A task-based message passing framework

Francois Gingras

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fullfilment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

20/06/2018

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Francois Gingras

Entitled: A task-based message passing framework

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.
Signed by the examining committee:

_____ Chair
Dr. R. Jayakumar

_____ Examiner
Dr. D. Goswami

_____ Examiner
Dr. T. Glatard

_____ Supervisor
Dr. T. Eavis

Approved By _____
Chair of Department or Graduate Program Director

_____20___ _____
Dean of Faculty

ABSTRACT

A task-based message passing framework

Francois Gingras

Over the past decade, it has become clear that parallel and distributed programming will occupy an increasingly larger proportion of a developer's work. While numerous programming languages and libraries have been built to facilitate working with concurrency, developer work is still difficult and error-prone.

In this thesis, we propose a task-based message passing framework. The proposed framework combines the actor model with message passing functionality to offer a useful and efficient way to implement parallel and distributed algorithms. The framework is intended to be part of a novel C compiler that will offer built-in task and message features. Perhaps most importantly, the new framework aims to be intuitive and efficient.

We have used the framework to implement a parallel sample-sort and a client-server application. Our results demonstrate both strong performance for a parallel sorting algorithm and scalability that extends to thousands of concurrent messages. In addition, we have developed a client server app that emphasizes the intuitive nature of the development cycle for the new model. We conclude that the proposed message passing framework would be well suited to concurrent development environments and offers a simple and efficient way to build applications for the new wave of multi-core hardware platforms.

ACKNOWLEDGEMENTS

I would like to thank Dr. Todd Eavis for supervising and providing advice, knowledge and experience throughout the completion of this thesis.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the course of the past decade, it has become clear that parallel and distributed systems will play an increasingly significant role in the technology sphere. In addition to the traditional use of multiple core servers to speed up analysis or simulation, new domains such as the Internet of Things, cloud computing, and mobile technologies all require major distributed architectures as well [53] [50].

That said, it is very challenging to successfully implement a concurrent, parallel or distributed system on a large scale at a reasonable cost. One reason is that the current tools used to achieve this are complex and notoriously difficult to use in real-world environments. Though many programming languages offer complete synchronization libraries, the developer is still fully responsible to properly use them. To avoid the high cost of starting from scratch, teams usually rely on a principal architecture library. However, later in the project the lack of control or the complexity of the library often leads to high additional costs to the project.

For example, OpenMPI is a well-known message passing library that has become a defacto standard in the field. However, over time, the library has become very complex, with hundreds of additional functions intended to help developers with

many challenges such as fault-tolerance, guarantee of delivery or high availability. Other technologies such as CUDA, depend on proprietary hardware.

In any case, developing such applications requires a lot of resources and is time-consuming, error-prone and very expensive. We address these issues in this thesis by proposing a simple and efficient task-based message passing framework. The library is to be intuitive and to work across multiple development environments. In short, it has to provide everything a programmer needs to implement a concurrent, distributed or parallel application. The framework will be integrated into a novel C compiler and will be directly compatible with existing C libraries. This thesis presents the core features of the framework and describes its design with functional prototypes and evaluations.

1.1 Overview

The proposed framework will take the form of a library directly ready for language integration. Using the C language, the framework will allow a developer to easily implement concurrent, parallel or distributed applications. We will achieve this by offering a task-based message passing programming style built directly into the language compiler. To avoid creating an overly complex framework, we have limited our functionality to message passing and task abstraction. Our main assumption is that by efficiently implementing these components, we will outperform or at least match existing library performance, while significantly minimizing developer work.

The aforementioned compiler is part of a related project. The new compiler is, in fact, a front-end that will generate additional source code required to support new task abstraction. The compiler will have access to the rich C ecosystem of tools and

libraries. Our target platform includes today's common CPU architectures - general purpose single and multicore CPUs used for concurrent and parallel algorithms. We note that the model is not intended for specialized environments such as GPUs and data-intensive cloud applications. Moreover, the framework is designed to abstract the communication layer for intra-process, inter-process, and distributed network communication. That said, the focus of this thesis will be on intra-process and inter-process. Extensions to support fully distributed communication will be left to future work.

1.1.1 Message based communication

Message passing is central to this framework. Each task, a primitive executable actor, will use messages to communicate with other tasks. One key challenge of implementing a message passing system is how to efficiently store, retrieve and send messages. The framework explores the use of shared memory to achieve efficiency and communication transparency between threads and processes. Figure 1.1 shows an example of a task-based architecture. Notice how task A and B are on the same process, while task C is on a different process. This simple illustration demonstrates how the framework is intended to abstract inter-process communication with inter-thread communication, in order to easily offer developers scalability for their application.

1.2 Evaluation

The proposed framework will be evaluated using three applications: sample-sort, a parallel sorting algorithm that serves to illustrate that our framework can achieve

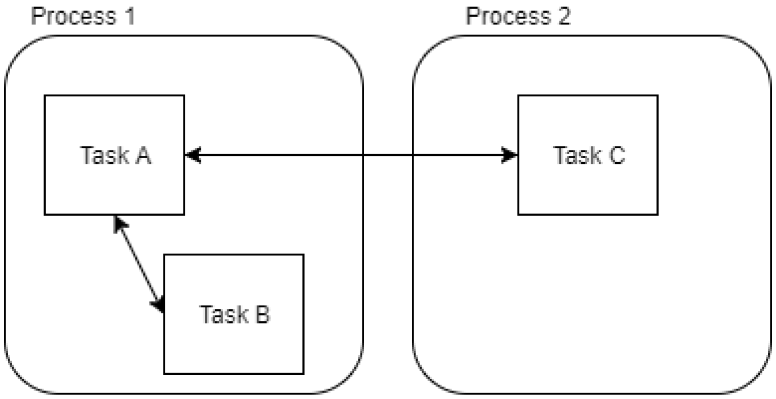

Figure 1.1: Task-based architecture example.

state-of-the-art parallel performance; a client-server application, to measure message throughput in inter-process or inter-thread communication and; a weather station application to showcase task discovery functionality. We will also use the weather application to discuss the complexity and usability of the system.

1.3 Thesis organization

The thesis is organized as follows:

Chapter 2 explains important concepts required to understand the fundamental problem and challenges. We discuss multi-core, concurrent, and parallel and distributed programming issues and we will also cover message passing and the actor model, two core aspects of the framework. Finally, we will review existing technologies similar to our framework and discuss the problems and the challenges such technologies have encountered.

Chapter 3 presents the new framework. We will go into the details of the implementation of each feature. This chapter will start with a design overview and will then

review task implementation, message implementation, message queues, code generation, message strategies, shared memory, and task repositories. By the end of this chapter, the reader should fully understand the core features of the framework.

Chapter 4 presents the evaluation of the task system. Each key feature is measured and compared with existing equivalents. To evaluate features, three applications were developed. The parallel sorting algorithm, sample-sort, has been developed to measure parallel performance. The client-server application measures message throughput with our message and task abstractions. Lastly, the weather station application showcases task discovery and discusses usability and complexity.

Chapter 5 presents the conclusions drawn from the research and proposes targets for future work.

Chapter 2

Background Material

2.1 Introduction

In early 1985, Microsoft released Windows 1.01, showcasing multitasking capabilities [29] on a single-core processor. Two programs could be run, but never simultaneously. In 2002, IBM released Power 4, the first 1Ghz multi-core processor [65]. Not long after, in the early 2000s, Intel and AMD were releasing the first general usage multi-core processors. In a modern computer, core count can vary from 8 to many more. Moreover, in cloud computing, distributed virtual machines can see core counts grow into even larger ranges. An important point to emphases is how quickly these multi-core processors became the norm in modern computers and how programming languages and libraries have had to keep up with the increasing core counts.

Multiprocessing or multi-processor differs from multi-core CPU where the system is composed of two or more CPU. There has been multiprocessing system in mainframes and supercomputers for nearly 40 years [8].

In fact, multi-core processors opened up a whole new field of computer science. Before that, even if it was possible to run two processes, they were never actually executed at the exact same time. With multiple processing units, this was now

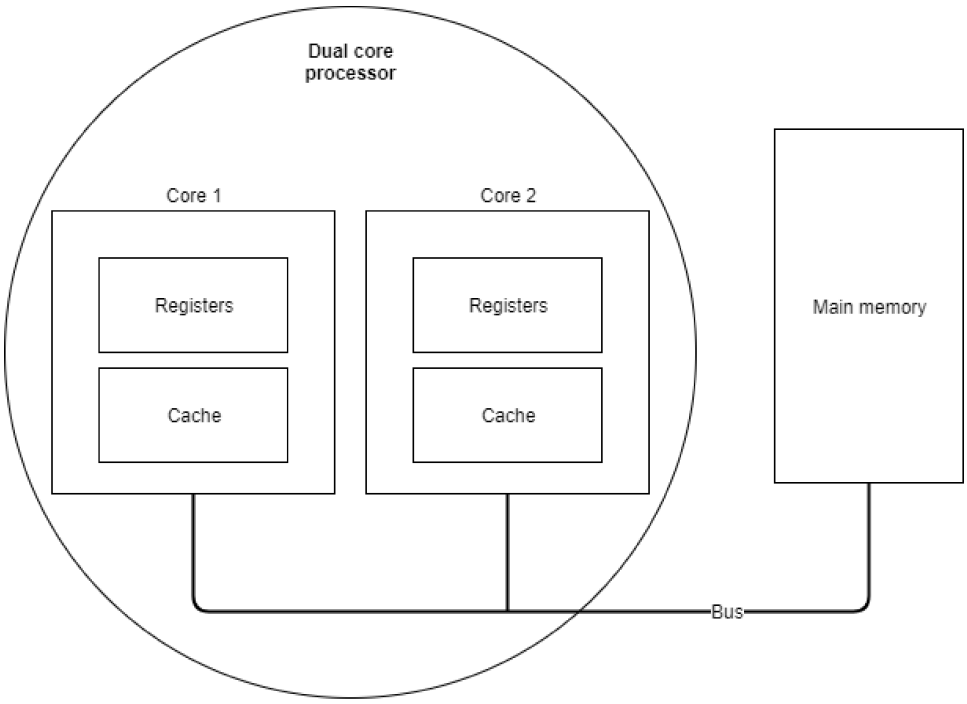

Figure 2.1: A dual-core processor architecture.

possible. Multi-core programming, also known as multithreading, became part of the developer job. The challenge was to program the application using threads and processes in order to harvest the power of having more than one core in the processor. In addition to efficiently using all cores, of course, program performance must also scale well when running on varying core counts. The multi-core programming section in this chapter will go into more detail about the potential problems and challenges developers may face when programming in a multithreaded environment.

Over the years, multiple technologies have evolved to facilitate multi-core programming. Many of these are now part of the standard libraries of various languages. Programming languages usually offer various features to help the programmer such as threads, synchronization primitives, asynchronous tasks, condition variables, mutexes, atomic updates and many more. Some languages have dedicated keywords or

even compiler checks to help the developer. For instance, Java offers a rich selection of classes within its *concurrent* package [16]. At the same time, many libraries were developed to allow developers to build complex parallel and distributed applications, including Akka, OpenMPI, OpenMP, Apache Spark [21] [22] [4] [2].

Even today, it is still quite difficult for developers to use the proper tools or libraries for their needs. Some libraries are either unnecessarily complex or may not be well suited to the problems. On the other hand, if a developer decides to do their own concurrency using primitive programming language features, they are responsible for creating safe code and must test it extensively for synchronization problems. No matter what option is used, the process of developing complex concurrent systems is expensive, in terms of developer resources.

In this chapter, we will explore background materials and existing work related to this thesis. Section 2.2 will discuss multi-core programming and the challenges of properly using it. Then, in Section 2.3 and 2.4, we will review concepts relevant to the actor model and message passing, two core elements of our framework. Lastly, Section 2.5 will discuss existing systems and libraries, their advantages and disadvantages, and the problems they try to solve. At the conclusion of this chapter, the reader should understand what problems the proposed framework is trying to solve and why we have chosen to move forward with the actor model and message passing. The reader should also have a better sense of where the framework could position itself in the industry.

2.2 Multi-core programming

Moore's law [60] states that the number of transistors doubles every two years. While this prediction proved accurate for several decades, mono-core processors eventually reached their limit and multi-core processors took over the market.

In the computer science context, a **thread** is a basic unit of CPU utilization. It has its own ID, program counter, and shares the same process code [61]. A thread is created within a process and is sometimes described as a lightweight process. There are two forms of threads: kernel threads refer to those created and scheduled by the operating system, and green threads, which are entirely managed by an external library. Green thread implementation is usually lighter and offers more control to the developer, making them a very attractive option for application with complex thread requirements.

In a multi-core processor, two or more threads can run at the same time permitting, what is called **multithreading** [52]. Multi-core programming allows code to run in truly parallel fashion. With this capability, developers can now build applications that harvest multi-core performance advantages and, therefore, execute more instructions in a given unit of time.

Manycore processor is a fairly new term that defines a processor built with a large number of cores [44]. These processors are built for a high degree of parallel processing and usually rely on specific hardware. Their massive parallel capabilities come at the expense of single thread process in term of performance and usability, therefore, these processors are not suited for general usage.

2.2.1 Protecting shared resources in a multithreading environment

Multi-core programming presents many challenges for developers. One of them is simply how to code, structure and design multithreaded software, a problem explicitly addressed by the thesis framework described in this document.

Another major challenge is how to handle memory when multiple threads can access it. For example, in order to generate a specific output, a process needs to execute a sequence of operation on a memory location. During this time, another process begins to execute a second sequence of operations on the same location resulting in the first process producing the incorrect output. this general issue is of what is known as a *race* condition [55]. A race condition happens when the application depends on the sequence or timing of two or more threads.

To avoid this problem, developers have come up with multiple solutions, including:

1. Locking
2. Message passing
3. Transactional memory

Firstly, **locking** is the act of reserving a memory location for a certain thread or process. The owner thread is then responsible for locking and releasing it. There are numerous ways to lock memory, some more efficient in specific scenarios. These locking mechanisms are called *synchronization primitives*. One such example is a *semaphore* variable used to control access to a shared resource.

Most modern programming languages offer a variety of synchronization primitives to allow developers to properly lock memory and prevent race conditions. Even with

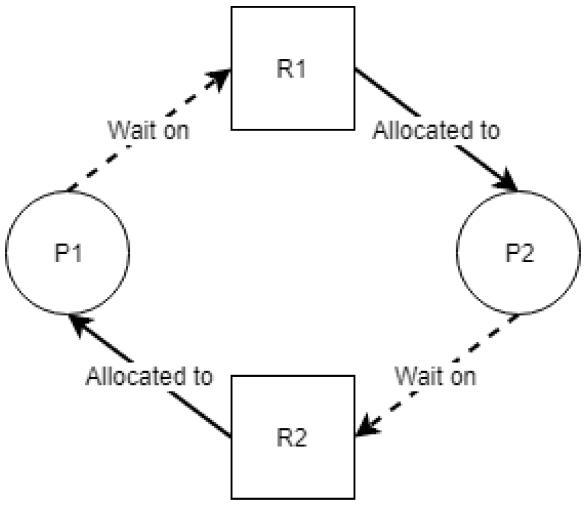

Figure 2.2: Two processes in deadlock state.

correct locking, however, other problems may still occur, with deadlock being probably the most common [36]. A deadlock is a state where a group of threads are each waiting on other threads, preventing the program from progressing further. Figure 2.2 illustrate a common circular deadlock where two processes wait on a resource that is allocated to the other.

Another issue is *starvation*, when one thread is constantly locking a memory location, meaning no other threads can access it. the term *lock contention* refers to the total amount of time threads spend waiting. In fact, increasing the number of threads and cores makes the use of locking less efficient. Specifically, even with good locking strategies, as the thread count grows, the total wait time will proportionally increase leading to more locking contention [63]. One solution is to minimize the requirement for locking by using good programming techniques or by introducing immutable variables [31]. Functional programming language takes the latter approach, and do not usually allow mutable state, instead relying on copying objects to avoid shared states, and therefore preventing race conditions or starvation. This solution does not come

free of course, as copying objects can be expensive.

In the real world, programmers are fully responsible for properly synchronizing their programs. This task ends up being error-prone and hard to validate. First, the developer is responsible for identifying variables at risk and adding protection where these variables are used. This work is even harder on a large codebase. In addition, the developer must really understand the timing of his application in order to not introduce potential deadlocks. The burden of this work falls on the developer with little help from tools and/or the compiler and typically requires extensive testing to validate.

One way to avoid locking is to use **message passing**, where *actors* communicate using messages. An actor can be a thread, a process or an object representing an execution flow. They communicate by sending and receiving messages. Many modern frameworks are based on this technique, allowing them to achieve high concurrency. Distributed systems also utilize message passing mechanisms, as this can be an effective way to communicate over a network. Of course, message passing affects performance because messages must be stored and handled, in addition to the challenges associated with message loss or message corruption.

One major advantage of message passing is the capacity to scale according to the number of actors. Because each actor can store multiple messages in its queue, or send many messages at once, it can do more work before having to wait. Similar to locking, minimizing the number of messages usually improves overall performance. However, implementing an algorithm using message passing instead of locking is fundamentally different. The framework presented in this thesis favors message passing over locking but, as will be seen later, the supporting queue must still use locking internally.

