Project no. 288570

PARA PHRASE

Strategic Research Partnership (STREP)
PARALLEL PATTERNS FOR ADAPTIVE HETEROGENEOUS MULTICORE SYSTEMS

Prototype Implementation of run-time system optimisations
D5.3

Due date of deliverable: 31st Dec 2014

Start date of project: October 1st, 2011

Type: Deliverable
WP number: WP5
Task number: T5.3

Responsible institution: UNITO
Editor and editor’s address: Marco Aldinucci, UNITO

Version 2.1 (Revision: 2166)

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PU</td>
<td>Public</td>
</tr>
<tr>
<td>PP</td>
<td>Restricted to other programme participants (including the Commission Services)</td>
</tr>
<tr>
<td>RE</td>
<td>Restricted to a group specified by the consortium (including the Commission Services)</td>
</tr>
<tr>
<td>CO</td>
<td>Confidential, only for members of the consortium (including the Commission Services)</td>
</tr>
</tbody>
</table>
Contents

1 Introduction 4
  1.1 Target platforms ........................................... 4

2 Optimisation via Platform and code inspection support 6
  2.1 Platform inspection support ................................. 6
  2.2 Code inspection support .................................... 8
  2.3 Optimisation by way of thread Mapping and Pinning ...... 8

3 Optimised OpenCL and CUDA support 10
  3.1 Support for iterative stencil-reduce ....................... 11
  3.1.1 Stencil-Reduce-Loop API ................................ 11
  3.1.2 The FastFlow implementation ............................ 14
  3.1.3 Performance ............................................. 15

4 Performance optimisations on distributed platforms 17
  4.1 OFED support for InfiniBand network ..................... 17
  4.1.1 Message Passing Library for IB .......................... 18
  4.1.2 Development of FastFlow patterns based on the Message Passing library ........................................... 23
  4.1.3 Reference .................................................. 24
  4.1.4 Examples ................................................. 25
  4.1.5 Tests ....................................................... 26
  4.2 Experimental support for A3CUBE Ronniee .................. 28

5 Other enabling features for run-time support optimisations 32
  5.1 Active and passive mediators ............................... 32
  5.2 Dynamic re-configuration and over-provisioning .......... 33
  5.3 Optimisation policies ....................................... 34
Executive Summary

This Deliverable consists of a software prototype accompanied by this short technical note sketching the main design choices adopted in the software and explaining how to obtain and run the software. This document is accompanied by two software packages that are available on the project website at http://www.paraphrase-ict.eu/Deliverables/.

- **D5.3-ff_code_r282.tgz**: The source code and instructions of the C/C++ WP5 development framework with tests and examples. This package also includes material from D2.4.

- Remote installation accessible via ssh at paracool.di.unito.it on request.

- No VMs images are provided since they are not suitable to test GPU features and multi-core optimisation features.
Positioning of Deliverable D5.3

This Deliverable is the third deliverable the work package WP5, *D5.3: Prototype Implementation of run-time system optimisations*. The work package has three main objectives:

1. to develop the specification of a unified hardware-oriented set of APIs which virtualise hardware resources and enables programmers to interact seamlessly with the hardware infrastructure;
2. to develop a set of compiler optimisations to take advantage of the hardware capabilities; and,
3. to develop low-level support for allocating software components to the available heterogeneous hardware.

These objectives will be achieved in three main tasks:

T5.1 the identification of a hardware virtualisation API encompassing a uniform layer of synchronisation, communication, and mapping primitives for several different hardware resources (e.g. CPUs and GPGPUs);

T5.2 the definition of a lightweight component model bridging the abstraction gap from mechanisms defined in T5.1 and functional and extra-functional features from WP2 and WP3; and,

T5.3 the definition of a staged compilation system to support the lifting of pattern optimisations at the earliest possible stage (e.g. static and launch-time).

This deliverable (D5.3) reports the achievements of task T5.3 and addresses Objective 3 above. The main input to this deliverable consists of Deliverable D5.2 *Low-Level Virtualisation (Platform-Specific Deployment)*.

Deviations of Deliverable D5.3 with respect original DoW

- Transformation of compositions of patterns into others at compilation time has not been implemented because the same effect can be obtained by source-to-source transformations obtained by code refactoring (covered by WP4). The optimisation focus has been shifted on supporting global optimisation of the network of process-components (Chap. 5) and its mapping onto hardware platform (Chap. 2) and

- The design and prototyping of the run-time support for a novel RDMA-capable networking technology has been added to the workplan to enhance project exploitation (Sec. 4.1).
Chapter 1

Introduction

Deliverable D5.3 is a prototype plus a technical report about the Prototype Implementation of run-time system optimisations. The software package, which also includes material from D2.4 builds on top of D5.2. It is made available for download on the project website at \texttt{http://paraphrase-ict.eu/}. The software is distributed as two versions:

- a tarball with sources and instructions for compiling tests and examples. Tests can be found in the \texttt{D5.3-ff_code_r282/ff_r282/tests/} directory.
- a Linux x64 virtual machine with pre-installed software that can be run on the Virtualbox and VMWare virtual machine players. In the latter case, hardware accelerators (GPGPUs) are not available since their virtualisation is not well supported by current technology.

Note that this deliverable exclusively focus on the optimisation of the C++ run-time, and in particular of the FastFlow-based implementation of the Paraphrase Hardware Abstraction Layer (HAL), which is described in deliverable D5.1.

1.1 Target platforms

The ParaPhrase project targets heterogeneous platforms that can be described in the most general case as a physical or virtualised cluster of multicore platforms equipped with hardware accelerators. The reference platform can be characterised as follows:

- x86 multicore platforms, either 32 or 64 bit. Platforms based on the ARM Cortex MP (including NVidia ARM+GPU) are also supported. PowerPC-based multicore platforms are partially supported.
- Linux OS with the \texttt{gcc} compiler. MacOS (>10.5) with the \texttt{gcc/clang} compiler and Windows XP/7 with Visual studio 10 are also partially supported.
• NVidia GPGPUs with OpenCL and CUDA run-time support.

