Project no. 288570

PARAPHRASE

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

Applications Porting Report
D6.4

Due date of deliverable: March 31, 2013; extended to April 30, 2013

Start date of project: October 1st, 2011
Type: Deliverable
WP number: WP6
Task number: T6.3

Responsible institution: SCCH
Editor and editor’s address: Holger Schöner, SCCH, Software Competence Center Hagenberg GmbH

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

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

This deliverable is a description of the porting work done in the ParaPhrase project. Its aims, the status, issues encountered, and next steps are discussed.

After a slow start due to the early project start and the need to acquaint the developers with technology new to them, communication and progress have intensified over the last months. Now, with a delay of approximately one month, the goals for D6.4 have been reached, and all use cases are available in a basic or even complete parallelized version. Most of the technology currently available in ParaPhrase is applied in one or the other use case, notable exceptions so far being GPGPU and distributed patterns. They can now serve as basis for technology evaluation, and feedback has already been provided to technological workpackage partners.

While conceptually simple to apply, the pattern based approach of ParaPhrase has shown some issues in its current implementation state when practically applied to the use cases. One main issue is the complexity of porting work, especially on the C++ side, which is currently being addressed by work on refactoring for C++. Another issue was the need for explicit synchronization and intensive fine tuning of communication among workers (threads), to achieve reasonable scaling. Partially this optimization work is still in progress, and ParaPhrase will have to show in future how refactoring, application specific and heterogeneous patterns, and static and dynamic mapping can address this.

This deliverable is accompanied by two further ones. D6.3 (confidential to project partners and commission services) provides information about access to ported code and the evaluation platform. D6.5 contains the evaluation of the use cases for which porting is described here.
Contents

Executive Summary ................................................. 1

1 Introduction .................................................. 3

2 Haar Transform – ESL ........................................... 5
   2.1 Summary of Use Case ....................................... 5
   2.2 Porting ....................................................... 7
       2.2.1 1D Haar Transform .................................. 8
   2.3 Discussion and Conclusion ................................. 10

3 Molecular Dynamics – HLRS .................................. 11
   3.1 Summary of use case ....................................... 11
   3.2 Porting ....................................................... 13
   3.3 Discussion and conclusion ................................. 15

4 Waste Water Processing – SCCH ............................... 17
   4.1 Summary of use case ....................................... 17
   4.2 Porting ....................................................... 17
   4.3 Discussion and conclusion ................................. 24

5 Discrete Optimization – SCCH ................................. 25
   5.1 Summary of use case ....................................... 25
   5.2 Porting ....................................................... 26
   5.3 Discussion and conclusion ................................. 32

6 Conclusion ..................................................... 33
1. Introduction

This Deliverable documents the status of work regarding task T6.3 “Porting and Annotation of Applications” in workpackage WP6 “Use Cases, Requirements and Evaluation”: the approach taken, the experiences gained, and the decisions reached about future work. It is based on deliverable D6.2 and the use cases selected therein by the partners. After this introduction, there is a section for each of the use cases, giving a short summary, the experiences gained during porting, current status, and a discussion and conclusion for the use case. A final section provides a summary of the experiences, conclusions and suggestions of the partners. Partners contributing most to this deliverable are SCCH (Software Competence Center Hagenberg, lead), ESL (Erlang Solutions Ltd.) and HLRS (High Performance Computing Center Stuttgart).

Work on the use cases covers several aspects. Besides representing a typical and important problem for the respective partner, tools and technology provided by technical workpackage partners are exercised and evaluated, and porting experience is used for comparison with alternative parallelization approaches. Exercising technology entails the use of different ways to express parallelism, i.e. different pattern hierarchies used for parallelization, sometimes in the same application. This is at the core of the ParaPhrase approach, and the use case sections will provide details. Implementation variants using different pattern hierarchies test the patterns and their implementation, and it is also possible to obtain performance information, which might be used to improve work on static and dynamic mapping. The use cases also provide examples and test cases for the work on refactoring tools, with both sequential and parallelized versions, and by using different pattern hierarchies to perform the same computations. Additionally, the implementations are written to exercise ParaPhrase technology on different hardware and show potential for scaling in different environments.

Taking all these aims into account requires multiple versions of the software and iterations of development. As currently work in the technical work packages is not finalized, the use cases will be extended in future to make use of newly available tools and technology. The basis for first usage of tools and libraries was made available in summer 2012 by initial implementations of WP2 partners (FastFlow), and by the availability of first pattern implementations and refactoring tools in Erlang. Since then, WP6 partners have started their porting efforts, and the main objective of the initial work leading to this deliverable was in building experience
with ParaPhrase’s pattern based approach, and in porting of applications to Fast-Flow and the developed Erlang infrastructure. This status of work is described in the following sections. Generally, partners have by now completed the first iteration of parallelization of their respective use cases, testing one or several ways of parallelization, often for a simplified version of the final application. Deliverable D6.5, due at the same time as this deliverable, reports upon performance and efforts spent. Extension to support the full application and exercise and evaluate newly available technology will continue during the coming months, until the end of the project in autumn 2014. At the end of the project, reports will be produced in Deliverable D6.6 about final status of use case porting and evaluation.

This deliverable is also complemented by deliverable D6.3 “Ported Applications”, which provides information about how to access, use and test the code developed so far. It also describes how to access the cluster machine set up at Robert Gordon University for ParaPhrase and which is selected as the main development machine and evaluation target for ParaPhrase use cases and for delivery of the code.\footnote{For a description of that cluster see deliverable D6.5.} As a consequence, that deliverable is restricted as confidential to project partners and commission services.
2. Haar Transform – ESL

2.1 Summary of Use Case

In mathematics, the Haar wavelet [8] is a sequence of “square-shaped” functions that can be used to approximate any square-integrable real function. An important example of the use of Haar wavelets in Computer Science is the Discrete Haar Transform, which is heavily used in image and signal processing. The Discrete Haar Transform consists of applying the operation

\[ y = T x T^T, \] (2.1)

for an input vector \( x \) and a fixed Haar matrix \( T = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \), producing the output vector \( y \). A 1-dimensional Haar Transform, where \( x \) is a pair of real numbers, is used in signal processing (e.g. sound compression). A 2-dimensional Haar transform, on the other hand, where \( x \) is 2x2 matrix, is used for image compression, as for each 2x2 matrix of pixels \( x \), it gives a 2x2 matrix of pixels \( y \) as a result, where most of the energy of \( x \) is contained in the top left pixel of \( y \). Applying the Discrete Haar Transform to an image (signal) consists of splitting the image (signal) into 2x2 subimages (pairs of elements) and applying the Haar Transform to each subimage (pair). See Figure 2.1 for the example of an image before and after the Discrete Haar Transform is applied.

We consider two use cases of applying the Discrete Haar Transform:

1. audio compression, where a 1D Haar Transform is applied to a stream of audio files; and,

2. video compression, where a 2D Haar Transform is applied to a stream of images captured by a camera.

