Agricultural Reform: More Efficient Farming Using Advanced Parallel Refactoring Tools

Christopher Brown, Vladimir Janjic and Kevin Hammond
School of Computer Science
Scotland, UK.
Email: {cmb21,vj,kh}@st-andrews.ac.uk

Holger Schöner
Software Competence Center Hagenberg GmbH, Austria.
Email: holger.schoener@scch.at

Kamran Idrees, Colin W. Glass
HLRS
Stuttgart, Germany.
Email: {idrees,glass}@hlrs.de

***Abstract*—Modern multicore systems offer huge computing potential. Exploiting large parallel systems is still a very challenging task, however, especially as many software developers still use overly-sequential programming models. refactoring tool support that allows the programmer to introduce and tune parallelism in an easy and effective way, exploiting high-level parallel patterns such as farms and pipelines. Using our approach, we achieve speedups of up to 21 on a 24-core shared-memory system for a number of realistic use-cases.**

I. Introduction

Despite recent trends towards increasingly parallel multicore/-manycore systems, software engineering practices are lagging behind. Most current application designers and programmers are not experts in writing parallel programs. Knowing *where* and *when* to introduce parallel constructs can be a daunting and seemingly *ad-hoc* process. As a result, parallelism is often introduced using an abundance of low-level concurrency primitives, such as explicit threading mechanisms and communication, that typically do not scale well and can lead to problems of deadlock, race conditions etc. Furthermore, software engineering tasks such as porting to other parallel platforms and general code maintenance can then require huge efforts and often rely on good parallel systems expertise.

This paper exploits a structured parallel approach based on combining parallel *task farms* with parallel/sequential *pipelines* to form complex parallel programs, using a new *refactoring*-based methodology. A sophisticated refactoring tool *guides* programmers through the parallelisation process, suggesting sensible parallelisations, and automatically inserting the appropriate parallel code at each step. This significantly reduces the difficulty of parallelisation, allowing the programmer to focus on the logic of their application, rather than on complex low-level issues. It also yields code that is more intuitive and easier to maintain. Unlike fully-automatic approaches, the programmer can exploit application knowledge and experience to direct the parallelisation process, allowing transformations cannot be absolutely shown to be safe, for example. This paper applies our refactoring tool and approach to a number of realistic use-cases in C++. We show that, using our methodology, we are able to obtain good speedups on large multicore platforms, with minimal work required from the programmer. Specifically, the main research contributions of this paper are:

1) we introduce a number of novel parallel refactorings for C++ using the FastFlow [1] skeleton library;
2) we show how our refactorings can be used to tune performance using nested skeleton configurations;
3) we demonstrate the applicability of our refactorings to a number of realistic medium-scale use-cases; and,
4) we study the performance and scalability of the refactored use-cases on a 24-core multicore machine.

While our results are given in terms of C++ and FastFlow, the approach that we have taken here is completely generic, however, and can be exploited in many other language, parallelism and communication settings, such as pThreads, OpenMP, MPI, Python or Java. Provided the underlying primitives are sufficiently rich, it is not necessary to exploit a pre-packaged skeleton library or other structured parallel primitives.

A. Introducing Parallelism using Refactoring

Refactoring is the process of changing the structure of a program, while preserving its functionality [19]. Unlike automatic program compilation and optimisation, refactoring emphasises the software development cycle, principally by: i) improving the design of software; ii) making software easier to understand; and iii) encouraging code reuse. This leads to increased productivity and programmability. This *semi-automatic* approach is more general than fully automated parallelisation techniques, which typically work only for a very limited set of cases under specific conditions, and are not easily tractable. Furthermore, unlike e.g. simple loop parallelisation, refactoring is applicable to a wide range of possible parallel structures, since parallelism is introduced in a structured way through algorithmic skeletons.

Our parallel refactoring methodology is shown in Figure 1. Here, the programmer starts with a sequential application, without any introduced parallelism. The first step is to *identify* where parallelism will be introduced; this step requires the programmer to reason about the structure of the program. However, it requires no expert knowledge on parallelisation, as a relatively basic understanding of the program structure, combined with information from performance/profiling statistics, is sufficient to indicate potential parallel targets. The programmer then identifies the unit(s) of work for the parallelism by *identifying and introducing component structures*. Components are introduced automatically by the refactoring tool, under programmer guidance. The third step is to *introduce the desired skeletal structure* for the parallelism, where the refactoring tool also introduces the required code automatically. The result is a new parallel program. The programmer is free to try

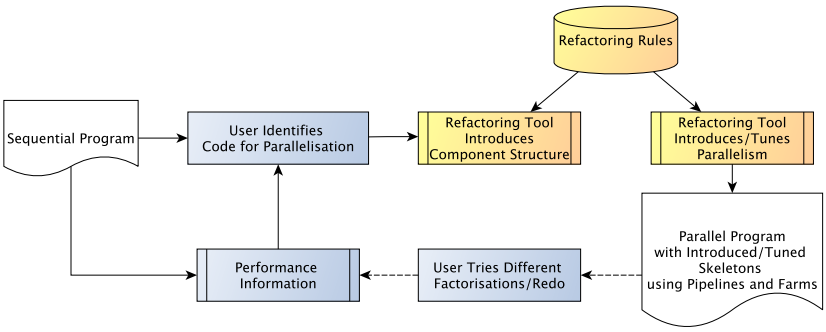

Figure 1. The Parallel Refactoring Methodology, starting from an initially sequential program, and using performance information to guide the introduction of specific parallel patterns and their associated skeleton implementations.

different parallelisations of the application, retracing previous refactoring steps if necessary. The process may be repeated, using new performance information gained from executing the refactored program, further honing the parallel performance.

