Project no. 288570

PARA\textsc{Phrase}

Strategic Research Partnership (STREP)
\textsc{Parallel Patterns for Adaptive Heterogeneous Multicore Systems}

Final Homogeneous Implementation
\textbf{D2.8}

Due date of deliverable: Sept 30\textsuperscript{th}, 2014
Actual submission date: Dec 31\textsuperscript{st}, 2014

\textit{Start date of project: 1\textsuperscript{st} October 2011}

\textit{Type: Deliverable}
\textit{WP number: WP2}
\textit{Task number: T2.3}

\textit{Responsible institution: UNIPI}

\textit{Editor \& and editor’s address: Sonia Campa and Massimo Torquati, UNIPI}

Version 1.0 / Last edited by S. Campa / XXXX

\begin{tabular}{|l|l|}
\hline
\textbf{Dissemination Level} & \textbf{Project co-funded by the European Commission within the Seventh Framework Programme} \\
\hline
PU & Public \\
PP & Restricted to other programme participants (including the Commission Services) \\
RE & Restricted to a group specified by the consortium (including the Commission Services) \\
CO & Confidential, only for members of the consortium (including the Commission Services) \\
\hline
\end{tabular}
Executive summary

The ParaPhrase deliverable D2.8 is a prototype deliverable, and this document is the companion document describing the main features of the prototype relative to the final implementation of the ParaPhrase patterns targeting homogeneous (i.e. CPU only) architectures.

The document is divided into two main parts: the first one covering the implementation of patterns within FastFlow, the second one covering the implementation of the patterns in Erlang.

Each part of the document details i) how to obtain and install the code, ii) what patterns are implemented in the prototype and iii) where the relative documentation may be accessed.

The position of this deliverable is shown in the Figure below.
# Contents

1 Introduction 3  
2 FastFlow patterns 4  
  2.1 Installation ......................................................... 4  
  2.2 Pattern list .......................................................... 4  
  2.3 Differences w.r.t. previous releases ............................ 5  
  2.4 Parallel for .......................................................... 5  
      2.4.1 The FastFlow ParallelFor pattern ......................... 5  
      2.4.2 The FastFlow ParallelForReduce and ParallelForPipeReduce patterns .......................... 8  
  2.5 Access to documentation ........................................... 10  
3 Erlang patterns 13  
  3.1 Installation .......................................................... 13  
  3.2 Pattern list .......................................................... 13  
  3.3 Differences w.r.t. previous releases ............................ 14  
  3.4 Access to documentation ........................................... 15
Chapter 1

Introduction

This document is the document pairing the software release of the final implementation of the *ParaPhrase* pattern set targeting homogenous architectures\(^1\). Within the document we synthetically describe the contents of this software release.

*ParaPhrase* activities have been designed to deal with two different programming environments: one based on C++/FastFlow and the other one based on Erlang. The software release relative to this deliverable and, as a consequence, this document reflect this dichotomy. The document is structured in two main parts. The first one describes the contents of the C++/FastFlow release (Chap. 2). The second one describes the contents of the Erlang based release (Chap. 3). In both cases, four different aspects will be considered and discussed:

i) how the software may be obtained and installed for subsequent usage

ii) which are the supported patterns, among those discussed in *ParaPhrase* deliverable D2.5 [3]

iii) which are the relevant changes with respect to the previous “initial” release of the *ParaPhrase* pattern set implementation targeting homogeneous architectures, and

iv) where the documentation relative to the software may be found and accessed.

\(^1\) CPU only hardware, not including/targeting GPUs
Chapter 2

FastFlow patterns

This part of the document collects all the information relative to the final release of the FastFlow based homogenous implementation of the ParaPhrase pattern set as described in D2.5 [3].

2.1 Installation

The final implementation of the FastFlow based ParaPhrase pattern implementation can be downloaded from the ParaPhrase web site at:

www.paraphrase-ict.eu/Deliverables

Being a “header-only” implementation, FastFlow is provided as a set of source files that must be properly included in the user source files. In order to obtain the expected performances, the user sources files, including the FastFlow header files and exploiting ParaPhrase patterns must be used turning compiler optimizations on (e.g. using a -O3 flag when using the GNU compiler).

The source tree includes two subdirectories including applications/kernels/synthetic code using the ParaPhrase patterns.

The ParaPhrase pattern set is provided under the FastFlow LGPL open source license.

