Project no. 288570

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

Heterogeneous Implementation of Initial Generic Patterns

D2.4

Due date of deliverable: 31st March 2013

Start date of project: October 1st, 2011
Type: Deliverable
WP number: WP2
Task number: T2.3

Responsible institution: UNIPI
Editor and editor’s address: Sonia Campa, UNIPI

Version 1.0 (Revision: 2044)

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PU Public</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PP Restricted to other programme participants</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>KE Restricted to a group specified by the consortium</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CO Confidential, only for members of the consortium</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Executive Summary

Deliverable D2.1 [6] introduced the initial set of generic patterns for the ParaPhrase project. The patterns in the initial set can be used to implement parallel applications according to the structured parallel programming methodology that is typical of the design patterns and/or algorithmic skeletons approaches. Deliverable D2.2 [7] described the implementation of the initial set of patterns targeting homogeneous multicore architectures. This deliverable describes the implementation of the initial generic pattern set on heterogeneous architectures, that is on architectures containing both multicore CPUs and General Purpose GPUs (GPGPUs). We focus on the implementation of the data parallel patterns from the initial pattern set (map and reduce), since the task parallel patterns that were described there (pipeline and farm) do not fit the data parallel execution model that is typical of GPGPUs.

This deliverable describes two different implementations of the initial data parallel patterns. The first implementation provides data parallel skeletons implementing the initial data parallel pattern set on GPUs. These are provided as suitable FastFlow abstractions. The second implementation provides similar skeletons as Erlang functions that can be seamlessly used in any Erlang program. In both cases, the GPU implementation of the data parallel skeletons runs on top of OpenCL, although some CUDA support is also provided.
The position of this deliverable with respect to the Paraphrase project as a whole is shown above. It represents the evolution of Deliverables D2.1 and D2.2 to cover heterogeneous (CPU/GPU) platforms. By the end of the project, it will be used to provide input to the implementations of the final set of patterns on both homogeneous platforms (Deliverable D2.8) and heterogeneous platforms (Deliverable D2.9).
Contents

1 Introduction ........................................ 4

2 Targeting Initial Patterns to (GP)GPU systems using C++ (FastFlow) ................. 6
  2.1 System requirements .......................... 6
  2.1.1 Downloading and installing FastFlow ......... 6
  2.2 API ............................................. 7
  2.2.1 Using the OpenCL implementation .......... 8
  2.2.2 Using the CUDA implementation .......... 9
  2.3 Example ....................................... 9
  2.4 Contents of the distribution archive ......... 11

3 Targeting Initial Patterns to (GP)GPU systems using Erlang ............................ 12
  3.1 System requirements .......................... 12
  3.2 API ............................................. 13
  3.2.1 API usage .................................... 15
  3.3 Example ....................................... 16
  3.4 Contents of the distribution archive ......... 17

4 Distribution ........................................ 18
  4.1 Software repository ......................... 18
  4.2 Virtual Machine .............................. 19
Chapter 1

Introduction

The initial set of generic patterns (introduced in [6] and summarized in Table 1.1) included two main kinds of pattern:

**Data parallel patterns** such as the *map* and *reduce* patterns; and

**Stream parallel patterns** such as the *pipeline* and *farm* patterns.

The implementation of the patterns included in the initial generic pattern set on multicore CPUs has already been described in Deliverable D2.2 [7]. The current deliverable describes the implementation of the data parallel patterns on heterogeneous architectures, comprising combinations of CPUs and (GP)GPUs. Since stream parallelism cannot be usefully exploited on GPGPUs, we do not consider the implementation of stream parallel patterns on GPGPUs

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>Wrapper for the sequential user code</td>
<td>Singleton</td>
</tr>
<tr>
<td>Pipeline</td>
<td>Chain of stages (patterns) in which the output of each stage represents the input of the next stage</td>
<td>Stream</td>
</tr>
<tr>
<td>Farm</td>
<td>Replication of a set of identical computing elements (patterns) with customizable dispatching and collecting strategy of the input/output data items/results</td>
<td>Stream</td>
</tr>
<tr>
<td>Map</td>
<td>Replication of a set of identical computing elements (Seq), each computing a partition of an input data collection</td>
<td>Collection</td>
</tr>
<tr>
<td>Reduce</td>
<td>Reduction to a single value applied over an input collection</td>
<td>Collection</td>
</tr>
</tbody>
</table>

Table 1.1: Initial generic patterns set as defined in Deliverable D2.1
As planned, two distinct implementations of the data parallel patterns have been produced for GPGPUs, covering both the functional and the imperative/object-oriented programming paradigms:

1. in FastFlow, the C/C++ pattern framework; and
2. in the functional language Erlang.

