One Tier Dataflow Programming Model for Hybrid Distributed- and Shared-Memory Systems

Javier Fresno Arturo Gonzalez-Escribano Diego R. Llanos
Universidad de Valladolid
{jfresno,arturo,diego}@infor.uva.es

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 of exploiting the parallelism by concurrently executing the different stages when their dependencies have been met.

In this paper we introduce a new parallel programming model and framework based on the dataflow paradigm. Its features are: It is a unique one-tier model that supports hybrid shared- and distributed-memory systems; it can express activities arbitrarily linked, including cycles; it uses a distributed work-stealing mechanism to allow Multiple-Producer/Multiple-Consumer configurations; and it has a run-time mechanism for the reconfiguration of the dependences network which also allows to create task-to-task affinities.

We present an evaluation using examples of different classes of applications. Experimental results show that programs generated using this framework deliver good performance, and that the new abstractions introduce minimal overheads.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Data-flow languages

Keywords Dataflow programming, Distributed systems, Dynamic computation, Parallel programming models, Streaming computation

1. Introduction

The most common programming tools for parallel machines are based on message passing libraries, such as MPI [14], or shared memory APIs like OpenMP [6]. 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 [4], CnC [5], OpenStream [17], or S-Net [13]) reduce the complexity of creating a parallel program because the programmer only has to define the sequential stages and its dependencies. It is the runtime resposability to control the sequential stages execution and perform the data synchronizations to exploit parallelism.

However, these models lack either: A unique representation for shared- and distributed-memory architectures; dependences structures involving feedback loops, free of concurrency problems; a generic system to represent MPMC (*Multiple-Producer/Multiple-Consumer*) configurations; mechanisms to reconfigure dependences at run-time; or they do not have ways of intuitively express task-to-task affinities which would allow a better exploitation of data-locality across state-driven activities.

In this paper, we propose HitFlow, a new dataflow parallel programming model that extends a previous proposal [10]. It uses a single representation for both shared- and distributed-memory models. It introduces a generic form of describing a program as a reconfigurable network of activities and typed data containers arbitrarily interconnected. It presents an abstraction for an MPMC channel system that includes a work-stealing, load-balancing mechanism. Our solution also allows task-to-task affinity to be set to exploit data locality.

We present an evaluation of our proposal using examples of three different application classes. We describe how they are represented in our model, showing how to express different types of parallel paradigms, and static and dynamic synchronization structures. Moreover, experimental work has been carried out to prove that the programs generated using our framework achieve good performance in comparison with manual implementations using message passing, or compared with FastFlow [4], another state-of-the-art tool for dataflow programming. These experiments show that the overheads introduced by the new abstractions do not have a significant impact.

The rest of the paper is organized as follows. Section 2 describes our proposed parallel programming model. A discussion about its usage is given in Section 3 while Section 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 Section 7.

2. 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 [16]. 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 and output role of places for each transition. A transition takes one token from each of its input places and performs some activity

HLPGPU 2016, January 19, 2016, Prague, Czech Republic.
Copyright © 2016 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

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, $P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of places and $T = \{t_1, t_2, \ldots, t_m\}$ is a finite set of transitions composed of modes: $t_i = \{m_1, m_2, \ldots, m_o\}$. Each mode m_i is a tuple $\langle f, I, O, \text{next}\rangle$ where $I \subseteq P$ are the input channels, $O \subseteq P$ are the output channels, f is the sequential function, and next $\in \{m_1, m_2, \ldots, m_o\} \cup$ END is the selected next mode.

Modes are used to define mutually exclusive activities inside the transitions that dynamically reconfigure the network. A mode enables a subset of connections to input places or output places. For each mode, the user defines a function to process inputs, the associated places and the default next mode that will be executed when the current one finishes. A transition with several modes 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 each 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. Task 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. In the first phase, it calculates 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 of the resulting matrix using the data obtained from the previous phase. As is shown in Fig 2, it is possible to create a network to model this kind of problem without using the modes. However, using the modes, we can fold that network adding two different activities in the transitions, one for each phase of the algorithm. Thus, each transition can perform the two required stages sharing its assigned portion of the matrix, avoiding communications of the matrix portions, that would imply sending big tokens through places.

