Parallelism by means of multithreading in Clean

Author:
J.J.H Groothuijse

Supervisor:
dr. P.W.M. Koopman
Second reader:
dr. P.M. Achten

Master's Thesis
Computing Science
Radboud University Nijmegen
April 17, 2016

Abstract

The current generation of processors can execute multiple threads at once. In order to get all performance from the current generation of processors, programs must leverage multithreading such that each processor core gets a share of the work. `Clean` is a lazy pure functional programming language, which means expressions in `Clean` can be evaluated independently of each other. This allows parallel evaluation to be used without changing the semantics of a program.

However, the current implementation of `Clean` does not support either parallel evaluation or multi-threading.

Using the `par` and `pseq` combinators, a `Clean` programmer is able to use multithreading to increase performance, given that the program has portions that can be evaluated in parallel. This work presents four variations of the `par` and `pseq` combinators, implemented in `Clean`.

The presence of a garbage collector complicates the design of the `par` and `pseq` combinator. The way the garbage collector works, prevents us from sharing data directly between threads. Instead, data is packed and copied, to allow the communication needed to distribute work and gather the results.

These combinators have been used to implement case studies, to give insight in the expected overhead and scalability of the combinators. The case studies show that the combinators are suited to execute `Clean` programs efficiently on multi core processors. Using a quad core processor, most case studies show a significant speed-up.

Contents

1 Introduction **7**

2 Background **9**
2.1 Concepts . . . 9
2.2 Computer hardware . . . 10
2.3 Problems in utilizing hardware . . . 11
2.4 Goal . . . 13

3 Starting point **15**
3.1 Runtime system support . . . 15
3.2 Concept of multithreading in `Clean` . . . 15
3.3 Limitations of the starting point . . . 16
3.4 Thread libraries . . . 17

4 Implementation of combinators **19**
4.1 Thread creation . . . 19
4.2 Copying the graph . . . 19
4.3 Synchronization . . . 20
4.4 Variants of thread creation . . . 21
4.5 Asymmetry of operational semantics . . . 23
4.6 Strictness . . . 24
4.7 Uniqueness . . . 24

5 Case Studies **25**
5.1 Fibonacci case study . . . 25
5.2 Parallel map case study . . . 28
5.3 Merge sort case study . . . 29
5.4 Traveling salesman problem case study . . . 30

6 Comparison with other languages **33**

7 Related work **37**

8 Future work **41**

9 Conclusion **43**

A Results for examples **49**

B Cross language examples **51**

Chapter 1

Introduction

In 2005, AMD introduced the first dual core CPU for the desktop market. Dual core CPUs essentially are two processors sharing the same chip, doubling the maximum number of calculations per second compared to a single core CPU. To harness this new calculation throughput, a program has to consist of multiple threads, at least one for each core. Since then four, six and eight cores have become common place, meaning single threaded applications can only utilize a fraction of current hardware.

Multithreading allows a program to be executed such that more than one piece of the program is executed at once.[1] Traditionally threads are used to run multiple tasks simultaneously by a program, for instance to keep the interface buttons responsive while calculating something, or while having multiple network or file system actions.

`Clean` is a functional programming language based on graph reduction. Together with `Haskell`, `Clean` is one of the very few so called lazy pure functional programming languages. Being a lazy language means that function arguments are only evaluated when needed. This contrasts with most other programming languages which use eager evaluation, meaning that all function arguments are evaluated before calling a function. Being a pure functional language means that there are no mutable variables and no side effects. These restriction lead to referential transparency, enabling reasoning about equality of functions and data. Referential transparency and free evaluation order make these languages in principle well suited for parallel evaluation.

A `Clean` program is made up of reduction rules and a root graph. The reduction rules are specified by functions. Graphs contain nodes, nodes have a node-id [Groningen, 1990]. Using node-ids allows for nodes to be shared among graphs. Instead of evaluating a node multiple times, a shared node is evaluated only once. Having a node-id also allows graphs to be cyclic [Nöcker et al., 1991b]. When executing the program, the root graph is reduced using the reduction rules. When the graph can no longer be reduced, it is said to be in *normal form* and the program is done. If a graph representing a data structure is evaluated to *head normal form*, the structure of the root node of the graph is revealed, but no more [Hartel et al., 1995]. In this thesis the words *calculate*, *compute* and *evaluate* all mean *reduce*.

The last major rewrite of `Clean` dates from the early 2000s, several years before multi cores became available in the consumer market. Earlier versions of `Clean`, `Concurrent Clean`, allowed the programmer to annotate graphs which could be evaluated in parallel [Nöcker et al., 1991a]. `Concurrent Clean` evaluated parts of the reduction graph on different computers. The last rewrite ended support for parallel annotations as used in the late 90s. Currently `Clean` does not actively support multithreading and no operators or combinators exist to make use of multithreading.

The goal of this project is to make it easy to use multithreading in `Clean` by providing `Clean` primitives to start new threads and to return the result to the current thread.

There are two challenges for the efficient execution of Clean programs on a multi core processor. The first is to decide which parts of the program can be executed on another core. The second challenge is to limit the interaction between cores working on the same program, so that threads don't have to wait. The first is solved by having the programmer explicitly annotate how work may be split. To solve the second challenge, several variations of the combinator are implemented to see which interaction model is best at limiting the interaction between cores.

[1]This was always possible by running the program multiple times at once, but using multithreading each thread gets access to the same memory allowing these threads to collaborate more efficiently.

This report documents the process of executing the project, it warns for pitfalls and summarizes the lessons learned. This document can be used as a guide and starting point for future attempts at providing multithreading primitives in `Clean`.

Chapter 2 *Background* defines concepts used in this thesis, provides an overview of the problems that this project addresses and states the goal of the project. In chapter 3 *Starting Point* the current state of development in `Clean` is discussed, including work that was done to make `Clean` thread safe and `Clean` modules that were adapted to serve in this project. It then goes on to outline which libraries were used outside of `Clean`, why these are necessary and how these libraries are used. The steps needed to implement the combinators are described in chapter 4 *Implementation.* In chapter 5 *Use cases*, a few simple use cases illustrate usage of the combinators and what performance characteristics should be expected in terms of overhead and scaling. In chapter 6 *Comparison with other languages*, one of the use cases is implemented in `C`, `C++`, `Java` and `Haskell`, to compare speed-ups and efficiency. Chapter 7 *Related Work* discusses a selection of papers on the subject of graph rewriting, parallel programming, distributed programming and concurrency. Chapter 8 *Future Work* outlines possible improvements and additions to the current implementation. Chapter 9 *Conclusion* reflects on the process and the results, with recommendations for possible future implementations.

Chapter 2

Background

The following sections first define concepts that will be used throughout this thesis, then use those concepts to describe the environment of this project and the problems associated with multithreading, before describing the goal of this project.

2.1 Concepts

The following concepts are used throughout this thesis.

Parallelism

Parallelism in the context of computing science means that a computation is split in parts, and that those parts are executed at the same time. The ultimate goal is to have the computation take less time.

Process

A process is a running application [Vyssotsky et al., 1965]. A process has its own memory that no one else can read from[1] or write to. Multiple processes use inter-process communication, called IPC, to communicate with each other. IPC can be provided by the operating system or through facilities like the file system and network. IPC includes Pipes (Unix sockets), message queues and shared memory [pipe, 2016, messagequeue, 2016, shm, 2016]. These facilities add overhead.

Thread

A thread is the smallest unit of program execution that can be managed by the operating system. A process contains one or more threads [DaveMcCracken, 2002]. A thread shares its memory with other threads of the same process. This enables communication between threads, without copying data. It also gives rise to a number of things that can go wrong, which we discuss in section 2.3 Problems. Depending on the available resources these threads can be executed interleaved or in parallel.

Garbage Collection

Memory management in `Clean` is done using garbage collection. When `Clean` needs new memory, it is allocated on the end of the heap, which is a cheap operation. When the heap is full however, the request for new memory starts `Cleans` garbage collector. The garbage collector inspects the heap to find memory that can no longer be reached (garbage) to be reclaimed. Reclamation happens by compaction, data is moved and references to data are updated accordingly [Groningen et al., 1991]. Moving data and updating references is not something that can be done in parallel with another operation using that same data.

[1]Forked processors can read from their parents memory due to copy-on-write semantics

2.2 Computer hardware

This section introduces the current state of affairs in computer hardware, outlining areas where the problems discussed in the next section arise from.

CPUs

Legacy desktop CPUs[2] can only work on one thread at a time. To create a multi tasking environment, the operating system features task schedulers. These schedulers allow programs to run interleaved on the CPU.

Processors with hyperthreading act as two processors to the operating system. These processors have the administrative capacity to allow two threads to run simultaneously, but they lack the execution units to allow two threads to run at twice the speed a single thread could run.

Current CPUs do have multiple cores. Multi core CPUs essentially are multiple processors on one chip, sharing memory. In order to use all cores, each core must have its own thread to work on.

Memory

The state of the program is not only stored in memory. A crucial part is stored in the registers on the CPU. Register values can be copied to and from memory. Registers allow faster access to data and compilers try keep important and often used data in these registers. Registers are also essential to the processor itself, it keeps track of the stack frame pointer and the program counter[3].

Part of the memory is cached in the processor to speedup access. This means a copy of the memory is made inside special cache memory. This cache memory resides on the CPU and has a very limited capacity. This means it only copies a tiny portion of the memory. The CPU is responsible for copying to and from memory and for administration.

Cache Hierarchy

Modern CPUs have multiple cores, each core is seen as a processor by the operating system. These cores have their own execution units and registers and in general each core has its own cache, up to a certain level of cache. So parts of the memory is copied to caches and registers, which are not shared among cores, while the main memory itself is shared among cores. The memory topology is of vital importance to the attained speedup of a multithreaded program. Caches keep a local copy of a tiny part of the memory. When the threads mutate memory that is copied in the cache of another core, an effect known as *false sharing* occurs, see figure 2.1 (a). Cache coherency mechanisms continuously force the other core to clear its cache and fetch data from main memory. False sharing can result in performance degradation compared to a single threaded version of a program.

The cache hierarchy is a result of a cost/performance trade off for CPUs, trying to get maximum performance from a limited chip size. A larger cache requires the more logic for its address decodes, and it has a longer critical path, increasing latency. So large caches tend to be slower than small ones. But larger caches can store more data and thus have a better hit rate. Because of this, CPUs typically have a small fast cache followed by a larger and slower cache. [Paul Genua, 2004]

To combat false sharing, different cores must access memory that is far enough away from each other, see figure 2.1 (b).

[2]Consumer grade x86 processors commonly used in desktops and laptops

[3]Only the program counter is truly essential, programs can be written in assembly such that they do not require a stack frame pointer.

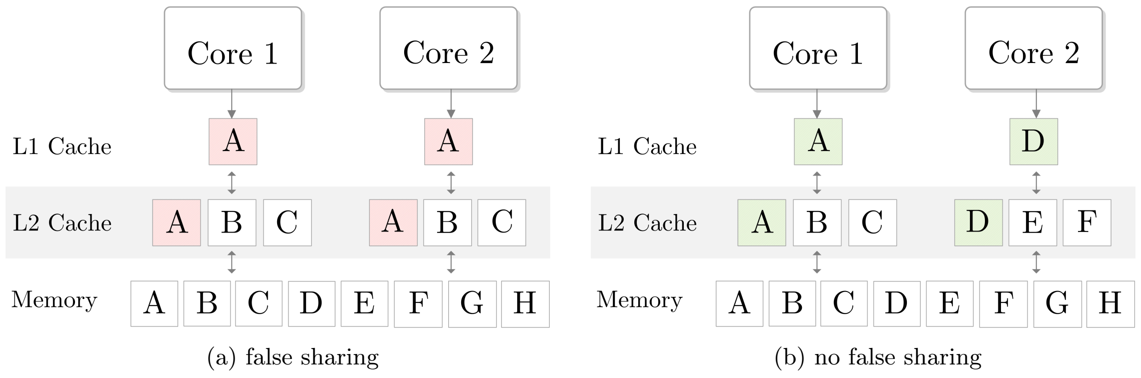

Figure 2.1: Memory topology of a dual core processor. Illustration by Kim Weustink.

2.3 Problems in utilizing hardware

Single threaded applications utilize only a fraction of the computational hardware of modern PCs. To get maximum performance out of ordinary desktop computers, we must use multithreading. Multithreading introduces a number of problems, related to memory use, distribution of work and garbage collection. Each of these problems is discussed in detail in this section.

