UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

C++ library for parallel programming on a distributed system

Author:
Yulianna TYMCHENKO

Supervisor:
Oleg FARENYUK

A thesis submitted in fulfillment of the requirements
for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

APPLIED
SCIENCES
FACULTY

Lviv 2021

Declaration of Authorship

I, Yulianna TYMCHENKO, declare that this thesis titled, "C++ library for parallel programming on a distributed system" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"To the stars who listen— and the dreams that are answered."

Sarah J. Maas

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

C++ library for parallel programming on a distributed system

by Yulianna TYMCHENKO

Abstract

The goal of this work was to develop an efficient and convenient API for distributed parallel computing and implement it as a library. The library provides an interface to describe parallel computation on distributed systems in terms of tasks and their relations. A high level of task description allows to manage and maintain task execution on a cluster efficiently. The library handles distributed task dispatching, scheduling, basic I/O functionalities, and node communication. As a result, the programmer doesn't have to worry about cluster management and can focus on algorithms. The idea of the library was inspired by the Intel Thread building blocks library for the shared memory systems.

Acknowledgements

Firstly, I want to express my gratitude to my supervisor and professor Oleg Farenyuk for his guidance and encouragement during this work. I am particularly grateful to him for sharing his knowledge and for his constant readiness to help.

I would also like to thank the Applied Science Faculty of the Ukrainian Catholic University for giving me the opportunity to become a part of their community, and for the atmosphere they created for us. I am grateful to our professors who shared their passion for science and taught us how to dream big.

My sincere gratitude to my parents, for their unconditional love and faith in me, and to my sister, who always stands by my side, even though we are not alike.

I am thankful for the moments, that brought my friends, whom I deeply love and appreciate, into my life.

Last but not least, I want to thank my cousin who inspired me to try programming 9 years ago.

Contents

Declaration of Authorship i

Abstract iii

Acknowledgements iv

1 Introduction 1

2 Background Knowledge 2
2.1 Parallel programming . . . 2
2.2 Distributed Systems . . . 2
2.2.1 Definition . . . 2
2.2.2 Distribution Transparency . . . 2
2.2.3 Scalability . . . 2
2.2.4 Types of Distributed Systems . . . 3
Cloud computing . . . 3
Cluster computing . . . 3
Grid computing . . . 3
2.2.5 Distributed System Architecture . . . 3
Centralized organizations . . . 3
Client-Server Architecture . . . 3
Multi-tiered Architecture . . . 3
Three-tiered Architecture . . . 3
2.2.6 Decentralized organizations . . . 3
Peer-to-Peer Architecture . . . 3
Advantages of Distributed Computing . . . 4

3 Related Works 5
3.1 Thread Building Blocks . . . 5
3.1.1 Concept Overview . . . 5
3.1.2 Generic parallel algorithms . . . 5
3.2 Apache Spark and Hadoop MapReduce . . . 5
3.2.1 Concept Overview . . . 5
3.2.2 Hadoop MapReduce . . . 5
Algorithm . . . 5
Strengths and Weaknesses . . . 6
3.2.3 Spark . . . 6
Architecture . . . 6
Spark Driver . . . 6
Spark Executor . . . 6
Spark Cluster Manager . . . 6
Stages . . . 7
Strengths and Weaknesses . . . 7

4 Developed Solution 8
4.1 Solution Architecture . . . 8
4.1.1 parallel_for and parallel_reduce . . . 8
Syntax overview . . . 8
Workflow . . . 9
4.1.2 parallel_pipeline . . . 9
Syntax overview . . . 9
Filter components . . . 9
Workflow . . . 10
4.2 RabbitMQ . . . 11
4.2.1 Overview . . . 11
4.3 Solution Architecture . . . 11
Reliability . . . 11
Routing and Dispatch . . . 12
4.3.1 AMQP-CPP . . . 12
4.3.2 Parallel processing challenges . . . 12
Connections and Channels . . . 12
4.4 Generic parallel algorithms . . . 13
Templates . . . 13
decltype . . . 13
type_traits . . . 14
4.4.1 Serialization . . . 14
RPC . . . 14
4.4.2 parallel_papeline . . . 15
Filter creation . . . 15
Chaining . . . 15
Data transferring . . . 15

5 Benchmarks 17

6 Conclusion & Future Works 19
6.1 Conclusion . . . 19
6.2 Future Works . . . 19

A Implementation 20
A.1 Code . . . 20

Bibliography 21

List of Figures

4.1 Pipeline Workflow 10
4.2 RabbitMQ Components 11
4.3 RabbitMQ Direct Exchange 12
4.4 Remote Procedure Call client-server example 14
4.5 Simplified Diagram of Filter Chaining 16

5.1 The relations between time, computational power and grain size. 1-4 – experiment number, time – in seconds, a – grain size 1000, b – grain size 100. 17
5.2 The consumption of the messages in RabbitMQ 18

Listings

4.1 parallel_for and parallel_reduce syntax 8
4.2 example class for parallel_reduce body parameter 8
4.3 parallel_pipeline syntax 9
4.4 input filter syntax 10
4.5 Example of user specialization 13
4.6 make_filter definition 15
4.7 operator&() definition 15

List of Abbreviations

AMQP	**A**dvanced **M**essage **Q**ueuing **P**rotocol
DAG	**D**irected **A**cyclic **G**raph
I/O	**I**nput **O**utput
IP	**I**nternet **P**rotocol
RAII	**R**esource **A**cquisition **I**s **I**nitialization
RDD	**R**esilient **D**istributed **D**atasets
RPC	**R**emote **P**rocedure **C**all
SMP	**S**ymmetric **M**ultiprocessor **P**arallelr
TBB	**T**hread **B**uilding **B**locks
TCP	**T**ransmission **C**ontrol **P**rotocol
TTL	**T**ime **T**o **L**ive

Chapter 1

Introduction

Since the development of the first computer the increase in the demand for computational power has brought enormous enhancement in the way computers are built and the techniques how to program them.