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. 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 [11].

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).

```
class MyTransition: public Transition {
public:
  void execute(){      // User activity method
    double intask;
    get(&intask);   // Retrive a token from the place
    double outtask = process(intask)
    put(&outtask);  // Put the token into the output
  }
};
```

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

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 its 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 places. The `get` method retrieves one token for each of the active input places of the current mode. On each activity method invocation, HitFlow ensures that the `get` method can be called once. Additional calls to `get` will block until there are again at least one token in each input place. If the framework detects that the producers that feed the input places have ended their modes, meaning no more tokens will be produced (end-mode tokens in the input places) and the input places are already empty of tokens of the active mode, the call to `get` will throw an exception. 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 the producer transitions have sent a mode-end signal indicating that they have finished the activity in that mode, and all the tokens in the places, that were generated in the previous mode, have been processed. At this moment, the transition sends end-mode signals tokens to the active output places and automatically evolves to the next-programmed mode. The next-programmed mode can be changed by calling the `Transition::mode` method 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 with a 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. Fig. 5 shows a simple code

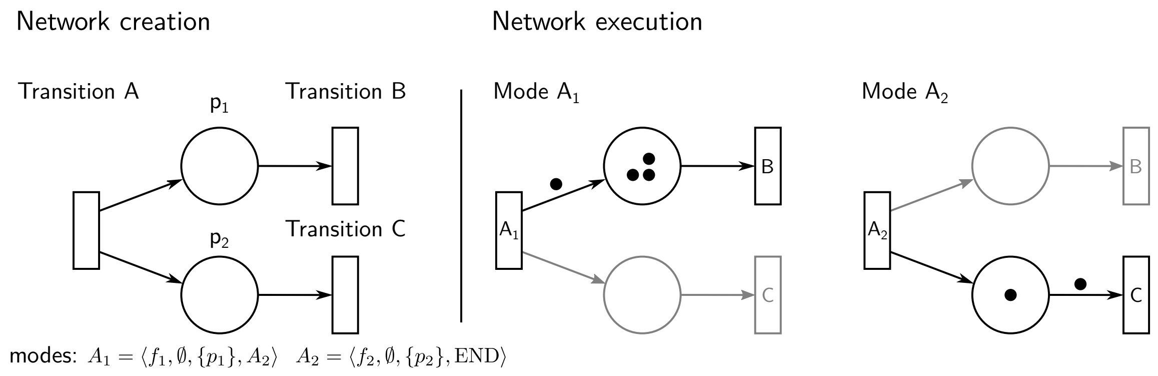
Network creation
Network execution
Transition A
p_1
Transition B
p_2
Transition C
Mode A_1
Mode A_2
B
A_1
C
A_2
modes: $A_1 = \langle f_1, \emptyset, \{p_1\}, A_2 \rangle$ $A_2 = \langle f_2, \emptyset, \{p_2\}, \mathrm{END} \rangle$

Figure 1: Example of a network using modes. Transition A has two modes (A_1 and A_2), each mode enables an output channel connecting A with B or A with C.

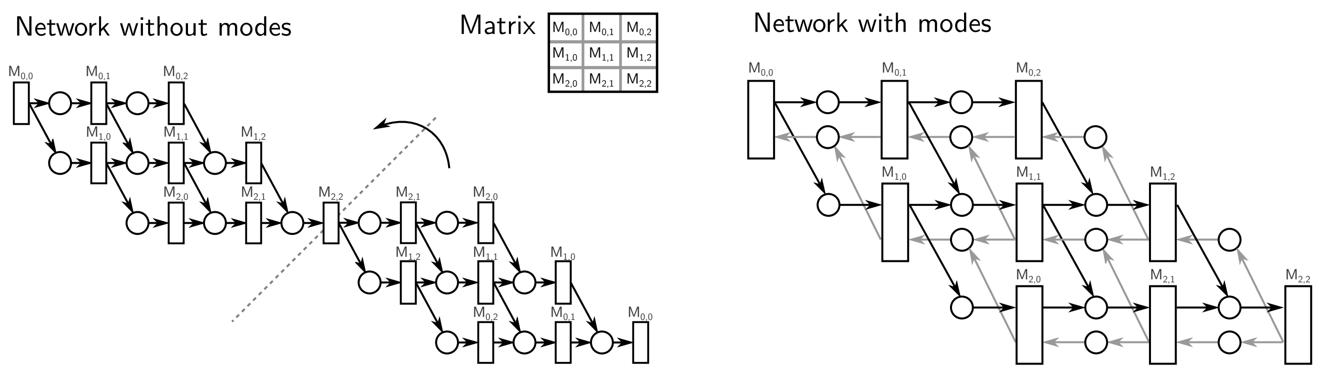
Network without modes
Matrix
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$
Network with modes

