UNIVERSITÀ DI PISA
SCUOLA SUPERIORE SANT'ANNA
Master degree in Computer Science and Networking

MASTER THESIS

A Distributed-Memory run-time for *FastFlow*'s building-blocks

Supervisors
Massimo TORQUATI
Gabriele MENCAGLI

Candidate
Nicolò TONCI

Academic year 2019-2020

Abstract

We are witnessing a proliferation of interconnected devices capable of generating unbounded sequences of data items like sensor readings. An increasing number of applications have to process such huge amounts of data in real-time, generate alerts or notifications to the users, and extract insights from the raw data. Those requirements intensify the need for new high scalable applications and programming interfaces for shared-memory and distributed architectures.

Along this line, the thesis aims to propose a new distributed-memory run-time system for *FastFlow*, a C++ structured parallel programming library originally targeting shared-memory platforms. The new run-time system enables an easy transition between shared-memory *FastFlow* applications to equivalent distributed-memory applications, allowing the exploitation of the parallel resources of multiple machines. A set of tests validates the proposed run-time system demonstrating a significant performance gain in exploiting the resources of multiple interconnected machines.

Contents

List of Acronyms

API	Application Program Interface
BB	Building Block
CPU	Central Processing Unit
DaSP	Data Stream Processing
DSL	Domain Specific Language
EOS	End Of Stream
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array
GPGPU	General Purpose Graphics Processing Unit
HPC	High Performance Computing
ICT	Information and Communication Technologies
IoT	Internet of Things
JVM	Java Virtual Machine
MPI	Message Passing Interface
MPMC	Multi-Producer Multi-Consumer
MPSC	Multi-Producer Single-Consumer
NFS	Network File System
RDD	Resilient Distributed Dataset
RDMA	Remote Direct Memory Access
RTS	Run-Time System
SPE	Stream Processing Engine
SPMD	Single Program Multiple Data
SPSC	Single-Producer Single-Consumer
SPMC	Single-Producer Multi-Consumer
VM	Virtual Machine
WC	Word Count

List of Figures

Listings

List of Tables

Introduction

In recent years we are witnessing a continuous increase in data production, as never happened before. This data, nowadays, is produced both by humans and machines (including sensors). This trend is powered by the high diffusion of the Internet of Things (IoT) devices and the vast explosion of tools for Big Data Analytics. Also, in the High-Performance Computing (HPC) domain, numerous scientific experiments ranging from particle accelerators to financial analysis are generating large volumes of data that require to be processed as soon as possible.

If, on the one hand, there is a need to analyze a large number of data, on the other hand, the market offer machines equipped with multi/many-core, several hardware accelerators, such as GPGPUs or FPGA, and high-performance and high-bandwidth interconnection devices. Those machines usually compose clusters to increase performance and achieve scalability. Also, in the cloud computing setting, to accomplish fault tolerance and implement geo-distribution of the applications, multiple interconnected multi-core servers (deployed as virtual machines) are typically used[2].

Therefore, the highest priority nowadays is to provide programmers with effective parallel programming tools targeting single multi-core servers as well as multiple interconnected multi-core platforms. However, shared-memory architectures and distributed-memory architectures, such as a cluster of interconnected machines, require different techniques and tools to support efficient parallelism exploitation. There are multiple parallel programming frameworks available within the shared-memory domain, providing either low levels abstractions and high-level algorithmic skeletons [3]. *Intel TBB*[4], *SkePU*[5] and *FastFlow*[6, 7, 8, 9] are some examples. However, the standard de facto is OpenMP[10]. In the distributed-memory domain, we have a sharp distinction between distributed processing frameworks targeting Big-Data analysis and tools targeting High-Performance Computing. However, in the context of HPC, the standard de facto is MPI[11]. OpenMp and MPI are usually used in conjunction to enable hybrid parallelism. Even if they are very efficient and optimized, they share common problems: a rather low-level abstraction presented to the application programmer and a poor separation of concerns (SoC)[12] resulting from the mix of business logic code and

system code. In addition, they expose two different programming models to the programmers, who should be remarkably expert in both models to squeeze the maximum performance from the underlying platform. Such problems contribute to hindering the ease of use of MPI+OpenMP programming model. Several algorithmic skeleton-based parallel libraries offer support for distributed-memory architectures. They raise the level of abstraction to mitigate programming difficulties and provides a single programming model and functionalities to easily balance the workload between nodes of the cluster and cores of the single node. Some of them are *GrPPI* [13], *SkePU 3*[5, 14], *Muesli*[15] and *GeoSkelSL*[16].

In this thesis we propose a distributed-memory run-time system support for the latest version of the *FastFlow* programming library originally targeting cache-coherent shared-memory multi-cores. The main objective is to enable an easy transition from pure *FastFlow*'s shared-memory applications running on a single multi-core server to distributed-memory applications capable of exploiting the resources of multiple nodes. The distributed RTS presented in this thesis has been designed to minimize the programmers' efforts in fulfilling this transition. It employs raw sockets for inter-process communications and supports two mechanisms of data serialization. However, the *FastFlow* programmer does not have to deal with low-level networking aspects since, thanks to our proposed APIs, the programming model exposed has not changed with respect to the one exposed on a single shared-memory machine.

In addition, to speedup the testing of the distributed RTS, an additional software module has been designed and developed. This module (called `dff_run`) simplifies the deployment and launch of the distributed *FastFlow* application. It has been implemented along the lines of the well-known MPI launcher (*mpirun*)[17].

Finally, to validate the proposed approach, multiple benchmarks and application examples are presented. Most notably, we measure the capacity of fully exploiting the available bandwidth and the benefit, in term of speedup, introduced in two different use cases: Word Count and a video filtering application implementing the Lane Detection.

More specifically, throughout the thesis we present the following contributions: (a) the design and implementation of the new *FastFlow* distributed-memory run-time system, which makes the transition from the shared-memory to the distributed-memory domain easy and smooth; (b) the implementation of the loader module `dff_run` for simplifying the deployment and execution of distributed *FastFlow* applications; (c) the evaluation of the performance and the evaluation of the correctness of the design proposed by using different applications.

The whole implementation of the proposed RTS, along with a large set of tests, is available online as open-source code in the official *FastFlow*'s *GitHub*

repository[1].

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

- *Chapter 1*: the chapter is devoted to the description of some of the available structured parallel programming frameworks, distributed systems for Big Data, the MPI standard, and the high-level parallel frameworks employing MPI to target distributed-memory architectures.

- *Chapter 2*: it presents the background needed to understand the contributions of the thesis. It focuses on the description of *FastFlow* library, with a detailed description of its structure, the provided parallel building-block, the high-levels framework built on top of it, and a brief description of the legacy distributed-memory support.

- *Chapter 3*: the chapter presents the description of the design of the proposed distributed-memory run-time, employing a top-down approach.

- *Chapter 4*: it provides a brief description of the *loader* module, the structure of configuration files and list of the possible improvements and features that can be introduced.

- *Chapter 5:* the chapter focuses on evaluating the correctness and performances of the proposed distributed RTS. Through three different applications, various aspects have been evaluated, such as the ability to fully exploit the available bandwidth, the capacity to provide speedup to different applications, and the proof of correctness of the routing of messages.

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

[1] `https://github.com/fastflow/fastflow`

Chapter 1

Background

Writing an efficient and bug-free sequential code is not a trivial task. Rewriting the same application exploiting a parallel architecture is even more painful. The programmers need to have full knowledge of the underlying architecture and parallel computing techniques. For instance: how the cache coherence works in that particular architecture, the presence of hardware accelerators, and how to synchronize shared data are important aspects to consider to produce fast and efficient parallel code. Fortunately, some frameworks, built on top of primitives given by every programming language, come up to facilitate the development. Frameworks abstract the low-level threading details necessary for effectively utilizing multi-core processors and free the programmer to write the error-prone synchronization procedures. Those libraries also increase the portability of applications, taking care of parameter tuning (e.g., optimal parallelism degree) as a function of the target architecture. The de facto standard for shared-memory parallel programming is *OpenMP*[10]. In addition, in the '80s researchers introduced the *algorithmic skeletons* methodology[18][3] (also know as parallel patterns), and the scientific results of the last decades have clearly shown that relatively few patterns are sufficient to exploit the architectures of modern multi-core processors efficiently. Hence, several parallel programming frameworks have introduced those patterns with the possibility to nest them in order to implement complex applications. Some available libraries are: *Intel TBB*[4], *SkePU*[5], *FastFlow*[6].

1.1 Distributed Systems for Big Data

In recent years, we have witnessed a rapid growth in data production, the so-called *data deluge*. Such a huge amount of data is available due to the widespread diffusion of connected sensors (e.g., like in smart cities, and smart industrial processes) and because of the pervasiveness of ICT technology in human life. The

concept of Big Data[19] is literally changing the methods to solve problems and gives answers to fundamental questions in science, medicine, business, economy, and so forth [20]. Thus, there is a huge demand for tools capable of processing those massive volumes of data, either in real-time or offline (i.e., deferred or asynchronously), and able to exploit the ever-increasing power of computing and networking resources. For stringent real-time demand, we talk about Data Stream Processing (DSP) applications. DSP application takes continuous and possibly infinite data streams and produces streams of results. Since the analysis should be immediate, strong performance requirements must be enforced in DSP frameworks: high throughput and low latency. There are available on the market multiple Streaming Processing Engines (SPE) targeting clusters of commodity servers [21, 22, 23, 24] (by scaling horizontally, or scaling-out, the application in distributed memory environments). Those solutions are usually built on top of instances of the Java Virtual Machine (JVM) to be platform agnostic. The most widely used products are: Apache Storm [21], Apache Spark Streaming[22], Apache Kafka[23] and Akka[24]. Instead, offline (or batching) computations require the full availability of the complete dataset to process, which is stored and available. Since datasets can be even Petabytes of data, they are usually scattered in multiple machines using distributed filesystems (e.g., NFS). Hence, to exploit data locality and to avoid useless movement of data, batch engines [25, 22] tend to compute on each node the partition of data locally available, scaling out the computation by taking advantage of the whole parallelism degree available in a cluster of interconnected machines. In this field, the leading frameworks are: Apache Hadoop[25], which provides the MapReduce programming model, and Apache Spark[22].

Apache Storm Apache Storm is an open-source distributed real-time data stream processing framework. The main goal of the framework is stream processing. The approach is analogous to the one given by Apache Hadoop, but in real-time instead of batch processing with map-reduce jobs. Basically, Storm provides abstractions that allow the programmer of a data stream processing application to avoid the manual setup of a network of queues and workers. Fault tolerance, scalability and language-agnostic programming are also key properties of this framework. To express a DSP application, the programmer has to define a topology, which is the top-level abstraction representing the network of interconnected operators. A topology is a directed acyclic data-flow graph where operators are running forever absorbing data from the configured sources. Processing operators may be of two types: (a) *Spouts*, which represent sources, and (b) *Bolts*, which consume data from (possible) multiple input streams, execute a transformation, and emit results in (possible) multiple output streams. All operators in

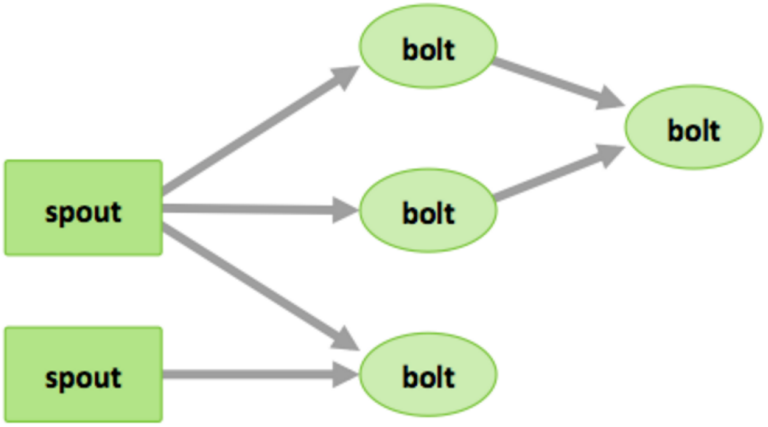

Figure 1.1: High-level view of a Storm topology.

the topology (or even of more running topologies) execute concurrently with the possibility to specify the parallelism degree of each of them. A stream is defined in Storm as an unbounded sequence of structured records. The connection of nodes through streams follows a static pattern. To receive a stream, bolts need to subscribe to streams produced by other bolts or spouts. A variety of simple bolt abstraction types are provided by the framework, like filters, aggregations, joins, and storage/retrieval from the persistent store. Indeed, by extending suitable API, the programmer can use its business-logic operators.

Apache Spark Apache Spark is an open-source fault-tolerant and general-purpose distributed engine providing APIs in multiple programming languages, such as Java, Scala, Python, and R [26]. The engine supports general execution graphs and targets either large-scale data processing. Through the streaming library, it can execute streaming jobs in the same way as batch jobs. It has become the de-facto standard because of its in-memory programming model, particularly suitable for iterative applications such as machine learning algorithms. Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. An RDDs can be created through deterministic operations on either data in persistent storage or other RDDs. Applications in Spark are composed of chains of transformation followed by an action. RDD transformations return a pointer to new RDDs and generate dependencies between RDDs. Each RDD in a dependency chain has a function for calculating its data and has a pointer (i.e., the dependency) to its parent RDD. Actions, placed at the end of the transformation chain, triggers the whole computation and handle results. A simple Word Count

application examples in Spark is shown in Listing 1.1.

```
1 text_file = sc.textFile("hdfs://...")
2 counts = text_file.flatMap(lambda line: line.split("␣")) \
3                   .map(lambda word: (word, 1)) \
4                   .reduceByKey(lambda a, b: a + b)
5 counts.saveAsTextFile("hdfs://...")
```

Listing 1.1: Word Count algorithm in Apache Spark

At line 1 the base is created by reading a specific file, then intermediate RDDs are created by `flatMap`, `map` and `reduce` transformations, finally the computation is triggered and results saved in a file by `saveAsTextFile` action.

1.2 Message passing interface (MPI)

The Message Passing Interface (MPI) [11] is a standard for writing message-passing parallel programs, introduced in the late '93. Basically, it is a specification for the developers and users of libraries addressing the message-passing parallel programming model. MPI has become a de-facto standard for communication among processes modeling parallel programs running on a distributed memory system, accepted by both the academic and industrial worlds. The standard aims to be: practical, portable, efficient, and flexible. Both point-to-point and collective communications are supported. The first implementation of the first released version of MPI was MPICH [27] in late '96. Then in 2004, with the objective to unify all the available implementations, *Open MPI* [28] was released. Nowadays, the latter can be considered the main implementation of MPI used in the majority of top 500 supercomputers. The adopted optimizations, such as hardware acceleration, e.g., RDMA, make the implementation very powerful and efficient. The main targets of *Open MPI* are C, C++, and Fortran, but bindings are available for many other languages, including Perl, Python, R, Ruby, Java.

So far, we have cited frameworks targeting just shared-memory architectures (e.g., OpenMP, Intel TBB) and frameworks for inter-process communication (e.g., MPI) targeting distributed-memory architectures. However, to push and achieve high degrees of scalability, multiple shared-memory multi-core platforms are required. The conjunction of shared-memory parallelism and distributed-memory parallelism is called hybrid parallelism. To fulfill this kind of need, techniques combining shared-memory parallelism and distributed-memory parallelism are often required: for instance MPI + OpenMP [29] [30]. Regardless of their performance, both libraries provide low-level abstractions and require some significant

knowledge to fine-tune the target application. Some pattern-based frameworks introduced extensions to deploy shared-memory applications to distributed-memory systems with reduced programming effort so to relieve the programmer from dealing with low-level communication and/or synchronization details. For instance: *GrPPI* [13] which is based on a MPI backend[31], *Muesli* [15] framework which has a builtin support for MPI, or *FastFlow*, which, in early stage, had a limited support to inter-process communication trough *ZeroMQ*[32].

GrPPI GrPPI[13] is an open-source generic and reusable parallel pattern programming interface developed at the University Carlos III of Madrid. It provides an intermediate layer between developers and existing parallel programming frameworks targeting multi-core processors, such as ISO C++ Threads, OpenMP, Intel TBB, and FastFlow. The interface leverages modern C++ features, metaprogramming concepts, and generic programming to switch between those frameworks. The parallel patterns supported by GrPPI are for both stream processing and data-parallel applications. Recently the MPI execution policy has been introduced [31]. This new back-end supports both pure distributed and hybrid (i.e., distributed + shared memory) parallel executions of pipeline and farm patterns. Communication between processes is handled by distributed queues implemented on top of MPI primitives. The benefits of using this approach with respect to the de facto standard of MPI+OpenMP are numerous and involve programmability, flexibility, and readability. On Listing 1.2 is shown an example of a distributed pipeline, where the first and third stages are sequential and the second stage is replicated two times.

```
grppi::parallel_execution_mpi ex{argc, argv, grppi::two_sided};
grppi::pipeline(ex,
    [x=1, n]() mutable -> optional<double> {
        if (x<=n) return x++;
        else return {};
    },
    grppi::farm(2, [](double x){
        return x*x;
    }),
    [](double x) { cout << x << endl; }
);
```

Listing 1.2: Example of GrPPI distributed pipeline

SkePU 3 SkePU[5] is an open-source skeleton programming framework for multicore CPUs and multi-GPU systems. In version 3, the authors introduced the

cluster backend to support also distributed memory architectures. Basically, this backend generates StarPU task code using the MPI interface of the SarPU RTS [14]. The framework also provides a distributed version of smart data containers, such as Vector, Matrix, and so forth, with the same interface as the shared-memory, but with default distributions. Each cluster node runs a single copy of the SkePU executable on top of a local instance of StarPU in SPMD style. SPMD stands for *Single Program Multiple Data* and is a programming technique where all the processes execute the same program; the distinction in the executions among different processes occurs by differentiating the program flow based on the local rank of the process. Each node only executes the operations that involve local partition of the container, following the "owner compute rule". To use the cluster backend, no syntactic changes in the SkePU code of the application are required, enforcing the SkePU's strict portability principle. The framework forces to have a master process (i.e., the SPMD process having rank 0) to perform I/O operations. Hence, it provides semi-automatic mechanisms to annotate the master I/O code and to synchronize data from/to the master process.