B. Patterns and Skeletons

In our approach, parallelism is described in terms of high-level parallel patterns. For this paper, we restrict ourselves to two classical parallel patterns, which we consider to be among the most useful and most common:

- The *Farm* (Δ) pattern is present in computations where a single function, f, is applied to a set of inputs, $x_1, \ldots, x_m$. Parallelism arises from applying the same function to different inputs at the same time.
- The *Pipeline* ($\|$) pattern models a parallel pipeline, where a sequence of functions, $f_1, f_2, ..., f_m$ are applied, in turn, to a sequence of independent inputs, $x_1, x_2, ..., x_n$. The output of f_i becomes the input to f_{i+1}, so that parallelism arises from executing $f_i(x_k)$ in parallel with $f_{i+1}(f_i(x_{k-1}))$.

The actual implementation of these parallel patterns is expressed in terms of *algorithmic skeletons* [10], [11], abstract parallel code templates that are instantiated with application code and other key information to give concrete parallel programs. Skeletons abstract away low-level complexities such as thread creation, communication, synchronisation, and load balancing. In this paper, we use the FastFlow [1] skeleton library for C++.

II. Refactoring C++ to Introduce Parallelism

This section discusses three refactoring examples to introduce computationally intensive parallel components, task farms, and parallel pipelines, respectively.

A. Introduce Component Refactoring

This refactoring allows the programmer to identify a computationally intensive entity (a parallel "component") that can subsequently be encapsulated as part of a parallel computation, e.g. as a worker in a farm or pipeline. The refactoring introduces an instance of a Component class[1], encapsulating the selected computational entity *hygienically* (i.e. the component is free from external side-effects). Step 1 of Figure 2 shows an example of using the *Introduce Component* refactoring on the sequential convolution code from Algorithm 1. In this example, the programmer has introduced two components. The refactoring is first applied to the generate method, producing a Component declaration, genStage. The call to generate is then replaced by a call to genStage.callWorker, with the appropriate parameters. The process is repeated for the filter method, producing the filterStage component.

[1]A subclass of Fastflow's base ff_node code.

B. Introduce Farm Refactoring

The *Introduce Farm* refactoring has two variants:

- *Introduce Farm (Declaration):* This refactoring introduces a new FastFlow *farm* declaration. A Component instance is chosen to be used as the worker function for the farm. An example is shown in Step 3 of Figure 2, where the refactoring has introduced a new farm declaration, gen_farm, using the gen_stage component from Step 2. The nworkers parameter (the number of farm workers) must either be previously defined or else be given as an argument to the refactoring. A similar refactoring is applied in Step 4 to give the filter_farm declaration.
- *Instantiate Farm (Instantiation):* The second variant instantiates a previously-defined farm. Step 3 instantiates one of the pipeline stages using the newly created gen_farm. All other stages in the *Pipeline* are preserved.

C. Introduce Pipeline Refactoring

Our final refactoring similarly has two variants:

- *Introduce Pipeline (Declaration):* This refactoring inserts a new FastFlow *pipeline* declaration, analogously to introducing a farm declaration. This is shown in Step 2.
- *Instantiate Pipeline (Instantiation):* For this refactoring, the programmer must select a C++ **for** loop. Step 2 shows the result after refactoring. Here, the original **for** loop has been refactored into a three-stage *pipeline*, pipe, where the first stage is a StreamGen stage. This is a C++ class instance that models the streaming input to the pipeline. The refactoring is not limited to only C style arrays: C++ STL data structures, such as std:vector< > objects,

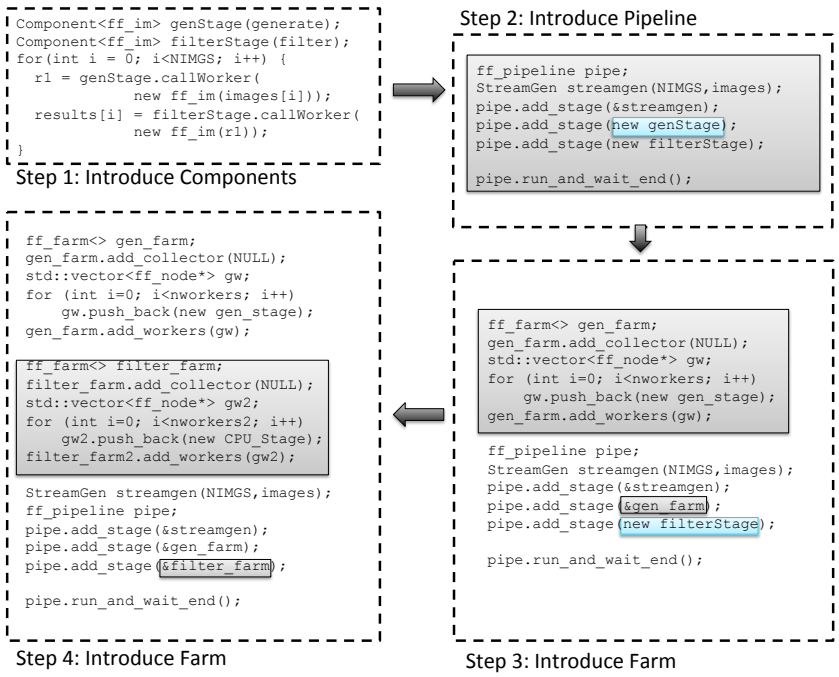

Figure 2. Refactoring the Image Convolution Algorithm (Algorithm 1), first introducing parallel components (Step 1), then introducing a parallel pipeline (Step 2), before adding two levels of farm (Steps 3 and 4).

for example, may also be considered The second stage, a function, genStage, is added as a pipeline stage; the third stage, filterStage, is added as the final stage. Finally, the pipeline waits for the result of the computation, using a run_and_wait_end() method call. Any dependencies between the output of genStage and the input of filterStage are detected automatically.