For each variant, it is possible to observe two sources of parallelism:

1. outer parallelism – The same operation (Discrete Haar Transform) needs to be applied to a set of independent inputs (a stream of audio files or images). This is relatively coarse-grained data-parallelism, which can be implemented using a task farm; and,
2. *inner parallelism* – The application of Discrete Haar Transform to one audio file or image exhibits a lot of fine-grained data-parallelism, where the same operation is applied to a set of independent subvectors or submatrices.

In this deliverable, however, we only present the parallel versions of the two use cases where the outer parallelism is exploited, using a *Farm* Erlang skeleton. We leave exploitation of inner parallelism as future work, where we plan to use an *OpenCL* kernel for dealing with fine-grained data-parallelism in applying the Discrete Haar Transform to one audio file/image, and to use Erlang/OpenCL bindings to execute this kernel within the Erlang Virtual Machine. This *OpenCL* binding will be wrapped inside a skeleton structure where both the *OpenCL* version and the Erlang version can be operated in parallel, allowing us to study the effectiveness of Heterogeneity in Erlang.
2.2 Porting

In this section we illustrate the process of porting the sequential Haar use-case to the Erlang parallel skeleton library, Skel, which was developed as part of ParaPhrase under WP2 [4]. The basic program comprises a three-stage function composition. In the first stage, a file is read that comprises either a video or an audio file. These images or audio files are then passed to the second stage, where a 1D Haar Transform is applied to the audio samples, and a 2D Haar Transform is applied to the image samples. Finally, in the third stage, the transformed audio and images are sent across a network to a server for processing. The high-level structure of the algorithm, can be seen in Figure 2.2. In order to parallelise this algorithm, it was decided to introduce a task farm for the middle stage, Haar, creating a Skeleton call to a Skel task farm, where the number of workers is controlled by the user via an input argument. The Skel implementation uses a basic accelerator mode to distribute tasks to the workers from the emitter. This means that tasks are distributed to idle workers, rather than in round robin fashion. Each worker computes the Haar Transform over the input and returns the results to a Skel collector. Both the Skel emitter and collector are hidden from the user, and are provided by the basic library framework. Figure 2.3 shows how this skeleton applies to the Haar Transform. In the figure, we have a stream of input elements, \(X_1, X_2, X_3, \ldots, X_4\), which could represent images or audio files. Each element is passed through an emitter, where the element is sent to an idle worker process. Each worker process applies the Haar Transform operation to the element in parallel, sending the result to the collector. The result of the whole operation is a list of new transformed elements, \(X_1', X_2', X_3', \ldots, X_4'\).
2.2.1 1D Haar Transform

The porting of the 1D Haar Transform proceeded in two stages, where the original algorithm is shown in Listing 1. Here the sequential 1D Haar Transform, sequential1d, is defined as mapping a function, haar_1d_wrapper, over a list of Vectors.

Stage 1: Introducing a Task Farm

Introducing parallelism into the program was done by first identifying the sub-expression in the program that generated the output list, and where each operation on the list could be computed in parallel. In the 1D Haar Transform example, the operation haar_1d_wrapper is converted into a task farm, where each worker computes the 1D Haar Transform for an input vector. The result of the task farm is a list of transformed vectors, as illustrated in Listing 2.

Here, the program can be broken down into a number of key components:

- `skel:run` denotes a call to the top-level run function in the skel library, where the parameter to run is a (nested) Skeleton;
- `farm` denotes a farm skeleton;
- `seq` denotes the workers of the farm skeleton are the sequential function, 1d_haar_wrapper;
- `24` denotes the number of farm workers; and, finally,
- `Vectors` is the input list of tasks.

Stage 2: Chunking

While using a task farm for the 1D Haar Transform creates a reasonable amount of parallelism, the parallelism is too fine-grained and the program does not scale as we would typically expect. This is a common problem in the early stages of writing parallel programs. To combat this, we introduce chunking to the task farm, allowing us to group together a number of small tasks into one
larger parallel task, where each parallel thread operates over a sub-list rather than just one element. We want each worker to be busy, so we chunk by groups of 4 elements, \((2048/4 = 512)\) tasks for each worker). By chunking in this way, we also decrease the communication costs, and reduce parallel overheads. Chunking can generally be achieved in a variety of different ways. In our example, we modify the task farm, manually refactoring it to a pipeline with a partition and combine stage, as illustrated in Listing 3.

We also have to introduce two new functions, partition and combine, as illustrated in Listing 4. Here, combine simply takes a list, where the head of the list, \(X\), is also a list, and appends \(X\) to the combined tail of the list, \(Xs\). Partitioning a list is implemented by the Partition function, where a new list is created that is ChunkSize elements in length. This sublist is then appended to the remaining sub-lists of the input list, List. The partitioning terminates when no new sublists can be created.

### 2D Haar Transform

The porting process for the 2D Haar Transform proceeded in a similar way to the 1D Haar Transform, where the sequential algorithm is shown in Listing 5.

Here, a function, \(\text{haar}_2d\_wrapper\) is applied to each element, \(I\), of the list of input images, \(\text{Images}\). Porting this code to use a skel task farm comprised of rewriting the above code into a call to the skel library, introducing a farm skeleton,
where the work function is `haar_2d_wrapper`, as shown in Listing 6.

Here we introduce a call to `skel:run`, which is the top-level skeleton call, parameterised by a nested skeleton. In our example, we use the `farm` skeleton, with 24 workers, and each worker is a sequential function, `haar_2d_wrapper`. In this example, it is not necessary to employ chunking, as the tasks are already large enough to give sufficiently large computation, without saturating the system with an abundance of parallel tasks.

### 2.3 Discussion and Conclusion

The porting process proved to be reasonably straightforward, most likely as a result of the use-case being relatively simple. However, it does show some interesting ideas for refactoring. Clearly, introducing the task farm in Skel could be automated by a refactoring tool, and is indeed already implemented and discussed in D4.1, where a sub-expression is selected for farming and the user gives the number of workers, resulting in a new task farm introduced in the code. A chunking refactoring has also been introduced and described in a recent paper by Brown et al. [4]. However, the user is left to manually derive the partition and combine functions themselves, although it is probable that these functions will be the same in most cases, and so the partition and combine functions should be added to the Skel library. Furthermore, the process of deciding which refactoring to perform could be automated by integrating cost models into the refactoring tools. Such cost-models could automatically profile a selection of the code, and then use this profile information to instantiate cost models, therefore informing the user on the possible “best” refactoring choices to make. In addition to this, such cost model instances could also be used to warn a user if they are about to refactor their code into a less optimal one. For ParaPhrase, we intend to develop refactoring tools that integrate performance models, providing an interactive tool chain for developing effective parallel programs. In the future, we wish to extend this use case by introducing an Erlang wrapper for OpenCL, allowing us to exploit fine-grained parallelism on a GPU architecture. We also intend to extend the Skel library to allow for heterogeneous skeleton integration. Both of these will be developed under ParaPhrase as part of WP4 and WP3.
3. Molecular Dynamics – HLRS

3.1 Summary of use case

Molecular dynamics (MD) simulates interaction between molecules [1]. In principle, given an initial set of positions and velocities of molecules, the following time progression of a set of interacting molecules may be ascertained. After the system is initialized with the initial positions and velocities of molecules, calculation of forces is done on all molecules in the system. Finally Newton’s equations of motion are integrated to advance the positions and velocities of molecules. The simulation is advanced until the computation of time evolution of system is completed for the required length of time.