This will allow direct comparisons to be made between the two paradigms, as well as ensuring that generic, language-independent solutions are produced for the ParaPhrase refactoring, patterns and other tools and technologies. In both cases, the data parallel patterns have been implemented using proper algorithmic skeletons implemented on top of OpenCL. C/C++/FastFlow and Erlang programmers may declare and use these skeletons using a compact, simple and complete set of macros that are designed to simplify parallel programming effort, and provide straightforward targets for the ParaPhrase refactoring tools.

In the case of the FastFlow programming environment, the macros exploit a class library (developed within the ParaPhrase project), built on top of OpenCL. This class library allows sub-tasks of the data parallel computations to be mapped to either CPU or GPGPU cores, as required. Minimal, proof-of-concept support for CUDA has also been included. This allows FastFlow programmers to use macros to directly map computations to nVidia GPGPUs via CUDA rather than OpenCL. In the case of the Erlang programming environment, the macros exploit SKEPU (http://www.ida.liu.se/~chrke/skepu/) to target either GPGPUs (through OpenCL or CUDA) or CPUs (through OpenMP). SKEPU is an open source library developed at Linkoping University as part of the EU FP7 PEPPHER project.

The software release for this deliverable comprises:

1. a tarball with the code relative to the FastFlow/C++ implementation,
2. a tarball with the code relative to the Erlang implementation, and
3. a virtual machine image (64 bits) with a precompiled, working version of both the FastFlow/C++ and the Erlang GPGPU data parallel patterns.

It has been made publicly accessible through the ParaPhrase web site (http://www.paraphrase-ict.eu/Deliverables/d2-4-prototype-sources).
Chapter 2

Targeting Initial Patterns to (GP)GPU systems using C++ (FastFlow)

The C++ (FastFlow) implementation provides map and reduce skeletons over an array of generic elements. The map skeleton has been implemented using both OpenCL and CUDA frameworks, whereas the reduce skeleton has been implemented only using OpenCL.

2.1 System requirements

The system requirements are:

- Linux (Kernel version 2.6.18 or higher, GNU C/C++ compiler version 4.2 or higher, 64bit Ubuntu or CentOS distributions)
- FastFlow [5] (version 2.0.1 or higher)
- CUDA (driver 4.1 or higher)
- OpenCL (either version 1.0 or higher)

The software has been tested on different architectures with Intel Nehalem or Sandy Bridge (E5520, E5-2650) or AMD Magny Cours (Opteron 6176) processors and different nVidia GPUs including C2050 (Tesla), GTX 660 (Kepler) or GTX 285 GPUs.

2.1.1 Downloading and installing FastFlow

FastFlow can be downloaded from the Sourceforge SVN repository using the following command:
or from the ParaPhrase web site ([http://www.paraphrase-ict.eu/Deliverables/d2-4-prototype-sources](http://www.paraphrase-ict.eu/Deliverables/d2-4-prototype-sources)) by downloading the fastflow-2.0.1.tar.gz tarball and uncompressing the files with a `tar xvf fastflow-2.0.1.tar.gz` command. The instructions for compiling all tests and examples provided with the package are contained in the `BUILD.ME` file in the top level directory.

The version on the web site is the one actually associated to and described in this deliverable. The version on sourceforge may change with new releases in the near future. To test the features described in this paper is therefore highly recommended to use the tarball uploaded on the ParaPhrase web site, that contains all the files mentioned in the following sections.

### 2.2 API

This section shows which API/classes are needed to implement a `map` and a `reduce` skeleton using OpenCL and CUDA in a C++ program. The API provides the C++ programmers with two distinct set of macros:

- macros used to define functions as needed to implement `map` and `reduce` skeleton, and
- macros used to instantiate the skeleton implementation.