Figure 2: SmithWaterman network structure with and without modes.

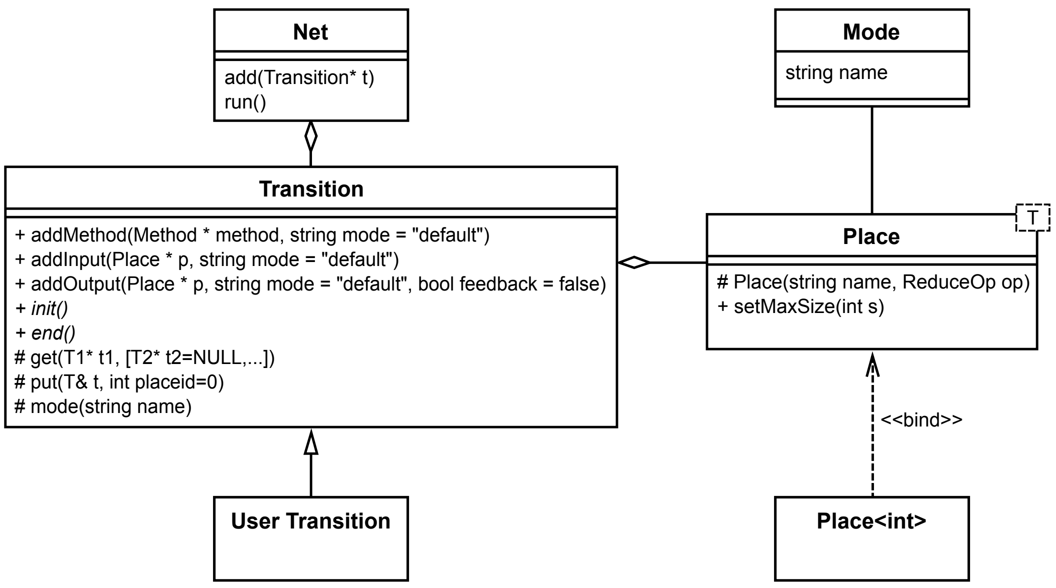
Net
add(Transition* t)
run()
Mode
string name
Transition
+ addMethod(Method * method, string mode = "default")
+ addInput(Place * p, string mode = "default")
+ addOutput(Place * p, string mode = "default", bool feedback = false)
+ init()
+ end()
# get(T1* t1, [T2* t2=NULL,...])
# put(T& t, int placeid=0)
# mode(string name)
Place
T
# Place(string name, ReduceOp op)
+ setMaxSize(int s)
<<bind>>
User Transition
Place<int>

Figure 3: UML diagram of the framework.

```
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
```

Figure 5: HitFlow example of the network creation.

with 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 prototype framework.

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.

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 needed, the tokens are transmitted and rearranged between the queues an input place on the transitions. This solution builds a distributed MPMC queue mechanism that exploits data locality, and is more scalable than a centralized scheme where a single process manages all the tokens of a place. However, this is a solution that introduces coordination challenges.

