HitFlow: A Dataflow Programming Model for Hybrid Distributed- and Shared-Memory Systems

**Javier Fresno · Daniel Barba ·
Arturo Gonzalez-Escribano ·
Diego R. Llanos**

Abstract Dataflow programming consists in developing a program by describing its sequential stages and the interactions between them. The runtimes supporting this kind of programming are responsible for exploiting the parallelism by concurrently executing the different stages as soon as their dependencies are met.

In this paper we introduce a new parallel programming model and framework based on the dataflow paradigm. It presents a new combination of features that allows to easily map programs to shared or distributed memory, exploiting data locality and affinity to obtain the same performance than optimized coarse-grain MPI programs. These features include: It is a unique one-tier model that supports hybrid shared- and distributed-memory systems with the same abstractions; it can express activities arbitrarily linked, including non-nested cycles; it uses internally a distributed work-stealing mechanism to allow Multiple-Producer/Multiple-Consumer configurations; and it has a runtime mechanism for the reconfiguration of the dependences and communication channels which also allows the creation of task-to-task data affinities. We present an evaluation using examples of different classes of applications. Experimental results show that programs generated using this framework deliver good performance in hybrid distributed- and shared-memory environments, with a similar development effort as other dataflow programming models oriented to shared-memory.

Keywords Distributed systems · Dynamic computation · Parallel programming models · Streaming computation

J. Fresno · D. Barba · A. Gonzalez-Escribano · D. R. Llanos
Departamento de Informática, Universidad de Valladolid, Valladolid, Spain

J. Fresno E-mail: jfresno@infor.uva.es
D. Barba E-mail: daniel@infor.uva.es
A. Gonzalez-Escribano E-mail: arturo@infor.uva.es
D. R. Llanos E-mail: diego@infor.uva.es

1 Introduction

The most common programming tools for parallel machines are based on message passing libraries, such as MPI [1], or shared memory APIs like OpenMP [2]. These tools allow the programmers to exploit the capabilities of the machines by explicitly define the parallel sections inserted in the sequential code and program inter-process synchronizations and communications.

On the other hand, stream and dataflow libraries and languages (such as FastFlow [3], CnC [4], OpenStream [5], or S-Net [6]) reduce the complexity of creating a parallel program because the programmer only has to define the sequential stages and its dependencies. It is the responsibility of the runtime to control the sequential stages execution and perform the data synchronizations. However, these models do not present specific features to express some computational patterns, or to obtain communication-efficient implementations on distributed processes.

In this work we propose a novel combination of features for dataflow programming models: (a) A single one-tier representation for shared- and distributed-memory architectures; (b) Description of a program as a reconfigurable network of activities and typed data containers arbitrarily interconnected, with a generic system to represent distributed *Multiple-Producer/Multiple-Consumer* (MPMC) configurations; (c) Support for dependence structures that involve non-nested feedback loops; (d) A mechanisms to reconfigure dependences at runtime without creating new tasks; and (e) A mechanism to intuitively express task-to-task affinities which would allow a better exploitation of data locality across state-driven activities. As a proof of concept we have devised HitFlow, a new dataflow parallel programming model and framework that extends a previous proposal [7] to include all these features. Table 1 shows a comparison of different dataflow solutions in terms of these features.

This combination of features allows the creation of networks of tasks that can be mapped to message-passing processes with a fixed scheduling. The capacity of reconfiguring the dependences and activities of a task allows the runtime modification of the communication pattern used at each computation stage, without the need of creating or scheduling new tasks. Tasks can allocate on their local contexts buffers, or data parts assigned with a classical data partition policy, that persist across different stages. In this way, data can maintain the affinity with the message-passing processes and across related tasks, avoiding costly migrations and optimizing the communications. This scheme leads to implementations with similar performance and scalability than programs manually developed and optimized using message-passing models, such as MPI.

We present an evaluation of our proposal using examples of four different application classes. We describe how they are represented in our model, showing how to express different types of parallel paradigms including static and dynamic synchronization structures. Experimental work has been carried out to prove that the programs generated using our framework achieve good performance in comparison with manually developed implementations using

	FastFlow	CnC	OpenStream	S-Net	HitFlow
Single tier model		✓		✓	✓
Reconfigure dependencies		✓			✓
Allows tasks affinities					✓
MPMC configurations	✓	✓	✓		✓
Feedback loops	✓	✓		✓	✓

Table 1: Comparison of dataflow libraries.

both message-passing libraries such as MPI, and state-of-the-art tools for parallel dataflow programming, like FastFlow [3] or CnC [4]. These experiments show that the overheads introduced by the new abstractions do not have a significant impact on performance. Finally, an analysis of different development effort metrics shows that the cost of programming using our proposal, targeting hybrid distributed- and shared-memory systems, is similar to other shared-memory dataflow approaches, highly reducing the programming cost comparing with using message passing directly.

The rest of the paper is organized as follows. Section 2 describes our proposal. A discussion about its usage is given in Sect. 3 while Sect. 4 shows the implementation details. Section 5 presents the experimental work carried out to test the implementation. Section 6 describes some related work in the field. Finally, the conclusions of the paper are in Sect. 7.

2 The HitFlow model

In this section we present HitFlow, a new parallel programming framework implemented in C++ that exploits dataflow parallelism for both shared- and distributed-memory systems. The HitFlow programming model takes its notation from Colored Petri nets [8]. A HitFlow program is a network composed of two kinds of nodes, called *places* and *transitions*. The places are shared-data containers that keep *tokens*, while the transitions are the sequential processing components of the system. Transitions are connected by directed channels to places, with the direction determining the input or output role of places for each transition (see Fig. 1). A transition takes one token from each of its input places and performs some activity with them. It may then add tokens to any/all of its output places. This activity is repeated while there are tokens arriving to the input places.

We propose the computation inside the transitions to be mode-driven. Using a mathematical notation, a program or computation is represented by $P = \{p_1, p_2, \ldots, p_n\}$ a finite set of places, and $T = \{t_1, t_2, \ldots, t_m\}$ a finite set of transitions.

The transitions are composed of modes: $t_i = \{m_i^1, m_i^2, \ldots, m_i^o\}$. Each mode m_i^j is a tuple $\langle f, I, O, \text{next} \rangle$, where f is a sequential function, $I \subseteq P$ are the input channels, $O \subseteq P$ are the output channels, and $\text{next} \in \{m_i^1, m_i^2, \ldots, m_i^o\} \cup \text{END}$ is the mode that will be activated after the current mode m_i ends.
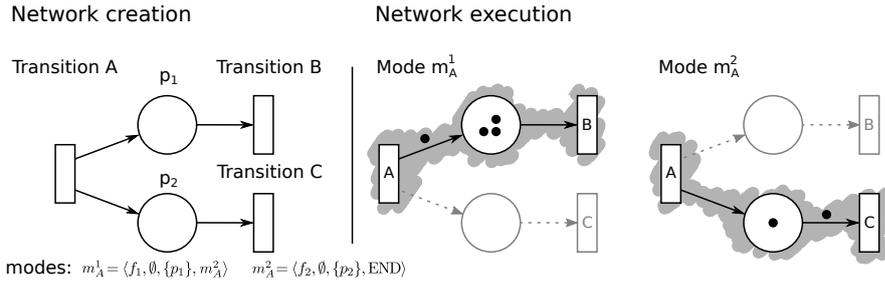
Fig. 1: Network example with modes. Transition A has two modes (A_1 and A_2), each mode enables a different output channel connecting A with B or A with C.

