can give us insight about the overhead of the *wait* version. We expect this overhead to be neglectable for bigger matrix sizes.

We can not give a specific speedup expectation, but the independence of the data should be beneficial even for high thread executions. Also the *wait* version should benefit from more threads by doing a *wave front* execution.

Continue vs Pheet task We expect that the Pipeline *continue* version and the Pheet task version perform exactly the same. The *continue* version only has a very slight overhead for creating a Pipeline iteration instance compared to the plain Pheet task version.

The Pipeline iteration object creation is not thread safe and has to be performed in a single thread. Pipeline iteration objects have to be registered in a central data structure in the Pipeline Environment. We expect some contention for high thread counts because of this.

Pheet Tasks vs C++ threads The C++ threads are directly scheduled by the operating system whereas the Pheet tasks are executed by worker threads. The bookkeeping of this tasks queues is expected to have a negative impact on the Pheet task version.

To our knowledge no comparisons of Pheet tasks and C++11 threads have been published.

Because we create a C++ thread for each iteration we have to limit the concurrently running threads to the number of Pheet worker threads for comparable results. It is unclear if this thread limitation done using the hwloc [Bro+10] library has a direct impact on the performance.

We modified the benchmark on the Ceres system to feature C++ working threads, because the Ceres system does not allow pinning of threads. This is expected to affect performance because less threads mean less overhead and allow better compiler optimizations.

We tested two Pheet schedulers [Wim14] (BasicScheduler, StrategyScheduler) to investigate the effect of the chosen scheduler on the performance. See Section 1.2.2 for details about Pheet schedulers.

Throtteling We run the benchmark with a throttle limit of 512 iterations and compared it to a run without a throttle limit. We expect that this restriction of concurrently running iterations increases the locality and therefore reduces the cache misses. A

$$\begin{aligned} a &= m[y, x] & &\equiv m[0,3] \\ b &= m[x, d-1-y] & &\equiv m[3,4] \\ c &= m[d-1-y, d-1-x] & &\equiv m[4,1] \\ d &= m[d-1-x, y] & &\equiv m[1,0] \end{aligned}$$

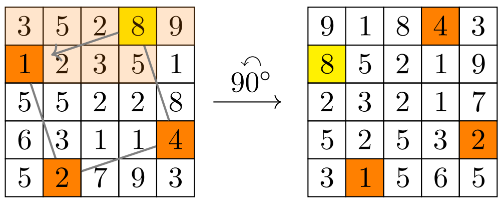


Figure 4.9: In place rotation of a (quadratic) matrix by 90° counter clockwise. All items addressed by the two nested for loops are shown in a light orange. The yellow cell is an arbitrary cell ($y = 0$, $x = 3$) in range of the two loops. Grey arrows show how the four cells are shifted to perform the rotation.

better overall run time is expected for the throttle version in executions with high thread counts.

4.9.4 Implementation

The algorithm to rotate the matrix 90° counter clockwise is shown in Algorithm 4.1. For easier understanding the rotation of a single cell is shown in Figure 4.9. The light orange areas show the area of the two nested loops. The yellow square is an arbitrary cell of the matrix. The grey arrow shows the shift of the other touched cells (orange).

Algorithm 4.1: Rotate Matrix 90° counter clockwise

```
Input: A reference to the matrix $m$ and a scalar dimension $d$
Output: The rotated matrix 90° counter clockwise in place
for $y \leftarrow 0$ to $d/2$ do
    for $x \leftarrow y$ to $d-1-y$ do
        $a \leftarrow m[y][x]$ ;
        $b \leftarrow m[x][d-1-y]$ ;
        $c \leftarrow m[d-1-y][d-1-x]$ ;
        $d \leftarrow m[d-1-x][y]$ ;
        rshift $a, b, c, d$ ;    // right shift:  $a \leftarrow b$;  $b \leftarrow c$;  $c \leftarrow d$;  $d \leftarrow a$
    end
end
```

4.9.5 Reference Implementation

To measure the overhead of our Pipeline implementation, a version using Pheet tasks and a C++11 thread version have been created.

Both reference implementations are only compared to the *continue* Pipeline version and thus do not perform any synchronization between threads. The reference versions execute

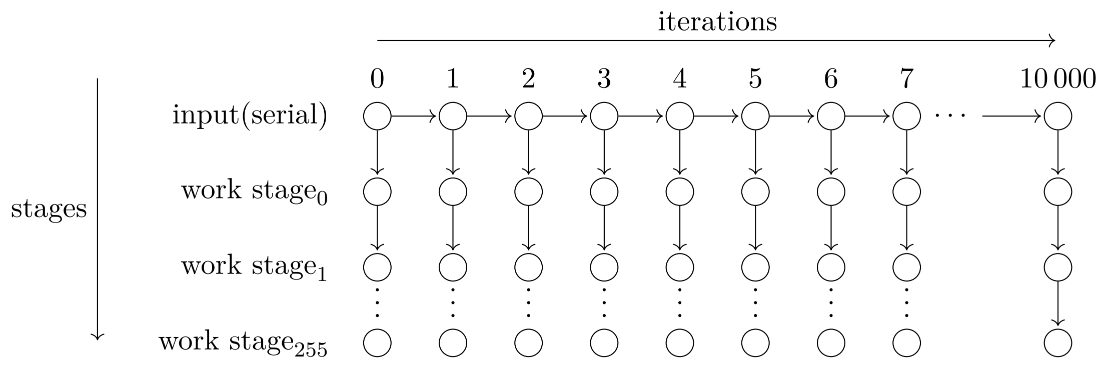


Figure 4.10: Continue version of the synthetic benchmark Pipeline with 256 work stages and the $n = 10\,000$ iterations. This diagram shows the large test instance. The small instances instead uses 1000 iterations. Each stage in the Pipeline performs a single matrix rotation and *waits* for the preceding stage of the same iteration. All work stages are parallel stages to benchmark the *continue* version code path and determin the general overhead of our Pipeline implementation.

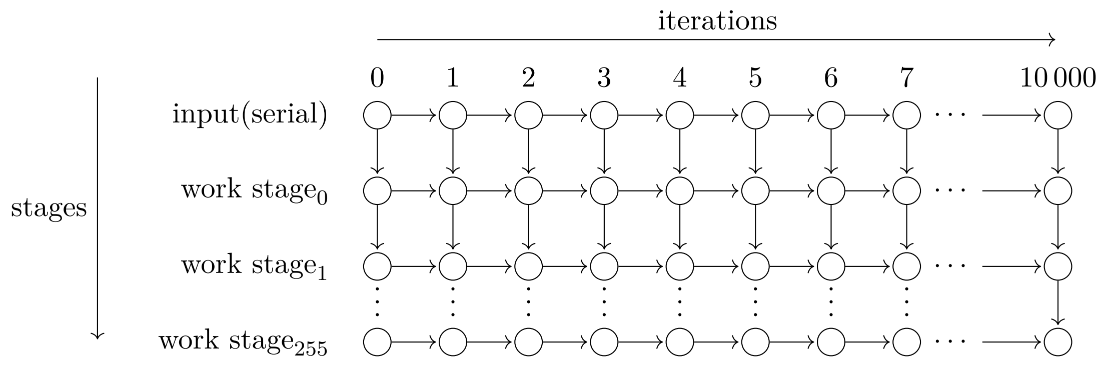


Figure 4.11: Wait version of the synthetic benchmark Pipeline with 256 work stages and the $n = 10\,000$ iterations. All work stages are serial stages where every iteration waits for the preceding iteration. This benchmark shows the additional synchronization overhead of the wait version.

256 matrix rotations using recursive function calls. The Pipeline version calculates each matrix rotation in a dedicated stage which yields more overhead.

Pheet Reference

The Pheet task version creates a task for each iteration. The tasks are executed by the Pheet scheduler using worker threads. The Pheet Environment allows to restrict the number of worker threads to test scalability.

Because no synchronization is used in this version it provides an upper bound and shows the maximum speedup for the given Pipeline structure. For each iteration a Pheet task was created which results in 10 000 tasks. The concurrently running Pheet tasks are restricted using the Pheet Environment object.

C++ Reference

A version using *C++11 threads* to measure the overhead of the Pheet scheduler was created. In Linux C++11 threads are implemented with Pthreads.

In the C++ version for each iteration a C++ thread was created. The number of concurrently running threads was restricted using the *hwloc* [Bro+10] library.

CHAPTER 5

Benchmark Environment

5.1 Mars - 8 Intel Xeon

The first system used for benchmarks has eight Intel Xeon E7-8850 CPUs with ten cores each. Cores run at a frequency of 2 GHz and have a shared 24 MB L3 cache per CPU[1]. Each core can run two threads (Hyper Threading) to hide memory latency resulting in a total of 160 threads. Each core features a 32 kB L1 cache per thread and a shared 256 kB L2 cache per core. The machine is a fully cache coherent NUMA system with eight NUMA nodes. The schema can be seen in Figure 5.1.

The theoretical speedup limit is 80 (8 CPUs $\times$ 10 cores).

It has a total of 1 TB main memory and is running a *Debian Testing* on a `4.1.0-1` Linux kernel. We used *gcc* `5.2.1`. Additional tests were performed with the Intel 64 bit C++ compiler `icpc` in version `14.0.1`.

We used hwloc `1.11.2` for our tests on this system.

In the performance analysis of a similar hardware configuration the Cern Openlab [SJ11] data shows a first break in speedup at around 20 threads. The next break happens at around 40 threads but there is still speedup till 80 threads. They used the *HEP-SPEC2006* benchmark [Spe] for their measurements.

5.2 Saturn - 4 AMD Opteron

The Saturn benchmark system is equipped with four AMD Opteron 6168 CPUs. Each of these CPUs has twelve cores running at 1.9 GHz and has 12 MB of L3 cache. Each core

[1]Intel. *Intel® Xeon® Processor E7-8850.* URL: `http://ark.intel.com/products/53575/` (visited on 05/01/2016).

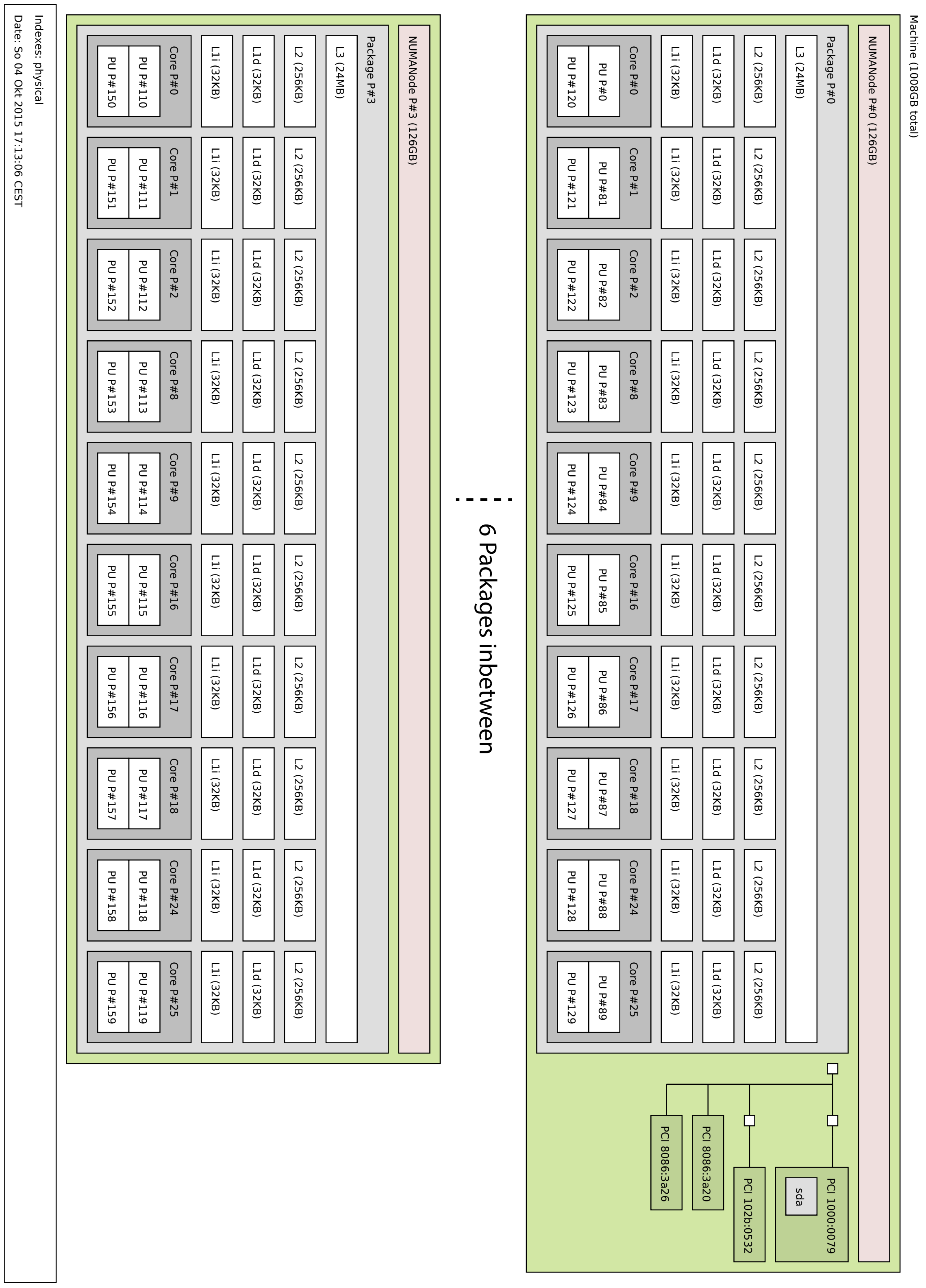


Figure 5.1: Processor and memory topology on the *Mars* benchmark system. The system is equipped with eight Intel Xeon E7-8850 CPUs having 10 cores each. Each CPU can run two threads resulting in a total of 160 threads. For easier readability six CPUs in between have been omitted. Diagram generated with *lstopo* which is part of the *hwloc* [Bro+10] library.

features 512 kB of L2 cache. Two CPUs are on one die thus NUMA effects are expected to take place above 24 threads.

The system has a total of 128 GB of DDR3-1333 memory and is running a *Debian Testing* on a `4.6.2-2` Linux kernel. We used *gcc* `5.4.0`.

The theoretical speedup is 48 (4 CPUs $\times$ 12 cores).

We used hwloc `1.11.3` for our tests on this system.

5.3 Ceres - 4 Oracle SPARC T5

The SPARC system consists of four Oracle SPARC T5-4 CPUs each having 16 cores (SPARC S3). The system has a total 1 TB of DDR3 main memory and is running Oracle Solaris 11 (SunOS `5.11`) and gcc `4.8.2`. The overall topology is shown in Figure 5.3.

The theoretical speedup limit is 64 (4 CPUs $\times$ 16 cores). Although the parallel execution of up to two instructions in once cycle may increase this limit.

Each CPU has 8 MB L3 cache which is not shared among CPUs. Interconnection distance between two arbitrary CPUs is always only a single hop. Each core has 128 kB of L2 cache and can run eight threads which results in a total of 512 threads.

The large number of eight threads per CPU is used to hide memory latency (cache misses, long latency) from the execution. The schema of the Sparc S3 core is shown in Figure 5.2. In each cycle one out of eight ready threads is picked. This prevents wasting of core cycles when one thread experienced a cache miss. To avoid starvation a LRU (least recently used) algorithm is used. Each core features two pipelines resulting in up to two instructions executed per cycle (2-way superscalar). Instructions are executed *Out-of-Order*[2] resulting in good single thread performance.

The system is a *cache coherent* NUMA system (cc-NUMA). Oracle Solaris uses a *First Touch*[3] policy which means that the thread using the data for the first time *owns* it. Touched data will be stored in a memory space connected to the processor executing the thread. In practice this means that data initialization should be parallelized and each thread should initialize his portion of data.

We used hwloc `1.11.0` for our tests on this system. The Sun C++ compiler `5.13` lacks support for `std::atomic` and therefore could not be used to compile the Pheet framework. g++ `4.8.2` also had some issues with `std::to_string` which could be mitigated by patching this method in our implementation.

[2] Oracle. URL: `http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/o13-024-sparc-t5-architecture-1920540.pdf` (visited on 10/10/2015).

[3] Oracle. URL: `http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/o13-060-t5-multicore-using-threads-1999179.pdf` (visited on 10/10/2015).

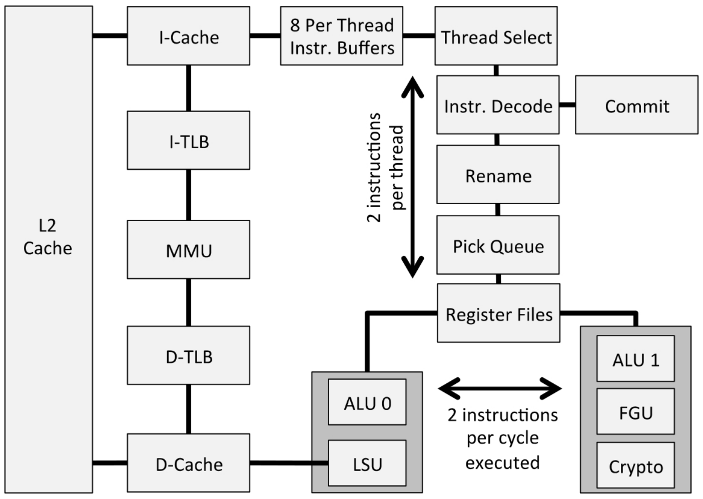


Figure 5.2: Block diagram of the SPARC S3 core used in our SPARC T5-4 ceres system. It shows the two pipelines which execute two instructions per cycle. In addition the eight instruction buffers for the eight threads per core are shown in the upper part of the schema. During each cycle one of the ready threads is selected and executed [Orab].

5.4 Pluto - 2 Xeon Phi Coprocessors

The host system utilizes two Intel Xeon E5-2650 with eight cores each. It has a total of 256 GB of main memory. The main system is running with CentOS 7 on a `3.10` Linux kernel.

