Project no. 288570

PARAPHRASE

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

Implementation of low-level component support
D5.2

Due date of deliverable: 31st March 2013

Start date of project: October 1st, 2011

Type: Deliverable
WP number: WP5
Task number: T5.2

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

Version 1.0 (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>KE</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>

Project co-funded by the European Commission within the Seventh Framework Programme
Contents

1 Introduction .............................................. 5
    1.1 Target platforms .................................. 6

2 C/C++ with FastFlow ..................................... 7
    2.1 Process-components .................................. 7
    2.2 Communication and synchronisation (channels) .......... 9
    2.3 Offloading ........................................... 10
    2.4 Extra-functional features .............................. 10
    2.5 Examples ............................................. 11
    2.6 Compiling and running the tests and examples ......... 12

3 Erlang .................................................. 13
    3.1 Process-components ................................ 13
    3.2 Communication and synchronisation (channels) .......... 13
    3.3 Offloading ........................................... 14
    3.4 Extra-Functional Features .............................. 14

4 Distribution ............................................. 16
    4.1 Software repository .................................. 16
    4.2 Virtual Machine ....................................... 17
    4.3 Inclusion of the D2.4 VM in the D5.2 VM ................ 17
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 three software packages that are available on the project website at http://www.paraphrase-ict.eu/Deliverables/.

- **ParaPhraseVM-v3.tar**: A Linux x86_86 virtual machine (VM) image with D5.2 software installed. The virtual machine can be opened in either Oracle VirtualBox or VMWare players. Due to current VM technology limitations, GPU hardware cannot be accessed from within the VM. Most of the tests designed for GPUs can be run on CPUs using OpenCL Intel run-time support. The VM includes joint material from both D5.2 and D2.4.

- **fastflow-2.0.1.tar.gz**: The source code and instructions of the C/C++ WP5 development framework with tests and examples (see Sec. 2). This package also includes material from D2.4.

- **Erlang-0.3.tar.gz**: The source code and instructions of the Erlang WP5 development framework (see Sect. 3). Note that this package depends on OpenCL. OpenCL cannot currently be used within a Virtualbox VM environment since this system is missing the required fglrx kernel module.
Positioning of Deliverable D5.2

This Deliverable is the second deliverable the work package WP5, Low-Level Virtualisation (Platform-Specific Deployment). 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 (via refactoring methods defined in WP4) the lifting of pattern optimisations at the earliest possible stage (e.g. static and launch-time).

This deliverable (D5.2) reports the achievements of task T5.2 and addresses Objective 2 above. The positioning of this deliverable with respect to other deliverables is shown in Fig. 1. The main input to this deliverable consists of Deliverable D5.1 Hardware Virtualisation API report, that in turn receives inputs from three different and distinct directions:

1. **from the hardware** analysis of the current generation of (and prospective next generation of) hardware architectures with a particular focus on heterogeneous platforms, including distributed platforms equipped with fast networks such as Mellanox Infiniband.

2. **from the hardware and software virtualisation interfaces** defined in WP3, and specified in Deliverable D3.1. Initial progress includes a series of experiments on the HAL offloading mechanisms to allow FastFlow to coordinate GPU-based computations.
3. from the software/patterns deliverables D2.1 (Initial Generic Pattern report), and D6.1 (Project Requirements Analysis) describing basic requirements of software artifacts in the project, performance, performance portability, dynamic adaptivity, orchestration model, features of the Erlang language and typical C++ pattern-based libraries (e.g. FastFlow).

On this basis, D5.2 abstracts the primitive features of different hardware platforms and introduces a low-level programming model suitable as a target for compilation/code generation of dynamically adaptable patterns.
Chapter 1

Introduction

Deliverable D5.2 is a prototype plus a technical report about the implementation of low-level component support. 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 http://paraphrase-ict.eu/. The software is distributed as two versions:

- a tarball with sources and instructions for compiling tests and examples.
- 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 the pure Erlang implementation cannot be executed on the supplied virtual machine due to a bug in the OpenCL bindings interacting with a 64-bit Erlang runtime. This is a known issue which is currently being addressed. The code has therefore been developed for a 32-bit Erlang runtime.