• Clusters built using TCP/IP networking and InfiniBand on top of the OFED stack.

• Physical or virtualised platforms. Presented software can run on public cloud infrastructures, such as Amazon EC2. On virtualised platforms, GPGPUs are not currently supported.
Chapter 2

Optimisation via Platform and code inspection support

A first opportunity to optimise the parallel code for both multi-core and GPGPUs is to match the configuration of the run-time support with the target execution platform. For this it is necessary to inspect the main features of the parallel code and the execution platform. These sources of information can be matched according to different policies in order to optimise different features. This matching is covered by WP3, whereas the mechanisms for inspection and the configuration of the run-time support are part of the WP5 (and described in this deliverable).

2.1 Platform inspection support

Platform inspection aims to describe the main features of the target platform. In this respect, the principal source of information is the Operating System. In the ParaPhrase FastFlow C++ run-time support, platform inspection is introduced as a facade pattern, which provides methods to retrieve information about processors, memory, and accelerators. The facade provides the interface between FastFlow and several third-party platform inspection libraries:

- the Hwloc library, available for different OSes [7];
- the native Linux inspector, which is based on the analysis of the /proc directory;
- the REPARA inspector, developed in the companion EU project (FP7 REPARA STREP) [17, 19].

The last inspector is able to retrieve a large number of information from both multi-core platforms and accelerators.
Architecture  FastFlow inspector is a set of c++ classes and wrappers able to supply information about the host machine where it runs. The aim is to allow programs using FastFlow to take decisions on CPU pinning, task size, worker number, etc.

The library is inspired by the result of the Repara project, which is partially reused. The class structure is inspired by the internal class structure of Repara, trying to give a one-to-one match between the data collection performed in Python and the c++ new classes. In addition to this, the inspector is built to support other inspectors, such as the one supplied by HWLoc and direct parsing of the Linux proc file system. Developers have a choice to collect data in different ways depending on their needs and on the installed software on the host machine.

The higher level of the architecture is a set of user level calls, where it is possible to query the hardware configuration. The interface shows to the user the same calls and data structure regardless of the underlying implementation chosen. In the lower level, several implementations of the user interface are available. Each one of them is able to read data from an external tool or directly from the operating system. The lowest level shows the tools or libraries involved by the implementation. The end user will just have to create an instance of the right implementation, call the methods and use the returned data structures.

Usage examples are reported in `extras/mapping/tests`. 

Figure 2.1: Connection between peers.
2.2 Code inspection support

Code inspection aims at statically profiling expected thread load in the FastFlow run-time. As described in deliverable [1], FastFlow applications are eventually compiled into a graph of nodes. A node is implemented by a thread. They can be structurally categorised in nodes (i.e. workers) realising a Single-Producer-Single-Consumer behaviour, and mediator nodes, i.e. minode and monode, realising a Multiple-Input-Single-Output and Single-Input-Multiple-Output behaviours, respectively. In FastFlow, these two categories of nodes are typically characterised by a different behaviour with respect to computing pressure on executing core.

Mediators are used as either schedulers or collectors. In their default behaviour (i.e. in the common case) they do not exhibit a compute load. However, their behaviour can be extended to compute an user-defined function. In the contrary, nodes are typically used to compute.

Code inspection implements two basic features:

- categorise all nodes according to their expected compute load;
- provide a description of the graph of nodes.

This information enable the optimisation of applications onto target platform by way thread pinning and thread mapping, especially in platforms with multiple sockets and contexts (i.e. hyper-threading).

Usage examples are reported in extras/mapping/tests.

A simple print from code inspection output is:

```
[ PIPE ([ WORKER docomp ]
  [ FARM ( EMITTER ) ( COLLECTOR )
    [ WORKER docomp ] ( WORKER docomp ) ( WORKER docomp )
  ],
  [ FARM ( EMITTER docomp ) ( WORKER docomp ) ( WORKER docomp )
    [ WORKER docomp ]
  ]
]
```

where EMITTER and COLLECTOR are mediator nodes, whereas other nodes are workers.

2.3 Optimisation by way of thread Mapping and Pinning

A proper deployment of a FastFlow application onto a multi-core platform can be used to optimise applications at the launch time by way of of thread pinning and thread mapping. These two techniques should be coupled to be effective.

- mapping of thread onto a core. It can be used to 1) reduce inter-socket data exchanges by mapping (whether possible) subgraphs within the same socket; 2) avoid the mapping of compute intensive threads onto different contexts of
the same (they share ALUs), promote the couple mapping of a worker thread and a mediator thread onto different contexts of the same core to minimise synchronisation latency and maximise core utilisation;

- pinning of a thread to a core: minimise OS jitter (fine grain applications), enhance locality and cache performance.

Communication bandwidth and synchronisation latency of FastFlow nodes mapped onto different contexts of the same code, different cores of the same socket, and different sockets are reported in [3].

The D5.3 includes a FastFlow prototype with all the support to map and pin threads to cores according a user-defined policy. This can be done either a launch-time (together with thread creation) or dynamically in any moment of the execution. Both platform inspection and code inspection support can be executed before the actual deployment of the parallel run-time in such a way this information can be used to optimise the deployment before thread creation. Dynamic mapping and pinning is instead aimed to support dynamic re-configuration of applications.
Chapter 3

Optimised OpenCL and CUDA support

The support of coupled usage of multi-core and GPGPUs is addressed with the support for a general meta-pattern (low-level pattern), i.e. *stencil-reduce-loop* pattern, able to support the implementation of various data-parallel patterns running on heterogeneous platforms. Both CUDA and OpenCL can be exploited for kernel code.