Problems related to memory use

Sharing memory and allowing mutations of memory to happen in parallel gives rise to these problems:

1. When data structures are shared and mutated, some threads can see (partly) outdated memory.
2. Invariants cannot be guaranteed for shared data structures.
3. All global variables are shared by default, compounding the latter two problems.

1. Different threads typically run on different cores. Each core has cache containing a local copy of data in memory, see subsection 2.2 for details. While CPUs do have a cache coherency, optimizations in CPUs and compilers may change the order of execution of instructions, leading to inconsistency. Optimizations that do not lead to inconsistency in a single thread, can lead to inconsistency when using multiple threads. [Meixner et al., 2006] Furthermore, for registers no such coherency mechanism exists, so only load and store operations propagate changes.

Another problem with mutating shared data is *false sharing*, which occurs when pieces of memory in cache are updated by another core. This causes the cache coherency mechanism to invalidate cache lines and thus reduce performance. When this happens frequently, the benefit of multithreading can be completely negated. See section 2.2 for more details.

2. Using multithreading, memory is available to read from and write to, by multiple threads. To ensure correctness of programs, programmers use invariants. Invariants are assertions that hold when entering and exiting a block of code. [Hoare, 1972] While executing the block, the invariant may be broken, so long as its restored before exiting. In normal programs that means that any code that is not inside the code block, or called from without the block, can assume that the invariant holds. In multithreaded programs this assertion does not hold for invariants concerning memory that is shared with other threads.

3. A global variable has a fixed memory location, because of this, all threads will always have access to it, so all global variables are shared. In shared memory the previous two problems can occur.

To combat this situation, locks are introduced. Locks allow exclusive access to certain resources. When a thread acquires a lock, other threads requesting that lock wait until the first thread releases its lock. However, locks can lead to diminished performance and give rise to deadlocks, when not used appropriately.

Problems related to the distribution of work

There is no obvious way to distribute work automatically, because one can not decide whether a computation takes long enough to make parallelizing worthwhile.

Problems related to garbage collection

Garbage collectors delete and move data. While a garbage collection cycle is running, the heap can be in an inconsistent state. This means that without special care, in a multithreaded program some threads may see inconsistent data, due to garbage collection. Several garbage collection strategies are implemented in other languages. For this project we focus on how to make it thread safe. Approaches by other languages include pausing all threads in `Java`, parallel garbage collection `Haskell` and one shared heap and a separate private heap per thread in `OCaml`.

```
eulerPlusFib :: Int → Int
eulerPlusFib n = x par y pseq (x + y)
where
    x = fib n
    y = euler n
```

Figure 2.2: Example of use of the par and pseq operators

2.4 Goal

The goal of this project is to make it easy to use multithreading in Clean.

Past versions of Clean, formerly called Concurrent Clean, allowed the programmer to specify parallel evaluation by annotations. This suggests parts of the graph could be computed on a different computer. A major rewrite ended this support for declarative parallelism. Currently Clean does not actively support multithreading, so no operators or combinators exist to specify potential parallel evaluation.

To solve the distribution of work problem, the programmer needs to define the distribution explicitly. To that end, the par and pseq combinators are selected as interface for explicit thread management [Marlow et al., 2009]. Using par and pseq combinators, a Clean programmer is able to use multithreading to increase performance, given that the program has portions that can be evaluated in parallel.

Reasons for choosing par and pseq include:

- They are well defined in literature.
- Other languages have successfully implemented them.
- They are low level enough to build other abstractions on.
- They are high level enough to encapsulate representations of threads and locks.
- They conceptually integrate well in lazy functional languages.

Semantics

The par and pseq combinators should have special operational semantics. Their presence should indicate part of the calculation can be done by another thread.

```
(par)  infixl 2 ::  a b → b
(pseq) infixl 3 :: !a b → b
```

The operator `par x y` evaluates x to head normal form, on a different thread, while evaluating y on the main thread. It returns y when both y and x are evaluated. The operator `pseq x y` first evaluates x and then returns y. When par and pseq are used in expression `x par (y pseq z)`, x is calculated by another thread while y is simultaneously calculated by the main thread. When both x and y are normalized they can be used in expression z.

This definition allows room for design decisions, which we explain in chapter 4 *Implementation*, where we introduce four different variations. These semantics differ slightly from the par and pseq operator in Haskell, see section 4.4 for details.

Chapter 3

Starting point

The starting point for this project is unpublished work that was done by John van Groningen.

John has made a version of the runtime system to support multithreading and created a number of sample programs, to show the feasibility of multithreading in `Clean`.

This work has resulted in support in the runtime system, a conceptual model for using multiple threads and a low-level implementation of an example. After a short recap of the problems surrounding parallelism in `Clean`, the remainder of this chapter discusses each of these parts.

As stated in section 2.1, when multithreading is used all global variables become shared by default. While `Clean` does not allow a user to specify such global variables, its runtime system uses global variables for its internal bookkeeping.

The shared address space supplied by multithreading allows us to directly share graphs. However, references to graphs can be invalidated by the garbage collector. This leads to a design where the graph specifying work to be outsourced, is copied to the heap of another thread. Since graphs can have references to other graphs, a deep copy is required. This involves packing the graph in a format such that it can be copied to the destination thread, where it is unpacked.

3.1 Runtime system support

`Clean` uses a runtime system written in `C`. This runtime system uses a number of global variables, like the stack pointer and heap pointer. The stack pointer and heap pointer are not meant to be shared and the program becomes corrupted if multithreading is used. So, `Clean` is not thread safe. To make `Clean` thread safe, John has modified the runtime such that each thread gets its own version of these global variables, by providing a thread specific offset in a register. Each thread also has its own heap. The runtime also contains an entry point function, to be called when a new heap and stack are allocated. This allows `Clean` threads to run completely independent.

3.2 Concept of multithreading in `Clean`

The use of one garbage collector per thread solves many problems commonly associated with multithreading in a garbage collected language, but it comes at a price: subgraphs must be copied from memory of one thread to memory of the other thread.

Packing and copying the graph

`Clean` contains functions to pack graphs, these functions are used by the `ITask` [itasks, 2015] system. The `ITask` system is a task-oriented programming toolkit, which uses the graph packing and copy functions for persistent storage. We can not directly use these functions, as we need to be able to specify where the packed graph will reside, in order to avoid problems with garbage collection. However, the core logic for packing and unpacking the graph can be completely reused.

Example multithreaded programs

Using this thread safe version of `Clean`, John has implemented three programs demonstrating the use of multiple threads. This program uses mostly `ABC` and machine code embedded in `ABC`, to let the threads communicate with each other. This communication happens via memory allocated by an ABC instruction to call `malloc`, its address is then transfered to each thread by a parameter of the `pthread_create` method.

The examples consist of three programs:

- One program to demonstrate how to copy a node from one heap to another.
- One program that does a sanity check using both multithreading and graph copying functions.
- A GTK(GIMP Toolkit, a multi-platform toolkit for creating graphical user interfaces [gtk, 2015]) program that calculates and draws a fractal using multiple threads.

Contained in the programs are small pieces of code, that do the following things:

- Create a thread, by calling a `C` function from the `pthreads` library.
- Share memory / resources, by calling the `C` function `malloc` and giving each thread the pointer that `malloc` returned.
- A mechanism for waiting on other threads, using wait functions from the `pthreads` library (`sem_open`, `sem_post`, `sem_wait`).
- A mechanism to copy `Clean` graphs into strings and from strings back into graphs again.

3.3 Limitations of the starting point

While these example programs demonstrate the use of multithreading in `Clean`, they lack any form of abstraction layer. Changing the program now needs in-depth knowledge of the ad-hoc allocated memory structure, so for now it contains no functionality that is useful to other programmers.

For this project that means it is the perfect starting point for creating those abstractions (primitives) to add multithreading to `Clean` such that a programmer can easily use it. Problems that have been left open include the following:

- To spark threads and use their results.
- Primitives for the programmer.
- A mechanism to cache threads (thread pools).
- A way to limit the total number of threads.

3.4 Thread libraries

It is common practice when creating software to use libraries of existing software, such that abstractions for the hardware and operating systems are already made. This not only saves work during the project itself, but also during maintenance, as those libraries get updated to reflect new hardware and operating system changes. The choice for libraries can be very important in the long run, as some libraries eventually disappear while others remain supported.

In my opinion, open source libraries are the safer option, because other people can continue development after the original authors have stopped. Whether this actually happens depends on the state of the project and the number of users.

This project uses two libraries. The first is the `pthreads` library, which is part of the `POSIX` standard and abstracts the way threads are created on all major operating systems [DaveMcCracken, 2002]. The second is the `stdatomics` library, part of the `C11 ISO` standard and used to abstract from hardware when communicating between threads and thus possible communication between cores [Vafeiadis et al., 2015]. The remainder of this chapter focuses on these libraries and how they are used.

Pthreads

`pthreads` is a standard to manage threads in all major operating systems. This section briefly describes the functions of `pthreads` that are used by multithreaded `Clean` and how they are used.

sem_open, sem_wait, sem_post

We use semaphores supplied by `pthreads` to coordinate multiple threads. Semaphores are a synchronization primitive [DIJKSTRA, 1968]. Currently we use semaphores to give the ability for one thread to wait on another thread. This was implemented in `pthreads` specifically for this purpose, so we assume it has decent performance and that it is in fact thread safe. Moreover, in the documentation we read it uses signals to wake up threads, ensuring limited overhead on the CPU [semwait, 1997].

Although a possible alternative approach is using `pthread_join`, using semaphores offers a better path towards thread pools. This is because `pthread_join` waits until a child thread ends. When implementing thread pools, we want a thread to do work and synchronize, and we don't want a thread to end.

pthread_create

`pthread_create(thread, attr, start_routine, arg)` creates a new thread and makes it executable. This includes making a new program counter and making the operating system aware of the presence of this thread [pthreads, 1997]. In the start routine we supply the address of `clean_new_thread`, which is a built-in `Clean` function to initialize a heap and stacks for a `Clean` program. In `arg` we supply a pointer to a space where we store a pointer to a semaphore, a `Clean` function and the location of the copied graph.

C11 stdatomics

In `C11`, a recent `C` standard, `stdatomics.h` was added to the standard library, to standardize atomic operations and *sequential consistency*. It ensures that compliant programs, when compiled by compliant compilers, run as expected on every platform. It contains the functions needed to atomically read from and write to shared variables, in the context of multithreading.

When a memory model is sequentially consistent, memory accesses by multiple threads are interleaved and totally ordered. This contrasts all recent hardware architectures, in which only single threaded programs are sequentially consistent [Vafeiadis, 2015].

According to the `C11` standard, data races using non-atomic access result in completely undefined behavior. Such data races are treaded as programming errors. Only programs without those data races are `C11` compliant programs [Vafeiadis, 2015].

C atomic_load(const volatile A* obj)

"Atomically loads and returns the current value of the atomic variable pointed to by `obj`*. The operation is atomic* `read` *operation"* [atomicload, 2015]. This operation is used to get the current thread count. By

using this particular function, the memory reordering of both the compiler and processor are constrained to force sequentially consistent behavior, while dealing with a shared variable.

`_Bool atomic_compare_exchange_strong( volatile A* obj, C* expected, C desired )`

"Atomically compares the value pointed to by `obj` *with the value pointed to by expected, and if those are equal, replaces the former with desired (performs* `read-modify-write` *operation). Otherwise, loads the actual value pointed to by* `obj` *into expected (performs load operation)"* [atomiccompare, 2015]. This function is used to increment or decrement the thread count, knowing what the thread count was right before executing. If the value has changed in the mean time, the function returns false and our local variable gets updated to the current value (as if we `called atomic_load` again). We loop until this function returns true. So we repeat until the value is not updated in the mean time (by other threads), this makes sure no updates are lost. A process diagram of this process given in figure 3.1.

