International Journal of Parallel Programming (2023) 51:1–21
https://doi.org/10.1007/s10766-022-00750-5

Distributed-Memory FastFlow Building Blocks

Nicolò Tonci[1] · Massimo Torquati[1] · Gabriele Mencagli[1] · Marco Danelutto[1]

Received: 30 August 2022 / Accepted: 21 November 2022 / Published online: 2 December 2022

Abstract
We present the new distributed-memory run-time system (RTS) of the C++-based open-source structured parallel programming library *FastFlow*. The new RTS enables the execution of *FastFlow* shared-memory applications written using its *Building Blocks* (`BBs`) on distributed systems with minimal changes to the original program. The changes required are all high-level and deal with introducing *distributed groups* (*dgroup*), i.e., logical partitions of the BBs composing the application streaming graph. A *dgroup*, which in turn is implemented using *FastFlow*'s `BBs`, can be deployed and executed on a remote machine and communicate with other *dgroups* according to the original shared-memory *FastFlow* streaming programming model. We present how to define the distributed groups and how we faced the problem of data serialization and communication performance tuning through transparent messages' batching and their scheduling. Finally, we present a study of the overhead introduced by *dgroups* considering some benchmarks on a sixteen-node cluster.

Keywords High-level parallel programming · Distributed programming · Parallel patterns · Algorithmic skeletons · Building blocks

✉ Massimo Torquati
massimo.torquati@unipi.it

Nicolò Tonci
nicolo.tonci@phd.unipi.it

Gabriele Mencagli
gabriele.mencagli@unipi.it

Marco Danelutto
marco.danelutto@unipi.it

[1] Computer Science Department, University of Pisa, Pisa, Italy

1 Introduction

High-end computing servers show a clear trend toward using multiple hardware accelerators to provide application programmers with thousands of computing cores. However, many challenging applications demand more resources than those offered by a single yet powerful computing node. In these cases, application developers have to deal with different nested levels and kinds of parallelism to squeeze the full potential of the platform at hand.

In this scenario, the C++-based *FastFlow* parallel programming library [1], initially targeting multi/many-core architectures, aspires to define a *single programming model* for shared- and distributed-memory systems leveraging a streaming data-flow programming approach and a reduced set of structured parallel components called `Building Blocks` (BBs). *FastFlow*'s `BBs` provide the programmer with efficient and reusable implementations of essential parallel components that can be assembled following a LEGO-style model to build and orchestrate more complex parallel structures (including well-known algorithmic skeletons and parallel patterns) [2]. With `BBs`, the structured parallel programming methodology percolates to a lower-lever of abstraction [3].

In this paper, we present the new distributed-memory run-time system (RTS) introduced in the `BBs` software layer of the *FastFlow* library aiming to target both scale-up and scale-out platforms preserving the programming model. It enables the execution of *FastFlow* applications written using `BBs` on distributed systems. Already written applications require minimal modifications to the original shared-memory program. New applications can be first developed and debugged on a single node, then straightforwardly ported to multiple nodes. The resulting distributed applications can still be recompiled to run on a single node with the native shared-memory RTS without modifying the code and with no degradation of performance. The motivations that have led us to work at the `BB` software level of the *FastFlow* library are twofold: a) provide the programmer with a quick and easy porting methodology of already written *FastFlow* data-streaming applications to distributed systems by hiding all low-level pitfalls related to distributed communications; b) prepare a set of mechanisms (e.g., specialized RTS `BBs`, serialization features, message batching) that can be used as the basis for building high-level ready-to-use parallel and distributed exploitation patterns (e.g., Map-Reduce, D &C).

We present the idea of *FastFlow*'s *distributed groups* and its associated API as well as some experimental results that validate functional correctness and provide preliminary performance assessments of our work.

The outline of the paper is as follows. Section 2 presents an overview of the *FastFlow* library and its `BBs`. Section 3 introduces the *distributed group* concept and semantics. Section 4 presents the experimental evaluation conducted. Section 5 provides a discussion of related works and Sect. 6 draws the conclusions and possible future directions.

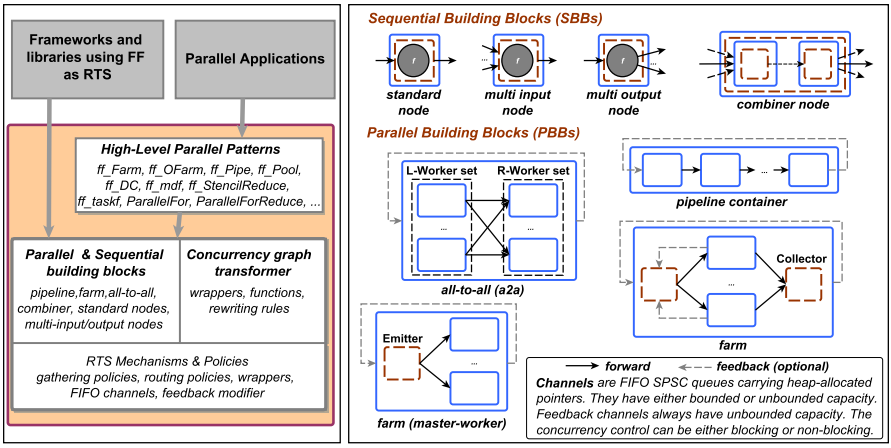


Fig. 1 (`Left`) *FastFlow* software stack. (`Right`) Graphical notation of the *FastFlow*'s Building Blocks

2 *FastFlow* Overview and Background

The *FastFlow* library is the result of a research effort started in 2010 with the aim of providing application designers with key features for parallel programming via suitable parallel programming abstractions (e.g., ordered farm, pipeline, divide &conquer, parallel-for, macro data-flow, map+reduce, etc.) and a carefully designed RTS [1]. The structured parallel programming methodology [4] was the fertile ground that has allowed the development of the initial idea and then guided the *FastFlow* library implementation.

The latest *FastFlow* version (v. 3.x) has been released in 2019 where the lower-level software layers have been redesigned, and the concept of *Building-Block* introduced to support the development of new patterns and domain-specific libraries [2]. In addition to the *farm* and *pipeline* core components two new `BBs` have been added, namely *all-to-all* and *node combiner*. Furthermore, a new software layer called *Concurrency graph transformer* is now part of the *FastFlow* software stack. Such a layer is in charge of providing functions for the concurrency graph refactoring to introduce optimizations (e.g., fusing parallel `BBs`) and enhancing the performance portability of the applications. *FastFlow*'s software layers are sketched in the left-hand side of Fig. 1.

Building Blocks (BBs). *Building Blocks* [2, 3] are recurrent data-flow compositions of concurrent activities working in a streaming fashion, which are used as the basic abstraction layer for building *FastFlow* parallel patterns and, more generally *FastFlow* streaming topologies. The Data-flow streaming model and the `BBs` are the two fundamental ingredients of the *FastFlow* library. Following the same principles of the structured parallel programming methodology, a parallel application (or one of its components) is conceived by selecting and adequately assembling a small set of well-defined `BBs` modeling both data and control flows. Differently from "pure" algorithmic skeleton-based approaches, where highly specialized, reusable, and efficient monolithic implementations of each skeleton are defined for a

given architecture, the BB-based approach provides the programmer with efficient and reusable implementations of lower-level basic parallel components that can be assembled following a LEGO-style methodology to build and orchestrate more complex parallel structures [2]. They can be combined and nested in different ways forming either acyclic or cyclic concurrency graphs, where graph nodes are *FastFlow* concurrent entities and edges are communication channels. A communication channel is implemented as a lock-free Single-Producer Single-Consumer (SPSC) FIFO queue carrying pointers to heap-allocated data [5]. Collective communications involving multiple producers and/or consumers are realized through broker nodes employing multiple SPSC queues. More specifically, we consider *Sequential Building Blocks* (SBBs) and *Parallel Building Blocks* (PBBs). SBBs are: the *sequenatial node* (in three versions) and the *nodes combiner*. PBBs are: the *pipeline*, the *farm* (in two versions), and the *all-to-all*. The right-hand side of Fig. 1 shows the graphical notation of all BBs. A description of each BB follows.

