PARA PHRASE

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

First report on Experimental Evaluation
D6.5

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: T6.2

Responsible institution: HLRS
Editor and editor’s address: José Gracia, HLRS, University of Stuttgart

Version 0.2 (Revision: 2001)

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PU Public</td>
<td></td>
</tr>
<tr>
<td>PP Restricted</td>
<td></td>
</tr>
<tr>
<td>KE Restricted</td>
<td></td>
</tr>
<tr>
<td>CO Confidential</td>
<td></td>
</tr>
</tbody>
</table>

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

Dissemination Level:

- **PU** Public
- **PP** Restricted to other programme participants (including the Commission Services)
- **KE** Restricted to a group specified by the consortium (including the Commission Services)
- **CO** Confidential, only for members of the consortium (including the Commission Services)
Executive Summary

This report gives a detailed description of the experimental validation of the Fast-Flow programming model, particularly for C++ and Erlang.

For some of the applications, in particular those originally written in C or C++, the porting work was challenging, in particular when the application logic and its implementation in C++ classes had to be refactored to accommodate the FastFlow library for optimization. In fact, one of the motivations for this task, was to evaluate FastFlow for real world applications, not only academic small scale test kernels. The encountering of issues was therefore anticipated. It is worth noting, that most issues encountered by the application developers have been either fixed, or a feasible work-around provided until a fix is available (see D6.4).

However, all applications have been ported successfully to the FastFlow programming model, some showing already encouraging performance, others just as proof of concept without claiming significant performance increase over the sequential version. The tangible results of this task are FastFlow enabled applications with the encouraging perspective for successful follow-up performance optimization, and the identification of performance problems to be addressed by the FastFlow developers.
# Contents

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

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

2 Experimental validation platforms ......................... 6
   2.1 CRAY XE6 Hermit at HLRS ................................. 7
   2.2 Xookik Cluster at RGU ..................................... 8

3 Haar Wavelet Transform Use-Case ....................... 9
   3.1 Brief Use-Case Description ............................... 9
   3.2 Evaluation ................................................... 9
      3.2.1 Evaluation Metrics ................................... 9
   3.3 Evaluation Results ........................................ 10
      3.3.1 Porting Time ........................................... 10
      3.3.2 Execution Time ......................................... 10
      3.3.3 Speedups ............................................... 10
   3.4 Discussion and Conclusion ............................... 12

4 Molecular Dynamics Use-Case ............................ 14
   4.1 Brief Use-Case Description ............................... 14
   4.2 Evaluation ................................................... 14
      4.2.1 Evaluation Metrics ................................... 14
   4.3 Discussion and Conclusion ............................... 19

5 Waste Water Processing Use-Case ...................... 20
   5.1 Brief Use-Case Description ............................... 20
   5.2 Evaluation ................................................... 20
      5.2.1 Evaluation Metrics ................................... 20
      5.2.2 Evaluation Results ................................... 22
   5.3 Discussion and Conclusion ............................... 27

6 Discrete Optimization Use-Case ......................... 29
   6.1 Brief Use-Case Description ............................... 29
   6.2 Evaluation ................................................... 30
1. Introduction

The ParaPhrase project aims to produce a programming model which increases overall productivity by improving programmability on one hand and resource utilization efficiency on the other hand. The approach taken to achieve this goal is to support the developers in exploiting a variety of parallel programming patterns (programmability), which in turn can be mapped dynamically to the available hardware resources in order to match application needs and hardware capabilities (efficiency).

More specifically, the programmability issue is addressed by the provision of a set of high-level programming patterns (as defined in deliverables D2.1 and D2.3) that cover typical workflows in industrial and scientific applications, and by the provisioning of a set of tools that assist the developer to refactor a given code in order to expose and utilize the programming patterns.

Parallel efficiency of the application is being addressed by the FastFlow library for C++ (and similarly Erlang) which lowers the high-level patterns onto low-level skeletons. These skeletons exist in a variety of implementations, each well optimized for a specific hardware architecture, as for instance multi-cores or GPUs. The FastFlow library further takes care of efficient run-time mapping of computation tasks onto the available compute resources.

This deliverable is the outcome of task T6.4 which has the objective of doing an initial port of real world production codes to the ParaPhrase programming model, in particular for C++ and Erlang. These initial ports are then used to evaluate the current state of the ParaPhrase programming model.

The use-cases presented are an Erlang OTP application (ESL, section 3), Molecular Dynamics (HLRS, section 4), Waste Water Processing (SCCH, section 5), and Discrete Optimization (SCCH, section 6). All use-cases, with the exception of the first, were originally implemented in C or C++; these have been parallelized using the FastFlow library. The first use-case is implemented in the Erlang programming language using the Erlang Skel framework, i.e. the Erlang analogue of the FastFlow library for C++.

As the refactoring tools and some of the high-level patterns are still being developed as anticipated for this stage of the project, this deliverable naturally concentrates on the evaluation of 1) the ease of application of the pattern-based approach to real world C/C++ or Erlang codes, and 2) the performance scaling with increasing number of available compute cores. An evaluation of the ease of use
and productivity gain of the tools will be presented in deliverables further along the project.
2. Experimental validation platforms

The day-to-day development work on the use-cases in being done on various locally available platforms ranging from workstations to small clusters. For the sake of comparability and reproducibility of the experimental validation we have chosen two target systems, namely the cluster Xookik at RGU, as a typical representative of compute resources available to SMEs or academic institutions, and the CRAY XE6 Hermit at HLRS as representative for state-of-the-art high-performance computing facilities. At time of this writing Hermit has fallen from its initial rank 12 in the TOP 500 list (November 2011) down to rank 27 in the current list (November 2012). Still, Hermit is one of the fastest and most sophisticated systems in Europe.

The following table summarizes the resources used in the project:

<table>
<thead>
<tr>
<th>Name / Institution</th>
<th>Type of processor</th>
<th>Nodes</th>
<th>Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hermit (HLRS)</td>
<td>AMD Opteron 6276 (Interlagos)</td>
<td>3552</td>
<td>113.664</td>
</tr>
<tr>
<td>Xookik (RGU)</td>
<td>Intel Xeon X5675 (Westmere-EP)</td>
<td>4</td>
<td>48</td>
</tr>
</tbody>
</table>

The different platforms have different operating system, libraries and different compiler infrastructures that makes comparisons between them impossible. All shared-memory benchmarks will be run at the RGU cluster. The cluster consists of Intel CPUs with hyperthreading. However, hyperthreading is disabled by the system administrators (which is a usual practise on HPC systems). Thus, the influence of additional hyperthreading reasources on the performance of FastFlow could not be assessed. The distributed-memory benchmarks should be executed on Hermit. In addition shared-memory benchmarks may also be executed on Hermit for comparison. The table below details coverage of use-case with respect to platforms

<table>
<thead>
<tr>
<th>Use-case</th>
<th>local resources</th>
<th>Xookik shared</th>
<th>Xookik distributed</th>
<th>Hermit shared</th>
<th>Hermit distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haar Wavelet Transform</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Molecular Dynamics</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wast Water Processing</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Discrete Optimization</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.1 CRAY XE6 Hermit at HLRS

Hermit is a CRAY XE6 supercomputer. The peak performance of the whole system is about 1 PFLOP/s (3552*294.4=1045708.8 GFLOP/s). There are 3552 dual socket G34 compute notes. This means the whole system has 7104 CPUs with 16 cores/CPU. All in all an application can be run on 113,664 cores. The system is build of AMD Opteron(tm) 6276 (Interlagos) processors.