MD code is used for basic research into High-performance computing. One code is used for multiple data structures, algorithms and parallelization strategies. Our in house MD code (CMD – Computational Molecular Dynamics) implements two algorithms for the computation of interactions between the molecules. These are the two use cases ported to the FastFlow parallel programming model and hence their performance is evaluated using FastFlow.

The first use case is named as CMD BasicN2, where the interaction of each molecule is computed with all the molecules in the domain. It is a compute intensive and highly data parallel problem. The second use case is named as CMD BasicN2MoleculeBlocks.
**Molecule Blocks**, where the domain (a large molecule container) is decomposed into molecule cells and then molecules are distributed among these molecule cells. Figure 3.1 shows the basic structures of **BasicN2** and **Molecule Blocks** use cases. In the **Molecule Blocks** use case, the interaction of molecules in one cell is computed only with molecules in the same cell and its neighbour cells, unlike the **BasicN2** use case. Therefore, it reduces the computation effort to the complexity of $O(N)$ compared to the **BasicN2** use case which has $O(N^2)$ complexity. It further reduces the computation and communication with the other cells in the system by calculating the interaction with only 13 neighbour cells instead of 26 neighbour cells (for 3 dimensional space). This is accomplished using Newton’s 3rd law of motion which makes it possible to avoid the calculation of force twice among a pair of molecules. So when an interaction between a pair of molecules is computed, the forces of both molecules are modified. Figure 3.2 shows an example for the calculation of interaction of molecules in 2 dimensional space for **Molecule Blocks** use case, where the interactions of molecules of a centered cell are computed with only its 4 neighbour cells (yellow coloured) instead of all 8 neighbour cells. When the interactions between molecules of two cells are computed, forces of molecules from both the cells are modified. Hence, as shown in the figure, the centered cell modifies only the forces of its molecules and molecules of its right neighbour cells (yellow coloured). Whereas when the orange cells compute the interactions with their 4 right neighbour cells and modify the forces of molecules of these 4 neighbour cells, then this results in achieving the computation of interactions of molecules of the centered cell with all of its 8 neighbour cells. Although the use of Newton’s 3rd law of motion lowers the computation effort, it raises the requirement of synchronization mechanism when modifying the forces of molecules of a neighbour cell. Therefore the synchronization mechanism is considered in **FastFlow** implementation of this use case.

Currently for the **BasicN2** use case, we utilize the **Farm** pattern of **FastFlow**. For the **Molecule Blocks** use case, we also have exercised the **Farm** pattern but with a locking interface for synchronization among worker threads.
Listing 7 Farm pattern used in BasicN2 (main.c)

```c
// Pseudo Code (main.c)

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

    /* Parsing of input arguments – identifies number of worker threads */
    ff_farm<> accelerator(true); /* Declare FastFlow Farm in Accelerator Mode */
    std::vector<ff_node *> workersVector; /* Declare vector of Farm worker threads */
    /* Add workers to Farm in the grid generator routine */
    grid_generator(&target_ensemble, SC, &domain, psp, &accelerator,
                   &workersVector);

    ... start_timer = timer();
    Simulation loop {
        Pre–Force Integration();
        Force–Calculation(); /* Parallelized using FF Farm in Accelerator mode */
        Post–Force Integration();
        ... 
    }
    end_timer = timer();
    return 0;
```

3.2 Porting

Implementation of CMD BasicN2 use case In the BasicN2 use case, 99.91% of the simulation time is spent on the force calculation routine. Therefore only the force calculation routine is parallelized using the FastFlow programming model. A FastFlow Farm is created with the given number of worker threads before the simulation is started. The number of threads is controlled using the input argument, ‘−−num−threads’. The Fastflow Farm is used in the accelerator abstraction to parallelize only the part of simulation which computes forces on molecules. Once the simulation is started, inside the basicN2_calc_forces routine the Farm is launched and the molecules are divided among its worker threads to compute the forces acting on them. Each thread computes the forces and accumulates the potential energy of its molecules. After distribution of molecules for force calculation, the main thread is frozen to wait for other threads to finish the computation routine. Once all threads have finished their task, a reduction operation is performed by the main thread to calculate the sum of potential energy of all the molecules in the system. To avoid the overhead of launching the farm again and again, it is not terminated, but rather the wait_freezing() method of the farm is used, so that the same farm can be used in the later simulation steps. This is shown in the pseudo code given in Listings 7 and 8.

Implementation of CMD Molecule Blocks use case In the Molecule Blocks use case, most of the simulation time is spread among multiple routines (as shown
Listing 8. *Farm* pattern used in *BasicN2* (basicN2.c)

```c
// Pseudo Code (basicN2.c)

void basicN2_calc_forces(  
    void *container,  
    real *U_pot,  
    ff_farm<> *accelerator) {
    basicN2 *mc = (basicN2 *) container;
    real U_pot_tmp = 0.;

    if (accelerator->run_then_freeze() < 0) {
        error("running accelerator\n");
    }

    for(int i = 0; i < num_threads; i++)
        accelerator->offload((void *) FF_EOS);
    accelerator->wait_freezing();

    /* Reduction Operation for global sum of Potential Energy */
    for(i = 0; i < config.num_threads; i++)
        U_pot_tmp += U_pot_worker[i];
    *U_pot += U_pot_tmp;
}
```

Table 3.1: Profile of CMD *Molecule Blocks* use case.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Time spent on routine (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Force Calculation</td>
<td>41.74</td>
</tr>
<tr>
<td>Kinetic Energy Calculation</td>
<td>15.36</td>
</tr>
<tr>
<td>Pre-Force Integration</td>
<td>10.43</td>
</tr>
<tr>
<td>Post-Force Integration</td>
<td>4.22</td>
</tr>
</tbody>
</table>

In Table 3.1. Therefore to achieve better speed up, the most expensive routines of simulation are parallelized using the *FastFlow* programming model. Like the *BasicN2* use case, a *FastFlow Farm* is created before the actual simulation starts. Once the simulation is started, inside each of the expensive routines of simulation the main thread launches the farm. After the farm is launched, the molecule cells are distributed among worker threads to perform the parallelization of a task specific to the routine to be parallelized and then the main thread waits for other threads to finish their computation by executing the *wait_freezing()* routine. As the routines in the simulation can not be executed in parallel, rather each routine is parallelized itself, and this can be done by using the single farm to parallelize multiple routines. This is done by passing a pointer to the task structure to the worker (or service) routine of the farm. The task structure has one member that
identifies which routine is required to be parallelized by the farm (as shown in figure 6.4.4). Inside the worker (or service) routine of farm, switch cases are used to parallelize multiple routines of simulation. After all threads finish the computation of their specific task, the main thread performs the reduction operation (if required) to calculate some parameter of the complete system.

