Project no. 288570

PARAPHRASE

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

Initial Implementation of Application-Specific Patterns

D2.6

Due date of deliverable: September 30, 2013
Actual submission date: September 30, 2013

Start date of project: 1st October 2011

Type: Deliverable
WP number: WP2
Task number: T2.2

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

Version 1.0 / Last edited by S. Campa /
Executive summary

This deliverable describes the initial implementation of a subset of application specific and high level patterns among those described in Deliverable [3] and [2]. The implementation we consider here is only targeting heterogeneous hardware platforms, being heterogeneous platform the subject of another, forthcoming deliverable. We selected the set of patterns to implement on the basis of the priorities expressed by the beneficiaries, in particular by those involved in the WP6 (Use cases) activities. The patterns considered more useful or more urgent to be introduced in a prototypal implementation has been developed in order to provide them a test framework on which starting to write code following the ParaPhrase methodology. As a consequence, in the next deliverable it could happen that slight modification will occur (maybe in the type or number of parameters for specific calls or in the adding of new/still not included methods) but the overall programming infrastructure can be considered assessed.

The deliverable is divided in two major parts: Chapter 2 describes the application specific pattern implemented on top of FastFlow; Chapter 3 presents the patterns implemented on top of the Erlang platform. The set of patterns is not exactly the same in the two cases, as of the different use cases and necessities by the beneficiaries implementing applications and use cases with C++/FastFlow and Erlang ParaPhrase frameworks.

The deliverable introduces each of the implemented domain specific or high level pattern by showing its programmatic interface and sample usage code.

The positioning of this deliverable is outlined in Fig. 1. It is based on the list of application specific patterns reported in [2] and their implementation will definitively assessed by D2.8 (Final homogenous implementation) and D2.7 (Heterogenous implementation of application specific patterns).
Figure 1: Positioning of Deliverable D2.6: w.r.t. other WP2 deliverables (top) and w.r.t. the forthcoming deliverables from other WPs (bottom).
## Contents

1 **Introduction** 4  
1.1 Download and install the code 4  

2 **FastFlow implementation** 5  
2.1 Divide and conquer 5  
2.2 Search 7  
2.3 Sort 9  
2.4 Pool evolution pattern 9  
2.5 Orbit pattern 13  
2.6 Stencil Pattern 14  
2.7 Work-flow graph interpreter 19  

3 **Erlang implementation** 27  
3.1 Divide&Conquer 27  
3.2 Pool 29  
3.3 Search 32  
3.4 Sort 32  
3.5 Orbit 33
Chapter 1

Introduction

This deliverable introduces a prototype implementation of a subset of application-specific patterns among the ones suggested by our project partners. The document is organized in two parts:

- in Chapter 2 we will show the FastFlow implementation of Divide and conquer, Search, Sort, Pool evolution, Orbit, Stencil and Work-flow graph interpreter
- in Chapter 3 we will show the Erlang implementation of Divide and conquer, Pool evolution, Search, Sort and Orbit

For each pattern we will propose the programming interface and an example of use. Being all kind of “high level” patterns, these patterns have been all implemented in terms of the “core” generic pattern defined in [1]. Moreover, some of them—as described later on in this document—can be defined as instances of other, more general, application-specific patterns in the set.

1.1 Download and install the code

The code can be downloaded and/or tested from two different sources:

- the ParaPhrase project web site, under the Deliverables tab, or through the direct url

  [http://www.paraphrase-ict.eu/Deliverables/d2-6-prototype/at_download/file](http://www.paraphrase-ict.eu/Deliverables/d2-6-prototype/at_download/file)

- by logging into the project host platform titanic.di.unipi with the account name paraphrase. Login will be only available through a password which will be e-mailed explicitly requesting to campa@di.unipi.it. The account will provide a pre-configured environment in which all the code is already installed and ready to be used.
Chapter 2

FastFlow implementation

2.1 Divide and conquer

Pattern interface The divide and conquer pattern has been introduced in [3] as a generic, high level pattern. Its behaviour can be represented by the following pseudo-code:

```plaintext
function DIVIDE&CONQUER(divide, basecase, solve, conquer)
    if basecase(X) then
        return solve(X)
    else
        return conquer(map
            (divide&conquer(divide, basecase, solve, conquer))
            (divide(X)))
    end if
end function
```

The interface of the pattern is provided as follows:

```cpp
template<typename T>
class dc: public ff_node {
public:
    // constructor : to be used in non-streaming computations
    dc(T task, // the task to be computed
        int pardegree, // the parallelism degree
        vector<T> div(T t), // the divide function
        bool isbc(T t), // the isbasecase function
        T solvebc(T t), // the solvebasecase function
        T cnq(vector<T> t)) ; // the conquer function

    // constructor : to be used in streaming applications
    dc(int pardegree, // the parallelism degree
        vector<T> div(T t), // the divide function
        bool isbc(T t), // the isbasecase function
        T solvebc(T t), // the solvebasecase function
        T cnq(vector<T> t)) ; // the conquer function

    // ..

};
```
The Divide and conquer pattern is a data parallel pattern that is internally implemented as a \texttt{f f\_farm}. The user is required to provide a pointer to the input data, the maximum parallelism degree to be used in the parallel computation of the divide\&conquer as well as the four user-defined functions specializing the divide\&conquer:

- function \textit{div}: \(T \rightarrow T[]\) which splits an input data of type \(T\) in subproblems;
- function \textit{isbc}: \(T \rightarrow \text{bool}\) which returns \textit{true} if a task \(T\) is a base case task;
- function \textit{solvebc}: \(T \rightarrow T\) which solves a base case task of type \(T\);
- function \textit{cnq}: \(T[] \rightarrow T\) which defines how to conquer sub-results of type \(T\).

Usage example The following code outlines the usage of the pattern, defining Fibonacci as a divide\&conquer pattern:

```cpp
#include <vector>
using namespace std;

bool fib_bc(int i) {
    if (i<2) return true; else return false;
}

vector<int> fib_div(int i) {
    vector<int> *v = new vector<int>();
    v->push_back(i-1);
    v->push_back(i-2);
    return *v;
}

int fib_solve(int i) {
    return 1;
}

int fib_cnq(vector<int> s) {
    cout << "cnq " << endl;
    int sol =s[0]+s[1];
    return sol;
}

int main(int argc, char * argv[]) {
    ...
    dc(n, nw, fib_div, fib_bc, fib_solve, fib_cnq);
    dc.run_and_wait_end();
    cout << "Computed " << dc.get_result() << endl;
}```
2.2 Search

The search pattern looks for items in a collection $A$ which can be a direct graph, a tree or an array.

**Pattern interface** In the following we consider the interface of the Search pattern working on a direct graph $G = (V, A)$ where $V$ is a list of vertexes (or nodes) and $A$ is a list of edges (or arcs) that connects two nodes.

For the sake of conciseness, we provide here only the interface of the Search pattern working on a graph data structures, which is the most complex one with respect to trees and arrays.

The FastFlow framework provides the class `ff_graphsearch` that can be used either for one single search computation on a given direct graph, or on a stream of tasks each one having a specific type $T$ which encapsulates all information needed to run the graph search.