Modes are used to define mutually-exclusive activities inside the transitions, and dynamically reconfigure the network. A mode enables a subset of connections to input places or output places. The sequential function is executed when tokens arrive in the input places of the active mode. A transition changes its mode when all the tokens from the active mode have been processed. To detect that there are no more tokens remaining or pending to arrive to the input places, special signal tokens are used to inform of a mode change (*mode-change signal*). The change of mode in a transition automatically sends mode-change signals to all its output places. Thus, signals are propagated automatically across the network, flushing tokens produced on the previous mode, before changing further transition to the new mode. When a transition change its mode, input and output places are reconfigured according to the new mode specification. An example of a network with modes can be seen in Fig. 1. The network has a transition (A) with two modes. On each mode, the transition will send tokens to a different destination B or C.

Finally, the modes can be used to enable data locality, defining task-to-task affinities. Tasks implemented as functions of different modes in the same transition are mutually exclusive and are executed by the same thread so they can share data structures. For example, data affinity is used in the Smith-Waterman algorithm, which is one of the benchmarks discussed in the experimental section. This benchmark performs a two-phase wavefront algorithm (see Fig. 2). In the first phase, tasks calculate the elements of a matrix starting from the top left element. The second phase is a backtracking search that starts from the bottom-right element, and each task works on a part of the the matrix obtained in the first phase. As it is shown in Fig. 2, it is possible to create a network to model this kind of problems without using the modes. However, using the modes, we can fold that network by adding two different activities in the transitions, one for each phase of the algorithm. Thus, each transition can perform the two required stages by sharing its assigned portion of the matrix, avoiding communications of the matrix portions that would imply sending big tokens through places.
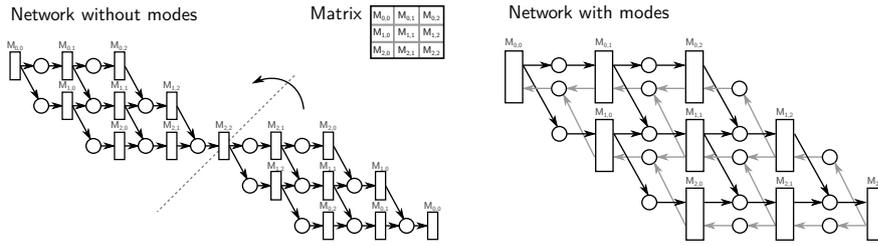
Fig. 2: Smith-Waterman network structure with and without modes.

3 Programming with HitFlow

We have developed a prototype of a framework to implement parallel programs in accordance with the proposed model. The current prototype relies on POSIX Threads Programming (Pthreads) and the standard Message Passing Interface (MPI) to support both shared- and distributed-memory architectures. We decided to use Pthreads in the prototype because the C++ Standard Library threads where not fully supported at the time the development began. Porting the current code to use native C++ threads would be straightforward.

This section explains the key features of the programming framework. It contains a summary of the HitFlow API, a description of how to build a program network, and details about the mode semantics. The main HitFlow classes are shown in the UML diagram in Fig. 3. A table with the API methods can be found in [9].

3.1 Building transitions

To use this framework, the user has to create a class which extends the provided `Transition` class with the sequential activities of the program (See example in Fig. 4). The `init` and `end` methods can be extended to execute starting and ending actions before and after the execution of the program. The user classes should introduce one or more new methods with arbitrary names to encapsulate the code for particular mode activities. The association between modes and activity methods is established when building the network (see section 3.2).

The activity method is automatically called when there are tokens to be processed in the input places declared for its mode. If there are no input places for a particular mode, it will be called just once. The user-defined activity methods can use the `Transition::get` or `Transition::put` methods to retrieve tokens from, or append tokens to the current mode places. The `get` method retrieves one token for each of the active input places. On each activity method invocation, HitFlow ensures that the `get` method can be called once. Additional calls to `get` will block until there is at least one token in each input

```
            ┌──────────────┐                    ┌──────────────┐
            │     Net      │                    │    Mode      │
            ├──────────────┤                    ├──────────────┤
            │ add(Transition* t) │              │ string name  │
            │ run()        │                    └──────────────┘
            └──────────────┘
```
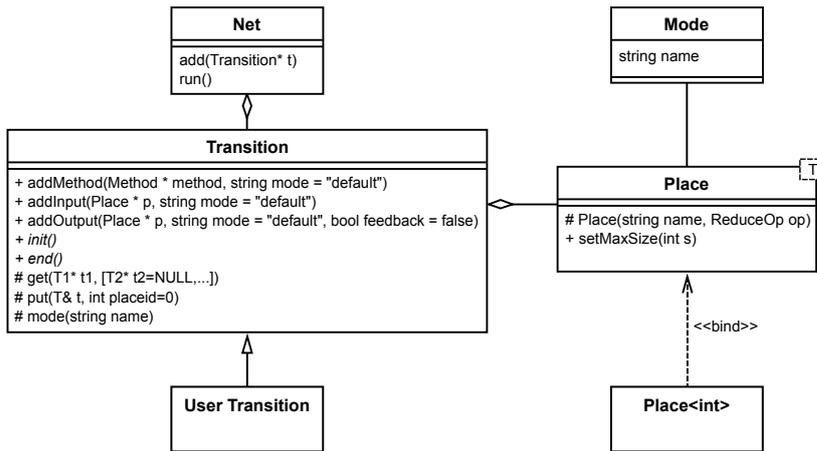
Fig. 3: UML diagram of the framework.

place. The `put` method adds a token to a specific output place. The output place can be selected by its identifier using the second argument of the `put` method. It can be omitted if there is only one active output place in the mode.

A mode automatically finishes when: (a) The producer transitions have sent a mode-end signal indicating that they have finished the activity in that mode; and (b) All the tokens that were generated in the previous mode have been consumed from the input places. At this moment, the transition sends end-mode signal tokens to the active output places and automatically evolves to the next-programmed mode. The next-programmed mode can be changed by calling the method `Transition::mode` at any time. If it is not changed by the user, the default next mode is *END*, that is used to finish the computation.

The example in Fig. 4 extends the `Transition` class by declaring a user activity method. The method retrieves a token from one place, processes it, and sends the result to an output place.