node. It defines the unit of sequential execution in the *FastFlow* library. A node encapsulates either user's code (i.e. business logic) or RTS code. Based on the number of input/output channels it is possible to distinguish three different kinds of sequential nodes: *standard node* with one input and one output channel, *multi-input node* with many inputs and one output channel, and *multi-output node* with one input and many outputs. A node performs a loop that: (i) gets a data item (through a memory reference) from one of its input queues; (ii) executes a functional code working on the input data item and possibly on a state maintained by the node itself by calling its service method (`svc()`); (iii) puts a memory reference to the resulting item(s) into one or multiple output queues selected according to a predefined (i.e., on-demand, round-robin) or user-defined policy (e.g., by-key, random, broadcast, etc.).

nodecombiner. It enables to combine two SBBs into one single sequential node. Conceptually, the combining operation is similar to the composition of two functions. In this case, the functions are the service functions of the two nodes (i.e., the *svc()* methods). This SBB promotes code reuse through fusion of already implemented nodes and it is also used to automatically reduce the number of threads implementing the concurrent graph when possible.

pipeline. The pipeline is the topology builder. It connects BBs in a linear chain (or in a toroidal way if the last stage is connected to the first one with a *feedback channel*). Also, it is used as a *container* of BBs for grouping them in a single parallel component. At execution time, the *pipeline* models the data-flow execution of its BBs on data elements flowing in streaming.

farm. It models functional replication of BBs coordinated by a sequential master BB called *Emitter*. The default skeleton is composed of two computing entities executed in parallel (this version is called *master–worker*): a multi-output Emitter, and a pool of BBs called *Workers*. The *Emitter* node schedules data elements received in input to the *Workers* using either a default policy (i.e., round-robin or on-demand) or according to the algorithm implemented in the business code defined in its service method. Optional *feedback channels* connect *Workers* back at the *Emitter*. A second version of the *farm*, comprises also a multi-input

BB called *Collector* in charge of gathering results coming from *Workers* (the results can be gathered either respecting *farm* input ordering or without any specific ordering). Also in this version, optional *feedback channels* may connect both *Workers* as well as *Collector* back to the *Emitter*.

all − to − all. The all-to-all (briefly *a2a*) defines two distinct sets of *Workers* connected according to the *shuffle communication pattern*. Therefore, each Worker in the first set (called L-Worker set) is connected to all Workers in the second set (called R-Worker set). The user may implement any custom distribution policy in the L-Worker set (e.g., sending each data item to a specific Worker of the R-Worker set, broadcasting data elements, executing a by-key routing, etc). The default distribution policy is round-robin. Optional *feedback channels* may connect R-Worker with L-Worker sets, thus implementing an *all-to-all communication pattern*.

`BBs` can be composed and nested, like LEGO bricks, to build concurrent streaming networks of nodes executed according to the data-flow model. The rules for connecting `BBs` and generating valid topologies are as follows:

1. Two `SBBs` can be connected into a pipeline container regardless of their number of input/output channels.
2. A `PBB` can be connected to `SBBs` (and vice versa) into a pipeline container by using *multi-input* (*multi-output*) sequential nodes;
3. Two `PBBs` can be connected into a pipeline container either if they have the same number of nodes, or through multi-input multi-output sequential nodes if they have different number of nodes at the edges.

To help the developer when possible, the RTS automatically enforces the above rules transforming the edge nodes of two connecting `BBs` by using proper node wrappers or adding helper nodes via the *combiner* `BB`. For example, in the *farm* `BB`, the sequential node implementing the *Emitter* is automatically transformed in a multi-output node. Additionally, if an *all-to-all* `BB` is connected to a *farm*, then the *Emitter* is automatically transformed in a *combiner node* where the left-hand side node and the right-hand side node are multi-input and multi-output, respectively. The *Concurrency graph transformer* software layer (see Fig. 1) provides a set of functions to aid the expert programmer statically change (parts of) the *FastFlow* data-flow graphs by refactoring and fusing `BBs` to optimize the shape of the concurrency graph.

All high-level parallel patterns provided by the *FastFlow* upper layer (e.g., *ParallelFor*, *ParallelForReduce*, *Ordered Farm*, *Macro Data-Flow*, etc.) were implemented with the sequential and parallel *BBs* presented [2].

BBs usage example. A simple usage example of a subset of *FastFlow* `BBs` is presented in Fig. 2. In the top-left part of the figure, we defined three sequential nodes: `Reader`, `Worker`, and `Writer`. The `Reader` node takes as input a comma-separated list of directory names and produces in output a stream of *file_t* data elements each associated to a file contained in one of the input directories (the `ff_send_out` call at line 8 is used to produce multiple outputs for

```
1  #include <ff/ff.h>
2  using namespace ff;
3  struct Reader:ff_node_t<file_t> {
4   Reader(const char dir[])
5     :dir(dir){}
6   file_t *svc(file_t*) {
7    for(auto file:dir_iterator(dir))
8      ff_send_out(new file_t(file));
9    return EOS; // end-of-stream
10  }
11  const char[] dir;
12 };
13 struct Worker:
14       ff_node_t<file_t,result_t>{
15  result_t *svc(file_t *in) {
16   auto r=search(in,"Hello");
17   return (r ? r : GO_ON);
18  }
19 };
20 struct Writer:
21        ff_minode_t<result_t>{
22  result_t *svc(result_t*in){
23   R.push_back(in);
24   return GO_ON; // keep going
25  }
26  void svc_end() {print_result(R);}
27  std::vector<result_t*> R;
28 };
```

```
26 // ----- (Version 1) ----
27 int main() {
28  ff_pipeline pipe;
29  pipe.add_stage(new Reader(
30        "dir1,dir2,dir3,dir4"));
31  pipe.add_stage(new Worker());
32  pipe.add_stage(new Writer());
33  return pipe.run_and_wait_end();
34 }
```

```
26 // ----- (Version 2) ----
27 int main() {
28  ff_a2a a2a; // all-to-all
29  a2a.add_firstset<Reader>({
30       new Reader("dir1,dir2"),
31       new Reader("dir3,dir4")});
32  a2a.add_secondset<Worker>({
33       new Worker(),new Worker(),
34       new Worker()});
35  ff_pipeline pipe;// main pipe
36  pipe.add_stage(&a2a);
37  pipe.add_stage(new Writer());
38  return pipe.run_and_wait_end();
39 }
```

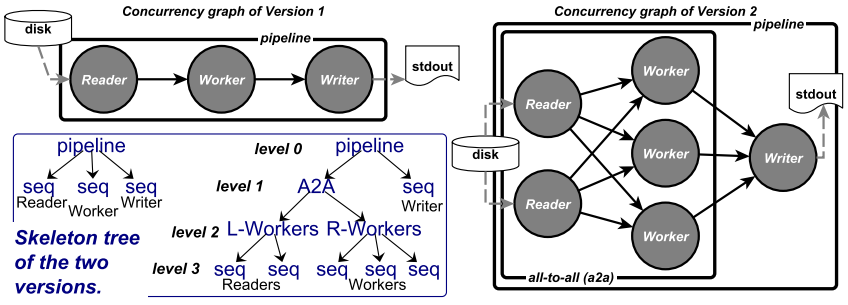


Fig. 2 Simple usage example of the *FastFlow* `BBs` implementing two different parallel skeletons of the same program. `Top`): (*Version 1*) three-stage pipeline of `SBBs`, and (*Version 2*) two-stage pipeline where the first stage replicates two times the `Reader` node and three times the `Writer` node by using the *all-to-all* `PBB`. `Bottom`): Graphical schema of the two versions and their skeleton trees showing the nesting levels of `BBs`

a single activation of the service method `svc()`); in the end, the `Reader` produces the special value *End-Of-Stream* (`EOS` at line 9) to start the pipeline termination of the next `BBs`. The `Worker` node executes a given `search` function on each input file, and then it produces in output only non-empty matching (the special value `GO_ON` at line 17 is not inserted into the output channel and

is meant to keep the node alive and ready to receive the next input). Finally, the `Writer` node collects all results, one at a time, and then writes the final result on the standard output by using the function `print_result` in the `svc_end()` method. Such method is called once by the *FastFlow* RTS when the node receives (all) the `EOS`(s) from the input channel(s) and before terminating.

