Paraphrasing: Generating Parallel Programs using Refactoring

Christopher Brown[1], Kevin Hammond[1], Marco Danelutto[2], Peter Kilpatrick[3], Holger Schöner[4], and Tino Breddin[5]

[1] School of Computer Science, University of St Andrews, Scotland KY16 9SX, UK.
[2] Dept. Computer Science, Univ. Pisa, Largo Pontecorvo 3, 56127 PISA, Italy.
[3] Sch. Electronics, Electrical Eng. and Comp. Sci., Queen's University Belfast, UK.
[4] Software Competence Centre Hagenberg GmbH, Austria.
[5] Erlang Solutions, London. UK.

Email: chrisb@cs.st-andrews.ac.uk, kh@cs.st-andrews.ac.uk, marcod@di.unipi.it, p.kilpatrick@qub.ac.uk, Holger.Schoener@scch.at

Abstract. Refactoring is the process of changing the structure of a program without changing its behaviour. Refactoring has so far only really been deployed effectively for sequential programs. However, with the increased availability of multicore (and, soon, *manycore*) systems, refactoring can play an important role in helping both expert and non-expert parallel programmers structure and implement their parallel programs. This paper describes the design of a new refactoring tool that is aimed at increasing the programmability of parallel systems. To motivate our design, we refactor a number of examples in C, C++ and Erlang into good parallel implementations, using a set of formal pattern rewrite rules.

1 Introduction

Despite Moore's "law" [**?**], uniprocessor clock speeds have now stalled. Rather than using single processors running at ever-higher clock speeds, and drawing ever-increasing amounts of power, even consumer laptops, tablets and desktops now have dual-, quad- or hexa-core processors. **Haswell**, Intel's next multicore architecture, will have eight cores by default. Future hardware is likely to have even more cores, with *manycore* and perhaps even *megacore* systems becoming mainstream. This means that programmers need to start *thinking parallel*, moving away from traditional programming models where parallelism is a bolted-on afterthought towards new models where parallelism is an intrinsic part of the software development process. One means of developing parallel programs that is attracting increasing interest is to employ parallel patterns, that is, sets of basic, predefined building blocks that each model and embed a frequently recurring pattern of parallel computation. An application is then a composition of these basic building blocks, that may be specialized by providing (suitably wrapped) sequential portions of code implementing the business logic of the application. Examples of such patterns include *farms*, *pipelines*, *map-reduces*, etc. By taking a pattern-based approach the application programmer can focus on providing the business code and, having identified a parallel pattern (composition) that is

suitable for his/her application from a set of available patterns, can then get "for free" the necessary behind-the-scenes code, such as that for implementing synchronization and communication among the parallel activities. We thus envisage that an application programmer will begin with a sequential version of his/her business code and proceed to introduce parallelism by selecting parallel patterns that are suitable for the application at hand *and* for the target architecture. This, of course, requires expertise in pattern usage. It also requires suitable software support to facilitate introduction of patterns into the existing (sequential) code. This paper addresses this latter issue by exploring the use of refactoring as a means of bringing parallelism to business code via patterns.

1.1 Using Refactoring for Parallelism

Refactoring is the process of changing the internal structure of a program, while preserving its behaviour. The term *refactoring* was first introduced by Opdyke in 1992 [**?**], but the concept goes back to the fold/unfold system proposed by Darlington and Burstall in 1977 [**?**]. In contrast to general program transformations, such as *generic programming*, the key defining aspect of refactoring is its focus on *purely structural changes* rather than on changes in program functionality. Some advantages of refactoring are as follows:

- Refactoring aims to *improve software design*. Without refactoring, a program design will naturally decay: as code is changed, it progressively loses its structure, especially when this is done without fully understanding the original design. Regular refactoring helps tidy the code and retain its structure.
- Refactoring makes software *easier to understand*. Refactoring helps improve readability, and so makes code easier to change. A small amount of time spent refactoring means that the program better communicates its purpose.
- Refactoring helps the programmer to *program more rapidly*. Refactoring encourages good program design, which allows a development team to better understand their code. A good design is essential to maintaining rapid, but correct, software development.

Our refactoring tool will be developed as part of the ParaPhrase project, a new 3 year EU Framework-7 research project. ParaPhrase will use refactoring together with high-level design patterns[6] to introduce parallelism into sequential programs. In this paper we outline the design for the ParaPhrase refactoring tool that will refactor sequential programs written in C/C++ and Erlang to introduce parallelism, and will also refactor parallel programs in C/C++ and Erlang into more efficient implementations. By targeting C/C++ and Erlang we can demonstrate the effectiveness of our approach and its applicability to different paradigms.