All necessary macros are defined in the `<ff/map.hpp>` FastFlow file. The functions used in the `map` and `reduce` skeletons must be written in C/C++ using the following interface:

```cpp
FFMAPFUNC(fname, basictype, param, code)
FFREDUCEFUNC(fname, basictype, param1, param2, code)
```

where:

- `fname` is the function name;
- `basictype` is the basic type of the input/output array;
- `paramX` is the name of the formal parameter(s) of the function `fname`;
- `code` is the function code written in C/C++.

As an example, a function computing the square of a floating point number may be defined as:

```cpp
FFMAPFUNC(square, float, x, return(x*x) );
```
a function computing the $\text{reduce}((\oplus, \langle x, y \rangle)$ of a pair of floating point numbers is defined as:

\[ \text{FFREDUCEFUNC}(\oplus, \text{float}, x, y, \text{return}(x+y)) ; \]

The interface to instantiate (and to delete) the map and reduce skeletons is:

\[ \text{NEWMAP} \text{(name, task\_type, mapf, V, sz)} \text{ defines a map running the function } \text{mapf} \text{ on the input array } V \text{ of size } sz, \text{ task\_type is the input and output type of the map;} \]

\[ \text{NEWMAPONSTREAM} \text{(task\_type, mapf)} \text{ defines a map running the function } \text{mapf} \text{ on the input task of type } \text{task\_type} \text{ received from the input stream;} \]

\[ \text{NEWREDUCE} \text{(name, task\_type, reducef, V, sz)} \text{ defines a reduce running the function } \text{reducef} \text{ on the input array } V \text{ of size } sz, \text{ task\_type is the input and output type of the reduce;} \]

\[ \text{NEWREDUCEONSTREAM} \text{(task\_type, reducef)} \text{ defines a reduce running the function } \text{reducef} \text{ on the input task of type } \text{task\_type} \text{ received from the input stream;} \]

\[ \text{DELETEMAP} \text{(name)} \text{ destroy the map instance with the name } \text{name;} \]

\[ \text{DELETEREDUCE} \text{(name)} \text{ destroy the reduce instance with the name } \text{name.} \]

where

- \text{name} is the map/reduce name;
- \text{task\_type} is the input/output task type which must be a subclass of the interface class \text{baseTask} as defined in the \text{<ff/map.hpp>} file;
- \text{mapf/reducef} is the map/reduce function name;
- \text{V} is the input (output) array of size \text{sz};
- \text{sz} is the size of the input (output) array.

### 2.2.1 Using the OpenCL implementation

In order to use the OpenCL implementations of the map and reduce skeletons, it is necessary to define \text{FF\_OCL}. This can be done either using the compiler flag \text{-DFF\_OCL}, or with an explicit definition as shown below:

\[ \text{#if !defined(FF\_OCL)} \]
\[ \text{#define \text{FF\_OCL}} \]
\[ \text{#endif} \]
\[ \text{#include \text{<ff/map.hpp}>} \]
2.2.2 Using the CUDA implementation

Similarly, to use the CUDA implementation of the map skeleton, it is necessary to define `FF_CUDA`:

```c
#if !defined(FF_CUDA)
#define FF_CUDA
#endif
#include <ff/map.hpp>
```

2.3 Example

Here we outline the core code that is necessary to implement a simple `pipeline(seq,farm(map),seq)` skeleton using FastFlow, where the map skeleton possibly runs on the GPGPU using OpenCL. The complete example can be found in the `<fastflow-home-dir>/tests/ocl/pipeFarmMap.cpp` file.

```c
// define the implementation back-end to use
#if !defined(FF_OCL)
#define FF_OCL
#endif
#include <ff/map.hpp>
#include <ff/farm.hpp>
#include <ff/pipeline.hpp>

// MAP function definition: just a simple increment
FFMAPFUNC(mapf, float, elem, return (elem+1.0) );

// this is the input/output task type for the MAP
// it must be a subclass of the baseTask interface
template<typename T>
class oclTask: public baseTask {
public:
    typedef T base_type;
    oclTask(){};
    oclTask(T *task, size_t s):baseTask(task),s(s) {}
    void setTask(void* t) {
        task=((oclTask<T>*)t)->getInPtr();
        s =((oclTask<T>*)t)->size();
    }
    size_t size() const { return s;}
    size_t bytesize() const { return s*sizeof(base_type); }
protected:
    size_t s;
```
// ... main definition and other code

// pipeline definition
ff_pipeline pipe;