```cpp
// base type of the graph node
class base_node_t {

    // compare operator
    virtual bool operator ==(const base_node_t&) const = 0;

    // returns the list of output nodes connected with this node
    virtual void out_nodes(std::deque<base_node_t*>&) const = 0;

    // base type of the graph node
    template<typename T=std::null_graph_t>
    class ff_graphsearch : public ff_node {

        public :

            // onstream is true if the graph pattern works on a stream of tasks
            ff_graphsearch (bool onstream);

            // stops the search after the first match
            template<typename base_node_t>

            // constructor
            ff_graphsearch (bool onstream);

            // stops the search after the first match
            void out_nodes(std::deque<base_node_t*>&) const = 0;

            // base type of the graph node
```

Figure 2.1: Searching elements in a direct graph.
bool search ( const base_node_t &start , const base_node_t &element,
base_node_t * results );

// searches all elements in the graph
template<typename base_node_t>
bool search ( const base_node_t &start , const base_node_t &element,
std::deque<base_node_t*> &results);

// sets parallelism degree
void setNumWorkers(const int nw);

}

The pattern requires that graph’s nodes have to be defined as a subclass of the
base_node_t interface type. This means that the two interface methods operator== and
out_nodes have to be defined for each graph node. The first method allows to
check if the input node passed as parameter is equal to the current node, the second
method returns a list of output vertexes directly connected with the current one (see
Fig 2.1).

When the ff_graphsearch pattern is used as a stage of a pipeline pattern or as
a worker of a farm pattern, it gets in input and produces in output tasks of type T
specified as template parameter. The type T has to provide at least the following
interface methods:

// returns true if the search has to stop after the first match
const bool first_of () const;
// returns the starting node of the graph used for the search
const base_node_t & getStart () const;
// returns the elment to search in the graph
const base_node_t & getElement() const;
// returns the pointer to the data structure where the results will be stored
std::deque<base_node_t*> getResults() const;

Usage example In the following a simple example demonstrating how to use the
ff_graphsearch pattern in a C++ sequential program is shown.

myGraph *graph;
...
// populate the graph

// get one node of the graph
node_graph_t & start = graph->getNode();
node_graph_t * result;

ff_graphsearch <> S(false);
S.setNWorkers(8);

if (S.search ( start , element , result ))
std::cout << "Element found\n";
else
std::cout << "Element not found\n";
2.3 Sort

The interface of the sort pattern has been designed as follows:

```cpp
template<typename T>
class sort : public ff_node{
public:
    // constructor : to be used in non-streaming computations
    sort(vector<T> data, // pointer to input data array,
          int pdegree, // parallelism degree
          int compare(T e1, T e2) // function comparing array items
          ) ;

    // constructor : to be used in streaming computations
    sort(int pdegree,
          int compare(T e1, T e2) // function comparing array items
          ) ;
};
```

The sort pattern can be seen as an application of the divide and conquer pattern. In fact, the initial implementation of the sort pattern is implemented as divide and conquer pattern where:

- non base case arrays are split into smaller arrays
- base case arrays (those whose length is smaller than a given threshold) are sorted with any sequential sorting algorithm
- conquer is implemented according to the divide function. If the divide just split in parts the original array, then the conquer may be implemented as a mergesort; in case the divide has been implemented as in the quick sort algorithm, then conquer is just a juxtaposition of the sorted subarrays.

2.4 Pool evolution pattern

The Pool evolution pattern has been presented in [3] as a generic, high level pattern. Its behaviour can be represented by the following pseudo-code in which P defines a population of individuals (i.e. a collection of items of type \textit{P\_type}).

```cpp
while not(t(P)) do
    \quad N = e(s(P))
    \quad P = P \cup f(N, P)
end while
```

The pattern iterates over a number of generations limited by the termination function \( t : P\_type \rightarrow bool \) which returns \textit{true} if a given termination condition has been satisfied. At each iteration, function \( s : P\_type \rightarrow P\_type \) selects a subset of the population which is evolved by applying the evolution function \( e : P\_type \rightarrow P\_type \) and then passed through a filter function \( f : P\_type \rightarrow P\_type \) that
possibly computes the fitness of the new individuals and filters those worth to be included in the population to be submitted to the next iteration of the pattern.

**Pattern interface**  The pattern interface has been designed as follows:

```cpp
1 template<typename T>
2 class poolevolution : public ff_node {
3 public:
4   // constructor: to be used in non-streaming applications
5   poolevolution ( vector<T>∗ pop,  // the initial population
6                   vector<T>∗ select ( vector<T> pop),  // the selection function
7                   vector<T>∗ evolution(T individual ),  // the evolution function
8                   vector<T>∗ filter ( vector<T> pop),  // the filter function
9                   bool ∗ termination ( vector<T> pop));  // the termination function
10
11   // constructor: to be used in streaming applications
12   poolevolution ( vector<T>∗ select ( vector<T> pop),  // the selection function
13                   vector<T>∗ evolution(T individual ),  // the evolution function
14                   vector<T>∗ filter ( vector<T> pop),  // the filter function
15                   bool ∗ termination ( vector<T> pop));  // the termination function
16
17   // the function returning the result in non streaming applications
18   vector<T> get_result () ;
19
20   // changing the parallelism degree
21   void setParDegree( int pardegree );
22
23   // these are used to set up data parallel versions of the sequential phases
24   // set up parallel termination function as a mapreduce (internal) pattern
25   void setParTermination ( int pardegree,  // par degree setup
26                   T ∗ mapTerminationFun(T t),  // map function
27                   T ∗ redTerminationFun(T t1, T t2));  // reduce function
28   // set up parallel filter function as a mapreduce (internal) pattern
29   void setParFilter ( int pardegree,  // par degree setup
30                   T ∗ mapFilterFun(T t ),  // map function
31                   T ∗ redFilterFun (T t1 , T t2 ));  // reduce function
32
33   // set up parallel selection function as a mapreduce (internal) pattern
34   void setParSelection ( int pardegree,  // par degree setup
35                   T ∗ mapSelectionFun(T t),  // map function
36                   T ∗ redSelectionFun (T t1, T t2 ));  // reduce function
37
38   // these are used to set up stream parallel versions of the sequential phases
39   void setParTermination ( int pardegree );
40   void setParSelection ( int pardegree);
41   void setParFilter ( int pardegree );
42 };
```

The pattern has two constructors, one to support standalone execution of the patterns, that is execution only processing the parameter input population, and one to support execution of the pattern over items appearing onto an input stream.

By default, the pattern has been implemented in such a way each one of the selection, filtering and termination functions are computed sequentially, thus having the parallel computation of the evolution function as the only source of parallelism. However, the user can decide that some of these sequential stages have to be executed in parallel. The pattern supports either task farm or map-reduce parallel
implementation of all the selection, filtering and termination stages, to be used respectively in stream parallel or in standalone contexts. To the purpose, two types of methods are provided for each operator:

- a set of methods of type