Since their appearance in the High-Performance Computing arena, GPGPUs have been widely perceived as data-parallel computing machines [16]. This belief stems from their execution model, which prohibits any assumption about work-items/threads execution order (or interleaving) in a kernel execution. This in turn requires the avoidance of true data dependencies among different parallel activities. It quickly became clear that the best approach to programming GPGPUs is to “think data-parallel” by way of “data-parallel building blocks” [16], i.e. data parallel skeletons [10–12]. For this reason, GPGPUs kernels are typically designed to employ the *map-reduce* parallel paradigm, where the *reduce* is realised as a sequence of partial (workgroup-level) GPGPU-side reduces, followed by a global host-side reduce. Thanks to GPGPUs’ globally shared memory, a similar pattern can be used to map computation over stencils (i.e. data overlays with non-empty intersection), provided they are accessed in read-only fashion to enforce deterministic behaviour. Often, this kind of kernel is called in host code in a loop body (e.g. up to the convergence of the process).

Specifically, *stencil-reduce-loop* is designed as a FastFlow pattern, which can be nested in other stream parallel patterns, such as *farm* and *pipeline*, and implemented in C++ and OpenCL [5] or CUDA [4,6].

The *stencil-reduce-loop* pattern:

- It is sufficiently general to subsume *map, reduce, map-reduce, stencil, stencil-reduce*, and, crucially, their usage in a loop, i.e. implementing the previously mentioned “data-parallel building blocks”. Also, it is more expressive than previously mentioned patterns.
- It simplifies GPGPU exploitation. In particular, it takes care of device detection, device memory allocation, host-to-device and device-to-host memory copy and synchronisation, reduce algorithm implementation, management of persistent global memory in the device across successive iterations, and enforces data race avoidance due to stencil data access in iterative computations.

- It can transparently exploit multiple CPUs or GPGPUs (sharing host memory) or a mix of them. Also, the same host code can exploit both a CUDA and OpenCL implementation (whereas the kernel functions should match the selected language). The CUDA implementation is not described in this paper (see [4]).

- It can be extended to support multiple GPGPUs on different platforms (via MPI or other messaging protocol). This is not described in this paper, but planned as future work.

3.1 Support for iterative stencil-reduce

3.1.1 Stencil-Reduce-Loop API

Let \( \text{map } f[a_0, a_1, \ldots] = [f(a_0), f(a_1), \ldots] \) and \( \text{reduce } \oplus [a_0, a_1, \ldots] = a_0 \oplus a_1 \oplus \ldots \), where \( f \) is the elemental function, \( \oplus \) is the combinator (i.e. a binary associative operator), and \([a_0, a_1, \ldots] \) an array (e.g. the pixels of an image). These parallel paradigms have been proposed as patterns both for multicore and distributed platforms, GPGPUs, and heterogeneous platforms [10–12]. Let \( \text{stencil} \) be a map, except that each instance of the elemental function accesses neighbours of its input, offset from its usual input [15]. They are well-known examples of data-parallel patterns, since the elemental function of a map/stencil can be applied to each input element \( a_i \) independently from each other, and also applications of the combinator to different pairs in the reduction tree of a stencil can be done independently, thus naturally inducing a parallel implementation.

The framework provides constructors for these patterns, in the form of C++ macros that take as input the code of the elemental function (combinator) of the map/stencil (reduce) patterns, together with an eventual read-only structure (i.e. the \textit{environment}) that can be regarded, from a pure functional semantics perspective, as a function parameter. The language for the \textit{kernel} code implementing the elemental function, which is actually the business code of the application, has clearly to be platform-specific (i.e. CUDA/OpenCL for GPGPUs, C++/OpenCL for multicore CPUs), since FastFlow is a header-only library and does not provide any compiler. Unlike the CUDA version, fully supporting C++ class template abstractions, the current OpenCL version still leverages on a C-style macro due to limitations of OpenCL 1.1 (C++ class templates not supported). A redesign of the pattern implementation for OpenCL 2.1 is currently underway [13].
OpenCL_STENCIL_KERNEL /* elemental function */
/* kernel parameters */
MF_kernel /* kernel id */, float /* basic argument type */,
input /* input—array name */,
env /* read—only environment name */,
len /* length of the input—array */,
i /* input—array index name */,

/* begin OpenCL kernel code */
float sum = input[i] * env[i];
if (i > 0) sum += input[i-1];
if (i < (len-1)) sum += input[i+1];
return sum / 3;
}

CPP_REDUCE_KERNEL /* combinator */
/* kernel parameters */
SUM_kernel /* kernel id */, float /* basic argument type */,
a /* first argument name — i.e. left input of the combinator */,
b /* second argument name — i.e. right input of the combinator */,
env /* read—only environment name */,
/* begin CPP kernel code */
return (a + b);
}

Figure 3.1: 1D-convolution stencil-reduce example.
In order to use the proposed pattern, the user has to write three macros:

- \((a)\) that defines the parameters and the elemental OpenCL function that will be applied to each element using the desired stencil shape.

- \((b)\) that defines the OpenCL combinator function that will be applied after the map phase.

- \((c)\) the stencil-reduce macro that takes as argument the user-defined datatype that will encapsulate all required data and macros as described in \((a)\) and \((b)\).

The user can additionally choose the convergence criterion that determines when the iterative mechanism should stop, based, for example, on the result obtained from the combinator function or by simply defining an upper bound on the number of iterations.

Fig. 3.1 sketches the implementation of a simple convolution filter followed by a sum operator. In the stencil phase, a classic 1D-convolution filter \((N\text{-neighbours MeanFilter with } N \text{ odd})\) is applied on input array \(A\), producing output array \(A'\).