[6] A *parallel (design) pattern* is a natural language description of a problem and of the associated solution techniques that the parallel programmer may use to solve that problem; an *algorithmic skeleton* is a programming entity used to implement a parallel design pattern. Here, for simplicity, we use the terms interchangeably.

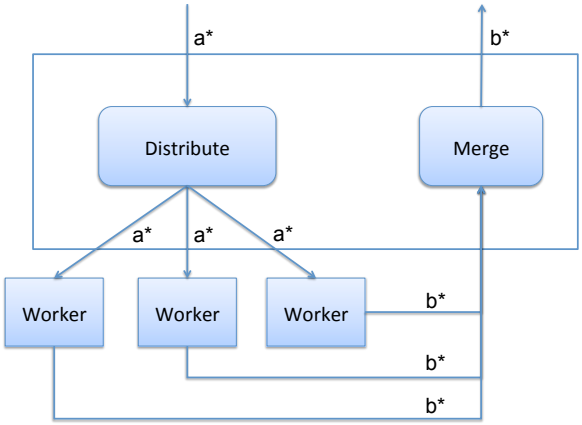

Fig. 1. A Typical Task Farm Showing a Master *Distribute* Function and the *Worker*s.

Listing 1 Sequential C Program Showing a Set of Tasks and a Worker Function *Before* the Refactoring Process

```
1  int main(int argc, char *argv[]) {
2    compute();
3  }
4  void compute() {
5      int i, task[MAX_TASKS];
6      for (i=0; i<MAX_TASKS; i++) {
7          task[i] = ... ;
8          payload(task[i]);   // set up some "tasks"
9      }
10 }
```

The specific technical contributions of this paper are:

1. we show how structured transformation techniques can enhance the *programmability* of parallel systems through refactoring;
2. we present a novel design for a new, generic, refactoring system that aims to transform programs into efficient parallel implementations, exploiting pattern-based rewrite rules that operate on systems of well-structured software components; and,
3. we present a number of new examples showing how refactoring can be used to aid a programmer in implementing parallelism.

Listing 2 Parallel C Program Showing an MPI Farm *After* the Refactoring Process

```
1  #include <mpi.h>
2  int main(int argc, char *argv[]) {
3    int np, rank;
4    MPI_Init(&argc, &argv);
5    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
6    MPI_Comm_size(MPI_COMM_WORLD,&np);
7    if (rank == 0) {
8       compute(np-1);
9    } else {
10      worker_component();
11   }
12   MPI_Finalize();
13 }
14 void compute(int workers) {
15     int i, task[MAX_TASKS];
16     for (i=0; i<MAX_TASKS; i++) {
17         task[i] = ... ;
18     }
19     for (i=0; i<workers; i++) {
20         MPI_Send(&task[i], 1, MPI_INT, i+1, i, MPI_COMM_WORLD);
21     }
22     while (i<MAX_TASKS) {
23       MPI_Recv(&temp, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
24       who = status.MPI_SOURCE;
25       tag = status.MPI_TAG;
26       result[tag] = temp;
27       MPI_Send(&task[i], 1, MPI_INT, who, i, MPI_COMM_WORLD);
28       i++;
29     }
30     for (i=0; i<workers; i++) {
31       MPI_Recv(&temp,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
32       who = status.MPI_SOURCE;
33       tag = status.MPI_TAG;
34       result[tag] = temp;
35       MPI_Send(&task[i], 1, MPI_INT, who, NO_MORE_TASKS, MPI_COMM_WORLD);
36     }
37 }
38 int computation(int x) {
39     return (payload(x));
40 }
41 void worker_component(){
42    int result, task;
43    MPI_Recv(&task, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
44    tag = status.MPI_TAG;
45    while (tag != NO_MORE_TASKS) {
46      result = computation(task);
47      MPI_Send(&result, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
48      MPI_Recv(&task, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD,&status);
49      tag = status.MPI_TAG;
50    }
51 }
```

2 Motivation

To motivate our refactoring design, we explore a simple refactoring example[7] that introduces a *task farm* [**?**] skeleton in C. A task farm is implemented as a series of *Worker* functions which are mapped to processor nodes. Each worker takes a set of *tasks* from a *master* and runs some computation that produces a *result* for each task. All results are fed back to the master. This is shown in Figure 1, where *Distribute* is the master, the sequence of inputs is shown as *a** and the sequence of results (which may be a different type to the tasks) is shown as *b**. The *Merge* function merges the results as they are delivered.