Every refactoring that introduces new code has a corresponding inverse: for example, *Farm Elimination* inverts *Introduce Farm*; and *Pipeline Elimination* inverts *Introduce Pipeline*. This allows any combination of rules to be inverted (undone). The refactorings are also fully nest-able, allowing, for example, a farm, such as $\Delta(f1 \circ f2)$ (a task farm, Δ, where the worker is a function composition, $\circ$, of two components, *f1* and *f2*), to be refactored into $\Delta(f1 \parallel f2)$ (transforming the composition into a parallel pipeline, $\parallel$). In this way farms and pipelines may be formed and reformed in any way that is necessary for the parallel application.

Our refactoring prototype[2] is implemented in Eclipse, under the CDT plugin. The programmer is presented with a menu of possible refactorings to apply. The decision to apply a refactoring and introduce a skeleton is made by the programmer. Once a decision has been made, all required code-transformations are performed automatically. We therefore rely on the programmer to make informed decisions, and can exploit any knowledge/expertise that they may have, but do not require him/her to have deep expertise with parallelism.

III. Use Cases

In this section we illustrate the use of the refactorings above on a set of medium-scale realistic benchmarks. For each use case, we start with the original sequential implementation in C++, and apply the refactorings from Section II in order to obtain an optimised parallel version. The refactoring process demonstrated here relies on programmer knowledge to know *when* and *where* to apply the refactorings, based on profiling information and knowledge about the patterns from Section I-B, and following the methodology of Section I-A. In order to properly evaluate our refactoring tool, parallelisation was performed twice for each use-case: once using the refactoring tool and once on a purely manual basis. The refactoring-based parallelisation will be discussed in detail below. In all cases, Both versions give almost identical performance. However, the development time using refactoring was much faster, giving a clear and significant advantage of nearly one order of magnitude over the manual implementation.

[2]Available at: http://www.cs.st-andrews.ac.uk/~chrisb/refactorer.zip

Algorithm 1 Sequential Convolution Before Component Introduction

```
1 for(int i = 0; i<NIMGS; i++) {
2   r1 = generate(new ff_image(images[i]));
3   results[i] = filter (new ff_image(r1));
4 }
```

A. Image Convolution

Image convolution is a technique widely used in image processing applications such as blurring, smoothing and edge detection. The sequential structure (Figure 2, Step 1), consists of two stages. The first stage, genStage, reads an image from a file, while the second stage, filterStage, applies a filter to each image. The convolution process is typically applied to a stream of images. Computationally, the filtering stage requires a scalar product of the filter weights with the input pixels

within a window surrounding each of the output pixels:

$$output\_pixel(i,j) = \sum_m \sum_n input\_pixel(i-n, j-m) \times filter\_weight(n,m) \quad (1)$$

Step 1: Introduce Components. The refactoring process proceeds in a number of steps. The first step is to identify the components required as worker stages of the skeletons to be introduced. In order to do this, the programmer first selects the generate function call on Line 2 of Algorithm 1, and uses the *Introduce Component* refactoring. The result of this is a component declaration, genStage together with a call to the method callWorker, as shown in Figure 2. The programmer repeats this step for the filter function call in Algorithm 1. The names for the components are given as a parameter to the refactoring tool before the refactoring is performed. The refactoring tool automatically checks for conflicts in scope.

Step 2: Adding a Pipeline. The second step is to add a *Pipeline* skeleton and identify that genStage and filterStage are the two stages of the pipeline. First, a new pipeline declaration, pipe, is introduced, using the *Introduce Pipeline* refactoring. The location of the new pipeline declaration is defined by the cursor position in the Eclipse IDE. In this example, the programmer chooses to introduce the new declaration just before the **for** loop. The next step is to add the two stages to the pipeline, by selecting the **for** loop, using the *Introduce Pipeline Stages* refactoring. The refactoring automatically checks the body of the **for** loop for Component instances, replacing calls to callWorker with pipe.addStage. In addition, the refactoring tool also automatically adds a preliminary streaming stage, streamgen, which acts as a stage that streams images to the genStage pipeline stage. This is determined as being the first argument to genStage before the refactoring took place. The refactored code is shown in Figure 2, Step 2.

Step 3: Farming the Pipeline Stages. The next step in the refactoring process is to farm the two pipeline stages. First, the programmer declares a new farm declaration, gen_farm (the name of the farm is programmer-defined) using the *Introduce Farm* refactoring. Then, the programmer selects the expression **new** genStage as the farm worker, and performs the *Instantiate Farm* refactoring. The refactoring tool then automatically adds the new worker to the gen_farm farm. The refactored code is shown in Figure 2, Step 3. Finally, the second stage of the pipeline is also farmed. Again, a new farm declaration is introduced, filter_farm, the expression **new** filterStage (second stage of the pipeline) is selected and the *Instantiate Farm* refactoring is performed. The result is shown in Figure 2, Step 4, where the filter_farm declaration adds the filterStage component as its worker. Additionally the farm is also added as the final stage of the pipeline.

