NTNU
Norwegian University of
Science and Technology

ProXC++ - A CSP-inspired Concurrency Library for Modern C++ with Dynamic Multithreading for Multi-Core Architectures

Edvard Severin Pettersen

Master of Science in Cybernetics and Robotics
Submission date: June 2017
Supervisor: Sverre Hendseth, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Project Description

Create a concurrency framework for C++, based on abstractions of Communicating Sequential Processes (CSP). The focus of this framework should be the utilization of multi-core architectures. Present a design and implementation of such framework, and discuss limitations and uses.

Assignment given: 01. january, 2017
Supervisor: Sverre Hendseth, ITK

Preface

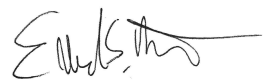
This master thesis presents the results of the project work, affiliated with the course TTK4900 from Department of Engineering Cybertnertics (ITK) at the Norwegian University of Science and Technology (NTNU). The project was carried out during the spring semester in 2017, starting in January and ending in June.

The majority of the work has gone into the implementation of the runtime scheduler, as well as debugging strange heisenbugs. As an afterthought, who would have guessed multi-core systems are easy to debug (spoiler: they are not). Hopefully, this thesis reflects the amount of work gone into the project development.

The reader is expected to basic knowledge within programming, especially concurrent programming. Having experience with a modern dialect of C++, at least C++11 or later, helps as well.

I would like to thank both of my supervisors, Sverre Hendseth and Øyvind Teig, for great feedback and knowledge regarding concurrent system design and paradigms, and for enjoyable and interesting academic evenings at Solsiden.

Edvard Severin Pettersen



Trondheim, 2017-06-04

Abstract

Ever since mass-market processors transitioned from single-core to multi-core architectures, software could no longer rely on an increase in sequential performance for an increase in software performance. Now, developing high-performance software on multi-core architectures requires to exploit the apparent parallelism. Concurrent programming is the main tool for developing such software, but programmers struggle to create correct and scalable concurrent systems.

It is argued in this thesis Communicating Sequential Processes (CSP) is a great model for creating correct and expressive concurrent systems. Further, it is argued combining the parallel nature of CSP with a dynamic multithreaded runtime system sets the foundation for creating high-performant and scalable software for multi-core architectures.

This thesis details the development of ProXC++ – a CSP-influenced concurrency library for modern C++, which is built around dynamic multithreading. Dynamic multithreading is implemented as a collection of lightweight processes cooperatively scheduled on multiple schedulers. The runtime system follows the hybrid threading model, where processes are implemented as user-threads, and each runtime scheduler runs on its own kernel-thread. Runtime schedulers employ randomized work stealing for load balancing ready processes to idle schedulers.

A detailed design and implementation of ProXC++ is presented, with focus on dynamic multithreading. New and existing algorithms are described, mostly for how process management, inter-process communication and synchronization is administered by the runtime schedulers. A series of benchmarks with various degrees of parallelism is performed. ProXC++ yields promising performance results, however some issues with the work stealing algorithm are highlighted and discussed.

ProXC++ is concluded as a successful project, providing expressive and correct abstractions for creating concurrent programs and is able to exploit the parallelism in multi-core architectures. Some potential candidates for future work is outlined, including implementing support for networking.

The ProXC++ library is publicly released as an open-source project with an MIT license, available free of charge on GitHub.

Sammendrag

Helt siden konsumerprosessorer gikk ifra enkjernet til flerkjernet prosessorarkitekturer kunne ikke lenger programvare være avhengig av økning i ytelse øke med den sekvensielle ytelse for prosessorer. Høy ytelses programvare på flerkjernet arkitekturer krever nå å utnytte den åpenbare parallelismen. Samtidig programmering er hovedverktøyet for å utvikle slike programvarer, men programmerere sliter med å skrive korrekt og skalerbart samtidighets-systemer.

Denne avhandlingen argumenterer at Communicating Sequential Processes (CSP) er en bra model for å lage korrekte samtidighets-systemer som har kraftig uttrykkskraft. Videre så argumenteres det at ved å kombinere den parallele naturen i CSP med et dynamisk multitrådet kjøresystem kan lage et solid fundament for høy ytelses og skalerbare programvare for flerkjernet arkitekturer.

I denne avhandlingen presenteres arbeidet som er gjort på ProXC++ – en CSP-insperert samtidighetsbibliotek for moderne C++, som bygges på dynamisk multitråding. Dynamisk multitråding implementeres med lettvektsprosesser som bruker sammarbeidende kjøring planlagt av flere planleggere. Kjøresystemet følger en hybrid trådmodel, hvor prosesser er implementert som brukertråder, og hver planlegger kjører på sin egen kjernetråd. Planleggerne bruker randomisert arbeidstjeling for å balansere arbeid mellom ledige planleggere.

Et detaljert design og implementasjon av ProXC++ er presentert, hvor fokuset er på dynamisk multitråding. Nye og eksisterende algoritmer er forklart, hovedsaklig på hvordan prosesstyring, inter-prosesskommunikasjon og synkronisering er forvaltet av planleggerne. En rekke tester med varierende grad av parallelisme er utført. ProXC++ gir lovende resulterer, men en del problemer med arbeidsstjelingsalgoritmen er fremhevet og diskutert.

ProXC++ er konkludert med å være et vellykket prosjekt, basert på dens uttrykkskraft og korrekte abstraksjoner for å lage samtidighets-systemer og at den klarer å utnytte parallelismen i flerkjernet arkitekturer.

Biblioteket ProXC++ er publisert some åpen kildekode med MIT lisens, og er tilgjengelig gratis på GitHub.

Contents

Chapter 1

Introduction

Computers are for many the greatest engineering feat in the 20th century, taking into consideration the vast complexity, precision, and knowledge it has required to reach the state of computers we have today. Since its inception with the transistor in the 40s, the computer has been subjected to numerous changes in different technology to further increase the performance. Such technology includes the decrease in size of transistors down to nanometers, faster clock rates for processors, and most notably the transition from single-core to multi-core architectures.

When entering the 21st century, the main drive behind increasing processor performance was increasing the transistor count in the processor, following the exponential growth described by Moore's law for decades. However, since 2005 this strategy was no longer sustainable. The physical limitations of the Dennard scaling were starting to show, and a shift from single-core towards multi-core architectures was a response to this limitation. This shift is, of course, an oversimplification of the issue, but multi-core architectures are pretty much the norm among desktop processors in this day and age.

With the transition to multi-core architectures, software can no longer naïvely rely on increase in processor performance for an increase in software performance. Software has to now exploit the parallelism in multi-core processors, which requires concurrent programming. Concurrent programming has long existed before multi-core processors, but is now the main tool for developers to write scalable software

which can utilize the parallel power in multi-core architectures.

Concurrent programming revolves around having multiple threads of execution running concurrently (simultaneously) in a program. Such programs are called concurrent systems. Even though concurrent programming is a powerful and expressive paradigm to write software in, it also is difficult to reason whether a concurrent system behaves as specified by the programmer. What makes it hard to write correct and well-behaved concurrent systems has to do with concurrency adding significant complexity to the system. Additionally, it is an added mental overhead for the programmer. Even the simplest and most subtle errors in a program can explode into the most obscure and hard-to-track bugs. Being able to write expressive concurrent systems, as well as being able to reason about the correctness of the system, are probably the biggest challenges with concurrency.

Communicating Sequential Processes (CSP) was an effort by Hoare [1] to harness the expressiveness of concurrent programming while being able to prove the correctness of such models. CSP is by its own a formal language used to describe concurrent models. These models described by CSP is a parallel composition of sequential processes, which only communicates through mutually agreed message-passing constructs. This inherently inhibits any types of race-conditions with shared memory to ever occur. However, the real power of CSP comes from the ability to reason about the correctness of these models, such as the absence of deadlocks and livelocks.

Several programming languages has incorporated CSP formalism into the language, such as occam-π [2], XC [3] and Go [4], but the mainstream popularity has not been great; Go is an exception, which ranks as one of the most popular languages to date. As a response, a collection of transpilers[1] and libraries has been created for more popular and well-established programming languages, such as C [5], C++ [6, 7, 8], Python [9], and Java [10].

Concurrent systems written in CSP has a great potential for exploiting the parallelism on multi-core architectures, as CSP is inherently parallel. CSP also has the added benefit of providing a model which can be reasoned about the correctness

[1] Source-to-source compiler, compiling source code written in one language to source code in another language

of the behaviour. However, not many CSP-based frameworks takes advantage of the potential parallelism in multi-core architectures. Existing libraries are either outdated for modern use, or does not satisfy performance wise. Chapter 3 takes this discussion further and argues there is a need for a modern solution.

With the lack of a modern CSP library for C, ProXC was developed as a result of the work done by Pettersen [5]. ProXC aimed to show the possibilities such a library could have. The focus of ProXC was however on the provided abstractions rather on performance, and support for multi-core architectures was not implemented. In this thesis, the work done on ProXC is continued with the aim to provide a portable CSP library for multi-core architectures.

This thesis presents the work and results of ProXC++, a portable CSP library for modern C++ with support for multi-core architectures. ProXC++ is a concurrency library which uses dynamic multithreading combined with lightweight processes to achieve proper parallel computing on multi-core architectures. The runtime system employed by ProXC++ consists of a number of schedulers equal to the number of available logical processor cores. Each scheduler is responsible for scheduling and running these lightweight processes on a kernel-thread, and uses work stealing to load balance work between schedulers.

1.1 Project Status of ProXC

As this thesis is a continuation of the work done in the project thesis by Pettersen [5], a short status report of ProXC the project is presented.

As of writing this thesis, ProXC has not undergone any major development, except for some bug fixes. The API has remained unchanged, as well as any of the major issues pinpointed by the project thesis.

1.2 Thesis Structure

The thesis is structured into three parts. Part I discusses the theoretical basis of this thesis, as well as arguing the motivation behind CSP with multi-core support: Chapter 2 gives an introduction to concurrent programming, and details all relevant

theoretical knowledge for which this thesis is based on. Chapter 3 argues there is a motivation for a CSP library with multi-core support, and provides a summary of existing solutions and how they work.

Part II details the work regarding ProXC++: Chapter 4 details the library specification. Chapter 5 and Chapter 6 presents respectively the design and implementation. Chapter 7 presents examples of how ProXC++ is used along with code examples. Chapter 8 performs a benchmark of ProXC++ compared to existing solutions, both highlighting the difference between single-core and multi-core implementations.

Part III discusses different aspects of CSP and multi-core, limitation and uses of ProXC++, and concludes the thesis: Chapter 9 compares both projects, drawing the differences and similarities between the multi-core and single-core library. Chapter 10 explains the challenges with implementing a CSP framework with multi-core support. Chapter 11 discusses the limitations and uses of ProXC++. Chapter 12 lists a set of potential future work. Chapter 13 draws a conclusion for the thesis.

Lastly, the appendices list the used acronyms (appendix A), and a complete listing of all benchmarks tested in Chapter 8 (appendix B). The last section in the thesis is the list of references.

Part I

Preliminaries

Chapter 2

Theoretical Background

This chapter gives a more in-depth explanation of different topics mentioned in the introduction, as well as other topics relevant to the thesis. These topics cover the required background knowledge, and each topic is presented on its own. The reader is encouraged to skim over this chapter, and rather come back and read more thoroughly when coming across a topic later in the thesis.

2.1 Concurrent Programming

Concurrent programming, or concurrent computing, is a form of computing to express programs or systems which execute multiple sequential computations in interleaving time periods, giving the impression of simultaneous execution. These computations are said to be running *concurrently*, compared to *sequentially* (one completing before the next start). These individual computations are often called *processes*, *tasks* or *threads*, which indicate a separate execution point. Do not confuse the term *process* in the context of concurrent programming with an individual program running in the context of an operating system.

The notion of concurrency stems from the limitations of sequential programs and how all programs, in the end, are translated to machine code. Given the program is executed on a uniprocessor, only one machine code instruction will be

executed at any given moment[2]. Since all computations in a sequential program must be executed sequential, it can be unintuitive how to model and implement systems which are inherently parallel. Concurrency aims to provide an abstraction level to bridge this limitation. Concurrent systems are therefore much more expressive than sequential systems, since it does not matter whether the program is executed in parallel or not, e.g. on a multiprocessor or uniprocessor.

It is important to note that concurrent programming is not the same as parallel programming. Concurrency is a form of abstraction, disregarding how the actual program execution is achieved. Parallelism refers to, in contrast to concurrency, the condition of a program being executed on multiple processor cores at once. One could therefore say that concurrency is possible on both uniprocessors and multiprocessors, while parallelism is only possible on multiprocessors.

Concurrency is a great tool for programmers, allowing concurrent systems to be expressed in a much more intuitive manner. It does however introduce an added mental overhead for the programmer, and is much more error-prone compared to other types of programming paradigms. As a result, many programmers resort to other, less error-prone programming paradigms than concurrent programming [11].

2.1.1 Threading Models

Threading is the foundation of concurrency, which allows multiple computations to be executed simultaneously. Some sort of threading mechanism must be implemented on a platform to provide concurrency. On *operating systems* (OS), threading mechanisms falls mainly into three types of threading models: *user-threads*, *kernel-threads*, or *hybrid-threads*, which is a combination of the two first models. Brown [7, sec. 1] goes into further detail on the three models, and a short summary is presented below.

[2]Machine code execution is vastly more complex than presented here, e.g. pipelining and instruction-level parallelism (ILP), but the sequential nature of program execution still stands

User-Threading

User-threads are a cooperative scheduling of threads executed in user space[3], and is called a `M:1` threading model. `M:1` means running `M` user-threads on a single kernel-thread. These user-threads must cooperate on scheduling as the scheduling is non-preemptive, meaning a running task is executed until completion or yields. Context switching and scheduling between these user-threads is happening unbeknownst to the OS, resulting in much faster context switch times. Consequently, the OS cannot however help with scheduling, and system blocking calls in any user-thread blocks all user-threads on the given kernel-thread. Only one user-thread can run on a kernel-thread at any given moment.

Kernel-Threading

Kernel-threads are often directly supported in OS kernels and is called a `1:1` threading model. `1:1` means scheduling a kernel-thread onto an available processor core of the system. Kernel-threads often use preemptive scheduling, meaning threads are given a priority, and the running thread is interrupted and later resumed whenever a thread with higher priority is ready. The OS is responsible for scheduling said threads. Kernel-threads has no problems with blocking calls, as the OS can schedule any other kernel-thread during a blocking call. Since the OS has full control over the scheduling, it can much better utilize the available processor resources and time usage for each thread, compared to user-threading. Context switching is however much slower than user-threads because of overhead and kernel-space[4] crossing.

Hybrid-Threading

Hybrid-threads is a combination of user-threads and kernel-threads, and is called a `M:N` threading model. `M:N` means running `M` user-threads over `N` kernel-threads. In other words, hybrid-threading runs multiple kernel-threads, with each kernel-thread running multiple user-threads. Blocking calls can still cause issues with

[3]Regarding operating systems, user space is a set of locations where normal user processes run

[4]Regarding operating systems, kernel space is the location where the code of the kernel is stored and executed

unnecessarily blocking other user-threads, but is possible to mitigate with running the blocking user-thread on its own kernel-thread. Scheduling user-threads among the kernel-threads can be much more difficult compared to user-threading, as the OS cannot help with utilizing the available processor resources.

2.1.2 Concurrency Concepts and Primitives

Having multiple sequential executions running concurrently is not very useful if they cannot cooperate. Some form of interaction or communication between the computations must exist, which in turn requires coordination of access to shared resources. This coordination is called *concurrency control*. Concurrency control means ensuring the correct and intended result from interactions between concurrent processes are upheld.

To be able to manipulate shared resources safely requires introducing a couple of new primitives and concepts. Below is a non-exhaustive list of prevalent concurrency primitives presented.

Atomic Operations

An operation is said to be *atomic* or *linearizable* if it appears to the rest of the system as instantaneous. In concurrent systems, multiple processes can access the same shared resource at the same time. If one of the processes are changing the contents of a shared resource while another process is using the same resource, it is possible the operation results in an invalid or undefined state. It is obvious such situations requires atomic operations to force a linear sequence of well-defined observable operations.

Atomic operations exist both as low-level and high-level primitives. At the bottom we have processor instructions which are used to manipulate memory atomically. These usually include *atomic read / write*, *atomic swap*, *test-and-set*, *fetch-and-add*, *compare-and-swap*, and *load-link / store-conditional*. Modern processors usually support these types of instructions [12].

Further, these instructions are used to implement higher level primitives and non-blocking algorithms. Examples of higher level primitive include semaphores,

mutual exclusion locks, and monitors.

Critical Sections

Concurrent access to shared resources can result in an invalid or undefined state, as stated above. Regions of code of which concurrent access by multiple processes were to cause erroneous behaviour are called *critical sections* or *critical regions*. These regions must therefore be protected by some sort of synchronization or lock mechanism. Critical sections usually access shared resources such as a data structures, IO operations or network sockets, where multiple concurrent access would result in incorrect behaviour [13].

Consider the following program: two processes, `A` and `B`, tries to respectively increment and decrement the shared resource `count` by one, initial value of 0. See Listing 2.1 for reference.

Process A

```
1  // Critical section
2  count = count + 1
```

Process B

```
1  // Critical section
2  count = count - 1
```

Listing 2.1: Example of a critical section causing a race condition.

If both process `A` and `B` are executing the critical section at the same time, it is possible for the `count` resource to equal some whole number instead of the expected result 0. This comes from the fact the incrementing and decrementing operations are not atomic operations, but a three-stage operation of read-modify-write.

If process `A` starts executing the increment, but is preempted before the write stage, process `B` can theoretically complete an arbitrary amount of decrement operations before process `A` is resumed. When process `A` is resumed, the old modified value of `count` is now written back instead of the updated value. This illustration is a classic example of a *race condition*, meaning the order and timing of which the operation is completed determines the outcome.

To achieve correct behaviour the critical section must be protected, usually done through some form of mutual exclusion. Consequently, mutual exclusion makes the critical section an atomic operation, as the read-modify-write operation is made

indivisible to other processes.

Semaphores

Semaphore is in software terms a data structure used to control access to a shared resource between multiple processes and to synchronize between processes. Invented by Dijkstra [14], and is one the simplest concurrency primitives used to build higher level concurrency control structures.

As described in Downey [15, chapter 2], a semaphore is like an integer with three differences:

1. When you create the semaphore, you can initialize its value to any integer, but after that the only operations you are allowed to perform are increment (increase by one) and decrement (decrease by one). You cannot read the current value of the semaphore.
2. When a thread decrements the semaphore, if the result is negative, the thread blocks itself and cannot continue until another thread increments the semaphore.
3. When a thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked.

Blocking in this sense means the scheduler will suspend the blocking thread until a corresponding event or operation which causes the thread to be unblocked. Both the increment and decrement operations are atomic, meaning multiple processes can concurrently access a semaphore.

Semaphores usually comes in two flavours: *counting* and *binary* semaphore. A counting semaphore allows an arbitrary resource count, while a binary semaphore allows only a resource count of 0 and 1 (hence the name binary).

Semaphores are incredibly simple by definition, and are therefore often used to create more complex concurrency synchronization and structures. This includes structures such as *locks*, *monitors* and other *synchronization patterns*. See Downey [15, chap. 3-7] for a more complete overview of such constructs.

Mutual Exclusion (Mutex)

Mutual exclusion, *mutex* for short, is used for protecting critical regions from concurrent access and to prevent race conditions. One can view mutexes as binary semaphores, with one important difference: mutexes has a notion of ownership; only the process which was successful in acquiring the mutex can release it.

Looking back at the example in Listing 2.1, wrapping a mutex around the critical section hinders process `A` and `B` of accessing the shared resource `count` simultaneously, and effectively turns the increment and decrement operation into an atomic operation. See Listing 2.2 for reference.

Process A

```
1  mutex.wait()
2  // Critical section
3  count = count + 1
4  mutex.signal()
```

Process B

```
1  mutex.wait()
2  // Critical section
3  count = count - 1
4  mutex.signal()
```

Listing 2.2: A critical section surrounded by mutual exclusion, removing the race condition.

Note that using mutual exclusion is very error-prone. Mutual exclusion might have unwanted side-effects and conditions such as deadlocks, starvation and priority inversion, which are explained in further detail in Subsection 2.1.3.

2.1.3 Common Pitfalls

As a consequence of using concurrency control, common pitfalls and potential problems such as *race conditions*, *deadlocks*, *livelocks*, *starvation*, and *priority inversion* must be taken into consideration. These are unwanted conditions or program states which are caused by concurrent operations and interactions resulting in an erroneous state.

Race Condition

As shown in Listing 2.1, a race condition occurs when the output of an operation is dependent on the timing and sequence of other processes. Race conditions is never an intended behaviour of a program, as it is deemed undefined behaviour.

Undefined behaviour comes from the fact that the result of a race condition is the non-deterministic result of timing between threads.

Deadlock

Deadlock is a state in which a group of processes are each waiting for other members of the same group to release a lock, effectively halting progression. A deadlock is a direct consequence of enforcing mutual exclusion in critical sections.

As described in Silberschatz et al. [16, pp. 239], a deadlock can only occur in a system if and only if all four of the *Coffman conditions* [17, pp. 70] are held simultaneously. The conditions are as follows:

1. **Mutual exclusion** – the resources involved must be unshareable.
2. **Hold and wait** – a process holding at least one resource and requesting additional resources.
3. **No preemption** – a resource can only be voluntarily released by the process holding it.
4. **Circular wait** – each process must be waiting for a resource held by another process, which in turn is waiting for the first process to release the resource.

There are several ways to handle deadlocks, but the three main approaches are *ignoring*, *detection* and *prevention*. Ignoring is simply to assume a deadlock never occurs. Detection allows deadlocks to occur. If a deadlock is detected, the deadlock symmetry is broken by either terminating one or more of the deadlocked processes, or involuntarily preempt one or more of the deadlocked resources. Prevention is to simply prevent one of the four Coffman conditions from ever occurring.

The simplest example of a deadlock is the following scenario: two processes, `A` and `B`, tries to acquire two mutexes, `mutex_A` and `mutex_B`. Both processes acquire the mutexes in opposite order. A deadlock occurs when either process acquires its first lock, e.g. process `A` acquires `mutex_A`, and is subsequently preempted. The other process is resumed and acquires its first mutex, e.g. process `B` acquires `mutex_B`. Now, when both processes attempts to acquire its second mutex, both will promptly block forever, as both processes require each other to release the mutex. A deadlock has now occurred. See Listing 2.3 for reference.

Process A

```
1  mutex_A.wait()
2  mutex_B.wait()
3  // some work
4  mutex_B.signal()
5  mutex_A.signal()
```

Process B

```
1  mutex_B.wait()
2  mutex_A.wait()
3  // some work
4  mutex_A.signal()
5  mutex_B.signal()
```

Listing 2.3: Example of a simple deadlock between two processes.

Livelock

Livelock is similar to deadlock, as it is a state in which processes are not blocked, but are refrained from making progression. A livelock can occur when e.g. processes are too busy responding to each other to resume work.

Livelock is a rarer condition than deadlock, and can be harder to detect as the processes are not blocked when livelocked.

Starvation

Starvation is a condition where a process is denied further progress by perpetually being denied access to required resources or denied running time by higher priority processes.

The absence of starvation in concurrent algorithms is called *liveness*, which is a property guaranteeing all processes are able to make progress within a finite time.

Priority Inversion

Priority inversion is a situation in scheduling where a higher priority process is indirectly preempted by a lower priority process, usually because of mutual exclusion.

Consider the following example: process `H`, `M` and `L` has the priority ordering `p(H) > p(M) > p(L)`, and `H` and `L` both try to acquire the resource `R`. If `L` acquires the resource and is promptly preempted by `H`, `H` becomes blocked until `L` releases the resource. It is now possible for `M` to run over `L`, because of higher priority. Effectively `H` cannot run as `L` cannot release the resource, since `M` is preempting `L`. This is called *priority inversion*. See Figure 2.1 for reference.