Over the years the engineers and developers have made several big breakthroughs, and at first high-performance computers switched from single central processors to mane (parallel) processor ones. Then multiprocessing invaded personal computers and later – mobile devices.

Parallel computers can be divided into the following categories [1]:

- Symmetric Multiprocessor Parallel (SMP) Computers – multiple processing units sharing a memory, connected to the same I/O bus and running the same Operating system (OS).
- Multicore Parallel Computers – have a processing unit with multiple cores and shared memory. Are kind of an SMP.
- Distributed Parallel Computers - consists of multiple processing units with their own memory each.

Another classification is based on the processor-memory design architecture. There are Shared Memory Architecture and Distributed Memory Architecture [1].

The advancement and the variety of designs have led to the evolution of programming concepts. Programming of parallel computers has always been a problematic, since historically computer programs were written sequentially and humans have difficulties thinking using current (low-level) parallel programming concepts. Whereas, to achieve a speedup from the parallel computer, the program has to be written in a way to exploit this parallelism. That is where the parallel programming languages arrive. In order to exploit the parallelism for the existing sequential programming languages, were implemented multiple libraries.

This thesis focuses on the creation of the C++ library, which can be used to facilitate the development of the distributed system, a precisely parallel system with distributed memory. The idea of the library was inspired by the Intel Thread building block for the shared memory systems[2].

Chapter 2

Background Knowledge

2.1 Parallel programming

Distributed computing systems are typically deployed for high-performance applications often originating from the field of parallel computing [3].

2.2 Distributed Systems

2.2.1 Definition

"A distributed system is a collection of autonomous computing elements that appears to its user as a single coherent system" [3].

This definition points out two characteristic features of distributes system. The first one, that this is a collection of computing elements, which can exist independently. These elements usually are referred to as nodes, can be a device or software. The second, that users believe they are working with a single coherent system. Thus, those nodes need to collaborate. While working with the distributed system we should take into account that the system contains multiple networked nodes, hence at any point of execution one of them can fail.

2.2.2 Distribution Transparency

One of the most important goals in building a distributed system is to hide the distribution itself. More precisely, if two nodes are separated by a large distance, they should stay hidden. This concept is called distribution transparency. The degree of distribution transparency has its trade-off, too. For example, many Internet applications try to hide the failures by repeatedly attempting to contact the server before finally giving up. As a result, these attempts can slow down the whole system [3].

2.2.3 Scalability

When it comes to the distributed architectures, one can easily associate them with scaling-out. There are three techniques applicable for better horizontal scaling [3]:

- hide communication latencies – use the asynchronous communication and reduce it as much as possible;
- partitioning and distribution – involves splitting components into smaller parts and distribute over the system;
- replication where it is beneficial.

2.2.4 Types of Distributed Systems

Distributed computing infrastructures can vary widely. However, there are three general categories: Cloud Computing, Grid Computing, and Cluster Computing.

Cloud computing

uses distributed computing to provide customers with highly scalable cost-effective infrastructures and platforms.

Cluster computing

It is a more general approach and refers to all ways in which homogeneous computers and their computing power can be combined in clusters.

Grid computing

It is similar to the cluster computing model but with more hybrid architecture. Grid computing system contains the resources from different organizations, which have to collaborate. Such a collaboration is realized in form of a virtual organization. This organization grants access to its computational resources and storage.

2.2.5 Distributed System Architecture

Centralized organizations

Client-Server Architecture

The processes are divided into two groups: server and client. A server is a process responsible for a specific service. And the client is in the process of requesting the service by sending a request and waiting for a response. Communication between the two is implemented through the network protocol, mostly TCP/IP connections[3].

Multi-tiered Architecture

Three-tiered Architecture

An expanded version of a client-server architecture with the distinction into three logical layers. Such an architecture provides a variety of possibilities to distribute an application across several machines. Many applications have the: user interface layer, processing layer, and data layer. Then if we have to distinguish only two kinds of machines, we have a lot of options[3].

2.2.6 Decentralized organizations

Peer-to-Peer Architecture

The peer-to-peer architecture organizes the distributed processing by placing the logically different components on the same machine. This is also known as horizontal distribution. The processes that constitute a peer-to-peer system are equal and each process can act as a client and a server at the same time[3].

Advantages of Distributed Computing

- Unlimited horizontal scaling – one can add a new computing element at any time
- Fault Tolerance – if one part of the system is down, others could still work properly
- Redundancy – because of the usage of many small machines, they do not need to be prohibitively expensive [4].

Chapter 3

Related Works

3.1 Thread Building Blocks

3.1.1 Concept Overview

Thread Building Blocks is a C++ template library for shared-memory parallel programming and heterogeneous computing. The working principle is to divide the task into small chunks and run them on the available CPU cores. Rather than breaking up a program into functional blocks and assigning a separate thread to each, TBB emphasizes data-parallel programming, enabling multiple threads to work on different parts of a collection. Parallel programming concepts with operating directly over threads may be tricky for designing a large and highly scalable system. Hence, the software creating some layer of abstraction between a developer and actual logical threads is a good solution [2].

