BBFlow: a Java implementation of FastFlow building blocks

Contents

Introduction

FastFlow is a general-purpose *C++* programming framework for heterogeneous parallel platforms. Like other high-level programming frameworks, such as Intel *TBB* and *OpenMP*, it simplifies the design and engineering of portable parallel applications [1]. *FastFlow*, developed across over one decade, implements the concept of building blocks. The building blocks are a set of primitive parallel components that could be used to model the parallel behavior of a wide range of computations across multiple application domains [2] [3].

Building blocks can be used in a *LEGO-style* approach, where bricks can be either complex pre-assembled and tested structures or elementary bricks. Each building block can be used as it is or combined with others forming a new type of building block.

Composition of these blocks, using communication channels, models all the parallelism exploitation aspects of the user computation.

This thesis aims at providing a *Java* based version of *FastFlow* building blocks and making a performance comparison between the two versions. Therefore, a project called *BBFlow,* reimplementing the main building blocks of the original project, was developed and tested.

BBFlow is a *Java* implementation of all sequential and parallel building blocks of *FastFlow*.

The *BBFlow* project is developed using only the classes included in *JDK* without any external library. All the implemented building blocks exploits the most common *Java* classes in order to try to keep the code as clear as possible.

BBFlow also implements all types of shared-memory communication channels present in *FastFlow* to interconnect the building blocks; in addition, we also implemented *TCP network channels* present in *Distributed FastFlow (D-FastFlow)* [4].

The ultimate goal of the *BBFlow* project is to compare the performance differences between the *C++ FastFlow* implementation and a *Java* one.

BBFlow does not pretend to reach the same degree of *efficiency* of *FastFlow*, for two main reasons: the programming language differences (memory management, *Java Virtual Machine*, etc.) and the fine-tuning job done on *FastFlow* over many years.

For this reason, a comparison, between the high-efficient *C++* implementation and the *Java* one using just default classes present in *JDK*, is very interesting.

The thesis is organized into six chapters, briefly described as follows:

- *Chapter 1* presents the background needed to understand the goal and the challenges of the thesis. It focuses on a brief description of *FastFlow* library that is used as the theoretical and practical base of our work. It also describes our new proposed implementation of *FastFlow* library in a different programming language.
- *Chapter 2* describes the proposed *BBFlow* project, its structure and its basic usage.
- *Chapter 3* describes in detail all of the implemented sequential and parallel building blocks, the implementation choices and the differences with the source *FastFlow* library.
- *Chapter 4* focuses on the description of the communication channels used to interconnect the parallel and sequential building blocks. It describes both shared-memory and network communication channels.
- *Chapter 5* focuses on the performance analysis of communication channels and building blocks. It also includes different performance comparisons between *FastFlow* and *BBFlow* components.
- *Chapter 6* presents a use case application developed exploiting the *BBFlow* library. It highlights advantages and drawbacks of our library and includes a performance analysis of the computation in different scenarios.

Finally, the *Conclusion* section summarizes the results obtained in this thesis and the experimental outcomes, providing some ideas of the possible future directions.

Attached to the thesis, there is the *Appendix A* containing the manual of the *BBFlow* project.

Chapter 1

Background

FastFlow is a parallel-programming library and is the result of a research work started in 2010 by a joint effort of the *Parallel Programming Model Group of the University of Pisa* and the *Parallel Computing Research Group of the University of Turin*. The library aims to provide key features for parallel programming to application designers. It abstracts a set of tools and patterns and provides a carefully designed *Run-Time System* (*RTS*). *FastFlow* comes as a *C++* header-only library and the parallel programmer can include it and exploits all available features. The language chosen for the library, *C++*, is dictated by the *efficiency* needs. *C++* is designed to be a compiled language, meaning that it is generally translated into machine language that can be understood directly by the system, making the generated program highly efficient. It is a versatile programming language that can be both low and high level. The low-level programming allows fine-tuning a project according to the machine and the operating system specifications. For example, the programmer can manually manage the memory exploiting *cache* locality and alignment. Even if the *C++* language is powerful and can be used on different level of abstractions, it has several drawbacks. If it is used as low-level, the portability to different machines or operating systems is reduced and the programmer have to do additional work. In addition, the language itself does not provide many high-level abstractions to the programmer and most of them must be developed or included using third-party libraries. From these and other considerations, we thought to rebuild *FastFlow* in a different programming language. The language chosen for the new library called *BBFlow* is *Java*.

FastFlow

FastFlow provides to the programmers various modular ready-to-use stream-parallel and data-parallel patterns. Those modules can be composed and customized to build complex parallel programs. Therefore, a generic parallel application is expressed by connecting patterns and building blocks in a data-flow *pipeline*.

In the latest version of *FastFlow* (ver. 3), there are a small set of highly efficient, customizable and composable building-blocks that can be composed and nested in many different ways. The building blocks are interconnected using communication channels of type *FIFO* (*First-input First-output*) and *SPSC* (*Single-producer Single-consumer*) that can be *blocking* or *non-blocking* (more details in *Chapter 4*). The building blocks available to the user are of two types: *sequential* and *parallel*. Sequential building blocks are *Node* and *Node combiner*. Parallel building blocks are *Farm*, *Pipeline* and *All-to-all* (see *Chapter 3* for details). These building blocks come with a support code (provided by the *RTS*) allowing the user to choose to use them as they are or override part of them to reach the desired degree of customization. All the building blocks and communication channels present in *FastFlow* are implemented in *BBFlow*.

Java and *BBFlow*

The *BBFlow* project is developed using *Java* programming language as described in the previous sections. Latest *Java* version *(Oracle Java 17)* was chosen for this project. *Java* is a lot different from *C++* and we were expecting from the beginning different performance results of *BBFlow*.

One main difference between *Java* and *C++* is the memory management. It is manual in *C++*, but in *Java* is totally in charge of the *JVM* (*Java Virtual Machine*). Indeed, even *Struct* or *Union* are not present in *Java* programming language. In addition, in *Java* there is the *Garbage Collector* that disposes allocated memory adding overhead (e.g., during the *Stop-The-World* phase) and not allowing to the user to implement custom *destructors*. The *Garbage Collector* simplifies the programmer experience, but in performance sensitive context, it can become a limitation. Another *Java* limitation is the lack of low-level system calls that can be useful when every millisecond is critical. Just to make an example, the *Thread sleep* of *Java* is very unreliable and we needed to find a workaround to it (see *Chapter 4* section *Non-blocking queues*).

On the other hand, *Java* has many big advantages. One of them is the high-level abstraction of the language that makes the *source code* clearer and the massive amount of embedded *libraries* available. The default libraries included in the *JDK (Java Development Kit)* contain almost everything a programmer needs and complex applications can be developed with few rows of code. Another big advantage is the *Java* portability. The *Java* language is platform-independent and the *JVM* is available for almost any common operating system.

The *Oracle Corporation* released in the last years many different versions of *Java*. In each version, there are more or less important changes [5]. In particular, from version *14* there is the *"Pattern matching for InstanceOf"* that permits, after checking if a variable is instance of another, to avoid further casting and use the variable directly. In addition, from version *15*, there is the possibility to activate "*The Z Garbage Collector*" that is a scalable low latency garbage *collector* (see in *Chapter 6,* The *Z Garbage Collector*) and it is very useful when managing big amount of data and *efficiency* allocation/deallocation is needed. There are different important features we exploited and there are others that can be tested in future works. For example, there is a new feature, actually in preview, called *"Foreign Function and Memory API"*. It will be the replacement of the actual *JNI* (*Java Native Interface*). This new feature will allow calling *native* or *external functions* and access memory *outside* the *JVM* [6].

Tools used

There are different tools used to develop the *BBFlow* project. As mentioned before, we used *Java version 17* and its *JDK*. The source code was written using the *"IntelliJ IDEA" IDE* that comes with many useful features like code refactoring, code completion and code debugging. To test the implementation, many different synthetic computations, inspired by the ones found in *FastFlow*, were run. Each computation was debugged and verified through the *IDE*. Therefore, the results were compared with ones obtained running the same synthetic computations using *FastFlow* library. Each *BBFlow* test was compiled using *"javac"* and executed using *"java"* commands. Other command line tools like *"iftop"* or *"time"* were used to measure network traffic and the process execution time. In addition, we generated the *Java* documentation using the *"javadoc"* command.

Chapter 2

BBFlow

The *BBFlow* project is a parallel-programming library developed in *Java*. *BBFlow* provides an abstraction to the user that can realize its parallel software without taking care of underlying mechanisms. It is inspired by *FastFlow* and implements the same sequential and parallel building blocks present. Building blocks can be interconnected through different type of channels. The user application, including *BBFlow* package, can run on a single multi-core machine (using *shared memory channels*) or on a network of workstations (using *TCP network channels*).

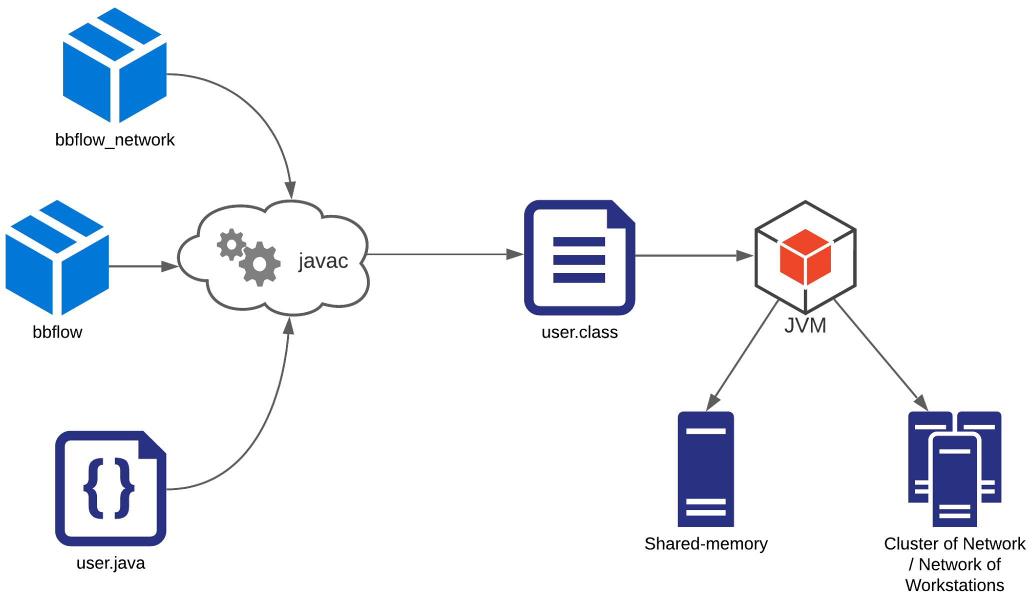


Figure 1: Compilation and execution of a parallel program using *BBFlow* library.

BBFlow comes as a library composed by two packages: *bbflow* and *bbflow_network*. The first one, *bbflow* includes the core of the project, while the second one, *bbflow_network* includes the client-server implementation of the *TCP channels*.

To use the library is sufficient to include, in its own project, the two packages using *"import"*. From that moment, all the classes and methods will be available to the application designer. The source code is compiled using "*javac*" command and a *class* file containing *byte-code* is produced. The compiled application can be executed using "*java*" command that will invoke the *JVM* that will interpret the application *(Figure 1).*

There are many simple and complex synthetic computations present in the project directory under directory "/tests":

- The root directory contains many functional tests of the main components using both type of jobs available (inline or not)
- The *"ff_ tests"* directory contains many *FastFlow* tests reimplemented in *Java* using *BBFlow library*. These tests are used for benchmarking and functional testing.
- The *"benchmarks"* directory contains all benchmarks executed and explained in the Benchmarks chapter. In addition, for each benchmark computation, there are the sequential version and the *C++* version used for the *FastFlow/BBFlow* comparison.
- The *"benchmarks/distributed"* directory contains all benchmarks of the TCP network channels (used in *Chapter 5* in *Network queues* and *Distributed FastFlow*). In addition, there is the *C++* version of each *Java* test used for the *FastFlow/BBFlow* comparison.
- In the "*MSOM"* directory, instead, there is a sample use case application using *BBFlow* packages (see the *Chapter 5*).

Settings

In order to use *BBFlow* packages, users must enforce of the settings in the class *bbflow.bb_settings*. This class defines different static variables that reconfigure the actual behavior of the package. There are different settings and their behavior will be described in details in this thesis.

The settings available are six, briefly described as follows:

- The *BOUNDED boolean* variable is used to decide if the communication channels between building blocks are *bounded* (limited amount of items) or not. *False* by default.
- The *BLOCKING* boolean variable is used to decide if the communication channels between building blocks are of type *blocking* (with *blocking* synchronization mechanisms) or not. *False* by default.
- The *defaultBufferSize integer* variable defines the maximum number of items of the *Bounded* communication channels. It is equal to

Integer.MAX_ VALUE (maximum value possible for an *Integer*) by default.

- The *backOff integer* variable defines an internal passive waiting time in *nanoseconds*. This time is used in different read-write operations on communication channels. The *backOff* technique is a way to shrink the busy-waiting phases alternating them with short passive waiting phases obtained by executing micro-sleeps (*thread* sleeps of few *microseconds*). *1000ns* by default.
- The *serverPort integer* variable is the base *TCP* port number used in the *TCP channels*. Each channel will have a different *TCP* port number assigned. Each *TCP* port number can be calculated as (*serverPort + channel id*). Port *44444* by default.
- The *bufferedTCP boolean* variable defines if the network channels should have a buffer or not. *True* by default.

More details about each setting and configuration can be found in *Chapter 4*.

Chapter 3

Building blocks

Introduction

BBFlow reimplements the building blocks model of *FastFlow*, where the user has *sequential* and *parallel blocks* available. Building blocks are ready to use objects that enable the programmer to create his parallel patterns following a *LEGO*-style approach. The most suitable blocks can be smartly assembled to solve a parallel problem. The blocks can be connected each other using different communication channels as detailed in *Chapter 4*.

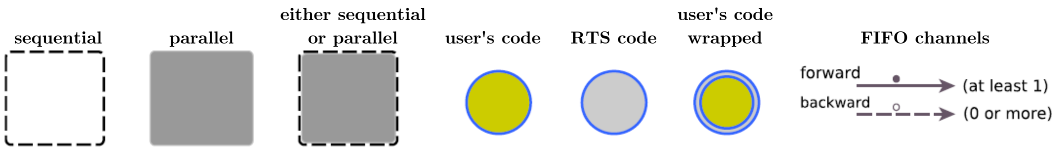


Figure 2: Symbols used to describe *FastFlow* building blocks

To represent the building blocks, we will use the same graphical symbols of *FastFlow* as briefly recalled in *Figure 2* and described in *FastFlow* documentation [7]. The basic abstraction of the building block set is the *node*. It encapsulates either user's code or *RTS (Run-Time System)* code. User's code can be also wrapped by a *BBFlow node* executing *RTS* code. In this way, input and output data can be manipulated and filtered before and after the execution of the business logic code.

In the *BBFlow* implementation, each *node*, called *ff_node*, encapsulates the user's code through another class called *defaultJob*. The *defaultJob* class provides to the user different methods, called by the *ff_node* on runtime, which can be overridden with the user's code.

```java
package tests;
import bbflow.*;

// Anonymous class declaration

public class myclass {
  public static void main (String[] a) {
    defaultJob<Long,Long> myjob = new
    defaultJob<>() {
      public Long runJob(Long x) {
        x += 2;
        return x;
      }
    };

    ff_node mynode = new ff_node(myjob);
  }
}
```

```java
package tests;
import bbflow.*;

// External class declaration

public class myjob extends
defaultJob<Long,Long>{
    public void runJob() {
        // computation
    }
}

public class myclass {
  public static void main (String[] a) {
    myjob job = new myjob();
    ff_node mynode = new ff_node(job);
  }
}
```

Code 1: *ff_node* and *defaultJob* classes initialization

The user class extending *defaultJob* can be instantiated in the following ways:

- As an inline *Anonymous* class, directly in the programmer's code. An example on the left side of the *Code 1*.
- As a standard class with a given name. An example on the right side of the *Code 1*.

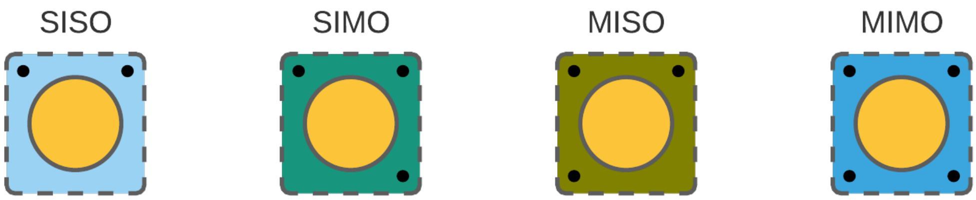


Figure 3: Cardinality of building block *nodes*

The *nodes* can have different channel cardinalities and they are represented as in *Figure 3*. Each *node* could have one or more input channels or one or more output channels. There are four types of graphical blocks to represent them:

- *Single-input single-output (SISO)*: one input and output channels.
- *Single-input multi-output (SIMO)*: one input and multiple output channels.
- *Multi-input single-output (MISO)*: multiple input and single output channels.
- *Multi-input multi-output (MIMO)*: multiple input and multiple output channels.

Depending on which kind of *node* the user needs, the two methods shown in the *Code 2* can be overridden.

```
public U runJob(T element);
public void runJobMulti(T element, LinkedList<ff_queue<U>> channels_output);
```

Code 2: Methods available for *workers* declared *anonymously*

In the *Code 2*, the first method is for standard *nodes* with cardinality one-input and one-output *(SISO)*. The method is called every time the *node* receives a new element of type *T*. This method must return value of type *U* or *null*. In case a *null* is returned, nothing will be sent to the output channel.

The second method in the *Code 2* is for multi-input or multi-output *nodes (SIMO, MISO* and *MIMO)*. The method is called every time the *node* receives a new element from one of the input channels (in *Roundrobin*) and the output items can be sent using directly the output channels (passed as parameter of the method) or using different *RTS* calls (listed in *Code 3*) available in all kinds of *nodes*.

```
void sendOut(U element); //sends element to the next output channel
(Roundrobin)
void sendOutToAll(U element); //sends element in broadcast to all out channels
void sendOutTo(U element, int index); //sends element to channel 'index'
```

Code 3: Methods available to send elements on the output channels

Another way to extend the *defaultJob* class is creating a standard class (as shown in the *Code 1*). In this case, a method called *runJob()* without parameters is called in loop. In this case, everything is in charge of the programmer that must manage the read and write on the channels (*channels* access is available through *in* and *out* global variables) and the *EOS (End Of Stream:* special mark used to manage streams). This approach is more useful when the implementation requires a particular degree of customization (see *Chapter 6* for a case of use).

Anyway, in both *anonymous* and standard implementation, the user has the ability to choose to be wrapped or not by *RTS* changing *runType* variable in *defaultJob* class (more details in the *BBFlow* documentation).

```java
package tests;
import bbflow.*;

public class myclass {
  public static void main (String[] a) {
    defaultJob<Long,Long> myjob = new
    defaultJob<>() {
      public Long runJob(Long x) {
        x += 2;
        return x;
      }
    };

    ff_node mynode = new ff_node(myjob);
  }
}
```

```java
package tests;
import bbflow.*;

public class myclass {
  public static void main (String[] a) {
    defaultJob<Long,Long> myjob = new
    defaultJob<>() {
      public Long runJob(Long x) {
        if (x%5 == 0) {
            x += 2;
            sendOut(x);
        }

        return null;
      }
    };

    ff_node mynode = new ff_node(myjob);
  }
}
```

Code 4: *SISO node* using *Anonymous* class declaration.

In the *Code 4* is shown the *defaultJob Anonymous* declaration. On the left side, there is the programmer task returning value after the sum. On the right side, the resulting value is sent through the usage of the *RTS* call *sendOut(U element)*.

```java
package tests;

import bbflow.defaultJob;
import bbflow.ff_queue;

public class myjob extends defaultJob<Long,Long> {
  public void runJob() throws InterruptedException {
    Long received;
    ff_queue<Long> in_channel = in.get(0);

    received = in_channel.take();
    if (received == null) { // EOS received
      in.remove(0); // Removing input channel
      return;
    }

    received += 2;
    sendOut(received);
  }
}
```

Code 5: *SISO node* using standard class declaration and manual channel management.