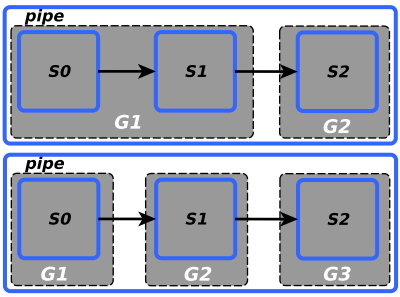
In the top-right part of Fig. 2 the three sequential nodes defined are instantiated and combined in two different concurrent streaming networks: the *Version 1)* is a standard 3-stage pipeline; the *Version 2)* is a 3-stage pipeline in which the first stage is an `a2a BB` replicating two and three times the `Reader` and `Worker` nodes, respectively. They will be automatically transformed into multi-output and multi-input nodes. At the bottom of the figure are sketched the schemes of the two versions and their skeleton trees showing the levels and nesting of `BBs`. The leaves of the tree are implemented as POSIX threads in the *FastFlow* RTS.

Previous *FastFlow* distributed RTS. The first versions of the *FastFlow* library (before v. 3.x) provided the programmer with the possibility to execute *FastFlow* programs on a distributed system [6]. Based on the ZeroMQ communication library, the distributed RTS was developed in 2012 by Massimo Torquati. Later a tiny message-passing layer atop InfiniBand RDMA was also implemented as a ZeroMQ alternative [7]. To support inter-process communications, the old *FastFlow* `node` was extended with an additional "external channel" (either in input or in output). The extended node was called `dnode`. Edge nodes of the *FastFlow* data-flow graph, once transformed into `dnodes`, could communicate with `dnodes` of others *FastFlow* applications running on different machines, through a pre-defined set of communication collectives (i.e., *unicast*, *onDemand*, *Scatter*, *Broadcast*, *fromAll*, *fromAny*). The programmer had to annotate each `dnode` with the proper collective endpoint to make them exchange messages according to the selected communication pattern.

There are many differences between the previous (old) version and the new one presented in this paper. We report here only the most relevant points wholly redesigned in the new version. In the old version, the programmer should explicitly modify the edge nodes of a *FastFlow* program to add the `dnode` wrapper with the selected communication pattern. It also had to define two non-trivial auxiliary methods for data serialization. Moreover, the old version exposed two distinct programming models to the programmer, one for the local node (i.e., streaming data-flow) and one for the distributed version (i.e., Multiple-Programs, Multiple-Data with collectives). Finally, the old version did not provide the *FastFlow*'s system programmer with any basic distributed mechanisms to define new high-level distributed patterns.

3 From Shared- to Distributed-Memory *FastFlow* Applications

This section presents the *FastFlow* library extensions to execute applications in a distributed-memory environment. By introducing a small number of modifications to programs already written using *FastFlow*'s `BBs`, the programmer may port its shared-memory parallel application to a hybrid implementation (shared-memory plus distributed-memory) in which parts of the concurrency graph will be executed




```
1 auto G1 = pipe.createGroup("G1");
2 auto G2 = pipe.createGroup("G2");
3 G1 << S0 << S1;
4 G2 << S2;
```

```
1 S0.createGroup("G1");
2 S1.createGroup("G2");
3 S2.createGroup("G3");
```

Fig. 3 Pipe example: creating *dgroups* from a three-stage *FastFlow* pipeline

in parallel on different machines according to the well-known SPMD model. The resulting distributed application will adhere to the same data-flow streaming semantics of the original shared-memory implementation. The modifications consist of identifying disjoint *groups* of `BBs`, called *distributed groups* (or simply *dgroups*), according to a small set of rules described in the following. Then mapping such distributed groups to the available machines through a JSON-format configuration file.

Each *dgroup* represents a logical partition of *FastFlow*'s `BBs` implementing a portion of the *FastFlow* streaming concurrency graph. It is implemented as a process that runs alone or together with other *dgroups* on a remote node. Furthermore, to exploit the full potential of a single node, a *dgroup* is internally realized as a shared-memory *FastFlow* application properly enriched with customized sequential `BBs` and node wrappers to transparently realize the communications among *dgroups* according to the original data-flow graph.

The API to define the distributed groups comprises two functions: the *dgroup* creator `createGroup` function, and the *dgroup* inclusion implemented through the C++ operator '<<'. A *dgroup* can be created from any level 0 or level 1 `BB` in the *FastFlow* skeleton tree. The `createGroup` function takes as an argument a string that uniquely identifies the distributed group. The inclusion operator is helpful when the programmer wants to create multiple *dgroups* from a single `BB`, and only a subset of its nested `BBs` need to be included in a given *dgroup*. It is worth noting that distributed *FastFlow* applications containing *dgroup* annotations can still be executed, upon recompilation disabling the distributed RTS, on a single node with its native shared-memory RTS without any modifications to the code (and any impact on the performance).

Figure 3 shows a generic application structured as a pipeline of three, possibly parallel, `BBs` where distributed groups are derived in two different ways. In the first case (on the top part of the figure) the *G1 dgroup* is composed of the first two stages while *G2* contains the last stage. In this example `S0`, `S1` and `S2` can be any valid nesting of the available `BBs`. As sketched in the code on the right-hand side of the figure, the two groups are created from the same pipeline at lines 1 and 2, and then the direct children `BBs` (i.e., those at level 1 of the skeleton tree) are included in the correct *dgroup* (lines 3 and 4). A *dgroup* created from a pipeline must have all included `BBs` contiguous to respect the pipeline order. The second case (bottom part

of Fig. 3) shows how to create 3 *dgroups* directly from the 3-stage pipeline. Indeed, when we create a *dgroup* from a `BB` at level 1 of the skeleton tree, we might be implicitly expressing the willingness to put the whole `BB` inside the *dgroup*. If this is the case, then there is no need to include all the nested `BBs` manually as the RTS automatically includes them all.

The `BBs` that can be added to a *dgroup* by using the '<<' operator are the direct children of the `BB` from which the group has been created. This constraint has two reasons: (1) to keep the implementation simple and manageable; (2) to not reduce too much the granularity of the single groups to have a coarse enough concurrency graph to be efficiently executed in a single multi-core node and thus capable of exploiting the available local parallelism. However, such constraints might be relaxed in future releases of the *FastFlow* library.

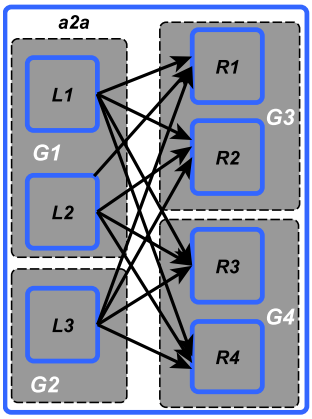
From the `node` and `farm` `BBs` it is possible to create only one *dgroup*,[1] whereas from the `pipeline` and `all-to-all` `BBs` it is possible to create multiple *dgroups*. Finally, a `BB` can be included only in a single *dgroup*.