CPU Facts:

- An Interlagos CPU is composed of 2 Orchi Dies. An Orchi Die consists of 4 Bulldozer modules
- 2*4*2= 16 Cores/CPU @ 2.3 GHz (up to 3.2 GHz with TurboCore)
- 32MB L2+L3 Cache, 16MB L3 Cache
- 2*2*2 channels of DDR3 PC3-12800 bandwidth to 8 DIMMs (4GB each)
- Direct Connect Architecture 2.0 with HyperTransport HT3: 6.4 GT/s*16 Byte/Transfer= 102.4 GB/s
- supports ISA extensions SSE4.1, SSE4.2, SSSE3, AVX, AES, PCLMULQDQ, FMA4 and XOP
- Flex FP: Bulldozer modules (2 cores) share a single 2*128= 256bit floating point unit
- Peak performance per socket: 2.3*4*16= 147.2 GFLOP/s

The total amount of compute note RAM is 126 TB (3072*32+480*64=129024 GB). As High Speed Network between the nodes CRAY Gemini is used.

Architecture: The system is devided into service nodes, compute nodes and pre-post precessing nodes. Service nodes are login nodes and MOM nodes. MOM nodes are used for placing user jobs of the batch system into execution. On the login node the request for compute node ressources will be done. Via a cross-compiler the code can be compiled on one of these two nodes. Compute nodes are only available for users using the batch system and the Application Level Placement Scheduler (ALPS)

Software Features: The system runs with the Cray Linux Environment (CLE) 4 operating system. This is operating system is based on SUSE Linux Enterprise Server (SLES) 11. Torque and moab are used for the batch system. On the system there are many development tools in different versions available. The default compiler is the cray compiler in version 8.1.6. Therefore the compiler
2.2 Xookik Cluster at RGU

The RGU cluster is a distributed memory configuration with four identical compute nodes and an interactive headnode for code compilation and job submission. The interactive headnode incorporates two Intel Xeon E5620 2.4GHz CPUs with four individual cores per processor.

On each of the four compute nodes, two Intel Xeon X5675 Westmere-EP CPUs, with six 3.07GHz cores per processor, 50 GB of physical memory and an NVIDIA Tesla M2090 GPU are present. This brings the cluster-wide compute-core count to 48 and the total physical memory to 200GB. The Westmere-EP CPUs offer hyperthreading to multiplex two logical threads on a single physical core. This feature seems to be favorable for the FastFlow programming model.

Infiniband provides high-bandwidth, low-latency interconnection between the cluster nodes which also have access to a shared filesystem and local temporary scratch space. The Base Operating system is Scientific Linux 6.2 with the Alces HPC Software Stack preconfigured with tools for deployment, performance and availability monitoring. Cluster job scheduling is managed by Sun Grid Engine. Installations of NVIDIA's CUDA toolkit, OpenMPI and compilers from Intel and PGI are available.
3. **Haar Wavelet Transform Use-Case**

3.1 **Brief Use-Case Description**

This use-case is described in more detail in Deliverable D6.4. To briefly summarise, the use-case uses a Haar wavelet transformation, applied to Audio Files (for a 1D Haar Transform) and Images (for a 2D Haar Transform). The primary purpose of the Haar Transform is to apply a compression to the image or audio sample, thereby allowing it to be efficiently sent across a network for processing.

We evaluate our 1D Haar Transform over 2048 audio files, split into samples (vectors) of size 4400 elements. In D6.4, we ported the 1D Haar to two Skeleton versions using the Erlang Skel framework, from D2.2. In the first version, we introduce a simple task farm, with 24 workers, to execute the 1D Haar Transform operation in parallel to the input list of audio samples. In the second version, we add partitioning, thereby grouping the audio samples into a larger task (or 4 audio samples). This has the effect of increasing the granularity of the computation and reducing communication and synchronisation overheads. For the 2D Haar Transform, we ported our application to use a task farm, for 24 images, of 1024 * 1024 dimensions.

3.2 **Evaluation**

3.2.1 **Evaluation Metrics**

- **Porting time**: the total time spent for porting the application’s serial version to the parallel programming model. This excludes time taken for benchmarking the application.

- **Execution time**: the total time spent in execution; and,

- **Absolute Speedups** of the parallel implementations against the serial versions.
3.3 Evaluation Results

3.3.1 Porting Time

1D Haar Transform  Porting of the sequential code to use the Skel library was reasonably fast, taking half a day of effort, with most of the effort in building a task farm which calls the sequential 1D Haar Transform as a worker kernel. The chunked version required another half day of effort to implement the required chunking (partitioning) functions, as well as the corresponding decomposition and recomposition functions in Erlang.

2D Haar Transform  The porting of the 2D Haar Transform took 1 hour, as most of the process was similar to introducing the task farm as for the 1D Haar Transform, and therefore the experienced gained the first time around proved invaluable.

3.3.2 Execution Time

All measurements have been made on the Xookik cluster at RGU.

1D Haar Transform  Figure 3.1 shows the runtime results for the 1D Haar Transform, for 2048 audio files, split into audio samples of 4400 elements. The blue line shows the simple task farm variant, and the red line shows the task farm with additional chunking. The figure clearly illustrates the the chunking variant is slightly faster than the standard task farm version, when executed on 8 cores and more. Up to 8 cores, the runtimes are similar, this is most likely due to the task allocation on fewer cores, where there are fewer workers, so more tasks are allocated to each worker. On 12 cores, the average runtime for the non-chunking version was 0.75 seconds, and for the chunking version 0.71 seconds; a difference of 0.04 seconds.

2D Haar Transform  Figure 3.3 shows the runtime results for the 2D Haar Transform, for 24 images, each with a dimension of $1024 \times 1024$. On 12 cores, the average runtime was 3.3 seconds vs. 31 seconds for the sequential version.

3.3.3 Speedups

1D Haar Transform  For the 1D Haar Transform, we executed the application over 2048 audio samples, each with a sample size of 4400. This translates to 20480 vectors, each with 4400 elements, where a single execution of a 1D Haar Transform operates over a single vector. Figure 3.2 shows the speedup results for the 1D Haar Transform up to 24 cores. In the figure, the blue line corresponds to the speedups from the simple task farm variant of the application, with 12 static workers, from 1 – 12 cores. The red line shows the chunked version, where the input list of vectors are partitioned into groups of 4 vectors, each with a sample size of 4400. The chunked version still uses a task farm, with 12 static workers, where each
worker function is mapped over the partition of tasks. The simple task farm variant (blue line) shows a maximum speedup of 8.99, where the version with chunking shows an improvable speedup of 9.55. Although more investigation is needed at this stage, initial speculation suggests that in the simple task farm variant, there is an abundance of parallelism, where the system is saturated with many fine-grained tasks. In the chunking version, the number of tasks is reduced, but the computation size is increased, therefore reducing the communication overheads.