With respect to WP5, the software implements the Hardware Abstraction Layer as described in deliverable D5.1. The ParaPhrase project actually supports two implementations of the HAL:

1. A C/C++ implementation based on the FastFlow library (Chapter 2).
2. An Erlang implementation (Chapter 3).

The HAL realises a low-level concurrent programming model based on a CSP/Actor hybrid model where processes are named and the data paths between processes are clearly enumerated. According to D5.1, there are four main abstractions:

1. Process-component model;
2. Communication and synchronisation (channels);
3. Offloading (exploiting hardware accelerators);
4. Extra-functional attributes and features.

We refer the reader to Deliverable D5.1 for an extensive description of each of these and their characterisation.

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 and PowerPC are also partially supported.
- Linux OS with the gcc compiler. MacOS (>10.5) with the gcc/clang compiler and Windows XP/7 with Visual studio 10 are also partially supported.
- NVidia GPGPUs with OpenCL run-time support. CUDA is partially supported.
- Clusters built using TCP/IP networking. Native support for Infiniband/OFED is ongoing.
- 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

C/C++ with FastFlow

FastFlow is a C++ header-only template library with a layered design. The architecture of FastFlow is shown in Fig. 2.1; WP5 C/C++ HAL consists of an evolution of the two lowest layers of the FastFlow library.

2.1 Process-components

Process-components are implemented by the ff::node class, which provides the abstraction for a thread of control. It is used to encapsulate sequential portions of code (via the virtual svc method) implementing functions to be run concurrently. A node repeatedly pops a message pointer from a channel (input interface), executes the svc method and pushes one or more messages into a channel (output interface). Node input is received via the method parameter and produces an output either via the return statement of the svc method or via ff::ff_node::ff_send_out method. A node can be initialised and finalised by way of the svc_init() and svc_end() methods, respectively. It is the superclass of all process-components nodes. Once created, a ff::node can be run using the run method. The wait method waits for the termination of a node. A node terminates when it receives a NULL pointer in its input interface.

In the shared-memory programming model, the ff::minode and ff::monode subclasses extend ff::node to have either a multiple input or output interface. These are respectively meant to connect many-to-one and one-to-many nodes in a lock-less memory-fence-free fashion. They are called mediator nodes. Typically, mediator nodes are attached (either statically or dynamically) to nodes. Two mediator nodes can also be attached together if and only if one of the two mediator nodes is attached along a single interface. Since they extend ff::node, they inherit the above mentioned methods. In particular, the svc method is used to implement any kind of collective communication and message scheduling. It can be overridden to implement user-defined message management and filtering.

These concepts are shown graphically in Fig. 2.2. Graphs of nodes can be either statically or dynamically built by attaching nodes and mediator nodes. It can
be observed that since the layer is specifically designed to support parallel patterns, not all graphs can be built. However, arbitrarily complex cyclic graphs can be built. We refer to the graph built with the above rules as ff-graph.

In the distributed memory programming model, graphs of ff-graphs can be defined. Each ff-graph is contained in a process (i.e., two ff-graphs do not natively share memory). Each ff-graph can be enriched with special nodes, called ff::dnodes. The ff::dnode class extends ff::node class and provides either input or output interface (or both, ff::dinout) on either network channels or shared-memory channels. On network channels, ff::dnodes exhibit a blocking behaviour. A network channel can be either symmetric (1-to-1) or asymmetric (1-to-N or N-to-1). Asymmetric channels can be configured for different behaviours, such as unicast,
broadcast, multicast, scatter, gather, etc. \texttt{ff::dnode} is a network mediator node that is typically used to connect two or more \texttt{ff-graphs} in a producer-consumer fashion.

### 2.2 Communication and synchronisation (channels)

As mentioned above, channels are meant to connect nodes. They can be created within the class implementing either producer or consumer nodes. Once created, a channel exports a free interface (either output or input), that can be attached to its partners. The assembly mechanism is different for \texttt{ff::nodes} and \texttt{ff::dnodes}, i.e. for shared-memory and network channels, respectively.