3.1.2 Generic parallel algorithms

The term algorithm in the TBB context stands for a set of its features. The library provides a variety of generic patterns with a fork-join nature. Meaning that all they start from the one thread of execution. Then, when the thread encounters the parallel algorithm, it splits the workload between different threads. When all the work was finished the execution merges back together and continues on the initial thread [2].

3.2 Apache Spark and Hadoop MapReduce

3.2.1 Concept Overview

In contrast to TBB, Spark and Hadoop MapReduce are Big Data frameworks, designed to run on the distributed systems. That is also a convenient tool when it comes to Cloud Computing, which has become mainstream now.

3.2.2 Hadoop MapReduce

Algorithm

The computational model of MapReduce consists of a combination of such functions as Map, Shuffle, and Reduce. One important note, that the framework operates exclusively on <key, value> pairs, that is, both the input and the output of the job is the set of pairs, conceivably of different types. The master node inputs the incoming data splits it into chunks and transfers it to the worker nodes. These nodes apply the mapping functions on their local data and store the intermediate results in in-memory buffers that spill over to the local file system. Here comes to work Shuffle.

Generally, that is the process when the output from mappers gets sorted by key and is transferred to reducers. Sorting in Hadoop saves time for the reducer, as it can take all the pairs with one key per task and generate key-value pair as output [5].

Strengths and Weaknesses

MapReduce is a programming model and an associated implementation for processing and generating large data sets. It is highly available, scalable, and fault-tolerant [6]. However, there are a couple of drawbacks for Hadoop MapReduce, for example, low processing speed. The Reduce task input is an output of the Map task. Whereas there is a shuffling stage before the Reduce, the execution of the Reduce begins only after all the Map tasks were finished. In addition, MapReduce is a batch processing framework, as a result, the performance is lower as we need some time for data to be stored [5]. MapReduce is inefficient for multi-pass applications which require low latency.

3.2.3 Spark

Architecture

Apache Spark has a well-defined and layered architecture. Spark architecture is based on two main abstractions:

- RDD (Resilient Distributed Datasets) - a collection of data items that are split into partitions and can be stored in memory on worker nodes. The RDD's support two types of operations: transaction and action.
- DAG (Directed Acyclic Graph). Direct transformation is an action that transitions data partition state from A to B. Acyclic transformation can not return to the older partition.

In general, Spark follows a master/slave architecture with two daemons and a cluster manager.

Spark Driver

The Driver invokes the `main()` and creates the Spark Context. That is the place, where scheduling occurs. The Driver translates RDD's into the execution graph and splits them into stages and saves the metadata. The next step is to divide into tasks and execute them in workers.

Spark Executor

This distributed agent responsible for task execution. Each application has its executor, which performs the data processing, as well as reads and writes the data from external sources. The executor is also responsible for storing the computational results.

Spark Cluster Manager

- an external service responsible for acquiring resources on the Spark cluster and allocating them to a spark job.

Stages

As for the MapReduce, there are mainly two stages associated with the Spark: ShuffleMapStage and ResultStage.The ShuffleMapStage is the intermediate phase for the tasks, which prepares data for another stage. In the Spark context, there are two types of transformation: narrow and wide. The narrow one does not require shuffling, for example, `map()`, whereas wide need shuffling across various partitions. Spark jobs are executed in the pipeline where narrow transformations are combined into a single stage and wide requires the different stages to communicate across different partitions. Each stage is an input for the other following stages. The ResultStage is a final step to the spark function for the particular set of tasks in the spark job [7].

Strengths and Weaknesses

In the beginning, Apache Spark was intended to become the programming model which supports a much wider class of application than MapReduce, without loss of maintenance its automatic fault tolerance. This goal has succeeded. Apache Spark can be faster 100 times than MapReduce as a result of in-memory computing, and there is support for different applications, which are common in analytics [8].

Although, this framework still has some disadvantages. The Apache Spark does not support a file management system, as a result, additional integration should be done. In addition, if one will try to use Spark cluster with HDFS (Hadoop Distributed File System) they can come across a problem with small files. The last one is its cost because in-memory computation with lots of RAM usage is expensive [9].

Chapter 4

Developed Solution

4.1 Solution Architecture

When we started working on the solution, we considered different user interfaces to implement. The first idea, probably the easiest from the development view, was to wrap the data transferring between different nodes and create an MPI-based solution. But this approach was on the lower abstraction level than we wanted to achieve at the end. So, finally, we chose to build the TBB-like interface and scale it up to work on distributed systems.

Our current solution provides such execution patterns as: parallel_for, parallel_reduce, and parallel_pipeline. We will look through each of these in more detail.