Internally, the distributed places are implemented using *ports* that move the tokens from the source to one of the destination transitions. Input and output ports are linked using *channels*. Figure 6 shows how the arcs of the model are implemented using ports. There are five 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, it will have several input token 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 shows the internal port objects and explains the details about the communications and buffering. Figure 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 operation on the tokens and sends some of them back to the producer using the place *B*. Figure 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 real 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 are 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. Finally, the user method is

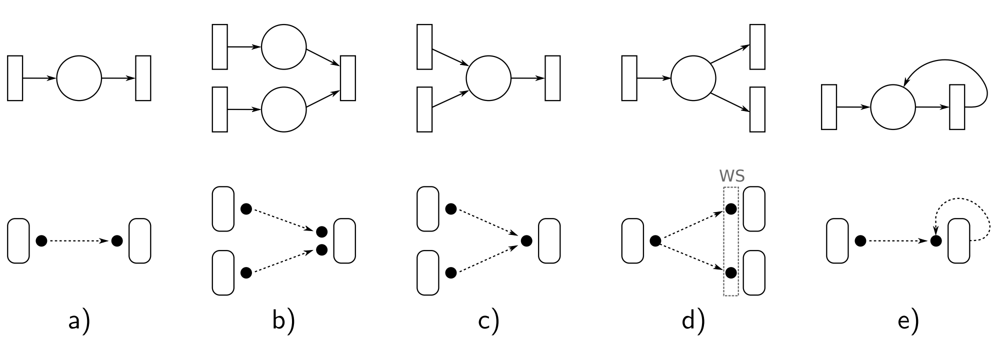

Figure 6: Translation from the model design to its implementation. WS: work-stealing.

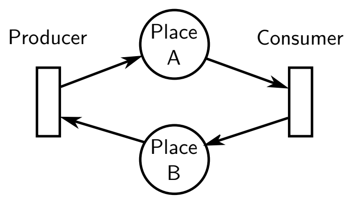

Figure 7: Small example network with two transitions.

automatically called to process the available tokens when there is at least one token in each input queue.

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 token 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 that same size. The figure also represents the MPI communication buffers for the two processes (6 and 6').

With the HitFlow runtime, it is not possible to produce a deadlock 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 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 in 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 the new features that are being studied to be included in MPI-4. Such as Allocate Receive communications [15] that allocate memory internally for incoming messages to eliminate buffering overhead when receiving unknown-size messages, and Communication Endpoints [8] that allow the threads inside a process to communicate as if they are separate ranks.

4.4 Work-stealing

To solve load unbalances when a place has several output transitions, HitFlow 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 `Transtion::get` method while the work-stealing mechanism takes or adds tokens using the top end. When a transition processes all the tokens in its queue, the HitFlow runtime will try to obtain more tokens. First it will select a victim from within 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 there are no more tokens available in any of the transitions, a distributed voting-tree scheme is performed.

5. Case Studies: HitFlow Evaluation

In this section, three case studies are discussed to test whether the model is suitable to represent 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, to detect potential overheads, and also allows us to compare with other solutions.

The next two benchmarks are two implementations of a real application. They present very different implementations of the Smith-Waterman algorithm, an algorithm to perform local alignments of protein sequences. One of them is swps3 [18], a highly optimized implementation that extensively uses vector instructions. The other one is a parallelization based on the implementation developed by Clote [7]. The first one is a simple task-farm application, while the second represents a complex combination of wavefront and reduction operations.

5.2 Performance

Experimental work has been conducted to show that the implementation of HitFlow achieves a good performance compared with other frameworks and with manually optimized implementations. We use two different experimental platforms with different architectures: A multicore shared-memory machine and a distributed cluster of shared memory multicores. The shared-memory system, Atlas, has 4 AMD Opteron 6376 processors with 16 cores each at 2.3GHz, and 256GB of RAM. The distributed system, CETA-Ciemat, is composed of several MPI nodes with two Quad Core Intel Xeon processors (4 cores, 2.26GHz) each, thus each node con-