Distributed groups from a single `a2a` `BB` can be derived in different ways, either by cutting the `a2a` graph horizontally or vertically or in both directions (i.e., oblique cuts). Let us call *inter-set cuts* those graph's cuts that group *BBs* from both L- and R-Worker sets of an *a2a*, and *non-inter-set cuts* those graph's cuts that group only L- or R-Worker *BBs*. Vertical and non-inter-set horizontal cuts produce only distributed communications between L-Worker and R-Worker `BBs`. Differently, in horizontal inter-set cuts, some *dgroups* contain both L-Worker and R-Worker `BBs` of the `a2a`. Therefore, some communications will happen in the shared-memory domain, while others will happen in the distributed-memory domain. When applicable, for example, for the *on-demand* and *round-robin* data distribution policies, the distributed RTS will privilege local communications vs. distributed communications.

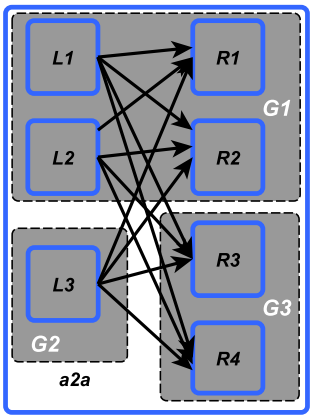
An example of four groups produced as a result of a vertical and two non-inter-set horizontal cuts for an `a2a` `BB` is sketched in the left-hand side of Fig. 4. In this example, some L-Worker `BBs` are grouped together in a single group (as a result of non-inter-set cuts). The same for some R-Worker `BBs`. Instead, in the right-hand side of Fig. 4, the same `a2a BB` is split into three *dgroups* by making two cuts, one horizontal inter-set producing the group *G1* that aggregates `BBs` from the two Worker sets, and one vertical cut producing two distinct *dgroups*, *G2* and *G3*. With such a division, communications between *L1*, *L2* and *R1*, *R2* `BBs` are local through SPSC shared-memory channels. In contrast, all the other pairs of communications among the L-Worker and R-Worker sets are distributed. The different kinds of communications are handled transparently by the *FastFlow* RTS. Notably, for remote communications, data types must be serialized. At the bottom of Fig. 4 is reported the code needed for creating the *dgroups* for both cases.

So far, we have introduced the basic grouping rules through simple generic examples showing the small extra coding needed to introduce *dgroups* in a *FastFlow* program. In the next example, we give a complete overview of a still straightforward but complete distributed *FastFlow* `BB`-based application. This application, sketched

[1] For the `farm`, this is a limitation of the current release that will be relaxed in the future.




```
1 a2a.createGroup("G1") <<L1 <<L2;
2 a2a.createGroup("G2") <<L3;
3 a2a.createGroup("G3") <<R1 <<R2;
4 a2a.createGroup("G4") <<R3 <<R4;
```




```
1 auto G1 = a2a.createGroup("G1");
2 a2a.createGroup("G2") <<L3;
3 a2a.createGroup("G3") <<R3 <<R4;
4 G1 <<L1 <<L2 <<R1 <<R2;
```

Fig. 4 A2A example: creating *dgroups* from a *FastFlow* `a2a` by aggregating some L-Worker and some R-Worker `BBs`. There are no restrictions on how to aggregate L- and R-Workers

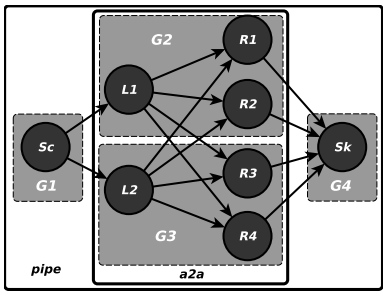
in Fig. 5, is made of a 3-stage pipeline in which the first and last stages are sequential `BBs`, and the middle one is a 2×4 `a2a BB`. The distributed version comprises four groups: *G1* containing the source node, *G2* containing the top half of the `a2a`, *G3* containing the bottom half part of the `a2a`, and *G4* containing the sink node. In this example, all the `dgroups` are created directly from the level 1 `BB`, whereas the inclusion operator is used to assign the `a2a`'s Worker `BBs` to the desired group. The definition of the `dgroups` is listed from line 24 to line 29 in the code listing on the left-hand side of Fig. 5. The changes compared to the shared-memory version are as follows: (1) at line 1 the include file `dff.hpp` enabling a set of distributed RTS features; (2) the `DFF_Init` function at line 6 needed to identify the *dgroup* name to execute and to collect all needed information provided by the launcher (e.g., location of the JSON configuration file); (3) the previously discussed annotations needed to create *dgroups* (from line 24 to 29). No modifications to the business logic code of the sequential nodes are needed. The JSON configuration file containing the mapping $<$ *dgroup* $-$ *host* $>$ is shown on the right-hand side of the figure. The mandatory information in the JSON file is the *name* of the group and the *endpoint* (i.e., the hostname or IP address). All other attributes are optional and are meant for optimizing the performance.

In line 1 we specify to use MPI instead of the default TPC/IP transport protocol. In line 3 we specify that the *dgroup G1* will send out messages in batches of 10 per destination, and in lines 9 and 13, in the same way, we specify that the *dgroups G2* and *G3* will send results to *G4* in batches of 20 messages. The batching of messages is *completely transparent* to the application, and it is particularly helpful for small-size messages to optimize the network link bandwidth. In addition, the JSON

```
#include <ff/dff.hpp>
#include <nodesDef.hpp>
using namespace ff;
int main(int argc,char *argv[]){
  // distributed RTS init ------
  DFF_Init(argc,argv);

  // ---- FastFlow graph -------
  SourceNode Sc;
  LeftNode   L1, L2;
  RightNode  R1,R2,R3,R4;
  SinkNode   Sk;
  ff_pipeline pipe;
  ff_a2a a2a;
  a2a.add_firstset<LeftNode>(
            {&L1, &L2});
  a2a.add_secondset<RightNode>(
         {&R1,&R2,&R3,&R4});
  pipe.add_stage(&Sc);
  pipe.add_stage(&a2a);
  pipe.add_stage(&Sk);

  // --- distributed groups ----
  auto G1=Sc.createGroup("G1");
  auto G2=a2a.createGroup("G2");
  G2 << L1 << R1 << R2;
  auto G3=a2a.createGroup("G3");
  G3 << L2 << R3 << R4;
  auto G4=Sk.createGroup("G4");
  // -- distributed execution --
  return pipe.run_and_wait_end();
}
```




```
{"protocol" : "MPI",
 "groups": [{
  "batchSize": 10,
  "name": "G1",
  "messageOTF": 64,
  "endpoint":"node1"
  },{
  "name": "G2",
  "batchSize": 20,
  "threadMapping": "0,2,4,6,8",
  "endpoint": "node2"
  }, {
  "batchSize": 20,
  "name": "G3",
  "endpoint": "node3"
  },{
  "name": "G4",
  "endpoint": "node4"
}]}
```

Fig. 5 A complete application example composed of a 3-stage pipeline: `multi-output node`, `a2a`, `multi-input node`. The distributed version comprises four `dgroups`, two coming from the `a2a` `BB` through an horizontal inter-set cut. The definition of Sk,Sc,L_i, and R_i nodes, is not shown

configuration file may contain other non-functional attributes to regulate some low-level knobs of the *FastFlow* RTS, for example the thread-to-core affinity mapping for each *dgroup* (`threadMapping`), or to set the size of the logical output queues representing the distributed channels through the attributes `messageOTF`/`internalMessageOTF`, which set the maximum number of "on-the-fly" messages for a channel. Currently, the configuration file must be entirely provided by the user in charge of mapping between *dgroups* and hosts. However, we defined the mechanisms that will allow us to automatize the generation of the JSON file with simple ready-to-use mappings.