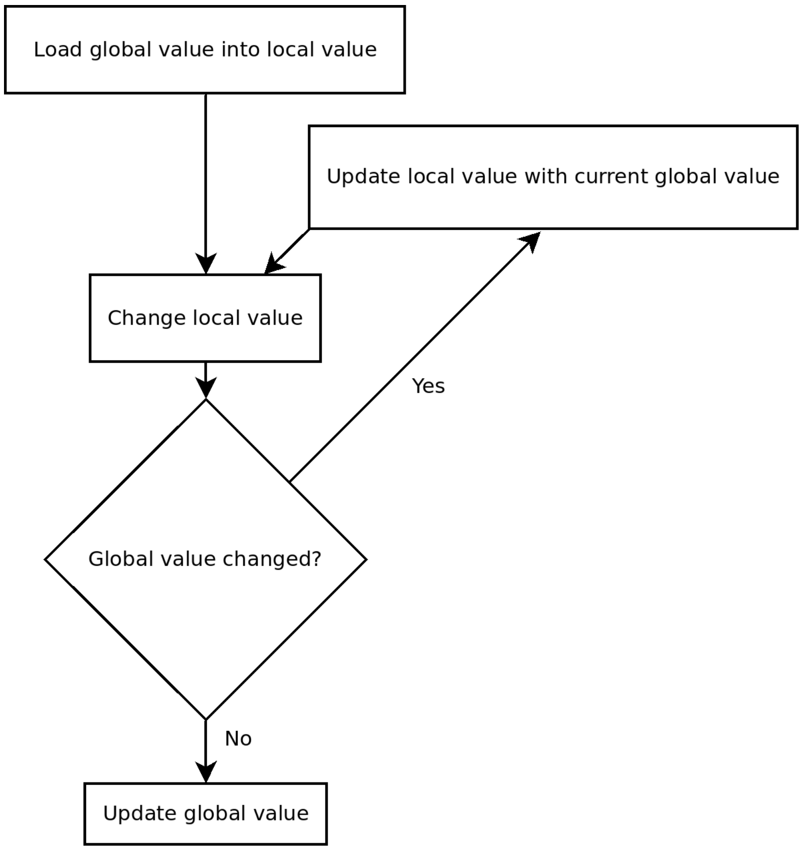
Load global value into local value
Update local value with current global value
Change local value
Yes
Global value changed?
No
Update global value

Figure 3.1: Process diagram of illustrating the typical atomic compare and exchange usage. The nodes "Global value changed", "Update local value with current global value" and "Update global value" are all implemented by `_Bool atomic_compare_exchange_strong( volatile A* obj, C* expected, C desired )`, and are all executed as one atomic operation.

Chapter 4

Implementation of combinators

This chapter outlines implementation details and ratio behind the implementation decisions.

As explained in section 2.4, the goal of the project is to make it easy to use multithreading in `Clean`. This is implemented, by providing `Clean` primitives that split and divide work over multiple threads. The `par` and `pseq` combinators are those primitives. The `par` operator expresses potential parallelism, `x par y` means `x` and `y` can be evaluated in parallel. The `pseq` operator expresses sequentiality, `x pseq y` means `x` is evaluated before `y` is evaluated. Informally, the operational semantics of `x par y pseq z` can be described as follows:

1. Both `x` and `y` are evaluated in parallel.
2. After `x` and `y` are evaluated, `z` is evaluated and returned.

Evaluating `x` and `y` in parallel requires creating threads and moving work to those threads. In order to evaluate `x` and `y` in parallel, the `par` operator evaluates `x` on a new thread while at the same time evaluating `y` on the original thread. Threads are created using the `pthread` library discussed in chapter 3.4, the implementation details can be found in section 4.1.

Copying the graph is addressed in section 4.2.

To wait until both `x` and `y` are evaluated before evaluating `z` requires a mechanism to wait for thread completion. The implementation, using the semaphores of the `pthreads` library, is detailed in section 4.3.

We experimented with several thread creation schemes. Section 4.4 provides an overview of these variants. Section 4.5 explains how using a limited number of threads, can lead to asymmetry in the `par` and `pseq` operators, with respect to CPU utilization and provides a solution. Strictness and uniqueness of the operators are discussed in section 4.6 and section 4.7.

4.1 Thread creation

The `pthread` library, described in section 3.4, is used to create a thread such that the operating system schedules its execution. To keep track of the thread in `Clean`, a small piece of memory, a 40 byte block, is allocated to describe the thread. It contains the address of the `Clean` function to be executed, the size of the heap for the thread (offset 8), 8 bytes of storage for the return value (offset 16), the address of the semaphore (offset 24) and the address of the workload (offset 32). The purpose of the semaphore is synchronization between threads, details can be found in section 4.3.

4.2 Copying the graph

How garbage collection works and why it is problematic

While `Clean` itself and its execution model are very well suited for parallel execution, its garbage collection mechanism is not. To manage the heap, `Clean` allocates objects on the heap until it runs out of space. `Clean` uses two garbage collection schemes [Groningen et al., 1991]. The copying collector uses only one half of the heap. When this half is full, all nodes that are reachable are copied over to the other half. While being very fast, this limits memory usage to half the heap. For programs that continuously use almost half of the heap space, garbage collection gets triggered too often. So when that happens, `Clean` switches to another collection scheme. The sliding compaction is a mark-scan scheme, which marks all accessible nodes, then scans the heap twice. The first scan updates all forwarding pointers, the second moves the nodes and updates backward pointers. Both of these schemes involve changing the heap by moving nodes and updating pointers.

If we allow graphs to become shared among threads, this causes a problem. During a garbage collection cycle, a shared graph can be deleted by the thread that created it. Even if a graph is not deleted it can be moved, which also invalidates the pointer held by the other thread. Because of these problems, sharing graphs among threads is not feasible with the current garbage collector.

Safe graph copying

The chosen concept of multithreading in `Clean` prevents cache problems by creating threads that work on their own graph. When a thread is created, a graph to be reduced is moved to it. The `copy_to_string` and `copy_from_string` functions provide the ability to store a graph in a string, and recreate the graph from a string. These methods create and use a string in the heap, to store the graph. Since heap memory can be reclaimed or moved by the garbage collector, a reference residing outside of heap H to a string inside heap H can be invalidated each time the garbage collector runs. To alleviate this problem, a version of `copy_to_string` and `copy_from_string` was made to store a graph in a separate piece of memory. This piece of memory is called *graph buffer* from now on. Each thread gets its own graph buffer. The graph buffer stores the graph in a packed representation.

The graph buffer is allocated outside of the `Clean` heap, see figure 4.1. Since the packing operation is already necessary in order to move graphs across heaps, using this buffer imposes only minor additional overhead. In the current implementation the size of the graph buffer is fixed.

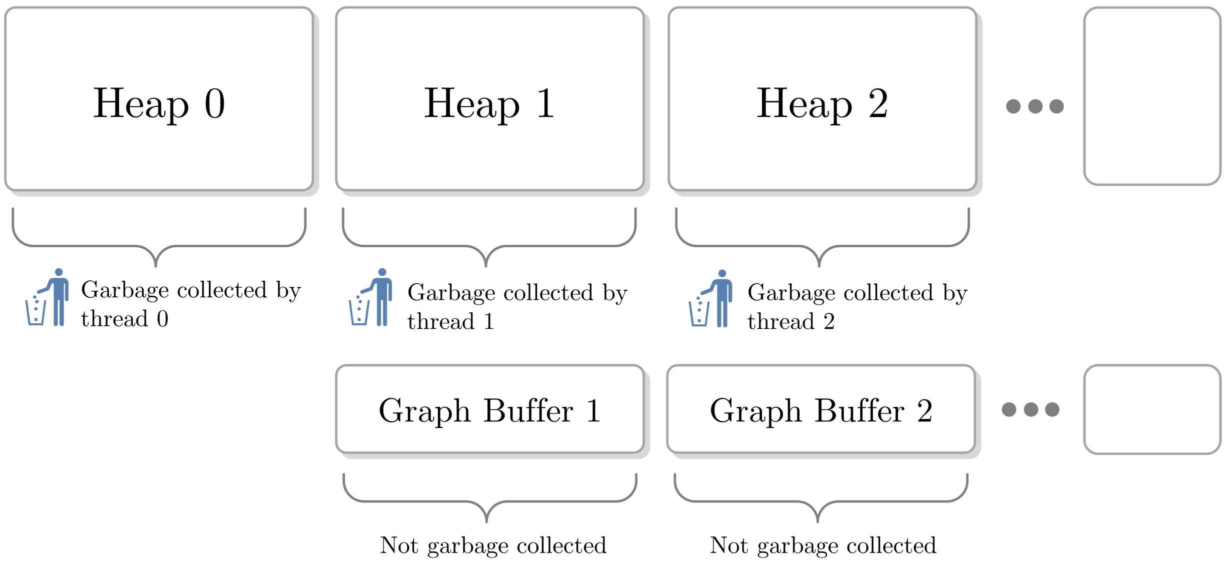

Figure 4.1: Memory layout of a multithreaded `Clean` program. Illustration by Kim Weustink.

Hypothetically, instead of copying to the graph buffer, the heap of the newly created thread could also be used to store the packed graph. Because the new thread has not yet started, no garbage collection cycle will remove it. This requires much tighter integration with the `ABC`-machine, because the string should be placed on the `A-stack`. Therefore, this option is not chosen for this project, but future multithreading primitives in `Clean` may want to consider it.

4.3 Synchronization

After a thread is created, the function that created the thread returns immediately. The newly spawned child thread starts reducing the graph. When the graph is in head normal form, it is packed again in the graph buffer and the child thread signals its completion, using the `sem_post` function on the semaphore pointer stored in its thread descriptor.

The parent thread first reduces its own graph to head normal form, and then waits for the child thread to finish, using the `sem_wait` function. If the child thread was already finished this returns immediately, otherwise execution waits until the child thread is done.

These actions of waiting and notifying when done allow synchronization between threads.

4.4 Variants of thread creation

Several variations of the `par` and `pseq` operator are created to test different implementation choices. Experiments have been carried out with these variants, which are discussed in chapter 5.

`pseq1` Default version

The default `par` and `pseq1` implementation is based on the `par` and `pseq`, found in the Haskell module `Control.Parallel` [Marlow et al., 2009].

```
par  :: a → b → b
pseq :: a → b → b
```

In Haskell, the `par` function indicates its arguments may be evaluated in parallel and returns the second argument. The `pseq` function specifies order, its first argument must be evaluated before its second argument is evaluated and returned [Peyton Jones, 2008].

Our `par` and `pseq1` support similar use, but work differently. Instead of the first two arguments to `par` and `pseq` being evaluated implicitly (without data dependency), in this implementation the last argument is a function that receives the first two arguments in head normal form. This approach is more consistent with the lazy nature of the language.

```
// Simplified signature for par and pseq1
(par)   infixl 3 :: a (a → c) → c
(pseq1) infixl 4 :: b (a b → c) → (a → c)
```

Because the second argument of `pseq1` needs the types of both arguments of `par` to check for type safety, the type signature of `par` and `pseq1` had to change from the original operators like implemented in Haskell. In the Haskell `par` and `pseq`, the last parameter is a graph of which the first two parameters can be subgraphs. Because we chose to have the last parameter be a function that receives the first two parameters in head normal form, we must make sure the type of this last parameter matches the first two.

`pseq1` produces a function that is used as the second argument of `par`. The second argument of `pseq1` is of type `(a b → c)`. The a is repeated in the result type `(a → c)` of `pseq1`. Because this result type is the argument to `par`, we can specify `par` such that it forces consistent type, forcing the second argument of `pseq` to match the type of its first argument, `a`, with the type of the first argument of `par`, also named `a`. The next example shows how `par` and `pseq1` can be used to calculate two values simultaneously, using multiple threads.

```
// Illustrative example of par and pseq1 usage

ack :: Int Int → Int
ack 0 y = y + 1
ack x 0 = ack (x-1) 1
ack x y = ack (x-1) (ack x (y-1))

Start :: String
Start = (ack 4 2) par (ack 4 1) pseq1 (λx y. "ack 4 2:" +++ toString x
                                       +++ "ack 4 1:" +++ toString y)
```

The default `par` and `pseq1` version uses only a fixed number of threads. If less than a certain number threads are active, the `par pseq1` operators cause one more thread to be used. If the maximum number of threads is reached, the graphs are evaluated sequentially and no additional threads are created by par and pseq1. This version is the default, because it is closest to the par and pseq combinators of Haskell.

