Project no. 288570

PARAPHRASE

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

Enhanced Pattern Transformation System
D4.2

Due date of deliverable: 31 July 2013

Start date of project: October 1st, 2011

Type: Deliverable
WP number: WP4
Task number: T4.1

Responsible institution: University of St Andrews
Editor and editor’s address: Christopher Brown, University of St Andrews

Version 0.1 (0.1)

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

Deliverable D4.1 introduced the initial pattern transformation system. In addition to describing an initial refactoring tool for Erlang, D4.1 also described a number of new refactorings for Erlang, such as *Introduce Pipeline*, *Introduce Farm* and *Introduce Map*.

This deliverable describes the enhanced pattern transformation system, where we describe a new refactoring tool implemented for C++ in the Eclipse IDE. Following the same trend as in D4.1, we also introduce a number of new refactorings for C++ and FastFlow, such as *Introduce Pipeline* and *Introduce Farm*. In addition, we also introduce a new *Component* class, and provide a refactoring to shape C++ code into a hygienic state ready for parallelisation. We also provide a number of use-cases demonstrating the C++ refactoring tool against a set of examples taken from the Industrial Partners, SCCH and HLRS.

Positioning of Deliverable 4.2

The positioning of this deliverable (D4.2) with respect to other deliverables is shown in Figure 1. In particular, the work presented in D4.2 is based upon the work presented in D2.2 (Homogeneous Implementation of Parallel Patterns), focussing on FastFlow and C++. In addition to this, the refactorings were implemented after evaluating the use-cases described in D6.4. D4.2 will provide input for D4.3 (The Pattern Transformation User Interfaces).
Figure 1: The Positioning of D4.2
1. Introduction

This deliverable describes the second phase of the Refactoring Tools implemented for the Pattern Transformation System, as described in T4.1. The work here builds heavily on the skeleton work as proposed in D2.4 for C++ and also the Use-Cases as reported in D6.4.

D4.1 reported an initial transformation system, targeted at Erlang demonstrating a number of new refactorings that introduce and tune parallelism, such as *Introduce Pipeline*, *Introduce Farm*, *Introduce Map* and *Introduce Chunking*. Many of these refactorings are carried forward for FastFlow and C++ as described here, in a new tool, built into Eclipse that supports the full C++11 standard. The refactorings reported in this deliverable follow the basic premise as those in D4.1, but are enhanced and tailored to fit the work by the Industrial Partners (SCCH and HLRS) and also the work done for FastFlow.

1.1 What is new in this deliverable?

- We introduce a new refactoring tool for C++, developed in Eclipse, using the CDT plugin. The refactoring tool covers the full C++11 standard, preserving layout and comments in the refactored source code.

- We introduce new refactorings for C++ that introduce and tune parallelism, using the Fastflow skeleton library. The new refactorings include: *Introduce Farm*, *Introduce Pipeline*, and *Identify Component*.

- We present new use cases, demonstrating the effectiveness of our refactoring tools for C++.

- We demonstrate that our refactoring tool can be used by the industrial partners, including SCCH and HLRS, by refactoring two use cases, taken from D6.5.
2. Refactoring C++

This chapter introduces a new refactoring tool for C++, built into the Eclipse developer IDE and framework. The new C++ refactoring tool supports a number of new refactorings, supporting skeletons implemented as part of WP2 for FastFlow.

2.1 Refactoring in Eclipse

Our refactoring prototype is implemented in Eclipse, using the CDT plugin. The programmer is presented with a menu of possible refactorings to apply. The decision to apply a refactoring is made by the programmer. Once a decision has been made, any required transformation is performed automatically. In this way, we can rely on programmers making informed decisions about which refactorings to apply, but do not rely on them necessarily having expertise with parallelism or skeletons. Figure 2.1 shows a sample screenshot of our refactoring tool, where the programmer is presented with a menu of possible parallel refactorings to apply, such as Identify Component, Introduce Farm and Introduce Pipeline, as described below.

2.1.1 The CDT Plugin

The CDT project provides a fully functional C and C++ integrated Development Environment based on the Eclipse platform. Features include: support for project
creation and a managed build for various toolchains, standard make build, source navigation, various source knowledge tools, such as type hierarchy, call graph, include browser, macro definition browser, code editor with syntax highlighting, folding and hyperlink navigation, source code refactoring and code generation, visual debugging tools, including memory, registers, and disassembly viewers.