4.1.1 parallel_for and parallel_reduce

Syntax overview

```
1 template<typename Body, typename Container>
2 void parallel_for(const Container &container,
3                          Body &body, RabbitClient &client);
4
5 template<typename Body, typename Container>
6 void parallel_reduce(const Container &container,
7                          Body &body, RabbitClient &client);
```

LISTING 4.1: parallel_for and parallel_reduce syntax

Before discussing the working principle, let us describe the usage requirements. Firstly, we need to introduce some library components used to operate with the data structures passed to `parallel_for` and `parallel_reduce`. The `blocked_container` and `blocked_range` manage the user-defined containers used in parallel patterns for splitting the given container before sending it to workers. The `blocked_range` is used internally to define the sub-ranges for each portion of data for the one task. This range can be recursively subdivided into two parts with the `split_()` method forming a tree-like structure. The division stops when the received parts represent portions of work equal to the grain size. Afterward, the container constructor invokes and builds the sub-containers.

```
1 class Foo {
2 public:
3     int value;
4
5     Foo();
6     void operator()(const Container &container);
7     void join(Foo &rhs);
8
9     template<class Archive>
```

```
10     void serialize(Archive &ar, const unsigned int version) {
11         ar & value;
12         };
13     friend class boost::serialization::access;
14 
15 };
```

LISTING 4.2: example class for parallel_reduce body parameter

The `Body& body` has several requirements to satisfy.

1. Default constructor;
2. `operator()` for accumulation of the container;
3. `join()` method to be used for the reduction
4. serialization method
5. serialization class as a friend to grand access;

Listing 4.1 illustrates that both patterns have the same signature, but the requirements are not. The `Body& body` for the `parallel_for` do not need to have `join()` for the obvious reason.

Workflow

Once the execution starts, the splitting stage begins and the results are sent to the RabbitMQ. At this step, the producer spreads the associated work across worker nodes and starts waiting for the RPC results. The producer's receiving stage consists of two steps: consuming the messages from the RabbitMQ and delivering them to the local concurrent queue. On the main node, we also create threads for joining the incoming responses in parallel with their consumption. When all pieces of work are done, the execution continues in the single initial thread.

4.1.2 parallel_pipeline

The power of pipelining has become indisputable since instruction-level parallelism was introduced. Pipelines and filters are still useful patterns for component-based data transformation. Pipelines allow to express a sequence of processing transformations on the data stream. In our solution, the filters operate on data passed through the local queues and RabbitMQ [10],

Syntax overview

```
1 void parallel_pipeline(const filter_t<void, void> &filter_chain,
2                              RabbitClient &client);
```

LISTING 4.3: parallel_pipeline syntax

Filter components

are the processing units of the pipeline. Filters are used in conjunction with lambda expressions. The role of this component is to define the type of execution and to create the proper queue between two filters. There are three types of filters:

- `serial_in_order` – uses `std::deque`
- `parallel` – uses `concurrent_queue`
- `serial_out_of_order` – uses `concurrent_queue`

The concurrent_queue is our thread-safe wrapper of std::deque and it is usually used while we want to retrieve the result messages from RabbitMQ. One more important fact about the filter types: the first and last filters have the input and output purpose accordingly. Filters have defined operator() which invokes the given lambda repeatedly until processes all the input data. We use different types of queues mentioned above to store intermediate results between filters. Subsequently, to gather input data in the initial filter we needed to implement one more "helper" – flow_control.

```
make_filter<void, int>(filter::serial_in_order,
                       [&](flow_control &fc) -> int { while(...){
                                   ...
                                   return some_int;
                                   }
                                   fc.stop();
                                   return 0;});
```

LISTING 4.4: input filter syntax

The filter re-invokes the passed anonymous function before reaches the fc.stop(). The output one just have no returning queue, so that its lambda is returning void.

The make_filter creates the filter object of the correct type of filter. The chaining of filters can be built with operator&.

Workflow

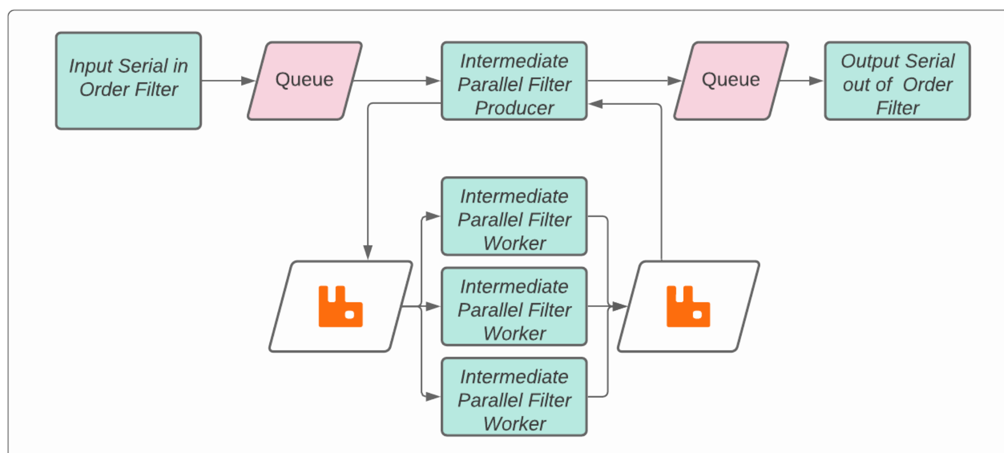


FIGURE 4.1: Pipeline Workflow

As described earlier, we need the input filter to receive the information and pass it on to the next filter in the chain. Only the parallel type of filter runs on worker nodes. The mechanism is the following when the execution encounters a parallel filter we call the overloaded operator(parallel) with the RPC implementation.

When we receive back the results, they are stored in the concurrent_queue and then the execution moves on to the next filter. The main advantage of this pipeline is that we can asynchronously deliver the RPC result and there is no need to wait for all jobs to return. However, the filter which is waiting for the data from concurrent_queue blocks when the queue is empty and in processing state.