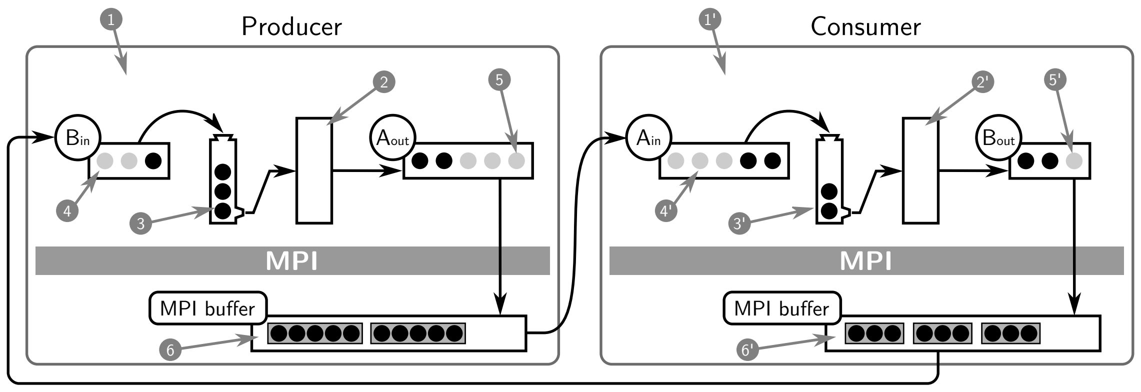

Figure 8: Description of the different buffers, data structures and control elements involved in the communications. Legend: (1) MPI process. (2) User transition object. (3) Internal token queue for the transition. (4) Input buffer port. (5) Output buffer port. (6) MPI communication buffer.

tains 8 cores. For experimentation in this paper, we have used 8 nodes to match the 64 cores of Atlas.

5.2.1 Mandelbrot set

For the Mandelbrot benchmark, we compare the HitFlow version against a manual MPI version, and two versions using FastFlow [4], a structured parallel programming framework originally targeting shared memory multi-core architectures with specific extensions to support combined distributed- and shared-memory platforms. The first FastFlow implementation only uses the shared-memory layer. The second implementation includes the use of both the distributed and the shared-memory layers. The shared-memory pure version is directly the implementation included in the FastFlow distribution examples, that uses the FastFlow farm pattern. 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 [3]. This last version has a farm pattern inside each distributed node. There is also an upper tier producer that coordinates the work and feeds tasks to the nodes, with their lower tier farms.

All the implementations use a farm structure that processes the grid by rows. The manually developed one implements a simple farm algorithm in MPI. 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 the HitFlow channel implementation, and the work-stealing mechanism.

Figure 9 shows the results of the Mandelbrot implementations. The programs calculate the set in a grid of $2^{14} \times 2^{13}$ elements. The programs use up to 1 000 iterations to determine if each element belongs to the set leading to many low cost tasks to be processed. FastFlow shared-memory implementation scales better in Atlas, since the internal lock-free queues take advantage of the architecture. HitFlow implementation is simpler than the distributed FastFlow version because it uses a unique tier for both shared- and distributed-memory architectures. It shows worse scalability in the shared-memory machine but obtains the same results as FastFlow in the distributed one. This shows that HitFlow channel and work-stealing implementation have a great scalability in distributed environments, there that is still room for improvement in shared memory machines.

5.2.2 Swps3

For the swps3, we compare the original version [18], which is implemented using pipe and fork system calls to create several pro-

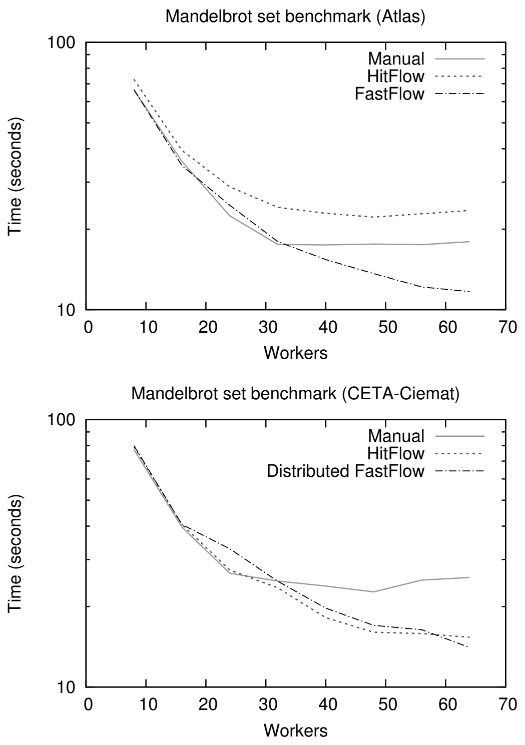

Figure 9: Mandelbrot set benchmark results.