Within an \texttt{ff-graph} (i.e. in the \texttt{shared-memory} programming model) \texttt{ff::dnodes} are implemented with Single-Writer-Single-Reader FIFO queues, either bound or unbound. They are designed to manage memory pointers that act as synchronisation tokens whereas the message payload (i.e. referenced data) can be accessed via the shared memory. The default implementation of either bound or unbound queues is based on native FastFlow queues, implemented in the \texttt{SWSR-buffer} and \texttt{SWSR-ubuffer} classes. Both queue implementations are wait-free and memory-fence-free and exhibit state-of-the-art performance.

A channel is created within a node with the \texttt{create_input_buffer} or \texttt{create_output_buffer} methods of the \texttt{ff::node} class. Their free binding can be retrieved with \texttt{get_in_buffer} or \texttt{get_out_buffer} methods of the \texttt{ff::node} class. Eventually, two nodes can be linked using the \texttt{set_input} or \texttt{set_output} methods of the \texttt{ff::node} class. Examples of the usage of these constructs are provided in the \texttt{tests/layer2-tests-HAL} directory. The following fragment of code sketches how to connect two nodes in a pipeline fashion.

```cpp
class N: public ff_node {
    ...

main() {
    N n1, n2;

    n2.create_input_buffer(100);
    n1.set_output(n2.get_in_buffer());

    n1.run();
    n2.run();

    n1.wait();
    n2.wait();
    ...
}
```

Channels in different \texttt{ff-graphs} (i.e. in the \texttt{distributed memory} programming model) are implemented using the ZeroMQ library, which supports TCP/IP message transport. Network channels are created on both producers and consumers via the \texttt{ff::dnode} constructor, which requires the programmer to specify:
• **channel type** of type `CommImpl`, e.g. `zmqOnDemand`, `zmqFromAny`, etc.

• **channel id**, i.e. a unique text label.

• **host address**, i.e. a TCP/IP host:port.

The `ff::dnode` class provide the programmer with two virtual methods for message serialisation and de-serialisation: the `prepare` and `unmarshalling` methods, respectively. Examples of the use of network channels can be found in the `tests/d` directory.

### 2.3 Offloading

Offloading onto hardware accelerators (GPGPUs) is currently implemented via OpenCL. The dialogue with the GPGPU is delegated to `ff::oclnode`, which maintains a persistent binding with a hardware accelerator device. An `ff::oclnode` extends `ff::node` class with the following features:

- **hardware accelerator setup** (`svc_SetUpOclObjects`): create an OpenCL context, create an OpenCL command queue, set OpenCL kernel launch parameters, and create device memory objects.

- **hardware accelerator release** (`svc_releaseOclObjects`): release kernel, command queue, and release device memory objects.

- **kernel execution** (`svc`), typically including H2D memory copy, kernel launch, and D2H memory copy.

One or more `ff::oclnodes` can be bound to a hardware accelerator device. Zero or more HW accelerators can be used. In the former case, kernels are executed in one or more cpu cores. Examples of the usage of these constructs can be found in `examples/spd-denoiser/opencl/` and `tests/ocl`.

### 2.4 Extra-functional features

**System inspection.** A number of functions are provided that can dynamically inspect the system configuration. They include, for example: detection of the number and kind of GPGPUs, the number and kind of CPUs, the CPU frequency, the CPU cache configuration, and process numbering.

**Run-time monitoring.** The `ff::node` class is equipped with a number of pre-defined run-time monitors based on processor hardware counters (currently supported on x86_64 machines). These can be enabled at compile time by defining `TRACE_FASTFLOW`. Information is accounted for each node, such as the number of received tasks (stream items), the average/min/max execution time for the user
code (svc_ticks), the number and time spent on busy waiting for input and output tasks (due to empty or full buffers). Monitor information can be dynamically queried during the run. An example report is:

... Node N1
ID: 1  work-time (ms): 33.595
    n. tasks   : 50
    svc ticks  : 24216510 (min= 72714 max= 17871948)
    n. push lost : 0 (ticks=0)
    n. pop lost : 15604 (ticks=15604000)