Synchronization in the force-calculation routine is achieved using pthread locks. pthread_mutex_trylock routine is used to acquire the locks on two cells among whom interaction is to be computed. The use of the non-block lock requesting routine allows to avoid the deadlock among threads. Such that if a thread acquires the lock on a cell and it does not get lock on its neighbour cell, it releases this lock so that this lock can be acquired by other worker thread waiting to acquire lock on this cell.

3.3 Discussion and conclusion

The porting of use case CMD BasicN2 to the FastFlow programming model was as easy to do as to port this use case to OpenMP Programming model because it required only a single compute intensive routine of the simulation kernel to be parallelized. For this compute intensive routine, we create a FastFlow Farm with a fixed number of worker threads once and reuse the same set of worker threads when this routine is called in later simulation steps. To achieve this we only needed to move the code inside the compute intensive routine to the service routine of worker class. However we faced an issue while porting the CMD MoleculeBlocks use case to FastFlow as we wanted to re-use the same set of worker threads for parallelizing the multiple routines of this use case. To achieve this, we needed to move all the code of these routines to the service routine of the worker class and there we were able to parallelize a specific routine depending on the input argument to the service routine. Hence for a large modular code where we have multiple routines in some file and if these routines can not be executed in parallel due to data dependencies among them, we need to move the code of the multiple routines to a single service routine or to have a dedicated farm for each of the routines to be parallelized. Having a dedicated farm for each routine had an overhead of farm creation in our case due to computation per thread being so fine grained. However for an application where computations per thread are not so fine grained, it is suggested to have a dedicated farm for each routine to be parallelized (i.e. move the code of each routine to be parallelized to its respective worker class’s service routine). For our use case where the computations are fine grained, a lot of code structure needed to be changed for porting it to FastFlow (i.e. moving the code to be parallelized to a single service routine).

For the BasicN2 use case, we currently use the Farm pattern of FastFlow, although we intend to employ the Map pattern for this use case in the future. Making use of the Farm pattern in accelerator mode was more straightforward to start with. Similarly, for the Molecule Blocks use case, we currently use the Farm pattern of
FastFlow, but with a locking interface. However in future, we also plan to employ the Map pattern for this use case, because it will be more suitable for this kind of application where a reduction operation is required after each thread performed its task. Finally we would like to exercise the Stencil Pattern which exactly describes the pattern of this use case where an elemental function can not only access a single element of input collection but also a set of ‘neighbours’. For a stencil pattern, the programmer should not explicitly take care of the synchronization mechanism when accessing the neighbour cells: the implementation of stencil pattern should take care of this.
4. Waste Water Processing – SCCH

4.1 Summary of use case

One of the two use cases selected at SCCH in Deliverable D6.2 is an analysis of waste water processing data to find causal dependency structures among the features of the data set [9]. The method intended to be used is a combination of the Granger Causality approach [2] and the Graphical Lasso [6]. The implementation is based on an existing sequential version of the algorithm in SCCH’s C++ Machine Learning library mlpp. Because of the size of the data set with over 5000 features, and the necessity to multiply this number (by about 10 to 50 for this use case) when using the Granger Causality approach, the problem becomes too big for sequential analysis.

A parallelization of this approach is possible on multiple levels ranging from completely independent blocks of computations via coupled updates to learned weights, to linear algebra operations, and it is part of the use case to use ParaPhrase technology to evaluate the appropriate levels or mix of these for given architectures. Target architectures for this use case are mainly multi-core shared memory machines, because of the mainly fine grained communication structure and computational blocks. Parts of the algorithm might be suitable for parallelization on GPGPUs (e.g. covariance matrix computations or other linear algebra operations).

The main goals are on the one hand making application of the use case feasible computation-wise and with reasonable implementation effort, and to be an application of and motivation for ParaPhrase technology, on the other hand. Relevant technologies include patterns for shared memory multi core systems, mainly Farm, possibly Map/Reduce and GPGPU patterns; refactoring for parallelization and switching between parallelization strategies; and target machine mapping of possibly nested pattern hierarchies.

4.2 Porting

Sequential Algorithm As a basis for understanding the parallelization, we first introduce the sequential algorithm. We will concentrate on the structure of the problem and abstract from the mathematical details unimportant for parallelization.
The sequential implementation is available in two variants (working on the data correlation matrix, or directly on the data matrix) in the src/mlpp/ggm/gelnet.hpp header file and src/mlpp/ggm/gelnet_cov.cpp and src/mlpp/ggm/gelnet_mat.cpp implementation files, in functions t_gelnet and t_gelnet_mat, respectively.\footnote{See deliverable D6.3 for a description of access to the source code on the RGU cluster machine.}

The Graphical Lasso gives an estimate of the correlation matrix $C$ of a given data set, and of its inverse $P = C^{-1}$. Additionally, a “Lasso” regularization is applied to the elements of $P$, i.e. the optimization cost function contains a term penalizing the magnitudes of the entries in a linear way. This “Lasso” regularization is known to yield sparse solutions, with many 0 values in the estimated $P$. Furthermore, the inverse of a covariance matrix resulting from real world data usually only has non-zero entries for rows and columns for which the corresponding features are directly dependent \cite{6}. This makes the Lasso a good regularization term for estimating the graph structure from noisy data. The strength of regularization is called $\lambda$, in the Deliverables as well as in the code (\texttt{lambda}).

The algorithm builds representations for $P$ and $C$ in the following way: It first initializes $C$ by the data correlation matrix $S$, and partitions it into blocks by searching for Connected Components, in which all features are connected by chains of covariance above the used regularization parameter value $\lambda$. For these, the entries can be determined independently in the following, which can save much time with respect to estimating all entries jointly.

Then, for each block, the regularized inverse $P$ and the corresponding $C$ are computed. This process iterates over each feature in the block, solving a regression problem to estimate the row/column in $P$ corresponding to the feature, based on the other elements of $P$. This iteration is performed until convergence.

The regression problem itself is Lasso regularized as well, and solved using the Coordinate Descent algorithm. This is another loop to convergence, where in each iteration each element of the parameter vector is estimated based on the current estimates of the other elements.

So in total, there are 5 nested loops. One is an iteration over independent components, which is easily parallelizable. Two iterate some process until convergence and are not parallelizable. The remaining two iterate over features, and are parallelizable to a certain data dependent degree \cite{3}. Where features are uncorrelated, the computations for these are independent and parallelizable without a problem; but the stronger the correlations between features, the more the convergence of the loops is reduced by assuming independence and computing the corresponding parameters in parallel.