cesses in the same machine, with the FastFlow and the HitFlow versions. The structure of this benchmark is a farm with an emitter. For a fair comparison, we have developed the FastFlow and HitFlow versions starting with the sequential code of the original swps3 benchmark. We have not used the original example included in FastFlow [2], since it uses some memory allocation optimizations and does not work for the bigger sequences chosen as input for our experiments, 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

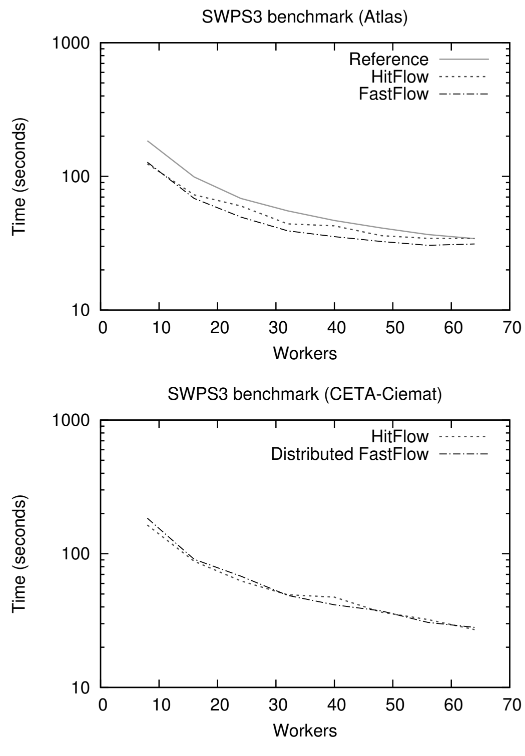

Figure 10: Swps3 benchmark results using the protein sequence Q8WXI7 as inputset.

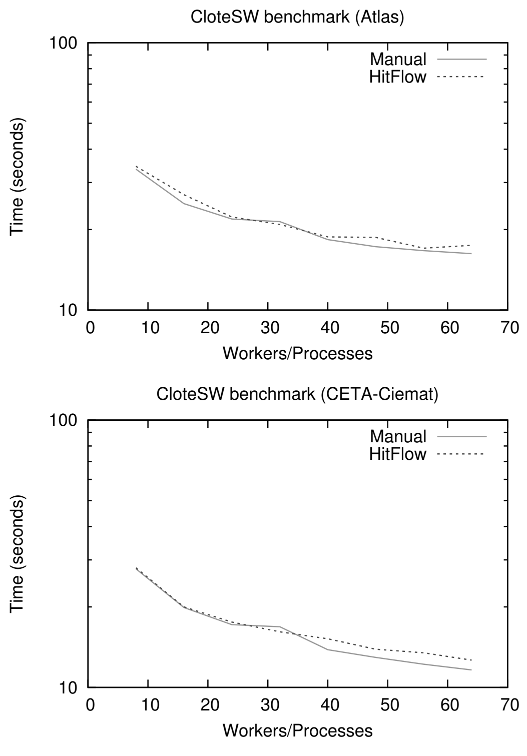

Figure 11: Clote's Smith-Waterman benchmark results.

(UniProt) [1]. This database consists of 544 996 sequences which minium 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. With this example, we test the HitFlow communication and work-stealing performance on a real application.

Figure 10 shows the experimental results for the sequence named Q8WXI7, which has 22 152 proteins. For the shared-memory machine, both HitFlow and FastFlow surpass the results of the reference version. FastFlow obtains a slightly better performance than HitFlow. In the cluster, there is no significant difference between both versions. 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 Clote's algorithm

The third benchmark, CloteSW, is a different implementation of the Smith-Waterman protein alignment that aims to compare two big sequences [7]. For this benchmark, we compare 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 a 100 000 x 100 000 elements matrix to be calculated with the alignment. 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 inverse order. We have developed a manual C++ & MPI version and a HitFlow version. The HitFlow implementation use the mode structure described in Fig. 2. The use of the modes in HitFlow allows the data affinity between phases of the benchmark to be defined, avoiding extra communications or coordinations. The results are shown in Fig. 11. Both versions show a similar performance.