Node N2
ID: 2  work-time (ms): 33.68
    n. tasks   : 50
    svc ticks  : 24085984 (min= 65678 max= 17886528)
    n. push lost : 0 (ticks=0)
    n. pop lost : 15598 (ticks=15598000)

... Pinning. HAL performance can significantly depend on node-to-core pinning. While mapping policies are studied in WP3, WP5 provides lower-level mechanisms to pin ff::nodes to cores, either statically or dynamically. The C/C++ HAL implements two main ways to pin a ff::node to a core:

- at ff::node creation time. This feature is supported via a gcc intrinsic operation that make it possible to create the thread implementing a ff::node on a specific core.
- at any time during the run, via the ff_mapThreadToCpu and ff_getMyCpu functions.

2.5 Examples

Several simple tests can be found in the tests/ directory. In particular:

- in tests/tests/layer2-tests-HAL:
  - 11_ff_nodes_graph.cpp: a graph is defined, assembled and run using both layer1 and layer2 features.
  - 12_generic_dag.cpp: a direct acyclic graph is defined, assembled and run using layer2 features.
  - 12_generic_dag_pinning: a direct acyclic graph is defined, assembled and run using layer2 features. Nodes are dynamically pinned and re-pinned to other cores.
  - 12_cyclic_graph.cpp: a cyclic graph is defined, assembled and run using layer2 features.
- `l2_cyclic_graph_trace.cpp`: a cyclic graph is defined, assembled and run using `layer2` features. Run-time statistics are collected.
- `l2_mapping_utils`: system inspection.

- The directory `tests/` contains regression tests for all layers.
- The directory `tests/d` contains regression tests for the distributed version of the HAL.
- The directory `tests/ocl` contains regression tests for GPGPUs/OpenCL.

### 2.6 Compiling and running the tests and examples

A description of how to compile and run the tests and examples can be found in the `README` and `BUILD.ME` files.
Chapter 3

Erlang

3.1 Process-components

Erlang includes the notion of a process, whose implementation includes the primitives `send` (also written as `(!)`), `receive` and `spawn`. For the purposes of WP5, process components map 1:1 to Erlang processes. Each process is scheduled concurrently by the Erlang scheduler. The data parallelism which occurs by executing code on an OpenCL device operates at a finer level of granularity than normal Erlang processes, since a single data element may come from the composition of a numeric array composed of bytes, integers, floats and doubles. This implies that there are two views of process components for differing levels of granularity. References to the process components on an OpenCL device are intended to refer to an individual instance of a running OpenCL kernel which has its own control flow and where the mechanisms for handling parallelism are provided by OpenCL's `NDRange` operation. This distinction is important since it draws the line between the Erlang runtime notion of a process and that which executes upon an OpenCL device such as a GPU. An operation may in this context be applied to many data elements at the same time within a single Erlang process.

3.2 Communication and synchronisation (channels)

There are two relevant types of channels. These have different responsibilities and distinguish between executing code in a distributed environment versus executing code locally.

TCP/IP Channels. Communication between Erlang nodes uses TCP/IP. Each Erlang node that is responsible for executing OpenCL code contains a server which listens for incoming data on a specified port. It is assumed that the particular OpenCL kernel that is to be used is configured via the application configuration, allowing multiple distinct kernels to reside within the same codebase and interchanged based on the requirements. The node can be seen as a black box which
streams input from a client or server, processes the data via OpenCL and streams
the results back to the client/server.

**OpenCL Channels.** While loading data into OpenCL device memories, chan-
nels may be created which copy data to/from the device based on certain events.
For example, consider the case where two matrices are to be multiplied in a spa-
tially parallel implementation. The two matrices must first be loaded into device
memory using OpenCL channels, and then operated on by the kernel. The required
number of OpenCL channels can be created. This allows multiple OpenCL kernels
to be executed in succession on a single node, and means that dataflow patterns
can be made to branch based on the results that have been obtained from previous
kernel executions. This capability allows for dataflow patterns to be expressed in
terms of communicating OpenCL kernels.