  ```
  void setParXXX(int pardegree);
  ```

  where XXX can be one among Termination, Selection or Filter. These methods replicates the instances of the genetic operator with pardegree replicas (in detail, it is rewritten as a farm of pardegree workers);

- a set of methods of type

  ```
  void setParXXX(int pardegree,
                 T * mapXXXFun(T t),
                 T * redXXXFun(T t1, T t2));
  ```

  where XXX can be one among Termination, Selection or Filter, let the XXX stage to split the input data provided with the constructor in partitions, each partition being evaluated by the mapXXXFun function and, eventually, all the partition subresults are reduced by the reduceXXXFun function. At a lower level, the stage is rewritten as a mapreduce or a map pattern (if reduceXXXFun defines a null pointer).

**Usage example**  In this example we will show how an application instancing a pool pattern should be structured. In particular, we are implementing an application in which selection and evaluation of the fitness can be done in parallel. Our instance of the problem is one in which:

- each individual of the population is tagged as selected or not-selected by mean of some criterion

- individuals are evaluated in parallel and those exploiting a very good fitness could generate new individuals and/or mutate

- the new or mutated individuals are added to the original population

At instance time, the evaluation is computed in parallel but also the filter stage can be turn to a parallel stage by invoking the corresponding setParFilter method while the selection and the termination stages are sequential. Since the parallel stages can be expressed as task-parallel computations, we simply need to instantiate the pattern by calling the corresponding constructor and to specify the parallelism degree to be applied to the new parallel stage.
Alternatively, we could specify a data-parallel implementation of the pattern, maybe exploiting a parallel version of the selection stage: for instance, we could be interested in selecting and evaluating specific individuals by splitting the population in
NUMPART chunks and then measuring fitness on and filtering only the selected individuals. In such case, the piece of code needed to be called in the main of our application is

```
1  // builds the pattern for a non-streaming application
2  poolevolution<INDIVIDUAL> pool(init_population, selection, evolution, filter, termination);
3
4  // transforms the selection stage into a mapreduce
5  pool. setParSelection(NUMPART, mapSelection, reduceSelection);
6
7  ...
```

and we would need to implement functions `mapSelection` and `reduceSelection` in order to manipulate the partitions. They would appear as follows:

```
1  /*--- functions needed for implementing mapreduce selection ---*/
2  INDIVIDUAL * mapSelection(INDIVIDUAL t) {
3    INDIVIDUAL * new_t =... // select / deselect individual t
4    return new_t;
5  }
6
7  INDIVIDUAL * reduceSelection(INDIVIDUAL t1, INDIVIDUAL t2){
8    INDIVIDUAL * new_t =... // build new_t on the basis of t1 and t2
9    return new_t;
10  }
```

### 2.5 Orbit pattern

As defined in the [3] the orbit pattern computes the transitive closure of a set $S$ through a set of generators $G$:

1. repeat
2. for all $x \in S$ do
3.   for all $g \in G$ do
4.     $New = g(x)$
5.   for all $n \in New$ do
6.     if not$(n \in S)$ then
7.       $S = S \cup \{n\}$
8.   end if
9. end for
10. end for
11. until no more items added to $S$

The orbit pattern has been therefore implemented by properly instantiating a pool evolution pattern, where

- the `termination` function evaluates if more items has been added to the pool;
- the `selection` function is implemented as the identity (all the items in the set are selected to be passed to generators, actually);
• the *evolution* function consists in the parallel evaluation of the functions in the set of generators over every selected item of the pool and in the production of new items;

• the *filter* function inserts the new items into the pool if they have not already present in the pool.

The pattern interface is defined as follows:

```cpp
1 template<typename T>
2 class orbit : public ff_node {
3 public:
4 // constructor : used in non−streaming computations
5 orbit (vector<T> set, int pardegree);
6 // constructor : used in streaming computations
7 orbit (int pardegree);
8 // add generator
9 void add_generator (vector<T> gen(T));
10 // get result from non streaming computation
11 vector<T> get_result ();
12};
```

The pattern may be used just calling the constructor with the proper parameters, then adding the relevant generator functions with a set of calls to the `add_generator` method and eventually invoking the proper `run_and_wait_end` method as for all the other ParaPhrase patterns provide by FastFlow.

## 2.6 Stencil Pattern

Stencil computations are a class of iterative kernels which update data elements according to some pattern called stencil. They are most commonly found in many scientific and engineering applications like computation fluid dynamics, computer graphics, modelling and simulation methods. In these computations the items in the input data set (typically represented as a 2D- or 3D-matrix) dynamically change over a number of iterations representing time steps. Each item change depends on the previous value of that item and on the value of its neighbour items in the collection. The number of neighbours and the distance between the considered item and its neighbours characterizes the stencil pattern.

Here we focus on static stencil, i.e. stancil computations in which the shape of the stencil remains the same over all time steps. Dynamic stencil computations, instead, are considered as a particular instance of the work-flow graph interpreter pattern (see Sec. 2.7).

For simplicity’s sake, we consider in the following 2D-based stencil, i.e. input/output data sets are bi-dimensional matrices. In the left-hand side of Fig. 2.2 are sketched some possible 2D-stencil pattern. We denoted with *Xradius* and *Yradius* the maximum distance w.r.t. the considered element, in the X and Y directions,
respectively. Depending on which kind of boundary conditions have to be used in the application, at the edges of the data structure could be present the so called ghost-zone. For the 2D case, they can be present only at the edges of the rows or of the columns or at both edges. The values in the ghost cells are typically particular values and the way in which those values are choosen represents different boundary conditions. Periodic boundary conditions are the simplest to implement: values needed for the right ghost cell are simply copied from just inside the left boundary and so on with all boundaries. It is worth to notice that, for this particular case, the ghost-zone might not be used at all provided that extra checks are necessary for identifying edge cells.

Generally, stencil computations work as follow: values computed at Step\textsubscript{i} depend on values calculated at Step\textsubscript{i-1}. For this reason typically two data structures are used, one holding old values (the ones computed in the Step\textsubscript{i-1}) and one for the new values. The 2 data structures are then swapped before executing the next iterative step of the algorithm.

In some cases the grid space is so big that it is not possible to use 2 data structures of the same size. A smaller buffer for computing new data has to be used instead of a complete grid so that the computation of the single step is further divided in sub-steps. The data structures used for this kind of computation are represented in Fig. 2.6 and the pseudo code describing the computation of one single step is sketched in the following (no ghost-zone are present):

