Project no. 644235

REPHRASE

Research & Innovation Action (RIA)
REFACTORING PARALLEL HETEROGENEOUS RESOURCE-AWARE APPLICATIONS – A SOFTWARE ENGINEERING APPROACH

Initial Report on Ported Applications
D6.4

Due date of deliverable: 31/12/2016

Start date of project: April 1^{st}, 2015

Type: Deliverable
WP number: WP6

Responsible institution: Software Competence Center Hagenberg
Editor and editor's address: Michael Rossbory, Software Competence Center Hagenberg

Version 0.1

Project co-funded by the European Commission within the Horizon 2020 Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Change Log

Rev.	Date	Who	Site	What
1	02/12/16	Michael Rossbory	SCCH	Document structure
2	26/12/16	Michael Rossbory	SCCH	Stochastic Local Search
3	25/01/17	Javier Garcia Blas and J. Daniel Gar-cia	UC3M	Medical Imaging
4	20/01/17	Yasser Aleman	CIBERSAM	Verification of the obtained results of pHARDI
5	31/01/17	Michael Rossbory	SCCH	Summary, Introduction, Conclusion
6	07/04/17	Imre Pechan	EvoPro	Railway Diagnosis System

Executive Summary

This deliverable is the report about the initial porting process of the use cases Railway Diagnosis, Medical Image Processing and Stochastic Local Search that have been described in D6.3 using RePhrase tools and technologies that are currently available. Applied technologies and tools are described and an initial performance evaluation is presented. Most of this initial porting has been done manually using the parallel patterns described in D2.1 and implemented in D2.4 using either pure FastFlow or the newly developed GrPPI. Initial application of tools such as the refactoring, pattern discovery or verification tool is described as far as they could be applied on the use cases.

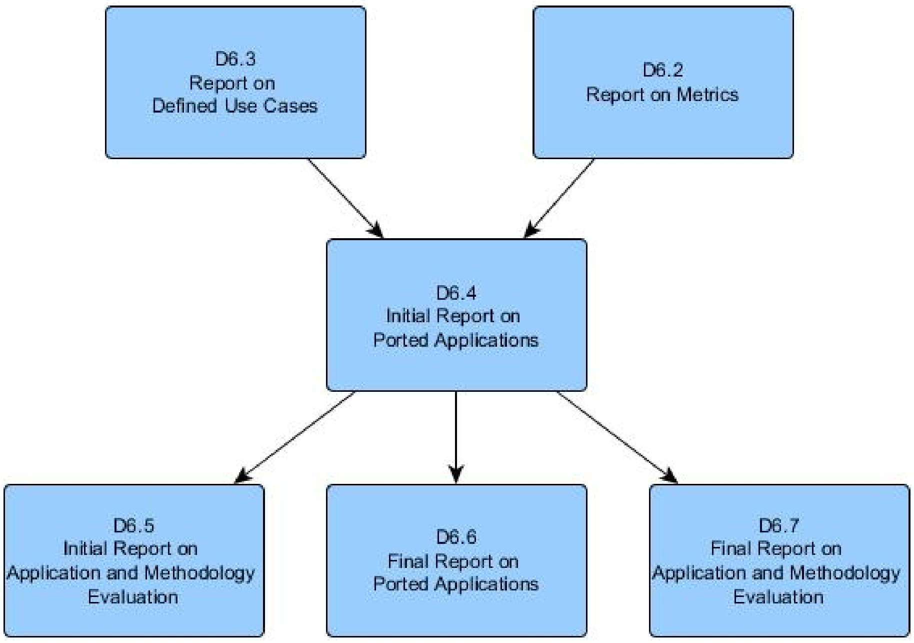


Figure 1: Dependencies of D6.4

Contents

1 Introduction

This deliverable reports the initial porting process of the selected use cases using RePhrase methodologies and tools as far as they were available at about project month 20. The use cases will only be described briefly as far as needed to explain the parallelization applied on the use cases. A detailed description of the use cases has already been given in D6.3 (Report on Selected Use Cases).

In the following chapters every use case partner reports about what RePhrase tools and technologies, mainly developed in WP2 and WP3, have been used to port the initial sequential implementation to a parallel implementation. Furthermore an initial evaluation of the parallel implementations compared to the sequential implementation concerning performance has been performed. This initial evaluation focuses on functionality and performance improvements gained throw parallelization. A detailed evaluation based on the metrics defined in D6.2 will be reported in D6.5.

Currently available are most of the initial patterns defined in D2.1 and implemented in D2.4 in an initial implementation. High-level patterns, such as the pool pattern are currently only available when using the FastFlow interface. The pattern interface does only allow for specifying the parallelism degree. Additional non-functional parameters can not be defined so far. The refactoring, pattern discovery and verification tool are in an early development phase and still lack off functionality and usability to fully support (semi)automatic parallelization and verification of the use cases. Current application of the tools will be reported in this deliverable. Using a continuous evaluation and feedback process the tools are improved to finally support the needs of the use cases.

2 Railway Diagnosis System

2.1 Introduction

Evopro's use case has been described in detail in deliverable D6.3. In a nutshell the use case consists of two parts: the sensor algorithm which is executed on the samples that are being produced by 24 analogue sensors and the post-processing of that measurement data in order to produce the final load results. In D6.3, we have already elaborated a list of potential parallelisation options, taking also into account the physical requirement originating from the industrial deployment of the eRDM system. Thus, in order to measure the effectiveness of code parallelisation for later tool benchmarking purposes, that is, to create a clear baseline, we have selected that part of the code base which seemed to be the most promising source of demonstrable results. Also, we have evaluated a few parallelisation patterns, three different embedded platforms and various simulated sensor measurements in order to survey the gains through direct parallelisation. During this work the code was ported manually; the following sections describe the consecutive steps and corresponding results. We also created a parallel code version using the automatic code transformations of the ParaFormance tool; these results are presented in 2.5.

2.2 Selected algorithm

The sensor algorithm calculates the convolution over the signal samples measured by the strain gauge bridges. The calculation takes places through two parallel FIR (finite impulse response) filters, each having 52 pre-calculated TAPS (filter coefficients). The output of the calculation is the multiplied sum of the current and the previous input signal samples and the filter coefficients. The relevant code in sequential format is shown below in Listing 2.1.

Listing 2.1: Original sequential filtering code

```
const float* bp_coeffs = RCBProc::bandpass_coefs;
const float* hil_coeffs = RCBProc::hilbert_coefs;
unsigned int numOfCoeffs = RCBProc::filterLength;

//BENCHMARK START
double startTime = Utils::getTimeSec();

```

```
//filtering
for (unsigned int rcbIdx = 0; rcbIdx < numberOfRCBs; rcbIdx++) {

    float* inputSignal = RCBs[rcbIdx].rcbProc.rawData.data();
    float* filteredSignal = RCBs[rcbIdx].rcbProc.filteredData.data();
    unsigned int numOfSamples = RCBs[rcbIdx].rcbProc.rawData.size();

    for (unsigned int sampIdx = 0; sampIdx < numOfSamples; sampIdx++) {
        filteredSignal[sampIdx] = 0.0f;
        if (sampIdx >= numOfCoeffs - 1) {
            float bp_temp = 0.0;
            float hil_temp = 0.0;
            for (unsigned int coeffIdx = 0; coeffIdx < numOfCoeffs;
               (cont.)coeffIdx++) {
                bp_temp += bp_coeffs[coeffIdx] * inputSignal[sampIdx -
                   (cont.)coeffIdx];
                hil_temp += hil_coeffs[coeffIdx] * inputSignal[sampIdx -
                   (cont.)coeffIdx];
            }
            filteredSignal[sampIdx] = sqrt(bp_temp * bp_temp + hil_temp *
               (cont.)hil_temp);
        }
    }

}

//BENCHMARK STOP
std::cout << "Total program run time:" << Utils::getTimeSec() - startTime
   (cont.)<< "sec" << std::endl;
```

As can be seen in Listing 2.1, there is a three-fold nesting of loops to carry out the convolution. The outer loop represents the sensors, the middle loop stands for the actual samples and the inner loop is where the actual calculation of the convolution with the pre-set filter coefficients takes place. From our current perspective we labelled both the outer loop (due simulated data) and the inner loop (due granularity) as not relevant, so we focused on the refactoring of the middle loop along the FastFlow patterns.

2.3 Chosen platforms

Evopro's use case fits well state-of-the-art heterogeneous platforms, due to its industrial deployment, preferable multi-core ARM based many-core GPU SoCs. In order not to prematurely constrain the spectrum of potential hardware solutions we opted for parallelisation via FastFlow, however, we evaluated both multi-core CPU and many-core GPU solutions individually. The following hardware platforms have been tried:

- Xeon Server (48 CPU-cores, 2816 GPU-cores)
  - 2x Intel Xeon E5-2695 v2 CPU (x86) (30M Cache, 2.40 GHz) (12 physical cores each, i. e. 48 cores in total by HyperThreading)
  - 8x16 GB of DDR3 1600 MHz memory

  - AMD Radeon R9 290X graphics card (2816 cores, 4GB GDDR5 / 512 bit)

- Nvidia Jetson TK1 board (4 CPU-cores, 192 GPU-cores)
  - NVIDIA 4-Plus-1 Quad-Core ARM Cortex-A15 CPU (2.32 GHz)
  - NVIDIA Kepler GPU with 192 CUDA cores
  - 2 GB of DDR3L 933MHz memory

- Gizmo 2 board (2 CPU-cores, 80 GPU-cores)
  - AMD GX-210HA Dual-Core CPU (x86) (1M Cache, 1 GHz), i. e. 2 cores
  - AMD Radeon HD 8210E with 80 cores
  - 1GB of DDR3 1600 MHz memory

2.4 Parallelisation with manual refactoring

The focus of manual parallelisation is the convolution along the input data samples; therefore three Fastflow parallelisation patterns seemed to be the best candidates for the task: parallel_for (targeting the CPU), stencil, and farm combined with stencil (targeting the GPU). The three different solutions are elaborated in the following subsections.

2.4.1 Applying the parallel_for pattern

The kernel can be trivially embedded into the FastFlow Task structure: the original convolution code can be copied verbatim, the context parameters must be provided via the parallel_for construct. Finally, the Task structure gets these parameters via its constructor (see Listing 2.2).

The invocation of the kernel is also straight-forward: at the place of the middle for loop Listing 2.1, this time a Task is created, then the stages are set-up and finally a FastFlow pipe is constructed. Then, by calling run_and_wait_end() on the pipe the kernel is invoked; the rest of the code remains untouched (see Listing 2.3).

Listing 2.2: Filtering kernel code using parallel_for pattern

```
struct Task{

    Task(float* inputSignal, const float* bp_coeffs, const float*
       (cont.)hil_coeffs, float* filteredSignal, unsigned int numOfSamples)
       (cont.) :
    inputSignal { inputSignal }, bp_coeffs { bp_coeffs }, hil_coeffs {
       (cont.)hil_coeffs }, filteredSignal { filteredSignal }, numOfSamples
       (cont.) { numOfSamples } {}

    float* inputSignal;
    const float* bp_coeffs;
    const float* hil_coeffs;
    float* filteredSignal;
    unsigned int numOfSamples;
};

struct StageInit : ff_node_t<Task> {

    StageInit(Task& taskInit) : task(taskInit) {}
    Task *svc(Task *in) {
        ff_send_out(&task);
        return EOS;
    }

private:
    Task& task;
};

struct Stage: ff_Map<Task> {

    Stage(unsigned int& numOfCoeffs, unsigned int& numOfSamples) :
    numOfCoeffs { numOfCoeffs }, numOfSamples { numOfSamples }/*, ff_Map<
       (cont.)Task> { 4 }*/ {}

    Task* svc(Task* in) {
        float* inputSignal = in->inputSignal;
        const float* bp_coeffs = in->bp_coeffs;
        const float* hil_coeffs = in->hil_coeffs;
        float* filteredSignal = in->filteredSignal;
        ff_Map<Task>::parallel_for(0, numOfSamples, [bp_coeffs, &
           (cont.)filteredSignal, hil_coeffs, inputSignal, this](unsigned
           (cont.)int sampIdx) {
            filteredSignal[sampIdx] = 0.0f;
            if (sampIdx >= numOfCoeffs - 1) {
                float bp_temp = 0.0;
                float hil_temp = 0.0;
                for (unsigned int coeffIdx = 0;coeffIdx < numOfCoeffs;
                   (cont.)coeffIdx++) {
                    bp_temp += bp_coeffs[coeffIdx] * inputSignal[sampIdx -
                       (cont.) coeffIdx];
                    hil_temp += hil_coeffs[coeffIdx] * inputSignal[sampIdx
                       (cont.) - coeffIdx];
                }
                filteredSignal[sampIdx] = sqrt(bp_temp * bp_temp +
                   (cont.)hil_temp * hil_temp);
            }
        });
        return GO_ON;
    }

private:
    unsigned int& numOfCoeffs;
    unsigned int& numOfSamples;
};
```

Listing 2.3: Filtering kernel call site code using parallel_for pattern

```
const float* bp_coeffs = RCBProc::bandpass_coefs;
const float* hil_coeffs = RCBProc::hilbert_coefs;
unsigned int numOfCoeffs = RCBProc::filterLength;

//Benchmark START
double startTime1= Utils::getTimeSec();

```

```
for (unsigned int rcbIdx = 0; rcbIdx < numberOfRCBs; rcbIdx++) {

    float* inputSignal = RCBs[rcbIdx].rcbProc.rawData.data();
    float* filteredSignal = RCBs[rcbIdx].rcbProc.filteredData.data();
    unsigned int numOfSamples = RCBs[rcbIdx].rcbProc.rawData.size();

    Task task { inputSignal, bp_coeffs, hil_coeffs, filteredSignal,
       (cont.)numOfSamples };
    StageInit stage_init(task);
    Stage stage { numOfCoeffs, numOfSamples };
    ff_Pipe<Task> pipe(stage_init, stage);

    pipe.run_and_wait_end();
}

//BENCHMARK STOP
double endTime= Utils::getTimeSec();
std::cout <<"Program run time:" << endTime - startTime1 << "sec" << std::
   (cont.)endl << std::endl;
```

2.4.2 Applying the stand-alone stencil pattern

Stencil is the obvious parallelisation pattern for carrying out convolutions since it allows the accumulation of the results via the map-reduce operation. We relied on FastFlow's stencil pattern over OpenCL when we defined the kernel.

The kernel definition shown in Listing 2.4 is again mirroring the inner for loop of Listing 2.1, but this time it relies on a macro to facilitates the injection of the context variables (e.g. indexes to input samples, coefficients) into the body of the kernel. oclTask is the specialised FastFlow task structure that sets up the data input/output relation to and from the kernel (see Listing 2.5).

Finally, the kernel is invoked by first initialising an oclTask instance with the sample input data and then mixing it in with the rest of the parameters, including the name of the kernel. When the final data structure is ready, a GPU unit is selected and run_and_wait_end() is called on it (see Listing 2.6).