2.2 Pattern list

The final implementation of the ParaPhrase pattern set in FastFlow includes the following patterns (see D2.5 [3], D2.6 [4] and D2.7 [5] for description of these patterns):

Core/generic patterns Pipeline, Farm, Map, Reduce, with the possibility to implement iterative computation through the feedback attribute

High level patterns Divide and conquer, Search, Sort, Pool evolution and Workflow graph interpreter patterns
**Domain-specific patterns**  Stencil, instances of Pool and Work-flow interpreter patterns

2.3 **Differences w.r.t. previous releases**

The final release of the ParaPhrase pattern set in FastFlow targeting homogeneous architectures mainly differs from the initial release in three different respects:

- small bug fixes have been introduced in the code to get rid of errors and malfunctioning detected while using the initial pattern implementation (not reported in this deliverable)

- part of the macros used to introduce high level syntax versions of the patterns in the initial implementation have been replaced by different implementation of the same abstractions exploiting new C++11 features (see Sec. ??)

- the PARFOR pattern has been introduced as a syntactic, user friendly substitution of the map pattern (see Sec. 2.4).

2.4 **Parallel for**

A sequential iterative kernel with independent iterations is also known as a *parallel loop*. A map computation is a particular case of the parallel loop. Parallel loops may be clearly parallelized by using the map or farm skeletons, but this typically requires a substantial re-factoring of the original loop code with the possibility to introduce bugs and not preserving *sequential equivalence*. For this reason, in the FastFlow framework there is a set of data parallel patterns implemented on top of the basic FastFlow skeletons to ease the implementation of parallel loops.

2.4.1 **The FastFlow ParallelFor pattern**

The FastFlow ParallelFor pattern can be used to parallelize loops having independent iterations. The class interface of the ParallelFor pattern is defined in the header file: parallel_for.hpp. The constructor accepts three optional arguments:

\[
\text{ParallelFor(const long maxnworkers=FF\_AUTO,}\]
\[
\text{ bool spinWait=false, bool spinBarrier=false);}\]

The first parameter sets the maximum number of worker threads that can be used in the ParallelFor (that is the maximum parallelism degree), the second argument sets non-blocking run-time for parallel computations, the third parameter sets non-blocking barrier implementation. For all arguments, default initialization values are provided.
The `ParallelFor` object encapsulates a number of `parallel_for` methods, which differ from each other in the number of arguments they get and for the signatures of the function body. A single `ParallelFor` object can be used as many times as needed to run different parallel-for instances (different loop bodies). Nested invocations of `ParallelFor` methods are not supported.

The loop body may be a standard function or a C++11 lambda-function. The following list presents the most useful parallel-for member functions provided by the `ParallelFor` class:

```cpp
// default values: step=1, grain=FF_AUTO
parallel_for(first, last, bodyF, nworkers);
// default value: grain=FF_AUTO
parallel_for(first, last, step, bodyF, nworkers);
// no default value
parallel_for(first, last, step, grain, bodyF, nworkers);
// loop body function signature
bodyF = F(const long idx);

// version for having access to the thid in the bodyF
parallel_for_thid(first, last, step, grain, bodyF, nworkers);
// thid is the id of the thread executing the body function
bodyF = F(const long idx, const int thid);

// explicit management of internal loop on sub-partitions
parallel_for_idx(first, last, step, grain, bodyF, nworkers);
bodyF = F(const long begin, const long end, const int thid);
```

Given the following sequential loop:

```cpp
auto F = [](const long i) { return i*i; }
for(long i=1; i < 100; i +=2) A[i] = F(i);
```

we can write the following FastFlow parallel-for:

```cpp
ParallelFor pf;
auto F = [](const long i) { return i*i; }
pf.parallel_for(1,100,2,[&A](const long i) { A[i]=F(i);});
```

or, by using the `parallel_for_idx` member function, we have:

```cpp
ParallelFor pf;
auto F = [](const long i) { return i*i; }
pf.parallel_for_idx(1,100,2,[&A](const long begin,const long end, const int thid){
    std::cout << "Hello I’m thread " << thid << " executing iterations (" << start <<"," << end <<")\n";
    for(long i=begin; i<end; i += 2) A[i]=F(i);
});
```
The `parallel_for_idx` is just a “low-level” version of the `parallel_for`, where the internal loop, iterating over all iterations assigned to the worker, has to be written directly by the user. This may be useful when it is needed to execute a pre-computation (executed in parallel) before starting the execution of the loop iterations, or in general for debugging purposes.

It is important to remark that, when `spinWait` is set to `true`, in some particular cases, the body function is called the first time with `begin==end==0` so it would be safe to test this condition at the beginning of the `parallel_for_idx` method (i.e. `if (begin==end) return;`).