```plaintext
numsteps = Ysize/Youtsize; // Ysize is a multiple of Youtsize
radiussize = (Yradius*Xsize);
blocksize = (Youtsize*Xsize);
last = ((Youtsize-Yradius)*Xsize);
for (step = 0, start =0; step < numsteps; ++step, start +=Youtsize) {
    // stencil computation
    for (size_t i = start , i0=0; i < (Youtsize+start ); ++i, ++i0)
        for (size_t j = 0; j < Xsize; ++j)
            ...
}
```

Figure 2.2: Some possible stencil patterns on a 2D-grid (left). Ghost-zone added to a 2D-grid space (right).
Figure 2.3: In place computation using a smaller output buffer and extra buffer for edge rows.

Pattern interface

The 2D stencil pattern is implemented by the \textit{stencil2D} class of the FastFlow framework. The \textit{stencil2D} class has 2 constructors: one used in non-streaming computation and one used in case the stencil pattern is a stage of a pipeline pattern or as a worker of the \textit{farm} pattern. The most important methods of the \textit{stencil2D} are reported in the following:

```
1  template<typename T>
2  class stencil2D : public ff_node {
3    public:
4      // constructor1 : used for non-streaming computation
5      stencil2D(T *Min, T *Mout,
6                  const size_t Xsize, const size_t Ysize,const size_t Youtsize ,
7                  int nw, int Yradius=1, int Xradius=1, bool ghostcells = false ,
8                  const size_t chunksize=DEFAULT_STENCIL_CHUNKSIZE);
9
10     // constructor2 : used for streaming computation
11      stencil2D(int nw, int Yradius=1, int Xradius=1, bool ghostcells = false ,
12                  const size_t chunksize=DEFAULT_STENCIL_CHUNKSIZE);
13
14    void initInFunc ( init_F_t F, void *extra );
15    void initOutFunc( init_F_t F, void *extra );
```
The user provides, together with the input and output buffers (Min and Mout respectively), also a number of function pointers used for:

1. initialising the input matrix using \textit{initInFunc} (optional);
2. initialising the output matrix using \textit{initOutFunc} (optional);
3. sequential pre-computation using \textit{preFunc} (optional). The function will be called before starting the stencil computation;
4. post-computation using \textit{postFunc} (optional);
5. computing the stencil on the single cell using \textit{computeFunc} (mandatory);
6. computing the termination condition using \textit{reduceFunc} (mandatory).

The initialization functions and the stencil function are executed in parallel using a given number of threads (nw, which is provided by the user). The implementation template of the stencil pattern is described in the following pseudo-code:

```c
if (initIn) initIn(Min,Xsize,Ysize); // concurrent initialization
if (initOut) initOut(Mout,Xsize,Youtsize); // concurrent initialization
rVar = reduceVar; iter = 0;
do {
    swap();
    if (beforeFor) beforeFor(Min,Xsize, Ysize, rVar);
    // each element of the Mout matrix is computed independently using nw threads
    FF_PARFORREDUCE_START(loopCompute, rVar, identityValue, i, Xstart, Xstop, Xstep,
        chunkSize, nw) {
        for (long j=Ystart ; j< Ystop; j+=Ystep) {
            Mout[i*Xsize+j] = computeFReduce1(i,j,Min,Xsize,Ysize, rVar);
        }
    } FF_PARFORREDUCE_F_STOP(loopCompute,rVar,reduceOp);
    if (afterFor) afterFor(Mout,Xsize, Ysize, rVar);
} while(++ iter <maxIter && iterCondition(rVar , iter ));
```

At each iteration: first the input and output matrices are swapped, then the \textit{beforeFor} function is executed sequentially, next each element of the \textit{Mout} matrix is computed using a parallel for, then the \textit{afterFor} function is executed sequentially.
and finally the do-while termination condition is evaluated calling the function \textit{iterCondition}.

The invocation of the \textit{computeFReduce1} function for each matrix element could introduce too much overhead in particular if the stencil computation is fine-grain. To overcome this limitation, the stencil pattern interface has been augmented with some extra methods which allow the user to have higher control on the computation granularity. In particular, the compute function provided by the user using the method \textit{computeFunction}, can be a function that works on the single element (as shown above), on a row of elements and on the entire matrix. In the last two cases, the parallel loop computation is executed inside the user function using the \textit{FF_PARFOR/REDUCE} patterns provided by the \textit{FastFlow} framework. The initialization and termination phases needed to correctly use the \textit{FF_PARFOR/REDUCE} are completely in charge of the stencil pattern and hidden to the programmer.

\textbf{Usage example} As an example we consider the program to solve a finite difference discretization of Helmholtz equation using the Jacobi iterative method on a 2D-grid. This is a classical 5-points (4 neighbours) stencil computation on a grid of \(m \times n\) points. The 2D-Jacobi algorithm can be implemented as a double loop nest traversing the complete computational domain by updating each grid point using 2 distinct matrices.

The pseudo-code of the algorithm is the following:

```c
1 while(k<=maxit && error > tot ) {  
2  /* copy new solution into old matrix */  
3  for ( int j=0;j<m;j++)  
4    for ( int i=0;i<n;++i) uold[i+m*j]=u[i+m*j];  
5  /* computes the stencil and residual */  
6  for ( int j=1;j<(m-1);++j)  
7    for ( int i=1;i<(n-1);++i) {  
8      resid = compute_resid(f , i , j , uold,ax,ay) ;  
9      /* updates solution */  
10     u[i + m*j] = uold[i + m*j] − omega × resid ;  
11     /* accumulates residual error */  
12     error =error + resid × resid ;  
13    }  
14    error = sqrt ( error ) / (n+m); k++;  
15 }
```

Using the \textit{stencil2D} pattern the entire algorithm can be written as follow:

```c
1 // instantiate stencil pattern  
2 stencil2D<double> stencil (u,uold,m,n,NUMTHREADS,1,1,false);  
3 4 stencil −>initInFunc(initU) ;  
5 stencil −>initOutFunc(initUold) ;  
6 7 stencil −>computeFunc(stencilF, 1,m−1, 1,n−1,1);  
8 stencil −>reduceFunc(condF, maxit, reduceOp, 0.0) ;  
9 10 stencil −>run_and_wait_end();
```

where:
• `initU` initialise the single cell of the \( u \) matrix. The function is called in a parallel loop in order to execute the initialization phase in parallel;

• `initUold` initialise the single cell of the \( uold \) matrix as in the previous case;

• `computeFunc` executes the stencil for each pair \((i, j)\) updating the \( u \) matrix and reading values from the \( uold \) matrix;

• `reduceFunc` reduction function used to evaluate the error and for terminating the computation;

• `run_and_wait_end` starts the stencil computation and wait for termination.

### 2.7 Work-flow graph interpreter

A work-flow (WF) graph interpreter pattern models those computations where the tasks to be computed as well as the relative ordering of the task computations are expressed through a work-flow graph \( G = (N, A) \). The set of nodes \( N \) in the graph are instructions, that is functions computing one or more outputs from one or more inputs.

When the function code represent sequential instructions and arcs represent data dependency among instructions, the WF interpreter is also known as data-flow (DF) interpreter.

The data-flow execution model is a well-known computing paradigm extremely attractive for parallel processing. It consists in an asynchronous way to execute instructions based only on the availability of their input arguments (pure data dependencies, i.e. read-after-write).

Instead of expressing parallelism at the single instruction level which is typically too fine-grain to obtain good performance on modern general purpose multi-core architectures, (large) portions of the sequential code having pure functional dependencies between input parameter and output results, are considered resulting in macro-dataflow (MDF) instructions. The MDF program is therefore represented as a graph whose nodes are computational kernels and arcs read-after-write dependencies.

The MDF instructions interpreter is in charge of scheduling fireable instructions and managing data dependencies as fast as possible to avoid introducing new computational bottlenecks. To this end, the following three points represent fundamental aspects for an efficient implementation of a MDF interpreter:

**Construction of the task graph (DAG)**

In the general case the task graph could be very large; its generation time can affect significantly the computation time, and the memory required to store the entire graph may be extremely large. To alleviate these issues, a widely used solution consists in generating the graph during the computation, such that only a “window” of
the graph is maintained in memory. This also allows to overlap tasks computation with the graph generation.

**Handling task dependencies**

two main operations on the graph need to be properly optimised: i) update dependencies after the completion of previously scheduled tasks; ii) determine ready (fireable) tasks to be executed. These operations should be done as fast as possible and in parallel with tasks computation, to avoid affecting the performance of the parallel application.

**Scheduling of fireable tasks**

a task having all input dependencies resolved (the so called fireable task) may be selected by the interpreter for execution. This selection needs to be done considering both task-affinity assignment of tasks to instruction executors in order to better exploit cache level hierarchies, and parallelism in order to maintain the number of ready tasks as big as possible during the execution to prevent stalls. The first optimisation leverages on the fact that graph arcs represents data dependencies, so that if task \( i \) unlocks execution of task \( j \), then they share at least one of the dependencies. Executing task \( j \) on the same instruction executor that ran task \( i \) as soon as possible, increases the probability that the common data reside in the cache hierarchy. The second optimisation leverages on the fact that a graph node with an higher degree of output arcs, might unlock a larger number of tasks. Following this principle, the scheduling policy should select with higher priority among all fireable tasks those ones that have the higher number of outgoing edges. This can be accomplished by ordering fireable tasks with respect to the degree of the related node on the graph.

In order to reduce memory consumption, in-place computation is generally used on shared memory platforms instead of the classic dataflow approach. To this end, dataflow graphs have to be enriched by additional anti-dependencies (write-after-read) between tasks for removing the need of costly copies of the original data structure. This at the price of a possible lower parallelism between MDF instructions. In many cases, the task graph is not very big or memory consumption is not an issue, then the task graph can be statically generated and stored in main memory at the beginning of computation thus not incurring in any extra overhead during execution.

The Work-flow graph interpreter is a very general pattern and can be used to execute in parallel almost any programs. The parallel execution efficiency depends on different factors in particular: i) on the size of the graph, ii) on the kind of dependencies and, iii) on the computation granularity of the single tasks. Complex dynamic stencil-based computations such as those coming from Dense Linear Algebra domain (as LU, QR and Cholesky block-based factorisations), may be executed in parallel using the work-flow/MDF pattern, provided that particular
attention is paid for tuning the block (or tile) size, which greatly influences the computational granularity.

### 2.7.0.1 Possibly iterated numerical library calls pattern

This is a particular instance of the work-flow pattern in which graph tasks contain numerical library function calls. This pattern is quite common in linear algebra domain where algorithms are often built as a list of (possibly iterated) calls to numerical libraries such as BLAS and LAPACK functions. The computation graph can be built statically or dynamically during the computation.

A MDF interpreter may be implemented as a two-stage pipeline: a sequential stage called Graph Descriptor (GD), which defines and executes the user algorithm that eventually produces the instructions for building the DAG, and a parallel stage that generates (a portion) of the DAG using a suitable data representation and then executes in parallel the resulting DAG nodes as soon as their input dependencies are satisfied. The partial results of the computation are typically produced in output as a stream of tasks or stored in the shared memory by updating a data structure.

The parallel stage is logically composed by 2 concurrent entities: a task scheduler (called ESched) and a set of replicated workers (Ws) which are the real interpreter of the MDF instructions. The Esched receives in a non-deterministic way, both new instructions coming from the GD and also completed tasks coming from the set of workers. It generates dynamically the graph nodes during the computation upon receiving instructions from GD. From the sequential order in which tasks are generated by the GD, the ESched computes a partial ordering that ensures computation correctness, and, by evaluating data dependencies among tasks, it adds the corresponding node/edge to the DAG structure. A completed task coming from one of the workers may either activate new tasks ready to be scheduled for execution, or trigger the termination condition.

Since the memory required to store the entire DAG may be huge, only a “window” of the graph structure is maintained in main memory in order to limit the
amount of memory used. When the number of generated graph nodes reaches a predetermined threshold (that can be tuned by the user), the channel from GD to Esched is temporary disabled; this way the graph generation is halted and the ESched handles only completed tasks coming from the workers. When the number of available instructions returns below the threshold, the channel from the GD node is re-enabled. The resulting skeleton structure (called ffMDF) is sketched in Fig. 2.4.

The GD thread executes the user-defined function which generates task’s nodes. The user provides a function (Algo) describing the algorithm. In the Algo function, the AddTask procedure (implemented by the ffMDF runtime) is used to define the operations composing the graph.

It requires, a function pointer to the real kernel code (i.e. low level PLASMA wrappers or LAPACK wrappers), the number and the list of parameters used by the function. For each parameter the user has to specify its size and its mode. The mode specifies if the parameter is used in INPUT or OUTPUT, in such a way that the corresponding graph dependencies may be built. A special mode, VALUE, is required for those parameters that are directly evaluated inside the GD and do not concur to the DAG creation. All tasks generated via the AddTask function are packed and sent to the Esched thread, which creates the corresponding DAG node.

ESched, non-deterministically receives completed task notifications from the set of workers and new instructions, produced by the AddTask function, executed in the user function from GD. The pseudo-code of the Esched thread is sketched in the following:

```plaintext
while(!ComputationEnded)
{
    receive_task(T);
    if (task_from_workers(T)) {
        update_graph_dependencies();
        for (int i=0; i<num_workers; ++i) {
            if (select_ready_task(T))
                schedule_task(T);
            else break;
        }
        if (graph_size() < Threshold)
            enable_input_channel();
        else {
            compute_task_dependencies(T);
            add_task_to_graph(T);
            if (graph_size() > Threshold)
                disable_input_channel();
        }
    }
}
```

The generic worker execute the following pseudo-code:

```plaintext
while(!ComputationEnded) {
    get_new_task(T);
    compute_task(T);
    return_task(T);
}
```
Task scheduling policies

Worker threads receive tasks through an on-demand protocol, i.e. tasks are scheduled upon request. This policy ensures a very good work load balancing without using more complex and costly work-stealing techniques. When multiple fireable tasks are available, they are enqueued in a local buffer by the Esched. When the worker completes the computation on a task, a notification is sent back to ESched, so that it can update the graph with the new dependencies, and the computed node is removed from the graph freeing memory space.

The currently available ffMDF scheduling policies (SchedP) are the following ones:

- SIMPLE (S) the tasks to be executed are selected on a FIFO order basis: the first task becoming fireable is the first one executed; this is considered the basic scheduling strategy. It entirely relies on the FastFlow task-farm support;

- LOCALITY FIFO (LF) a locality-oriented scheduling, implemented by using multiple ready queues, one per worker thread. Tasks that become fireable after the completion of a given task executed by the worker \(i\), are inserted in a ready queue associated to the worker \(i\). Tasks scheduled to the worker \(i\) are extracted in FIFO ordering by the ready queue \(i\). When the queue is empty, tasks are stolen from other workers queues, implementing a kind of centralized work stealing strategy.

- LOCALITY LIFO (LL) another locality-oriented scheduling that works exactly as the LF policy, with the only exception that tasks are extracted from the ready queues in a LIFO order (i.e. the last inserted task is the first to be extracted), possibly guaranteeing even more cache locality than the previous case.

- PARALLELISM (P) a parallelism-oriented scheduling, in which ready tasks are kept in a single queue. The first task to be executed is the ready task with the higher number of forward dependencies in the DAG, following the concept expressed in Section 2.7. This policy has been implemented by using a single priority queue.

- LOCALITY PARALLELISM (LP) a mix of parallelism and locality-oriented scheduling policies, in which we ensure locality by using a queue per worker thread, as in the LL and LF policies, and parallelism by extracting from the queues using the priority mechanism of the P policy.

Although more complex policies can be added, we tried to keep them as simple as possible in order to avoid the case in which the ESched stage is the main bottleneck of the ffMDF pattern when fine grain DAGs are executed.
Pattern interface The programmer that uses the ffMDF pattern has to provide the function that generates the graph tasks (for the case in which the graph is generated dynamically) or, in case the graph is statically available, has to provide a table representing the entire graph structure with all task dependencies already set. The two options are available by using two different constructors of the FastFlow class `ff_mdf` implementing the ffMDF pattern.

```cpp
1 struct param_info {
2   uintptr_t tag; // unique tag
3   direction_t dir; // can be INPUT, OUTPUT or VALUE
4 };
5
6 struct basic_f_t {
7   virtual void call ()=0;
8 };
9
10 struct table_entry_t {
11   table_entry_t () : id (0) ,dep_counter(0) ,Function( nullptr ) {};
12   unsigned long id; // task identifier
13   unsigned long dep_counter; // dependencies counter
14   base_f_t ∗Function; // task function
15   std::vector<unsigned long> cont; // indexes to other tasks
16 };
17
18 template<typename T=null_mdf_t>
19 class ff_mdf: public ff_node {
20   public :
21     // constructor1 : graph dynamically generated
22     template<typename Tparam>
23     ff_mdf(void (∗userF)(ff_node ∗const , Tparam ∗const), Tparam ∗const arg, bool onstream);
24
25     // constructor2 : graph statically available and dumped into the 'table' parameter
26     ff_mdf(std :: vector< table_entry_t ∗> &table, bool onstream);
27
28     // AddTask is called inside 'userF'
29     template<typename... Param>
30     void AddTask(std::vector<param_info> &P, void (∗F)(Param ...) , Param ... args);
31
32     // sets scheduling policy
33     void setPolicy ( policies_t kind);
34     // sets parallelism degree
35     void setNumWorkers(const int nw);
36
37     void setOutstandingTaskSize ( size_t numtasks);
38     void setThreshold ( size_t th=0);
39
40     ...;
41 }
42
The first class constructor accepts the user function `userF` which will be called inside the GD stage. The `userF` generates the graph tasks by calling the `AddTask` method of the `ff_mdf` class. The methods `setOutstandingTaskSize` and `setThreshold` allow to control the maximum number of tasks that can be produced by the GD stage but not yet considered for building the graph, and the maximum size (in terms of number of tasks) of the graph stored in memory during the computation, respectively. In case the `ff_mdf` pattern is used as a pipeline stage or as a farm
worker, the boolean parameter in the constructor has to be set to ‘true’. Stream
data elements have each one the same type $T$.

The second constructor is used instead when the graph is statically generated
and dumped into the dynamic vector ‘table’. The table contains for each each entry
a graph task with the following information: the instruction identifier, the depen-
dencies counter correctly initialized to the number of input dependencies for the
task, the task function that has to be executed, and a list of table entries (indexes)
that represent the output edges of the task (the input dependencies of other tasks of
the graph).