6. Related work

HitFlow is a complement of Hitmap, a library for automatic but static hierarchical mapping, with support for dense and sparse data structures [9, 12]. The Hitmap library focuses on data-parallel techniques and lacks support for dataflow applications. In a previous work [10], 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.

HitFlow has several similarities with FastFlow [4], 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. 12 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. Regarding the system targeted, the FastFlow group has develop a distributed memory extension using a two tier model [3]. However, this solution forces the programmer to manually divide the program structure for the available memory spaces and use a different mechanism of external channels to communicate the tasks. In this sense, HitFlow makes the program design independent from the mapping. An extended comparison including development effort metrics of the previous two stage pipeline example using distributed FastFlow can be found in [11].

HitFlow networks are similar to CnC (Concurrent Collections [5]) 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 of the steps. 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.

OpenStream [17] is a dataflow OpenMP extension where dynamic independent tasks communicate through streams. The programmer exposes data flow information using pragmas to define the stream input and output task. This allows arbitrary dependence patterns between tasks to be created. A stream can have several producers and consumers that access the data in the stream using a sliding window. The OpenStream runtime ensures the coordination of the different elements. These streams are equivalent to the HitFlow places and they also relay on work stealing. However, OpenStream does not natively support distributed memory hence cannot handle load balancing at the cluster level. Trying to transfer this model to distributed memory involves the same problems as the use of distributed FastFlow.

S-Net [13] 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.

7. Conclusions

This paper presents a new parallel programming model and framework based on a dataflow paradigm. It allows programs to be described as a network of communicating activities in an abstract form. The system allows to implement 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 run-time, 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 manual implementations. Moreover, the results obtained in a distributed-memory environment show a similar performance to FastFlow, a dataflow programming framework for multi-core platforms.

This generic framework will allow us to focus research on the best mapping policies that can transparently target heterogeneous platforms for specific or generic combinations of parallel paradigms, allowing us to build powerful parallel patterns using a common and generic framework. Although, the experimental results show that the distributed support in our current prototype achieves good performance, there is still room for improvement on the thread-level communication implementation for shared memory systems.

Acknowledgments

This research has been partially supported by MICINN (Spain) and ERDF program of the European Union: HomProg-HetSys project (TIN2014-58876-P), CAPAP-H5 network (TIN2014-53522-REDT), and COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS), and by the computing facilities of Extremadura Research Centre for Advanced Technologies (CETA-CIEMAT), funded by the European Regional Development Fund (ERDF). CETA-CIEMAT belongs to CIEMAT and the Government of Spain.

References

[1] UniProt Knowledgebase (UniProtKB). `www.uniprot.org/`.

[2] M. Aldinucci, M. Meneghin, and M. Torquati. Efficient Smith-Waterman on Multi-core with FastFlow. In M. Danelutto, J. Bourgeois, and T. Gross, editors, *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 195–199, Pisa, Italy, 2010. IEEE Computer Society. ISBN 978-1-4244-5672-7.

[3] M. Aldinucci, S. Campa, and M. Danelutto. Targeting distributed systems in fastflow. In *Proceedings of the Euro-Par 2012 Parallel Processing Workshops*, volume 7640 of *Lecture Notes in Computer Science*, pages 47–56. Springer Berlin Heidelberg, 2013.

[4] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. FastFlow: high-level and efficient streaming on multi-core. In S. Pllana and F. Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing. Wiley, 1st edition, 2014. ISBN 0470936908.

[5] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasirlar. Concurrent collections. *Scientific Programming*, 18(3-4):203–217, Aug. 2010. ISSN 1058-9244. . URL `http://dx.doi.org/10.1155/2010/521797`.

[6] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann, 1 edition, 2001. ISBN 1-55860-671-8.

[7] P. Clote. Biologically significant sequence alignments using Boltzmann probabilities. Technical report, 2003. URL `http://bioinformatics.bc.edu/~clote/pub/boltzmannParis03.pdf`.

[8] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur. Enabling MPI interoperability through flexible communication endpoints. In *20th European MPI Users's Group Meeting, EuroMPI '13, Madrid, Spain - September 15 - 18, 2013*, pages 13–18, 2013. . URL `http://doi.acm.org/10.1145/2488551.2488553`.