2.2 Implementing the Refactorings

While refactorings differ from each other in their side-conditions and transformation rules, their implementation normally follows a similar pattern. A refactoring normally involves the following steps:

- Transform the program source into an internal representation; this is currently done automatically by the CDT parser plugin, where it transforms the source code into an Abstract Syntax Tree (AST).
- Locate the focus of the refactoring. Usually the user is required to highlight a particular function or expression in the program. For example in identifying components, the user must select a function call, which is located within the AST using location information supplied by Eclipse.
- Validate the side conditions of the refactoring.
- Perform the refactoring transformation. Currently the AST is the modified directly by the refactoring.
- Present the refactored program to the user. This requires the AST to be pretty printed back into source code, the Eclipse editor automatically refreshes the buffer, and the refactored program is presented.

2.3 Refactorings

For each refactoring, the following is introduced where appropriate:

- A general description of the refactoring, giving an overview of the purpose of the refactoring and how it would change the structure of the program. This general description is also intended to give a motivation for the refactoring in question.
- An example showing the effects of the refactoring and its inverse operation. Performing a refactoring *left to right* typically introduces a skeleton structure, and from *right to left* eliminates the structure. The examples presented in this chapter are only intended to show the basic principles of the refactorings, further examples are given in the next chapter.
• A set of conditions from left to right. A refactoring is made up of a set of side-conditions and a set of transformation rules. The side-conditions must be met in order for the transformation to take place, otherwise the refactoring tool terminates with an error message.

• A set of transformation rules, describing the structure source code changes the refactoring performs.

2.3.1 Identify Component

Identify Component introduces a Component instance for a selected function call. The purpose of the Component class is twofold:

1. It provides a provision for hygienic code, meaning that a component must be free of side-effects and be referentially transparent. A component must also adhere to the component specification as outlined in D3.1. In this deliverable, however, we assume that the function call to be selected for a component is already hygienic, and identified as such by the user before the refactoring is performed.

2. A component allows the selected function call to be used as a worker in a FastFlow skeleton, where FastFlow workers must be an instance of ff_node. We note that this encapsulation is not required for Erlang, due to higher-order functions, and, indeed, even if another C++ skeleton library is used that may not require the workers to inherit in this way.

Example An example of identifying and introducing a component is shown in Figure 2.2. The left column shows before the refactoring has taken place, and the right column shows after the refactoring has taken place. In the left column, the user has highlighted the code block worker(new ff_task_t(i,j));. This function call is transformed into a new Component declaration (in this example the refactoring has introduced the Component instance as w), where the function name is passed as a parameter to the constructor of the Component class. The refactoring also introduces a call to the callWorker method, which invokes the worker function, passing in the formal parameter to callWorker as an actual parameter to the worker function.

The component definition, Component, is provided as part of the Fastflow interface and is defined in Figure 1. A component is defined as a C template, parameterised over the type of the parameter to the user selected function, worker. The component class inherits from ff_node, overloading the svc function, which is called by FastFlow to denote the worker computation to be performed within a skeleton. In our component definition, svc simply calls the worker function, the actual parameter to worker is taken from the FastFlow task pool, the head of which is the parameter to the svc function, task. The result of the worker function is added to the task pool of the next skeleton stage, unless the component
Listing 1 The Component Class

```cpp
template <class T> class Component : public ff_node {

public:
    Component(T ∗ (∗worker)(T ∗)): worker(worker), final(false) {}
    Component(T ∗ (∗worker)(T ∗), T ∗ results[]):
        idx(0), results(results), worker(worker),
        final(true) {}

    void ∗ svc(void ∗ task) {
        if (final == true) {
            T ∗ taskin = (T ∗) task;
            results[idx++] = (∗worker)(taskin);
            return GO_ON;
        } else {
            T ∗ taskin = (T ∗) task;
            T ∗ taskout = (∗worker)(taskin);
            return ((void ∗)taskout);
        }
    }

    T ∗ callWorker(T ∗ x) {
        T ∗ res = worker(x);
        delete x;
        return res;
    }

protected:
    bool final;
    T ∗ (∗worker)(T ∗); // the worker function
    int idx;
    T ∗∗ results;
};
```

is used as a single worker, in which case the result is returned. If the component is to be used as a single stage skeleton, then an extra parameter is passed to the constructor, to collect the results of the computation.

Conditions Suppose we select the function `worker`, and the user-defined new name is `w` then the side-conditions for introducing a component are:

- The new name for the `Component` instance, say `w`, is introduced by the user when he/she runs the refactoring. The new name must not, therefore, conflict with any other name within the scope in which `worker` is selected and defined.
- The selected `worker` must be a function call.
- The `worker` function must be defined at the global level.
The `worker` function must have an arity of one, and the type of the parameter must be a structure, or a single type.