4.2 RabbitMQ

4.2.1 Overview

RabbitMQ is a message-queuing software also known as a message broker. This is a powerful tool to use if one needs to implement message queuing. We decided to use RabbitMQ as a task broker since it provides a certain level of reliability. In addition, its clients are mostly implemented to perform asynchronously, which implies we can keep our workers busy until we have some other work to be handled. One more reason for choosing RabbitMQ is that it has a user-friendly interface and remove the need to implement additional layers for message passing through the network.

4.3 Solution Architecture

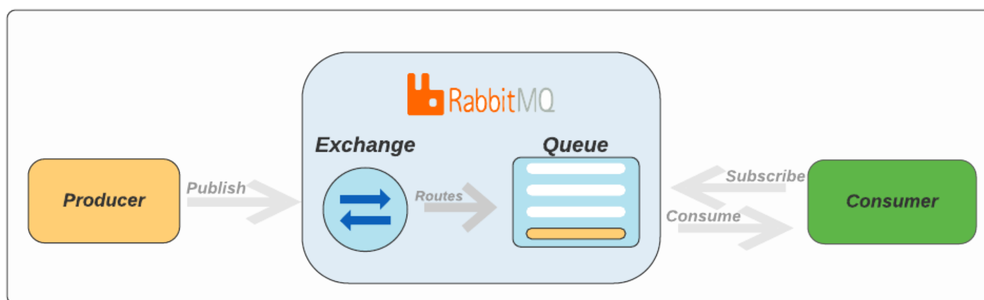


FIGURE 4.2: RabbitMQ Components

The basic architecture of a message queue is the following – some producers create messages and deliver them to the broker, and consumers, which connect to the queue and subscribe to the messages to be processed. The published message is first sent to the exchange, which is responsible for the routing messages to corresponding queues, depending on the binding and routing keys.

Reliability

RabbitMQ allows sending and receiving messages reliably. One can configure the TTL for messages per queue, as long as its persistence.

In condition, if one wants to make sure the messages survive the broker restart, the queue should be declared as durable and messages are sent with the persistent delivery mode.

The RabbitMQ persistence level has two components: the queue index and the message store. The index holds the information about the message location within the queue, along with the delivery information.

The message store is a key-value store for messages, shared among all queues in the server. There is an option, messages can be stored directly in the queue index, or written to the message store.

In any case, RabbitMQ provides the ability to acknowledge or reject the given message. Provided that, if the execution encounters some issues, the message can be redelivered back to the queue and won't be lost.

Routing and Dispatch

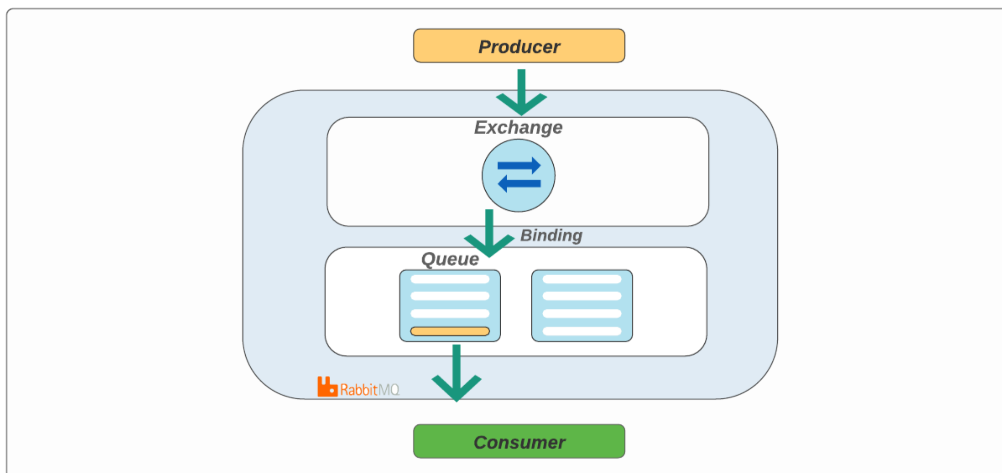


FIGURE 4.3: RabbitMQ Direct Exchange

The RabbitMQ has flexible routing. But for our implementation there was no need for additional complexity, so we used the direct exchange, hence a message goes to the queue with the binding key that exactly matches the routing key of the message. Binding is a connection between exchange and queue [11].

Last but not least, RabbitMQ, by default, has a round-robin dispatching. Meaning that each message will be sent to the next consumer in sequence so that they are distributed evenly. For our solution the fair dispatching is not an ideal solution, because if the execution of each task can take a long time, we can end up in a situation with two workers when there are both heavy and light messages, one worker can be constantly busy and the other one will do hardly any work. To avoid such kind of behavior we set the prefetch count to 1. This tells RabbitMQ not to give more than one message to a worker at a time [12].

4.3.1 AMQP-CPP

AMQP-CPP is a C++ library for communicating with a RabbitMQ message broker. This library can be used to parse and generate data frames required for work with the RabbitMQ server. The library is fully asynchronous and does not do any blocking (system) calls. For this project, we have used the Linux-only TCP module for the networking part. Providing that, the library needs to be built with the special option turned on [13].

4.3.2 Parallel processing challenges

When it comes to parallel processing, we need to be aware of some RabbitMQ specifics to create a high-performance application.