Usage example  As an example, let us consider the simplified code needed to
instantiate and run the Cholesky factorisation algorithm using the ffMDF pattern.
The programmer have to select one of the available scheduling policy $SchedP$, and
to provide a function pointer (the $taskGen$ function) which describes the algorithm
and that will be executed by the GD stage of the FastFlow run-time.

```cpp
struct Parameters {
  float_complex_t *A; // work matrix
  int nb, bs; // n. of blocks and block size
} &P = { A, nb, bs};

ff_mdf<> chol(taskGen, &P, false) ;
chol . setPolicy (LP);
chol . setNumWorkers(nworkers);
chol . setThreshold (nworkers∗64);
chol . setOutstandingTaskSize (2048);
chol . run_and_wait_end();

void taskGen(Parameters *const P){
  A = P->A; nb=P->nb, bs=P->bs;
  std::vector<param_info> Param;
  for (k=0;k<=(nb−1);k++) {
    for ( i=0;i<=(k−1); i++) {
      const param_info _1={&A[k∗bs∗bs∗nb+k∗bs],INPUT}, _2={&A[k∗bs∗bs∗nb+i∗bs],INPUT},
          _3={&A[k∗bs∗bs∗nb+k∗bs],OUTPUT};
      Param.push_back(_1); Param.push_back(_2); Param.push_back(_3);
      AddTask(Param,F_CHERK,&A[k∗bs∗bs∗nb+k∗bs],&A[k∗bs∗bs∗nb+i∗bs],
               &A[k∗bs∗bs∗nb+k∗bs]);
    }
  }
  for (j=(k+1); j<=(nb−1); j++) {
    for (i=0; i<=(k−1); i++) {
      const param_info _1={&A[k∗bs∗bs∗nb+k∗bs],INPUT}, _2={&A[k∗bs∗bs∗nb+i∗bs],INPUT},
          _3={&A[j∗bs∗bs∗nb+k∗bs],INPUT}, _4={&A[j∗bs∗bs∗nb+k∗bs],OUTPUT};
      Param.push_back(_1); Param.push_back(_2); Param.push_back(_3); Param.push_back(_4);
      AddTask(Param,F_CPOTF2,&A[k∗bs∗bs∗nb+k∗bs],&A[j∗bs∗bs∗nb+i∗bs],
               &A[j∗bs∗bs∗nb+k∗bs],&A[j∗bs∗bs∗nb+i∗bs],
               &A[j∗bs∗bs∗nb+k∗bs],&A[j∗bs∗bs∗nb+k∗bs]);
    }
  }
```
Param.clear();
const param_info _1={&A[k*bs*bs*nb+k*bs].INPUT}, _2={&A[j*bs*bs*nb+k*bs].INPUT}, 
 _3={&A[j*bs*bs*nb+k*bs].OUTPUT};
Param.push_back(_1); Param.push_back(_2); Param.push_back(_3);
AddTask(Param,F_CTRSM,&A[k*bs*bs*nb+k*bs],&A[j*bs*bs*nb+k*bs],
 &A[j*bs*bs*nb+k*bs]);
  } // for j
  } // for k
}
Chapter 3

Erlang implementation

The Erlang implementation of the high level and domain specific patterns is a proof-of-concept implementation built on top of the generic patterns provided by the skel library (https://github.com/ParaPhrase/skel). The skel library implements the generic patterns described in [1]: sequential wrapper, pipeline, farm, map and reduce.

This Erlang implementation is a partial implementation of the high level patterns described in [3], being this an initial implementation to be refined and extended to fully support the development of the ParaPhrase Erlang use cases, as developed in work package WP6. In particular, of the high level and domain specific patterns listed in [3]:

- divide&conquer, search, pool have been implemented on top of skel
- sort and orbit have been implemented as instances of the divide&conquer and pool patterns, respectively
- work-flow graph interpreter and stencil pattern have not been implemented being specific “numeric” patterns not of interest for the Erlang use cases and applications considered so far in the project.

The high level and domain specific patterns implemented for this deliverable will be eventually included in the skel library, once the implementation will be refined.

The following sections describe how the different patterns implemented may be used and the main mechanisms used to exploit parallelism.

3.1 Divide&Conquer

The parallel implementation of divide&conquer actually implements in parallel the computation of the sub-tasks obtained from the divide phase.

The parallel implementation of the divide and conquer pattern is derived from the sequential implementation:
dc(Isbase, Base, Divide, Conquer) ->

fun(X) ->

BC = Isbase(X),

case (BC) of

ture ->

apply(?MODULE, Base, [X]);

false ->

Subprobs = apply(?MODULE, Divide, [X]),

Subsols = lists :map(apply(?MODULE, dc, [Isbase, Base, Divide, Conquer], Subprobs),

Res = apply(?MODULE, Conquer, [Subsols]),

Res

end

end.

by substituting line 9 by the proper call to an `skel:map`. The different parameters of the divide&conquer pattern denote:

- the function returning `true` if the input data represents a base case and `false` otherwise (`Isbase`)
- the function computing the solution of a base case (`Base`)
- the sub problems to be solved in case the input data do not represent a base case (`Divide`)
- the function used to build the divide&conquer result out of the subproblems results (`Conquer`).

Considering the whole computation tree of a divide&conquer computation, as many processes as the total number of items in the different `Subprobs` lists are created. Proper definition of `Isbase` (and consequently of `Base`) parameter(s) may help avoiding to generate an unnecessarily large number of processes/subtasks and overall to properly tune the amount of parallelism actually exploited.

A second implementation of the divide&conquer pattern is provided by the `dcl` higher order function. `dcl` (divide&conquer with limited parallelism) has the signature of the `dc` higher order function with the addition of a numerical parameter:

```
dcl(Isbase, Base, Divide, Conquer, Pardegree) -> ...
```

The parallelism exploited, in this implementation, is limited by the `Pardegree` parameter in that the divide&conquer computation tree forks parallel processes to compute the `Subprobs` in parallel only up to `Pardegree` times. After the limit has been reached, any of the `Subprobs` item is actually computed sequentially.

**Usage example** The following code shows how the divide&conquer pattern may be used to implement a quicksort.

```erlang
%% check base case; the length of the base case is given as a parameter
isbase(N) ->

fun(L) ->
```
(length (L) < N)
end.

%%% solve base case
base(X) ->
lists : sort (X).

%%% collect smaller/equal from list
smaller(_,[]) ->
[;]
smaller(X,[Y|Ry]) ->
case (X < Y) of
   true ->
   smaller(X,Ry);
   false ->
   [Y|smaller(X,Ry)]
end.

%%% collect largers from list
larger(_,[]) ->
[;]
larger(X,[X|Ry]) ->
larger(X,Ry);
larger(X,[Y|Ry]) ->
case (X > Y) of
   true ->
   larger(X,Ry);
   false ->
   [Y|larger(X,Ry)]
end.

%%% divide non base case lists : this is quick sort
divide ([X|Y]) ->
Smaller = smaller(X,Y),
Larger = larger(X,Y),
[Smaller, [X], Larger].

parsort (N,X) ->
(dc(isbase(N),base, divide ,conquer))(X).

3.2 Pool

The pool evolution pattern is an high level pattern that has been defined to model parallelism in wide range of cases, namely in all those cases where a set of values is processed to make it evolve iteratively (see [3]).