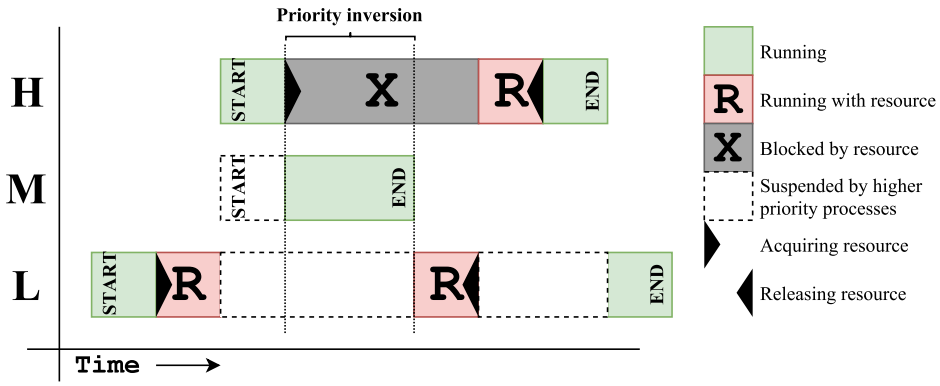


Figure 2.1: Example of priority inversion with three processes.

Bounded priority inversion is when it can be proven priority inversion only occurs for a finite amount of time, while *unbounded priority inversion* cannot prove this.

2.2 Communicating Sequential Processes

Communicating sequential processes (CSP) is a formal mathematical language used in computer science to describe and model concurrent systems. First introduced by Hoare [1], and was originally described as a parallel composition of sequential disjoint processes with primitives for input and output, combined with guarded commands [18]. Communication was solely through message-passing, which permitted synchronized communication between named processes.

Since its inception, CSP has undergone numerous transformations. As Abdallah et al. [19] explains, most of the subsequent research has focused on a process algebra known as *Theoretical CSP* (TCSP), which suppresses the imperative aspect of CSP [20].

The strong points of CSP and TCSP are the ability to reason about the correctness of the system being modelled. Since all communication between processes are limited to message-passing, no primitive race conditions such as memory operations cannot occur. Further on, properties of the CSP models can be reasoned about, such as liveness, safety and deadlock.

In the industry, CSP is used to specify and verify the correctness of concurrent systems, especially communication and security protocols. The most prominent tool used is the *failure-divergence refinement* (FDR) checker [21]. FDR can statically prove if a concurrent system refines a given specification, has the correct liveness and safety properties, as well as the absence of deadlocks and divergence. It is obvious such tools are powerful for systems that must prove its correctness before being deployed.

The ideas and expressiveness of CSP models has influenced the creation of concurrent programming languages, most notable occam [22], Ada [23], XC [3] and Go [4]. CSP frameworks has also been developed for programming languages which does not natively support CSP influenced concurrency, such as ProXC [5] for C, C++CSP2 [7] and C++CSP [8] for C++, JCSP [10] for Java, PyCSP [9] for Python, and much more.

It is understandable CSP is a great framework for concurrent systems. It provides an expressive and correct framework for modelling and implementing such systems, and allows to prove the correctness of said implementation. This alleviates some of the mental overhead, as well as reducing the error-prone nature of concurrent programming.

2.3 Memory Ordering

Out of all CPU operations, memory accesses are among the slowest. Following Moore's law, CPU instruction performance has increased at a much greater rate than memory performance. Multilevel caches has been used to a great extent to bridge this gap in performance, however, utilization of said caches could be improved. This is where *memory reordering* comes in. To increase the performance of memory access further and properly utilize the hardware parallelism, memory operations can complete out of order [24].

Several types of memory-consistency models exist. *Sequential consistency* is a guarantee that all processes agree on the order memory operations occur, even if they are completed out of order. The easiest configuration for sequential consistency is running all processes on a uniprocessor. On multi-core architectures, how-

ever, memory reordering can create inconsistencies between processes and proper care must therefore be taken to enforce correct memory ordering.

There are different types of memory reordering, which usually falls into the categories of *compiler reordering* and *processor reordering*. As the name implies, compiler reordering is memory reordering done by the compiler at compile time, while processor reordering is memory reordering done by the processor at runtime.

Compiler reordering is obviously different from compiler to compiler, as it is implementation-defined for each compiler. Processor reordering is also surprisingly different from CPU to CPU architecture. A *memory model* is therefore used to describe what types of memory ordering to expect at runtime for a given processor architecture, which again falls into the two categories *weak* and *strong* memory models.

As Preshing [25] explains it, a weak memory model can expect all types of memory reordering. This means any load or store operation can be reordered with any other load or store operation, and can ne reordered by both the compiler or processor. In contrast, a strong memory model limits the types of memory reordering. More specifically, only `store-load` reordering is permitted.

To enforce correct memory ordering, some type of *memory barrier* must be used [26, 27]. These exists as both software and hardware semantics, and are generally implemented as some sort of *acquire* and *release* semantics [28].

Memory ordering is important to take into consideration when creating non-blocking algorithms, as the memory operations in critical regions must have a sequential consistency between processors. More of this explained in Section 2.4.

2.4 Non-Blocking Algorithms

To fully utilize multi-core processors, most programs are multiprogrammed. Preemptive multiprocessor operating systems has the unfortunate effect of degrading performance in synchronized parallel programs if the preemption is ill-timed, and access to shared data structures is usually the root of the cause. As Michael and Scott [29] explains, these data structures need to ensure concurrent access is consistent and well-formed across processes, which is usually implemented by protecting

critical sections with mutual exclusion. Mutual exclusion locks scales poorly in performance on time-sliced multiprogrammed systems [30], due to preemption of processes holding locks. The preempted process must be rescheduled and release the lock before any waiting processes can progress.

One principal strategy to mitigate ill-timed preemptions are *non-blocking algorithms*. Non-blocking algorithms have the property any process cannot cause failure nor suspension of any other process, despite failures or suspension of itself. Two types of guarantees are usually associated with nonblocking algorithms to describe how strong this property is: *lock-free* and *wait-free*.

Lock-freedom guarantees system-wide progress, and can be defined as follows: given a meaningful definition of progress, an algorithm is lock-free if at least one process in a program makes progress if all processes are given sufficient running time.

Wait-freedom guarantees process-wide progress, and can be defined as follows: an algorithm is wait-free if every operation in the algorithm has an upper bound of computational time before it completes. Wait-freedom implies lock-freedom, which makes wait-freedom a stronger guarantee than lock-freedom.

Preshing [27] describes the *lock* part of non-blocking algorithms (or as lock-free programming he refers it to) as "the possibility of 'locking up' the entire application in some way, whether it's deadlock, livelock – or even due to hypothetical thread scheduling decisions made by your worst enemy". This means code not containing any locks, such as mutexes, may still not be lock-free.

It is rare a concurrent program is entirely lock-free or wait-free, which is why the focus is on non-blocking algorithms rather than non-blocking programs. Generally, a set of high contention operations between processes are set out to be non-blocking, such as access and manipulation to shared data structures.

A handful of techniques are used to implement non-blocking algorithms, most commonly a combination of atomic operations, memory barriers and general patterns. While the atomic primitives `read-modify-write` and `compare-and-swap` forms the basis of most non-blocking algorithms, memory models and memory ordering needs to be taken into consideration as well. Since different processors have different memory models, it is not given processes agree on the order in which

memory operations occur. Memory reordering can cause memory inconsistencies between processes, which is why sequential consistency or memory barriers are needed, especially on multi-core architectures [25, 26, 27, 28].

2.5 Dynamic Multithreading

Exploiting the potential parallelism on multi-core architectures requires fully utilizing the available logical processor cores. Amdahl's law [31] argues that the maximum potential speedup in a program with infinite number of parallel processors is limited to $^1/_s$, given the fraction of sequential work in the program is s. Because of Ahmdal's law, multiprogrammed concurrent programs needs to identify the potential parallelism and convert sequential work into parallel work to exploit the parallelism on multi-core architectures. This responsibility is mostly up to the programmer to identify, and in most cases is easier said than done. Simple constructs such as loops and loosely coupled sections are easy to detect and convert [32, 33], but a program-wide parallelism is far from intuitive to detect and how to schedule such parallelism effectively on multi-core architectures.

Dynamic multithreading is a strategy to abstract away the scheduling and instead focus on detecting the units of parallel work in the program. The idea is letting the program spawn and synchronize new processes or tasks which execute some kind of computation, and some sort of runtime takes care of the scheduling.

Based on the threading models (see Subsection 2.1.1) the naïve approach would be spawning kernel-threads for each process. The operating system is responsible for creating, scheduling and destroying the processes, as well as scheduling the processes among the available processor cores. However, kernel-threads are expensive to create, context switching between kernel-threads are relatively slow, and scales very badly in performance with an increase in kernel-threads running simultaneously. This severely limits the use of processes, and makes it very unintuitive on how to distribute the parallel work efficiently.

Using hybrid-threads, a combination of user-threads running on multiple kernel-threads, is an improvement on only kernel-threads. Each spawned process is a user-thread, scheduled on a static number of kernel-threads equal to the number

of available cores. User-threads are much cheaper to create, context switching is relatively much faster, and scales much better performance-wise. The problem is how to efficiently schedule and distribute said user-threads between the available kernel-threads, since an intermediate runtime environment has to exist between the user-threads and the OS.

Two principal scheduling strategies exist to efficiently distribute and schedule processes among a static number of kernel-threads, namely *work stealing* and *work sharing*.

2.5.1 Work Stealing

Given a processor with P logical cores, P kernel-threads are created with each running a scheduler. Each scheduler has a pool of ready work. Whenever a schedule runs out of ready work, it selects another scheduler (called victim) and tries to steal work. If the steal was successful, it continues the stolen work. Spawning a process simply pushes the spawned process to the ready pool [34].

Different types of work stealing algorithms exist, but the main emphasis is on the victim selection and stealing procedure. The most popular algorithm is the randomized work stealing algorithm first detailed in Blumofe and Leiserson [34]. Whenever a scheduler is empty for ready work, a victim is chosen uniformly at random. If the victims ready pool is not empty, try to steal work. If successful, resume said work, else start over the selection procedure.

The most common way to represent ready work is by using a double-ended queue, or *deque* for short, where the ends of the deque are called *top* and *bottom*. The scheduler owning the deque has access to the bottom of the deque, where it can push and pop ready work. Any other schedulers can try to steal from the top of the deque if it is non-empty. Much research has gone into creating these deques as efficient and low overhead as possible [35, 36], as they can be high contention area between kernel-threads.

Many runtime frameworks for dynamic parallel computations has been created with work stealing [37, 38], and with great success performance wise.

2.5.2 Work Sharing

As the same setup as work stealing, P kernel-threads are created on a processor with P logical cores with each running a scheduler. The big contrast to work stealing is work sharing manually distributes each work among the schedulers whenever new work is spawned.

Usually work stealing is preferred over work sharing, as work stealing causes less process migration between schedulers. As Blumofe and Leiserson [34, pp. 721] argues, "Intuitively, the migration of threads occurs less frequently with work stealing than with work sharing, since when all processors have work to do, no threads are migrated by a work stealing scheduler, but threads are always migrated by a work sharing scheduler".

Chapter 3

CSP + Multi-Core = True ?

The motivation described in the introduction is further elaborated in this chapter on why a CSP framework with multi-core support would be desirable. Further, a review of existing solutions of CSP programming languages and libraries with multi-core support is presented and how they work.

3.1 Motivation behind CSP and Multi-Core

As mentioned in the introduction, software can no longer rely on an increase in processor performance for an increase in software performance. High-performance software aiming to utilize the full potential of multi-core architectures needs to do two things: identify the parallel units of work in a program, and efficiently run the parallel work using the available processor resources.

Why is it important to identify parallel units of work in a program? The short answer is Amdalh's law [31]. Amdalh's law argues the maximum speedup in a program with infinite number of parallel processors is limited to $^1/_s$, given the fraction of sequential work in the program is s. If a program has a fraction of 50% sequential work the maximum speedup possible is 2, and a fraction of 25% sequential work yields a maximum speedup possible of 4. There exists more refined models of Amdalh's law developed for different multi-core architectures [39], but the point is the same; parallelism is key.

If a programmer cannot reduce the fraction of sequential work, or in other words cannot increase the fraction of parallel work, there is not much to gain from multi-core architectures. Increasing the fraction of parallel work usually consists of identifying parallelism in sequential work, and subsequently converting said work to parallel units of work. Identifying parallelism is usually the easiest part, e.g. loops and decoupled sections of code are easy to identify and argue whether it is parallel or not. How to convert sequential work to parallel work is however another matter, requiring details of scheduling and synchronization.

When parallel work has been identified, how do you efficiently schedule and run the parallel work on the available processor resources? When you want the program to be scalable for all multi-core architectures, simply hard coding for a given architecture will not suffice. Dynamic multithreading is a solution, which has strategies for dynamically distribute parallel work among available logical processor cores. Strategies include work stealing and work sharing, but the core philosophy is letting the dynamic multithreading take care of scheduling and synchronization, while the programmer specifies what the parallel work is. Whether dynamic multithreading is implemented with the program or as part of a runtime system underneath, achieves the same goal.

This thesis argues CSP is a good candidate for creating high-performance software for multi-core architectures. First of all, CSP provides expressive and correct abstractions for creating concurrent systems, which also can be statically reasoned whether certain specifications and safety properties are met. Secondly, CSP inherently defines its parallel units of work, as all CSP models are a parallel composition of sequential processes. This inherent parallel nature of CSP essentially identifies all higher level units of parallel work in a program. Now, if a dynamic multithreading scheme were to be employed together with a CSP framework, it would enable exploiting parallelism in multi-core architectures.

3.2 Existing Solutions of CSP and Multi-Core

Combining CSP with dynamic multithreading is no new invention. Even though dynamic multithreading has usually not been the focus with CSP frameworks,

multiple frameworks do exist with dynamic multithreading, including programming languages and programming libraries. These frameworks vary in availability and age, ranging from proprietary programming languages to open-source libraries. Below is a non-exhaustive list of dynamic multithreaded CSP frameworks summarized.

3.2.1 Programming Languages

The minority of programming languages have made concurrency a native part of the language design, yet alone CSP-based concurrency. For those languages with CSP as a native influence, dynamic multithreading has usually not been the focus. The first CSP-based languages were designed for microprocessor hardware, and with the years transitioned to general-purpose programming languages. Below are the three most influential CSP-based programming languages with dynamic multithreading presented.

occam on the transputer

Occam, a concurrent programming language, was the first programming language to build on the CSP process algebra. It was developed by INMOS [40] and first appeared in 1983. The main motive behind occam was to develop a concurrent language to run on the transputer microprocessor [41], also developed by INMOS.

The transputer microprocessor was the first general purpose microprocessor designed for parallel computing, allowing multiple transputers to easily be connected without the requirement for a complex communication bus. Running on a cluster of transputer microprocessors, occam programs allowed for concurrent systems to be running as true parallel computing.

Sadly, occam was initially only for proprietary use, as support for occam was limited to the transputer microprocessor architecture. Multiple implementations of later occam versions and dialects has been made for more general purpose computers, but none has had support for true parallel computing in mind.

XC on the XCore

XC is a concurrent programming language, developed for real-time embedded parallel computing [42]. XC targets the XCore processor architecture [43], both developed by XMOS. XC as a programming language is based on C with custom syntax extensions and restrictions. Concurrency primitives are a native design of XC, providing features which corresponds to the architectural resources on the XCore.

The XCore processor architecture is an embedded multi-core architecture, aimed to be used for parallel computing. An XCore processor supports concurrent execution of up to eight threads, with native support for inter-thread and inter-processor communication and thread scheduling. The main selling point of XCore is the deterministic execution of parallel threads, allowing concurrent system to have real-time constraints combined with true parallel computing.

However, just as occam, XC is only for proprietary use, only supporting the XCore microprocessor architecture. No other implementations of the XC programming language exists, making XC unavailable for the mainstream audience.

Go

Go, also called Golang, is a concurrent programming language, developed by Google [44]. In contrary to occam and XC, Go was developed for the widespread platform use rather than some proprietary hardware. Since its launch in 2009, Go has made its way to platforms such as all desktop platforms, various microprocessors, as well as mobile devices.

The Go language has built-in concurrency primitives as a part of the language, including lightweight processes and channels. A major selling point of Go is the use of dynamic multithreading, where concurrent programs written in Go will by default utilize the available logical processor cores on the running processor architecture. This, together with native support for asynchronous IO, has made Go very popular for high-performance networking and multiprocessing systems.

3.2.2 Programming Libraries

Several CSP-based concurrency libraries have been developed for more popular and established programming languages not supporting CSP natively. Below are the most influential CSP-based concurrency libraries with dynamic multithreading presented.

C++CSP2

C++CSP2 [7] is a concurrency library for C++, providing concurrency primitives and abstractions based on CSP. C++CSP2, which is the successor of C++CSP [6], focuses on extending the original work by implementing a many-to-many threading model, essentially providing dynamic multithreading.

C++CSP2 achieves dynamic multithreading by allowing the programmer to define which processes run on which threads. The methodology is that the programmer must identify and define the course-grained parallelism between processes, and lets the library take care of the fine-grained parallelism between processes on a given thread.

C++CSP2 was the first major implementation of a concurrency library for C++ which offered CSP abstractions and dynamic multithreading. However, sine its release in 2007, the library is not very compatible with modern C++. With lack of modern C++ semantics, as well as in general being hard to develop with, has given C++CSP2 not much widespread use among C++ programmers.

C++CSP

C++CSP [8], **not** to be mistaken by the precursor of C++CSP2, is a concurrency library for C++. C++CSP aims to provide a CSP-based concurrency library for modern C++ which utilizes dynamic multithreading.

C++CSP implements its dynamic multithreading with a kernel threading model, where each process is a kernel-thread. Scheduling of the processes is therefore the responsibility of the OS, rather than the runtime system of the library. The runtime system design is very different from how C++CSP2 implements dynamic multithreading.

Since its release in 2016, C++CSP is a rather new programming library. Even though it is much more suited for more modern development of concurrent C++ programs, the dynamic multithreading implementation can be argued to not be suitable for high-performance and scalable concurrent systems.

Boost.Fiber

Boost.Fiber [45] is a concurrency library for C++, developed as a part of the C++ Boost libraries. From the project description, "*Boost.Fiber provides a framework for micro-/userland-threads (fibers) scheduled cooperatively*". A selection of concurrency primitives which are based on CSP is provided by Boost.Fiber, such as channels.

Boost.Fiber has support for dynamic multithreading to cooperatively schedule fibers across multiple processor cores. Dynamic multithreading is implemented by having multiple schedulers running on multiple kernel-threads, using work stealing for distributing work among the schedulers.

Boost.Fiber is a fairly new library, and has great potential for widespread use as its a part of the very popular and acknowledged Boost library collection. However, very few of the abstractions provided by Boost.Fiber are based on CSP, and lacks some very important abstractions such as alting on multiple channels. However, it is as of writing this thesis in active development, and could in the future expand the feature set with more well sought after CSP abstractions.

Part II

ProXC++ – The Library

Chapter 4

Library Specifications

This chapter introduces ProXC++, the library developed for this thesis. The specifications of ProXC++ includes a description of the feature set, the library API, target platforms and external dependencies.

4.1 Library Description

ProXC++ (pronounced "proxy plus plus") is a CSP-influenced concurrency library for modern C++, specifically aimed at multiprogrammed parallel programs. Modern C++ in this regard meaning support for C++14 standard and later. The central ambition of ProXC++ is to provide expressive and safe concurrency to C++ programs which fully and effectively utilizes available computational resources on multi-core architectures.

ProXC++ is a runtime system using a hybrid threading model. Work stealing is employed for load balancing between the schedulers, each running on an available logical processor core.

Available concurrency primitives in ProXC++ are fork-and-join parallelism, strict message-passing, simultaneous event handling, and soft real-time requirements.

The name "ProXC++" is a continuation of the original library ProXC [5], where ProXC++ targets C++ in contrast to ProXC targeting C. Since ProXC++ aims to be an improvement over ProXC, the "*plus plus*" could also be viewed as an

"*incremental better*" version of thelibrary. This is of course only intended to be humorous, as I personally like "*plus plus*" better than slapping a "2" at the end of the name.

4.2 Library Features

The complete set of features in ProXC++ is as follows:

- Multi-core support by default
- Lightweight threads called processes
- Per process operations via a separate namespace
- Channels; synchronous, unidirectional, type-safe, one-to-one message-passing between processes
- Parallelism; fork-and-join parallelism on a set of processes
- Replicators for parallel, generating dynamic number of parallel processes
- Alting; choice over multiple alternatives
- Alternatives of types channel read, channel write, and timeouts on timers
- Alternatives guarded on a boolean value
- Replicators for alting, generating dynamic number of choices
- Timers; types of relative, repeating, and absolute timeouts
- Soft real-time requirements for process suspension, channel operations, and alternation operations

4.3 Library API

Including ProXC++ in code requires including the header file `#include <proxc.hpp>`. All API related types and methods resides in the `proxc` namespace.

Lightweight process has the type `Process`. The process constructor takes a function pointer and the corresponding arguments for the given function pointer. A process can be implicitly created with the `proc` function. Or, an arbitrary number of processes can be implicitly created with the `proc_for` function, which takes any pair of Process iterators of pre-allocated processes, or an integer range and a function pointer and generates processes.

Processes can be spawned in parallel with the `parallel` function. The `parallel` function takes one-or-more processes and runs them in a fork-join model, meaning the calling processes will be suspended until all spawned processes has terminated.

Each process can be directly handled through a set of functions in the `this_thread` namespace, including process id, explicitly yielding, and explicit suspension for a given duration.

Channels has the type `Chan<T>`. The two channel ends, sending and receiving, has the types `Chan<T>::Tx` and `Chan<T>::Rx` respectively. Arbitrary number of channels can be created statically on the stack and dynamically on the heap. Statically created channels has the type `ChanArr<T,N>`, and dynamically created channels has the type `ChanVec<T>`. Both channel containers support indexing with brackets.

Timers reside in the `timer` namespace, and exists as the three types of timers `Egg`, `Repeat` and `Date`. All timers can be constructed with the `std::chrono` time points and durations.

Waiting on multiple channel operations simultaneously can be achieved with alting. Alting has the type `Alt`, which takes zero-or-more alternatives. These alternatives are created by function chaining appropriate methods on the alting object. Lastly, the `Alt::select` method is called which waits and chooses a ready alternative.

4.4 Target Platforms

ProXC++ is targeted for desktop environments, especially multi-core architectures. However, it should support all platforms that have access to a C++14 compliant compiler and the Boost C++ libraries [46] as well as the Boost.Context library [45] (as Boost is portable). "*Should*" is used here, because ProXC++ is only tested on 64-bit x86 Linux platform as of writing this thesis.

4.5 Dependencies

ProXC++ uses a handful header-only libraries from the Boost C++ libraries [46], and the compiled library Boost.Context [45] for portable and fast context switching between execution contexts. Boost.Context is used as a foundation of the user-thread implementation. As of why not rolling with a handwritten implementation of context switching, compared to ProXC, is with the simple reason of Boost Context being portable out of the box and writing context switching library is not the focus of this thesis.

Chapter 5

Design

First of all, ProXC++ is a concurrency library for C++. Only recently as of C++11 has the C++ programming language added concurrency primitives to the standard library, such as threading and futures. However, C++ lacks any standard support for concurrency in user space.

Designing a concurrency library for C++ with dynamic multithreading requires some notion of threading mechanism in user space, which in turn requires a runtime system. This runtime system acts as an invisible layer between the program and the OS, handling resources and logistics of scheduling the different processes running in the program.