B. Ant Colony Optimisation

Ant Colony Optimisation (ACO) [3] is a heuristic for solving NP-complete optimisation problems, inspired by the behaviour of ants living in real ant colonies. In each iteration of ACO alogrithm, each ant independently computes a solution to the problem, with the solution being partially guided by a *pheromone trail*. To compute one component of a solution, an ant (with probability q) follows the pheromone trail for that component or (with probability $1-q$) it performs a biased random selection of the component. In this way, different ants produce different (but similar) solutions. When all ants are done, the best solution is chosen and the pheromone trail is updated according to that solution and the next iteration starts.

Algorithm 2 Sequential Ant Colony

```
1 for (j=0; j<num_iter; j++) {
2   for (i=0; i<num_ants; i++)
3     cost[i] = solve(new ff_task_t(num_jobs, i,
4       &(results[i*num_jobs]), process_time, weight,
5       deadline, tau));
6   best_t = pick_best(num_ants, num_jobs, cost,
7                      results, &best_result);
8   update(num_jobs, best_t, best_result, tau);
9 }
```

In this paper, we consider Single Machine Total Weighted Tardiness Problem (SMTWTP) as an optimisation problem to which we apply ACO. We are given n jobs, where each job i is characterised by its processing time, p_i, deadline, d_i, and weight, w_i. The goal is to schedule execution of jobs in a way that achieves minimal total weighted *tardiness*, where the tardiness of a job is defined by $T_I = \max\{0, C_i - d_i\}$, where C_i is the completion time of the job i, and the total tardiness of the schedule is defined as $\sum w_i T_i$. In the ACO solution to the SMTWTP problem, in each iteration each ant independently computes a schedule. The pheromone trail that guides the schedules is defined by matrix τ, where $\tau[i,j]$ is the preference of assigning job j to the i-th place in the schedule.

C++ Implementation: Algorithm 2 shows the extract of the sequential code for the SMTWTP instance of the ACO class of algorithms, written in C++. The most relevant code for refactoring and parallelisation is the iteration loop, where in each iteration every ant computes a solution, the best solution is chosen and the pheromone trail is updated. The solve function produces a solution based on the arrays of processing times, deadlines and weights of jobs and the pheromone trail. The pick_best function picks the best solution and update updates the pheromone trail, tau. The potential for parallelisation lies in the fact that each ant can compute its solution completely independently. The remaining two phases of each iteration, i.e. picking the best solution and updating the pheromone trail, are relatively cheap and inherently sequential. Therefore, parallelising the ACO involves introducing a farm, where each worker computes the solution for one ant.

Step 1: Identifying Components. The first step in the refactoring process is to identify the component that will be used as the worker in the farm. In the our example from Algorithm 2, we identify the function call to solve as a worker function, and introduce a Component instance, called WorkerComponent, using the *Introduce Component* refactoring.

Step 2: Adding a Farm. The next step is to introduce a new farm declaration, using the *Introduce Farm* refactoring. This new declaration is added just before the **for** loop at Line 1, so that it can be used within the **for** loop body.

Step 3: Adding Tasks to the Farm. The final step is to add the tasks to the farm, using the *Instantiate Farm* refactoring. For this, the **for** loop is selected, enabling the refactoring to replace calls to the callWorker function with farm.offload, passing the

Algorithm 3 Ant Colony Refactored for Parallelisation

```
1  Component<ff_task_t> w(solve);
2  ff_farm<> farm(true);
3  farm.add_collector(NULL);
4  std::vector<ff_node*> workers;
5  for (int i =0; i<num_workers; i++)
6      workers.push_back(new WorkerComponent);
7  farm.add_workers(workers);
8    for (j =0; j<num_iter; j++) {
9      farm.run_then_freeze();
10     for (i =0; i<num_ants; i++) {
11       farm.offload(new ff_task_t(num_jobs, i,
12         &(results[i*num_jobs]), process_time, weight,
13         deadline,tau);
14     }
15     farm.offload((void *)FF_EOS);
16     farm.wait_freezing();
17     best_t = pick_best(num_ants, num_jobs, cost,
18                        results, &best_result);
19     update(num_jobs, best_t, best_result, tau);
20   }
```

actual callWorker parameter to offload instead. The code after refactoring is shown in Algorithm 3.

C. Molecular Dynamics

Molecular Dynamics (MD) simulates the behaviour of a system on the atomic scale [2]. Interactions between molecules are evaluated explicitly and Newton's equations of motion are numerically integrated over time, leading to an extreme spatial and temporal resolution. However, this leads to significant computational costs and hampers the adoption of MD as the method of choice in more complex real-world applications. With the rise of HPC computing, circumstances are changing. MD is used routinely in more and more cutting-edge technologies and research. This requires MD codes to be highly parallel, thus a lot of effort has been put into porting and developing new, hardware-specific codes. For research on MD from a HPC point of view, a highly modular code is being developed in pure C, *CMD*. The goal is to have a single code featuring multiple widely used data structures and all widespread parallelisation methods.