Lastly, **transactional memory** tries to solve concurrency problems by allowing programs to define a set of instructions that must be completed atomically. This mechanism is strongly influenced by the database transaction model. There are two forms of transactional memory: software and hardware. Hardware transactional memory is when supporting hardware detects a conflict and executes a rollback of some instructions in order to avoid the memory conflict. As the name suggests, hardware transactional memory requires the participation of additional hardware components. On the other hand, software transactional memory is provided solely by software and can be associated with a programming language or an existing library. The biggest disadvantage of software transactional memory is that high overhead often results in lower performance relative to classic synchronizations.

2.2.2 Parallel computing

One particular aspect of multi-core computing is parallel programming, where the goal is to boost performance by splitting execution and running the application on multiple cores simultaneously. Algorithms must be modified in order to partition them into multiple simples tasks. Numerous libraries have been developed to make parallel computing at least somewhat simpler and more accessible [21] [22] [10].

The first challenge in this context is to determine a strategy to split the algorithm into simple executable tasks. Some libraries, such as CUDA [10], are directly integrated into the compiler by adding new syntax and keywords. Other libraries, such as Microsoft .NET Task Parallel Library (TPL) [24], rely on objects to abstract the parallel computation. In both cases, developers have to modify their algorithm in order to run it in parallel.

The second challenge is memory management. If all executable tasks have to rely

on a single mutable memory object, then the program will not truly be able to run in parallel. One solution is to take a copy of the data before running the task. This solution may work for a small object, but large objects cannot be efficiently copied. When memory management is poorly done, a parallel algorithm may have high lock contention, leading to poor overall performance.

A recent approach to achieve good parallel performance is harvesting the power of graphics processing units (GPU) to execute parallel computation. CUDA [10] is a framework developed by Nvidia that uses GPUs for general purpose processing. It is directly integrated with multiple languages such as C, C++, and Fortran. It is then compiled by a special compiler. CUDA is used in major artificial intelligence libraries such as Theano [27], Tensorflow [26], and Torch [9] to accelerate model computing, through this approach only works on proprietary hardware.

2.2.3 Distributed systems

Distributed systems are distinct from multi-core or parallel programming by having different applications running on different processes and hosts. They need to communicate in order to do their work and propagate data. Figure 2.3 shows an example of a distributed system. One key aspect of distributed programming is that it usually uses messages to communicate over the network or between processes. Such systems typically require more configuration so that all applications can connect to each other. Networking also adds significant configuration and debugging overhead.

A major difference related to parallel programming is the use of distributed memory instead of shared memory. Because each application may run on different hosts, individual processes do not have access to a large shared pool of memory. Every piece of memory that needs to be communicated has to be written inside a message

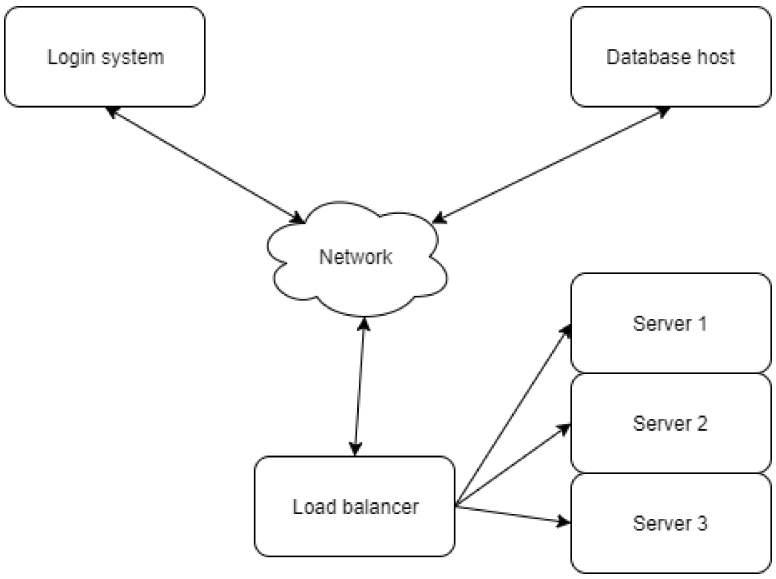

Figure 2.3: A distributed architecture example.

for transportation. Every host also has to connect and be aware of the presence of others hosts, as well as the network topology. Also, in the case of a distributed system over a network, the application does not have access to a global clock and may receive messages in random order depending on network communication capacity.

Because of the complexity of a distributed system, developers usually rely on a library to implement their applications. The library makes it easier to work with the network and will abstract communication between processes. Developers can also build their own distributed program by using low-level socket programming, available in nearly all programming languages.

2.3 Actor model

First presented in 1973, the actor model is a well-studied approach in which each actor is a universal primitive of concurrent computation [46] [30]. It was developed to

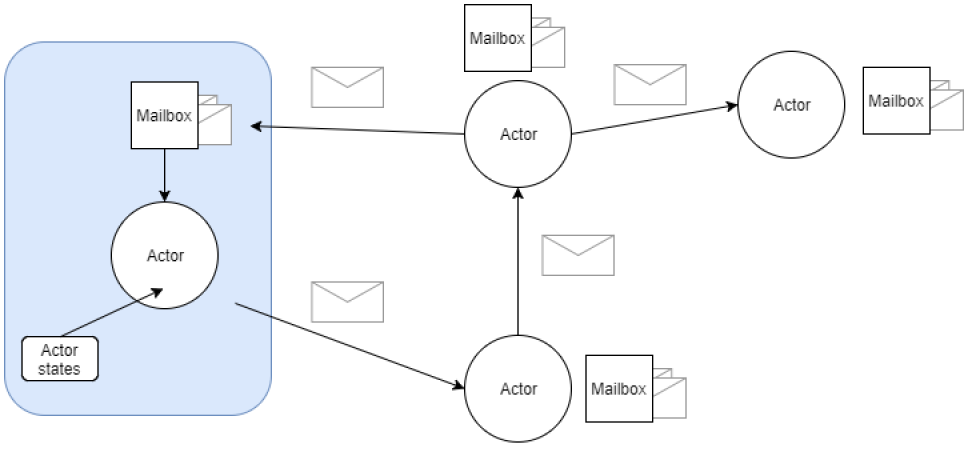

Figure 2.4: Actor model.

exploit massive parallel architectures and provide a general framework for concurrent computation. As foreseen by the original author, the actor model gained momentum when multi-core and massive parallel computers became available. New libraries and frameworks were developed based on this concurrency model.

The actor is described as a computational agent which sends and receives communication, executes computation, and creates new actors. Each actor keeps a local state that evolve over time. An actor is therefore not immutable. The actor model can, in theory, execute any kind of computation.

Following the actor model in its purest form is not always easy or realistic. A study done in 2013 [64] discovered that nearly 80% of actor models written in Scala combined the actor model with other concurrent models. One reason for this is that the actor model requires a lot of development. Another important reason is that the model itself has some limitations. For instance, because actors can not share memory, each data structure must be copied into the second actor space using messages. This implies two copies of the same data - one in the message and a second in the receiving actor space.

Another implementation detail is how the messaging service will be implemented. The model defines the *mailbox*, a queue where each message is stored before retrieval, but does not define any storage order. Different actor model implementations usually have different ways of implementing the message queue. The queue could be first-in-first-out or a priority queue, allowing the receiver to skip through it and prioritize certain messages over others. A priority queue is a very appealing method of ensuring that the actor reads messages in the proper order and ensures the correct flow of the algorithm.

Issues such as the order of message arrival or loss of messages are not directly defined in the model. In practice, an actor can be a kernel thread, a user thread or any object that can process instructions in its own memory. The scheduling is not defined in the model and, therefore, is decided at the implementation level. All of these decisions will affect the cost of developing an actor-based library.

Akka [2] and Erlang [32] are two examples of actor model implementations. They both offer actor abstraction and their own scheduling. Similar to an actor-based approach, our framework will use a task-based paradigm. We will utilize the general concept of actors, but the final implementation may not perfectly fit the model, as this is not the intention. For example, one example the divergence is the need for shared references between actors for large data structures.

2.4 Message passing

Message passing is not only for actor-based models, but is a very generic way to communicate. Message passing can be used to communicate over a network, between processes, between threads or even to execute a remote procedure call. A simple

web-service works using messages. A client sends a request (a HTTP message) over the network, and gets a response message back. All data needed for the request must be written in the message.

One important aspect of message passing systems is whether communication is done synchronously or asynchronously. Similar to a function call, synchronous message passing is when the caller waits for the operation to complete before doing anything else. It is significantly less complex to implement, but does not offer the capability to continue computing while the message is being processed. By contrast, asynchronous message passing is non-blocking and the process can continue computing while messages are being executed. Asynchronous messaging is more complex to implement because it requires a supporting system to queue and deliver messages.

Message communication does not come without performance overhead, especially for asynchronous messaging. Each message has to be created, the data has to be written into it and finally, the message has to be sent to an appropriate destination. This process is repeated at the destination where the message has to be received, read and the action has to be performed. When communicating over the network, the overhead of network communication is so large that the cost of packaging messages is typically not very relevant. However, using messages between threads and processes is a lot more expensive in a relative sense, as a simple function call or a simple lock acquisition is a very cheap instruction to perform.

Asynchronous message storing is usually done in a first-in-first-out queue or a prioritized queue. The data structure has to be properly synchronized using locking primitives. This can lead to scaling problems such as high locking contention. One way to avoid this is to use a *lock-free* instruction offered by the operating system. A

lock-free instruction is, in fact, an atomic operation. A normal lock is built around these atomic operations, but if we only use the atomic part of the lock to manipulate the data structure, we get a lock-free data structure. Compare-and-swap or a fetch-and-add are two common lock-free instructions [45]. One recent article proposes a queue that uses the fetch-and-add instruction [66]. Such a queue could in fact be suited to our own framework.

Despite these complications, it is likely that we will eventually see an increasing number of libraries using messages instead of classic synchronization or function calls. One key reason for this is the increasing need for scalable abstractions in parallel, distributed and cloud systems. In fact, message passing is fundamental in certain concurrency models. For example, a new advancement is on-chip message passing that proposes hardware support for message passing between processor cores [51]. By adding message passing hardware capabilities, it is possible to open up a vast field of research in which message-based frameworks and algorithms could execute and scale faster with these augmented CPU's.

2.5 Previous Work

2.5.1 OpenMPI

OpenMPI stands for the Open Message Passing Interface project. MPI is a standardized and portable message passing standard [18], developed by researchers from academia and industry. The first committee effort was carried out in 1991 and the MPI 1 standard was released in November 1992. Since then, numerous development iterations of the standard have been released and many implementations developed. The current standard is MPI 3.1 [38], while MPI 4.0 is the development effort and aims

to focus on support for hybrid programming models, fault tolerant MPI applications, persistent collections and one-sided communication [17]. In this context, hybrid programming refers to combining shared-memory and distributed-memory programming models [54].

One major MPI implementation is OpenMPI, an open source implementation built by researchers from academia and industry that combine severals existing projects [21]. The project was born as a merge of three MPI implementations: FT-MPI from the University of Tennessee, LA-MPI from Los Almos National Laboratory and LAM/MPI from Indiana University [40]. OpenMPI focus is generic high-performance computing using the message passing standard. Following the merge, the project evolved in sync with the MPI standard. The latest version (3.0.0) was released in September 2017. The project supports the latest standard and offers numerous language bindings and extra tools to support a wide variety of project development targets.

In 2004, OpenMPI developers published a paper defining the goals, concepts, and design of the implementation [39]. Other than a good MPI implementation, their primary objective was to offer a production quality library that supports a wide range of parallel machines, high-performance clusters and other technologies such as TCP/IP, shared memory, Myrinet [34], and Infiniband [57]. Before OpenMPI, it was necessary to choose the appropriate implementation depending on what system was to be used to build the application. OpenMPI merged all these technologies into one general implementation. As OpenMPI is used in complex applications, it also offers optional features to check data integrity and monitor network transmission errors and faulty applications.

Supporting all these technologies for such a long time is difficult. The current implementation is very complex and contains hundreds of functions and configurations. That being said, the goal of the designers was to support a large array of technologies targeted for generic high-performance computation. The library is still in continuous development.

One recent challenge for the industry is to process very large data sets. New technologies such as Spark [4], Hadoop [28] and Google cloud platform [14] have been developed to solve these issues. A recent experiment aimed to compare these technologies with two supervised machine learning algorithms [59]. The results show that OpenMPI outperformed Spark by more than one order of magnitude while, at the same time, stating that Spark offers better data management infrastructure and is better when dealing with errors. The authors concluded that Spark and Hadoop may be preferred due to their greater usability.

2.5.2 MVAPICH

MVAPICH is another MPI implementation similar to OpenMPI, but with a different purpose. The implementation is developed by a group of researchers from Ohio State University. The project is sponsored by numerous groups and is used to power several top 500 supercomputers. The latest software family, MVAPICH2, is based on the MPI 3.1 standard and supports InfiniBand, Ethernet/iWARP and RoCE networking technologies [20]. Others MPI implementation are GridMPI [15] and MPICH-Madeleine [19].

In contrast to OpenMPI, MVAPICH aims to support many different discrete technologies. They offer six software families where each aims to support a very specific array of technologies. For instance, MVAPICH2-Virt is built to support scalable MPI

Family	Supported technologies
MVAPICH2	Support for InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE
MVAPICH2-X	Advanced MPI features, OSU INAM, PGAS (OpenSHMEM, UPC, UPC++, and CAF), and MPI+PGAS programming models with unified communication runtime
MVAPICH2-GDR	Optimized MPI for clusters with NVIDIA GPUs
MVAPICH2-Virt	High-performance and scalable MPI for hypervisor and container based HPC cloud
MVAPICH2-EA	Energy aware and High-performance MPI
MVAPICH2-MIC	Optimized MPI for clusters with Intel KNC

Table 2.1: MVAPICH software families.

for hypervisors and containers, based on a high-performance computing cloud. Table 2.1 summarizes each family and their targeted technologies.

While OpenMPI aims to be a more general-purpose message passing library and MVAPICH is intended for specific technologies, they should both have similar performance results in similar environments. A study done by researchers from Ecole Normale Superieure compared four different MPI implementations [42]. They measured each implementation's performance when used over a long distance network and in a heterogeneous system. They concluded that after proper configuration, each MPI implementations had very similar performance characteristics.

Due to its nature, MVAPICH is likely to be more expensive to implement. It has a more open, research-oriented structure and, as a result is very complex to use. They have an extensive performance section on their website and a list of recent publications. On the other hand, OpenMPI is more adapted to industry. They provide a generic and complete MPI library.

2.5.3 Erlang

Erlang is a general purpose functional programming language designed to build massively scalable real-time systems and distributed fault-tolerant systems [13]. The first release was in 1986 and it was part of a development effort by the Ericsson company. The language was built to improve the development of telephony applications.

In this thesis, we are particularly interested in the concurrency model of Erlang. As Erlang is a functional language and does not rely on mutable state, it doesn't suffer from race conditions. The language is based on the actor model, where each running object is an Erlang process [32], and offers built-in asynchronous message passing between actors. Erlang uses green threads and implements its own scheduling and memory management. The language can then be used to build highly scalable real-time applications with high availability. What makes the Erlang model so flexible is that it uses asynchronous message passing so that threads do not block when sending messages.

At any time in the execution of a process, the user can create a new process by calling the `spawn` function. This function starts a new process with a programmer defined function as an entry point. The spawn function also takes optional arguments and returns a unique process identifier. The exclamation mark symbol `!` is used to send a message. The `receive` keyword delimits a block in which Erlang will retrieve a message. In this block, it is possible to use pattern matching to filter messages in the queue. Code Sample 2.1 shows how to use `spawn` at line 29 and `!` at line 7, 11, and 24. Using these functions, it is possible to implement very complex distributed message models [12] with minimal code.

```
1
2 -module(tut15).
```

```
3
4  -export([start/0, ping/2, pong/0]).
5
6  ping(0, Pong_PID) ->
7      Pong_PID ! finished,
8      io:format("ping␣finished~n", []);
9
10 ping(N, Pong_PID) ->
11     Pong_PID ! {ping, self()},
12     receive
13         pong ->
14             io:format("Ping␣received␣pong~n", [])
15     end,
16     ping(N - 1, Pong_PID).
17
18 pong() ->
19     receive
20         finished ->
21             io:format("Pong␣finished~n", []);
22         {ping, Ping_PID} ->
23             io:format("Pong␣received␣ping~n", []),
24             Ping_PID ! pong,
25             pong()
26     end.
27
28 start() ->
29     Pong_PID = spawn(tut15, pong, []),
30     spawn(tut15, ping, [3, Pong_PID]).
```

Listing 2.1: Erlang ping pong sample [11]

Erlang also offers a repository function to retrieve a process identifier associated with a given name. Using the `register` function, a process can map a process name to a given process identifier. One interesting use of this function is that it can be used to redirect a message to a new process in case of failure or migration. This is key to implementing fault-tolerance and high availability. Process identifier resolution is done automatically when using the `!` symbol.