In the example, the OpenCL version of the stencil pattern is used, thus exploiting automatic GPGPU offloading. Note that the programming model is data-oriented rather than thread-oriented, since the \(i\) index refers to the input array space rather than the work-items (i.e. threads) space - which is in turn the native programming model in OpenCL. In the multi-device version of the stencil-reduce-loop, the user is provided also with a local index \(i'\) that takes values over the space of the device-local subset of the input array. Thus, in order to exploit multiple GPGPU devices on a single input array, the user effort amounts to just refactoring the stencil kernel code, replacing \(i\) with \(i'\) when accessing the input array. It also has to specify two additional parameters for the instantiation of the pattern, namely:

- the number of devices to be used

- the maximum width of the neighbourhood accessed by the elemental function when called on each point of the input array

Note that the second parameter could be easily determined by a static analysis on the kernel code.

In the reduce phase, the residual array \(|A' - A|\) is passed through a reduce operator (a standard Sum) which computes the sum of its elements. In the example, the C++ version of the reduce pattern is used.

Note that, from an ease-of-development perspective, the user has to provide only the kernel code of the elemental function of the Mean Filter stencil-based operator and the combinator of the Sum reduce-based operator. Moreover, from a performance portability perspective, the user does not have to care about the underlying platform and has to provide only the kernel code body, since FastFlow manages all the kernel header definition, memory transfers and platform heterogeneity during the application deployment.
Usage examples of the stencil-reduce-loop meta-pattern can be found in tests/ocl/StencilReduceLoop. Runnable source code examples are included for both

- **one-shot** (i.e. single task):

  1. StencilReduceLoop/oclStencilReduce.cpp
  2. StencilReduceLoop/oclMap.cpp
  3. StencilReduceLoop/oclReduce.cpp
  4. StencilReduceLoop/oclMapReduce.cpp

- **and streaming**:

  1. StencilReduceLoop/farm_oclMap.cpp
  2. StencilReduceLoop/farm_oclReduce.cpp
  3. StencilReduceLoop/pipeFarmMap.cpp

usages styles of the patterns.

### 3.1.2 The FastFlow implementation

The effectiveness of the approach is discussed in [4–6], where a variational image restoration algorithm is used as a benchmark. In algorithms of this class, the convergence of the process is typically evaluated at each iteration, the reduction of results across three successive iterations. This requires the management of the GPGPU’s global memory across multiple iterations, i.e. across different kernel invocations. The persistence of the GPGPU global memory across multiple kernel invocations is quite common in iterative applications because it drastically reduces the need for Host-to-Device and Device-to-Host copies, which can severely limit the speedup. This also motivates the explicit inclusion of the iterative behaviour in the stencil-reduce-loop pattern design which is one of the differences with respect to solutions adopted in other frameworks, such as SkePU [10].

The general schema of the stencil-reduce-loop pattern is described in Fig. 3.2. The stencil-reduce-loop runtime is tailored to efficient loop-fashion execution. When a task is submitted to be executed by the devices the pattern is deployed onto, the runtime takes care of allocating on-device memory buffers and filling them with...
input and environment data via Host-to-device copies. Inside the loop, before off-loading the actual kernel invocation to the devices, the runtime realises a partial synchronisation between the devices by Device-to-Device copying the overlapping portions of the output buffers – instead of copying the whole output. Then it swaps the on-device I/O memory buffers, since the input for an iteration is the output of the previous one. Then a kernel call is enqueued and the previously discussed two-stage reduction is performed.

### 3.1.3 Performance

The image restoration system presented in [4–6] has been tested for performance measurement. All experiments reported in this section were conducted on an Intel workstation with 2 eight-core double-context (2-way hyperthreading) Xeon E5-2660 @2.2GHz, 20MB L3 shared cache, 256K L2, and 64 GBytes of main memory (also equipped with two NVidia Tesla M2090 GPGPUs) with Linux x86_64. Figure 3.3 reports the throughput (i.e. frames per second) obtained by running different deployments of the restore stage over a video stream (under different noise-level conditions), thus exploiting different parallelisation schemas over different combinations of both CPU and GPGPU usage. A purely CPU deployment is included as a baseline, in which each frame is passed through a stencil-reduce-loop OpenCL version of the filter, deployed onto the (cores of the) CPU. Notice that the baseline configuration exploits multicore parallelism onto each single frame, thus resulting in an intra-frame parallelisation schema. The baseline is compared against different GPGPU-deployments of the stencil-reduce-loop OpenCL version of the filter. First a version deployed onto a single GPGPU is included, where intra-frame parallelism is exploited by implementing the stage as a FastFlow farm with a single worker executing the stencil-reduce-loop on the GPGPU. Moreover,
a two-GPGPU intra-frame version is included, which is implemented as a straightforward refactoring of the previous one. Finally, a version using a multi-level parallelisation schema is shown which is implemented with a FastFlow farm of 2 workers (i.e. inter-frame parallelism), each running the OpenCL stencil-reduce-loop version of the filter (i.e. intra-frame parallelism) on a dedicated GPGPU. The maximum speedup obtained with respect to the baseline CPU version (which has a parallel implementation) ranges from 8.22 for 10% noise to 19.04 for 90% noise.

As expected, mapping the restore stage onto GPGPUs greatly accelerates the application. Also, multiple GPGPUs can be used with almost linear speedup. It is worth noticing that in the OpenCL case it is possible to deploy the restore stage on both CPUs and GPGPUs, actually mapping different workers of the farm on different devices, thus fully using the aggregate computational power of CPUs and GPGPUs for the restore stage.
Chapter 4

Performance optimisations on distributed platforms

4.1 OFED support for InfiniBand network

The run-time support of the FastFlow framework for distributed platforms has been initially designed to target TCP/IP networks by way of the third-party 0MQ library [2]. The 0MQ library has been introduced in FastFlow to simplify handshaking and set up of the distributed collection of processes of the run-time system. Despite the native support for high-performance networks (such as the InfiniBand network) was in the development roadmap of 0MQ, at today it has not been developed (and it seems exited from the development road-map). FastFlow native support for the InfiniBand (IB) network aims to optimise performance of distributed applications on top of the FastFlow distributed run-time support.