Listing 2.4: Filtering kernel code using stencil pattern

```
FF_OCL_STENCIL_ELEMFUNC_2ENV(mapfenv, float, size, k, M, float, bp_coeff,
   (cont.)float, hil_coeff,
    (void)size;
    float bp_temp = 0.0;
    float hil_temp = 0.0;

    if (k >= 52){
        for(int i = 0; i < 52; i++) {
            bp_temp += bp_coeff[i] * M[k-i];
        }
        for(int i = 0; i < 52; i++) {
            hil_temp += hil_coeff[i] * M[k-i];
        }

        return sqrt(bp_temp*bp_temp + hil_temp*hil_temp);
    } else {
        return 0.0;
    }
);
```

Listing 2.5: Task definition for filtering kernel call site code using stencil pattern

```
struct oclTask: public ff::baseOCLTask<oclTask, float> {

    oclTask() :
        M(nullptr), out(nullptr), env(nullptr), size(0), result(0.0) {
    }

    oclTask(float *input, float * output, size_t size, env_t * env) :
        M(input), out(output), env(env), size(size), result(0.0){
    }

    void setTask(oclTask *t) {
        assert(t);
        setInPtr(t->M, t->size);
        setOutPtr(t->out, t->size);
        setEnvPtr(t->env->bp_coeffs, 52);
        setEnvPtr(t->env->hil_coeffs, 52);
        setReduceVar(&t->result);
    }

    float * M;
    float * out;
    env_t * env;
    const size_t size;
    float result;
};
```

Listing 2.6: Filtering kernel call site code using stencil pattern

```
void do_gpu_filtering(std::vector<float>& inpSignal, std::vector<float>&
   (cont.)filteredSignal) {

    env_t env;

    oclTask task(inpSignal.data(), filteredSignal.data(), inpSignal.size()
       (cont.), &env);

    ff_stencilReduceLoopOCL_1D<oclTask> srl(task, mapfenv, "", 1.0,
       (cont.)nullptr, 1, 52);
    srl.pickGPU();
    srl.run_and_wait_end();

    return;
}
```

2.4.3 Applying the stencil and map patterns combined

In the case of the FastFlow farm pattern, the kernel definition remains the same stencil-based one as already shown in Listing 2.4. However, this time explicit Emitter, Collector and Worker definitions are necessary. The Emitter prepares the input samples for processing, the Worker invokes the kernel with the needed parameters as already shown in Listing 2.6. Nevertheless, there is a slight difference here since there is no need to call run_and_wait_end() as the Collector has taken over that role (see the map node definitions in Listing 2.7).

Finally, the farm is constructed by connecting together the Emitter, the Collector and the Worker nodes of the FastFlow network. When the run_and_wait_end() is invoked on the farm the convolution kicks off on the input samples (see Listing 2.8).

Listing 2.7: Nodes for the map-based filtering kernel

```
struct Emitter: ff::ff_node {
    Emitter(std::vector<std::vector<float>> & inpSignals, std::vector<std
       (cont.)::vector<float>> & filteredSignals, env_t & env) :
        inpSignals(inpSignals), filteredSignals(filteredSignals), env(env)
           (cont.) {
    }

    void * svc(void *) {
        for(unsigned i = 0; i < inpSignals.size(); i++) {
            oclTask * T = new oclTask(inpSignals[i].data(),
               (cont.)filteredSignals[i].data(), inpSignals[i].size(), &env
               (cont.));
            ff_send_out(T);
        }

        return EOS;
    }

    std::vector<std::vector<float>> & inpSignals;
    std::vector<std::vector<float>> & filteredSignals;
    env_t & env;
};

struct Collector: ff::ff_node_t<oclTask> {
    oclTask * svc(oclTask * t) {
        return GO_ON;
    }
};

struct Worker: ff::ff_stencilReduceLoopOCL_1D<oclTask> {
    Worker(std::string mapf, const size_t nacc) :
        ff_stencilReduceLoopOCL_1D<oclTask>(mapf, "", 1.0, nullptr, nacc,
           (cont.)52) {
        pickGPU();
    }
};
```

Listing 2.8: Call site code for the map-based filtering kernel

```
void do_gpu_filtering_farm(std::vector<std::vector<float>> & inputSignals,
   (cont.) std::vector<std::vector<float>> & filteredSignals,const int
   (cont.)nWorkers) {

    //const int nWorkers = 3;
    const int nAccelerators = 1;

    env_t env;
    std::vector<std::unique_ptr<ff_node>> w;

    for(int i = 0; i < nWorkers; i++) {
        w.push_back(make_unique<Worker>(mapfenv, nAccelerators));
    }
    Emitter e(inputSignals, filteredSignals, env);
    Collector c;
    ff_Farm<> farm(std::move(w), e, c);
    farm.run_and_wait_end();
}
```

2.4.4 Measurement results

We have carried out some performance measurements on the different parallelisation patterns in order to measure their speed-up gains. Also, we have generated various input sample data to model the different train length and carriage distribution. Although the measurements are only precursory of a more detailed metrics driven analysis at a later stage, the preliminary results are rather promising indeed. The measurements on Figure 2.1 show the execution times on the Xeon server under various train length. Parallelisation is very efficient, it shows almost constant execution times independently of the length of the train. This is in stark contrast with the sequential code where single CPU limitation results in polynomial increase of execution time.

Similar results were obtained for the other two platforms, but the CPU and GPU limitations of the embedded computing boards distorted the constant execution times whenever the hardware limitations kicked in. The respective measurements are shown on Figure 2.2 and 2.3.

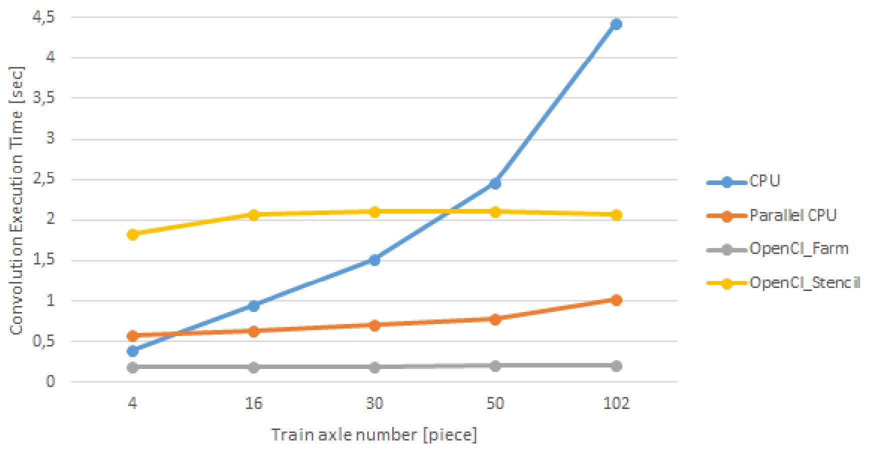


Figure 2.1: Results on Xeon server

2.5 Parallelisation with automatic refactoring

We created another version of the use case code, relying on the automatic transformations provided by the ParaFormance tool. Again we focused on the filtering code just like in the previous sections. In this case, however, we applied a farm pattern targeting the CPU, and parallelized the outermost loop shown in Listing

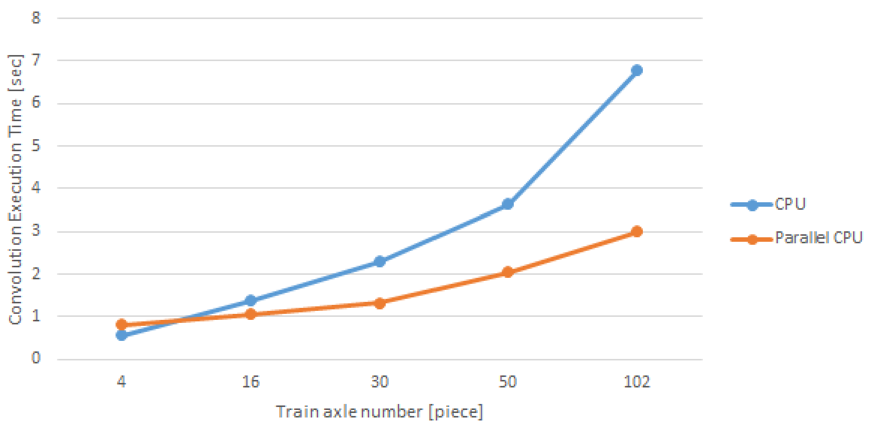


Figure 2.2: Results on Jetson board

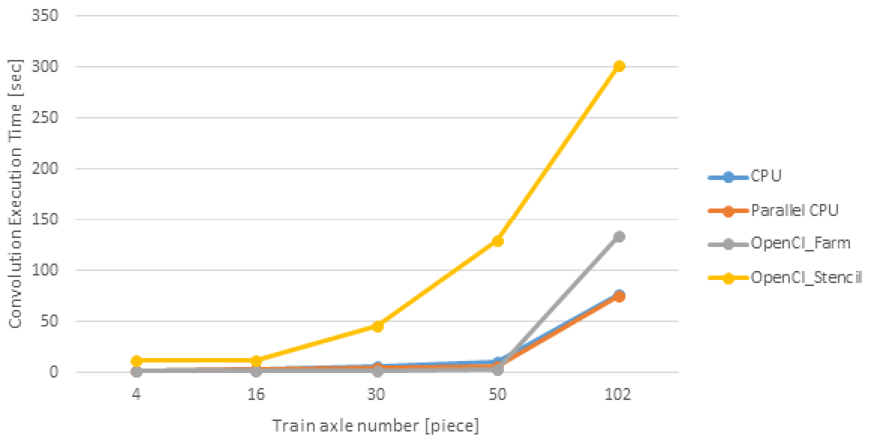


Figure 2.3: Results on Gizmo board

2.1. During code transformation the following automatic steps were carried out (supported by the ParaFormance tool):

- First the body of the for loop to be implemented as a farm pattern was extracted to a function, so that the loop body contains only a single function call.
- Then the function argument list was refactored so that the function operates only on a single argument (implemented as a tuple object).
- In the next step a Component object instantiation was inserted automatically in the code, based on the for loop.
- Then the FastFlow farm declaration was inserted, according to the created Component object and the for loop.
- Finally, the FastFlow farm implementation code was added.

Result of the first two steps can bee seen in Listing 2.9, and the code implementing the farm (result of the latter three steps) is shown in Listing 2.10.

Listing 2.9: Transformed filtering function

```
void filtering_fun(unsigned int rcbIdx, unsigned int numOfCoeffs, const
   (cont.)float* bp_coeffs, const float* hil_coeffs,std::vector<tRCB>& RCBs
   (cont.)) {
{
    float* inputSignal = RCBs[rcbIdx].rcbProc.rawData.data();
    float* filteredSignal = RCBs[rcbIdx].rcbProc.filteredData.data();
    unsigned int numOfSamples = RCBs[rcbIdx].rcbProc.rawData.size();
    for (unsigned int sampIdx = 0; sampIdx < numOfSamples; sampIdx++) {
        filteredSignal[sampIdx] = 0.0f;
        if (sampIdx >= numOfCoeffs - 1) {
            float bp_temp = 0.0;
            float hil_temp = 0.0;
            for (unsigned int coeffIdx = 0; coeffIdx < numOfCoeffs;
            coeffIdx++) {
                bp_temp += bp_coeffs[coeffIdx] * inputSignal[sampIdx -
                   (cont.)coeffIdx];
                hil_temp += hil_coeffs[coeffIdx] * inputSignal[sampIdx -
                   (cont.)coeffIdx];
            }
            filteredSignal[sampIdx] = sqrt(
            bp_temp * bp_temp + hil_temp * hil_temp);
        }
    }
}
}

void filtering_fun(const std::tuple<unsigned int, unsigned int, const
   (cont.)float *, const float *, vector<tRCB, allocator<tRCB>>&> &args) {
    return filtering_fun(std::get<0>(args), std::get<1>(args), std::get
       (cont.)<2>(args), std::get<3>(args), std::get<4>(args));
}
```

Listing 2.10: Generated taskfarm

```
const float* bp_coeffs = RCBProc::bandpass_coefs;
const float* hil_coeffs = RCBProc::hilbert_coefs;
unsigned int numOfCoeffs = RCBProc::filterLength;

//BENCHMARK START
double startTime = Utils::getTimeSec();
Component<std::tuple<unsigned int&,unsigned int&,const float*&,const float
   (cont.)*&,std::vector< tRCB,std::allocator< tRCB> > &> > component(
   (cont.)filtering_fun);
ff_farm<> farm = true;
std::vector<ff_node*> workers;
for (int i = 0; i < 8; i++)
    workers.push_back(&component);
farm.add_workers(workers);
farm.run_then_freeze();
//filtering
for (unsigned int rcbIdx = 0; rcbIdx < numberOfRCBs; rcbIdx++)
    farm.offload(std::forward_as_tuple(rcbIdx, numOfCoeffs, bp_coeffs,
       (cont.)hil_coeffs, RCBs));
farm.offload((void*) (FF_EOS));
farm.wait_freezing();

```

```
//BENCHMARK STOP
std::cout << "Total program run time:" << Utils::getTimeSec() - startTime << "sec" << std::endl;
```

This code needs some further refinements before compilation, which must be currently made manually. The final code for the filtering function and the definition of the Component class can be seen in Listing 2.11. The altered code for the taskfarm using 16 workers is shown in Listing 2.12.