The tokens are C++ variables of any type, handled using template methods. The marshaling and unmarshaling is done internally with MPI functions. The basic types (`char`, `int`, `float`, ...) are enabled by default. User-defined types require the programmer to declare a data type invoking the HitFlow function (`hitTypeCreate`) that internally generates and registers the proper MPI derived type.

3.2 Building the network

Once the transition classes are defined, the programmer builds the network in the `main` function of the C++ program. This implies creating transition and place objects, associating the activity methods, input, and output places to modes on the transitions, and finally adding the transitions to a Net object.

```
1  class MyTransition: public Transition {
2  public:
3    void execute(){      // User activity method
4      double intask;
5      get(&intask);    // Retrieve a token from the place
6      double outtask = process(intask);
7      put(&outtask);  // Put the token into the output
8    }
9  };
```

Fig. 4: HitFlow example of the creation of a Transition extending the basic Transition class.

```
1   Place<double> placeA, placeB;  // Declare the places
2   placeA.setMaxSize(10);        // Set the place size
3
4   MyTransition transition;
5
6   // Add the method and places to modeA
7   transition.addMethod(&MyTransition::execute,"modeA");
8   transition.addInput(&placeA,"modeA");
9   transition.addOutput(&placeB,"modeA");
10  ...
11
12  Net net;                // Declare the net
13  net.add(&transition); // Add the transition
14  net.run();              // Run the net
```

Fig. 5: HitFlow example of the network creation.

Fig. 5 shows a simple code to build a network using the previously shown `MyTransition` transition.

The first step is to create the places that will be used in the application (line 1). The `Place` class is a template class used to build the internal communication channels. The size of the place defines the granularity of the internal communications: It is an optimization parameter that represents the number of packed tokens that will be transferred together. The user can set it in accordance with the token generation ratio of the transition.

The next step is to set the activity method and the inputs and outputs for each mode. The `addInput`, `addOutput`, and `addMethod` methods, have an optional parameter to specify the mode. When this parameter is not specified, a default END mode is implicitly selected. Lines starting at 7 set the activity method, an input place, and an output place for the default mode. Multiple calls to the `addInput` or `addOutput` for the same transition mode, allow MPMC constructions to be built.

Finally, all the transitions are added to a `Net` class that controls the mapping and the execution (lines 12 and 13). Line 14 invokes the `Net::run` method that starts the computation.

3.3 Mapping

Using HitFlow, the programmer can provide a mapping policy to assign transitions to the available MPI processes. If it is not provided, there is a default fallback policy implementing a simple round-robin algorithm. MPI processes with more than one mapped transition automatically spawn additional threads to concurrently execute all the transitions. HitFlow implementation solves the potential concurrency problems introduced by synchronization and communication when mapping transitions to the same process (see Sect. 4.3). In the current prototype, the mapping policies should provide an array associating indexes of transitions to MPI process identifiers.

4 Implementation details

This section discusses some of the implementation challenges associated with the model, and how they have been solved in the current framework implementation.

4.1 Targeting both shared and distributed systems

One of the main goals of the framework is to support both shared- and distributed-memory systems with a single programming level of abstraction. The user-defined transition objects that contain the logic of the problem are mapped into the available MPI processes. Since there may not be enough processes for all of the transitions, threads are spawned inside the processes if needed. Only one thread is spawned for each transition, to execute the user function and its port activities asynchronously to other transitions. The main thread on each MPI process initializes the runtime data structures, launch the threads for the transitions mapped to it, and wait for them to finish. Coordination between the spawned treads, to use the shared structures of the runtime system, is done using mutexes and condition variables.

4.2 Distributed places

The HitFlow places are not physically located in a single process. Instead, they are distributed token containers. A place is implemented as multiple queues of tokens located in the transitions that use that place as input. When load balance requires it, the tokens are transmitted and rearranged between the queues on the transitions[1]. This solution builds a distributed MPMC queue mechanism that exploits data locality, and is more scalable than a centralized

---

[1] In the current implementation, MPI communications are always used to move tokens from queue to queue, even when the queue objects are mapped into the same MPI process. Although this simplifies the implementation, and MPI communications are highly optimized in shared-memory, this decision clearly opens possibilities for further optimization.
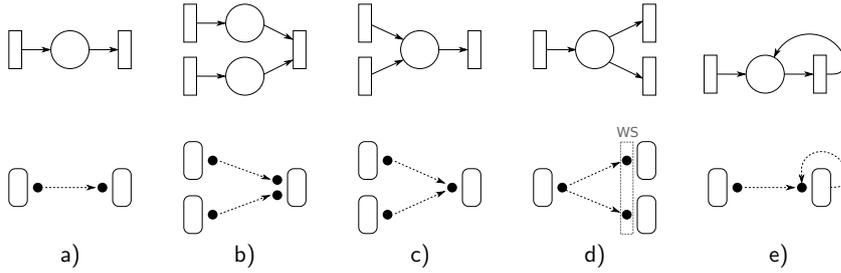
Fig. 6: Translation from the model design to its implementation. WS: work-stealing.

scheme where a single process manages all the tokens of a place. However, this is a solution that introduces coordination challenges that will be discussed below.

Internally, the distributed places are implemented using *ports* that manage the movement of the tokens from the source to one of the destination transitions. Input and output ports are linked using *channels*. Fig. 6 shows how the arcs of the model are implemented using ports. There are five possible situations:

(a) When a place connects two transitions, a channel will be constructed to send the tokens from the source to the destination.
(b) When there are two or more input places in a transition, the transition will have several input ports, each of them connected to the corresponding source.
(c) When two or more transitions send tokens to a common place, the destination will have a single port that will receive tokens, regardless of the actual source.
(d) If a place has several output transitions, any of them can consume the tokens. To allow this behavior, when a place is shared by several destinations, the source will send tokens in a round-robin fashion to each output port. This can lead to load unbalance if the time to consume tokens in the destinations is not compensated. To solve this, a work-stealing mechanism is used to redistribute tokens between the destination transitions.
(e) When a transition uses the same place as input and output, the token will flow directly to the input port for efficiency reasons.

4.3 Ports, buffers, and communications

This section describes the internals of the port objects and explains the details about the communications and buffering. Fig. 7 shows an example of a two-transition network. There is a producer that generates tokens which are sent to a consumer using the place A. The consumer presumably performs a filter
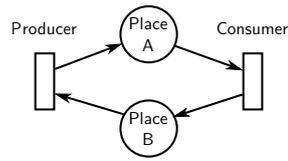
Fig. 7: Small example network with two transitions.

operation on the tokens and sends some of them back to the producer using the place B. Fig. 8 describes the internal structures of the previous example.