2D Haar Transform Figure 3.4 shows speedup results for the 2D Haar Transform, over 24 images, each 1024 * 1024 in size. As it can be seen, the application starts to scale reasonably well, tailing off at around 12 cores with a speedup of 7.2. After 16 cores, the application scales much better, with a maximum speedup of 9.2 on 12 cores. This is most likely due to the fact that there are only 12 tasks: on 12 cores, each task is likely to be allocated to a worker operating in a separate thread, and this is not the case with fewer than 24 cores, where the task allocation becomes irregular. One solution to this could be to make the Skel emitter allocate tasks to idle workers, rather than use a round robin distribution. Further experiments are needed here.

Figure 3.1: Runtime figures for a 1D Haar Transform, for 2048 audio files, with a sample size of 4400.
3.4 Discussion and Conclusion

We have presented evaluation results for a 1D Haar Wavelet Transform and 2D Haar Wavelet Transform. For the 1D Haar, we presented two variants, both using a task farm skeleton from the Erlang Skel library. The first variant was a basic task farm skeleton with 24 workers, and the second variant was a task farm skeleton with partitioning. Our evaluation showed that the basic task farm variant achieves an average speedup of 8.9, where the partitioning version gives an improved speedup of 9.5, for 12 cores. For the 2D Haar Transform, we evaluated the application using a simple task farm skeleton from the Erlang Skel library, demonstrating a speedup of 9.2 of 12. Clearly this shows reasonable and scalable speedups of the Haar Transform use-case. For future work, we intend on performing more experiments on a range of platforms, and also evaluating an OpenCL variant, which would also allow us to take into account GPU architectures.

Figure 3.2: Speedup figures for a 1D Haar Transform, for 2048 audio files, with a sample size of 4400.
Figure 3.3: Runtime figures for a 2D Haar Transform, for 24 images, 1024*1024.

Figure 3.4: Speedup figures for a 2D Haar Transform, for 24 images, 1024*1024.
4. Molecular Dynamics Use-Case

4.1 Brief Use-Case Description

This use-case is described in more detail in deliverable D6.4.

Our CMD (Computational Molecular Dynamics) code implements two algorithms for the computation of interactions between the molecules. These two algorithms are ported to FastFlow parallel programming model and evaluated for their performance on FastFlow.

The first use-case is 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. It has complexity $O(N^2)$.

The second use-case is CMD MoleculeBlocks, where the large domain is decomposed into cells and then molecules are distributed among the cells. In this case, interactions of molecules of a cell are computed only with half of the neighboring cells (13 instead of 26 for a 3D space) and it also utilized Newton’s 3rd law of motion which raises the requirement of synchronization mechanism. This is not a highly data parallel application like BasicN2 use-case and also has many dependencies among threads.

The CMD BasicN2 use-case is evaluated for the problem size of 68,000 molecules, whereas the CMD MoleculeBlocks use-case is evaluated for the problem size of 1,000,000 molecules. Both codes are compiled with GCC 4.7.2. Also for FastFlow versions of use-cases, threads are pinned with following mappings: main thread to core 0, emitter thread to core 1, worker threads to cores 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, and 0, respectively.

4.2 Evaluation

4.2.1 Evaluation Metrics

Porting Time: This is the total time spent for porting the application’s serial version to a parallel programming model. This excludes time taken for benchmarking the application.

Simulation Kernel Execution Time: This is the total time spent on simulation loop in CMD code which incorporates 7-8 routines. The simulation kernel is executed 10 times for each fix number of worker threads. The benchmark results
Figure 4.1: Execution time of use-case CMD BasicN2 with FastFlow and OpenMP shows minimum, maximum and average value of execution time for fix number of worker threads.

**Speedup:** This is speedup of simulation kernel after porting it to parallel programming model. It is the ratio of simulation kernel’s average serial execution time to average parallel execution time.

### 4.2.2 Evaluation Results

#### 4.2.2.1 Porting Time

**BasicN2 Use-Case:** The porting of BasicN2 use-case to FastFlow programming model took 2 weeks, which includes the initial learning and exercising the concepts of FastFlow parallel programming model. Whereas doing the same on OpenMP took 2 days.

**MoleculeBlocks Use-Case:** The porting of use-case MoleculeBlocks solely took 2-3 weeks including optimization of this use-case to achieve better performance using FastFlow. Initially only force calculation routine was parallelized with FastFlow which took 2-3 days. But to achieve a decent speedup up in this use-case, more than a single simulation kernel routine is needed to be parallelized. This is because there is very less computation per thread in this use-case compared to the BasicN2 use-case. The rest of the time was spent in porting the other routines of simulation kernel to FastFlow and optimizing the code to achieve better perfor-
Figure 4.2: Execution time of use-case CMD MoleculeBlocks with FastFlow and OpenMP

Figure 4.3: Speedup of use-case CMD BasicN2 with FastFlow and OpenMP
Figure 4.4: Speedup of use-case CMD MoleculeBlocks with FastFlow and OpenMP.

Figure 4.5: Execution time of the use-case BasicN2 with FastFlow and OpenMP on Hermit.
### Table 4.1: Averaged measured execution times and standard deviation for the two use-cases BasicN2 and MoleculeBlocks implemented with FastFlow (FF) and OpenMP, respectively.

<table>
<thead>
<tr>
<th>cores</th>
<th>BasicN2 FF time [s]</th>
<th>BasicN2 OpenMP time [s]</th>
<th>MoleculeBlocks FF time [s]</th>
<th>MoleculeBlocks OpenMP time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22.819 ± 0.011</td>
<td>22.126 ± 0.008</td>
<td>2.335 ± 0.002</td>
<td>2.364 ± 0.003</td>
</tr>
<tr>
<td>2</td>
<td>11.435 ± 0.026</td>
<td>11.132 ± 0.025</td>
<td>1.442 ± 0.004</td>
<td>1.659 ± 0.031</td>
</tr>
<tr>
<td>3</td>
<td>7.978 ± 0.012</td>
<td>7.443 ± 0.013</td>
<td>1.090 ± 0.002</td>
<td>1.252 ± 0.113</td>
</tr>
<tr>
<td>4</td>
<td>5.985 ± 0.021</td>
<td>5.631 ± 0.054</td>
<td>0.926 ± 0.001</td>
<td>1.088 ± 0.027</td>
</tr>
<tr>
<td>5</td>
<td>4.802 ± 0.014</td>
<td>4.621 ± 0.003</td>
<td>0.819 ± 0.001</td>
<td>0.903 ± 0.008</td>
</tr>
<tr>
<td>6</td>
<td>3.993 ± 0.006</td>
<td>3.885 ± 0.005</td>
<td>0.923 ± 0.005</td>
<td>0.849 ± 0.001</td>
</tr>
<tr>
<td>7</td>
<td>3.417 ± 0.006</td>
<td>3.337 ± 0.007</td>
<td>0.792 ± 0.001</td>
<td>0.753 ± 0.001</td>
</tr>
<tr>
<td>8</td>
<td>2.997 ± 0.008</td>
<td>2.921 ± 0.006</td>
<td>0.759 ± 0.002</td>
<td>0.745 ± 0.002</td>
</tr>
<tr>
<td>9</td>
<td>2.667 ± 0.010</td>
<td>2.598 ± 0.007</td>
<td>0.733 ± 0.001</td>
<td>0.697 ± 0.017</td>
</tr>
<tr>
<td>10</td>
<td>2.399 ± 0.008</td>
<td>2.340 ± 0.003</td>
<td>0.681 ± 0.003</td>
<td>0.691 ± 0.013</td>
</tr>
<tr>
<td>11</td>
<td>2.234 ± 0.001</td>
<td>2.190 ± 0.001</td>
<td>0.671 ± 0.001</td>
<td>0.648 ± 0.001</td>
</tr>
<tr>
<td>12</td>
<td>2.064 ± 0.001</td>
<td>2.022 ± 0.001</td>
<td>0.720 ± 0.004</td>
<td>0.673 ± 0.001</td>
</tr>
</tbody>
</table>