Finally, Erlang is also natively distributed, which is uncommon for a programming

language. When two processes are on different hosts, they can communicate using a configuration cookie that contains the name of the Erlang node and basic authentication information. When sending a message the developer simply needs to provide the Erlang node name. Each Erlang node is fully independent; therefore, each node has its own registry. To help with this, registry functions can be used on a given node. Because Erlang nodes are decentralized, developers need to understand the topology of the network in order to properly use it. Specifically, because Erlang does not offer an automatic distributed topology function, developers need to be aware of each Erlang node and functions they expose.

Erlang is a complete general purpose actor-based programming language and strongly influences our framework. One drawback is that the language's functionality is built over a virtual machine that adds computational overhead and reduces efficiency. Erlang will never outperform C/C++. Instead, Erlang is built for high throughput computing, making the language not perfectly suited for big data problems.

Another issue with Erlang is the fact that the language simply isn't very popular among developers. In 2017, it ranked 38th in the TIOBE index [25]. In general, functional languages are less frequently used as imperative languages because of fundamental differences with the more common imperative programming style. As a final point, Erlang lacks the object-oriented mechanisms typically favored in modern software engineering.

2.5.4 Rust

Rust [23] is a fairly new language. First released in 2010, it is designed to be a safe, concurrent and practical language. Rust's developers clearly state that concurrency

is becoming very important in modern computing and that software developers do not always have the right tools to do their work properly [7].

Rust offers many modern facilities to support good concurrency. Rust has a powerful ownership system that binds variables to an owner. At compilation, Rust will make sure that only one owner is bound to a mutable state. Using this system, Rust can detect a potential race condition at compilation. The system makes use of two *traits*: send and sync. Send indicates that an object of this type can have its ownership transferred safely between threads. Sync indicates that the object implementation is memory safe. This implies that the object is immutable and can be used by multiple concurrent threads at the same time.

```
1
2  use std::thread;
3  use std::time::Duration;
4
5  fn main() {
6      let mut data = vec![1, 2, 3];
7
8      for i in 0..3 {
9          thread::spawn(move || {
10             data[0] += i;
11         });
12     }
13
14     thread::sleep(Duration::from_millis(50));
15 }
```

Listing 2.2: Rust thread spawn and mutable states.

In code sample 2.2, the program will not compile because the reference is owned by all three threads. In Rust, references are immutable, hence the compiler error. One way to make this code work properly is to copy the data. In Rust, every standard library data structure is provided with extensive concurrency features such as cloning and immutability. By taking a copy of the object, the thread has only one reference

and can now be the owner of the object and, hence, modify it.

Rust offers inter-thread message communication using *Channels*. These are used to send signals and optional data. Channels allows one thread to wait on other threads that may include response data. Finally, Rust offers an array of classical synchronization primitives such as mutex, atomic and barrier.

Rust is a fairly new language trying to take its place in the vast ecosystem of programming languages. It offers good concurrency solutions, but lacks comprehensive choices. Their message framework offers too little to build either distributed or complex message-based applications. In addition, the language does not offer an actor abstraction and still relies on the developer to carry out the synchronization.

2.5.5 Charm++

Charm++ is an object oriented parallel programming language based on C++ [49]. It uses actors and message-passing to abstract parallel execution through shared objects. The language was proposed in 1993 and is still maintained as of today [5].

At the root of the programming language we can find the `charmc` compiler that will combine augmented c++ files, headers and interface definitions into a portable executable. The compiler is not natively compatible with C++ and cannot interface with standard C++ libraries. The interface file defines which functions can be remotely invoked. Listing 2.3 shows a simple interface example.

```
1 module hello {
2
3   array [1D] Hello {
4     entry Hello();
5     entry void sayHi(int);
6   };
7
```

```
8 };
```

Listing 2.3: Charm++ interface definition example

Charm++ is intended to run in a multi-processor cluster or supercomputer. Through its runtime, the program will be distributed across all processors, with remote execution transparent to the developer. The runtime performs scheduling, load balancing and fault-tolerance operations. In general, the language is intended to simplify developer work when implementing parallel algorithms.

The biggest issue with this language is the fact that it is not natively compatible with others C++ libraries. To compensate, Charm++ team offers tools to bridge with CUDA, OpenMP and MPI [6]. The language is mainly used by research teams to implement complex parallel.

2.5.6 OpenMP

OpenMP [22] is a programming library used for parallel execution. It takes the form of compiler directives, libraries, and configuration options to execute code on multiple processors and cores. The library offers bindings to numerous programming languages and operating systems. OpenMP is developed by a non-profit organization and is sponsored by industry [37].

```
1  int main (int argc, char *argv[])
2  {
3  int nthreads, tid;
4
5  /* Fork a team of threads giving them their own copies of variables */
6  #pragma omp parallel private(nthreads, tid)
7    {
8
9    /* Obtain thread number */
10   tid = omp_get_thread_num();
11   printf("Hello␣World␣from␣thread␣=␣%d\n", tid);
12
```

```
13    /* Only master thread does this */
14    if (tid == 0)
15      {
16      nthreads = omp_get_num_threads();
17      printf("Number␣of␣threads␣=␣%d\n", nthreads);
18      }
19
20    }  /* All threads join master thread and disband */
21
22  }
```

Listing 2.4: OpenMP hello world

The library uses a fork and join model [56] to split compatible code blocks and distribute the execution over multiple cores. The developer is responsible for targeting code that can be forked, as seen in code sample 2.4. The library can be used on multi-core processors or within a supercomputer with multiple processors.

OpenMP works on shared memory and relies on locking and copying to resolve shared states. It can suffer from lock contention if the implementation is not optimized for parallel execution. It is recommended to use good programming practices and avoid shared data structures. OpenMP also suffers from scalability problems when there is an large number of threads. It has been shown that the overhead growth is linear or super-linear with the number of threads and cores [48].

Modern architectures are turning towards hybrid models in which OpenMP can be used with MPI. Typically, OpenMP is used for parallel execution within a node, while MPI is for communication between nodes. One article published in 2009 [58] compared hybrid, OpenMP-only and MPI-only architectures on a common set of problems. They concluded that a hybrid model can help scalability greatly, but pointed out that it is hard to find the right balance between parallel execution and inter-node communication cost. They propose fully optimizing for parallel execution

before adding nodes and network communication. They also pointed out that the lack of standardization between the different systems and network architectures prevents full optimization of the code.

OpenMP is not the only library for parallel execution. CUDA [10] is developed by Nvidia to run parallel code on GPUs. CUDA is extensively used in machine learning to compute models using the GPU's parallelization capabilities. Both CUDA and OpenMP provides additional languages instructions to parallelize execution, and neither provides message passing capabilities.

2.5.7 Transactional memory on many-cores with network-on-chip

Transactional memory [43] aims to simplify concurrent programming by grouping memory accesses in a transaction. The concept is very similar to a database transaction system. Transactional memory can be implemented for memory that is shared between threads and processes. It works on an *optimistic* concurrency control, in which each thread can enter a critical section but may need to abort in the case of a conflict. Transactional memory can be implemented by software or by hardware. Software implementations tends to add a significant overhead by monitoring each shared state and transaction [35]. Hardware transactional memory tries to solve this by having dedicated hardware that detects conflict and executes rollback on shared states.

One interesting project is TM^{2}C [41], a transactional memory protocol for many-core systems. TM^{2}C explores the use of network-on-chip for low communication latency between cores. Their library comes with FairCM, a distributed contention manager that ensures transaction termination and fair use of the transactions for each

core. Their experiments scale well on many-core processors for hash and map-reduce operations.

Transactional memory is still in early development. It faces numerous challenges before it can be considered to be an equivalent model to classic methods such as locking and message passing. Most software transactional memory is less efficient than other models due to the computation overhead.

2.6 Conclusions

In this chapter, we introduced concurrent programming and described its relevance to this thesis. We defined the actor model and message passing, two core concepts of our proposal. We note that many existing libraries focus either on a single problem, or specifically on parallel execution or distributed systems. Our framework intends to support both parallel and distributed capabilities.

Many existing libraries suffer from high complexity and low usability, in part because of how quickly the market has evolved around massive parallel architectures and high scalability. Using Erlang and Rust as motivation, we will explore the idea of integrating message-based concurrency with a broadly use high performance programming language. The major challenge will be to offer robust task and message passing functionality without suffering from performance loss and development complexity.

Chapter 3

Task-based message passing framework

3.1 Introduction

We propose a task-based message passing framework to support rapid development of reliable, scalable applications for parallel and distributed systems. The framework is an intuitive and unified system that can support parallel, concurrent and distributed execution. To achieve these goals, we built a lightweight messaging framework that provides central message passing communication. This chapter will present the details of each major feature of that framework.

First, we will discuss the general design of the framework. This section gives a general overview of the library from a programmer's perspective. We will then describe how the tasks and messages are implemented and how to use them effectively. These two sections are important because they expose the developer interface to the framework.

The remaining section will review the internal implementation. We will cover how we use shared memory to store message queues and abstract inter-process communication. Because the framework is intended to be integrated into a novel C compiler,

we next explore the use of generated code to provide an object-oriented paradigm. We then describe how the developer can use message strategies to optimize the flow of application content. We complete the chapter with a discussion of the process repository.

We emphasize that the central goal of this framework is to provide simple and efficient task and message abstractions. Each component was created with this objective in mind. As much as possible we will highlight the developer's expected contributions relative to the library functionality.

3.2 The framework

Multi-core programming has been accessible to developers for quite some time. For example, programming languages usually offer threads and locking primitives. At the same time many developers rely on external libraries to build complex parallel or distributed application.

In either case, an application will at some point have to share resource between different threads or processes. As noted, one solution is to lock that resource, use it, and then release the lock. Another approach however, is to use message passing.

The task-based message passing framework proposed in this thesis will use the C language, augmented with a extended syntax. The augmented backend C code will be compiled by a novel compiler and then fed to a standard C compiler. Figure 3.1 summarizes this process.

The C programming language is well suited for our project for a number of reasons, including:

- The generated C code will be compiled by a state of the art C compiler that will

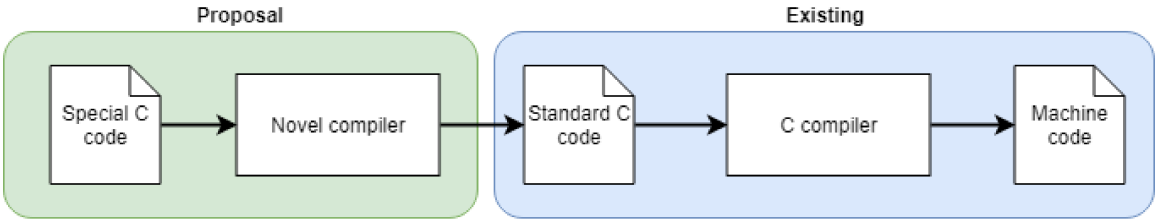

Figure 3.1: Compiling process.

produce native code portable to virtually all modern computing environments.

- C is well suited for high-performance computing.
- The C language has a well-known syntax and an enormous user base that generated over time a large eco-system of tools and libraries.
- C is an imperative language with a well-understood programming model. For many, this will be more comfortable and accessible than functional languages like Erlang and Haskell.

The original language does not offer native message passing capabilities without using external libraries. Our goal is to offer simple and efficient message passing and task abstractions in C while profiting from the large eco-system it offers.

As a concrete example. the use case described in Figure 3.2 will be relatively easy to implement using tasks and messages. In this case, tasks would represent the weather service, as well as each client. When a client wants to find out about the weather, it sends a subscribe message to the service. The weather service will then broadcast the weather to every subscribed client. For the developer, the work would be to implement a task that represents a station and a task for the client. This approach would require at least two new message types: a subscribe message and a broadcast message.

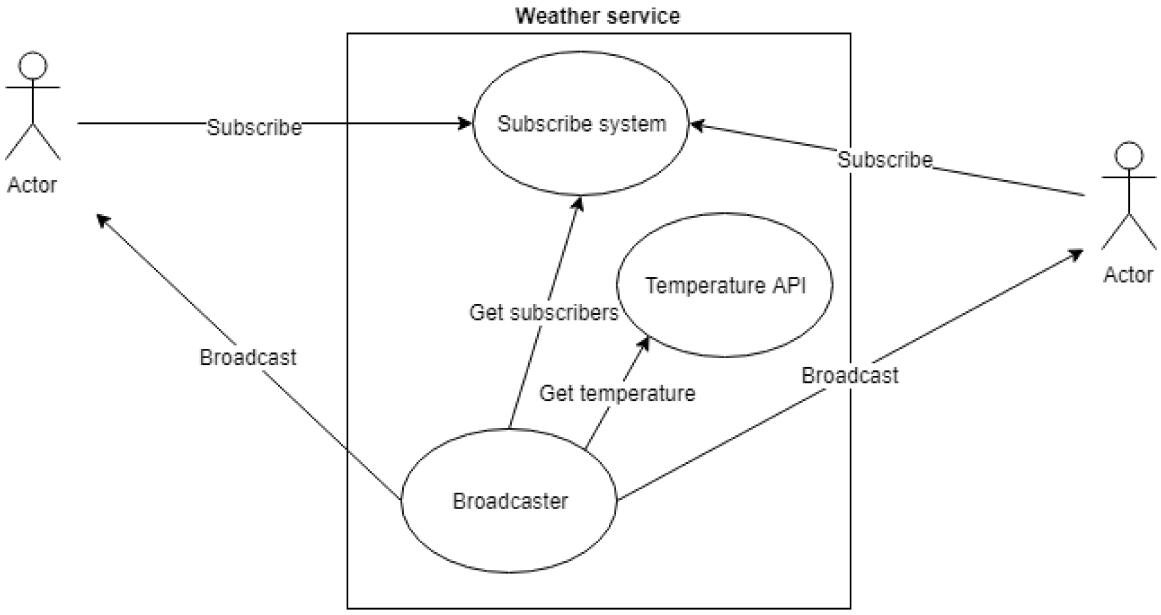

Figure 3.2: Simple weather use case.

3.2.1 Terminology

Before we look deeply at the framework, we want to clarify the terms we will be using during this thesis.

Task. A task is a lightweight process in a state of execution. It is spawned from the code, has his own memory and can communicate through message-passing with another task.

Message. A message is used to communicate between task. Data is embedded within the message and then transmitted to the destination task through shared memory.

System. The system is the central module loaded at the beginning of the execution phase. The module contains all functions needed to manage tasks and send and receive messages.

User code. User code is what the developer will directly construct. In figure 3.1

it would represent the special C syntax that will be fed to the novel compiler.

Generated code. The generated code is the result of the initial compilation. It is standards compliant C code and will be fed to a standard C compiler.

Shared memory. Shared memory represents blocks of memory that multiple processes have access to at the same time. We use it to store messages and system memory. From the process view, it is no different than regular memory.

3.3 General design

The *system* is the central library module of the framework. It contains functions to create and destroy message queues, send and receive messages, and provide message flow strategies and repository functions. The root application process has the responsibility of initializing the system. We will see later how the system is initialized when two or more processes are used.

After the system is initialized, a task can be created. Similar to object-oriented programming, tasks and messages are instantiated through a create function. The system will generate the message queue and assign a unique identifier to the task. A task can be divided into two logical sections: the *user* code, and the *generated* code. The generated code is transparently produced by the supporting compiler. The generated code defines the interface used to send messages, employ message strategies, and handle message types and repository functions.

In contrast, the task's user code is where the developer will implement the business logic, as well as the message management logic. In the task implementation Section 3.4, we will review two different approaches for receiving messages. A full task implementation can be found in Appendix A.

Function	Description
`System_create`	Creates the system and initializes the shared segment
`System_acquire`	Acquires an existing shared segment
`send`	Sends a message to a given task
`receive`	Receives a message from a given task queue
`dropMsg`	Drops a message from a given task queue
`getMsgTag`	Gets message tag from a given task queue
`createMsgQ`	Creates message queue for a given task
`destroy`	Destroys the system and deletes the shared segment
`message_notify`	Puts a task to sleep if there is no message in the queue
`message_wait`	Yields the CPU
`message_immediate`	Returns true if the message queue is empty for a given task
`repository_set_name`	Sets a repository name for a given task
`repository_get_id`	Gets a task identifier associated to a given name

Table 3.1: System functions.

Once the task has been instantiated, the next logical step is to start sending and receiving messages. In fact, messages are also defined with a generated and a user part. The generated component does the heavy lifting and manages creation, cloning, rebinding, and defines the underlying data structure. Section 3.5 on message implementation contains details about the internal structure and rebind function. The only work that remains for the developer at this point is handling state by providing the desired data accessors.