The system is equipped with two Intel Xeon Phi 7120P extension cards. Overall topology including extension cards is shown in Figure 5.5.

Each of the cards runs a customized Linux on 61 cores with 16 GB of memory. The full featured Linux system runs with a customized kernel (`2.6.38`). It features 244 software cores (61 cores $\times$ 4 threads) running at 1238 MHz. Each core has 512 kB of L2 locally.

The theoretical speedup limit are the 61 cores of a single coprocessor card.

The schema of the coprocessor card is shown in Figure 5.4. Only six of the 61 cores are shown for visual reasons. Each core has an associated L2 cache which is kept fully coherent using the distributed TD (Tag Directory)[4]. All components are connected by a ring interconnect [Rei12].

Whilst the Intel Xeon Host CPUs are cache coherent and share access to the same main memory, the Xeon Phi is a cache coherent SMP (Symmetric Multi-Processor) which is connected to other cards and devices only via PCIe bus. There is no hardware cache

[4]Beilun Wang, Lin Gong, and Chunkun Bo. URL: `https://www.cs.virginia.edu/~lg5bt/files/Intel%20Xeon%20Phi%20Coprocessor.pdf` (visited on 10/10/2015).

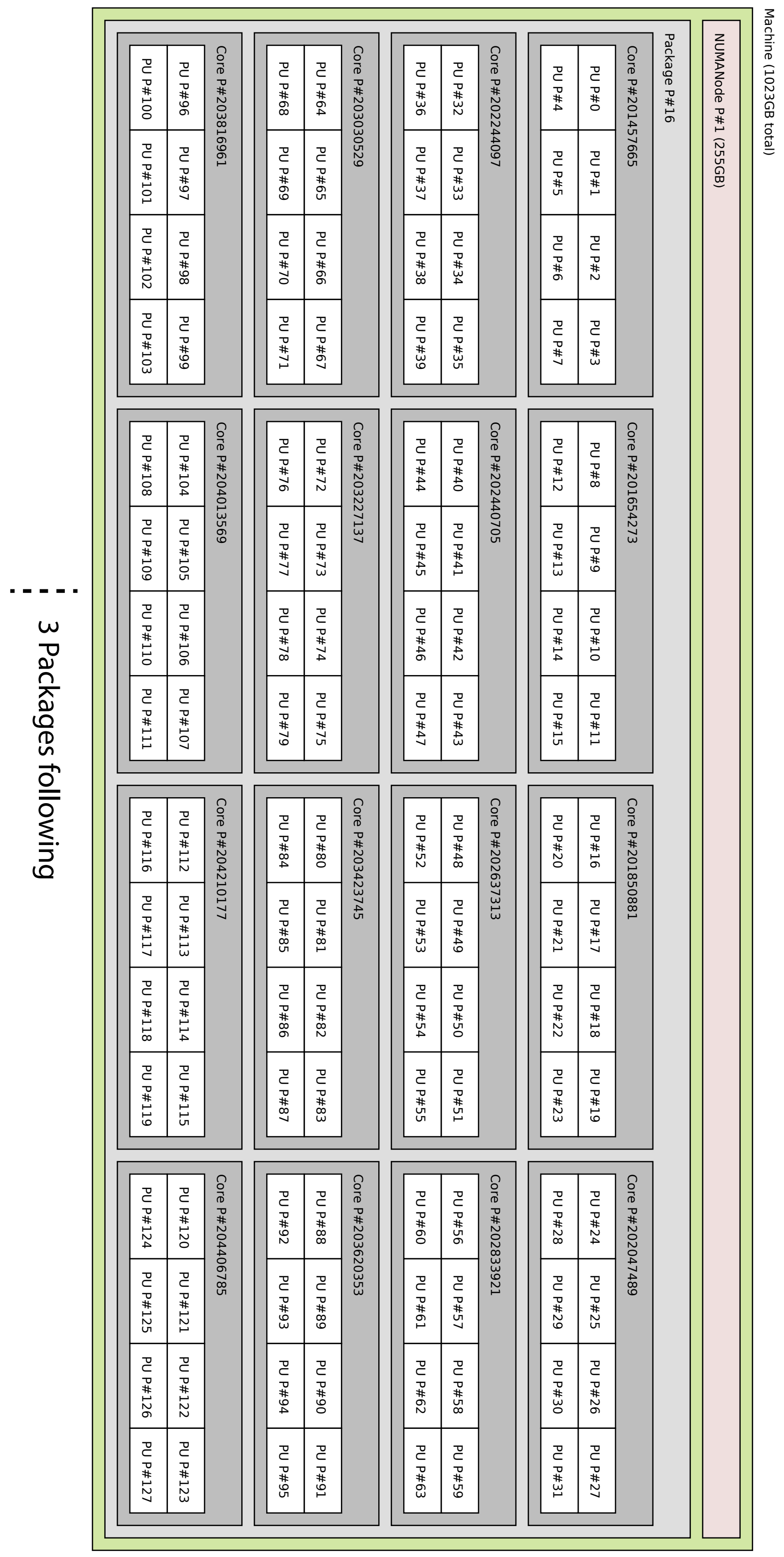


Figure 5.3: Processor and memory topology on the *Ceres* benchmark system. The system is equipped with four SPARC T5-4 processors. Each processor has 16 cores running at 3600 MHz. Each core can run eight hardware threads resulting in a total of 512 threads. For easier reading only the schema of the first of the four CPUs is displayed. Diagram generated with *lstopo* which is part of the *hwloc* [Bro+10] library.

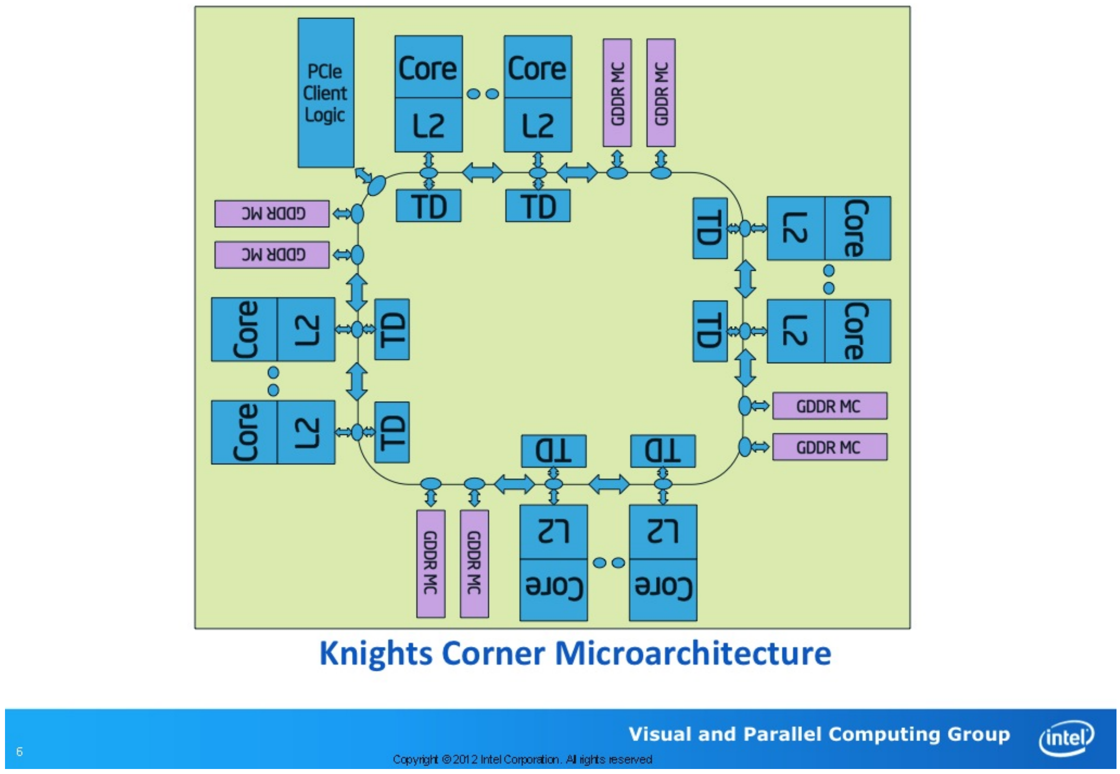

PCIe Client Logic
Core
L2
TD
GDDR MC
Knights Corner Microarchitecture
6
Visual and Parallel Computing Group
Copyright © 2012 Intel Corporation. All rights reserved
intel

Figure 5.4: Internal architecture of the Intel Xeon Phi coprocessor card. Only a subset of the 61 cores is displayed for visual reasons. TD (Tag Directory) is distributed and used to keep different L2 caches fully coherent with each other. CPUs are connected using a ring interconnect [Rei12].

coherency between two Xeon Phi coprocessors or between a Xeon host CPU and a Xeon Phi coprocessor [Rei12].

The large number of threads per core and the 512-bit SIMD instructions are key to performance. SIMD instructions perform operations on multiple data elements at once. Intel recommends to run at least two threads on each core as this always improves the overall performance [Rei12].

Our Pipeline implementation is not optimized for the SIMD instruction set, but the Intel compiler may optimize the code of our benchmarks.

To run applications on the extension cards the Intel compiler flag `-mmic` has to be used. All used libraries also have to be compiled with this flag. The application is executed using a special Intel Loader (`micnativeloadex`) [Ama14]. We used the Intel 64 bit compiler `icpc` in version `14.0.01`.

Previous versions of gcc (< 5) did not support Intel MIC instructions. This offloading support was added partially in gcc 5 which allows OpenMP offloading to Intel MIC targets[5]. Intel recommends gcc (patched MIC architecture support) only for building the embedded Linux kernel but not the actual application because it lacks SIMD instruction support. The Intel XE composer studio compiler is recommended to build applications because it offers full vector instruction support on the MIC architecture[6].

Benchmarks were run entirely native on one of the coprocessor cards. The host system was only used to cross compile binaries but was not utilized in the benchmarks itself. It is possible to execute a program on the host system and offload work to both of the coprocessor cards which is called *hybrid execution* [Dok+12]. Hybrid execution goes beyond the scope of this thesis and is not explained in detail. It would be beneficial for performance and is left open for future research. It is noted that the newly released gcc 6.1 supports hybrid executions.

[5]GCC. *GCC5 Offloading Support.* URL: `https://gcc.gnu.org/wiki/Offloading` (visited on 05/01/2016).

[6]Intel. *Intel and Third Party Tools and Libraries available with support for Intel® Xeon Phi^{TM} Coprocessor.* Oct. 2014. URL: `https://software.intel.com/en-us/articles/intel-and-third-party-tools-and-libraries-available-with-support-for-intelr-xeon-phitm` (visited on 10/10/2015).

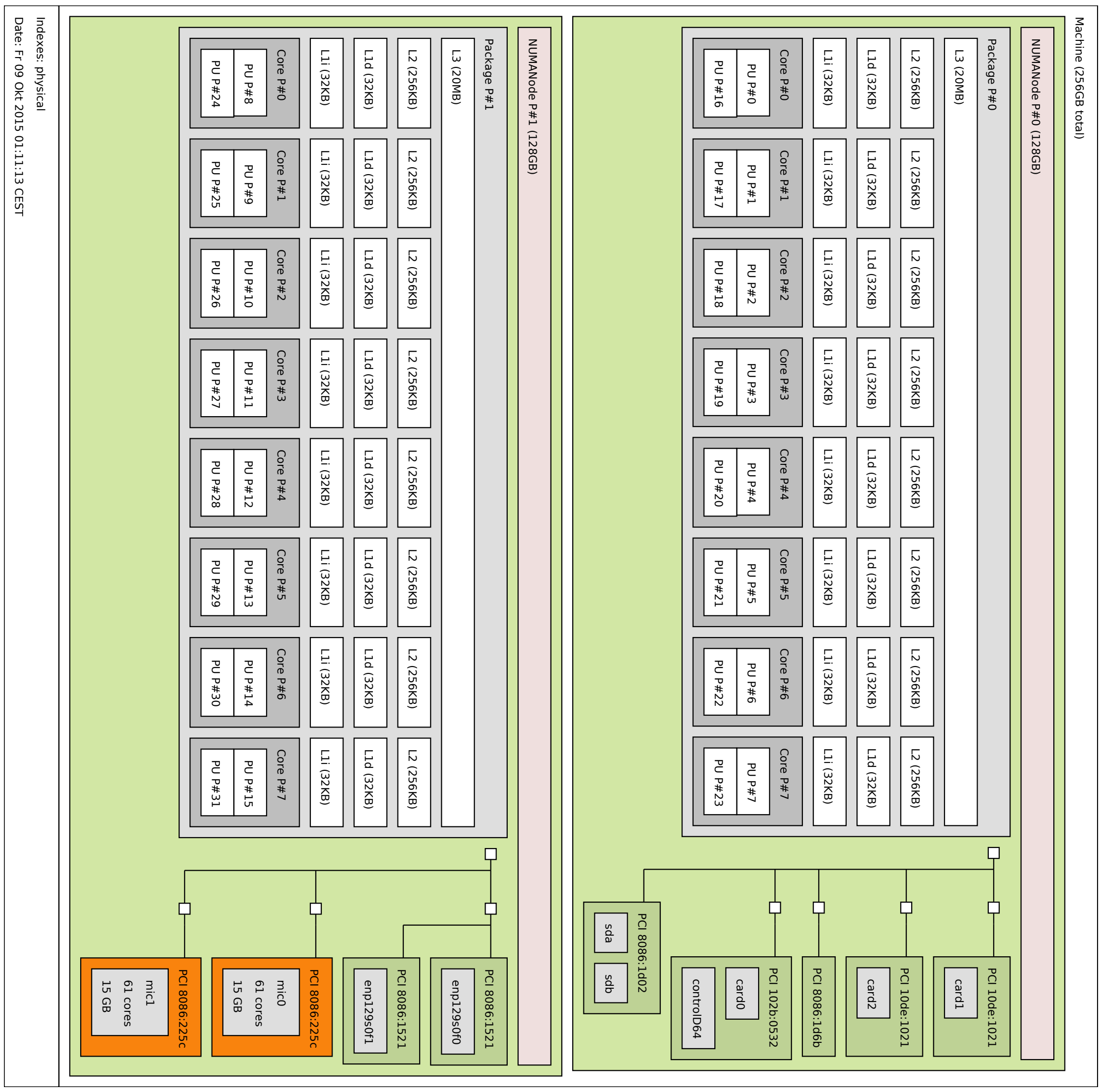


Figure 5.5: Processor and memory topology of the *Pluto* benchmark system. Diagram generated with *lstopo* which is part of the *hwloc* [Bro+10] library. This topology mainly shows the host system containing two Intel Xeon CPUs. The two Xeon Phi extension cards connected via the *PCIe bus* are visualized in *orange* at the bottom right. Each extension card has 61 cores and 15 GB of memory.

CHAPTER 6

Benchmarks Results

6.1 Dedup

The results of the Dedup benchmark can be seen in the following figures:

- Figure 6.1 shows the absolute benchmark execution times on a linear scale. Only the thread range from 1 to 32 is displayed here to allow comparison with related work.
- Figure 6.2 shows the speedup relative to a single thread execution. Execution speeds for single thread executions differ significantly between the different implementations. Therefore this plot is only used to measure the scalability of the different implementations.

6.1.1 Results

We expected a speedup of at least 10 for 16 threads for the unmodified Pthread version [Lee+13]. Lee et al. published a speedup of their Pipeline implementation of 6.7 for 16 threads. On our test systems we only achieved a speedup of 5 for 16 threads. No further improvements could be measured after 16 threads and beyond.

Our Pipeline implementation was slower than the reference Pthread version as it was expected. The only exception is the execution of the *native* instance with 16 threads on our Pluto system where our version outperformed the Pthread implementation.

We tried to show that a bigger instance would yield more parallelism and result in a better speedup above eight threads. Our expectation was based on the *Gustafson's law* [JM12] and on an evaluation of the spawned iteration numbers for each instance. The relevant speedup is shown in Figure 6.2.

On the left side the *simlarge* instance is shown. On the right side the *native* (Fedora disk image) instance is shown. The native instance is about four times bigger than the *simlarge* instance. The difference between the two instances can be clearly seen in the absolut time plot in Figure 6.1.

Against our expectation for this benchmark no improvement in the speedup could be seen in the bigger test instance. Clearly Figure 6.2 shows that the speedup is the same for both test instances on all our systems. The Pthread reference implementation did not yield more speedup from a bigger instance size.

Figure 6.1 shows that the total execution time even goes up after 16 threads. The speedup does not get better after 32 threads thus the plots with more threads are not shown.

We assume that this is the case for our implementation because of worker threads performing unsuccessful steal attempts. It is not clear why the Pthreaded version also suffers from this problem.

We measured the *cache misses* and *cache references* during our benchmark runs using the Linux kernel perf interface. We experienced a massive increase of the cache references after eight threads only for our Pheet Pipeline version. The Pthreads reference version has a constant number of cache references for all runs using different thread counts. The number of *cache misses* in the Pipeline execution did only grow slowly with more threads. Compared to our Pheet Pipeline version, executions of the Pthread reference had more cache misses, by a factor of four. This issue is left open for future research.