// first sequential stage of the pipeline
// which produces the stream of vectors
pipe.add_stage(new ArrayGenerator(size, streamlen));

// farm definition with default collector
ff_farm<> farm;
farm.add_collector(NULL);

// create the farm’s workers
std::vector<ff_node *> w;
for(int i=0;i<nworkers;++i)
  // each worker is a MAP
  w.push_back(NEWMAPONSTREAM(oclTask<float>, mapf));

// adding the MAP workers to the farm
farm.add_workers(w);

// 2nd stage of the pipeline
pipe.add_stage(&farm);

// last stage of the pipeline which gathers all
// vectors produced by the farm
pipe.add_stage(new ArrayGatherer);

// run the pipeline and wait for termination
pipe.run_and_wait_end();

To compile the program, type make pipeFarmMap inside the ocl folder. An excerpt of the output produced by the execution of the program when using a vector size of 20, a stream length of 100 and 4 workers for the farm, on a multi-core machine with 1 GPU, is shown below:

$ ./pipeFarmMap 20 100 4
MEMSIZE 2147287040
THRESHOLD 536870912
max application number for device id=0x7fc87c015bd0 is 3
MEMSIZE 33638199296
THRESHOLD 536870912
max application number for device id=0x7fc87c025970 is 62
selected GPU id =0
selected GPU id =0
selected GPU id =0
selected CPU id=1
3.00 4.00 5.00 6.00 7.00 8.00 9.00 10.00 11.00 12.00 13.00 ...
7.00 8.00 9.00 10.00 11.00 12.00 13.00 14.00 15.00 16.00 ...
11.00 12.00 13.00 14.00 15.00 16.00 17.00 18.00 19.00 20.00 ...

2.4 Contents of the distribution archive

The distribution archive fastflow-2.0.1.tgz contains the FastFlow run-time and a set of examples and tests. The OpenCL and CUDA example tests can be found in the tests/ocl and tests/cuda folders respectively. Each folder contains a Makefile for building the tests.
Chapter 3

Targeting Initial Patterns to (GP)GPU systems using Erlang

The corresponding Erlang implementation provides several skeletons:

1. a map skeleton, implementing a map of a unary function over a list of (floating point) numbers,
2. a mapzip skeleton, implementing a map of a binary function over the pairs that are obtained by zipping together two input lists of the same length;
3. a reduce skeleton, implementing a reduce of a binary (associative and commutative) function over a list of floating point numbers, and
4. a mapreduce skeleton, implementing a mapreduce over a list of floating point numbers. The list is first processed as in a map applying to all elements a unary function and then the resulting list is “summed up” using a binary function to produce the final result.

The map and reduce skeletons implement the high level patterns summarised in Table 1.1. The mapzip and mapreduce skeletons provide specialised versions of these patterns that are useful in some contexts. A mapzip(f, L1, L2) computes the list whose $i$-th element is the result of computation of the function $f$ over the two $i$-th elements of list $L1$ and $L2$. A mapreduce(f,⊕,L) computes a reduce(⊕, (map(f,L))). This mapreduce is not identical to the Google mapreduce pattern, but rather composes a single reduce operation with a single map.

3.1 System requirements

The provided implementation uses SKEPU to target GPUs. It should run on any Linux system that provides:

- Erlang [1–3] (version Erlang R15B01 (erts-5.9.1) or higher),
How to install

**Erlang**
- Standard packages exist in all distributions
  - e.g. `apt-get install erlang` (Ubuntu Linux)

**SKEPU**
- [http://www.ida.liu.se/~usmda/skepu/](http://www.ida.liu.se/~usmda/skepu/)

**CUDA**

**OpenCl**

**Perl**
- Comes standard with Linux distributions

**OpenMP**
- Comes with standard GNU c++ compiler (version 4.0 or higher)

<table>
<thead>
<tr>
<th><strong>Tool</strong></th>
<th><strong>How to install</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Erlang</td>
<td>Standard packages exist in all distributions</td>
</tr>
<tr>
<td></td>
<td>e.g. <code>apt-get install erlang</code> (Ubuntu Linux)</td>
</tr>
<tr>
<td>SKEPU</td>
<td><a href="http://www.ida.liu.se/~usmda/skepu/">http://www.ida.liu.se/~usmda/skepu/</a></td>
</tr>
<tr>
<td>Perl</td>
<td>Comes standard with Linux distributions</td>
</tr>
<tr>
<td>OpenMP</td>
<td>Comes with standard GNU c++ compiler (version 4.0 or higher)</td>
</tr>
</tbody>
</table>

Table 3.1: Required software tools.