These par and pseq1 operators are really a facade for another function, that takes three arguments and does the actual work. The function that does the actual work is called parseq1 and takes 3 arguments. Each variant has its own parseq function. Since parseq1 is called from pseq1 and not from par, we only need one par function for all variants.

pseq2 Using an alternate graph instead of sequential execution

The pseq2 variant takes one more argument.

```
// Simplified signature for par and pseq1
(par)   infixl 3 :: a (a → c) → c
(pseq2) infixl 4 :: b ((a b → c), c) → (a → c)
```

This argument is an alternative way to compute the result. When the number of threads are exhausted, this alternative graph is returned. This prevents attempts to parallelize deeper in the graph, thus saving on overhead. In the next example a function called mfib calculates Fibonacci using multiple threads. When all threads are in use, it switches over to a single threaded variant.

```
// Illustrative example of par and pseq1 usage

fib :: Int → Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

mfib :: Int → Int
mfib 0 = 1
mfib 1 = 1
mfib n = (mfib (n-2)) par (mfib (n-1)) pseq2 (+, fib n)
```

In example above, if at any point no thread is available and fib n is evaluated, then it does not request another thread again. This can decrease overhead, at the cost of potentially less resource utilization.

pseq3 Using a thread pool

A thread pool can be used decrease the overhead of creating new threads and semaphores each time par and pseq1 are called. Starting new threads takes time, so reusing old threads can save time and thus lower the overhead when repeatedly using multithreading. To this end, a thread pool is created. Upon first use of a thread, a certain number of threads are initialized with corresponding semaphores and graph buffers, one for each thread. When another thread is requested, it either gets a free thread or a result indicating that no free thread is available at this time. In this latter case par reverts to sequential execution. The two variants above do not use a thread pool, a third variant, pseq3, uses a thread pool.

```
// Simplified signature for par and pseq3
(par)   infixl 3 :: a (a → c) → c
(pseq3) infixl 4 :: b (a b → c) → (a → c)
```

pseq4 Unlimited number of threads

If the algorithm already has a means to control the number of threads it will try to spawn, then the thread counter and control logic of par pseq1 operators add unnecessary overhead. To remove this overhead a version was created, that does not adhere to any maximum number of threads. The pseq4 variant spawns a new thread on every call.

```
// Simplified signature for par and pseq4
```

```
(par)   infixl 3 :: a (a → c) → c
(pseq4) infixl 4 :: b (a b → c) → (a → c)
```

4.5 Asymmetry of operational semantics

During experimentation a usage pattern was discovered, in which all threads where used, but only one thread was still calculating. Figure 4.2 illustrates this situation and its effect on CPU usage.

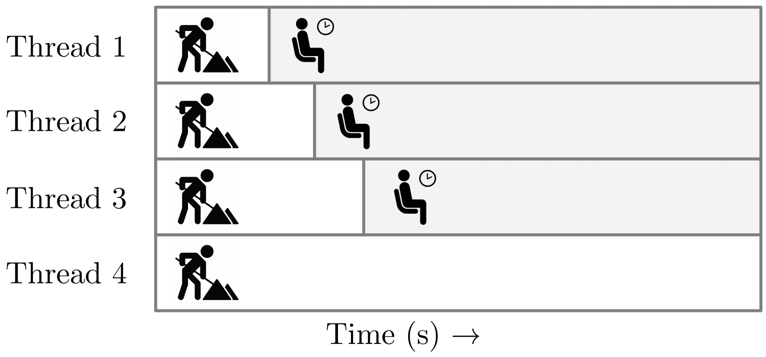

Figure 4.2: The effect on productivity of reverting to one thread. Illustration by Kim Weustink.

The cause of this problem is that the operational semantics of the `par pseq1` operators are asymmetrical. This means that `x par y pseq1 (λx y.z)` is not operationally equivalent to `y par x pseq1 (λy x.z)`. In an attempt reduce the number of threads, these operators only create one new thread to evaluate two tasks. One of the tasks is performed on the current thread. Specifically the first argument, `x`, is evaluated on a newly created thread, while the second argument, `y`, is evaluated on the current thread. This means that if the first argument is more work, contains potential parallelism and does so recursively, then it drains all threads and proceeds to evaluate on 1 thread, while the other threads are waiting. Figure 4.3 shows a graph which would cause this.

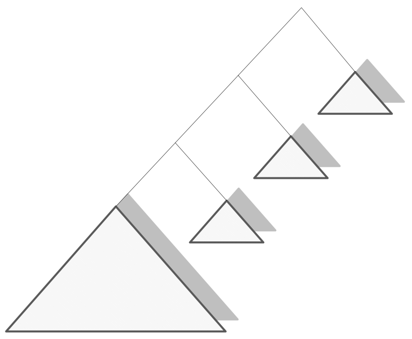

Figure 4.3: In this graph each left branch is always more work than the right branch. Illustration by Kim Weustink.

This is not an edge case. Unless the work can be split in two perfectly equal parts, this problem occurs.

This asymmetry is a problem, because the user might not know which graph should be the first, and which graph should be the second argument. One of the original problems explained in section 2.3, computing for all two graphs, which graph takes longer to normalize, is undecidable. This also means we can not assume the programmer is *always* able to know. So it makes sense to create a `par pseq` variant that is symmetrical, such that we can parallelize efficiently, without knowing which part of the work takes longer.

The solution is to limit only the number of *active* threads. In order to do so, when a thread needs to wait, it decrements the thread count. This allows room for other threads to be spawned and increases resource utilization. While a program still has potential parallelism behind the `par` and `pseq` operators, the situation depicted in figure 4.2 does not occur.

This solution is only viable for the default `par` and `pseq1` operators and the variant with an alternative graph `pseq2`. In the other variants, because of their asymmetry, better performance is expected when the second argument of `par` is more work than the first.

4.6 Strictness

The `par` and `pseq` operators are strict in both arguments. If any of the arguments do not normalize, then the `par` or `pseq` operator does not normalize to head normal form. This is a departure from the `par` and `pseq` in `Haskell`, which is only strict in the second argument of both functions [Marlow et al., 2009].

The `par` and `pseq` operators normalize the first two arguments to head normal form and then give these normalized graphs to the third argument. In order to be able to do work in parallel, the first two arguments must not be in head normal form. If they are in head normal form, then no work is left to do in parallel. So the arguments to the `par` operator must not be annotated as strict, because doing so forces the arguments to be in head normal form when `par` is called.

4.7 Uniqueness

All variants of the `par` and `pseq` combinator support uniqueness on both arguments.

Sharing work across threads is at odds with uniqueness. A unique node may, by definition, not be shared. However, the `par` and `pseq` operators do not *share* graphs with other threads.

The `par` and `pseq` operators evaluate the graph of the second argument on the current thread. This graph is not copied or passed to another function, so it preserves uniqueness. The other graph is packed to be evaluated by another thread. After packing a graph, the original graph is no longer used. Thereby only one reference to a graph is kept, and so the first argument also preserves uniqueness.

The following program demonstrates passing arguments to `par` and `pseq1` and writes to two files in parallel.

```
// Opens two files to append, writes "hello" to both in parallel and then closes the files again.
Start world
♯ (success, file1, world) = fopen filename1 FAppendText world
| not success = abort ("Failed to open " +++ filename1)
♯ (success, file2, world) = fopen filename2 FAppendText world
| not success = abort ("Failed to open " +++ filename2)
= (file1 ⋘ "hello") par (file2 ⋘ "hello") pseq1 (closeFiles world)
where
    filename1 = "out1.txt"
    filename2 = "out2.txt"
    closeFiles :: *World *File *File → *World
    closeFiles world x y
    ♯ (_, world) = fclose x world
    ♯ (_, world) = fclose y world
    = world
```

In this case the function supplied to `pseq1` is also unique since it captures `world` which is of type `*World`. In order to allow `par` and `pseq1` to be called this way, the types must be appropriately annotated.

```
// Actual signature for par and pseq1
(par) infixl 3 :: .a .(.a → .c) → .c
(pseq1) infixl 4 :: u:b v:(.a → .(u:b → .c)) → w:(.a → .c), [w ≤ u,w ≤ v]
```

The variant that uses an alternative graph instead of sequentially executing the first two arguments, `pseq2`, also supports uniqueness, but this combination is very hard to use due to the presence of this alternative graph. The example above can not be implemented using `pseq2`, since specifying an alternative graph that does not reference the files, does not implement the desired behavior.

For `pseq3` and `pseq4` the example can be implemented, analogous to the version using `pseq1`.

Chapter 5

Case Studies

This chapter illustrates the usage of multithreading in `Clean`, using the `par` and `pseq` combinators. These case studies show that the combinators work and that in some cases a speedup of 3.7 is possible. They are not meant to show that one way of dividing work always performs better than some other way.

This chapter consists of examples demonstrating all `par` and `pseq` variations.

- The first case study shows the usage of all the `par` and `pseq` operator variants, using a naive Fibonacci function. The Fibonacci function is used in other papers and lets us compare results factoring in environment changes.
- In section 5.2 a more abstract example, a potential `pmap` function, is proposed. The `pmap` function is a higher level combinator, its implementation shows abstraction layers can be build on top of `par` and `pseq`.
- In section 5.3 a merge sort is implemented using the `par` and `pseq` combinator. The merge sort algorithm exposes the biggest weakness of our implementation, because the data to work on is copied repeatedly.
- A more concrete use case is given in section 5.4, where an implementation of the traveling salesman problem are discussed. The traveling salesman problem is implemented using the `pmap` function, showing that the abstraction works on more complex examples.

Measurements

All programs are tested on a quad core system, specifically an `AMD Athlon II X4 635` processor using `Linux kernel 3.13.0 x86_64`. The running time and CPU usage is measured using the `linux` utility `time`. The `time` utility provides three metrics: `system` which is not used, `real` which is called elapsed time and `user` which is called CPU time. Using these metrics to compare to a single threaded version, we can measure overhead using CPU time and measure speedup using the elapsed time.

To improve accuracy and repeatability on each test run each program is executed and measured five times. Of those five times the slowest result deleted. The rationale for the deleting the slowest result, is that the running time increases, when another process on the test system get scheduled.

5.1 Fibonacci case study

Our first case study is the Fibonacci function. The Fibonacci function is used to demonstrate basic usage of the different variants.

This Fibonacci was chosen for its simplicity, it can be used to demonstrate how to use `par` and `pseq`. Furthermore, it is used in other papers, allowing some basis for comparison.

A non threaded version of fib:

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

pseq1: Parallel fib

In the parallel version, `pfib`, if `n` is high enough, then `pfib (n-1)` and `pfib (n-2)` are evaluated in separate threads. The cutoff value for `n` is 25. For `pfib` calls with `n` smaller than 25, no threads are spawned.

```
pfib :: !Int → Int
pfib n
| n < 25 = fib n
         = pfib (n-2) par pfib (n-1) pseq1 (+)
```

Since `n` only gets lower, it makes no sense to keep checking if `n` is low enough. Hence, we can improve performance by having the algorithm switch to a non parallel version, once `n` is low enough.

pseq2: Alternative graph

As explained in 4.4, a variant of the `par` and `pseq` operators, `pseq2`, returns an alternative graph when the maximum number of threads is reached.

```
pfib n
| n < 25 = fib n
         = pfib (n-2) par pfib (n-1) pseq2 (+, fib n)
```

pseq3: Threadpool

Another variant, discussed in section 4.4, used a fixed size pool of eight threads. An example program using this thread pool looks as follows:

```
pfib n
| n < 25 = fib n
         = pfib (n-2) par pfib (n-1) pseq3 (+)
```

pseq4: Limiting the number of threads in the algorithm

Another approach is to count and restrict the number of threads in the algorithm, to a more reasonable number. The next example allows the user to set a bound on the maximum number of thread created. By allocating twice as many threads to the most demanding recursive call, the work is better balanced.

```
pfib b 0 = 1
pfib b 1 = 1
pfib 0 n = fib n
pfib b n = pfib (b / 3) (n-2) par pfib (2 * b / 3) (n-1) pseq4 (+)

Start = pfib 12 42
```

In this example the first argument is the number of available threads. If the algorithm itself can limit the number of threads being used, then not using the built-in thread counter reduces overhead and can improve performance. That is why this example uses `pseq4`, which always creates a new thread.

Expectations

We don't expect a speedup close to four. The number of recursive calls to `par` and `pseq` is high enough to add significant overhead.

Of our variants we expect `pseq1` to be the slowest, since all other variants have some strategy to improve performance over `pseq1`. We expect `pseq2` to be a minor improvement over `pseq1`, because while reducing overhead, the algorithm misses potential parallelism after switching to the alternative graph. We expect `pseq3` and `pseq4` to be the fastest performers. `pseq4` distributes the work optimally, while adding only a slight overhead. `pseq3` also distributes work better than `pseq1`, but can potentially add more overhead than `pseq4`.

Results by variant

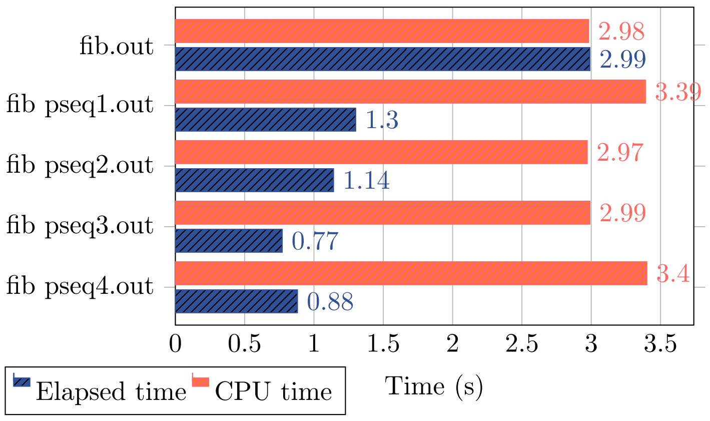

Figure 5.1: Test results using a quad core system, calling `fib` and `pfib` with 42 as argument `n`.