CMD features two distinct data structures. *BasicN2*, used for small to medium sized systems where all intermolecular distances are evaluated in order to identify interaction partners, and *MoleculeBlocks*. There, all molecules are sorted into cells, reducing the search for interaction partners to O(N) and therefore used for large systems (also called linked cell). In this paper we consider the BasicN2 variant for refactoring. Profiles of the BasicN2 use case show that the force calculation routine dominates the simulation time (e.g. 99.91% for 68,000 molecules). Therefore, only the force calculation routine needs to be parallelised.

Step 1: Identifying Components. Algorithm 4 shows the simplified code of the sequential version of the force calculation routine. The first step is to identify the worker components, by selecting the function call to calc_forces_real and choosing the *Identify Component* refactoring. Likewise, the programmer performs this refactoring for the calc_forces_halo function.

Step 2: Introducing a Farm Declaration. The next step in the refactoring process is to introduce a new farm declaration.

Algorithm 4 Sequential Molecular Dynamics (MD)

```
1  void basicN2_calc_forces(void *container, real *U_pot) {
2      basicN2 *task = (basicN2 *) container;
3      long i, j;
4      real U_pot_tmp = 0.;
5      for(i  = 0; i  < config.num_threads; i++)
6          calc_forces_real(task);
7      *U_pot /= (2 * 6.0);
8      for(i  = 0; i  < config.num_threads; i++)
9          calc_forces_halo(task);
10 }
```

Algorithm 5 MD Refactored for Parallelisation

```
1  void basicN2_calc_forces(void *container, real *U_pot,
2          ff_farm<>* farm1, ff_farm<>* farm2) {
3      basicN2 *task = (basicN2 *) container;
4      long i, j;
5      real U_pot_tmp = 0.;
6      for(i  = 0; i  < config.num_threads; i++)
7          farm1->offload(task);
8      farm1->offload((void *)FF_EOS);
9      farm1->wait_freezing();
10     *U_pot /= (2 * 6.0);
11     for(i  = 0; i  < config.num_threads; i++)
12         farm2->offload(task);
13     farm2->offload((void *)FF_EOS);
14     farm2->wait_freezing();
15 }
```

Component declarations from the previous step need to be *lifted* to the main function (to re-use the same farm in multiple simulation iterations). Currently, this step must be performed manually by the programmer, however, this could easily be implemented as a refactoring. Once the Component declarations are lifted to the main function, the programmer can select the first Component declaration and choose the *Introduce Farm* refactoring. The name of the introduced farm is chosen as farm1 and the number of workers N is specified. This process is repeated for the halo routines, by selecting the second worker declaration haloWorker and introducing a second farm, farm2.

Step 3: Adding Tasks to the Task Farm. The final refactoring step is to add tasks to the two new farms that were introduced in the previous step. In order to do this, the new farm declarations, farm1 and farm2 must be passed as parameters to the basicN2_calc_forces function. This is currently a manually performed step (it could be implemented as refactoring). Once the farms are added as function parameters, the programmer first identifies farm1 and selects the first **for** loop (at line 6 in Algorithm 4) and chooses the *Instantiate Farm* refactoring. This refactoring replaces the Component method call to callWorker with farm1.offload, using the parameters to callWorker as parameters to the offload function. This process is then repeated for farm2 and the second **for** loop at Line 9. Algorithm 5 shows the result of the refactoring process

D. Graphical Lasso

In Machine Learning, there is often the need to determine a dependency structure graph between features in a data set. The Graphical Lasso [13] allows us to do this, based on the data correlation matrix, but eliminating correlation between features where this is introduced indirectly because of intermediate

Algorithm 6 Sequential Graphical Lasso

```
1  typedef MatT arma::mat;
2  void t_gelnet_ff(const MatT& S, double lambda, MatT& C,
3          MatT& P, ...) {
4      arma::uword p = S.n_cols; // problem size: #variables
5      std::vector< std::vector< arma::uword > > components;
6      connected_components( S, L, components );
7      arma::uvec block;
8      arma::uword bi, j;
9      for( arma::uword c=0; c < components.size(); c++ ) {
10         block = arma::sort( arma::conv_to< arma::uvec >::
11             from( components[c] ) );
12         C( block, block ) = S( block, block );
13         for( bi=0; bi < block.n_elem; bi++ ) {
14             j  = block(bi);
15             C(j,j ) = S(j,j ) + lambda;
16         }
17         if ( block.n_elem > 1 ) {
18             t_precision2weights( P, block );
19             gelnet_inverse( S, lambda, block, C, P, ... );
20             t_weights2precision( P, C, block );
21         } else {
22             j  = block(0);
23             P(j,j ) = 1.0 / C(j,j );
24         }
25     }
26 };
```

features. Common implementations use iterative coordinate descent optimisation to perform the involved matrix inversion. If the expected graph structure is sparse, this optimisation is usually combined with a lasso regularisation, by which sparse graph structures are favoured. Additionally, for real data, the correlation matrix often factorizes into blocks where features between different blocks have no correlation above the value used for the lasso regularisation. In this case, the optimisation process on these blocks becomes independent, and inversion of such blocks can be performed independently and in parallel. We first parallelised this algorithm manually, using the FastFlow library. When the refactoring tool became available, we produced a second version. The relevant part of the sequential implementation is given in Algorithm 6 in simplified form. After determining a factorizing block structure (connected components, Line 6) of the matrix to work on, $\mathbf{S}$, a **for** loop (Line 9) iterates over all components, and prepares variables for a following call of gelnet_inverse() (Line 19), which implements the coordinate descent.