Instead, the *Code 5* is an example of a standard class extending *defaultJob* where the channels are managed manually. The *RTS* support is still available and the example makes use of *sendOut(U element)*. This code has the same behavior of the left code in the *Code 4.*

Sequential building blocks

There are two types of sequential building blocks, *ff_node* and *ff_comb*.

ff_node is the basic *node* entity where the user can add its code and consists in a *Java Thread* running a *Runnable defaultJob* class. Every other block is constructed on top of *ff_node*. Each *node* can have one or more input/output channels and can be connected with other blocks. The *ff_node* encapsulates a *defaultJob* extended class, containing user's code and *RTS* code. The methods called during a *node* execution are shown in the *Code 6.*

```
public void init();
public void runJob(); // in case of non-anonymous class
public U runJob(T element); // anonymous class + cardinality 1in-1out
public void runJobMulti(T element, LinkedList<ff_queue<U>> channels_output);
// anonymous class + multi-in/out
public void EOS();
```

Code 6: *defaultJob* methods that can be overwritten by the user

The *init* method is called after the *node* starts, independently from the cardinality of the *node*. It is useful to prepare the *node* to receive the elements, e.*g*. to instantiate objects, to send elements to the output channel(s) in case the *node* acts as a generator, etc.

The *runJob*/*runJobMulti* methods are called according to type of class (*Anonymous* or not) and the channels cardinality of the *node* chosen. The user, when implementing a *node*, should override the correct method according to the choice made.

The EOS method, instead, is called when the stream of items ends on all input channels. The *EOS* method is called only when *RTS* is wrapping user's code. Without *RTS* support, the user must manage *EOS* directly in the *runJob* method as shown in the *Code 5* example (more details about *EOS* in the *Chapter 4*).

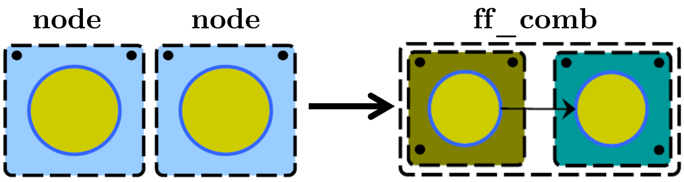


Figure 4: Combine transformation of two standard *nodes*

The sequential *node combiner*, ff_*comb*, combines two *nodes* into a single one. The combination is linear and the first *runJob* output goes directly to the second *runJob* input. i*nit()* and *EOS()* methods of both combined *nodes* are called in the final *node*. In practice the new *ff_comb node* acts as a single *node* encapsulating the two *nodes* functions inside. The *ff_comb node* runs on a single *Thread*.

The resulting *node* of the *ff_comb* has cardinality multi-input/multi-output *(MIMO)* independently from the source *nodes (Figure 4)*. This is an advantage and not a drawback, because *MIMO nodes* work even if there is only one channel connected and the programmer can combine any *node* without taking care of the cardinalities. A *MIMO node*, when running, scans all input channels for new elements and if there is only one channel, the same channel is read until the *EOS* is reached. The same thing happens on the output channels that will write the channels available, even if only one.

Parallel building blocks

There are three types of parallel building blocks present in *BBFlow:*

- *The Farm* building block (*class ff_farm*) is a ready to use classical *farm* template consisting, by default, by an *Emitter*, *n workers* and a *Collector*. The *Emitter* process gathers input tasks from the input stream and schedules these tasks for execution on one of the available *workers*. *Workers*, in turn, receive tasks to be computed, compute them sending the result to the *Collector*.
- *The Pipeline building block* composes in a *pipeline* manner as set of building blocks (*class ff_pipeline*). The *Pipeline* parallel pattern can be described as a chain of stages (building blocks) where the flow of data traverses the stages one by one. The *Pipeline* ensures that every *node* of the chain is executed in parallel.
- *All-to-all* connects two different farms in different configurations (*class ff_all2all*). The combination of two farms has as a main reason the bottleneck removal of the *Collector/Emitter*.

Even if each of them is just a composition of *nodes* (*ff_node*) and channels, they are a powerful and flexible abstraction for the user. Each parallel building block is explained in detail in *FastFlow* documentation so we will focus mainly on implementation choices.

ff_farm building block

The *Farm* building block, as described before, is composed, by default, by an *Emitter*, *n* *workers* and a *Collector*. The *Emitter* is connected to all *workers* and the *workers* are connected to the *Collector* (as shown in *Figure 5).*

Emitter and *Collector* *RTS* are defined by *defaultEmitter* and *defaultCollector* classes, both extending *defaultJob* class. They can be both removed from the *Farm* and replaced by user's code (two examples in the manual: Code 21 and Code 22). The two methods *connectEmitterWorkers()* and *connectWorkersCollector()* can be user to reconnect the new *nodes* to *workers*. There is also a way to instantiate a *farm* without any *node* where the user can customize every element having anyway the *RTS* support.

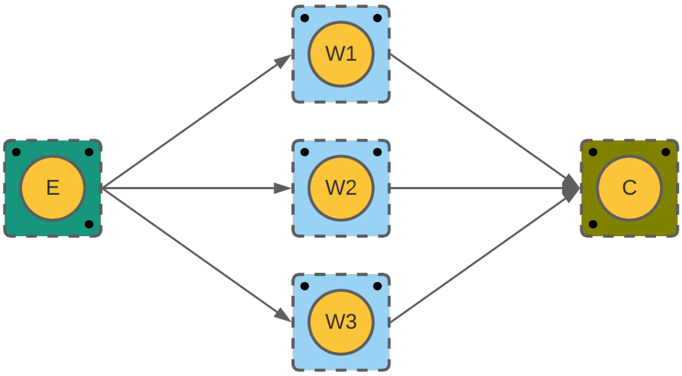


Figure 5: *ff_farm* building block default topology

Emitter implements three kind of item distribution to the *workers*: *Roundrobin*, *Scatter* and *Broadcast*

Roundrobin, given *n* output channels (equal to number of *workers*), distributes elements one per *worker* starting from the first *worker*. Once last *worker* is reached, the distribution continues restarting from the first *worker*. Even if the *Emitter* is in *Roundrobin*, the *EOS* signal received from *Emitter*'s input channel is propagated to all *workers*, in *broadcast*. The *EOS* signal is propagated in *broadcast* in all kinds of *BBFlow nodes* to guarantee correct termination of the entire network.

Emitter with *Scatter* distribution, instead, splits the data received from input channel, which is of type *Collection*, into *n* chunks and sends each of them to the *workers* in *Roundrobin* manner. If the size of input collection is not divisible by *n*, the leftover chunks are distributed uniformly starting from first *worker*. The input data to split must be at least of size *n* to ensure to have enough chunks.

The last distribution scheme, *Broadcast*, sends the received item to all *workers*. The item is a passed as a reference so that all *workers* will share the same object, reducing memory accesses. This can cause a concurrency problem if a *worker* writes on the received shared *object*. In this case, the user is responsible to manage synchronization mechanisms or replace *Emitter* with a custom solution.

Actually, there are not already implemented *feedback channels* in *BBFlow*. A *feedback channel* is a *node* interconnection directed in the opposite direction than the standard data flow. Even if these channels are not a standard element of *BBFlow*, they can be created manually instantiating new channels and connecting them properly. The interconnection is possible through *addInputChannel* or *addOutputChannel* methods provided by the *RTS* (interconnection example in *Code 16*).

On the other hand, *Collector*, implements three ways to collect elements received from the *workers*: *Roundrobin*, *Firstcome*, *Gather.*

Roundrobin works as for the *Emitter*, the *Collector* waits the element from each input channel one by one and when last channel reached, the process restart from the first one. This way to collect objects blocks the *collector* until the element is received on the target channel, even if there are other data available from the other *workers*. When a *Worker* finishes its tasks, the *EOS* signal received from the *Emitter* will be propagated to the *Collector*. Once the *Collector* receives the *EOS* from one channel, the channel will not be considered anymore in the *Roundrobin* scanning; this will avoid any critical situation where the *Collector* waits indefinitely for a new item from a terminated *worker*.

Firstcome is a way to overcome to the *Roundrobin* limitations and consists in scanning all the channels and collecting items as soon as they are available. The scan in executed in round-robin using *poll()* function and a *backoff* time. More details are present in the *Chapter 4.*

Gather that is commonly on the other side of *Scatter*, aggregates the chunks received from *workers (one item from each one of the workers)* in a single *Collection* that is sent to the output.

When instantiating an *ff_farm* object, *input* and *output* types must be specified correctly in order to make *Scatter* and *Gather* work as detailed.

The *Emitter* and *Collector* configuration in *Roundrobin* or in *Scatter/Gather*, guarantee the total ordering between input and output, so the user can easily implement an *Ordered Farm* (*Farm* that provides a *total ordering* between input and output). Another way to implement it, is to use custom *packets* (*items*) containing a *packet id*. The *packet id* can be used to keep track of the order of the items and decide which is the next *packet* that should be sent to the output channel. More detailed information and an example about the *Ordered Farm* using the *packet labeling*, can be found in the *Appendix A*.

ff_pipeline building block

The *ff_pipeline* building block is a chain of multiple *nodes*, sequential or parallel, connected each other using communication channels. Each *node* will run in parallel and *ff_pipeline* acts just as a container, abstracting *nodes* interconnection and other methods that the *pipeline* stages have in common. *Nodes* inside the *pipeline* can be connected together in many different topologies to cover all the possible common combinations.

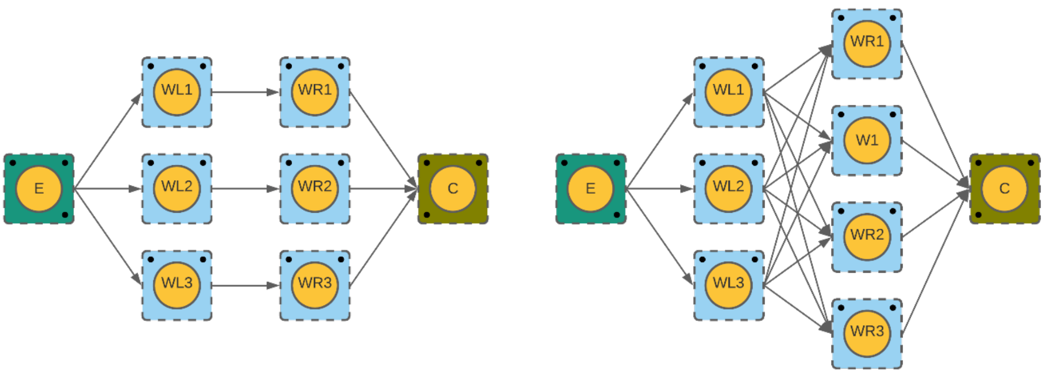


Figure 6: *Pipeline N-to-N* and *NxM* configurations

There are five different interconnection topologies:

- *1-to-1*: only one channel is created between the two *nodes* where the first one is the *producer* and second one is the *consumer*. Can be used on any *nodes*, including *ff_farm* if *Emitter* or *Collector* are present (depending on which side the *ff_farm* is).

- *1xN*: *N* channels connecting one *node* from the left to *N nodes* on the right. For example, this is the standard configuration for an *Emitter*, connected to all *workers*. This configuration can be used to replace an *Emitter* or to create a custom network.
- *Nx1*: the opposite of *1xN*, where *N nodes* from the left are connected to one single *node* on the right. This is the standard configuration of a *Collector*.
- *N-to-N*: this allows connecting *N nodes* one by one in 1-to-1 interconnection (left side of *Figure 6)*. Useful if two farms, without *Collector* and *Emitter* respectively, should connect their *workers* in order that left-*worker* sends data to the right-*worker*. The cardinality of the two farms must be the same; otherwise, the *NxM* topology will be used.
- *NxM*: this connects *N nodes* to *M nodes* in all-to-all configuration (right side of *Figure 6)*. Each left *node* is connected to each right *node*. The resulting topology implies that each *worker* (from both sides) become a multi-input/multi-output.

The default constructor of *ff_pipeline* takes only two *nodes*, with the chosen topology, and each new *node* can be added to the *pipeline* with *appendBlock* method. 1-to-1 interconnection is used by default if no configuration specified.

ff_all2all building block

In the *ff_farm* building block, even if is powerful and flexible, the *Emitter* or *Collector* can become a bottleneck of the *node* being centralized and sending/receiving data from many *workers*. For this reason, in different applications, it is necessary to remove the centralization point and interconnect the *farm* together. In the *ff_all2all* block, two set of *workers L-workers* (from left *farm*) and *R-workers* (from right *farm*) can be composed in many different ways. All the possible compositions we implemented are taken from the *FastFlow* documentation [7].

After the *farms* composition, the resulting *ff_all2all* building block is a *node* and can be used as a *pipeline* of two farms. In fact, the *ff_all2all* block shares many different configurations with *ff_pipeline* (like *NxM*).

There are many examples of *ff_all2all* building block in *BBFlow* tests *directory*, which reproduce the same ones present in *FastFlow*.

The user, when creating an *ff_all2all* building block, can provide as a support, two different *nodes R* and *G* that will be combined respectively to left and right *workers*. These two *R* and *G nodes* enable each *worker* to communicate properly in the new all-to-all topology with other *workers*. For example, *R-workers*, in the original right *farm*, if are not multi-input, a new *G node* combined with them will allow receiving data from multiple *L-workers*.

The *ff_all2all* constructor does not take any parameter. The composition of the two farms is done through *combine_farm* method.

The *ff_all2all.combine_farm* method shown in *Code 25* takes five parameters: left *Farm*, right *Farm*, *R node*, *G node* and *merge* variable.

All the possible combinations of the All-to-all building block, taken from *FastFlow* documentation, are:

If *merge* variable is *false,* we can distinguish the following cases:

- Both *R* and *G* are not *null*: it produces exactly the topology sketched in *Figure 7*.
- Only the *R* is not *null*: it produces a *pipeline* of a single *farm* whose *Worker* is an *all-to-all* building block where the *nodes* of *L-Workers* are a composition of first *Farm's Workers* and the *node R*.
- Only the *G* is not *null*: it produces a *pipeline* of a single *farm* whose *Worker* is an *all-to-all* building block where the *nodes* of the *R-Workers* are a composition of the *G node* and second *Farm's Workers*.
- Both *R* and *G* are *null*: it produces the topology sketched in *Figure 6* (right side) where *first Farm's Emitter* and *second Farm's Collector* are removed and *Workers* are connected in *pipeline* in an *NxM* configuration.

If *merge* variable is *true,* we can distinguish the following cases:

- Both *R* and *G* are not *null*: this produces a *pipeline* of two farms where the first one does not have the *Collector* while the second one has as *Emitter node*, the composition of the *R node* and *G node* (this is the case sketched in *Figure 8*).
- Only the *R* is not *null*: this produces a *pipeline* of two farms where the first *farm* does not have the *Collector node* while the second *farm* has as *Emitter* the *R node* (this transformation substitutes the *Emitter* of the second *farm* and removes the *Collector* of the first *farm*, if present).

- Only the *G* is not *null*: this is equivalent to the previous case (Only the *R* is not *null*) where *R*=*G*.
- Both *R* and *G* are *null*: in this case, if the parallelism degree of the two farms is the same, it applies the *farm* fusion transformation (sketched in *Figure 6,* left side). If the parallelism degree of the two farms is different, the transformation applied is the same as the case when *merge=false.*

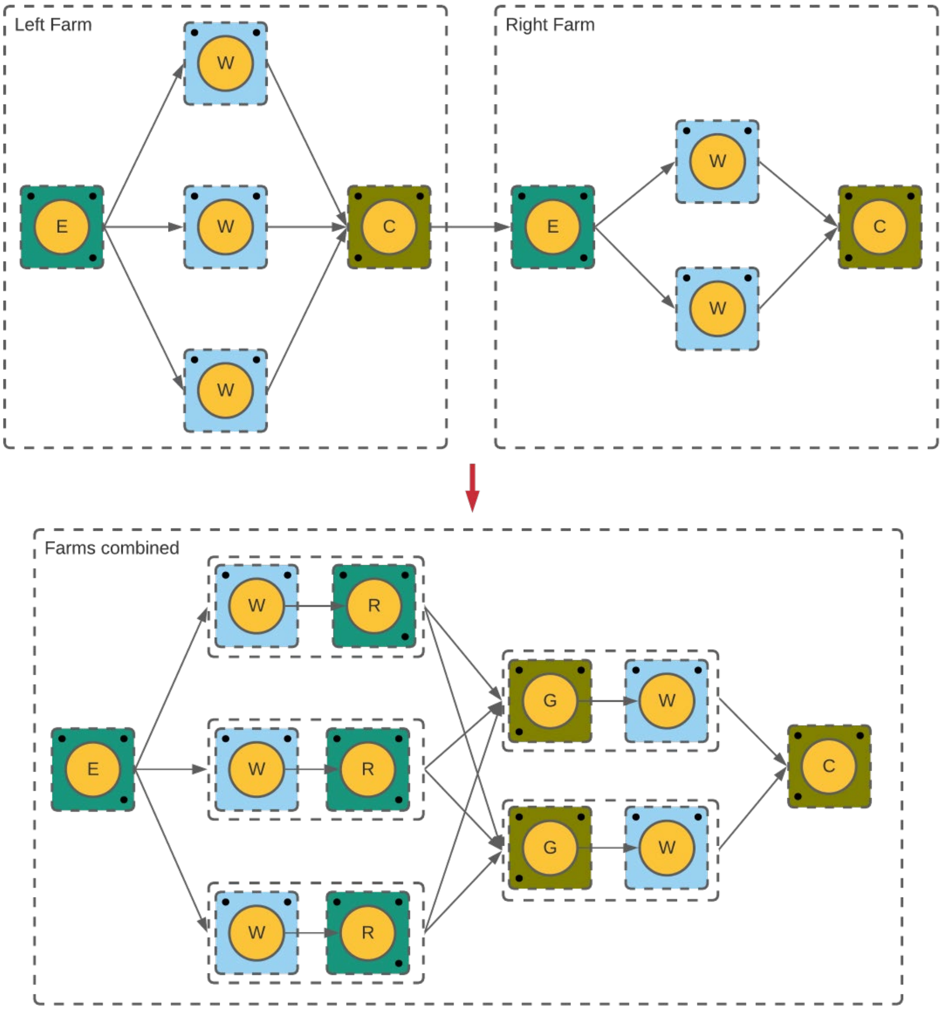


Figure 7: *Combining* two *farms* with user-defined *Collector* and *Emitter*

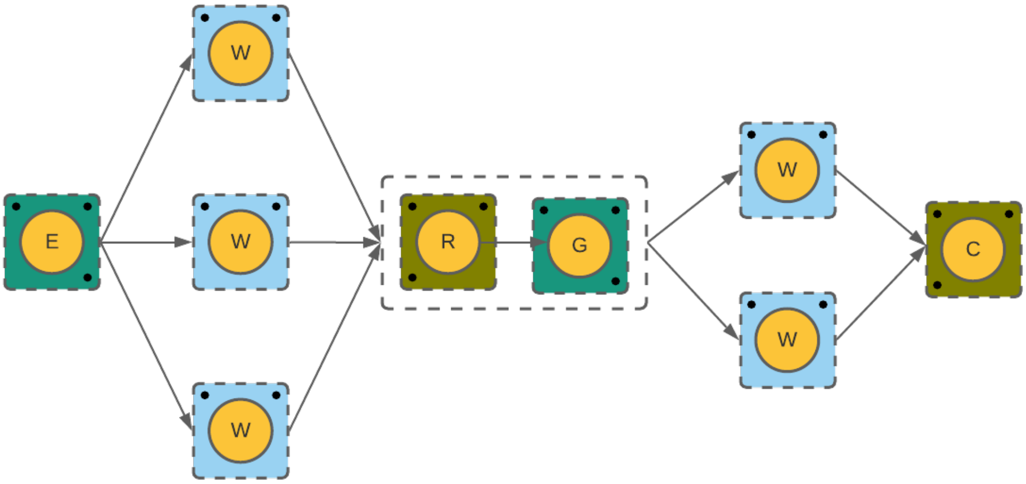


Figure 8: *Farm* composition in *pipeline* using custom combined R and G *nodes*

Chapter 4

Communication Channels

Introduction

In this chapter, we describe the *BBFlow* communication channels used to interconnect all types of building blocks letting them to communicate each other.

A communication channel between two blocks is a *1-to-1 FIFO* (*First-In First-Out*) queue. Each channel is of type *Single-Producer Single-Consumer* (*SPSC*). In *SPSC* queues, the *producer* pushes new element in the queue and the *consumer* pops it.

Producer and *Consumer* roles cannot be swapped so if the two entities need a bi-directional communication, two channels are actually needed.

The channel is abstracted by the class *ff_queue* implementing two class of queues that are *blocking* and *non-blocking*. Both queues can be *bounded* or not. In the user application, all types of queues can coexist all together granting a very good degree of freedom to the *BBFlow* user.

The data items exchanged through the channels are objects and are passed "by reference" from one *node* to another.

A new layer of abstraction of the queues, extending a *non-blocking unbounded* queue, is the network channel implemented by the class *ff_queue_TCP*. The queue, from the user point of view, acts as a local queue except for the fact it communicates through *TCP* network using a simple object serialization/deserialization protocol rather than using shared-memory.