Listing 2.11: Altered filtering function along with Component class definition

```
void filtering_fun_inner(unsigned int rcbIdx, unsigned int numOfCoeffs, const float* bp_coeffs, const float* hil_coeffs, tRCB& RCBs) {
{
    float* inputSignal = RCBs.rcbProc.rawData.data();
    float* filteredSignal = RCBs.rcbProc.filteredData.data();
    unsigned int numOfSamples = RCBs.rcbProc.rawData.size();
    for (unsigned int sampIdx = 0; sampIdx < numOfSamples; sampIdx++) {
        filteredSignal[sampIdx] = 0.0f;
        if (sampIdx >= numOfCoeffs - 1) {
            float bp_temp = 0.0;
            float hil_temp = 0.0;
            for (unsigned int coeffIdx = 0; coeffIdx < numOfCoeffs;
            coeffIdx++) {
                bp_temp += bp_coeffs[coeffIdx] * inputSignal[sampIdx - coeffIdx];
                hil_temp += hil_coeffs[coeffIdx] * inputSignal[sampIdx - coeffIdx];
            }
            filteredSignal[sampIdx] = sqrt(
            bp_temp * bp_temp + hil_temp * hil_temp);
        }
    }
}
}

std::tuple<unsigned int, unsigned int, const float *, const float *, tRCB&>* filtering_fun(std::tuple<unsigned int, unsigned int, const float *, const float *, tRCB&> *args) {
    filtering_fun_inner(std::get<0>(*args), std::get<1>(*args), std::get<2>(*args), std::get<3>(*args), std::get<4>(*args));
    return args;
}

template <class T>
class Component: public ff_node {

    public:

    Component(T* (*worker)(T*)): worker(worker), final(false) {}

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

    void* svc(void* task) {

        if (final == true) {
            T* taskin = (T*) task;
            results[idx++] = (*worker)(taskin);
            return GO_ON;
        } else{
```

```
        T* taskin = (T*) task;
        T* taskout = (*worker)(taskin);
        return ((void *)taskout);
    }
}
T* callWorker(T* x) {
    T* res = worker(x);
    delete x;
    return res;
}

protected:

T* (*worker)(T*); // the worker function
bool final;
int idx;
T** results;
};
```

Listing 2.12: Altered taskfarm code

```
const float* bp_coeffs = RCBProc::bandpass_coefs;
const float* hil_coeffs = RCBProc::hilbert_coefs;
unsigned int numOfCoeffs = RCBProc::filterLength;
int numWorkers = 16;

//BENCHMARK START
double startTime = Utils::getTimeSec();

ff_farm<> farm(true);
std::vector<ff_node*> workers;
for (int i = 0; i < numWorkers; i++)
    workers.push_back(new Component<std::tuple<unsigned int,unsigned int,
       (cont.)const float*,const float*,tRCB&> >(filtering_fun));
farm.add_workers(workers);
farm.run_then_freeze();
//filtering
vector<std::tuple<unsigned int,unsigned int,const float*,const float*,tRCB
   (cont.)&> > inputVec;
for (unsigned int rcbIdx = 0; rcbIdx < numberOfRCBs; rcbIdx++) {
    inputVec.push_back(std::tuple<unsigned int,unsigned int,const float*,
       (cont.)const float*,tRCB&>(rcbIdx, numOfCoeffs, bp_coeffs,
       (cont.)hil_coeffs, RCBs[rcbIdx]));
    farm.offload((void *) &inputVec.back());
}
farm.offload((void*) (FF_EOS));
farm.wait_freezing();
for (int i = 0; i < numWorkers; i++)
    delete workers[i];

//BENCHMARK STOP
std::cout << "Total program run time:" << Utils::getTimeSec() - startTime
   (cont.)<< "sec" << std::endl;
```

2.5.1 Measurement results

Performance measurement for the described taskfarm-based implementation exploiting the ParaFormance automatic transformations were performed on our Xeon-based server. Figure 2.4 shows the results we got for the CPU-based implementations running on this platform; the sequential code, the parallel_for-based version (these are also shown on Figure 2.1) and the taskfarm-based versions. The last one gives the best performance by far, being more than one order of magnitude faster compared to the sequential version for each input size (an average of x10.6 thanks to the 16 parallel workers applied). In contrast, the parallel_for-based solution only achieves a moderate speedup (x2.37 averagely for the different input size values), which indicates that selecting the middle for loop for parallelisation in the original code (see Listing 2.1) is suboptimal on the CPU (although it suits the GPU architecture well, due to the large number of potential parallel workers processing distinct samples of the input signal). On CPU, processing the different sensors in parallel (leading to lower number of parallel workers, but less overhead) is favourable.

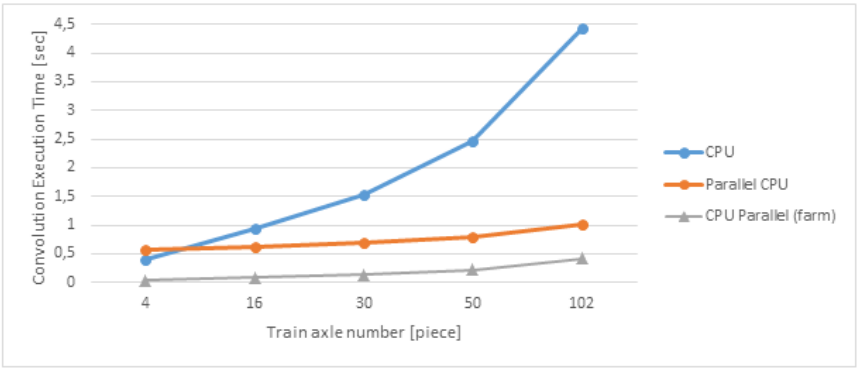


Figure 2.4: Comparing CPU-based results acquired on Xeon server

3 Medical Image Processing

3.1 Introduction

In recent years, diffusion magnetic resonance imaging (MRI) has played an important role in the investigation of morphological features of brain tissues. When the diffusion is constrained by the presence of obstacles, this technique yields information about the confining geometry. It provides a new dimension of MRI contrast based on water mobility, because of the different cellular environment experienced by water molecules. The study of this phenomenon has made possible a deeper knowledge of the microanatomy of the living brain and others complex tissues, due to its noninvasive nature.

In this sense, the use of the MATLAB language has become over the years *de facto* standard for designing and prototyping a wide range of applications in both science and engineering areas. Because of its easy programming model, the scientific and technical communities have chosen to use this high-level language for developing application prototypes, which require the computation of linear algebra problems. By contrast, many of these prototypes have moved to a production stage without being properly adapted and optimized to handle large workloads. Therefore, today it is possible to find numerous examples of MATLAB-implemented applications running on high-performance production platforms but not fully exploiting the benefits of such parallel hardware.

From the set of methods and algorithms proposed to date for analyzing this type of medical imaging, we will focus on the collection of MATLAB routines, namely HARDI-Tools `http://neuroimagen.es/webs/hardi_tools/`, dedicated to the intravoxel reconstruction from High Angular Resolution Diffusion Imaging (HARDI) data. Within the set of techniques included in HARDI-Tools, in this deliverable, we have focused on the parallelization of the RUMBA-SD method (the source code presented in this work is available at the project Web site[1]), namely pHARDI. However, because of the modular design of the proposed solution, including new methods in the future will not require considerable efforts.

[1]See: `https://bitbucket.org/fjblas/phardi`

3.2 pHARDI Architecture

As shown in Fig. 3.1, pHARDI has a layer-based design, which allows the use of multiple linear algebra accelerators in a wide range of devices, such as multi-core devices GPU (both CUDA and OpenCL) or even co-processors, like Intel Xeon Phi. In the case the platform that does not incorporate any accelerator, our solution can also run on multi-core processors using highly-tuned linear algebra libraries. We use Armadillo on top of the linear algebra accelerators for providing a common interface. Additionally, we take advantage of Armadilllo for reflecting most of the MATLAB language (e.g. $remap$, $reshape$, etc.).

With the aim of supporting GPU devices, we have developed two different versions: one that is totally based on the Armadillo library and a second leveraging the ArrayFire library. To develop the second approach, only fragments of code that are computationally more expensive, namely kernels, have been implemented using ArrayFire routines. It is important to remark that both solutions use a *column-major* logical layout of the matrices and volumes handled, in both CPU and GPU. Therefore, no additional changes in the memory access pattern are required in the code for running both versions.

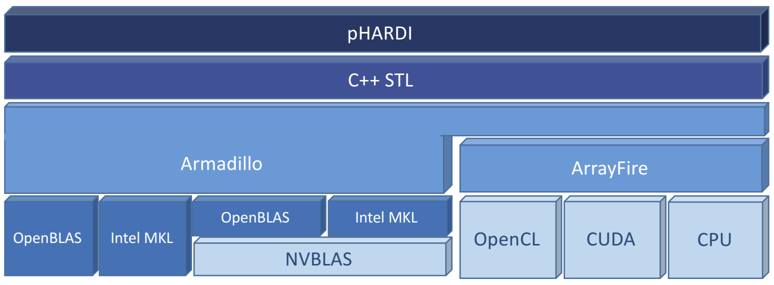


Figure 3.1: Layered software architecture of pHARDI.

We note that the only drawback of using Armadillo is the lack of parallelization of element-wise operations (as shown in Subsection 3.4). In order to cope with this, we have parallelized these operations using OpenMP. However this approach limits the portability to accelerated solutions like NVBLAS. In contrast, ArrayFire supports automatic element-wise operations, which facilitates the development and increases performance.

Regarding data management, we have chosen the ITK library [7], which facilitates the management for reading and writing data in different formats related to the medical imaging area, such as DICOM and Nifti. Another advantage of using ITK is the support for automatic file compression. This fact, significantly reduces the storage space required by the applications.

pHARDI supports two data access patterns for processing the input data. The first pattern separately processes each slice from the input data volumes ($x \times y \times z$), resulting in a total of z slices, each one of $x \times y \times t$ voxels, being t the number of orientations. The second layout processes all pixels from the volumes in a single matrix. This matrix has a dimension of $n \times m$ elements, where n corresponds with the amount of evaluated orientations (e.g. 100) and m with the amount of voxels on each volume (e.g 125 megapixels).

Listing 3.1: pHARDI computation kernel: intravox fiber reconstructor in RUMBA-SD.

```
for (size_t i = 0; i < Niter; ++i) {
    Ratio = mBessel_ratio<T>(n_order,Reblurred_S);

    #pragma omp parallel for simd
    for (size_t k = 0; k < SR.n_cols; ++k )
        for (size_t j = 0; j < SR.n_rows; ++j)
            SR(j,k) = Signal(j,k) * Ratio(j,k);
    KTSR = KernelT * SR;
    KTRB = KernelT * Reblurred;

    #pragma omp parallel for simd
    for (size_t k = 0; k < fODF.n_cols; ++k)
        for (size_t j = 0; j < fODF.n_rows; ++j)
            fODF(j,k) = fODF(j,k) * KTSR(j,k) / (KTRB(j,k) + std::
               (cont.)numeric_limits<double>::epsilon());

    Reblurred = Kernel * fODF;

    #pragma omp parallel for simd
    for (size_t k = 0; k < Signal.n_cols; ++k)
        for (size_t j = 0; j < Signal.n_rows; ++j)
            SUM(j,k) = (pow(Signal(j,k),2) + pow(Reblurred(j,k),2))/2 - (
               (cont.)sigma2(j,k) * (Signal(j,k) * Reblurred(j,k)) / sigma2
               (cont.)(j,k)) * Ratio(j,k);

    sigma2_i = (1.0/N) * sum( SUM , 0) / n_order;

    #pragma omp parallel for
    for (size_t k = 0; k < sigma2_i.n_elem; ++k)
        sigma2_i(k) = std::min<T>(std::pow<T>(1.0/10.0,2),std::max<T>(
           (cont.)sigma2_i(k), std::pow<T>(1.0/50.0,2)));

    sigma2 = repmat(sigma2_i, N, 1);
}
```

Listing 3.1 shows the most time consuming part of RUMBA-SD. This code is executed a determined number of iterations ($Niter$). Each iteration is composed by three matrix multiplications and multiple element-wise multiplications. As far as we know, Armadillo lacks of a parallel implementation of element-wise multiplication calls, so these calls have been parallelized by OpenMP, while preserving the data access primitives of Armadillo. As explained before, Armadillo stores matrices in a row-major way. Loops are also parallelized by applying the $simd$ OpenMP pragma, with the aims of vectorizing the content of each loop.

3.3 Initial Porting Setup based on GrPPI

The aims of this section is to detail the manual steps carried out for paralellizing an initial implementation of pHARDI using GrPPI. In this case, we have identified that the most suitable parallel patterns are Pipelineand Farm. A reduced code fragment of pHARDI paralellized by GrPPI can be seen in Listing 3.2. Initially, we manually identify the parallel patterns that fit with the existing code. First, we identify that the loop in charge of processing each slice of the input data can be adapted to a Pipelineparallel pattern (Line 4), given that each slice can be acquired (Line 5), processed (Line 23), and stored (Line 58) independently. So, the three stages of the Pipelinepattern follow this processing order. Then, the processing stage (second stage) can be transformed in a Farmparallel pattern. This middle stage can process in parallel the calculation of the ODF resulting matrix. The input data of this stage corresponds with a modified matrix obtained in the previous stage and passed by parameter in the lambda function. In all the stages, we found that the use of tuples is a good alternative to forward the corresponding ouputs to the next stage.

One of the mains advantages of GrPPI in conjunction with pHARDI use case is the possibility of overlapping both computation and I/O-related stages, reducing the total execution time as demonstrated in the evaluation section of this chapter.

Listing 3.2: pHARDI paralellized by using GrPPI.

```
1
2  parallel_execution_omp p{};
3  parallel_execution_omp f{};
4  Pipeline( p,
5      // Pipeline stage 0
6      //[&]() {
7        [=, &Vmask, &Vdiff, &nslice]() {
8            LOG_INFO << "Processing Stage 0: Slice " << nslice ;
9            if (nslice == zdiff)
10              return optional<paramStage0>();
11           else {
12              Cube<T> Idiff(xdiff, ydiff, Ngrad);
13              for (int graddir = 0; graddir < Ngrad; ++graddir) {
14                     Idiff.slice(graddir) = Vdiff[graddir].slice(nslice) %
                          (cont.)Vmask.slice(nslice); // product-wise
                          (cont.)multiplication
15              }
16              paramStage0 tupleIdiff = std::make_tuple(nslice, Vmask.slice(
                   (cont.)nslice), Idiff);
17              nslice++;
18              return optional<paramStage0>(tupleIdiff);
19            }
20       },
21
22       Farm(f,
23             [=, &Kernel, &ind_S0](paramStage0 tupleIdiff) {
24                int slice     = std::get<0>(tupleIdiff);
25                LOG_INFO << "Processing Stage 1: Slice " << std::get<0>(
                     (cont.)tupleIdiff);
26                Mat<T> Vmasks = std::get<1>(tupleIdiff);
27                Cube<T> Idiff = std::get<2>(tupleIdiff);
28
29                std::cout << Vmasks.size() << std::endl;
30
31                size_t  totalNvoxels = Vmasks.n_elem;
32
33                    Cube<T> globODFslice (xdiff,ydiff,Nd,fill::zeros);
```

```
               Row<T> ODF_iso(ydiff,fill::zeros);

               Cube<T> S0_est (xdiff, ydiff, ind_S0.n_elem);
               Mat<T> S0_est_M (xdiff, ydiff);

               for (int i = 0; i < ind_S0.n_elem; ++i) {
                   S0_est.slice(i) = Idiff.slice(ind_S0(i));
               }


            ....

               Mat<T>allIndexesODF = repmat(inda.t(),ODF.n_rows,1);
               Mat<T> ODFindexes = allIndexesODF + totalNvoxels * repmat(
                  (cont.)linspace<Mat<T>>(0, ODF.n_rows - 1, ODF.n_rows )
                  (cont.),1,inda.n_elem);
               for (int j = 0; j < ODF.n_cols; ++j) {
                       for (int i = 0; i < ODF.n_rows; ++i) {
                        globODFslice.at(ODFindexes(i,j)) = ODF(i ,j);
                       }
               }

               return std::make_tuple(slice, globODFslice, slicevf_CSF,
                  (cont.)slicevf_GM, slicevf_WM, slicevf_GFA);
            }),

      [&]( paramStage1 tupleRes ) {
            int slice             = std::get<0>(tupleRes);
            Cube<T> globODFslice = std::get<1>(tupleRes);
            Mat<T> slicevf_CSF   = std::get<2>(tupleRes);
            Mat<T> slicevf_GM    = std::get<3>(tupleRes);
            Mat<T> slicevf_WM    = std::get<4>(tupleRes);
            Mat<T> slicevf_GFA   = std::get<5>(tupleRes);

            LOG_INFO << "Processing Stage 2: Slice " << slice ;

....

            for (int i = 0; i < xdiff; ++i) {
                Index4DType coord;
                coord[0] = i; coord[2] = slice;
                for (int j = 0; j < ydiff; ++j) {
                    coord[1] = j;
                    for (int k = 0; k < Nd; ++k) {
                        coord[3] = k;
                        imageODF->SetPixel(coord, globODFslice(i,j,k));
                    }
                }
             }
             WriteImage<Image4DType,Image3DType, NiftiType>(ODFfilename,
                (cont.)imageODF, slice);
      }
);
```

3.4 Experimental Evaluation

In this section, we detail the experimental evaluation carried out that demonstrates the benefits of the migrated application. The experiments have been conducted using different multi-core processors and accelerators and several highly-tuned linear algebra libraries on the bottom of Armadillo and ArrayFire. In the following we describe in detail the target platform, software and configurations used during the experimentation phase.