Step 1: Preparation for refactoring. The code as shown in Algorithm 6 does not yet fulfil the preconditions for component identification and farm introduction refactorings. To enable this, some local variable declarations (Lines 7–8) are moved into the loop body, the loop body is extracted as a new function (t_gelnet_ff_for_component; Eclipse CDT supports this via an existing sequential refactoring), and the arguments of the new function are extracted into a task structure (Algorithm 7, Lines 1–11; this has to be done manually as there is currently no appropriate refactoring).

Step 2: Identifying Components. Next, the *Identify Component* refactoring is applied, introducing the component wrapper t_gelnet_ff_worker as described in Section II-A.

Step 3: Introducing a Farm Declaration. In the next step, refactoring is used to introduce the farm declaration before

Algorithm 7 Graphical Lasso Refactored for Parallelisation

```
1  struct t_gelnet_ff_task {
2      t_gelnet_ff_task(const MatT& _S, double _lambda,
3          MatT& _C, MatT& _P, std::vector<arma::uword>
4          _component, ...): S(_S), lambda(_lambda),
5          C(_C), P(_P), component(_component), ... {};
6      const MatT& S;
7      double L;
8      MatT& C;
9      MatT& P;
10     std::vector<arma::uword> component;
11 };
12 void t_gelnet_ff(const MatT& S, double lambda, MatT& C,
13         MatT& P, ...) {
14     arma::uword p = S.n_cols; // problem size: #variables
15     std::vector<std::vector<arma::uword>> components;
16     connected_components( S, L, components );
17     ff::ff_farm<> farm(true); // set accelerator mode
18     farm.add_collector(NULL);
19     std::vector<ff::ff_node*> workers;
20     for (int i=0; i<nworkers; ++i)
21         workers.push_back(new t_gelnet_ff_worker);
22     farm.add_workers(workers);
23     farm.run_then_freeze();
24     for( arma::uword c=0; c < components.size(); c++ ) {
25         farm.offload(new t_gelnet_ff_task(S, lambda,
26                 &C, &P, components[c], ...));
27     }
28     farm.offload((void *)FF_EOS);
29     farm.wait_freezing();
30 };
```

the **for** loop (Algorithm 7, Lines 17-23), here with nworkers workers specified. In this step, the farm is also filled with the workers of the component t_gelnet_ff_worker introduced during the preceding step.

Step 4: Adding Tasks to the Task Farm. Finally, the whole **for**-loop is marked, and refactoring is used to replace the component call in the **for** loop by farm offloading of the tasks (Algorithm 7, Lines 24–26). The final refactored code is given in Algorithm 7, excluding the function t_gelnet_ff_for_component introduced in the step "Preparation for refactoring" (which just contains the former loop body including the moved local variable declarations), and the standard component wrapper introduced by refactoring, t_gelnet_ff_worker.

IV. Benchmark Results

All measurements have been made on a dual 2.3GHz 12-core AMD Opteron 6176 architecture, running Centos Linux 2.6.18-274.el5, and averaged over 5 runs. Figure 3 shows speedup results for each of the use cases. The results for the convolution of 500, 1024*1024 images, is shown in the left side of the figure. The convolution is defined as a two stage pipeline ($\|$), with the first stage being a farm (Δ) that generates the images (G), and the second stage is a farm that filters the images (F). In the convolution, the maximum speedup obtained from the refactored version is 6.59 with 2 workers in the first farm and 8 workers in the second farm. There are also three workers for each pipeline stage, (two for the farm stages, and one for the initial streaming stage), plus threads for the load balancers in the farm, giving a total of 15 threads. Here, the nature of the application may limit the scalability. The second stage of

the pipeline dominates the computation: the first stage takes on average 0.6 seconds and the second stage takes around 7 seconds to process one image, resulting in a substantial bottleneck in the second stage.

The ant colony optimisation was executed with 2000 jobs and with 3000 ants (each ant corresponds to one task). We can observe a linear speedup up to 7.29 using 8 farm workers, after which the performance starts to drop noticeably. We observe only relatively modest speedups (between 3 and 4) with more than 12 farm workers. The Ant Colony Optimisation is quite a memory-intense application, and all of the farm workers need to access a large amount of shared data (processing times, deadlines and weights of jobs, together with τ matrix), especially since we are considering instances where the number of jobs to be scheduled is large. Since the architecture on which we have tested is NUMA, the drop in performance is due to expensive memory accesses for those farm workers that are placed on remote cores (i.e., not in the same processor package). The fact that the decrease in performance occurs at about 10 farm workers confirms this: this is exactly the point where not all of the farm workers can be placed on cores from one package[3].

For the BasicN2 use case (shown in Listing 3 in the right column), the refactored version achieves a speedup of 21.2 with 24 threads. The application scales well, with near linear speedups (up to 11.15 with 12 threads). After 12 threads, the speedups decrease slightly, most likely because the refactored code dynamically allocates memory for the tasks during the computation, resulting in some small overhead. FastFlow also reserves two workers: for the load balancer and the farm skeleton, so the maximum speedup achievable with 24 threads is only 21.2. BasicN2 gives scalable speedups due to its data-parallel structure, where each task is independent, and the main computation over each task dominates the computation. In Listing 3, we also show the manually parallelised version in FastFlow, which achieves comparable speedups of 22.68 with 24 threads. The manual refactored code achieves slightly better speedups due to the fact that only one FastFlow farm is introduced in the code. However, in the refactored version, due to the tool's limitation, we introduce two FastFlow farms with an additional barrier point between them. The refactoring tool does not yet have provision to merge two routines, which can be achieved by an experienced C++ programmer.