**Parallelization of “cd_cov” and “cd_mat”** Until now, we have concentrated parallelization on three of these loops. We started by parallelizing the innermost of the mentioned loops, i.e. the iteration over features in the inner regression. We used the approach described in \cite{3}, but adapted it to use the pattern based approach of ParaPhrase and wrote the code using the library FastFlow. The
Figure 4.1: Basic pattern hierarchy for the “cd_mat” implementation. Light blue blocks show threads, green block is a data structure. Blue arrows show communication of the task data structure, dashed black arrows show synchronization (wait_freezing()), green arrows show read access (from the whole vector), and red arrows write access (concurrent, but without conflict to separate parts of the vector). The main thread performs a (small sized) reduce operation after waiting.

code for these two variants is available in files src/mlpp/linear/elnetspace..hpp,cpp, in functions t_elnet_cov_ff (called “cd_cov” approach in the following, because the elastic net Lasso implementation uses Coordinate Descent for optimization) and t_elnet_mat_ff (“cd_mat”).

This was our first experience with FastFlow, and we started with an implementation similar to the tutorial code of the Farm acceleration pattern. We encountered performance problems in the beginning, which led us to try out different variants of the Farm pattern. We used collectors for merging the computed updates, or performed them in the main thread doing the offloading, or used pthread synchronization to perform updates in the workers directly; we tested offloading to a Farm accelerator as well as calling the Farm pattern directly; and we experimented with pinning. WP2 partners in Pisa supported us in finding a good FastFlow formulation. But in the end, the performance problem turned out to be a performance bug in the used linear algebra library, Armadillo, which needed 1600 ms for summing up matrix elements, where later a manual loop did the same in 100 ms. This computation was done in the sequential initialization of the regression, and because of the small size of the computations involved in the parallel part, no speedup was possible then.

Beside the use of the manual loop, we also moved from optimization on the correlation matrix (which we call the “cd_cov” variant), to a variant (“cd_mat”)
Listing 10 Farm pattern used in “cd_mat” (file src/mlpp/linear/elnet.cpp)

```cpp
void t_elnet_mat_ff(in X, in y, in lambda, out w, ... ) { 
    arma::uword p = X.n_cols; // size of problem
    arma::uword nworkers = ...; 
    // initialization of computation
    ... 
    // initialize FastFlow farm pattern
    ff :: ff_farm<> farm(true); // set accelerator mode
    std::vector<boost::shared_ptr<ff::ff_node> > workers;
    for (arma::uword i=0; i<nworkers; ++i)
        workers.push_back(new ff_elnet_mat_worker(X, y, &w, lambda, ...));
    if (farm.add_workers(workers) < 0) ... 
    std::vector<ff_elnet_mat_task> tasks(p);
    for (arma::uword iter = 0; iter < max_iter; ++iter) {
        for (arma::uword j=0; j<p; ++j ) farm.offload(&tasks[j]);
        farm.offload(EOS);
        farm.wait_freezing();
        for (arma::uword j=0; j<p; ++j) {
            // perform reduction step
            ... 
        } 
        if (/* convergence achieved */) break;
    }
    farm.offload(EOS);
    farm.wait();
}
```

working directly with the data matrix. This variant has more computations to do for each feature and thus is slower at the moment (as the data contains more samples than features), but avoids a direct representation of the covariance matrix. This will be important later on, when we combine the Graphical Lasso with the Granger Causality approach. There, time lags are considered between features, and the number of features to be analysed by Graphical Lasso is multiplied by the number of considered lags (which will be between 10 and 50, for the waste water use case). The resulting correlation matrix would have up to $5000 \times 50 \times 5000 \times 50 > 60$ billion entries, making the “cd_mat” version much more attractive than the “cd_cov” variant. Because the second version has more work to do for each feature, as a side effect we also obtain better speedups by parallelization. We increased this effect by introducing chunking in the implementation. This is one of the two parallelization approaches for which we report performance in Deliverable D6.5.

Figure 4.1 shows the final choice of pattern (Farm) and variant (accelerator usage, no collector, synchronization using wait_freezing()), reduce operation in main
Figure 4.2: Basic pattern hierarchy for the “glasso_comp” implementation. *Farm* pattern structure is very similar to “cd_mat” in Figure 4.1. But tasks are completely independent and no read or write conflicts occur, and no reduction step is necessary after the *Farm* completes. Tasks can be of much different size in this case.

thread after synchronization). Reads of concurrent workers might obtain outdated or current values of the vector, depending on whether the write has taken place before. This way we can avoid expensive synchronization between workers, which is traded off for a potential increase in iterations necessary for convergence (but this is unavoidable for parallelized coordinate descent anyway). During runtime of the workers, the main thread and emitter should consume little to no time, because the main thread is sleeping until all workers are completed after the trivial offloading is performed, and the emitter should only fill the workers’ queues and then also go to sleep after receiving an end-of-stream token. All tasks should require the same amount of work. This implementation is also outlined in the pseudo code in Listing 10.

We have performed an alternative implementation of “cd_mat” using OpenMP parallelization. Our performance evaluation indicates comparable performance of Fastflow and OpenMP ports. Details are given in D6.5, for performance as well as porting effort. Regarding latter, the OpenMP port was somewhat quicker to implement as only few lines had to be changed/introduced. But we expect this difference in porting effort to be reduced or eliminated with appropriate refactoring tools later in this project.

**Parallelization of “glasso_comp”** The second variant used for performance reporting in D6.5 is “glasso_comp”, which is implemented in function t_gelnet_ff in file src/mlpp/ggm/gelnet_ff.cpp. The pattern hierarchy is shown in Figure 4.2, and pseudo code is provided in Listing 11.

This is the parallelization of computation on the independent blocks/components of the matrix. It is a relatively prototypical use of the *Farm* pattern, which has also been used as one of the first examples for C++ refactoring in cooperation
with WP4. As the components are located in completely disjunct parts of the matrix, there is no synchronization necessary between parallel workers. Parallelization works well using this approach, if the considered data set has several medium sized components, more specifically not one dominating one. As computation time for a component is at least (depending on the necessary iterations for convergence) of complexity $O(n^3)$, with $n$ being the number of features in a block, computing a dominant component might take longer than computation of all the others together. The possibility for dominant components motivates the need for parallelization on multiple levels of the Graphical Lasso algorithm, for later enabling the adaptation of the appropriate strategy to data characteristics.

“glasso_comp” has been used to test the Introduce Farm refactoring of the C++ refactoring tool developed in WP4. Resulting code is very similar, with the main differences to manually ported code being the use of smart pointers in the latter instead of raw pointers, and a different way of wrapping the worker function in a component. As expected, experiments have shows that these do not seem to have any influence on performance, and we do not report separately on performance of the semi-automatically refactored version in D6.5.

Like “cd_mat”, this code has also been ported using OpenMP. Performance evaluations are not yet complete, so we cannot yet discuss performance and speedup for that port in D6.5. Initial results show in general similar characteristics of Fastflow and OpenMP ports, with sometimes one and other times the other port having slightly better performance, depending on the target platform.