**Transformation Rules**

- The selected function `worker(x)` is transformed into two statements:

  ```
  Component<ff_task_t> w(worker);
  w.callWorker(new ff_task_t(x));
  ```

  where:

  - `ff_task_t` is the type of `x`, and is defined as a structure with a constructor accepting `x` as a parameter; and,
  - `w` is the user-defined name for the `Component` instance.
Comments

- It would be possible to implement a refactoring for making the arguments to a function into the fields of an appropriate data type, possibly as part of the Identify Component refactoring or a separate step.

- The code in Figure 1 does not test for overflow of the result array, and it does also not allow for infinite streams. It would be possible to extend the Component class to deal with dynamic data structures using smart pointers from C++.

2.3.2 Introduce Pipeline Declaration

Introduce Pipeline Declaration simply introduces a new Fastflow pipeline declaration with a user-defined name. This declaration is used by the Introduce Pipeline Stages refactoring, which adds component stages to the introduced pipeline declaration.

```cpp
Component<ff_image> s1(georef);
Component<ff_result> s2(denoise);
ff_pipeline pipe;
for(int i=0;i<NIMGS;++i) {
    images[i] = new Image;
    res = s1.callWorker(new ff_image(images[i]));
    results[i] = s2.callWorker(new ff_image(res));
}
```

Figure 2.3: Before and After Effects of Introducing a Pipeline Declaration

Example Figure 2.3 shows an example of introducing a pipeline declaration. Here, the user simply specifies the position for the new declaration in the editor, and gives the new name for the pipeline declaration. The refactoring then inserts a new declaration, with the user-defined name, providing the name avoids capture (the name doesn’t conflict with any other names in scope), otherwise an error message is given to the user.

Conditions

- The user-defined pipeline name must not conflict with any other name in the scope in which the pipeline declaration is to be inserted.

- The type `ff_pipeline` must be in scope where the pipeline is declared.
Listing 2 The StreamGen Class

```c++
template <class T>
class StreamGen: public ff_node {
    public:
        StreamGen(int N, T* input[]): N(N), input(input) {}
    void* svc(void* task) {
        for(int i = 0; i < N; i++) {
            ff_send_out((void*) input[i]);
        }
        return NULL;
    }
    protected:
        int N;
        T** input;
};
```

Transformation Rules  The refactoring inserts `ff_pipeline pipe` into the source code at the position specified by the user, where the name, `pipe`, is user-defined.

2.3.3 Introduce Pipeline Stages

Introduce Pipeline Stages takes an existing pipeline declaration and adds a number of selected components as the various pipeline stages to the existing pipeline.

Example  Figure 2.4 shows, in the left column, before the refactoring is applied, where the user selects a for loop with a number of statements corresponding to a function composition. In the example, the user selects the following for loop:
for(int i=0;i<NIMGS;++i) {
    images[i]= new Image;
    res = s1.callWorker(new ff_image(images[i]));
    results[i] = s2.callWorker(new ff_image(res));
}

Here, s1 and s2 are Component types, and the result of s1 is given as a parameter to s2. This code is refactored into the code shown in the orange box in the right column. Here, the refactoring tool first replaces the for loop altogether with an instance of a StreamGen class (the definition of which is given in Figure 2). This takes the list of tasks (in this case image) and sends them to the first stage of the skeleton. Finally, the refactoring adds the stages, s1 and s2 to the already-defined pipeline. The result of the call to s1.callWorker() (res) is passed as a reference to the final pipeline stage, as the second parameter. The result of the pipeline will be placed into this parameter. Finally the pipeline is executed, using the run_and_wait() function.

Conditions  Suppose the user selects the following code:
for(int i = 0; i < N; i++) {
    r1 = s1.callWorker(x[i]);
    r2[i] = s1.callWorker(r1);
}
then the conditions for introducing a pipeline skeleton are:

- A selected pipeline declaration, pipe, must be in the scope at the point of the for loop. The pipeline declaration is given as a parameter to the refactoring.
- r1 and r2 must be declared in scope.
- i must be used within the for loop.
- s1 and s2 must be declared as Component types (see Figure 1 for the definition of a Component type).
- The result from s1 must be used in the parameter to s2.
- The parameter to s1 must be an array indexed by the for loop counter.
- If the loop contains statements that are not component calls, the refactoring ignores them, unless they use the result of a component, in which case the refactoring terminates with an error. The result of a component may only be used as a parameter to other components.
Transformation Rules

- A new declaration, StreamGen<ff_image> streamgen(N, x), is added in place of the selected for loop, passing in the parameter to s1 without the index (x) and also the for bound (N). The name for streamgen is generated by the refactoring tool and guaranteed to avoid name capture.

- streamgen is added a stage to the pipeline, immediately after the pipeline declaration.