### 4.2.2.2 Simulation Kernel Execution Time

**BasicN2 Use-Case** The Benchmark for BasicN2 simulation kernel execution time is shown in figure 4.1. It compares the execution time of simulation kernel on FastFlow and OpenMP. We can see that the OpenMP version is very slightly faster as compared to the FastFlow version for low number of worker threads (as shown in standard deviation in table 4.1), but as the number of worker threads is increased we achieve almost same execution time with FastFlow as we do with OpenMP.

**MoleculeBlocks Use-Case** The Benchmark for the MoleculeBlocks simulation kernel execution time is shown in figure 4.2 for the Xookik cluster and in figure 4.5. It compares the execution time of simulation kernel on FastFlow and OpenMP. We can see that the FastFlow version is significantly faster than the OpenMP version for up to 5 worker threads (Single NUMA Node on RGU Cluster has 6 cores). FastFlow implementation uses the farm pattern in accelerator mode. It has two additional threads apart from worker threads compared to OpenMP, which are the main and emitter threads. FastFlow utilizes one core for the main thread, whereas first worker thread is mapped to the same core as the emitter thread. So for 5 worker threads FastFlow implementation utilizes one complete NUMA node of RGU cluster which has 6 cores. As the number of worker threads are increased beyond 5, OpenMP version performs slightly better than the FastFlow version.
4.2.2.3 Speedup

**BasicN2 Use-Case** The Benchmark for the speedup of BasicN2 simulation kernel is shown in figure 4.3. We observe almost same speedup with the FastFlow version as we do with the OpenMP version.

**MoleculeBlocks Use-Case** The Benchmark for the speedup of MoleculeBlocks simulation kernel is shown in figure 4.4. We observe that FastFlow version is significantly faster on a single NUMA node (as shown in standard deviation in table 4.1). However beyond single NUMA node, the OpenMP version is slightly faster than FastFlow version.

4.3 Discussion and Conclusion

With both BasicN2 and MoleculeBlocks use-cases we have observed good speedup of our simulation kernels on FastFlow when compared with their OpenMP versions. However there is still a need to efficiently map the threads to the cores with FastFlow when executing the application on more than one NUMA node as memory access latency depends on the locality of the data in memory of NUMA nodes. Therefore an automatic efficient thread mapping is required by the FastFlow implementation.
5. Waste Water Processing Use-Case

5.1 Brief Use-Case Description

The Lenzing waste water processing use-case is implemented in C++ using FastFlow as well as the Machine-Learning library mlpp. Because of the problem structure, we initially target homogeneous multicore machines. In a later stage, we consider implementing low-level parts using GPGPU-enabled FastFlow patterns, as well. Of the available levels where parallelization can be performed, two levels are implemented and evaluated already:

**Connected components** By exploiting the fact, that for a given regularization parameter $\lambda$ the estimated inverse covariance matrix $W$ assumes a block structure and optimization for each block becomes a completely independent problem, these blocks (connected components) can be optimized in parallel. An issue to consider here is, that the block structure of a problem cannot be predicted (and for small amounts of regularization will not be present), and individual blocks can be of very different sizes.

**Coordinate descent** At the base of the optimization of the graphical lasso lies the coordinate descent algorithm for actually iteratively estimating the elements of the inverse covariance matrix. Coordinate descent can be parallelized across features (matrix columns or rows), up to a data dependent degree of parallelism [1].

Details are provided in Deliverable D6.4. Evaluation has taken place on artificial data, randomly generated to emulate distributions and correlation structures appropriate for the algorithm and problem setting, and on actual data from SCCH customer Lenzing.

5.2 Evaluation

5.2.1 Evaluation Metrics

We performed development and initial evaluation at SCCH on *Tirana*, a Linux systems with a 12 cores (dual-hexacore, Intel Xeon X5690, 3.47 GHz) with HyperThreading support, and 24GB of memory, running under Linux 64bit (Ubuntu
Thus, characteristics are similar to individual nodes of the evaluation cluster *Xookik* at RGU, allowing us to perform development locally under conditions more or less similar to the evaluation platform.\(^1\) We encountered a few issues regarding compiler and library versions, but as expected, performance and scaling are relatively similar between these systems. Where not explicitly mentioned, this section reports performance on ParaPhrase *Xookik* cluster (only using one of the four 12-core nodes, because the implementation is not using distributed patterns).

The execution times we report in the following are based on the computer microsecond clock (as provided by the `boost::posix_time` library). We time complete functions, including sequential initial/final parts, where present. What we do not take into account is the time for the computation of the data covariance matrix, which is already provided as an argument to the relevant functions (and is a candidate for parallelization using GPGPU support later on). Sequential parts included are mainly the computation of the connected components (blocks) and initialization of helper variables, setup of *FastFlow* patterns, and copy operations for variables as necessary for the parallelization. Because we sometimes observed some speedup of the parallel version using one worker, with respect to the sequential version, we assume that there is some caching going on during the first sequential run. We circumvented this effect (being a disadvantage for the sequential version, otherwise) by calling the sequential version twice, and only reporting on the second timing.

Speedup is computed from the raw execution times as detailed in the previous paragraph, by dividing the sequential algorithm execution time by that of parallel version. The problem size is kept the same while the number of workers is varied.

Because of the pattern structures implemented, reporting execution times or speedup for a given number of worker does usually not imply the same number of threads. Instead, in *FastFlow* farm workers are provided their input data by emitters, and collectors take care of the results produced by the workers, and both of these components run in their own threads. These are usually not critical, as their work load is small compared to that of the workers. Especially the Emitter does not perform any complicated (floating point) computations, and should not affect the workers much, or consume additional cores not reported by the number of workers. Actually HyperThreading seems to well take care of the computations needed by these extra threads, without the need for additional cores. The collector thread can sometimes be avoided completely, when the workers can write their result directly to shared memory (to locations specific to each worker, to avoid race conditions). This technique is currently used for both evaluated parallelized algorithm parts.