In the next paragraphs, we focus on how queues are implemented and which are their limits.

Blocking queues

The simplest implementation for a *FIFO* channel, in a multi-threaded context where different threads try to access the same data structure, is the synchronized queue.

The *producer* and the *consumer* of the channel synchronize themselves using a lock on a shared object and only one thread can have access to the queue keeping the other thread waiting for its turn.

Synchronization mechanisms affect the performance of the threads execution and if the throughput is slower than the item processing time, the queue becomes a bottleneck. In this case, a *non-blocking* queue can offer a better solution to the user.

This queue is implemented extending the already existing *Java* queue class *LinkedBlockingQueue* that is an optionally *bounded blocking* queue based on linked *nodes*.

We chose a queue based on a linked list considering the fact that the queue is *FIFO* (so *producer* pushes element at the tail of the list and *consumer* pop it from the head) and dynamic (so can contain a different amount of elements in each phase of the user application execution).

Using the *LinkedBlockingQueue*, we do not need to pre-allocate memory space for the elements and we can push/pop elements in linear time. *Java* class *LinkedBlockingQueue* typically has higher throughput than *Java* array-based queues.

This queue can be *bounded* or *unbounded*, letting the user to decide, in case of *bounded*, the size of the linked list during the initialization.

The *bounded* queue implements different methods to push an element, including *put* and *offer.* The *put* method waits indefinitely up to the point in time where there is enough space available to host the new item. The *offer* method tries to push the item (with or without a timeout) and if there is not enough space available, the function returns *false*.

Anyway, even the *unbounded* queue presents a size limit that is 2^{31} (the max value of integer). Although, in the most practical uses, it can be considered fairly *unbounded*.

Non-blocking queues

The *non-blocking* queues implementation employs an efficient "wait-free" algorithm based on the one described by Michael and Scott [8] and it is based on the existing *Java* class called *ConcurrentLinkedQueue*.

Again, the queue is based on a linked list, preserving previous enounced properties. The *Java* default class is *unbounded*, so the *bounded* version called *squeue* is an extension of the *ConcurrentLinkedQueue* taking inspiration from the *bounded* version of the queue in *Hadoop HBase* [9].

Having the *Java* class *ConcurrentLinkedQueue* a *size()* method non-linear in time, we implemented in *squeue* an *AtomicLong* variable that keeps the actual size of the queue and is atomically updated.

When pushing or popping an element, if the operation *must* succeed, the technique called *backoff* is used [1]. The operation waits passively, executing a thread sleep for few microseconds, until it will be possible to actually push or pop the item. Lower *backOff* values implies higher load while higher values imply low channel responsivity; so a tradeoff should be found and a good range of value in typical application is between *100ns* and *500ns*.

The *backOff* setting can be changed using *bb_settings* class and in particular critical applications, lowering the value to *100ns* could be ideal to reduce the execution time.

Initially, as asynchronous waiting *Thread.sleep()* function was used. Unfortunately, this function has a very low precision and the execution time of user' application turned out to be too volatile and as consequence, the *scalability* was affected.

To overcome to this problem, a new function called *sleepNanos* is now present in the *ff_queue*. The method works as follows:

- If the waiting time left is greater of the precision of the *Thread.sleep()* (calculated around of 2ms), the *Thread* goes to sleep for 1ms using the sleep function.
- Instead, if the waiting time left is smaller, the *Thread.yield()* method is called.

Yield method is used to inform the scheduler that the current thread is willing to relinquish its current use of processor, but it would like to be scheduled back soon as possible. *Scheduler* takes into consideration the request of the thread and will decide if continue with a context switch or to leave the current thread running. The probability of a context switch is non-zero, but it is less than *Sleep*. *Sleep* should be preferred in any common case, but different empirical studies [10] [11] and our tests demonstrated that *Yield* has a smaller slowdown on the overall performances and the total sleep time can be well controlled.

Channels implementation

The class *ff_queue* implements all the queues above (*blocking/non-blocking bounded/unbounded*) in a transparent way for the end user.

When a new ff_*queue* is instantiated, the user can choose which type of queue to use and its size (in case of *bounded* one).

The operations allowed on each channel are:

- *put(T element):* inserts the specified element at the tail of the queue, waiting if necessary for space to become available. There is no waiting time in *unbounded* queues; in this case, the call immediately succeeds.
- *offer(T element, [timeout]):* inserts the specified element at the tail of the queue if it is possible to do so immediately without exceeding the queue's capacity and optional timeout, returning *true* upon success and *false* if this queue is full or time exceeded.
- *take():* retrieves and removes the head of the queue, waiting if necessary until an element becomes available.
- *poll([timeout]):* retrieves and removes the head of this queue, or returns *null* if this queue is empty. If the optional timeout specified, waits the element the requested amount of time returning it or *null* if time exceeded.
- *size():* returns the number of elements in this queue.
- *setEOS():* sets in the queue that the End Of Stream (*EOS*) has been reached. No new elements will be accepted anymore.
- *getEOS():* check if the *EOS* is set for the queue.

The *End Of Stream (EOS)* is used to inform *nodes* connected to the channels that they should terminate their execution once the queue will be empty. *EOS* can be set even if there are still elements in the queue, so the end user should check if both *getEOS()* is *true* and the size of the queue is *0*. Once *EOS* is set in a channel, no new items can be pushed to the queue and this will avoid any critical concurrent situation.

The *take()* method return *null* if *EOS* is set like *poll().* So if when retrieving an element the returned value is *null* and the *getEOS()* is *true*, the *node* can terminate.

Network Channels

On top of the *ff_queue* class, a new one called *ff_queue_TCP* extends the channel implementation over *TCP/IP* network.

A single-threaded client-server communication will be established between the two communicating *nodes*. The channel is still *SPSC* and only the *client* sends elements to the *server*.

Each *node* instantiates a new *ff_queue_TCP* but only the *server* keeps a queue data structure in memory where the elements received by the *client* are stored.

Client, during *put* and *offer operations*, directly sends serialized items over the network to the *server*. The serialization is possible using the classes *ObjectOutputStream* on the *producer* and *ObjectInputStream* on the receiver. Any non-primitive type will be serialized and sent on the channel. On the other side, the receiver unserializes the data and casts to the correct type.

```
ff_farm stage1 = new ff_farm<Integer,Double>(n_workers, workerJob);
stage1.addOutputChannel(new ff_queue_TCP(ff_queue_TCP.OUTPUT, 1, "10.0.0.4"));
```

Code 7: Example of initialization of *network output channel (client)*

When *EOS* reached, a special string *"EOS"* is sent from the *client* to the *server* communicating that the *stream* reached its end.

On the *server* side, both operations *take* and *poll* act like on local queues being elements and data structure resident in local memory.

```
ff_node stage2 = new ff_node<Double,Integer>(new complete_farm_testOutnode<Double,Integer>(15));
stage2.addInputChannel(new ff_queue_TCP(ff_queue_TCP.INPUT, 1));
```

Code 8: Example of initialization of *network input channel (server)*

This queue extends the *non-blocking* and *unbounded* version of the *ff_queue*. No synchronization mechanisms are present between *server* and *client*. *Server* is just the receiver and does not send any data to the *client*.

The creation of the channel is done instantiating an *ff_queue_TCP* class in two modes: *INPUT* for the server and *OUTPUT* for the *client* (as shown in the two pieces of code: *Code 7* and *Code 8*). The channels are added as input and output channels using the two methods available for each building block *addInputChannel* and *addOutputChannel*.

The channel is identified by an *id* specified during channel creation and must be the same on both *nodes* communicating each other.

The *TCP* port used is dependent on the *id* of the channel. Every channel will have different *TCP* port and once *EOS* reached, the connection will be terminated and the *server* (receiving elements) will close its socket. The *TCP* port of a channel is calculated using the global *bb_settings.serverPort integer* variable. The channel port number can be calculated as (*serverPort + channel id*). Default value of the global *serverPort* is *44444*. As an example, a channel with id 4, will listen/connect on port *44444+4 = 44448*.

The connection between *client* and *server*, in case of failure, try the reconnection indefinitely. Therefore, when items are sent through the channel, they are guaranteed to arrive.

Chapter 5

Performance comparison

The comparison of *BBFlow* with *FastFlow* will be investigated on different levels. The differences are many, starting from the programming language up to the implementation, but we will try cover main aspects, benefits and drawbacks of each implementation.

BBFlow project has not as purpose to be a clone of *FastFlow*, but just to reimplement the building blocks structure to make an efficient comparison between the implementation between two different programming languages, *Java* and C++.

In *BBFlow,* we tried to implement all the main functionalities required to test and compare the differences in performance between the two projects.

This project is modeled trying to replicate the *FastFlow* structure and its implementation choices in order to make a comparison with the same general assumptions.

Programming language

BBFlow is developed in *Java* and runs on the actual latest version: *Java 17*. We tried to realize it using already-existing *Java* classes (for queues, threads, etc.) and see how a high-level implementation will affect the performances.

FastFlow project is tuned and optimized in deep, realized and refined across many years and we did not pretend to have the same degree of optimization; instead, we want to make the raw comparison with a high-end *Java* implementation and the fine-tuned one in *C++.*

The main difference is that *Java* language runs on a *JVM* and this add a good overhead during execution. Running a *Java* program involves either interpreting. *JVM* instructions, compiling them into instructions of the underlying hardware, or directly executing them in a hardware implementation of the *JVM.* The *JVM* loads the class file (pre-compiled bytecode) containing the program's entry point and execution begins. The program may reference other class files, which are loaded in turn. In addition, to maintain the integrity of the *Java* execution model, the *JVM* checks that variety of semantic constraints are met, both within a class file and between class files [12]. All these operations executed by the *JVM* impact on the execution time having a distributed drop in performances. On the other hand, *Java* language has as advantage that is completely portable and independent from the target architecture, unlike the *C++* implementation that is fine-tuned for the target platforms and migration to new architectures requires some amount of work.

Another main difference is the memory space occupied, in particular to host variable types. All blocks in *BBFlow* use generics to specify types of input and output of each building block. This helps the user to receive and send the correct data type and avoid casting every element. As a drawback, generic types impose to use wrapper classes, like *Integer* for *int*, wasting memory space, adding overhead to reach the element value, and loosing alignment of variables in memory.

Generic types, even if removed from blocks letting the casting task to the user, are present in many other *Java* data structures like *Collections* (*ArrayList*, *LinkedList*, ...) which imposes the usage, anyway, of non-primitive types.

```java
Integer n = 7;
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(n);
Integer m = list.get(0);
```

```java
Integer n = 7;
ArrayList<Object> list = new ArrayList<Object>();
list.add((Object) n);
Integer m = (Integer) list.get(0);
```

Code 9: *Java Generics* in compilation phase

The reason behind the fact *Java* generics do not allow primitives is due to backward compatibility with old versions. *Generics* in *Java* are entirely compile-time construct and all types are casted to *Object (see Code 9)*, forcing to have non-primitive types [13].

Benchmarks

In the following chapters, we analyze the performances of our *Java* implementation running on some test machines different patterns for both *Java* and *C++* projects.

The machine used in the shared memory benchmarks is a *dual-CPU* server with *2xAMD EPYC 7551* and *128GB* of *RAM*. Each *CPU* has 32 cores and 64 threads. The *CPU base* clock frequency is *2.0GHz* and *3.0GHz* on *boost* [14]. The operating system running on that workstation is *Ubuntu 18.04*.

Two other machines are used for *TCP/IP channels* testing. Both of them are *single-CPU* with *AMD Ryzen 7 5800X* and *128GB* of *RAM*. The *CPU* has *8 cores* and *16 threads*. The processor *base* frequency is *3.8GHz* and *4.7GHz* the *boots* one [15]. The two servers are interconnected through a *Gigabit switch* and are in the same datacenter. The *Round Trip Time* (*RTT*) between them is of about *0.4ms*. The operating system running on these workstations is *Ubuntu 20.04*.

Queues

The type of queues used to interconnect building blocks in *BBFlow*, are the same as *FastFlow*, *FIFO SPSC* queues. Their *efficiency* has been widely investigated in the scientific works [8] [16] [17] [18].

In *BBFlow,* we used already-existing data structure to build the queues. The main component of the channels is *Java LinkedList*. Linked list is a linear collection of elements where each one points to the next one. Theoretically, being the queues *FIFO*, this list should be in our case the most efficient one across all *JCF* (*Java Collection Framework*). The reason is that we add and remove elements only to/from the head or tail of the queue. These two operations are linear in linked lists. In addition, the memory management is more efficient respect to the arrays implementation, being the memory required to host the items not allocated in advance.

Everything is theoretically clear, but the *Java* implementation reserves surprises. Things change quickly when the *consumer* of a queue is slower to process items than the time spent by the *producer* to send the elements. When the size of the queue increases, the performances degrades quickly and any other *ArrayList* implementation become a better choice [19].

For this reason, as a further optimization, new types of list should be tested as a possible replacement.

Anyway, if the number of elements in the queues do not exceed *1K*, the performance differences between *LinkedList* and other possible alternatives are negligible.

We implemented two main types of shared-memory queues, as explained in detail in *Chapter 4*, *blocking* and *non-blocking*.

Blocking queues are based on *Java* class *LinkedBlockingQueue* that extends the *LinkedList* adding *blocking* synchronization mechanisms. The concurrent access to the queue is managed by the *Java* class that exploits different locking mechanisms.

The *non-blocking* queues are based on *Java* class *ConcurrentLinkedQueue* that implements wait-free mechanisms [8].

Figure 9: *Producer* and *Consumer* interconnected in *pipeline*

After implementation, the two types of queues were compared running the same benchmark using both *BBFlow* and *FastFlow* project. The benchmark is done using two building blocks (*nodes*) connected in *pipeline (Figure 9)*. The first *node*, the *generator* (*Producer*), produces a stream of items (numbers of type *long*). The second *node*, the *receiver* (*Consumer*), receives the items from the input channel and manipulates their values (simple arithmetic multiplication by *2*).

The same synthetic computation, implemented using both projects *(Code 10)*, is executed on a different amount of data and on both types of queues.

We tried to send elements (from *1k* to *100M*) from the *generator* to the *receiver* measuring the *Java* performances. The *Long* object compared with *C++* long primitive and the *JVM* overhead were expected to have an impact on the queues performance.

```java
defaultWorker<Long, Long> Emitter = new
defaultWorker<>() {
    public Long runJob(Long x) {
        return null;
    }
    public void init() {
        for (long i = 1; i <= finalN; ++i) {
            sendOut(i);
        }
        sendEOS();
    }
};

defaultWorker<Long, Long> Filter1 = new
defaultWorker<>() {
    public Long runJob(Long x) {
        x *= 2;
        return null;
    }
};
```

```cpp
struct Emitter: ff_node_t<long> {
    int ntask;

    long *svc(long*) {
        for(long i=1;i<=ntask;++i) {
            long *t;
            t = (long*)malloc(sizeof(long));
            *t = i;
            ff_send_out(t);
        }
        return EOS;
    }
};

struct Filter1: ff_node_t<long> {
    long *svc(long *in) {
        *in = (*in)*2;
        return GO_ON;
    }
};
```

Code 10: Same *Java* and *C++* implementation of the benchmark tests

To mitigate the *JIT (Just in Time)* overhead of *JVM (Java Virtual Machine)* for non-already loaded classes, in the *BBFlow* project is present a class called *preloader* that can be used to preload all the commonly used classes of the project before starting the main task. This class provides a static method called *preloadJVM* that invokes an empty method for each *BBFlow* class in order to let the *JVM* to preload the entire library. The benchmarks with the *preloader* (called *BBFlow PL*) are included in the *Table 1.*

Non-blocking	**1K**	**10K**	**100K**	**1M**	**10M**	**100M**
FastFlow	0,4	1,012	7,8	76	716	7035
BBFlow	8	25	69	173	751	7092
BBFlow PL	8	17	58	159	715	7034

Blocking	**1K**	**10K**	**100K**	**1M**	**10M**	**100M**
FastFlow	0,343	0,929	5,15	50,79	482	4818
BBFlow	7	32	183	1165	13383	65806
BBFlow PL	5	21	64	221	5451	48379

Table 1: *2-nodes* benchmark on *blocking* and *non-blocking queue*

The registered times in the *Table 1* are in *milliseconds* and they are obtained averaging multiple tests on the same machine.

Non-blocking queue

The *non-blocking* queues results (shown in the *Table 1)* highlight clearly how the *JVM* affects the computation performances. When the execution time of a computation is less than *100ms*, the *JVM* overhead weights enough to make the execution ten times longer than using *FastFlow*. The *Java* overhead can be reduced using the *BBFlow preloader* and obtain a better execution results. With the *preloader*, the execution time reduces of many percentage points giving the possibility to *BBFlow* to reach the same performance of *FastFlow* on long runs. In fact, when the data stream is long enough, the *JVM* overhead impact become negligible.

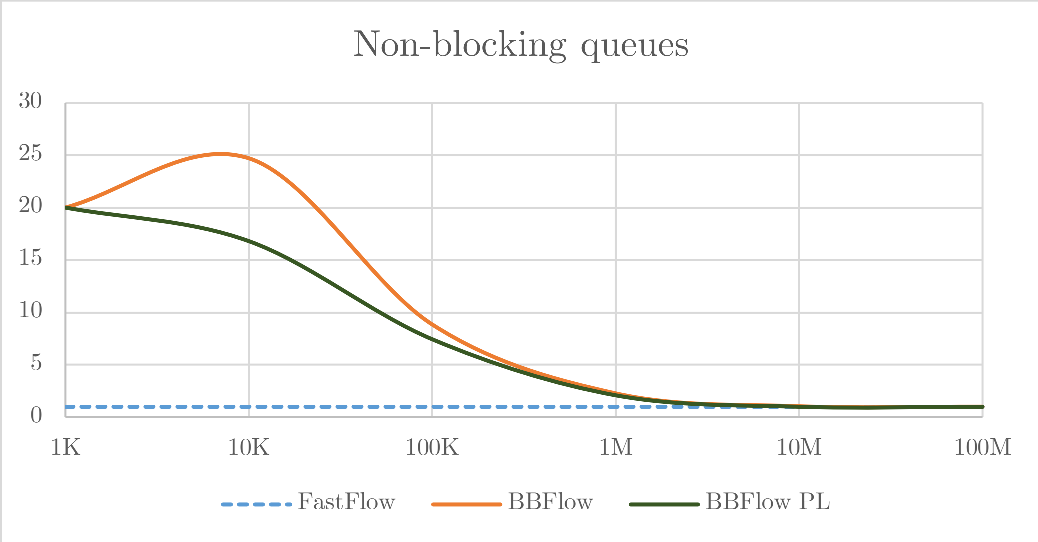


Figure 10: *BBFlow non-blocking queue* performance compared with *FastFlow*.

In the chart in *Figure 10*, the results are clearer and it is easier to see how the *BBFlow* behavior is. When the *preloader* is used, the curve become more linear; therefore, using the *preloader* could be a good idea on particularly short execution time applications.

The impact of the non-primitive *Long* object seems to be negligible on this benchmark, at least on the long run.

The *non-blocking Java* implementation should be anyway improved if small amount of elements are present in the queues, because this is a common scenario where the *consumer* is fast enough to consume the data or the amount of items in the network are less than few thousands. The transmission time on the queue of *20* times more, is too much even considering the many programming language drawbacks.

***Blocking* queue**

In the *blocking* queue, the *BBFlow* performances are completely different from the *non-blocking* ones *(Table 1)*. Instead, in *FastFlow*, the differences between the two queues (*blocking* and *non-blocking*) are less noticeable even if, from the results of benchmarks executed on *FastFlow,* unexpectedly emerged that *blocking* queues are faster than *non-blocking* ones.

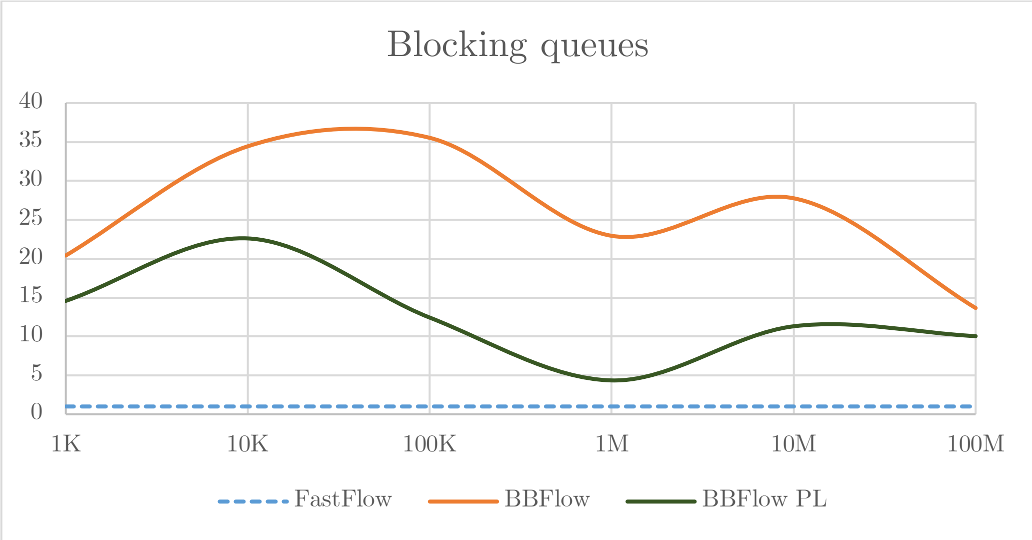


Figure 11: *BBFlow blocking* queue performance compared with *FastFlow*.