**Iteration scheduling**

Three distinct iteration scheduling types are currently possible in parallel-for computations:

1. **default static scheduling**: the iteration space is (almost) evenly partitioned in large contiguous chunks, and then they are statically assigned to workers, one chunk for each worker.

2. **static scheduling with interleaving** $k$: the iteration space is statically divided among all active workers in a round-robin fashion using a stride of $k$. For example, to execute 10 iterations (from 0 to 9) using a concurrency degree of 2 and a stride $k = 3$, then the first thread executes iterations 0, 1, 2, 6, 7, 8 and the second thread executes iterations 3, 4, 5, 9. The default static scheduling is obtained by setting a stride $k = \text{iterationspace}/nworkers$.

3. **dynamic scheduling with chunk** $k$: in this case no more than $k$ contiguous iterations at a time are dynamically assigned to computing workers. As soon as a worker completes computation of one chunk of iterations, a new chunk (if available) is selected and assigned to the worker. The run-time tries to select as many contiguous chunks as possible in order to better exploit spatial locality. This allows to have a good trade-off between iterations affinity and load-balancing.

By default the **default static scheduling** is used. In general, the scheduling policy is selected by specifying the `grain` parameter of the `parallel_for` method. If the `grain` parameter is not specified or if its value is 0, then the **default static scheduling** is selected. If `grain` is greater than zero, then the **dynamic scheduling** is selected with $k = \text{grain}$. Finally, to use the **static scheduling with interleaving** $k$ the `parallel_for_static` method must be used with $k = \text{grain}$. Note that, if using the `parallel_for_static` with `grain` parameter set to zero, then the **default static scheduling** policy is automatically selected.
2.4.2 The FastFlow \textit{ParallelForReduce} and \textit{ParallelForPipeReduce} patterns

The FastFlow \textit{ParallelForReduce} is used to perform a parallel-for computation plus a reduction operation (by using a combiner function named $reduceF$ in the following) over a sequence of elements of type $T$. In order to deterministically compute the result, the reduction function needs to be associative and commutative.

The constructor gets the same parameters as the \textit{ParallelFor} constructor:

\begin{verbatim}
ParallelForReduce<T>(const long max=FF_AUTO,
                      bool spinwait=false, bool spinBarrier=false);
\end{verbatim}

The \textit{ParallelForReduce} class provides all the parallel-for methods already provided by the \textit{ParallelFor} class and a set of additional parallel\_reduce member functions:

\begin{verbatim}
parallel\_reduce(var, identity, first, last,
                bodyF, reduceF, nworkers);
parallel\_reduce(var, identity, first, last, step,
                bodyF, reduceF, nworkers);
parallel\_reduce(var, identity, first, last, step, grain,
                bodyF, reduceF, nworkers);
bodyF =F(const long idx,T& var);
reduceF =R(T& var,const T& elem);

parallel\_reduce\_thid(var, identity, first, last, step, grain,
                     bodyF, reduceF, nworkers);
// thid is the id of the thread executing the body function
bodyF =F(const long idx,T& var,const int thid);
reduceF =R(T& var,const T& elem);
// explicit management of internal loop on sub-partitions
parallel\_reduce\_idx(var, identity, first, last, step, grain,
                     bodyF, reduceF, nworkers);
bodyF =F(T& var,const long begin,const long end,const int thid);
reduceF =R(T& var,const T& elem);
\end{verbatim}

The \textit{reduceF} function specified in all parallel\_reduce methods, executes the reduction operation.

As an example, let’s consider the case of the summation of all elements of an array (see Fig. 2.1) in which a parallel-for is used for initialising the array $A$ and 3 different parallel\_reduce calls are used for computing the final sum (the reduction variable) using the default static scheduling, the static scheduling with interleaving and the dynamic scheduling, respectively.

It is worth noting that, the reduction variable in the parallel\_reduce function member is passed as first argument by reference. The formal parameter used in the \textit{bodyF} lambda and passed by reference, has no relation with the reduction variable. It is generally convenient to declare the formal parameter with the same name of the reduction variable in order to not change the user code in the \textit{bodyF}. 