The internal communications are handled by `Port` objects. The transitions have a port for every input or output place. The ports have a buffer where the tokens are stored. The size of the buffer is determined by the maximum number of tokens that can be stored at the same time in the place that it represents, as defined by the user with the `Place::setMaxSize` method. The size of the buffer also has an extra space for the message headers and other information that must be sent along with the tokens. When tokens are sent to a place, they are first stored in the output port buffer. The HitFlow runtime library decides when to perform the actual communication. By default, it will try to maximize the port buffer usage, packing as many tokens as possible to minimize the number of MPI messages to be sent, without delaying communications.

In addition to the input port buffers, the transitions have queues to store the tokens received. There is a queue for each input place. When an incoming MPI message is received, the input port buffer associated to the channel is used to retrieve the tokens and store them in the corresponding queue where they can be accessed by the transition `get` method. Unlike the buffers, which have a limited memory space assigned, the queues grow dynamically and are only limited by the host memory.

In Fig. 8, the producer transition (2) and the consumer transition (2') are executed in two different processes (1 and 1', respectively). Since both transitions have only one input place, they have only one input queue (3 and 3'). The size of the place A is 5, thus the output port of the producer (5) and the input port of the consumer (4') have a buffer for 5 elements. In contrast, the size of B is 3, so its port buffers (4 and 5') have size 3. The figure also represents the MPI communication buffers for the two processes (6 and 6'). If there are several transitions mapped to the process, all the elements except the MPI buffers (6 and 6') are replicated for each transition, and they are managed by its own thread.

The HitFlow runtime ensures a deadlock-free behavior due to port buffer exhaustion, even in unbalanced networks with cycles. Consider for example the network depicted in Fig. 8. Assuming that the producer and consumer send tokens with a very unbalanced ratio, causing the port buffer of the two transitions to become exhausted, it will not cause a deadlock. The runtime will keep receiving messages and storing them in the local and unlimited transition
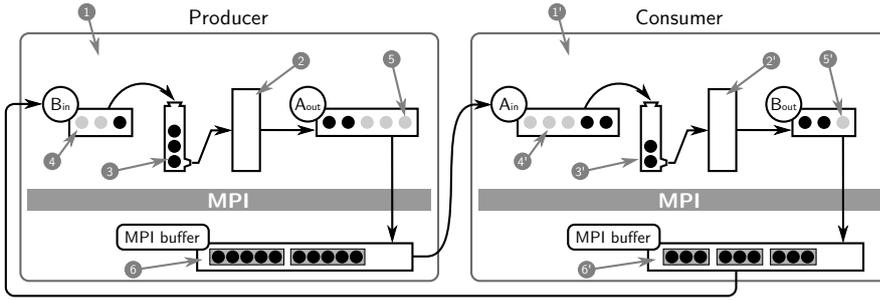
Fig. 8: Description of the different buffers, data structures and control elements involved in the communications. Legend: (1) MPI process. (2) User transition object. (3) Internal token queue for the transition. (4) Input buffer port. (5) Output buffer port. (6) MPI communication buffer.

queue. Thus, the only limitation will occur when one of the processes depletes the host memory.

However, due to a limitation of the MPI-3 standard that only allows one MPI buffer per process, it is possible to produce a deadlock when several transitions are mapped to the same MPI process using threads. If two transitions are mapped to the same process, they share the same MPI buffer. Thus, the messages of one transition could consume all the buffer memory, preventing the other transition from performing its communications. This opens the possibility of producing a deadlock on the progression of the whole network. This problem can be solved using new features that are proposed for MPI-4, such as Allocate Receive communications [10], that allocate memory internally for incoming messages to eliminate buffering overhead when receiving unknown-size messages, and Communication Endpoints [11] that allow the threads inside a process to communicate as if they were at separate ranks.

4.4 Work-stealing

To solve load unbalances when a place has several output transitions, Hit-Flow uses a work-stealing mechanism to redistribute tokens between the consumers. The token queues that were presented in Sect 4.3 are in fact double-ended queues. The user function retrieves the tokens from the bottom with the `Transition::get` method, while the work-stealing mechanism takes or adds tokens using the top end. When a transition consumes all the tokens in one input queue, the HitFlow runtime will try to obtain more tokens. First it will select a victim between the other transitions in the work-stealing group, and then it will send a request message. Depending on the number of available tokens in the victim, it can send some of its tokens back or send a message denying the request. In order to determine when the tokens have been consumed in all the distributed queues of a single place, and the work-stealing

should stop, a distributed voting-tree scheme is performed. It also implements a mechanism to distinguish tokens of different modes, and manage the signals indicating both mode changes and computation end.

5 Case Studies: HitFlow Evaluation

In this section, four benchmarks representing four case studies are discussed to show the expressiveness of the model for different kinds of applications, and to check the performance of the framework.

5.1 Benchmarks

The first benchmark calculates the Mandelbrot set, an embarrassingly parallel programming application that helps us to test the basic functionalities of our proposal, detect potential overheads, and also to compare our implementation with other solutions.

   The next two benchmarks are two very different implementations of a real application, the Smith-Waterman algorithm, that performs local alignments of protein sequences. The first one is swps3 [12], a highly optimized implementation that extensively uses vector instructions whenever possible. It is a simple task-farm application. The other one is a parallelization based on the implementation developed by Clote [13], it represents a complex combination of wavefront and reduction operations. Finally, the last benchmark solves the Poisson equation in a discretized 2D space using an iterative Jacobi solver. This kind of benchmark is a typical kernel computation in many problems usually associated with data parallelism. It represents a static parallel structure, not based on dataflow parallelism. The last two benchmarks are implemented efficiently in MPI using a coarse-grain data partition, with fixed data affinities across computation stages, and with dependence loops. HitFlow is specifically designed to efficiently implement this kind of problems with a dataflow approach. We discuss below the problems encountered to implement them with other chosen tools.

5.2 Performance study

Experimental work has been conducted to show that the implementation of HitFlow achieves a good performance compared with manually optimized implementations directly programmed using a message-passing paradigm, and with other dataflow parallel programming frameworks. We use two different experimental platforms with different architectures: A multicore shared-memory machine and a heterogeneous distributed cluster of shared memory multicores. The shared-memory system, Heracles, has 4 AMD Opteron 6376 processors with 16 cores each at 2.3 GHz, and 256 GB of RAM. The distributed system is composed of 6 distributed nodes: a Intel Xeon (24 cores, 1.9 GHz), another

Xeon (12 cores, 2.1 GHz), a Intel I7 (8 cores, 3.2 GHz), and three Quad Core Intel processors at 2.4 GHz connected with Gigabit Ethernet network technology. All the nodes in the two platforms use CentOS Linux release 7 and the programs have been compiled using GCC version 4.8.3 with -O3 optimization flag.