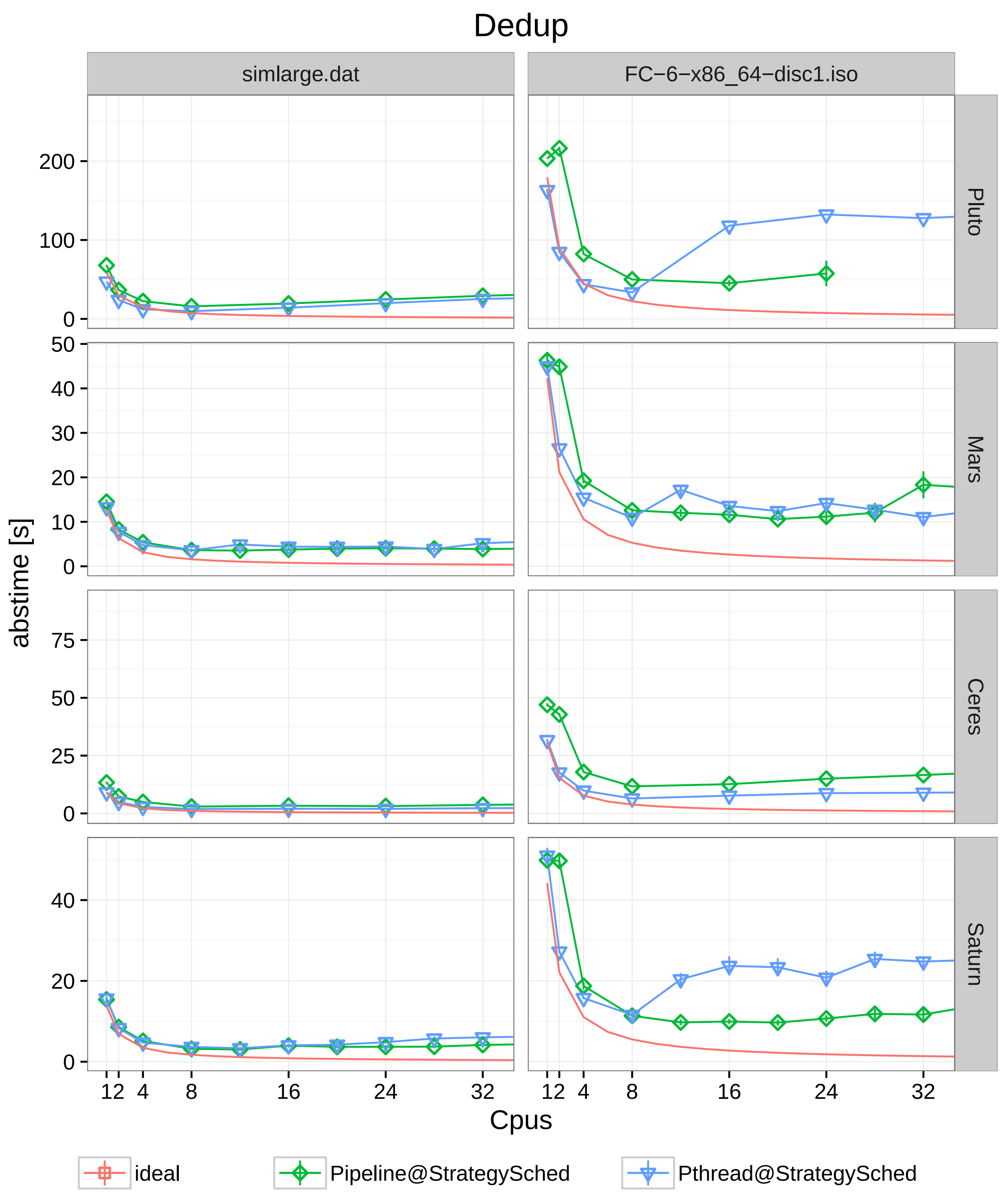


Figure 6.1: Dedup absolute benchmark times in seconds showing up to 32 threads. Ideal execution time based on the fastest single thread execution is shown in red. On the left side the *simlarge.dat* instance is shown, whereas on the right side the larger *Fedora Core 6* CD image is shown. Both instances are part of the Parsec benchmark instances [Bie11].

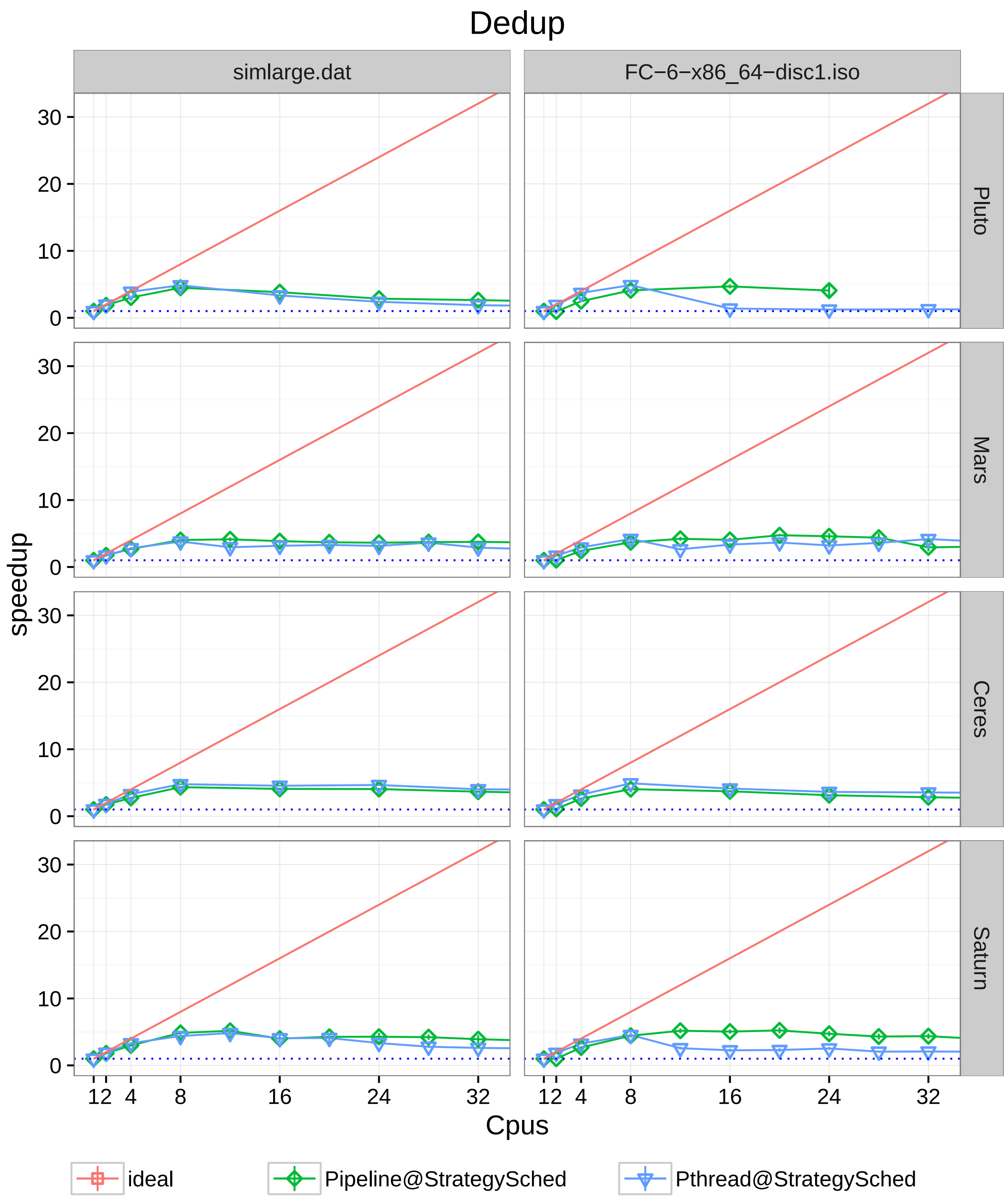


Figure 6.2: Dedup speedup relativ to a single core execution showing up to 32 threads. Ideal speedup is shown in red. On the left side the *simlarge.dat* instance is shown, whereas on the right side the larger *Fedora Core 6* CD image is shown. Both instances are part of the Parsec benchmark instances [Bie11].

6.2 PrefixSum

The results of the PrefixSum benchmark can be seen in the following figures:

- Figure 6.3 shows the absolute benchmark execution times on a linear scale. Only the thread range from 1 to 32 is displayed here to allow comparison with related work.
- Figure 6.4 shows the speedup relative to a single thread execution. Execution speeds for single thread executions differ significantly between the different implementations. Therefore this plot is only used to measure the scalability of the different implementations.

We expected speedup till 48 cores because of previously published work [Wim14].

6.2.1 Results

In the range from 1 to 4 threads our Pipeline PrefixSums implementation outperforms the reference implementation which is based on Pheet tasks. After four threads we do not see any further speedup of our Pipeline version. The Pheet tasks reference implementation still shows slight speedup till 32 cores on all benchmark systems except Ceres.

We see very different results regarding the architecture: On the Ceres and Pluto systems the Pipeline version always outperforms the Pheet task version. On Saturn and Mars the Pheet tasks version beats the Pipeline version after four (Mars) and eight (Saturn) threads.

The *sequential implementation* of PrefixSums always outperforms both other versions on Saturn, Ceres and Pluto. On Mars the Pheet task based version outperforms the sequential version at around four threads. Our Pipeline version does never outperform the sequential version.

The *speedup* of the Pipeline version ends at around four threads on all test systems. Only the Pheet tasks version has a good speedup till 32 threads on the Mars system.

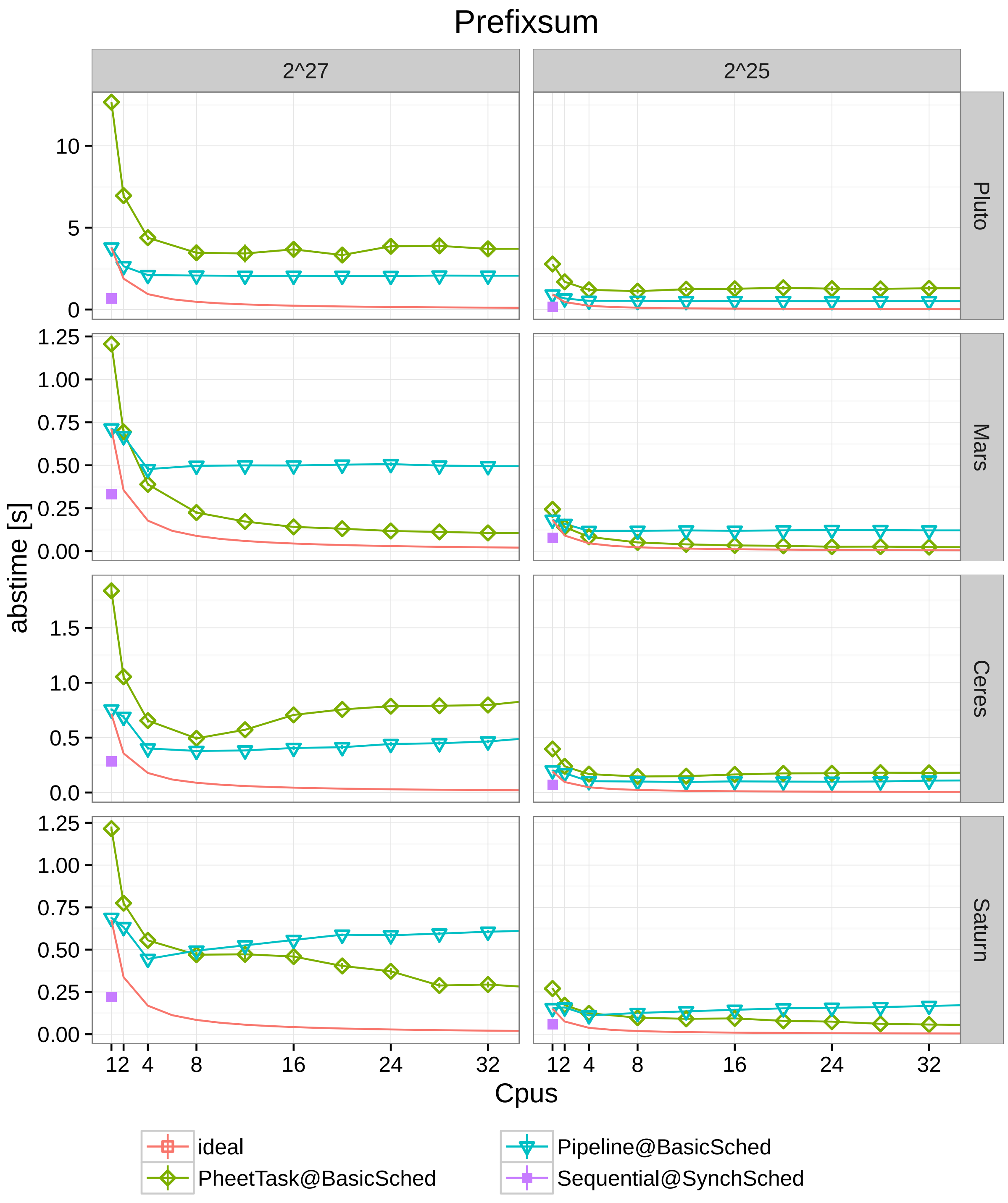


Figure 6.3: Prefixsum absolute benchmark times in seconds showing up to 32 threads.

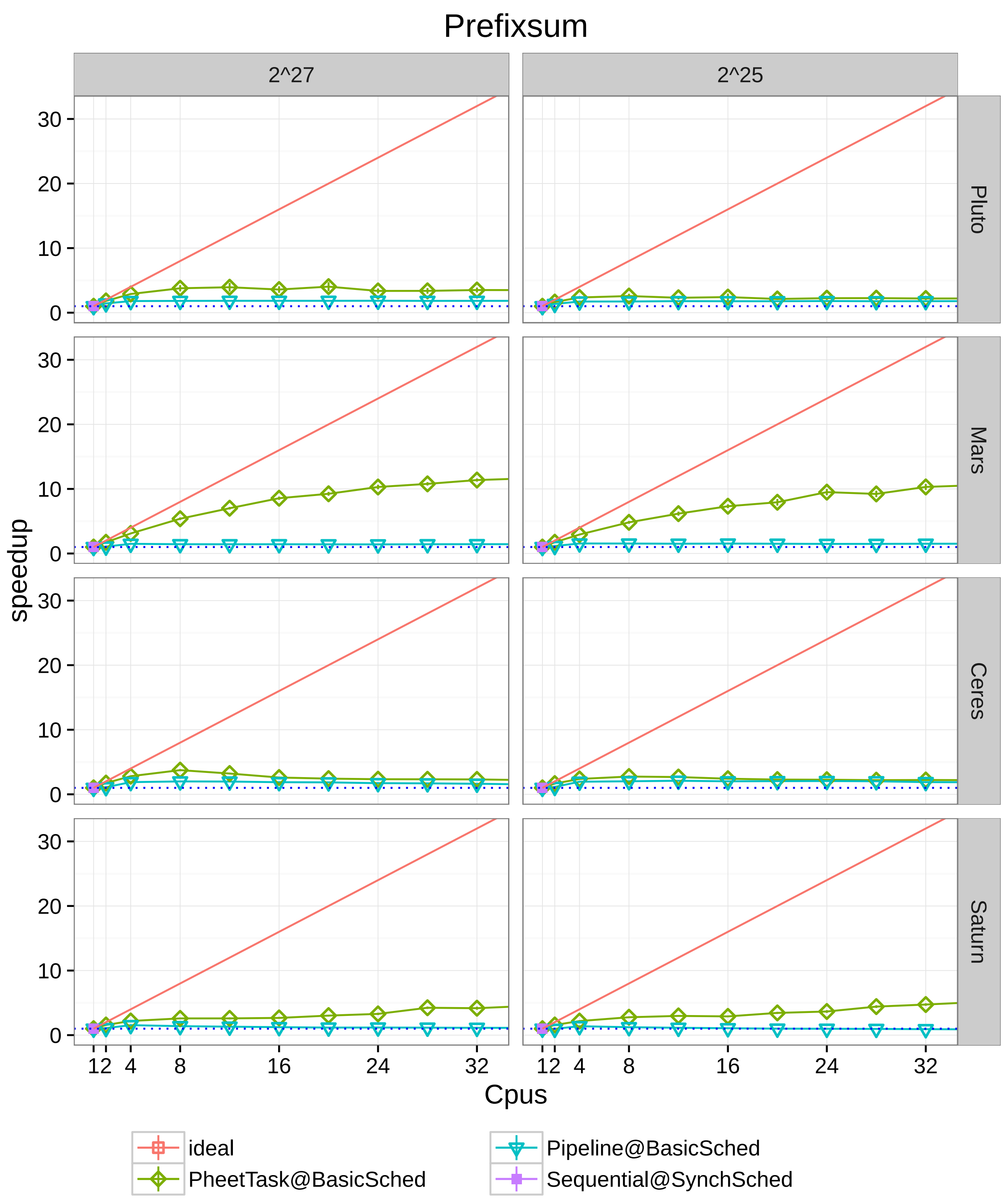


Figure 6.4: Prefixsum speedup relativ to a single core execution showing up to 32 threads.

6.3 X264

The results of the x264 benchmark can be seen in the following figures:

- Figure 6.5 Shows the absolute benchmark execution times on a linear scale. Only the thread range from 1 to 32 is displayed here to allow comparison with related work.
- Figure 6.6 shows the speedup relative to a single thread execution. Execution speeds for single thread executions differ significantly between the different implementations. Therefore this plot is only used to measure the scalability of the different implementations.
- Figure 6.7 shows the speedup relative to a single thread execution. It shows the thread range from 1 to 160. In theory the speedup should grow till 160 threads on the Mars system.

The x264 implementation is strongly optimized for a x86 architecture. We experienced multiple memory alignment problems on our Ceres system. Also the MIC command set of the Pluto system is not beneficial for x264.

Therefore this benchmark was only ported successfully to our Mars system.

6.3.1 Results

We can clearly see that our Pipeline implementation benefits from a bigger input size. This is the case because our implementation relies on independent IDR-frames to execute the encoding work in parallel. In longer video sequences more IDR-frames appear and therefore more parallelism can be exploited.

The Pthread reference version utilizes marcoblock level parallelism and also features a good speedup for the *small benchmark* instance. The speedup for this version drops at 32 threads because there are only 32 frames to encode.

On the *large benchmark instance* with 480 frames we see a good speedup of our Pipeline implementation till 32 cores. We used an IDR-frame interval of 10 frames thus resulting in 48 key frames for the bigger benchmark instance. Nevertheless we see a stall of the speedup at around 32 threads on Mars.