As detailed in Deliverable D6.4, so far we implemented four parallelization variants for the graphical lasso algorithm, “cd_cov”, “cd_mat”, “glasso_comp” and “glasso_mid”. Two of these, “cd_mat” and “glasso_comp”, appear to be useful with respect to convergence behaviour and work load in the workers, and we re-

---
\(^1\)Two GPU boards are available as well, but not used currently in this use-case.
\(^2\)Relatively late for this deliverable we found out, that Hyperthreading actually is disabled on *Xookik*, which probably explains most of the remaining performance differences.
Figure 5.1: Scaling of parallel components algorithm “glasso_comp” on Tirana (left) and Xookik (right) for randomly generated correlated multivariate Gaussian data with 1000 features. Blue shows execution time results on left axis (mean over 7 runs; errorbars show corrected sample standard deviation across runs). Red shows speedup on right axis. X-axis shows performance of sequential code version “seq” on the left, and the parallel version using different numbers of workers along the axis. For interpretation see text in Section 5.2.2

port performance for them in the following. Implementation effort for all of these is documented as well, for comparison to alternative parallelization approaches. Such development effort is based on the time recording facility used at SCCH. The numbers are adapted by estimates to reflect the fact that not each short action is recorded separately there, but tasks like looking up documentation occur frequently during development. We report only hours directly connected to the parallelization effort and learning about FastFlow, not connected to sequential development, to repository maintenance, documentation writing, etc.

Additionally, source lines of code (SLOC) are reported. These are lines of code excluding empty lines and comment lines, as determined by the tool cloc\(^3\) on source files consisting just of the relevant code. For each variant, these are the lines for those functions touched (or introduced) by the parallelization. That is, the main function being parallelized, and all helper functions and helper data structures which were modified or introduced.

We have already started with an alternative implementation of the algorithms using OpenMP; the version for “cd_mat” is ready and we report results for it. As more comparison implementations become available, we plan to compare the ease of coding and code maintenance using OpenMP and ParaPhrase technology.

### 5.2.2 Evaluation Results

**Connected components** Artificial data for evaluation is randomly sampled from a multivariate Gaussian with its covariance matrix having a blocked structure (i.e.\(^3\)http://cloc.sourceforge.net/
with several sets of features having higher correlations among themselves than to features in other sets). The blocks have high but random correlation among the features in the blocks, and very little correlation with features of other blocks. Before sampling, the block-randomly initialized covariance matrix is preprocessed to yield a feasible (positive definite) covariance matrix. The blocks contain 60 features mostly (except for the last one), to yield several components of a size inducing reasonable workload. For the regularization parameter used, $\lambda = 0.08$, the estimated component sizes are $120, 120, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60$, and several components of size 1.

As computation for each component in “glasso_comp” is completely independent, we expected very good scaling behaviour, subject to presence of components of similar size in the data. As Figure 5.1 shows, this was not achievable completely. Speedup can be considerable, but it was not linear, neither for the artificial data
shown in that Figure, nor later on for our real data. Beside some problems during initial parallelization (discussed later on), the problem turned out to be mainly in the data generation. The correction of the covariance matrix to be positive definite turned out to be a high computational burden (solving several eigenvector problems), which takes longer than the actual inverse covariance matrix computation. Thus we can only generate artificial data with a limited number of components and component sizes, which limits the potential for scaling. For our real data, we have one dominating component with a few medium sized following (see Figure 5.2). As a consequence, to generally achieve significant speedup, we will need to improve our parallelization approach by a combination of parallelizations on different levels, with an appropriate load balancing of workers.

A more detailed analysis of Figure 5.1 still shows some interesting aspects. First, we see that the parallel version using just one worker performs approximately as fast as the sequential version (on Xookik even slightly faster, despite using the second sequential run for comparison). This indicates that the overhead of using FastFlow is no problem, at least at this problem size. Using two workers, the speedup is about 1.8, showing that good scaling can be possible. Perfect scaling probably is prevented by the estimation of the block structure of the matrix in the initial sequential part.

For higher numbers of workers, the speedup stagnates. Besides the sequential block structure estimation, another reason can be the sizes of the components. There are two large components of 120 features each, which are well parallelizable on two workers. The other components are much smaller, with some 60-feature components coming next. As the graphical lasso algorithm implemented for each component has at least cubic complexity in feature size (maybe more, depending on the number of iterations necessary for convergence), there is not that much work left for a larger number of additional cores. Not clear yet is the difference in scaling for 8 workers on Tirana and Xookik, despite the otherwise relatively similar timing results.

For the real waste water processing data set by Lenzing (Figure 5.2) we obtained similar speedup on Tirana and Xookik, thus we just report about the latter one. At the selected regularization parameter value, there is a dominating block with 342 features, with just a few medium sized blocks (101, 52, 42 features) remaining. This limits the speedup to about 1.05 for 3 workers. A solution for such data will be the planned incorporation of parallelization on lower levels (coordinate descent, which is not yet integrated into the Lenzing analysis program; also parallelization of the covariance matrix computation), and on higher levels (computation of the regularization path across several regularization parameters).

Regarding the mentioned initial parallelization problem, we obtained very bad execution times for worker threads, when compared to a sequential code in the main thread. Despite doing exactly the same computations in the same code and function, execution times were about 50% longer. This occurred even if just one worker was used, and also if pinning was applied to threads to make sure they ran on the same processor (i.e. in the same NUMA-node of the dual-hexacore system).
The reason is not completely clear yet, but we were able to alleviate that problem by letting workers not compute in separate parts of a global variable, but by copying those parts into smaller worker-local variables. Interestingly, we did not see this behaviour for the artificial data, maybe because of the somewhat smaller problem size.

**Coordinate descent** Evaluation of the second version of parallelizing the graphical lasso code results in the execution times and scaling shown in Figure 5.3, for *FastFlow* parallelization as well as for *OpenMP* parallelization. The algorithm parallelizes the updates of the weights in the coordinate descent algorithm [1]. Because in the sequential version updates of following features use previously computed results, a parallel update does not converge in the same number of iterations as does the sequential version. How many more iterations are needed depends on the degree of parallelism, but also on the problem characteristic. Namely, the correlation between features can limit sensible parallelism to apply.

The data used for evaluation in the figure was a randomly (normal/Gaussian) generated data matrix of 100,000 rows by 1,000 columns (features). As seen in the plot, for this data, for up to about 4 workers the parallelizations works well (although not linearly, because of some overhead and increases in iterations), with a maximum speedup of 2.0 to 2.4 at about 8 to 10 workers. Then, the speedup stagnates (*OpenMP* on *Xookik*) or degrades (*FastFlow* on *Xookik* and *Tirana*; *OpenMP*...
Table 5.1: Report on the effort in hours spent on parallelization of different code variants (FF: *FastFlow*, OMP: *OpenMP*). Only effort directly related to parallelization is detailed.