Connections and Channels

The AMQP protocol has a mechanism called channels that is a sort of virtual TCP connection. It is recommended that each process only creates one TCP connection, using multiple channels in that connection for different threads. Sharing channels

between threads would have a serious negative effect on the performance, as most clients don't make channels thread-safe [14].

For our solution we fork the worker process with a connections. In addition, the AMQP-CPP library can be integrated with different I/O processing event loops. We used the `boost::asio` handler, from the AMQP-CPP TCP module.

Since we have some blocking operations in our workflow, the event loop should run in a separate thread. This is because when we lock the execution thread, the event loop can't proceed with the handling of the asynchronous requests.

4.4 Generic parallel algorithms

Templates

When one considers using the library, it means that there will be support for a wide variety of types and structures as long as they match the algorithm. In C++ templates provide the main support for this kind of generic programming. Templates provide compile-time polymorphism [15].

The main advantage of template usage is that it handles both built-in and user-defined types. The only requirement is to match the usage of its arguments.

The single template defines every template argument that the user might want to substitute. Hence, one can also specify different behavior for certain template arguments by creating a specification. For example, use a different implementation for pointers. To achieve this, one should create the alternative definition, called user-defined specializations, after the base one [16].

```
1 template<typename T>
2 void foo(T &value);
3
4 template<>
5 void foo(void*);
```

LISTING 4.5: Example of user specialization

Since our implementation includes templates for various intentions, there is a need to introduce some notion of [17]:

- *"methods with template dependent signature"/"methods with template independent signature"* for methods that have template type arguments or return a template type value and vise versa;
- *"methods with template dependent implementation"/"methods with template independent implementation"* for methods that refer in the implementation to the template parameter type and vise versa;
- *"template dependent fields"/"template independent fields"* for fields referring to a template parameter in their declaration and vise versa;

decltype

The `decltype()` is an built-in type operator that returns the declared type of its argument. It is used for deducing the type of more complex than simple initializer, for example, the type of expression [15].

type_traits

In `<type_traits>` standard library provides type functions to determine properties and to generate the new types from the existing ones. This feature is usually used at the compile time with the metaprogramming [15].

Type functions can be used to check whether the type is the same as in the type predicate or check the specific properties, e. g. `is_const()`, `is_default_constructable()`.

There are also type generators, which produce the type given other type as an argument. For example `remove_cont<X>` produces type like `X`, but without `const`.

There are also different modifications, but for the sake of simplicity, we discussed only ones used in the solution implementation.

Using templates, type traits, Resource Acquisition Is Initialization (RAII) concept, and rvalue/lvalue references one can implement efficient and safe resource management as a part of the library.

4.4.1 Serialization

RPC

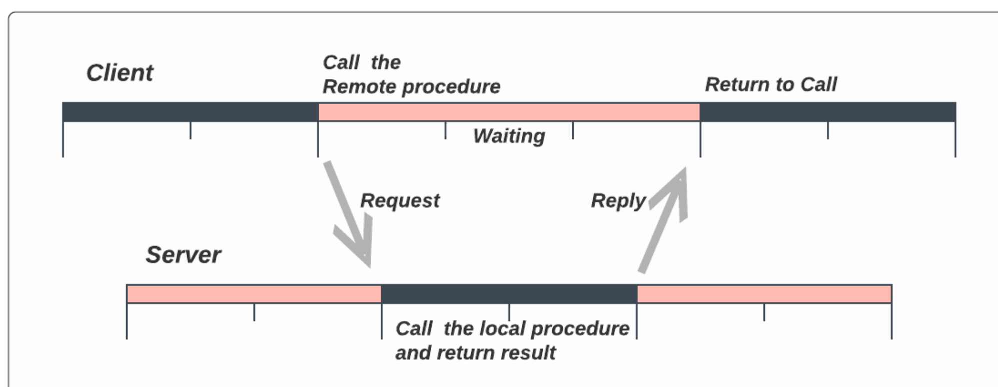


FIGURE 4.4: Remote Procedure Call client-server example

RPS stands for remote procedure call and is widely used in distributed computing. The idea behind the RPC is to make the remote procedure call look like local.

Since we are working with the distributed system, we can not rely on the shared memory and simply reference the parameter we want to pass. So that, we have to implement the parameter passing mechanism for our remotely executed function.

As mention before the `Container` is a template parameter for the template container class and the `Body` for `parallel_for` and `parallel_reduce` is a template parameter that should be deduced to the user defined type.

Therefore we need the serialization logic for the data we want to deliver via a message broker. The serialization of the object does not need additional transformation and type deduction. That means, we can create an envelope directly from the serialized archive and publish it to RabbitMQ. With the result retrieval, things become more complicated. The deserializer is a template function, this implies, we have to pass the type of object we want to create.

```
1 const AMQP::Message message = ...;
2 auto val = deserialize<remove_cvref_t<decltype(body)>>(message.body())
```

Listling 4.7 demonstrates the approach used for the deduction of the template parameter body. Although, from listling 4.1 we remember, body is passed as a constant reference, we use type function to remove constant reference.

4.4.2 parallel_papeline

```
template<typename I, typename O, typename Body>
   filter_wrapper<I, O>
   make_filter(filter::mode mode, const Body &body) {
                return new filter_node_leaf<T, U, Body>(mode, body);}
```

LISTING 4.6: make_filter definition

```
   template<typename T, typename V, typename U>
   filter_wrapper<T, U> operator&(const filter_wrapper<T, V> &left,
   const filter_wrapper<V, U> &right) {
       return new filter_node_join(*left.root, *right.root);}
```

LISTING 4.7: operator&() definition

Pipeline building turned to be a complicated task, due to the purpose of creating filters with template dependant parameters.