- *Platform.* The evaluation has been carried out on a machine consisting of two multi-core Intel Xeon E5-2630 v3 processor with a total of 8 physical cores running at 2.40 GHz, hyperthreading activated, equipped with 128 GB of RAM, and executing Linux Ubuntu 14.04 x64 OS. This machine is also equipped with a NVidia Tesla K40 and a GTX 680 under CUDA version 7.5. The compilers used are GCC 5.1 and Intel 15.1. After that, the source code has been compiled using both `-O3` and `-DNDEBUG` flags.

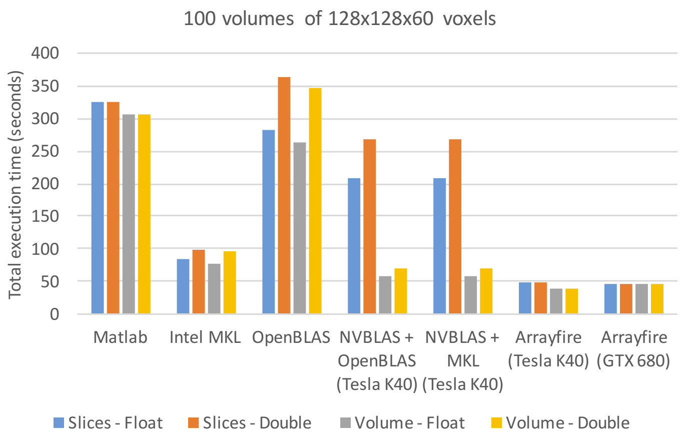
- *pHARDI configuration.* The experimental results using the pHARDI framework have been obtained using different linear algebra libraries, concretely OpenBLAS, Atlas, NVBLAS, Intel MKL, and ArrayFire. In the case of Atlas, we compiled the application using the auto-tuned optimal parameters. It is important to remark that all the experiments have been performed using single and double precision floating point numbers. To guarantee integrity of the results, we performed five consecutive executions and computed the average execution times.

- *Input data.* For each of the linear algebra libraries tested within pHARDI, we run the application using a real diffusion MRI dataset acquired from healthy subject. Specifically, whole-brain HARDI data were acquired in a 3T Philips Achieva scanner (Sant Pau Hospital, Barcelona) with a 8-channel head coil along 100 different gradient directions on the sphere in q-space with constant $b = 2000\,\text{s/mm}^2$. Additionally, $1b = 0$ volume was acquired with in-plane resolution of $2.0 \times 2.0\,\text{mm}^2$ and slice thickness of 2 mm. The acquisition was carried out without undersampling in the k-space (i.e., $R = 1$). The final dimension of this dataset is $128 \times 128 \times 60 \times 101$ voxels.

For the experimental evaluation, we have used two different data layouts for processing the input data. In both cases, the reconstruction process is obtained after 300 iterations of the RUMBA-SD algorithm.

3.4.1 Accelerators Evaluation

Fig. 3.2 (left) plots the execution time (left) and the respective speedups achieved, including I/O time (right). As can be seen, the versions that attained the highest performance are the ones using the ArrayFire implementation. Obviously, the main advantage of this version is due to the high computational capacity of the GPUs. In this case, both BLAS3 and element-wise operations are offloaded to the GPU. However, we observe that there are not remarkable differences between the two GPU models analyzed. In the future, we intend to carry out a more detailed analysis



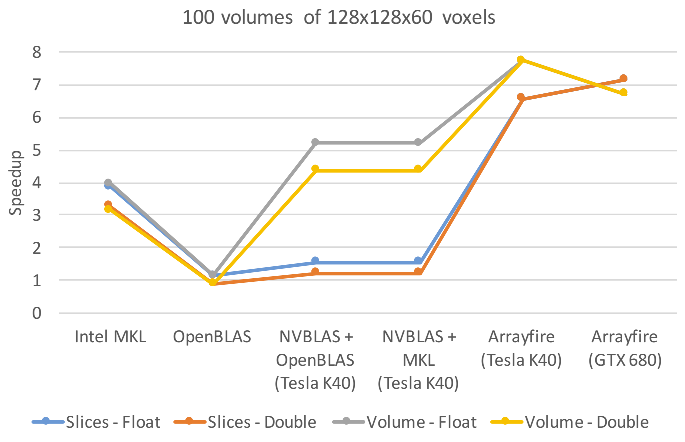


Figure 3.2: Overall execution time of pHARDI on different linear algebra accelerators.

on this aspect. An additional observation is that the version linked against the Intel MKL library delivers the best performance for the CPU-based cases. Although the MATLAB version of the applications uses underneath the same Intel MKL library for executing linear algebra kernels, our C++-ported implementation makes it more efficient thanks to the use of parallelized element-wise multiplication and optimized I/O operations via the ITK library. Finally, we do not contemplate Atlas in Fig. 3.2, mainly due to the bad performance obtained, even for tuned versions.

It is also important to highlight that NVBLAS version is only able to offload to the GPU BLAS3-related operations. This shortcoming substantially limits the room for improvement in this case. Additionally, for each offloaded operation a memory transfer from host to device arises, limiting, even more, the performance of this approach. For example, this process is not required in ArrayFire, and thanks to its API, the developer can specify which variables should be maintained in the device memory. This feature is especially important for iterative applications, needing to compute more than once a given operation over the same data.

Given the experimental results, we observe that the approach based on NVBLAS (in case of the volume pattern) is a competitive solution, where it is not needed to modify the initial Armadillo code.

Finally, Fig. 3.2 (right) reports the speedup reached by each of the linear algebra solutions, comparing with MATLAB as reference. We observe that reduce the overall execution time by $8\times$. In case of Intel MKL, the major improvement comes from the use of OpenMP, reaching an improvement of $4\times$ over MATLAB.

3.4.2 NVBLAS Evaluation

Fig. 3.3 plots a comparative analysis of NVBLAS, where the tile dimension varies from 256 to 16384 elements (threads). The tile dimension corresponds with the parameter $cublasXtSetBlockDim$ of cuBLAS. It is important to note that not all the BLAS Level-3 calls are offlloaded to the GPU. That decision is based on a simple heuristic that estimates if the BLAS call will execute for long enough to amortize the PCI transfers of the input and output data to the GPU [8].

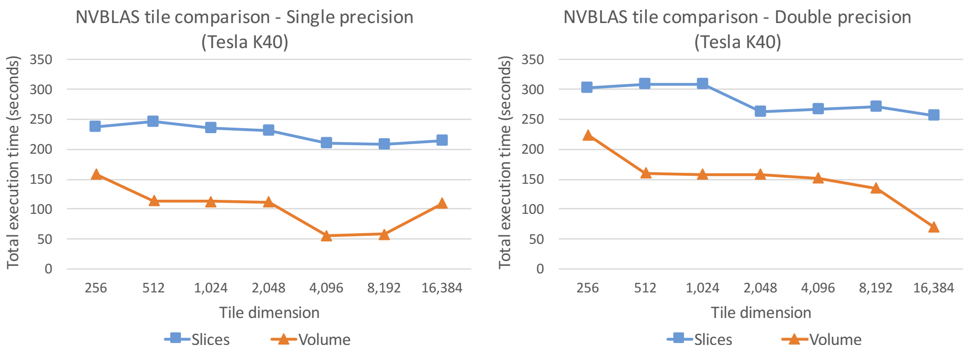


Figure 3.3: Evaluation of different tile dimensions in NVBLAS, for single and double precision.

Fig. 3.4 compares two GPUs (GTX 680 and Tesla K40) in terms of the tile dimension. We observe that there is not a significant different in terms of performance. However, the selection of an adequate tile dimension is a key factor.

3.4.3 pHARDI over GrPPI

In this subsection, we present the evaluation results of pHARDI in combination with GrPPI. The evaluation was carried out by executing the transformed source code presented in Listing 3.2. In all the evaluated cases, we have executed pHARDI with the number of cores available in the previously presented platforms. In other words, GrPPI has been configured to use as thread number, the number of the computer cores. Finally, the available threads for matrix multiplications have been reduce and limited to one to do not interfere with the GrPPI computational threads.

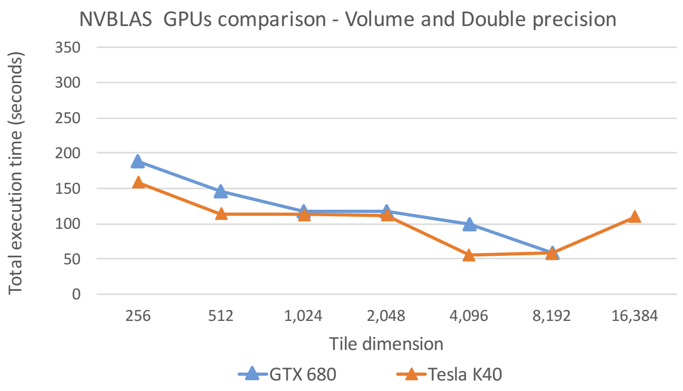


Figure 3.4: Evaluation of different GPUs, varing the tile dimension at double precision and the volume access pattern.

Table 3.1: pHARDI paralellized by using GrPPI over OpenMP and standard C++1 threads.

Implementation	Time (in seconds)	Speedup
Matlab	∼1,200.00	-
pHARDI baseline (Intel MKL)	120.20	∼10× faster than MATLAB
pHARDI GrPPI-OMP	82.82	45% faster than baseline
pHARDI GrPPI-THR	66.76	∼2× faster than baseline

The evaluation shown in Table 3.1, we have compared the overall execution time of pHARDI (included the I/O phases) over four different scenarios: MATLAB, a pHARDI relying on Intel MKL, pHARDI over GrPPI and OpenMP as acceleration support, and pHARDI under GrPPI using C++ threads. The table summarized the execution time (in seconds) and a comparative speedup with MATLAB.

In Table 3.1, we observe that the faster solution is pHARDI on top of GrPPI using C+11 threads. The results show that the version based on GrPPI is 2× faster than the baseline approach introduced in Section 3.2 and 40× faster than the MATLAB implementation. The obtained execution times are motivated by a better use of the computational resources.

3.4.4 PPAT

In Listings 3.3 and 3.4, we show the lines of codes identified by PPAT. Regarding Listing 3.3, which represents the source code related with the image initial processing, most of the identified lines are currently parallelized by using OpenMP. Additionally, Listing 3.4 demonstrates that the most expensive computational parts of the kernel of pHARDI are also identified.

Listing 3.3: pHARDI analyzed by PPAT. Algothirm manager code region.

```
MAP FOUND ON: ./fjblas-phardi-a699a3dadc57/include/
  (cont.)multi_intravox_fiber_reconstruction.hpp:139
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:141
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:143
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:174
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:176
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:284
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:419
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:420
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:430
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:432
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:452
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:759
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:770
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:787
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:788
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:798
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:809
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:819
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:829
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:839
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:841
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:861
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:875
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:1176
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:1187
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:1204
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:1205
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:1216
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:1218
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:1230
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:1244
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:1246
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:1258
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:1260
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:1273
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:1275
MAP FOUND ON: ./include/multi_intravox_fiber_reconstruction.hpp:1277
```

Listing 3.4: pHARDI analyzed by PPAT. RUMBA SD kernel.

```
MAP FOUND ON: ./include/intravox_fiber_reconst_sphdeconv_rumba_sd.hpp:108
MAP FOUND ON: ./include/intravox_fiber_reconst_sphdeconv_rumba_sd.hpp:112
MAP FOUND ON: ./include/intravox_fiber_reconst_sphdeconv_rumba_sd.hpp:113
MAP FOUND ON: ./include/intravox_fiber_reconst_sphdeconv_rumba_sd.hpp:124
MAP FOUND ON: ./include/intravox_fiber_reconst_sphdeconv_rumba_sd.hpp:131
MAP FOUND ON: ./include/intravox_fiber_reconst_sphdeconv_rumba_sd.hpp:140
MAP FOUND ON: ./include/intravox_fiber_reconst_sphdeconv_rumba_sd.hpp:147
MAP FOUND ON: ./include/intravox_fiber_reconst_sphdeconv_rumba_sd.hpp:43
MAP FOUND ON: ./include/intravox_fiber_reconst_sphdeconv_rumba_sd.hpp:44
```

3.5 Discussion

This deliverable has presented a case study where an application, initially implemented in the MATLAB language, has been ported to C++ language. The benefits are numerous, particularly due to its robustness, flexibility, and portability. Concretely, the medical use case ported is now compliant with a wide range of parallel hardware, such as multi-core processors, accelerators and co-processors, and multiple highly-tuned linear algebra libraries. We believe that this new application can tremendously aid researches in the area o studying the diffusion MRI of the human brain to get responses, as the experiments require now much less time to complete. Although the porting task can be cumbersome, we observed that using modern C++ libraries, such as Armadillo or ArrayFire, for performing linear algebra operations, greatly alleviates the burden of the developer carrying out the task.

In general, we observed that the execution time of the migrated application is $8\times$ faster, on average, than the original MATLAB version. The performed experiments demonstrate that the pair Armadillo and NVBLAS provide multiple advantages: a similar source code, automatic parallelization on BLAS level 3, and a good performance. As a remark, one of the main issues we observed for the ArrayFire library is the limited support for developing applications using multiple GPUs.

The approach introduced in this work is extended to also implement other important intravoxel reconstruction methods, including model-free techniques like q-ball imaging [12] and its extensions [1, 2, 6, 11], diffusion orientation transforms [4, 9], diffusion spectrum imaging [3, 13], as well as approaches based on parametric diffusion models and other spherical deconvolution algorithms [10] (e.g., for more details see the evaluation study by [5]). A further evaluation of these methods will be present in future deliverables.

4 Stochastic Local Search

4.1 Introduction

In this particular use case we address an optimization problem currently encountered in the slitting of metal sheets used in the production of electrical transformers. The use case has already been described in detail in D6.3. Therefore only a course overview of the important aspects will be given here.