All library functionality, such as channels, builds upon the framework provided by the runtime system. Essentially, anything must be done via the runtime system, which is why the limitations of the runtime system functionality limit the potentiality of the library functionality.

This chapter contains a full design of ProXC++, and explains some of the design choices. The runtime system is first designed, and subsequently the library features are designed based on the runtime system functionality.

5.1 Runtime System

Before discussing the design of the runtime system, the responsibilities of the runtime system are first identified. A ProXC++ program contains a set of processes, where each process represents some point of execution. The runtime system must be able to spawn, schedule, and synchronize these processes, i.e. process management. A scheduler could be responsible for process management.

For a single-core runtime system, only a single scheduler makes sense. However, how many schedulers are optimal for a dynamic multithreaded runtime system? Given the maximum number of parallel executions on a processor architecture is equal to the number of available logical processor cores, the optimal number of schedulers is the number of available logical processor cores, as each scheduler runs on its own logical processor core. If more schedulers were online than logical processor cores, situations would occur where multiple schedulers are struggling to run on the same logical processor core, which would be unproductive.

Another important factor is how process management is shared among schedulers. Two principal strategies exists, namely *centralized* or *distributed* management. Given a centralized management, all schedulers share the same control structures, including a shared process queue for which processes are to be scheduled from. All schedulers share the responsibility for all processes in a program. Given a distributed management, all schedulers has a set of processes for which it has the responsibility of managing. Schedulers share nothing, essentially meaning all processes has a parent scheduler.

A centralized management is easy to implement, as the entire runtime system such as control structures and processes are shared between schedulers. Schedulers do not have to worry about distributing processes for scheduling, as all processes are scheduled through the same queue. However, it does not scale as well with an increase in processor cores, as it creates contention between the increase in schedulers accessing the shared control structures. Distributed management is much harder to implement, as now each scheduler has its own set of control structures and processes. The main issue is how to efficiently distribute ready processes between schedulers. If done right, distributed management scales very well with an

increase in logical processor cores.

Given that ProXC++ aims to be a high-performance concurrency library, distributed process management is necessary for catering to all multi-core architectures.

To summarize, the runtime systems consists of a number of schedulers equal to the number of available processor cores. Process management is distributed among the schedulers, meaning each scheduler has responsibility for a set of processes.

5.1.1 Scheduler

The scheduler is the core of the runtime system, providing the logic and control structures necessary for process management. Given a scheduler, a set of zero-or-more unique processes are said to be under the schedulers management. This scheduler is the *parent scheduler* of said processes, which are called *work processes*.

A scheduler is either *running* or *idle*. A running scheduler has one-or-more work processes which are ready to be scheduled for execution, or simply called ready. When the scheduler runs out of ready work processes, it becomes idle.

Ready work processes are stored in a process queue called the *ready queue*. The scheduler uses the ready queue to determine whether it has ready work processes or not, i.e. an idle scheduler has an empty ready queue. When a scheduler has a non-empty ready queue, a work process is removed from the ready queue and resumed execution. Scheduling and rescheduling a work process is equivalent to adding the process to the ready queue.

As processes are scheduled in user space, the scheduler relies on *cooperative scheduling*. Cooperative scheduling entails running processes voluntarily yielding running time to the scheduler. If a process never yields to the scheduler, the process will run forever.

The main challenge of the scheduler is how to distribute superfluous ready work processes to schedulers which are idle. Two principal strategies exist: *work sharing* and *work stealing*. Work sharing consists of manually distributing new work processes among the schedulers. Work stealing consists of stealing ready work from other schedulers when a scheduler runs out of ready work, i.e. becoming

idle.

Work stealing is usually preferred over work sharing as it causes less process migration. Process migration is when management responsibility of a process is transferred between schedulers, or in other words the process changes its parent scheduler. Work stealing is however more complex to implement than work sharing, but scales much better on distributed multiprogrammed systems. Therefore, work stealing is employed as the load distribution strategy for the schedulers.

5.1.2 Processes

Processes represent some point of execution. As stated above, all work processes has a parent scheduler which corresponds to the scheduler running on a particular processor core. A scheduler is also a process but differs from work processes, as a work process represents meaningful work from the running program.

A work process with a corresponding parent scheduler may spawn new work processes. These new processes are scheduled by the parent scheduler. A work process may also synchronize or join on other work processes, meaning waiting for the other work process to terminate.

Some work processes may suspend itself, waiting for other work processes to reschedule the suspended process. The suspending work process simply yields to the scheduler. Suspended processes are not stored in any queues, as other work processes will reschedule the suspended process rather than the scheduler.

Work processes may also suspend, or sleep, until a given or derived time point. These processes are added to the parent schedulers *sleep queue*. The parent scheduler will frequently examine the sleep queue and reschedule any work processes with an expired time point. If a work process sleeps but is rescheduled by another work process before the time point is expired, the scheduler will reschedule and remove the waking process from the sleep queue.

5.2 Feature Design

The feature set of ProXC++ is what the programmer uses to create his or hers concurrent systems. The design of the feature set is important to provide the necessary abstractions for the programmer, which is presented in this section.

5.2.1 Timers

Three types of timers are available: *egg*, *repeat* and *date* timer. Timers are used for either explicit process suspension, or for timeout on channel or alting operations.

Egg timer is used for relative timeout. Just as an egg timer in real life, it is used to countdown for a specified period. It is not a one-shot timer, meaning the same timer can be reused multiple times. The countdown begins at the start of an operation, effectively resetting the timer if already used.

Repeat (or loop) timer is used for a periodic repeating timeout. The repeat timer will timeout in a periodic fashion, given a specified period. The timer only resets after a timeout. Compared to the egg timer, the repeat timer is also not a one-shot timer and can be reused, but the repeat timer does not reset the countdown at the start of an operation.

Date timer is used for absolute timeout. It is a one-shot timer, and will always be expired after a timeout. The date timer timeouts at a specified time point. The timer can never be reset, and therefore survives multiple operations.

5.2.2 Parallel Statement

Processes can spawn new processes in parallel with the *parallel statement*. The parallel statement takes one or more processes, either as single process statements or process replicators, and spawns and runs these processes in parallel. The spawned processes run concurrently with any other processes currently running in the program. The parallel statement is the only way to spawn new processes with ProXC++.

The parallel statement follows the fork-join model [47, pp. 88], where a sequential execution branches off at a designated point into parallel work, and subse-

quently joins/merges at another designated point and resumes the original sequential execution. Figure 5.1 gives a simple illustration of the fork-join model.

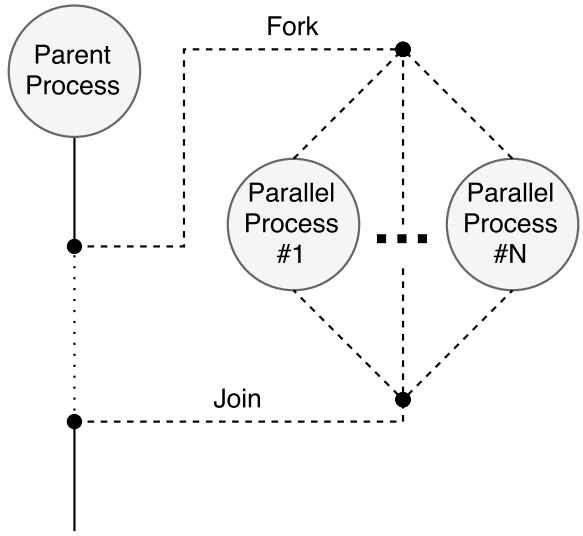


Figure 5.1: A parent process spawns N parallel processes following the fork-join model.

The process calling the parallel statement is the parent process. When calling the parallel statement, the parent process is suspended until all processes within the parallel statement have spawned, executed and terminated. When all parallel processes have terminated, the parent process resumes execution.

Processes to be executed in parallel can be defined in two ways: either as a single process or as a replicated process. See Subsection 5.1.2 for a more detailed explanation of processes.

5.2.3 Channels

Channels forms the only means of communication as well as synchronization between work processes via message-passing. Given a channel, some type of message can be transmitted between a sender and a receiver.

Hereafter, the term "*Tx*" denotes a channel end for which a work process is sending on, and the term "*Rx*" denotes a channel end for which a work process is receiving on. The term "*participant*" denotes either a Tx or Rx.

An synchronous channel implies both a Tx and a Rx must be at a channel to complete a channel operation, e.g. a Tx cannot transmit a message without a Rx ready to receive. If only one of the two required participants is ready to transmit, the participant must block (wait) until the other side is ready. This kind of behaviour is called *rendezvous*. A synchronous channel is also unbuffered, given that no messages are stored intermediately in the channel.

The opposite of a synchronous, unbuffered channels are asynchronous, buffered channels, where messages are buffered if there are no receivers ready. Tx never waits regardless of a ready Rx or not, while Rx must only wait if the buffer is empty. See Figure 5.2 for an illustration of the difference between synchronous and asynchronous channels.

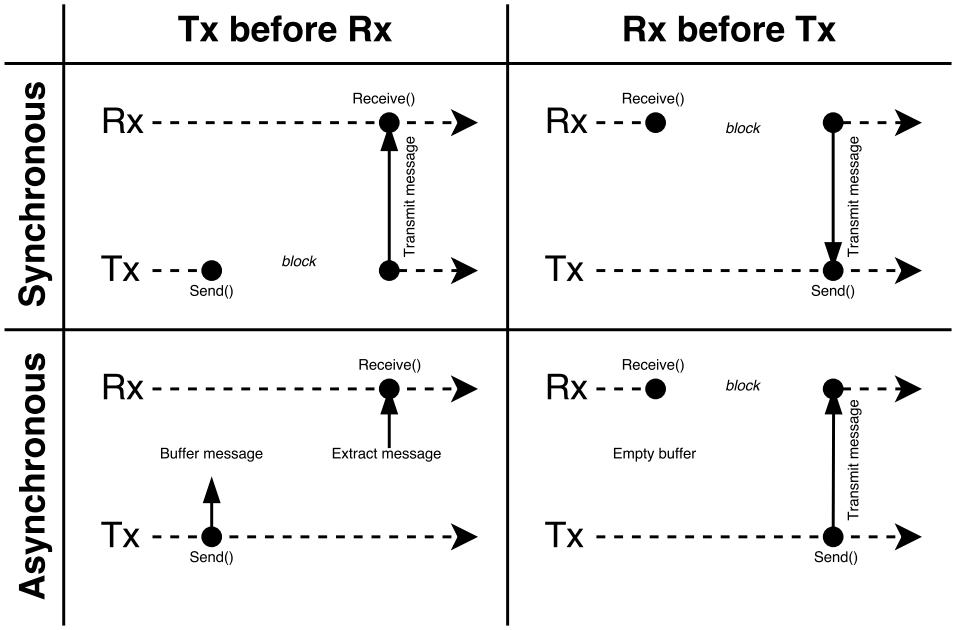


Figure 5.2: Different configurations of synchronous and asynchronous channel communication. It is given the asynchronous channel has an infinite buffer size.

Synchronous channels are usually preferred over asynchronous channels, as it is more deterministic. When a channel operation is completed, both participants know the opposite participant is in a particular state. This is not the case for asynchronous channels. Regarding memory use, synchronous channels always re-

quire constant memory while asynchronous channels has to allocate memory for its buffer.

A unidirectional channel only allows messages to be transmitted in one direction. In contrast, a bidirectional channel allows messages to be transmitted in both directions. Unidirectional channels are more restrictive than bidirectional channels. The argument for unidirectional channels is a follows: given a bidirectional channel, it is possible for both participants to disagree which direction a channel transmission is for a given moment, i.e. both are participant are trying to send or receive, causing a deadlock. The direction of a channel transmission for a unidirectional channel is always given, and can bever cause any deadlocks.

Some channel design has the concept of limiting how many unique processes can access and use a channel. This concept is called *disjointness rules* in XC [3, pp. 32] and *usage rules* in occam [2, pp. 116]. In essence, these "rules" specify if either end of a channel, sending and receiving, can only be used by a unique process or any processes. Usually denoted by *one/any(senders)-to-one/any(receivers)*, which gives the four configurations: *one-to-one*, *one-to-any*, *any-to-one*, and *any-to-any*. Both XC and occam follows the one-to-one design.

Ensuring both Tx and Rx agrees on the type of message being transmitted has to do with *type safety*. If both participants of a channel operations disagrees in the type of the transmitted message, a *type error* occurs. Type safety is therefore to discourage or prevent type errors. It is obviously desirable for a type safe channel, but it is highly dependent on support from the programming language and/or the compiler to enforce type safety. A compromise is size safe channels, where both participants agree on the size of the message rather than the type. Size safety is not as desirable as type safety, but is much more doable to enforce.

Channels have a notion of being open or closed. An open channel operates as normal, but a closed channel will deny any operations being completed. A channel always starts as open, and can be closed in one of two ways: an explicit close from either participant, or either channel end goes out of scope, i.e. is no longer available. When a channel becomes closed it remains closed. A closed channel is used to signal participants when no more values will be sent on the channel, which is useful to communicate completion to the opposite channel end.

Considering the different channel confifugrations above, channels in ProXC++ are synchronous and unbuffered, unidirectional, one-to-one and type safe.

5.2.4 Alting

Alting is a construct allowing to wait on multiple alternatives simultaneously, selecting an alternative when one or more alternatives are ready. Alting is also sometimes called *selective choice*. Each alternative has an optional corresponding block of code (closure) which is executed if chosen. Alternatives can also be guarded by a boolean value, enabling or disabling certain alternatives on certain conditions. These boolean guards are evaluated at the initialization of the alting procedure.

Three types of alternatives exists: channel operations, timers, and skip. Channel alternatives include both sending and receiving.

The alting procedure works as follows: at initialization, zero-or-more alternatives are enabled for selection. Each enabled alternative is checked if ready to synchronize. If one or more alternatives are ready, one is selected. If none are ready, the alting procedure suspends and waits for the first alternative to become ready. This first ready alternative is automatically selected. When the alting procedure has selected an alternative, the corresponding closure is executed if present.

The alternatives operates, when enabled, as follows:

1. **Channel operation** – specifies a channel send or receive. A channel send is accompanied with the item to transmit, while a channel receive can optionally capture the received item in the alternative closure. The alternative becomes ready when the opposite participant of the channel is present.
2. **Timeout** – is a timer with a corresponding timeout. If the alting process has not selected an alternative before timeout, the timer alternative is selected.
3. **Skip** – is an alternative which is always ready. It can be compared to the `default` case in switch cases. If no alternatives are ready at the alting initialization, the skip alternative is selected.

Multiple channel alternatives can be enabled for an alting procedure. Multiple timer alternatives can also be enabled for an alting procedure, however, only the

timer with the shortest deadline will be registered. If multiple skip alternatives are enabled, only the first skip alternative is registered.

An channel alternative can either be selected *actively* or *passively*. Actively selecting involves the alting procedure performing the selection, while passively selecting involves external events performing the selection. A channel alternative can also only be selected after being entered. Entering a channel alternative is essentially registering the interest of the alting procedure to perform the channel operation. Leaving a channel alternative performs the opposite, unregistering this interest.

5.3 Runtime Algorithms

A selection of new and existing algorithms are used for the different runtime and library features, which are presented in more detail in this section.

5.3.1 Work Stealing Algorithm

As each scheduler has their unique set of work processes, work stealing is employed to distribute superfluous work processes from a running scheduler to an idle scheduler.

The work stealing algorithm centers around the ready queue. As long as ready queue contains ready processes, the scheduler will continue to resume work processes from the queue. However, when the ready queue becomes empty, the scheduler resorts to work stealing. Randomized work stealing as described in Blumofe and Leiserson [34] goes as follows:

1. **Select victim** – another scheduler, called a victim, is randomly chosen.
2. **Try stealing** – the current scheduler tries to steal some work processes from the victim. If failed, restart procedure;
3. **Resume work** – else, the stolen work processes are stored in the ready queue, and a work process is resumed.

As simple as the algorithm sounds, some details must be taken into consideration. Is it feasible for an idle scheduler to repeatedly retrying to steal work until

it succeeds? Stealing a process is the same as process migration, and process migration does cause some overhead. Consider the following programs which have a few number of parallel processes or many dependent processes. An aggressive work stealing scheme would cause many unnecessary process migrations, compared to none for a single-core runtime system.

To mitigate unnecessary process migrations by overaggressive work stealing, an idle scheduler can either wait for a short period of time before retrying, or it could wait until signaled by a scheduler with superfluous work processes.

5.3.2 Channel End Algorithm

The channel algorithm is different whether the channel end is alting or not. A set of observations are postulated in Lists 5.1 to 5.3 to form the basis of the channel algorithm.

1. A channel operation consists of a single Tx and a single Rx arriving in either sequential order, both of which can be alting or non-alting.
2. When a channel end has arrived for a channel operation, another channel end of same type cannot arrive before that channel operation has completed.
3. Any channel end may at most suspend once during a channel operation.
4. The last channel end to arrive at a channel operation will always complete the item transmission.
5. If a channel end is suspended during a non-timed channel operation, it is only rescheduled if the channel operation completed or the channel closed.
6. During a timed channel operation, a suspended channel end may also be rescheduled if a timeout occurs.

List 5.1: Observations for both non-alting and alting channel ends.

1. The non-alting channel end arriving first at an channel operation will always suspend.
2. The channel end arriving last at an channel operation will never suspend.
3. A process holding both channel ends for a channel will always block indefinitely if operating on the given channel.
4. A channel end never leaves the channel operation after arrival unless it completes, the channel closes, or times out, i.e. a channel end always commits to a channel operation.

List 5.2: Observations for non-alting channel ends.

1. A channel end never explicitly suspends during a channel operation. This is done indirectly by the alting procedure.
2. A channel end may leave the channel operation after arrival and return later, i.e. a channel end may not commit to a channel operation.

List 5.3: Observations for alting channel ends.

A key observation to make from the observations in Lists 5.1 to 5.3 is the symmetry of the channel operation; it is invariant whether Tx or Rx is completing the operation. This makes the algorithm very similar for both sending and receiving on a channel. Another observation to make is how non-alting channel ends commit, while alting channel ends may not commit. This skew in committal channel ends allows for important assumptions in the algorithm.

Non-Alting Channel End Algorithm

A non-alting channel end has two cases to consider: either the other channel end is non-alting or alting. A channel operation for a non-alting channel end can be considered to consists of three possible steps:

1. **Check channel is open** – return if the channel is closed, else continue.
2. **Check for opposite channel end** – if an opposite channel end is present at the channel, try selecting the channel end. If successful, complete the transmission, reschedule opposite channel end and return ok. Else, continue.
3. **Wait for opposite channel end** – register the channel end in the channel and suspend. When rescheduled, check if the item was transmitted or not and return appropriately.

Note that the entire procedure is surrounded by mutual exclusion using the channel lock, because channel ends must arrive at the channel in a sequential order.

To elaborate on Step 2, selecting the opposite channel end is necessary since an alting opposite channel end is not committal to the channel operation, even though it is present. Selecting a non-alting channel end is always successful since it is always committal, while an alting channel end may fail. This selection process

is further explained in Subsection 5.2.4.

Completing the transmission in Step 2 involves transmitting the item and setting the consumed flag, and checking the transmission in Step 3 involves checking and resetting the consumed flag.

The consumed flag is necessary to signal the suspended channel end if the item was transmitted, even though the channel has been closed. Consider this situation: Tx enters the channel and is suspended, waiting for Rx. Rx enters the channel, completes the transmission, and reschedules Tx. Before Tx resumes execution, Rx closes the channel. Now, when Tx resumes it is impossible to tell if Tx was rescheduled because of completed operation or channel closing. The consumed flag is here to signal the rescheduled channel end was rescheduled because of either case.

Alting Channel End Algorithm

An alting channel end has the same two cases to consider: either the other channel end is non-alting or alting. The non-alting case is trivial, with reasoning that if the opposite end is present then simply complete the transmission. The alting case is however more complicated.

Before describing the algorithm, first consider this situation: Two processes both alting on the same two channels, however, each process has the Tx of one channel and Rx of the other, making a cycle. See Figure 5.3 for illustration. Both of these alting processes must somehow agree on which case they both select. Some synchronization between two alting channel ends must therefore exist. Note that this alt-to-alt synchronization must be asymmetrical to avoid deadlocks.

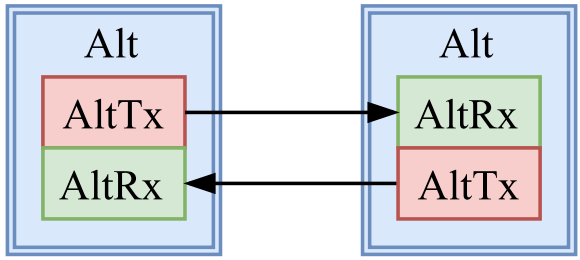


Figure 5.3: Illustration of the cyclic alting problem.

A value with three possible states is used for alt-to-alt synchronization. The three states are *none*, *offered*, and *accepted*. Given all alting channel ends have a well defined static ordering, the alt-to-alt synchronization protocol says the lower ranked channel end is to offer synchronization, while the higher ranked channel end is to accept. Note that for this to not be symmetric, the ordering must be invariant of the channel end type.

Given this alt-to-alt synchronization, the alting channel end algorithm is as follows:

1. **Check alt-to-alt synchronization** – if an opposite channel end is present and is alting, check synchronization state. Else, continue. If offered, accept and complete the alt-to-alt transmission. Else, return and try later.
2. **Check channel is open** – return if the channel is closed, else continue.
3. **Check for opposite channel end** – return if there is no opposite channel end present, else continue.
4. **Check if opposite channel end is non-alting** – if the opposite channel is non-alting, complete the transmission, reschedule opposite channel end and return ok. Else, continue.
5. **Check rank of opposite channel end** – compare the rank of the opposite channel end with self.
6. **Self has lower rank** – offer synchronization and wait until response. If the sync was accepted, return ok. Else, return appropriately.
7. **Self has higher rank** – check synchronization state. If offered, complete transmission and accept synchronization. Else, return appropriately.

The reason for checking the alt-to-alt synchronization in Step 1 before checking the channel is open has to do with acquiring the channel lock. Since the lock is held by the opposite channel end while offering synchronization, an offered synchronization must be resolved before acquiring the channel lock.

In Step 4, the alting channel end does not need to synchronize with the opposite channel end if it is non-alting, due to selecting a non-alting channel end always succeeds.

The synchronization procedure in Step 6 is a little more convoluted than stated,

since the alt-to-alt synchronization is only needed if the opposite alting procedure is checking. If it is waiting, normal selection is used.

The synchronization procedure in Step 7 is more straightforward. If synchronization is offered, complete the transmission and accept. If it is not offered, check the alting procedure state. If the alting procedure state is checking, try later. If the state is waiting, try normal selection.

5.3.3 Alting Algorithm

The alting algorithm works in three phases: *checking*, *waiting*, and *completing* phase. The waiting phase is only entered if the checking phase results in no selected alternatives. Below are the three phases explained in more detail:

1. **Checking phase** – is the active phase of the procedure, where alternatives can be actively selected. Each enabled channel alternative is entered. Next, all channel alternatives are checked if ready. If one or more are ready, all ready channel alternatives are actively selected in random order. When the first alternative is successfully selected, the alting procedure continues to the completing phase. If no channel alternatives are ready or all channel alternatives results in a failed selection, the alting procedure continues to the waiting phase.
2. **Waiting phase** – is the passive phase of the procedure, where alternatives can be passively selected. If a skip alternative is enabled, the skip alternative is now selected and the procedure continues to the completing phase. If a timer alternative is enabled the procedure suspends itself until the timer expires, else the procedure suspends itself indefinitely. When the procedure is rescheduled, it is either because of a channel alternative passively selecting or the timer alternative has expired, and the corresponding alternative is selected.
3. **Completing phase** – each channel alternative which was entered are now left, and the closure of the selected alternative is executed if present. This completes the alting procedure.