Muesli The Muenster Skeleton Library (Muesli)[15] is a C++ programming library enabling the programming of heterogeneous clusters equipped with multi-core CPUs as well as many-core GPUs and co-processors by implementing the concept of algorithmic skeletons. It uses the current standard message passing interface MPI to facilitate the communication between processes and adopts the SPMD model. The framework provides an abstraction layer for inter-process communication, which is transparent to the programmer. As seen previously in SkePU, Muesli provides a set of distributed data containers (i.e.m vector, matrix and sparse matrix) to implement data parallelism. Along with distributed data containers, it is shipped with a set of parallel patterns such as pipe, farm, divide&conquer, also supporting task parallelism and combinations of both task and data parallelism. Additionally, to fully exploit each machine's parallel resources, the framework employs OpenMP for the intra-process concurrency.

GeoSkelSL GeoSkelSL[16] is the DSL top-level layer of GeoSkel. The latter is a framework aiming to cover the recurring computations on rasters in geo-sciences. GeoSkelSL is written in C++, employs MPI for processes communications, and its DSL is embedded in Python. Applications can be executed in the Python context or translated in a parallel program that can be compiled for a target architecture (e.g., a cluster for interconnected multi-core machines). To handle and store datasets, the framework requires a distributed file system (e.g., NFS). Like other abstraction layers we have seen so far, the library provides parallel containers (i.e., data structures) to store inputs and, typically, they are matrices describing terrains and parallel patterns (e.g., stencils) to be customized and applied on

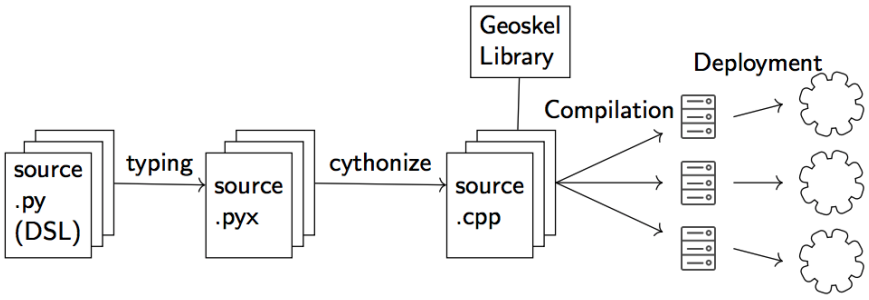

Figure 1.2: Derivation process from the DSL source code to the parallel program

parallel data structures. The derivation from Python to C++ is based on the Cython library. A Cython code (.pyx) can produce external modules than can be used in a standard C++ program using the GeoSkelSL C++ run-time system. The derivation process from the DSL source code to the parallel program is shown in Figure 1.2.

Chapter 2

FastFlow

FastFlow [6, 7, 8, 9] 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, provisioning abstractions, and a carefully designed Run-Time System (RTS). *FastFlow* results in a structured parallel programming library[1] composed of multiple layers of abstraction and relative API available to programmers. It was developed to fully support highly efficient stream parallel computations on heterogeneous multi/many-cores with the general aim of targeting three important challenges: performance, programmability, and portability, also known as "the 3P". *FastFlow* comes as a C++ header-only template library and in the last decade, it has been evolving significantly. From the early versions (in particular versions 1 and 2), the user can model parallel applications as a structured directed graph of concurrent processing nodes. Precisely, the concurrency graph is built by assembling sequential, parallel building-blocks and higher-level parallel patterns (e.g. pipeline, farm, divide&conquer, stencil, parallel-for, map, map+reduce, etc.). *FastFlow*'s efficiency derives firstly from the optimized implementation of the base communication and synchronization techniques [33].

The latest version of *FastFlow* (version 3), has been released in 2019 [34]. This version is the one considered in this thesis. The lower-level software layers have been redesigned, and the concept of *building-block* introduced. In addition to the *farm* and pipeline core components of the previous releases, two new building-blocks have been added, namely *all-to-all* and *node combiner*. Furthermore, a new software component called *concurrency graph transformer* is now part of the *FastFlow* software stack. Such a layer is in charge of providing mechanisms for refactoring the user-defined graph introducing optimizations, and enhancing the

[1]The library is currently released open-source under the LGPLv3 license: https://github.com/fastflow/fastflow

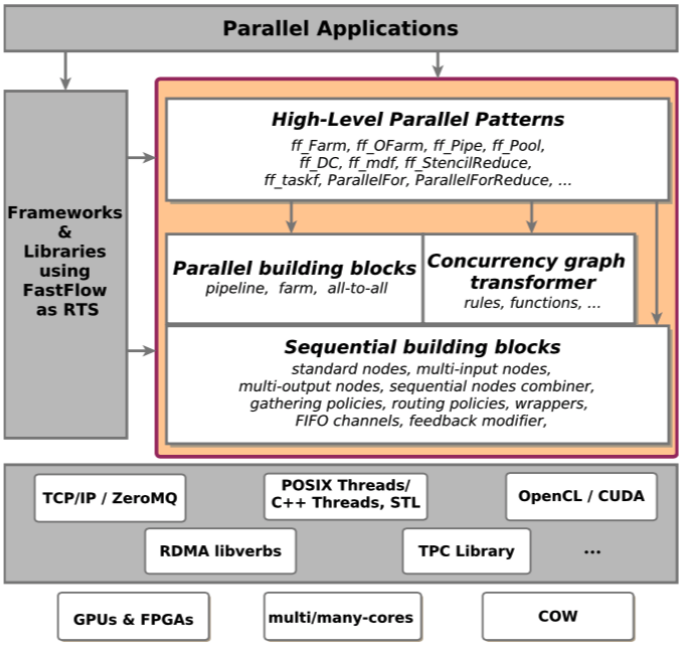

Figure 2.1: *FastFlow* version 3, software layers

performance portability of the applications. The overall layers structure of version 3 is sketched in Figure 2.1.

2.1 Library usage

As briefly described at the beginning of this chapter, the library's provides the programmers with various modular ready-to-use stream-parallel and data-parallel patterns. Those modules can be composed and customized (i.e., extending the library public C++ classes) to build complex parallel programs. Therefore, a generic parallel application is expressed by connecting patterns and building-blocks in a data-flow pipeline (usually the outermost pattern). The library's design allows the user to either write an application from scratch or accelerate specified parts of an existing codebase. Once the graph has been built, it can be executed by invoking the `run` method of the external object and subsequently the `wait` method to wait for the program termination. Alternatively, the library provides a method (`run_and_wait_end`) which combines *run* and *wait* calls, making the execution of the data-flow graph synchronous. Both style of usage are illustrated in Figure 2.2. On the left-hand side of Figure 2.2, a pipeline of four stages is created at a given point and is executed synchronously, waiting for its termination. Instead, on the right-hand side of the same figure, the execution of

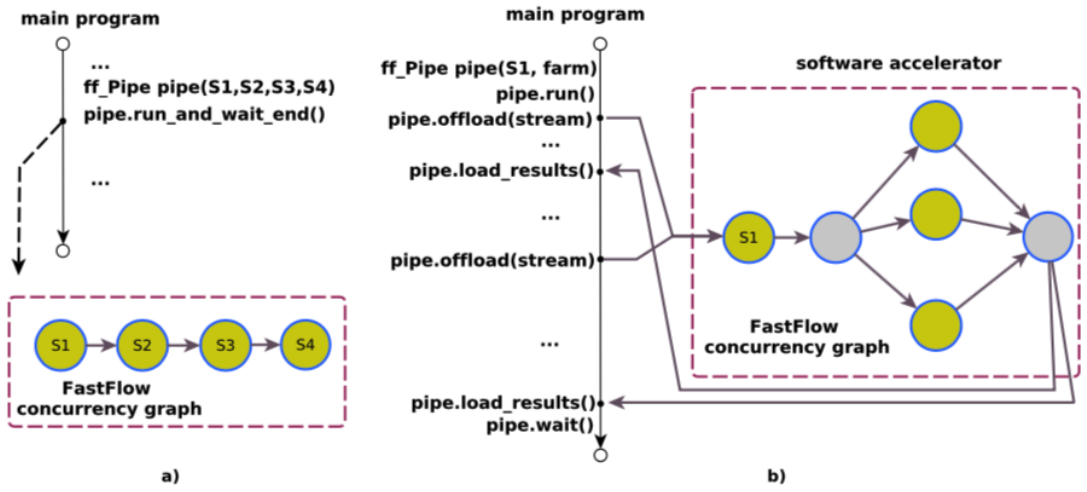

Figure 2.2: **a)** Standard usage of the *FastFlow* library: build the data-flow graph and synchronously runs it. **b)** *FastFlow* used as *software accelerator*, data are then fed to the data-flow graph directly by the main thread.

some user-defined kernels is offloaded to a parallel unit. In our example, there is a pipeline containing a sequential node and another parallel component (farm), fed from the main thread instantiating the accelerator. This mode mimics what happens when using hardware accelerators.

In order to give the flavor of how a simple program can be approached in different ways using *FastFlow*, a straightforward example is going to be presented in different implementations. On Listing 2.1, the farm building-block is employed to compute $\sum_{i=0}^{N} F(V[i])$ in parallel, where F is a user-defined function, and V is a vector of size N. All the items of the collection V are streamed towards the pool of farm Workers by the farm emitter (i.e., Source node); then the reduction phase is computed by the farm Collector (i.e., Sink node). Note the final output value of the Source node, the *EOS*, which denotes the end of the stream, and it is used to terminate all the downstream concurrent entities in the graph. The use of the low-level farm building-block allows the programmer to have complete control over the parallelization and customize the program quickly.

```
1 struct Source: ff_node_t<float> {
2     Source(std::vector<float>&D):V(V){}
3     float* svc(float *) {
4         for(size_t i=0;i<D.size();++i) ff_send_out(&V[i]);
5         return EOS; // End-Of-Stream
6     }
7     std::vector<float>& V;
```

```
};
struct Worker: ff_node_t<float> {
    float* svc(float* in) { return F(*in);  }
};
struct Sink: ff_node_t<float> {
    float* svc(float *in) { sum +=*in; return GO_ON; }
    float sum = 0.0;
};
int main(int argc, char *argv[]) {
   ...
    Source S(V); // Source object
    std::vector<ff_node*> W; // pool of 4 Workers
    for(size_t i=0;i<4;++i) W.push_back(new Worker());
    Sink R; // Sink object
    ff_farm farm(W,S,R); // farm building block
    farm.run_and_wait_end(); // run and wait termination
    ...
}
```

Listing 2.1: Parallelization with *FastFlow*'s farm low level building-block of $\sum_{i=0}^{N} F(V[i])$

An alternative approach employing a high-level parallel patterns, requires less programming effort by abstracting all the low-level parallel modules. The very same computation of Listing 2.1 is re-implemented in Listing 2.2 using the high-level *ParallelForReduce* pattern.

```
int main(int argc, char *argv[]) {
   ...
    ParallelForReduce<float> pfr; // creating the pattern
    float sum{0.0}; // reduction variable
    pfr.parallel_reduce(sum,0,0,D.size(),
        // map+reduce function
        [&](const long i, float& sum) { sum+=Fun(i); },
        [](float& v, const float& e) { v+=e;}, // reduce function
        4); // parallelism degree
    ...
}
```

Listing 2.2: High-level version of the same code presented in Listing 2.1 using *FastFlow*'s *ParallelForReduce*

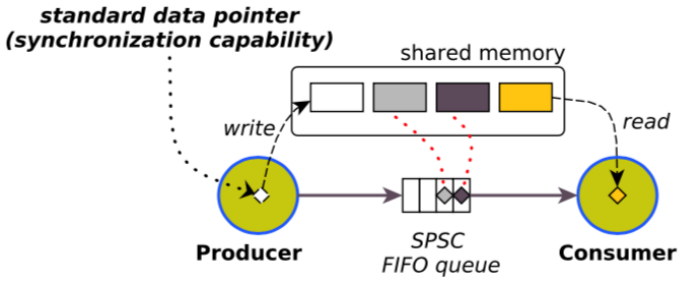

Figure 2.3: Producer-consumer semantics in *FastFlow* through SPSC FIFO channel.

2.2 Building Blocks

The building-blocks are concurrent components, and represent essentials elements for building parallel applications in *FastFlow*. Building-blocks are implemented by a proper composition of *nodes*. Nodes are connected via data-flow channels. Therefore an essential component of the building-block is the lock-free Single-Producer Single-Consumer (SPSC) First-In-First-Out (FIFO) queue implementing the channel [33]. It can have either bounded or unbounded capacity. Those queues carry pointers to data allocated in the shared address space. Sending references equals to transfer the ownership of the data pointed by the reference. The node that receives a reference is supposed to have exclusive access to the referenced data. A simple schema of the producer-consumer semantics implemented in *FastFlow* through SPSC queues is shown in Figure 2.3. Collective communications such as Single-Producer Multi-Consumer (SPMC), Multi-Producer Single-Consumer (MPSC), and Multi-Producer Multi-Consumer (MPMC) are realized through broker nodes employing multiple SPSC queues. Shared concurrent passive data structures are avoided to prevent any memory fence introduced by mutual exclusion/locking instructions and reduce the cache-coherence memory traffic. Default emitter and collector nodes of the farm building-block are broker nodes. Those mediator nodes read from one or more SPSC queues a data reference and write it into one or more SPSC queues. Moreover, using mediator nodes allows customizing the communication behaviors or combine data management with computation. Lastly, employing just load/store memory operation, thus avoiding atomic operation, gives a significant advantage on performances, especially on fine-grained computations.

The building-blocks are either sequential or parallel. Sequential building-blocks are simple nodes, with possible multiple input channels or multiple out channels, or the sequential combination of multiple sequential nodes, implemented by the *combine* building-block. Parallel building-block, instead, are concurrent components built upon a proper assembly of multiple nodes connected by SPSC

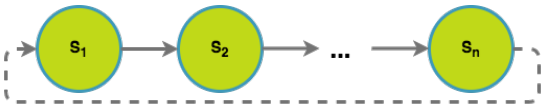

Figure 2.4: Pipeline building-block. The dashed arrow is the optional feedback channel between the last and first stages of the pipeline.

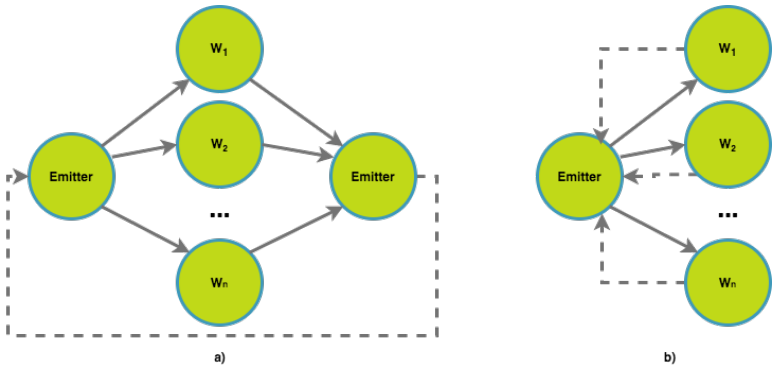

Figure 2.5: Farm building-block. a) Typical configuration of the farm with emitter and collector. The dashed arrow is the optional feedback channel. b) Farm pattern without an explicit collector (a.k.a. *master-worker* configuration).

channels. The fundamental parallel building-blocks are: *pipeline*, *farm* and *all-to-all*. The *pipeline* is used for linking building-blocks and, in general, to express data-flow pipeline parallelism. The pipeline topology is sketched in Figure 2.4. The farm represents the functional replication, in which tasks are dispatched by a centralized entity called *emitter*, and results may be gathered by another entity called *collector*. Both emitter and collector can be extended to follow a user-defined policy for task dispatching and results gathering. The typical topology of a whole farm pattern is shown in Figure 2.5-a. Furthermore, the collector can be collapsed with the emitter, employing feedback channels from workers to the emitter (see Figure 2.5-b). This schema is also called master-worker. The *all-to-all* building-block models functional replication (without a centralized entity as it occurs with the farm) and the shuffle communication pattern between function replicas. This building-block is useful when either the emitter or the collector represents a network bottleneck and needs to be replicated. A description of this parallel pattern is shown in Figure 2.6. To sum up, the reduced set of sequential and parallel building-blocks, along with the ability to be customized and composed, enables the also known as LEGO-style approach to parallel programming [34, 35].

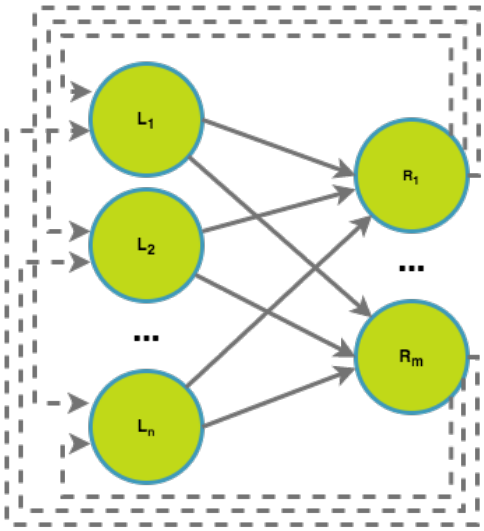

Figure 2.6: All-2-All building-block. The dashed arrows represents the optional feedback channels.