In the *blocking* queues, preloading the classes makes a huge difference in the execution time reducing it up to 50%.

As shown in the *Figure 11*, now the *BBFlow* behavior seems to be more casual and after *1M* of elements sent over the queue, there is a slowdown that makes the distance between *FastFlow* and *BBFlow* wider. In this case, the cause of the problem is the *LinkedList* class. The *LinkedList Java* list performs worse [19] if there are many elements in it as explained in *Chapter 4*.

As a further improvement, we can consider reimplementing the *LinkedBlockingQueue* or use an alternative (a class present in *JCF* or a third-party library).

Farm

The *Farm* is one of the fundamental building blocks of *FastFlow* and therefore we compared performances of the *farm* using *BBFlow* versus those achieved using regular *FastFlow*.

The benchmarks related to the *farm* building block, considers a basic *farm* topology composed by an *Emitter*, *N workers* and a *Collector (Figure 12)*.

The *Emitter* generates a stream of items (numbers) that are sent to the *Workers*. Once a *Worker* finishes its task, the result of the computation is sent to the *Collector*.

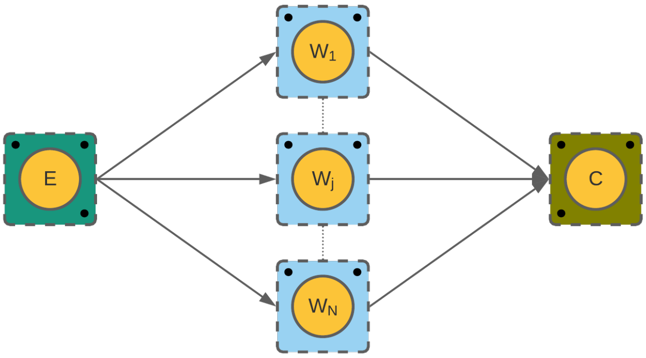


Figure 12: *Farm* topology used in the benchmark

For both programming languages, *Java* and *C++*, we realized a sequential version of the synthetic computation in order to measure the *Tseq* needed to eventually compute the *Speedup* of each version.

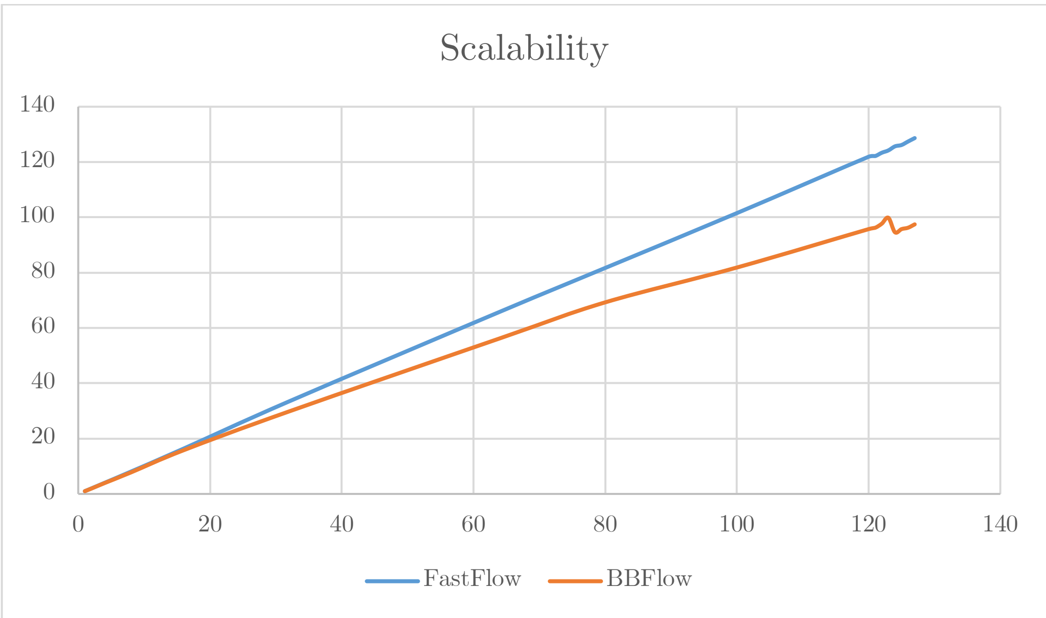


Figure 13: *Scalability* comparison of *Farm* between *BBFlow* and *FastFlow*.

From *scalability* point of view, as shown in the *Figure 13*, *BBFlow* has a good behavior even if the performances decrease while the number of *workers* approaching the number of cores present on the machine. The maximum *scalability* value reached using *BBFlow* is 100 with 123 *workers* and 121 with 127 *workers* using *FastFlow*.

Considering the amount of overheads added by the *Java* language, the results seems satisfactory and in line with expected performances. There are many empirical comparisons between *Java* and *C++* language demonstrating that *Java* is slower of *C++* in sequential and parallel applications [20] [21] [22].

During these benchmarks, a problem discussed in previous chapter raised again, the usage of *non-primitive* objects. In this test *farm*, in *BBFlow* implementation, the items exchanged on the queue between *Emitter/Workers* and *Workers/Collector* are of type *Long*. The active task done by the *Workers* consists in computing a multiplication and a division on the received number one million of times. Working directly on the *Long* number received, the execution time increased drastically making *BBFlow* version incomparable with *FastFlow*. The *efficiency* was worse of at least 40%. As a workaround, the received data on the *worker* is casted to long *primitive* and the mathematical operations are done on it *(Code 11)*. It is casted again by *Java* when returned.

```
defaultWorker<Long, Long> Worker1 = new defaultWorker<>() {
    public Long runJob(Long x) {
        long y = x;
        for (int i=0; i<1000000; i++) {
            y *= 1000;
            y /= 999;
        }
        return y;
    }
};
```

Code 11: *Worker* task where received number is casted to *long primitive*.

The access to the value of a *non-primitive* variable has a cost and if many accesses should be done on it, like in this case, a cast to a *primitive* type could reduce drastically the execution time. The access cost derives from the fact that the value of a *non-primitive* variable, that is an *Object*, is inside and must be accessed every time the value is needed. Accessing a *non-primitive* variable, that is a class with a *primitive* variable inside, millions of times will add a huge overhead to the computation.

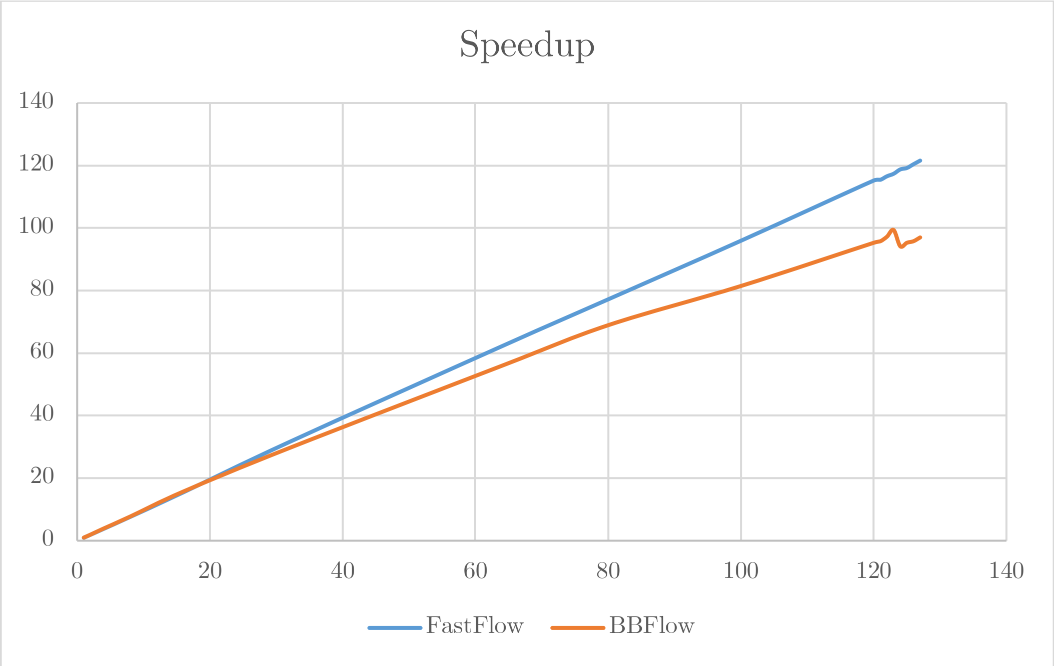


Figure 14: *Speedup* comparison of *Farm* between *BBFlow* and *FastFlow*.

In the *Speedup* chart *(Figure 14)*, we compared parallel version with sequential one and we obtained similar results saw in the *Scalability* chart. The maximum *speedup* reached by *BBFlow* is 100 with 123 *workers*. The test application, theoretically, is completely scalable, but the serial part and overheads introduced by the *Java*, limits the maximum *Speedup* limit that is calculable using the *Amdahl's law* [23].

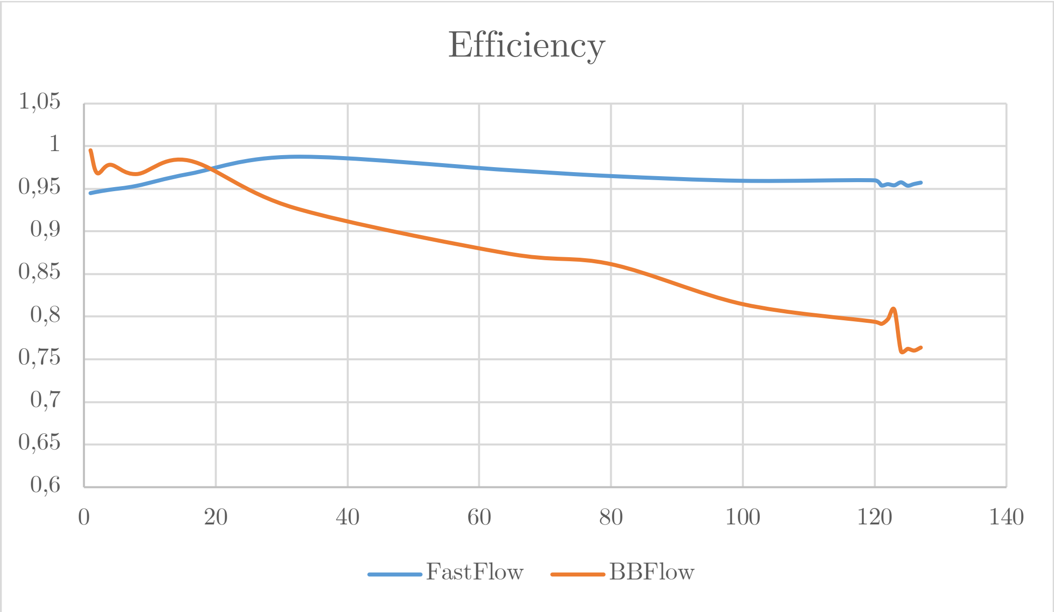


Figure 15: *Efficiency* comparison of *Farm* between *BBFlow* and *Fastflow*.

From the *Efficiency* chart in *Figure 15,* it is clear how performance of *BBFlow* degrades while increasing the number of threads with respect to *FastFlow* that remains almost stable. At 123 cores, where is the maximum *speedup*, the *efficiency* of the *Java* implementation is of about 0.81. The *FastFlow* one is 0.95.

Interestingly, the *Efficiency* of *BBFlow* is greater than the *FastFlow* one until 16 cores reached. After that, *FastFlow* becomes more efficient as seen in the chart.

Pipeline

Another important part of *FastFlow* that should be compared with *BBFlow* is the *Pipeline* building block.

The *pipeline* connects a set of *nodes* using shared memory or network channels where each *node* (that can be any building block) runs on its own. Each *node* receives data from one or more *nodes* and send data to other *nodes*.

In the *pipeline* benchmark, we test a *pipeline* composed only by *single-input/single-output nodes* where each of them will compute, for a good amount of time, arithmetic computations on received value. Each *node* will run on a different thread and the number of *nodes* will be *N+2* (including also *Generator* and *Collector*).

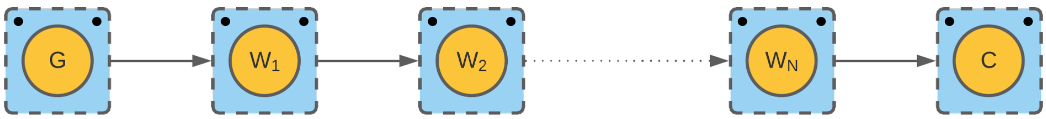


Figure 16: *Pipeline* topology used in this benchmark.

The *pipeline* test includes a *generator* and a *collector node* before first and after last *worker (Figure 16)*. The two more *nodes* run on their own thread like *Emitter* and *Collector* does in the *Farm*. Anyway, they do not do any task apart sending and receiving data from the queues.

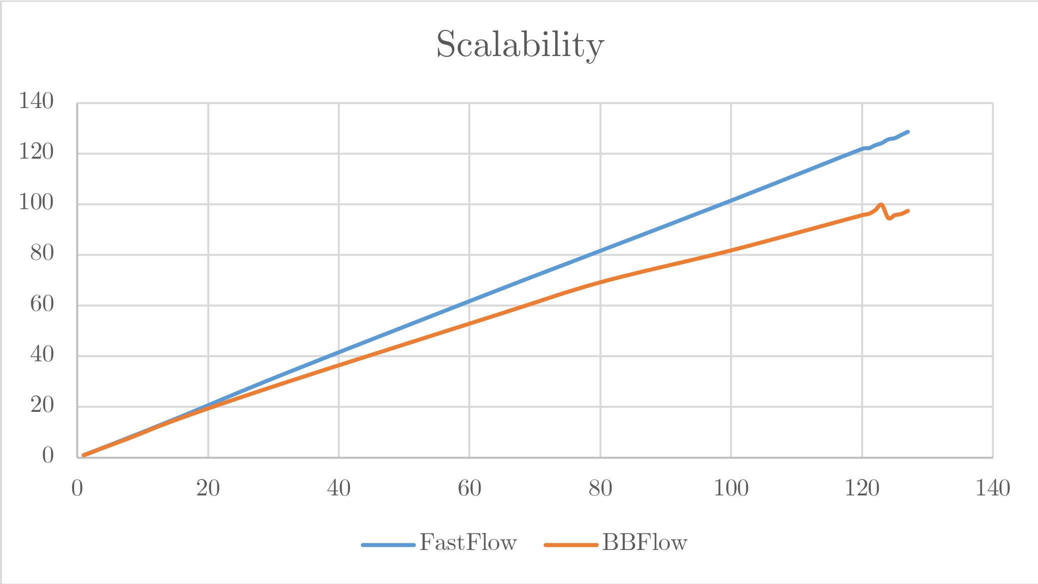


Figure 17: *Scalability* comparison of *Pipeline* between *BBFlow* and *FastFlow*.

Both results of *scalability (Figure 17)* and *speedup* (*Figure 18*) are similar to the *Farm*. The main gain is on *Emitter* and *Collector,* now absent. Now we have a *Generator* and a *Collector* that are single-output and single-input respectively. There is no more waiting time by *Worker* and *Collector* caused by the scanning of other input channels. The stream of data now crosses straightly the entire *pipeline* and each *node* will wait new task only on the first and unique input channel.

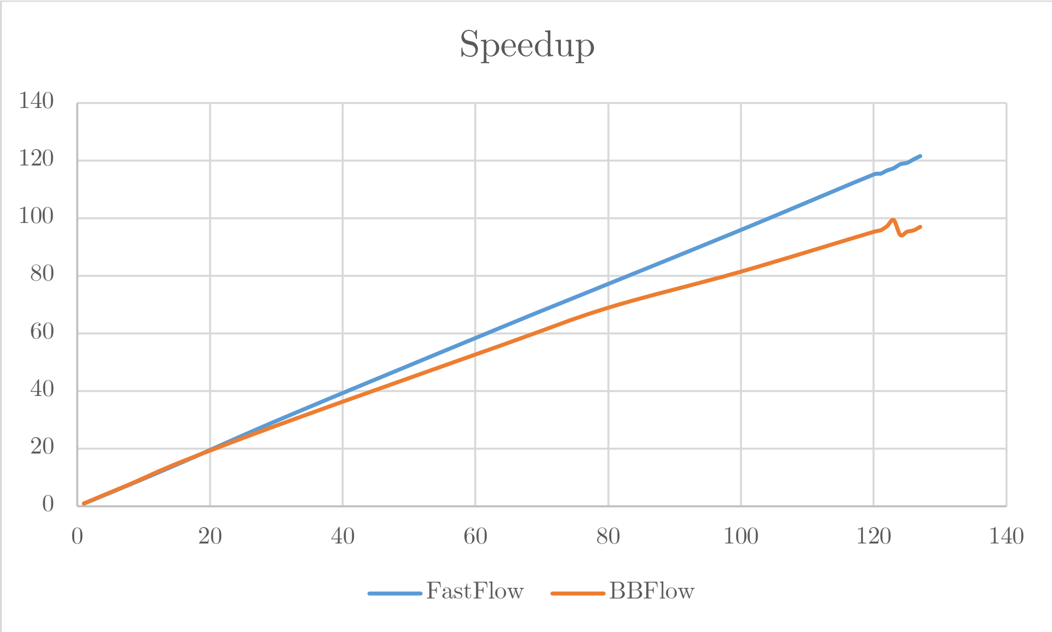


Figure 18: *Speedup* comparison of *Pipeline* between *BBFlow* and *FastFlow*.

The maximum *Scalability* is 100 with 123 *workers* for *BBFlow* and 127 with 127 *workers* for *FastFlow*. Instead, the maximum *Speedup* is 100 with 123 *workers* for *BBFlow* and 121 with 127 *workers* for *FastFlow*.

The results are almost identical to those achieved with the *Farm* topology.

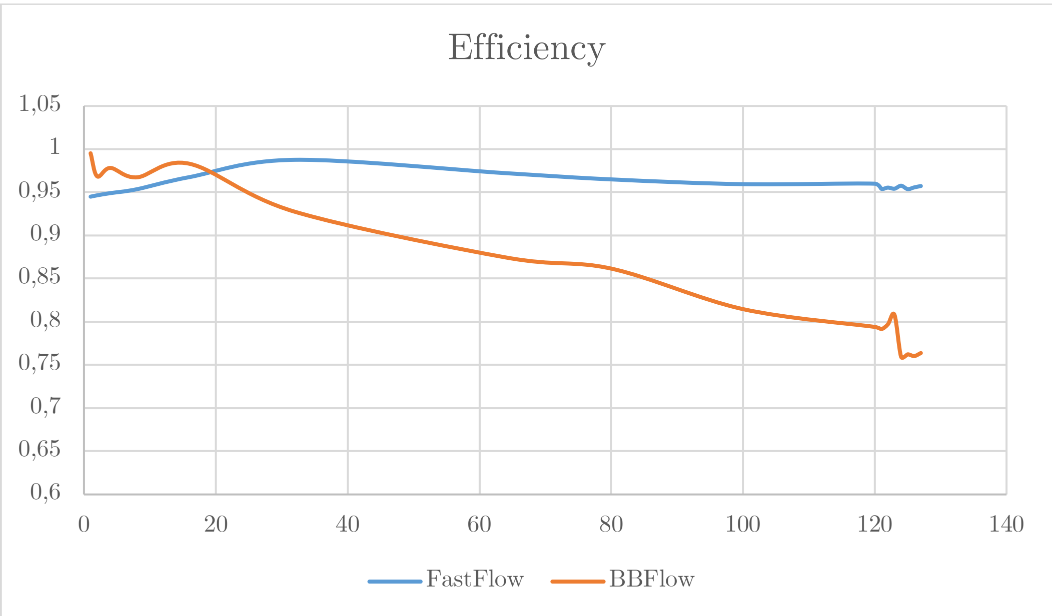


Figure 19: *Efficiency* comparison of *Pipeline* between *BBFlow* and *FastFlow*.

In addition, the *efficiency* chart *(Figure 19)* shows the same kind of behavior seen with *Farm*. *BBFlow* reacts better with less cores and after 16 *workers,* the *efficiency* degrades.