Note that if any channel alternatives becomes ready after it has been checked

by the alting procedure in the checking phase, that alternative has to wait until the procedure enters the waiting phase to try passively select itself.

Given a number of channel alternatives are ready during the checking phase, each alternative must be resolved before continuing to the waiting or completing phase. Resolving a channel alternative involves trying to complete the channel operation, which either results in a success or fail. The first channel alternative which results in a success short-circuits the selection order and continues to the completing phase. However, an alt-to-alt channel operation has a slightly larger overhead because of alt-to-alt synchronization, which means trying to complete the channel operation may result in a try-later result. A try-later result means the alt-to-alt synchronization cannot be resolved right now, and must be tried later. The selection order therefore skips the alternative and tries later. All channel alternatives must be resolved with a failed result before continuing to the waiting phase.

Chapter 6

Implementation

The library is written in C++, with standard C++14 dialect. The reader is expected to have a fair understanding of C++, and being familiar with standard C++11 dialect or newer is recommended. Detailed explanations of the C++ programming language is not presented here. Refer to any C++ reference (e.g. Stroustrup [48]) for more details.

6.1 Data Structures

A set of data structures is commonly used by the runtime system. Apart from the C++ Standard Template Library (STL), the most notable data structures are *intrusive containers*, *concurrent queues*, and *mutual exclusion locks*.

6.1.1 Intrusive Containers and Pointers

Equivalent to any other containers, intrusive containers store some kind of data in some sort of way. The difference is how the container stores the necessary data used to organize the data. A non-intrusive container is responsible for storing the necessary data, while for an intrusive container the elements are responsible for storing the necessary data. In other words, the element becomes "aware" of being a part of the intrusive container. Usually, intrusive containers are implemented with the elements having *hooks* as data members. These hooks contains all the

necessary data used by the intrusive container to store the elements.

An intrusive pointer is the intrusive equivalent of a smart pointer with shared ownership, releasing a dynamically allocated object when all owners have released the pointer. The owner counter is stored in the object rather in the pointer.

Intrusive containers and pointers offer better performance compared to non-intrusive containers, as they minimize memory allocations and better memory locality. Intrusive containers and pointers are however much less maintainable, and are much harder to modularize.

Boost Intrusive containers and Boost Intrusive pointers are used as the implementation of intrusive containers and pointers.

6.1.2 Concurrent Queues

Concurrent queues are queues which are safe to use concurrently, often denoted as *thread safe*. The most common approach is taking a non-thread safe queue and enforcing mutual exclusion around the critical regions. This approach however is not desirable, as it has very low throughput in multiprogrammed programs.

Concurrent queues often differentiate between single or multiple producers and consumers. Producers are processes which insert elements into the queue, and consumers are processes which remove elements from the queue. The runtime system uses the variants *single-producer-multiple-consumer* (SPMC) queues and *multiple-producer-single-consumer* (MPSC) queues.

A SPMC queue is used as a double-ended queue for work-stealing, implemented with the Chase-Lev algorithm [35] combined with efficient work-stealing for weak memory models [36].

An MPSC queue is used by schedulers to signal other schedulers a work process is to be rescheduled on the corresponding scheduler, implemented with the design presented by Vyukov [49].

6.1.3 Mutual Exclusion Locks

Creating a complete non-blocking system is usually impossible for a multiprogrammed system, and sometimes resorting to mutual exclusion in critical regions

are unavoidable. Different types of locks are suitable for different situations. Whether the lock is often contested, meaning multiple kernel-threads are trying to acquire the lock simultaneously, and if the lock is short-term held or not, will affect the performance.

Brown [7, pp. 196–199] performs a case study on different mutexes, describing various mutex algorithms and provides a benchmark and analysis of their performance. The conclusion from the case study is that for low contested and short-term held mutexes, spinlocks yields best performance regarding low latency.

For multi-core architectures, the *test-and-test-and-set* (TTAS) spinlock is generally favorable as it causes less memory contention than the standard spinlock. Instead of constantly trying to test-and-set the lock, it waits until the lock appears free. Different variants of the TTAS spinlock includes constant/exponential backoff during contention and cache friendly atomic operations.

For most inter-process runtime procedures, a TTAS spinlock with exponential backoff during contention is used. For idle schedulers waiting, the standard `std::mutex` lock is used.

6.2 Runtime System Implementation

The runtime systems consists of two major implementation parts: implementation of lightweight processes, and implementation of a runtime scheduler.

Processes are implemented as user-threads, and are called *contexts* in the runtime. Processes use contexts as its back end, meaning processes represent a meaningful part of computation while contexts represent the actual processor state of the computation. At runtime, the scheduler manipulates contexts when transferring control of execution, or *context switches*, between processes.

The runtime differentiates between three types of contexts, and if a context type is dynamic or static, i.e. a context can migrate between schedulers or not. The context types are as follows:

1. **Main context** – context of a kernel-thread. When the main context returns, the corresponding kernel-thread terminates. Main contexts are static; they

cannot migrate between schedulers.

2. **Scheduler context** – context of the scheduler. The scheduler context only returns when the main program exits. Scheduler contexts are static; they cannot migrate between schedulers.
3. **Work context** – context of a work process. Work contexts are dynamic; they can migrate between schedulers.

The initialization procedure of the runtime system is invoked with the first call to the scheduler. Given there are N online logical processor cores, the procedure spawns $N-1$ additional kernel-threads, as the initial main kernel-thread is included. A scheduler is initialized for each kernel-thread, representing the runtime environment for the corresponding thread. Schedulers are accessed via thread local static methods, which point to the scheduler object.

The context of the `main()` function of a program, also called the initial main context, is special. It is the only context which represents some productive work of the running program, but cannot migrate between schedulers. All other main contexts (from the other spawned kernel-threads) only joins the scheduler context, and operates invisibly to the programmer. Main contexts cannot migrate because they are calling the runtime constructors. When the main context returns, the destructors are called for the corresponding runtime objects, such as the scheduler. If the main context where to migrate and return on a kernel-thread other than its origin, the runtime cleanup would result in erroneous behaviour.

When the initial main context returns, the cleanup procedure of the runtime system is invoked if and only if the initialization procedure has been invoked previously.

Only the scheduler for which the initial main context resides on will begin as running. All other initialized schedulers will begin as idle.

Figure 6.1 displays a rough outline of how the contexts, user-threads and kernel-threads are organized relative to each other. Note that the number of work contexts on each kernel-thread are zero-or-more, but the accumulative sum of all work contexts on all kernel-threads equals M.

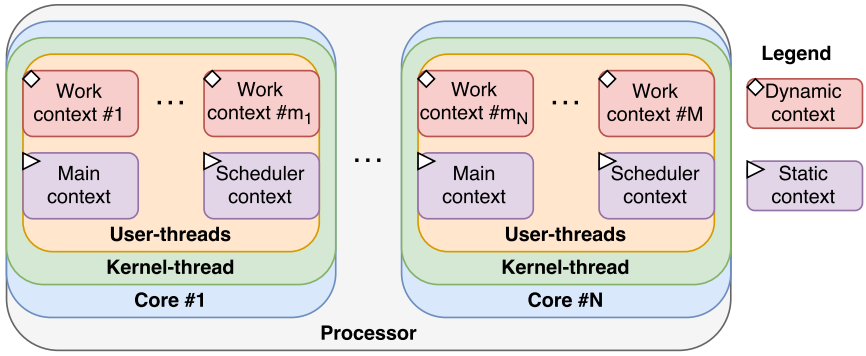


Figure 6.1: Overview of the runtime system with M work contexts, given N online processor cores.

6.2.1 Processes

Processes are a vital part of ProXC++. As stated many times before, processes represent some computation of the total program, which in code translates to running a function concurrently with the rest of the system. Constructing a process requires supplying a function pointer and its corresponding arguments. The process object constructs a context object out the function pointer and arguments, and stores an intrusive pointer to the context object as a data member. In other words, the process is no more than an opaque type to a context object. The programmer implicitly creates new contexts through processes, while the scheduler under the hood operates on the contexts. See Listing 6.1 for reference.

Listing 6.1: Minimal process type.

```
1  class Process {
2  private:
3      using CtxPtr = boost::intrusive_ptr<proxc::Context>;
4      CtxPtr ctx_ptr_{nullptr};
5  public:
6      template<typename Fn, typename... Args>
7      Process(Fn&& fn, Args&&... args);
8  };
```

The context object represents the execution state of the computation. Each context object contains an execution context, which is the actual execution state of the computation. The execution context is implemented by the Boost.Context

library [45], which encapsulates context switching and manages the associated context' stack. Note that a context object refers to the context type defined by the runtime system, and execution context refers to the Boost.Context object which defines the actual processor state.

The context object does not implement much functionality, other than wrapping the execution context state with additional control structure data and intrusive data members. The functionality is implemented in the scheduler, which is explained in further detail in Subsection 6.2.2.

Execution contexts can either create a context of the current running execution context, or takes a function pointer of the type `void(void*)`. Transfer of control flow between execution contexts is done by calling another execution context with the overloaded `void* operator()(void*)` method. Parameter passing is possible through void pointer casting. When transferring control to another execution context, an optional pointer can be passed. If this is the first transfer of control to the execution context, the parameter will be passed as the void pointer argument for the function. Else, the void pointer will be passed as the return value of the transfer of control operator.

Listing 6.2 gives an example of how execution contexts work. An execution context of the currently running context is made, as well as an execution context of a lambda function. The output of the code snippet will print `main` and `work` in that order. Refer to the Boost.Context documentation for a more thorough explanation of the execution context implementation and API [45].

When creating a context object, an enclosing entry function of the type `void(void*)` is created, which calls the received function with its arguments. This entry function, called trampoline, handles the void pointer argument from the execution context, calls the process function, and calls the terminate procedure when the process function returns. This way, all terminating processes can be gracefully resolved by the trampoline function while remaining invisible to the programmer. Entry functions are created with generic lambdas, allowing to create generic entry function calling over any type of function pointer and arguments. This is explained in further detail in Subsection 6.2.2.

The context object contains the type of the context, the execution context, the

Listing 6.2: Transfer of control between execution contexts.

```
using ex_ctx = boost::context::execution_context;
ex_ctx main_ctx{ex_ctx::current()};
ex_ctx work_ctx{[&main_ctx](void* vp){
    std::string* msg = static_cast<std::string*>(vp);
    std::cout << *msg << std::endl; // prints `main`
    std::string work_msg{"work"};
    // transfer of control to main_ctx
    main_ctx(static_cast<void*>(&work_msg));
    // never returns
}};
std::string main_msg{"main"};
// transfer of control to work_ctx
void* vp = work_ctx(static_cast<void*>(&main_msg));
std::string* msg = static_cast<std::string*>(vp);
std::cout << *msg << std::endl; // prints `work`
```

entry function for the particular process, and a pointer to the parent scheduler. Each process also has its own spinlock, used for inter-process synchronization. Additionally, control data used by the scheduler is stored in the context object, such as intrusive hooks and time points for timed suspension. See Listing 6.3 for a stripped down version of the context class definition.

Each context object also contains a wait queue, containing other contexts which are waiting for the termination of the given context object. This queue, together with the context object spinlock, is used to implement process joining.

Listing 6.3: Minimal context type.

```
class Context {
private:
    using ExCtxT = boost::context::execution_context;
    using EntryFn = delegate<void(void*)>;
    using TimePointT = std::chrono::steady_clock::time_point;
    ExCtxT ex_ctx_;
    EntryFn entry_fn_{nullptr};
    proxc::Scheduler * scheduler_ptr_{nullptr};
public:
    TimePointT time_point_{TimePointT::max()};
    proxc::Alt * alt_{nullptr};
    proxc::Spinlock splk_;
    /* impl defined */ wait_queue_{};
    /* intrusive data members */
};
```

6.2.2 Scheduler

The scheduler is the corner piece of the runtime system. It has the sole responsibility of managing the different processes, including creating, scheduling, and synchronizing processes. The scheduler runs as its own context, invisible to the programmer.

When the scheduler is initialized by the runtime initialization procedure, the scheduler context enters the scheduler event loop. The scheduler event loop is where the scheduler context resides during the lifetime of the entire program.

A scheduler is implemented as a class object, consisting of context queues and process management logic. See Listing 6.4 for reference.

Listing 6.4: Minimal scheduler type.

```
1  class Scheduler {
2  private:
3      struct Initializer;
4      bool exit_{false};
5      proxc::Context* running_ptr_{nullptr};
6      proxc::Spinlock splk_;
7      /* impl defined */ policy_;
8      /* impl defined */ work_queue_{};
9      /* impl defined */ sleep_queue_{};
10     /* impl defined */ terminated_queue_{};
11     /* impl defined */ remote_queue_{};
12 public:
13     static proxc::Scheduler* self();
14     static proxc::Context* running();
15 };
```

Scheduler Initialization

The scheduler has two static methods: `self()` and `running()`. The `self()` method returns a pointer to the scheduler object which is the parent scheduler for the current running kernel-thread, and can be called from any context. The `running()` method returns a pointer to the context object currently running on the kernel-thread, and can be called from any context.

The `self()` method is the starting point of the runtime system, for which the first call to will trigger the initialization procedure. The method contains a static thread local variable of the scheduler initializer object, which is a Schwarz counter.

A Schwarz counter, also known as a nifty counter, is a C++ idiom for ensuring a non-local static object is initialized before its first use and destroyed only after last use of the object, where the non-local static object in this case is the scheduler object.

The first call to the scheduler initializer constructor on each kernel-thread will initialize the scheduler object for the corresponding thread, and simultaneously set its static thread local pointer member to the scheduler object. This static thread local pointer is the pointer which is returned by `self()` method. Additionally, the `running()` method returns the `running_ptr_` member from the returned scheduler object, which is retrieved by the `self()` method.

Listing 6.5: Static scheduler methods.

```
1  // Schwarz counter
2  struct Scheduler::Initializer {
3      thread_local static Scheduler* self_{nullptr};
4      thread_local static std::size_t counter_{0};
5      Initializer() {
6          if (counter_++ == 0) {
7              /* initialize scheduler */
8              /* member self_ is set */
9          }
10     }
11     ~Initializer() {
12         if (--counter_ == 0) { /* destroy scheduler */ }
13     }
14 };
15 Scheduler* Scheduler::self() {
16     thread_local static Initializer init;
17     return Initializer::self_;
18 }
19 Context* Scheduler::running() {
20     return Scheduler::self()->running_;
21 }
```

Process Queues

From Listing 6.4 a total of five queues are used by the scheduler for process management: *ready*, *work*, *sleep*, *terminated* and *remote* queue.

The ready queue is called a scheduling policy, or just policy for short. The scheduling policy is an abstraction for the scheduler, for which ready processes are enqueued to and ready processes to resume are dequeued from. The scheduling

policy is responsible for storing the ready processes, and how and in which order the processes are enqueued and dequeued is up to the implementation of the scheduling policy. As of writing this thesis, the default policy is hard coded with the work stealing policy. However, any type of scheduling policy could be implemented, such as round robin. Following the algorithm described in Subsection 5.3.1, the queue is implemented as a combination of a double-ended queue, deque for short, and a doubly linked list. The deque is used for dynamic processes and the doubly linked list is used for static processes. This is to avoid static processes from being stolen by other schedulers. The deque has two distinct ends called *top* and *bottom*. Enqueueing and dequeueing a dynamic process to the dequeue pushes and pops the process from the bottom end, respectively. The doubly linked list acts as a FIFO queue. The work stealing commences when both the deque and the doubly linked list are empty. A deque from another scheduler is chosen at random, and a process is tried to be stolen from the top end. Pointers to all deques from all schedulers are stored in a static list. Note that work stealing is happening unbeknownst to the scheduler, as the scheduler only enqueues and dequeues processes to the scheduler policy.

The work queue, which holds all processes the scheduler is the parent of, is implemented as an intrusive doubly linked list. The sleep queue, which holds all processes suspended until a time point, is implemented as an intrusive multiset[5]. The terminated queue, which holds all process which has terminated and can be destroyed, is implemented as an intrusive doubly linked list. Both work, sleep and terminated queues are only manipulated by a single scheduler and are therefore not thread safe, i.e. no other schedulers can access these queues safely.

The remote queue is a concurrent MPSC queue, where the managing scheduler is the consumer while any other schedulers are the producers. Whenever a scheduler reschedules a process and is not the parent scheduler, the context is placed in the remote queue to the parent scheduler. The parent scheduler transitions processes in the remote queue and enqueues them in the scheduling policy. In essence, the remote queue is used to signal schedulers when their processes are to be rescheduled,

[5]A multiset is an associative container that contains a set of objects, which allows multiple keys with the same values.

since remote schedulers cannot safely enqueue processes to the scheduling policy.

Process States and Transitions

The scheduler imposes a set of states for a process and how a process transitions between these states. An overview of the finite state machine of the states and transitions are shown in Figure 6.2. The different process states imply the following:

- **Ready** – a newly created process always starts in the ready state. A ready process is enqueued to a ready queue and is ready to be resumed by the parent scheduler.
- **Running** – a process is currently executing on a kernel-thread.
- **Ended** – a process has terminated and is enlisted in the terminated queue. An ended process can be destroyed by the parent scheduler.
- **Suspend** – a process is waiting indefinitely. Another process might reschedule the suspended process.
- **Sleep** – a process is waiting until a given time point. A sleeping process is enlisted in the sleep queue and is rescheduled by the parent scheduler when the time point has expired or is rescheduled by another process.
- **Join** – a process is waiting for another process to terminate. The joining process is enlisted in the wait queue of the process, and will be rescheduled when the process terminates.
- **Remote** – a process is rescheduled by an another process which does not have the same parent scheduler. A remote ready process is enlisted in the remote queue.

Some observations can be made by the transition diagram in Figure 6.2. A ready process can only transition to running. Only a running process can terminate. A transition to and from running is the same as a context switch between two processes. The remote state can be seen as an intermediate ready state. When a running process transitions from running to one of the states suspend, sleep or join, the next transition must be ready state. A transition between the three states suspend, sleep or join cannot occur.

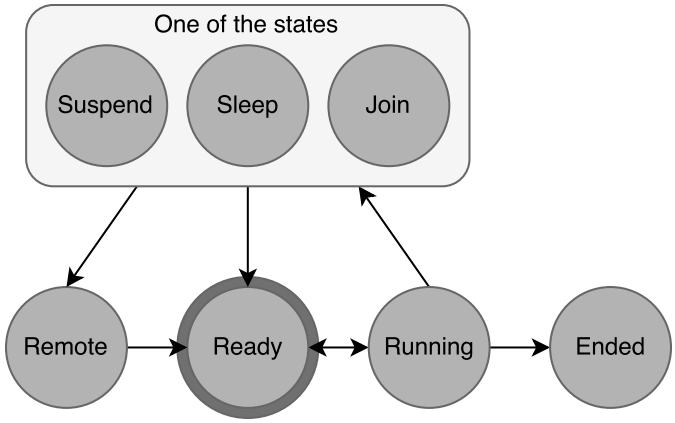


Figure 6.2: Finite state machine of states and transitions for a process.

Scheduler Functionality

The scheduler implements a set of methods for manipulating processes, such as process management and inter-process synchronization. It is important the scheduler implement the necessary functionality, as this forms the basis of what library features can be implemented. The following functionality is available for a process:

- **Process attaching and detaching** – allows a process to be attached and detached to a scheduler, i.e. setting and removing a parent scheduler to a process.
- **Process rescheduling** – allows a process to reschedule other processes.
- **Process suspension** – allows a process to suspend execution, either indefinitely, for an another process, or until a given time point.
- **Inter-process synchronization** – allows a process to synchronize on other processes, meaning the process waits for the other process to terminate.
- **Context switch operations** – allows a process to perform certain operations immediately after a context switch.

To further elaborate on the points above, attaching and detaching a process is necessary when work stealing. As all schedulers have a set of processes of which it is responsible for, attaching a process is to register this responsibility. Detaching would unregister this responsibility. When a process migrates between two schedulers, the process must detach of its previous parent scheduler and attach to its

new parent scheduler.

A process rescheduling another process requires checking whether they have the same parent scheduler or not. With the same parent scheduler, the process is enlisted directly to the ready queue. If they have different parent schedulers, the process is first enlisted to the remote queue, then enlisted to the ready queue when the parent schedulers transitions the process. If a rescheduled process was currently suspended with a timeout, the process is also removed from the sleep queue when enlisted to the ready queue.

Process suspension falls into two categories: waiting until some time point, or wait indefinitely. A process waiting until some time point enlists the process to the sleep queue. However, it is possible for other processes to reschedule the waiting process before the timeout occurs. A process waiting indefinitely may only be rescheduled by other processes.

Inter-process synchronization, also called joining, allows for processes to wait for other processes to terminate. Each process has its corresponding spinlock, which is used to enforce mutual exclusion access to the process termination flag. When a process terminates, the lock is acquired, and the wait queue is checked. All processes enlisted to the wait queue is rescheduled, and lastly the lock is released.

When a process is joining on another process, the lock is first acquired, and the termination flag is checked. If the termination flag is not set, the process enlists itself on the wait queue of the joining process, and the process suspends itself with releasing the lock. If the termination flag is set, the operation returns immediately. Note that it is safe for processes to access the wait queue as its guarded with mutual exclusion.

Context switch operations allow for processes to perform an action after the context switch has occurred. This is necessary when a process can only perform said action after itself has yielded execution. Currently, two operations can be performed with a context switch operation: releasing a lock and rescheduling a process. Releasing a lock after a context switch is necessary when a process releasing a lock could result in the given process being rescheduled by an another process. An example of this is inter-process synchronization. Rescheduling a process after a context switch is necessary when a process want to reschedule itself,

such as processor yielding.

Scheduler Event Loop

The scheduler event loop is the where the entirety of the scheduler process is executing. The event loop consists of the following: check exit condition, process cleanup, process transitions, and process scheduling. See Listing 6.6 for pseudo code reference.

What is important to note is that whenever the scheduler context switches to an another process, the scheduler process is enqueued to the ready queue before context switching. This ensures the scheduler is always available to context switch back to when running as a process.

Listing 6.6: Scheduler event loop pseudo code.

```
while ( true ) {
    if (exit_condition()) {
        break;
    }
    // cleanup terminated contexts
    cleanup_terminated();
    // transition contexts to ready
    transition_remote(); // remote -> ready
    wakeup_sleep();      // sleep  -> ready
    // schedule ready context if any. Else, wait
    context = scheduling_policy.dequeue();
    if (context != nullptr) {
        // scheduler must always be available
        scheduling_policy.enqueue( scheduler_context );
        // context switch to ready process
        context.resume();
        // scheduler is now running
    } else {
        // sleep until first timeout or idle wakeup
        scheduling_policy.suspend_until( next_wakeup() );
    }
}
// scheduler is exiting, cleanup scheduler
scheduler_cleanup();
// lastly, context switch to main
main_context.resume();
// never returns
```

6.3 Library Feature Implementations

Library features provide the framework for the library for which are visible for the programmer. All features use the runtime system to implement its functionality, and this section explains this in further detail.

6.3.1 Timers

The three types of timers described in Subsection 5.2.1 are represented through a common abstract class interface, shown in Listing 6.7. Instances of an egg or repeat timer convert the specified time duration to a time point, while the date timer already specifies a time point. This time point is stored in the base interface class. All timers support duration and time points from the standard library `std::chrono`.

Listing 6.7: Timer abstract class interface.

```
1  class timer::Interface {
2  protected:
3      using TimePointT = /* implementation defined */;
4      TimePointT time_point_;
5  public:
6      virtual void reset() = 0;
7      virtual bool expired() = 0;
8      bool operator<(Interface const& other) const
9      { return time_point_ < other.time_point_; }
10     TimePointT const& get() const { return time_point_; }
11 };
```

When a timer is supplied for a timed operation, the reset method is called. For the egg timer, a reset results in calculating new time point. For the repeat timer, the next periodic time point is calculated if expired, else the time point remains the same.