- The components identified in the selected code are added as stages to pipe in the order that they appear in the original code:

  ```cpp
  pipe.add_stage(&s1);
  pipe.add_stage(&s2);
  ```

- Finally, the statement `pipe.run_and_wait_end();` is added.

2.3.4 Introduce Farm Declaration

Introduce Farm introduces a new FastFlow farm skeleton, passing in a selected component as a worker function.

```cpp
Component<ff_task> s1(f);

for(int i=0;i<N;++i)
  for(int j=0;j<N;++j)
    {          
      s1.callWorker(new ff_task_t(i,j));
    }
```

Figure 2.5: Before and After Effects of Introducing a Farm Declaration

Example  Figure 2.5 shows an example of introducing a new farm declaration. In the figure, the left column shows before the refactoring takes place, and the right column shows after the refactoring has taken place. In the example, the user selects the component s1, and the refactoring introduces a new farm, farm (named by the user), and adds the component, s1, as a worker to the farm. The number of workers
to the farm are also user defined. In this example, the user specifies that there are 8 workers to the farm. The worker declaration, \(w\), is introduced by the refactoring, avoiding name capture. Finally, the farm is executed and frozen, awaiting tasks to be offloaded to its task queue, via the \texttt{farm.run\_then\_freeze();} function call.

**Conditions**

- The user-defined farm name must not conflict with any other name in the scope in which the farm is to be introduced.
- The type \texttt{ff\_farm} must be in scope where the farm is declared.

**Transformation Rules**

- The refactoring inserts \texttt{ff\_farm<> farm(true)} into the source code at the positions specified by the user, where \texttt{farm} is a user-defined name.
- A vector of type \texttt{ff\_node} is introduced after the farm declaration.
- The user specifies the number of farm workers, \(N\). A for loop is introduced instantiating the vector with \(N\) instances of a user selected component, \texttt{s1}.
- The worker vector is added to the farm.
- Finally, the farm is executed in a frozen state, awaiting tasks to be uploaded to its task queue.

### 2.3.5 Introduce Farm

**Example** Figure 2.6 shows an example of introducing tasks to a farm. In the figure, the user selects the for loop in the left column, and the result of the refactoring is shown in the right column. Here, the refactoring searches for occurrences of a component \texttt{callWorker} within the body of the for loop, and replaces it with offloading a task to the farm. The original parameter to the \texttt{callWorker} function is used as the parameter to the \texttt{farm\_offload} (i.e., it becomes a new task in the farm). After the for loop, an end of stream signal is sent to the farm, via \texttt{FF\_EOS}. Finally, the farm is executed and suspended, via the \texttt{farm\_wait\_freezing()} command. The FastFlow farm, \texttt{farm}, is identified as the farm to add the tasks into by the user before the refactoring is performed.

**Conditions**

- The selected for loop must have an instance of a component, with a call to the \texttt{callWorker} method. The result of the component must not be used before the termination of the for loop, otherwise the refactoring cannot postpone the retrieval of the component results until the end of for loop.
The selected for loop must only have a single instance of a component, otherwise the refactoring has no way to assign workers and tasks to the farm.

The type of the parameter to the component instance must unify with the type of the parameter to the farm worker.

A FastFlow farm must be in scope and passed as a parameter to the refactoring.

Transformation Rules

- Statements within the for loop of the form s1.callWorker(X), where s1 is a Component instance, are refactored into farm.offload(X), where farm is the selected farm, and X is the parameter passed to callWorker.

- Immediately after the for loop, the refactoring tool introduces the following statements:

```c
farm.offload((void *)FF_EOS);
farm.wait_freezing();
```
3. Use Cases

This chapter demonstrates the refactorings introduced in Chapter 2 on a number of realistic use-cases. The Molecular Dynamics and the Graphical Lasso glasso_comp use case are taken from D6.4, provided by the industrial partners, SCCH and HLRS. Here we only report the refactoring process. D6.5 shows the speedup and performance figures that result from the refactoring of these two use cases.

3.4 Ant Colony Optimisation

Ant Colony Optimisation (ACO) [2] is a heuristic for solving NP-complete optimisation problems, inspired by the behaviour of ants living in real ant colonies. An ACO algorithm consists of a number of iterations. In one iteration, each ant independently computes a solution to the problem, with the solution being partially guided by a pheromone trail produced by ants. To compute one component of a solution, an ant (with the designated probability \( q \)) follows the pheromone trail for that component or (with the probability \( 1 - q \)) it performs a biased random selection of the component. In this way, different ants generally produce different (but similar) solutions. After the iteration is finished, and all ants have computed solutions, the best solution is chosen and the pheromone trail is updated according to that solution. After that, the next iteration, where ants compute new solutions based on the new feromone trail, is started.