For the sake of completeness, the pseudo-code of Fig. 6 gives a complete overview of the semantics checks of the two functions provided by the distributed-memory *FastFlow* RTS, namely *createGroup* and *addToGroup*, the latter mapped onto the inclusion operator '<<'. We also included the *runGroup* function, which

1: **function** CREATEGROUP(bb, $groupName$)
2: **if** $\exists G : name(G) = groupName$ **then** *error()*
3: **end if**
4: **if** $type(bb) \in \{seq, comb, farm\} \wedge \exists G : parent(G) = bb$ **then** *error()*
5: **end if**
6: **if** $level(bb) > 1$ **then** *error()*
7: **end if**
8: **if** $level(bb) = 0 \wedge \exists G : level(parent(G)) = 1$ **then** *error()*
9: **end if**
10: $Groups \leftarrow Groups \cup \{G_{groupName}\}$
11: **end function**
12: **function** ADDTOGROUP(bb, g)
13: **if** $bb \notin parent(g)$ **then** *error()*
14: **end if**
15: **if** $\exists g' : bb \in g'$ **then** *error()*
16: **end if**
17: $g \leftarrow g \cup \{bb\}$
18: **end function**
19: **function** RUNGROUP(g)
20: **if** $type(parent(g)) = pipe$ **then** $checkPipe(g, parent(g))$
21: **end if**
22: $parseJSON()$
23: ffg $\leftarrow buildFFnet(g)$
24: run(ffg)
25: **end function**

name(g)	returns the unique name of the group g
parent(g)	returns the BB from which the group g has been created
type(bb)	returns bb type $\in \{seq, comb, farm, pipe, a2a\}$
checkPipe(g, p)	checks contiguousness of pipe stages p in g
level(bb)	returns the nesting level of the BB bb in the skeleton tree
parseJSON()	parses the configuration file
buildFFnet(g)	generates the *FastFlow* concurrent graph for the group g
run(g)	executes the *FastFlow* graph implementing the group g

Fig. 6 Pseudo-code of the rules used for creating valid *dgroups*. The *runGroup* contains the pseudo-code to build the *dgroup* and run it as a native *FastFlow* application

the programmer does not directly call since it is automatically invoked by RTS passing the proper group name. It builds the *FastFlow* graph implementing the given *dgroup* and runs it.

Data serialization. Data serialization/deserialization (briefly data serialization from now on) is a fundamental feature of any distributed RTS. It is the process of transforming a possibly non-contiguous memory data structure into a format suitable to be transmitted over a network and later reconstructed, possibly in a completely different computing environment, preserving the original data. In the distributed *FastFlow* RTS, data serialization can be carried out in two different ways. The programmer may select the best approach, between the two, for each

```
1 struct data_t {
2   std::string key;
3   uint64_t id, ts;
4
5   template<class Archive>
6   void serialize(Archive & ar){
7       ar(key,id,ts);
8   }
9 };
```

```
1  struct data_t {
2    char key[MAXWORD];
3    uint64_t id, ts;
4  };
5  template<class Pair> // <ptr,size>
6  void serialize(Pair &p, data_t *d){
7    p={(char*)d, sizeof(rdata_t)};
8  }
9  template<class Pair> // <ptr,size>
10 void deserialize(const Pair &p,
11                    data_t *&d){
12   d = new (p.ptr) data_t;
13 }
```

Fig. 7 (`Left`) *Cereal*-based serialization function of the `data_t` type. (`Right`) custom serialization of a memory-contiguous version of the `data_t` type enabling zero-copy transfers

data type flowing into the inter-group channels (i.e., the data types produced/received by the edge nodes of a *dgroup*).

The first approach employs the *Cereal* serialization library [8]. It can automatically serialize base C++ types as well as compositions of C++ standard-library types; for instance, a `std::pair` containing a `std::string` and a `std :: array < int >` objects can be serialized without writing any extra line of code. *Cereal* requests a serialization function only for user-defined data types. A user-defined data type containing an explicit (yet straightforward) serialization function is sketched on the left-hand side of Fig. 7. The second approach allows the user to specify its serialization and deserialization function pair entirely. This might be useful, when feasible, to avoid any extra copies needed by the serialization process itself. This method is beneficial when the data types are contiguous in memory (i.e., trivial types in C++), thus a zero-copy sending protocol can be employed. An example of this custom approach is shown on the right-hand side of Fig. 7.

dgroups implementation and program launching. A *dgroup* is implemented through the *FastFlow*'s `farm BB`. The Emitter is the *Receiver*, and the Collector is the *Sender*. The `farm`'s Workers are the `BBs` of the original application graph included in that particular *dgroup* (either implicitly or explicitly). The `BBs` that communicate with the Sender and/or Receiver via shared-memory *FastFlow* channels are automatically wrapped by the RTS with class wrappers that transparently perform serialization activities on the input/output data-types of the `BBs`. Such activities happen in parallel with data communications. Horizontal inter-set cuts in an `a2a` are implemented using customized `BBs` in the L-Worker and R-Worker sets. Concerning *FastFlow* program launching, we have designed a software module called `dff_run`. It takes care of launching the application processes, each with the appropriate parameters (e.g., *dgroup* name), following the mapping host-group described in the JSON configuration file. When the transport protocol is MPI, the `dff_run` is just a wrapper of the well-known *mpirun* launcher. It produces a suitable `hostfile/rankfile` which will be handled by *mpirun*. The MPI backend guarantees high-end HPC cluster accessibility.

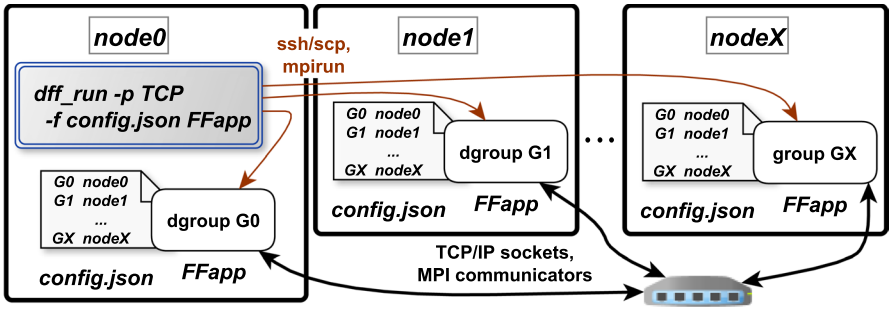


Fig. 8 Logical schema of the launching of a *FastFlow* distributed application with `dff_run`

A simplified overview of the launching phase when using the TCP/IP protocol as a communication layer is sketched in Fig. 8. The current version of the `dff_run` launcher does not deploy the *FastFlow* executables and the JSON file on the remote hosts. This limitation will be addressed in the next releases.

4 Experimental Evaluation

Experiments were conducted on the *openhpc2* cluster hosted by the Green Data Center of the University of Pisa. It has 16 nodes interconnected at 1 Gbit/s. Each node has two Intel Silver Xeon 4114 CPUs running at 3.0 GHz for a total of 20 physical cores (40 hardware threads) and 128 GB of RAM.