2.3 Frameworks on top of *FastFlow*

WindFlow *WindFlow* [36] is a Data Stream Processing (DaSP) parallel library targeting multi-cores and accelerators. It aims to integrate the algorithmic skeleton methodology in streaming environments. Each skeleton represents a predefined pattern in a high-level abstraction hiding all the structures and complexities of parallel applications. Multiple parallel patterns are ready to use in the *WindFlow* library, more complex patterns can be built by means of composition and nesting of the basic ones. All the patterns are accessible through an high-level fluent-interface API. The latter has a compositional approach centered on the execution entities and how they are connected, instead of expressing how data should be transformed at each step of computation. A generic application is made by defining the corresponding Data-Flow graph, made out of nodes (i.e. operators) and connections as in *FastFlow*. The fundamental operators are two: the *source* operator, which generate the stream as a sequence of homogeneous type items, and the *sink* operator, which absorbs the input stream and possibly stores the received results into storage systems. The streaming items are encapsulated in C++ 17 optional containers, where an undefined object represents the end-of-stream. Operators applying transformation to streams are: *Map*, which applies a transformation to each received element producing exactly a single output; *FlatMap*, which act like *Map* but can produce zero, one or multiple outputs from a single consumed input item; *Filter*, which apply a boolean predicate on each received element, dropping all the elements for which the predicated output values is false;

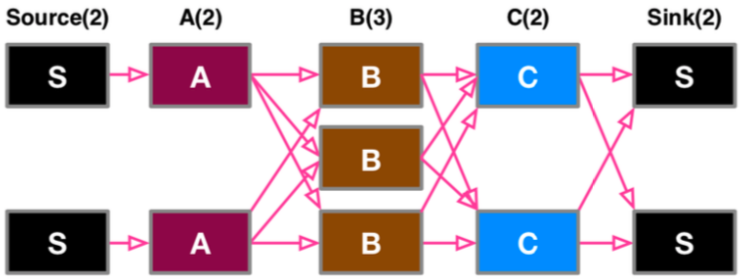

Figure 2.7: *WindFlow* example of *MultiPipe* structure

Accumulator, which executes a fold function on the received inputs partitioned by key. Additionally, the library provide windowed operators, to implement stateful "actors". Windowed operators are: *Keyed Farm*, which executes a function in parallel on different key partitions; *Windowed Farm*, which executes a windowed function in parallel on distinct streaming windows; *Paned Farm*, which acts like Windowed Farm but on overlapping windows; and *Windowed MapReduce* which execute a windowed function exploiting data parallelism. The data-flow graph expressing a generic application is modeled by the *MultiPipe*, which is made of multiple pipelines working in parallel. Communication between operators of stage i and i+1 can be either direct or shuffle. An example of a graph is depicted on Figure 2.7. Under the hood the *MultiPipe* construct is implemented in *FastFlow* nesting *all2-to-all* building-block.

RPLsh RPLsh [37, 38] is a C++ tool that allows: to design a parallel application through patterns, to explore different alternatives of the same automatically refactored application, to evaluate some non-functional properties of patterns, and to generate *FastFlow* code, provided the business logic code. RPLsh is a shell where expression describing patterns can be defined, refactored, and optimized according to well-known predefined rules. For each expression, it is possible to evaluate some non-functional properties, such as latency and service time, and generate working C++ code based on the *FastFlow* library. The supported patterns are: `Seq` the sequential code wrapper, `Comp` the sequential composition, `Pipe`, `Farm`, `Map`, `Reduce` and `DC`, the divide & conquer parallel pattern. Supported refactoring rules, instead, are listed in Table 2.1. About optimization rules, the tools have two possible policies to follow: the one aiming to use the maximal number of parallel resources available and the one aimed to use the minimum number of resources preserving the maximal achievable performance (i.e., not introducing bottleneck in the data-flow graph). Finally, to generate the working C++ code, the user has to supply its business logic code by extending the pure virtual C++ classes provided by RPLsh.

pipe introduction/elimination	$pipe(\Delta_1, \Delta_2) \equiv comp(\Delta_1, \Delta_2)$
farm introduction/elimination	$farm(\Delta) \equiv \Delta$
pipe associativity	$pipe(pipe(\Delta_1, \Delta_2), \Delta_3) \equiv pipe(\Delta_1, pipe(\Delta_2, \Delta_3))$
comp associativity	$comp(comp(\Delta_1, \Delta_2), \Delta_3) \equiv comp(\Delta_1, comp(\Delta_2, \Delta_3))$
map/comp distributivity	$map(comp(\Delta_1, \Delta_2)) \equiv comp(map(\Delta_1, \Delta_2))$
map/pipe distributivity	$map(pipe(\Delta_1, \Delta_2)) \equiv pipe(map(\Delta_1, \Delta_2))$
map collapse	$map(map(\Delta)) \equiv map(\Delta)$
farm collapse	$farm(farm(\Delta) \equiv farm(\Delta)$
pipe deletion	$pipe(\Delta) \equiv \Delta$
comp deletion	$comp(\Delta) \equiv \Delta$
map to divide & conquer	$map(\Delta) \equiv DC(\Delta)$

Table 2.1: RPLsh refactoring rules

Overall, RPLsh supports programmers in all the phases needed to design, implement and tune a structured parallel application.

SPar SPar [1] is a C++ tool providing the user an annotation-based language for modeling stream parallel applications. The annotated sequential code is translated in an intermediate C++ code where calls to the parallel run-time are injected. SPar supports the introduction of pipeline and farm parallel patterns. Using C++ annotations rather than compiler pre-processed directives (i.e., pragma) gives the application developer more flexibility, allowing to annotate type, classes, code blocks, and so on. To give the flavor of SPar, in Listing 2.3, is presented a snippet of a video streaming computation with OpenCV annotated with SPar. In particular, the example shows a two-stage pipeline in which the first stage is replicated (i.e., task-farm) and the second is sequential. `ToStream` identifies the parallel region and the source feeding that. Inside the parallel region two `Stage` annotations specify the pipeline components along with the corresponding dependencies that are captured by the `Input` and `Output` attributes. As anticipated, the first stage may be computed independently over different input objects, so through the `Replicate` annotation, the stage is wrapped in a task-farm pattern.

```
[[spar::ToStream, spar::Input(res,channel,src,S)]] for(;;){
  total_frames++;
  inputVideo >> src;
  if (src.empty()) break;

  [[spar::Stage, spar::Input(res,channel,src,S),
    spar::Output(res),spar::Replicate()]]{
    vector<Mat> spl;
    split(src, spl);
    for (int i =0; i < 3; ++i){
     if (i != channel) spl[i] = Mat::zeros(S, spl[0].type());
    }
    merge(spl, res);
    cv::GaussianBlur(res, res, cv::Size(0, 0), 3);
    cv::addWeighted(res, 1.5, res, -0.5, 0, res);
    Sobel(res,res,-1,1,0,3);
  }

  [[spar::Stage, spar::Input(res)]] {
    outputVideo << res;
  }

}
```

Listing 2.3: Video streaming computation using SPar's annotations. Taken from [1]

2.4 Legacy distributed-memory support

In the early stages of *FastFlow*, to scale to thousands of cores and exploit huge amounts of memory, a run-time system was introduced targeting distributed-memory platforms (i.e., clusters of interconnected machines)[32]. With this support, the framework turned out to have a two-tier programming model: (a) lower tier, which represents the conventional shared-memory behavior; (b) upper tier, which implements the coordination among a set of distributed nodes executing the lower tier computations. Specifically, at the lower tier, the programmer designs a typical *FastFlow* graph, utilizing stream parallelism and the shared physical memory. The data-flow graph describing the application, at this level, is implemented using concurrent threads inside a single process abstraction. Consequently, multiple processes can be connected using suitable communication patterns over the network. The upper tier offers such communication patterns. Thus, to commu-

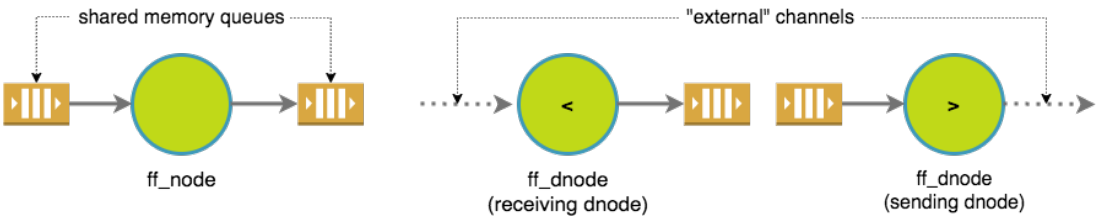

Figure 2.8: *FastFlow*'s `ff_node` vs `ff_dnode`

nicate to other processes, edge-nodes of each *FastFlow* application have to be defined as `ff_dnode`. A `ff_dnode` is actually a `ff_node` in which a channel, either input or output, is external, i.e., connecting the edge-node with one or more edge-nodes of other *FastFlow* processes. No memory is shared among the processes, even if they are running on the same host. All inter-process interactions have to be implemented using explicit communications, which are offered by *FastFlow* RTS and are completely transparent to the user due to the careful design of the `ff_dnode`. In Figure 2.8 are shown the differences between `ff_node` and `ff_dnode`, in terms of communication mechanisms. Note that a distributed node having both input and output external channel is not supported since the minimal pure *FastFlow* application is composed of at least two nodes. The transport layer used for the external channel is ZeroMQ [39], which is an LGPL open-source communication library supporting various transports methods (intra/inter-process, TCP/IP, UDP multicast, etc.). It also offers an asynchronous communication model that facilitates complex asynchronous message-passing networks, ensuring good performances. In particular, all the external communication are built on top of ZeroMQ using the `ROUTER` socket, which permits to route a message to a specified recipient provided the peer identifier, and the `DEALER` socket, which is used for fair queuing on input and for performing load-balancing on output. Multiple communication patterns can be employed specializing the *FastFlow* edge nodes. The communication collectives provided in the second tier are summarized in Table 2.2.

Each communication pattern has a unique identifier, i.e., the `channel-name` and each `ff_dnode` mast have an identifier, not necessarily unique for the external channel in the range $\{x|0 \leq x < \#peers\}$. Hence, the tag `channel-name:dnode-id` is used as a unique identifier for the peer connection. The data serialization is handled trough two virtual methods of `ff_dnode` which implement zero-copy marshalling and unmarshalling.

This distributed support is now obsolete since it has been discontinued in more recent releases of the *FastFlow* library.

unicast	unidirectional point-to-point communication between two peers
broasdcast	sends the same input data to all connected peers
scatter	sends different parts of the input data to all connected peers
onDemand	the input data is sent to one of the connected peers, the choice of which is taken at run-time on the basis of the actual work-load (typically implemented as request-reply protocol)
fromAll	collects different parts of the data from all connected peers combining them in a single data item (also knows as *all-gather*)
fromAny	collects one data item from one of the connected peers

Table 2.2: Communication collectives available in the legacy distributed-memory support of

Chapter 3

Design of the *FastFlow* distributed RTS

3.1 Goals

At the beginning, *FastFlow* was designed to target in an efficient manner shared-memory shared-cache multi-core architectures only. Subsequently, as described in Section 2.4, a distributed-memory support was introduced in the library. Transforming a shared-memory application to a distributed-memory application using the legacy approach requires multiple changes to the code base by the programmers. Every node communicating to other processes, also knows as edge-nodes, must be partially re-implemented using the `ff_dnode` interface. This requires also the implementation of the marshalling and un-marshalling functions for the type that need to be serialized, and the explicit identification of any collective communication among edge nodes. Furthermore, the launch of the processes and the internal setup phases are left to the programmer. Starting from *FastFlow* version 3, the distributed run-time has been discontinued.

In this thesis, we propose a new distributed-memory RTS for the latest version of *FastFlow*. The primary goal of such new RTS is to make the transition from a single computing node to multiple nodes as simple and transparent as possible for the the programmer. In particular, we wish to reduce the changes that need to be applied to the shared-memory FastFlow version of an application to make it capable of being executed in a distributed platform. Therefore, starting from a fully-working shared-memory *FastFlow* program, the main changes to be applied by the programmers can be summarized as follows:

1. include the `dff.hpp` header file, and insert a call to `DFF_Init` at the very beginning of `main` function to initialize the distributed RTS;

2. identify suitable *groups of nodes* of the original data-flow graph to be de-

ployed as a separated distributed component; edges nodes of such groups must be annotated;

3. provide a function for serialization/deserialization for each non-standard or user-defined data type flowing between distinct groups of nodes;

4. build a *JSON* configuration file to explicitly provide the *groups to host mapping*; the *JSON* file will be used also by the FastFlow's loader (*dff_run*, described in Chapter 4) for the deployment and execution of the groups of nodes.

Furthermore, by employing the *FastFlow*'s *loader* module we are going to hide most of the complexity required to launch multiple remote processes with the right parameters in one shot.

From a technical standpoint, the design aims to be as modular as possible and to reuse the available *FastFlow* building-blocks. The modularity gives us the possibility to add other implementations of any modules or completely replace some components of the whole system with alternative ones based on different building-block compositions. For instance, it will be straightforward to implement the pair of communication nodes (i.e., *sender* and *receiver* described in Section 3.6) by using the MPI[11] primitives. Moreover, by implementing the distributed RTS leveraging the *FastFlow* building-blocks, the resulting implementation is lightweight and makes use of the existing optimizations introduced for the building-blocks.

3.2 Design choices

FastFlow applications can be quite complex in terms of data-flow graphs. The variety of available building-blocks and the possibilities of nesting them together leads to an explosion of possible cases that the RTS needs to take into account and manage. As such, during the design of the distributed RTS, some issues arose due to the complexity of the building-blocks nesting, which forced us to make some design choices to reduce the overall complexity of the implementation. Most of the choices, that we are going to list in the following, can be relaxed in the next releases.

We will call *groups of nodes* (or simply *groups*) the sub-set of nodes of the FastFlow concurrency graphs that will be deployed in a separated process (i.e., the tiniest deployment unit). The primary design choices we made are the following ones:

- We support only groups made from *pipeline* and *all-to-all* parallel building-blocks. We do not consider the *farm* building-block. This is not a particular issue since a farm can be implemented using an *all-to-all*.

- The root building-block of the whole data-flow graph of an application must be a *pipeline*. For instance, if the whole application is expressed as a single *all-to-all*, the latter have to be wrapped by a *pipeline* of just one single stage. From the performance standpoint, this is transparent since no performance degradation is introduced.

- A sequential node cannot mix shared-memory and distributed-memory communications. For example, a node with multiple output channels cannot have some of them in the distributed-memory domain and others in the shared-memory domain. All of them must be in a single domain. Mixing the domain between input and output channels is instead possible. For example, a node having all input channels in the distributed-memory domain can have all output channels either in the distributed-memory or in the shared-memory domain. Vice versa is also possible. Such design choice limits the way an *all-to-all* can be split into multiple groups. Specifically, it is not possible to create a group with a subset of nodes coming from both the first and the second set. Either the whole *all-to-all* is included in the group, or a group can contain only a subset of (or all) nodes of the first/second set.

- To simplify the implementation when considering pipelines, we decided that a group created from a *pipeline* must contain all its stages. Consequently, to split a pipeline into two or more groups, it must be refactored into two or more distinct pipelines.

- Since a group of nodes is a unit of deployment, distinct groups cannot be composed nor nested.

3.3 Group identification

A group is a sub-graph of the original data-flow concurrency graph. In this section, we describe how to define the sub-graphs to create a distributed group. As discussed in the design choices, a group can be created from a *FastFlow*'s building-block, specifically, from a *pipeline* and an *all-to-all*. Into a group can be inserted only nodes that are children of the building-block from which the group was built. From a *pipeline* can be built only one group; the one containing the whole *pipeline*, i.e., all the stages. From a *all-to-all* can be built multiple groups containing just nodes from a single set. A group with nodes from both sets is not possible unless the whole *all-to-all* is inserted in the group.

To properly build a group and to define which part of the base building-block to include, edge nodes of the group should be annotated by using the operator `<<` (or `<<=`) and inserted in the input set and/or output set for the group. A

node with both input and output channels in the shared-memory domain must be annotated both in input and output. To perform data serialization, the RTS needs to know the types of data in input and output. Therefore, the operators require typed references to nodes. Note that all edge nodes must be annotated to build the group. Error during the annotation may lead to undefined behaviors or run-time errors.

The following examples are helpful to understand the principles to define groups by annotating the edge nodes. An empty group cannot be deployed and results in a run-time error.

Example 1. Assume we have an application composed of a pipeline of four sequential stages. The first stage generates tasks, and the last stage consumes tasks. This simple application can be built in multiple ways in *FastFlow*. The simplest one is by employing a single pipeline building-block. A representation of the resulting data-flow graph in *FastFlow*is depicted in Figure 3.1. Now, suppose we want to split this application into two processes that can be deployed in two different machines to exploit their resources. In particular, we want to cut the pipeline into two parts, the first two stages in a process and the other two stages in another process. To do that, we need to build two groups. Our original application is composed of one pipeline, and from the previously described constraints, exactly one group can be created from a single pipeline block. So, we need to refactor the application to have at least two pipelines, useful to create the groups as described before. In particular, since a group created from a pipeline must contain all the stages, we need a pipeline containing the first two stages and another pipeline containing the last two stages. A third pipeline, which represents the topmost one of the application, connects the two pipelines into a single pipeline. The resulting refactored application is reported in Figure 3.2. The applied refactoring does not change the behavior of the application nor the performance in shared memory. At this point, it is possible to create the groups and annotate the input and output nodes for each group. An input (output) node can be any node that has an input (output) channel that links two groups. An input-output node represents a node having inter-process communications both in input and output. In our example, we have just one output node (S_2) belonging to the first group (i.e., left-most pipeline) and just one input node (S_3) belonging to the second group (i.e., the right-most pipeline).

The resulting pure shared-memory *FastFlow* code implementing our application is written down in Listing 3.1. In particular, we instantiate the nodes from line 4 to 7, we create and fill up the pipeline from line 8 to 12, and finally, we run the application at line 14.

Instead, the shared-memory version is reported in Listing 3.2. First of all, note the different include file at line 1 in order to use the distributed memory

features and the call to `DFF_Init` at line 4, required to setting up the distributed run-time system. Then we build the graph composed by 3 pipelines as described before and depicted in Figure 3.2. From `p1` we define the group *G1* at line 11 and we specify that the last stage of `p1`, *S2*, has an external output connection, at line 12. In the same way, at line 19, we define the group *G2* from `p2`, and we set *S3* as an input node for the group, i.e., it has an external input connection. The execution statement at line 26 remain unchanged with respect to the shared memory version.

```
1  #include <ff/ff.hpp>
2  ...
3  int main(int argc, char *argv[]){
4    S1 stage1(params);
5    S2 stage2(params);
6    S3 stage3(params);
7    S4 stage4(params);
8    ff_pipeline myPipe;
9    myPipe.add_stage(&stage1);
10   myPipe.add_stage(&stage2);
11   myPipe.add_stage(&stage3);
12   myPipe.add_stage(&stage4);
13
14   myPipe.run_and_wait_end();
15   return 0;
16 }
```

Listing 3.1: Pure shared-memory pipeline of 4 stages in *FastFlow*

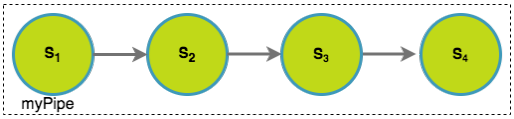

Figure 3.1: Plain pipeline of 4 stages.