An example ACO algorithm that we consider in this deliverable is computing a solution to a Single Machine Total Weighted Tardiness Problem (SMTWTP) [5]. In SMTWTP, we are given \( n \) jobs, whereas each job \( i \) is characterised by its processing time, \( p_i \), deadline, \( d_i \), and weight, \( w_i \). The goal is to schedule execution of jobs in a way that achieves minimal total weighted tardiness. The tardiness of a job \( i \), in a schedule is defined by \( T_i = \max\{0, C_i - d_i\} \), where \( C_i \) is the completion time of the job \( i \) in that schedule. The total tardiness of the schedule is defined as \( \sum w_i T_i \).

In the ACO solution to the SMTWTP problem, in each iteration each ant independently computes a schedule. The pheromone trail that guides the computation of schedules is defined by a matrix \( \pi \), where \( \pi[i,j] \) is the preference of assigning job \( j \) to the \( i \)-th place in the schedule. Therefore, in each step of the solution computation, an ant will either pick the job with the highest preference for that
position, or will choose a biased random selection (again based on the pheromone trail). Once the iteration is finished and all ants have computed their schedules, the schedule that obtains the minimal total weighted tardiness is selected, and the pheromone trail is modified to increase chances of selecting job in the same order as in the currently found best solution.

### 3.4.1 C++ Implementation

Figure 3 shows the extract of the sequential code for the SMTWTP instance of the ACO class of algorithms, written in C++. The most relevant code for refactoring and parallelisation is the iteration loop, where in each iteration every ant computes a solution, the best solution is chosen and the pheromone trail is updated.

The `solve` function produces a solution based on the arrays of processing times, deadlines and weights of jobs and the pheromone trail. The `tau.pick_best` function picks the best solution and `update` updates the pheromone trail, `tau`.

### 3.4.2 Parallelisation

From the description of the ACO algorithm and Figure 3, we can observe that the potential for parallelisation lies in the fact that each ant can compute its solution completely independently of all other ants. Other two phases of each iteration, i.e. picking the best solution and updating the pheromone trail, are relatively cheap and inherently sequential. Therefore, the way to parallelise the ACO algorithm seems to be to introduce the map or task farm, where a task would consist of computing the solution for one ant.

#### Step 1: Identifying Components

The first step in the refactoring process is to identify the component that will be used as the worker in the farm. In the example from Figure 3, we identify the function call `solve` as a worker function,
Listing 4 Sequential Ant Colony with an Identified Component

```c++
for (j = 0; j < num_iter; j++) {
    for (i = 0; i < num_ants; i++)
        Component<ff_task_t> w(solve);

    cost[i] = w.callWorker(new ff_task_t(num_jobs,
                               i,
                               &(results[i*num_jobs]),
                               process_time,
                               weight,
                               deadline,
                               tau);
    best_t = pick_best(num_ants, num_jobs, cost,
                       results, &best_result);
    update(num_jobs, best_t, best_result, tau);
}
```

and introduce a Component instance, passing the solve function as a parameter. The affect of this refactoring is shown in Figure 4.

**Step 2: Introduce a Farm Declaration** The next step in the refactoring process is to introduce a new farm declaration. This new declaration is added just before the for loop, so that it can be used within the for loop body. This is particularly useful when there are a predetermined set of tasks upon whose completion the farm itself can be destroyed. The code in Figure 5 shows the result of applying the refactoring, where the Component declaration has been lifted by the user previous to the refactoring taking place.

**Adding Tasks to the Farm** The final step in the refactoring process is to add the tasks to the farm. In order to do this, the user selects the for loop. This enables the refactoring to replace calls to the callWorker function with farm.offload, passing the actual callWorker parameter to offload instead. The code after refactoring looks like this.

```c++
Component<ff_task_t> w(solve);
ff_farm<> farm(true);

farm.add_collector(NULL);

for (i = 0; i < num_workers; i++)
    workers.push_back(w);

farm.add_workers(workers);
farm.run_then_freeze();
for (j = 0; j < num_iter; j++) {
```
Listing 5 Introducing a new Farm declaration, adding w as the worker function

```c
Component<ff_task_t> w(solve);
ff_farm<> farm(true);

farm.add_collector(NULL);

for (i = 0; i < num_workers; i++)
    workers.push_back(w);

farm.add_workers(workers);

farm.run_then_freeze();

for (j = 0; j < num_iter; j++) {
    for (i = 0; i < num_ants; i++)
        Component<ff_task_t> w(solve);

    cost[i] = w.callWorker(new ff_task_t(num_jobs,
        i,
        &(results[i*num_jobs]),
        process_time,
        weight,
        deadline,
        tau);

    best_t = pick_best(num_ants, num_jobs, cost,
        results, &best_result);

    update(num_jobs, best_t, best_result, tau);
}

for (i = 0; i < num_ants; i++)
    farm.offload(new ff_task_t(num_jobs,
        i,
        &(results[i*num_jobs]),
        process_time,
        weight,
        deadline,
        tau));

farm.offload((void*)FF_EOS);
farm.wait_freezing();

best_t = pick_best(num_ants, num_jobs, cost, results, &best_result);
update(num_jobs, best_t, best_result, tau);
```
3.5 Molecular Dynamics