5.2.1 Mandelbrot set

For the Mandelbrot benchmark we compare the HitFlow version against a manually developed MPI version, two versions using FastFlow [3] (one for shared-memory and another one for distributed-memory), a version using Intel CnC [4], and another one that uses OpenMP 3.0 tasks in the shared-memory system. All the implementations use a farm structure that processes the grid by rows. The HitFlow version uses a network with a producer transition and several worker transitions connected by a single place. This is a very simple benchmark used to test both the HitFlow channel implementation, and the work-stealing mechanism. The FastFlow pure shared-memory version is the implementation included in the distribution examples. We have developed the distributed version using the two-tier model of the extended FastFlow library that supports both shared and distributed memory using different classes [14]. The CnC version is the one provided in the distribution examples.

Fig. 9 shows the results of the Mandelbrot implementations. The programs calculate the set in a grid of $2^{14} \times 2^{13}$ elements. They use up to 1,000 iterations to determine if each element belongs to the set, leading to many low-cost tasks to be processed if fine grain parallelism is used. The granularity chosen is 100×100 elements per task.

The FastFlow, CnC, and OpenMP versions obtain the same performance in the shared-memory architecture. HitFlow and the manual version have an overhead due to the use of the MPI communications instead of direct use of the shared memory mechanisms. In the distributed architecture, all versions show the same scalability except FastFlow, whose two-tier approach cannot take advantage of the heterogeneous cluster because it uses a static task distribution. Previous experiments using a homogeneous cluster showed that FastFlow achieved the same performance as HitFlow [15]. This shows that HitFlow channel and work-stealing implementation have a great scalability in distributed environments, while there is still room for improvement in shared memory machines. Specific shared-memory communication mechanisms should be used internally between transitions mapped to the same MPI process. This can be done without modifying the model features.

5.2.2 Smith-Waterman: Swps3

We use as reference the original version of swps3 [12], which is implemented using `pipe` and `fork` system calls to create several processes in the same machine. We compare it with FastFlow, CnC, and HitFlow versions. The structure of this benchmark is a farm with an emitter. For a fair comparison, we have
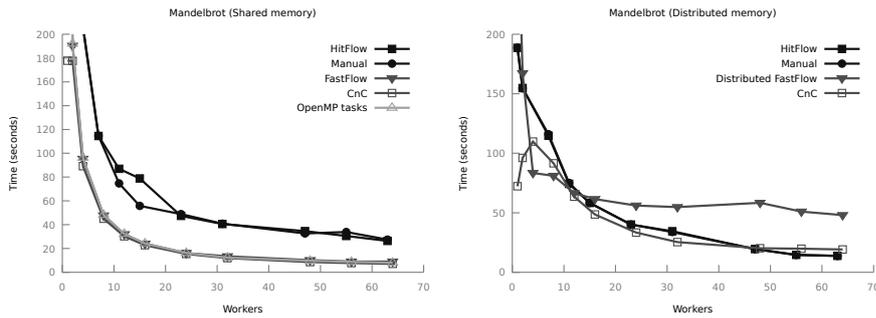
Fig. 9: Mandelbrot set benchmark results.

developed the FastFlow, CnC, and HitFlow versions starting with the sequential code of the original swps3 benchmark to implement the tasks functions. We have not used the original example included in FastFlow [16], since it uses some memory allocation optimizations and it does not work for the big sequences chosen as input for our experiments, which are needed to generate enough workload for our target systems. All the versions match a single protein sequence to all the proteins from a database of sequences. We have used the UniProt Knowledgebase (UniProtKB) release 2014_04, a protein information database maintained by the Universal Protein Resource (UniProt) [17]. This database consists of 544,996 sequences which minimum length is 2, its maximum is 35,213, and its average is 355. Each sequence in the database is a task that will be fed to a farm worker, so they can be matched concurrently.

Fig. 10 shows the experimental results for a representative case, the sequence named Q8WXI7, which has 22,152 proteins. Experiments with other sequences showed similar behaviors. For the shared-memory machine, all versions except CnC show a similar performance. Using CnC leads to a very simple implementation as there are no dependencies among the different calculations. Results show reasonable scalability but very poor performance. This behavior can also be noticed in other example applications provided with the CnC distribution, for example the Jacobi benchmark in Sect. 5.2.4. Like the previous benchmark, FastFlow implementation shows worse scalability in the cluster due to the heterogeneous architecture. We can conclude that HitFlow can be used for this kind of real applications with minimum performance degradation thanks to the proposed implementation.

5.2.3 Smith-Waterman: Clote's algorithm

The third benchmark, CloteSW, is a different implementation of the Smith-Waterman protein alignment that aims to compare two big sequences [13]. For this benchmark, we compare in the shared architecture two sequences of 100,000 elements. They are bigger than any of the sequences used in the previous experiment. For this case, the Smith-Waterman algorithm requires
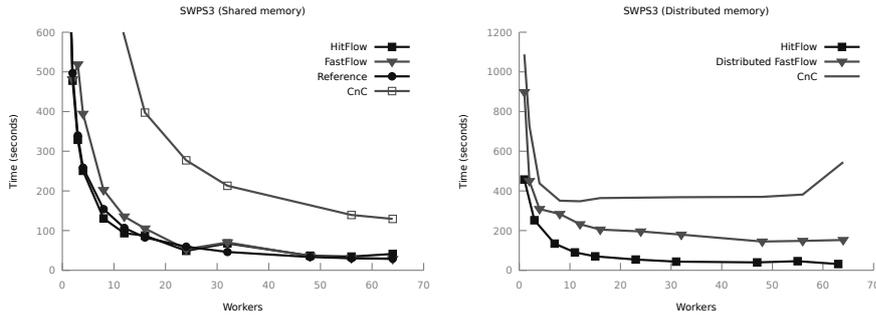
Fig. 10: Swps3 benchmark results using the protein sequence Q8WXI7 as in-putset.

to compute the values of a matrix with 100,000 x 100,000. Due to memory limitations in some of the distributed nodes, we use sequences of 30,000 elements in the cluster. The computation is broken down into pieces, following a distributed wavefront structure. The benchmark has several phases: First, it populates the alignment matrix following the wavefront structure. Then, it performs a reduce operation to determine the maximum match sequence. Finally, it uses a backtracking method to compose the sequence traversing the wavefront structure in the reversed order. The backtracking stage can be implemented as a different mode in the same transitions, creating data affinities that avoid extra data communications or synchronizations (recall Fig. 2).

We have developed and executed versions for shared memory using Fast-Flow, CnC, HitFlow. and C++ with MPI (Manual). The FastFlow version for shared memory uses the FastFlow's *ff_mdf* dataflow skeleton which implements the *macro dataflow* pattern, responsible for scheduling, and that allows the declaration of data dependencies.

The same C++/MPI and HitFlow programs can be used in distributed memory. However we have not been able to obtain correct programs with the other tools. The FastFlow distributed version was not possible to be implemented due to the early stage of development of the distributed support. Some dataflow constructions can only be used in shared-memory environments. As it is stated in [14], more work is needed to allow the user to use distributed versions of the different parallel skeletons.