- **SKEPU** [4] (version 1.0),
- **CUDA** (driver 4.1 or higher),
- **OpenCL** (version 1.0 or higher),
- **Perl** (version 5 or higher, Perl is needed to run the script processing the `user.h` macros and generating the `mapper.erl` wrapper),
- **OpenMP** (as provided by g++ version 4.1.2 or higher, OpenMP is needed only in case we want to test the skeleton implementation using only CPU cores and not using OpenCL to target the CPU cores).

Most of these packages may be installed on a standard Linux distribution using the provided .deb or .rpm packages or by downloading a specific tarball from the package website. Table 3.1 shows the standard web location/Linux packages relative to the needed tools. The only non-standard package is SKEPU. The installation of this package simply consists of unpacking a tar.gz file and including the unpacked include directory, using a `-I <pathtoskepuinclude>/include` compiler flag.

SKEPU has been adopted to factorize over the effort needed to address different kind of GPUs. SKEPU actually targets either GPUs and CPUs through either OpenCL (GPU and CPU), CUDA (GPU only), or OpenMP (CPU only). The different hardware and middleware systems are targeted by simply defining proper macros at the very beginning of the SKEPU code files (see documentation at [http://www.ida.liu.se/~usmda/skepu/doc/html_v1.0/](http://www.ida.liu.se/~usmda/skepu/doc/html_v1.0/))

### 3.2 API

The API provides the Erlang programmers with two distinct set of macros:

- some macros, inherited from SKEPU, supporting the definition of unary and binary functions as needed to implement `map` or `reduce` worker functions, and

---

13
• some macros, specific to the ParaPhrase skeleton implementation, supporting the Erlang programmer in the definition of the skeletons supported by the library. This second set of macros, all with names prefixed by ERL_GPU_, accept two or three parameters: the name(s) of the parameter functions (one in case of map, mapzip and reduce, and two in case of mapreduce) and the name of the skeleton.

The macros inherited from SKEPU include:

**UNARY_FUNC**(name, type, param, body) defines a unary function named name accepting a parameter param of type type and process it through the code in body to get a result of type type. As an example, a function computing the square of a float may be written as:

```erlang
UNARY_FUNC(sq, float, x, return(x*x); )
```

**BINARY_FUNC**(name, type, param1, param2, body) defines a binary function named name accepting two parameters param1 of and param2 of type type and process them through the code in body to get a result of type type. As an example, the function summing two floats may be defined as:

```erlang
BINARY_FUNC(sum, float, x, y, return(x+y); )
```

The original macros used to define heterogeneous skeletons are instead:

**ERL_GPU_MAP**(fname, skname) defines a skeleton named skname computing the map of the function fname defined as a UNARY_FUNC over a list of floating point numbers.1 As an example, a map computing the list of the squares of the input list items may be defined as:

```erlang
ERL_GPU_MAP(sq, mapsquare)
```

In common with the other macros described in this section, this macro must be defined in the file user.h. The newly defined skeleton (mapsquare in the case above) will then become accessible to Erlang code as a normal mapsquare/1 function, accepting a list as a parameter and returning the list of the squares, computed on the GPU.

**ERL_GPU_REDUCE**(fname, skname) defines a skeleton named skname computing the reduce of the function fname defined as a BINARY_FUNC over a list of floating point numbers.

1we restricted all the skeletons to work on floating point lists only. Differently typed skeletons may be easily derived from the ones processing the floating point numbers
**ERL_GPU_MAPZIP***(fname, skname)*** defines a skeleton named skname computing the *map* of the function *fname* defined as a *BINARY_FUNC* over a list of pairs of floating point numbers derived by zipping the two input lists together. **ERL_GPU_MAPZIP**(sum, skzips) defines an Erlang skeleton function such that

```
skzips([1.0,2.0],[3.0, 4.0]).
```

computes on the GPU and returns

```
[4.0, 6.0]
```

**ERL_GPU_MAPREDUCE***(fname, oplusname, skname)*** defines a mapreduce skeleton named skname computing first the map(fname) and then the reduce(oplusname) on the input list.