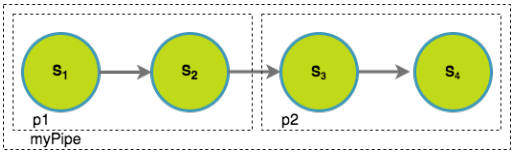

Figure 3.2: Refactored Pipeline of Figure 3.1

```
1  #include <ff/dff.hpp>
2  ...
3  int main(int argc, char *argv[]){
4    DFF_Init(argc, argv);
5
6    S1 stage1(params);
7    S2 stage2(params);
8    ff_pipeline p1;
9    p1.add_stage(&stage1);
10   p1.add_stage(&stage2);
11   auto g1 = p1.createGroup("G1");
12   g1.out << &stage2;
13
14   S3 stage3(params);
15   S4 stage4(params);
16   ff_pipeline p2;
17   p2.add_stage(&stage3);
18   p2.add_stage(&stage4);
19   auto g2 = p2.createGroup("G2");
20   g2.in << &stage3;
21
22   ff_pipeline myPipe;
23   myPipe.add_stage(&p1);
24   myPipe.add_stage(&p2);
25
26   myPipe.run_and_wait_end();
27   return 0;
28 }
```

Listing 3.2: Distributed-memory version of the application in Listing 3.1

Example 2. We consider now a second application composed of a single *all-to-all* with three workers in the left set (a.k.a. first set) and two workers in the right set (a.k.a. second set). Nodes of the first set generate tasks and nodes of the second set consume tasks.

```
#include <ff/ff.hpp>
...
int main(int argc, char *argv[]){
 ff_a2a myA2a;
 vector<LWorker*> firstSet;
 vector<RWorker*> secondSet;

 for(int i=0; i<3; i++)
   firstSet.push_back(new LWorker
    (param));

 for(int i=0; i<2; i++)
   secondSet.push_back(new
    RWorker(param));

 myA2a.add_firstset(firstSet);
 myA2a.add_secondset(secondSet);

 myA2a.run_and_wait_end();
 return 0;
}
```

Listing 3.3: Pure shared-memory pipeline of 4 stages in *FastFlow*

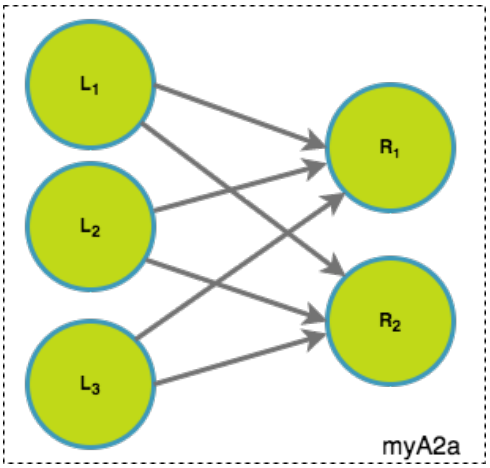

Figure 3.3: Plain A2A of 4 stages.

```
#include <ff/dff.hpp>
...
int main(int argc, char *argv[]){
 DFF_Init(argc, argv);

 ff_a2a myA2a;
 vector<LWorker*> firstSet;
 vector<RWorker*> secondSet;

 auto l1=myA2a.createGroup("L1");
 auto l2=myA2a.createGroup("L2");
 auto r=myA2a.createGroup("R");

 for(int i=0; i<3; i++)
   firstSet.push_back(new LWorker
    (param));

 for(int i=0; i<2; i++) {
   auto w = new RWorker(param);
   secondSet.push_back(w);
   r.in << w;
 }

 l1.out << firstSet[0]
        << firstSet[2];
 l2.out << firstSet[1];

 myA2a.add_firstset(firstSet);
 myA2a.add_secondset(secondSet);

 ff_pipeline myPipe;
 myPipe.add_stage(&myA2a);
 myPipe.run_and_wait_end();
 return 0;
}
```

Listing 3.4: Distributed-memory version of the application in Listing 3.1

We wish to split up the application into three distinct groups: one group containing the first and last node of the first set, one group containing just the second node of the first set, and a group for all the nodes of the second set. Since we can create multiple groups from a single *all-to-all* building-block, no code refactoring is needed in this case. We only need to insert the *all-to-all* building-block into a pipeline since we require that every distributed memory application must be a pipeline. The resulting application is sketched in Figure 3.3. As in the previous example, we describe and compare the resulting *FastFlow* shared-memory and distributed-memory code, respectively in Listing 3.3 and in Listing 3.4.

The shared-memory code base is very straightforward: we instantiate the first set and second set workers in the body of for loop at line 8 and line 11, respectively. Then we add the workers to the instantiated `ff_a2a` building-block at line 14-15. Finally, we run the application by calling `run_and_wait_end`. Apart from the include and the call to `DFF_Init`, the distributed memory version differs at lines 11-13 to define the three groups from the same `myA2a` building-block. At line 20 to annotate all the workers of the second set belonging to group R as input nodes, lines 23-25 annotate nodes of the first set as output nodes in the right group (i.e., *L1* and *L2*), and finally lines 30-32 insert our *all-to-all* in a single-stage *pipeline*.

So far, we described how to construct the executable to work in SPMD style. All the groups require the same binary with the right parameters to run. The parameters are two, and they are passed through the command-line and handled by the `DFF_Init` function at the beginning of the program. `DFF_Init` remove from the argc and argv variables, non-positional parameters required by the run-time preserving the ordering of other positional parameters required by the business logic application and defined by the programmer. The two parameters are: (a) `--DFF_GName` used to pass the name of the group to run (similar to the rank in MPI), and (b) `--DFF_Config` used to pass the path of a configuration file. The configuration file is a JSON text file containing the list of all the groups composing the application, identified by their name, along with meta-data. This meta-data contains the endpoint (i.e., address:port pair) in which the group is listening for incoming connections and the list of groups to connect to. The file structure is not complex; in Listing 3.5 and in Listing 3.6 are reported the configuration file for the Example 1 and Example 2, respectively. Briefly, the main object has a single fields called `groups` which contains an array of objects; each object has the `name` field which is a *string* and is a required field, the endpoint represented by a *string*, and the output connection list (i.e., `OConn`) represented by an *array* of *strings* which contains the name of groups to connect to.
Finally, to execute the whole application, we run the binary n times, where n is the number of groups with the correct parameters. For instance, to run the application presented in Example 1 on a single machine, the following commands can be used:

```
$ ./example1 pParam1 pParam2 --DFF_GName=G1 --DFF_Config=example1.json
```

and on another shell:

```
$ ./example1 pParam1 pParam2 --DFF_GName=G2 --DFF_Config=example1.json
```

```
{
 "groups" : [
  {
   "endpoint" : "localhost:8004",
   "name" : "G1",
   "OConn" : ["G2"]
  },
  {
   "name" : "G2",
   "endpoint": "localhost:8005"
  }
 ]
}
```

Listing 3.5: Configuration file (JSON) for Example 1.

```
{"groups" : [
  {
   "name" : "L1",
   "endpoint" : "localhost:8004",
   "OConn" : ["R"]
  },
  {
   "name" : "L2",
   "endpoint" : "localhost:8005",
   "OConn" : ["R"]
  },
  {
   "name" : "R",
   "endpoint": "localhost:8006"
  }
 ]
}
```

Listing 3.6: Configuration file (JSON) for Example 2.

3.4 From one to multiple *FastFlow* applications

In this section, we are going to start exploring the internals of the new distributed run-time system. In particular, we describe how a single data-flow graph representing the whole shared-memory application is split into multiple distinct *FastFlow* data-flow graphs representing groups. Each group is a valid *FastFlow* application made of a single *farm* building-block with (possibly) heterogeneous workers. The emitter is a custom communication node that receives tasks from other groups or, instead, is the default farm emitter if the group does not have incoming connections (i.e., the group is a source). Similarly, the collector is a custom communication node that sends tasks to other groups, or, instead, it is not present if the group does not have outgoing connections (i.e., the group is a sink). The farm building-block is used for the group implementation because it has just one instance of each communicating node (sender and receiver). The

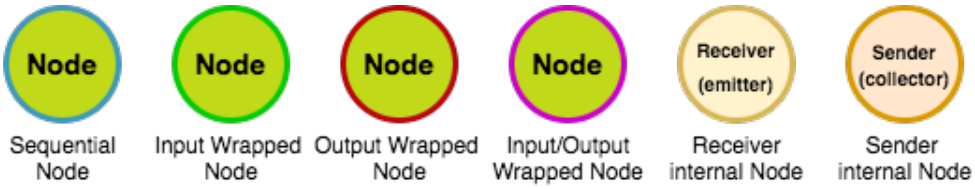

Figure 3.4: Legend showing the types nodes involved in the structure of a process.

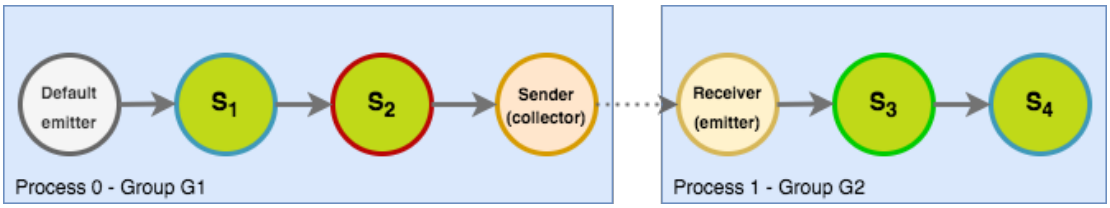

Figure 3.5: Groups internal structure of the Example 1 described in Section 3.3. Solid arrows are shared-memory communications, dashed arrows are distributed-memory communications.

farm workers can be single nodes or a complex nesting/composition of building-blocks for which the programmer has annotated all the inputs and outputs nodes. Indeed, only the nodes that directly communicate through shared-memory with the communication nodes need to be annotated to be wrapped appropriately. Such wrapping of nodes is required to perform encapsulation/decapsulation and serialization/deserialization of input/output data types. These operations will be described with details in the next section. Therefore, for a source group, the only nodes to be annotated are the output ones, while, for a sink group, the only node to be annotated are the input ones. The graphical notation for the types of nodes involved in the internals of a group is shown in Figure 3.4. These notations will be used in the following examples.

Provided that all edge nodes were annotated correctly, a group created from a pipeline will have just one worker for the internal farm implementing the group (i.e., the pipeline itself). Considering the Example 1 of Section 3.3, we will have a farm building-block in each group with one worker. The first group (G1), a source group, has as receiver node the default farm emitter and as sender node the farm collector. The second group (G2), a sink group, has the farm emitter as a receiver node (in this case, the collector is not present). The whole network of nodes of the two groups implementing Example 1 is depicted in Figure 3.5.

A group created from an *all-to-all*, may have multiple workers. In particular, it will have all the workers from a single set that have been annotated correctly.

For instance, suppose to have an *all-to-all* that has in the first set three nested *all-to-all*, each having two nodes in their first set and one node in the second set. To place those three building-blocks in a single group build from the outer *all-to-all*, it is necessary to annotate all the six nodes of input and three nodes of output. The resulting group is composed of a farm of three workers. Each worker is an *all-to-all* where nodes are properly wrapped to be able to exchange messages with communications operators.

To give the flavor of all the rules employed in a complex application, let us consider the network shown in Figure 3.6. The data-flow graph is composed of a pipeline of two stages. The first one is a *all-to-all* and the second one is a sequential node. The *all-to-all* has three building-block in the first set: the first is an *all-to-all* with two nodes in the first set and one in the second set; the second and the third are instead two pipelines of two sequential nodes. The second set of the *all-to-all* is composed of two workers: a pipeline of two sequential stages and a single sequential node. We want to split this concurrent graph in four distinct groups: group 0 with only the nested *all-to-all* (i.e., nodes LL_1, LL_2, LR_1), group 1 with the two nested pipeline of the first set of the outer *all-to-all* (i.e., nodes LL_3, LR_2, LL_4, LR_3), group 2 with all the second set of the same outer *all-to-all* (nodes RL_1, RR_1, R) and group 3 with just the sequential node of the main pipeline (node S). Note that, as described in the previous sections, to create the group of process 3, the sequential node needs to be inserted in a pipeline of just one stage. Supposing that all needed code refactoring and annotation were successful, the network of groups and their internals is depicted in Figure 3.7.

3.5 Data serialization

Communication wrappers are one of the most important modules of the distributed RTS. They allow pure shared-memory *FastFlow* nodes containing business-logic code to communicate with other nodes running on a separate group. As a result, the user can turn a shared-memory application into a distributed-memory application without modifying or adapting the implementation of any operator. The main objectives of wrappers are to perform serialization/encapsulation and decapsulation/deserialization operations.

There are three types of wrappers:

- `WrapperOUT`: this module wraps up nodes that will communicate just in output with other remote nodes. It takes the pointer of data returned by the wrapped node and the destination ID, performs the serialization of the referenced data, builds the message structure by encapsulating all the information and the serialized data, and forwards it to the attached commu-

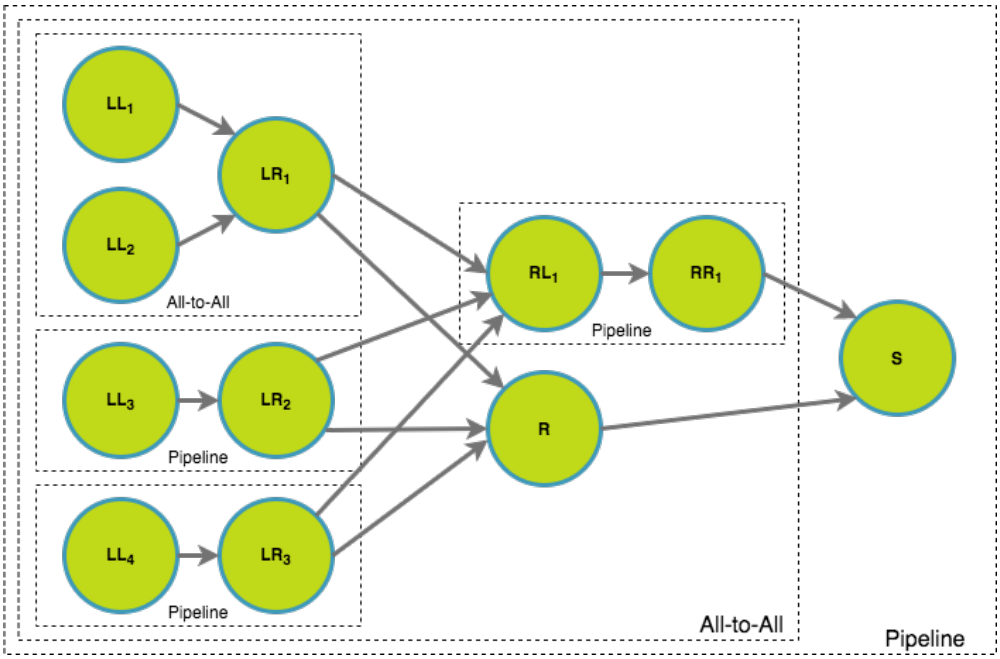

Figure 3.6: Data-flow graph representing a valid complex *FastFlow* application.

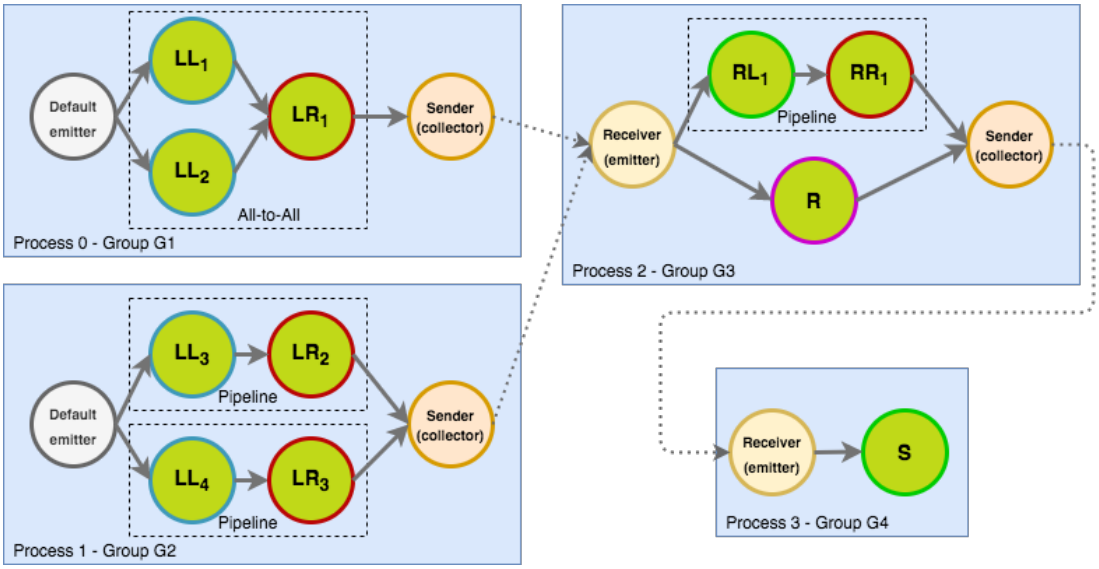

Figure 3.7: Network of distributed groups and their internal structure implementing the *FastFlow* shared-memory concurrent graph depicted in Figure 3.6. Solid arrows are shared-memory communications, dashed arrows are distributed-memory communications.

nication node. The `WrapperOUT` makes the wrapped node a multi-output node regardless of the type of wrapped node. In case the original node was a multi-input, and it is necessary to preserve the type of the node, a *combine* building-block is automatically constructed in which the first node is the original one and the second node is a `Forwarder` node wrapped by the `WrapperOUT`.