It is presented here a redesign of FastFlow distributed run-time support based a native IB transport, namely OFED (OpenFabrics Alliance Distribution). It allows to develop native software for InfiniBand and it has been used by most of well known software such as MPI distributions (MVAPICH, OpenMPI, . . . ), as well as native IB file systems, a number of commercial solutions and scientific open-source experiments.

Despite OFED is InfiniBand-specific, its API is designed to capture most of the general hardware features exposed by almost any RMDA-capable high-performance networks. For this, the run-time support of FastFlow for OFED can be expected to be easily ported also to other technologies (such as A3CUBE Ronniee network).

The FastFlow integration for IB/OFED consists on two main independent modules:

1. Implementation of a Message Passing library for IB
2. Development of FastFlow patterns using the Message Passing library

The idea behind this is to reuse as much as possible the patterns implementation developed for TCP and to give a full compatibility of the current distributed
FastFlow developed software on IB. A detailed description of material (with comparative testing) can be found in [18].

4.1.1 Message Passing Library for IB

This library has been implemented from scratch. It is a simplified message passing library, where the main purpose is to develop just a simple and well performing interface targeted at our purposes. From the user viewpoint, the library consists on:

- Transport layer
- Descriptor
- Read and Send calls

The most simple implementation of a software connecting two peers based on such library consists of a few steps:

- Transport layer creation
- Descriptor creation
- Initialisation of descriptor
- Loop on send and/or recv calls.
- Disconnection and finalisation

However, because of the InfiniBand architecture, internally the library is more complex. OFED requires that two peers connect themselves using a native protocol to take advantage of the InfiniBand network.

In the development of the library, we identified some internal modules:

- Nodes Connection
- Memory registration
- Work request
- Shared FIFO queue
- Data dispatch
- Data reception
- Completion queue processing
- Flow control

**Nodes Connection** The most simple and common option to connect to peers is to use a specific library named *RDMA Connection Manager* (RDMACM). This event based library allows connecting two peers through the usage of the TCP stack based on IP over IB. This choice allows to simplify the required calls for the initial connection, allowing to easily integrate the system software such as middleware already available at the user’s site.

The main difference between a native TCP connection and our implementation is how the peers uses TCP ports. While in TCP the actual communication happens to a random port after the two peers established a first contact to a specific port, with RDMACM this process has to be re-developed. To simplify this process, we
assumed that the processes are able to be assigned to a set of unused ports at launch time.

Therefore we implemented a simple connection protocol with a static port assignation:

- The passive side listens to a number of ports equal to the number of active sides.
- The active side connects to the passive side through a system or user statically assigned port.
- The passive side processes all the incoming connections, one by one.
- The connection is done when all the passive sides are connected to all the active sides. As a connection is established, a \texttt{rdma\_cm\_id} structure for each active side will be created on the passive side, while, on the active side, a single \texttt{rdma\_cm\_id} structure is created to identify the opened connection.

**Memory registration** InfiniBand supplies \textit{Remote Direct Memory Access} (RDMA), a way to access remote memory areas between nodes, where the network card can
directly access the memory on the local host and manages some data structures needed to coordinate the communication between peers.

The operation of Memory Registration is needed to define the memory areas used for RDMA. It consists of "pin down" memory by the OS and exchanging Rkeys between the peers. Our implementation follows this order:

- Memory allocation (malloc)
- Locally record the allocated memory to the network card.
- Rkey dispatch: a special structure containing the memory area information is sent to another host.

To avoid races when both peers have to share a memory area, the dispatch of Rkey is managed in different ways if the node is passive or active: the passive node send the Rkey and waits the Rkey back from the active node; the active node instead waits to receive Rkey from before sending it (see figure 4.2).

**Work request** When a node wants to send a part of its memory to a peer, it need to perform a Work Request (WR). This is an operation where a structure has to be filled with information about the memory area to be sent, which is a subset of the registered memory. Since InfiniBand supplies many ways to fill this structure, we created an abstraction for internal use where the structure is filled with the needed information.

**Shared FIFO Queue** The FIFO queue is a main structure in our implementation (see figure 4.3. Sending a message consists on enqueue it and create a Work Request, while receiving messages is a dequeue operation. Both sides have to update the FIFO queue structure: while the sender updates the head information, the receiver will update the tail.
The structure containing the queue has to be directly accessed by the memory card: this means that both sender and receiver has to allocate an area where to store the information and call the Memory Registration before accessing it. The queue has been implemented to circularly use this area, defining a head pointer (the location where a new data has to be inserted), and a tail pointer (where the next data will be removed). The implementation manages the situation of full and empty queue, as well as the passage of the pointers from the end to the beginning of the buffer.

To implement the onDemand protocol, the memory shared by the receiver also contains a counter to the request of new data; this has to be incremented and sent each time that the node is available to receive a new message from the sender.

**Data dispatch**

As mentioned before, data dispatch is supplied by a `send` call, which calls `enqueue` and creates a Work Request. Eventually, the flow control and completion queue processing are called, to ensure the correct utilisation of the InfiniBand data structures.

To make the process of sending messages faster and independent from the receiver state, a cache has been implemented so that, when the queue is full because the receiver is too slow, the sender doesn’t have to wait and can continue to produce data (see figure 4.4).

**Data reception**

The `recv` call wait for an incoming message on queue. Because of the asynchronous nature of InfiniBand, the recv method has to enforce synchronisation in order to satisfy the requirements of the data streaming model, in which the receiver side is always synchronous on the incoming data. Data reception is made by polling the Completion Queue until a new message is received, and calling the dequeue method of the FIFO queue.

**Completion Queue processing**

The Completion Queue is a structure supplied by OFED and hardware managed where the messages following a Work Request
are stored. Depending on the type of request, there may be a completion message for a send, while there’s always a completion message for a recv.