On this bigger instance the Pipeline version has a better scalability up to 48 threads compared to our reference Pthread version.

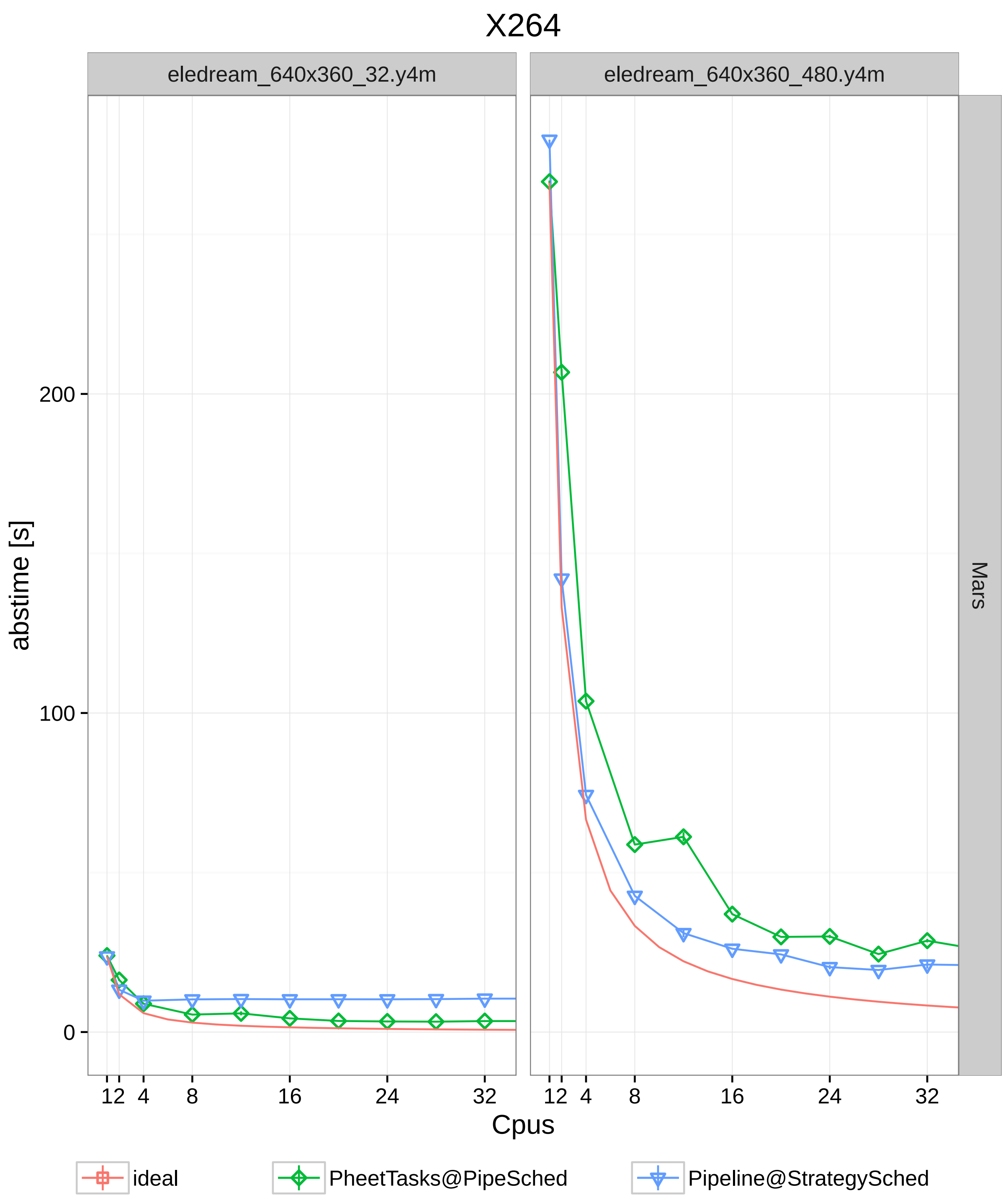


Figure 6.5: X264 absolute benchmark times in seconds showing up to 32 threads. For the left side 32 frames of the Elephants Dream [Roo+06] movie were used as input to the x264 encoder. On the right side 480 frames had been encoded.

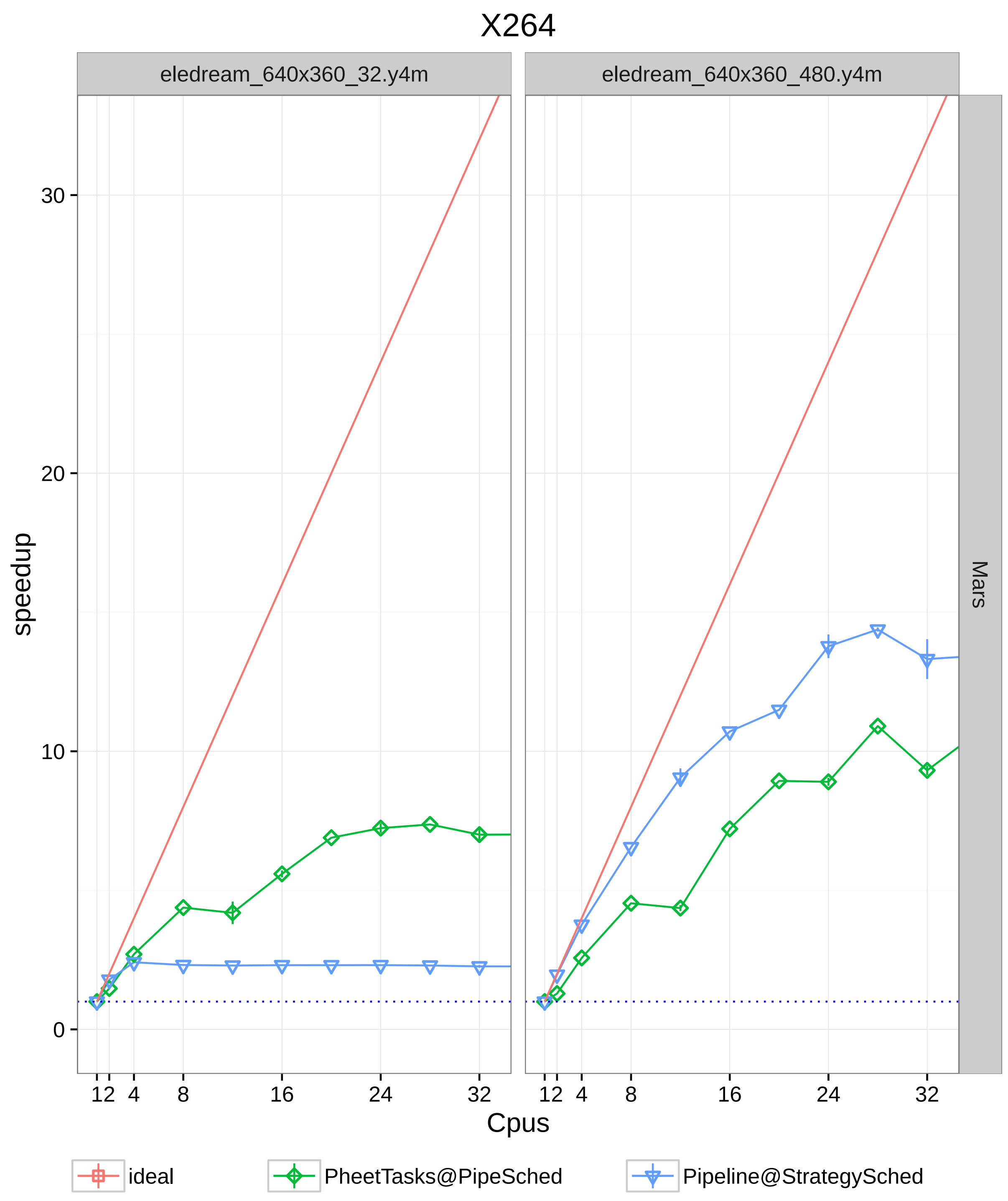


Figure 6.6: X264 speedup relativ to a single core execution showing up to 32 threads. For the left side 32 frames of the Elephants Dream [Roo+06] movie were used as input to the x264 encoder. On the right side 480 frames had been encoded.

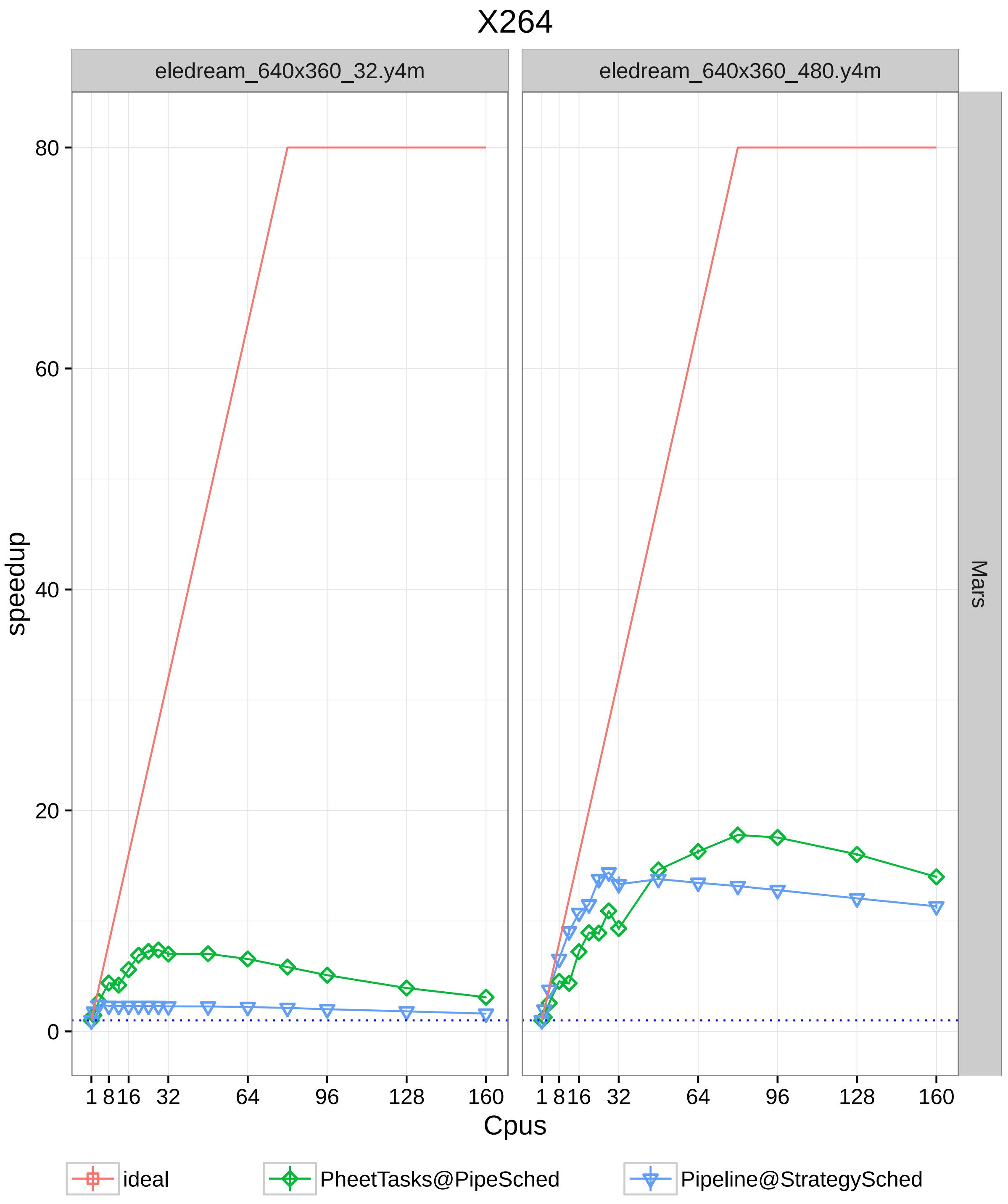


Figure 6.7: X264 speedup relativ to a single core execution showing up to 160 threads. For the left side 32 frames of the Elephants Dream [Roo+06] movie were used as input to the x264 encoder. On the right side 480 frames had been encoded.

6.4 Ferret

The results of the Ferret benchmark can be seen in the following figures:

- Figure 6.8 shows the absolute benchmark execution times on a linear scale. Thread range from 1 to 32 is displayed here to allow comparison with related work.
- Figure 6.9 shows the speedup relative to a single thread execution. Execution speeds for single thread executions differ significantly between the different implementations. Therefore this plot is only used to measure the scalability of the different implementations.
- Figure 6.10 shows the speedup relative to a single thread execution. It shows the thread range from 1 to 160.

As we expected the absolute runtime for the larger benchmark instance (*native*) is about 10× bigger compared to the small test instance. This can be seen in Figure 6.8.

We see an almost ideal speedup on all systems till four threads using the small instance and till eight cores using the large instance. After eight (simlarge) and 32 (simnative) threads the speedup slows down on all four benchmark systems.

We expected a better scalability of our Pipeline version because of work-stealing. In Figure 6.9 we see that our Pipeline version outperforms the Pthread version after four threads for the small instance and after eight threads for the big instance.

But if we look also at the absolute times in Figure 6.8 we can see that our Pipeline version is still slower than the Pthread version from 1 to 16 threads on the small instance and from 1 to 32 threads on the big instance. Only on the Saturn system our Pipeline version outperforms the Pthread version at all thread counts.

On the big instance after 32 threads hardly any speedup can be seen in Figure 6.10. Only the Ceres systems improves slightly till 200 threads for the Pipeline version.

Although the Pluto system features a very different memory architecture it also shows a slowdown at about 20 threads.

On the Saturn system we get a near ideal speedup curve till the theoretical speedup limit of 48 on this system.

We measured the *cache references* and *cache misses* on our Mars system using the Linux perf kernel interface. For our Pipeline version we experience a rapid growth of cache references after 96 threads. For the cache misses we see a constant growth which changes speed at around 32 threads. In relation to the cache references we see around 30 % cache misses. The rise of the cache references corresponds with a drop of speedup at 96 threads.

For the Pthread reference implementation we see a very constant and similar count of cache references from 4 to 160 threads. The cache misses are almost identical but slightly higher.

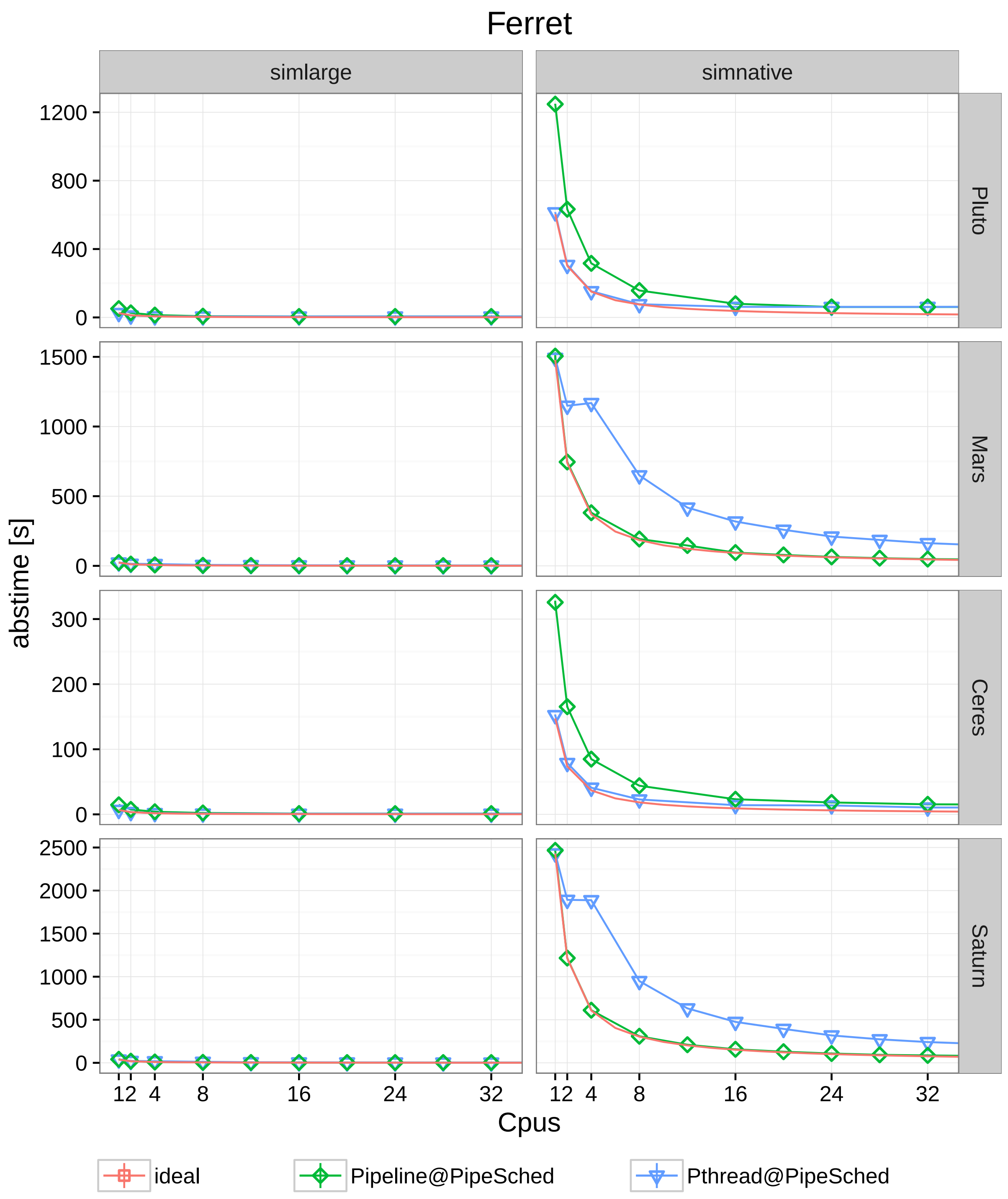


Figure 6.8: Ferret absolute benchmark times in seconds showing up to 32 threads. The left side shows the small test instance. The right size shows the big instance.