Filter creation

To create a filter the function `make_filter()` should be invoked. See the listling 4.6, parameter `mode` is the type of the filter and `body` – lambda function. The `make_filter()` function creates the the `filter_wrapper` object, later used for chaining. The `filter_node_leaf` is implicitly converted to `filter_wrapper` by non-explicit constructor.

Chaining

The filters are chained with the `operator&()`. The operator takes the left and the right filter and creates a join of the same type – `filter_wrapper`, but with reduced input and output types. For example, `filter_wrapper<void, std::string> & filter_wrapper<std::string, void>` turns into `filter_wrapper<void, void>`. When all the filters are added to the chain, we use the postorder traversal of a given tree and add the leaves to our pipeline.

The reason why we need so many proxy classes is that the pipeline doesn't know about templates and since we need the template dependant queues for input and output, we had to create some additional layers.

Data transferring

The figure 4.5 is intentionally simplified to illustrate the connections between classes. It demonstrates not only the relations between filter classes, but also shows the usage of `value_wrapper`, `queue_helper` and the `pointer_helper`. Initially, there should have been only one template class – `queue_helper` which pointer was passed forward to the next filter in the chain, cast to `void*` and typecasted back.

The problem occurred with asynchronous reply processing callback, the pointer was not valid by the time of callback execution. We intended to connect the message retrieval with the intermediate results queue to provide the next filter with the input data. As a result, we developed another solution. In current implementation the `queue_helper` is not a template class. Moreover, the values stored in queues are not template dependant. The `pointer_helper` and `value_wrapper` are template classes

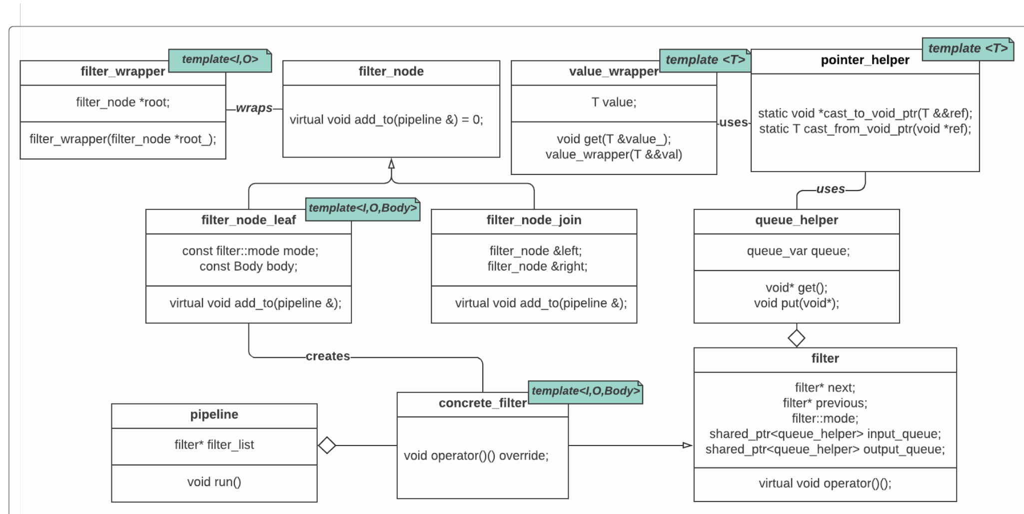


FIGURE 4.5: Simplified Diagram of Filter Chaining

with template dependant methods. The underlying logic is the following: every filter creates its shared pointer on the output queue, point the next filter input to this queue. The queue holds the value as the general-purpose pointer. Every time the `body` needs the input and produces the output `pointer_helper` casts it to the pointer on `value_wrapper` and back. The `value_wrapper` is nothing more than container for secure casting.

Chapter 5

Benchmarks

Firstly, let us set up and explain the testing environment. For the experiment we have chosen Cloud Based solution on Microsoft Azure. Were created 4 virtual machines to test the different configurations of the computational power and how it results on the performance.

The configuration of the machines used for testing:

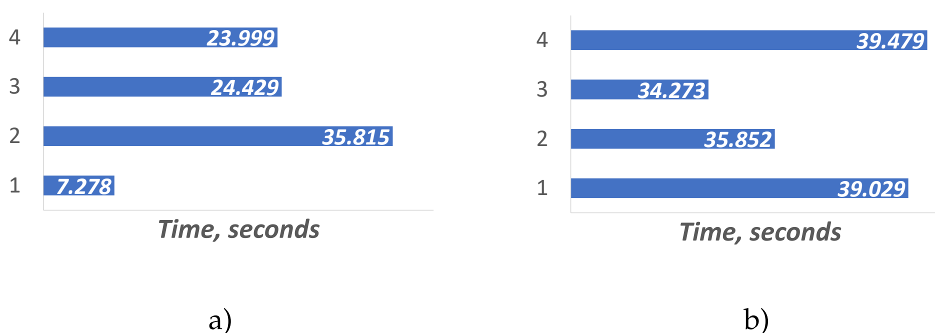
- OS – Ubuntu 20.04 LTC,
- RAM – two nodes with 8 Gb RAM and 2 with 16 Gb RAM,
- vCPU – 2 for each.