The first test evaluates the throughput attainable using different message sizes as well as the impact of varying the `batchSize` attribute in the JSON configuration file without modifying the program. This test considers two nodes of the cluster each running a *dgroup* and using the default TCP/IP protocol. The results are shown in Fig. 9 (top left-hand side). The ideal throughput is the one measured using the `netcat` network utility program using the TCP/IP protocol. As expected, the transparent batching feature is particularly useful for small messages and becomes less relevant for messages bigger than 512B. A batch size of 8–32 messages is enough to reach the maximum throughput attainable. The same test has been repeated using the MPI protocol (MPI over TCP/IP). The results are sketched on the bottom left-hand side of Fig. 9. No appreciable differences are present between the two protocols.

The second test evaluates the cost of the automatic serialization using the Cereal library. We compared Cereal-based serialization (the default) to manual serialization of a memory-contiguous data type that allows the RTS to perform a zero-copy message transfer. In this case, to avoid potential bottlenecks introduced by the network, both sender and receiver *dgroups* are mapped on the same node on different CPUs through the *threadMapping* configuration file attribute. The top right-hand side of Fig. 9 shows the results obtained. The two serialization approaches behave the same for messages smaller than 8KB, while above that threshold, manual serialization has less overhead, as expected. However,

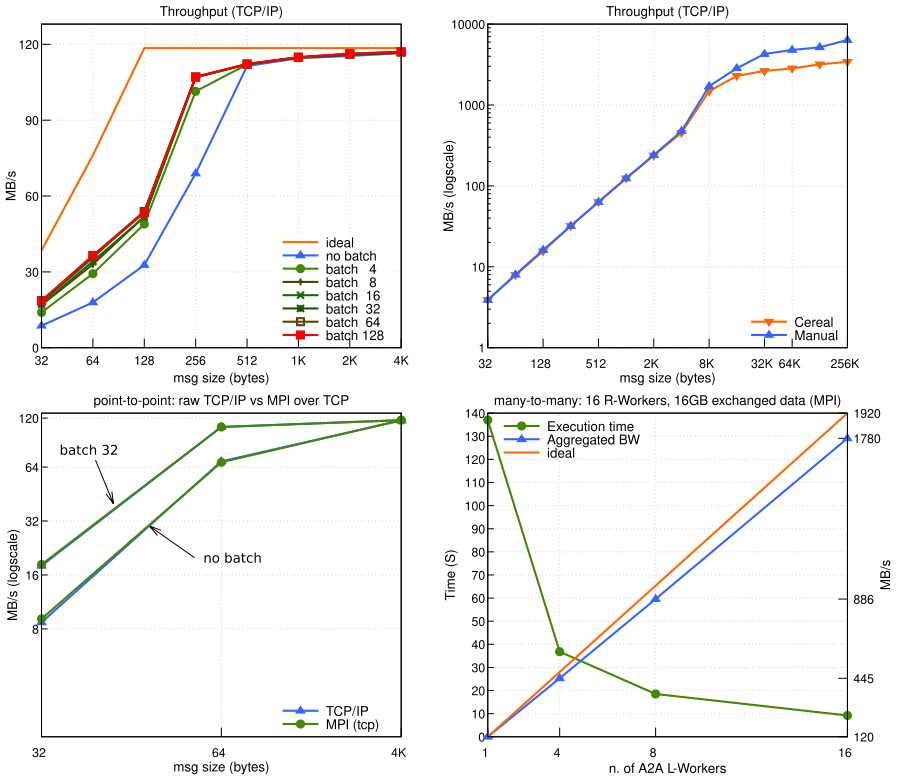
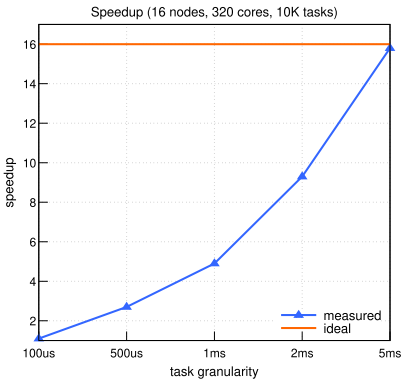


Fig. 9 Benchmarks. `Top left)`: Throughput between two cluster nodes (and two *dgroups*) varying the message size and the *batchSize*. `Top right)`: Throughput between two *dgroups* on a single cluster node by using Cereal-based vs. Manual serialization (no batching, TCP protocol). `Bottom left)`: Throughput comparison between TCP/IP and MPI over TCP/IP protocols considering two *dgroups* on two cluster node. `Bottom right)`: Execution time and aggregated bandwidth (in MB/s) for a single *a2a* considering 16 R-Worker *dgroups* and varying the L-Worker *dgroups*. 16GB data exchanged using the MPI over TCP/IP protocol

significant variances in performance are located above 1.2 GB/s. Therefore, we could expect almost no differences for applications running on clusters whose interconnection is up to 10 Gbit/s.

The third test evaluates the *a2a* aggregated bandwidth. We considered an *a2a* with a fixed number of 16 R-Workers each in a distinct *dgroup*; a file of 16 GB data that has to be partitioned among the R-Workers and a varying number of L-Workers, acting as producers, each in a separated *dgroup*. The generic L-Worker reads a contiguous portion of the file and sends the partition chunks to the proper R-Worker (e.g., with 4 L-Workers, the L-Worker 1 sends 4 chunks of 1 GB to the R-Workers 4, 5, 6, 7, respectively). Except for the case of 16 L-Workers and 16 R-Workers, all communications among *dgroups* are forced to be remote by avoiding mapping L- and R-Worker *dgroups* on the same node. The results obtained are shown in Fig. 9 (bottom right-hand side). The aggregated



Task Grain (ms)	Ideal time (S) 1 node 20 Workers	Exec Time (S) 16 nodes 320 Workers	Speedup
0.1	0.5	0.464	1.08
0.5	2.5	0.915	2.73
1	5	1.02	4.90
2	10	1.075	9.30
5	25	1.583	15.79

Task Grain (ms)	N. of tasks assigned to Workers		
	in Master dgroup	in other dgroups	
		Average	StdDev
0.1	89011	732.1	10.1
0.5	36048	4263.5	12.9
1	20339	5309.2	8.2
2	10680	5952.9	3.5
5	6239	6250.1	15.1

Fig. 10 Master–Worker experiment. (`Left`) Speedup varying the task's computation granularity. (`Right-Top`) Execution time (in seconds) and speedup for each task grain (in milliseconds). (`Right-Bottom`) tasks computed by the Master *dgroup* and by all others *dgroups*

bandwidth scales well with the number of L-Workers reaching a maximum of 1780 MB/s, about 93% of the total attainable bandwidth.

The next test mimics a Master–Worker parallel pattern implementation using *FastFlow* `BBs`. The starting point is a *FastFlow* shared-memory micro-benchmark using a `a2a BB`, in which a single *multi-output* sequential `BB` in the L-Worker set implements the Master, and a set of sequential `multi-input BBs` in the R-Worker set implement the Workers. The Master generates 100K tasks at full speed. Each task is a message whose payload is 128B. For each input task, the Workers execute a controlled synthetic CPU-bound computation corresponding to a predefined time (we considered values in the range $0.1 - 5$ milliseconds). The task scheduling policy between L- and R-Worker sets is *on-demand*. The distributed version is derived from the shared-memory benchmark by cutting the *a2a BB* graph both horizontally and vertically. The horizontal inter-set *dgroup* aggregates the Master and 20 Workers of the R-Worker set. The vertical *dgroups* aggregate the remaining Workers of the R-Worker set (20 Workers for each *dgroup* to fill in all physical cores of a node). Distinct *dgroups* are deployed to different cluster nodes. The tables on the right-hand side of Fig. 10 summarize the results obtained using all physical cores of the *openhpc2* cluster (i.e., 320 cores in total). All tests have been executed using a transparent batching of 32 messages and 1 as the maximum number of messages on-the-fly. The baseline is the ideal time on a single node considering the task granularity, e.g., for tasks of 100 μs, the ideal execution time is 500 ms. The speedup increases with computational granularity, and the number of tasks computed by local Workers in the Master group is inversely proportional to the task granularity. This is what we can expect from the *on-demand* task scheduling policy that privileges local *dgroup* Workers: the coarser the grain of tasks, the higher the number of tasks sent to remote Workers. This point is confirmed by the data measured and reported in the table on the bottom right-hand side of Fig. 10. When the task granularity is small (e.g., 0.1 ms), local Workers in the Master *dgroup* receive much more tasks than Workers