Reading the completion queue is useful to synchronise operations between peers. On the receiver peer it is used to wait a message on the queue, polling the completion queue until a message arrives. On the sender side, a completion is generated only when a specific RDMA write Working Request has been made. In our implementation we chose to use completions on the sender to implement the flow control.

**Flow control** InfiniBand gives a partial responsibility to the programmer to decide if and how to regulate the amount of data sent between the peers. In our implementation, the flow control is important to avoid filling the InfiniBand outgoing queues and overwriting the buffer not yet completed by the send operations.

The FIFO queue already perform a flow control. In fact, it is not possible to fill the queue and the outgoing flow is blocked when the queue is full. However this is not enough, and the flow control also has to consider not to overload the InfiniBand internal data structures.

A more effective control is in fact performed comparing the number of currently sent messages with the number of received completions. The difference between such numbers gives the amount of pending work requests, and so it is mandatory to keep this number lower than the outgoing queue.
4.1.2 Development of FastFlow patterns based on the Message Passing library

Once the Message Passing library development was over, porting FastFlow patterns was a more simple task. In fact our work partially re-used the code written to integrate FastFlow on TCP/IP networks.

The supported patterns are:

- Unicast (one to one)
- Broadcast
- AllGather
- FromAny
- Scatter
- OneToMany
- OnDemand

Unicast (one-to-one) This is the most simple pattern, connecting to single peers in one direction. The pattern didn't required any effort because it exactly match the message passing library: a send call corresponds to a single send call of the message passing library and the same for the recv call.

Broadcast Broadcast is a one-to-many pattern in which a single peer sends the same message at the same time to multiple peers.

Our implementation of such pattern consists on opening multiple one-to-one connection and copy each outgoing message to all the outgoing buffers. This is currently a naive implementation that performs well only on a limited number of peers, until the time to copy the outgoing buffer exceeds the computing time on the workers.

AllGather This is a many-to-one pattern in which incoming messages are read in a specific order from the peers. As all the patterns connecting multiple peers, it is implemented using multiple one-to-one patterns. The implemented recv call just loops through the connected peers, waiting until each of them has sent their message. To perform this, a blocking recv call is made for each connected peer.

FromAny This is again a many-to-one pattern where incoming messages may arrive from any connected node in any order. In this case the recv call loops through the nodes waiting that one of them has sent a message. The main difference with AllGather is that, in this case, it is not predictable which will be the peer sending the next message. In fact, before calling the blocking recv on a connected peer, the receiver tests which of the nodes sent messages with a non-blocking call.
**Scatter**  Scatter is a one-to-many pattern, where data are sent to peers in a specific order. Being the pattern implemented with multiple one-to-one connection, the only challenge was to call a send for each connected peer in the correct order.

**OnDemand**  OnDemand is a one-to-many pattern in which messages are sent to the first peer that sent a request. In this case, the main protocol requirement is to implement a mechanism to request a task from a receiving peer.

Each receiving peer share a counter to the sender, which is incremented and sent through a specific RDMA call when it is available to accept a new message. The sender side loops through the receivers to and choose the first one that incremented their counter to choose which has to receive the next message.

### 4.1.3 Reference

**class ofedTransport**  Define the transport layer. The constructor prepares the internal data structures needed by the node for the connection.

Constructor ofedTransport(int <procId>)

- **procId**: it is used by FastFlow to identify the process in which this call has been performed

**class descriptor**  Define the equivalent of a TCP/IP descriptor. It is needed to create a connection between peers, where the connection may be one-to-one, one-to-many and many-to-one depending on the parameters.

Constructor descriptor(string <name>, int <peers>, ofedTransport* <transport>, bool <passive>, bool <sender>)

- **name**: A string giving a name to the connection. All the peers involved must share the same name.
- **peers**: The number of peers to be connected.
- **transport**: A pointer to the transport structure.
- **passive**: If true, identifies the server peer, the one accepting connections.
- **sender**: If true, identifies the sender peer, the one allowed to call the send method

Method void init(string <addr>, int nodeId) perform the connection stage.

- **addr**: The remote connection address:port or the local address:port
- **nodeId**: A node identifier

Method ofedConnection** conn = d->getId(); It returns an array of pointers to the data structure identified each opened connection
class ofedConnection  It contains the description of a connection. It supplies

```c
conn[0]−recv(m);
for (long i=0; i<NUM_OF_MESSAGES; i++) {
    msg_t m;
    conn[0]−recv(m);
    long r =*((long*)(m.getData()));
    // (do something with r)
}
d−close();
t−closeTransport();
```

4.1.4  Examples
4.1.4.1 Simple producer-consumer with the Message Passing library

Producer

```c
ofedTransport* t = new ofedTransport(0);
descriptor* d = new descriptor("SERV", 1, t, 1, 1);
d−init(ADDRESS, 0);
ofedConnection* conn = d−getId();
for (long i=0; i<NUM_OF_MESSAGES; i++) {
    char* c = new char[SIZE];
    memset(c, 0, SIZE);
    *((long*)c) = i;
    conn[0]−send(m);
    delete c;
}
d−close();
t−closeTransport();
```

Consumer

```c
ofedTransport* t = new ofedTransport(0);
descriptor* d = new descriptor("SERV", 1, t, 0, 0);
d−init(ADDRESS, 0);
ofedConnection* conn = d−getId();
for (long i=0; i<NUM_OF_MESSAGES; i++) {
```
msg_t m;
conn[0]->recv(m);
long r = **((long*)(m.getData())));
// (do something with r)
}

d->close();
t->closeTransport();

4.1.5 Tests

Unidirectional bandwidth test In this test, messages are sent continuously from a process located in peer A to a process in peer B. The message sizes range from 10 bytes to 400,000 bytes. We show the results of a single pipe from two implementations: FastFlow using native InfiniBand support and TCP support through ZeroMQ. We also compare the results with the command ib_write_bw, a standard diagnostic test for RDMA, supplied with OFED, giving a reference bandwidth measure, and a similar MPI test.