Network queues

In *BBFlow*, we introduced a new type of channel that is a network channel exploiting *TCP/IP* protocol. The channel is of type *non-blocking* and *unbounded* and is constructed on top of a client-server paradigm in a single-threaded context. Every channel has its own server that listen on a unique port. More information can be found in *Chapter 4.*

The same benchmarks done for the shared memory queues have been run using the network channels.

Again, as synthetic computation, we propose a two *nodes pipeline (Figure 20)* where this time the *nodes* are interconnected using the *TCP channel*.

Figure 20: *Producer* and *Consumer* in *pipeline* interconnected by a *TCP channel*

The benchmark is executed running the computation with different batches of data. The *Producer* sends items of type long to the *Consumer* a number of times between *1K* and *100M*.

This *Pipeline* is tested on two different network setups:

- The two *nodes* are on the same machine and they are interconnected through localhost interface. The workstation used in this case is the one with the two AMD EPYC™ 7551 (detailed specifications in the *Benchmarks* section).
- The two *nodes* are on different machines connected through a Gigabit Ethernet switch. Both workstations are with AMD Ryzen™ 7 5800X CPU (detailed specifications in *Chapter 5/Benchmarks*).

The results obtained are compared with the *BBFlow non-blocking* queue to see how the network implementation affects the performances.

Execution time (ms)	**1K**	**10K**	**100K**	**1M**	**10M**	**100M**
Standard non-blocking	8	17	58	159	715	7034
Unbuffered	140	220	641	4515	45656	485801
Buffered 1ms	55	75	257	1218	14751	125385

Table 2: Execution time comparison between *shared-memory non-blocking* queues and *localhost network queues*

Execution time (ms)	**1K**	**10K**	**100K**	**1M**	**10M**	**100M**
Standard non-blocking	8	17	58	159	715	7034
Unbuffered	140	289	1162	7078	69154	699994
Buffered 1ms	117	146	282	1356	13242	131987

Table 3: Execution time comparison between *shared-memory non-blocking queues* and *Ethernet network queues.*

The initial implementation of the network channel was using the *ObjectOutputStream* class directly without a buffer. Every element sent to the channel was automatically sent over the network.

During benchmarks, we noticed that this was causing a big overhead, both in *Consumer* and *Producer*. For this reason, the implementation changed including a *Buffer* (*BufferedOutputStream*) [24] that flushes the information every certain amount of time *(1 ms)* or if the *EOS* reached. The *Buffered* version decreased the execution times of about *4* times with the *Pipeline* on *localhost* and of more than *5* times on *Ethernet*.

In the *Table 2* and in the *Table 3* there are results of the benchmark executed with *nodes* connected through loopback and Ethernet interfaces. Looking the tables is immediately clear how the physical network link between the two *nodes* affects the transmission. Ethernet network adds another overhead to the computation, but the transmission is substantially proportional. In fact, in *Figure 21* we can see how the buffered network queues takes almost the same amount of time to transfer the same amount of data.

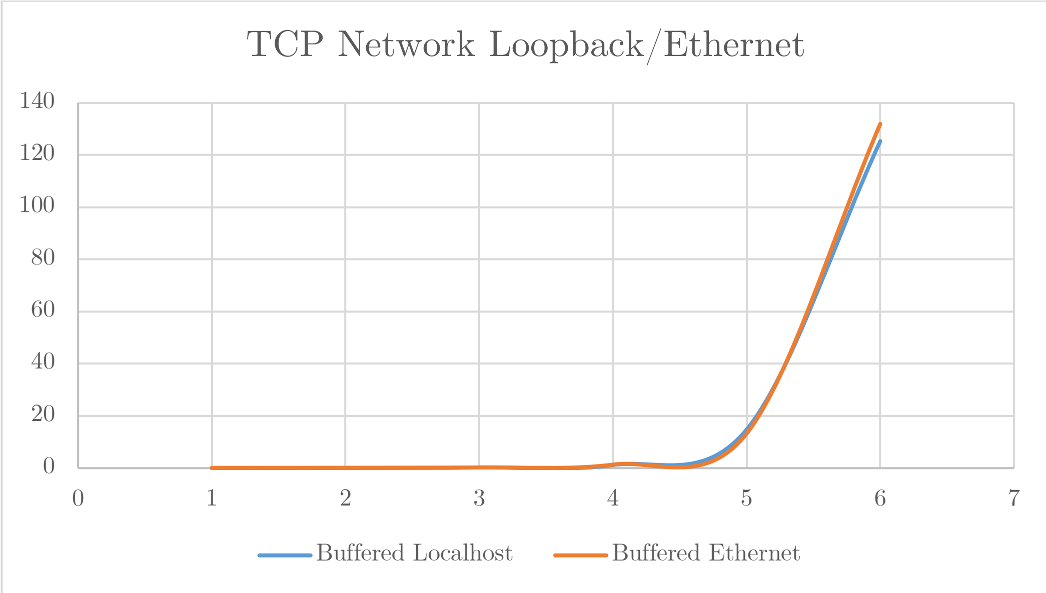


Figure 21: Two-stages *Pipeline* execution time (seconds) on *Loopback* and *Ethernet*

In the *Figure 22,* the chart shows the performances of the *network queues* (buffered and unbuffered) compared to the *shared-memory* ones.

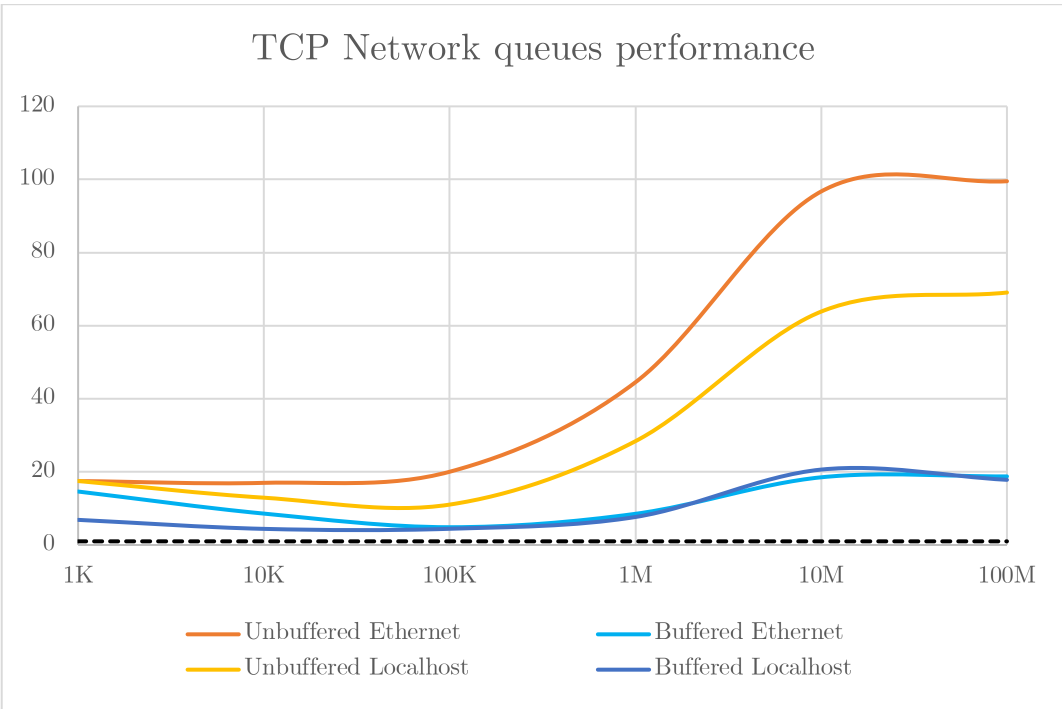


Figure 22: Performance comparison chart between network queues and shared-memory queues

Looking at it, it is immediately clear how the *Buffer* reduces the execution time and why we decided to implement it. Therefore, from this moment, we consider the buffered network queues for all other network comparisons present in this thesis.

The execution time of the *buffered network channel* is *5* times slower of the classic *non-blocking queue* up to *1M* of elements. After that, the performance starts to drop reaching a *20* times slower channel.

The main reasons of the general degraded performances are two: one obvious is the network management. Sending and receiving data implies synchronization between client and server with the addition of the all TCP/IP stack overhead [25]. The second reason is the serialization. All the elements are serialized using *Java ObjectOutputStream* and deserialized using *ObjectInputStream* [26]. This process of *non-primitive* serialization makes the entire process much slower.

The frequency of flushing can be changed modifying the variable *objectClient.flushThreshold* (*nanoseconds*) that has a value of *1ms* by default.

In case the user needs, there is also the possibility to disable the channel buffering using *bb_settings.bufferedTCP* (set it to *False*).

Distributed FastFlow

Recently, in the *FastFlow* project, the support to the network channels was introduced [4]. The channels are *TCP* based and exploit *Object* serialization like *BBFlow*. The serialization in *FastFlow* is done using a *C++* library called *Cereal* [27]. The serialized data are exchanged through *TCP* channels.

A comparison between *Distributed FastFlow* and *BBFlow* was done highlighting different interesting results. The channels are tested again using two *nodes* in *pipeline* interconnected by a *TCP* channel through *Gigabit Ethernet* interfaces. Different of amount of items are transferred (between 1K to 100M) to see how the two implementations perform.

Execution time (ms)	**1K**	**10K**	**100K**	**1M**	**10M**	**100M**
BBFlow TCP Buffered	117	146	282	1356	13242	131987
FastFlow TCP	7	29	385	4078	40598	403029

Table 4: Table of execution time (*ms*) of *BBFlow/FastFlow TCP channels* on *pipeline*

Looking at the *Table 4*, we can notice again, on the *Distributed FastFlow,* the programming language advantage. On small execution time, the overhead introduced by the *JVM* decreases *BBFlow* performances respect to *FastFlow*.

Things start to change when the impact of the *JVM* starts to weigh less on the execution time. In fact, from *100K* elements (*282ms*) the situation reverses. *FastFlow* starts to be slower and the execution time became *3* times the *BBFlow* one *(Figure 23)*. In practice, if the computation lasts more than few hundreds of *milliseconds*, network queues of *BBFlow* are significantly faster.

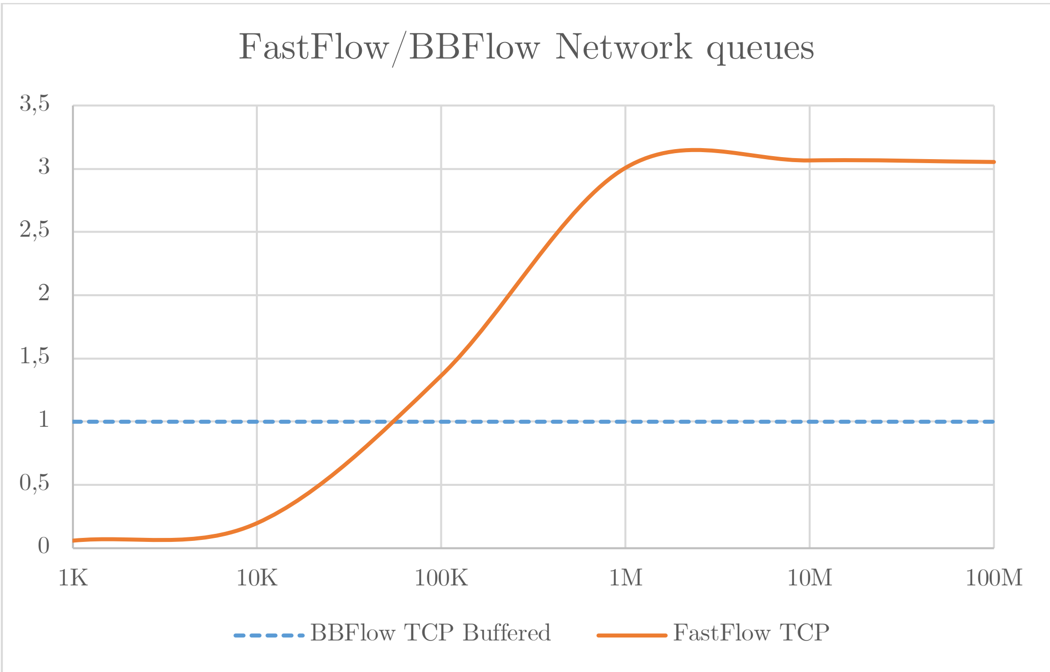


Figure 23: Comparison chart between *FastFlow* and *BBFlow* network queues

There are different reasons for these results. The *Java* programming language abstracts completely the network interface and the protocols and it was optimized through many years to perform as better as possible. In addition, using the *BBFlow* library, the buffering mechanisms and the periodic flushing leaves the time to the thread involved to do other tasks and reduce the overhead added by the network transmission.

The impact of the implementation of the *TCP* channels on *FastFlow* is bigger than expected. In fact, the shared-memory *non-blocking* queues were performing better in *FastFlow*; *BBFlow* was reaching the same performances, without surpassing *FastFlow*, only after *10M* elements transmitted on the *pipeline*. Now, instead, the network implementation makes *FastFlow* performs worse from *100K* elements transmitted.

One other reason affecting *Distributed FastFlow* performances could be the *Cereal* serialization library adding overhead to the transmission, but further investigation is necessary.

Farm with network queues

Another interesting comparison between the two implementations is the *Farm* building block where *Emitter*, *Workers* and *Collector* are interconnected using network channels (*Figure 24*).

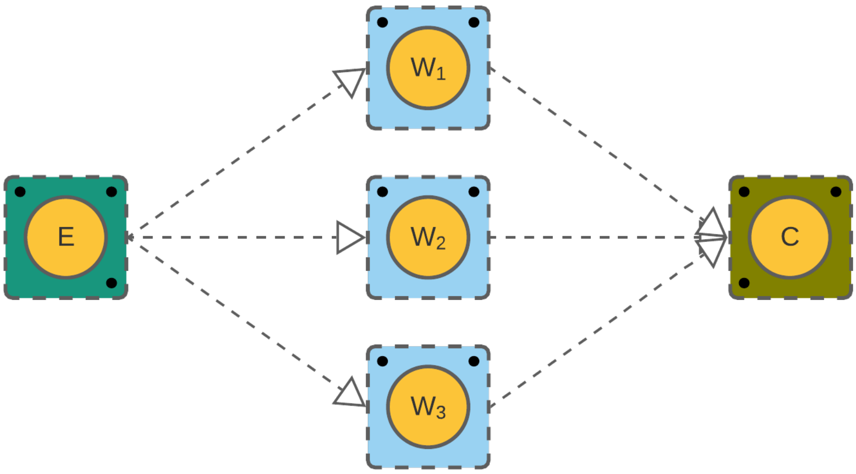


Figure 24: *Farm* topology with three *workers* and *network channels.*

The same data batches are tested as before. In this case, we are sending items through the network from the *Emitter* up to the *Collector*. The benchmarks are executed again on different machines connected through a *Gigabit Ethernet* (detailed specifications in *Chapter 5/Benchmarks*). The *Emitter* and the *Collector nodes* run on the first machine while the *Worker nodes* run on the second one.

```
struct Node3: ff_node_t<myTask_t>{
    myTask_t* svc(myTask_t* t){
      t->num  += get_my_id();
        return t;
    }
};
```

```
Worker = new defaultWorker<>() {
    public Long runJob(Long x) {
        x += id;
        return x;
    }
};
```

Code 12: *C++/Java Worker* computation in *Farm* with *network channels.*

Both versions, *Java* and *C++*, have three *workers* and they do a simple sum operation on the element received before sending it to the *Collector (Code 12)*. Both *Emitter* and *Collector* are configured in *Roundrobin*.

Execution time (ms)	1K	10K	100K	1M	10M	100M
BBFlow TCP Buffered	51	147	300	1205	13266	150619
FastFlow TCP	30	39	278	4977	53443	533750

Table 5: Execution time (*ms*) of a *Farm* with *network channels*

From the results shown in the *Table 5*, the *JVM* still adds some overhead to the *Java* execution and it is noticeable especially on short execution time. Anyway, like other benchmarks, on longer executions the overhead impact is amortized during time.

The results on *BBFlow Farm* are almost identical to the one of the two *nodes pipeline*. That is because the time spent to send elements from *Emitter* to the *Workers* overlaps with the transmission time from *Workers* to the *Collector*. The main reason is because the *BBFlow* channels are single-threaded and every channel has its own connection in a different thread.

Therefore, meanwhile the *Worker* receives the items, the thread managing connection with the *Collector* continues to send elements.

Another point in favor of *BBFlow* is the *Buffering*. Being both reading and sending operations buffered and the flushing periodic, the elements are sent and received in bulk.

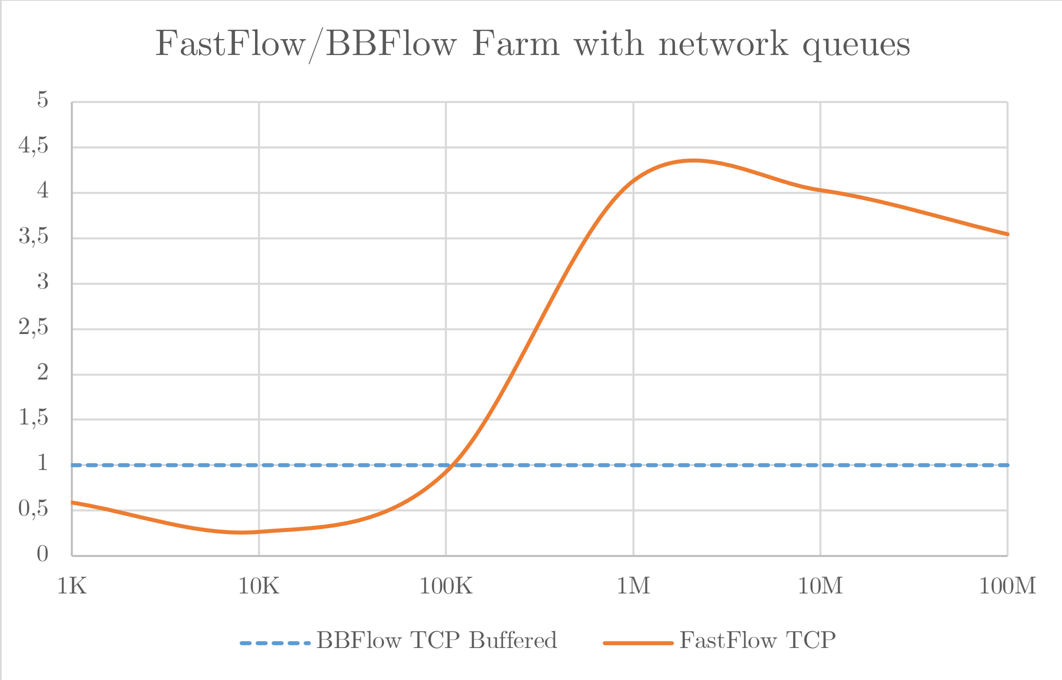


Figure 25: Comparison chart between *FastFlow* and *BBFlow Farm* with *network queues*

The *Figure 25* shows the performance using the *FastFlow* library respect to the *BBFlow one*. It is clear that *FastFlow* is about three/four times slower. At the beginning, there is still the *JVM* overhead that weights on the performances, but after few hundreds of milliseconds, its impact became negligible.

Overall performances

The performance results obtained from the previous analysis confirm our expectations. The *FastFlow* is a project thought and developed across multiple years and its degree of optimization is amazing. The presented *BBFlow Java* project uses only built-in components and data structure already existing in the *JDK*. In addition, there is not any low-level memory management and it does not take into account the processor caching. The *BBFlow* exploits all possible abstractions given by the *Java* programming language and it is obvious that cannot reach the same degree of tuning of *FastFlow* without a deeper customization.

The different benchmarks on shared memory of *Queues*, *Farm* and *Pipeline* show that there is a constant overhead added by the *JVM* and by the internal memory management. This overhead affects the global *efficiency* and reduces the *scalability*. Despite of the lower *efficiency*, the *BBFlow Java* implementation demonstrates to have good potentialities and with a better fine-tuning, *BBFlow* could reach much better performances.

The situation is the reversed when using the network queues. The *BBFlow* project is few times more efficient than the *Distributed FastFlow* (*D-FastFlow*) implementation. The main reason is that the *D-FastFlow* is a quite recent extension of *FastFlow* and it is not yet fine-tuned as it should. The *Java* network implementation developed and improved across many decades, shows to be more efficient.

There are different improvements that can be applied to the *BBFlow* project and many new *Java* features that can be exploited. In the *Future works* section, there are some ideas for further improvements.

Chapter 6

Use case

In the *Chapter 5,* we shown the performances of many synthetic computations using the *BBFlow* library and we compared them with the same synthetic computations using the *FastFlow* library. In this chapter, we try to confirm the *BBFlow efficiency* results in real life applications. For this reason, we developed a sample application exploiting a custom parallel pattern that can mimic a common scenario.