The results are shown in Fig. 11. The CnC version obtains the worst results with a high difference. FastFlow shows the best performance in shared-memory. The use of message passing in the HitFlow and the Manual versions requires more time for the low-level marshalling and movement of data buffers. However, they can be executed in distributed memory directly, obtaining the same good performance and scalability.
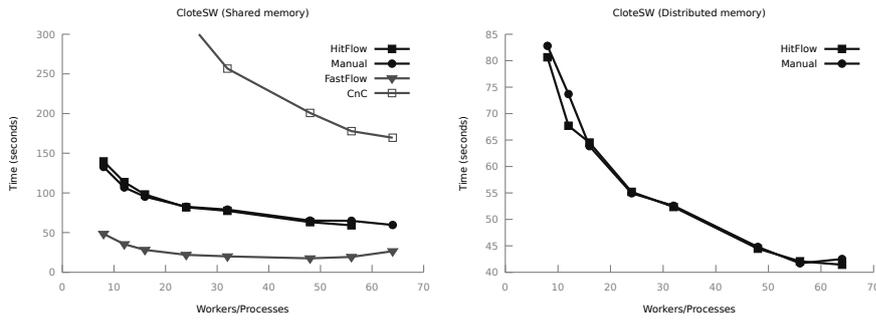
Fig. 11: Clote's Smith-Waterman benchmark results.

5.2.4 2D Jacobi solver

The last benchmark tested is a Jacobi solver that performs 1,000 iterations of a 4-point stencil computation in a 10,000 x 10,000 bidimensional grid. We compare HitFlow against a manually developed version using C and MPI (Manual) in the distributed-memory system, and we also compare against OpenMP, FastFlow, and the version provided by CnC in the shared-memory case. FastFlow benchmark is developed using the stencil data parallel skeleton which, among other parameters, accepts the appropriate function to update each cell. When trying to develop this benchmark using distributed FastFlow, we encountered the same problems as in Clote's version of Smith-Waterman and we were unable to implement it. The Manual version is a classical stencil implementation that divides the grid into portions and uses a neighbor synchronization communication structure to exchange border data on each computation iteration. The HitFlow version uses the same partition policy. Each partition is assigned to a transition that communicates with its neighbors, sending and receiving the data of the borders using places in two different modes.

The results in Fig. 12 show that HitFlow obtains a similar performance to manual C+MPI in distributed memory, and also similar to FastFlow, and OpenMP versions for shared-memory. The implementation provided by CnC does not show a good performance, and it is not prepared for running on distributed memory. As expected, the distributed experiments show a degradation of performance when changing from one single node to several heterogeneous nodes. However they obtain a good scalability when more nodes are added. These results show that the HitFlow model can also be applied to problems that are usually solved using static data parallel models.
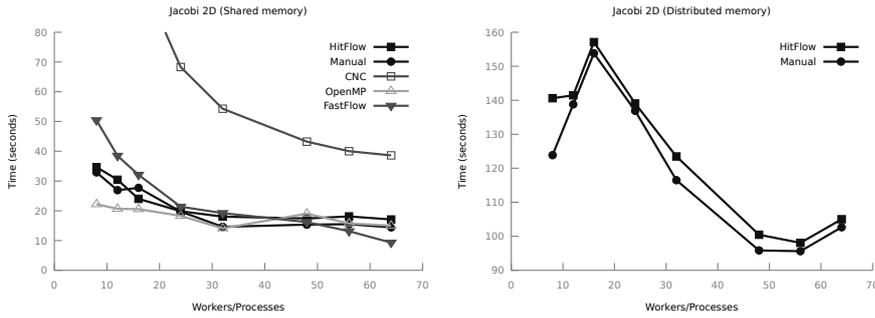
Fig. 12: Jacobi 2D results.

5.3 Code complexity

In this section, we use several code complexity and development-effort metrics to compare HitFlow codes with other proposals. For this comparison, we use three classical development effort metrics: The number of code tokens, McCabe's cyclomatic complexity [18], and Halstead's development effort [19]. The number of tokens detected by the programming-language parser, measures the code volume of C/C++ programs better than the number of code lines. McCabe's cyclomatic complexity is a quantitative measure of the number of linearly independent paths through a program's source code. Finally, the Halstead's development-effort metric is also a quantitative measure based on the number of operators and operands in the source code. They are related to the mental activity needed by a programmer to develop the code, and to the amount of test cases needed to check the program correctness. Low cyclomatic complexity and Halstead's development effort indicate codes which are simpler to develop and debug. These metrics are typically used in the assessment of software design complexity.

We have selected the Mandelbrot benchmark because it is a simple benchmark, and we have more implementations using different programming tools. Table 2 shows the measures obtained for each metric, for the different versions of the benchmark. The metrics clearly show that dataflow abstractions allow the representation of the target program with less development effort than using directly MPI. The shared-memory FastFlow and CnC versions, followed by the HitFlow version are the simplest implementations. However, the regular FastFlow version cannot be used in distributed-memory systems, and CnC needs some tuning to run it in a distributed environment. The version that uses the distributed-memory support of FastFlow leads to the bigger metrics values. This is due to the use of the two-tier model, that forces to implement separately the coordination logic for the distributed processes, and the logic used for shared memory inside the nodes.

A full example of a simple pipeline application implemented in HitFlow, and in FastFlow supporting only shared-memory, can be seen in Fig. 13. It

Metric	Manual MPI	FastFlow	Dist. FF	CNC	HitFlow
Tokens	552	471	935	565	518
McCabe	34	33	57	24	32
Halstead	8.65E+5	4.46E+5	13.1E+5	4.16E+5	4.93E+5

Table 2: Complexity comparison

can be observed that the codes for the nodes activities and coordination are very similar, while FastFlow present neat higher-level abstractions. It reduces the code complexity thanks to the use of skeletons to build the network. This approach could also be used on top of HitFlow. This is furtherly discussed at the end of the Related Work section.

The results indicate that, using the techniques presented in this work, dataflow abstractions in general can efficiently exploit hybrid shared- and distributed-memory using a one-tier programming model, and reducing the development effort comparing with directly using message-passing interfaces.

6 Related work

In this section we first comment the differences between our current proposal and the previous work of our group in the same research line. Then, we discuss conceptual similarities and differences with other dataflow or task-network oriented programming models. We focus the discussion on features that have implications in the programming strategies, the implementation techniques used, and the mapping of the tasks in the context of distributed processes.