The following pseudo code shows how the test was developed. In MPI the send and recv are just replaced with MPI_Send and MPI_Recv. The FastFlow test is composed by a producer generating a number of messages of the same size and a consumer, which processes such messages. The distributed pipe connecting the two nodes is, in one case, shared through the RDMA mechanism and, in the second case, the ZeroMQ implementation running on the top of TCP/IP over InfiniBand.

The results shown in Table 4.1 prove the good performance improvement for FastFlow on InfiniBand using our native library with respect to ZeroMQ using TCP/IP over InfiniBand (IPoIB). It also shows that our solution gives similar results of MPI for most of message sizes, with some advantage in the range of 25k/64k sized messages. MPI gives better performance for messages bigger than 200k and smaller than 10K, because of the highly optimized implementation. While we consider hard to reach the same transfer rate of MPI, we think it is possible to improve our solution with better transfer rates with some future software optimization.
Table 4.1: Comparing throughput of different implementations of the unidirectional bandwidth test for several message sizes.

We expected a big difference between using native InfiniBand against ZeroMQ on TCP/IP over IB, but the second performed under expectations for the biggest sized messages. We suspect that the network cards we tested may be better configured to achieve a better bandwidth. In any case, the support for TCP/IP is attractive only when the network does not have InfiniBand hardware.

**Ping pong** The ping-pong benchmark is a standard test used for the measurement of latency in a distributed environment. It sends a single message from a node to another and back, measuring the round trip time, for different message sizes. The pseudo-code of the ping-pong test is shown in the following:

```plaintext
1 Host A:
2     init connection
3    message = new Message(size)
4   for (i=0; i<LOOPS; i++)
5       send(message)
6       message = recv()
7
8 Host B:
9     init connection
10    start timer
11   for (i=0; i<LOOPS; i++)
12      message = recv()
13     send(message)
14    end timer
```

Table 4.2 compares the mpptest ping-pong implemented using MPI [14], against the equivalent FastFlow version using ZeroMQ over InfiniBand and using the IPoIB.
<table>
<thead>
<tr>
<th>message size (bytes)</th>
<th>MPI/IB (us)</th>
<th>FastFlow/IB (us)</th>
<th>FastFlow/ZMQ/IPoIB (us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1.47</td>
<td>1.66</td>
<td>45</td>
</tr>
<tr>
<td>100</td>
<td>1.75</td>
<td>1.92</td>
<td>46</td>
</tr>
<tr>
<td>1,024</td>
<td>3.91</td>
<td>3.96</td>
<td>49</td>
</tr>
<tr>
<td>5,000</td>
<td>6.66</td>
<td>6.56</td>
<td>61</td>
</tr>
<tr>
<td>10,000</td>
<td>9.13</td>
<td>9.03</td>
<td>71</td>
</tr>
<tr>
<td>25,000</td>
<td>13.74</td>
<td>16.39</td>
<td>84</td>
</tr>
<tr>
<td>50,000</td>
<td>21.5</td>
<td>28.34</td>
<td>120</td>
</tr>
</tbody>
</table>

Table 4.2: Communication latency (microseconds) of the ping-pong benchmark for different message sizes and implementations

driver, and the same version using our native InfiniBand library. This test again shows the benefits, for FastFlow, to adopt our solution in a native InfiniBand environment instead of using IPoIB driver and a TCP/IP-based communication library. Comparing to mpptest, our implementation behaves well for small sized messages, giving good results. It suffers some extra latency when the message size increases, because of the memory copy on the sender side. In fact, while data streams may benefit from sending data asynchronously, the ping pong test makes data sending synchronous, because every send has to wait for a read. This makes the latency of the send call more evident. In our tests, a simulation of a zero-copy send where the copy is just not performed, cancels the gap between the two versions.

4.2 Experimental support for A3CUBE Ronniee

This section reports a preliminary experiences with A3CUBE Ronniee networking technology. The reported material is aimed to show that the same approach used to support the OFED/IB stack in the FastFlow run-time can be also used for other networking technologies. This does not mean necessarily that the A3CUBE Ronniee technology (and similar technologies) cannot be used in a more advanced fashion. A specific discussion on this latter topic will be reported in the final exploitation report deliverable (which is confidential).

Ronniee is an emerging network technology based on a shared memory model. The process of integrating Ronniee and FastFlow has partially reused the implementation for InfiniBand, with some simplification thanks to the model. The work for Ronniee focused on a number of modules:

- Initialization
• Memory mapping and connection
• Shared FIFO queue
• Data dispatch
• Data reception
• Flow control

The work made for distributing FastFlow on TCP/IP was still valid and it has been reused as it was for InfiniBand.

Initialization  The stage of initialization is when each peer calls a set of functions to set up the internal data structures needed for the correct card functionality. At this time there’s no connection between the nodes and there’s no need to synchronize the peers.

Initialization of the network card is performed in the transport initialization and it consists on calling `DPAInitialize`. In the initialization of descriptor, the node ID is read through calling `DPAGetLocalNodeId`. After such calls, it will be possible to call memory mapping functions and begin to connect.

The needed parameter to initialize the descriptor is a tuple in the format nodeid:port. This format has been chosen to keep a compatibility to the typical TCP/IP address so that it is possible, in future, to perform a translation between IP address and RONNIEE node ID. The meaning of the tuple is quite straightforward: nodeid is an identifier associated to the network card, typically an integer number such as 4, 8, 12 and so on. The port, instead, will be used in the memory mapping stage and it will be an ID number associated to a memory segment.

Memory mapping and connection  After the initialization, each peer needs to allocate the data structures needed to implement the shared FIFO queue and to make them available remotely to the peer.