The problem corresponds to a 2 dimensional bin-packing problem with constraints, since the objective corresponds to appropriately placing a set of metal stripes (bands) into a set of available metal coils, so that the overall metal waste is minimized. The optimization is subject to several constraints concerning the properties (e.g., weight and quality) of the produced bands, the admissibility of the slitting sequence of the available coils, as well as overall properties of the resulting assembly.

The large number of constraints (as well as their nonlinear nature) does not allow for using standard optimization algorithms for addressing this particular problem. In particular, the computation of feasible solutions (i.e., solutions that satisfy all constraints) is not straightforward, making the optimization especially hard. Thus, an optimization algorithm has been designed which is based on local stochastic search which allows to incorporate problem specific search operators, where a set of possible alterations in the allocation of bands is considered for locally searching for improvements in the cost function. This set of possible alterations is formulated in the form of processing blocks (workers), and any initially admissible allocation is going through a large sequence of such blocks.

Due to the large number of constraints, convergence to local optima is rather common while the probability of escaping from such local optima is very small (since it would require a large number of reverse alterations to be realized). Because of this, the probability of convergence to the global optimum may only increase if a) a large number of initial admissible solutions is considered which are processed independently, and b) reprocessing from earlier stages is allowed when a processing path has already converged to a local optimum. Both (a) and (b) may significantly increase the number of processing paths that are explored throughout the optimization, thus increasing the probability for converging to the global optimum. However, a large number of initial admissible solutions and reprocessing from earlier stages significantly increases the processing time. In fact, in order to maintain a reasonable overall processing time, reprocessing is only possible when it starts from admissible solutions that are close enough to the currently best solution.

Parallelization of the processing steps of the optimization algorithm allows for a) larger number of initial admissible solutions and b) reprocessing from stages further away from the currently best solution, while maintaining the processing time at the same levels as in the sequential version. Thus, through parallelization, we may increase the number of processing paths followed throughout the optimization (i.e., we may increase the probability for convergence to the global optimum) while maintaining the processing time at the same level.

4.1.1 Evaluation Metric

Besides the traditional performance metrics to evaluate the efficiency of a parallel implementation (e.g. execution time, speedup, utilization) our main algorithm specific metric is the number of processed solutions per time unit. Measuring the number of processed items is important for two reasons:

- The ratio of processed items per time unit and the number of used processing units gives information about the speedup and scalability of the parallel implementation.
- The higher the number of processed items in a single optimization run the more processing paths have been followed throughout the optimization which increases the probability of finding an optimal solution.

4.2 Parallel Pattern Implementations

Basis for the parallelization was a sequential implementation of the algorithm described in D6.3. In this initial application port we focused on the iterative optimization part to increase the number of processed items in a single optimization run, since this improves the probability to find an optimal solution. Figure 4.1 depicts the iterative core part of the optimization algorithm. Simply put, slitting plans residing in a pool are processed by a chain of optimization steps, inspected by the filter component and put back into the pool. This process is continued until a certain termination criterion is met.

In the initial sequential implementation the slitting plans are optimized one after the other. Only after one slitting plan has been finished processing, the next one can be picked from the pool. But since each slitting plan is completely independent from the others and carries all information needed for optimization the optimization steps can be applied on all slitting plans in the pool in parallel. The filter is more complicated to parallelize since it has several global states that are updated when processing a slitting plan.

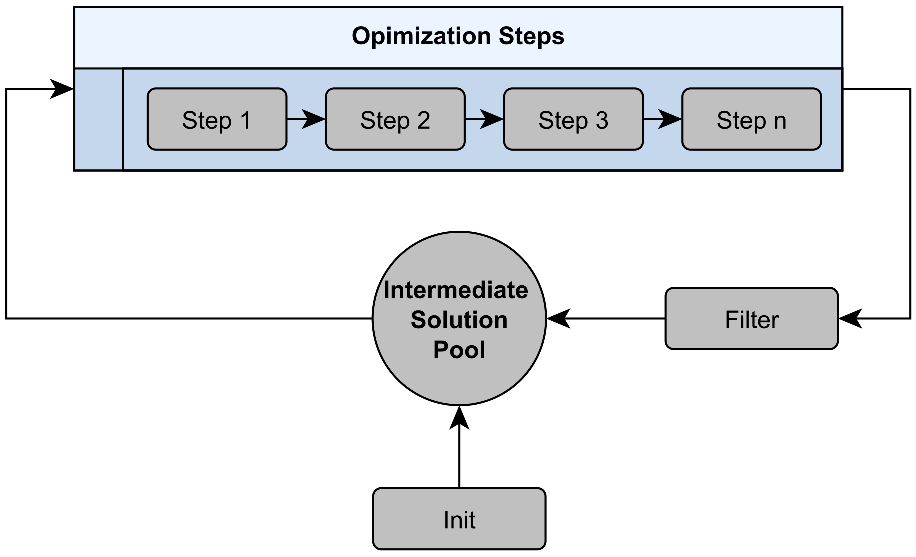


Figure 4.1: Diagram of iterative optimization.

The next two chapters describe two different parallel implementations. The first one uses high-level parallel patterns, while the second one uses core streaming patterns. In this initial port of the application the implementation has been done manually without any tool support since at the time of writing this deliverable the tools where not major enough to cope with the complex code structures and refactorings needed to introduce parallelism (semi-)automatically. We will deal with the tool application on this use case separately in chapter 4.3.

4.2.1 Synchronous Implementation

In the first implementation we applied the pool pattern, as introduced in D2.5 chapter 2.1, on this use case. Though initially designed for genetic algorithms, the semantics of this pattern perfectly fits to this kind of optimization algorithms. To apply this pattern several refactorings of the existing code had to be performed in order to map the different code parts to the four functions (selection, evolution, filter, terminate) of the pool pattern. Initially picking an item from the pool, optimizing and filtering it has been done in one main loop. This had to be split in to several functions manually. Tool support is only feasible in a very limited scale since restructuring the code requires knowledge of semantics of the algorithm. Figure 4.2 depicts the structure of the pool pattern applied to the use case.

Currently only an implementation of this patterns based on FastFlow is available. As soon as an implementation based on GrPPI is available, we will switch to this implementation in order to be able evaluate the performance using different backends.

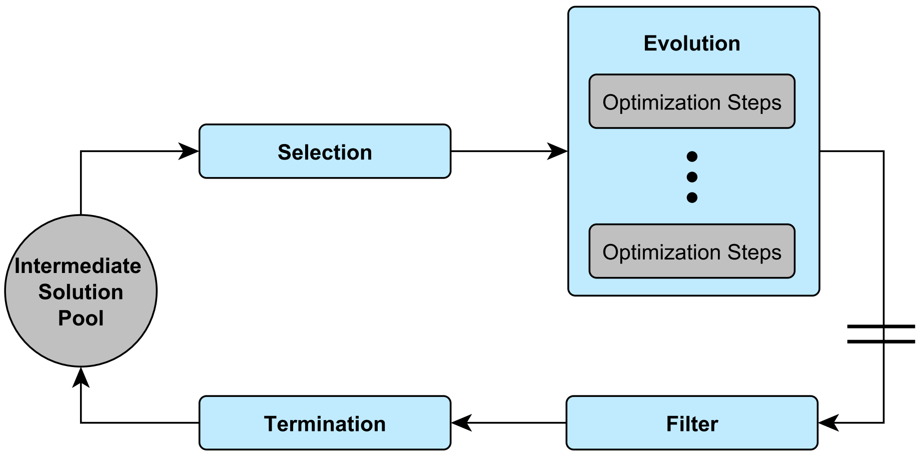


Figure 4.2: Diagram of applied Pool-Evolution Pattern.

The effort of applying this pattern is limited to mapping the existing code to the functions for selection, evolution, filter and termination.

- The *selection* function simply returns all items of the pool, since in every iteration all candidate solutions are processed.
- The *evolution* function can be directly mapped to the optimize function.
- To *filter* the solution after the evolution phase the already existing filter function can be used.
- The *terminate* function simply has to check a boolean flag that has been set during the filter process.

The following code snippet illustrates the instantiation and use of the pool evolution pattern:

```
poolEvolution<shared_ptr< Workpiece<T> >, Env_t<T>>
    pool(_numWorker, _pool,
         selection, evolution, filter, termination,
         Env_t<T>(this));

pool.run_and_wait_end();
```

The template class of the pool pattern needs two template arguments for instantiation:

- Type of the items in the pool
- Type of an optional structure that holds additional data needed during pool execution

Constructor parameters:

- `_numWorker`: the number of workers used for execution the pool pattern; equal to the parallelization degree
- `_pool`: iterable structure that holds the pool items
- `selection, evolution, filter, termination`: pointers to functions
- `Env_t<T>(this)`: needed environment data needed during optimization process

The command *pool.run_and_wait_end()* synchronously executes the pool pattern and returns once the execution terminates.

In the implementation of the pool evolution pattern only the evolution of the pool items is executed in parallel. This means for our particular case that the optimize function is applied on all selected candidate solutions simultaneously. Further loop parallelization within the optimize function is possible by simply nesting e.g. a par-for pattern. But this is only feasible on systems where the number of cores is higher than the number of solutions processed in parallel. Otherwise switching between the threads would hurt performance more than the gain from additional parallelization. Additional speedup can be achieved by parallelizing the filter function, for example by nesting another pattern like par-for or pipe. But as stated earlier, global states in the filter need to be protected from concurrent updates, which reduces the possible performance gain.

The initial implementation of the pool pattern only provides a synchronous implementation, meaning that before the filter is called all selected items must have finished the evolution function.

4.2.1.1 Preliminary Evaluation

In this evaluation we optimized the slitting plan of a transformer core with a weight of about 200 tons.

- *Platform.* The evaluation has been carried out on a 20 core Intel Xeon (two NUMA nodes 10 cores each) with 128 Gb RAM running Ubuntu 12.02.
- *Optimization Setup.* The optimization has been done using the sequential implementation as the base line. Afterwards multiple runs with increasing numbers of threads have been executed. For each number of threads the average of ten runs has been calculated. For the pool size two different values (16, 32) have been chosen. The execution time for one optimization run was always two minutes.

- *Metrics.* For each run two key metrics for our use case have been measured. The number of processed items has been measured as the metric to rate the performance of the application. The weight of the selected metal coils needed to build the transformer core indicates the quality of the optimization process.

Figure 4.3 and figure 4.3 depict the results of the evaluation. Roughly spoken, with increasing number of threads the number of items processed in a single run increases and the weight of coils needed to build to transformer core decreases. A more detailed analysis shows that the speedup is not constant for the different runs. For a small number of threads (up to 4) an almost linear speedup could be achieved. The speedup flattens if more threads are used. The trend of the speedup is illustrated in figure 4.5 and figure 4.5. Furthermore it can be seen that the performance between to consecutive runs is not always equal. There are two main reasons for that.

- *Sequential Parts.* The filter in this implementation is still sequential and poses the main bottleneck now. The more items are processed in parallel in the evolution part, the longer the queue in front of the filter gets. Therefor the speedup decreases with the number of threads.

- *Farm Queue Length.* The reason for different performance increases between two consecutive runs is the length of the item queues of the farm workers in the evolution function. The available items are assigned equally to the number of workers, e.g. using eight workers for 32 items leads to equal queues of length four for each worker. In this case all workers will finish at about the same time. Increasing to ten threads leads to eight workers with queues of length three and two workers with queues of length four. The eight workers with three items assigned will finish before the two with four items, but have to wait until the others are finished. This is illustrated in figure 4.7 and figure 4.8. It shows the correlation between the number of processed items in a single execution and the reverse of the longest queue length. A parallelisation approach based on item streaming as described in the next chapter should help to avoid this.

4.2.2 Asynchronous Implementation

The advantage of high-level patterns such as the pool evolution pattern is its abstractness, hiding all aspects of parallelization or pattern composition from the developer. But encapsulation of these aspects reduces the possibilities of the developer to control or tune the parallelization.

The current implementation of the pool pattern itself has one issue that hurts performance of our application. For every iteration a fixed number of items is selected from the pool (or in our case all items). Evolution of all these items has to