Parallelization of “glasso_mid” We pursued a fourth approach for parallelization, named “glasso_mid” in the following. This consists of a parallelization similar to the one for Coordinate Descent used for “cd_cov”, but one level higher. It parallelizes the loop over features for computation of the regularized inverse P. The motivation here is the similarity to the “cd_cov” approach, and the fact that the workers have much more work to do for each feature (the Lasso regression problem for that feature). We experimented with different ways of distributing the available workers between “glasso_mid” for the dominating component(s), and “glasso_comp” for the medium/smaller components. Unfortunately so far we could not come up with a distribution of work and an order of processing of the features, that would lead to a speedup. The problem seems to be that by working on features of the dominating component in parallel, too much information about the concurrent feature computations is missed, leading to an increase in iterations necessary for convergence. This increase more than undoes speedups by parallelization, so far. So we do not report on performance for this approach explicitly.

**Future work** The Graphical Lasso is a regularized optimization, and the question arises which value for $\lambda$ is the appropriate one, and which features are important for different settings of $\lambda$. Usually this is determined by model selection, i.e. trying out different values (the regularization path), evaluation of each solution on test data,
Listing 11 *Farm* pattern used in “glasso_comp” (file src/mlpp/ggm/gelnet_ff.cpp)

```cpp
void t_gelnet_ff(in S, in lambda, out C, out 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 );
        // initialize FastFlow farm pattern
        arma::uword nworkers = ...
        ff :: ff_farm<> farm( true ); // set accelerator mode
        std::vector< boost::shared_ptr<ff::ff_node> > workers;
        for (arma::uword i=0; i<nworkers; ++i)
            workers.push_back( new t_gelnet_ff_worker() );
        if ( farm.add_workers(workers) < 0 ) ... 
        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], ...));
        }
        if ( farm.run() < 0 ) ... 
        for( arma::uword c=0; c < components.size(); c++ ) { 
            farm.ofload(&tasks[c]);
        }
        farm.ofload(EOS);
        farm.wait();
    }
```

and choosing the one yielding best performance on that test data. In principle, this is an easily parallelizable algorithm again, as the computations for each $\lambda$ are completely independent. Two issues have to be considered, though. Firstly, the computations become more and more complex as $\lambda$ decreases, because the solution becomes less sparse. This might lead to the computations for the lowest value(s) for $\lambda$ dominating the others. Secondly, solutions for similar values of $\lambda$ tend to be similar as well. So there might be an advantage in coupling the computations for neighbouring values of $\lambda$, which would have an impact on the parallelization strategy as well.

In the basic form of the Graphical Lasso, first the correlation matrix has to be computed. This can be a considerable part of the whole computation (depending on $\lambda$ and the resulting sparseness of the solution). So we plan to apply parallelization to this function as well. The *Map/Reduce* pattern will probably be a good way to implement this, where available on GPGPUs, otherwise on multiple CPU cores. In the end, when the Graphical Lasso is sufficiently performant, it will be adapted to take into account the Granger Causality principle in time series. An efficient implementation of this combination will probably give further potential for parallelization.

Currently we are also continuing our work on *OpenMP* versions of the code
so far parallelized with FastFlow. This is intended to obtain experience for the comparison of implementation effort between ParaPhrase and other technology, and to derive other dis-/advantages.

4.3 Discussion and conclusion

The problem encountered with the performance bug in Armadillo took more time than anticipated for the first parallelization variant, so the current status of implementation, and in consequence D6.5, cannot yet give a comprehensive overview of the speedup possible for Graphical Lasso using a combination of “cd_mat” and “glasso_comp”. This will be the next focus, followed by exploitation of further parallelization opportunities relevant for the use case, and for exercising ParaPhrase technology (C++ refactoring, Map/Reduce pattern, GPGPU support, work on balancing parallelization on different levels as appropriate for data).

Specifically regarding refactoring, “glasso_comp” will be a test case for developing the refactoring tools, in cooperation with WP4. So far, introducing the necessary wrapper code for FastFlow patterns has been a considerable manual coding effort, which could be largely automatized and made more user friendly by refactoring. At a later stage, refactoring could also help the programmer in choosing the right FastFlow formulation of a pattern, taking into account knowledge about the target platform and models of the cost of execution. It could then also guide and perform transformations between different pattern formulations of a parallelization (e.g. using a Farm vs. Map/Reduce). We expect that such tool support will help to reduce the parallelization overhead when using ParaPhrase, when compared to e.g. OpenMP. It will also provide an advantage to other parallelization approaches concerning the ease of support of multiple target architectures, e.g. by explicit or implicit transformation of pattern hierarchies to ones appropriate for a new target.

Considering the work on mapping performed in WP3, later on in the project we expect the need for appropriate balancing to be an interesting test case for exercising WP3 technology. Given an initial characterization of the block structure of the inverse matrix \( P \) to be determined, we could give estimates for the amount of work to be performed for components, which might help refactoring and mapping to optimize the final placement of components and selection of the number of workers on different levels of the hierarchy.
5. Discrete Optimization – SCCH

5.1 Summary of use case

The second of the two use cases selected at SCCH in Deliverable D6.2 is a use case that has been derived from a project with a company residing in the area of metal sheet processing. The problem that has to be solved deals with optimization of material consumption in the production process. This kind of problem is widespread and often arising in various companies e.g. reel cutting in paper mills [5]. Optimization is achieved using algorithms from the area of constraint programming, precisely solving a discrete optimization problem with constraints in our particular use case [7]. Solving this kind of optimization problem can be computationally very intensive depending on for example the constraints to be met or the degree of optimization to be achieved. Therefore parallelization of the optimization is a worthwhile goal.

The initial algorithm to be parallelized (refer to Simplex Algorithm in Section 5.2) follows an iterative and step-wise approach for solving optimization problems. Therefore the pipeline and farm pattern will be the first patterns to be used for parallelization. Further alternative implementations may also use the map pattern. Furthermore the not yet available pool pattern seems also suitable.

The intended target architectures are mainly multi/many-core shared memory systems. But also distributed architectures are conceivable if the optimization iterations or the single optimization steps, respectively, are computationally intensive enough to offset the overhead of distributing and collecting the objects to optimize. Furthermore additional constraints that prevent isolated optimization might restrict the use of distributed systems.

The expected main advantages gained by a parallel implementation following the ParaPhrase approach are the following:

- The used algorithm is stated to find an optimal solution in a finite number of iterations. Depending on the problem this number of iterations might be very high and lead to long execution times. Breaking earlier will return an admissible but not an optimal solution. The more iterations performed, the better the solution will be. A parallel implementation will lead to a better solution in the same time and therefore to e.g. reduced production costs.
- Optimization problems and the algorithms to solve them are often quite complicated. Using low level parallelization approaches make implementation
even more complicated and therefore harder to understand, maintain and extend and are more error prone. The high level pattern based approach of ParaPhase is expected to solve this.

5.2 Porting

This section describes the parallelization approach that has been chosen, the reasons for that decision and the current state of implementation. Furthermore, important intermediate steps and encountered issues during implementation are outlined.

Before the parallelization approach and its implementation are detailed, the actual chosen optimization algorithm and its sequential implementation that has been ported afterwards are described. The description of the algorithm will be constrained to its basic concept as far as needed to understand the parallelization approach. The mathematical details of the algorithm are left out intentionally.