These results show that different parallel strategies make a difference in both the running time and the system resource usage. `pseq3` has the lowest latency while `pseq2` incurs the least overhead. All variants attain a speedup larger than three and the maximum measured overhead is 14% in this example.

As expected, `pseq1` was the slowest and `pseq2` offers a minor improvement in speed with very low overhead. Both `pseq3` and `pseq4` are noticeably faster. We also expected `pseq3` to have more overhead than `pseq4`, but this is not the case. In fact, `pseq3` has ten percent less overhead than `pseq4` and is ten percent faster.

Scaling

Using an alternative version of `fib`, we can count how many recursive calls are made:

```
nfib 0 = 1
nfib 1 = 1
nfib n = 1 + nfib (n-1) + nfib (n-2)
```

Using this number of recursive calls, we can calculate how many function calls per second a program makes. When we compare theses numbers for different values of `n`, we can see how the programs scales.

We expect to see almost flat lines, indicating there is no problem with scaling.

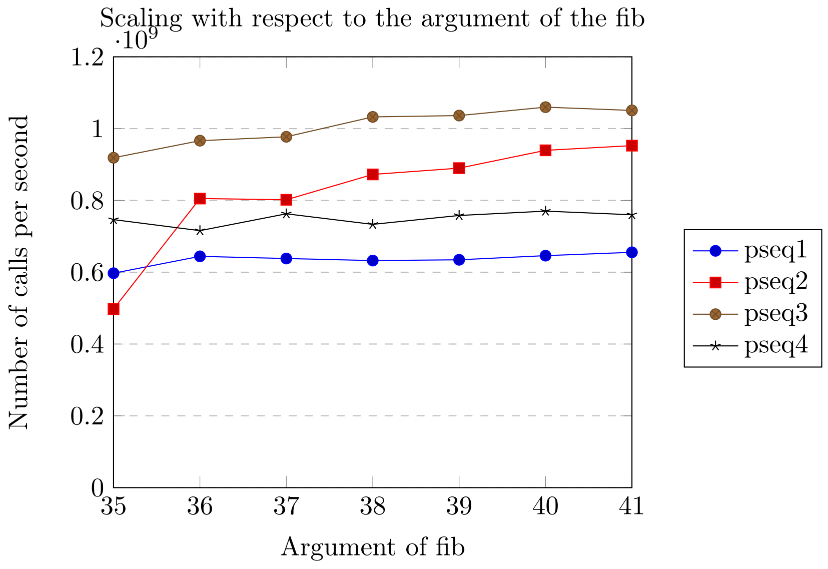

Figure 5.2: Results for implementations across parameter values of the fib function

In figure 5.2, we can see that there is no performance degradation when the number of requests for new threads increases. This means they scale well. The difference between the lines corresponds to the previous result, `pseq3` is clearly faster.

5.2 Parallel map case study

In this example we make a more abstract function. We define a parallel version of the `map` function, called `pmap`. We do this to show that abstractions can be built on top of the `par` and `pseq` functions, meaning that `par` and `pseq` can be used to create libraries.

Listing 5.1: parallel map function.

```
pmap f [] = []
pmap f [x:xs] = (f x) par (pmap f xs) pseq1 λx y.[x:y]

fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

Start = pmap (fib) [40, 40, 40, 40, 40, 40, 40, 40]
```

The `pmap` function works by calculating `f x` in parallel with `pmap f xs`, so the tail is evaluated in parallel. This implementation of `pmap` creates a new thread for every element of the list and it evaluates the spine of the list almost instantly. That makes this `pmap` function strict in its list argument.

Listing 5.2: A single threaded version using map is used as reference.

```
Start = map (fib) [40, 40, 40, 40, 40, 40, 40, 40]
```

We expect that `pseq4` gets close to a speedup of four, because it spawns seven additional threads and divides the work evenly among all threads. `pseq2` is expected to not have a significant speedup, after switching to the single threaded graph, there is still work with potential parallelism left. We expect that both `pseq1` and `pseq3` have better resource utilization than `pseq2`, but worse than `pseq4`.

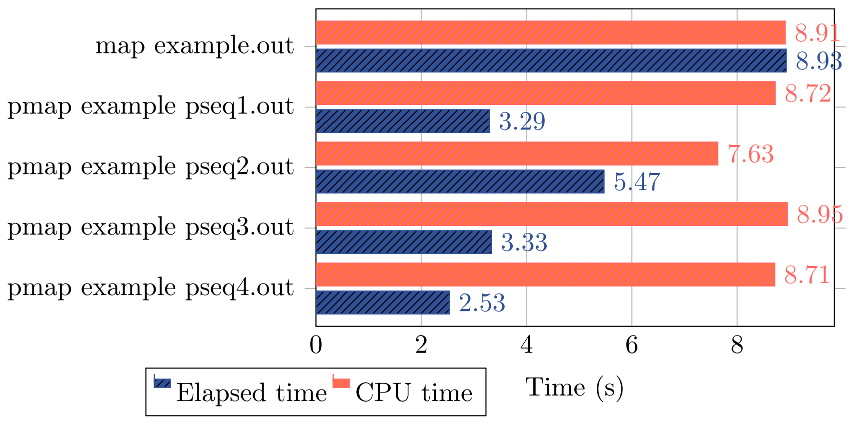

Figure 5.3: PMap test results using a quad core system.

As expected in this example the `pseq1` the speedups remain low, as we can see in figure 5.3. We can see in the results, that `pseq1` has the same overhead as `pseq4`, which was also expected. Due to lower resource utilization, it has a lower speedup, 2.7 for `pseq1` against 3.5 for `pseq4`.

`pseq4` scores well, but slightly less than expected. Moreover, it creates a thread for every element, which will not scale to very large lists.

5.3 Merge sort case study

Merge sort orders a collection, using a divide and conquer strategy. It takes an element [1] and divides the collection into two parts, one part for all elements of the collection smaller than the chosen element and one part for all elements larger than the choosen element. On these two parts, merge sort recursively applies itself. Now merge sort only needs to append the two parts with the chosen element in the middle, to return an ordered collection. Recursion stops when the collection size is less than two, because collections of size zero and one are always ordered.

This case study was chosen because it fits our par and pseq implementation poorly. Our par and pseq combinators require that data to be processed is copied. Merge sort works on large data sets. Copying those datasets is very costly. This means that merge sort gives insight into how much overhead copying adds, and what kind of algorithms are unsuitable for our par and pseq.

Our merge sort implementation uses unboxed arrays as collections and integers as elements. The time complexity for the merge sort in our example is $O(n * log(n))$, which is appropriate for merge sort. However, its space complexity is also $O(n * log(n))$, while merge sort optimally uses only $O(n)$ space. Instead of using two buffers and switching source and destination at every recursive call, each recursive call in our implementation allocates its own (smaller) buffer. The extra allocations make this version slow compared to the optimal merge sort.

Listing 5.3: merge sort, the mergeTo function merges two sorted arrays into one existing array.

```
mergesort data
♯ (size, data) = usize data
| size < 2 = data
♯ leftSize = size/2
♯ (data, left) = cpy 0 data 0 (createArray leftSize 0) leftSize
♯ rightSize = size-leftSize
♯ (data, right) = cpy leftSize data 0 (createArray rightSize 0) rightSize
= mergeTo data (leftSize - 1) (mergesort left) (rightSize - 1) (mergesort right)
```

Merge sort works by merging two sorted halves of a data set. The sorted halves are sorted by recursion, which stops when the data size reaches one or zero. The merge action is not parallelized, only the two recursive calls are executed in parallel. This limits the attainable speedup. The merge step has $O(n)$ complexity. When four threads are used, the last merge step is executed by one thread and second to last merge step is still only executed by 2 threads. Only the deeper recursive calls are distributed over 4 threads, explaining the limited effect when going from two to four threads. This effect is illustrated in figure 5.4.

Listing 5.4: parallel merge sort.

```
pmergesort data 0 = mergesort data
pmergesort data t
♯ (size, data) = usize data
| size < 2 = data
♯ leftSize = size/2
♯ (data, left) = cpy 0 data 0 (createArray leftSize 0) leftSize
♯ rightSize = size-leftSize
♯ (data, right) = cpy leftSize data 0 (createArray rightSize 0) rightSize
♯! (left, right) = pmergesort left (t/2) par pmergesort right (t/2) pseq1 λx y.(x,y)
= mergeTo data (leftSize - 1) left (rightSize - 1) right
```

We expect the additional resource utilization to be negated by the need to copy the data to and from the graph buffer.

Due to the unique arrays, pseq2 could not be used. This is because pseq2 requires an alternative graph to compute the solution, but when the original computation works on a unique array, the alternative graph cannot work on that same array.

Due the size of the graph buffer, pseq3 could not be used. The threads in pseq3 are pre-allocated. They have a smaller size graph buffer, to compensate for the fact that they are pre-allocated and thus always take up this memory.

[1]elements can be randomly selected

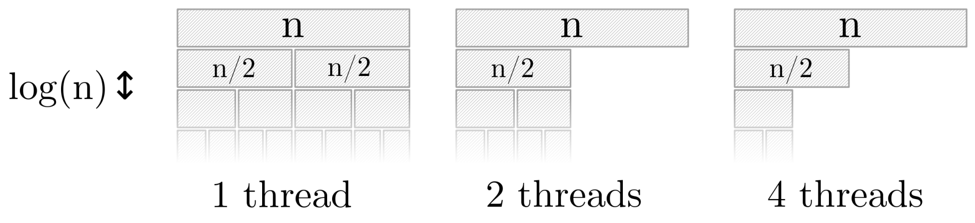

Figure 5.4: Reduced effectiveness when parallelizing recursive calls in a divide and conquer algorithm. Surface area represents execution time. We can see the top call is not parallelized, the second level is only split in half.

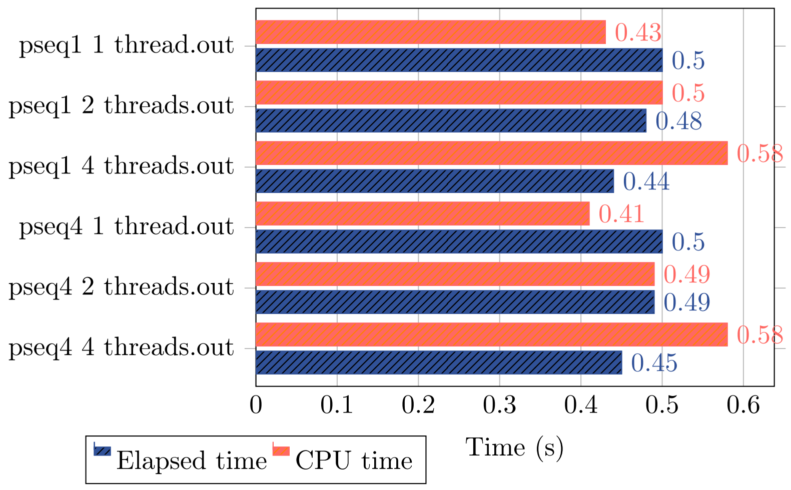

Figure 5.5: `Merge sort` test results using a quad core system, ordering 2000000 numbers

We can see that the CPU time in figure 5.5 rises steadily when more threads are used. Since thread creation is not a significant factor, this overhead must be attributed to copying data between threads. Thread creation is not a significant factor, because we are only creating two or four threads.

Contrary to our expectations, using more threads did allow a small speedup. In figure 5.5 we see that both `pseq1` and `pseq4` have a minuscule speedup using two threads, 1.04 times and 1.02 times respectively. Furthermore, there is a minor speedup using four threads, 1.13 times and 1.11 times respectively.

5.4 Traveling salesman problem case study

In this case study we use the earlier defined `pmap` function, to solve the traveling salesman problem. This case study is therefore more concrete, demonstrating a possible utility of the `par` and `pseq` combinators.

In this example we use a table to represent the costs of traveling from one city to another. This table is symmetrical across the diagonal and the diagonal itself only contains zeros. So, we only need one half of the table, minus the diagonal to store all costs. In our implementation we use a two dimensional array of reals, where the arrays in the outer array are progressively smaller, e.g. for four cities we have and outer array of size three with inner arrays of size three, two and one. Furthermore, we refer to cities using integers.