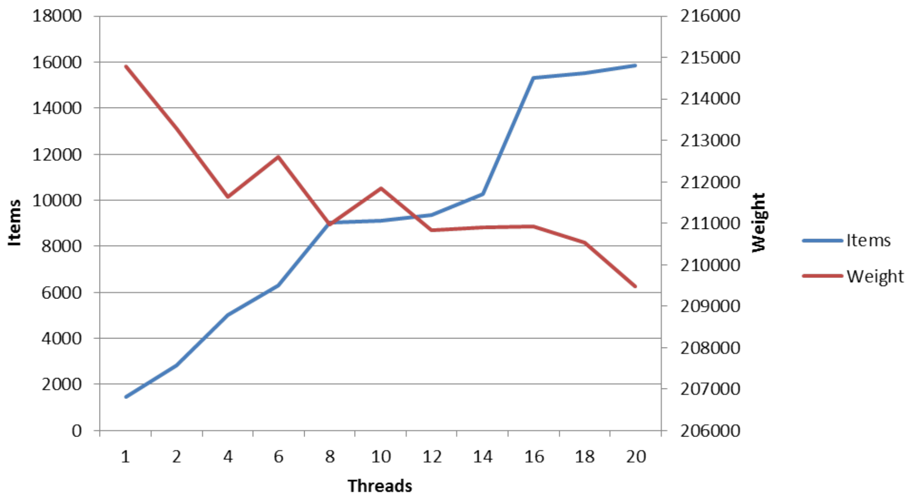


Figure 4.3: Number of processed items vs. coil weight. Pool size 16

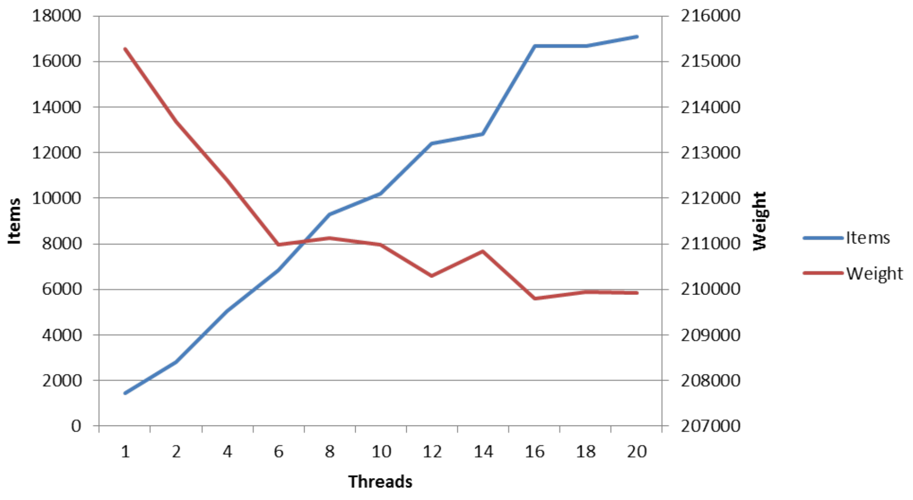


Figure 4.4: Number of processed items vs. coil weight. Pool size 32

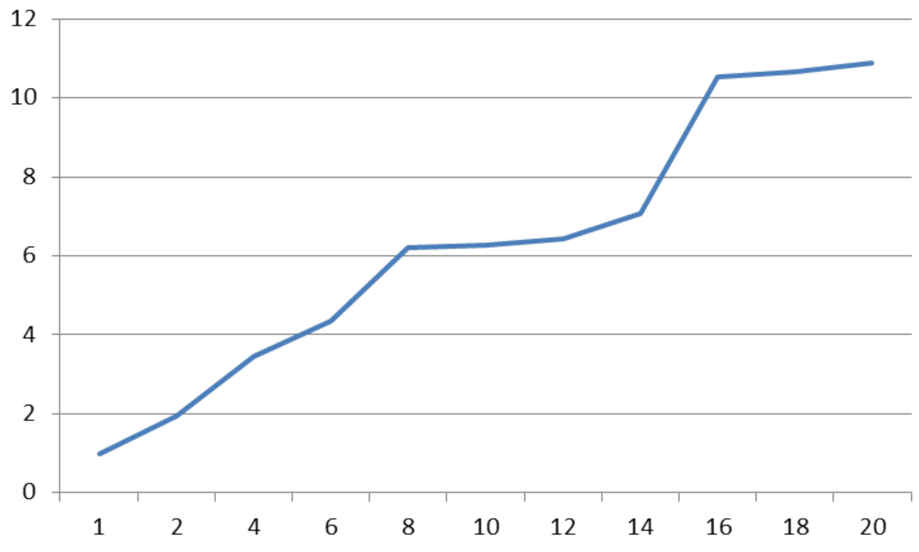


Figure 4.5: Speedup with increasing number of threads. Pool size 16

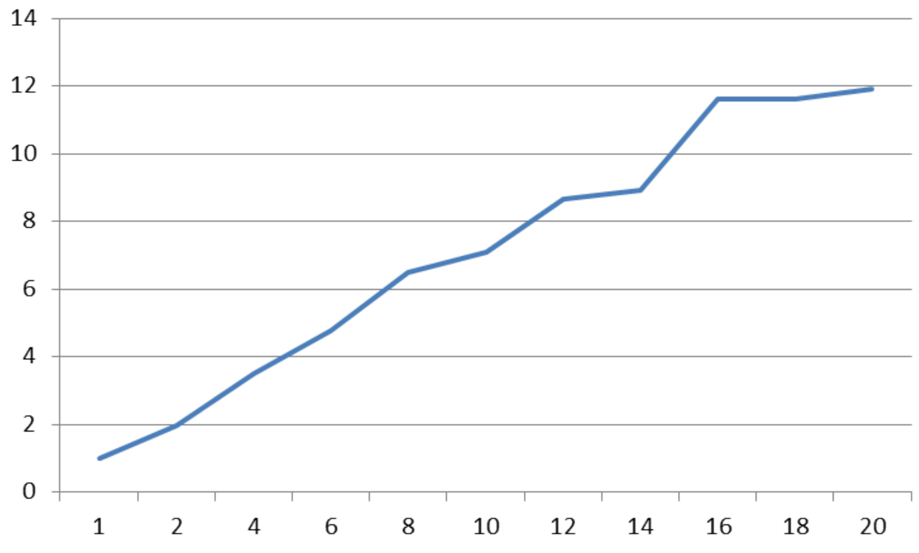


Figure 4.6: Speedup with increasing number of threads. Pool size 32

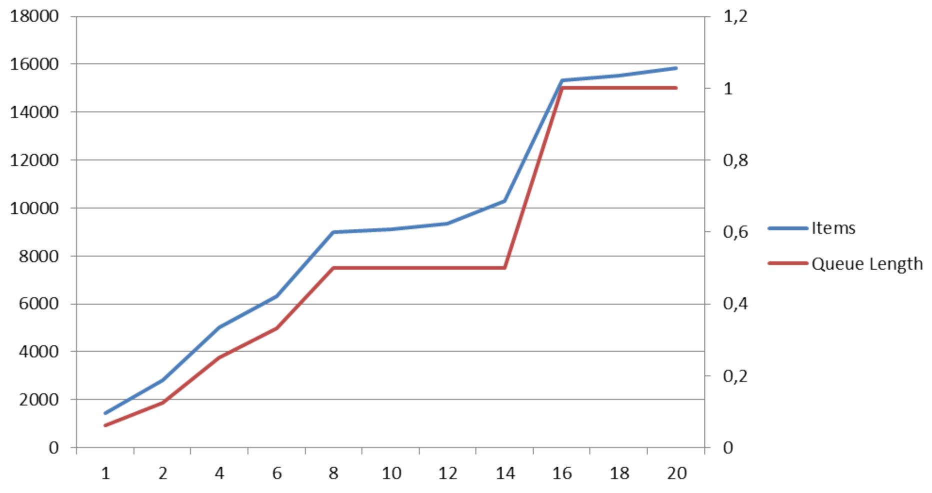


Figure 4.7: Number of processed items vs. farm worker queue length. Pool size 16

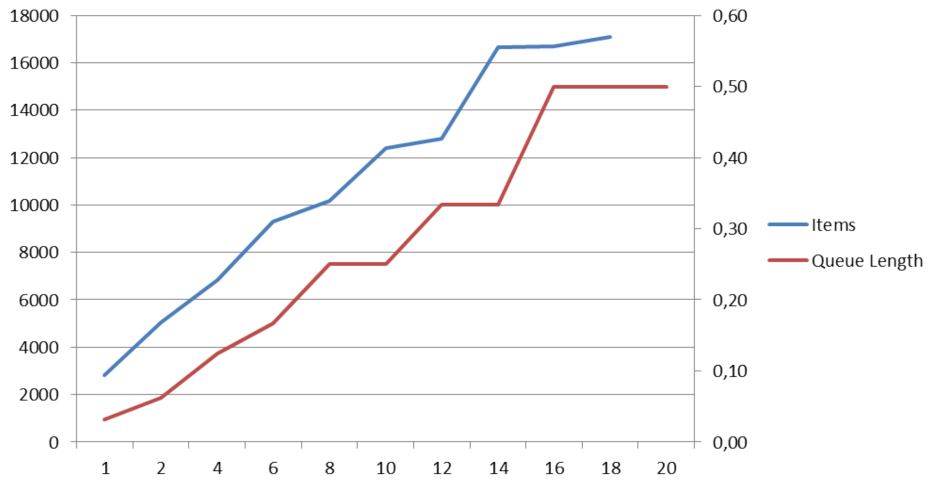


Figure 4.8: Number of processed items vs. farm worker queue length. Pool size 32.

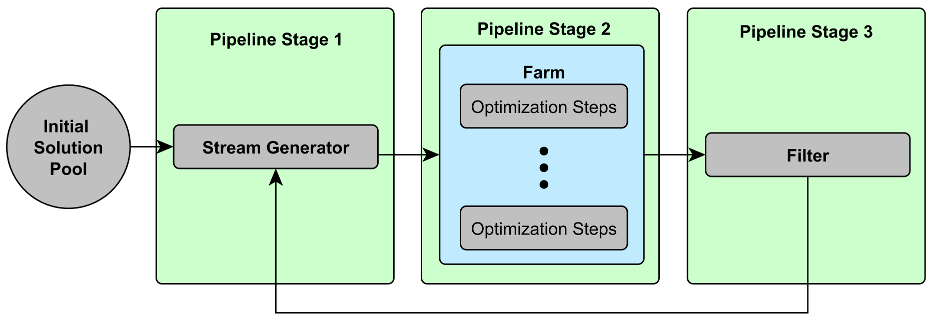


Figure 4.9: Diagram of Streaming Parallel Pattern structure.

be finished before the filter function is invoked. This introduces a synchronisation point between evolution and filter (depicted in Figure 4.2 with two parallel bars between Evolution and Filter) insuring that all items have been processed exactly for the same number of iterations. In our optimization algorithm such a synchronization is not necessary. It does not matter whether an items has been processed more often than others and filtering a particular item only relies on a global state that might be updated during the filter process but not on the current state of the other items. Due to the stochastic nature of this algorithm the processing time for each item in the evolution phase can not be predicted and may differ between single items. Therefor faster items have to wait in order to be filtered until all items have been processed by the evolution function, thus reducing the potential speedup. Apparently this additional synchronization might even hurt performance.

An enhanced implementation of the pool pattern that allows for enabling synchronization or not has been designed but not implemented yet. Therefore we decided to take a different approach for the moment.

Instead of processing all pool items in every iteration, the items can be seen as a circulating stream. Items are streamed to the optimization stage and as soon as an item is finished it is passed instantly to the filter without waiting for other items to finish. If the filter decides that an item should stay in the optimization loop it is again streamed to the optimization stages until the algorithm terminates.

This concept is implemented using the streaming patterns pipeline, farm and stream iteration pattern. Details about these patterns can be found in deliverable D2.1 and D2.4. The outermost pattern constitutes the stream iteration pattern. It initially generates the stream from the items in the solution pool and keeps the items streaming until the termination criterion is met. Nested in the stream iteration pattern is a three stage pipeline. The first stage generates the stream from the pool. The second stage consists of a farm pattern containing the optimization stages. The third stages acts as the filter. Figure 4.9 depicts how the these patterns are composed. The stream iteration pattern around the inner pattern structure should connect the third and the first pipeline stage keeping the items in the loop. Unfortunately there is no GrPPI implementation of this pattern available yet. Therefore the last and the first stage are connected using a Single-Producer-Single-Consumer-Queue and checking the termination criterion is done in the first stream generating stage. This construct will be replaced with the stream iteration patterns as soon as it is available. Since the stream iteration pattern will use quite the same concept there will be no considerable performance differences, but will reduce code complexity.

Listing 4.1: Parallelization using Streaming Parallel Patterns.

```

boost::lockfree::spsc_queue<wp_type, boost::lockfree::capacity<100> >
    (cont.)buffer;
for (int i = 0; i < pool_.size(); i++) {
  buffer.push(pool_.get_element(0));
  pool_.erase(0);
}

parallel_execution_omp p{3}, f_evo{(this->_numWorker - 4)}, f_filter{1};

std::atomic<bool> break_opt(false);
int new_pool_size = pool_.size();

//stage 0: create stream of items
//pull items from spsc queue
//check if done
auto stage_0 = [&break_opt, &buffer]()
{
  wp_type wp;
  if ( break_opt )
    return optional<wp_type>();
  while(!buffer.pop(wp))
  {
    continue;
  }
  return optional<wp_type>(wp);
};

//stage 1: farm of evolution functions
auto stage_1 = Farm(f_evo, [this](wp_type wp)
{
  pthread_t tid = pthread_self();
  int threadid = omp_get_thread_num();
  wp->get_opti_object().setThreadId(threadid);
  wp->set_error(false);
  bool error;

  // Going through the working units (i.e., all the constraints)
  for (int i = 0; i < this->id_vec_.size(); i++)
  {
    error = this->working_units_copies_[threadid][this->id_vec_[i]]->run(*
        (cont.)wp);
    if (error)
    {
      wp->set_error(true);
      break;
    }
  }
  return wp;
});

```

```
//stage 2: filter solution; put them back to buffer
auto stage_2 = [this, &break_opt, &buffer, &new_pool_size](wp_type wp)
{

  ...

  for(wp_type& item : output)
  {
    while (!buffer.push(item))
      continue;
  }
};


Pipeline(p, stage_0, stage_1, stage_2);
```

4.2.2.1 Preliminary Evaluation

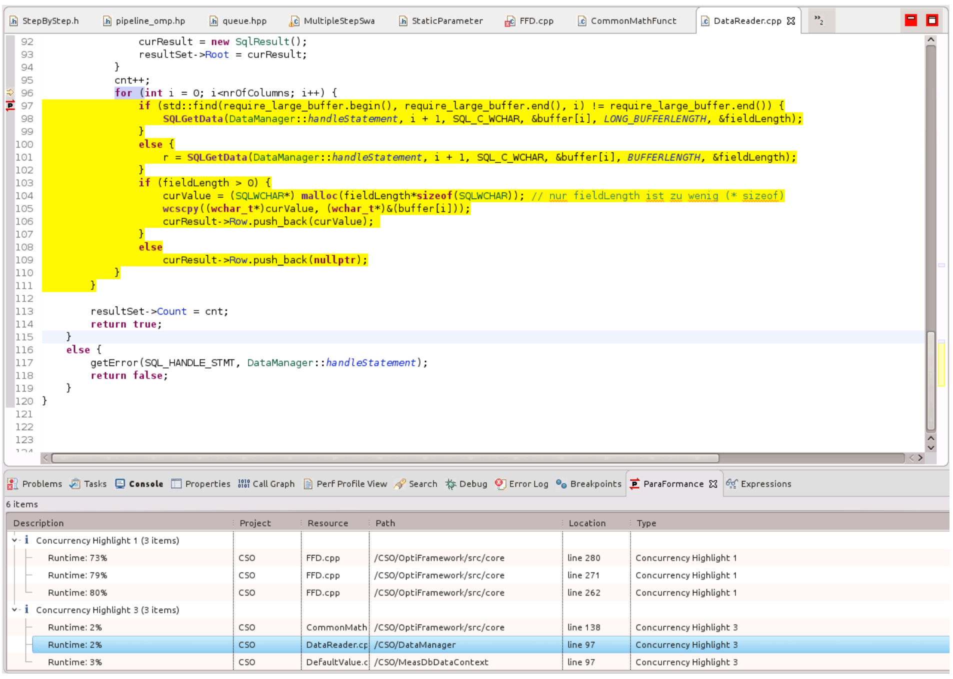
Proper evaluation could not be done using the current implementation of the used patterns. The main reason is that the nested farm pattern uses a spin waiting approach for polling for items in the queues that are used to connect the stages of the outer pipeline. These causes the application to freeze in almost all runs because all available threads are busy polling for items and the thread that should fill the queue that connects the last and the first pipeline stage does not get execution time. Furthermore the current farm implementation ensure that the items leave the farm in the same order as they entered it. This ordering costs a lot of execution time and is not needed for this use case. Since the pool pattern does not keep the order of the items a fair comparison of these two implementations could not be done.

4.3 Tools Application

So far we evaluated the Refactoring Tool and the Parallel Pattern Analyzer Tool (PPAT) on this use case. At the moment both tools are not sophisticated enough to deal with the code of the use case well. More details of the evaluation will be given in the next two chapters. Furthermore we started to evaluate the Verification Tool, but two make it work some changes have to be done in the use case code. The tool relies on the fact that the application produces the same results for two runs if the same input arguments are provided. At the moment the stochastic implementation of our use cases does not produce the same result for two runs. We have started to change this in order to be able to use this tool, and as an additional benefit make the application more testable.