<table>
<thead>
<tr>
<th>Task</th>
<th>FF/generic</th>
<th>cd_cov</th>
<th>cd_mat</th>
<th>glasso_comp</th>
<th>glasso_mid</th>
<th>cd_mat</th>
<th>OMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF/OMP reading</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>basic variant</td>
<td>20</td>
<td>5</td>
<td>15</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Debugging, tuning</td>
<td>25</td>
<td>25</td>
<td>5</td>
<td>20</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing, timing</td>
<td>10</td>
<td>5</td>
<td>20</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Interestingly, *OpenMP* and *FastFlow* have very similar performance characteristics. The main differences are a higher overhead for the *OpenMP* version (the parallel implementation with one thread is significantly slower than the sequential implementation), but a better behaviour when the number of workers/threads is above the number of available cores on the machine (12). Otherwise, we often (as in the presented plots in Figure 5.3), but not always, observed slightly smaller computation times for the *FastFlow* variant.

As previously discussed, because speedup using parallelized coordinate descent is in principle limited, we are looking at parallelization potential on other levels as well. In principle, the parallelization of connected components and coordinate descent would complement each other well, because the former will work well for several medium sized components, while the latter can accelerate computation for the large/dominating component(s). The smaller components are not an issue, because their computation does not take significant amounts of time. Whether such a combination in practice can deliver the intended speedup still has to be evaluated, as both parallelization variants have not yet been combined in one program.

**Programming effort** Deliverable D6.4 gives an introduction to the four developed parallelization variants. In Table 5.1 we report on the approximate working hours spent on different tasks during parallelization of these. One of the variants so far was also implemented using *OpenMP*. Regarding initial reading of documentation, both parallelization approaches are relatively similar, in our experience. Because of the minimal invasiveness of *OpenMP* parallelization, the algorithm was quickly parallelized, although tuning still involved testing out a few formulation variants. An initial implementation of a *FastFlow* pattern is more invasive, although conceptually still very straightforward. Because of the verbosseness, testing out different implementation variants required more effort for the *FastFlow* variants. We hope, that this will be alleviated in future by the use of the envisaged refactoring tool.

Additionally, Table 5.2 reports on the source lines of code (SLOC) of the various variants. The numbers so far seem to indicate that for parallelization of small
Table 5.2: Source lines of code (SLOC) for sequential and parallel versions of parallelization variants.

<table>
<thead>
<tr>
<th>Parallelization</th>
<th>cd_cov</th>
<th>cd_mat</th>
<th>glasso_comp</th>
<th>glasso_mid</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential</td>
<td>49</td>
<td>53</td>
<td>67</td>
<td>120</td>
</tr>
<tr>
<td>FastFlow</td>
<td>102</td>
<td>127</td>
<td>126</td>
<td>394</td>
</tr>
<tr>
<td>OpenMP</td>
<td></td>
<td></td>
<td></td>
<td>71</td>
</tr>
</tbody>
</table>

to medium complex problems (the first three variants), 2-2.5 times the amount of code is needed for the parallelized parts using FastFlow. For the relatively complex “glasso_mid”, about 3.3 times the amount of code is needed. The OpenMP version of “cd_mat” just introduces a few pragmas, thread-local variables, an include, and some extra lines for optimized synchronization, and thus has a smaller impact on code size than the FastFlow version.

### 5.3 Discussion and Conclusion

So far, two usable acceleration approaches by parallelization of graphical lasso parts have been developed. Both have their individual advantages and disadvantages, and it remains to combine them to a comprehensive approach which could work well on dominant large matrix components, as well as on a large number of medium sized components. Further potential for parallelization, also for distributed systems, arises in the computation of whole regularization paths (evaluations for different settings of the regularization parameter, to choose an appropriate one). These could be treated independently, but a more sophisticated parallelization taking into account relations between neighbouring regularization parameter values might provide further advantages.

Regarding ParaPhrase technology, we have gained initial experience by several parallelization implementations. For a realistic comparison to alternative parallelization approaches (most notably OpenMP), we are waiting for practically usable ParaPhrase refactoring tool support, and more experience with OpenMP parallelization of further algorithm parts.

Understanding the concept of ParaPhrase technology, FastFlow mainly in the moment, turned out not to be a problem. It is relatively easy to develop parallelization approaches using patterns like Farms and Pipelines. On the other hand, it turned out to be quite involved to find an optimal formulation for these patterns. A huge part of the reported effort was spent on implementing different variations (use of FastFlow as an accelerator or calling the skeletons directly, usage of collectors or writing back results into specific parts of global memory, or letting the main thread perform reduce steps, ...). In the end, this turned out not to be as important as imagined in the beginning, because our initial performance problems were mainly due to a performance bug in the matrix/vector library used. So other
users might have different experiences here. Anyway, we are thankful for helpful discussion of issues with partners mainly from UNIPI and WP6. An advantage of the use of FastFlow has already been the possibility for it to run on several different target machines, operating systems, compilers. Porting to another architecture has not been a problem so far.

For the future we furthermore anticipate, that such optimization issues will be hidden, or testing different variants made easier, by the possibility to use refactoring tools. Especially when refactoring can be supported by cost models for the resulting skeletons, we expect this to become much easier and informative for the programmer. Also, development using the planned application specific patterns might be helpful in this respect. We expect these to be readily tuned to the specific application area, and they should provide a familiar interface to the programmer. Both of these have further potential to ease the development and maintenance of parallel applications.
6. Discrete Optimization Use-Case

6.1 Brief Use-Case Description

This section briefly describes the use-case just as detailed as needed to understand the following evaluation. For details about the use-case, the used algorithm and its implementation it is referred to deliverables D6.2 and D6.4. The use-case itself has been derived from a project with a company residing in the area of metal sheet processing. The core 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. The optimization is achieved using algorithms from the area of constraint programming, more precisely for our use-case solving a discrete optimization problem with constraints.

As described in deliverable D6.4, a framework for discrete optimization called OptiFramework has been developed to solve different optimization problems using various algorithms.

The algorithm currently implemented for this use-case is iterative and stepwise and possesses some similarities with the well-known Simplex Algorithm. The basic idea of the algorithm is the following (a detailed description is provided in D6.4). It consists of two phases. An initial one, that is only executed once and second phase that performs optimization iterative and step-wise. In the first of the two phases it is tried to find an initial admissible solution that fulfills one required constraint, which is needed to start phase two. If such a solution could not be found the algorithm terminates immediately.

Based on this initial solution found in the first phase, the second phase iteratively enforces an increasing number of constraints interleaved with optimization steps of the partial solutions. To pursue different optimization paths the initial solution is copied several times and pushed into the pool of intermediate solutions. In each iteration an intermediate solution is picked from the pool and passed through the optimization chain. If the new objective function value after passing the chain is better than a certain threshold, the intermediate solution is considered worth for further optimization and again several copies are pushed into the pool. Otherwise the intermediate solution is dismissed.