The system functionality, as well as each task message queue, is stored on shared memory segments. This allows multiple processes to access the same system and perform inter-process communication. This is a key design feature of the framework. The message queue is implemented as a fixed-size circular array. Each message's data

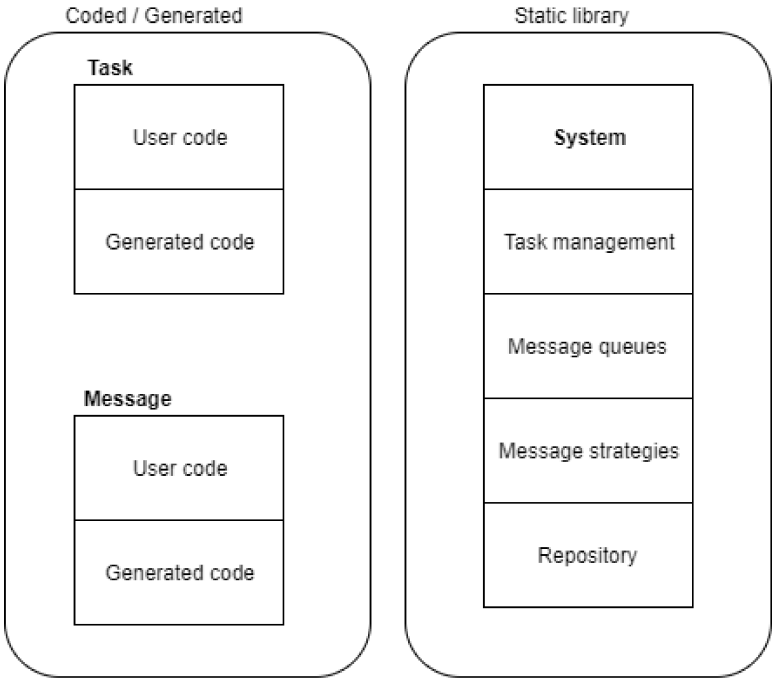

Figure 3.3: General framework design.

is linearized inside the shared segment, and the system shared segment contains all task information.

Figure 3.3 illustrated how the framework is structured. To summarize the design, the system is the central library module and creates tasks and handles message communication. The system also offers repository and message strategy functions. We will cover these later in this chapter.

As for the developer, their work can be broken down into three main aspects: initializing the system, defining messages and implementing task logic. This makes the developer's work very simple, as the library is responsible for performing complex task and message handling.

3.4 Task implementation

Similar to the actor model, a task is an executable unit. It has local memory and functions to communicate with other tasks. In our framework, a task is a running kernel thread and it is separated into two parts - the generated code and the user code. The code sample 3.1 provides an example of what the developer would write in the file to define a task and base functions.

```
1  // Include message type to use
2  #include "IntMessage.h"
3  #include "StringMessage.h"
4
5  task TaskA {
6          // Define tags
7          tag tag_a = 0;
8          tag tag_b = 1;
9
10         // States
11         int my_int_state = 0;
12         char my_char_state = 'b';
13 };
14
15 // Required functions
16 static void start(TaskA this) { /* code */ }
17 static void receive(TaskA this) { /* code */ }
18
19 // Message handling functions
20 static void handle_tag_a_msg(TaskA this, IntMessage message) { /* code
21 static void handle_tag_b_msg(TaskA this, StringMessage message) { /* co
22
23 // Task related function
24 int myFunction(TaskA this, int a, int b) { /* code */ }
```

Listing 3.1: Task syntax example

Function	Description
`send`	Interface to system send function
`receive`	Task receive function
`TaskName_create`	Creates the task
`start`	Task start function
`run`	Thread entry point
`handle_*Msg`	Task messages handling function. Each message type has its own generated function.
`message_notify`	Puts the task to sleep if there is no message in the queue
`message_wait`	Yields the CPU
`message_immediate`	Returns true if the message queue is empty for the task
`repository_set_name`	Sets a repository name for the task
`repository_get_id`	Gets a task identifier associated to a given name

Table 3.2: Task functions.

3.4.1 Generated functions

The compiler will examine user-defined tasks in order to generate supporting functions. Table 3.2 contains the list of functions available when working on a task. Most of these functions are incorporated into the generated code. We will discuss how the code can be generated in Section 3.8.

3.4.2 User functions

The developer is required to implement two functions: `start` and `receive`. As shown in Listing 3.2, the start function is the entry point of the task and it is where most business logic will be or at least called from this function. From this point, the developer can now send and receive messages. The receive functions will handle message retrieval, but it is up to the developer to choose the actual technique.

```
1  static void start(SampleTask this){
2          // ..do something
3
4          // Check for first message
5          receive(this);
6
7          //... do something else
8
9          // check for second message
10         receive(this);
11 }
```

Listing 3.2: Start function code sample.

We identify two simple approaches: direct receive and loop receive. The direct receive will not loop until a message has arrived and will directly return control if no message is present. The looping receive will loop until a message has arrived before returning control. In general the looping method is the most flexible and we used it in all of our examples. Listing 3.3 provides a code sample of the receive function for our weather application. Later in the message strategies section, we will introduce the `message_immediate` function, a mechanism that allows the programmer to check whether any message is available at any moment. Mixing the looping receive method with the `message_immediate` function allows the highest level of versatility while keeping the complexity low.

```
1  static void receive(WeatherClientTask this){
2          int tag = Comm->getMsgTag(Comm, this->taskID);
3
4          // Looping until a message is present
5          while (tag < 0) {
6                  tag = Comm->getMsgTag(Comm, this->taskID);
7          }
8
9          Message msg;
10
11         // match the message to the right message "handler"
12         switch (tag) {
13         case WEATHER_STATION_NAME_MSG:
14                 msg = Comm->receive(Comm, this->taskID);
15                 handle_WeatherStationNameMsg(this, (TextMsg)msg);
16                 break;
17         default:
18                 Comm->dropMsg(Comm, this->taskID);
19         }
20 }
```

Listing 3.3: Looping receive code sample.

In the `receive` method, the message *tag* is used to call the appropriate handling function. The message tag is different from the message type in that it is defined in the task. The message type refers to the concrete code that is used to implement the message, while the message tag is a simple integer used to identify the message from the application or programmer perspective.

Handler functions are associated with tags defined in the task. They are declared in the generated code, but the implementation must be done by the developer in the user code, as illustrated in Listing 3.4. It is similar to the concept of an abstract interface, where each function must be defined, in order to avoid a compiler error. Each tag is associated with one message type, but multiple tags may use the same type. The relevant handling method is called in the receive function.

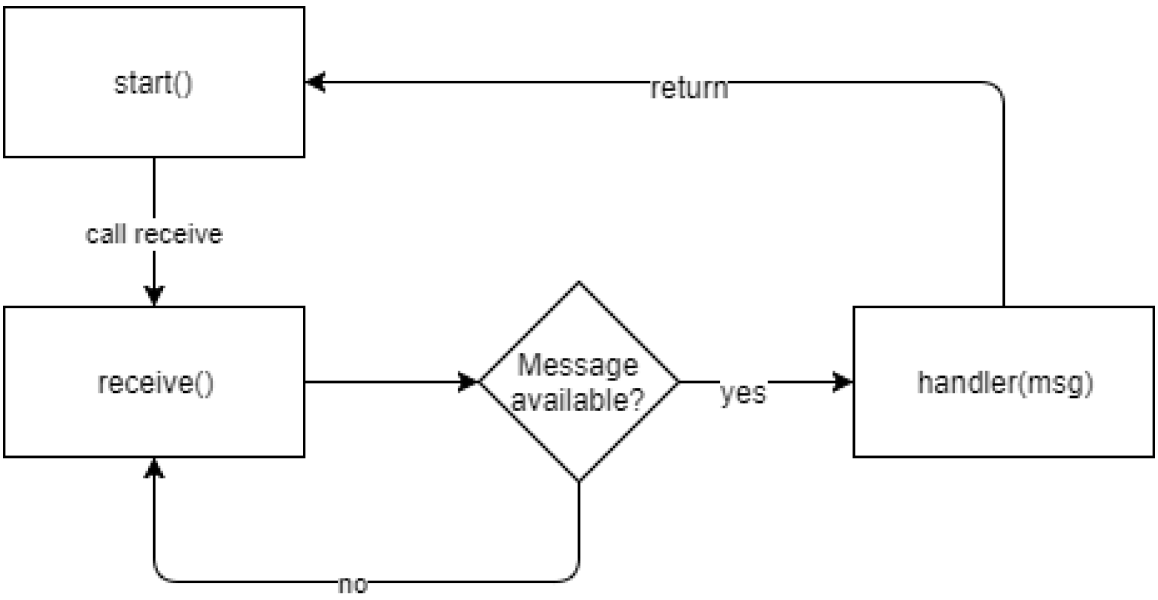

Figure 3.4: Task code flow.

```
1 static void handle_WeatherStationNameMsg(WeatherStationTask this,
2                                          TextMsg textMsg) {
3         printf("Weather␣station␣received␣name␣:␣%s\n",
4                 textMsg->getMsg(textMsg));
5         strcpy(this->station_name, textMsg->getMsg(textMsg));
6 }
```

Listing 3.4: Handling function code sample.

The flow of the receive function can be summarized by Figure 3.4. The `start` function is called directly after task creation. In this function, we can send and receive messages. When calling `receive`, we loop until a message is present in the queue, at which point we call the associated handler function. The handler function will process the message, make local changes, and then return control. If we want to send a message, we must first create it and then use the generated function `send`, along with the destination task identifier.

The task is a central part of the framework and the applications should be designed

with the task in mind. That said, the developer's work will simply consist of defining tasks and implementing business logic through message communication.

3.5 Message implementation

Following the task, the message is the second core concept of the framework. Messages are sent between tasks and stored in a queue. Implementation is similar but requires some additional work by the developer. We note that an object-oriented paradigm in which messages are instantiated (with some limitations) can be used with polymorphic rules. The implementation is again separated into two parts - the generated code and the user code. Code sample 3.5 provides an example of what the developer might write to define a sample message. A full message implementation can be found in Appendix A.

```
1
2  message IntArrayMessage {
3          // Data structure
4          int size = 0;
5          int* values;
6
7          // Functions
8          int getValue(int pIndex);
9          int getSize();
10         void setValues(int pCount, int* pValues);
11 };
```

Listing 3.5: Message syntax example.

Each time a message is sent, its payload will be cloned inside the message queue and when received, the message will be cloned again into the process memory space. By doing this, the original data is safe from race condition. Note that data copying is a fundamental element in virtually all message passing system. When a task receives

a message, because it is a copy, it can modify the data without any risk of synchronization problems. The message should do a deep copy of the data but, as discussed later, we leave the developer the possibility to pass references to avoid copying large objects. In our parallel sorting implementation, for example, we use a message to pass the reference to an array of different tasks. Passing pointers will not work outside the process space, and therefore cannot be used in inter-process tasks.

3.5.1 Message data structure

The underlying data structure of a message is a simple C struct with message data configured using user-declared functions. Each user-defined message has a common structure, similar to a parent class in object-oriented programming. The UML diagram, shown in Figure 3.5, depicts the object-oriented message model used within the framework. Of course, the C language does not provide objects natively. Instead, the base message structures are produced by the code generator. The developer is only responsible for implementing functions from the message sub-class.

When defining message data, it is important to note that the message has to keep track of its total size. Listing 3.6 shows a function example from a message that keeps track of its size by setting the `msg_size` variable from the base message structure. We need this value so we know how large the message will be in the queue.

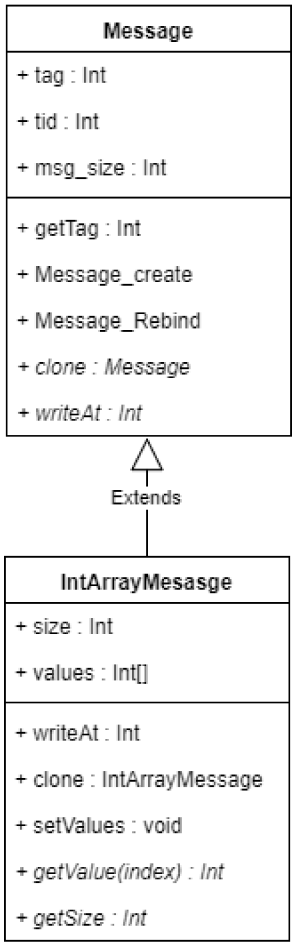

Figure 3.5: Parent message in OO paradigm.

Function	Description
`MessageName_create`	Creates the message
`MessageName_rebind`	Rebinds functions and pointers to the local process space
`clone`	Performs a copy of the message
`writeAt`	Linearizes the message to a given position
`getTag`	Returns message tag
`getSize`	Returns total message size

Table 3.3: Message functions.

```
1
2  static void setValues(IntArrayMsg this, int count, int val[]) {
3          this->size = count;
4          this->values = malloc(this->size * sizeof(int));
5          for (int i = 0; i < count; i++) {
6                  this->values[i] = val[i];
7          }
8
9          // Size tracking
10         this->msg_size = sizeof(struct IntArrayMsg) + (count * sizeof(i
11 }
```

Listing 3.6: Keeping track of size

3.5.2 Generated functions

The relevant generated functions are `create` and `rebind`. The create function is straightforward and consists of allocating memory for the underlying data structure and associating function pointers to the process space.

Because messages may be sent by another process, the rebind function has to associate function pointers with process space addresses. The rebind function should be called just after recovering the message from the message queue. However, when recovering a message from the message queue, it is not possible to know what its concrete type is. To address this constraint we implement a mapping between a

unique message type identifier (*tid*) and a rebind functions array. The mapping is generated at compilation and is detailed in Section 3.8.

3.5.3 User functions

On the user side, the developer has a number of functions to implement. First, the `writeAt` function is used when copying a message to the shared segment of the message queue. The role of this function is to store the entire message inside the queue, including additional dynamic data. Previously, we stated that the message must keep track of its size while updating data. The `writeAt` function will be called on a position where there is enough space for the message, given its size. Fortunately, the front-end compiler will eventually feature a component that automatically linearizes and tracks the size of framework objects.

Second, the `clone` method will do a deep copy of the message in process memory. This function is used when retrieving a message from the message queue, just after the mapping. The message is copied to the local space and the associate data in the message queue is deleted.

Implementing a new message is a little more work than creating a task, but we support the programmer by using an object-oriented approach and embracing code reusability. So while implementing a new message type is probably the most complex effort, when it is done message types can be reused in other modules or projects. To summarize, message and task implementation represent the developer's primary work in the framework. The remaining sections in this chapter cover the more complex components that are managed by the library.

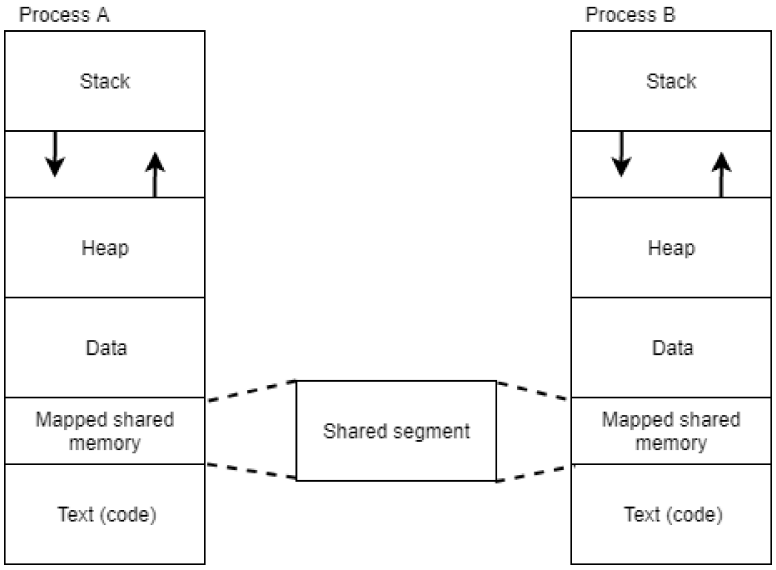

Figure 3.6: Process memory with shared segment mapping.

3.6 Shared memory

One key goal of the task system was to abstract the communication between threads or processes. To do this, we use shared memory to store system code and message queues. Because communication is done over shared memory, it abstracts the communication fabric, allowing tasks to be be in different processes, not just a single process. Shared memory also came with challenges and limitations. Technically, we use memory mapped files with shared memory. Each shared memory segment is, in fact, a file that is never physically written to the disk. The file is mapped to a memory location and this location can be accessed by multiple processes. Figure 3.6 illustrate how two processes map a shared segment.

The first issue was to determine when to acquire the segment, as well as defining its creation parameters. Each shared segment requires a unique name and a size. For a task, the shared memory segment is used for the memory queue, and it is created

at the time of task instantiation. The name of the segment is constructed from the task's unique identifier and the size is established. Note that a task segment may have been created by a process other than the sending process. In this case, the sender will acquire the task segment at the time of the first send.

For the system, the root process must call `system_create` in order to create a shared segment. The size is calculated from the size of the system's underlying structure. Because it is difficult to have a dynamic data structure within a shared segment, the system has a fixed maximum task count that determines the size of the structure. Inside the system shared segment, we store the current task count, the wait and signal thread variables, the condition variables used to put the task to sleep, and the task repository.

The second issue with a shared segment is when to delete it. If we delete them too early, one process may write to a deleted address location, causing a memory access violation. A task could conceivably provide a status to the system before shutting down, but this approach would be prone to race conditions. To solve this issue, we delete all shared segments upon destruction of the system. Note that in a later section, we will cover error handling and task crashes.

The system library is centralized and the developer never directly uses the shared memory. By abstracting the communication fabric in this way, we make the system much less complex. The cost of creating and using shared memory is very low. After acquiring it, the cost to access a shared memory location is the same as any other memory location from the local space of the process.