Molecular Dynamics (MD) simulation models the interactions between molecules in a system [3]. Once the system is initialized, forces are computed among all the interacting molecules in the system. Then, the Newton equations of motion are integrated to advance the positions and velocities of the molecules. The simulation is carried out until the computation of the time evolution of the system is completed for a specified length of time.

Here we have used our in-house Molecular Dynamics code CMD, which is cultivated for fundamental research into high performance computing. The serial CMD is ported to FastFlow programming model, utilizing the programming patterns provided by it. The refactoring tool is employed to transform the serial CMD to Fastflow, exploiting the programming patterns as a perfunctory measure. The transformed code should exploit the FastFlow patterns in somewhat similar manner to the manual implementation of the same.

CMD features multiple MD data structures, algorithms and parallelization strategies and thus allows for quantitative comparisons between them. Two widely used data structures are implemented - with corresponding algorithms - for the computation of interactions between molecules in the system, “BasicN2” and “Molecule-Blocks”. The BasicN2 use case has been exercised with the refactoring tool.

In the BasicN2 use case, molecules are spatially distributed in a large 3D domain. The distance between every pair of molecules in the domain is computed. For every pair with a distance lower than the cut-off radius, forces acted on the molecules are computed. It is a compute intensive and highly data parallel problem with the compute complexity of O(N^2).

3.5.1 Parallelization of BasicN2

Profiling the BasicN2 use case apprises that the force calculation routine dominates the simulation time (e.g. 99.91% for 68,000 molecules). Therefore, only the force calculation routine is parallelized. A farm is used in the accelerator abstraction to parallelize only the part of the simulation which computes forces on molecules.

Step 1: Identifying Components

Listing 6 shows the pseudo code of the serial version of the force calculation routine in the BasicN2 use case. The first step in the parallelization is to identify the worker components. The user does this by selecting the function call to calc_forces_real and chooses the Identify Component refactoring. Likewise, the user also performs this refactoring for the calc_forces_halo function. The result of applying both of these refactorings is shown in Listing 7.

Step 2: Introducing a Farm Declaration

The next step in the refactoring process is to introduce a new farm declaration that will be used in the parallelization. Component declarations from the previous step need to be Lambda Lifted to the
Listing 6 Sequential version of force calculation routine

```c
void basicN2_calc_forces(void *container, real *U_pot) {
    basicN2 *mc = (basicN2 *) container;
    long i, j;
    real U_pot_tmp = 0.;
    /* Force Calculation Start */
    for(i = 0; i < mc->NumMolecules; i++) {
        calc_forces_real(new ff_task_t(i, j, config.num_threads, mc, U_pot));
    }
    *U_pot /= (2 * 6.0);
    for(i = 0; i < mc->NumMolecules; i++) {
        for(j = 0; j < mc->halo->N; j++) {
            calc_forces_halo(new ff_task_t(i, j, config.num_threads, mc, U_pot));
        }
    }
    /* Force Calculation Ends */
}
```

main function. Currently, this step must be performed manually by the user, although could be easily implemented as a refactoring.

Once the Component declarations are lifted to the main function, the user can start to introduce the FastFlow farm declarations. The user does this by first selecting the first Component declaration in the main function and choosing the Introduce Farm Declaration refactoring, specifying the name of the farm as accelerator1 as a parameter to the refactoring, and also specifying N as the number of workers for the farm. The user repeats this operation, selecting the second worker declaration for haloWorker, introducing another farm, accelerator2. Listing 8 shows the code of the main function where the farm pattern is declared and pointer to it is passed to the force calculation routine for parallelizing it.