In the initial parallel implementation the chain of optimization steps is parallelized using a farm of pipelines, where each stage of a pipeline corresponds to a single optimization step. The emitter of the farm takes one solution from the set
of admissible solutions computed so far and emits it to an optimization pipeline. After the solution has been passed through the pipeline, the collector calculates the new objective function value and decides whether to put the solution back to the set, maybe even more than one copy of it, for further optimization or to eliminate it. For details about the parallelization approach it is again referred to deliverable D6.4.

Implementation and initial evaluation of this use-case has been done on the local system Tirana which is described in section 5.1.

6.2 Evaluation

6.2.1 Evaluation Metrics

The evaluation of the parallelization of the use-case is split into two parts. The main part deals with evaluation of the performance of the parallelized algorithm. Therefore the source code has been instrumented with additional statements to measure the overall execution time of one run of the program. The measurement is started after the configuration files has been read and stopped after a solution has been found (The test case and the parameters were chosen in a way that the problem is solvable). The measurements are not constrained to the parallelized or respectively sequential parts containing the code were the actual iterative optimization takes place, but also include the initialization parts, that are different between the sequential and the parallel implementation. For measuring the execution time, the timer class provided by FastFlow has been used. The measures are in microseconds and converted into seconds with two decimals afterwards. The denoted values are averaged values over three runs (whereaa outliers have been excluded).

The implementation of the simplex algorithm provides several parameters (already been explained in more detail in deliverable D6.4) that have a direct influence on how the optimization is performed. Therefore only the parameters, which have an affect on the computational load and the number of threads, are outlined here, since those parameters have been varied in the different evaluation runs.

NetLength: This parameter defines the length of the optimization chain, which equals to the number of optimization steps performed in an iteration of the simplex algorithm. Since the number of steps corresponds to the number of stages in the pipeline, it further influences the number of instantiated threads in the parallelized implementation.

ObjectNum: As already mentioned above the admissible solutions used in the second phase of the simplex algorithm are maintained in a pool. Since phase one returns only a single solution, this solution is copied several times at the beginning of phase two, so that different optimization strategies can be tried to find the best one. The parameter ObjectNum defines how often the initial solution is copied at beginning of phase two and pushed into the pool. Therefore this parameter affects the number objects that circulate simultaneously, which affects the computational workload.
The evaluation runs were performed with different combinations of values of these parameters. The ranges for the values are the following:

**NetLength:** [8; 40] steps. This range has been chosen regarding to the specifications of the evaluation hardware platform (12 physical cores and additionally 12 hyper-threaded logical cores). This interval leads to a range of threads, starting with a number of threads less than the available physical cores and ending with a number of threads larger than the sum of physical and logical cores. This way all important ranges and transitions of thread numbers are included in the evaluation. This is important for analysing the program behaviour with increasing number of threads with respect to absolute execution time and relative speedup compared to the sequential version. Facts concerning the scheduling behaviour (especially the hyper-threading awareness of the scheduling) are expected.

**ObjectNum:** [25, 50, 75, 100]. This set of parameter values is chosen according to the constraints a solution has to fulfil in our test case. Less than 25 objects leads to very bad solutions and short running times especially when higher values for the parameter NetLength are used, because the pool is empty too soon. Choosing 100 objects leads to execution times and workloads high enough for appropriate evaluation. More objects do not lead to much better solutions nor to more computational workload, just to longer queues of the pipeline stages.

With the results of the evaluation runs we want to analyse the program according to:

- Changes in behaviour of the parallelized and sequential algorithm, respectively, using different parameter values (that lead to different number of threads and variable computational workload).
- The differences of the sequential and parallel version, especially regarding the achieved speedup.

The second, less formal, evaluation part deals with the effort, required to implement the current parallel version using FastFlow based on the already existing sequential algorithm. This effort is again split, on the one hand, into the required changes of the existing sequential code in order to be able to perform parallelization at all, and on the other hand, the assessment of the actual parallel implementation using FastFlow. This comprises the following (ordered chronologically):

- The first step was to get to know FastFlow and its usage. Therefore the documentation about FastFlow and the available patterns has been read. Some of the provided examples had been implemented to get more practice. Changing the programming environment to be able to use FastFlow took nearly no effort.
- Much effort and time was needed to gain knowledge about the optimization algorithm and its sequential implementation in order to be able to perform parallelization at all. How is the optimization actually done, what parts of the code correspond to this, which parts can be parallelized and what are the possible race conditions that arise due to parallel execution and so on.
• The iterative and step-wise approach of the used algorithm made the selection of appropriate pattern hierarchy quite easy. Therefore the design process for the parallel implementation using the pattern approach took little time.

• The main time of the parallel implementation was needed to refactor the existing sequential implementation to make parallelization even possible. The existing implementation was not meant to be parallelized. Many global resources were used, which lead to race conditions. Classes that were instantiated only once in the sequential implementation but several times in the parallel, had e.g. no appropriate copy constructors or assignment operators. Time was needed for the implementation itself, but also to get to know the existing implementation in detail to write appropriate without changing the behaviour of the algorithm.

• After preparing the existing code for parallelization the actual implementation of the initial version using the patterns provided by FastFlow was quite straightforward and took much less time than previously expected.

On the whole about two-thirds of the time was spent on understanding and refactoring the existing implementation in a way that the parallelization could be done. The remaining time was consumed by reading documentation and get to know the use of FastFlow and implementing the parallel version.

This evaluation and comparison of effort and gained advantages should give a basis for decision if the achieved speedup due to parallelization is worth the effort of its implementation. Furthermore, since also alternative parallelization approaches should be implemented, a comparison of needed effort is important, especially compared to the achieved speedup.

6.2.2 Evaluation Results

The first step in evaluation was to determine the number of instantiated threads, their workload and the number of used cores, caused by the selected architecture of used parallel patterns, depending on the selected parameters described above. The observed result was the number of threads always equals to the value of NetLength plus three. The value of NetLength determines the number of pipeline stages, where each stage corresponds to one thread. The additional three threads are the main thread of the program and a thread for emitter and collector of the farm, respectively. During program execution the number of fully utilized cores was always two less than the number of threads. The reason is that the main thread is just waiting for the farm to finish and does not use CPU time and that the emitter has nearly no computations to perform and mostly just waits for objects to arrive in the pool (no busy-waiting). On the whole, this is exactly the expected behaviour.

In the second step, the execution time was recorded with a NetLength from 15 to 20. With each NetLength one run with all values of the range for ObjectNum was performed. As expected the increase of the value for ObjectNum leads to higher execution times in the sequential as well as in the parallel version. The same is true for increased NetLength values. Calculating the resulting speedup achieved
by parallelization showed that the higher the value for ObjectNum was, the greater was the speedup ranging from about 5 to over 7.

The intention of the third step was to evaluate the execution time and speedup over the whole range of values for NetLength as described above, leaving the ObjectNum at a constant value of 100. The results are visualized in Figures 6.1, 6.2 and 6.3.