The Graphical Lasso use case gives a scalable speedup of 9.8, for 16 cores, and stagnates afterwards. This is similar to manually ported FastFlow code, and to results obtained with OpenMP (which achieved a maximum speedup of 11.3 on 16 cores). Although the tasks parallelised here are, in principle, independent, we expected significant deviation from linear scaling for higher numbers of cores, because of cache synchronisation (disjunct but interleaving memory regions are updated in the tasks), and an uneven size combined with a limited number of tasks (48). At the end of the computation, some cores will wait for the completion of remaining tasks. The observed performance matched our expectations, providing considerable speedup with a small investment in manual code changes.

Table I shows approximate porting metrics for each use case, with the time taken to implement the manual parallel FastFlow implementation by an expert, the time to parallelise the sequential version using the refactoring tool, and the lines of code introduced by the refactoring tool. Clearly the refactoring tool gives an *enormous* saving in effort over the manual implementation of the FastFlow code.

[3]FastFlow reserves some cores for load balancing, the farm emitter/collector.

Table I. APPROXIMATE IMPLEMENTATION TIME, MANUAL VS. REFACTORING

	Man.Time	Refac. Time	LOC Intro.
Convolution	3 days	3 hours	58
Ant Colony	1 day	1 hour	32
BasicN2	5 days	5 hours	40
Graphical Lasso	15 hours	2 hours	53

V. RELATED WORK

Refactoring has a long history, with early work in the field being described by Partsch and Steinbruggen in 1983 [16], and Mens and Tourwé producing a survey of refactoring tools and techniques in 2004 [15]. The first refactoring tool system was the *fold/unfold* system of Burstall and Darlington [9] which was intended to transform recursively defined functions. There has so far been only a limited amount of work on refactoring for parallelism [17]. We have previously [18] used Template Haskell [20] with explicit cost models to derive automatic farm skeletons for Eden [14]. Unlike the approach presented here, Template-Haskell is compile-time, meaning that the programmer cannot continue to develop and maintain his/her program after the skeleton derivation has taken place. In [4], we introduced a parallel refactoring methodology for Erlang programs, demonstrating a refactoring tool that introduces and tunes parallelism for Skeletons in Erlang. Unlike the work presented here, the technique is limited to Erlang is demonstrated on a small and limited set of examples, and we did not evaluate reductions in development time. Other work on parallel refactoring has mostly considered loop parallelisation in Fortran [22] and Java [12]. However, these approaches are limited to concrete and fairly simple structural changes (such as loop unrolling) rather than applying high-level pattern-based rewrites as we have described here. We have recently extended HaRe, the Haskell refactorer [6], to deal with a limited number of parallel refactorings [7]. This work allows Haskell programmers to introduce data and task parallelism using small structural refactoring steps. However, it does not use pattern-based rewriting or cost-based direction, as discussed here. A preliminary proposal for a language-independent refactoring tool was presented in [5], for assisting programmers with introducing and tuning parallelism. However, that work focused on building a refactoring tool supporting multiple languages and paradigms, rather than on refactorings that introduce and tune parallelism using algorithm skeletons, as in this paper.

VI. CONCLUSIONS AND FUTURE WORK

This paper has introduced a novel refactoring approach for parallelising medium-scale realistic use-cases in C++, using FastFlow skeletons. The refactoring approach to parallelisation has demonstrated that easy and scalable speedups are achievable (in one case, up to 21 on a 24-core machine) with a massively reduced programmer workload. Based on readily available and easy to use profiling tools, the programmer identifies the most computationally intensive parts of his/her