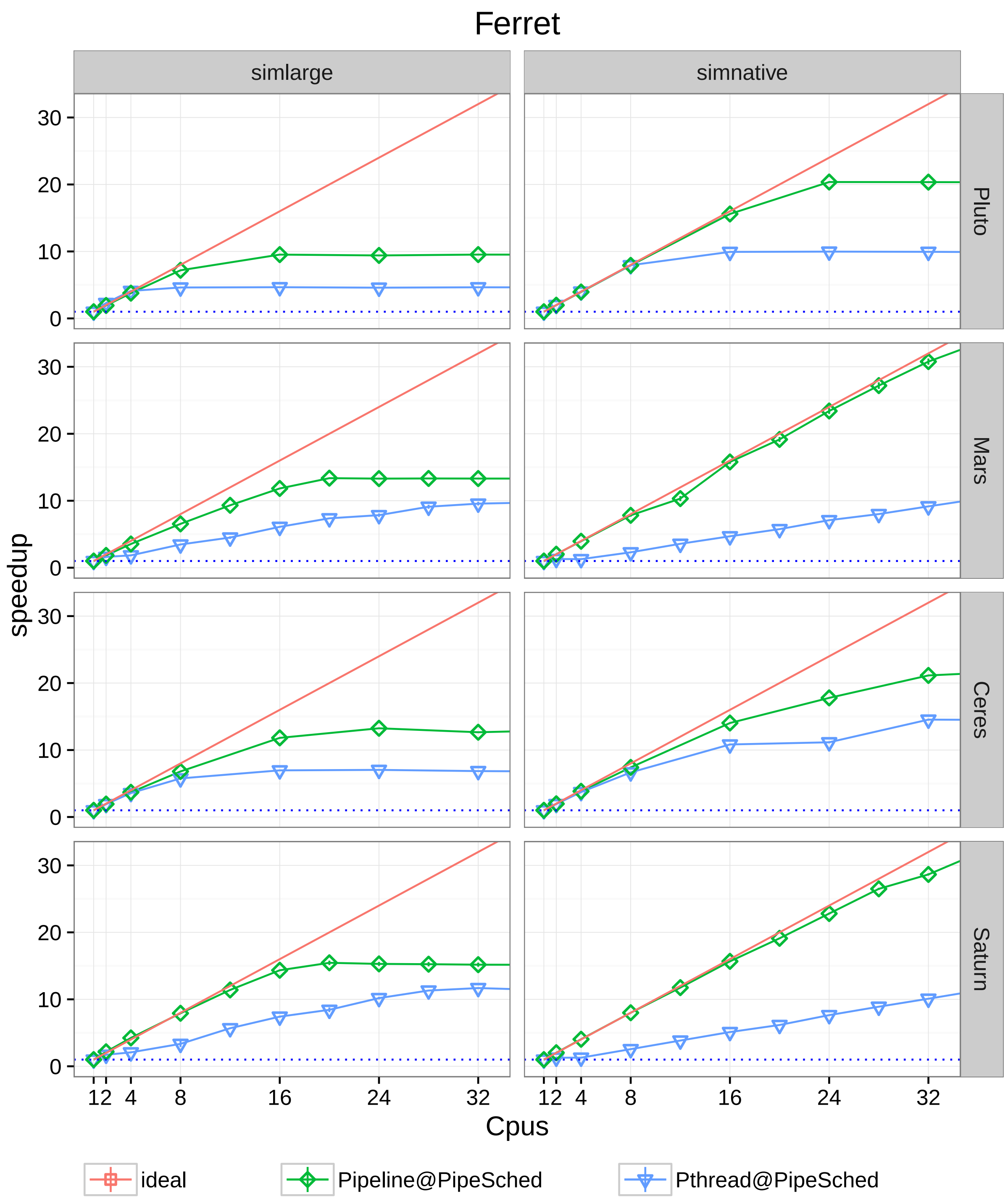


Figure 6.9: Ferret speedup relativ to a single core execution showing up to 32 threads. The left side shows the small test instance. The right size shows the big instance.

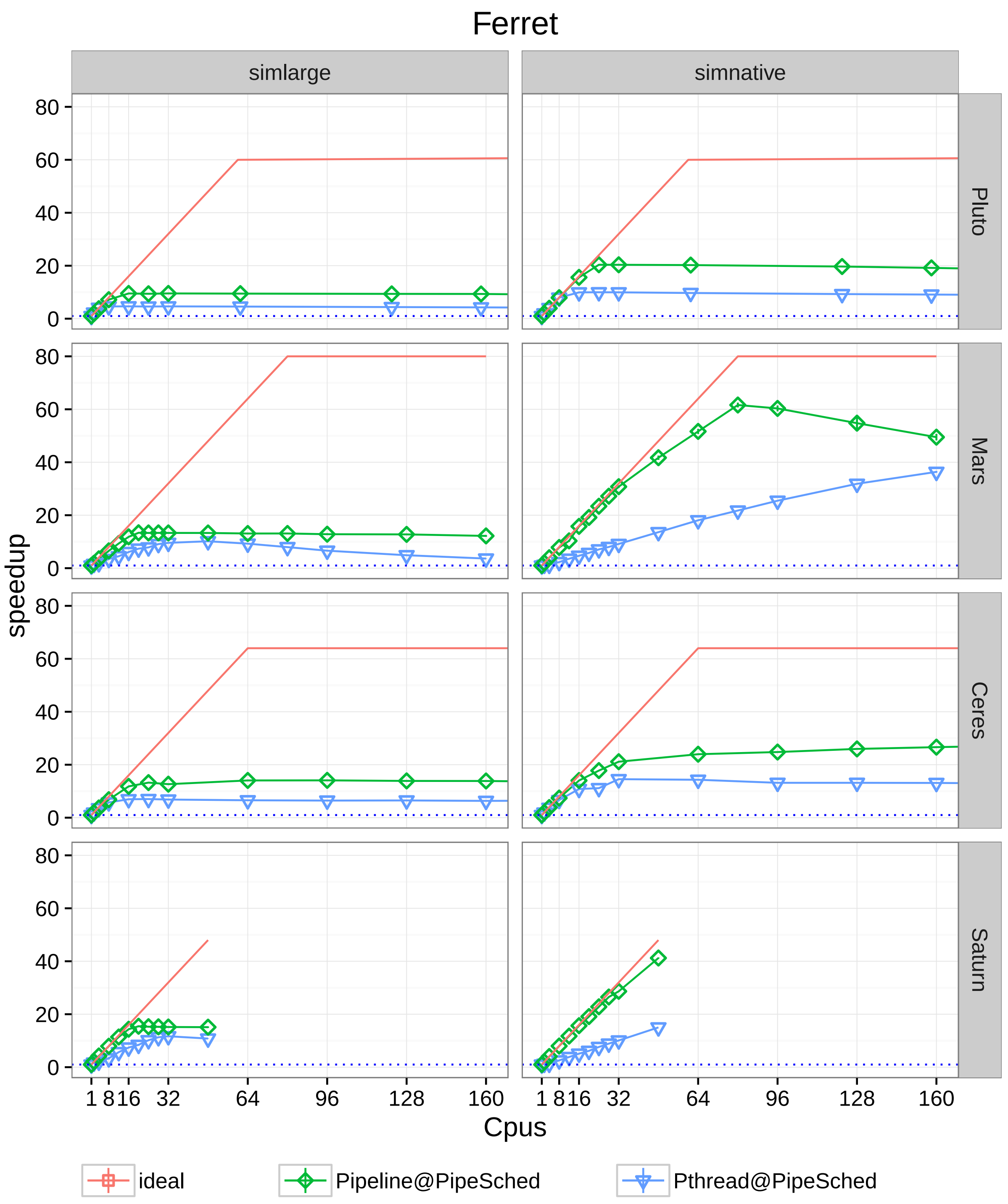


Figure 6.10: Ferret speedup relativ to a single core execution showing up to 160 threads. The left side shows the small test instance. The right size shows the big instance.

6.5 Synthetic

The synthetic Pipeline benchmark was designed to measure different characteristics. The different versions are explained in Section 4.9.

6.5.1 Wait vs Continue

This version shows the scalability of the Pipeline *wait* version in comparison to the *continue* version. Both versions have been executed with two different Pheet scheduler (BasicScheduler and StrategyScheduler see Section 1.2.2) to measure the impact of the used Pheet scheduler.

We expected that the continue version outperforms the wait version because there are no dependencies between the iterations.

The absolute execution times in Figure 6.11 show clearly that the wait version is slightly slower than the continue version. We expected the difference to be bigger. The difference can be seen more clearly in the Figure 6.12 which shows the speedup from 1 to 32 threads. After eight threads the speedup of the continue versions still grows almost ideally whereas the speedup stalls for the wait versions.

At 16 threads we see the next drop in the speedup on the *Ceres* system. This drop occurs for both the continue and the wait version. We assume that this drop on the Ceres systems occurs because each CPU has 16 cores. After 16 threads cross CPU memory contention slows down the overall progress. We see no big effect of the eight thread queues per core on the Ceres system.

The *Pluto* systems features good speedup until 128 threads. This speedup is visualized in Figure 6.13 which shows thread counts from 1 to 160. We assume that the Xeon Phi can benefit from two threads per core but not much more with more threads. In theory the Xeon Phi card has Hyper Threading with four threads per core. In addition we see a decrease in speedup (increase of total execution time) after more than 192 threads on this system.

6.5.2 Throtteling

The Pipeline wait version and the continue version were executed with a throttle limit of 512 iterations and throttling disabled.

We expected the throttled version to outperform the non-throttled version because of increased locality.

In Figure 6.11, which shows the absolute runtime, we see that throttling only effects the *wait* version. The speedup in Figure 6.12 shows that the *continue+throttle* version is slightly better than the *continue* version.

For the *Pluto* system we can see a drop in speedup for the version without throtteling at twelve threads. The throttled version only slows down very slightly. In the speedup plot in Figure 6.13 ranging from 1 to 160 threads we can see that the peak speedup of the throttled wait version is at 120 threads.

On the *Ceres* system the throttled wait version and the non-throttled version are much closer. Starting at 48 threads we see a bigger speedup difference in Figure 6.13.

6.5.3 Continue vs Pheet task vs C++ threads

This benchmark shows the overhead of the Pipeline continue version to the Pheet task version. The overhead is mainly creation and registration of the Pipeline iteration objects.

We expected a negative impact of the Pheet bookkeeping on the performance.

Due to a technical restriction of the *hwloc* library [Bro+10] on the Ceres system we can only constrain the number of threads to one or not constraint at all. For this benchmark only the requested number of Pthreads is started. The matrix rotation work is evenly distributed to these Pthreads.

In the absolute execution time results in Figure 6.14 we can see that our Pipeline implementation is faster than the Pthread implementation even for an execution with only one thread. This is the case because the Pthread version has to create 10 000 threads whereas the Pheet version only has to create 10 000 tasks and a single worker thread. Creating a task does not require a context switch in the operating system and thus is faster.

We executed the Pheet task versions with the Pheet BasicScheduler and the StrategyScheduler. In the speedup plot in Figure 6.15 the better scalability of the Pheet versions can be seen more clearly. We also see in this plot that the Pipeline version outperforms the Pheet task version. We assume that creating a PipelineIteration objects is more lightweight than creating a Pheet task object.

On the *Pluto* system we see almost linear speedup till 32 cores for the Pheet version.

6.5.4 Scheduler

To test the effect of different Pheet schedulers the Pipeline continue version and the wait version were executed using the PipelineScheduler (derived from the StrategyScheduler) and the BasicScheduler of Pheet. See details about the Pheet schedulers in Section 1.2.2.

For the absolute execution time the plot is shown in Figure 6.17. It clearly shows that the wait version suffers strongly from the BasicScheduler. This effect can be seen on all our test systems. The investigation of this scheduler bottleneck is out of the scope of this thesis.

The speedup plot in Figure 6.18 shows more clearly the differences for the *continue* version. In the range of 1 to 8 threads both scheduler perform equally well. For more threads we get different results on our benchmark systems.

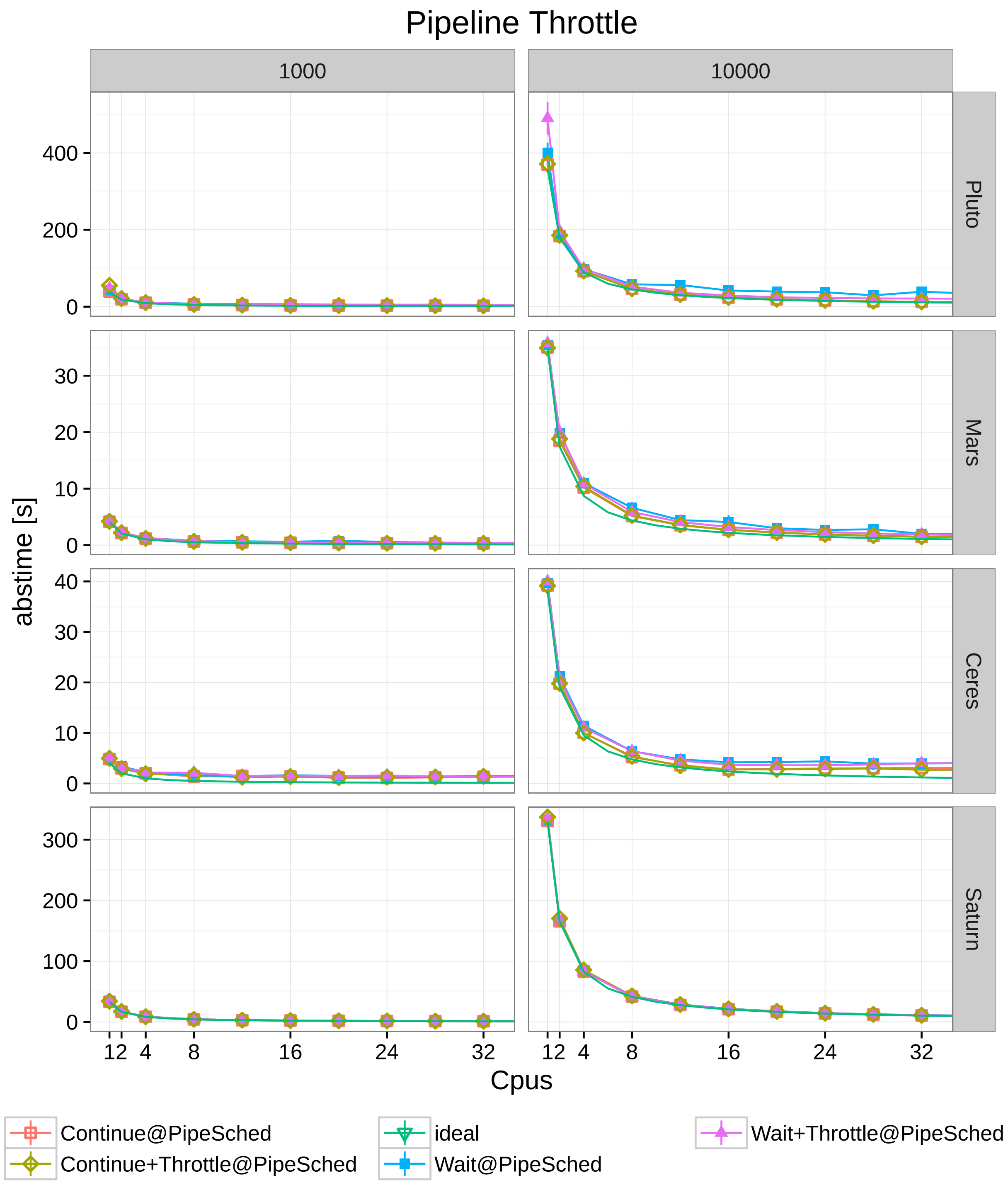


Figure 6.11: This figure shows the pipeline *continue version* vs *wait version* executed with two different throttle limits (see Section 2.1.2). The throttling was set to 512 iteration for one run and disabled for the other. This plot shows the absolute benchmark times in seconds from 1 to 32 threads.