HitFlow is a complement of Hitmap, a library for automatic but static hierarchical mapping, with support for dense and sparse data structures [20, 21, 22]. The Hitmap library focuses on data-parallel techniques and does not have a native support for dataflow applications. In a previous work [7], we introduced a first approach to a dataflow model that could be used as a Hitmap extension. The model introduced in this paper generalizes several restrictions of the previous one, introducing a complete generic model to represent any kind of combinations of parallel structures and paradigms. The differences with the previous Hitmap extension can be summarized as: (1) We present a general MPMC system where consumers can consume different task types from different producers. (2) It supports cycles in the network construction. (3) The new model introduces a concept of mode inside the processing units to reconfigure the network, allowing mutually exclusive functions in a transition, and to intuitively define task-to-task affinity with an easier mapping to fixed-scheduled MPI processes.

S-Net [6] is a declarative coordination language. It defines the structure of a program as a set of connected asynchronous components called boxes. S-Net only takes care of the coordination: The operations done inside boxes are defined using conventional languages. Boxes are stateless components with only a single input and a single output stream. From the programmers' perspective,

the implementation of streams on the language level by either shared memory buffers or distributed memory message passing is entirely transparent.

HitFlow has several similarities with FastFlow [3], a structured parallel programming framework targeting shared memory multi-core architectures. FastFlow is structured as a stack of layers that provide different levels of abstraction, providing the parallel programmer with a set of ready-to-use, parametric algorithmic skeletons, modeling the most common parallelism exploitation patterns. HitFlow transition API is similar to FastFlow. Fig. 13 shows a full example of a simple pipeline application to compare both of them. The main differences are that the HitFlow framework is designed to support both shared- and distributed-memory with a single tier model. It includes a transparent mechanism for the correct termination of networks even in the presence of feedback-edges, and mode-driven control to create affinity between transitions in distributed memory environments. The FastFlow group has developed an extension to FastFlow to target distributed nodes, using a two tier model [14]. However, this solution forces the programmer to implement separately the coordination logic for the distributed processes, and the logic used for shared memory inside the nodes. It uses a different mechanism of external channels to communicate the tasks. In this sense, HitFlow makes the program design independent from the mapping between shared-memory and distributed-memory levels.

HitFlow networks are similar to CnC (Concurrent Collections [4]) graphs. CnC is a parallel programming model where the computation is defined by serial functions called computation steps and their semantic ordering constraints. Like HitFlow transitions, CnC steps communicate through message-passing as well as shared memory using shared entities called item collections. One of the differences between HitFlow and CnC is that CnC allows the programmer to give the scheduler hints about the thread affinity. However, CnC steps only execute one activity each one with its own memory space. Thus it is not possible to define task to task affinities in the way HitFlow transitions do, to better map the task networks to MPI processes without incurring in communication cost penalties.

There are some proposals that support task parallelism introducing annotations in the sequential source code. For example, the OpenMP 3.0 task primitives and the dependency extensions introduced in version 4.0 of the standard [23]. The programmer exposes data flow information using pragmas to define the stream input and output task. The runtime ensures the coordination of the different elements. Other dataflow proposals based on annotations are: OpenStream [5], OmpSs [24], and StarPU [25]. All these proposals simplify the development of task parallel programs in shared-memory, and OmpSs and StarPU also support environments with accelerator devices, and even distributed memory. These models rely in load balancing mechanisms that dynamically map tasks with an arbitrary granularity level defined by the programmer. On the other hand, our approach is designed to simplify the expression of task networks with a flexible granularity, and to allow the creation of affinities between tasks and distributed processes. The tasks can reconfig-

ure their activity and/or communication channels to change the computation and communication structure across different stages. The purpose is to better exploit locality and to reduce the communication costs.

Skeleton libraries present an approach that have a higher level of abstraction than our dataflow model, with flexible implementations for different target architectures and hybrid platforms (see e.g. SkePU [26] or Muesli [27]). They typically use a two-tier model, not related to the target platform, but to the programming paradigm. They distinguishing between two different types of skeletons (task- or data-parallel oriented). These types cannot be composed in any form. Data-parallel skeletons can only be the leaves of the composition tree. Data affinities across different stages, which means different hierarchies of skeletons, are not properly defined. Finally, the amount of included skeletons do not support all the applications classes supported by a generic dataflow programming model that can express task or data-parallel computations with the same abstraction, supporting arbitrarily connected transitions and places, with dependences loops. In the context of HitFlow, skeletons could be used as higher-level abstractions to transparently generate common tasks networks, by combining a limited set of structures. FastFlow already exploits this approach as we discussed at the end of the previous section.

7 Conclusions

This paper presents a parallel programming model and framework with a novel combination of features designed to easily map dataflow programs to distributed-memory processes. It allows programs to be described as a network of communicating activities in an abstract form. The system allows the implementation of applications from simple static parallel structures, to complex combinations of dataflow and dynamic parallel programs. The description is decoupled from the mapping techniques or policies, which can be efficiently applied at runtime, automatically adapting static or dynamic structures to different resource combinations. Our current framework transparently targets hybrid shared- and distributed-memory platforms.

We present an evaluation with examples of different classes of dynamic and static applications. Experimental performance results show that the overhead introduced by our abstractions has minimal impact compared with manually developed implementations using MPI. Comparisons with other dataflow programming tools show that HitFlow can better express some classes of programs designed for distributed environments, while its implementation can be improved for shared-memory. Comparisons of development effort metrics indicate that HitFlow codes have a similar development cost than other dataflow abstractions. HitFlow codes present a much lower complexity than manually developed MPI codes, and obtain the same performance and scalability.

This generic framework will allow us to focus our research on the best mapping policies that can transparently target heterogeneous platforms for

```
1   #include <hitflow.h>
2   using namespace hitflow;
3
4
5   class StageA: public Transition {
6      int numtasks;
7   public:
8      StageA(int t): numtasks(t){};
9
10     void create(){
11        long task;
12        for(int i=0; i<numtasks; i++){
13           task = i;
14           put(task);
15        }
16
17     }
18  };
19
20  class StageB: public Transition {
21     long sum;
22  public:
23     void init(){
24        sum = 0;
25
26     }
27     void process(){
28        long task;
29        get(&task);
30        sum += task;
31     }
32     void end(){
33        cout << "Sum " << sum << endl;
34     }
35  };
36
37  int main(int nargs, char * vargs[]){
38
39     HitFlow::init(&nargs, &vargs);
40
41     StageA st_a(10);
42     StageB st_b;
43
44     Place<long> place("long container");
45
46     st_a.addOutput(&place,"createTasks");
47     st_a.addMethod(&StageA::create,"createTasks");
48     st_b.addMethod(&StageB::process,"processTasks");
49     st_b.addInput(&place,"processTasks");
50
51     Net net;
52     net.add(&st_a); net.add(&st_b);
53     net.run();
54
55     return 0;
56  }
```

```
1   #include <ff/pipeline.hpp>
2   using namespace ff;
3
4   class StageA: public ff_node {
5      int numtasks;
6   public:
7      StageA(int t): numtasks(t){};
8
9      long * svc(void * intask){
10
11        for(int i=0; i<numtasks; i++){
12           long * task = new long(i);
13           ff_send_out(task);
14        }
15        return NULL;
16     }
17  };
18
19  class StageB: public ff_node {
20     long sum;
21  public:
22     int svc_init(){
23        sum = 0;
24        return 0;
25     }
26     long * svc(long * intask){
27        sum += *intask;
28        delete intask;
29        return GO_ON;
30     }
31     void svc_end(){
32        cout << "Sum " << sum << endl;
33     }
34  };
35
36  int main() {
37
38     ff_pipeline pipe;
39     pipe.add_stage(new StageA(10));
40     pipe.add_stage(new StageB());
41     if (pipe.run_and_wait_end()<0)
42        return -1;
43     return 0;
44  }
```

Fig. 13: A full pipeline example in both HitFlow (left) and shared-memory only FastFlow (right) frameworks.

specific or generic combinations of parallel paradigms, allowing us to build powerful parallel patterns using a common and generic framework.