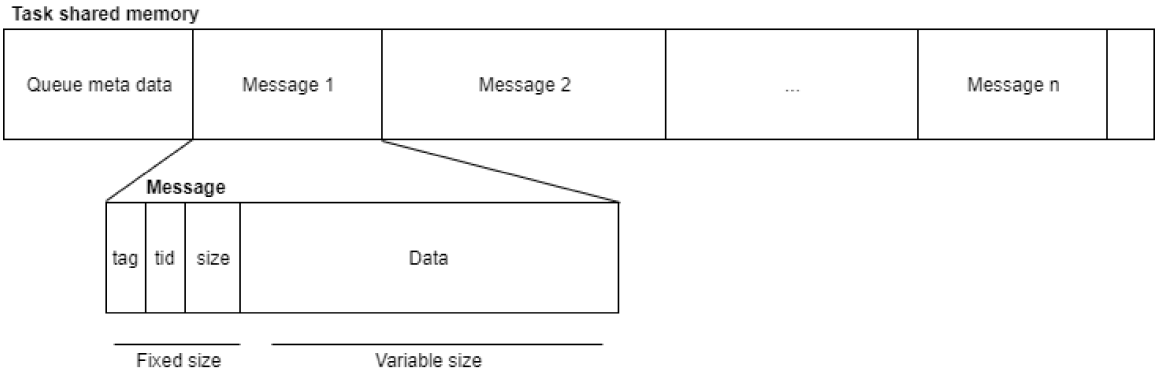

Figure 3.7: Message queue internal structure.

3.7 Message queue

When a task sends a message, the system is actually storing the message inside the destination task's message queue. Each task has one message queue and it is clearly a key component of the framework. The queue is a one-consumer n-producer circular queue and it is entirely managed by the system library, with support from the generated code. The developer does not directly use the queue. As noted above, it is stored inside a shared memory segment, accessible from any process or thread of the host. Figure 3.7 shows the internal structure of the message queue stored within the shared memory segment.

When inserting into the queue, the message is flattened inside the segment by the `writeAt` method of the message. Queue state variables `head`, `tail`, `size` and `rollover_position` are stored at the beginning of the shared segment. Because the queue is circular, it can rollover and store messages at the beginning of the shared segment when it gets to the end. We will discuss how a full queue is handled shortly.

Table 3.4 lists all functions available for the queue. Algorithm 1 describes how the message is enqueued in the shared segment. The first step is to identify the

Function	Description
`CreateQueue`	Creates and initializes a queue in a shared segment
`AcquireQueue`	Acquires an existing queue shared segment
`DeleteQueue`	Deletes a queue shared segment
`Peek`	Gets the message at the head of the queue without removing it
`IsEmpty`	Checks if a queue is empty
`Enqueue`	Enqueues a message at the tail of the queue
`Dequeue`	Dequeues the message at the head of the queue

Table 3.4: Queue functions.

tail position and then write the message at this location. Similarly, Algorithm 2 describes how to retrieve a message from the shared segment. Again, the algorithm identify the read position and then retrieves the first part of the message, including the size. From the size, it can then update the head position to point to the end of the retrieved message. It then uses a mapping to call the `rebind` function associated with the retrieved message type. The mapping is explained in Section 3.6. This is an important step for any IPC based task model because it will reset the function pointer to process address space.

Because the message queue is a shared segment, it must be created or acquired. The queue is created when the task is instantiated and may be acquired at the first send of a given task. Each process has one or more tasks and keeps a list of pointers to each queue of shared segments. The acquire logic is visualized in Figure 3.8. If a task is created in another process, then the pointer is still not initialized in the other processes. As such, the pointer must be acquired by the sending process in order to store the message inside the shared segment. To avoid having all processes acquiring all message queues, we only acquire the queue pointer at the first send. This adds a small overhead to the first send, but avoids excessive queue acquisition.

Algorithm 1 Enqueue function

Input: A message structure with a reference to the write_at function
Output: Error code or success code

```
    {Prepare writing position}
 1: Compute future tail position from message size
 2: if Queue is not rolled over then
 3:    if Future tail will roll over then
 4:       Update future tail at the beginning of the shared segment
 5:       Set queue rollover position to current tail position
 6:    end if
 7:    if Data will be overwritten then
 8:       return  Queue full code
 9:    end if
10: else
11:    if Data will be overwritten then
12:       return  Queue full code
13:    end if
14: end if
15:
16: {Write the message into the shared segment}
17: Call write_at function (message, future_tail)
18: Update queue tail position to future tail
19: if Rollover position > 0 then
20:    Set queue roll over flag
21: end if
22: Increment queue size by 1
23:
24: return  Success code
```

Algorithm 2 Dequeue function

Input:
Output: Message or null if empty
{Prepare reading position}
if Queue is empty **then**
  return null
end if
Set read position to queue head
if Read position is on rollover position **then**
  Update read position at the beginning of the queue
  Reset rollover flag
end if

{Read the message}
Read message structure with size, tid and tag
Update head position to the end of the message
Decrease queue size by 1
Call rebind function using the system mapping
return Retrieved message

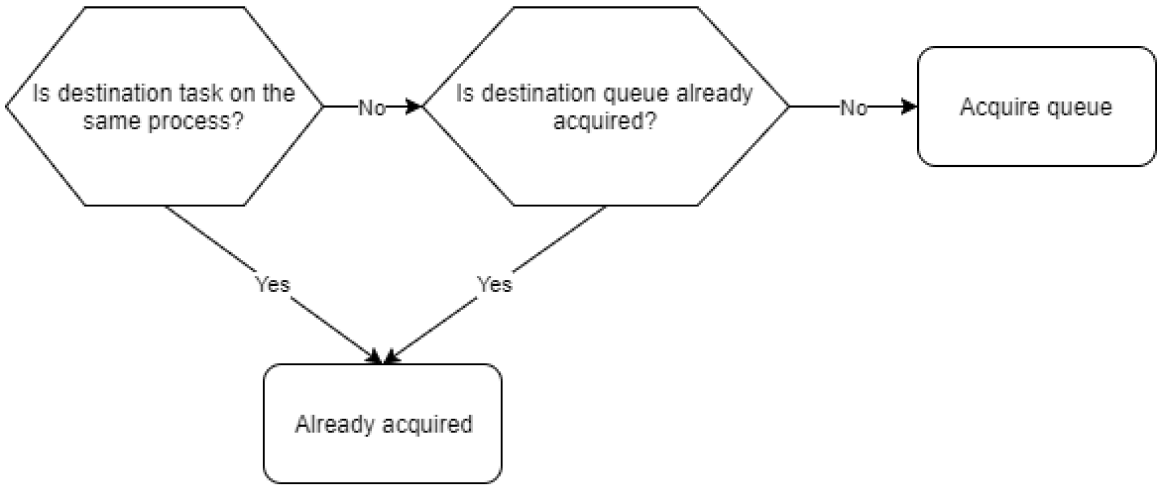

Figure 3.8: Queue acquire flowchart.

The message queue is currently limited to a fixed size. If the queue is full, the `send` method will return an error code that must be handled by the developer. Lastly, the shared segment is deleted at the time of system shutdown, not task deletion. This prevents segmentation faults from an unwanted write to a deleted shared segment.

Not giving access directly to queues was intended to minimize development complexity. In exchange, messages currently have to track size and linearize themselves. Eventually, this will be done by the frontend compiler. Since performance is critical, both enqueue and dequeue functions should theoretically run in O(1) complexity because they do not rely on a linear scan of the queue. Our queue uses classic locking to prevent a race condition when two processes send messages to the same task at the same time.

3.8 Generated code

The framework uses extensive code generation to reduce the work of the developers. This generated code is created during compilation by a new compiler frontend that ultimately feeds code to a standard backend C compiler. Because the final version of the frontend compiler is not yet available, a particular effort was made in this research project to validate the feasibility of the generated code.

Recall that we use generated code for three distinct components:

- Task
- Message
- System mapping

The developer provides **message** and **task** definitions using a special, C-compatible

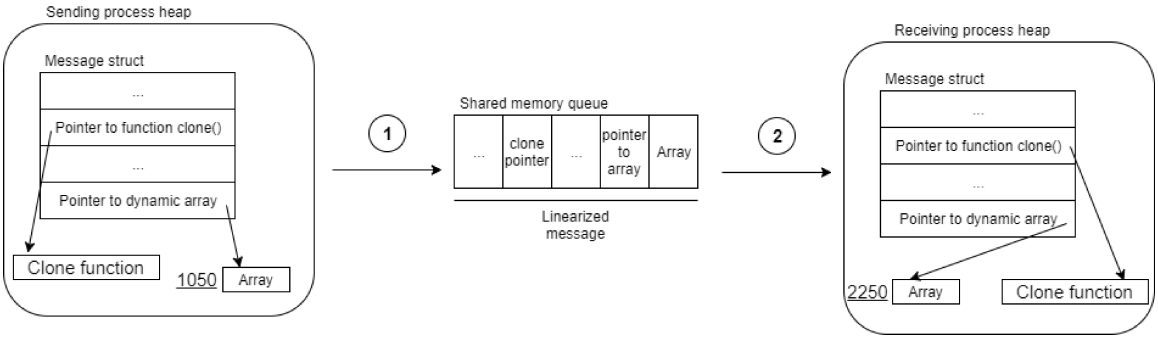

Figure 3.9: Pointer mapping issue illustrated.

syntax. Listing 3.1 and 3.5 provide examples of what the syntax could look like using the final compiler. For a task, the compiler will generate a structure containing the underlying task data. In addition to any task-specific variables, the data structure contains the task identifier and the thread reference. Based on Table 3.2, `create`, `send`, `run`, message strategies and repository functions are all generated. Most of these functions are relatively simple to produce.

In fact, `create` is the most complex generated function. The function starts by allocating memory for the structure. It will then request from the system a new task identifier, initialize data members to appropriate default values and start the thread. At the end of the creation process, the function returns the task identifier to the original thread so that it can be used to send messages.

On the message side, the `create` function is also generated. The primary objective is to bind message function pointers to the appropriate implementation functions. The underlying message structure contains the data and function pointers. This presents an object paradigm to the message model. The `rebind` function is also generated and its role is to rebind pointers after message queue retrieval.

Figure 3.9 shows an example of how pointers are passed from a sending process

to a receiving process. The figure also illustrate that pointers will change if the message is sent between two distinct processes. First, the sender's message stores a pointer to the array location (here it is 1050.) The message is then linearized into the message queue and includes any dynamic data structures that were created by the programmer. While in the message queue, both array and clone pointers still have the same value from the sending process address space. When the receiving process dequeues the message, the array gets a new memory location. Since the pointer still points to the original address 1050, the pointer must now be rebound to the new address 2250. This is why we need a map that associate message type with concrete rebind functions. This map is generated by the compiler and available in each running process.

The **mapping** is prepared during code generation and Figure 3.10 illustrates the retrieval process using the system mapping in order to fix pointers to the receiver's address space. At compilation, the compiler assigns a unique type identifier to each message type. Later, using the list of message type identifiers, it generates a map of identifier/rebind functions. This map is used to call the appropriate rebind function when retrieving a message from a task message queue.

Note that the generated code is not intended to be accessible to the developer. We want to avoid situations where changes are made inside generated functions and new compilation would subsequently erase them.

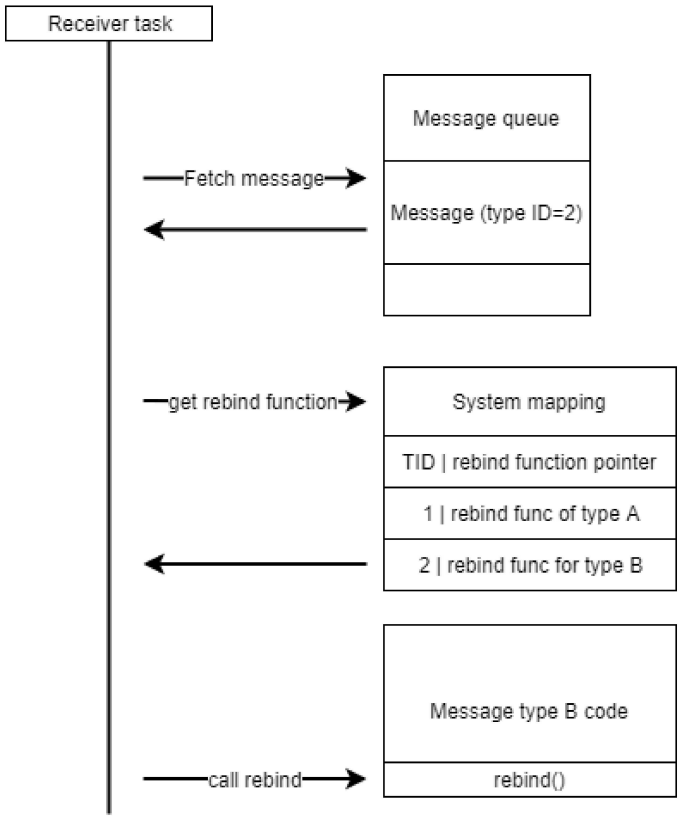

Figure 3.10: System mapping at message retrieval.

3.9 Message strategies

Depending on the application, the time between individual messages can be very long or very short, possibly leading to unwanted waiting periods. If the time between messages is very short, the system has to provide the fastest possible message throughput. On the other hand, if the time between messages is very long, a task could "busy loop" and waste considerable CPU time waiting for a new message.

To solve these issues, we provide three message *strategies* to optimize the message flow of the application. The strategies include `message_wait`, `message_notify` and `message_immediate`.

To avoid busy looping, the `message_notify` function puts, a task to sleep if there is no message in the message queue. In general, a task will wait for a message by busy looping but message-notify offers the possibility of putting the task to sleep and then notifying it of message arrival. In theory, we could notify a sleeping task when sending a message, but that may fail if the notify is called before the task is put to sleep. Missing the notify would therefore cause the task to sleep indefinitely. To avoid this problem, when the system is created, we start a background thread with the sole goal of notifying tasks when at least one message is in the queue. We call this thread the "wait and signal loop". This loop takes a small amount of CPU time but solves the missing notify problem. In practice, this solution turned out to be an important optimization. The `message_notify` solution can be used when a task is expected to wait for a long time between two messages. This messaging strategy should be called just before receiving and will not put the thread to sleep if there is a message in the queue. Using this technique, the worst-case scenario would be if the task just misses the notify and would have to wait for a full loop to occur before receiving it. We used

this strategy in our parallel sorting implementation to put to sleep any tasks that are not currently computing and thus wasting CPU time.

The next strategy is `message_wait` and is intended to provide high message throughput by simply yielding the CPU. This strategy can be used when a task will have to wait for a response right after sending a message, but for a very short delay. The task has no computation to do and expects the next message to come shortly. This strategy is particularly useful in a request-response scenario. We used this strategy in our client-server application.

The last strategy is `message_immediate`, which will return true if there is no message in the message queue. This strategy is generally used to provide a higher level of concurrency. It can be employed to look at the message queue and do additional work if there are no available messages. It can also be used to process a message that acts as an interruption to the normal process. For example, if the application can receive client subscriptions at any time, message-immediate can be used to look at the message queue for new subscriptions and then continue normal processing. We used this strategy in our weather station application, where a client will register with a station in order to receive weather data.

These three strategies are offered to optimize the message flow of the application. That being said, we recommend to not begin the implementation with an explicit strategy and instead wait until the prototype application is complete then find possible bottlenecks and apply the appropriate strategy at that point. When working with the framework, the first goal should always be to solve the fundamental problem with simple messages and tasks.

3.10 Repository

Currently, the unique identifier of a task is generated when the task is created. A task can both receive an existing identifier or create a new task but in both cases, the parent task will have access to the identifier. The repository is used when the current task communicates with an existing task, but it is not possible to directly receive its identifier. In short, the repository maps programmer defined names to identifiers, with this data stored on the system shared segment. To use the repository, a task can assign a name to itself by calling `repository_set_name` and can get an identifier by calling `repository_get_id`. Both of these functions are present in the task's generated code and can be used directly by the developer.

The name has a fixed maximum size because it will be stored in the system shared segment. A repository look-up currently works in O(n) time because it has to perform a linear search. This could of course, be improved to O(1) by implementing a hashmap that works on shared memory.

The repository is optional and tasks do not require a name. The functionality is typically used to communicate with a public task. There are numerous cases in which this is necessary. For instance, if one task acts as a server while others function are clients, the server can register itself with the repository and then be accessed by clients using that name. Figure 3.11 represent this example.

3.11 Summary

In this chapter, we discussed the design and implementation of a task-based message passing framework. The framework offers a simple and efficient way to use tasks and messages for a C application. We exploited shared memory facilities to abstract

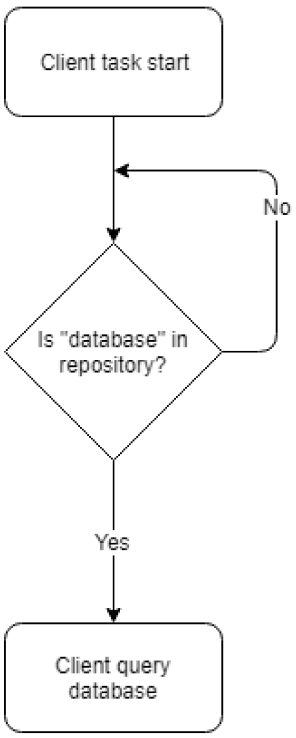

Figure 3.11: Repository usage example.

communication between processes. Within these shared segments, we use a dynamic messages queue to store each message.