- `WrapperIN`: this wrapper encloses a node that receives data in input from other remote nodes. It receives from the communication node (i.e., from the farm's emitter) an encapsulated message, performs the decapsulation and the data deserialization, then invokes the `svc` method of the wrapped node passing the pointer of the deserialized data. From the point of view of the wrapped node, the pointer and all the available meta-data, such as sender ID, are presented as if the sender node were directly attached to the actual node with a shared-memory channel. `WrapperIN` makes the wrapped node a multi-input node. In case the wrapped node is a multi-output node, a *combine* building-block is automatically constructed in which the first node is a `Forwarder` node wrapped by the `WrapperIN` and the second node is the original one.

- `WrapperINOUT`: this module implement the functionalities of the `WrapperIN` and the `WrapperOUT` at the same time. It is used to wrap nodes that communicate both in input and output with remote nodes. For instance, this is the case of the node R of the previous example depicted in Figure 3.7. Those nodes are directly attached in input and output to the communication nodes (i.e., they are a single worker node in the farm implementing the group).

To perform serialization and deserialization operations, wrappers require to detect the types of the wrapped node. Hence, wrappers require to be built from a typed sequential node (`ff_node_t`, `ff_minode_t` and `ff_monode_t`). For this reason, the distributed run-time force the annotation of input and output nodes one by one. This represents one of the main limitations of the proposed support.

All the wrappers support two kinds of serialization/deserialization mechanisms: an automatic and transparent to the user conversion leveraging *cereal*[1][40] library; and a user-guided conversion, which is particularly useful when the data type is contiguous in memory. The first one does not require additional code for

[1]Cereal is a header-only C++11 serialization library. Cereal takes arbitrary data types and reversibly turns them into different representations, such as compact binary encodings, XML, or JSON. Cereal was designed to be fast, lightweight, and easy to extend. It has no external dependencies and can be easily bundled with other code or used standalone.

standard data types. Instead, for user-defined structs and classes, the library requires to implement a template method called *serialize*. An example is provided in Listing 3.7.

```
struct MyDataStructure {
    std::string name; int age; float score;

    template <class Archive>
    void serialize( Archive & ar ){
        ar( name, age, score );
    }
}
```

Listing 3.7: Example of *cereal*'s serialize function of a user defined struct.

The second method gives the possibility to the user to define a custom low-level serialization/deserialization pair of functions or use the default ones for types that are contiguous in memory and for which the size is known at compile time. Custom functions can be provided with the node during the input/output annotations by using the `<<=` operator. The pair of functions for such second method is called *transform* and *finalizer* and are employed for serialization e deserialization purposes, respectively. The signature of the functions can be deduced by the default ones reported in Listing 3.8.

```
template<typename T>
std::pair<char*, size_t> transform(T* object){
    return std::make_pair(object, sizeof(T));
}

template<typename T>
T* finalizer(char* p, size_t){
    return new (p) T;
}
```

Listing 3.8: Default functions for custom serialization

Performance comparison of the two proposed serialization mechanisms is given in Section 5.1.1, both between groups running on the same machine and between groups running in different machines interconnected by 1 Gbit/s network interfaces.

The End-Of-Stream (EOS) message (and all other "special" messages used in *FastFlow*) is treated by the wrapper as in standard *FastFlow* nodes. Specifically, upon receipt of an EOS message, it is forwarded to the next node(s) and then the execution of the wrapper node is terminated.

3.6 Communication nodes

Communication nodes of the group are the only nodes in charge of communicating over the network. They represent gateways for any message that flows between groups. There are two kinds of communication nodes in a group: receivers and senders. Since a group is implemented as a farm building-block, they are implemented as the farm's emitter and collector, respectively. The receiver implements a socket-based server[2] which supports multiple input connections using a single-threaded model exploiting the Unix's *select* System Call. It receives messages from the connected clients and forwards them to the correct recipient by inspecting the message's header fields and accessing the local routing table. The message will be decapsulated later by the wrapper of the recipient node.

The sender of a group is connected to the receivers of other groups through persistent TCP/IP connections. It collects from all the internal nodes the messages that need to be sent to other groups. Like the receiver, by inspecting the header fields of messages, it can send each message to the correct group. The routing protocol is pretty simple; each receiver sends its local routing table to all the connected clients (i.e., sender nodes).

The sender, also, implements a sleep-retry-loop policy for the connection setup in the start-up phase, which does not force to launch the receiver nodes always before the sender nodes.

Message structure The protocol used to communicate over the network is quite simple and uses just one type of low-level message. The message has to carry information about the sender, the destination, the indexing of the original data-flow graph, the size of the serialized data, and the array of byte representing the serialized data. The actual structure of the message is depicted in Table 3.1. All the fields are sent in *Network Byte Order*.

int	int	long	
Sender ID	Channel ID	Size	serialized DATA

Table 3.1: The first field is a integer indicating the ID of the sender, the second field is an integer indicating who is the recipient, the third field is a long integer representing the size of the serialized data, which is transmitted/received immediately after.

No acknowledgment or error correction mechanisms are used at the application level, since TCP provides reliable, ordered, and error-checked delivery of a stream of bytes between pairs of processes.

[2]Sender and receiver support both `AF_INET` and `AF_LOCAL` socket types. Type of socket must be defined at compile time using REMOTE or LOCAL macros, respectively.

Routing protocol Each sender builds its routing table by merging all the tables received by the remote groups (i.e., by the receiver of the remote groups) and the relative socket channel during the phase of the initial handshake. The receivers build their local routing table by knowing the structure of the original data-flow graph and by inspecting the directly attached neighbors. Therefore, no particular protocol is needed to handle routing table transfers; tables are serialized and sent to all the connected peers. An example of how the routing table is built and structured either in the receiver and sender nodes is given in Figure 3.8. The testing of the message routing will be described in Section 5.2 using the shuffle phase of the map-reduce Word Count application.

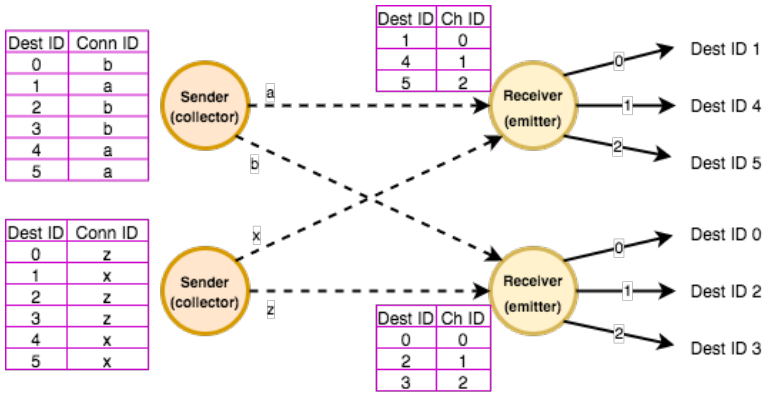

Figure 3.8: The receiver nodes build their local routing table being aware of the attached destination nodes. The list of reachable destination is sent to each client connected (i.e., from each receiver node to each connected sender node). Sender nodes make the association of reachable destination and the socket ID from which they received the list to build their routing table.

Termination Both receiver and sender nodes can handle and propagate *FastFlow*'s EOS message for implementing the termination protocol. EOS messages are represented by a massage whose size is 0. The sender of the group waits until it receives an EOS from all input attached nodes to propagate it to the distributed groups connected. The receiver node of the group waits until an empty message (i.e., the EOS) is received from all the connected remote groups to propagate the *FastFlow* EOS to all attached nodes. This simple protocol is sufficient to ensure a graceful termination of all nodes if the data-flow graph is a direct acyclic graph (DAG). When feedback channel support will be introduced in the distributed RTS (this feature is already in the future work list), additional synchronization mechanisms will be required to handle graceful termination.

Chapter 4

The *dff_run* loader

The chapter aims to present the problems of launching a distributed application composed of several components. Specifically, as a solution to such problems, we describe a software component called *dff_run* that takes care of the start-up phase of the executable resulting from the compilation of a generic *FastFlow* application using the distributed RTS proposed in this thesis.

If the launching phase of an application composed of two groups deployed locally is rather simple, imagine what can happen during the launching phase of an application composed of tens of groups deployed remotely. The setup of the environment, either local or remote, and the launching phases are critical operations that require multiple steps. The list of operations is lengthy and can involve the following actions: dispatch the code base, dependencies, and configuration files to all the target machines; compile the application taking care of each architecture; execute the application with the right parameters; properly collect the outputs of each process by saving it to files for future analysis or by redirecting it to the programmer terminal for real-time monitoring and troubleshooting; kill local or remote processes in case of unresponsive execution. For complex applications using several modules, all the previous operations can be cumbersome to do and error-prone. Most of the time, program scripting helps to automatize the majority of the operations. However, this solution requires an additional effort for the programmer, which has to write a custom and correct script for each application. To relieve the programmer from this responsibility, we have designed and implemented a software tool capable of executing some of the actions required to run a distributed-memory application composed of multiple processes. This software is called `dff_run`, and can be compared in some sense to the well-known MPI equivalent `mpi_run`. Since the main focus of this thesis is to design the distributed-memory run-time system in *FastFlow*, the proposed version of the loader is tailored to *FastFlow* programs, providing some features useful for testing

and troubleshooting. As previously stated, the number of operations that a complete loader module can support is vast. Therefore, there are many opportunities for improvements and the introduction of new features. The loader can perform the remote execution of processes taking care of the parameters, concentrate the standard output of all the processes in a single shell, and kill stuck processes. The loader does not currently manage to dispatch the codebase and remote compilation. Such missing features are somehow not needed in those platforms in which all the machines are homogeneous (i.e., a recompilation in each machine is not required), and the file system is shared between all the machines (i.e., the dispatch of the binaries is automatically handled under the hood). Local processes are forked directly by the loader, while remote processes are executed forking a Secure Shell (ssh) process with the needed commands to execute a process remotely[1].

Concretely, the loader takes in input the JSON configuration file of the application, the path to the binary, the positional arguments to be passed to each process. Concerning the output produced by the processes, *dff_run* shows the standard output of the specified groups[2] and the elapsed time of the whole application, useful for benchmarking applications. For instance, to execute the whole application of Example 1 of the previous section, the only command is:

```
$ ./dff_run -V -f example1.json ./example1 pParam1 pParam2
```

A logical schema of how the loader works is depicted in Figure 4.1.

4.1 Configuration files

Configuration files are crucial for executing distributed applications. As seen before, they are used by all the processes in order to run correctly and create connections to other possibly remote processes. It is also crucial for the loader to customize the launching of every single process composing the application. The chosen file format is JSON [41], which is either human-readable and the syntax results relatively compact. Configuration files are parsed by the run-time system and by the loader using *Cereal* library, already employed for the "automatic" serialization of messages in the RTS. We preferred *Cereal* to other existing libraries for the JSON parsing because of its simplicity, flexibility and to avoid introducing other dependencies to the whole system.

The configuration files include the definition of all the groups composing the application it refers to. For each of them, is it possible to define the following

[1] Currently only the loader supports ssh key-based authentication.

[2] To view the output of all the processes, it is possible to use the `-V` flag, instead to specify only some groups it is possible to use the `-v` followed by the list of group ids separated by comma.

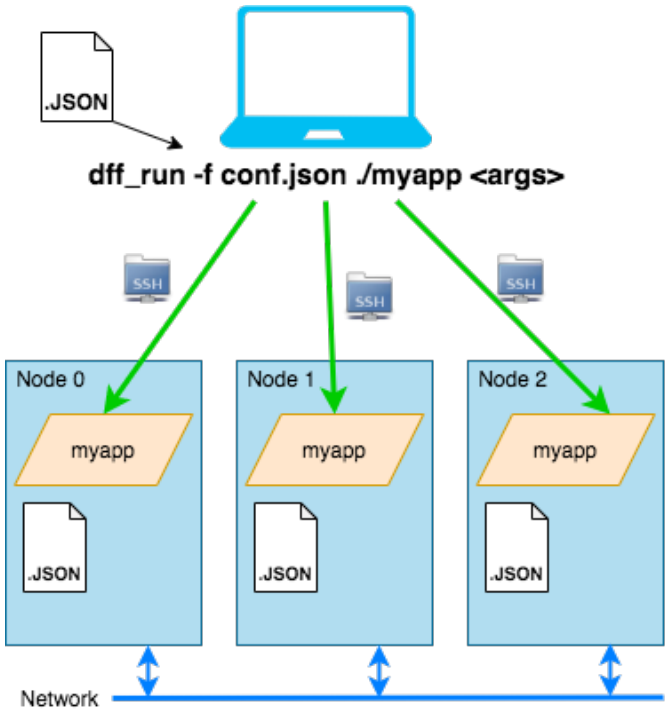

Figure 4.1: The *dff_run FastFlow* loader.

metadata:

- **name**: the name uniquely identifies a group. The name of a group is given during group creation, invoking the `createGroup` method of a building-block. The name is case sensitive and must be unique. This field is strictly required.

- **endpoint**: a pair of values ¡hostname, port¿ that has a dual utility. The first one is to set the endpoint to which the process is going to listen for incoming messages. The second one is to give directives to the loader of which machine/host has to be used to deploy that particular group. For instance, if the hostname is `localhost` or is equal to the one in which the loader is running, the `dff_run` will fork the process locally; instead, if the hostname is different, it will fork a remote process using the Secure Shell (ssh). Port value is ignored for source groups, since they do not have any receiver. This field can be omitted for source groups running in localhost.

- **OConn**: the list of groups the current group is connected to. This field can be omitted for sink groups since they do not have any outgoing connection by definition and do not run any sender node inside the process.

- **preCmd**: it represents a command that can be executed before the main one in the launching phase. For instance. it is useful for execute the processes using `taskset` or `time` commands. Suppose we want to cap all the processes to some given cores, by defining for each group a taskset directive, such as "taskset -c 0,1,2,3" in the `preCmd` field of the configuration file, the loader will automatically execute our application concatenating in front the taskset directive.

A sample configuration file of an application made of four processes is reported in Listing 4.1. Note that the groups `G2` and `G3` are using the `preCmd` directive to execute the process only on the specified cores.

```
{"groups" : [
    {
      "endpoint": "repara:0",               /* port is ignored */
      "name" : "G1",
      "OConn" : ["G2", "G3"]
    }, {
      "name" : "G2",
      "preCmd" : "taskset -c 25,26,27,28",
      "endpoint": "repara:8042",
      "OConn" : ["G4"]
    }, {
      "name" : "G3",
      "preCmd" : "taskset -c 0,8,16,24",
      "endpoint": "titanic:8043",
      "OConn" : ["G4"]
    }, {
      "name" : "G4",
      "endpoint": "repara:8044"
    }
  ]
}
```

Listing 4.1: Sample Configuration file in JSON text format of a 4 processes application.

Furthermore, the JSON format gives also the possibility to introduce other fields in case new features will be introduced.

4.2 Possible extensions

As stated above at the beginning of this chapter, the loader may include a vast number of features useful for deploying the multiple instances of our application. Those features help in making the critical phase of setting up and launching a distributed application painless. Some of the functionalities that can be introduced are:

- Remote compilation: the idea is to automatically dispatch the codebase of the application to all the nodes in which the application will run. Get information about the toolchain available for compile the source code. Perform the compilation on the codebase exploiting characteristics of each architecture, such as hardware accelerators, presence of particular network cards, enabling conditional compilation, and tuning of parameters at compile time.

- Handle CPU affinity: Deploying multiple applications in a single node with the same static CPU affinity policy leads to an inevitable degradation of performances since multiple processes will run on the same CPUs, not exploiting all the ones available. The first trivial solution could be disabling the default mapping of *FastFlow*, leveraging the operating system policy of thread mapping. Also, this case can lead to non-optimal performances. Hence, is required an entity that manages the CPU affinity of the processes deployed on the same machine. Multiple possible practical solutions can be investigated to squeeze the best performances from each shared machine.

- Automatic process/host mapping: The current version of the loader require the user to define the mapping between processes and the host of execution, enabling a high level of customization. However, there are situations in which we want to scale out some parts of an application. So we would like to have an automatic function that defines multiple replicas of a group and deploys them in a list of available machines. The automatic procedure will relieve the programmer to rewrite configuration files and refactor the application to support multiple replicas.

- Relative Paths support: Actually, the loader can handle just absolute paths in remote execution, which results quite limiting for complex applications deployed in several remote machines where the file-system is not synchronized (e.g., NFS). The wish is to specify a path for a group taking into account the machine of deployment to locate the binaries correctly on the remote machine.

Chapter 5

Evaluation

In this Chapter we present, through experiments, the tests we performed to evaluate the distributed *FastFlow* RTS described n Chapter **??**. All tests were carried out in two distinct environments. The first one is composed of two heterogeneous machines gigabit interconnected: *Repara* and *Titanic*. *Repara* is an Intel Xeon server composed of two sockets, each one of 12 cores 2-way hyperthreading, for a total number of 24 physical cores and 48 logical ones (i.e., hardware contexts). Instead, *Titanic* is an AMD EPYC composed of two sockets, each one of 32 cores able to run two distinct threads for a total number of 64 cores and 128 hardware contexts. Both machines belong to the Parallel Programming Models group of the Department of Computer Science of the University of Pisa.

The second test environment is composed of a flat gigabit interconnected cluster of 16 Intel Xeon machines called *openhpc2*. Each machine has two CPU sockets, each CPU has 10 cores 2-way hyper-threading, for a total of 40 hardware contexts. *openhpc2* cluster is hosted by the University of Pisa, in the new Green Data Center located in San Piero a Grado (PI).