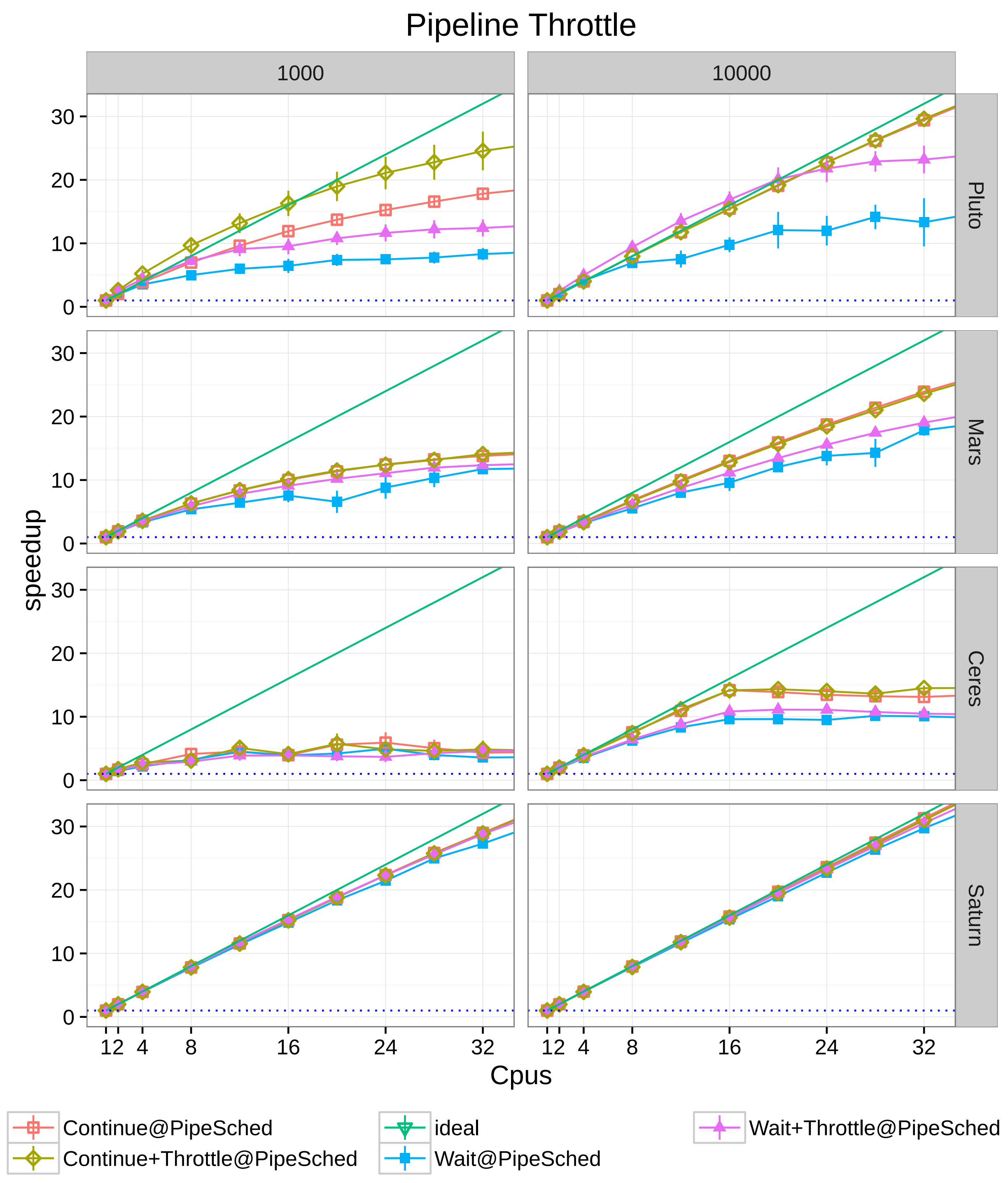


Figure 6.12: This figure shows the Pipeline *continue version* vs *wait version* executed with two different throttle limits. The throttling was set to 512 iteration for one run and disabled for the other. This plot shows the speedup relativ to a single core execution showing up to 32 threads.

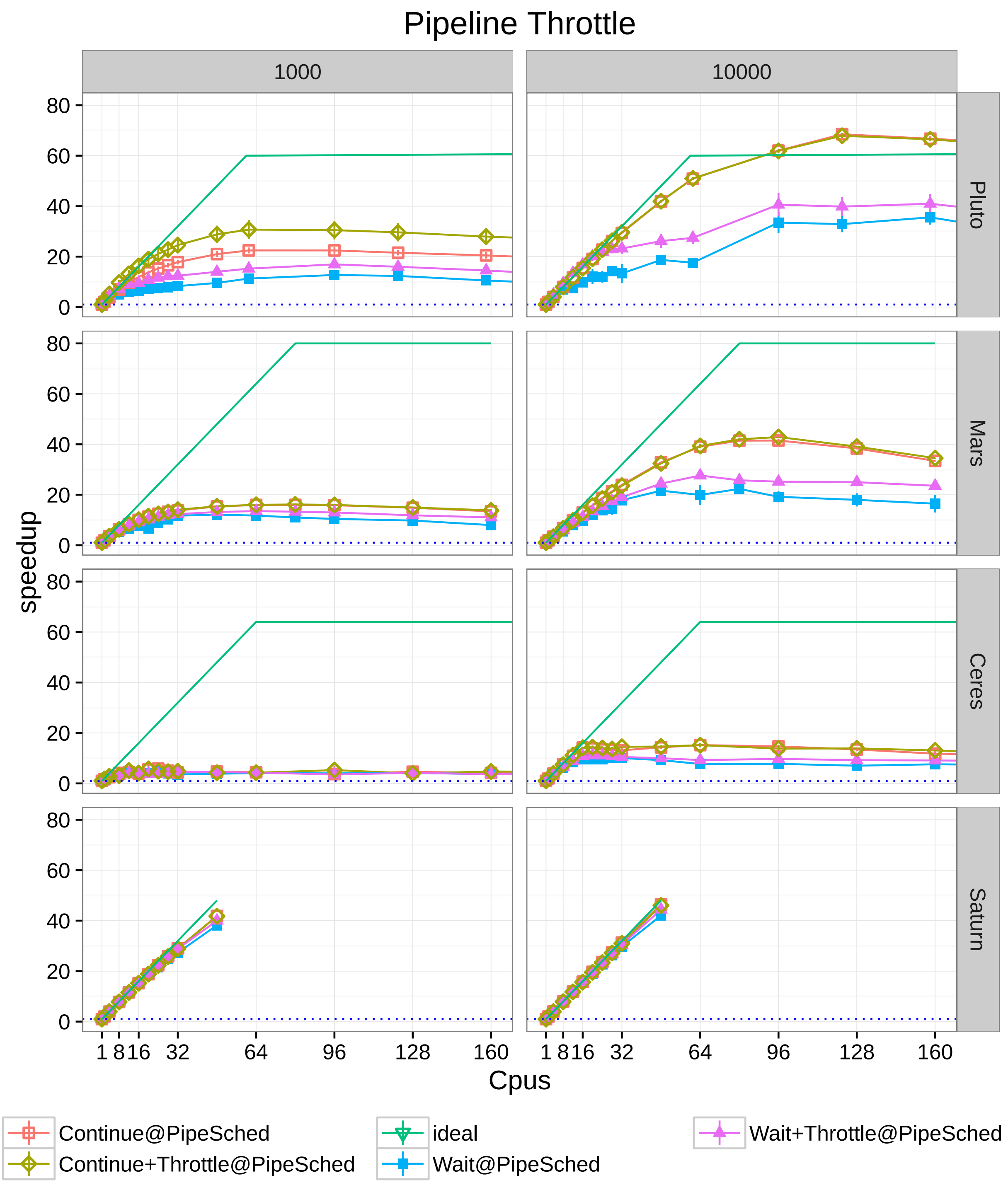


Figure 6.13: This figure shows the Pipeline *continue version* vs *wait version* executed with two different throttle limits. The throttling was set to 512 iteration and disabled in the run. Pipeline speedup relativ to a single core execution showing up to 160 threads.

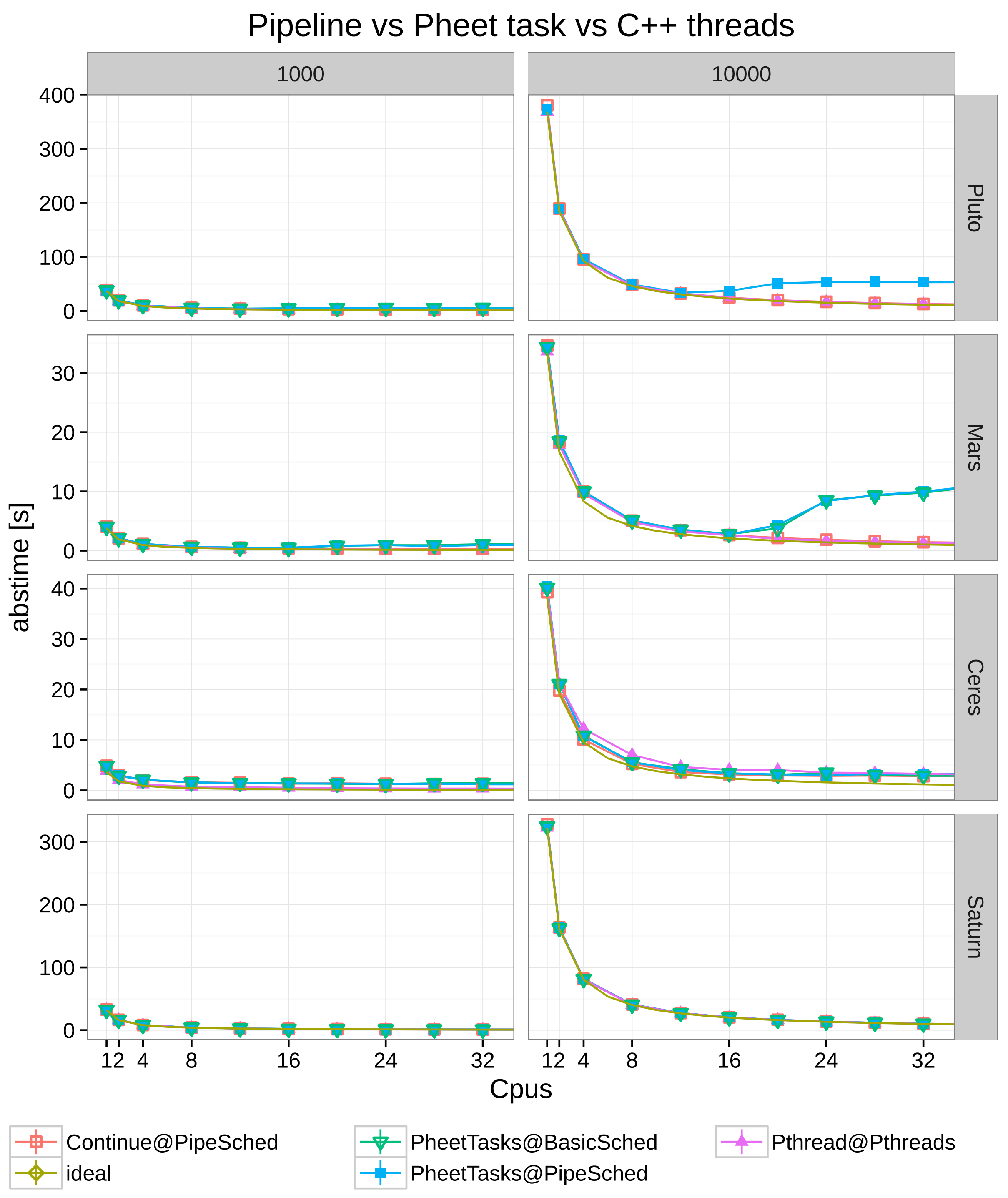


Figure 6.14: This benchmark compares our Pipeline implementation with Pheet tasks and with Pthreads (C++11 threads). It shows the absolute times in seconds from 1 to 32 threads. The Pheet Pipeline scheduler had some stability issues on our Saturn system and was therefore replaced by the BasicScheduler on this system.

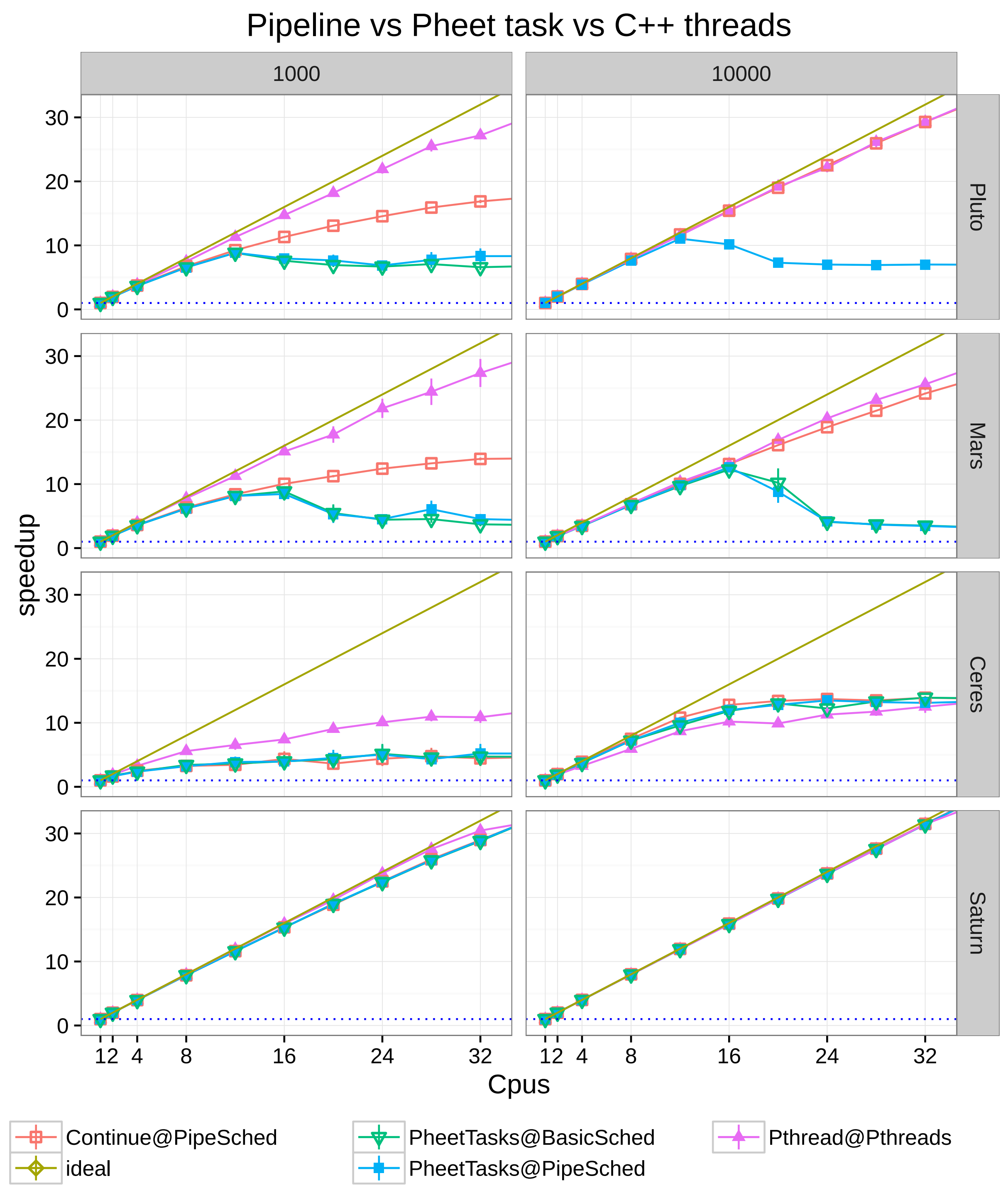


Figure 6.15: This benchmark compares our Pipeline implementation with Pheet tasks and with Pthreads. It shows the speedup of 1 to 32 threads relative to a single core execution. The Pheet Pipeline scheduler had some stability issues on our Saturn system and was therefore replaced by the BasicScheduler on this system.

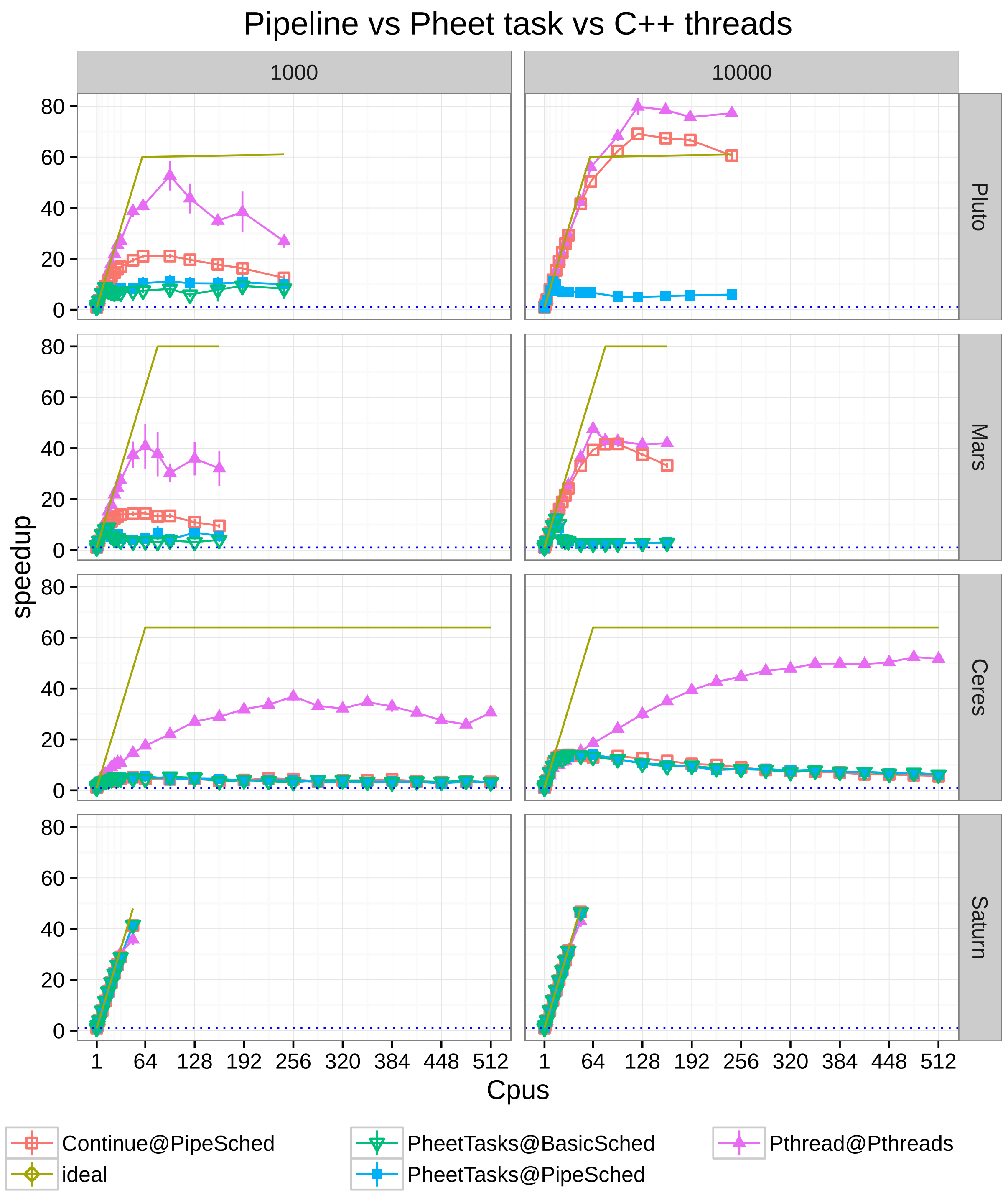


Figure 6.16: This benchmark compares our Pipeline implementation with Pheet tasks and with Pthreads. It shows the speedup up to 512 threads relative to a single core execution. The Pheet Pipeline scheduler had some stability issues on our Saturn system and was therefore replaced by the BasicScheduler on this system.