8
```cpp
#include <vector>
#include <iostream>
#include <ff/parallel_for.hpp>
using namespace ff;

constexpr size_t SIZE = 1048576;

int main(int argc, char *argv[]) {
    int nworkers = (argc > 1) ? atoi(argv[1]) : 2;
    // creates the array
    std::vector<double> A(SIZE);

    ParallelForReduce<double> pfr(nworkers);
    // fill out the array A using the parallel-for
    pfr.parallel_for(0, SIZE, 0, [&](const long j) { A[j] = j * 1.0; });

    // loop body function, computes per-thread local sum of portion of the input array
    auto bodyF = [&A](const long j, double &sum) { sum += A[j]; };

    // reduce function called (sequentially) for the final summation of all
    // values computed by the threads
    auto reduceF = [](double &sum, const double elem) { sum += elem; };

    // static scheduling (default iteration scheduling policy)
    double sum = 0.0;
    std::cout << "Computing sum with " << std::max(1, nworkers / 2) << " workers, default static scheduling\n";
    pfr.parallel_reduce(sum, 0.0, 0L, SIZE, bodyF, reduceF, std::max(1, nworkers / 2));
    std::cout << "Sum = " << sum << "\n";

    // static scheduling with fixed interleaving
    sum = 0.0;
    std::cout << "Computing sum with " << nworkers << " workers, static scheduling with interleaving 1000\n";
    pfr.parallel_reduce_static(sum, 0.0, 0L, SIZE, bodyF, reduceF, std::max(1, nworkers / 2));
    std::cout << "Sum = " << sum << "\n";

    // dynamic scheduling with fixed chunk size set to 1000
    sum = 0.0;
    std::cout << "Computing sum with " << nworkers - 1 << " workers, dynamic scheduling chunk=1000\n";
    pfr.parallel_reduce(sum, 0.0, 0L, SIZE, 1, 1000, bodyF, reduceF, nworkers);
    std::cout << "Sum = " << sum << "\n";
    return 0;
}
```

Figure 2.1: Computing the sum of all array elements using different iteration scheduling.
The ParallelForPipeReduce uses a different skeleton implementation with respect to the ParallelForReduce pattern. It computes a map function and a sequential reduce function in a pipeline fashion.

This pattern is useful when the reduce function has to be computed sequentially, for example because there are concurrent write accesses to a memory location (so they have to be serialised using a lock), or because the reduction operator is not fully commutative. In these cases, the typical solution is to execute the map part (for example using a parallel-for) and then after the map the reduce code sequentially. This implementation may be expensive because a full barrier (between the map part and the reduce part) is required. The ParallelForPipeReduce pattern allows execution of the map and reduce parts in pipeline without any barriers.

The ParallelForPipeReduce pattern requires to explicitly send the tasks to the reduce stage inside the body of the map function. The ParallelForPipeReduce class defined in the parallel_for.hpp file provides only 2 parallel-for member functions:

```cpp
parallel_for_idx(first, last, step, grain, bodyF, nworkers);
bodyF = F(const long begin, const long end, const int thid);

parallel_reduce_idx(first, last, step, grain, bodyF, reduceF, nworkers);
bodyF = F(T& var, const long begin, const long end, const int thid);
reduceF = R(T& var, const T& elem);
```

As an example, in the Fig. 2.2 is reported the FastFlow code for computing the simple dot-product algorithm using both the ParallelForReduce and the ParallelForPipeReduce patterns.

2.5 Access to documentation

The documentation relative to the FastFlow final implementation of ParaPhrase patterns targeting homogeneous architectures is available through different ways:

- the tarball with the final implementation includes all the necessary tools to generate up to date DoxyGen documentation of the main FastFlow pattern features and library calls available
- a version of the same DoxyGen documentation may be accessed through the FastFlow web site\(^1\) (see Fig. 2.3).
- the FastFlow web site has a comprehensive “Tutorial” section [1]

Complete documentation of the FastFlow implementation of the ParaPhrase pattern set (including homogeneous and heterogeneous architecture targeting) will be included in the forthcoming deliverable D7.3 [6].