Self-Organizing Map

The application realized, is an implementation of a Neural Network called Self Organizing Map *(SOM)* [28].

Both parallel and sequential implementations are present in the *BBFlow* repository.

Briefly, the *SOM* consists in a matrix of *nodes* (*neurons*). Each *node* is associated with an *n*-dimensional vector (*weight vector*) of numbers. Each *weight vector* stores information learned during the training process.

In the learning phase, a set of *input vectors* (having same dimension of the *weight vectors*) are used to train the neural network. Each *input vector* is compared with each *weight vector* finding the closest one (e.*g*., using Euclidean distance). Once the closest *weight vector* is found (inside the *target neuron*), the neuron involved is trained together with its neighborhood (using a *neighborhood function*) [29]. After the *SOM* learning process is finished, neighboring weight vectors in the matrix contain similar values respect to far ones; so similar data stay close in the matrix giving some sort of automatic clustering.

In the parallel version, we implemented both search and learn phase in the *SOM* matrix.

The search phase, as said before, consists in a comparison between *input vector* and all *weights vectors* of all matrix *nodes*.

A good way to parallelize this operation is to split the *SOM* matrix in multiple parts and do this search in parallel in each piece.

In this way, the size of the matrixes decreases, giving a better possibility to the *CPU* to exploit caching of the data.

Once the *target neuron* is found in each matrix, an external *node* must choose the better one comparing their distance with the *input vector* and finding the correct position.

The learning phase, with the split matrix, works as usual. The only situation that should be managed is when the neighboring function needs to access to neurons in adjacent matrixes. For this reason, a communication channel between adjacent matrixes must be created. In this way, neighborhood can be involved in the learning process.

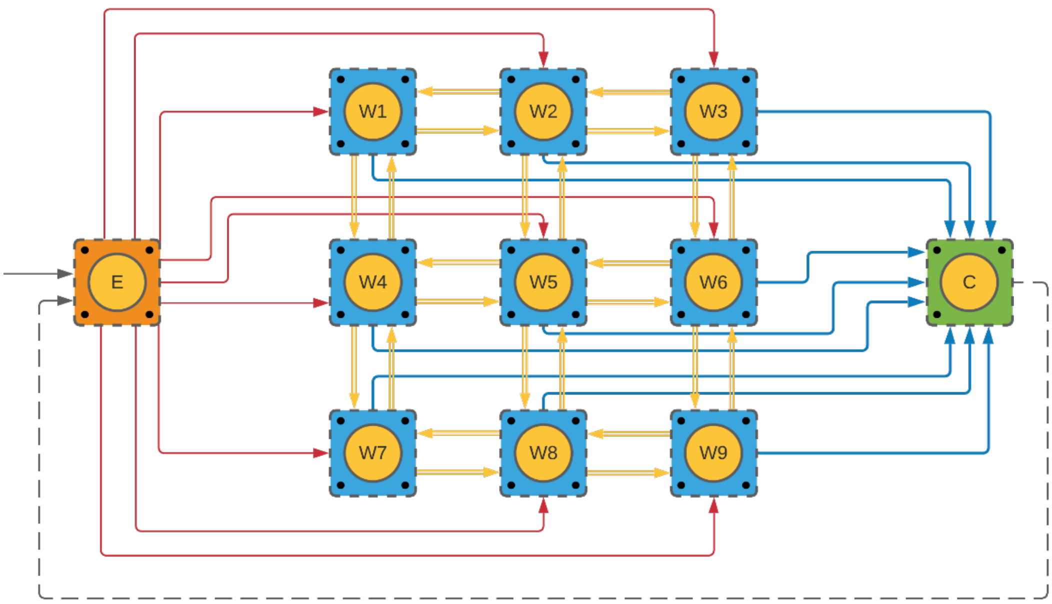


Figure 26: *Self-Organizing Map* topology

A good topology, to represent this parallel version of *SOM*, is to use a modified *Farm* building block where each *worker* contains a part of the *SOM* matrix and the adjacent *workers* are connected each other (as shown in the *Figure 26).* This resulting topology is not a standard building block composition, but a customization made possible by the *BBFlow* flexibility. In fact, the *workers* are manually connected each other exchanging tasks. This is possible thanks to the *BBFlow* library allowing manual creation of the channels between *nodes*. In addition, the custom *runJob* method, declared in a standard class, allows manual channels management, giving a high degree of freedom. We have chosen to present this sample application, using a non-standard building block composition, because I have a wide experience with this pattern and I developed a more complex version of it in *C++* for my current job. Therefore, this topology takes inspiration from a real neural network implementation actually used in the company I am working for. The final idea is to rebuild and replace the company's *SOM* implementation with a new one using the *BBFlow* library.

The *Emitter* has the role to assign the task to the target *worker(s)* (search, learn, etc.) and the *Collector* aggregates the data received by the *Workers* deciding what to do.

At the beginning, all *nodes* in the *farm* are waiting for a command. Once the *emitter* receives from the first input channel a new task to do, communicates it to the *Workers*.

In case of a search task, the *emitter* sends to all *workers* to search for the input vector. The *Emitter*, once it assigned a task to the *Workers*, does not receive any other task leaving them in the queue. This is done to avoid overlapping between different tasks, especially during learning phases.

Once the *workers* finish its search task, they send the result (*weight vectors* found in the *target neuron*) to the *Collector* that will compare all weight vectors received choosing the best position found in the matrix.

The *Collector's* result is sent to the *Emitter* as an *ACK*, using a *feedback channel* interconnecting *Collector* with the *Emitter* that now can proceed to execute new tasks.

Instead, in case of a learn task, the *Emitter* sends the learning command only to the involved *workers*, where the *target neuron* is. *Emitter* sends also a special command to the *target neuron's neighborhood* to inform them to listen on their all input channels. This is done to prepare adjacent *workers* to receive an eventual learning task from the target *worker*.

This special command is important to avoid that all *workers* are listening all time on all input channels. The number of input channels of each *worker* are maximum 1+4, where the first one is the command channel, where *workers* receive new tasks from *Emitter* and the others are the channels connecting them with adjacent *workers*.

On the same way, each *worker* has maximum *1+4* output channels, one with the *Collector* and others with adjacent *workers*.

In this way, almost all the time, the *workers* listen only on a single channel (the command one) without polling or waiting on multiple channels.

When a *worker* receives a task to execute, e.*g*., the learning one, an *ACK* is sent to the requester to inform it the task is completed.

All the tests on this *SOM Farm* are done using a matrix of size about *1024x1024* with *double* weight vectors of size *3*, representing *RGB* data between 0 and 255.

The total memory size of the matrix is of 24MB and will be split in *N* parts of size *MxM*.

The matrix must be divisible by *MxM* so the sizes of the test matrix will be between *1023x1023* and *1030x1030* to have the possibility to test with all possible *M* and reach the target number of *workers* running, equal about to the number of cores.

All values in the tests are random, both the *SOM* matrix and input vectors.

While the learning task of this parallel application cannot scale because it involves only a few vectors and matrixes, the search task can be parallelized being done on the entire matrix. The *workers* during the search phase are completely independent.

The data exchanged inside the *Farm* are of type *SOMData*, special packets that include instructions, results, and eventual *forward instruction*. The *forward instructions* are used to route a packet to another *worker* (e.g., during learning phase of adjacent *workers*).

```
case SOMData.LEARN_NEIGHBOURS:
    if (element.redirect != null) {
        int t = element.redirect;
        element.redirect = null;

        if (out.get(t) != null) {
            //System.out.println("Redirecting training vector");
            element.replyredirect = position;
            sendOutTo(element, t);
        } else {
            // no SOM to train, reply with ACK
            SOMData rd = new SOMData(SOMData.LEARN_FINISHED, id);
            rd.communicationType = SOMData.LISTEN_NEIGHBOURS;
            if (MSOM.DEBUG) rd.debugString = "ACK";
            sendOutTo(rd, position);
        }
```

Code 13: Redirecting a received packet if the target *node* exists

The *forward instructions* are necessary to reach *workers* not directly connected with the requester (example code of *SOMData* packet routing in *Code 13*). For example, if a *worker* is executes the *learn* task on position *[0,0]* of its matrix, *TOP* and *LEFT workers* will be contacted, but also *TOP-LEFT worker* must be involved (for position *[-1,-1]* for example). So, the packet intended to the *TOP-LEFT*, is sent to *TOP worker* that will redirect to its *LEFT*. In addition, replies sent by a *worker* will be routed to the requestor.

The execution of the *Farm* terminates when the *Emitter* receives an *EOS* command that will be propagated across the entire network.

Performances

The performances of this *SOM* implementation are measured executing a search and learn task over the same input vector, *100k* times. The *input vector*, *weight vector* and *SOM* matrixes are of type *double (primitive)*. The usage of primitive variables is possible because both vectors are exchanged through the queues inside the *SOMData* packet object.

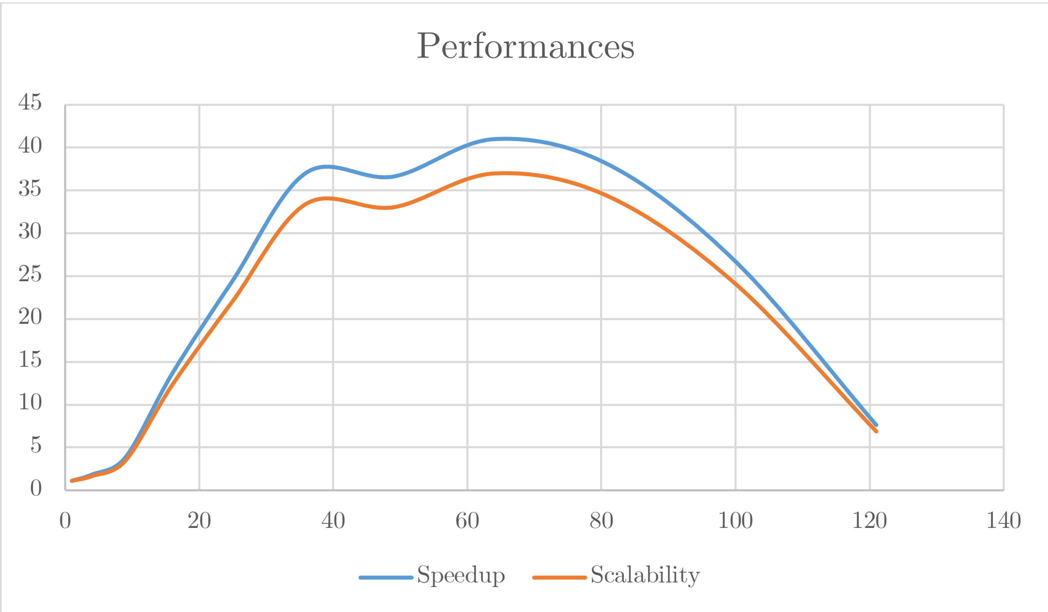


Figure 27: *Speedup* and *Scalability* of the multi-core *SOM* implementation

The *scalability* of the application is good until 36 *workers* with a value of 34. After that, the *efficiency* starts to decrease and *scalability* touches the *maximum* value of 36 with 64 *workers*. The same thing happens with the *speedup* reaching its maximum value of 41 with 64 *workers (Figure 27)*.

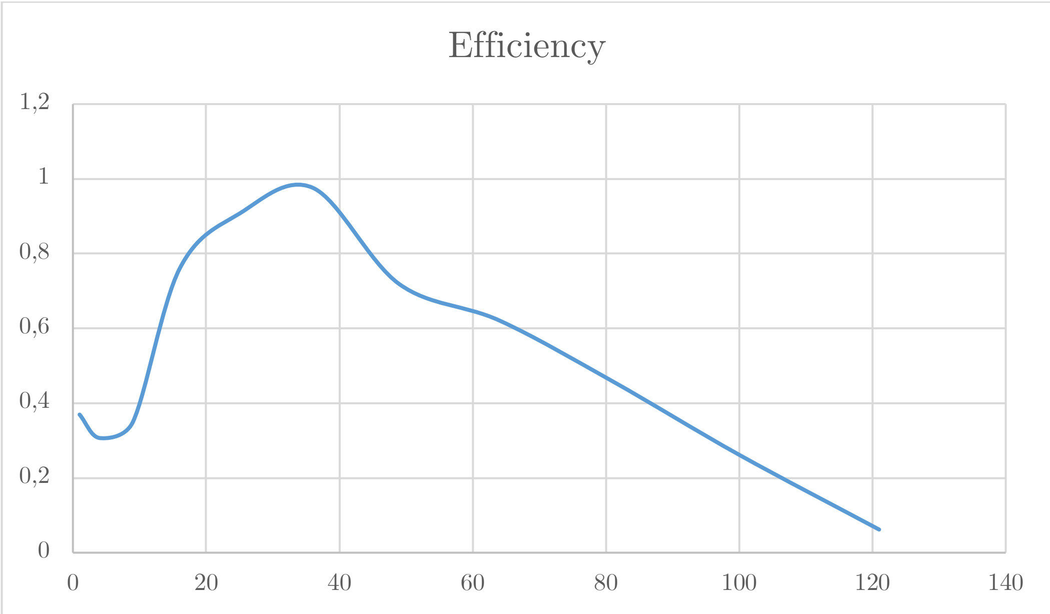


Figure 28: *Efficiency* chart of the multi-core *SOM implementation.*

From the *efficiency* chart *(Figure 28)* is clear that with 36 *workers* (38 threads including *Emitter* and *Collector*) we reach the maximum *efficiency* touching 0.976.

At the beginning, the complexity of the implementation affects the execution time leading to an *efficiency* less than 0.4. When the number of *workers* increase, the overhead added by our *SOM* topology became negligible and the application starts to scale up pretty well.

After 40 cores, the performance decreases. One reason could be the *collector*, a single centralization point that starts to be a bottleneck and leads to an increase of the execution time.

The *efficiency* chart confirms the results obtained with synthetic computations; in fact, its shape resembles the *efficiency* charts seen in the previous chapter.

To reach these performances, we needed to add many optimizations in all *nodes* of the application. The major optimizations are:

- When the *Collector* waiting search results, it listens only on channels where it has not yet received the result. Therefore, meanwhile *Collector* receives search results from the *workers*, the number of the channels where the *Collector* listens is reduced.
- *Workers* read only from the channel with the *Emitter*. They start to check other channels only when the *Emitter* informs them that they could be involved in a learning task.

- *Emitter* listens from one channel only even if it has two input channels (an input channel and a *feedback* channel). First, the *Emitter* waits for a command from the input channel. When received, it informs the *Workers* and starts listening to the *feedback* channel only. When a feedback received, the cycle starts again.

Many other improvements can be applied on the *SOM* implementation. The main one consists in reducing the amount of data exchanged between *nodes*. For example, the number of *ACKs* sent between *Workers*, *Collector* and *Emitter* are for sure responsible of some performance degradation.

The *Z Garbage Collector*

From the *Java* version *15*, a new *garbage collector* called *Z Garbage Collector* (also known as *ZGC*) is present in the *JVM*. At the beginning, it was experimental and not advised for production uses. Recently, the new *garbage collector* was officially released replacing the actually deprecated *CMS (Concurrent Mark Sweep) garbage collector* [30].

The *ZGC* is a scalable low latency *garbage collector* designed to meet the following goals:

- *Pause times* are of the sub-millisecond order.
- *Pause times* do not increase with the *heap*, *live-set (variables/objects* left in the *heap* after a *garbage collection)* or *root-set* size *(local variables/objects* and *static variables/objects).*
- Handle *heaps* ranging from *MB* to *16TB* in size.

Pause times are small intervals where the user application is suspended. This phase is called *Stop-The-World (STW)* and it is started by the *garbage collector* when it needs to do operations on the *heap* that can interfere with the normal execution. In the *ZGC*, this *STW* phase is reduced to the bare minimum and it does not depend on the *heap* size.

These characteristics make *ZGC* a good fit for server applications, where large *heaps* are common, and fast application response times are a requirement.

The *ZGC* is not the default *Java garbage collector* and can be enabled with a specific command line argument *(-XX:+UseZGC).* More details can be found in the *BBFlow Manual*.

The Self-Organizing Map sample is a good example that can require the use of *The Z Garbage Collector*. In fact, our *SOM* implementation manages big *heaps* and requires fast response times.

For this reason, we decided to run again the previous benchmarks on the *SOM* with the *ZGC* enabled and compare the results with the ones without *ZGC*.

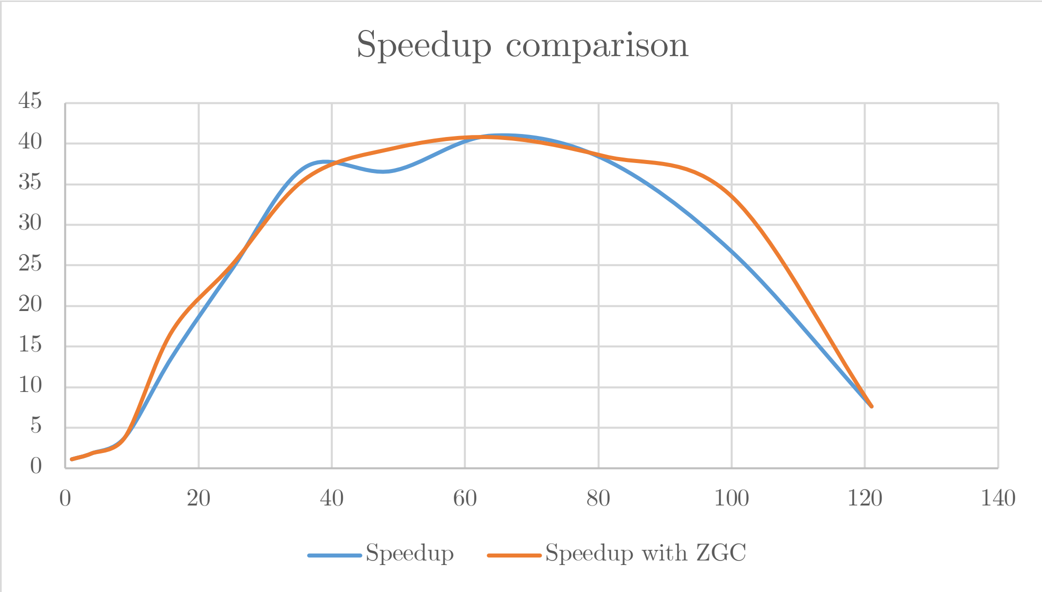


Figure 29: *Speedup* comparison between *SOM* running on a *JVM* with or without *ZGC* enabled.

The *speedup* comparison (shown in *Figure 29*) between *SOM* with or without *ZGC* shows that with the *ZGC* enabled the curve is smoother and the *speedup* remains almost constant for longer and its value starts to decrease later (around 95 *workers*).

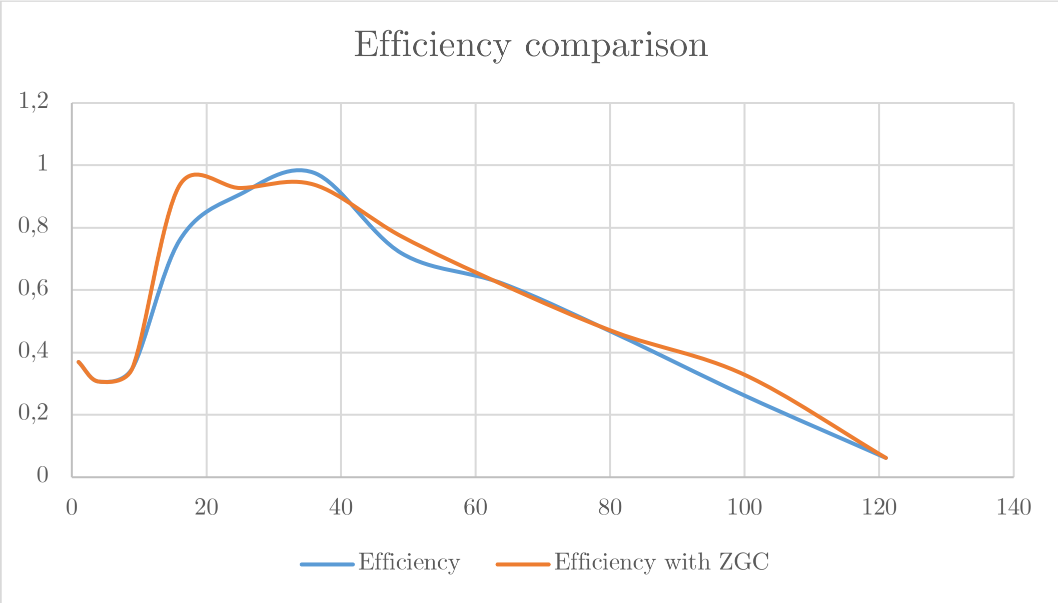


Figure 30: *Efficiency* comparison between *SOM* running on a *JVM* with or without *ZGC* enabled.