Step 3: Adding Tasks to the Task Farm The final refactoring step is to add tasks to the new farms that are introduced in the previous step. In order to do this, the new farm declarations, accelerator1 and accelerator2 must be passed as parameters to the basicN2_calc_forces function. This is currently a manually performed step (although could be easily implemented). Once the farms are added as function parameters, the user first identifies farm1 and selects the first for loop (at line 6 in Listing 6) and chooses the Introduce Farm Tasks refactoring. This refactoring replaces the Component method call to callWorker with farm1.offload, using the parameters to callWorker as parameters to the offload function. This process is then repeated for farm2 and the second for loop at Line 11. Listing 9 shows the addition of the worker threads to the farm.
void basicN2_calc_forces(void *container, real *U_pot) {
    basicN2 *mc = (basicN2 *) container;
    long i, j;
    real U_pot_tmp = 0.;
    /* Force Calculation Start */
    for(i = 0; i < mc->NumMolecules; i++) {
        Component<ff_task_t> w(calc_forces_real);
        w.callWorker(new ff_task_t(i, j, config.num_threads, mc, U_pot));
    }
    *U_pot /= (2 * 6.0);
    /* Force Calculation Ends */
    for(i = 0; i < mc->NumMolecules; i++) {
        for(j = 0; j < mc->halo->N; j++) {
            Component<ff_task_t> w(calc_forces_halo);
            w.callWorker(new ff_task_t(i, j, config.num_threads, mc, U_pot));
        }
    }
}
Listing 8 Introduction of Farm Pattern (to be used for parallelization) inside the main routine

```cpp
int main(int argc, char *argv[]) {
    ... 

    /* Mapping of Threads to Cores — Need to adapt for different machines */
    const char worker_mapping[] = "1,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0";

    /* Pin main thread is to core 0 */
    if (ff_mapThreadToCpu(0) < 0) abort();

    Component<ff_task_t> realWorker(calc_forces_real);
    /* Declaration of FastFlow Farm and Worker Threads */
    ff_farm<> accelerator1(true);  /* Farm in accelerator mode */
    std::vector<ff_node *> workersVector;  /* Vector of Farm worker threads */

    for(int i =0; i< N ; i ++)
        workersVector.push_back(realWorker);
    accelerator1.add_workers(workersVector);
    accelerator1.run_then_freeze();

    Component<ff_task_t> haloWorker(calc_forces_halo);

    ff_farm<> accelerator2(true);  /* Farm in accelerator mode */
    std::vector<ff_node *> workersVector2;  /* Vector of Farm worker threads */

    for(int i =0; i< N ; i ++)
        workersVector2.push_back(haloWorker);
    accelerator2.add_workers(realWorker2);
    accelerator2.run_then_freeze();

    ... 

    LOG(Info, "Starting main simulation loop.
");
    start_timer = timer();
    for(stime = simulation.start_time; stime < simulation.end_time; stime += simulation.dt) {
        ... 
        basicN2_calc_forces(container, &ensemble.U_pot, &accelerator1, &accelerator2);
        ...
    }
    end_timer = timer();
    ...
}
```
Listing 9 Final shape of refactored code of the force calculation routine

```c
void basicN2_calc_forces(void *container, real *U_pot) {
    ff_farm<> farm1, farm2;
    basicN2 *mc = (basicN2 *) container;
    long i, j;
    real U_pot_tmp = 0.;
    /* Force Calculation Start */
    for (i = 0; i < mc->NumMolecules; i++) {
        farm1->offload(new ff_task_t(i, j, config.num_threads, mc, U_pot));
    }
    farm1->offload((void *)FF_EOS);
    farm1->wait_freezing();
    *U_pot /= (2 * 6.0);
    /* Force Calculation Ends */
    for (i = 0; i < mc->NumMolecules; i++) {
        for (j = 0; j < mc->halo->N; j++) {
            farm2->offload(new ff_task_t(i, j, config.num_threads, mc, U_pot));
        }
    }
    farm2->offload((void *)FF_EOS);
    farm2->wait_freezing();
}
```


# Graphical Lasso glasso_comp

One of the use cases selected at SCCH for ParaPhrase is an analysis of waste water processing data to find causal dependency structures among the features of the data set [6]. The method used is a combination of the Granger Causality approach [1] and the Graphical Lasso [4]. Details about this use case, its implementation and parallel performance have been provided in Deliverables D6.4 and D6.5.

Here, we choose the algorithm part denoted as “glasso_comp” in these deliverables as a test case for the C++ refactoring tool. A simplified version\(^1\) of the relevant part of its original sequential implementation is given in Listing 10. After determining a factorizing block structure (connected components) of the matrix to work on, \(S\), a for loop iterates over all components, and prepares variables for a following call of \texttt{gelnet_inverse_mat()}.\(^2\)

\(^1\) the actual code hands over parameters which are not used in the function itself; template parameters have been replaced by the typedef for clarity; parts not important for and not affected by the refactoring are left out (".
. ." parts)