### 3.3 Offloading

Offloading involved the assignment of work to distinct units of hardware. In the
context of WP5, offloading does not include the ability to decide where units of
work should execute – such load balancing is handled by WP3 mapping mech-
anism. Instead, the focus is simply on the ability to offload. Each OpenCL sup-
ported device which is attached to the machine may be offloaded to. In situations
where one node is responsible for multiple devices, these may also be simultane-
ously offloaded to. The work which is to be executed on an OpenCL device may
be offloaded by streaming the data to the node responsible for the device.

### 3.4 Extra-Functional Features

**Hardware Accelerators.** OpenCL was used as the hardware accelerator API
since it includes all of the functionality that is necessary to execute code on de-
vices such as the GPU. CUDA is also a viable possibility and may be utilised in
the future depending on project requirements.

**Monitoring.** Erlang includes much inbuilt functionality for monitoring the sta-
tus of processes within a node, and of whole nodes in a distributed setting. The
framework that has been developed in WP5 does not make use of individual pro-
cess monitoring since OpenCL execution occurs only within one process. Where
monitoring can be of great benefit however is between node boundaries. If one
worker node goes down or otherwise becomes unavailable, this can be addressed
by an external load balancing mechanism.

**Streaming.** The base framework is designed in such a way that data should be
streamed across the network between nodes. For this framework, the communi-
cation is designed for use within a distributed memory environment rather than a shared memory environment. It would be feasible in an augmented version of the framework to also include the ability to stream across in-memory channels, such as can be seen in the FastFlow framework. This would necessitate changes to the Erlang distribution protocol to allow for a new shared-memory implementation.
Chapter 4

Distribution

All the software mentioned in this deliverable may be accessed through the Parallel Computing group at University of Torino web site (http://alpha.di.unito.it/?page_id=110) and in the ParaPhrase web site at (http://www.paraphrase-ict.eu/Deliverables/). The following sections shortly outline the web site repository structure and the virtual machine that is included as part of the software deliverable.

4.1 Software repository

The software release associated with this deliverable includes:

- a tarball with the version of FastFlow implementing the ParaPhrase C++ HAL (fastflow-2.0.1.tar.gz). Build instructions, libraries dependencies for tests and examples are described in the BUILD.ME text file within the tarball. Supported Platform/OS combinations and additional notes are described in the README text file within the tarball.

- a tarball with the version of Erlang implementing the ParaPhrase Erlang HAL (Erlang-0.3.tar.gz).

- a virtual machine with both frameworks installed (described in Sec. 4.2 below). Installation instructions are described in the README.md text file within the tarball.

The two tarballs may be accessed also on

- the ParaPhrase SVN repository (access restricted to project beneficiaries): software bundles in the directory WPfolders/WP5folder/HAL/. In the directory, two tarballs are present: fastflow-2.0.1.tar.gz and Erlang-0.3.tar.gz hosting the software bundles described in Chap. 2 and 3, respectively.
4.2 Virtual Machine

For convenience, a virtual machine (VM) image for the distribution has been created with VirtualBox and exported in the OVF format. This format may be imported by both VMware and VirtualBox virtual machine players, either using the command line interface or the virtual machine player GUI. The image is based on a 64-bit Linux Debian 6 distribution (X86-64 Debian 6). Once loaded, the machine may be used with the following login details:

```
login: paraphrase
passwd: paraphrase
```

The virtual machine image has been prepared so that 4 virtual CPU cores are used. If the VM user’s host machine has less cores, then a more convenient number of CPU cores ($k \leq \text{available cores}$) may be configured when the OVF virtual machine image is imported. One key limitation of the virtual machine is that it does not support GPUs. This is because neither VirtualBox nor WMplayer provide GPU virtualisation. The virtual machine may, however, be used to test all features and software on CPUs. In particular, the GPU integration of FastFlow relies on OpenCL and may therefore be run using CPU cores rather than GPUs.

4.3 Inclusion of the D2.4 VM in the D5.2 VM

The Virtual machine released in this deliverable (D5.2) has been cooperatively built together with D2.4 from WP2, by extending it with WP5-specific examples. This means that the VM released with D5.2 fully includes the material released in the VM associated with D2.4. The same relation holds for the tarball `fastflow-2.0.1.tar.gz`.

This relation does not hold for the tarball `Erlang-0.3.tar.gz`, which is specific of the D5.2.