Multiple timers can be supplied for some operations, such as alting. Since timers have a static time point after a reset, the closest time point is chosen when there are multiple timers.

An explicit process suspension with a given timer is simply enlisting the process to the scheduler sleep queue with the corresponding time point. When the time point is reached, the scheduler will transition the process from the sleep queue

to the ready queue. The time points are checked by the scheduler process in the scheduler event loop with the `Scheduler::wakeup_sleep()` method. Suspending a process with a timer will immediately return if the time point is already reached.

A timed operation differs slightly from a timed suspension. Whenever the process waits for some event during the operation, the process is enlisted in the sleep queue. Now, one of two things may happen: either the process is rescheduled by some other process, or the operation times out and is rescheduled by the scheduler. If the process was rescheduled by some other process, the scheduler removes the process from the sleep queue and is enqueued in the ready queue. If the time point expires, the scheduler transitions the process from the sleep queue to the ready queue just as a timed suspension. Either way, the process is removed from the sleep queue and enqueued in the ready queue.

Note that if the process is rescheduled by some other process which does not share the same parent scheduler, the process is enlisted in the remote queue first, but is not removed from the sleep queue. It is not until the scheduler transitions the process from remote to ready during the `Scheduler::transition_remote()` procedure the process is removed from the sleep queue.

6.3.2 Parallel Statement

The parallel statement, following the design presented in Subsection 5.2.2, has two obvious phases: the *fork* and *join phase*.

During the fork phase, each process to be executed in parallel is spawned by the parent process, one by one. The parent process is in this context the process calling the parallel statement. Spawning involves creating the process, attaching the process to the current scheduler, and schedules the process. When all parallel process has been spawned, the parent process enters the join phase.

The join phase consists of the parent process joining all parallel processes, one by one. Joining a process involves waiting until the process has terminated. One of two things happens when joining: either the process has not terminated and is still executing, or it has terminated. If the process has terminated, the parent process continues the join phase. If not, the parent process waits. When the process

terminates, it will wake up the parent process.

Pseudo code for the parallel implementation is presented in Listing 6.8.

Listing 6.8: Parallel statement pseudo code.

```
1 /* fork phase */
2 for_each process in parallel_processes {
3     process.fork();
4 }
5 /* join phase */
6 for_each process in parallel_processes {
7     process.join();
8 }
```

The parallel statement is quite simplistic, as it only enqueues new processes to the current scheduler and waits for their termination. Much of the simplicity comes from the lack of a *sequential* statement, simplifying the design significantly.

6.3.3 Channels

Following the design in Subsection 5.2.3, channels exist in one flavour: synchronous and unbuffered, unidirectional, one-to-one, and type safe.

Channel objects are of the type `Chan<T>`, which are composed of the two channel end objects `Chan<T>::Tx` and `Chan<T>::Rx`. `Tx` and `Rx` can send and receive on a channel, respectively. Creating a channel object allocates the two channel ends with the `channel::create<T>()` method. The two channel ends are accessible via channel methods. The channel method `Chan<T>::ref_tx/rx()` returns a reference to either channel end, while the method `Chan<T>::move_tx/rx()` returns a moved channel end object. See Listing 6.9 for reference.

Listing 6.9: Channel object type.

```
1 template<typename T>
2 struct Chan : public std::tuple<Tx<T>,Rx<T>> {
3     using Tx = Tx<T>;
4     using Rx = Rx<T>;
5     using TplT = std::tuple<Tx<T>,Rx<T>>;
6     Chan() : TplT{ channel::create() } {}
7     Tx & ref_tx();  Rx & ref_rf();
8     Tx   move_tx(); Rx   move_rx();
9 };
```

Two channel containers are also supplied: `ChanArr<T,N>` for static allocation on the stack, and `ChanVec<T>` for dynamic allocation on the heap. The underlying container of `ChanArr<T,N>` is a `std::array`, and the underlying container of `ChanVec<T>` is a `std::vector`. All related methods and types associated with the underlying container is accessible with the corresponding channel container. This means channel containers support indexing with brackets, e.g. `[i]`. See Listing 6.10 for channel container type definitions.

Listing 6.10: Channel container types.

```
1  template<typename T, std::size_t N>
2  struct ChanArr : public std::array<T,N> {
3      using ArrT = std::array<T,N>;
4      using ArrT::ArrT;
5      ChanArr() : ArrT() {}
6  };
7  template<typename T>
8  struct ChanVec : public std::vector<T> {
9      using VecT = std::vector<T>;
10     using VecT::VecT;
11     ChanVec(std::size_t n) : VecT(n) {}
12 };
```

Both channel containers has the methods `collect_tx/rx()`, which returns a container of all corresponding channel ends in the container. The returned container type is the same as the underlying channel container.

Channel operations on a channel end, alting or not, can be timed with a timer. Both sending and receiving channel ends can be used in alting, compared to occam and XC which only permits receiving channel ends.

Channels can be closed. When a channel is closed, no more channel operations can be completed on the given channel. Closing a channel cannot be undone. A channel closes when either one of the channel ends goes out of scope, or one the channel ends explicitly closes the channel.

Channel ends are movable but non-copyable, meaning channel ends must explicitly pass ownership between scopes. As each process in itself is an independent running scope, channel ends being non-copyable ensures only one process holds and owns a channel end at any given time. If a process were to pass a channel end to another process, the ownership of the given channel end must be moved.

Channel ends are no more than class object holding a shared pointer to the underlying channel implementation. The channel implementation is of type `ChannelImpl<T>`, containing the entire channel functionality. Channel ends are therefore no more than wrapping types, restricting the access to the channel implementation. Whenever a channel is to be allocated, a channel implementation object is dynamically allocated with the shared pointer `std::shared_ptr`, and both channel ends are constructed with the shared pointer of the channel implementation.

The channel implementation follows the algorithm detailed in Subsection 5.3.2. Some important details regarding the channel implementation is that all channel operations, such as closing, send, receive, etc. are enclosed with a spinlock that belongs to the channel implementation object. The spinlock effectively serializes all access to the channel implementation. However, some parts of the channel end algorithm require the participant to wait. The lock is therefore released after the process is suspended, using a context switch operation. If the lock were to be released before suspending, the opposite channel end could theoretically complete the channel operation and reschedule the current channel end before it suspended.

The alting channel end implementation is slightly more complicated than the non-alting implementation, as it has a alt-to-alt synchronization procedure before trying the acquire the channel implementation lock.

6.3.4 Alting

The alting procedure is implemented as a class object of type `Alt`, which is allocated on the stack. The channel alternative is composed of two different channel alternative objects, each for the sending and receiving case. Both alternative objects inherit from a common abstract class interface. The timer and skip alternatives are both allocated as a data member in the alting object.

After creating an alting object, alternatives can be created by chaining function calls to the alting object. The four functions `send`, `recv`, `timeout` and `skip` creates a channel send, channel receive, timeout and skip alternative, respectively. The channel alternative functions have a corresponding replicator function for creating a dynamic number of channel alternatives over a dynamic number of channel ends

These are called `send_for` and `recv_for`. All alternative creating functions also has an corresponding guarded function, which is the function name with `_if` appended, e.g. `send_if`. Lastly, the function call `select` performs the alting procedure and consumes the alting object in the process. The `select` function call is always last. See Listing 6.11 for a code example.

Listing 6.11: Code example of alting.

```
1  Alt()                                // creates alting object
2      .send(tx, item)                  // w/o guard, w/o closure
3      .recv_if(cond1, rx)              // w   guard, w/o closure
4      .timeout(timer, [](){})          // w/o guard, w   closure
5      .skip_if(cond2, &some_func)      // w   guard, w   closure
6      /* more alternatives can be inserted here */
7      .select();                       // selects an alternative
```

An important observation to make is that all alternative functions simply generate and store alternatives in the alting object. However, some additional care has to be taken into consideration. Multiple alternatives can be created on the same channel end. If this is the case, then the alting object chooses one at random for which is used during the alting procedure. If both ends of the same channel are detected as alternatives, then all alternatives for that channel are discarded.

Timeout alternatives are stored as a single entry in the alting object. Whenever a timeout alternative is created, the timeout period is checked against the current timeout period, initialized to the maximum value. If the timeout period is lower than the current one, the new timeout alternative is swapped in place.

Skip alternatives are stored as a single entry in the alting object. Compared to the timeout alternative, only the first skip alternative is stored. All subsequent skip alternatives are discarded.

The alting procedure follows the algorithm presented in Subsection 5.3.3. A spinlock is used for mutual exclusion during the alting procedure to enforce active and passive selection. An atomic flag and a pointer are used to set the “winner” of the selection. Listings 6.12 and 6.13 shows an illustration of the active and passive alting.

During the checking phase, the alting object holds the lock. If the alting pro-

cedure actively selects an alternative, the atomic flag is set and the pointer is set accordingly. The alting procedure lock is only released when the procedure advances to the waiting or completing phase. Only after the alting procedure continues to the waiting or completing phase is the lock released. Any alternative that becomes ready during the checking phase and tries to select passively must acquire the alting lock first. This allows the alting procedure to enforce all alternatives trying to passively select to wait until the waiting or completing phase.

If the alting procedure enters the waiting phase, the alting procedure suspends itself and the lock is released. Note that the lock is released after the suspension, using context switch operation. The first alternative to acquire the lock and set the atomic flag is allowed to set the winner pointer. This alternative also reschedules the alting procedure.

Listing 6.12: Active alting of the alting procedure.

```
8  lock = spinlock.aquire();
9  ready_selected = checking_phase();
10 winner = if ! ready_selected {
11     if skip { skip_alternative }
12     else {
13         // atomically suspend and release lock
14         timeout = scheduler.alt_wait( lock );
15         if timeout { timeout_alternative }
16         else       { channel_alternative }
17     }
18 } else { ready_alternative };
19 completing_phase(winner);
```

Listing 6.13: Passive alting of a process.

```
1 lock = spinlock.aquire();
2 if ! alt.atomic_flag.test_and_set() { return false; }
3 alt.winner = channel_alternative;
4 scheduler.reschedule(alt.context);
5 return true;
```

The alting procedure uses a uniform random distribution to randomly choose an alternative if multiple alternatives are ready during the active phase. Using a uniform random distribution makes the alting procedure fair and non-deterministic. Whether an alting procedure prefers determinism over fairness or not is not up for

this thesis to discuss, and for all practical reasons the alting procedure favours fairness over determinism.

Chapter 7

Examples of Usage

This chapter introduces how ProXC++ is used in C++ programs. Only the basics of ProXC++ are presented, introducing all necessary features for understanding the library. First of all, ProXC++ must be installed on the work environment. ProXC++ is available for free as in speech on GitHub [50]. Following the install guide on the readme should be sufficient. For more advanced examples, check out the examples in the `examples/` folder on the GitHub project.

All library related types and methods reside in the `proxc` namespace, which will be omitted in all code examples. It is given the header file `#include <proxc.hpp>` is included in all code examples shown in this chapter, as well as in all C++ files programming with ProXC++. The linker flag `-lproxc` must also be passed during compilation.

Programs using ProXC++ does not need to initialize or cleanup the library. This is done automatically by the library. The first call to the library which invokes the runtime system will initialize the library, and the cleanup procedure is invoked when the program exits. No cleanup will be invoked unless the library has been initialized.

7.1 Processes

At the core of ProXC++ are lightweight processes. These processes are a separate scope of execution which can run concurrently with the rest of the program. In code, these processes are represented by a function and its corresponding arguments. The main function of any C++ program is also implicitly a process. This function must have a return type of `void`, but can have any type and number of arguments. Listing 7.1 shows different function prototypes where some qualify running as a process and some do not.

Listing 7.1: Different function prototypes which do and do not qualify as a process.

```
1 void good_func1();                  // ok
2 void good_func2(std::string msg); // also ok
3 int         bad_func1();      // not ok, return type not void
4 std::string bad_func2(int y); // also not ok
```

Processes are stored as process objects of the type `Process`. These process objects can be created and stored freely in any container. A process object can be created explicitly with the object constructor, which takes a function pointer and its corresponding arguments. Alternatively, a process object can be created implicitly with the library method `proc`. Listing 7.2 illustrates the difference.

Listing 7.2: Process creation.

```
1 Process my_process{&my_func, arg1, arg2};
2 auto other_process = proc(&other_func, other_arg);
```

Process creation also has perfect forwarding of arguments, meaning any type of arguments being expensive to copy or being non-copyable can correctly be moved into processes.

A dynamic number of processes can also be created, which requires to either explicitly allocate each process before hand in any container or to generate them on the fly. The library method `proc_for` takes either a pair of input iterators to any container of processes, or a pair of integers which defines the integer range [start; end⟩ and a function pointer which takes an integer as an argument. The

method returns a process which runs the dynamic numbers of processes in parallel. See Listing 7.3 for reference. Calling the `proc_for` method only creates the parallel process, and does not run them.

Listing 7.3: Creating a dynamic number of processes.

```
1  std::vector<Process> procs;
2  for (int i = 0; i < N; ++i)
3      procs.emplace_back(&some_func, some_data[i]);
4  auto total_procs = proc_for(procs.begin(), procs.end());
5  // or
6  auto total_procs = proc_for(0, N,
7      [&some_data](int i){ some_func(some_data[i]); });
```

Creating a process is not useful if it cannot be executed in parallel with the rest of the system. The library function `parallel` takes one-or-more processes and runs them in parallel, following the fork-join model. The calling process will suspend until all parallel processes have terminated. See Listing 7.4 for reference.

Listing 7.4: Example of the parallel statement.

```
1  std::vector<Process> procs;
2  // ...
3  parallel(
4      proc(&my_func, 42),
5      proc([](){ /* lambda function */ }),
6      proc_for(procs.begin(), procs.end()),
7      proc_for(0, N, &calculate_func)
8  );
```

Process objects are non-copyable but movable, meaning the ownership of process objects must be moved between scopes. A process object is of one-time use. After a process has executed and terminated, the process object cannot be run again. The process object must be constructed once more to be executed again.

In any process context a set of operations are available for the current running process, accessible in the `this_proc` namespace. The operations includes getting the current process id, yielding, and suspending for a given duration or until a time point. The suspension operations supports time types from the standard library `std::chrono`. See Listing 7.5 for reference.

Listing 7.5: Per process operations for the current executing process.

```
1 auto id = this_proc::get_id();
2 this_proc::yield();
3 this_proc::delay_for(/* duration */);
4 this_proc::delay_until(/* time point */);
```

7.2 Timers

Three types of timers are available in ProXC++: egg, repeat and date timers. Timers allow for soft real-time requirements on certain operations. All timers have support for time period and duration declarations using the standard time library `std::chrono`.

All timer objects reside in the `timer` namespace. Timer objects are constructed with a supplied appropriate duration or time point, and are both copyable and movable. See Listing 7.6 for reference.

Listing 7.6: Constructing different timers.

```
1 timer::Egg    egg{/* duration */};
2 timer::Repeat rep{/* duration */};
3 timer::Date   date{/* time point */};
```

Egg timers are used for relative timeout periods. Just as an egg timer in real life, the timer starts when the operation starts, and expires when the period has passed. The egg timer can be reused after expiration, as the countdown is reset every time it is supplied for an operation. In a sense, the timer does not "survive" between multiple operations, as it resets.

Repeat timers are used for periodic timeout periods. Given a time duration, the repeat timer will expire every period equal to the time duration. When supplied with an operation, the repeat timer does not reset. Only when it expires does it reset, basically setting the next timeout point to the next period forward in time. Repeat timers do "survive" between multiple operations, as time timer only resets when it expires.

Date timers are used for absolute timeout periods. Given a time point, the date timer will expire when current time point has surpassed the given time point.

The date timer does "survive" between multiple operations, however, always stays expired after a timeout.

7.3 Channels

Processes use message-passing to communicate, which is realized through channels. The philosophy of CSP-based concurrency is processes should not share memory directly, as this is a potent problem often leading to race conditions. Rather, the processes should share memory by communicating, i.e. message-passing with channels.

Channels in ProXC++ exist only in one flavour: synchronous and unbuffered, unidirectional, one-to-one and type safe. Channel objects are of the type `Chan<T>`, and takes no additional arguments in its constructor. Channel objects are non-copyble but movable.

A channel object is a named tuple, containing the two channel end objects `Chan<T>::Tx` and `Chan<T>::Rx`, The channel ends `Tx` and `Rx` can respectively send and receive on the channel. Channel end objects are also non-copyable but movable.

Channel ends can be access directly from the channel object via the class methods `ref_tx` and `ref_rx`, which returns a reference to the channel end objects. Channel end objects can also be moved from the channel object with the class methods `move_tx` and `move_rx`. See Listing 7.7 for reference.

Listing 7.7: Channel creation and channel end access.

```
1 Chan<std::string> channel;
2 /* channel end access */
3 channel.ref_tx();  /* and */ channel.ref_rx();
4 /* channel end move */
5 channel.move_tx(); /* and */ channel.move_rx();
```

Given a channel of type `Chan<T>`, the channel end objects `Tx` and `Rx` can send data of the type `T` over the channel. Channel transmissions has perfect forwarding of data.

The channel end object `Tx` can either send data with the class method `send` or with the overloaded `operator<<`. The class method returns the explicit result of

the operation, while the overloaded operator only returns a boolean of whether the operation was a success or not. See Listing 7.8 for reference.

Listing 7.8: Sending on the Tx channel end object.

```
1 Chan<long> channel;
2 auto tx = channel.move_tx();
3 auto op_result = tx.send(some_data);
4 /* or */
5 bool success = tx << some_data;
```

The channel end object `Rx` can either receive data with the class method `recv` or with the overloaded `operator>>`. An important difference from the sending operations is the receive data must be preallocated before calling the receive methods. However, the return types resembles the return types of sending. See Listing 7.9 for reference.

Listing 7.9: Receiving on the Rx channel end object.

```
1 Chan<long> channel;
2 auto rx = channel.move_rx();
3 long some_data; // received data must be preallocated!
4 auto op_result = rx.recv(some_data);
5 /* or */
6 bool success = tx >> some_data;
```

Both channel end objects overload `operator()` for inline sending and receiving. However, inline channel operations does not return any indications whether the channel operation was a success or not. See Listing 7.10 for reference.

Listing 7.10: Inline channel operations.

```
1 Chan<T>::Tx tx; Chan<T>::Rx rx;
2 tx(data); // returns void
3 rx();     // returns T
```

A channel has a notion of being open or closed. A channel always starts as being open. When a channel is closed, no more channel transmission can be completed. A closed channel forever remains closed until destroyed. A channel becomes closed either by a channel end explicitly closing the channel with the class method `close`,

or a channel end object goes out of scope and is deallocated. A channel end can test whether the channel is closed or not with the class method `is_closed`, which returns true for a closed channel and false otherwise.

The channel end object `Rx` supports for range-based for loops, retrieving data as long as the channel is open. When the channel becomes closed, the for loop automatically breaks. See Listing 7.11 for reference.

Listing 7.11: Range-based for loops with the receving channel end.

```
1 Chan<T>::Rx rx;
2 for (auto data : Rx) {
3     /* do some work */
4 }
5 /* here, channel is closed */
```

A receiving channel end can pipe its input into a sending channel end, meaning all data received on the receiving channel end is forwarded as the output on a sending channel end. The overloaded `operator>>` is used to pipe data between a receiving and sending channel end, which returns a boolean indicating success or not. A successful piping is only when data is successfully received and sent. If either operation fails, the piping operation fails as well. See Listing 7.12 for reference.

Listing 7.12: Piping from a receiving to sending channel end object.

```
1 Chan<T>::Tx tx; Chan<T>::Rx rx;
2 while (rx >> tx) { /* piping is successful */ }
3 /* piping failed */
```

Lastly, a channel send or receive operation can be timed. A time duration or time point from the standard library `std::chrono` can be supplied with a channel operation. The channel operation will try to complete within the specified time, and returns either if the operation completed or timed out. The return value of the timed operation specifies the reason for why the reason channel operation returned.

7.4 Alting

If a process were to wait on multiple channel ends simultaneously, normal channel operations would not suffice. The alting construct however allows a process to simultaneously wait on multiple channel operations and complete one when one becomes ready. The channel operations, together with timeouts and skips, makes up what is called alting alternatives.

The four alternatives are as follows: channel send and receive takes an appropriate channel end and data, and performs the given channel operation. A timeout specifies a timer and wait until that timer has expired. A skip is always ready, much like the default keyword in a switch construct.

The alting procedure allows a process to synchronously wait for multiple alternatives. The alting procedure block the calling process until one of the alternatives can complete, and completes that alternative. A corresponding closure, which can be anything callable, is then executed if present. When the alting procedure finishes, including the executed closure has returned, the process resumes.

An alting procedure consists of zero-or-more alternatives, where the alting procedure waits for these alternatives to become "ready". When one-or-more alternatives are ready, the alting procedure chooses one alternative and executes a corresponding closure. Alternatives can be guarded by a boolean condition, which enables or disables an alternative for the alting procedure depending on the condition.

An alting procedure starts with creating an alting object, which has the type `Alt`. This alting object can create alternatives by function chaining multiple alternative methods, and lastly, call a selection method which consumes the alting object and resolves the alting procedure. See Listing 7.13 for reference.

The four alternative types channel send, channel receive, timeout and skip has the corresponding alternative methods `send`, `recv`, `timeout` and `skip`, respectively.

Each alternative method can prepend a guard with a boolean condition by calling the altered alternative method, which has an appended `_if` to the method name, e.g. `send_if`. All alternative methods, both with and without a guard, can have an optional closure. See Listing 7.13 for reference.

Listing 7.13: Example of the alting construct.

```
1  Alt()
2      .send(tx, 42)
3      .recv_if(cond1, rx)
4      .timeout(timer, &timeout_func)
5      .skip_if(cond2, [](){
6          /* skip lambda */
7      })
8      .select();
```

A dynamic number of channel send and receive alternatives can be generated. The class methods `send_for` and `recv_for` takes a pair of iterators to any container with the appropriate channel end object, and generates a corresponding alternative for each channel end. The class methods can also be guarded, following the same naming scheme as the single alternative methods.

The timer alternatives takes on of the three timer types presented in Section 7.2.

Chapter 8

Performance

Benchmarking performance of a concurrency library is not intuitive. Some might even argue performance is not important, but rather the abstractions are correct. However, some metrics of parallel programs can be tested.

For this benchmark, a set of concurrent programs are tested with various degrees of parallelism. Metrics such as concurrent throughput, sequential speed, and load balancing are factors which can influence performance.

To have some appropriately comparable benchmarks, entries from Section 3.2 are included. Occam and XC are however not included, as they require proprietary hardware to run on multi-core architectures. C++CSP2 is not included because of difficulties of implementing benchmark tests which did not result in spurious segmentation faults by the library during execution. Boost.Fiber is not included because the lack of the necessary CSP abstractions to create the benchmarks.

Additionally, single-core versions of ProXC++ and Go are included to observe the potential speedup in performance compared to multi-core. ProXC++ implements a single-core runtime system with a round robin scheduling policy, while Go implements single-core runtime system by setting the number of schedulers to one with `GOMAXPROCS`. C++CSP has no support for a single-core runtime system, and therefore only the multi-core version is included.

All code for the different benchmark tests can be found in appendix B.

8.1 Benchmark Setup

All benchmarks performed in this chapter are computed on the same machine; a desktop PC with an Intel® Core™ i7-4790 4GHz processor with a total of 8 logical cores (4 physical cores with 2 logical cores per physical core), 16GiB DDR3 RAM, running Ubuntu® 16.04 xenial, x86_64 Linux® 4.4.0 as operating system.