The pool pattern implemented for this deliverable exactly mirrors the definition given in [3], and in particular is defined as the high order function:

pool(Termination, Selection, Evolution, Filter) ->
fun(Set) ->
case (Termination(Set)) of
true ->
Set;
false ->
Selected = Selection(Set),
Rest = setdifference(Set, Selected),
Evolved = lists : map(fun ?MODULE:Evolution/1, Selected),
Filtered = Filter (Evolved),
Newset = union (Rest, Filtered ),
(pool (Termination, Selection, Evolution, Filter )) (Newset)
end.

where the map at line 9 has been suitably substituted by a call to the eskel:map. The different parameters of the pool patterns denote:

- the function returning true if the termination has been actually reached (Termination)
- the function selecting individuals from the current population to be submitted to the evolution process (Selection)
- the function computing the evolution of an individual (Evolution)
- the function filtering the evolved individuals to be inserted in the population set (Filter)

It is worth pointing out that, differently from what happens to the divide&conquer pattern, the map call does not imply recursive call to the pattern. Therefore the processes sparked to evaluated in parallel the Evolution function over the Selected items of the input set all terminate before the pattern is called recursively to evaluate the Newset.

A variant of the pool pattern is being considered that implements the selection, filter and termination phases in parallel. In this case, the three parameters relative to the involved functions will all be given in terms of mapreduce functions. The new pool pattern (called poolp) will therefore be called as:

\[ \text{poolp}(\text{Termmap, Termred, Selmap, Selred, }} \]
\[ \text{Evolution, Filtermap, Filterred}) \]

with the functions $X$map and $X$red being the part evaluated within the map and the reduce–respectively–of the $X$ phase.

**Sample usage** The following code shows how the pool pattern may be used to implement a dummy genetic algorithm computing “Hello world”.

```plaintext
%% genes are lists of integers
%% computes fitness as distance from the target gene
fitness_to_target ([], []) -> 0;
fitness_to_target ([X|Rx],[Y|Ry]) -> abs(X-Y) + fitness_to_target (Rx,Ry);
fitness_to_target (X,[]) -> length(X);
fitness_to_target ([], Y) -> length(Y).
fitness (Target) ->
fun(Gene) ->
fitness_to_target (Gene,Target)
end.

%% termination function
```
fitpop ([], _) →
[[]];
fitpop ([G|Rg], Fitness ) →
[ Fitness(G) | fitpop(Rg, Fitness ) ].
termination_to_target(Pop,Targetgene) →
Fitness = fitness(Targetgene),
Fit = fitpop(Pop, Fitness ),
lists :member(0,Fit).
termination(Targetgene) →
fun(Pop) → termination_to_target(Pop,Targetgene).

%% selection function
select_best_n(Pop, N, Targetgene) →
Fitness = apply( ?MODULE, fitness, [Targetgene] ),
Fit = fitpop(Pop, Fitness ),
Fitted = lists :zip(Pop, Fit),
Sorted = lists :sort( (fun({_,Fa},{_,Fb}) → (Fa < Fb) end, Fitted ) ,
Bests = [Indiv || {Indiv ,_} < lists :sublist( Sorted , trunc(N) ) ],
Rest = [Indiv || {Indiv ,_} < lists :sublist( Sorted , trunc(N+1),length(Sorted) ) ],
{Bests,Rest}.
select_to_target_N(Pop, N, TG) →
Fitness = apply( ?MODULE, fitness, [TG] ),
Fit = fitpop(Pop, Fitness ),
Fitted = lists :zip(Pop, Fit),
Sorted = lists :sort( (fun({_,Fa},{_,Fb}) → (Fa < Fb) end, Fitted ) ,
[ Indiv || {Indiv ,_} < lists :sublist( Sorted , trunc(N) ) ]).
select(N, TargetGene) →
fun(Pop) → select_to_target(Pop,N,TargetGene).

%% evolution function
mutate([_|Rx],1,New) →
[New|Rx];
mutate([X|Rx],N,New) →
[X|mutate(Rx,(N-1),New)].
mutation_to_max(X,Max) →
Len = length(X),
Candidate = random:uniform(Len),
New = random:uniform(Max),
mutate(X,Candidate,New).
mutation(Max) →
fun(X) → mutation_to_max(X,Max).

%% filter function : we use select again
%% instantiation of the pattern %%%%
evolve(Pop, N, Maxval, TargetGene) →
pool( termination(TargetGene), select(N, TargetGene),
mutation(Maxval), select(N, TargetGene)).
3.3 Search

The search pattern has been defined over Erlang digraphs. Parallelism is exploited by visiting in parallel all the nodes reachable from the currently visited nodes that have not been already visited. The code for the search patterns is the sequential code:

```erlang
searchNotVisited(G, From, Visited) ->
  fun(Target) ->
    case lists:member(From, Visited) of
      true -> [];
      false ->
        case Target =:= From of
          true -> [From];
          false ->
            NewVisited = [From | Visited],
            Next = digraph:out_neighbours(G, From),
            lists:flatten(lists:map(?MODULE:searchNotVisited(G, From, NewVisited), Next))
        end
    end
  end.
end.

search(G, Target, From) ->
  F = ?MODULE:searchNotVisited(G, Target, []),
  F(From).
```

where the map at line 11 has been substituted by a proper call to a parallel `skel:map`. The parameters of the search pattern denote:

- the graph to be searched (the digraph `g`)
- the node we look for (Target)
- the node we start the search from (From)

The pattern returns an empty list in case the Target node has not been found. A variant of the pattern calling a parameter callback on the found node(s) is being considered.

It is worth pointing out that in the general case of recursively defined data structures, the search pattern may be implemented in terms of the divide&conquer pattern.

3.4 Sort

The initial implementation of the sort pattern is defined in terms of the divide&conquer pattern. The sample code detailed in Sec. 3.1 shows how a quick sort algorithm may be implemented as a divide&conquer pattern instance.

As a further example, a mergesort implementation may be defined simply changing the implementation of the Divide and Conquer functions:
and re-using the other functions needed to instantiate the \texttt{dc} pattern already defined in Sec. 3.1:

\begin{verbatim}
1 parsort (N,X) -->
2   (dc(ishbase(N),base,divide_ms,conquer_ms))(X).
\end{verbatim}

\section*{3.5 Orbit}

The orbit pattern may be implemented in terms of the \texttt{pool} pattern by defining:

\begin{itemize}
  \item \textbf{Selection} as the identity function
  \item \textbf{Evolution} as the function applying all the generators of the orbit pattern to a single item of the population set
  \item \textbf{Filter} as the function filtering out all the items already present in the population set, and
  \item \textbf{Termination} as the function comparing the old population set with the current one.
\end{itemize}
Bibliography