We employ an object paradigm for our tasks and messages, though C is not natively an object-oriented language. To achieve this, we explored the use of generated code and presented a special syntax that will be fed to a novel frontend compiler. The combined user and generated code will then be fed into a standard C backend compiler. Given its implementation in C, the message passing code is compatible with any existing C libraries.

We concluded the chapter by presenting our message strategies, used to optimize the message flow of the application, and the name repository, a component that provides name and ID look-ups. In the next chapter, we will evaluate this framework on three applications we built and each will be compared to an equivalent implementation.

Chapter 4

Evaluation

4.1 Introduction

Now that we understand how the framework works, it is time to evaluate it. Our evaluation will aim to measure efficiency and complexity using a set of applications representing common use cases.

We will start by evaluating possibilities for parallel computation using an implementation of a parallel sample-sort algorithm. We compare the result with a standard quick-sort algorithm [47].

Next we look at message throughput. In this case, we built a client-server application that simulates a simple database. We use this application to measure the message throughput of the task system and compare achieved speed with an equivalent application that uses Unix domain sockets [62]. To complete message throughput evaluation we will assess our choice to use shared memory for inter-process communication.

Finally, we built a weather station application in order to demonstrate the use of the tasks repository. That application will also open discussion on complexity and usability of the framework.

Model	vCPU	Memory (Gb)
`m4.large`	2	8
`m4.xlarge`	4	16
`m4.2xlarge`	8	32
`m4.4xlarge`	16	64

Table 4.1: Amazon EC2 M4 instance type specification [3].

4.2 The Test Environment

Experiments were executed on Amazon Web Service EC2 virtual machines. Each virtual machine runs on Linux Ubuntu. Amazon EC2 virtual machines offer various types of compute instances. We used a general purpose M4 type that offers a good balance between computation power, memory, and network speed. The specifications of each M4 instance type we used are provided in Table 4.1.

Each virtual machine offers a variable number of vCPUs. As per the Amazon documentation, a vCPU of an M4 instance type is equal to one core of a 2.4 GHz Intel Xeon E5-2676 v3 processor. Each virtual machine is configured with the following software:

- Ubuntu 16.04.1 LTS (GNI/LINUX 4.4.0-66-generic x86_64)
- CMake 3.5.1
- Gcc 5.4.0
- Python 3.5.2

Each experiment is done using the command line without any graphical components running. Measures are automatically gathered using multiple `python` scripts.

Each script outputs results in a CSV file that is then used to generate charts. Time measurement is instrumented inside the code using the `clock_gettime` function. To scale the number of processors, we restart each machine using a higher Amazon instance type.

4.3 Test Results

The messaging framework is intended to work for concurrent, parallel and distributed applications. It should be easy to use and provide fast message communication. To evaluate the implementation, we developed three applications with the aim to test major features. First, a parallel sorting algorithm, sample-sort, will show the capability of carrying out efficient parallel execution. Second, a client-server application is used to measure message throughput and evaluate message strategies. Third, a weather station application demonstrates how the framework handles life-cycle, error handling and the repository. The last application will also highlight issues related to complexity and usability.

4.4 Sample-sort

The sample-sort is a divide and conquer comparison-based sorting algorithm well-suited for parallel execution [33]. Starting with an unsorted array, the algorithm begins by partitioning the array into sub-arrays called *buckets*. Each bucket then selects a set of samples. Using all collected samples, the algorithm builds a list of *splitters* that defines which values go in which buckets. Finally, each source bucket sends values to the appropriate destination bucket based on the splitters. Figure 4.1 shows an example starting from an unsorted array of 24 values and 3 buckets.

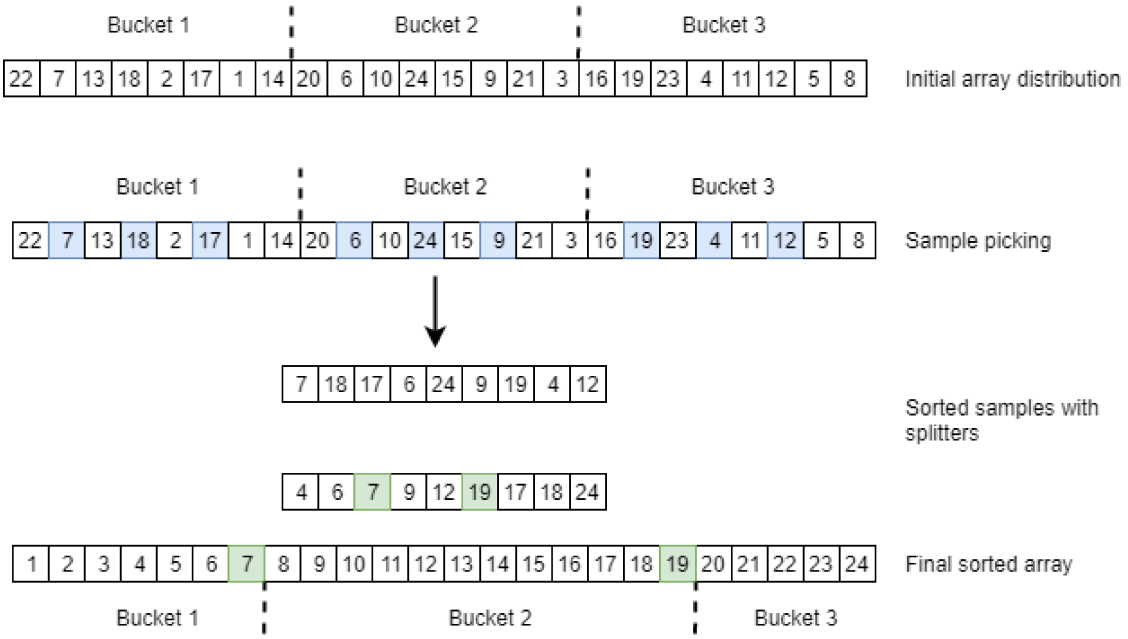

Figure 4.1: Sample-sort example.

To implement this algorithm we used one task per bucket and one central sample-sort task. The diagram in Figure 4.2 summarizes the messages exchanged between the sample sort task and the bucket tasks. We provided the application with the number of buckets we wanted to employ. The sample-sort task is then created and generates one task per bucket.

The sample-sort task starts by sending a reference from the main array to each bucket. We used a reference message type to avoid copying the large initial array to each bucket. Note here that, because the message contains a reference to a memory location, this form of message passing is intended for intra-process parallelism. With the reference, each bucket picks samples in their assigned region of the array. The sample-sort task then collects samples, generate splitters and sends them back to the buckets. Each bucket uses the splitter's information to propagate each value to the

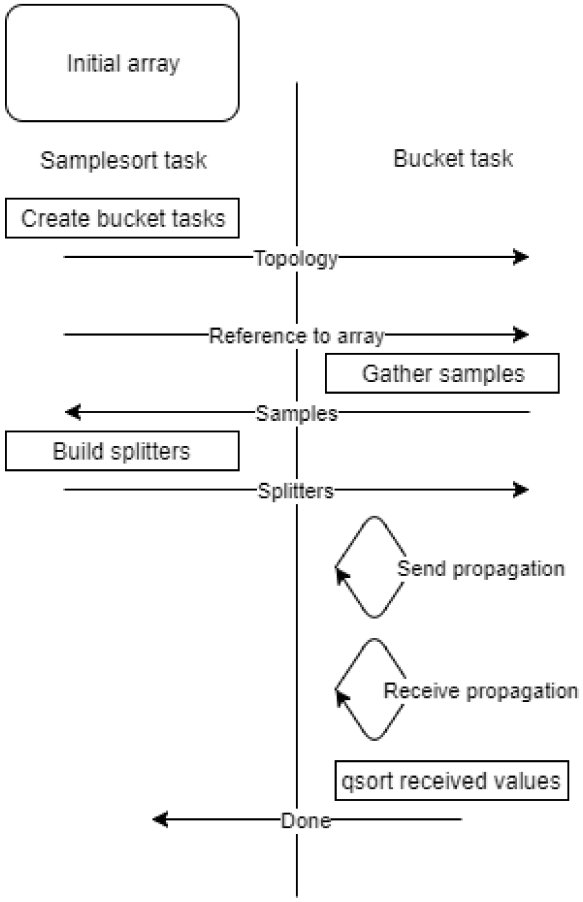

Figure 4.2: Sample-sort diagram.

Read time	Wait time	Bucket 1 propagation time	Bucket 1 sort time
13.417s	9.138s	1.43s	6.98s

Table 4.2: Result example for k=4 and N=100000000

Read time	Wait time
13.676s	8.952s
13.341s	9.189s
13.341s	8.977s
13.344s	9.356s
13.382s	9.218s

Table 4.3: Five examples of measurement for sorting 100 million elements on 4 buckets/cores.

appropriate buckets. Finally, after receiving data from all other buckets, each bucket performs a final quick-sort. When complete, each bucket only contains values that are found between its assigned splitters. Depending on the initial array and the chosen splitters, final buckets may not have an equal number of values.

We instrumented the code with multiple metrics, allowing us to gather time measurements for each part of the algorithm. We gathered 5 samples per test and we averaged the final result. Table 4.2 shows the important metrics from an execution of 100 million values. We provide more exhaustive result tables in Appendix B. Table 4.3 shows an example for samples gathered for the same tests and demonstrates the stability of the test environment. In this context, the wait time is the most important metric and represents the amount of time the sample-sort task waited on buckets before being able to process final results. In short, this is equal to the sorting time. The reading time is the amount of time it took to read the initial values and for each bucket, we also have propagation and sort times. Propagation is the time it took to send values to other buckets, while sort time is the last quick-sort time. The overhead

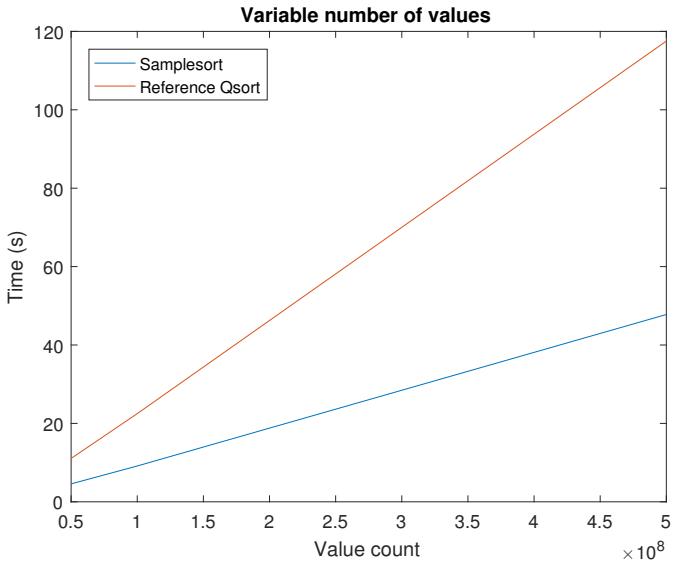

Figure 4.3: Variable array size.

of our implementation is the communication and orchestration with buckets.

We note that sorted results were validated by a script to make sure that the results were correct and no values were lost. The data input is randomly generated and was validated to avoid falling into a scenario where one bucket gets too many values. In real life, this would not be possible, but it wasn't the purpose of our measurement, as this is a generated issue with Sample sort.

We also measured the application on two variables, the number of cores and the array size. We used a standard quick-sort as a baseline reference to compare our result and our speed gain. In Figure 4.3, we vary the initial array size up to 500 million elements and compare the time to a simple quick-sort. In Figure 4.4, we scale the number of buckets and cores to see how the implementation scales with an initial array size of 100 million values. Figure 4.5 is the speed gain when scaling the number of cores compared to the quick-sort reference.

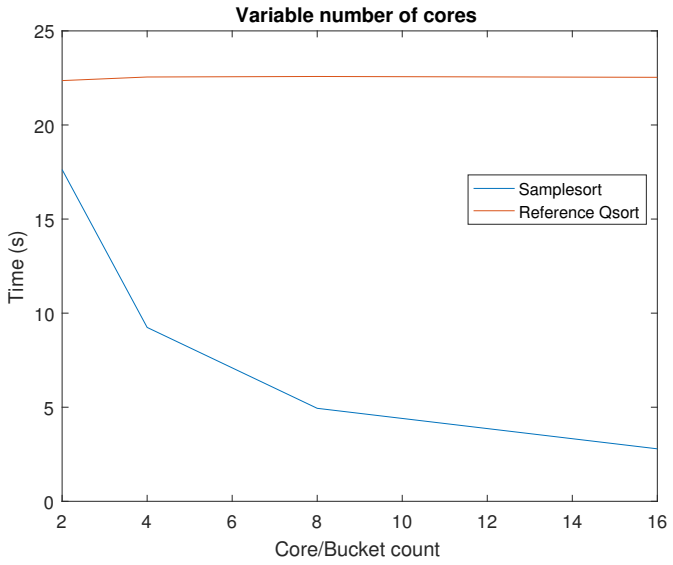

Figure 4.4: Variable cores and buckets.

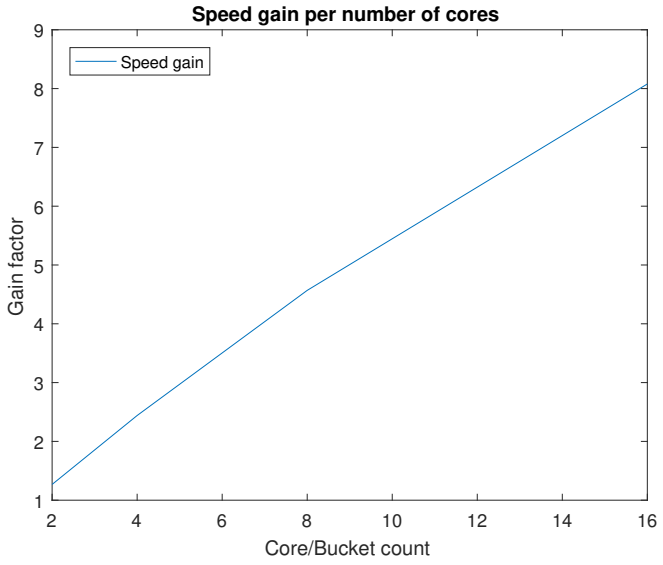

Figure 4.5: Linear speed gain over increasing bucket count.

4.4.1 Discussion

The goal of the application is to display true parallelism, with adequate performance for a parallel implementation. Figure 4.3 shows that the sort time remains stable when increasing the number of data values. In theory, the time complexity of the reference and the sample-sort is O(nlogn), but the data set in this case isn't really large enough to illustrate this directly. However, these results and the comparison with the reference indicates that the implementation does not suffer from additional overhead when processing a larger initial set. Note here that because we use reference messages our messages size stays relatively stable.

In order to refine our measurement, we evaluated how the framework behaved when increasing the number of cores and buckets. In Figure 4.4, we see that as we scale the number of cores, the sort time goes down proportionally. To evaluate the overhead of having more tasks, we calculated the speed gain displayed in Figure 4.5. The speed gain represents how much faster the implementation is compared to the reference. The reference is single-threaded and does not scale with an increasing number of cores. As we increase the core count, the speed gain stays proportionate relative to cores and buckets. This indicate the as we add new cores, the sort time is better in a relative sense. Also, we can assume that no significant overhead is added as we grow the number of cores given that the performance curve stays relatively straight. This indicates that our framework has minimal overhead with increased tasks and communication.

We went through multiple iterations to implement the sample-sort algorithm. In the first iteration, values were copied multiple times. This implementation was slow and used too much memory. The reference message was added at this stage so that

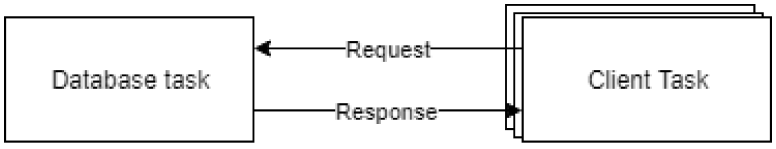

Figure 4.6: Database application tasks.

we didn't have to copy the large initial array. In the second iteration, at propagation, we sent one message per transferred value. This implementation used a large number of messages and slowed execution. We quickly learned that minimizing the number of messages should be the first optimization to be carried out. In the third and final iteration, we added proper message strategies. The results shown in this section were gathered using the final form of the application.

4.5 Client server

The second application aims to evaluate maximum message throughput and the effect of message strategies. To do this, we developed a client-server application that acts as a database. The server receives a request, executes the action and sends back the response to the client. In this case, the server can have multiple clients. Also, we built a reference application that uses the Unix Domain Socket technologies to send and receive messages. In the end, we expect the task system to be able to handle comparable messages volumes.