### 3.2.1 API usage

To use the API, the Erlang programmer should:

1. edit the *user.h* file to include the proper *UNARY_FUNC* and *BINARY_FUNC* macros as well as all the required **ERL_GPU_xxx** macros.
2. make sure that the file *mapper.cu* is compiled with either

   ```
   #define SKEPU_CUDA
   ```

   or

   ```
   #define SKEPU_OPENCL
   ```

   The Makefile in the distribution, already provides proper **make ocl** and **make cuda** targets defining the SKEPU_xxx macros with a `-D` flag and linking the appropriate libraries for OpenCL and CUDA, respectively.
3. prepare the NIF shared library by issuing a

   ```
   make cuda
   ```

   command (or alternatively, a **make ocl** command, to use OpenCL on GPUs, or a **make cpu**, to target CPU cores through OpenMP). The command also generates the *mapper.erl* file, hosting the actual Erlang code that is needed to make available the skeletons defined in the *user.h* file in the usual Erlang programming environment.
4. start the Erlang interpreter.

5. compile the library NIF interface by issuing:
   
   ```
   c(mapper).
   ```

At this point, the Erlang interpreter may be used to call any of the skeletons targeting the GPU.

### 3.3 Example

Assume that we want to compute simple map(square) or reduce(sum) operation over lists using the GPU. We first edit the user.h file and add the following lines:

```
UNARY_FUNC(sq, float, x, return(x*x));
BINARY_FUNC(sum, float, x, y, return(x+y));
```

```
ERL_GPU_MAP(sq,mymapsq)
ERL_GPU_REDUCE(sum,myreducesum)
```

We then compile the shared NIF library:

```
marcod@titanic:~/erlang_skeletons$ make cuda
./skprep.pl
nvcc mapper.cu -I /usr/local/lib/erlang/erts-5.9.1/include/
   -I include -w -Xcompiler -fpic -Xcompiler -shared -o mapper.so
ptxas /tmp/tmpxft_00006150_00000000-2_mapper.ptx, line 89;
   warning : Double is not supported. Demoting to float
marcod@titanic:~/erlang_skeletons$
```

Finally, we can start the Erlang interpreter and use the skeletons that have been defined:

```
marcod@titanic:~/erlang_skeletons$ erl
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:24:24]
[async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.9.1 (abort with ^G)
1> c(mapper).
   {ok,mapper}
2> L = [1.0, 2.0, 3.0, 4.0].
   [1.0,2.0,3.0,4.0]
3> mapper:mymapsq(L).
   [1.0,4.0,9.0,16.0]
4> mapper:myreducesum(L).
   10.0
5>
```
3.4 Contents of the distribution archive

The distribution archive erlang-gpu-skeletons.tgz unpacks the following material into the current directory:

- Makefile, needed to compile the skeletons.
- mapper.cu and mapper.cpp, hosting the skeleton implementation targeting GPU and CPU, respectively. These are completely managed through the Makefile.
- Skepu, a directory with the files necessary to compile SKEPU programs.
- skprep.pl a PERL script processing the macros defined in the user.h file and producing the files needed to compile, namely NIF_bindings.h and mapper.erl. This file is also completely managed through the Makefile.
- user.h a sample macro file.
Chapter 4

Distribution

All the software mentioned in this deliverable may be accessed through the ParaPhrase web site at (http://www.paraphrase-ict.eu/Deliverables/d2-4-prototype-sources). 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 with the GPU implementation of the initial skeleton set,
- a tarball with the Erlang version of the GPU implementation of the initial skeleton set, and
- a virtual machine with the full environment providing GPU implementations of the initial skeleton set (described in Sec. 4.2 below).

These three items are available for download on

- the ParaPhrase web site (http://www.paraphrase-ict.eu/Deliverables/d2-4-prototype-sources see Fig. 4.1)

The two tarballs may be accessed also on

- the ParaPhrase SVN repository (access restricted to project beneficiaries): software bundles in the directory WPfolders/WP2folder. In the directory, two subdirectories are present: FastFlow and erlang_skeletons hosting the software bundles described in Chap. 2 and 3.
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 VMware Player provide GPU virtualisation. The virtual machine may, however, be used to test all features and software on CPUs. In particular, the GPU implementation of initial pattern set in FastFlow relies on OpenCL and may therefore be run using CPU cores rather than GPUs.
Bibliography