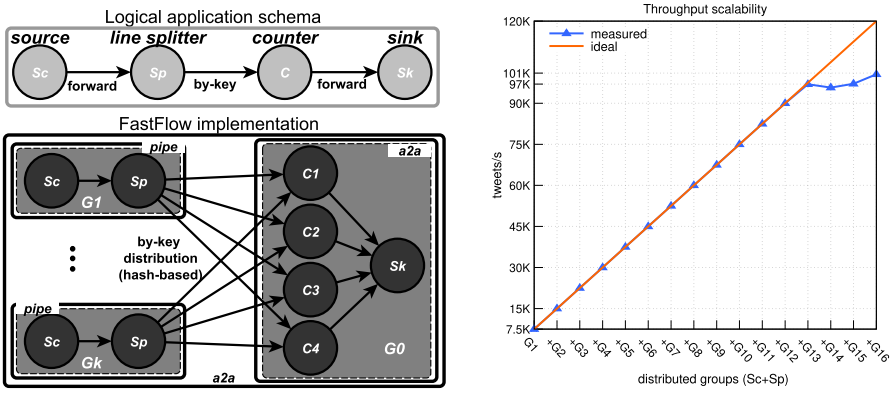


Fig. 11 WordCount test. (`Left`) Logical schema and the *FastFlow* implementation. (`Right`) Throughput scalability (tweets/s) increasing the number of source-splitter *dgroups*

in remote *dgroups*. As a final note for this test, since the completion time is quite short, an initial barrier has been artificially introduced in the *DFF_Init* to synchronize all *dgroups* and obtain a more accurate measurement. The reported times are thus the maximum time observed among all *dgroups* without considering the spawning time introduced by the `dff_run` launcher (which accounts for about 200 ms).

The last experiment is *WordCount*, a well-known I/O-bound streaming benchmark. Its logical data-flow schema is sketched on the top left-hand side of Fig. 11. There is a source stage (*Sc*) that reads text lines from a file or a socket; a line tokenizer or splitter (*Sp*) that extracts words from the input line and sends all words with the same hash value (called key) to the same destination; a counter (*C*) that counts the number of instances of each word received; and a sink (*Sk*) that collects words and prints all statistics (e.g., unique words, current number of words, etc.). The native shared-memory version obtains the best performance. However, with this test, we want to study the throughput scalability and stress the communications routing feature of the distributed RTS. Our test considers Twitter's tweets as text lines (max 280 characters including spaces), multiple replicas of the pair source-splitter stages (to emulate tweet streams coming from various sources), four replicas of the counter, and one sink stage. The *FastFlow* data-flow graph implementing the test is shown on the bottom left-hand side of Fig. 11. The grey rectangles identify the *dgroups*: each source-splitter replica is part of a *dgroup* (*G1*...*Gk* in the figure), whereas the counter replicas and the sink stage own to a single *dgroup* (*G0* in the figure). By running a single replica of the source-splitter *dgroup* and the counter-sink *dgroup* on the same cluster node on different CPU cores with `batchSize=32`, we found a maximum attainable throughput of about 120K tweets/s. Consequently, we configured each source stage to constantly produce $120/16 = 7.5$K tweets/s towards the splitter stage. Then, to stress-test the shuffle communication pattern with the by-key distribution, we replicated the source-splitter *dgroup* multiple times (up to 16 replicas). The results obtained in these tests are plotted on the left-hand side of Fig. 11. The scalability is linear up to 13 replicas for an aggregated throughput of about 97K tweets/s, then it flattens until we reach 16 *dgroup* replicas where the last

replica (i.e., *G*16) is executed on the same node of the counter-sink *dgroup G*0 (*G*16 and *G*0 are mapped on different CPU cores) thus leveraging more bandwidth due to the inter-node communications and reaching a maximum throughput of about 101K tweets/s (i.e., about 84% of the maximum).

4.1 Results discussion

Summarizing the results obtained: (a) the *FastFlow* shared-memory streaming model is transparently preserved when porting to distributed-memory domains, applications that use nontrivial communication patterns (e.g., hash-based message scheduling in WordCount), and with both horizontal as well as vertical cuts of the concurrent graph; (b) the designed distributed RTS can achieve close to nominal bandwidth on 1 Gbit/s networks using the TCP/IP and MPI over TCP protocols; (c) the transparent batching feature, is helpful to optimize communications for small messages without modifying the application code; (d) the RTS can efficiently balance tasks workload among multiple distributed groups yet privileging local communications (i.e., towards group local Workers) to minimize communication overheads; (e) streaming computations with computational tasks of a few milliseconds can benefit from the distributed RTS to reduce the execution time.

5 Related Work

The *de facto* standard parallel programming model for shared-memory parallel systems is *OpenMP*, whereas, in the HPC context, the most broadly used model is "MPI + X", where X is usually either OpenMP or CUDA [9, 10]. However, several higher-level parallel programming libraries or domain-specific languages (DSLs) have been proposed in the context of structured parallel programming [4]. Several of them are implemented in C/C++ (e.g., SkePU [11], GrPPI [12], SkeTo [13], SkelCL [14], Muesli [15]), some others are DSL-based such as Musket [16] and SPar [17]. High-level parallel programming frameworks abstract the low-level threading, communication, and synchronization details necessary for effectively utilizing parallelism and liberate the programmer from writing error-prone concurrent procedures.

Recently some of the concepts coming from the algorithmic skeletons and parallel design patterns research communities have also fertilized some commercial/industrial programming environments such as Intel TBB [18] for multi-core parallelism, Khronos SYCL [19] for heterogeneous many-core equipped with GPUs and Apache Spark [20] and Apache Flink [21] for cluster-level data-stream processing. In recent years there was a proliferation of frameworks aiming to ease the communication in distributed systems [22]. For example, in the HPC context, Mercury [23] leverages multiple HPC fabrics protocols to implement remote procedure calls. Instead, in big data analytics and cloud contexts, ActiveMQ and ZeroMQ[2] are among the most used messaging systems.

[2] ActiveMQ: https://activemq.apache.org/. ZeroMQ:https://zeromq.org/.

FastFlow [1, 2] has been developed in the context of structured parallel programming methodology, mainly targeting streaming applications. What primarily characterizes *FastFlow* compared to other notable approaches in the field is its ambition to offer different yet structured software layers to the system and application programmers. At the bottom level of the software abstraction, a reduced set of flexible, efficient, and composable `BBs` are provided for building new domain-specific frameworks such as WindFlow [24], and highly-distributed streaming networks. *BBs* mainly target parallel-expert programmers. At the higher level of the software abstraction, *FastFlow* provides some well-known parallel exploitation patterns (e.g., ParallelFor and D &C) mainly targeting application developers. Currently, all patterns provided by the library can be used only inside a single *distributed group*. In contrast to other skeleton-based frameworks that also support distributed systems (e.g., SkePU, GrPPI, Muesli), the principle in the *FastFlow* distributed `BBs` layer is that the set of actors composing the parallel program (i.e., nodes) is partitioned in a set of groups mapped to different nodes. Conversely, data streams are not automatically partitioned; the streams' items are routed to the destination according to the distribution policy selected. In the version described in this paper, the programmer uses a specific API to partition the set of nodes into distributed groups and thus defines their mapping. In future releases, automatic data partitioning will be provided to the users through distributed implementations of some well-known data-parallel patterns [25].