[9] J. Fresno, A. Gonzalez-Escribano, and D. R. Llanos. Extending a hierarchical tiling arrays library to support sparse data partitioning. *The Journal of Supercomputing*, 64(1):59–68, 2013. ISSN 0920-8542.

[10] J. Fresno, A. Gonzalez-Escribano, and 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)*, pages 320–326, July 22-25, Las Vegas, NV, USA., 2013.

[11] J. Fresno, A. Gonzalez-Escribano, and D. R. Llanos. Additional material for the paper: Dataflow programming model for hybrid distributed and shared memory systems. Technical Report IT-DI-2015-0003, Department of Computer Science, University of Valladolid, Spain,

```cpp
#include <hitflow.h>
using namespace hitflow;

class StageA: public Transition {
   int numtasks;
public:
   StageA(int t): numtasks(t){};

   void create(){
      long task;
      for(int i=0; i<numtasks; i++){
         task = i;
         put(task);

      }

   }
};

class StageB: public Transition {
   long sum;
public:
   void init(){
      sum = 0;

   }
   void process(){
      long task;
      get(&task);
      sum += task;

   }
   void end(){
      cout << "Sum " << sum << endl;
   }
};

int main(int nargs, char * vargs[]){

   HitFlow::init(&nargs, &vargs);

   StageA st_a(10); StageB st_b;

   Place<long> place("long container");

   st_a.addOutput(&place,"createTasks");
   st_a.addMethod(&StageA::create,"createTasks");
   st_b.addMethod(&StageB::process,"processTasks");
   st_b.addInput(&place,"processTasks");

   Net net;
   net.add(&st_a); net.add(&st_b);
   net.run();

   return 0;
}
```

```cpp
#include <ff/pipeline.hpp>
using namespace ff;

class StageA: public ff_node {
   int numtasks;
public:
   StageA(int t): numtasks(t){};

   void * svc(void * intask){

      for(int i=0; i<numtasks; i++){
         long * task = new long;
         *task = (long) i;
         ff_send_out(task);
      }
      return NULL;
   }
};

class StageB: public ff_node {
   long sum;
public:
   int svc_init(){
      sum = 0;
      return 0;
   }
   void * svc(void * intask){
      long * task = (long*) intask;
      sum += *task;
      delete task;
      return GO_ON;
   }
   void svc_end(){
      cout << "Sum " << sum << endl;
   }
};

int main() {

   ff_pipeline pipe;
   pipe.add_stage(new StageA(10));
   pipe.add_stage(new StageB());
   if (pipe.run_and_wait_end()<0)
      return -1;
   return 0;
}
```

Figure 12: A full pipeline example in both HitFlow (left) and shared-memory only FastFlow (right) frameworks.

2015. URL `http://www.infor.uva.es/~jfresno/reports/IT-DI-2015-0003.pdf`.

[12] A. Gonzalez-Escribano, Y. Torres, J. Fresno, and D. R. Llanos. An Extensible System for Multilevel Automatic Data Partition and Mapping. *IEEE Transactions on Parallel and Distributed Systems*, 25(5): 1145–1154, May 2014. ISSN 1045-9219.

[13] C. Grelck, S.-B. Scholz, and A. Shafarenko. A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI : Portable Parallel Programming With the Message-passing Interface*. MIT Press, 2 edition, 1999. ISBN 0262571323.

[15] D. Holmes. Ideas for persistant point to point communication. Technical report, MPI Forum Meetings, 2014. URL `http://meetings.mpi-forum.org/2014-11-scbof-p2p.pdf`.

[16] K. Jensen, L. M. Kristensen, and 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):213–254, Mar. 2007. ISSN 1433-2779.

[17] A. Pop and A. Cohen. A OpenStream: Expressiveness and Data-Flow Compilation of OpenMP Streaming Programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):53:1–53:25, 2013.

[18] A. Szalkowski, C. Ledergerber, P. Krähenbühl, and C. Dessimoz. SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2. *BMC research notes*, 1(107), 2008. ISSN 1756-0500.