Acknowledgements

References

1. W. Gropp, E. Lusk, A. Skjellum, Using MPI : Portable Parallel Programming With the Message-passing Interface, 2nd Edition, MIT Press, 1999.
2. R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon, Parallel programming in OpenMP, 1st Edition, Morgan Kaufmann, 2001.
3. M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Torquati, FastFlow: high-level and efficient streaming on multi-core, in: S. Pllana, F. Xhafa (Eds.), Programming Multi-core and Many-core Computing Systems, 1st Edition, Parallel and Distributed Computing, Wiley, 2017.
4. Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, S. Tasirlar, Concurrent collections, Scientific Programming 18 (3-4) (2010) 203–217. doi:10.1155/2010/521797.
   URL http://dx.doi.org/10.1155/2010/521797
5. A. Pop, A. Cohen, A OpenStream: Expressiveness and Data-Flow Compilation of OpenMP Streaming Programs, ACM Transactions on Architecture and Code Optimization (TACO) 9 (4) (2013) 53:1–53:25.
6. C. Grelck, S.-B. Scholz, A. Shafarenko, A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components, Parallel Processing Letters 18 (2) (2008) 221–237.
7. J. Fresno, A. Gonzalez-Escribano, D. R. Llanos, Runtime Support for Dynamic Skeletons Implementation, in: Proceedings of the 19th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), July 22-25, Las Vegas, NV, USA., 2013, pp. 320–326.
8. K. Jensen, L. M. Kristensen, L. Wells, Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems, International Journal on Software Tools for Technology Transfer 9 (3-4) (2007) 213–254.
9. J. Fresno, A. Gonzalez-Escribano, D. R. Llanos, Additional material for the paper: Dataflow programming model for hybrid distributed and shared memory systems, Tech. Rep. IT-DI-2015-0003, Department of Computer Science, University of Valladolid, Spain (2015).
   URL http://www.infor.uva.es/~jfresno/reports/IT-DI-2015-0003.pdf
10. D. Holmes, Ideas for persistant point to point communication, Tech. rep., MPI Forum Meetings (2014).
   URL http://meetings.mpi-forum.org/2014-11-scbof-p2p.pdf

11. J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, R. Thakur, Enabling MPI inter-operability through flexible communication endpoints, in: 20th European MPI Users's Group Meeting, EuroMPI '13, Madrid, Spain - September 15 - 18, 2013, 2013, pp. 13–18. `doi:10.1145/2488551.2488553`.
    URL `http://doi.acm.org/10.1145/2488551.2488553`
12. A. Szalkowski, C. Ledergerber, P. Krähenbühl, C. Dessimoz, SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2., BMC research notes 1 (107).
13. P. Clote, Biologically significant sequence alignments using Boltzmann probabilities, Tech. rep. (2003).
    URL `http://bioinformatics.bc.edu/~clote/pub/boltzmannParis03.pdf`
14. M. Aldinucci, S. Campa, M. Danelutto, Targeting distributed systems in FastFlow, in: Proceedings of the Euro-Par 2012 Parallel Processing Workshops, Vol. 7640 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 47–56.
15. J. Fresno, A. Gonzalez-Escribano, D. R. Llanos, One Tier Dataflow Programming Model for Hybrid Distributed- and Shared-Memory Systems, in: Proc. of Intl. Workshop on High-level Programming for Heterogeneous and Hierarchical Parallel Systems (HLPGPU), January 19, Prague, Czech Republic., 2016.
16. M. Aldinucci, M. Meneghin, M. Torquati, Efficient Smith-Waterman on Multi-core with FastFlow, in: M. Danelutto, J. Bourgeois, T. Gross (Eds.), Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP), IEEE Computer Society, Pisa, Italy, 2010, pp. 195–199.
17. UniProt Knowledgebase (UniProtKB), `www.uniprot.org/`.
18. T. J. McCabe, A complexity measure, IEEE Transactions on software Engineering SE-2 (4) (1976) 308–320. `doi:10.1109/TSE.1976.233837`.
19. M. H. Halstead, Elements of Software Science (Operating and Programming Systems Series), Elsevier Science Inc., New York, NY, USA, 1977.
20. J. Fresno, A. Gonzalez-Escribano, D. R. Llanos, Extending a hierarchical tiling arrays library to support sparse data partitioning, The Journal of Supercomputing 64 (1) (2013) 59–68.
21. A. Gonzalez-Escribano, Y. Torres, J. Fresno, D. R. Llanos, An Extensible System for Multilevel Automatic Data Partition and Mapping, IEEE Transactions on Parallel and Distributed Systems 25 (5) (2014) 1145–1154.
22. A. Moreton-Fernandez, A. Gonzalez-Escribano, D. Llanos, Exploiting distributed and shared memory hierarchies with hitmap, in: High Performance Computing Simulation (HPCS), 2014 International Conference on, 2014, pp. 278–286. `doi:10.1109/HPCSim.2014.6903696`.
23. OpenMP Architecture Review Board, OpenMP application program interface version 4.5 (November 2015).
    URL `http://www.openmp.org/mp-documents/openmp-4.5.pdf`
24. Barcelona Supercomputing Center, OmpSs specification 4.5 (December 2015).
    URL `http://pm.bsc.es/ompss-docs/specs`
25. C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, Starpu: A unified platform for task scheduling on heterogeneous multicore architectures, Concurr. Comput. : Pract. Exper. 23 (2) (2011) 187–198. `doi:10.1002/cpe.1631`.
    URL `http://dx.doi.org/10.1002/cpe.1631`
26. C. W. K. Johan Enmyren, 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 (HLPP 2010), ACM, 2010, pp. 5–14.
27. H. K. Philipp Ciechanowicz, Enhancing muesli's data parallel skeletons for multi-core computer architectures, in: Proceedings of 12th IEEE International Conference on High Performance Computing and Communications (HPCC 2010), IEEE, 2010.