Listing 5.5: Function to calculate costs between city `src` and city `dst` using `table` to lookup the costs..

```
cost :: {{ Real }} !Int !Int → Real
cost table src dst
| src == dst = 0.0
| src >  dst = table.[src].[dst]
             = table.[dst].[src]

generateTable :: Int → {{ Real }}
generateTable c = { { (toReal i) + (toReal j) * 10.0 \\ j ← [1..i + 1] } \\ i ← [0..c - 1] };
```

The algorithm itself uses no heuristic, it tries every option recursively and returns the route that yields the lowest cost. By limiting the cities to be considered as the first city of the route, this task can be slit across threads.

Listing 5.6: tsp calculating list with for every city the best route starting in that city.

```
tsp table [pos] = [(0.0, [pos])]
tsp table toVisit = [
    case tsp table (filter ((≠) pos) toVisit) of
        [x:xs] = let (c, path) = foldl min x xs in
            (c + cost table pos (hd path), [pos: path])
    \\ pos ← toVisit]
```

The `tsp` function returns a list of best routes, given a first city. From this list the best is selected and returned as result of the program. In the single threaded version the best is selected using a normal `foldl`. The parallel version applies `pmap` to the list with the best routes. The `pmap` function puts each element of the list in head normal form, in parallel. The head of this list has a data dependency on the rest of the list, computing the entire result. So each best route, given a first city, is calculated in parallel.

Listing 5.7: Singlethreaded version.

```
tsp_s table toVisit = case tsp table toVisit of [x:xs] = foldl min x xs;
```

Listing 5.8: Multithreaded version, uses `pmap` to evaluate list in parallel.

```
tsp_p table toVisit = case pmap (id) (tsp table toVisit) of [x:xs] = foldl min x xs
```

We expect to see speedups similar to the `pmap` function, since we are using it as parallelize our implementation.

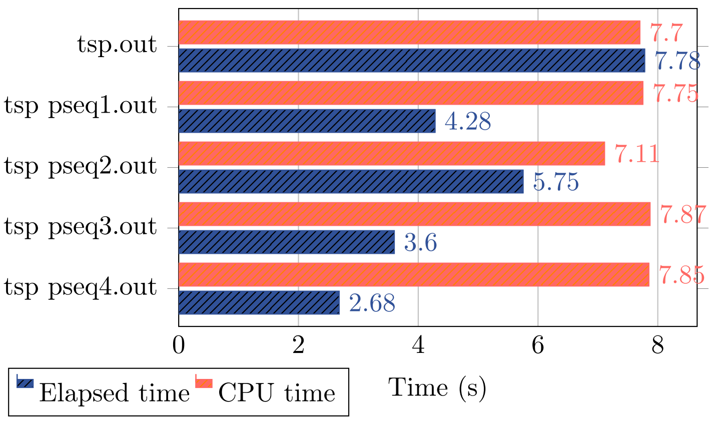

Figure 5.6: TSP test results using a quad core system with 11 cities

As we can see in figure 5.6, in this example using the `pmap` function the speedup is just under three. It hardly adds overhead and is well distributed across the resources.

Contrary to our expectation the speedups are much lower than in the `pmap` case study. This could be due to higher memory usage outside of the caches, which causes the memory controller to become a bottleneck. The memory controller is physically shared between all cores.

Chapter 6

Comparison with other languages

Other programming languages have similar features to use multiple cores. In this chapter we will compare multithreading libraries of these languages with our own implementation. The remainder of this chapter consists of implementation and experimentation notes, per language, and a discussion of the results in section 6. The languages `C`, `C++` and `Java` are chosen because they are widely used, the language `Haskell` was added because of its similarity with `Clean`.

As an example program, the `pfib` function from section 5.1 is used. Because the Fibonacci function we are using is not tail recursive, the functional languages do not have an unfair advantage in this scenario. The complete source code for all of these examples can be found in the appendix B. Figure 6.1 shows the results of the measurements.

C

`C` is the only language in this comparison that by itself does not offer any primitive for parallel programming. The `C11` standard gives primitives to make memory operations thread-safe and while it also prescribes standard threads library, no compiler currently supplies this library.

Two language extensions are used to supply `C` with parallel programming primitives, `OpenMP` targeting `Fortran`, `C` and `C++` and `Cilk` targeting `C` and `C++`.

- **Cilk**

  `Cilk` provides a language extension of `C` and runtime system based on a work-stealing scheduler [Blumofe et al., 1995]. The language extension consists of new keywords and even an list-comprehension-like array-notation. It is the only primitive which ran a version of the Fibonacci with an unlimited number of tasks, without error and in reasonable time using all cores. In our example the keywords `cilk_spawn` and `cilk_sync` are used to express the fork and join.

- **OpenMP**

  `OpenMP` consist of compiler directives and runtime functions [Dagum and Menon, 1998]. The compiler directives are well integrated and compiler errors are generated when the directives are flawed, for example when defining a variable which did not yet exist, as shared. Directives to indicate how something is parallelized are decoupled from directives that indicate parallel execution is used. This decoupling allows the same function with `OpenMP` directives to be used with and without parallelism.

  The `pragma task shared(n2)` directive forks off the function call to `parallel_fib(n - 2)`. The `pragma omp taskwait` directive lets the program wait for any task in its scope. `OpenMP`'s task based directive suffers the same operational asymmetry described in section 4.5. This means that if `parallel_fib(n - 1)` is forked off on another thread instead of `parallel_fib(n - 2)`, then the performance degrades. On the machine we used for testing, the latency increases by 50%.

C++

`C++` provides a number of primitives for parallel programming in its standard library, including a thread library with locks and condition variables [Stroustrup, 2014].

- **async**

  According to [Stroustrup, 2014] *"with async() you don't even know how many thread s will be used because that's up to async() to decide based on what it knows about the system resources available at the time of a call. For example, async() may check whether any idle cores (processors) are available before deciding how many thread s to use"*. By specifying `std::launch::async` as first argument to the `std::async` function, we are asking for the job to be executed on another thread. Joining is accomplished by the `get` operation, on the `future` of the task. This task was returned by our call to `std::async`.

- **packaged_task**

  A packaged task is slightly more low level. There is no logic to decide when threads are spawned, the user supplies the threads directly to the task. A `future` is used to represent the out-sourced computation. The `get` operation on this `future` will join and return this result. Templates are used to ensure type safety of `packaged_task` and `future`.

Java

- **ForkJoin**

  Introduced in `Java 7`, ForkJoin provides an object oriented interface to Tasks and Futures, targeted at task parallelism. The example program uses a thread pool to reuse the threads, the size of the pool was set to four.

- **Parallel Stream**

  Introduced in `Java 8`, Parallel Streams allow operations to be applied to a stream, in parallel. This results in parallel map and parallel reduce type functions. This primitive is targeted at data parallelism. In the example program, the `parallelFib` function creates a parallel stream consisting of the numbers n-1 and n-2, maps itself recursively over them and then sums up the results.

Haskell

`Haskell` provides a `par` and `pseq` functions in the `Control.Parallel` module for use in task parallelism. The number of threads used by the program is specified at call time. The `+RTS -N8` argument specifies that eight threads should be used.

Cross language results

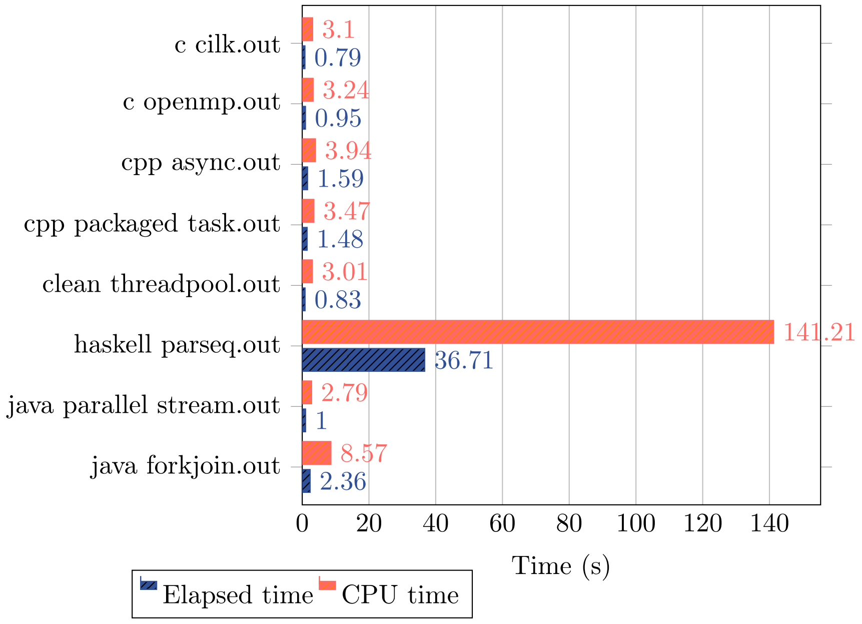

Figure 6.1: Cross language example results, all implemented the `pfib` function reverting to `fib` when n was smaller than 25.

The `Cilk` example has the lowest latency in this test. It achieves this using a reasonable amount of overhead. The `Clean` version using a thread pool comes in second, with less overhead than `Cilk`. `OpenMP` and the `Java Parallel Stream API` take up third and fourth place, getting very close in terms of latency. `Java Parallel Stream API` has the least overhead of all operators. All examples try to implement the same algorithm. For `Clean` and `OpenMP` the lesser recursive call was forked off to another thread. This variation was tested on all examples, but no other examples benefit from it. `Cilk` and `OpenMP` can also be used with `Fortran` and `C++`, but here we tested `C++` primitives found in the standard library.

Chapter 7

Related work

This chapter provides a short overview of work related to parallelism in pure functional languages.

Task parallelism

Task parallelism is parallelism by having multiple pieces of (sub)computation that can be computed independently of each other. These tasks may have nothing in common, they could be different functions and have different running times.

[Marlow et al., 2009] Provides an overview of `Multicore Haskell`. `Glasgow Parallel Haskell`, *GpH*, consists of two combinators, `par` and `pseq`. The first argument of `par` is stored as a *spark*, in the *spark pool*. A spark is work that can potentially be done in parallel. Idle processors can find useful work in the `spark pool`. The `pseq` function is used for sequencing, it makes sure its first argument is evaluated before the second argument. Our version of the `par` and `pseq` combinators was inspired by the `Haskell` implementation. `Haskells` runtime system, the `GHC`, uses lightweight threads, *Haskell threads*, which are multiplexed over operating system threads, *worker threads*.

The state of a Haskell thread, together with its stack, is kept in a heap-allocated thread state object, *TSO*. For each CPU (or CPU core) one *HEC*, a Haskell Execution Context, is maintained. This HEC contains, among other things, a message queue containing requests from other HECs, a local allocation area, a worker pool of spare worker threads and the spark pool.

Garbage collection is done very differently, because `Haskell` uses one heap that is shared by all threads. `Haskell` has a parallel garbage collector, but garbage collection only takes place when all HECs stop together, and agree to garbage collect. In contrast, in our implementation each thread has its own heap and the heaps are garbage collected independently.

[Kesseler, 1991] describes the early `Concurrent Clean` support for distributed computation. In these early versions, a cluster of computers was used and the network was considered the most important bottleneck. In current architectures, memory operations play a similar role. The paper presents an implementation of the `PABC` machine, for a transputer rack from Parsytec. Garbage collectors work locally and the problems of a global garbage collector are outlined. The system does not save registers when a context switch happens, so in certain sections (basic blocks and garbage collection) context switches are prohibited. This was accomplished by not using instructions that allowed context switches to occur. Problems of among others copying graphs, routing and load distribution are left as not entirely solved, though the discussion of copying graphs outlines the trade-offs. The `par` and `pseq` primitives described in this thesis work around the problems related to global garbage collection mentioned in [Kesseler, 1991], but are still confronted by the same trade-offs when communicating with other threads.

Instead of having the compiler decide what is done in parallel, this version of Concurrent Clean uses annotations to allow the programmer to specify jobs [Nöcker et al., 1991b]. The `P` annotation of `Concurrent` Clean serves a similar role to the `par` and `pseq` combinators. A `P` annotation indicates that an expression should be evaluated in parallel [Nöcker et al., 1991b].