Intel Core is a registered trademark of Intel Corporation, Linux is the registered trademark of Linus Torvaldsen in the U.S. and other countries, Ubuntu is a registered trademark of Canonical Ltd.

8.2 Benchmark Tests

Three type of benchmark tests are performed: *extended commstime*, *concurrent mandelbrot* and *concurrent prime sieve*. These three tests have various degrees of sequential and parallel characteristics, aiming to highlight the concurrency adaptation capabilities, the concurrent throughput, and how well each entry scales with an increasing amount of parallel work load.

8.2.1 Extended Commstime

The extended commstime test is a custom derivation of the original commstime test. The original commstime test [51] is a pseudo-standard benchmark for testing sequential channel communication between processes. Three processes called *prefix*, *delta* and *successor* creates a channel cycle, sending an integer in loops. Each loop increments the integer. A fourth process, called *consumer*, receives the integer on each loop cycle. The consumer receives an integer a number of times and calculates the average time to receive the integer. The commstime test mostly gives a metric for the overhead regarding channel communication.

Channel communication overhead is not that interesting of a metric with dynamic multithreading. Commstime is however a good metric for a programs adaptability of sequential programs, as the four processes in commstime creates a sequential dependency. I therefore propose the *extended commstime* test. Instead of three processes in a channel communication cycle, N processes are created in a chain of

channel communications, creating a variable sized cycle.

The extended commstime varies the chain length from $N = 1$ to 1000. For each chain length value of N, the time to receive 100 integers divided by the chain length is averaged over 50 runs.

The results of the extended commstime test are presented in Figure 8.1.

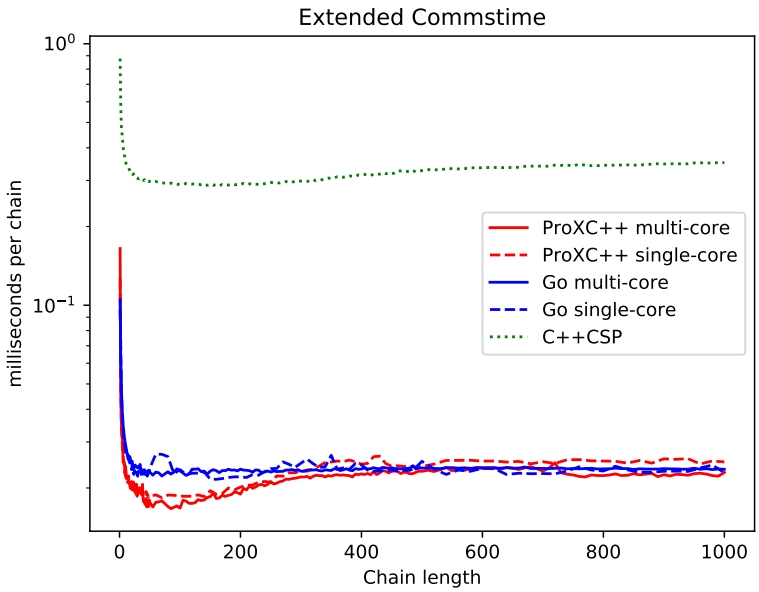


Figure 8.1: Results of extended commstime. The y-axis has a logarithmic scaling. Lower values are better.

8.2.2 Concurrent Mandelbrot

Some problems are embarrassingly parallel [52], where little to no effort is needed to separate the work load into parallel tasks. The Mandelbrot set is a perfect example of an embarrassingly parallel problem, where each point in the set can be calculated independently of each other.

Generating a Mandelbrot set is perfect for testing the parallel capabilities of the entries, such as how good the load balancing is.

For this benchmark, the Mandelbrot set is computed in the domain $x \in [-2.1; 1.0]$ and $y \in [-1.3; 1.3]$ with a resolution of a given dimension D. The set is split into D

number of lines on the y-axis, where each line consists of D points evenly distributed along the x-axis. Each line represents some parallel unit of work.

All lines are computed in a map-reduce manner, where some worker processes each calculates a full line at a time. Two approaches will be tested: a hard coded optimal distribution of work, and dynamic sub-optimal distribution of work.

Both approaches tests for the same; for a given dimension from $D = 1$ to 1000, how long does it take to calculate all lines, averaged over 100 rounds.

The hard coded optimal distribution is implemented by having the number of worker processes equal to the number of logical cores, 8 in this case. Each worker process receives a line to calculate from a channel, where a producer process is on the other end continuously sending new lines to calculate. A consumer process receives finished calculated lines on a channel from the worker processes.

Note that for the single-core versions, only 1 worker process is spawned as only a single core is available.

The results of the hard coded Mandelbrot test are presented in Figure 8.2.

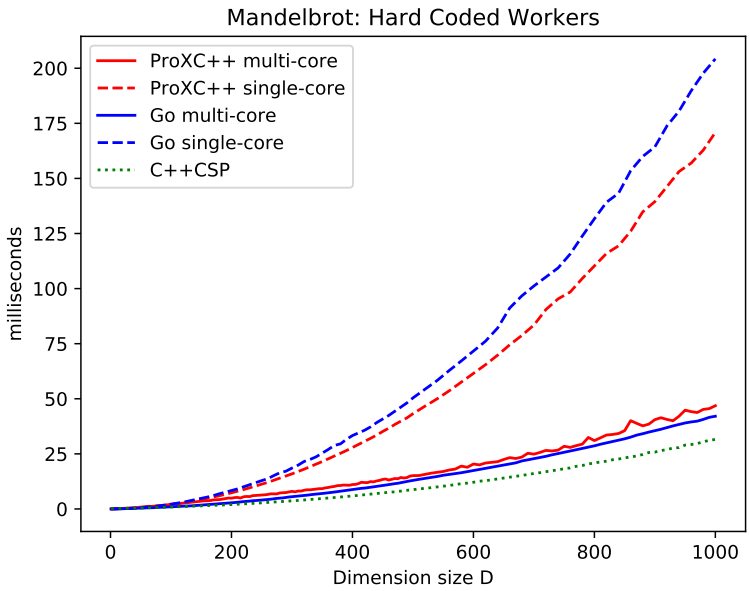


Figure 8.2: Results of concurrent Mandelbrot with hard coded number of workers. Lower values are better.

The dynamic sub-optimal distribution is implemented by having worker processes each dedicated for calculating a single line, meaning a total of D worker processes will be spawned for a dimension D. All worker processes will be spawned simultaneously. A consumer process will receive finished calculated lines on a channel from the worker processes, just as the hard coded Mandelbrot test.

The results for the dynamic Mandelbrot test are presented in Figure 8.3.

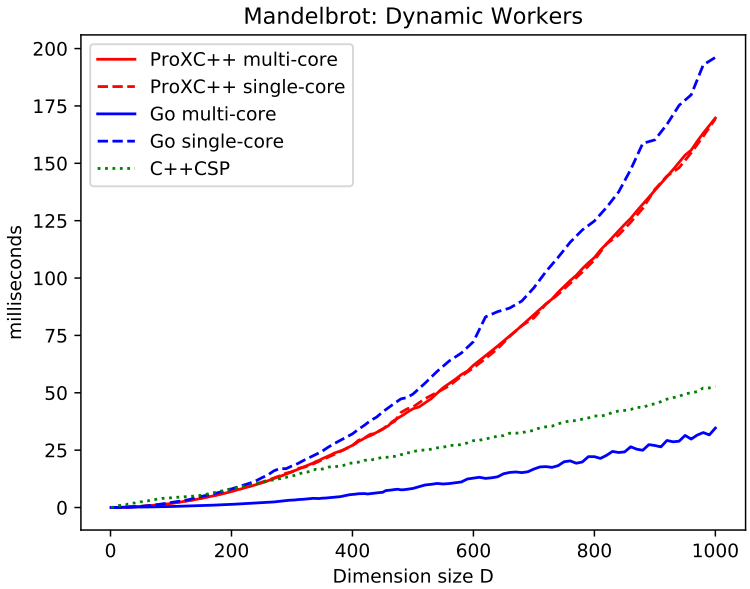


Figure 8.3: Results of concurrent Mandelbrot with dynamic number of workers. Lower values are better.

8.2.3 Concurrent Prime Sieve

The last benchmark is the concurrent prime sieve, popularized by the famously elegant piece of concurrent code in Go [53]. A prime sieve is a fast type of algorithm to find prime numbers, usually implemented as a sequential algorithm. A concurrent prime sieve is also an algorithm to find prime numbers, however daisy-chains processes to determine whether a number is a prime or not.

At the start of the daisy-chain is the *generator* process, which generates all numbers from 2 and above. Along the daisy-chain is *filter* processes, where each

filter represents a prime number along the number line. When the filter receives a number, the divisibility of that number is checked against the filters prime number. A non-divisible number is passed along the daisy-chain, while a divisible number is discarded. Given a daisy-chain of N filters, at the end of the daisy-chain is the Nth prime.

Note that the concurrent prime sieve algorithm is nowhere near as efficient as the sequential counterpart. The interesting merit of the concurrent prime sieve algorithm is the total concurrent throughput. As multiple integers can be pipelined simultaneously in the daisy-chain, it is possible to sieve multiple integers in parallel.

This benchmark test generates N prime numbers, where $N = 10$ to 1000. Given a N, the execution time per prime number is calculated over the average running time of 10 runs.

The results of the concurrent prime sieve test are presented in Figure 8.4.

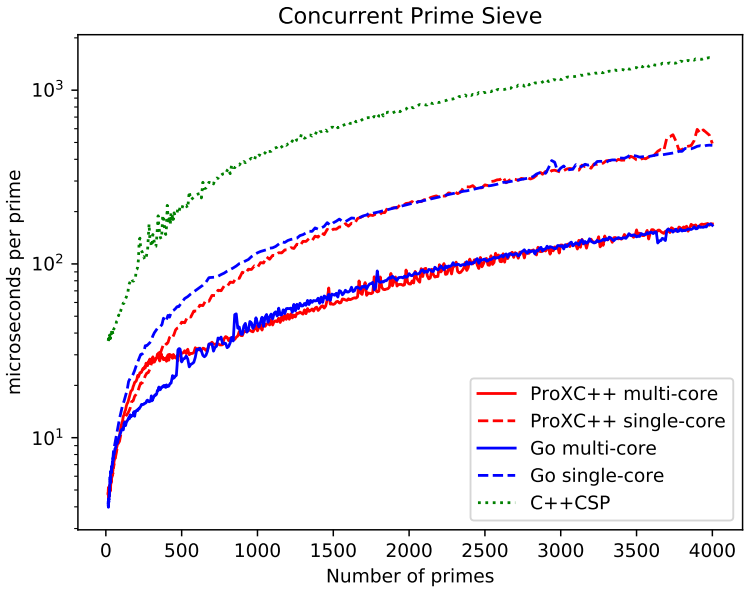


Figure 8.4: Results of concurrent prime sieve. The y-axis has a logarithmic scaling. Lower values are better.

8.3 Analysis

Starting with the extended commstime test, the results in Figure 8.1 shows both multi-core and single-core versions ProXC++ and Go equals in execution time. A small exception is ProXC++ being slightly faster than Go in the range $n \in [1; \sim 300]$. As the chain length increases, both ProXC++ and Go converges to a fixed execution time per chain, showing a linear increase in total execution time.

The single-core versions of both entries were expected to yield best results, as extended commstime is nothing but a long sequential dependent cycle of processes. That both multi-core versions matches, and sometimes surpasses, the single-core versions is a good result. C++CSP behaves very similar to ProXC++ and Go, however at a very much worse execution time per chain.

The results of the hard coded Mandelbrot test in Figure 8.2 were interesting, but expected. The multi-core and single-core versions of both ProXC++ and Go respectively increase in execution time with the same exponential trend. The single-core version was expected to perform worst, while the multi-core versions were expected to have a much better execution time. What is surprising is C++CSP being the best entry here. When thinking about it, C++CSP does have the most optimal setup here. Since each worker process in C++CSP is a kernel-thread, each thread can run independently of each other on each core. Considering C++CSP is the best possible outcome, both multi-core version of C++CSP and ProXC++ yields a decent result.

The results of the dynamic Mandelbrot test in Figure 8.3 however shows a different side. The same trends for the single-core versions can be observed. C++CSP slightly doubles in execution time, while Go almost matches the results of the hard coded Mandelbrot test for C++CSP. The big surprise is now multi-core ProXC++ equals single-core ProXC++. Further inspections of the running test processor utilization reveals the runtime system only fully utilizes a single processor core. What is happening is that when the worker processes are spawned, everyone is placed on the same ready queue on the same parent scheduler. All other idle schedulers fail to effectively steal work of the one parent scheduler, resulting in only one scheduler having work to run. This test clearly highlights that ProXC++ has issues with

effectively distributing a huge number of small work across the idle schedulers.

Lastly, the results of the concurrent prime sieve test in Figure 8.4 are probably the most promising results. Fully utilizing the parallel nature in the concurrent prime sieve algorithm is very unintuitive, and heavily relies on the runtime system to detect ready work and effectively distribute said work among idle schedulers. The results show that the single-core and multi-core versions of ProXC++ and Go both follow the same trend, where the multi-core versions yields a much better result than the single-core versions. C++CSP is a lot slower than the single-core versions of ProXC++ and Go, showing the lack of concurrent throughput in C++CSP.

An interesting observation to make is between the prime range $n \in [1; \sim 500]$ ProXC++ is a great deal slower than Go until it suddenly drop to equal. A possible explanation is not able to effectively distribute the ready work as the processes are too short lived, sort of similar to the same issue withe the dynamic Mandelbrot test. However, over a certain limit around ~ 500 processes the schedulers are able to steal the processes.

Part III

Discussions

Chapter 9

ProXC++ vs. ProXC

As ProXC++ is a continuation of the project ProXC [5], it is interesting to see the different design choices and capabilities the two projects exhibits.

This chapter compares and discusses the differences between the two projects ProXC++ and ProXC, which includes differences such as abstractions, library features, usability, and performance. This comparison should provide an insightful look into both libraries, as both cater to the same motivation; providing a CSP abstraction for a programming language with no native support for CSP-based concurrency.

9.1 Similarities and Differences

Table 9.1 gives a rough comparison between ProXC++ and ProXC. The main difference to take from the comparison is the more dynamic feature set ProXC++ provides compared to the feature set of ProXC. In ProXC++, a dynamic number of processes can be spawned in parallel, a dynamic number of alternatives can be alted on, etc. This dynamic approach allows to create more flexible and less hard coded programs, and consequently creating more maintainable and expressive concurrent systems.

	ProXC++	ProXC
Target langauge	The C++ programming language.	The C programming language.
Lightweight processes	Third party library, portable, customizable stack types.	Handwritten, not portable, hard coded stack types.
Process spawning	Synchronous parallel spawning of dynamic number of processes.	Synchronous/asynchronous nested parallel and sequential process spawning of fixed number of processes.
Channels	Synchronous and unbuffered, unidirectional, one-to-one, type safe.	Synchronous and unbuffered, bidirectional, any-to-any, size safe.
Alternation	Alting on dynamic number of alternatives. Alternatives include channel send and receive, timeouts and skip.	Alting on fixed number of alternatives. Alternatives include channel receive, timeouts and skip.
Threading model	M:N, hybrid threading model. Support for multi-core.	M:1, user threading model. No support for multi-core.

Table 9.1: Comparison of library specification and features between ProXC++ and ProXC

9.2 Various Capabilities

A big difference between the two is the threading model used, where ProXC++ uses a hybrid threading model while ProXC uses a user threading model. The use of threading models only affects the performance in concurrent programs. The same concurrent program running on both ProXC++ and ProXC should behave just the same. However, since the main development in processor architectures is in multi-core, there is an incentive to use the available resources for a potential performance gain.

Now, not everything is about performance. Some might argue correct and expressive abstractions are more important than performance. The abstractions available in ProXC++ and ProXC are both based on CSP. However, ProXC++ has a greater expressive power than ProXC because of replicators for parallel spawning and alternatives for alting. Due to this limitation, ProXC cannot express a dynamic numbers of processes and alternatives for alting.

Chapter 10

Challenges with a Multi-Core Library

Creating a dynamic multithreaded CSP library for multi-core architectures is mostly motivated by the potential performance increase in concurrent systems by utilizing the available cores. CSP is about creating a correct and expressive abstraction over concurrent systems rather than performance, but it is tempting to exploit the apparent parallelism in multi-core architectures because of the inherently parallel nature of CSP. Therefore, the rhetorical question is as follows: is the difficulty of implementing a dynamic multithreaded CSP library worth it?

10.1 Critical Sections

What separates a single-core vs. a multi-core implementation of a user-threaded runtime system is the ability with single-core runtime systems to reason and control which states processes are in, when they are running, etc. This reasoning is especially important in critical regions of the runtime system, being able to justify certain states in algorithms based on process states.

For a single-core runtime system, any critical regions and side effects can be fully reasoned about. Since only one process runs at any given moment, code running between descheduling points essentially becomes a mutual exclusion. Multi-core

runtime systems cannot follow the same reasoning, since it has much less control whether a given process is currently running on another processor core not. Critical regions in the runtime system must most likely enforce some sort of mutual exclusion.

10.2 Choice of Mutual Exclusion Locks

What kind of locks a multi-core runtime system uses to enforce mutual exclusion is also important. Different locks and their variations, such as OS mutexes, spinlocks, read-write locks, and so forth have a different impact on performance in different situations. For instance, given a lock is held for are short-term held, then spinlocks are preferred. For locks with high contention, meaning multiple actors are trying to acquire the lock simultaneously, some type of non-linear back off procedure is needed. Either way, choosing what kind of lock is suitable for a given situation requires knowledge of different locks and the situation itself.

10.3 Non-Blocking vs Mutual Exclusion Design

Sometimes, it is desirable to design a non-blocking design rather than mutual exclusion design. A non-blocking design is usually much more demanding to develop than a mutual exclusion design, as it is harder to prove the design is correct. Non-blocking design is often preferable over mutual exclusion, as it both scales better with a number of processor cores and has a better throughput, but does have a higher latency per operation wise compared to mutual exclusion.

10.4 Pinning Kernel-Threads to Processor Cores

Multi-core runtime systems must resort to some sort of schedulers, each running on a kernel-thread. A factor to consider is whether to set thread affinity for each kernel-thread, i.e. pinning each kernel-thread to a different processor core. The idea is to avoid the operating system from migrating kernel-threads between processor cores, which causes discrepancies in the runtime system. The counter argument is

the operating system can help with load balancing kernel-threads, when for instance a scheduler with a high work load runs on a less used processor core.

Chapter 11

Shortcomings and Limitations

Much of the shortcomings and limitations present in ProXC has been improved upon in ProXC++, which includes enforcing correct usage and the portability issues with the user-thread implementation. However, some issues are present in the current state of ProXC++.

11.1 Enforcing Correct Usage

ProXC++ goes to great lengths to enforce correct usage by using much of the semantic facilities present in the C++ programming language. However, all problems existing in C++ programs, such as null pointer dereferencing and memory leaks, are possible in ProXC++ programs.

It is impossible to create a system which always generates and enforces correct program behaviour, and somewhere does the line have to be drawn. For instance, channels in ProXC++ are one-to-one, and if multiple processes where to operate on the same channel end simultaneously, the result would be undefined behaviour. Channel ends have therefore been made non-copyable to enforce the channel end only having a single owner and user. It is however very easy to bypass this by sending a reference of the channel end when spawning a new process, making

multiple users still possible. Having shared memory between processes is also entirely possible, but is highly discouraged.

ProXC++ is all about providing a safe framework for creating concurrent systems. Creating a complete safe framework is impossible, as the foundation of C++ is inherently unsafe. Therefore, ProXC++ only enforces correct usage to some degree, and let the rest be up to the programmer.

11.2 Inefficient Load Balancing

ProXC++ uses work stealing for load balancing work between schedulers. A scheduler continues to run work as long as it has ready work. However, when a scheduler runs out of work and becomes idle, it tries to steal ready work from other schedulers. How these idle schedulers decide when to steal, how much ready work to steal, and so forth is not optimal nor efficient.

Currently, when a scheduler becomes idle it tries to steal once from a random scheduler. If the steal is successful it resumes the stolen work. However, if the steal fails the scheduler sleeps for 1 millisecond and tries again when it wakes up. There is no coordination between the schedulers, as a scheduler with lots of ready work has no way to indicate to other schedulers to steal from it.

This sort of brute-force approach to find ready work is of course not desirable, as a concurrent program with few parallel tasks will generate lots of unnecessary wake-ups of from idle scheduler trying to find work. This periodical wake-up can also cause unnecessary migration of tasks between schedulers when few sequential dependent tasks are running, which was highlighted in the concurrent Mandelbrot test in Subsection 8.2.2.

The unnecessary migration of tasks and wasteful wake-ups becomes negligible after an amount of parallel semi-independent tasks is surpassed in the program.

11.3 Wasteful use of Resources

The ProXC++ runtime system has to manage a great lot of resources, including user-thread management and inter-process communication. Much of the resources

used with user-thread management are one-time use, meaning nothing is reused of stack or control data structures related to the user-threads. This one-time use creates lots of unnecessary allocations of dynamic memory which could have been reused for later use.

A smarter runtime system could detect these one-time use resources and reuse the resource rather than deallocating them. This, of course, requires a much more complex resource management by the runtime system, and could potentially introduce a greater overhead.

11.4 Determinism and Real-Time Characteristics

An important selling point of CSP is the ability to reason how deterministic a CSP model is, usually via refining the model against a specification model. The very same attribute, determinism, is also important in real-time systems. A real-time system is where the time it takes to complete an operation is just as important as the result of the operation, and can have direct consequences of the system functionality.

The question is whether ProXC++ qualifies to be used for any real-time requirements. The abstractions provided by ProXC++ are very much deterministic, except for alting. Alting uses a pseudo-random distribution to choose an alternative when multiple alternatives are ready. The reasoning behind using a pseudo-random distribution is it favours fairness over determinism.

Another factor of determinism in ProXC++, which is unrelated to CSP, is dynamic multithreading. As the runtime schedulers rely on work stealing for distributing parallel work, the deterministic characteristic of the runtime system is affected. An idle scheduler chooses its victim by random, which consequently makes a process being stolen or not random. If a time critical operation relies on certain processes to be distributed among the schedulers to meet its deadline, it is inherently non-deterministic whether the deadline is met or not.

Chapter 12

Future Work

There are some opportunities for future work for which ProXC++ could benefit from. This chapter discusses the biggest potential contenders for future work, which has been detected during the project development.

12.1 More Efficient Runtime System

The desire for an efficient runtime system is a matter of course. An optimal library would be as efficient as possible, but is rarely the case. There are however some aspects of the current implementation of the runtime system that can be directly addressed as inefficient or sub-optimal, which could potentially increase the performance and overall processor utilization.

First, the work stealing protocol used by the schedulers is not working properly. Ready work is not properly being distributed if the type of work is short lived or rarely deschedules. Schedulers are also inefficient at finding ready work if they are idle. Currently, when a scheduler tries to steal work and fails, it sleeps for 1 millisecond and tries again. This is obviously not a good approach, as this generates unnecessary CPU time. Therefore, the work stealing protocol between schedulers has potential for improvement.

Regarding the stealing part, a potential improvement is to steal half the available work rather than one, which is what Go does. Given a runtime system with N

schedulers, a scheduler with X work can effectively distribute the total work on all schedulers with a minimum number of $N-1$ steals. Compared to only stealing one work at a time, a minimal amount of $X(N-1)/N$ steals must occur to effectively distribute the total work. If X is large, it is obvious to see the first approach is desirable.

Secondly, allocation of processes and process stacks could be improved. Currently, nothing is reused when destroying a process after returning. Stack allocations could be pooled, which is supported by the Boost.Context library.

Lastly, the alt-to-alt synchronization between an alting channel send and alting channel receive is horribly inefficient. This has mostly to do with a poor design of two alting processes selecting each other, where an alting channel end has to inefficiently spin with the channel lock acquired.