Similar to sample-sort, we instrumented the code to measure the total time it took for each client to complete their desired request count. We do not measure the total time on the server because each client does not wait for all other clients to be spawned before starting to send a request. We average the time it took for each client to complete their order and we run each test one time only. We could have run each

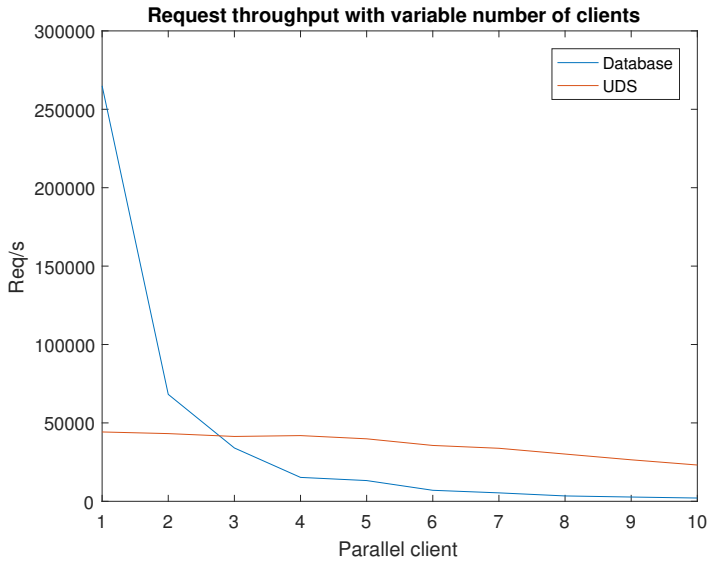

Figure 4.7: Maximum request throughput for parallel clients.

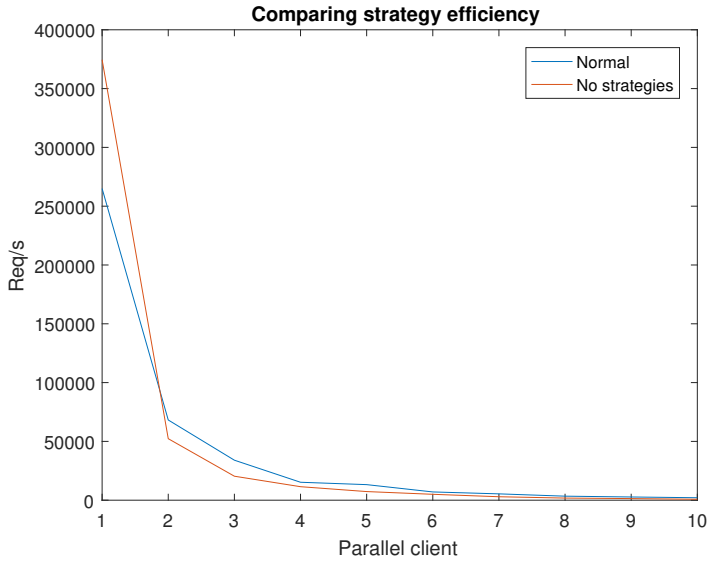

Figure 4.8: Message strategy comparison.

Nb client	Request count	Average time to complete
3	500000	14.700s
7	500000	92.863s

Table 4.4: Result example for 3 and 7 clients

test multiple time but the result would be very similar.

Table 4.4 shows an example of the gathered results. For 7 clients, each took an average of 92.863 seconds before completing their 500000 requests order. By dividing the number of requests by the total time, we compute the request per second each client was able to satisfy during the test. For example, with 7 clients in parallel, clients were able to complete an average of 5385 requests per second.

Figure 4.7 shows how many requests per client our system handles with a variable number of parallel clients. This measurement was done on a 4-core virtual machine. As expected, the task system can support very high message throughput, especially when the CPU is not fully loaded with tasks. We note, however, that performance does decrease as we load the CPU with more tasks than available cores. In this setting, the Unix Domain Socket reference shows very stable progress. We have multiple explanations for the difference.

- Our implementation uses mutual-exclusion to protect the message queue. As the server is retrieving more messages, it forces more clients to wait before adding the new message. The server is the main bottleneck, not the messaging model per se.

- The Unix Domain Socket is a mature technology with many years of development and has utilizes the state of the art message queue and locking mechanism

to optimize the flow. Our prototypes could not be extended to provide the same type of maturity.

- Our application simulates a database while the reference is simply sending and receiving messages.
- The message size in the reference is smaller.

Reworking the shared memory message queue and/or using a lock-free algorithm should improve our results comfortably. We could also adjust the reference to simulate a database load with similar message size. That being said, even this initial framework comfortably handles message throughput in the thousands per second.

With the database application, we also wanted to measure the effect of message strategies. In the application, we used a `message_wait` right after a client sends a message. This strategy yields the CPU, an approach that makes sense because we know that in order to get a response, the database task has to read the message. The likelihood that the server will process the request instantly is very low, so yielding the CPU should help. In Figure 4.8, we show the differences in terms of requests per second when using the wait strategy versus no strategy. When yielding the CPU, we get a small performance boost when the CPU is not fully loaded with tasks. After four tasks, the CPU is loaded and the tasks have a higher chance of busy-looping after regaining the CPU.

As noted earlier, we recommend implementing the application without using any messaging strategy first. Every strategy has a specific optimization purpose. In the database, we used a CPU yield because we knew that we wouldn't have the response message instantly. The sample-sort implementation used a `message_notify` strategy

to put long waiting tasks to sleep while waiting for buckets to finish. The effect of the strategy in the sample-sort was more visible because of the longer busy-looping time. In the database application, responses from the server are not slow enough to generate a gain by putting a task to sleep, but not fast enough to not busy looping when receiving, especially when the CPU is loaded. That being said, the goal of this application was to prove that our framework can handle a high volume of messages.

4.5.1 IPC methods evaluation

Early in the project, we decided to use shared memory for inter-process communication. In this section, we compare shared memory with `nanomsg` [1], an open-source library, and Unix domain sockets, a data communication link to exchange data between processes. The original assumption was that the shared memory would be faster and more versatile than other solutions. Of course, working with shared memory is not easy, but that is not a criterion for the system library.

We built a simple application where we have multiple clients each concurrently updating a single shared variable. For the shared memory, the variable is in the shared segment and we implemented an n-producer, one-consumer application to simulate a client-server architecture. Both `nanomsg` and domain sockets work using message passing. For each, we built a server that is responsible to store the shared variable, and clients that will request a value change by message. We calculated the total time it took for all clients to finish updating the value multiple times. Tests were carried out on the `m4.xlarge` virtual machine instance type.

In Figure 4.9, we measure how each technology behaves with an increasing number of parallel threads that concurrently update the shared variable. As expected, shared memory scales better than Nanomsg and Unix domain socket. The major difference

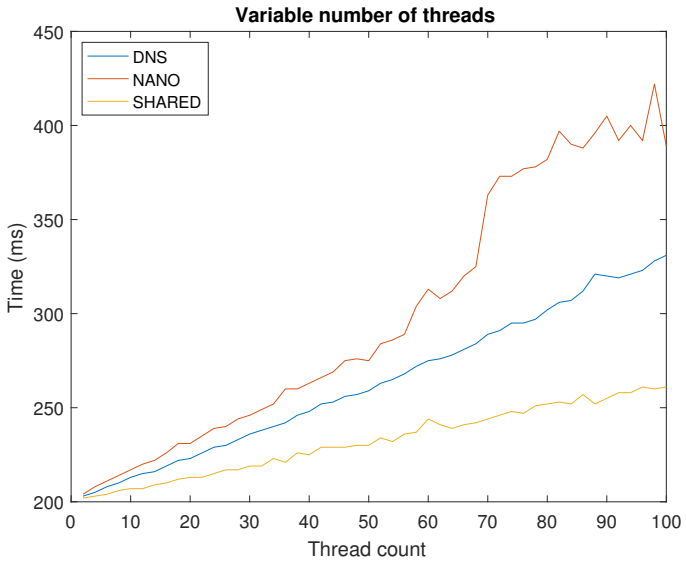

Figure 4.9: Variable number of threads.

is that domain sockets and `nanomsg` require message passing for each simple update, and therefore suffer from the overhead for packaging, sending and receiving messages.

In Figure 4.10, we measured how well the technology scales with an increasing number of updates and a fixed set of four clients. This time, shared memory is the least effective solution. The main bottle-neck here is likely in the producer-consumer implementation that uses two `pthread_mutexes`, causing a slowdown by locking and unlocking constantly.

We note that our framework prioritizes a large number of threads over speed of message sending. That being said, our choice to use shared memory remains appropriate. Versatility is one of our main aims, and we will be able to use shared memory for both inter-thread and inter-process communication. By contrast, `Nanomsg` is a library that offers complex communication patterns between threads, processes, and networks. Unfortunately, the framework adds a significant overhead and requires

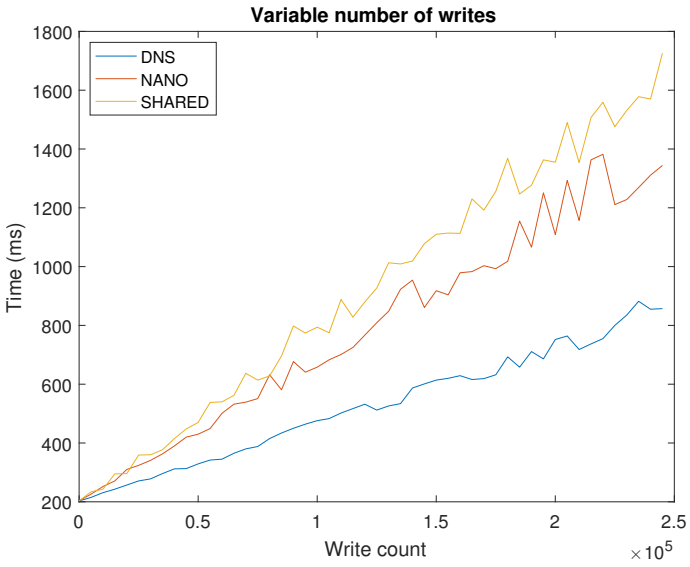

Figure 4.10: Variable number of writes.

more configuration.

4.6 Weather system

The weather station was the third and final application developed in order to evaluate the framework. The main goal of this application was to showcase the use of the task repository. We also wanted to evaluate the expected work of the developer when designing a more complex distributed application. The weather application consists of weather services and clients that can subscribe to receive weather broadcasts. The application is represented in Figure 4.11.

At start-up, the first task is required to create the system, while subsequent services only need to acquire it. Implementing an automatic switch for acquiring or creating the system would be feasible using a lock provided by the operating system.

At the start of each weather service task, each task records their name in the

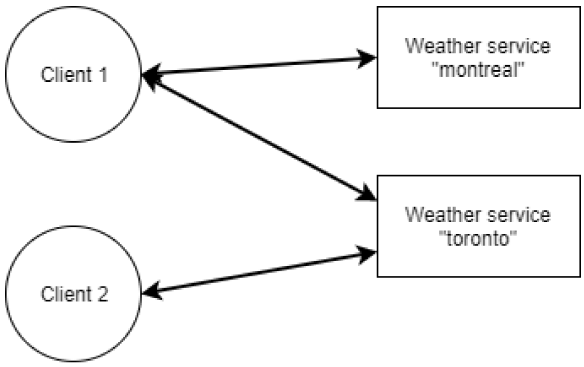

Figure 4.11: Weather application.

repository. This action also represents the moment when the service can begin receiving a new subscription. Using the weather station name, each client will subscribe to a given weather service to receive broadcasts. We used the repository to get the task identifier of the service and to detect service availability, similar to the example in Figure 3.11.

Figure 4.13 shows how the service was implemented. We used the `message_immediate` strategy to detect new subscribers. The main loop will accept new subscribers and then broadcast to all clients.

One advantage of using shared memory for the message queues is that if one client crashes, the queue is still available and not corrupted, and if the queue is full, we simply ignore that client in the broadcast. The service could automatically remove it from its subscriber list. If a station crashes, then a client would simply stop receiving broadcasts.

We emphasize that this whole service can be implemented in less than 50 lines of code. Listing 4.1 shows the core code function of our weather service task. As a reminder, our application could use any C library to gather weather data.

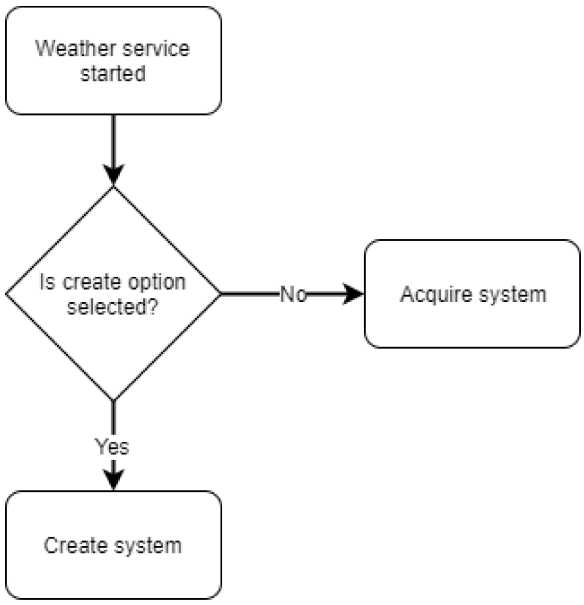

Figure 4.12: Weather service starting option.

```
1  /*
2   * This is the "main" method for the weather service task
3   */
4  static void start(WeatherStationTask this){
5    printf("Weather␣station␣task␣has␣been␣started\n");
6
7    // Receive station name and set repository
8    receive(this);
9    repository_set_name(this, this->station_name);
10
11   // Main loop
12   while (true) {
13     // Look for new subscriber
14     if (message_immediate(this) == 0) {
15       receive(this);
16     }
17
18     // Broadcast weather to all subscriber
19     IntArrayMsg temp_msg = IntArrayMsg_create(POST_WEATHER_MSG);
20     int vals[2] = {this->taskID, 31};
21     temp_msg->setValues(temp_msg, 2, vals); // Mock temperature
22     for (int i = 0; i < this->sub_count; i++) {
23       send(this, (Message)temp_msg, this->subs[i]);
24     }
25     temp_msg->destroy(temp_msg);
26   }
27 }
```

Listing 4.1: Weather service task implementation.

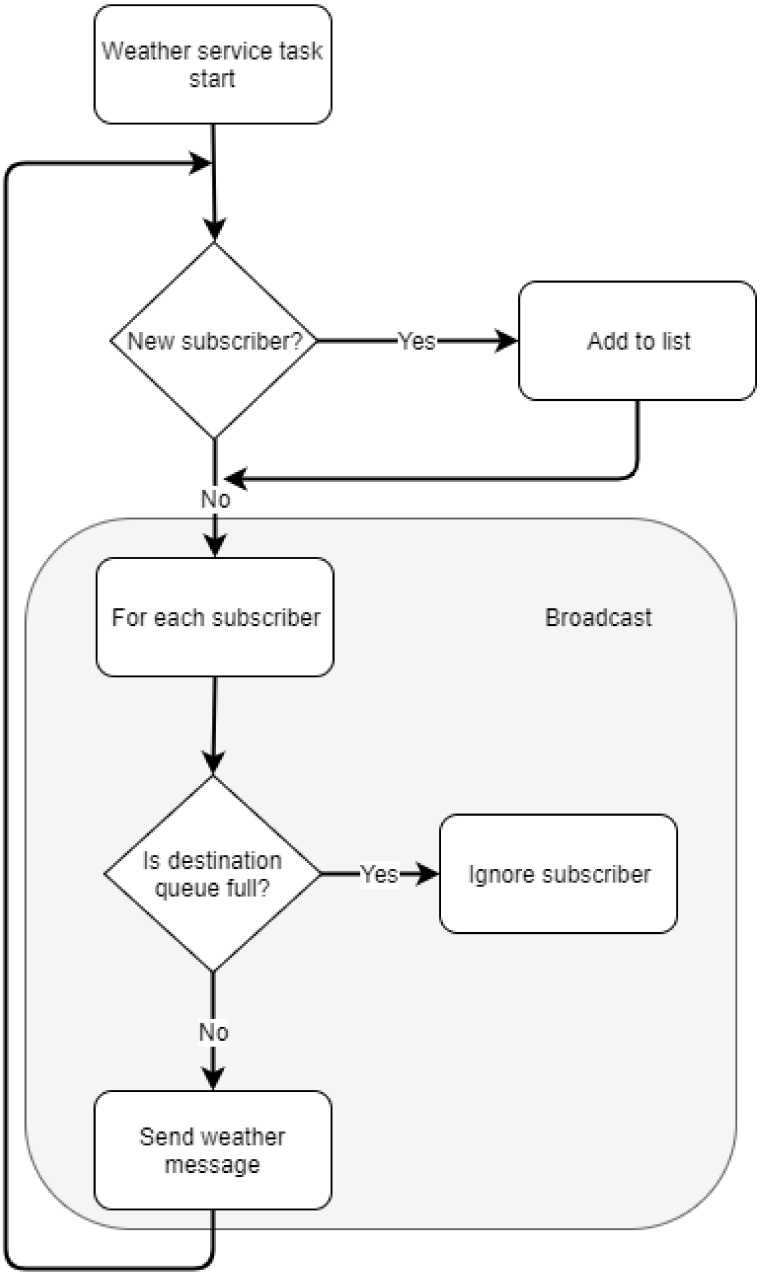

Figure 4.13: Weather service task flow.