5.1 Parametric benchmark

This application represents a synthetic program, fully parametrizable, to evaluate some specific aspects of the proposed distributed-memory RTS. The application consists of an *all-to-all* in which the workers of each set are homogeneous and the number of replicas is given as a parameter. The generic worker from the first set generates tasks, whereas the worker from the second set consumes tasks. The execution time to produce a task and the execution time to consume a task are provided as command-line arguments. Furthermore, it is possible to specify the size of the message that will be transferred between sources and sinks. Finally, the last parameter is the total number of tasks to be generated. Hence, the number of

Sink execution time	Time spent by the sink to consume a single task
Source execution time	Time spent by the source to generate a single task
Source parallelism degree	Number of replicas of sources
Sink parallelism degree	Number of replicas of sinks
# Task	Total number of task to generate. Each source will generate $\sim \frac{\#Tasks}{\lvert Sources \rvert}$
sizeof(Type)	Size of the exchange data type

Table 5.1: Parameters of the Parametric Benchmark

tasks generated per worker is approximately $\frac{|Tasks|}{|Sources|}$. Computing loads are always mimicked through the usage of an active wait function.

5.1.1 Point-to-point performance test

This test aims to benchmark point-to-point performance. It is based on the parametric benchmark described previously with a single producer and a single consumer, communicating entirely on a distributed-memory channel. Therefore, there are only two groups: the first one comprising the producer worker and the sender node introduced by the distributed RTS, the second one comprising the consumer node and the receiver worker introduced by the RTS. In this test, the active wait used to simulate a per-task workload is disabled in both operators. Thus, they operate at full speed. This benchmark allows us to evaluate how our RTS can saturate the available network bandwidth between the two groups and how the two available mechanisms of serialization (Cereal-based serialization vs. manual-based serialization) impact the overall performances. The benchmark has been executed either on a single machine and on two interconnected machines.

In the single machine case, even if it was possible to exploit the `AF_LOCAL` type of socket, it has been used the `AF_INET` socket type, which employs the full TCP/IP stack. The maximal reachable throughput is given by the bandwidth available between CPU and the main memory. The resulting throughput (measured in MegaByte/s) in the single machine configuration is plotted in Figure 5.1, in which we varied the message payload size. The two serialization mechanisms perform the same for small payloads, whereas for bigger messages, manual serialization overcomes the *Cereal*'s one.

The results obtained for the same benchmarks but with the two groups deployed on different interconnected machines (*Repara* and *Titanic*) are reported in Figure 5.2. Even in this case, we show what happens by varying the payload size of the messages. To have a baseline we tested the reachable bandwidth between the two machines, using *netcat*[42]. The commands used to measure the bandwidth reached by *netcat* are the following:

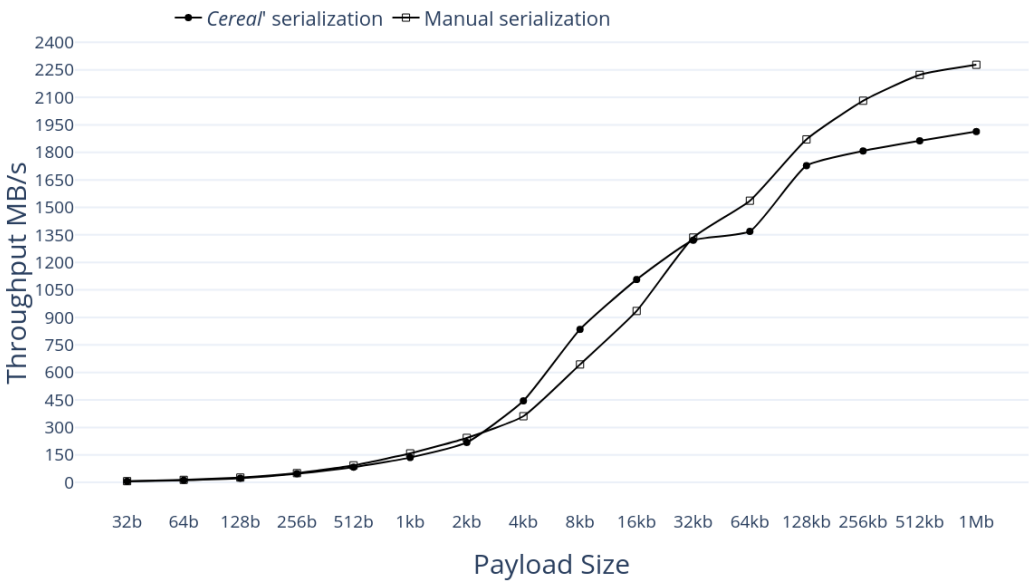

Figure 5.1: Point-to-point performance test on a single machine (Repara) varying the payload size of the messages and the type of serialization.

```
Repara:$ for i in 32 64 128 ...; do dd if=/dev/zero bs=$i count=1000000 | nc -N titanic 8000 ; done

Titanic:$ nc -lk -p 8000 > /dev/null
```

In particular, the results show that for small messages, it is hard to fill the link bandwidth. This is because each message at the application level corresponds to a TCP segment at the transport level and probably a single datagram at the network level. A possible solution to this issue is to introduce buffering at the sender side, i.e., waiting until a certain number of messages or a certain number of bytes are available before sending all of them in a single batch. However, this approach has an impact on the end-to-end latency. Choosing the most suitable value for the number of messages to wait for before sending the batch is not easy and deserves to be studied.

5.1.2 Distributed Farm Emulation

This test mimics a farm whose workers are distributed across multiple machines. It is based on a custom version of the parametric benchmark application, which enables the deployment of multiple replicas of the sinks into multiple processes. In particular, we have a single source representing the emitter of the distributed

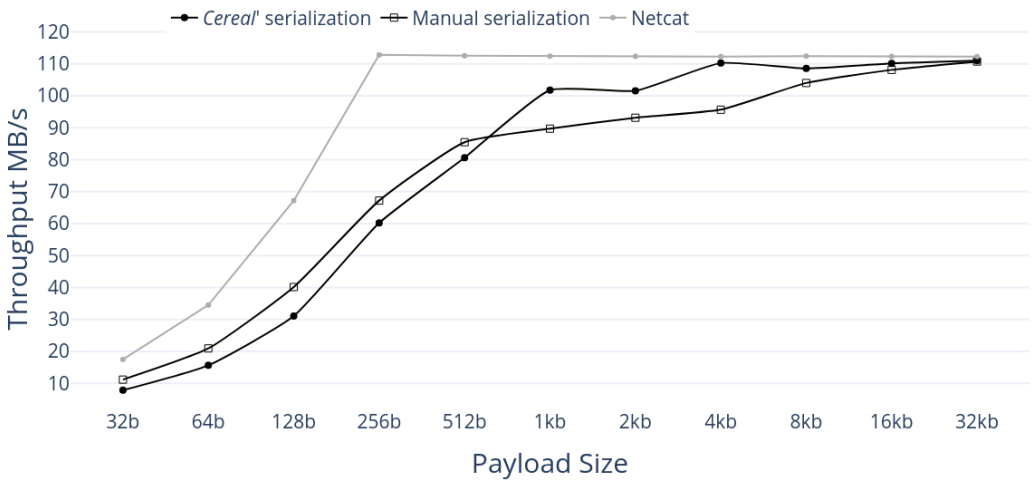

Figure 5.2: Point-to-point performance test between *Repara* and *Titanic* varying the payload size of the messages and the type of serialization. *Netcat*'s performances are used as a baseline to show the maximum attainable throughput between the two machines.

farm. The source generates 100k tasks at full speed (i.e., there is no execution time by the emitter). Each task has a size of 100byte. The workers simulate a computation that takes 2ms for each task. We assume to have a cluster of interconnected machines, each one able to run only two workers. This is a common situation for a cluster of IoT devices or in a cloud-based environment where Virtual Machines (VMs) are usually equipped with a small number of cores. The tests measure the completion time of the same application varying the number of processes, i.e., the total number of workers in the farm. The test has been executed on the *openhpc2* cluster of the University of Pisa, which is made of 16 machines. The resulting execution time are plotted in Figure 5.3. The bars represent the measured execution times, and the line traces the ideal execution time. We reported the average execution time of several runs. From the measured completion times, it is possible to compute the scalability of all the trials. The measured scalability and the ideal one are reported in Figure 5.4. The plot clearly shows that the scalability is very close to the ideal one for a small number of processes and slowly decreases when the number of distributed nodes used increases. This behavior is expected since, by increasing the number of nodes, the RTS needs to do more work to manage and synchronize all nodes.

Now, let us see what happens when increasing the intra-group parallelism

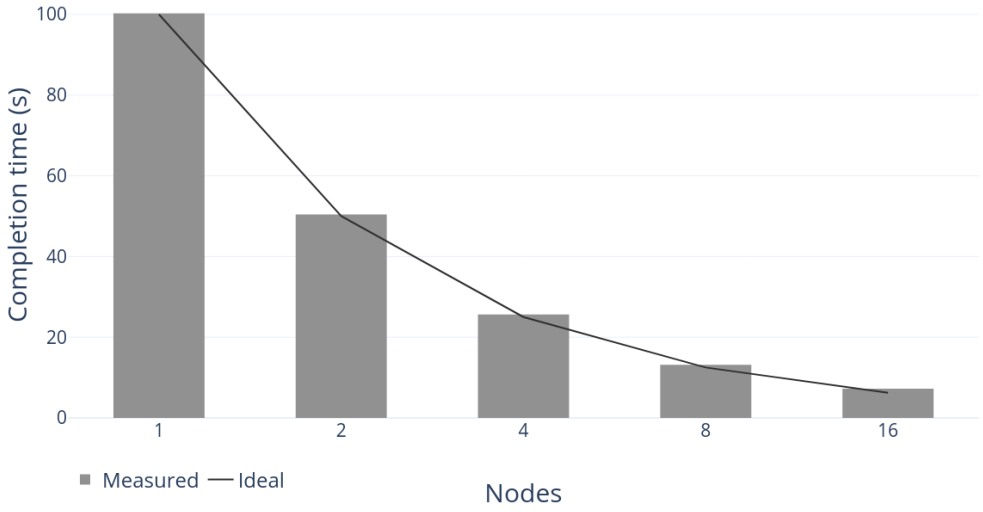

Figure 5.3: Completion time of the farm processing 100k tasks each one taking 2ms of execution time, varying the number of nodes, i.e., worker processes. Each node/process includes 2 actual workers/threads.

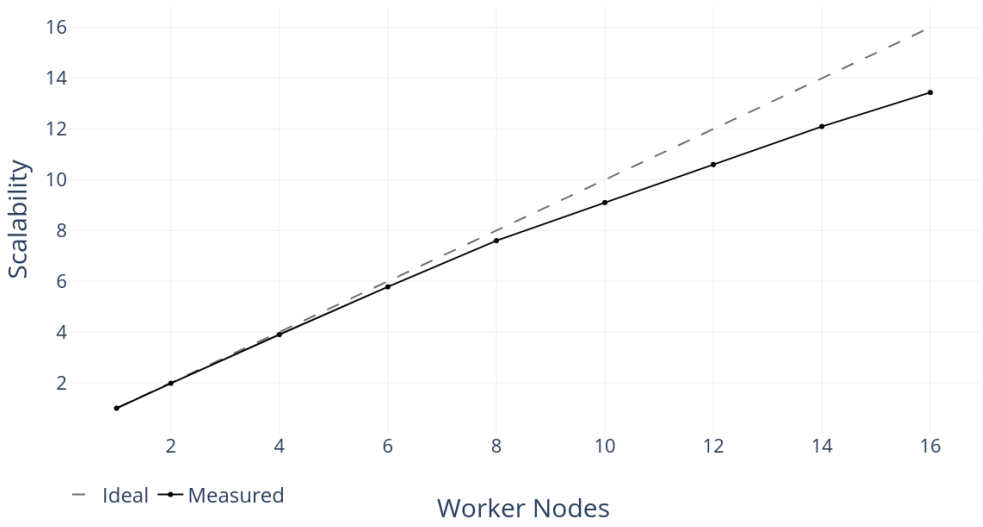

Figure 5.4: Computed scalability of the results showed in Figure 5.3.

Worker Processes	Total threads	Completion Time (s)	Scalability
1	20	10.79	
2	40	5.87	1.80
4	80	3.39	3.18
6	120	2.56	4.20
8	160	2.04	5.27
10	200	1.89	5.69
12	240	1.75	6.13
14	280	1.61	6.69
16	320	1.68	6.40

Table 5.2: Farm benchmark results using 20 threads/worker per process. Each process is running on a distinct node of the cluster. The execution time for each task is 2ms, representing a fine-grained application.

degree. Each machine belonging to the cluster has 20 physical core 2-way hyperthreading. We want to fill all the physical cores of the machine so that the resulting number of workers per group can increase up to 20. This value allows us to exploit all the computing capabilities of the given machine. The completion time and relative scalability of the application presented in the last paragraphs are reported in Table 5.2. The execution time of the single task is set to 2ms. Such value mimics a relative fine-grained application. By increasing the parallelism degree in each group produces better performances (i.e., a reduction in the completion time), at least up to 14 groups/nodes. Table 5.3 presents the results obtained running the same test but increasing the per-task execution time from 2ms to 100ms. This is a scenario that mimics a coarser-grain application. In this case, the trend is almost linear, by doubling the resources produces a cut of 50% in the completion time. Thus, the scalability is very close to the ideal one. A comparison of the scalability of the fine-grained application and the coarse-grain application can is reported in Figure 5.5.

Worker Processes	Total threads	Completion Time (s)	Scalability
1	20	500.91	
2	40	250.90	1.99
4	80	125.89	3.97
6	120	84.20	5.94
8	160	63.40	7.90
10	200	50.90	9.84
12	240	42.52	11.78
14	280	36.59	13.68
16	320	32.09	15.60

Table 5.3: Farm benchmark results using 20 threads per group. Each group is running on a distinct node of the cluster. The execution time for each task is 100ms, representing a coarse-grained application.

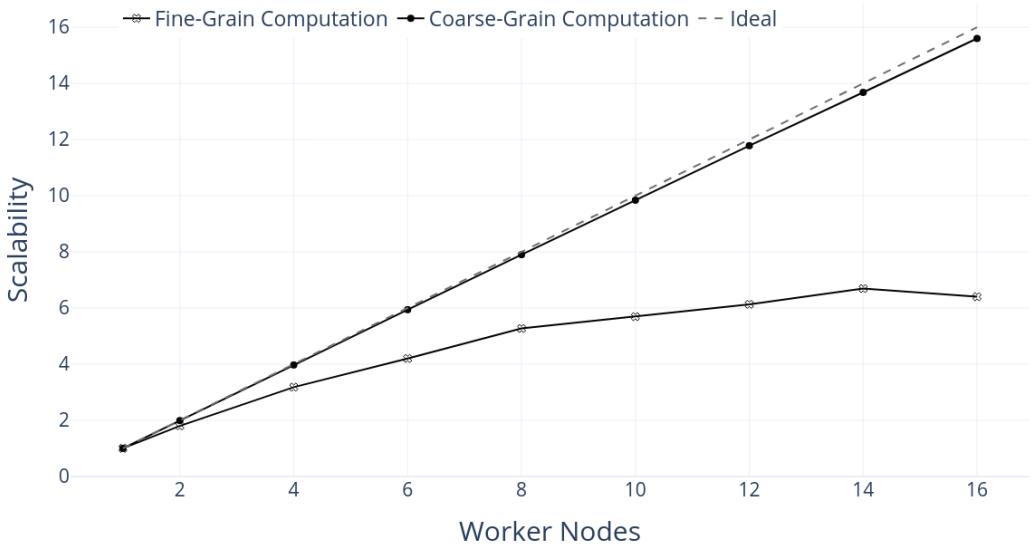

Figure 5.5: Scalability comparison of the fine-grained vs coarse-grained applications (data from Table 5.2 and Table 5.3, respectively.

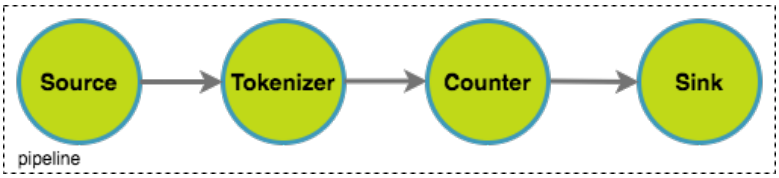

Figure 5.6: Data-Flow graph of the *Word Count* application.

5.2 Word Count

The Word Count (WC) application is an analysis tool to measure word frequencies in documents, books, web pages, etc. In general, it can be applied to any data-set that is a textual collection of words. For each word, the application returns the number of occurrences. Word Count can be modeled as a four-stage pipeline. The first stage is the source which produces a continuous stream of records by reading line-by-line a textual file. The second stage is the *tokenizer* which takes a line from the source and emits a string for each encountered word. It basically splits the line string using the space character as the separator. The third stage is the word counter, which takes single words and counts their occurrences. It has a counter for each word received until that moment; upon arrival of a word, it increments the corresponding counter. The last stage is the sink. It receives all results produced by the word counter, i.e., a word and its number of occurrences up to that instant. The pipeline modeling the application is depicted in Figure 5.6.

In order to make the application more scalable, it is split into two parts: on the one side, source and tokenizer; on the other side, counter and sink. In this way, it is possible to create an *all-to-all* in which we can have a generic number of replicas of the first two stages and the last two ones. The resulting structure of the application is quite simple: string, representing lines, are forwarded from the source to the tokenizer, then the strings, representing words, are hashed by key (i.e., the word) and forwarded to the corresponding counter replica and finally the result forwarded to the sink. Partitioning the stream using the single word as the distribution key does not guarantee load balancing of the workload across the parallel entities of the word counter node since a different frequency characterizes each word in natural languages. However, this version of word count employs the full capabilities of the *all-to-all* building-block and allows us to stress the message routing part of the distributed RTS. The implementation of the tokenizer node is shown in Listing 5.1. The structure of the *FastFlow* Word Count application, employing the *all-to-all* building-block, is sketched in Figure 5.7.