The compiler for this project is gcc (Ubuntu 9.3.0-17ubuntu1 20.04) 9.3.0.

The the task was executed with such configuration of the workers:

1. 1 worker on the same machine with producer,
2. 1 worker on separate machine,
3. 2 workers on separate machines and 1 on the same node with the producer,
4. 4 workers, two per separate node.

The task chosen for the testing stage is the classic – words counter. Let us proceed to the results.



a) b)

FIGURE 5.1: The relations between time, computational power and grain size. 1-4 – experiment number, time – in seconds, a – grain size 1000, b – grain size 100.

The key points of the performance testing:

1. To explore the dependency between workers.
2. To explore the RabbitMQ behaviour while high workload.
3. To explore the performance behaviour while workers are located on the same machine.
4. To explore the performance while workers are located on different machines.

Figure 5.1 illustrates that slightly larger grain size has overall better performance. Since, during the testing we noticed that the smaller grain size implies in the more

tasks posted to the RabbitMQ, therefore the rapid occurrences of the messages with no delay may impact its performance. On the other hand, in the situation where the grain size is near to the medium values we avoid the the network overload. Interestingly, there were lesser acceleration than was expected while the separate node was added. This means that the main delay is on the communication level. As a result, PCAM (Partitioning, Communication, Agglomeration, Mapping) methodology accent on reducing communication is still valid and important[18].

FIGURE 5.2: The consumption of the messages in RabbitMQ

The figure 4.5 demonstrates a high throughput in our application, because the messages in RabbitMQ get acknowledged quickly. This means, that both our workers and the main node don't block during the processing, which are the exactly expected behaviour.

Chapter 6

Conclusion & Future Works

6.1 Conclusion

As a result of this work, we have an implementation of the parallel patterns which can be used to write the programs for distributed systems. The provided interface can be useful for data processing, as well as for scientific computing. The developed solution has its restrictions. Since the implemented model represents fork-join architecture, the performance is limited by the availability of the main node, which needs to handle all the results. Nevertheless, our solution is suitable for smaller clusters.

6.2 Future Works

Since the development of a library is a considerable task, there are lots of possible improvements. The most important ones are about error handling and reliability, with the current approach, we assume that RabbitMQ handles the errors and redelivers the message, but there has to be more complex handling with the ability to reject the messages without significant loss. In addition to this, the application can not just ignore a loss, provided that there is a need to implement the mechanism of proper behavior in such a case. Another, yet important, improvement is to add optimizations to reduce the amount of data sent across the network and reduce the load from the main node.

Appendix A

Implementation

A.1 Code

The compiler for this project is gcc (Ubuntu 9.3.0-17ubuntu1 20.04) 9.3.0. The source code can be found here: https://github.com/neverlandjt/sheeppl

Bibliography

[1] Charles Saidu, Afolayan Obiniyi, and Peter Ogedebe. "Overview of Trends Leading to Parallel Computing and Parallel Programming". In: *British Journal of Mathematics Computer Science* 7 (Jan. 2015), pp. 40–57. DOI: 10.9734/BJMCS/2015/14743.

[2] *Advanced HPC Threading: Intel® oneAPI Threading Building Blocks*. URL: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onetbb.html#gs.156cim.

[3] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems*. Pearson Education, 2013. ISBN: 1292025522.

[4] *Distributed computing for efficient digital infrastructures*. URL: https://www.ionos.com/digitalguide/server/know-how/what-is-distributed-computing/.

[5] *Hadoop vs. Spark: A Head-To-Head Comparison*. 2020. URL: https://logz.io/blog/hadoop-vs-spark/.

[6] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.

[7] *Apache Spark Stage- Physical Unit Of Execution*. 2019. URL: https://techvidvan.com/tutorials/apache-spark-stage/.

[8] *Research: Apache Spark*. URL: https://spark.apache.org/research.html.

[9] *Limitations of Apache Spark - Ways to Overcome Spark Drawbacks*. 2018. URL: https://data-flair.training/blogs/limitations-of-apache-spark/.

[10] Jonathan Thaler. *System Architectures*. 2020. URL: https://homepages.fhv.at/thjo/lecturenotes/sysarch/pipes-and-filters.html.

[11] *RabbitMQ tutorial - Publish/Subscribe*. URL: https://www.rabbitmq.com/tutorials/tutorial-three-python.html.

[12] *Consumer Prefetch*. URL: https://www.rabbitmq.com/consumer-prefetch.html.

[13] CopernicaMarketingSoftware. *CopernicaMarketingSoftware/AMQP-CPP*. URL: https://github.com/CopernicaMarketingSoftware/AMQP-CPP.

[14] *Part 1: RabbitMQ Best Practices*. URL: https://www.cloudamqp.com/blog/part1-rabbitmq-best-practice.html.

[15] Bjarne Stroustrup. *The C++ Programming Language*. 4th. Addison-Wesley Professional, 2013. ISBN: 0321563840.

[16] David Vandevoorde, Nicolai M. Josuttis, and Douglas Gregor. *C++ Templates: The Complete Guide (2nd Edition)*. 2nd. Addison-Wesley Professional, 2017. ISBN: 0321714121.

[17] *Refactoring template bloat*. 2009. URL: https://eugenedruy.wordpress.com/2009/07/19/refactoring-template-bloat/.

[18] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201575949.