Simplex Algorithm The used algorithm is called Simplex Algorithm, which is popular in linear programming to solve linear optimization problems. The algorithm either solves such a problem exactly in finite time or states that it is unbounded or not solvable.

The simplex algorithm is a step-wise iterative algorithm that consists of two phases. In the first, an initial admissible solution is computed which is needed as a starting point for the second phase. If such solution can not be found, the problem is not solvable and the algorithm terminates immediately. The second phase is iterative, where each iteration tries to find an admissible solution for the given problem with a better objective function value than already found. An iteration itself consists of a chain of single optimization steps that are chosen heuristically from a set of possible steps. In each iteration one admissible solution is picked from the intermediate solutions that have been computed so far and passed through the chain of optimization steps. In the first iteration the set of intermediate solutions consists only of the one solution found in the first phase. After that the solution is either added to the set of intermediate solutions for a further optimization iteration or eliminated if further optimization is not possible. If the optimal solution has been found, the algorithm terminates.

Optimization Framework At the SCCH the generic optimization framework OptiFramework has been developed to solve various kinds of optimization problems using different algorithms. In its initial version the previously described Simplex Algorithm has been implemented to solve linear discrete optimization problems with constraints. This section describes its basic architecture, how the components map to the optimization algorithm and the sequential implementation of the algorithm. The framework consists of a generic part containing the basic template classes and a problem specific part whose components (e.g. particular optimization steps) derive from the base classes to solve a certain kind optimization problem.
(e.g. reel-cutting in paper mills). The actual implementation of the optimization algorithm is encapsulated in the generic part. Therefore the following coarse-grained description of its components will be constrained to that.

The main components comprise the following:

**Workpiece**: A Workpiece corresponds to an intermediate admissible solution of the Simplex Algorithm that holds all information about the solution. These objects are passed through the sequence of optimization steps for further optimization. Implementations for specific problems derive from this class.

**WorkpiecePool**: The WorkpiecePool holds all the work pieces and synchronizes access to them.

**Worker**: A Worker equates to one specific optimization step of the optimization sequence. Again specific problems derive from this class to implement the necessary optimization steps.

The sequential implementation uses a loop to perform the optimization. In each iteration one Workpiece is picked from the WorkpiecePool and passed through the sequence of Workers for optimization. At the end of the iteration the objective function value is calculated and the new solution is either deleted if further optimization is not possible or put back to the WorkpiecePool (or more copies of it) for a further optimization iteration.

The important parameters that have an an influence on the running time and the solution are:

**ObjectNum**: After the first phase of the algorithm only one admissible solution is available in the pool. Since the optimization steps include stochastic elements more copies of the initial solution are put into the pool. The number of copies is defined by this parameter.

**NetLength**: This parameter defines the number of optimization steps in the sequence.

**Iterations**: Since the number of iterations to find the optimal solution might be very high, an upper limit can be defined with this parameter.

**Parallelization approach** Since the algorithm is stepwise, iterative and the sequence of optimization steps, though heuristically assembled, is static in every iteration, parallelization using the pipeline pattern is obvious. The pipeline represents the optimization sequence where a single stage of the pipeline corresponds to a specific optimization step. The intermediate solutions are picked out of the pool and passed through the pipeline. In addition to the optimization stages, a last pipeline stage is added that performs the calculation of the objective function value and decides how the intermediate solution is further processed (eliminating it or putting a single or multiple instance of it into the pool).

But since generality and extensibility of the framework with additional algorithms and variations of them is a basic concept of the framework, a different parallelization approach has been chosen to accommodate this. Therefore the chosen approach is to use a farm pattern with nested pipelines as farm workers (Figure 5.1).
The components of the farm are the following:

**Emitter:** The emitter of the farm has access to the intermediate solution pool. It picks out the intermediate solutions from the pools and emits them to the worker(s). This replaces the loop in the sequential implementation that picked the solutions from the pool and passed them through the optimization steps.

**Worker:** A worker of the farm contains a nested pipeline that again corresponds to the sequence of optimization steps. This time only the optimization steps are included in the pipeline without an additional collector stage.

**Collector:** After a candidate solution has passed through the pipeline the collector of the farm is responsible for the calculation of the objective function value for this solution and the decision as to how the solution is further processed.

The emitter also picks solutions that have been put back and emits them again. This is repeated until either a solution is found or another of the previously mentioned termination criteria is reached and the best solution found so far is returned.

The use of a farm has two main advantages compared to the single pipeline approach.

- If enough cores are available on the target machine, more farm workers containing the same pipeline can be used which could lead to a higher performance.
- Variations of the algorithm could use optimization sequences with different optimization steps. The implementation of this would be very easy because only an appropriate `loadbalancer` class has to be written that is used by the emitter to decide to which pipeline an intermediate solution has to be passed.

Currently the first implementation of this approach has been finished and first evaluation runs have been performed. They are already very promising (with a speedup of almost 10 on a 12 core machine), but not all possibilities have yet been explored.
Figure 5.2: Illustration of a variation (described in Future plans in Section 5.2) of the initial approach, where the single farm collector is replaced by an additional stage in each pipeline that perform the collectors tasks.

Issues During implementation several issues were encountered. To some extent they were expected, based on prior experiences in parallelizing existing sequential implementations. Others were unexpected and some of them were directly caused by the use of FastFlow.

- Before any parallelization could be done, quite detailed knowledge of the algorithm and its existing sequential implementation had to be gained to identify the parts to be parallelized and to develop an parallelization approach. The more complicated task was identifying the parts that had to be changed to be suitable for a parallel implementation. This included e.g. shared resources and required synchronization without losing advantages of parallelization if possible. In the sequential implementation many objects were instantiated only once, but more times in the parallelized implementation. The lack of proper implementations of assignment and copy constructors led to many problems and required a detailed look into the existing source to enable development of proper implementations.

- The random number generator from a third party library used by many optimization steps has to be declared as a global variable. It therefore had to be decided to either synchronize access (which would lead to less performance
Figure 5.3: Illustration of a variation of the initial parallelization approach. Farm pattern with Emitter and Collector (that have access to the pool) and an arbitrary number of workers with nested Pipelines containing the optimization steps. Depending on the algorithm the pipelines contain the same sequence of optimization steps or different ones. The loadbalancer decides which intermediate solution goes to which pipeline.

- An issue that was caused by the use of FastFlow pertains to SmartPointers. In the initial sequential implementation the intermediate solutions in the pool were managed by SmartPointers to avoid memory leaks. Problems were caused by the interface definition of the svc method of the FastFlow worker class ff::node, because this method expects a pointer to the task as input parameter and also returns a pointer to the task. In this case the tasks were the intermediate solutions from the pool. Therefore it was necessary to omit the use of SmartPointers and use raw pointers.