The call needed to create a segment is `DPACreateSegment`. This call needs an unique identifier that when a segment is created, an ID number is associated to it, chosen at runtime: in our case, this is already passed as an initialization parameter. After the next calls (`DPAPrepareSegment`, `DPAMapLocalSegment` and `DPASetSegmentAvailable`) the segment will be mapped to the user address space and ready to accept connections from the peer.

Since both peers will have to share a part of their memory, a connection stage is performed on all of them. This is made through the call of `DPAMapRemoteSegment`, a synchronized call after which the remote segment is mapped locally. The function call needs to specify which node and segment ID to connect. Before the peer may write to such segment, a couple of calls (`DPACreateMapSequence` and `DPAStartSequence`) are still needed.
Figure 4.5: Example of connection protocol.

**Shared FIFO queue**  The queue has been reused as it was coded for InfiniBand; the documentation written for the InfiniBand section are still valid with some minor differences: some minor optimization performed to increase speed on InfiniBand are not needed here, so the code has been slightly simplified.

**Data dispatch**  Like for InfiniBand, data dispatch is a call to enqueue, with data caching when the queue is full. The main different with the previous implementation is that, in this case, there are no work requests to fill: data is automatically sent to the peer by the network card transparently without further efforts.

**Data reception**  Data reception is a call to dequeue, with busy waiting if the queue is empty. As for data dispatch, there’s no need to process any other data structure, as data are received directly on the local memory.

**Flow control**  Since there’s no data structures to regulate the communication between the peers, the only flow control implemented at software level are made on the FIFO queue: when the queue is full data are not sent until the peer changes the pointer to the tail. When data are not sent, the sender uses a caching mechanism to send data in a second time and avoid to block the program flow.
Figure 4.6: Memory registration.
Chapter 5

Other enabling features for run-time support optimisations

As described in the ParaPhrase Deliverable D5.2 [1], the FastFlow run-time support is composed of a graph of nodes (process-components, i.e. Posix processes or threads at the implementation level). This graph is build by composing subgraphs, each of them implementing patterns exploited in the application code. Composition is driven by pattern nesting, and realised by way of C++ meta-programming in such a way the resulting graph is not affected by deadlocks or live-locks. Also, this graph has a static nature and dimensioned to exploit the maximum parallelism configured either statically (e.g. in the application code) or at launch time (i.e. malleable executable). Nodes implemented as processes have a blocking behaviour, whereas nodes implemented as threads have a non-blocking behaviour. This design choice make it possible to efficiently target both coarse grain and fine grain parallelism, respectively [2, 3]. However, it should be noticed that non-blocking threads cannot be over-provisioned. Typically, one thread fully uses one hardware context (i.e. a core at OS level).

In the present report two additional enabling features for optimisation are introduced:

5.1 Active and passive mediators

The possibility to substitute active mediators with passive mediators, i.e. data structures and synchronisation protocols to coordinate nodes logically linked to the mediator. The substitution of active mediators with passive mediators helps to reduce the number of active threads, which may enhance power consumption and performance. Experimental results on both active and passive mediators show that none of the two approaches is always better than the other for performance (whereas passive approach is almost always beneficial for power consumption). Depending on the workload, either active or passive mediator approach is to be selected. As rule of thumb, the passive mediator approach is beneficial for perfor-
mance if the at least one of the following hypothesis holds true:

- coarse grain tasks;
- limited number of hardware cores with no hyper-threading;
- lock-step (synchronous) parallelism exploitation.

This features is exploited in several high-level patterns, which can be executed with either “active” or “passive” mediators. As an example in the ParallelFor pattern the following member function can be used to enable or disable the active mediators (default active):

```cpp
void f::ParallelFor::disableScheduler (bool onoff = true)
```

Examples of usage can be found in:

1. D5.3−ff_code_r282/ff_r282/tests/test_parfor_unbalanced.cpp
2. D5.3−ff_code_r282/ff_r282/tests/test_parforpipereduce.cpp

### 5.2 Dynamic re-configuration and over-provisioning

Over-provisioning, i.e. the possibility to start more threads than cores, is sometimes useful for the dynamic optimisation of applications, especially in the cases when the parallelism desgree cannot be statically sized.

Over-provisioning, however, does not match well with non-blocking threading, which typically employs lock-free synchronisations (and mutual exclusion is needed to suspend threads at the OS level).

To make over-provisioning possible also for non-blocking threads, the FastFlow run-time support provides the programmers with the possibility to dynamically “freeze” (i.e. suspend) part of the run-time in oder to dynamically alter the concurrently executing threads. Both single nodes and subgraphs can be frozen (and thawed). In the latter case, multiple cooperating nodes should be suspended and, in order to preserve correctness (no data-races, deadlocks, live-locks), a concurrent protocol is used. This protocol is actually the same used for termination with a different interpretation of the End-Of-Stream (EOS) token.

In FastFlow, a pattern can be freezed if started with a specific run method, i.e. `run_then_freeze()` which is available for each pattern. Once a pattern is suspended, all the nested patterns are suspended according to the previously mentioned protocol. This means that whole subgraph of nodes implementing the pattern composition will be suspended. The same pattern can be restarted by invoking again the same method. At the implementation level, the whole subgraph of nodes implementing the composition will be restarted in a nonblocking mode.

Several examples of usage are provided under the `tests/` directory, e.g.

1. D5.3−ff_code_r282/ff_r282/tests/test_freeze.cpp
2. D5.3−ff_code_r282/ff_r282/tests/test_taskf.cpp
3. D5.3−ff_code_r282/ff_r282/tests/test_stopstartall.cpp
5.3 Optimisation policies

The policies needed for the automatic optimisation of the run-time code via the static or dynamic selection of the best approach for each pattern and each workload is beyond the WP5, which aimed to provide the run-time support mechanism for sensing and steering deployed code at three different stages of application life cycle: compile, launch-time, and run-time. Several recent works attempted to address the problem via machine-learning approaches [8, 9].
Bibliography