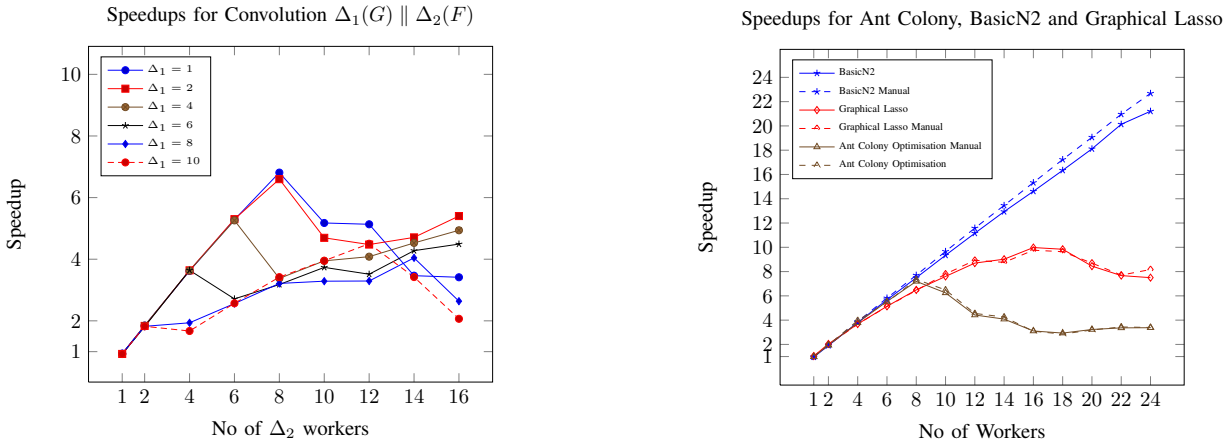

Figure 3. Refactored Use Case Results in FastFlow

code and simply points the refactoring tool towards them. The actual parallelisation is then performed by the refactoring tool, supervised by the programmer. This can give significant savings in effort, of about one order of magnitude. This is achieved without major performance losses: as desired, the speedups achieved with the refactoring tool are approximately the same as for full-scale manual implementations by an expert. In future we expect to develop this work in a number of new directions, including adding advanced performance models to the refactoring process, thus allowing the user to accurately predict the parallel performance from applying a particular refactoring with a specified number of threads. This may be particularly useful when porting the applications to different architectures, including adding refactoring support for GPU programming in OpenCl. Also, once sufficient automisation of the refactoring tool is achieved, the best parametrisation regarding parallel efficiency can be determined via optimisation, further facilitating this approach. In addition, we also plan to implement more skeletons, particularly in the field of computer alegbra and physics, and demonstrate the refactoring approach with these new skeletons on a wide range of realistic applications. This will add to the evidence that our approach is general, usable and scalable. Finally, we intend to investigate the limits of scalability that we have obvserved for some of our use-cases, aiming to determine whether the limits are hardware artefacts or algorithmic.

REFERENCES

[1] M. Aldinucci, M. Danelutto, P. Kilpatrick and M. Torquati. FastFlow: High-Level and Efficient Streaming on Multi-Core. *Programming Multi-core and Many-core Computing Systems*. Parallel and Distributed Comptuing. Chap. 13, 2013. Wiley.

[2] Michael P Allen. Introduction to Molecular Dynamics Simulation. *Computational Soft Matter: From Synthetic Polymers to Proteins*, 23:1–28, 2004.

[3] M. den Besten, T. Stuetzle, M. Dorigo. Ant Colony Optimization for the Total Weighted Tardiness Problem *PPSN 6*, p611-620, Sept. 2000.

[4] C. Brown, K. Hammond, M. Danelutto, P. Kilpatrick, and A. Elliott. Cost-Directed Refactoring for Parallel Erlang Programs. in *International Journal Parallel Processing. HLPP 2013 Special Issue*. Springer. Paris, September 2013. DOI 10.1007/s10766-013-0266-5

[5] C. Brown, K. Hammond, M. Danelutto, and P. Kilpatrick. A Language-Independent Parallel Refactoring Framework. in *Proc. of the Fifth Workshop on Refactoring Tools (WRT '12).*, Pages 54-58. ACM, New York, USA. 2012.

[6] C. Brown, H. Li, and S. Thompson. An Expression Processor: A Case Study in Refactoring Haskell Programs. *Eleventh Symp. on Trends in Func. Prog.*, May 2010.

[7] C. Brown, H. Loidl, and K. Hammond. Paraforming: Forming Haskell Programs using Novel Refactoring Techniques. *12th Symp. on Trends in Func. Prog.*, Spain, May 2011.

[8] C. Brown, K. Hammond, M. Danelutto, P. Kilpatrick, H. Schöner, and T. Breddin. Paraphrasing: Generating Parallel Programs Using Refactoring. In *10th International Symposium, FMCO 2011*. Turin, Italy, October 3-5, 2011. Revised Selected Papers. Springer-Berlin-Heidelberg. Pages 237-256.

[9] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *J. of the ACM*, 24(1):44–67, 1977.

[10] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Par. and Distrib. Computing. Pitman, 1989.

[11] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Par. Computing*, 30(3):389–406, 2004.

[12] D. Dig. A Refactoring Approach to Parallelism. *IEEE Softw.*, 28:17–22, January 2011.

[13] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Sparse Inverse Covariance Estimation with the Graphical Lasso. *Biostatistics*, 9(3):432–441, July 2008.

[14] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Func. Prog. in Eden. *J. of Func. Prog.*, 15(3):431–475, 2005.

[15] T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.

[16] H. Partsch and R. Steinbruggen. Program Transformation Systems. *ACM Comput. Surv.*, 15(3):199–236, 1983.

[17] K. Hammond, M. Aldinucci, C. Brown, F. Cesarini, M. Danelutto, H. Gonzalez-Velez, P. Kilpatrick, R. Keller, T. Natschlager, and G. Shainer. The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. FMCO. Feb. 2012.

[18] K. Hammond, J. Berthold, and R. Loogen. Automatic Skeletons in Template Haskell. *Parallel Processing Letters*, 13(3):413–424, September 2003.

[19] W. Opdyke. Refactoring Object-Oriented Frameworks. PhD Thesis, Dept. of Comp Sci, University of Illinois at Urbana-Champaign, Champaign, IL, USA (1992).

[20] T. Sheard and S. P. Jones. Template Meta-Programming for Haskell. *SIGPLAN Not.*, 37:60–75, December 2002.

[21] D. B. Skillicorn and W. Cai. A Cost Calculus for Parallel Functional Programming. *J. Parallel Distrib. Comput.*, 28(1):65–83, 1995.

[22] J. Wloka, M. Sridharan, and F. Tip. Refactoring for reentrancy. In *ESEC/FSE '09*, pages 173–182, Amsterdam, 2009. ACM.