12.2 C Wrapper API

ProXC++ is, first of all, a C++ library. It would however be interesting and potentially useful be able to use ProXC++ for C code as well, which would require to create a C wrapper API for ProXC++.

Now, why would it be desirable to use ProXC++ in C code when ProXC is readily available for creating CSP programs in C? The main arguments are ProXC++ is portable and has multi-core support, both of which would be much more challenging to implement in ProXC.

The biggest hurdle of creating a C wrapper API would be translating C++ semantics to C. ProXC++ uses a lot of of templates and move semantics to enforce both type safety, generalized processes, and unique ownership of channel ends. Such semantics have no intuitive translation, and a more restrictive and generalized wrapper API has to probably be created to fully work in C.

12.3 Improving on the Library Feature Set

The feature set provided by ProXC++ is a good foundation for creating any type of concurrent program with CSP abstractions. However, some library features could

potentially be a good extension to the current feature set.

Extended channel input or output [54], also called extended rendezvous, is about extending the synchronization in a channel send and receive operation. An extended rendezvous allows the sender or receiver to perform additional operations after the channel transmission has occurred, but before both channel ends resumes execution. One could see this as an "invisible" middle-man process performing some operation without breaking the existing synchronization.

Class defined processes, same as in C++CSP2 [7], could be an useful addition. Defining a process from a class point of view rather than a function allows for a more rigid definition of processes. A well-defined constructor, destructor and execution body can be specified, as well a more intuitive syntax can be used, i.e. calling a class constructor rather than a process constructor or a function allocator.

Asynchronous I/O operations are not supported with ProXC++. Any call to a blocking system call effectively blocks the entire kernel-thread, halting any progress of processes residing on the same kernel-thread. Go has support for asynchronous I/O operations, letting a scheduler with its processes continue on another kernel-thread while the blocking process halts the current kernel-thread. Asynchronous I/O could be used to implement support for networking and other blocking operations, but would require extensive development of the current implementation.

Chapter 13

Conclusion

Concurrent programming has long existed before multi-core architectures entered the mainstream market. With the ever increasing use of multi-core architectures, the demand for software which are scalable and fully utilize said processors are becoming more evident, and concurrent programming is the tool to realize such software.

For many programmers, it is hard to write correct concurrent systems with concurrent programming, and especially fail to utilize the available multiple processor cores. Communicating Sequential Processes (CSP) is a formal language for describing concurrent systems, which provides a safer yet expressive abstraction level by limiting all communication between processes to message-passing. CSP also allows to reason about the correctness of the concurrent system, whether certain specifications are met, etc. CSP-influenced concurrency has long been proven to serve as a powerful abstraction for creating concurrent systems.

This thesis argues that by combining the parallel nature of CSP with a dynamic multithreaded runtime system, correct and expressive high-performance concurrent programs can be created, which are able to fully utilize and scale with the increase of required parallelism in multi-core architectures.

The work presented in this thesis is the development of a design and implementation of ProXC++: a CSP-inspired concurrency library for C++. ProXC++ uses dynamic multithreading to effectively utilize all available logical cores on multi-core

architectures. The dynamic multithreading is implemented by having a number of runtime schedulers, equal to the available logical processor cores, each running on its own kernel-thread. The runtime schedulers employs work stealing for load balancing work dynamically.

ProXC++ and other CSP-influenced concurrency libraries with dynamic multithreading were benchmarked. The results from the benchmarks showed ProXC++ to be quite performant compared to the well-established programming language Go. However, some results highlighted underlying issues with the runtime system, especially how the current work stealing scheme for the runtime schedulers were flawed in certain situations.

The work done on ProXC++ should prove useful for designing and implementing runtime systems for dynamic multithreaded CSP-based concurrency frameworks. Especially, the design and implementation of the runtime scheduler could in itself be useful for other dynamic multithreaded concurrency frameworks, and not necessarily only for CSP-based concurrency frameworks.

A set of potential candidates for future work was identified and outlined for ProXC++. Improving the runtime scheduler and the work stealing efficiency is probably the most sought after improvement, which would only increase the multi-core performance of the library. Another sought out feature is asynchronous IO operations, which would open up implementing support for networking.

In conclusion, CSP has been proven to provide powerful abstractions for which can be exploited by multi-core processors. There is no denying dynamic multithreaded frameworks are the future of high-performance systems on multi-core architectures, and ProXC++ is one more example showing the potentiality CSP has on such systems.

13.1 Availability

ProXC++ is available online, open-source and free as in speech, on GitHub [50]. Any contributions to the library are welcomed. Any inquiries regarding this thesis or the library can contact the author via mail, edvard.pettersen@gmail.com, or through the project page on GitHub.

13.2 Acknowledgment

I would like to thank Øyvind Teig for the ideas and definitions of the egg, repeat and date timers.

Appendix A

Acronyms

API	Application Programming Interface
CSP	Communicating Sequential Processes
FIFO	First In, First Out
ILP	Instruction-Level Parallelism
MPMC	Multiple-Producer-Multiple-Consumer
MPSC	Multiple-Producer-Single-Consumer
OS	Operating System
SPMC	Single-Producer-Multiple-Consumer
SPSC	Single-Producer-Single-Consumer

Appendix B

Listing of Benchmark Code

Extended Commstime: ProXC++

```
1  #include <algorithm>
2  #include <chrono>
3  #include <iostream>
4  #include <proxc.hpp>
5
6  using namespace proxc;
7  using ItemT = std::size_t;
8  using ChanT = Chan< ItemT >;
9
10 constexpr std::size_t REPEAT = 100;
11 constexpr std::size_t RUNS = 50;
12
13 void chainer( ChanT::Rx in, ChanT::Tx out ) {
14     for ( auto i : in ) {
15         out << i + 1;
16     }
17 }
18
19 void prefix( ChanT::Rx in, ChanT::Tx out ) {
20     out << std::size_t{ 0 };
21     for ( auto i : in ) {
22         out << i;
```

```
23      }
24  }
25
26  void delta( ChanT::Rx in, ChanT::Tx out, ChanT::Tx out_consume ) {
27      for ( auto i : in ) {
28          if ( out_consume << i ) {
29              out << i;
30          } else {
31              break;
32          }
33      }
34  }
35
36  void consumer( ChanT::Rx in ) {
37      for ( std::size_t i = 0; i < REPEAT; ++i ) {
38          in();
39      }
40  }
41
42  void commstime( std::size_t chain ) {
43      std::size_t sum = 0;
44      for ( std::size_t run = 0; run < RUNS; ++run ) {
45          ChanT pre2del_ch, consume_ch;
46          ChanVec< ItemT > chain_chs{ chain + 1 };
47          std::vector< Process > chains;
48          chains.reserve( chain );
49          for ( std::size_t i = 0; i < chain; ++i ) {
50              chains.emplace_back( chainer,
51                  chain_chs[i+1].move_rx(),
52                  chain_chs[i].move_tx() );
53          }
54          auto start = std::chrono::system_clock::now();
55          parallel(
56              proc( prefix, chain_chs[0].move_rx(), pre2del_ch.move_tx↩
                    () ),
57              proc( delta, pre2del_ch.move_rx(), chain_chs[chain].↩
                    move_tx(), consume_ch.move_tx() ),
58              proc( consumer, consume_ch.move_rx() ),
59              proc_for( chains.begin(), chains.end() )
60          );
```

```
61          auto end = std::chrono::system_clock::now();
62          std::chrono::duration< std::size_t, std::nano > diff = end -↩
                start;
63          sum += diff.count();
64      }
65      std::cout << chain << "," << sum / RUNS << std::endl;
66  }
67
68  int main() {
69      for ( std::size_t chain = 1; chain < 50; chain += 1 ) {
70          commstime( chain );
71      }
72      for ( std::size_t chain = 50; chain < 500; chain += 5 ) {
73          commstime( chain );
74      }
75      for ( std::size_t chain = 500; chain <= 1000; chain += 10 ) {
76          commstime( chain );
77      }
78      return 0;
79  }
```

Extended Commstime: Go

```
1   package main
2   import (
3       "fmt"
4       "time"
5       "sync"
6   )
7   const (
8       REPEAT = 100
9       RUNS = 50
10  )
11  func chainer(wg *sync.WaitGroup, in <-chan uint, out chan<- uint) {
12      for item := range in {
13          out <- item
14      }
15      close(out)
```

```
16      wg.Done()
17  }
18
19  func prefix(wg *sync.WaitGroup, in <-chan uint, out chan<- uint) {
20      out <- 0
21      for item := range in {
22          out <- item + 1
23      }
24      close(out)
25      wg.Done()
26  }
27
28  func delta(wg *sync.WaitGroup, in <-chan uint, out, out_consume chan↩
        <- uint, ex chan bool) {
29      running := true
30      for running {
31          item := <-in
32          select {
33          case out_consume <- item:
34              out <- item
35          case <-ex:
36              close(out)
37              running = false
38          }
39      }
40      wg.Done()
41  }
42
43  func consumer(wg *sync.WaitGroup, in_consume <-chan uint, ex chan ↩
        bool) {
44      for i := 0; i < REPEAT; i++ {
45          <-in_consume
46      }
47      close(ex)
48      wg.Done()
49  }
50
51  func commstime(chain int) {
52      sum := int64(0)
53      for run := 0; run < RUNS; run++ {
```

```
54          var wg sync.WaitGroup
55          wg.Add(3 + chain)
56          del2pre_ch := make(chan uint)
57          consume_ch := make(chan uint)
58          chain_chs := make([]chan uint, chain + 1)
59          for i := range chain_chs {
60              chain_chs[i] = make(chan uint)
61          }
62          ex_ch := make(chan bool)
63          start := time.Now()
64          go prefix(&wg,   chain_chs[0], del2pre_ch)
65          go delta(&wg,    del2pre_ch,   chain_chs[chain], consume_ch,↩
                ex_ch)
66          go consumer(&wg, consume_ch,   ex_ch)
67          for i := 0; i < chain; i++ {
68              go chainer(&wg, chain_chs[i + 1], chain_chs[i])
69          }
70          wg.Wait()
71          sum += time.Since(start).Nanoseconds()
72      }
73      fmt.Println(chain, ",", sum/RUNS)
74  }
75
76  func main() {
77      for chain := 1; chain < 50; chain += 1 {
78          commstime(chain)
79      }
80      for chain := 50; chain < 500; chain += 5 {
81          commstime(chain)
82      }
83      for chain := 500; chain <= 1000; chain += 10 {
84          commstime(chain)
85      }
86  }
```

Extended Commstime: C++CSP

```
1  #include <chrono>
```

```
2  #include <functional>
3  #include <iostream>
4  #include <sstream>
5  #include <vector>
6  #include "csp.h"
7
8  using namespace csp;
9  using Chan = one2one_chan< std::size_t >;
10 using Tx = chan_out< std::size_t >;
11 using Rx = chan_in< std::size_t >;
12
13 void chainer( Rx in, Tx out ) {
14     for ( ;; ) {
15         out( in() );
16     }
17 }
18
19 void prefix( Rx in, Tx out ) {
20     out( 0 );
21     for ( ;; ) {
22         out( 1 + in() );
23     }
24 }
25
26 void delta( Rx in, Tx out, Tx out_consume ) {
27     for ( ;; ) {
28         std::size_t item = in();
29         out( item );
30         out_consume( item );
31     }
32 }
33
34 int main( int argc, char *argv[] ) {
35     if ( argc != 2 ) { return 1; }
36     std::size_t chain;
37     std::stringstream{ argv[1] } >> chain;
38     Chan consume_ch, pre2del_ch;
39     std::vector< Chan > chain_chs( chain + 1 );
40     std::vector< std::function< void() > > in_chain;
41     in_chain.reserve( chain );
```

```
42      for ( std::size_t i = 0; i < chain; ++i ) {
43          in_chain.push_back( make_proc( chainer,
44              chain_chs[i+1], chain_chs[i] ) );
45      }
46      auto start = std::chrono::system_clock::now();
47      par{
48          par{ in_chain },
49          make_proc( prefix,     chain_chs[0], pre2del_ch ),
50          make_proc( delta,      pre2del_ch, chain_chs[chain], ↩
                consume_ch ),
51          make_proc([&]( Rx in_consume ){
52              constexpr std::size_t repeat = 100;
53              for ( std::size_t i = 0; i < repeat; ++i ) {
54                  in_consume();
55              }
56              auto end = std::chrono::system_clock::now();
57              std::chrono::duration< std::size_t, std::nano > diff = ↩
                    end - start;
58              std::cout << chain << "," << diff.count() << std::endl;
59              std::exit(0);
60          }, consume_ch)
61      }();
62      return 0;
63  }
```

Hard Coded Mandelbrot: ProXC++

```
1   #include <algorithm>
2   #include <array>
3   #include <chrono>
4   #include <iostream>
5   #include <vector>
6   #include <proxc.hpp>
7
8   using namespace proxc;
9
10  constexpr std::size_t NUM_WORKERS = 8;
11  constexpr std::size_t ROUNDS = 100;
```

```
12 constexpr std::size_t MAX_ITER = 255;
13 /* x in [-2.1; 1.0] */
14 /* y in [-1.3; 1.3] */
15 constexpr double XMIN = -2.1;
16 constexpr double XMAX =  1.0;
17 constexpr double YMIN = -1.3;
18 constexpr double YMAX =  1.3;
19
20 struct MandelbrotData {
21     std::size_t line;
22     std::vector< double > values;
23     MandelbrotData() = default;
24     // make non-copyable
25     MandelbrotData( MandelbrotData const & ) = delete;
26     MandelbrotData & operator = ( MandelbrotData const & ) = delete;
27     // make moveable
28     MandelbrotData( MandelbrotData && ) = default;
29     MandelbrotData & operator = ( MandelbrotData && ) = default;
30 };
31
32 using LineChan = Chan< std::size_t >;
33 using DataChan = Chan< MandelbrotData >;
34
35 inline bool point_predicate( const double x, const double y ) ↩
       noexcept {
36     return ( x * x + y * y ) < 4.;
37 }
38
39 void mandelbrot( const std::size_t dim, LineChan::Rx line_rx, ↩
       DataChan::Tx data_tx ) {
40     const double integral_x = (XMAX - XMIN) / static_cast< double >(↩
           dim );
41     const double integral_y = (YMAX - YMIN) / static_cast< double >(↩
           dim );
42     for ( auto line : line_rx ) {
43         MandelbrotData data = { line, std::vector< double >( dim ) ↩
               };
44         double y = YMIN + line * integral_y;
45         double x = XMIN;
46         for ( std::size_t x_coord = 0; x_coord < dim; ++x_coord ) {
```

```
47              double x1 = 0., y1 = 0.;
48              std::size_t loop_count = 0;
49              while ( loop_count < MAX_ITER && point_predicate( x1, y1↩
                     ) ) {
50                  ++loop_count;
51                  double x1_new = x1 * x1 - y1 * y1 + x;
52                  y1 = 2 * x1 * y1 + y;
53                  x1 = x1_new;
54              }
55              double value = static_cast< double >( loop_count ) / ↩
                    static_cast< double >( MAX_ITER );
56              data.values[ x_coord ] = value;
57              x += integral_x;
58          }
59          data_tx << std::move( data );
60      }
61  }
62
63  void producer( const std::size_t dim, std::array< LineChan::Tx, ↩
        NUM_WORKERS > line_txs ) {
64      for ( std::size_t line = 0; line < dim; ++line ) {
65          Alt()
66              .send_for( line_txs.begin(), line_txs.end(), line )
67              .select();
68      }
69  }
70
71  void consumer( const std::size_t dim, std::array< DataChan::Rx, ↩
        NUM_WORKERS > data_rxs ) {
72      std::vector< std::vector< double > > results( dim );
73      for ( std::size_t i = 0; i < dim; ++i ) {
74          Alt()
75              .recv_for( data_rxs.begin(), data_rxs.end(),
76                  [&results]( MandelbrotData data ){
77                      results[ data.line ] = std::move( data.values );
78                  } )
79              .select();
80      }
81  }
82
```

```
83  void mandelbrot_program( std::size_t dim ) {
84      std::size_t sum = 0;
85      for ( std::size_t round = 0; round < ROUNDS; ++round ) {
86          ChanArr< std::size_t, NUM_WORKERS >    line_chs;
87          ChanArr< MandelbrotData, NUM_WORKERS > data_chs;
88          std::vector< Process > workers;
89          workers.reserve( NUM_WORKERS );
90          for ( std::size_t i = 0; i < NUM_WORKERS; ++i ) {
91              workers.emplace_back( mandelbrot, dim,
92                  line_chs[i].move_rx(),
93                  data_chs[i].move_tx() );
94          }
95          auto start = std::chrono::system_clock::now();
96          parallel(
97              proc_for( workers.begin(), workers.end() ),
98              proc( producer, dim, line_chs.collect_tx() ),
99              proc( consumer, dim, data_chs.collect_rx() )
100         );
101         auto stop = std::chrono::system_clock::now();
102         std::chrono::duration< std::size_t, std::nano > diff = stop ↩
                - start;
103         sum += diff.count();
104     }
105     std::cout << dim << ", " << sum / ROUNDS << std::endl;
106 }
107
108 int main() {
109     for ( std::size_t dim = 1; dim < 100; dim += 1 ) {
110         mandelbrot_program( dim );
111     }
112     for ( std::size_t dim = 100; dim < 500; dim += 5 ) {
113         mandelbrot_program( dim );
114     }
115     for ( std::size_t dim = 500; dim <= 1000; dim += 10 ) {
116         mandelbrot_program( dim );
117     }
118     return 0;
119 }
```

Hard Coded Mandelbrot: Go

```
1  package main
2  import (
3      "fmt"
4      "sync"
5      "time"
6  )
7  const (
8      ROUNDS = 100
9      NUM_WORKERS = 8
10     MAX_ITER = 255
11     /* x in [-2.1; 1.0] */
12     /* y in [-1.3; 1.3] */
13     XMIN = -2.1
14     XMAX =  1.0
15     YMIN = -1.3
16     YMAX =  1.3
17 )
18 type MandelbrotData struct {
19     line uint
20     values []float64
21 }
22 func point_predicate(x, y float64) bool {
23     return (x * x + y * y) < 4.0
24 }
25
26 func mandelbrot(wg *sync.WaitGroup, dim uint, lines_ch <-chan uint, ↩
       data_ch chan<- MandelbrotData) {
27     integral_x := (XMAX - XMIN) / float64(dim);
28     integral_y := (YMAX - YMIN) / float64(dim);
29     for line := range lines_ch {
30         data := MandelbrotData{ line, make([]float64, dim) }
31         y := YMIN + float64(line) * integral_y
32         x := XMIN
33         for x_coord := uint(0); x_coord < dim; x_coord++ {
34             x1 := 0.0
35             y1 := 0.0
36             loop_count := 0
```

```
37              for loop_count < MAX_ITER && point_predicate(x1, y1) {
38                  loop_count += 1
39                  x1_new := x1 * x1 - y1 * y1 + x
40                  y1 = 2.0 * x1 * y1 + y
41                  x1 = x1_new
42              }
43              value := float64(loop_count) / float64( MAX_ITER )
44              data.values[x_coord] = value
45              x += integral_x
46          }
47          data_ch <- data
48      }
49      wg.Done()
50  }
51
52  func producer(wg *sync.WaitGroup, dim uint, lines_ch chan<- uint) {
53      for line := uint(0); line < dim; line++ {
54          lines_ch <- line
55      }
56      close(lines_ch)
57      wg.Done()
58  }
59
60  func consumer(wg *sync.WaitGroup, dim uint, data_ch <-chan ↩
        MandelbrotData) {
61      results := make([][]float64, dim)
62      for i := uint(0); i< dim; i++ {
63          data := <-data_ch
64          results[data.line] = data.values
65      }
66      wg.Done()
67  }
68
69  func mandelbrot_program(dim uint) {
70      sum := int64(0)
71      for round := 0; round < ROUNDS; round++ {
72          lines_ch := make(chan uint)
73          data_ch := make(chan MandelbrotData)
74          start := time.Now()
75          var wg sync.WaitGroup
```

```
76          wg.Add( 2 + NUM_WORKERS )
77          go producer(&wg, dim, lines_ch)
78          go consumer(&wg, dim, data_ch)
79          for worker := 0; worker < NUM_WORKERS; worker++ {
80              go mandelbrot(&wg, dim, lines_ch, data_ch)
81          }
82          wg.Wait()
83          elapsed := time.Since(start)
84          sum += elapsed.Nanoseconds()
85      }
86      fmt.Println(dim, ",", sum/ROUNDS)
87  }
88
89  func main() {
90      for dim := uint(1); dim < 100; dim += 1 {
91          mandelbrot_program(dim)
92      }
93      for dim := uint(100); dim < 500; dim += 5 {
94          mandelbrot_program(dim)
95      }
96      for dim := uint(500); dim <= 1000; dim += 10 {
97          mandelbrot_program(dim)
98      }
99  }
```

Hard Coded Mandelbrot: C++CSP

```
1   #include <chrono>
2   #include <iostream>
3   #include <memory>
4   #include <string>
5   #include <vector>
6   #include "csp.h"
7
8   using namespace csp;
9
10  constexpr int NUM_WORKERS = 8;
11  constexpr std::size_t MAX_ITERATIONS = 255;
```

```
12  constexpr std::size_t ROUNDS = 100;
13  /* x in [-2.1; 1.0] */
14  /* y in [-1.3; 1.3] */
15  constexpr double xmin = -2.1;
16  constexpr double xmax = 1.0;
17  constexpr double ymin = -1.3;
18  constexpr double ymax = 1.3;
19
20  template<typename T>
21  using mobile = std::unique_ptr<T>;
22
23  struct mandelbrot_packet {
24      int line = 0;
25      std::vector< double > data;
26  };
27
28  inline bool point_predicate( const double x, const double y ) ↩
        noexcept {
29      return ( x * x + y * y ) < 2.0;
30  }
31
32  void mandelbrot( std::size_t dim, chan_in<int> in, chan_out< mobile<↩
         mandelbrot_packet > > out ) noexcept {
33      const double integral_x = (xmax - xmin) / static_cast< double >(↩
            dim);
34      const double integral_y = (ymax - ymin) / static_cast< double >(↩
            dim);
35      int line = in();
36      while (line != -1) {
37          double x, y, x1, y1, xx = 0.0;
38          std::size_t loop_count = 0;
39          mobile<mandelbrot_packet> packet = mobile<mandelbrot_packet↩
                >(new mandelbrot_packet());
40          packet->line = line;
41          packet->data = std::vector< double >( dim );
42          y = ymin + (line * integral_y);
43          x = xmin;
44          for ( std::size_t x_coord = 0; x_coord < dim; ++x_coord ) {
45              x1 = 0.0;
46              y1 = 0.0;
```

```
47              loop_count = 0;
48              while ( loop_count < MAX_ITERATIONS && point_predicate( ↩
                    x1, y1 ) ) {
49                  ++loop_count;
50                  xx = x1 * x1 - y1 * y1 + x;
51                  y1 = 2 * x1 * y1 + y;
52                  x1 = xx;
53              }
54              auto val = static_cast< double >( loop_count ) / ↩
                    static_cast< double >( MAX_ITERATIONS );
55              packet->data[ x_coord ] = val;
56              x += integral_x;
57          }
58          out( move( packet ) );
59          line = in();
60      }
61  }
62
63  void producer(chan_out<int> out, int lines, int num_workers) ↩
        noexcept {
64      for ( int i = 0; i < lines; ++i ) {
65          out( i );
66      }
67      for ( int i = 0; i < num_workers; ++i ) {
68          out( -1 );
69      }
70  }
71
72  void consumer(chan_in<mobile<mandelbrot_packet>> in, int lines) ↩
        noexcept {
73      std::vector< std::vector< double > > results( lines );
74      for ( int i = 0; i < lines; ++i ) {
75          auto packet = in();
76          results[ packet->line ] = std::move( packet->data );
77      }
78  }
79
80  void mandelbrot_program( std::size_t dim ) {
81      one2any_chan< int > lines;
82      any2one_chan< mobile< mandelbrot_packet > > data;
```

```
83      std::vector< std::function< void() > > workers;
84      for (int i = 0; i < NUM_WORKERS; ++i) {
85          workers.push_back( make_proc( mandelbrot,
86                  dim, lines, data ) );
87      }
88      std::size_t sum = 0;
89      for ( std::size_t i = 0; i < ROUNDS; ++i) {
90          auto start = std::chrono::system_clock::now();
91          par {
92              make_proc( producer, lines, dim, NUM_WORKERS ),
93              par( workers ),
94              make_proc( consumer, data, dim )
95          }();
96          auto stop = std::chrono::system_clock::now();
97          std::chrono::duration< std::size_t, std::nano > diff = stop ↩
                - start;
98          sum += diff.count();
99      }
100     std::cout << dim << ", " << sum / ROUNDS << std::endl;
101 }
102
103 int main() {
104     for ( std::size_t dim = 1; dim < 100; dim += 1 ) {
105         mandelbrot_program( dim );
106     }
107     for ( std::size_t dim = 100; dim < 500; dim += 5 ) {
108         mandelbrot_program( dim );
109     }
110     for ( std::size_t dim = 500; dim <= 1000; dim += 10 ) {
111         mandelbrot_program( dim );
112     }
113     return 0;
114 }
```