While the main goal of this application was to showcase the task repository, we found that developing a more complex distributed software was quite easy and required minimal coding. However, we identified two points that would require improvement, especially in a distributed context:

- Acquiring or creating the system should be automatic
- Implement a clean shutdown procedure

4.7 Conclusions

This chapter was dedicated to the evaluation of the task system. We developed three applications to evaluate different aspects of the framework. The first was a samplesort implementation to display parallel execution capabilities. We have shown that the framework is capable of providing good results for a parallel algorithm implementation. The second application was the client-server database. The goal of this application was to measure maximum message throughput and the effect of message strategies. It was also the first application to use inter-process communication between tasks. The last application was the weather station, showcasing repository functions. The weather application was a more complex distributed application that promoted a discussion about important design choices regarding system creation and task failure.

We believe these three applications were able to provide a good overview of the performance and capabilities one can expect from the framework, once the current implementation is integrated with the the compiler. In short, the framework provides appropriate tools to develop concurrent, parallel or distributed applications with minimal code and configuration. It can handle message throughput in the thousands, as

well as parallel execution, abstract inter-process and inter-thread communication.

Chapter 5

Conclusions

5.1 Summary

With the growing need for concurrency, new technologies are emerging to solve complex challenges such as core scalability, distributed architectures and massive data sizes. These systems usually rely on strong distributed or parallel libraries to acheive their goals. Numerous technologies currently exist to build these systems; however, they are largely aimed at only one type of architecture and are typically complex to use. Programming languages usually offer a basic set of features to create threads and synchronize memory access. Some languages, such as Erlang, go further and are built over the actor model with built-in message passing. Unfortunately, small and unique languages do not have access to large language ecosystems and libraries.

In this thesis, we proposed a task-based message passing framework written in C. The framework aims to be easy to use and to provide an efficient way to create parallel or distributed applications over an actor model. We explored the use of generated code, allowing the framework to exploit the large C ecosystem, as well as exposing an object-oriented programming style. We also used shared memory to abstract communication between threads and processes.

Using our framework, a developer should be able to implement a wide variety of parallel or distributed application simply by using tasks and message passing. We evaluated our framework to make sure we provided good parallel performance and high message throughput. We also covered design choices that the developer should make in order to create a robust application.

5.2 Future Work

Even thought the current implementation offers enough functionality to build a real application, more work must be done to improve various parts of the framework. Possible future work includes:

- *Green threads.* The current implementation is base on the `pthread` library that provides access to kernel threads that can sometime be too heavy for our needs. Similar to Erlang, we could implement a green thread model for massive concurrency with optimized scheduling and memory management that might sometimes be more appropriate than full kernel threads that rely on operating system scheduling.

- *Fully distributed system.* The current messaging implementation is limited to a single host. Network distribution would be a real challenge. It could be based on the Erlang model where intra-node communication is done using node naming and global network configuration. This solution is not automatic, however and highly reliant on the developer. A transparent network-capable distributed system would be an even bigger challenge for performance and usability.

- *Lock-free message queue.* The current message queue is the main bottleneck in

terms of maximum message throughput. It uses classic synchronization primitives and will suffer from locking contention with a high number of threads. Implementing a lock-free queue would be a big step in improving message throughput and faster inter-process communication. Back in Chapter 2, we introduced a fetch-and-add based lock-free queue that could be implemented in our framework [66].

- *Improved central system.* The current shared memory approach is limited in terms of space. It is not possible to have more than one system at a time and each shared segment has a fixed size. With multiple systems or variable shared segment sizes, it would be easier to work with a massive number of tasks. Reworking the system to improve how it manages tasks would improve IPC and scalability.

Bibliography

[1] About nanomsg. http://nanomsg.org/index.html. (Accessed on 01/25/2018).

[2] Akka — akka. https://akka.io/. (Accessed on 01/07/2018).

[3] Amazon ec2 instance types amazon web services (aws). https://aws.amazon.com/ec2/instance-types/. (Accessed on 11/04/2017).

[4] Apache spark - lightning-fast cluster computing. https://spark.apache.org/. (Accessed on 10/23/2017).

[5] Charm++: Applications. http://charmplusplus.org/applications/. (Accessed on 06/20/2018).

[6] Charm++: Capabilities. http://charmplusplus.org/capabilities/. (Accessed on 06/20/2018).

[7] Concurrency - the rust programming language. https://doc.rust-lang.org/book/first-edition/concurrency.html. (Accessed on 01/27/2018).

[8] The cray-2 computer system, 1985. http://s3data.computerhistory.org/brochures/cray.cray2.1985.102646185.pdf. (Accessed on 06/02/2018).

[9] Cuda semantics pytorch master documentation. http://pytorch.org/docs/master/notes/cuda.html. (Accessed on 01/27/2018).

[10] Cuda zone — nvidia developer. https://developer.nvidia.com/cuda-zone. (Accessed on 10/25/2017).

[11] Erlang – concurrent programming. http://erlang.org/doc/getting_started/conc_prog.html. (Accessed on 10/23/2017).

[12] Erlang message passing example. https://www.erlang.org/course/concurrent-programming. (Accessed on 10/23/2017).

[13] Erlang programming language. https://www.erlang.org/. (Accessed on 01/13/2018).

[14] Google cloud computing, hosting services & apis — google cloud platform. https://cloud.google.com/. (Accessed on 10/23/2017).

[15] Gridmpi. http://aist-itri.github.io/gridmpi/. (Accessed on 10/23/2017).

[16] java.util.concurrent (java platform se 7). https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html. (Accessed on 01/07/2018).

[17] Mpi 4.0. http://mpi-forum.org/mpi-40/. (Accessed on 10/23/2017).

[18] Mpi forum. http://mpi-forum.org/. (Accessed on 10/23/2017).

[19] Mpich-madeleine. https://runtime.bordeaux.inria.fr/mpi/. (Accessed on 10/23/2017).

[20] Mvapich. http://mvapich.cse.ohio-state.edu/. (Accessed on 10/23/2017).

[21] Open mpi: Open source high performance computing. https://www.open-mpi.org/. (Accessed on 10/23/2017).

[22] Openmp. http://www.openmp.org/. (Accessed on 10/25/2017).

[23] The rust programming language. https://www.rust-lang.org/en-US/. (Accessed on 10/27/2017).

[24] Task parallel library (tpl) — microsoft docs. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl. (Accessed on 05/12/2018).

[25] Tiobe index — tiobe - the software quality company. https://www.tiobe.com/tiobe-index/. (Accessed on 10/23/2017).

[26] Using gpus — tensorflow. https://www.tensorflow.org/programmers_guide/using_gpu. (Accessed on 01/27/2018).

[27] Using the gpu theano 1.0.0 documentation. http://deeplearning.net/software/theano/tutorial/using_gpu.html. (Accessed on 01/27/2018).

[28] Welcome to apache hadoop! http://hadoop.apache.org/. (Accessed on 10/23/2017).

[29] Windows version history. https://support.microsoft.com/en-us/help/32905/windows-version-history. (Accessed on 10/27/2017).

[30] Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.

[31] Jonathan Aldrich, Emin Gün Sirer, Craig Chambers, and Susan J Eggers. Comprehensive synchronization elimination for java. *Science of Computer Programming*, 47(2-3):91–120, 2003.

[32] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. Concurrent programming in erlang. 1993.

[33] Jessie Berlin, Gabriel Tanase, Mauro Bianco, Lawrence Rauchwerger, and Nancy M Amato. Sample sort using the standard template adaptive parallel

library. Technical report, Technical Report TR07-002, Parasol Lab, Dept of Computer Science, Texas A&M University, College Station, USA, 2007.

[34] Nanette J Boden, Danny Cohen, Robert E Felderman, Alan E. Kulawik, Charles L Seitz, Jakov N Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE micro*, 15(1):29–36, 1995.

[35] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):40:46–40:58, September 2008.

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

[37] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan 1998.

[38] Message Passing Interface Forum. A message-passing interface standard, version 3.1. 2015.

[39] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*, pages 97–104. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[40] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. *Open MPI: A Flexible High Performance MPI*, pages 228–239. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[41] Vincent Gramoli, Rachid Guerraoui, and Vasileios Trigonakis. Tm2c: a software transactional memory for many-cores. *Distributed Computing*, Aug 2017.

[42] Ludovic Hablot, Olivier Glück, Jean-Christophe Mignot, and Pascale Vicat-Blanc Primet. Etude d'implémentations mpi pour une grille de calcul. *RenPar 2008*, 2008.

[43] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2Nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[44] Jim Held, Jerry Bautista, and Sean Koehl. From a few cores to many: A tera-scale computing research overview. *white paper, Intel*, 2006.

[45] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[46] Carl Hewitt. Actor model of computation: scalable robust information systems. *arXiv preprint arXiv:1008.1459*, 2010.

[47] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[48] Christian Iwainsky, Sergei Shudler, Alexandru Calotoiu, Alexandre Strube, Michael Knobloch, Christian Bischof, and Felix Wolf. *How Many Threads will be too Many? On the Scalability of OpenMP Implementations*, pages 451–463. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[49] Laxmikant V Kale and Sanjeev Krishnan. Charm++: a portable concurrent object oriented system based on c++. In *ACM Sigplan Notices*, volume 28, pages 91–108. ACM, 1993.

[50] Kaivan Karimi and Gary Atkinson. What the internet of things (iot) needs to become a reality. *White Paper, FreeScale and ARM*, pages 1–16, 2013.

[51] Shashi Kumar, Axel Jantsch, J-P Soininen, Martti Forsell, Mikael Millberg, Johny Oberg, Kari Tiensyrja, and Ahmed Hemani. A network on chip architecture and design methodology. In *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*, pages 117–124. IEEE, 2002.

[52] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programm. *IEEE transactions on computers*, (9):690–691, 1979.

[53] Denise Lund, Carrie MacGillivray, Vernon Turner, and Mario Morales. Worldwide and regional internet of things (iot) 2014–2020 forecast: A virtuous circle of proven value and demand. *International Data Corporation (IDC), Tech. Rep*, 2014.

[54] Ewing Lusk and Anthony Chan. Early experiments with the openmp/mpi hybrid programming model. *Lecture Notes in Computer Science*, 5004:36, 2008.

[55] Robert HB Netzer and Barton P Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[56] L. Nyman and M. Laakso. Notes on the history of fork and join. *IEEE Annals of the History of Computing*, 38(3):84–87, July 2016.

[57] Gregory F Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[58] R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 427–436, Feb 2009.

[59] Jorge L. Reyes-Ortiz, Luca Oneto, and Davide Anguita. Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf. *Procedia Computer*

Science, 53(Supplement C):121 – 130, 2015. INNS Conference on Big Data 2015 Program San Francisco, CA, USA 8-10 August 2015.

[60] R. R. Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, June 1997.

[61] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating Systems Concepts*. Wiley, 2014.

[62] W Richard Stevens and Stephen A Rago. *Advanced programming in the UNIX environment*. Addison-Wesley, 2013.

[63] Nathan R Tallent, John M Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In *ACM Sigplan Notices*, volume 45, pages 269–280. ACM, 2010.

[64] Samira Tasharofi, Peter Dinges, and Ralph E Johnson. Why do scala developers mix the actor model with other concurrency models? In *European Conference on Object-Oriented Programming*, pages 302–326. Springer, 2013.

[65] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, Jan 2002.

[66] Chaoran Yang and John Mellor-Crummey. A wait-free queue as fast as fetch-and-add. *SIGPLAN Not.*, 51(8):16:1–16:13, February 2016.

Appendix A

Simple task sample

Code sample for the developed framework. This code sample represents all the work the developer has to do to create the system, implement a message type and create a task to use it. It do not include generated code.

main.c

```
1
2 /**
3  * Start a simple task main example
4  */
5 int main(int argc, char *argv[]) {
6         // Create the system
7         Comm = System_create();
8         printf("System␣addr␣%p\n", Comm);
9         printf("System␣shared␣data␣addr␣%p\n", Comm->data);
10
11        unsigned int simple_task = SimpleTask_create();
12        printf("Simple␣task␣id␣=␣%d\n", simple_task);
13
14        return EXIT_SUCCESS;
15 }
```

Listing A.1: Main for simple task sample

SimpleTask.c

```
1
2 #include "TaskSystem/Tasks/SimpleTask/SimpleTask.h"
3
4 #include "TaskSystem/System.h"
```

```
 5 #include "TaskSystem/fatal.h"
 6
 7 #include <stdio.h>
 8 #include <stdlib.h>
 9 #include <string.h>
10 #include <unistd.h>
11
12 // this file contains code that the language compiler/runtime
13 // would generated automatically
14 #include "TaskSystem/Tasks/SimpleTask/generated.h"
15
16 /*****************************
17  * Programmer Code
18  *****************************/
19
20 enum {A_MESSAGE_TAG};
21
22 /*
23  * This is the "main" method for the thread
24  */
25 static void start(SimpleTask this){
26
27         int important_int = 1000;
28
29         // Repository usage
30         unsigned int destination_id = repository_get_id(this,
31                 "destination_task_name");
32
33         // Send a int message to a destination
34         IntMsg int_to_send = IntMsg_create(A_MESSAGE_TAG);
35         int_to_send->setValue(int_to_send, important_int);
36         send(this, (Message)int_to_send, destination_id);
37         int_to_send->destroy(int_to_send);
38
39         // Message strategy : yield cpu
40         message_wait(this);
41
42         // Receive a message
43         receive(this);
44 }
45
46 static void receive(SimpleTask this){
47         int tag = Comm->getMsgTag(Comm, this->taskID);
```

```
48          while (tag < 0) {
49                  tag = Comm->getMsgTag(Comm, this->taskID);
50          }
51
52          Message msg;
53
54          // match the message to the right message "handler"
55          switch (tag) {
56          case A_MESSAGE_TAG:
57                  msg = Comm->receive(Comm, this->taskID);
58                  handle_AMessageTag(this, (IntMsg)msg);
59                  break;
60          default:
61                  printf("\nTask␣%d␣No␣Handler␣for␣tag␣=␣%d,
62 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣dropping␣message!␣\n", this->taskID, tag);
63                  Comm->dropMsg(Comm, this->taskID);
64          }
65 }
66
67 static void handle_AMessageTag(SimpleTask this, IntMsg intMsg) {
68          printf("Task␣received␣a␣message␣:␣%d\n", intMsg->value);
69 }
```

Listing A.2: SimpleTask implementation

IntMesage.c

```
1  #include "TaskSystem/Messages/IntMsg/IntMsg.h"
2  #include "TaskSystem/fatal.h"
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7
8  // this file contains code that the language compiler/runtime
9  // would generated automatically
10 #include "TaskSystem/Messages/IntMsg/generated.h"
11
12 /*****************************
13  * Programmer Code
14  *****************************/
15
16
17 static int getTag(IntMesage this){
```

```
18          return this->tag;
19  }
20
21  static BarMsg clone(IntMesage this){
22          IntMesage tmp = IntMesage_create(0);
23          tmp->tag = this->tag;
24          tmp->tid = this->tid;
25          tmp->msg_size = this->msg_size;
26
27          tmp->value = this->value;
28
29          return tmp;
30  }
31
32  static void destroy(IntMesage this){
33          free(this);
34  }
35
36  static int writeAt(IntMesage this, void* addr) {
37          IntMesage tmp = (IntMesage)addr;
38          tmp->tag = this->tag;
39          tmp->tid = this->tid;
40          tmp->msg_size = this->msg_size;
41
42          tmp->value = this->value;
43
44          return this->msg_size;
45  }
46
47
48  static int getValue(IntMesage this){
49          return this->value;
50  }
51
52
53  static void setValue(IntMesage this, int value){
54          this->value = value;
55  }
```

Listing A.3: IntMessage implementation

Appendix B

Sample-sort result tables for variable array size experiment

Sample-sort result tables for variable array size. Each line represents the averaged results of the test at N values. This experiment was done with 4 buckets on a 4-core virtual machine. Values of 0 mean the time it took to complete the task is virtually instantaneous.

N	File read time	Spawning buckets	Generate splitters	Wait time
50000000	6.717s	0s	0s	4.574s
100000000	13.4168s	0s	0s	9.1378s
500000000	67.4654s	0s	0s	47.7774s

Table B.1: Sample-sort task result table.

N	Get sample	Get splitters	Propagation	Rebuild	Sort	Value count
50000000	0s	0s	0.717s	0.1048s	3.5384s	12638275
100000000	0s	0s	1.4268s	0.2132s	7.442s	26187202
500000000	0s	0s	7.2124s	1.0342s	38.016s	126963778

Table B.2: Bucket 1 task result table.

N	Get sample	Get splitters	Propagation	Rebuild	Sort	Value count
50000000	0s	0s	0.7184s	0.1092s	3.6948s	13369761
100000000	0s	0s	1.4284s	0.2s	7.0344s	24326091
500000000	0s	0s	7.3478s	1.0082s	37.273s	123614756

Table B.3: Bucket 2 task result table.

N	Get sample	Get splitters	Propagation	Rebuild	Sort	Value count
50000000	0s	0s	0.7178s	0.0998s	3.326s	11633328
100000000	0s	0s	1.4408s	0.2066s	7.2216s	25008899
500000000	0s	0s	7.2088s	1.0112s	37.5374s	123275160

Table B.4: Bucket 3 task result table.

N	Get sample	Get splitters	Propagation	Rebuild	Sort	Value count
50000000	0s	0s	0.7182s	0.1016s	3.4884s	12358634
100000000	0s	0s	1.4358s	0.1984s	7.0392s	24477806
500000000	0s	0s	7.128s	1.044s	38.1188s	126146304

Table B.5: Bucket 4 task result table.