From the *Efficiency* chart *(in Figure 30)* is clearer the *ZGC* behavior. Our *SOM* implementation creates a number of sub-matrixes that is dependent on the number of *workers*. When the number of *workers* are lower, the matrixes are less and bigger. In this case the *ZGC* helps to reduce pauses of the execution (*STW*) increasing the *efficiency* of our implementation earlier. In fact, at about *16 workers*, the *efficiency* value is around *0.93*.

There are not substantial differences after about 30 *workers* between the two benchmarks.

After this benchmark, we can conclude that the *ZGC* can be a good optimization to take into account for particular cases where the amount data in the *heap* is big and the responsiveness is critical.

Conclusion

In this thesis, we describe how we reimplemented the *FastFlow* building blocks using the *Java* programming language. The *FastFlow* building blocks are concurrent components that are the fundamental elements of any structured parallel applications implemented using the *FastFlow* library.

The resulting *Java* implementation, called *BBFlow* and developed using *Java version 17,* is fully working and available open-source.

The building blocks implemented are both sequential and parallel (*Chapter 3*) and they can be interconnected each other with different type of communication channels (shared-memory or network channels; see *Chapter 4*).

The *BBFlow* project was tested during and after its implementation with many synthetic computations. It was also compared with *FastFlow* doing a deep performance analysis *(Chapter 5).*

From the comparison results, emerges that *BBFlow* has slightly reduced performances respect to *FastFlow* in a shared memory context, but better results on a network of workstations environment.

In addition, a sample use case *(Chapter 6)* was developed using *BBFlow* package to show the library versatility and performances on a real application.

This work has been very interesting and it highlights the advantages (pre-existing *Java* classes, portability, etc.) and the disadvantages (*JVM*, non-primitive data types, etc.) of the *Java* programming language in concurrent applications.

Future works

The *BBFlow* project can be extended and improved in several directions. The actual implementation exploits all common *Java* classes (already included in *JDK*) and replacing few of them with custom and fine-tuned new classes can surely improve performances.

Between all the improvement's ideas, the main one is to reimplement the lists of the queue. Actually, we are using the *LinkedList JCF* class for all type of queues. The *Java LinkedList* implementation, *blocking* or *non-*

blocking, suffers with many elements in it and its overhead is high as explained in the *Chapter 5*.

All of the queues, in particular the *non-blocking* one, should be reimplemented following the design present in *FastFlow* or in alternative, using a third-party library or another class present in the *JCF* (like *ArrayList* instead of *LinkedList* as literature documentation advice [19]).

Another important possible improvement emerged during the benchmarks is to limit the use of the *Java Generics*. *Java Lists* use *Generics* (so *Objects*) as elements. The memory access of *Objects* has a completely different cost of accessing a *primitive* variable. We tried two workarounds to overcome this limitation:

- When a *node* receives an item, cast it to a new *primitive* variable and run the task using the new variable *(Code 11).*
- Create a custom *Object* embedding multiple *primitive* items to transmit as shown with *packet labeling (Code 28)* and with *SOMData (Chapter 6, Self-Organizing Map)*.

Both tested workarounds are valid. In the first case, if the number of accesses to the received *item* is high enough to justify the memory allocation of a new variable, the programmer should consider doing the casting. In the second case, we have still the access to an *Object* (in read and write), but the *Object* accessing costs can be amortized embedding multiple items or more complex data structures in it.

Another good improvement that can be add in the *BBFlow* library, that for sure can be powerful for the user, is to add a built-in *Ordered Farm* version with *packet labeling (Code 28)*. Right now, the *Ordered Farm* can be realized in two ways, using *Roundrobin* configuration (on both *Emitter* and *Collector*) or implementing a new *node* after or instead of the *Collector*. This new *node* receives items from *workers* and using the *packet id,* outputs the items ordered.

The *BBFlow project*, during its development, was tested using the *Java JVM version 17* without any particular flag. The *JVM* can be tuned exploiting many different parameters (see in the *Chapter 6,* The *Z Garbage Collector* and the *Java options* in the *Appendix A*) and we tested only few of them during our benchmarks. There are different other options that could be tested on the *JVM* and many of them are described in literature documents [31].

References

[1] M. Aldinucci, M. Danelutto, P. Kilpatrick and M. Torquati, Fastflow: High-Level and Efficient Streaming on Multicore. In Programming multi-core and many-core computing systems, eds S. Pllana and F. Xhafa, 2017.

[2] M. Torquati and M. Danelutto, A RISC Building Block Set for Structured Parallel Programming, 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, doi: 10.1109/PDP.2013.17., 2013.

[3] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick and M. Torquati, Design patterns percolating to parallel programming framework implementation, International Journal of Parallel Programming, 2013. 10.1007/s10766-013-0273-6.

[4] N. Tonci, A Distributed-Memory runtime for FastFlow's building-blocks, Master thesis, University of Pisa, Pisa, IT, 2021. ETD-04222021-171721.

[5] M. Behler, Java Versions and Features, 2021, https://www.marcobehler.com/guides/a-guide-to-java-versions-and-features#_java_features_8_17.

[6] JEP 370: Foreign-Memory Access API (Incubator), 2019, https://openjdk.java.net/jeps/370.

[7] M. Torquati, Harnessing Parallelism in Multi/Many-Cores with Streams and Parallel Patterns, PhD thesis, University of Pisa, Pisa, IT, 2019. ETD-02132019-111752.

[8] M. M. Michael and M. L. Scott, Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms, Proceedings of the Annual ACM Symposium on Principles of Distributed Computing, 1996. 10.1145/248052.248106.

[9] Hadoop database: Apache HBase, a distributed, scalable, big data store. Version 1.1.7 2016.

https://hbase.apache.org/1.1/apidocs/org/apache/hadoop/hbase/util/BoundedConcurrentLinkedQueue.html.

[10] S. D. Stoller, Testing Concurrent Java Programs using Randomized Scheduling, 2002, Electronic Notes in Theoretical Computer Science, 70(4):142-157. 10.1016/S1571-0661(04)80582-6.

[11] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby and S. Ur, Framework for testing multi-threaded Java programs, 2003, Concurrency and Computation: Practice and Experience, 15, pp. 490-498. https://doi.org/10.1002/cpe.654.

[12] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson and M. Wolczko, Compiling Java just in time, IEEE Micro, vol. 17, no. 3, pp. 36-43, May-June 1997, doi: 10.1109/40.591653..

[13] D. Ghosh, Generics in Java and C++ - A Comparative Model, SIGPLAN Notices, vol. 39, 2004.

[14] CPU AMD EPYC™ 7551, https://www.amd.com/en/products/cpu/amd-epyc-7551.

[15] CPU AMD Ryzen™ 7 5800X, https://www.amd.com/en/products/cpu/amd-ryzen-7-5800x.

[16] J. Giacomoni, T. Moseley and a. M. Vachharajani, FastForward for Efficient Pipeline Parallelism: A Cache-Optimized Concurrent Lock-Free Queue, Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP, 2008. 10.1145/1345206.1345215.

[17] A. Morrison and Y. Afek, Fast Concurrent Queues for x86 Processors, ACM SIGPLAN Notices, vol. 48, 2013. 10.1145/2442516.2442527.

[18] S. Arnautov, P. Felber, C. Fetzer and B. Trach, FFQ: A Fast Single-Producer/Multiple-Consumer Concurrent FIFO Queue, IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2017. 10.1109/IPDPS.2017.41.

[19] D. Costa, A. Andrzejak, J. Seboek and D. Lo, Empirical Study of Usage and Performance of Java, 8th ACM/SPEC International

Conference on Performance EngineeringAt: L'aquila, Italy, 2017. 10.1145/3030207.3030221, pp. 389-400.

[20] L. Prechelt, An Empirical Comparison of Seven Programming Languages, Computer, vol. 33, 2000. 10.1109/2.876288.

[21] G. Nikishkov, Y. Nikishkov and V. Savchenko, Comparison of C and Java performance in finite element computations, Vols. Volume 81, Issues 24–25, Computers & Structures, vol. 81, 2003. 10.1016/S0045-7949(03)00301-8, pp. 2401-2408.

[22] A. Shafi, B. Carpenter, M. Baker and A. Hussain, A comparative study of Java and C performance in two large-scale parallel applications, Concurrency and Computation: Practice and Experience, vol. 21, 2009. 10.1002/cpe.1416.

[23] G. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, Proceedings of the April 18-20, 1967, Spring Joint Computer Conference AFIPS'67, vol. 30, 1967.

[24] R. Agarwal, D. Harrington and C. Gusman, Lego Mindstorm NXT Controller with Peer-to-Peer Video Streaming In Android, Journal of Computing Sciences in Colleges. 27. 243-252, p. 247.

[25] M. Pendergast, Performance, overhead, and packetization characteristics of Java application level protocols, ACM SIGITE Research in IT, vol. 8, 2011.

[26] S. Hirano, Y. Yasu and H. Igarashi, Performance evaluation of popular distributed object technologies for Java, Concurrency - Practice and Experience, vol. 10, 1998. 10.1002/(SICI)1096-9128(199809/11)10:11/13<927::AID-CPE390>3.0.CO;2-G.

[27] G. W. Shane and V. Randolph, Cereal, http://uscilab.github.io/cereal/.

[28] T. Kohonen, The self-organizing map, Proceedings of the IEEE, vol. 78, no. 9, pp. 1464-1480, 1990. 10.1109/5.58325.

[29] D. Miljković, Brief Review of Self-Organizing Maps, MIPRO 2017. 10.23919/MIPRO.2017.7973581.

[30] Oracle OpenJDK, The Z Garbage Collector Wiki, https://wiki.openjdk.java.net/display/zgc/Main.

[31] J. Pavelec, Optimization of JVM settings for application performance, Master thesis, Masaryk University, Brno, Czech Republic, 2017.

Appendix A

Manual

In this chapter, we explain how to use the *BBFlow* library in different scenarios. For each parallel and sequential building blocks described in this thesis, we provide one or more examples with the most interesting details. The examples include the usage of both types of communication channels: *shared-memory* and *network*.

Library installation

The *BBFlow Java* project can be found on the *Github* repository *https://github.com/st4ck/BBFlow*.

The project documentation is available at *https://st4ck.github.io/BBFlow* and contains description of classes and methods present in the *BBFlow* packages.

The first step to start to use the *BBFlow* library is to download the project from *Github* (using *git clone* or downloading the archive). In the root directory of the downloaded project, there are many files and directories:

- The *"docs"* directory contains the *Java* documentation of the project, the same one available online on *Github*.
- The *"src"* directory contains all the files needed to use the *BBFlow* library and different examples.
- The *"BBFlow.iml"* file is the project file of the *IntelliJ IDEA IDE*.
- The *"README.md"* file is the readme of the *BBFlow* project shown on Github.
- The *"LICENSE"* file is the license file. The *BBFlow* project is under *GNU General Public License v3.0*.

The *BBFlow* library was developed and tested using *Oracle Java* version 17. In order to use the library, the *Oracle JDK* version at least of version 17 must be installed on the system. The *JDK* comes with the two commands that are fundamental to compile and run *Java* applications: *"javac"* and *"java"*.

The *BBFlow* library comes with two *Java* packages, *bbflow* and *bbflow_network*, which can be included in the user project using *Code 14*.

```java
import bbflow.*;
```

Code 14: *bbflow* package import.

Once the library is included, the user can start to use all the classes and methods provided by the *BBFlow* library.

The user's application can be compiled using "javac" command and executed with "*java*" one. The example in *Figure 31* shows a typical usage.

```
user@h:~/BBFlow-master/src$ javac tests/ff_tests/combine2.java
user@h:~/BBFlow-master/src$ java tests.ff_tests.combine2.java
Worker2 (id=2) in=1
Worker2 (id=3) in=3
Worker2 (id=1) in=2
Worker2 (id=3) in=6
Worker2 (id=2) in=4
Worker2 (id=3) in=9
…
Filter2 received: 99
Filter2 received: 97
Filter2 received: 98
Filter2 received: 100
```

Figure 31: Typical usage of *JDK* to run an application

Library basic usage

The basic entity of each building block in *FastFlow* is the *ff_node*. The class *ff_node* contains a *Thread* that can execute the user's custom code. The custom code must be inside a *defaultJob* class.

```java
defaultJob <Long, Long> Worker1 = new defaultJob<>() {
    public Long runJob(Long x) {
        return x+1;
    }
};

ff_node stage1 = new ff_node(Worker1);
```

Code 15: *Node* creation using *anonymous* class

In the *Code 15*, the *node* called *stage1* implements the *Worker1* job. The *node stage1* receives a *long* value from the input channel and return it incremented by *1*.

The *nodes* can be interconnected together using different type of communication channels (shared-memory or network) manually or using already-existing building blocks.

The *defaultJob* classes provides different methods (like runJob) that can be overridden and we will enter in details in the next pages of the manual.

Nodes interconnection

The *nodes* can be interconnected creating manually the communication channels or using sequential or parallel building blocks. The manual interconnection is simple and is shown in the *Code 16.*

```
defaultJob <Long, Long> Worker1 = new defaultJob<>() {
    public Long runJob(Long x) {
        return null;
    }

    public void init() {
        for (long i = 1; i <= 1000; ++i) {
            sendOut(i);
        }
        sendEOS();
    }
};

defaultJob <Long, Long> Worker2 = new defaultJob<>() {
    public Long runJob(Long x) {
        System.out.println("Received item "+x);
        return null;
    }
};

ff_node stage1 = new ff_node(Worker1);
ff_node stage2 = new ff_node(Worker2);
ff_queue shm_channel = new ff_queue();
stage1.addOutputChannel(shm_channel);
stage2.addInputChannel(shm_channel);

stage1.start();
stage2.start();
stage1.join();
stage2.join();
```

Code 16: Two-stage *pipeline* with manual created channel

In the *Code 16*, two *nodes* are interconnected using a shared memory channel where stage1 is the *producer* and *stage2* is the *consumer*. Both *nodes* must be started manually using *start()* function and their completion waited with *join()* method.

stage1 generates numbers from *1* to *1000* and sends them to the output channel. The result of this interconnection is a *pipeline* formed by two *nodes* that can be rewritten as in *Code 17.*

```
defaultJob <Long, Long> Worker1 = new defaultJob<>() {
    public Long runJob(Long x) {
        return null;
    }

    public void init() {
        for (long i = 1; i <= 1000; ++i) {
            sendOut(i);
        }
        sendEOS();
    }
};

defaultJob <Long, Long> Worker2 = new defaultJob<>() {
    public Long runJob(Long x) {
        System.out.println("Received item "+x);
        return null;
    }
};

ff_node stage1 = new ff_node(Worker1);
ff_node stage2 = new ff_node(Worker2);
ff_pipeline two_stage_pipeline = new ff_pipeline(stage1,stage2);

two_stage_pipeline.start();
two_stage_pipeline.join();
```

Code 17: Two-stage *pipeline*

In this case, executing *start()* and *join()* method on the *pipeline* is sufficient to run and wait both *nodes* in the *pipeline*.

With manual channel interconnection, is possible to realize any pattern. Anyway, that is advised only when already existing building blocks cannot satisfy the user requirements.

Network interconnection

The *nodes* can be interconnected also using network channels that we presented in the *Chapter 4.* The creation of a network channel consists in an instantiation of two different *ff_queue_TCP* objects (one of type *INPUT* and one of type *OUTPUT*), one for each *node*.

In the example below, there is a *pipeline* composed by two *nodes* interconnected through the network.

```java
import bbflow.*;

public class pipeline_network {
  public static void main (String[] args) {
  preloader.preloadJVM();

  boolean node1 = false;
  String host;
  if (args.length < 2) {
    System.out.println("Please specify which node to start and the host");
    return;
  } else {
    if (Integer.parseInt(args[0]) == 0) { node1 = true; }
    host = args[1]; // host for the client node
  }

  defaultWorker<Long, Long> Node1 = new defaultWorker<>() {
    public Long runJob(Long x) { return null; }

    public void init() {
    for (long i = 1; i <=100; ++i)  sendOut(i);
    sendEOS();
    }
  };

  defaultWorker<Long, Long> Node2 = new defaultWorker<>() {
    public Long runJob(Long x) {
    x *= 2;
    System.out.println("Received "+x+" from "+position);
    return null;
    }
  };


  ff_node E = new ff_node(Node1);
  ff_node F = new ff_node(Node2);

  if (node1) E.addOutputChannel(new ff_queue_TCP(ff_queue_TCP.OUTPUT, 1, host));
  if (!node1) F.addInputChannel(new ff_queue_TCP(ff_queue_TCP.INPUT, 1));

  if (node1) { E.start(); E.join(); }
  else { F.start(); F.join(); }
  }
}
```

Code 18: Two *nodes* connected through a network channel

The first *node* (*Node1*) has as output channel an *ff_queue_TCP* object of type *OUTPUT* and the second *node* (*Node2*) has as input channel an *ff_queue_TCP* object of type *INPUT*. Each *node* can be started independently using the first command line argument. Once both are started (on the same or on different machines), the first *node* connects to the second one (using the address provided) and the items can be transferred through the network channel.

```
user@h1:~/ $ java tests.pipeline_network 0 192.168.1.2
```

Figure 32: Command line to start the first *node* of a network pipeline

```
user@h2:~/ $ java tests.pipeline_network 1 192.168.1.1
Received 4 from 0
Received 6 from 0
Received 8 from 0
Received 10 from 0
…
```

Figure 33: Command line to start the second *node* of a network pipeline.

In the *Figure 32* and in the *Figure 33,* there are the two command lines needed to start the two *nodes* on different machines. Once both of them are running, the transmission of the items starts. The two *nodes* terminate their execution once the second *node* receives the *EOS* signal.

Sequential building blocks

There are two sequential building blocks available: *ff_node* (*Node*) and *ff_comb* (*Node combiner*).

The *Node* is the essential entity of the building blocks and all parallel patterns are built on top of it. Each *Node* provides different methods that user can override and they are:

- *"init()":* this method is called immediately after the *node* is started. It is called independently from the number of input and output channels.
- *"T runJob(T item)":* this method is called every time the *node* receives a new item from one of the input channels. The item value to be sent to the output channel can be returned from this method. If the returned value is *null*, nothing will be pushed to the output channel. This method is called only if there is at least one input channel.
- *"runJobMulti(T item, List channels)":* this method should be used when the *node* has more than one input channels. The items are received in *Roundrobin* and this function is called every item. The user can send items to one of the output channels using the list of channels provided as the function parameter or one of the provided functions described in the next paragraphs.
- *"runJob()":* this method is called in loop until there are not any channels anymore (input or output). The management of items (sent

and received) and of *EOS* is in charge of the user. This function can be used if the programmer needs it for particular applications.

- *"setEOS()":* this method is called when all input channels received *EOS (End Of Stream)* signal and there are not any items to process.

In addition, the user can count on a set of methods that can be used everywhere in the *node* and main ones are:

- *"sendOut(T item)":* send an item to one of the output channels (in *Roundrobin*). The item is sent every time to the same channel if there is only one output channel.
- *"sendOutTo(T item, int channel)":* send an item to the channel specified. The channels are counted starting from *0*.
- *"sendOutToAll(T item)":* send the item to all output channels. The items sent are a reference of the source one, so it should be used by the receivers as a constant (read-only) to avoid concurrency issues.

The *node combiner*, instead, is a way to combine different *nodes* together in sequential way. All jobs contained in *nodes* will be merged together in a single sequential job. The *Figure 34* shows how the combiner works from the logical point of view.

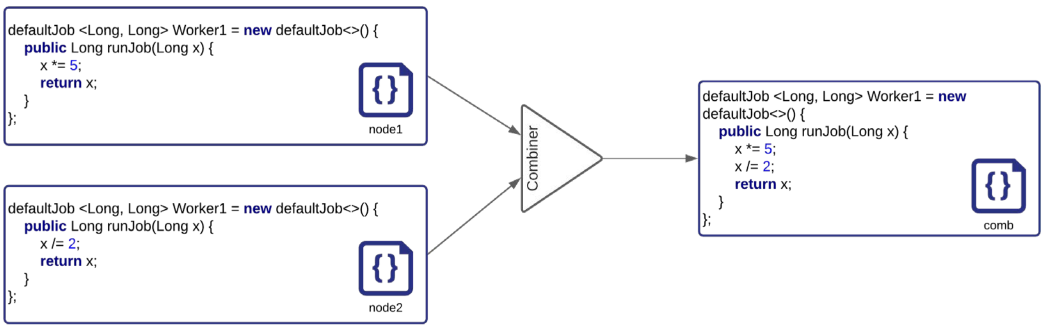


Figure 34: Logical behavior of the *Node combiner*

The resulting *node* contains a task that is the linear combination of the two source *nodes*.