[Groningen, 1992] describes the process of copying graphs, waiting for results calculated elsewhere and the various trade-offs of fine-grained and course-grained parallelism, polling and packing graphs. Waiting is implemented by polling and the resulting overhead is measured. While varying from program to program, the overhead remains quite low. The resulting programs are run on a system with sixteen simulated processors, the measured speedup is between 0.78 and 8.4. Our `par` and `pseq` operators do not use polling, instead they use signals, which are likely implemented as interrupts. As a result, the measured overhead of the `par` and `pseq` is much lower when we compare the results of the `fib` function.

[Trilla and Runciman, 2014] describes way of using profiling feedback information when compiling, so the compiler can determine which sparks should actually spark a new thread and which should not. Profiling assisted optimized compilation can be used to solve problems that are undecidable, by trying out variations on a representative workload. The downsides include the slower compilation times, difficulty finding a truly representative workload and the result being too specifically optimized for the host system, on which the compiler runs. The technique works on `par` and `pseq` combinators, like the ones implemented by this project and therefore it can be implemented on top of the current primitives.

[Beemster et al., 1993] discusses clustered architectures, wraps up pieces of `Concurrent Clean`, the `Parallel ABC Machine` and the Dutch parallel reduction machine project. It describes the hardware as multiple computers which form a cluster using an intra-cluster shared memory, and multiple clusters forming a network using the inter-cluster network.

A modern desktop PC roughly equates to a cluster in [Beemster et al., 1993] in terms of topology; these clusters had multiple processors and shared memory, which effectively matches of modern day multiple cores which share memory through multithreading.

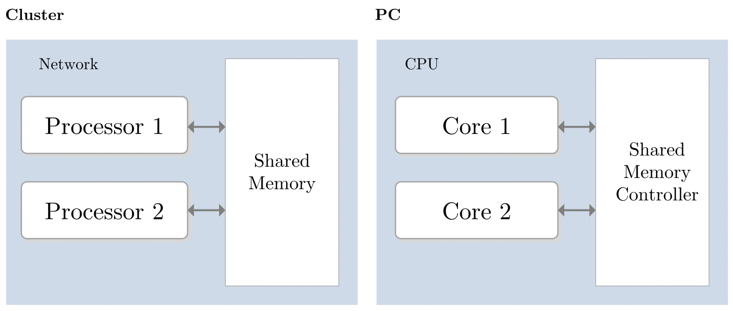

Figure 7.1: On the left: a cluster in [Beemster et al., 1993], on the right: a dual core CPU. Illustration by Kim Weustink.

In [Beemster et al., 1993] the `sandwich` primitive serves the same role as our `par` and `pseq`. The difference between the two is negligible. In `sandwich` the first argument is a function taking two arguments, in `par` and `pseq` the last argument is this function. Furthermore, in the `sandwich` primitive both expressions spawn a parallel *job*, while the `par` function only evaluates the first argument in a new thread.

Data parallelism

Data parallelism allows one operation (of any size) to be applied in parallel to a large dataset. This concept, of one operation, is more restrictive than task parallelism, which allows any operation to run in parallel. The fact that it parallelizes only one operation, allows it to scale automatically with the size of the dataset.

[Chakravarty et al., 2007], [Jones et al., 2008] and [Chakravarty et al., 2008] provide the current state of affairs of nested data parallelism in Haskell, based on `NESL`. Nested data parallelism allows data parallelism to be used on recursive structures. This process involves flattening the data and generating vectorized versions of the operations on the data. According to [Chakravarty et al., 2007] these vectorized operations are only feasible to generate in pure languages. This is an alternative approach based on data parallelism, our `par` and `pseq` primitives are classified as operators to specify task parallelism.

[Keller and Chakravarty, 1998] specifically discusses flattening of trees. This technique is used to transform a tree-like structure to a flat array. It enables algorithms to use tree like structures, while the parallelization mechanism works on flat arrays. This is an advanced technique for use in data parallel primitives. It enables users to specify algorithms on recursive data structures and still have it automatically parallelized.

While this should still not be considered the solution for all parallelization, it's a very effective solution when the amount of data to be processed large. The technique requires compiler changes and is therefore deemed out of reach for this project.

[Zsók et al., 2011] discusses algorithmic skeletons for computations on `D-Clean`. `D-Clean` is a distributed version of `Clean`, running on multiple computers. It features primitives to coordinate the distribution of work. These primitives operate on Channels, which act as streams or lists to send data through the system for processing. The presence of channels implies a disposition towards data parallelism, unlike our `par` and `pseq` operators which have a disposition towards task parallelism.

[Keller et al., 2012] explain a problem associated with flattening and vectorization[1], which can cause a performance penalty. Flattening of data involves vectorization all operations, so all scalar operations are transformed into vector operations. While this enables SIMD usages, it can cause memory overhead by introducing large intermediate structures.

Fusion is a technique to combat this situation. Fusion uses removes intermediate values by aggressive in-lining. Fusion has its drawbacks, such as the resulting program becoming very large and the resulting code can not be properly optimized by the GHC's code generator. This paper introduces analysis which allows the vectorization to be avoided when it offers no benefit.

[1]which is using the SIMD capabilities of processors to allow one core to do more calculations at once

Chapter 8

Future work

This chapter outlines a number of features that were not implemented in this project, but we expect would provide useful additions.

Specializing on primitive return types

The current implementation uses the same copy mechanism regardless of types. Using Generics to specialize on primitives we can prevent a `graph_copy_to` and `graph_copy_from` operation, by using the pointer to the graph to store primitive data instead. Thus, this specialization improves the efficiency.

Platform independence

Assembly code was used in parts of this project. Because the assembly code has to interface with `C`, and because the calling convention of `C` functions differs between Linux, Mac and Windows, this project only works on Linux. Thus, the current implementation only works on 64 bit Linux using x86-64 processors. `Clean` supports both 32 bit and 64 bit Linux, Mac and Windows. To get this project to work on the remaining five platforms, the assembly has to be ported to each of those platforms.

Preventing double work

In case the graph that is reduced in another thread, is included in the graph that is reduced locally, as illustrated in figure 8.1, we do not want to calculate that graph twice. Calculating the same graph twice is a violation of the graph reduction semantics and it wastes CPU cycles. To prevent graphs from being reduced twice, their contents is overridden after copying to the new thread took place.

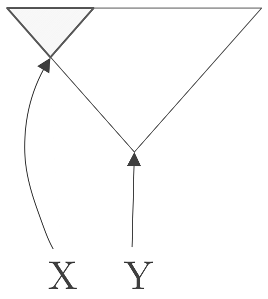

Figure 8.1: Graph x is included in graph y. Illustration by Kim Weustink.

The graph reduced on new thread can be replaced by a node which waits for the other thread to finish and then overrides itself with the correct value. Unfortunately, the graph reduced by the host thread can not be replaced in the heap of the new thread. This is because the new thread never receives the graph that its host is reducing. [Kesseler, 1994] presents a partial solution.

Skeletons for frequently occuring thread patterns

In the `tsp` example in section 5.4 parallelism is added using only a `pmap` over a list, this `pmap` was defined earlier in Examples. More skeletons need to be defined. Skeletons are used to separate algorithms, from a parallelization method [Danelutto et al., 1998].

Thread pool using more graph buffers

The current implementation the thread pool variant, `pseq3`, only uses a fixed number of graph buffers. While the thread pool saves on allocation time, it also means threads cannot be reused until after the result is read. Dynamically adding graph buffers while keeping the number of threads fixed, allows threads to be reused sooner.

Chapter 9

Conclusion

Hardware development in recent years has changed how programs can extract maximum performance out of a CPU. Each CPU now has several cores. These cores present themselves to the system as independent processors sharing the main memory, allowing the system to execute code in parallel. A single thread cannot be executed in parallel by multiple cores. In order to utilize all cores a program must have multiple threads.

It is our goal to allow programmers to utilize multi core hardware while using the `Clean` programming language. Clean currently has no convenient support for multithreading. This convenient support must be able to split work into pieces. The implementation must then be able to create and destroy threads and distribute those pieces of work over those threads.

By implementing the `par` and `pseq` combinators in `Clean`, we let the programmer specify where parallelism could be used. Several variants of `par` and `pseq` are implemented, which have different thread creation schemes.

The current implementation has some limitations. For example, the size of the work to be outsourced is limited, and data must be copied to and from another thread. These limitations hinder the use of the current implementation in data parallelism, where a relatively simple operation is applied to a large dataset.

Using examples we confirmed the strengths and weaknesses of our implementation and the variations on thread creations. In the traditional naive Fibonacci a speedup of three over a non multithreaded version was measured using a quad core system. Using a `pmap` on relatively small lists, the speedup is closer to four. When applied to data parallelism, the overhead increases but still allows some speedup. Compared to combinators in commonly used other languages, the performance is on par. These latter results depend on the inherent efficiency of the language, which in `Cleans` case is very good.

Of the four implemented variants, the `pseq3`, appears to be the most useful. This is the variant that uses a thread pool. While some improvements can be made to it, in our examples it scores very well. Moreover, it is very general and with the proposed improvements in place it will be as general as `pseq1`.

The current implementation only works on 64 bit Linux, but there are no known blocking issues preventing the development of support on other platforms.

Allowing the programmer to specify which graph are reduced in parallel, can be added to `Clean` by implementing a `par` and `pseq` combinator. The current implementation has much room for improvement, but is already usable and offers a decent speedup over a sequential version.

Bibliography

[atomiccompare, 2015] atomiccompare (2015). atomic compare exchange weak, atomic compare exchange strong - cppreference. `http://en.cppreference.com/w/c/atomic/atomic_compare_exchange`. Accessed: 2016-01-10.

[atomicload, 2015] atomicload (2015). cppreference. `http://en.cppreference.com/w/c/atomic/atomic_load`. Accessed: 2016-01-10.

[Beemster et al., 1993] Beemster, M., Hartel, P., Hertzberger, L. O., Hofman, R., Langendoen, K. G., Li, L., Milikowsku, R., Vree, W. G., Barendrregt, H. P., and Muldert, J. (1993). Experience with a clustered parallel reduction machine.

[Blumofe et al., 1995] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995). Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA. ACM.

[Chakravarty et al., 2008] Chakravarty, M. M. T., Leshchinskiy, R., Jones, S. P., and Keller, G. (2008). Partial vectorisation of haskell programs.

[Chakravarty et al., 2007] Chakravarty, M. M. T., Leshchinskiy, R., Jones, S. P., Keller, G., and Marlow, S. (2007). Data parallel haskell: a status report.

[Dagum and Menon, 1998] Dagum, L. and Menon, R. (1998). Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55.

[Danelutto et al., 1998] Danelutto, M., Cosmo, R. D., Leroy, X., and Pelagatti, S. (1998). Parallel functional programming with skeletons: the ocamlp3l experiment. In *In Proceedings of the 1998 ACM Sigplan Workshop on ML*, pages 31–39.

[DaveMcCracken, 2002] DaveMcCracken (2002). *POSIX Threads and the Linux Kernel*.

[DIJKSTRA, 1968] DIJKSTRA, E. (1968). Cooperating sequential processes. Technical report, Technological U., Eindhoven, The Netherlands.

[Groningen, 1990] Groningen, J. V. (1990). Implementing the abc-machine on m680x0 based architectures. Technical report, University of Nijmegen, NL.

[Groningen, 1992] Groningen, J. V. (1992). Some implementation aspects of concurrent clean on distributed memory architectures. Technical report, University of Nijmegen, NL.

[Groningen et al., 1991] Groningen, J. V., Nöcker, E., and Smetsers, S. (1991). Efficient management in the concrete abc machine. Technical report, University of Southampton, UK.

[gtk, 2015] gtk (2015). The gtk+ project. `http://www.gtk.org/`. Accessed: 2016-01-08.

[Hartel et al., 1995] Hartel, P. H., Hofman, R. F. H., Langendoen, K. G., Muller, H. L., Vree, W. G., and Hertzberger, L. O. (1995). A toolkit for parallel functional programming.

[Hoare, 1972] Hoare, C. (1972). Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281.

[itasks, 2015] itasks (2015). Itasks - clean. http://clean.cs.ru.nl/ITasks. Accessed: 2016-01-08.

[Jones et al., 2008] Jones, S. P., Leshchinskiy, R., Keller, G., and Chakravarty, M. M. T. (2008). Harnessing the multicores: Nested data parallelism in haskell.

[Keller and Chakravarty, 1998] Keller, G. and Chakravarty, M. M. (1998). Flattening trees.

[Keller et al., 2012] Keller, G., Chakravarty, M. M., Leshchinskiy, R., Lippmeier, B., and Peyton Jones, S. (2012). Vectorisation avoidance. *SIGPLAN Not.*, 47(12):37–48.