On the *Pluto* system which has a working *hwloc* library [Bro+10] we see that the PipelineScheduler (StrategyScheduler) outperforms the simpler BasicScheduler. The hwloc library pins Pheet worker threads to specific cores. This increases the locality because the worker can not be moved by the operating system between two tasks. On this system the speedups for the *continue* version are quite close for all thread counts.

On the *Ceres* system the BasicScheduler outperforms the PipelineScheduler. On the Ceres system the pinning of working threads to Sparc cores using the *hwloc* [Bro+10] library did not work as expected. The reason is that threads on Sparc systems can not be arbitrarily pinned on Sparc cores.

On Ceres we see a drop of speedup for both schedulers at the continue version at around 16 threads. We assume that the reason is that there are 16 cores on each Sun CPU and thus a greater thread count creates cross CPU traffic.

In the *small* benchmark instances with only 1000 iterations the overhead for task switching and iteration creation does not allow any speedup after four cores on the Ceres system. On the Pluto system we see speedup even after four threads but it is far away from ideal speedup.

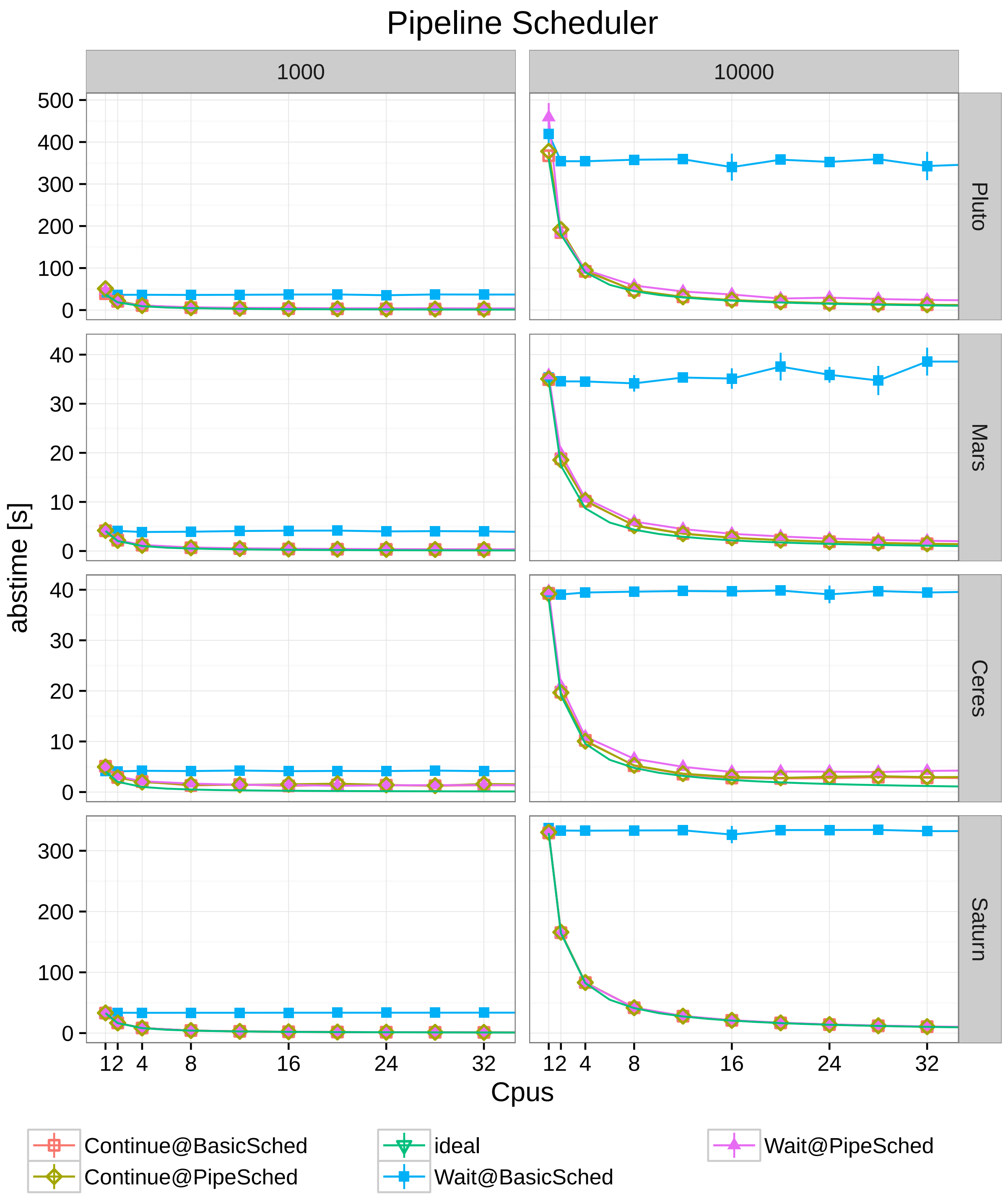


Figure 6.17: Measures the Pipeline wait and continue version with two different Pheet schedulers, namely the PipelineScheduler (StrategyScheduler) and the BasicScheduler. This test should give some insights about the scalability of different Pipeline scheduler. This plot shows the absolute execution times for executions from 1 to 32 threads.

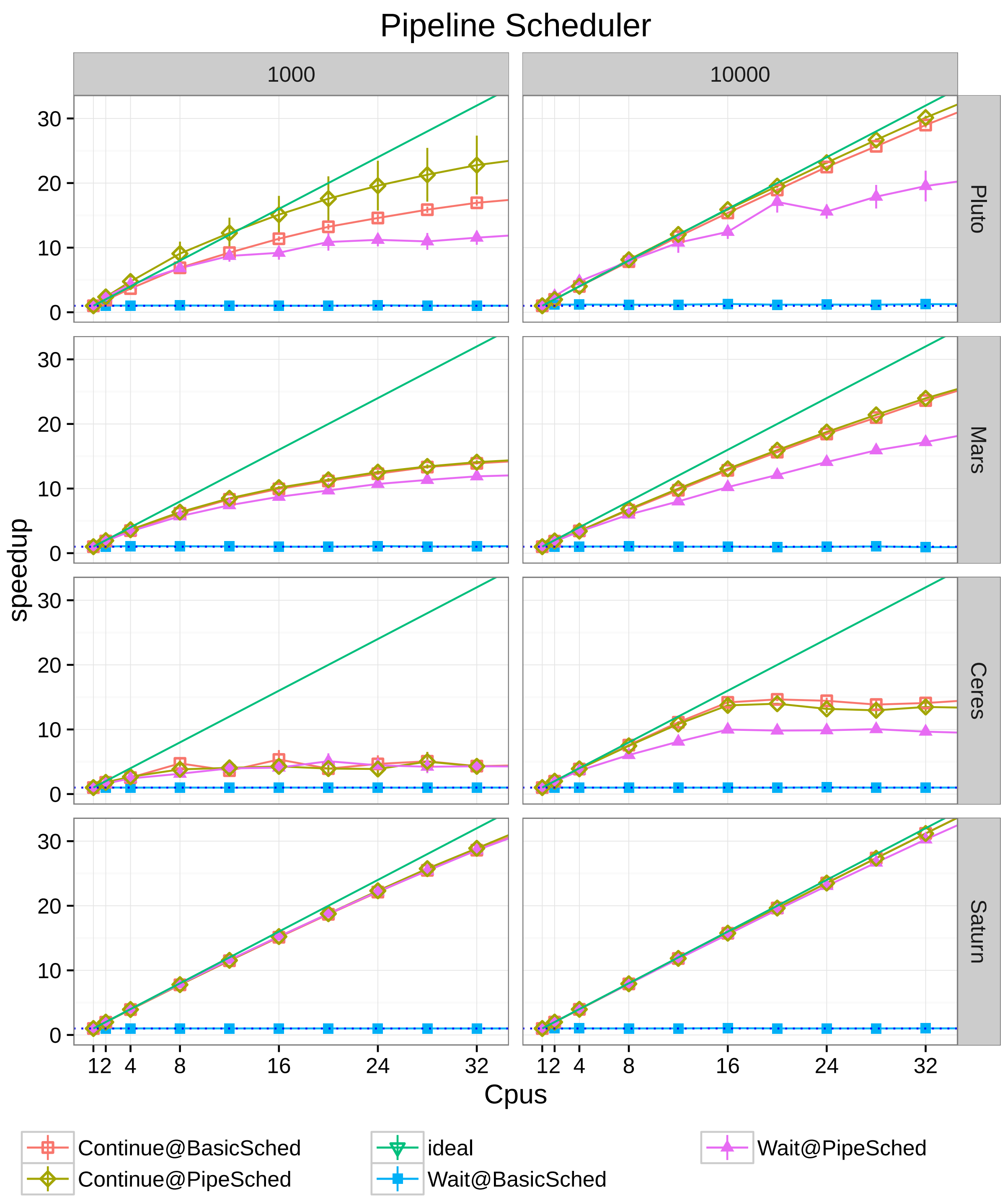


Figure 6.18: Measures the Pipeline wait and continue version with two different Pheet schedulers, namely the PipelineScheduler (StrategyScheduler) and the BasicScheduler. This test should give some insights about the scalability of different Pipeline scheduler. It shows the *speedup* relativ to a single core execution ranging from 1 to 32 threads.

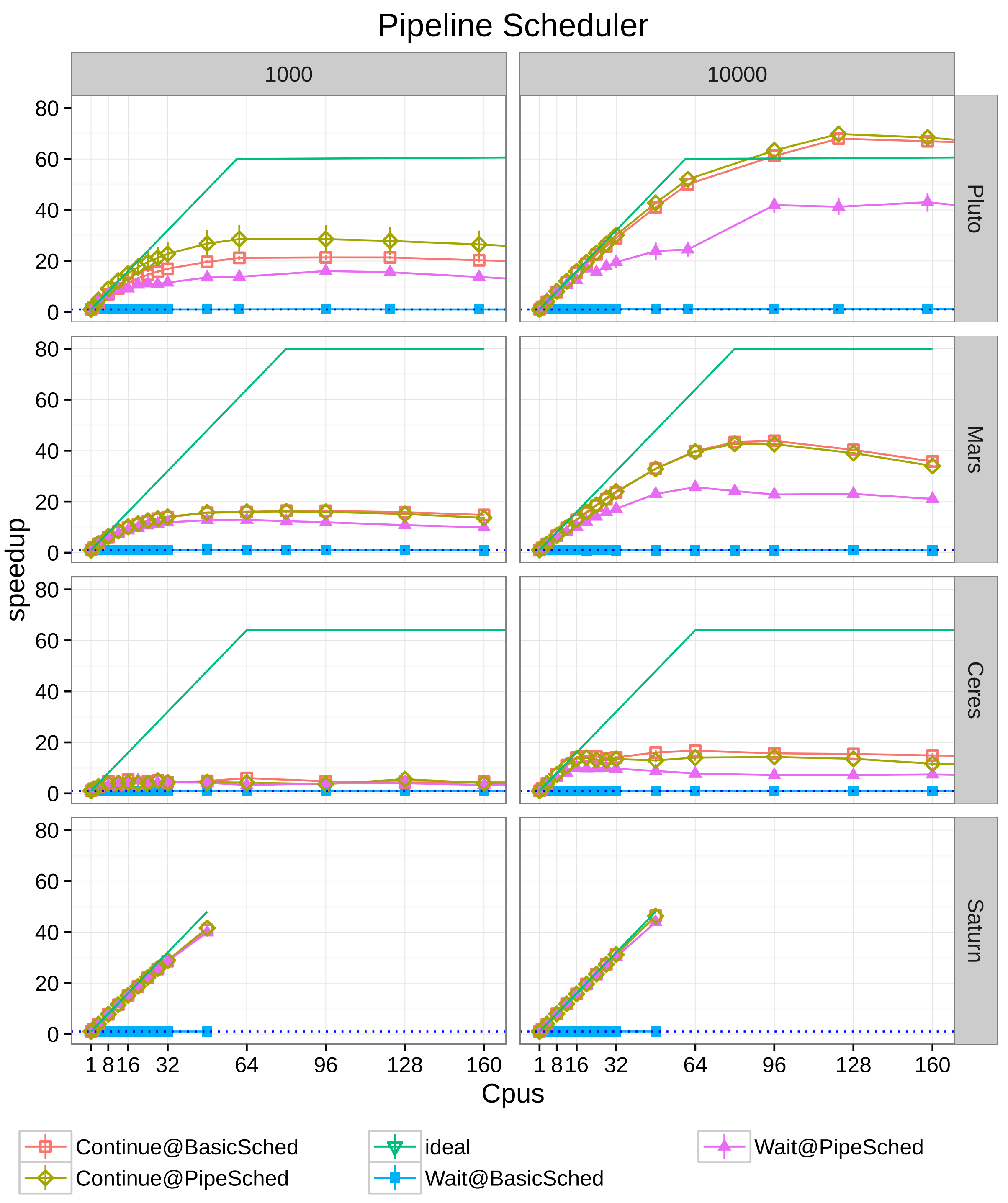


Figure 6.19: Measures the Pipeline wait and continue version with two different Pheet schedulers, namely the PipelineScheduler (StrategyScheduler) and the BasicScheduler. This test should give some insights about the scalability of different Pipeline scheduler. It shows the *speedup* relativ to a single core execution ranging from 1 to 160 threads.

CHAPTER 7

Summary

7.1 Pipelines in Pheet

We created a competitive implementation of the Parallel Pipeline pattern in the Pheet framework. The Pipeline pattern is very relevant for parallel programs because in this pattern the synchronization is performed by the framework and not by the user. This leads to less errors and better maintainable programs.

Our implementation was based on analyzing existing *state of the art* frameworks which support Pipeline construction. By analyzing the strong points and the weaknesses of other implementations we were able to define the main design goals for our implementation.

We defined the following points as our main design goals for our Pheet Pipeline implementation:

- Our benchmark systems are multi chip NUMA systems. All our systems are able to execute up to 60 threads in parallel. Some of our benchmark systems can even execute more than 160 threads concurrently. Therefore we optimized our implementation to be *lock-free* and scale well on multi CPU NUMA systems.

  We realized *lock-freedom* by strongly relying on the C++11 atomic data type. When used correctly it has much less overhead and produces less contention compared to a lock data structure. We also avoided creating centralized data structures to reduce cross chip memory accesses.

- *Short Pipeline definitions* make it easier to port legacy applications to our Pipeline interface. Shorter code also leads to less programming errors and increases the readability. Compared to Nabbit, definitions in our interface only require 10 % of code. Piper Pipeline definitions are the same size compared to our interface, but require hand compilation.

We rely on C++ Lambda functions to create unnamed local functions for every stage, which can access variables of the containing scope. The Pheet task scheduler requires to formulate every Pipeline stage as an independent task by design. Piper does not have this restriction because their scheduler can handle blocking tasks, while the Pheet scheduler does not support this.

- Some of our benchmarks require support for *non-linear* Pipelines. Non-linear Pipelines are more flexible and allow stages to produce multiple sub iterations for a single input iteration. Not every stage has to obey the same static schema for every iteration.

  The support for non-linear Pipelines leads to challenges regarding the unique identification of a Pipeline iteration. We solved this by using a hierarchical id for each iteration.

  Piper and Intel TBB do not support non-linear Pipelines. The DAG scheduler Nabbit and the new Intel flow interface allow specification of non-linear Pipelines.

- Most other Pipeline implementations support passing results from stage to stage for iterations. But most problem instances also require *passing results* from iteration to iteration, which is not supported by most other implementations. Providing a framework method to pass results from *iteration to iteration* reduces the chance of possible bugs and reduces the code complexity of the user code.

  We implemented this feature in our Pipeline and provide fast access to the results of adjacent iterations. The passing of results is optional, unless it is used no additional memory or computation time is used for storing results.

  Cilk and Intel TBB do not support result passing from iteration to iteration. Nabbit supports access to results of dependent nodes.

- Analyzing performance parameters in parallel programs in challenging. Most methods require special privileges or slow down the execution on many core systems.

  We used Pheet performance counters based on Hyperobject Reducers which scale very well for multi CPU systems. The disadvantage of these Pheet performance counters is that the results are only available after the Pipeline execution has terminated.

  To our knowledge none of the other implementations support performance counters based on Reducer Hyperobjects.

- Limiting the number of concurrently running iterations is an important issue for Pipeline programs to limit the used memory space.

  We implemented *throttling* of iterations in our scheduler and allow the user to dynamically change the throttle limit. We consider dynamic adaptions important in multiprogrammed environments.

  Piper and Intel TBB support static throttle limits. Nabbit does not support throttling.

7.2 Future Research

The following topics are open for future research:

We strongly rely on the *Lambda construct* and model every Pipeline stage as an own Lambda function expression. This is necessary because the Pheet scheduler can not handle blocking tasks correctly. A blocking task blocks the whole worker thread. Changing this would require deep changes in the Pheet scheduler which were beyond the scope of this thesis. With these changes the Pipeline declarations could be done in a single Lambda function which would increase readability.

Another open issue is that our implementation only allows to specify a *single data type for result values* for all stages. As a workaround we suggested using the C++ union data type, but this leads to none type safe code. Future work could investigate template methods to specify result types per stage in a type safe manner.

The usage of our type safe null values called *null traits* pose an additional burden to the user. Compiler error messages are very unclear in case this essential definition is omitted by the user.

During our benchmark runs we saw a huge increase of *cache references* for higher thread counts. This only affected the Pheet scheduler and not our Pthread reference implementations. Future work could investigate this issue.

We did not utilize the *strategy* support of our Pheet scheduler. Priortizing ealier iterations may increase locality and reduce the memory usage be reducing the number of concurrently active iterations.