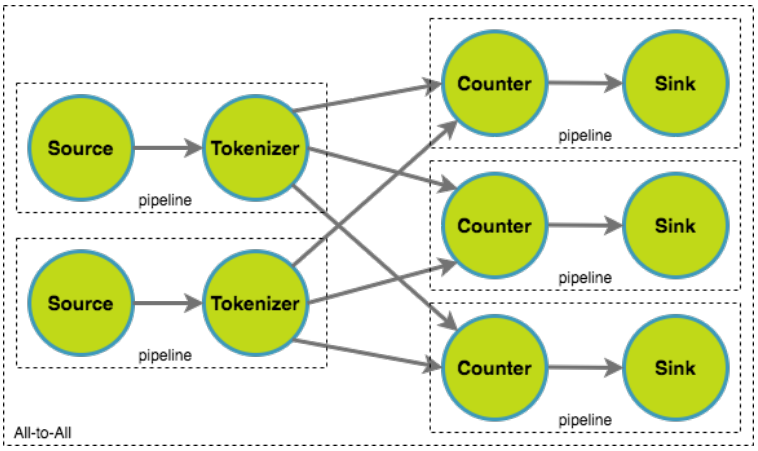

Figure 5.7: *all-to-all* based implementation of the *Word Count* test case where we can have multiple replicas of the ¡Source, Tokenizer¿ and ¡Counter, Sink¿ pipelines.

```
1  result_t* svc(tuple_t* in) {
2    char *tmpstr;
3    char *token = strtok_r(in->text_line, "␣", &tmpstr);
4    while (token) {
5      int ch = std::hash<std::string>()(std::string(token)) % noutch;
6
7      result_t* r = new result_t;
8      strncpy(r->key, token, MAXWORD-1);
9      r->key[MAXWORD-1]='\0';
10
11     ff_send_out_to(r, ch);
12     token = strtok_r(NULL, "␣", &tmpstr);
13   }
14   delete in;
15   return GO_ON;
16 }
```

Listing 5.1: Implementation of the tokenizer node, where words are hashed (line 6) in order to know which is the corresponding counter node. `noutch` represents the number of counter replicas.

Starting from the shared-memory version of the Word Count (i.e., the one in Figure 5.7) we now want to modify it to run into a distributed-memory platform. We want to deploy the workers of the first set of the *all-to-all* (i.e., the ¡Source, Tokenizer¿ replicas) into a separate process and the workers of the second set (i.e., the ¡Counter, Sink¿ replicas) into a second process. As already highlighted, this

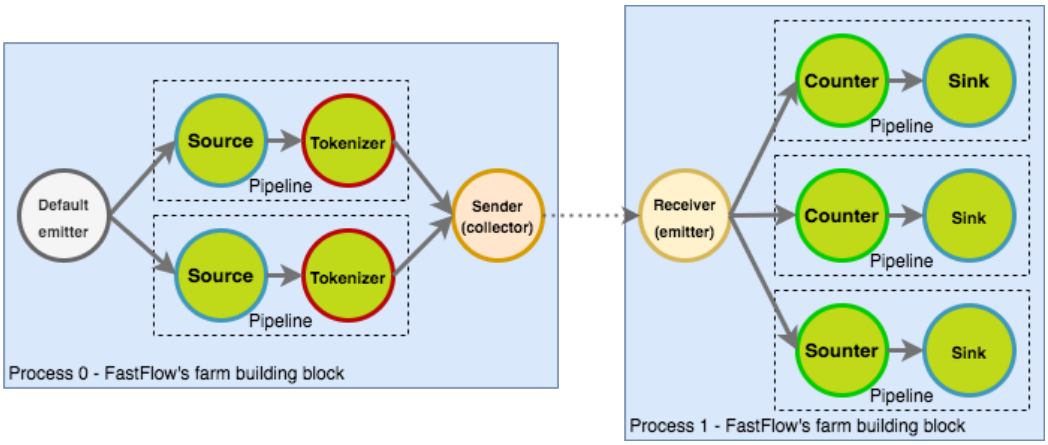

Figure 5.8: Word Count process network.

is a good example of an application that requires a relatively complex routing of messages exchanged between processes. A wrong routing would bring to an incorrect result. Therefore, this application serves as proof of the correctness of the implemented routing mechanisms, including the routing protocol.

The resulting network of processes and their internals is shown in Figure 5.8. As a reference, in Listing 5.2 and in Listing 5.3 are reported the code of the shared-memory and distributed-memory version, respectively.

To prove the correctness of the distributed version, particularly the routing of messages, we will compare the outputs of the distributed-memory and the shared-memory versions. Both versions use two counters/sink replicas. The text employed for this test is an extract of *La divina commedia* by Dante Alighieri, in particular, lines 4-5-6 of the first canticle. The results, reported in Figure 5.9 and Figure 5.10 refer the shared-memory and the distributed memory versions, respectively. As it can be seen, the result is the same.

```
#include <ff/ff.hpp>
...
int main(int argc, char *argv[]){
 vector<Counter*> C(sink_deg);
 vector<Sink*>    S(sink_deg);
 vector<ff_node*> L;
 vector<ff_node*> R;

 ff_a2a a2a();

 for(int i=0; i<source_deg; ++i){
   auto pipe = new ff_pipeline();
   pipe->add_stage(new Source());
   Splitter* sp = new Splitter(
    sink_deg);
   pipe->add_stage(sp);
   L.push_back(pipe);
  }

  for(int i=0; i<sink_deg; ++i){
   auto pipe = new ff_pipeline();
   S[i] = new Sink;
   C[i] = new Counter;
   pipe->add_stage(C[i]);
   pipe->add_stage(S[i]);
   R.push_back(pipe);

  }

 a2a.add_firstset(L);
 a2a.add_secondset(R);
 a2a.run_and_wait_end();
}
```

Listing 5.2: Shared-memory implementation of Word Count.

```
#include <ff/dff.hpp>
...
int main(int argc, char *argv[]){
 DFF_Init(argc, argv)
 vector<Counter*> C(sink_deg);
 vector<Sink*>    S(sink_deg);
 vector<ff_node*> L;
 vector<ff_node*> R;

 ff_a2a a2a();
 auto G1 = a2a.createGroup("G1");
 auto G2 = a2a.createGroup("G2");

 for(int i=0; i<source_deg; ++i){
   auto pipe = new ff_pipeline();
   pipe->add_stage(new Source());
   Splitter* sp = new Splitter(
    sink_deg);
   pipe->add_stage(sp);
   L.push_back(pipe);

   G1.out << sp;
  }
 for(int i=0; i<sink_deg; ++i){
   auto pipe = new ff_pipeline();
   S[i] = new Sink;
   C[i] = new Counter;
   pipe->add_stage(C[i]);
   pipe->add_stage(S[i]);
   R.push_back(pipe);

   G2.in << C[i];
 }

 a2a.add_firstset(L);
 a2a.add_secondset(R);
 ff_pipeline pipeMain();
 pipeMain.add_stage(&a2a);
 pipeMain.run_and_wait_end();
}
```

Listing 5.3: Distributed-memory version of the Word Count.

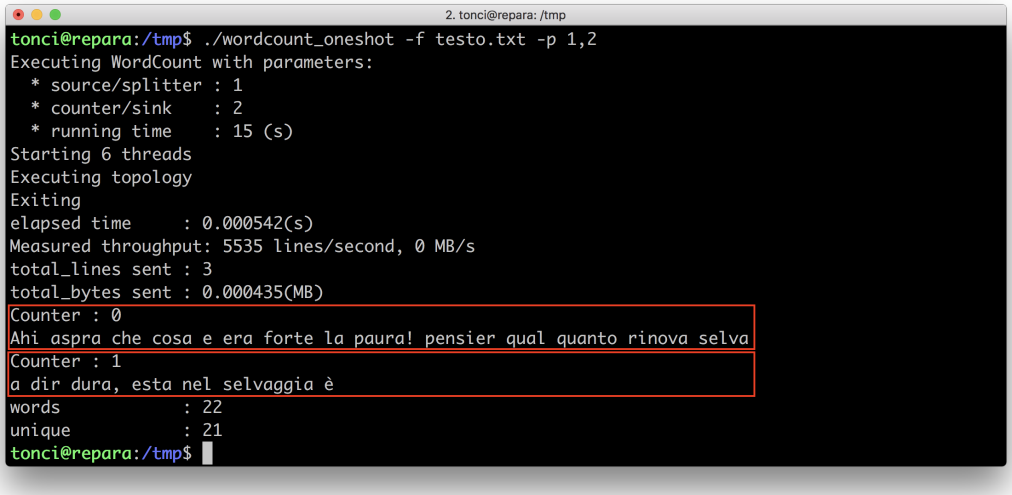

Figure 5.9: Shared-memory version of Word Count analyzing line 4-5-6 of the first canticle of *La divina commedia*, employing two counters.

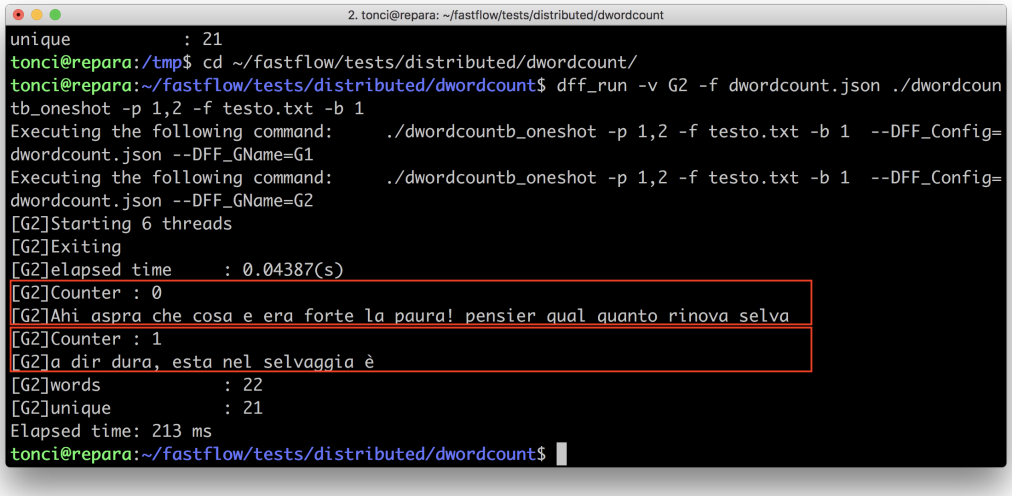

Figure 5.10: Distributed-memory version of Word Count analyzing the same input data-set of the shared-memory version.

5.2.1 Performance tuning

Word Count represents a very fine-grained application, particularly for the distributed memory domain. This is due to the fact that each word is sent over the network to another process implementing the counter. Words use just a few bytes. Hence, the cost of setting up a message, encapsulate it, and decapsulate it at the remote side, plays an important role in the total communication overhead. In particular, the total communication overhead is more significant than the time needed to process the single word, making the granularity of this application very fine. Considering these aspects, we expect that a shared-memory version of the Word Count out beats the distributed version unless the input file is so large to not fit into the RAM of a single server.

In order to amortize the costs of sending words related to setup overheads, we can buffer a parametric amount of lines before sending the words to the respective counters. As a consequence, the network throughput is also optimized. This technique is usually labeled as "batching". Several tests have been carried out to evaluate the scalability of the distributed version by varying the buffer size (B) and the number of replicas of each sub-modules. In the first test-set, both processes have been deployed in the same machine, i.e., *Repara*, and the results are shown in Figure 5.11. The second test-set, instead, has been carried out spreading the processes in two distinct machines, i.e., *Repara* for the source/splitting and *Titanic* for counting/sinking. The results of the latter test-set are shown in Figure 5.12.

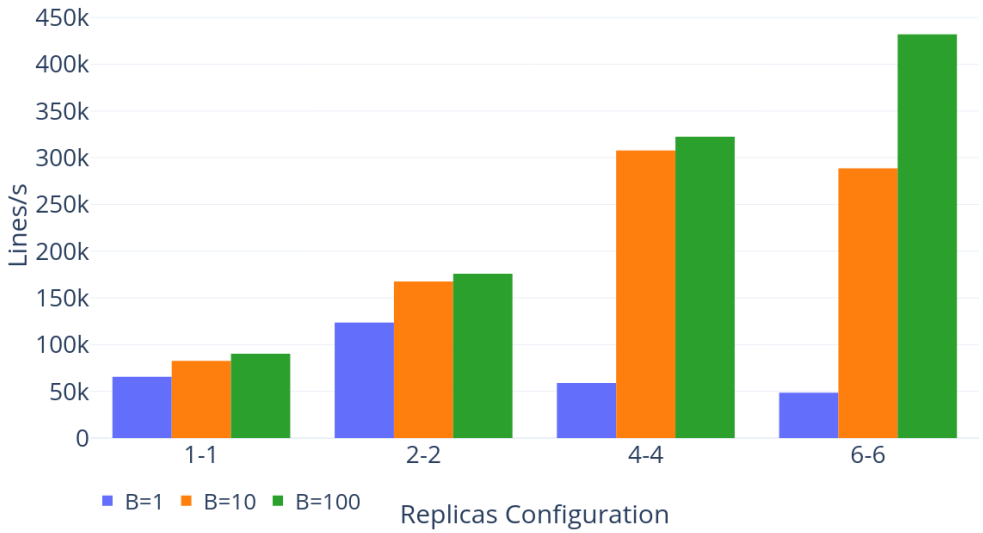

Figure 5.11: Performances of Word Count varying the buffering parameter. All processes on the same machine.

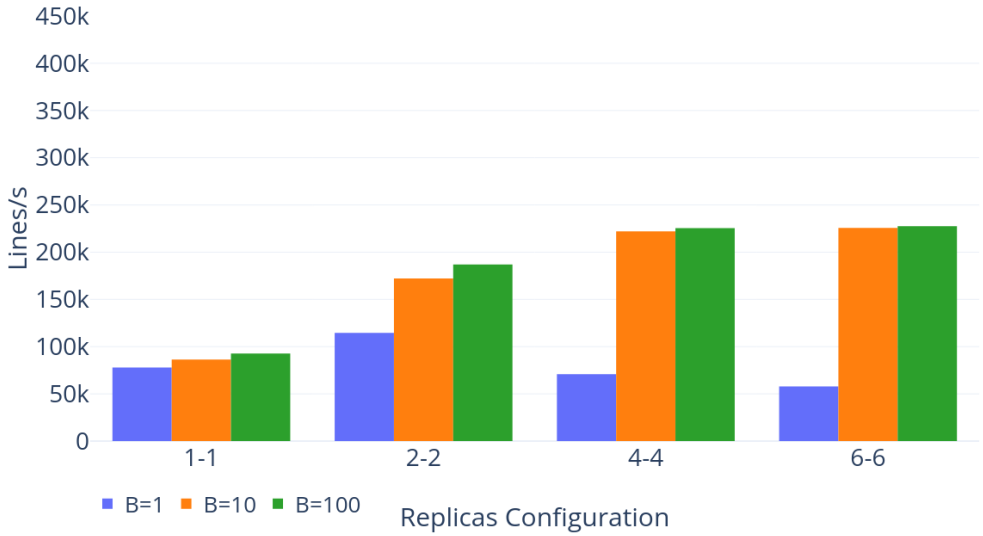

Figure 5.12: Performances of Word Count varying the buffering parameter. Processes on different machines interconnected at 1Gbit/s.

Figure 5.13: On the left-hand side is shown a frame of the input video. On the right-hand side the output frame after applying Lane Detection algorithm; the lane is highlight and the predicted movement is also overlaid in the frame.

5.3 Lane Detection

To evaluate the distributed run-time system in a coarse-grained application, we consider the lane detection algorithm[43] [44] in the field of computer vision and AI. Lane detection is a crucial module for autonomous cars and requires low latency (almost real-time in some cases) to be employed safely. In our case, we process a video stored in the file system and we write back a video where the lane is highlighted. The logical parallel structure of the application is that of a three-stage pipeline (read-compute-write). The application has been implemented using a pipeline of two stages: the first read the input video from the local disk sequentially, the second is an *all-to-all*.

Regarding the latter, the left-hand side set is composed of the actual workers computing a frame, and the right-hand side set is composed of just one sequential node that writes the resulting video back to the local disk. Since the video frame order has to be preserved, the last sequential node needs to reorder the frames if received out-of-order. The actual implementation mimics the farm pattern, as can be seen conceptually in Figure 5.14. The core algorithm makes extensive use of *OpenCV* library[45], which is the de-facto standard for real-time and deferred computer vision. *OpenCV* internally, unless recompiled with special flags not so well documented, adopt widely threading to parallelize the execution of the provided functionalities. In order to limit the use of all the available processors of the machines, *Repara* and *Titanic*, composing our test bench, we assume that each process runs inside a container with only four core available. Containerization[46, 47] allows to deploy and run distributed applications isolated without requiring the set-up of a VM for each application. Multiple isolated applications (or systems) run on a single host and use a single kernel, making containers more efficient than virtual machines, preserving the same feeling from the point of view of processes. Nowadays, is quite common to execute application inside cloud containers with few private resources, such as cores, memory, etc. This benchmark can be a good

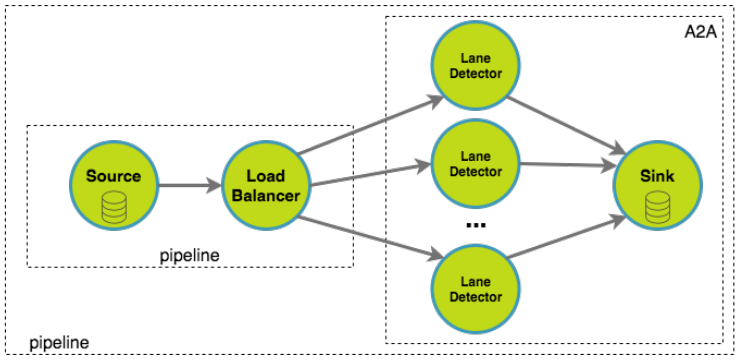

Figure 5.14: Lane Detection example data-flow graph. Source node read the video from the file-system. Sink node write back the transformed video to the file-system.

opportunity to investigate how distributed applications deployed in containers respond.