**Future plans** Currently the parallelization of the algorithm has been implemented in the way described in Paragraph Parallelization approach in Section 5.2. First evaluations have also already been performed (cf. Deliverable D6.5). The next planned development steps concern further development of the current approach,
use of patterns other than farm and pipeline and targeting platforms other than shared memory systems (Items ordered by priority from current point of view):

• Currently the farm uses only one worker containing the optimization pipeline, which is enough for the current target system. For systems with many more available cores using more than one pipeline could lead to better performance, if there is one core available for every thread of the multiple pipeline stages. Using multiple pipelines, the collector might become a bottleneck, because it has to evaluate every intermediate solution from every pipeline.

• Therefore another step will be trying to eliminate the farm collector and add it as a last stage to every pipeline. But at least the intermediate solution pool is shared by all these collectors and access has to be synchronized. Splitting the pool into different storage regions for every collector and only synchronizing the access of the emitter might improve the performance further. Another problem might be that the evaluation of an intermediate solution needs shared resources. In that case the access would have to be synchronized, too (Figure 5.2).

• In the example currently used for evaluation all optimization steps have almost the same execution time. Other examples will contain longer running optimization steps compared to the others. In that case it might be useful to add a farm in the pipeline for these long running steps, containing enough workers to compensate for the higher execution time.

• A further step will be to evaluate the suitability of distributed systems. The algorithm itself seems appropriate for this kind of architecture. The access to shared resources could cause problems with distributed optimization. It will also have to be evaluated whether the optimization steps or the whole pipeline are computationally of sufficient complexity to compensate for the additional overhead for item distribution and collection.

• To support the use of different optimization pipelines for every farm worker, an appropriate loadbalancer class has to be developed. This loadbalancer will then decide, based on some parameters, to which pipeline an intermediate solution from the pool has to be emitted. (Figure 5.3)

• Besides the patterns used in the initial approach, parallelization will also be evaluated using other patterns. E.g. the map pattern seems also be appropriate for these optimization problems and may be less algorithm dependent.

• For comparison of the ParaPhrase approach to other parallelization technologies, an implementation using e.g. OpenMP or OpenCL is also planned.
5.3 Discussion and conclusion

On the whole, the implementation of the specific parts containing the code for the actual parallelization using FastFlow took much less time than understanding and refactoring the existing sequential implementation to be able to parallelize it at all. The most time-consuming problems (as already stated in Issues in Section 5.2) were synchronizing the access to shared resources (in emitter and collector), identifying the objects that in the sequential implementation were only instantiated once, but multiple times in the parallelized version and implementing the required copy constructors and assignment operators and omission of SmartPointers for the intermediate solution objects. The latter was the greatest problem, because SmartPointers were used across the whole framework and many declarations and implementations had to be changed.

It will be interesting to see how far the refactoring tools of ParaPhrase will support the developer in this in future, and how much the effort will be reduced compared to manual refactoring.

The unexpected amount of time for refactoring is the main reason why currently only the initial approach has been implemented and evaluated (see D6.5) and the list of further development steps is still longer than planned.

Automation of problem or algorithm based parallelization will also pose a challenge. An already mentioned example is the possible need for a nested farm within the optimization pipeline to replace a long running optimization stage. Whether there are stages that need much more time for execution, or how much longer that time is, might not be known before the first run. This kind of dynamic problem or execution time dependent pattern architecture might be a harder problem to solve, but would be of much benefit to this use case.
6. Conclusion

Initial versions of all use cases have now been provided by WP6 partners. Initial problems have been encountered, but mostly have already been resolved in cooperation with technical workpackage partners. Now current implementations can be used as a basis for further improvement and implementation of more sophisticated use case applications, and more importantly for evaluation of specific parts of ParaPhrase technology.

Technology already used by the applications are skeletons (Farm pattern of Skel) and refactoring for Erlang, and the FastFlow skeleton library (Farm and Pipeline patterns) for C++. This will be extended in future by OpenCL interfacing from Erlang (to implement heterogeneous patterns), by usage of Map/Reduce, application specific, and heterogeneous patterns in FastFlow, by C++ refactoring, and by testing static and dynamic mapping, as they become available.

So far, porting experiences by WP6 partners have been mixed. In general the pattern based approach of ParaPhrase was easy to map to use cases conceptually. Devising an appropriate pattern hierarchy to express the parallelism for a given application was usually straightforward. Issues encountered during actual porting work were mainly of two kinds, at least regarding C++ use cases. First, the extraction of code into farm or pipeline workers was rather verbose and sometimes required major refactoring work (this is not the case for Erlang porting, as can be seen in Section 2). Second, a straightforward application of the pattern hierarchy often did not result in satisfactory performance; tuning of farm re-usage, synchronization and data granularity was often needed to get good performance (cf. deliverable D6.5).

WP6 partners expect these issues to be addressed by future development in the technical work packages. In particular, refactoring tool support has the potential to alleviate or completely remove both of these issues. If augmented by cost models and when interfacing with static and/or dynamic mapping, refactoring could also hint about good pattern formulations, thus addressing the second issue as well (at least for common cases). As soon as heterogeneous patterns are used, at least static mapping will become necessary to control the distribution of workload between CPU and GPGPU cores, and to avoid the need for multiple experiments by the programmer to determine a good manual mapping. Application specific patterns, on the other hand, have the potential to provide readily tuned solutions for common special cases, which will remove the need for explicit synchronization (or the
danger of inefficiency) for these cases (e.g. stencil pattern for the Computational Molecular Dynamics use case, or a pool pattern for the Discrete Optimization use case).

All use case partners were new to the FastFlow and Erlang pattern based approach. Though conceptually simple and clear, the issues encountered by all partners especially in porting the first non-trivial use case made progress somewhat slower than originally anticipated. Furthermore, the project start was earlier than anticipated, and several partners had problems with full commitment during the first project year. Still, with a delay of approximately one month the use case porting status is now about what was expected originally: availability of parallelized initial versions of all use cases. Not all partners were able yet to provide alternative implementations (e.g. using OpenMP or traditional parallel Erlang) for comparison of programming effort and performance. Otherwise, these use cases are usable as a basis for further evaluation of advanced ParaPhrase technology, and all partners have gained experience in porting, important for evaluation of the advanced technology in future.

Furthermore, in solving the issues encountered there was intensive communication with partners especially at UNIPI, UNITO, and USTAN. WP2 partners in Pisa provided support for some usage questions and performance problems regarding the use of FastFlow. Refactoring work for C++ has started, and there has been intensive discussion about the usage of use cases for development of refactoring, with WP4 partner in St. Andrews. These provided valuable insight as well in porting of the Erlang use case. And we also discussed how to apply cost models to the use cases as a basis for static, and later dynamic, mapping. Technical workpackage partners thus gained additional insight about problems and approaches new users of their technology have, which will help them to optimize code and documentation for further use. This will be important for organization of the first user community workshops in WP7. WP6 partners among themselves intensively discussed the common deployment platform, the RGU cluster, and evaluation criteria, as a basis for the current set of deliverables. A WP6 workshop with participation by several technical workpackage partners in March 2013 in Stuttgart was especially useful in this process. Partners feel that such intensive communication during the previous months provides a good basis for further progress, in technological and application contexts.