[Kesseler, 1991] Kesseler, M. (1991). Implementing the pabc machine on transputers.

[Kesseler, 1994] Kesseler, M. (1994). Uniqueness and lazy graph copying. copyright for the unique. In *In proceedings of the 6th International Workshop on the Implementation of Functional Languages, University of East Anglia*.

[Marlow et al., 2009] Marlow, S., Jones, S. P., and Singh, S. (2009). Runtime support for multicore haskell.

[Meixner et al., 2006] Meixner, A., Member, S., Sorin, D. J., and Member, S. (2006). Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. In *In Proc. DSN*, pages 73–82.

[messagequeue, 2016] messagequeue (2016). mq_overview(7) - linux man page. http://linux.die.net/man/7/mq_overview. Accessed: 2016-03-12.

[Nöcker et al., 1991a] Nöcker, E., Smetsers, J., van Eekelen, M., and Plasmeijer, M. (1991a). Concurrent clean. In Aarts, E., van Leeuwen, J., and Rem, M., editors, *PARLE '91 Parallel Architectures and Languages Europe*, volume 506 of *Lecture Notes in Computer Science*, pages 202–219. Springer Berlin Heidelberg.

[Nöcker et al., 1991b] Nöcker, E., Smetsers, S., van Eekelen, M., and Plasmeijer, R. (1991b). Concurrent clean.

[Paul Genua, 2004] Paul Genua, P. (2004). A cache primer. http://www.google.nl/url?q=http://www.nxp.com/files/32bit/doc/app_note/AN2663.pdf. Accessed: 2016-03-05.

[Peyton Jones, 2008] Peyton Jones, S. S. (2008). A turorial on parallel and concurrent programming in haskell.

[pipe, 2016] pipe (2016). pipe(7) - linux man page. http://linux.die.net/man/7/pipe. Accessed: 2016-03-12.

[pthreads, 1997] pthreads (1997). pthreads. http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_create.html. Accessed: 2016-01-10.

[semwait, 1997] semwait (1997). semaphores. http://pubs.opengroup.org/onlinepubs/7908799/xsh/sem_wait.html. Accessed: 2016-01-10.

[shm, 2016] shm (2016). shm_open(3) - linux man page. http://linux.die.net/man/3/shm_open. Accessed: 2016-03-12.

[Stroustrup, 2014] Stroustrup, B. (2014). *A Tour of C++*. Pearson Education, Inc.

[Sudkamp, 2005] Sudkamp, T. A. (2005). *Languages and Machines: An Introduction to the Theory of Computer Science (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Trilla and Runciman, 2014] Trilla, J. M. C. and Runciman, C. (2014). An iterative compiler for implicit parallelism.

[Vafeiadis, 2015] Vafeiadis, V. (2015). Formal reasoning about the c11 weak memory model.

[Vafeiadis et al., 2015] Vafeiadis, V., Balabonski, T., Chakraborty, S., Morisset, R., and Zappa Nardelli, F. (2015). Common compiler optimisations are invalid in the c11 memory model and what we can do about it. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 209–220, New York, NY, USA. ACM.

[Vyssotsky et al., 1965] Vyssotsky, V. A., Corbató, F. J., and Graham, R. M. (1965). Structure of the multics supervisor. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, AFIPS '65 (Fall, part I), pages 203–212, New York, NY, USA. ACM.

[Zsók et al., 2011] Zsók, V., Koopman, P., and Plasmeijer, R. (2011). Generic executable semantics for d-clean. *Electron. Notes Theor. Comput. Sci.*, 279(3):85–95.

Appendix A

Results for examples

program	elapsed time (s)	user time (s)	CPU usage	Speedup
fib.out	2.99 s	2.98 s	100 %	1.00
fib pseq1.out	1.30 s	3.39 s	261 %	2.30
fib pseq2.out	1.14 s	2.97 s	261 %	2.62
fib pseq3.out	0.77 s	2.99 s	390 %	3.89
fib pseq4.out	0.88 s	3.40 s	384 %	3.37

Table A.1: Fib test results using a quad core system

program	elapsed time (s)	user time (s)	CPU usage	Speedup
map example.out	8.93 s	8.91 s	100 %	1.00
pmap example pseq1.out	3.29 s	8.72 s	265 %	2.72
pmap example pseq2.out	5.47 s	7.63 s	140 %	1.63
pmap example pseq3.out	3.33 s	8.95 s	269 %	2.68
pmap example pseq4.out	2.53 s	8.71 s	343 %	3.52

Table A.2: Pmap test results using a quad core system

program	elapsed time (s)	user time (s)	CPU usage	Speedup
pseq1 1 thread.out	0.50 s	0.43 s	85 %	1.00
pseq1 2 threads.out	0.48 s	0.50 s	105 %	1.05
pseq1 4 threads.out	0.44 s	0.58 s	132 %	1.15
pseq4 1 thread.out	0.50 s	0.41 s	83 %	1.00
pseq4 2 threads.out	0.49 s	0.49 s	101 %	1.04
pseq4 4 threads.out	0.45 s	0.58 s	127 %	1.10

Table A.3: Mergesort test results using a quad core system sorting 2000000 ints

program	elapsed time (s)	user time (s)	CPU usage	Speedup
tsp.out	7.78 s	7.70 s	99 %	1.00
tsp pseq1.out	4.28 s	7.75 s	181 %	1.81
tsp pseq2.out	5.75 s	7.11 s	124 %	1.35
tsp pseq3.out	3.60 s	7.87 s	219 %	2.16
tsp pseq4.out	2.68 s	7.85 s	294 %	2.91

Table A.4: TSP test results using a quad core system visiting 11 cities

Appendix B

Cross language examples

program	elapsed time (s)	user time (s)	CPU usage	Speedup
c cilk.out	0.79 s	3.10 s	394 %	2.02
c openmp.out	0.95 s	3.24 s	341 %	1.68
cpp async.out	1.59 s	3.94 s	248 %	1.00
cpp packaged task.out	1.48 s	3.47 s	234 %	1.07
clean threadpool.out	0.83 s	3.01 s	363 %	1.92
haskell parseq.out	36.71 s	141.21 s	385 %	0.04
java parallel stream.out	1.00 s	2.79 s	280 %	1.60
java forkjoin.out	2.36 s	8.57 s	363 %	0.68

Table B.1: Cross comparison test results using a quad core system

C

Listing B.1: Source code for cilk.c, using the Cilk library and language extension in C.

```
// to be compiled with: gcc cilk.c -fcilkplus -lcilkrts

#include <stdio.h>
#include <cilk/cilk.h>

int fib(int n) {
  if (n < 2) return 1;
  else return fib(n - 1) + fib(n - 2);
}

int parallel_fib(int n) {
  if (n < 25) return fib(n);
  // Fork off work, possibly to another thread.
  int x = cilk_spawn parallel_fib(n - 1);
  // Compute the other side on this thread while waiting.
  int y = parallel_fib(n - 2);
  // Wait fork the forked work.
  cilk_sync;
  // Compute the result.
  return x+ y;
}

int main() {
  printf("%d", parallel_fib(42));
  return 0;
}
```

Listing B.2: Source code for c_openmp.out, demonstrating the OpenMP compiler directives in C.

```
// to be compiled with: gcc -std=c11 -fopenmp openmp.c

#include <stdlib.h>
#include <stdio.h>

int fib(int n) {
  if (n < 2) return 1;
  else return fib(n - 1) + fib(n - 2);
}

int parallel_fib(int n) {
  if (n < 25) return fib(n);
  int n1, n2;

  // Fork off work, possibly to another thread.
  #pragma omp task shared(n2)
  n2 = parallel_fib(n - 2);

  // Compute the other side on this thread while waiting.
  n1 = parallel_fib(n - 1);

  // Wait fork the forked work and compute final result.
  #pragma omp taskwait
  return n1 + n2;
}

int main() {
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf("%i\n", parallel_fib(42));
    }

  }
}
```

C++

Listing B.3: Source code for cpp_async.out, using the `std::packaged_task` library in `C++`.

```
#include <iostream>
#include <future>

int fib(int n) {
  return n < 2 ? 1 : fib(n-1) + fib(n-2);
}

int fib_async(int n) {
  if (n < 25) return fib(n);
  // Calculate fib n - 1 asynchronously, possibly on anther thread.
  auto fib_n_minus_one = std::async(std::launch::async, fib_async, n-1);
  // Compute the other side on this thread while waiting.
  int fib_n_minus_two = fib_async(n-2);
  return fib_n_minus_two + fib_n_minus_one.get();
}

int main() {
  std::cout << fib_async(42) << std::endl;
}
```

Listing B.4: Source code for cpp_packaged_task.out, showing the `std::async` library in `C++`.

```
// To compile with g++ please use: g++ -std=c++11 packaged_task.cpp -lpthread

#include <iostream>
```

```
#include <future>

int fib(int n) {
  return n < 2 ? 1 : fib(n-1) + fib(n-2);
}

int fib_task(int n) {
  // If n is below a certain threashold we will not use threads.
  if (n < 25) return fib(n);

  // We will calculate fib(n-1) on other thread and fib(n-2) on this thread.

  // Specify our task.
  std::packaged_task<int(int)> fib_n_minus_one_task { fib_task };
  // Set reference to get the result of the task.
  std::future<int> fib_n_minus_one { fib_n_minus_one_task.get_future() };

  // Create a thread, assign its argument and execute the task.
  std::thread task_thread { move(fib_n_minus_one_task), n-1 };
  // Detach so we don't need to join after completion.
  task_thread.detach();

  // Calculate other value on our thread.
  // Wait for the other thread, get its value and return.
  return fib_task(n-2) + fib_n_minus_one.get();
}

int main() {
  std::cout << fib_task(42) << std::endl;
  return 0;
}
```

Listing B.5: Source code for clean_threadpool.out, using the `par` and `poolseq` operators.

```
module fib_pseq3

import StdEnv
import parseq

fib :: !Int → Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

pfib :: !Int → Int
pfib n
| n < 25 = fib n
        = pfib (n-2) par pfib (n-1) pseq3 (+)

Start = pfib 42
```

Haskell

Listing B.6: Source code for haskell_parseq.out, using the `par` and `pseq` operators found in `Haskell`.

```
import Control.Parallel

-- compile with: ghc --make -threaded -rtsopts parseqfib.hs
-- and then run with: ./parseqfib.out +RTS -N8
-- replacing 8 with the desired number of threads.

fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)

```

```
parallelFib n = if n < 25 then fib n else n1 `par` n2 `pseq` r
  where
  n1 = parallelFib (n - 1)
  n2 = parallelFib (n - 2)
  r = n1 + n2

main :: IO ()
main = do putStrLn (show (parallelFib 42))
```

Java

Listing B.7: Source code for java_forkjoin.out, showing the `ForkJoinTask` framework introduced in `Java 7`.

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ExecutionException;

public class ForkJoinFib extends RecursiveTask<Integer> {

  final int n;

  ForkJoinFib(int n) {
    this.n = n;
  }

  @Override
  protected Integer compute() {
    if (n < 25) return fib(n);

    // Create tasks.
    ForkJoinFib n1 = new ForkJoinFib(n - 1);
    ForkJoinFib n2 = new ForkJoinFib(n - 2);

    // Fork off work, possibly to another thread.
    pool.invoke(n1.fork());

    // Compute the other side on this thread while waiting.
    Integer n2result = n2.compute();

    try {
      // Wait fork the forked work and compute final result.
      return n1.get() + n2result;
    } catch (InterruptedException | ExecutionException e) {
      // If anything went wrong, calculate it the safe way.
      return n1.compute() + n2result;
    }
  }

  static final ForkJoinPool pool = new ForkJoinPool(4);

  static int fib(int n) {
    if (n < 2) return 1;
    return fib(n - 1) + fib(n - 2);
  }

  public static void main(String[] args) {
    System.out.println(pool.invoke(new ForkJoinFib(42)));
  }
}
```

Listing B.8: Source code for java_parallel_stream.out, using `Parallel Streams` introduced in `Java 8`.

```
import java.util.stream.Stream;

/**
 * Using a data parallel feature of Java8, the parallel stream.
 */
public class StreamFib {

  private static int fib(int n) {
    if (n < 2) return 1;
    return fib(n - 1) + fib(n - 2);
  }

  private static int parallelFib(int n) {
    if (n < 25) return fib(n);
    else return
      Stream.of(n - 1, n - 2)      // stream of 2 ints
      .parallel()         // stream is parallel
      .mapToInt(StreamFib::parallelFib) // map recursion
      .sum();             // add up results
  }

  public static void main(String[] args) {
    System.out.println(parallelFib(42));
  }
}
```