4.3.1 Refactoring Tool

The refactoring tool could only be applied partially successful on the use case. The tool was able to find possibilities where to introduce additional parallel constructs (Figure 4.10, 4.11, 4.12 and 4.13. Unfortunately the tool was not able to identify the main loop that has been parallelized manually as a possibility for parallelism.



```
                curResult = new SqlResult();
                resultSet->Root = curResult;
            }
            cnt++;
            for (int i = 0; i<nrOfColumns; i++) {
                if (std::find(require_large_buffer.begin(), require_large_buffer.end(), i) != require_large_buffer.end()) {
                    SQLGetData(DataManager::handleStatement, i + 1, SQL_C_WCHAR, &buffer[i], LONG_BUFFERLENGTH, &fieldLength);
                }
                else {
                    r = SQLGetData(DataManager::handleStatement, i + 1, SQL_C_WCHAR, &buffer[i], BUFFERLENGTH, &fieldLength);
                }
                if (fieldLength > 0) {
                    curValue = (SQLWCHAR*) malloc(fieldLength*sizeof(SQLWCHAR)); // nur fieldLength ist zu wenig (* sizeof)
                    wcscpy((wchar_t*)curValue, (wchar_t*)&(buffer[i]));
                    curResult->Row.push_back(curValue);
                }
                else
                    curResult->Row.push_back(nullptr);
            }
        }

        resultSet->Count = cnt;
        return true;
    }
    else {
        getError(SQL_HANDLE_STMT, DataManager::handleStatement);
        return false;
    }
}
```

Description	Project	Resource	Path	Location	Type
Concurrency Highlight 1 (3 items)					
Runtime: 73%	CSO	FFD.cpp	/CSO/OptiFramework/src/core	line 280	Concurrency Highlight 1
Runtime: 79%	CSO	FFD.cpp	/CSO/OptiFramework/src/core	line 271	Concurrency Highlight 1
Runtime: 80%	CSO	FFD.cpp	/CSO/OptiFramework/src/core	line 262	Concurrency Highlight 1
Concurrency Highlight 3 (3 items)					
Runtime: 2%	CSO	CommonMath	/CSO/OptiFramework/src/core	line 138	Concurrency Highlight 3
Runtime: 2%	CSO	DataReader.cp	/CSO/DataManager	line 97	Concurrency Highlight 3
Runtime: 3%	CSO	DefaultValue.c	/CSO/MeasDbDataContext	line 97	Concurrency Highlight 3

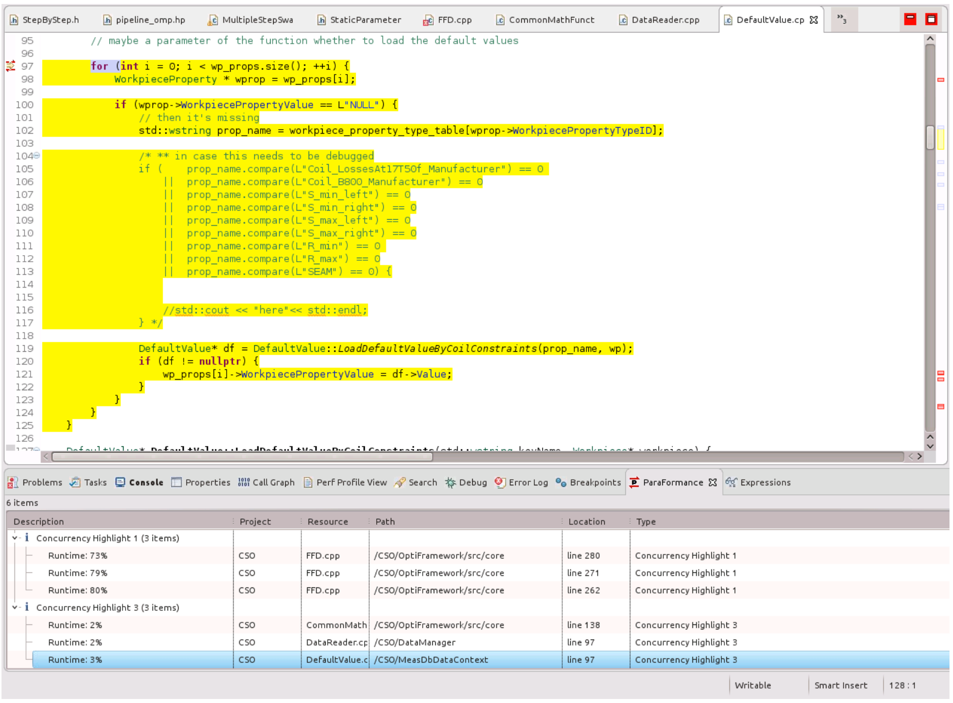
Figure 4.10: Parallelism found in DataReader.

The simple reason is that this loop is a while-loop and the tool is not yet able to cope with that. Apart from one location all the possibilities reside in parts of the code that are only executed once at the beginning of the application. Since it is the optimization part where the majority of the execution time is spent, the performance gain would be negligible. Further improvements of the tool will drastically increase its usability.

4.3.2 Parallel Pattern Analyzer Tool

The Pattern Discovery Tool, as defined in D2.3, identifies instances of parallel pattern candidates within C++ applications . The tool defines a set of program shaping techniques and methods in order to refactor sequential C++ programs into hygienic C++ code with equivalent functionality by eliminating non-hygienic code properties, such as side-effects and unnecessary task/data dependencies. It also describes the sets of requirements and properties for the initial pattern set that needs to be satisfied in C++ codes in order for a particular pattern to be introduced.

The Parallel Pattern Analyzer Tool (PPAT) introduced in D2.3, Chapter 4, is a prototype tool for detecting parallel patterns of the RePhrase project in sequential applications. The tool is completely independent of the re-factoring tool used, since it identifies parallel patterns, it performs a static analysis and avoids the use of profiling techniques, thus becomes much faster than other approaches; and it guarantees that parallel patterns detected comply with a series of requirements that

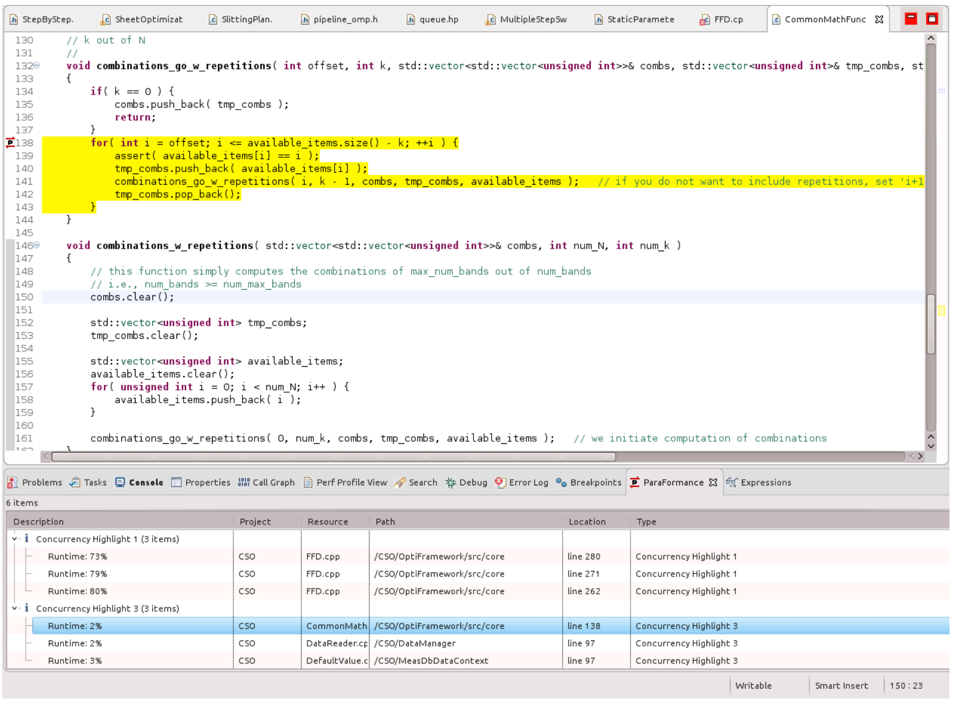


```cpp
        // maybe a parameter of the function whether to load the default values

        for (int i = 0; i < wp_props.size(); ++i) {
            WorkpieceProperty * wprop = wp_props[i];

            if (wprop->WorkpiecePropertyValue == L"NULL") {
                // then it's missing
                std::wstring prop_name = workpiece_property_type_table[wprop->WorkpiecePropertyTypeID];

                /* ** in case this needs to be debugged
                if (    prop_name.compare(L"Coil_LossesAt17T50f_Manufacturer") == 0
                    ||  prop_name.compare(L"Coil_B800_Manufacturer") == 0
                    ||  prop_name.compare(L"S_min_left") == 0
                    ||  prop_name.compare(L"S_min_right") == 0
                    ||  prop_name.compare(L"S_max_left") == 0
                    ||  prop_name.compare(L"S_max_right") == 0
                    ||  prop_name.compare(L"R_min") == 0
                    ||  prop_name.compare(L"R_max") == 0
                    ||  prop_name.compare(L"SEAM") == 0) {


                    //std::cout << "here"<< std::endl;
                } */

                DefaultValue* df = DefaultValue::LoadDefaultValueByCoilConstraints(prop_name, wp);
                if (df != nullptr) {
                    wp_props[i]->WorkpiecePropertyValue = df->Value;
                }
            }
        }
    }
```

6 items

Description	Project	Resource	Path	Location	Type
Concurrency Highlight 1 (3 items)					
Runtime: 73%	CSO	FFD.cpp	/CSO/OptiFramework/src/core	line 280	Concurrency Highlight 1
Runtime: 79%	CSO	FFD.cpp	/CSO/OptiFramework/src/core	line 271	Concurrency Highlight 1
Runtime: 80%	CSO	FFD.cpp	/CSO/OptiFramework/src/core	line 262	Concurrency Highlight 1
Concurrency Highlight 3 (3 items)					
Runtime: 2%	CSO	CommonMath	/CSO/OptiFramework/src/core	line 138	Concurrency Highlight 3
Runtime: 2%	CSO	DataReader.cp	/CSO/DataManager	line 97	Concurrency Highlight 3
Runtime: 3%	CSO	DefaultValue.c	/CSO/MeasDbDataContext	line 97	Concurrency Highlight 3

Figure 4.11: Parallelism found in DefaultValue.

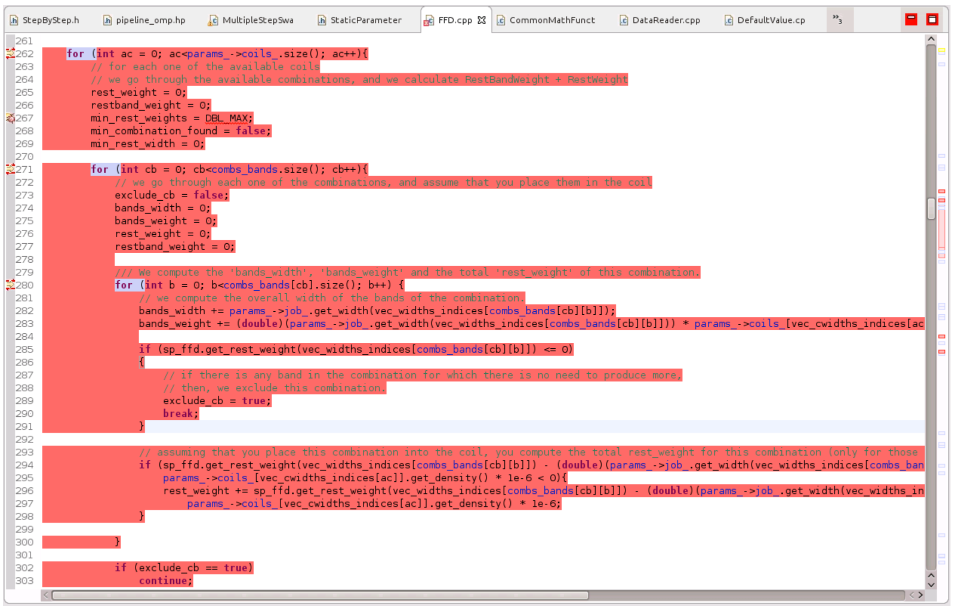


```cpp
    // k out of N
    //
    void combinations_go_w_repetitions( int offset, int k, std::vector<std::vector<unsigned int>>& combs, std::vector<unsigned int>& tmp_combs, st
    {
        if( k == 0 ) {
            combs.push_back( tmp_combs );
            return;
        }
        for( int i = offset; i <= available_items.size() - k; ++i ) {
            assert( available_items[i] == i );
            tmp_combs.push_back( available_items[i] );
            combinations_go_w_repetitions( i, k - 1, combs, tmp_combs, available_items );   // if you do not want to include repetitions, set 'i+1
            tmp_combs.pop_back();
        }
    }

    void combinations_w_repetitions( std::vector<std::vector<unsigned int>>& combs, int num_N, int num_k )
    {
        // this function simply computes the combinations of max_num_bands out of num_bands
        // i.e., num_bands >= num_max_bands
        combs.clear();

        std::vector<unsigned int> tmp_combs;
        tmp_combs.clear();

        std::vector<unsigned int> available_items;
        available_items.clear();
        for( unsigned int i = 0; i < num_N; i++ ) {
            available_items.push_back( i );
        }

        combinations_go_w_repetitions( 0, num_k, combs, tmp_combs, available_items );   // we initiate computation of combinations
```

6 items

Description	Project	Resource	Path	Location	Type
Concurrency Highlight 1 (3 items)					
Runtime: 73%	CSO	FFD.cpp	/CSO/OptiFramework/src/core	line 280	Concurrency Highlight 1
Runtime: 79%	CSO	FFD.cpp	/CSO/OptiFramework/src/core	line 271	Concurrency Highlight 1
Runtime: 80%	CSO	FFD.cpp	/CSO/OptiFramework/src/core	line 262	Concurrency Highlight 1
Concurrency Highlight 3 (3 items)					
Runtime: 2%	CSO	CommonMath	/CSO/OptiFramework/src/core	line 138	Concurrency Highlight 3
Runtime: 2%	CSO	DataReader.cp	/CSO/DataManager	line 97	Concurrency Highlight 3
Runtime: 3%	CSO	DefaultValue.c	/CSO/MeasDbDataContext	line 97	Concurrency Highlight 3

Figure 4.12: Parallelism found in CommonMathFunctions.



```
    for (int ac = 0; ac<params_->coils_.size(); ac++){
        // for each one of the available coils
        // we go through the available combinations, and we calculate RestBandWeight + RestWeight
        rest_weight = 0;
        restband_weight = 0;
        min_rest_weights = DBL_MAX;
        min_combination_found = false;
        min_rest_width = 0;

        for (int cb = 0; cb<combs_bands.size(); cb++){
            // we go through each one of the combinations, and assume that you place them in the coil
            exclude_cb = false;
            bands_width = 0;
            bands_weight = 0;
            rest_weight = 0;
            restband_weight = 0;

            /// We compute the 'bands_width', 'bands_weight' and the total 'rest_weight' of this combination.
            for (int b = 0; b<combs_bands[cb].size(); b++) {
                // we compute the overall width of the bands of the combination.
                bands_width += params_->job_.get_width(vec_widths_indices[combs_bands[cb][b]]);
                bands_weight += (double)(params_->job_.get_width(vec_widths_indices[combs_bands[cb][b]])) * params_->coils_[vec_cwidths_indices[ac

                if (sp_ffd.get_rest_weight(vec_widths_indices[combs_bands[cb][b]]) <= 0)
                {
                    // if there is any band in the combination for which there is no need to produce more,
                    // then, we exclude this combination.
                    exclude_cb = true;
                    break;
                }

                // assuming that you place this combination into the coil, you compute the total rest_weight for this combination (only for those
                if (sp_ffd.get_rest_weight(vec_widths_indices[combs_bands[cb][b]]) - (double)(params_->job_.get_width(vec_widths_indices[combs_ban
                    params_->coils_[vec_cwidths_indices[ac]].get_density() * 1e-6 < 0){
                    rest_weight += sp_ffd.get_rest_weight(vec_widths_indices[combs_bands[cb][b]]) - (double)(params_->job_.get_width(vec_widths_in
                        params_->coils_[vec_cwidths_indices[ac]].get_density() * 1e-6;
                }

            }

            if (exclude_cb == true)
                continue;
```

Figure 4.13: Parallelism found in FirstFitDecrease.

prove the correctness of the solution.