6 Conclusion

We extended the *FastFlow*'s `BBs` layer with a new RTS enabling the execution of `BBs`-based *FastFlow* applications on distributed platforms. Changes to the codebase required to port the applications to the hybrid shared/distributed-memory environments are minimal and straightforward to introduce. First experiments conducted on a 16-node cluster demonstrate that: (i) the new distributed RTS preserves the *FastFlow* programming model and does not introduce unexpected overheads; (ii) the transparent batching of messages is a useful feature for tuning the distributed application throughput. Future extensions will consider: (a) adding support for the `farm BB` and bearing cyclic *FastFlow* networks; (b) introducing heuristics for automatically defining *dgroups* to relieve the programmer from this decision; (c) augmenting the number of transport protocols provided to the user, and enabling the coexistence of multiple protocols on different zones of the *FastFlow*'s nodes graph; (d) developing some high-level parallel patterns using the distributed RTS; (e) expanding the functionalities of the *dff_run* launcher to improve the deployment phase by integrating with the most popular workload managers (e.g., SLURM).

Acknowledgements This work has been partially supported by the ADMIRE project, "Adaptive Multi-Tier Intelligent Data Manager for Exascale" (EuroHPC-01-2019, G.A. n. 956748), the EUPEX project, "European Pilot for Exascale" (EuroHPC-02-2020, G.A. n. 101033975), and the TEXTAROSSA project, "Towards Extreme Scale Technologies and Accelerators for EuroHPC HW/SW Supercomputing Applications for Exascale" (EuroHPC-01-2019, G.A. n. 956831).

Author Contributions Authors' contributions is as follows: NT: conceptualisation, investigation, software, experiments, writing—original draft. MT: conceptualisation, investigation, supervision, software, validation, writing—original draft. GM: conceptualisation, writing—review and editing. MD: conceptualisation, writing—review and editing.

Declarations

Conflict of Interest The authors have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. In: Programming multi-core and many-core computing systems, parallel and distributed computing (2017). https://doi.org/10.1002/9781119332015.ch13
2. Torquati, M.: Harnessing Parallelism in Multi/Many-Cores with Streams and Parallel Patterns. PhD thesis, University of Pisa (2019)
3. Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., Torquati, M.: Design patterns percolating to parallel programming framework implementation. Int. J. Parallel Program. **42**(6), 1012–1031 (2014). https://doi.org/10.1007/s10766-013-0273-6
4. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput. **30**(3), 389–406 (2004). https://doi.org/10.1016/j.parco.2003.12.002
5. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: An efficient unbounded lock-free queue for multi-core systems. In: Euro-Par 2012 Parallel Processing, pp. 662–673. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32820-6_65
6. Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., Torquati, M.: Targeting distributed systems in fastflow. In: Proceedings of the 18th International Conference on Parallel Processing Workshops. Euro-Par'12, pp. 47–56. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-36949-0_7
7. Secco, A., Uddin, I., Pezzi, G.P., Torquati, M.: Message passing on infiniband rdma for parallel run-time supports. In: 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 130–137 (2014). https://doi.org/10.1109/PDP.2014.23
8. Grant, W.S., Voorhies, R.: Cereal a c++ 11 library for serialization. https://github.com/USCiLab/cereal (2013)
9. Rabenseifner, R., Hager, G., Jost, G.: 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, pp. 427–436 (2009). https://doi.org/10.1109/PDP.2009.43. IEEE
10. Smith, L., Bull, M.: Development of mixed mode MPI/OPENMP applications. Sci. Program. **9**(2, 3), 83–98 (2001). https://doi.org/10.1155/2001/450503
11. Ernstsson, A., Ahlqvist, J., Zouzoula, S., Kessler, C.: Skepu 3: Portable high-level programming of heterogeneous systems and HPC clusters. Int. J. Parallel Program. **49**(6), 846–866 (2021). https://doi.org/10.1007/s10766-021-00704-3

12. López-Gómez, J., Muñoz, J.F., del Rio Astorga, D., Dolz, M.F., Garcia, J.D.: Exploring stream parallel patterns in distributed MPI environments. Parallel Comput. **84**, 24–36 (2019). https://doi.org/10.1016/j.parco.2019.03.004
13. Tanno, H., Iwasaki, H.: Parallel skeletons for variable-length lists in sketo skeleton library. In: European Conference on Parallel Processing, pp. 666–677 (2009). https://doi.org/10.1007/978-3-642-03869-3_63
14. Steuwer, M., Kegel, P., Gorlatch, S.: Skelcl—a portable skeleton library for high-level GPU programming. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PHD Forum, pp. 1176–1182 (2011). https://doi.org/10.1109/IPDPS.2011.269
15. Ciechanowicz, P., Kuchen, H.: Enhancing Muesli's data parallel skeletons for multi-core computer architectures. In: 2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC), pp. 108–113 (2010). https://doi.org/10.1109/HPCC.2010.23
16. Rieger, C., Wrede, F., Kuchen, H.: Musket: A domain-specific language for high-level parallel programming with algorithmic skeletons. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. SAC'19, pp. 1534–1543. ACM, New York, NY, USA (2019). https://doi.org/10.1145/3297280.3297434
17. Griebler, D., Danelutto, M., Torquati, M., Fernandes, L.G.: Spar: a DSL for high-level and productive stream parallelism. Parallel Process. Lett. **27**(01), 1740005 (2017). https://doi.org/10.1142/S0129626417400059
18. Kukanov, A., Voss, M.J.: The foundations for scalable multi-core software in intel threading building blocks. Intel Technol. J. (2007). https://doi.org/10.1535/itj.1104.05
19. Reyes, R., Lomüller, V.: Sycl: Single-source C++ accelerator programming. In: Parallel Computing: On the Road to Exascale, IOS Press, vol. 27, pp. 673–682 (2016). https://doi.org/10.3233/978-1-61499-621-7-673
20. Salloum, S., Dautov, R., Chen, X., Peng, P.X., Huang, J.Z.: Big data analytics on apache spark. Int. J. Data Sci. Anal. **1**(3), 145–164 (2016). https://doi.org/10.1007/s41060-016-0027-9
21. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink: Stream and batch processing in a single engine. Bull. IEEE Comput. Soc. Tech. Comm. Data Eng. **36**(4), 28–38 (2015)
22. Ramon-Cortes, C., Alvarez, P., Lordan, F., Alvarez, J., Ejarque, J., Badia, R.M.: A survey on the distributed computing stack. Comput. Sci. Rev. **42**, 100422 (2021). https://doi.org/10.1016/j.cosrev.2021.100422
23. Soumagne, J., Kimpe, D., Zounmevo, J.A., Chaarawi, M., Koziol, Q., Afsahi, A., Ross, R.B.: Mercury: enabling remote procedure call for high-performance computing. In: CLUSTER, pp. 1–8 (2013). https://doi.org/10.1109/CLUSTER.2013.6702617
24. Mencagli, G., Torquati, M., Cardaci, A., Fais, A., Rinaldi, L., Danelutto, M.: Windflow: high-speed continuous stream processing with parallel building blocks. IEEE Trans. Parallel Distrib. Syst. **32**(11), 2748–2763 (2021). https://doi.org/10.1109/TPDS.2021.3073970
25. Danelutto, M., Torquati, M.: Loop parallelism: a new skeleton perspective on data parallel patterns. In: 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 52–59 (2014). https://doi.org/10.1109/PDP.2014.13