7.3 Benchmark Results

We benchmarked our implementation using four real world benchmarks and one synthetic benchmark. No work comparing the Pheet scheduler with other task scheduler has been published yet, therefore we did only compare a Pipelined version of each benchmark with a reference Pthread version. All Pthread versions had been carefully optimized by the creators of these benchmarks. We did not expect to beat them in speed, but we showed that the code complexity is significantly less using our Pipeline interface.

Benchmark results for the *Dedup* file deduplication benchmark showed acceptable results. On three of our test systems our Pipeline implementation even outperformed the Pthreads reference implementation. Both implementations did not show any further speedup after eight threads. The best speedup reached at eight threads was only about four, which was lower than expected.

For the *PrefixSum* benchmark we saw very different results on our test systems. On two test systems our Pipeline version achieved a better speedup than the reference Pheet task based version. On nearly all of our test systems the speedup was limited to values around three. Only on the Mars system we saw steady speedup till 32 threads but only for the reference Pheet tasks version.

The popular H.264 video encoder *x264* was benchmarked only on our Mars system. Porting this benchmarks to different systems is a great challenge because of heavy use of low level code and assembly optimizations. We experienced a steady speedup for our Pipeline version which even outperformed the reference Pthread version but only on the bigger input instance. This was the case because in our Pipeline version parallelism is achieved using the independence of IDR-Frames whereas the Pthread reference relied on fine grained parallelism from macroblocks. The speedup for the Pipeline version was almost ideally till 24 threads.

The similarity search *Ferret* benchmark also showed different results on different test systems. On the Mars and on the Saturn test system our Pipeline version always outperformed the reference Pthread version. On our other test systems which feature different architectures the Pthread version was faster. On most of our test systems we saw almost linear speedup till 32-48 threads (regarding the test system).

We created a *synthetic benchmark* to measure the overhead of our Pipeline implementation. We performed matrix rotations inside the stages to simulate work. We compared the code for *wait* with the *continue* code path to investigate the synchronization overhead. We also investigated the effects of our *throttle* function to the overall performance. At the end we compared the performance with a Pheet task reference version and with a C++ threads (Pthreads) reference version. In conclusion we saw very good scalability of all versions. The continue version features ideal speedup like the reference C++11 threads version did. We achieved a *speedup peak* for the *wait version* (using throttling) of 40 running with 96 threads and for the version without throttling of 32 running with the same thread count on the Pluto system. We saw similar results on all benchmark systems.

To summarize we have shown that our Pipeline implementation theoretically can achieve a high speedup even for high thread executions. Most of the real world test cases did not yield a good speedup, but neither did the Pthreads reference versions. We assume that compilers and modern CPUs add many optimizations which make the serial execution very fast. This results in less speedup for executions using many cores. We can back this assumption by executing the benchmarks without compiler optimizations. These runs yield a better speedup curve (but longer runtime) on all benchmark systems.

List of Figures

List of Code Listings

Bibliography

[ABP98] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. "Thread Scheduling for Multiprogrammed Multiprocessors". In: *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures.* SPAA '98. Puerto Vallarta, Mexico: ACM, 1998, pp. 119–129. ISBN: 0-89791-989-0. URL: `http://doi.acm.org/10.1145/277651.277678`.

[Aga+07] Shivali Agarwal et al. "Deadlock-free Scheduling of X10 Computations with Bounded Resources". In: *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures.* SPAA '07. San Diego, California, USA: ACM, 2007, pp. 229–240. ISBN: 978-1-59593-667-7. URL: `http://doi.acm.org/10.1145/1248377.1248416`.

[Ale07] Michael J. Voss Alexey Kukanov. *The Foundations for Scalable Multi-Core Software in Intel ® Threading Building Blocks Methodology, Tools, and Techniques to Parallelize Large-Scale Applications: A Case Study Future-Proof Data Parallel Algorithms and Software on Intel ® Multi-Core Archite.* 2007.

[ALS10] K. Agrawal, C.E. Leiserson, and J. Sukha. "Executing task graphs using work-stealing". In: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS).* Apr. 2010, pp. 1–12.

[Ama14] S. Amanda. *Building a Native Application for Intel® Xeon Phi^{TM} Coprocessors.* Tech. rep. Intel, Mar. 2014. URL: `https://software.intel.com/en-us/articles/building-a-native-application-for-intel-xeon-phi-coprocessors` (visited on 10/10/2015).

[Amd67] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference.* AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485. URL: `http://doi.acm.org/10.1145/1465482.1465560`.

[Bie11] Christian Bienia. "Benchmarking Modern Multiprocessors". PhD thesis. Princeton University, Jan. 2011.

[BL12] Christian Bienia and Kai Li. “Characteristics of Workloads Using the Pipeline Programming Model”. In: *Proceedings of the 2010 International Conference on Computer Architecture*. ISCA’10. Saint-Malo, France: Springer-Verlag, 2012, pp. 161–171. ISBN: 978-3-642-24321-9. URL: `http://dx.doi.org/10.1007/978-3-642-24322-6_14`.

[BL93] Robert D. Blumofe and Charles E. Leiserson. “Space-efficient Scheduling of Multithreaded Computations”. In: *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*. STOC ’93. San Diego, California, USA: ACM, 1993, pp. 362–371. ISBN: 0-89791-591-7. URL: `http://doi.acm.org/10.1145/167088.167196`.

[BL99] Robert D. Blumofe and Charles E. Leiserson. “Scheduling Multithreaded Computations by Work Stealing”. In: *J. ACM* 46.5 (Sept. 1999), pp. 720–748. ISSN: 0004-5411. URL: `http://doi.acm.org/10.1145/324133.324234`.

[Ble89] Guy E Blelloch. “Scans as primitive parallel operations”. In: *Computers, IEEE Transactions on* 38.11 (1989), pp. 1526–1538.

[Boc+09] Robert L. Bocchino Jr. et al. “Parallel Programming Must Be Deterministic by Default”. In: *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*. HotPar’09. Berkeley, California: USENIX Association, 2009, pp. 4–4. URL: `http://dl.acm.org/citation.cfm?id=1855591.1855595`.

[BR97] Guy E. Blelloch and Margaret Reid-Miller. “Pipelining with Futures”. In: *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’97. Newport, Rhode Island, USA: ACM, 1997, pp. 249–259. ISBN: 0-89791-890-8. URL: `http://doi.acm.org/10.1145/258492.258517`.

[Bro+10] F. Broquedis et al. “hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications”. In: *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*. Feb. 2010, pp. 180–186.

[CJ11] Chi Ching Chi and Ben Juurlink. “A QHD-capable Parallel H.264 Decoder”. In: *Proceedings of the International Conference on Supercomputing*. ICS ’11. Tucson, Arizona, USA: ACM, 2011, pp. 317–326. ISBN: 978-1-4503-0102-2. URL: `http://doi.acm.org/10.1145/1995896.1995945`.

[Cor+09] T.H. Cormen et al. *Introduction to Algorithms*. 3rd ed. Cambridge, MA: The MIT Press, 2009. ISBN: 0262033844.

[Dok+12] Jiri Dokulil et al. “Efficient Hybrid Execution of C++ Applications using Intel(R) Xeon Phi(TM) Coprocessor”. In: *CoRR* abs/1211.5530 (2012). URL: `http://arxiv.org/abs/1211.5530`.

[Fri+09] Matteo Frigo et al. “Reducers and Other Cilk++ Hyperobjects”. In: *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA ’09. Calgary, AB, Canada: ACM, 2009, pp. 79–90. ISBN: 978-1-60558-606-9. URL: `http://doi.acm.org/10.1145/1583991.1584017`.

[Gam+95] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[GCC] GCC. *GCC5 Offloading Support*. URL: `https://gcc.gnu.org/wiki/Offloading` (visited on 05/01/2016).

[GTA06] Michael I. Gordon, William Thies, and Saman Amarasinghe. “Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs”. In: *SIGPLAN Not.* 41.11 (Oct. 2006), pp. 151–162. ISSN: 0362-1340. URL: `http://doi.acm.org/10.1145/1168918.1168877`.

[Guo+09] Yi Guo et al. “Work-first and Help-first Scheduling Policies for Async-finish Task Parallelism”. In: *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. IPDPS ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12. ISBN: 978-1-4244-3751-1. URL: `http://dx.doi.org/10.1109/IPDPS.2009.5161079`.

[Gus88] John L. Gustafson. “Reevaluating Amdahl’s Law”. In: *Commun. ACM* 31.5 (May 1988), pp. 532–533. ISSN: 0001-0782. URL: `http://doi.acm.org/10.1145/42411.42415`.

[Gut] Project Gutenberg. URL: `http://www.gutenberg.org/wiki/Main_Page` (visited on 01/10/2016).

[Hal+14] Halpern et al. *Task Region N3832*. Tech. rep. N3832. Programming Language C++ Proposal, Jan. 2014.

[Hal+15] Halpern et al. *Task Block N4411*. Tech. rep. N4411. Programming Language C++ Proposal, Apr. 2015.

[Inta] Intel. *Intel® Xeon® Processor E7-8850*. URL: `http://ark.intel.com/products/53575/` (visited on 05/01/2016).

[Intb] Intel. *Piper: Experimental Language Support for Pipeline Parallelism in Intel® Cilk^{TM} Plus*. URL: `https://www.cilkplus.org/piper-experimental-language-support-pipeline-parallelism-intel-cilk-plus` (visited on 05/10/2016).

[Int11] Intel. *Intel TBB Documentation*. Tech. rep. Intel Developer Zone, 2011. URL: `https://software.intel.com/de-de/node/517344` (visited on 09/10/2015).

[Int13] Intel. *Intel® Cilk^{TM} Plus Language Extension Specification Version 1.2.* Tech. rep. Intel, Sept. 2013. URL: https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm (visited on 09/20/2015).

[Int14] Intel. *Intel and Third Party Tools and Libraries available with support for Intel® Xeon Phi^{TM} Coprocessor.* Oct. 2014. URL: https://software.intel.com/en-us/articles/intel-and-third-party-tools-and-libraries-available-with-support-for-intelr-xeon-phitm (visited on 10/10/2015).

[Int15] Intel. *Intel TBB Documentation.* Tech. rep. Intel, 2015. URL: https://www.threadingbuildingblocks.org/docs/help/reference/algorithms/parallel_pipeline_func.htm (visited on 09/10/2015).

[ISO12] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++.* Geneva, Switzerland: International Organization for Standardization, Feb. 28, 2012, 1338 (est.) URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.

[JF10] Jaakko Järvi and John Freeman. "C++ Lambda Expressions and Closures". In: *Sci. Comput. Program.* 75.9 (Sept. 2010), pp. 762–772. ISSN: 0167-6423.

[JM12] B.H.H. Juurlink and C. H. Meenderinck. "Amdahl's Law for Predicting the Future of Multicores Considered Harmful". In: *SIGARCH Comput. Archit. News* 40.2 (May 2012), pp. 1–9. ISSN: 0163-5964. URL: http://doi.acm.org/10.1145/2234336.2234338.

[KR87] Richard M. Karp and Michael O. Rabin. "Efficient Randomized Pattern-matching Algorithms". In: *IBM J. Res. Dev.* 31.2 (Mar. 1987), pp. 249–260. ISSN: 0018-8646. URL: http://dx.doi.org/10.1147/rd.312.0249.

[Lee+13] I-Ting Angelina Lee et al. "On-the-fly Pipeline Parallelism". In: *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures.* SPAA '13. Montreal, Canada: ACM, 2013, pp. 140–151. ISBN: 978-1-4503-1572-2. URL: http://doi.acm.org/10.1145/2486159.2486174.

[Lib] LibreSSL. URL: http://www.libressl.org/ (visited on 01/10/2016).

[Lv+07a] Qin Lv et al. "Ferret: a toolkit for content-based similarity search of feature-rich data." In: *EuroSys.* Ed. by Yolande Berbers and Willy Zwaenepoel. ACM, May 16, 2007, pp. 317–330.

[Lv+07b] Qin Lv et al. "Multi-probe LSH: Efficient Indexing for High-dimensional Similarity Search". In: *Proceedings of the 33rd International Conference on Very Large Data Bases.* VLDB '07. Vienna, Austria: VLDB Endowment, 2007, pp. 950–961. ISBN: 978-1-59593-649-3. URL: http://dl.acm.org/citation.cfm?id=1325851.1325958.

[MRR12] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation.* Elsevier Science, 2012. ISBN: 9780123914439.

[MSM04] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming.* First. Addison-Wesley Professional, 2004. ISBN: 0321228111.

[MSS04] Steve Macdonald, Duane Szafron, and Jonathan Schaeffer. "Rethinking the pipeline as object–oriented states with transformations". In: *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments.* 2004, pp. 12–21.

[Nav+09] Angeles Navarro et al. "Analytical Modeling of Pipeline Parallelism". In: *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques.* PACT '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 281–290. ISBN: 978-0-7695-3771-9.

[Oraa] Oracle. URL: `http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/o13-024-sparc-t5-architecture-1920540.pdf` (visited on 10/10/2015).

[Orab] Oracle. URL: `http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/o13-060-t5-multicore-using-threads-1999179.pdf` (visited on 10/10/2015).

[PC11] Antoniu Pop and Albert Cohen. "A Stream-computing Extension to OpenMP". In: *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers.* HiPEAC '11. Heraklion, Greece: ACM, 2011, pp. 5–14. ISBN: 978-1-4503-0241-8. URL: `http://doi.acm.org/10.1145/1944862.1944867`.

[Pro] Fedora Project. URL: `https://getfedora.org/de/` (visited on 01/10/2016).

[RCJ11] Eric C. Reed, Nicholas Chen, and Ralph E. Johnson. "Expressing Pipeline Parallelism Using TBB Constructs: A Case Study on What Works and What Doesn't". In: *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, VMIL'11.* SPLASH '11 Workshops. Portland, Oregon, USA: ACM, 2011, pp. 133–138. ISBN: 978-1-4503-1183-0.

[Rei12] James Reinders. *An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors.* Tech. rep. Intel, Oct. 2012. URL: `https://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors_1.pdf` (visited on 10/10/2015).

[Ric11] I.E. Richardson. *The H.264 Advanced Video Compression Standard.* Wiley, 2011. ISBN: 9781119965305. URL: `https://books.google.at/books?id=k7nOAiIUo9IC`.

[Roo+06] Ton Roosendaal et al. *Elephants Dream.* Tech. rep. Blender, Mar. 2006. URL: https://orange.blender.org/download/ (visited on 10/13/2015).

[San+11] Daniel Sanchez et al. "Dynamic Fine-Grain Scheduling of Pipeline Parallelism". In: *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques.* PACT '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 22–32. ISBN: 978-0-7695-4566-0. URL: http://dx.doi.org/10.1109/PACT.2011.9.

[SJ11] A. Lazzaro S. Jarp and A. Nowak J. Leduc. "Evaluation of the Intel Westmere-EX server processor". In: (July 2011).

[Spe] Spec. *HEP-SPEC06 Benchmark.* URL: https://w3.hepix.org/benchmarks/doku.php (visited on 05/01/2016).

[Suk13] Jim Sukha. *Piper: Experimental Support for Parallel Pipelines in Intel Cilk Plus.* Tech. rep. Intel, Aug. 2013. URL: https://www.cilkplus.org/sites/default/files/experimental-software/PiperReferenceGuideV1.0_0.pdf (visited on 09/20/2015).

[UGT09] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. "Software Pipelined Execution of Stream Programs on GPUs". In: *In CGO '09: Proc. of the seventh annual IEEE/ACM Intl. Symp. on Code Generation and Optimization.* 2009.

[Ull75] J.D. Ullman. "NP-complete scheduling problems". In: *Journal of Computer and System Sciences* 10.3 (1975), pp. 384–393. ISSN: 0022-0000. URL: http://www.sciencedirect.com/science/article/pii/S0022000075800080.

[VCN13] Hans Vandierendonck, Kallia Chronaki, and Dimitrios S. Nikolopoulos. "Deterministic Scale-free Pipeline Parallelism with Hyperqueues". In: *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis.* SC '13. Denver, Colorado: ACM, 2013, 32:1–32:12. ISBN: 978-1-4503-2378-9. URL: http://doi.acm.org/10.1145/2503210.2503233.

[Vid] Videolan. *x264 the best H.264/AVC encoder.* URL: https://www.videolan.org/developers/x264.html (visited on 10/13/2015).

[Vos11] M. Voss. *How to make a pipeline with an Intel® Threading Building Blocks flow graph.* Tech. rep. Intel Developer Zone, 2011. URL: https://software.intel.com/en-us/blogs/2011/09/14/how-to-make-a-pipeline-with-an-intel-threading-building-blocks-flow-graph (visited on 02/10/2015).

[WGB] Beilun Wang, Lin Gong, and Chunkun Bo. URL: https://www.cs.virginia.edu/~lg5bt/files/Intel%20Xeon%20Phi%20Coprocessor.pdf (visited on 10/10/2015).

[Wim] Martin Wimmer. *Pheet.* URL: http://pheet.org/ (visited on 05/10/2016).

[Wim13] M. Wimmer. “Wait-free Hyperobjects for Task-Parallel Programming Systems”. In: *2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*. May 2013, pp. 803–812.

[Wim14] Martin Wimmer. “Variations on Task Scheduling for Shared Memory Systems”. PhD thesis. 2014.

[ZL77] J. Ziv and A. Lempel. “A universal algorithm for sequential data compression”. In: *Information Theory, IEEE Transactions on* 23.3 (May 1977), pp. 337–343. ISSN: 0018-9448.

[zli] zlib. URL: `http://www.zlib.net/` (visited on 01/10/2016).

[ZLP08] Benjamin Zhu, Kai Li, and Hugo Patterson. “Avoiding the Disk Bottleneck in the Data Domain Deduplication File System”. In: *Proceedings of the 6th USENIX Conference on File and Storage Technologies*. FAST’08. San Jose, California: USENIX Association, 2008, 18:1–18:14. URL: `http://dl.acm.org/citation.cfm?id=1364813.1364831`.