4.3.2.1 Evaluation in CSO Use-Case

In this subsection we present an experimental evaluation of PPAT on the CSO use case. The objective is to test PPAT on the CSO source code in order to analyze any potential loop that can be re-factored into parallel patterns. This evaluation have been performed with the following software:

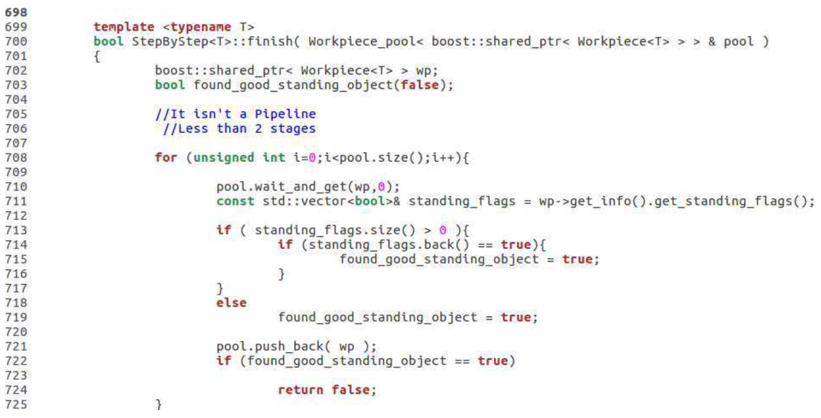
> The CSO code has been compiled with CMAKE V2.8.12.1, Boost C++ V1.55.

> The compilation of PPAT has been performed using the LLVM compiler infrastructure v3.7.0, with its Clang compiler and the extended attributes from REPARA.

The results of the evaluation methodology for the analyzed component of CSO source code is depicted in Table 4.1. The CSO code is basically divided in two major parts, the Core and OptiFramework. For simplicity, the most significant results are presented in here where only StepByStep.h is part of OptiFramework while the rest are part of the Core. We describe in the table the number of code lines, total number of loops in the source code, number of loops reported by PPAT, number of parallel patterns (pipelines, farm, map, reduce) detected during the evaluation. Every loop marked as PPAT(loops) in Table 4.1 is a loop reported by PPAT as having

Table 4.1: Statistics of OptiFramework, part of CSO source code

Test	Lines	Loops	PPAT (loops)	PPAT (patterns)
StepByStep.h	717	5	2	1
TestMethods.cpp	162	3	3	0
Remove.cpp	137	3	3	0
CommonMathFunctions.cpp	201	11	6	5



```
template <typename T>
bool StepByStep<T>::finish( Workpiece_pool< boost::shared_ptr< Workpiece<T> > > & pool )
{
        boost::shared_ptr< Workpiece<T> > wp;
        bool found_good_standing_object(false);

        //It isn't a Pipeline
         //Less than 2 stages

        for (unsigned int i=0;i<pool.size();i++){

                pool.wait_and_get(wp,0);
                const std::vector<bool>& standing_flags = wp->get_info().get_standing_flags();

                if ( standing_flags.size() > 0 ){
                        if (standing_flags.back() == true){
                                found_good_standing_object = true;
                        }
                }
                else
                        found_good_standing_object = true;

                pool.push_back( wp );
                if (found_good_standing_object == true)

                        return false;
        }
```

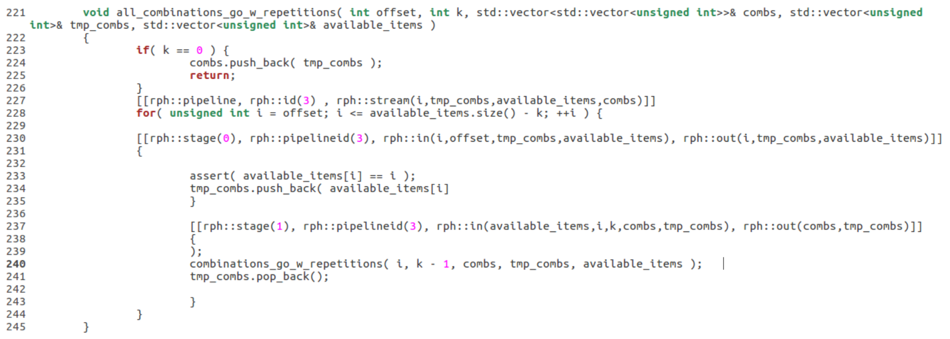
Figure 4.14: Annotated loop in StepByStep.h with a return clause

not enough stages to compute (*less than 2 stages*). Meaning that PPAT was not able to detect a pattern due to potential missing requirements in the loop, as described in D2.3, Chapter 3.

PPAT run successfully on the source code without errors. The evaluation reported different parallel situations described below. Out of the patterns reported in the evaluation, maps and pipelines were within the majority although, there was a farm reported but none reduce.

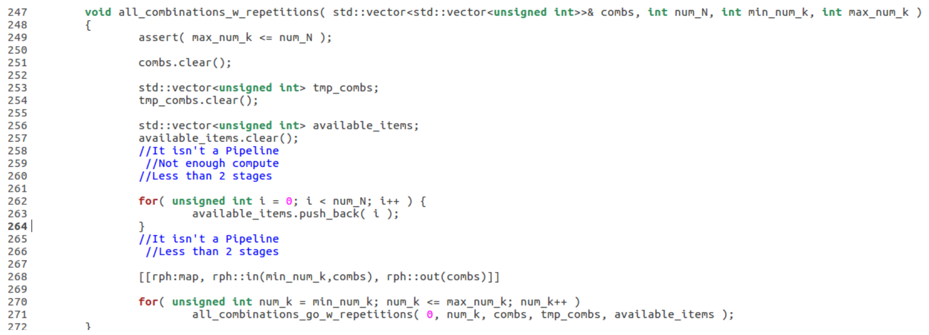
Part of the code of StepByStep.h has already been implemented with parallel patterns. While there exists three loops on the non-parallel implementation side of the code, one was reported as a two-stages pipeline, this is an expected result. As for the other two loops, they were annotated as having a *return* and a *break* statement respectively. Fig. 4.14 shows one of them. The *no break statements* within the code is one of the conditions that need to be met for the code to be evaluated as a potential parallel pattern (D2.3, Chapter 3).

The 3 loops in TestMethods.cpp and Remove.cpp have been annotated as *empty loops* with the use of *return* statements or *writing on a global variable*. Therefore, no patterns were recognized on them. The situation is similar in the two cases where, there are nested loops with an inner *continue* or *return* statement that might be propagated on the loops.



```cpp
        void all_combinations_go_w_repetitions( int offset, int k, std::vector<std::vector<unsigned int>>& combs, std::vector<unsigned
int>& tmp_combs, std::vector<unsigned int>& available_items )
        {
                if( k == 0 ) {
                        combs.push_back( tmp_combs );
                        return;
                }
                [[rph::pipeline, rph::id(3) , rph::stream(i,tmp_combs,available_items,combs)]]
                for( unsigned int i = offset; i <= available_items.size() - k; ++i ) {

                [[rph::stage(0), rph::pipelineid(3), rph::in(i,offset,tmp_combs,available_items), rph::out(i,tmp_combs,available_items)]]
                {

                        assert( available_items[i] == i );
                        tmp_combs.push_back( available_items[i]
                        }

                        [[rph::stage(1), rph::pipelineid(3), rph::in(available_items,i,k,combs,tmp_combs), rph::out(combs,tmp_combs)]]
                        {
                        );
                        combinations_go_w_repetitions( i, k - 1, combs, tmp_combs, available_items );
                        tmp_combs.pop_back();

                        }
                }
        }
```

Figure 4.15: Annotated pipeline in CommonMathFunctions.cpp



```cpp
247         void all_combinations_w_repetitions( std::vector<std::vector<unsigned int>>& combs, int num_N, int min_num_k, int max_num_k )
248         {
249                 assert( max_num_k <= num_N );
250
251                 combs.clear();
252
253                 std::vector<unsigned int> tmp_combs;
254                 tmp_combs.clear();
255
256                 std::vector<unsigned int> available_items;
257                 available_items.clear();
258                 //It isn't a Pipeline
259                  //Not enough compute
260                 //Less than 2 stages
261
262                 for( unsigned int i = 0; i < num_N; i++ ) {
263                         available_items.push_back( i );
264                 }
265                 //It isn't a Pipeline
266                  //Less than 2 stages
267
268                 [[rph:map, rph::in(min_num_k,combs), rph::out(combs)]]
269
270                 for( unsigned int num_k = min_num_k; num_k <= max_num_k; num_k++ )
271                         all_combinations_go_w_repetitions( 0, num_k, combs, tmp_combs, available_items );
272         }
```

Figure 4.16: Annotated map and Not-enough-compute in CommonMathFunctions.cpp

CommonMathFunctions.cpp is composed of a number of functions that contain loops, that caused PPAT to report a high number of pattern found, four pipelines and one map. Figure 4.15 shows an annotated pipeline found in CommonMathFunctions.cpp during the evaluation.

There are also loops where the computation is quite simplistic to be implemented with parallel patterns as it is the case of the loop depicted in Fig. 4.16 line 262, where there is an hygienic code although, there are not enough elements to report a pattern. While the loop in line 268 has been evaluated as a map.

The source code, specially the Core part of it, contains a high number of annotated loops where PPAT is not able to report as potential parallel patterns. Nevertheless, PPAT found relevant pipelines, maps and farms in the CSO source code and made annotations on potential implementation of patterns. Although, it seems to be reasonable to manually modify the Core source code in order to bypass the high number of annotations such that PPAT can detect them as parallel patterns in further evaluations.

5 Conclusion

In this deliverable the initial porting process of the use cases using RePhrase technologies and the initially performance evaluations results has been reported. Concluding we can say that we were able to successfully apply RePhrase technologies to parallelize existing code using parallel patterns. The evaluation results proved that this approve allows a reasonable performance increase comparable to handcrafted implementations while reducing complexity, development time and need for expert knowledge.

Bibliography

[1] Iman Aganj, Christophe Lenglet, Guillermo Sapiro, Essa Yacoub, Kamil Ugurbil, and Noam Harel. Reconstruction of the orientation distribution function in single- and multiple-shell q-ball imaging within constant solid angle. *Magnetic Resonance in Medicine*, 64:554–566, 2010.

[2] E J Canales-Rodríguez, L Melie-García, and Yasser Iturria-Medina. Mathematical description of q-space in spherical coordinates: Exact q-ball imaging. *Magnetic Resonance in Medicine*, 61(6):1350–1367, 2009.

[3] Erick Jorge Canales-Rodríguez, Yasser Iturria-Medina, Yasser Alemán-Gómez, Lester Melie-García, E J Canales-Rodriguez, Y Aleman-Gomez, and L Melie-Garcia. Deconvolution in diffusion spectrum imaging. *NeuroImage*, 50(1):136–149, 2010.

[4] Erick Jorge Canales-Rodríguez, Ching Po Lin, Yasser Iturria-Medina, Chun Hung Yeh, Kuan Hung Cho, Lester Melie-García, E J Canales-Rodriguez, and L Melie-Garcia. Diffusion orientation transform revisited. *NeuroImage*, 49(2):1326–1339, 2010.

[5] Alessandro Daducci, Erick Jorge Canales-Rodriguez, Maxime Descoteaux, Eleftherios Garyfallidis, Yaniv Gur, Ying Chia Lin, Merry Mani, Sylvain Merlet, Michael Paquette, Alonso Ramirez-Manzanares, Marco Reisert, Paulo Reis Rodrigues, Farshid Sepehrband, Emmanuel Caruyer, Jeiran Choupan, Rachid Deriche, Mathews Jacob, Gloria Menegaz, Vesna Prckovska, Mariano Rivera, Yves Wiaux, and Jean Philippe Thiran. Quantitative comparison of reconstruction methods for intra-voxel fiber recovery from diffusion MRI. *IEEE Transactions on Medical Imaging*, 33:384–399, 2014.

[6] Descoteaux. Regularized, fast, and robust analytical Q-ball imaging. *Magnetic Resonance in Medicine*, 58:497–510, 2007.

[7] Hans J Johnson, Matthew M McCormick, and Luis Ibanez. *The ITK Software Guide Book 1: Introduction and Development Guidelines-Volume 1*. Kitware, Inc., 2015.

[8] NVidia. CUDA Toolkit documentation: NVBLAS. http://docs.nvidia.com/cuda/nvblas, 2016.

[9] Vemuri BC Blackband SJ Mareci TH Ozarslan E, Shepherd TM. Resolution of complex tissue microarchitecture using the diffusion orientation transform (DOT). *Neuroimage*, 31(3):1086–103, 2006.

[10] J. Donald Tournier, Chun Hung Yeh, Fernando Calamante, Kuan Hung Cho, Alan Connelly, and Ching Po Lin. Resolving crossing fibres using constrained spherical deconvolution: Validation using diffusion-weighted imaging phantom data. *NeuroImage*, 42:617–625, 2008.

[11] A Tristan-Vega, S Aja-Fernandez, and C F Westin. On the blurring of the Funk-Radon transform in Q-Ball imaging. *Med Image Comput Comput Assist Interv*, 12(Pt 2):415–422, 2009.

[12] David S. Tuch. Q-ball imaging. *Magnetic Resonance in Medicine*, 52:1358–1372, 2004.

[13] Tseng WY Reese TG Weisskoff RM Wedeen VJ, Hagmann P. Mapping complex tissue architecture with diffusion spectrum magnetic resonance imaging. *Magn Reson Med*, 54(6):11377–86, 2005.