\(^1\)http://calvados.di.unipi.it/storage/refman/doc/html/index.html
```cpp
#include <ff/parallel_for.hpp>
#include <vector>
using namespace ff;

constexpr double INITIAL_VALUE = 3.0;
constexpr size_t size = 1048576;
constexpr int nworkers = 4;
constexpr size_t CHUNKSIZE = 0;

int main() {
    std::vector<double> A(size), B(size);
    
    // ParallelForReduce version
    double sum = INITIAL_VALUE;
    ParallelForReduce<double> pfr(nworkers, (nworkers < ff_numCores()));
    pfr.parallel_for(0, size, 1, CHUNKSIZE, [] (const long j) {
        A[j] = j * 3.14; B[j] = 2.1 * j;
    });
    auto Fsum = [](double& v, const double& elem) { v += elem; };
    pfr.parallel_reduce(sum, 0.0, 0, size, 1, CHUNKSIZE,
    [&] (const long i, double& sum) { sum += A[i] * B[i]; }, Fsum);
    printf("Sum =%g\n", sum);
}

// ParallelForPipeReduce version
double sum = INITIAL_VALUE;
ParallelForPipeReduce<double*> pfr(nworkers, (nworkers < ff_numCores()));
// A and B already initialized
auto Map = [] (const long start, const long stop, const int thid, ff_buffernode &node)
    { (cont.)
        if (start == stop) return ;
        double localsum = 0.0;
        for (long i = start ; i < stop; ++i)
            localsum += A[i] * B[i];
        node.put(new double(localsum));
    }
auto Reduce = [&] (const double* v) { sum += *v; };
pfr.parallel_reduce_idx(0, size, 1, CHUNKSIZE, Map, Reduce);
printf("Sum =%g\n", sum);
return 0;
}
```

Figure 2.2: Dot-product computation using 2 parallel-for-reduce versions.
Figure 2.3: FastFlow web page with the Doxygen documentation (sample)
Chapter 3

Erlang patterns

This part of the document collects all the information relative to the final release of the Erlang based homogenous implementation of the ParaPhrase pattern set as described in D2.5 [3].

The Erlang implementation of the ParaPhrase patterns is currently organized in two main packages: the skel library [2], hosting the core and generic ParaPhrase patterns and the high level pattern implementation prototype (HLPP), that also includes the implementation relative to the targeting of heterogeneous architectures, not relevant for this deliverable.

3.1 Installation

The skel library can be downloaded from github site at

https://github.com/ParaPhrase/skel

whereas the HLPP prototype (sk_hlp.tgz) may be downloaded from the ParaPhrase web site at

http://www.paraphrase-ict.eu/Deliverables

In both cases, the directories hosting the two software packages must be accessible from the environment of the Erlang interpreter used to run the application exploiting the ParaPhrase pattern libraries. skel is provided with a specific open source license included on the Github. The HLPP prototype is provided as open source LGPL.

3.2 Pattern list

The following patterns are supported, through the two different software packages:

Core/generic patterns Pipeline, farm, map, reduce with possibility to model iterative computation through the feedback construct (skel)
High level patterns  Divide and conquer (HLPP)

Domain-specific patterns  Pool evolution (HLPP)

3.3 Differences w.r.t. previous releases

The final implementation of the Erlang patterns does not present significative differences with the initial implementation, but for a number of minor bug and feature fixes.

The only notable difference consists in the release of a second version of high level patterns provided with the initial release (in sk_hlp-alt.erl file) which provides a slightly more efficient implementation by removing one step of the original implementation chain.

In the initial implementation, high level and domain specific skeletons are implemented on top of the skel generic skeletons. This means the Erlang code for the Pool or Divide&Conquer skeletons contains the definition of the call to a skel map skeleton. In turn, this implies the full toolchain of skel is invoked to interpret a skel workflow only made of the map call. Although this does not represent a large overhead, we’ve provided the base routines used to implement the map pattern in skel within the sk_hlp-alt.erl version of the high level patterns and then we have directly placed call to these low level routines in place of the original calls to the skel workflows.

This means the initial implementation of the Pool evolution pattern:

```erlang
1 dc(Isbase,Base,Divide,Conquer) ->
2   fun(X) ->
3       BC = Isbase(X),
4       case (BC) of
5         true -> Base(X);
6         false ->
7          [L] = skel:do([map, [seq, dc(Isbase,Base,Divide,Conquer)]], [X]),
8          L
9       end,
10      end.
```

has been provided as a:

```erlang
1 dc(Isbase,Base,Divide,Conquer) ->
2   fun(X) ->
3       BC = Isbase(X),
4       case (BC) of
5         true -> Base(X);
6         false ->
7          Subprobs = Divide(X),
8          Subsols = pardo:pardo(dc(Isbase,Base,Divide,Conquer,Mode),Subprobs);
9          Res = Conquer(Subsols)
10      end,
11    end.
```

in sk_hlp-alt.erl. This allowed to slightly improve the overall performance of the ParaPhrase pattern Erlang parallel applications, especially in those cases
where fine grain computations are used as (functional) parameters\(^1\) of the high level skeletons.

### 3.4 Access to documentation

The documentation of the Erlang implementation of the *ParaPhrase* patterns targeting homogeneous architectures may be accessed through:

- the README files in the distribution tarball
- the example files in the example subdirectory

---

\(^1\)i.e. as the *business logic*
Bibliography