Once the *nodes* are combined, they can be treated as a new *Node*. In the *Code 19* an example on the *ff_comb* usage.

```
ff_node stage1 = new ff_node(Worker1);
ff_node stage2 = new ff_node(Worker2);
ff_comb comb = new ff_comb(stage1,stage2);
```

```
comb.start();
comb.join();
```

Code 19: *Node combiner* of two *nodes*

Parallel building blocks

The parallel building blocks available to the user are *Farm*, *Pipeline* and *All-to-all*. All of them are widely described in this thesis and their usage it is simple.

The *Farm* building block is composed by default by an *Emitter*, *N Workers* and a *Collector*. *Workers* task must be specified during the *Farm* class instantiation and they can be of type *defaultWorker* or *defaultJob*.

```
defaultJob <Long, Long> Worker1 = new defaultJob<>() {
    public Long runJob(Long x) {
        return null;
    }

    public void init() {
        for (long i = 1; i <= 100; ++i) {
            sendOut(i);
        }
        sendEOS();
    }
};

defaultWorker<Long,Long> workerJob = new defaultWorker<>() {
    public Long runJob(Long x) {
        return x*2;
    }
};

defaultJob <Long, Long> Worker2 = new defaultJob<>() {
    public Long runJob(Long x) {
        System.out.println("Received item "+x);
        return null;
    }
};


int n_workers = 3;
ff_node generator = new ff_node(Worker1);
ff_farm farm = new ff_farm<>(n_workers, workerJob);
ff_node filter = new ff_node(Worker2);

ff_pipeline pipe = new ff_pipeline(generator,farm);
pipe.appendBlock(filter);

pipe.start();
pipe.join();
```

Code 20: Complex *pipeline* with two *SISO nodes* and a *Farm*

In the example in *Code 20*, there is a *pipeline* composed by a *Node*, a *Farm* and another *Node*. The *Farm's Emitter* receives items generated by *generator node* (*Worker1*) and the *filter node* (*Worker2*) receives from the *Farm's Collector* the results of *workerJob*. In this case, *Emitter* and *Collector* are provided by the *RTS* and the user specifies only the *worker's task*.

The *Emitter* and *Collector* can be easily replaced as shown in the examples in *Code 21* and *Code 22*.

```
defaultJob <Long, Long> Emitter = new defaultJob<>() {
    public Long runJob(Long x) {
        return null;
    }

    public void init() {
        for (long i = 1; i <= 100; ++i) {
            sendOut(i);
        }
        sendEOS();
    }
};

defaultWorker<Long,Long> workerJob = new defaultWorker<>() {
    public Long runJob(Long x) {
        return x*2;
    }
};

defaultJob <Long, Long> Collector = new defaultJob<>() {
    public void runJobMulti(Long x, LinkedList<ff_queue<Long>>
out) {
        System.out.println("Received item "+x+" from channel
"+position);
    }
};

ff_farm farm = new ff_farm(3, workerJob);
farm.removeEmitter();
farm.removeCollector();
farm.emitter = new ff_node(Emitter);
farm.collector = new ff_node(Collector);
farm.connectEmitterWorkers();
farm.connectWorkersCollector();

farm.start();
farm.join();
```

Code 21: *Emitter* and *Collector* replacement on a default *Farm*.

```
defaultJob <Long, Long> Emitter = new defaultJob<>() {
    public Long runJob(Long x) {
        return null;
    }

    public void init() {
        for (long i = 1; i <= 100; ++i) {
            sendOut(i);
        }
        sendEOS();
    }
};


defaultWorker<Long,Long> workerJob = new defaultWorker<>() {
    public Long runJob(Long x) {
        return x*2;
    }
};

defaultJob <Long, Long> Collector = new defaultJob<>() {
    public void runJobMulti(Long x, LinkedList<ff_queue<Long>>
out) {
        System.out.println("Received item "+x+" from channel
"+position);
    }
};

LinkedList<ff_node> workers = new LinkedList<>();
for (int i=0; i<3; i++) workers.add(new
ff_node(defaultJob.uniqueJob(workerJob)));

ff_farm farm = new ff_farm(0, null);
farm.emitter = new ff_node(Emitter);
farm.collector = new ff_node(Collector);
farm.workers = workers;
farm.connectEmitterWorkers();
farm.connectWorkersCollector();

farm.start();
farm.join();
```

Code 22: Adding custom *Emitter* and *Collector* on a empty *Farm*.

In the first example, the *Emitter* and the *Collector* are replaced in the following way:

- After creation of the *Farm* with default *Emitter* and *Collector*, both of them are removed using *removeEmitter* and *removeCollector* methods.
- The new *Emitter* and *Collector* are assigned to the *emitter* and *collector* variables of the *Farm*.
- The connection between *Emitter*, *Workers* and *Collector* is made using *connectEmitterWorkers* and *connectEmitterWorkers* methods.

In the second example, the *Farm* is instantiated without any *node* inside and all of them must be instantiated:

- New *Emitter* and *Collector* are assigned to the *emitter* and *collector* variables of the *Farm*.
- A set of *Workers* is assigned to the *workers* variable of the *Farm*.
- The connection between *Emitter*, *Workers* and *Collector* is made using *connectEmitterWorkers* and *connectEmitterWorkers* methods.

The ***Pipeline*** building block can be instantiated using the class *ff_pipeline.* The *nodes* in the *pipeline* can be interconnected using many different topologies described in detail in the *Chapter 3* of the thesis.

All types of building blocks can be interconnected using the *pipeline* pattern and the *1-to-1* interconnection is used by default.

The *pipeline* building block can be instantiated starting from two building other blocks. Any new block can be added to the *pipeline* using the method *"appendBlock()".*

```
defaultWorker<Long,Long> w0 = new defaultWorker<>() {
    public Long runJob(Long x) {
        return null;
    }

    public void init() {
        for (long i = 1; i <= 100; ++i) {
            sendOut(i);
        }
        sendEOS();
    }
};

defaultWorker<Long,Long> w1 = new defaultWorker<>() {
    public Long runJob(Long x) {
        return x*2;
    }
};

defaultJob <Long, Long> w2 = new defaultJob<>() {
    public Long runJob(Long x) {
        System.out.println("Received item "+x);
        return null;
    }
};

ff_node stage1 = new ff_node(w0);
ff_node stage2 = new ff_node(w1);
ff_node stage3 = new ff_node(w2);

ff_pipeline pipe = new ff_pipeline(stage1,stage2);
```

```
pipe.appendBlock(stage3);

pipe.start();
pipe.join();
```

Code 23: Three-stage *pipeline* example.

In the example above, a *three-stage pipeline* is instantiated. The *pipeline* is constructed starting from the first two stages and adding the third one using *appendBlock* method.

The *All-to-all* building block is used to merge different *Farms* and try to remove centralization points (*Emitter* and/or *Collector*). There are many different resulting topologies after the merge and all of them are described in the *Chapter 3* of the thesis.

```
ff_all2all stages = new ff_all2all();
stages.combine_farm(stage1,stage2);
```

Code 24: *All-to-all* usage example.

The example in the *Code 24* declares *All-to-all* building block where *stage1* and *stage2* are *Farms* and are combined using the method *combine_farm*.

The *combine_farm* method can take different parameters based in which resulting topology the user requires. The constructor can take different parameters as shown in *Code 25.*

```
void combine_farm(ff_farm<T,U> b1, ff_farm<V,W> b2,
ff_node<U,Object> R, ff_node<Object,V> G, boolean merge)
```

Code 25: *All-to-all Node combiner* constructor

The usage of *R*, *G* and *merge* variables is described in the *Chapter 3* of the thesis.

Examples

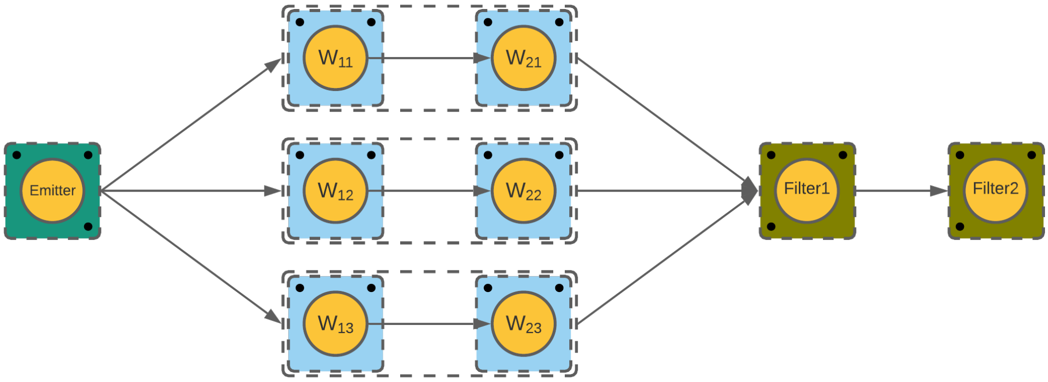
In the *BBFlow* project directory, there are different examples, simple or complex. In this section, we show few of them and explain how they are implemented.

combine2

```java
defaultWorker<Long, Long> Emitter = new defaultWorker<>() {
    public Long runJob(Long x) {
        return null;
    }

    public void init() {
        for (long i = 1; i <=100; ++i) {
            sendOutTo(i, ((int)i)%out.size());
        }
        sendEOS();
    }
};

defaultWorker<Long, Long> Worker1 = new defaultWorker<>() {
    public Long runJob(Long x) {
        return x;
    }
};

defaultWorker<Long, Long> Worker2 = new defaultWorker<>() {
    public Long runJob(Long x) {
        System.out.println("Worker2 (id="+id+") in="+x);
        return x;
    }
};

defaultWorker<Long, Long> Filter1 = new defaultWorker<>() {
    public void runJobMulti(Long x, LinkedList<ff_queue<Long>> out) {
        System.out.println("Received "+x+" from "+position); sendOut(x);
    }
};

defaultWorker<Long, Long> Filter2 = new defaultWorker<>() {
    public void runJobMulti(Long x, LinkedList<ff_queue<Long>> out) {
        System.out.println("Filter2 received: "+x);
    }
};

ff_node stage1 = new ff_node(Emitter);
ff_node stage3 = new ff_node(Filter1);
ff_node stage4 = new ff_node(Filter2);


ff_farm stage2 = new ff_farm<Long,Long>(0, null);
stage2.workers.push(new ff_comb(new ff_node(defaultJob.uniqueJob(Worker1,1)),new
ff_node(defaultJob.uniqueJob(Worker2,1))));
stage2.workers.push(new ff_comb(new ff_node(defaultJob.uniqueJob(Worker1,2)),new
ff_node(defaultJob.uniqueJob(Worker2,2))));
stage2.workers.push(new ff_comb(new ff_node(defaultJob.uniqueJob(Worker1,3)),new
ff_node(defaultJob.uniqueJob(Worker2,3))));

stage2.emitter = stage1;
stage2.collector = stage3;
stage2.connectWorkersCollector();
stage2.connectEmitterWorkers();

ff_pipeline all = new ff_pipeline(stage2,stage4);

all.start();
all.join();
```

Code 26: *combine2* synthetic computation

The combine2 example uses different building blocks: *Node*, *Pipeline*, *Farm* and *Node combiner.* The resulting parallel pattern generated by the code is the following:

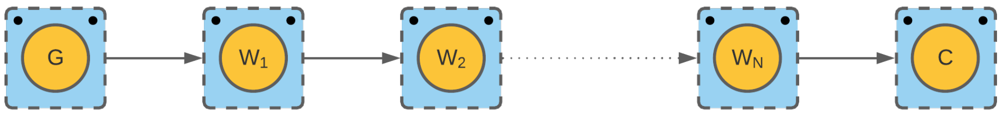


The structure is a two-stage *pipeline* where:

- The first building block of the *pipeline* is *Farm* that is instantiated without any *node*. *stage1* is assigned to *Emitter* and *stage3* is assigned to the *Collector* of the *Farm*. Three *workers* are added to the *Farm* and they are the *composition* (using *combine*) of two different *workers* (*Worker1* and *Worker2*).
- The second building block is a *node* (*Filter2*) receiving first stage's output.

benchmark_*pipeline*

This example is a benchmark used in our performance analysis. It is composed by a set of *nodes* connected in *pipeline*.



The number of *nodes* is *N+2* where *N* is the number of *workers* doing the computation inside of the *pipeline*. The other two *nodes* are a generator and a *collector* that receives the results.

The code is the following:

```
preloader.preloadJVM();
```

```java
bb_settings.backOff = 100;

int n = 1000;
int n_workers = 16;
if (args.length >= 2) {
    n = Integer.parseInt(args[0]);
    n_workers = Integer.parseInt(args[1]);
    if (args.length == 3) {
        bb_settings.backOff = Integer.parseInt(args[2]);
    }
}

Long finalN = Long.valueOf(n);
defaultWorker<Long, Long> Generator = new defaultWorker<>() {
    public Long runJob(Long x) {
        return null;
    }

    public void init() {
        for (long i = 1; i <= finalN; ++i) {
            sendOut(i);
        }
        sendEOS();
    }
};

int totalTask = 1000000/n_workers;
defaultWorker<Long, Long> Worker = new defaultWorker<>() {
    public Long runJob(Long x) {
        long y = x;
        for (int i=0; i<totalTask; i++) {
            y *= 1000;
            y /= 999;
        }
        return y;
    }
};

defaultWorker<Long, Long> Final = new defaultWorker<>() {
    public Long runJob(Long x) {
        return null;
    }
};

ff_pipeline all = new ff_pipeline(new ff_node(Generator), new
ff_node(defaultJob.uniqueJob(Worker)));

for (int i=0; i<n_workers-1; i++) {
    all.appendBlock(new ff_node<>(defaultJob.uniqueJob(Worker)),
ff_pipeline.TYPE_1_1);
}

all.appendBlock(new ff_node<>(Final), ff_pipeline.TYPE_1_1);

customWatch x = new customWatch();
x.start();
all.start();
all.join();
x.end();
x.printReport(true);
```

Code 27: Source code of the *pipeline* benchmark

This benchmark exploits different interesting classes present in *BBFlow*. For example, the *preloader* class is called at the beginning of the application to let the *JVM* to load commonly used classes and reduce the computation time. Instead, the *customWatch* class allows measuring the time spent by a computation. It works like a stopwatch and once the programmer starts the *watch*, it records the total time between *start* and *end* methods; the user could also register time steps using *watch* method. The time registered can be printed to screen using *printReport* method in *milliseconds* or *microseconds*.

The benchmark creates the *pipeline* described above adding one *node* after the other with *appendBlock* function. Every *worker node* added to the *pipeline* must be unique. The *worker nodes* are created from the same *defaultJob* object and therefore, they must be cloned in order to have different instances. The duplication of the *Worker* object can be done using the *defaultJob.uniqueJob*.

ordered_*farm*_labeling

This last example shows how an *ordered farm* can be implemented. There are different ways to achieve the same result; in this particular case, we attach a *"label"* to each item in order to have the possibility to sort them after the *Farm* computation.

The code of this implementation is the follow:

```
class packet<T> {
    T value;
    int id;
    public packet(T val, int id) {
        this.value = val;
        this.id = id;
    }
}

defaultWorker<packet, packet> Generator = new defaultWorker<>() {
    public packet runJob(packet x) {
        return null;
    }

    public void init() {
        for (Integer i = 0; i < 1000; ++i) {
            packet<Integer> x = new packet(i, i);
            sendOutTo(x, i % out.size());
```

```
        }
        sendEOS();
    }
};

defaultWorker<packet, packet> Worker1 = new defaultWorker<>() {
    public packet runJob(packet x) {
        try {
            Thread.sleep(5);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return x;
    }
};

defaultWorker<packet, packet> Filter1 = new defaultWorker<>() {
    TreeSet<packet> sortedqueue = new TreeSet<packet>(new
Comparator<packet>() {
        @Override
        public int compare(packet s1, packet s2) {
            if (s1.id<s2.id) { return -1; }
            else if (s1.id==s2.id) { return 0; }
            else { return 1; }
        }
    });

    int lastid = -1;
    public packet runJob(packet x) {
        if (x.id == (lastid+1)) {
            sendOut(x);
            lastid++;
            while(sortedqueue.size() > 0) {
                packet p = sortedqueue.first();
                if (p.id == (lastid+1)) {
                    sendOut(p);
                    sortedqueue.remove(p);
                    lastid++;
                } else {
                    break;
                }
            }
        } else {
            sortedqueue.add(x);
        }

        return null;
    }

    @Override
    public void EOS() {
        for (packet p : sortedqueue) {
            sendOut(p);
        }
        sortedqueue.clear();
    }
};

defaultWorker<packet, packet> Filter2 = new defaultWorker<>() {
    public void runJobMulti(packet x, LinkedList<ff_queue<packet>> out)
{
        System.out.println("Filter2 received: "+x.value);
    }
};
```

```java
ff_node stage1 = new ff_node(Generator);
ff_node stage3 = new ff_node(Filter1);
ff_node stage4 = new ff_node(Filter2);

int n_workers = 64;
if (args.length == 1) {
    n_workers = Integer.parseInt(args[0]);
}

ff_farm stage2 = new ff_farm<Integer, Integer>(0, null);
for (int i = 0; i < n_workers; i++) {
    stage2.workers.push(new ff_node(defaultJob.uniqueJob(Worker1, i)));
}

stage2.emitter = new ff_node(new
defaultEmitter(defaultEmitter.ROUNDROBIN));
stage2.collector = new ff_node(new
defaultCollector<Integer>(defaultCollector.FIRSTCOME));
stage2.connectWorkersCollector();
stage2.connectEmitterWorkers();


ff_pipeline all = new ff_pipeline(stage1, stage2);
all.appendBlock(stage3, ff_pipeline.TYPE_1_1);
all.appendBlock(stage4, ff_pipeline.TYPE_1_1);

customWatch x = new customWatch();
x.start();
all.start();
all.join();
x.end();
x.printReport(true);
```

Code 28: *Ordered Farm* exploiting *packet labeling*

Each item is of type *packet* that contains two variables, one that is the value of the packet and another one that contains the *id* (*label*) of the packet (increasing *integers*). The items are emitted by the *Generator* and sent to the *Farm*. The *Farm's emitter* distributes the items to the *workers* in *Roundrobin*. Each time a *worker* concludes its computation, the resulting item is sent to the *Collector*. The *Collector* receives the items from the *Workers* in a *Firstcome* configuration (try to retrieve items from all *workers* as soon as they are available). The unsorted stream of items received by the *Collector* are forwarded to the next *node* that is *Filter1*. The *Filter1 node* receives unsorted items and output them sorted using the *packet id*. When *Filter1 node* receives an item, it checks if the *packet id* is equal to *(last packet id sent to the next node)+1*. If *true*, the item is sent to the output channel, otherwise it is stored in a *sorted list*. Every time an item is sent to the next *node*, items present in the *sorted list* are check and sent if they are sequential. At the end of computation, once *EOS* reached, the *sorted list* is flushed and items are sent to the next *node*.

Java options

There are different *JVM* options that can be used to tune the applications performance. One of them is to activate the *Z Garbage Collector* (described in The *Z Garbage Collector*). Other options allow increasing memory and configuring the garbage *collector* behavior. A collection of few of them is listed in the table below.

-XX:+UseZGC	Activate the *Z Garbage Collector*.
-Xmx<size> (example –Xmx16G)	Set the Max *Heap* Size.
-Xms<size>	Set the Minimum *Heap* Size.
-XX:ParallelGCThreads	Set number of *Garbage Collector* threads.
-XX:UseLargePages	Increase the size of memory pages.
-Xlog:gc*	Show detailed *garbage collector* logs. Useful to check the garbage *collector* behavior.

There are many others options available that depends on the *Java* version used. All of them can be found on the *Java* documentation or they can be listed with their actual assigned value using the command:

```
user@h:~/ $ java -XX:+PrintFlagsFinal

[Global flags]
      int ActiveProcessorCount                              = -1
{product} {default}
    uintx  AdaptiveSizeDecrementScaleFactor                  =  4
{product} {default}
    uintx  AdaptiveSizeMajorGCDecayTimeScale                = 10
{product} {default}
    uintx   AdaptiveSizePolicyCollectionCostMargin         =   50
{product} {default}
    uintx  AdaptiveSizePolicyInitializingSteps              =  20
{product} {default}
    uintx  AdaptiveSizePolicyOutputInterval                  =  0
{product} {default}
    uintx AdaptiveSizePolicyWeight                          = 10
{product} {default}
    uintx  AdaptiveSizeThroughPutPolicy                      =  0
{product} {default}
    uintx AdaptiveTimeWeight                                = 25
{product} {default}
     bool AdjustStackSizeForTLS                          = false
{product} {default}
```

```
     bool AggressiveHeap                                      = false
{product} {default}
     intx AliasLevel                                              = 3
{C2 product} {default}
     bool AlignVector                                         = false
{C2 product} {default}
    ccstr AllocateHeapAt                                            =
{product} {default}
…
…
```