\(^2\)
Listing 11  glasso_comp code fragment after preparation for component introduction

```cpp
struct t_gelnet_ff_task {
    t_gelnet_ff_task(const MatT& _S, double _lambda, MatT& _C, MatT& _P,
        std::vector<arma::uword> _component, ...): S(_S), lambda(_lambda),
        C(_C), P(_P), component(_component), ... {};
    const MatT& S;
    double L;
    MatT& C;
    MatT& P;
    std::vector<arma::uword> component;
    ...
};

void t_gelnet_ff_for_component(t_gelnet_ff_task* task) {
    ...
}

void t_gelnet_ff(const MatT& S, double lambda, MatT& C, MatT& P, ...)
    arma::uword p = S.n_cols; // problem size: number of variables
    // initialization of computation
    ...
    // computation of components (sequential code)
    std::vector< std::vector< arma::uword > > components;
    connected_components( S, L, components );
    arma::uvec block;
    arma::uword bi, j;
    std::vector<t_gelnet_ff_task> tasks;
    for( arma::uword c=0; c < components.size(); c++ ) {
        tasks.push_back(t_gelnet_ff_task(S, lambda, C, P, components[c], ...));
        t_gelnet_ff_for_component(&tasks[c]);
    }
};
```

3.6.1 Parallelization of glasso_comp

**Preparation for refactoring**  The code as shown in Listing 10 does not yet fulfill the preconditions for component identification or even farm introduction refactorings. It has to be prepared. To achieve this, some local variable declarations are moved inside the loop body, the loop body is extracted as a new function (t_gelnet_ff_for_component; Eclipse CDT supports this via an existing sequential refactoring), and the arguments of the new function are extracted into an own task structure (this has to be done manually as there is currently no appropriate refactoring). The resulting code is shown in Listing 11, excluding the newly introduced function (which just contains the former loop body including the moved local variable declarations).

**Step1: Identifying Components**  Next, the refactoring is told the code which should make up the worker component, i.e. line 25 in Listing 11, and it introduces the component wrapper as described in Section 2.3.1.
Step 2: Introducing a Farm Declaration  In the next step, the refactoring is used to introduce the farm declaration before the for loop. In this step, the farm is also filled with workers of the component introduced in the last step “Identifying Components”.

Step 3: Adding Tasks to the Task Farm  Finally, the whole for loop is marked, and the refactoring is used to replace the function/component call in the for loop by farm offloading of the tasks. The resulting code is given in Listing 12.
4. Conclusion

This deliverable has introduced a new refactoring tool for C++ built into the Eclipse development framework and IDE aimed at introducing and tuning parallelism for FastFlow programs, building on the foundational work in D4.1. The refactorings themselves are described in terms of a set of pre-conditions and transformation rules. The pre-conditions must be met in order for the transformation rules to be fired by the tool. Although, the transformation rules are applied automatically, the choice of which refactoring to apply and where in the program to apply it is still a choice driven by the user. We feel that this choice is an important decision, and we discuss this point in more detail below.

The refactorings themselves have been produced following discussions from the Industrial Partners and through study of the use-cases provided in D6.4. In Chapter 3, we presented three realistic Use-Cases that demonstrate the refactorings in Chapter 2. We also outlined a number of new structural and program shaping refactorings that would enhance the user’s experience with parallelising their programs. These refactorings include the provision of introducing a new C++ structure to make the Component introduction easier, and also the lifting of declarations, so that introduced declarations such as the Component can be moved to a different scope in the program.

4.1 Using Refactoring Tools for Introducing Parallelism

Using a refactoring tool to introduce parallelism in a program instead of manually inserting the parallelism has many advantages.

- **Refactoring helps the programmer to program faster.** Using a parallel refactoring tool to introduce skeletons instead of manual insertion means that the programmer has to remember and understand less, allowing them to concentrate on the program design. Also, if the skeleton interfaces change, a refactoring tool can help automate the process of modifying the program to reflect the new interfaces.

- **Refactoring provides correctness-by-construction.** A refactoring tool, by virtue of its design, will not allow a user to break their programs. Introducing the wrong skeleton into a program is simply disallowed, assuming
the preconditions are correct. Furthermore, the refactoring tool will only provide a list of refactorings that are applicable to a selected portion of code, saving time and frustration.

- **Refactoring encourages a consistent software engineering discipline.** Programmers often write a program without thinking about future developers. They may understand their code at the moment that they are writing it, but in a short time the code may become difficult to understand. Refactoring helps the programmer make their code more readable and consistent with a particular style. This helps other programmers to read and understand the skeleton code, making it more amenable for future tuning and modification.

- **Refactoring helps the programmer find bugs.** As parallel refactoring tool helps improve the understanding of algorithmic skeletons, it also helps the programmer to verify certain assumptions they have made about the program.
Bibliography