The evolution of the execution times of the sequential implementation is shown in Figure 6.1. As expected the increase of the execution times is linear, on Tirana as well as on the Xookik cluster. The difference between the two platforms is that the Xookik cluster is much faster in the execution times raise not as fast as on Tirana. The reason for the linear increase is that in every run the number of steps in the optimization chain was increased by 1 and all available optimization steps need about the same time for their computations. If the execution times of the single steps would be very different (as they will in future scenarios) and since the steps are selected randomly, the values would still increase quite linear, but not as perfect as in this example.

More interesting are the results of the parallel implementation, shown in Figure 6.2. As expected the execution times increases with the length of the optimization pipeline, however not linearly. Again the results of the two platforms are quite similar, besides the different values of the absolute execution times.

On Tirana, in the range of 8 to 12 for the NetLength parameter, the execution time is almost constant, most likely because there are more physical cores than stages in the optimization pipeline. At the NetLength of 13 there is jump in the measured execution time (indicated by the green dashed line). The reason for this might be that there are no unused physical cores left and that from this point on the logical cores have to be used.

Upto a NetLength of 24 the execution time is again almost constant, as the available logical cores are used to work on additional pipeline stages. At 25 there is again a jump (indicated by the red dashed line), but this time a larger one. The
reason is probably that now there are more optimization steps (and threads) than cores and thus the cores now have to be shared among the threads.

So far the observed behaviour corresponds to what has been expected. Increasing the NetLength further, it was expected that the execution time again stays almost constant with jumps at multiple values of physical and logical cores. But what has been observed was a great jump (Indicated by the purple dashed line) at a NetLength of 31 (This behaviour was reproducible). So far a reasonable explanation therefore could not be found.

The chart of the Xookik cluster looks similar, but in detail there are differences. Again the execution times increases with longer optimization chains. As on Tirana, there are ranges were the execution times are almost constant or just slightly rising despite increasing number of steps, as well as clear jumps between these values (indicated by the colored dashed lines), but not as drastic as on Tirana. In contrary to the jumps on Tirana, where the jumps correlate with the number of available cores, such a direct correlation is hard to establish on Xookik. The characteristic jumps are at 16, 24 and 33. Definitely no multiples of the numbers of CPU cores.
Further tests have to be performed to find a reasonable explanation (e.g. pinning the threads to the cores of single node).

Figure 6.3 shows the achieved speedup of the parallel implementation compared to the sequential one. The jumps in execution time can again be seen very clearly (again indicated by the three dashed lines).

On Tirana, the highest speedup of nearly 10 was achieved for a NetLength of 24, when all optimization step is executed on a dedicated core. Again a correlation between threads and available cores seems to be the reason. This speedup corresponds to the expected one in deliverable D6.2 of 8 - 12 times.

On Xookik the highest speedup is 5, always reached just before a jump. Furthermore the achieve speedups do not vary as much as on Tirana. Again the reason for the zig-zag-pattern could not be found yet.

6.3 Discussion and Conclusion

Looking at the evaluation results concerned to speedup and scaling in comparison to the time and effort that had to be spent to achieve them, it is obvious that it is worth to invest time. Even when using FastFlow for the first time and with little experience in parallelization at all. More practice will reduce the effort of parallelization even more, because you get familiar with how to apply the parallel patterns and to avoid common pitfalls. This effect of reduced time due to more practice seems greater using FastFlow compared to low level approaches like OpenMP, because the high-level pattern-based approach is not as problem and implementation depended as low level approaches are.

Furthermore, it has to be noted that ParaPhrase does not yet offer all planned features and tools that will support the developer during the parallelization process, e.g. tools for refactoring. Availability of those tools will make parallelization even easier and faster.

A last point that has to be mentioned concerns extensibility and variability. The use-case’s optimization framework is still in an early stage of development. New discrete optimization algorithms will be added and existing ones will have to be changed due to changing requirements. The pattern-based approach makes a proper adaption of the parallelization easier and less expensive.
7. Conclusion

Several conclusions can be drawn from this first experimental evaluation, in particular concerning the areas of porting effort, performance and efficiency of the runtime, and expressiveness of the programming model.

Porting effort

Non-negligible porting effort was necessary for all applications, even if the specific amount varied greatly from one use-case to the other, ranging from few hours to several weeks. Only in two cases, we have quantitative estimates of porting effort for other parallel programming models, specifically OpenMP. In both cases, porting to the the lower-level model OpenMP was not more effort than porting to FastFlow. However, we would not claim that porting to OpenMP is simpler or requires less effort. Interestingly, in general the subjective perception (and expectation) of the programmers was that porting to FastFlow requires less effort than porting to other shared-memory programming models as pthreads or OpenMP.

Most of the necessary recoding was due to refactoring data structures and program logic in such a way as to satisfy the object-oriented nature of the FastFlow library. The overhead for writing worker classes, including constructor, etc, was considerable.

Performance

The performance of the ported applications was in all cases at least comparable to that of the original application. Where an alternative parallel implementation is available, the performance of the FastFlow version is comparable to it. In general, scaling to moderate number of workers/cores was good. Scaling efficiency could decrease for large number of cores, i.e. number of workers comparable to the number of cores available on the node. Depending on the application this behavior might have different causes.

If the granularity of tasks is very small, the overheads of FastFlow library become so large, that the performance drops significantly. This can be avoided by restructuring the application to use larger granularity, for instance by grouping different algorithmic steps in a single worker instance (see Molecular Dynamics and Haar Wavelet Transform use-cases).
A further cause for insufficient scaling efficiency is load-imbalances. In FastFlow this will become particularly severe whenever the number of threads (i.e. actual workers and supporting threads as emitters, collectors, etc) is larger than the number of physical cores available. Currently, the runtime system does not allow to assign tasks to threads in a way that would balance resource utilization across the complete workflow. Instead all threads compete with the same priority for CPU time slot from the operating system (see for instance Discrete Optimization use-case).

A different cause of performance penalties, particularly for memory intensive applications, are NUMA effects. When successive steps of the workflow are assigned to threads in different NUMA domains, this leads to decreasing cache utilization and/or longer memory access times (see Molecular Dynamics use-case).

Also, task allocation to workers in a farm currently is handled in a round-robin fashion, with is a good strategy only if the task have equal execution time and delays from, e.g. the operating system, are negligible (see for instance Haar Wavelet Transform use-case).

**Expressiveness**

The advantage ParaPhrase model is the pattern-based approach to parallelization. These patterns are generally easy to understand, as confirmed in D6.3. Because of the early stage of the project, however, not all high-level, algorithmic idioms present in the use-case applications could be provided as pattern already. In such cases, programmers had to fall-back to more complex, lower-level skeletons to achieve the intended mean. This situation, however, is expected to improve over the course of the project.

Considerable porting time was spent on refactoring code to be compatible with the FastFlow object-oriented approach, which indicates, that the expressiveness still lacks regarding applicability to legacy imperative language codes.

In addition, refactoring tools will become available in the second half of the project. Those will facilitate the taks of porting and add to the expressiveness of the model.
Bibliography