The distributed version can programmatically create as many groups as the number of workers, naming each group W concatenating a progressive number (W0, W1, ...). The source and the sink are always deployed in different distinct processes. The resulting shared-memory and distributed-memory codes are presented in Listing 5.4 and Listing 5.5, respectively.

```cpp
#include <ff/ff.hpp>
...
int main(int argc, char *argv[]){

 Source src(argv[1]);
 Sink sink(argv[2]);

 ff_a2a A2A;
 vector<LaneDetector *> w;

 for(int i=0; i<numWorkers; i++)
   w[i] = new LaneDetector();

 A2A.add_firstset(w);
 A2A.add_secondset({&sink});


 ff_pipeline pipe;
 pipe.add_stage(&src);
 pipe.add_stage(&A2A);

 pipe.run_and_wait_end();
 return 0;
}
```

Listing 5.4: Shared-memory implementation of the Lane Detector.

```cpp
#include <ff/dff.hpp>
...
int main(int argc, char *argv[]){
 DFF_Init(argc, argv);

 Source src(argv[1]);
 Sink sink(argv[2]);

 ff_pipeline srcPipe;
 srcPipe.add_stage(&src);
 srcPipe.createGroup("Source").out
     << &src

 ff_a2a A2A;
 vector<LaneDetector *> w;

 for(int i=0; i<numWorkers; i++){
  w[i] = new LaneDetector();
  auto g = A2A.createGroup("W"+i);
  g.in << w[i]; g.out << w[i];
 }

 A2A.add_firstset(w);
 A2A.add_secondset({&sink});
 A2A.createGroup("Sink").in << &
    sink;

 ff_pipeline pipe;
 pipe.add_stage(&srcPipe);
 pipe.add_stage(&A2A);

 pipe.run_and_wait_end();
 return 0;
}
```

Listing 5.5: Distributed-memory version of the Lane Detector.

We will compare the completion time of different configurations of the distributed version with the sequential one. Also, the sequential version has been executed as if contained in a container with only four core available. Since each process is executed in one container, from now on, process and container refer to the same thing.

The distributed configurations are: (R) 1 worker on *Repara*; (R/T) 2 work-

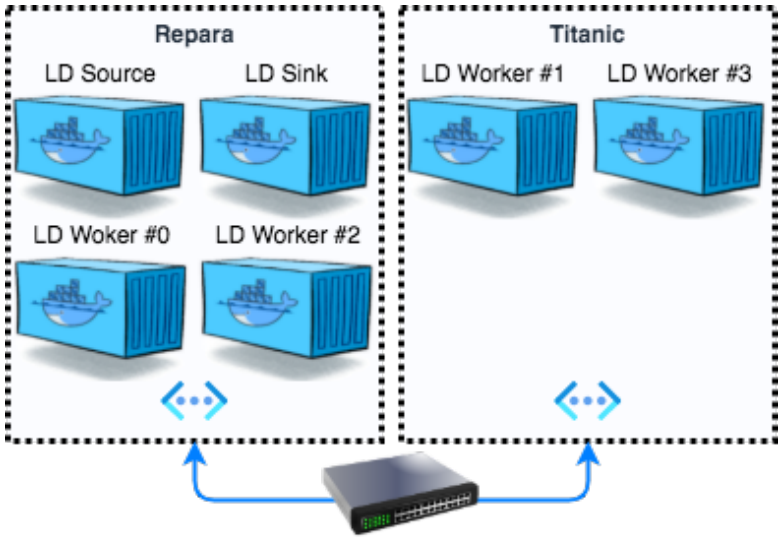

Figure 5.15: Lane detection containerized version. Each container provides four cores.

ers, one on *Repara* and one on *Titanic*; (R/R/T) 3 workers, two on *Repara* and one on *Titanic*; (R/R/T/T) 4 workers, two on *Repara* and two on *Titanic*. We change the number of workers of the algorithm and the mapping of containers/-machines. In this test, the source and the sink containers are always deployed in the same machine *Repara*. The R/R/T/T configuration is reported in Figure 5.15. The measured completion time of each configuration is plotted in in Figure 5.16. The results show that just splitting the read-compute-write pipeline in three distinct processes deployed in containers on the same machine (i.e., configuration R) gives benefits to the sequential version. In the R configuration, we are using three times the resources of the sequential one since we are employing three containers. The second distributed configuration (R/T), using two workers, does not scale as expected. This is because the worker deployed on *Titanic* is capped by the network representing the main bottleneck, and the scheduling policy is static. The proof that the network represents a bottleneck is given by a measurement not reported in the plot in which we deploy a single worker (like configuration R) but on *Titanic*. The completion time of this execution is approximately 60s, which is approximately +60% with respect to the reported result of the R configuration. Employing dynamic scheduling would mitigate the problem of two unbalanced workers. Alternatively, increasing the network bandwidth from 1Gbit/s to 10Gbit/s or 40Gbit/s would make the workers more balanced. Confirming the hypothesis of unbalanced workers is the third configuration R/R/T, in which we try to re-balance the workload between machines giving 2/3 of the

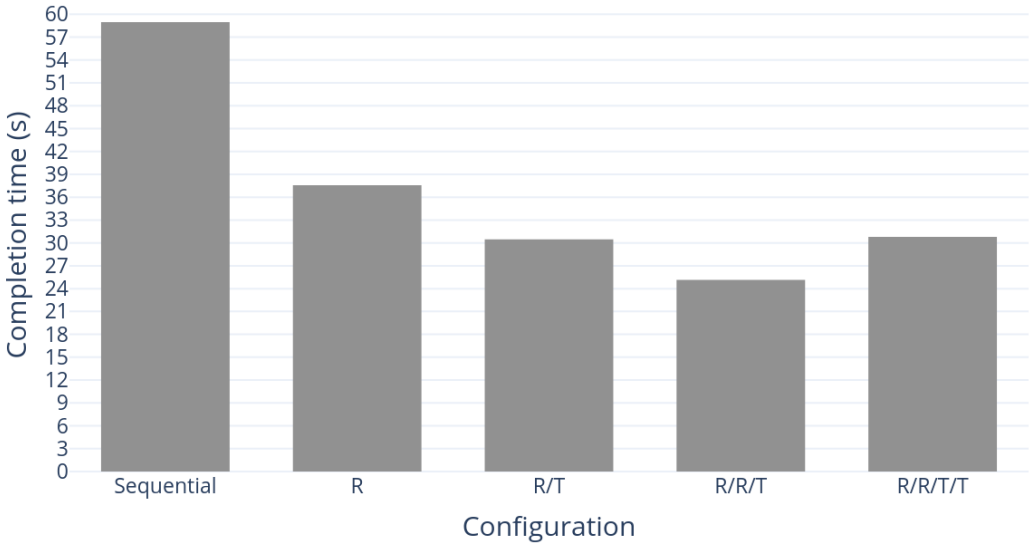

Figure 5.16: Lane Detection completion time in different configurations. All are distributed but the first one which is sequential. Configurations are labelled with the mapping of the workers: R (single worker deployed in *Repara*), R/T (two worker, one in *Repara* one in *Titanic*), and so forth.

tasks to workers in the same machine of the source/sink *Repara* and the rest 1/3 to the worker deployed on *Titanic*. This configuration results in the best one in terms of completion time. The last configuration (R/R/T/T) utilizing four workers, results again in an unbalanced configuration since we are sending only 50% of the task to workers in the same machine of the source/sink. Even if the number of workers is bigger than the R/R/T configuration, the performances are worse.

In this benchmark, to emulate the containers, processes have been launched with `taskset` with a given CPU affinity. The CPU affinity is a scheduler property that pins a process to a given set of CPUs on the system, obtaining the same effects as using Linux's *cgroups* to implement containers.

Conclusion

In this thesis, we have extended the *FastFlow* parallel programming library with a new distributed-memory run-time support. The introduced support enables the execution of generic *FastFlow* applications on distributed platforms to exploit the resources of multiple interconnected multi-core machines. The whole work described in this thesis has been carried out by keeping in mind the initial objectives. These consisted in making the transition of a fully working *FastFlow* shared-memory application to a distributed-memory one painless to the programmer.

Two main directives guided our work: the new API needed to introduce distributed groups in the *FastFlow* library must be easy to use; the distributed run-time system must be modular, efficient, and easy to extend.

The same directives guided the development of the *loader* software module (called *dff_run*), which aims to facilitate the launching of distributed *FastFlow* applications. Although such a module is still in its early stages, it was very useful to automatize the execution of the tests we conducted and accelerate troubleshooting during the development.

As demonstrated by the set of experiments we conducted on a small cluster of Linux machines connected by a switched Gigabit network, the proposed distributed run-time system is capable to achieve good performance figures in terms of sustained throughput and is a useful tool to increase the speedup of applications limited by the single machine resources.

We considered a test set made up of three different use-cases: a synthetic microbenchmark, a video filtering application, and the streaming WordCount application. The synthetic microbenchmark allowed us to perform multiple tests: (i) the *Point-to-Point* test to measure the achievable throughput by using the two proposed approach for the data serialization, and (ii) the emulation of a *distributed farm* pattern to evaluate how different computation grain in the distributed workers impacts the overall pattern performance. Instead, the *Word Count* application allowed us to prove the correctness of the message routing protocol and to analyze the impact of message packing as a potential optimization worth introducing directly at the distributed RTS level. Finally, the *Lane Detection* application served

as an example of an application that requires high bandwidth in the interconnection link to scale in the distributed-memory domain. Results obtained running the Lane Detection application on a single machine demonstrate that the RTS we developed does not introduce bottlenecks and that the performance of this application strictly depends on the available network bandwidth.

Most of the time of the thesis was spent on the software engineer aspects of the proposed run-time system, e.g., designing the mechanisms to make the introduced layer fully functional and designing the APIs provided to the end-user. The whole produced codebase is fully working, and it is publicly released as open-source. It is actually part of the main *FastFlow GitHub* repository.

Future works

Several improvements and several new features can be introduced starting from the version we developed. In our opinion, the most important and higher priority ones are the following:

- Relaxing group identification constraints: relaxing the constraints in the group identification allows a more flexible split of the original *FastFlow* data-flow graph and a faster transformation of shared-memory applications.
- Configuration file automation: automatize the creation of the configuration file by statically discovering how groups are going to interact. This will relieve the programmer of writing complex configuration files for big applications composed by several groups.
- Improving the loader: the loader we developed is a useful tool but still too basic. Multiple are the improvements that can be added to this software component. Most of them were presented in Section 4.2.
- Feedback channels: The current run-time supports only directed acyclic data-flow graphs, so it is not possible to express in any way loops and feedback channels. Feedback channels are graph edges directed in the opposite direction with respect to the standard data flow. Introducing the support of feedback channels in the *FastFlow* graph will allow us to extend the domain of algorithms that can be expressed, such as iterative ones.
- MPI communication back-end: Implementing communication nodes using MPI instead of raw TCP/IP, enables the exploitation of native collective communication channels and the support for RDMA[48]. RDMA will help to reduce the overheads allowing real zero-copy in all the distributed-memory communications.

References

[1] Dalvan Griebler, Marco Danelutto, Massimo Torquati, and Luiz Gustavo Fernandes. Spar: a dsl for high-level and productive stream parallelism. *Parallel Processing Letters*, 27(01):1740005, 2017.

[2] Anju Bala and Inderveer Chana. Fault tolerance-challenges, techniques and implementation in cloud computing. *International Journal of Computer Science Issues (IJCSI)*, 9(1):288, 2012.

[3] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel computing*, 30(3):389–406, 2004.

[4] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[5] Johan Enmyren and Christoph W Kessler. Skepu: a multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, pages 5–14, 2010.

[6] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core. *Programming multi-core and many-core computing systems, parallel and distributed computing*, 2017.

[7] Marco Danelutto and Massimo Torquati. Structured parallel programming with "core" FastFlow. In Viktória Zsók, Zoltán Horváth, and Lehel Csató, editors, *Central European Functional Programming School: 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers*, pages 29–75. Springer International Publishing, Cham, 2015.

[8] Marco Aldinucci, Massimiliano Meneghin, and Massimo Torquati. Efficient Smith-Waterman on multi-core with FastFlow. In Marco Danelutto, Tom Gross, and Julien Bourgeois, editors, *Proc. of Intl. Euromicro PDP 2010:*

Parallel Distributed and network-based Processing, pages 195–199. IEEE, February 2010.

[9] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. Accelerating code on multi-cores with FastFlow. In E. Jeannot, R. Namyst, and J. Roman, editors, *Proc. of 17th Intl. Euro-Par 2011 Parallel Processing*, volume 6853 of *LNCS*, pages 170–181, Bordeaux, France, August 2011. Springer.

[10] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[11] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[12] Walter L Hürsch and Cristina Videira Lopes. Separation of concerns. 1995.

[13] David del Rio Astorga, Manuel F Dolz, Javier Fernández, and J Daniel García. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, 29(24):e4175, 2017.

[14] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[15] Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. The Münster Skeleton Library Muesli: A comprehensive overview. ERCIS Working Papers 7, University of Münster, European Research Center for Information Systems (ERCIS), 2009.

[16] Kevin Bourgeois, Sophie Robert, Sébastien Limet, and Victor Essayan. Geoskelsl: A python high-level dsl for parallel computing in geosciences. In *International Conference on Computational Science*, pages 839–845. Springer, 2018.

[17] Samuel A Fineberg and Thomas S Woodrow. Mpirun: A portable loader for multidisciplinary and multi-zonal applications. 1994.

[18] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[19] Xindong Wu, Xingquan Zhu, Gong-Qing Wu, and Wei Ding. Data mining with big data. *IEEE transactions on knowledge and data engineering*, 26(1):97–107, 2013.

[20] Chris Anderson. The end of theory: The data deluge makes the scientific method obsolete. *Wired magazine*, 16(7):16–07, 2008.

[21] Apache Software Foundation. Apache storm. `https://storm.apache.org`.

[22] Apache Software Foundation. Apache spark. `https://spark.apache.org`.

[23] Apache Software Foundation. Apache kafka. `https://kafka.apache.org`.

[24] Lightbend. Akka. `https://akka.io`.

[25] Apache Software Foundation. Apache hadoop. `https://hadoop.apache.org`.

[26] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[27] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[28] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 97–104. Springer, 2004.

[29] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *2009 17th Euromicro international conference on parallel, distributed and network-based processing*, pages 427–436. IEEE, 2009.

[30] Lorna Smith and Mark Bull. Development of mixed mode mpi/openmp applications. *Scientific Programming*, 9(2, 3):83–98, 2001.

[31] Javier López-Gómez, Javier Fernández Muñoz, David del Rio Astorga, Manuel F Dolz, and J Daniel Garcia. Exploring stream parallel patterns in distributed mpi environments. *Parallel Computing*, 84:24–36, 2019.

[32] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Targeting distributed systems in fastflow. In *European Conference on Parallel Processing*, pages 47–56. Springer, 2012.

[33] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. An efficient unbounded lock-free queue for multi-core systems. In Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, pages 662–673, Berlin, Heidelberg, 2012. Springer.

[34] Massimo Torquati. *Harnessing Parallelism in Multi/Many-Cores with Streams and Parallel Patterns*. PhD thesis, University of Pisa, 2019.

[35] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati. Design patterns percolating to parallel programming framework implementation. *Int. J. Parallel Program.*, 42(6):1012–1031, December 2014.

[36] Gabriele Mencagli, Massimo Torquati, Andrea Cardaci, Alessandra Fais, Luca Rinaldi, and Marco Danelutto. Windflow: High-speed continuous stream processing with parallel building blocks. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[37] Leonardo Gazzarri and Marco Danelutto. A tool to support FastFlow program design. In *Proceedings of the International Conference on Parallel Computing (ParCo 2017)*, Advances in Parallel Computing, pages 687–697, Bologna, Italy, 2018. IOS Press.

[38] Marco Danelutto, Massimo Torquati, and Peter Kilpatrick. A DSL Based Toolchain for Design Space Exploration in Structured Parallel Programming. *Procedia Computer Science*, 80:1519 – 1530, 2016. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[39] Martin Sústrik et al. Zeromq. *Introduction Amy Brown and Greg Wilson*, 2015.

[40] W Shane Grant and Randolph Voorhies. cereal–a c++ 11 library for serialization. *URL https://github. com/USCiLab/cereal*, 2013.

[41] Lindsay Bassett. *Introduction to JavaScript object notation: a to-the-point guide to JSON*. " O'Reilly Media, Inc.", 2015.

[42] Jan Kanclirz. *Netcat power tools*. Elsevier, 2008.

[43] Abdulhakam AM Assidiq, Othman O Khalifa, Md Rafiqul Islam, and Sheroz Khan. Real time lane detection for autonomous vehicles. In *2008 International Conference on Computer and Communication Engineering*, pages 82–88. IEEE, 2008.

[44] Miguel Maestre Trueba. Lane detection for autonomous cars. https://github.com/MichiMaestre/Lane-Detection-for-Autonomous-Cars, 2017.

[45] Ivan Culjak, David Abram, Tomislav Pribanic, Hrvoje Dzapo, and Mario Cifrek. A brief introduction to opencv. In *2012 proceedings of the 35th international convention MIPRO*, pages 1725–1730. IEEE, 2012.

[46] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support paas. In *2014 IEEE International Conference on Cloud Engineering*, pages 610–614. IEEE, 2014.

[47] Claus Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

[48] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K Panda. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

Appendix A

Code Structure

The whole produced code can be found in the official *FastFlow GitHub* repository at the following link: `https://github.com/fastflow/fastflow`.

The main file `dff.hpp` used to include and to enable the distributed-memory support has been placed in the main directory of *FastFlow* (i.e., the one to be included during the compilation). The main implementation files of the new RTS are located in the `ff/distributed` directory. An overview of the implementation files is given in Table A.1. On the same directory is also located the `loader` sub-directory which contains the source code of the `dff_run` tool. Tests targeting the distributed-memory support, instead, are located in the `tests/distributed` directory.

Some other files of the *FastFlow* library have been edited to accommodate the new run-time system features.

File	Description
ff_dgroups.hpp	Singleton to index all the groups. Include DFF_Init and the other functions to enable the distributed-memory support.
ff_dgroup.hpp	Class representing the group. Contains the node annotation API and the internals mechanism to build the group.
ff_network.hpp	Contains all the helper structure and functions to communicate over the network.
ff_dreceiver.hpp	Class implementing the receiver communication node.
ff_dsender.hpp	Class implementing the sender communication node.
ff_wrappers.hpp	File including all types of node wrapper: WrapperIN, WrapperOUT, WrapperINOUT.

Table A.1: Files containing the implementation of the distributed-memory support.