Dynamic Mandelbrot: ProXC++

```
1  #include <algorithm>
2  #include <array>
```

```
3  #include <chrono>
4  #include <iostream>
5  #include <vector>
6  #include <proxc.hpp>
7
8  using namespace proxc;
9
10 constexpr std::size_t ROUNDS = 100;
11 constexpr std::size_t MAX_ITER = 255;
12 /* x in [-2.1; 1.0] */
13 /* y in [-1.3; 1.3] */
14 constexpr double XMIN = -2.1;
15 constexpr double XMAX =  1.0;
16 constexpr double YMIN = -1.3;
17 constexpr double YMAX =  1.3;
18
19 struct MandelbrotData {
20     std::size_t line;
21     std::vector< double > values;
22     MandelbrotData() = default;
23     // make non-copyable
24     MandelbrotData( MandelbrotData const & ) = delete;
25     MandelbrotData & operator = ( MandelbrotData const & ) = delete;
26     // make moveable
27     MandelbrotData( MandelbrotData && ) = default;
28     MandelbrotData & operator = ( MandelbrotData && ) = default;
29 };
30
31 using DataChan = Chan< MandelbrotData >;
32
33 inline bool point_predicate( const double x, const double y ) ↩
       noexcept {
34     return ( x * x + y * y ) < 4.;
35 }
36
37 void mandelbrot( const std::size_t line, const std::size_t dim, ↩
       DataChan::Tx out ) {
38     const double integral_x = (XMAX - XMIN) / static_cast< double >(↩
           dim );
```

```
39      const double integral_y = (YMAX - YMIN) / static_cast< double >(↩
            dim );
40      MandelbrotData data = { line, std::vector< double >( dim ) };
41      double y = YMIN + line * integral_y;
42      double x = XMIN;
43      for ( std::size_t x_coord = 0; x_coord < dim; ++x_coord ) {
44          double x1 = 0., y1 = 0.;
45          std::size_t loop_count = 0;
46          while ( loop_count < MAX_ITER && point_predicate( x1, y1 ) )↩
                {
47              ++loop_count;
48              double x1_new = x1 * x1 - y1 * y1 + x;
49              y1 = 2 * x1 * y1 + y;
50              x1 = x1_new;
51          }
52          double value = static_cast< double >( loop_count ) / ↩
                static_cast< double >( MAX_ITER );
53          data.values[ x_coord ] = value;
54          x += integral_x;
55      }
56      out << std::move( data );
57  }
58
59  void consumer( const std::size_t dim, std::vector< DataChan::Rx > ↩
        ins ) {
60      std::vector< std::vector< double > > results( dim );
61      for ( auto& in : ins ) {
62          auto data = in();
63          results[ data.line ] = std::move( data.values );
64      }
65  }
66
67  void mandelbrot_program( std::size_t dim ) {
68      std::size_t sum = 0;
69      for ( std::size_t round = 0; round < ROUNDS; ++round ) {
70          ChanVec< MandelbrotData > data_chs{ dim };
71          std::vector< Process > workers;
72          workers.reserve( dim );
73          for ( std::size_t i = 0; i < dim; ++i ) {
```

```
74              workers.emplace_back( mandelbrot, i, dim, data_chs[i].↩
                    move_tx() );
75          }
76          auto start = std::chrono::system_clock::now();
77          parallel(
78              proc_for( workers.begin(), workers.end() ),
79              proc( consumer, dim, data_chs.collect_rx() )
80          );
81          auto stop = std::chrono::system_clock::now();
82          std::chrono::duration< std::size_t, std::nano > diff = stop ↩
                - start;
83          sum += diff.count();
84      }
85      std::cout << dim << ", " << sum / ROUNDS << std::endl;
86  }
87
88  int main() {
89      for ( std::size_t dim = 1; dim < 100; dim += 1 ) {
90          mandelbrot_program( dim );
91      }
92      for ( std::size_t dim = 100; dim < 500; dim += 5 ) {
93          mandelbrot_program( dim );
94      }
95      for ( std::size_t dim = 500; dim <= 1000; dim += 10 ) {
96          mandelbrot_program( dim );
97      }
98      return 0;
99  }
```

Dynamic Mandelbrot: Go

```
1  package main
2  import (
3      "fmt"
4      "sync"
5      "time"
6  )
7  const (
```

```
8       ROUNDS = 100
9       MAX_ITER = 255
10      /* x in [-2.1; 1.0] */
11      /* y in [-1.3; 1.3] */
12      XMIN = -2.1
13      XMAX =  1.0
14      YMIN = -1.3
15      YMAX =  1.3
16  )
17
18  type MandelbrotData struct {
19      line uint
20      values []float64
21  }
22
23  func point_predicate(x, y float64) bool {
24      return (x * x + y * y) < 4.0
25  }
26
27  func mandelbrot(wg *sync.WaitGroup, line, dim uint, data_ch chan<- ↩
        MandelbrotData) {
28      integral_x := (XMAX - XMIN) / float64(dim);
29      integral_y := (YMAX - YMIN) / float64(dim);
30      data := MandelbrotData{ line, make([]float64, dim) }
31      y := YMIN + float64(line) * integral_y
32      x := XMIN
33      for x_coord := uint(0); x_coord < dim; x_coord++ {
34          x1 := 0.0
35          y1 := 0.0
36          loop_count := 0
37          for loop_count < MAX_ITER && point_predicate(x1, y1) {
38              loop_count += 1
39              x1_new := x1 * x1 - y1 * y1 + x
40              y1 = 2.0 * x1 * y1 + y
41              x1 = x1_new
42          }
43          value := float64(loop_count) / float64( MAX_ITER )
44          data.values[x_coord] = value
45          x += integral_x
46      }
```

```
47      data_ch <- data
48      wg.Done()
49  }
50
51  func consumer(wg *sync.WaitGroup, dim uint, data_ch <-chan ↩
        MandelbrotData) {
52      results := make([][]float64, dim)
53      for i := uint(0); i < dim; i++ {
54          data := <-data_ch
55          results[data.line] = data.values
56      }
57      wg.Done()
58  }
59
60  func mandelbrot_program(dim uint) {
61      sum := int64(0)
62      for round := 0; round < ROUNDS; round++ {
63          data_ch := make(chan MandelbrotData)
64          start := time.Now()
65          var wg sync.WaitGroup
66          wg.Add( 1 + int(dim) )
67          go consumer(&wg, dim, data_ch)
68          for line := uint(0); line < dim; line++ {
69              go mandelbrot(&wg, line, dim, data_ch)
70          }
71          wg.Wait()
72          elapsed := time.Since(start)
73          sum += elapsed.Nanoseconds()
74      }
75      fmt.Println(dim, ",", sum/ROUNDS)
76  }
77
78  func main() {
79      for dim := uint(1); dim < 100; dim += 1 {
80          mandelbrot_program(dim)
81      }
82      for dim := uint(100); dim < 500; dim += 5 {
83          mandelbrot_program(dim)
84      }
85      for dim := uint(500); dim <= 1000; dim += 10 {
```

```
86          mandelbrot_program(dim)
87      }
88  }
```

Dynamic Mandelbrot: C++CSP

```
1   #include <chrono>
2   #include <iostream>
3   #include <memory>
4   #include <string>
5   #include <vector>
6   #include "csp.h"
7
8   using namespace csp;
9
10  constexpr int NUM_WORKERS = 8;
11  constexpr std::size_t MAX_ITERATIONS = 255;
12  constexpr std::size_t ROUNDS = 100;
13  /* x in [-2.1; 1.0] */
14  /* y in [-1.3; 1.3] */
15  constexpr double xmin = -2.1;
16  constexpr double xmax = 1.0;
17  constexpr double ymin = -1.3;
18  constexpr double ymax = 1.3;
19
20  template<typename T>
21  using mobile = std::unique_ptr< T >;
22
23  struct mandelbrot_packet {
24      int line = 0;
25      std::vector< double > data;
26  };
27
28  inline bool point_predicate( const double x, const double y ) ↩
        noexcept {
29      return ( x * x + y * y ) < 2.0;
30  }
31
```

```
32 void mandelbrot(int line, int dim, chan_out< mobile< ↩
       mandelbrot_packet > > out) noexcept {
33     double integral_x = (xmax - xmin) / static_cast<double>(dim);
34     double integral_y = (ymax - ymin) / static_cast<double>(dim);
35     double x, y, x1, y1, xx = 0.0;
36     std::size_t loop_count = 0;
37     mobile< mandelbrot_packet > packet{ new mandelbrot_packet() };
38     packet->line = line;
39     packet->data = std::vector< double >( dim );
40     y = ymin + line * integral_y;
41     x = xmin;
42     for ( std::size_t x_coord = 0; x_coord < dim; ++x_coord ) {
43         x1 = 0.0;
44         y1 = 0.0;
45         loop_count = 0;
46         while ( loop_count < MAX_ITERATIONS && point_predicate( x1, ↩
               y1 ) ) {
47             ++loop_count;
48             xx = x1 * x1 - y1 * y1 + x;
49             y1 = 2 * x1 * y1 + y;
50             x1 = xx;
51         }
52         auto val = static_cast< double >( loop_count ) / static_cast↩
               < double >( MAX_ITERATIONS );
53         packet->data[ x_coord ] = val;
54         x += integral_x;
55     }
56     out( std::move( packet ) );
57 }
58
59 void consumer( chan_in< mobile< mandelbrot_packet > > in, int lines ↩
       ) noexcept {
60     std::vector< std::vector< double > > results(lines);
61     for ( int i = 0; i < lines; ++i ) {
62         auto packet = in();
63         results[ packet->line ] = std::move( packet->data );
64     }
65 }
66
67 void mandelbrot_program( std::size_t dim ) {
```

```
68      any2one_chan< mobile< mandelbrot_packet > > data;
69      std::vector< std::function< void() > > workers;
70      for ( int i = 0; i < dim; ++i ) {
71          workers.push_back(make_proc(mandelbrot, i, dim, data));
72      }
73      std::size_t sum = 0;
74      for ( std::size_t i = 0; i < ROUNDS; ++i ) {
75          auto start = system_clock::now();
76          par {
77              par(workers),
78              make_proc(consumer, data, dim)
79          }();
80          auto stop = system_clock::now();
81          std::chrono::duration< std::size_t, std::nano > diff = stop ↩
                - start;
82          sum += diff.count();
83      }
84      std::cout << dim << ", " << sum / ROUNDS << std::endl;
85  }
86
87  int main() {
88      for ( std::size_t dim = 1; dim < 100; dim += 1 ) {
89          mandelbrot_program( dim );
90      }
91      for ( std::size_t dim = 100; dim < 500; dim += 5 ) {
92          mandelbrot_program( dim );
93      }
94      for ( std::size_t dim = 500; dim <= 1000; dim += 10 ) {
95          mandelbrot_program( dim );
96      }
97      return 0;
98  }
```

Prime Sieve: ProXC++

```
1  #include <sstream>
2  #include <iostream>
3  #include <proxc.hpp>
```

```
4
5  using namespace proxc;
6  using ItemT = std::size_t;
7
8  void generate( Chan< ItemT >::Tx out, Chan< ItemT >::Rx ex ) {
9      ItemT i{ 2 };
10     while ( ex ) {
11         out << i++;
12     }
13 }
14
15 void filter( Chan< ItemT >::Rx in, Chan< ItemT >::Tx out ) {
16     ItemT prime = in();
17     for ( auto i : in ) {
18         if ( i % prime != 0 ) {
19             out << i;
20         }
21     }
22 }
23
24 int main( int argc, char *argv[] ) {
25     if ( argc != 2 ) { return -1; }
26     std::size_t n;
27     std::stringstream{ argv[1] } >> n;
28     Chan< ItemT >    ex_ch;
29     ChanVec< ItemT > chs{ n };
30     std::vector< Process > filters;
31     filters.reserve( n - 1 );
32     for ( std::size_t i = 0; i < n - 1; ++i ) {
33         filters.emplace_back( filter, chs[i].move_rx(), chs[i+1].↩
               move_tx() );
34     }
35     this_proc::yield();
36     auto start = std::chrono::steady_clock::now();
37     parallel(
38         proc( generate, chs[0].move_tx(), ex_ch.move_rx() ),
39         proc_for( filters.begin(), filters.end() ),
40         proc( [start,n]( Chan< ItemT >::Rx in, Chan< ItemT >::Tx ){
41                 in();
42                 auto end = std::chrono::steady_clock::now();
```

```
43                  std::chrono::duration< std::size_t, std::nano > diff↩
                        = end - start;
44                  std::cout << n << "," << diff.count() / n << std::↩
                        endl;
45              },
46              chs[n-1].move_rx(), ex_ch.move_tx() )
47      );
48      return 0;
49  }
```

Prime Sieve: Go

```
1   package main
2   import (
3       "fmt"
4       "time"
5       "os"
6       "strconv"
7   )
8   func generate(ch chan<- int) {
9       for i := 2; ; i++ {
10          ch <- i
11      }
12  }
13
14  func filter(in <-chan int, out chan<- int, prime int) {
15      for i := range in {
16          if i % prime != 0 {
17              out <- i
18          }
19      }
20  }
21
22  func main() {
23      n := func() int64 {
24          tmp, _ := strconv.Atoi(os.Args[1])
25          return int64(tmp)
26      }()
```

```
27      ch := make(chan int)
28      var prime int
29      start := time.Now()
30      go generate(ch)
31      for i := int64(0); i < n; i++ {
32          prime = <-ch
33          ch1 := make(chan int)
34          go filter(ch, ch1, prime)
35          ch = ch1
36      }
37      elapsed := time.Since(start).Nanoseconds()
38      fmt.Println(n, ",", elapsed/n)
39  }
```

Prime Sieve: C++CSP

```
1   #include <chrono>
2   #include <iostream>
3   #include <functional>
4   #include <vector>
5   #include "csp.h"
6
7   using ClockT = std::chrono::steady_clock;
8   using TimePointT = typename ClockT::time_point;
9   using Chan = csp::one2one_chan< long >;
10
11  void generate( Chan out ) {
12      long i = 2;
13      for ( ;; ) {
14          out( i++ );
15      }
16  }
17
18  void filter( Chan in, Chan out ) {
19      long prime = in();
20      for ( ;; ) {
21          long i = in();
22          if ( i % prime != 0 ) {
```

```
23              out( i );
24          }
25      }
26  }
27
28  void time_taker( std::size_t n, TimePointT start, Chan in ) {
29      long prime = in();
30      auto end = ClockT::now();
31      std::chrono::duration< std::size_t, std::nano > diff = end - ↩
            start;
32      std::cout << n << "," << diff.count() / n << std::endl;
33      std::exit(0);
34  }
35
36  int main( int argc, char *argv[] ) {
37      if ( argc != 2 ) { return 1; }
38      std::size_t n;
39      std::stringstream{ arg[1] } >> n;
40      std::vector< Chan > chans( n );
41      std::vector< std::function< void() > > procs;
42      procs.reserve( n - 1 );
43      for ( std::size_t i = 0; i < n - 1; ++i ) {
44          procs.push_back( csp::make_proc(
45              filter, chans[i], chans[i+1]
46          ) );
47      }
48      auto start = ClockT::now();
49      csp::par{
50          make_proc( generate, chans[0] ),
51          csp::par( procs ),
52          make_proc( time_taker, n, start, chans[n-1] )
53      }();
54  }
```

Bibliography

[1] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978. ISSN 0001-0782. doi: 10.1145/359576.359585.

[2] Geoff Barrett. occam 3 Reference Manual. Technical report, Technical report, Inmos Limited, March 1992. Available at: `http://wotug.ukc.ac.uk/parallel/occam/documentation`, 1992.

[3] Douglas Watt. *Programming XC on XMOS Devices*. XMOS Limited, 2009. ISBN 1907361030.

[4] Team Go. The Go Programming Language Specification. 2009.

[5] Edvard S. Pettersen. *ProXC – A CSP-Inspired Concurrency Library for the C Programming Language*. Department of Engineering Cybernetics, Norwegian University of Science and Technology, 2016.

[6] Neil C. C. Brown and Peter H. Welch. An Introduction to the Kent C++ CSP Library. In J. F. Broenink and G. H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 139–156, sep 2003. ISBN 978-1-58603-381-1.

[7] Neil C. C. Brown. C++CSP2: A Many-to-Many Threading Model for Multi-core Architectures. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 183–205, jul 2007. ISBN 978-1586037673.

[8] Kevin Chalmers. Development and Evaluation of a Modern C++CSP Library. In *Communicating Process Architectures 2016*, 2016.

[9] John M. Bjørndalen, Brian Vinter, and Otto J. Anshus. PyCSP - Communicating Sequential Processes for Python. In *Communicating Process Architectures 2007*, pages 229–248, 2007.

[10] Peter H. Welch, Neil C. C. Brown, James Moores, Kevin Chalmers, and Bernhard H. C. Sputh. Integrating and Extending JCSP. In *CPA*, volume 65, pages 349–370, 2007.

[11] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming.* Pearson Education, 2006. ISBN 032131283X.

[12] Ian A. Buck, John R. Nickolls, Michael C. Shebanow, and Lars S. Nyland. Atomic Memory Operators in a Parallel Processor, December 1 2009. US Patent 7,627,723.

[13] Michel Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations.* Springer Science & Business Media, 2012. ISBN 3642320279.

[14] Edsger W. Dijkstra. Over de Sequentialiteit van Procesbeschrijvingen. 1962 or 1963.

[15] Allen B. Downey. The Little Book of Semaphores. (2.2.1), 2016.

[16] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Principles, 7th Ed.* Wiley Student Edition. Wiley India Pvt. Limited, 2006. ISBN 9788126509621.

[17] Edward G. Coffman, Melanie Elphick, and Arie Shoshani. System Deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[18] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, 1975.

[19] Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders. *Communicating Sequential Processes. The First 25 Years*, volume 3525 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005. ISBN 9783540322658. doi: 10.1007/b136154.

[20] Stephen D. Brookes, C. A. R. Hoare, and Andrew W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM (JACM)*, 31(3): 560–599, 1984.

[21] FDR2 User Manual. Failures-Divergence Refinement. 2000.

[22] INMOS International. *occam 2 Reference Manual.* Prentice hall, 1988.

[23] Henry Ledgard. Reference Manual for the ADA Programming Language. 1983.

[24] Paul E. McKenney. Memory Ordering in Modern Microprocessors. *Interface*, 6:6, sep 2007.

[25] Jeff Preshing. Weak vs. Strong Memory Models. `http://preshing.com/20120930/weak-vs-strong-memory-models`, 2012. Accessed: 2017-05-30.

[26] Jeff Preshing. Memory Barriers are like Source Control Operations. `http://preshing.com/20120710/memory-barriers-are-like-source-control-operations/`, 2012. Accessed: 2017-05-30.

[27] Jeff Preshing. An Introduction to Lock-Free Programming. `http://preshing.com/20120612/an-introduction-to-lock-free-programming`, 2012. Accessed: 2017-05-30.

[28] Jeff Preshing. Acquire and Release Semantics. `http://preshing.com/20120913/acquire-and-release-semantics/`, 2012. Accessed: 2017-05-30.

[29] Maged M. Michael and Michael L. Scott. Nonblocking Algorithms and Preemption-safe Locking on Multiprogrammed Shared Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[30] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, 1991.

[31] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[32] Leonardo Dagum and Ramesh Menon. OpenMP: an Industry Standard API for Shared-Memory Programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[33] Arch D. Robison. Composable Parallel Patterns with intel Cilk Plus. *Computing in Science & Engineering*, 15(2):66–71, 2013.

[34] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[35] David Chase and Yossi Lev. Dynamic Circular Work-Stealing Deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28. ACM, 2005.

[36] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli.

Correct and Efficient Work-Stealing for Weak Memory Models. In *ACM SIGPLAN Notices*, volume 48, pages 69–80. ACM, 2013.

[37] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[38] Vaidy S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[39] Xian-He Sun and Yong Chen. Reevaluating Amdahl's Law in the Multicore Era. *Journal of Parallel and Distributed Computing*, 70(2):183–188, 2010.

[40] David May. Occam. *ACM Sigplan Notices*, 18(4):69–79, 1983.

[41] David May and Roger Shepherd. Occam and the Transputer. In *Proc. of the IFIP WG 10.3 workshop on Concurrent languages in distributed systems: hardware supported implementation*, pages 19–33. Elsevier North-Holland, Inc., 1984.

[42] Douglas Watt. *Programming XC on XMOS Devices*. XMOS Limited, 2009.

[43] David May. *The XMOS XS1 Architecture*. XMOS, 2009.

[44] Ivo Balbaert. *The Way to Go: A Thorough Introduction to the Go Programming Language*. IUniverse, 2012.

[45] Oliver Kowalke. The Boost C++ Libraries: Boost Context. `http://www.boost.org/doc/libs/1_64_0/libs/context/doc/html/index.html`, 2014–2017. Accessed: 2017-05-13.

[46] Boost. The Boost C++ Libraries. `http://www.boost.org/`, 2017. Accessed: 2017-05-13.

[47] Michael D. McCool, Arch D. Robison, and James Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2012.

[48] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 4th edition, May 2013. ISBN 978-0321563842.

[49] Dmitry Vyukov. Intrusive MPSC Node-Based Queue. `http://www.1024cores.net/home/lock-free-algorithms/queues/intrusive-mpsc-node-based-queue`, 2014.

[50] Edvard S. Pettersen. ProXC - A CSP-Inspired Concurrency Library for C and

C++. `https://github.com/edvardsp/libproxc`, 2017. Accessed: 2017-05-22.

[51] MA Roger. A Reconfigurable Host lnterconnection Scheme for Occam-based Field Programmable Gate Arrays. In *Communicating Process Architectures 2001: WoTUG-24: Proceedings of the 24th World Occam and Transputer User Group Technical Meeting, 16-19 September 2001, Bristol, United Kingdom*, volume 59, page 179. IOS Press, 2001.

[52] Barry Wilkinson and Michael Allen. *Parallel Programing*, volume 999. Prentice hall Upper Saddle River, NJ, 1999.

[53] Concurrent Prime Sieve in Go. `https://play.golang.org/p/9U22NfrXeq`, 2016. Accessed: 2016-12-14.

[54] Carl Ritson. Extended Outputs. `https://www.cs.kent.ac.uk/research/groups/plas/wiki/OEP/142`, 2006. Accessed: 2017-06-04.