The sequential program *before* the refactoring process is shown in Listing 1. The program itself is relatively simple: it simply creates some *tasks* and then performs a *computation* for each task. The computation itself is not important here, so we simply use the dummy function payload. In a real application, this function would be replaced by some meaningful computation for each task. The highlighted pieces of code are the parts that are needed as *inputs* to the refactoring tool. In the listing, the user has highlighted compute to act as the *Distribute* function; task[i] to represent the list of *Tasks* and payload(task[i]); to represent the *Worker* function. The refactorer will generate a *Merge* function, based on knowledge about C array processing. The user simply has to select these portions of code in a refactoring editor, and choose the *Introduce Task Farm* refactoring from the ParaPhrase refactorer. Preconditions, such as checking for non side-effecting code in the highlighted components would be done automatically, and the refactoring would fail if these conditions are not met. The refactored code is shown in Listing 2, which reveals the significant amount of boilerplate code that needs to be introduced to set up a task farm, including various low-level calls to MPI [**?**]. Most of the new code deals with accumulating the results and terminating the program. Once a worker processes a task, the result is returned to the master, and a new task is sent to the worker. When there are no more tasks, a termination message is sent to the worker. The important thing to note is that a refactoring tool will *automate all of these steps for the programmer*. The programmer can start with their sequential program, choose a *task farm* refactoring and have the refactoring tool produce the parallel version, complete with all the necessary MPI calls etc. For a complex program, this can be an *enormous* saving in effort. Even for a simple program, there is a significant saving in not needing to understand the detail of the MPI implementation.

In the remainder of this section, we demonstrate the *manual* steps that are needed to perform this refactoring by hand. We start with the simple C program shown in Listing 1. The first step in this refactoring process is to identify the *computation* component for the workers. In our example we identify the call to payload as the computation component and isolate this as a component:

```
1 int main(...) {
2 ...
3 }
4
```

[7] We use C and a task farm here for their familiarity and relative simplicity.

```
5  void compute() {
6      ...
7      for (i=0; i<MAX_TASKS; i++) { // set up some tasks
8          task[i] = ...
9          (void) computation(task[i]);
10     }
11 }
12
13 int computation(int x) {
14  return (payload(x));
15 }
```

The next stage is to identify the component that will represent the *workers* of the task farm. A refactoring tool will introduce this worker component automatically, by also introducing the MPI calls that send tasks to the workers.

```
1  int computation(int x);
2
3  int main(int argc, char *argv[]) {
4    int np, rank;
5    MPI_Status status;
6    MPI_Init(&argc, &argv);
7    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8    MPI_Comm_size(MPI_COMM_WORLD, &np);
9
10   if (rank == 0)
11      compute(np-1);
12   else
13      worker_component();
14
15   MPI_Finalize();
16 }
17
18 void compute(int workers) {
19   int i, task[MAX_TASKS];
20   MPI_Status status;
21   for (i=0; i<MAX_TASKS; i++) { // set up some tasks
22         task[i] = ...
23         (void) computation(task[i]);
24   }
25
26   for (i=0; i<workers; i++)
27     MPI_Send(&task[i], 1, MPI_INT, i+1, i, MPI_COMM_WORLD);
28 }
29
30 void worker_component() {
31    int result, task;
32    MPI_Recv(&task, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
33    result = computation(task);
34 }
```

The main function is modified so that the workers are evaluated by separate MPI tasks. The main MPI task becomes the master (compute). Tasks are sent

Listing 3 Sequential Erlang Program *Before* Task Farm Refactoring

```
1 -module(taskFarm).
2 -export([run/2]).
3
4 run(Fun, Parameters) ->
5     [Fun(P) || P <- Parameters].
```

to the workers in a round-robin manner. At this stage the results are not yet accumulated, and no termination checking has been introduced. This is done in the final stage of the refactoring, which produces the code in Listing 2. Although there were only a small number of steps needed to perform this refactoring by hand, the programmer needs to understand how to implement a task farm skeleton, including checking for termination, and also needs expert knowledge in the use of MPI. It would be very easy to make a mistake at any of these steps, which could make debugging the parallel version very difficult. A refactoring tool, on the other hand, which automates this for a programmer, can eliminate potential mistakes and so allow the programmer to focus their effort on program design, rather than on the intricate details of implementing skeletons.

2.1 Erlang Example

We now show how to refactor an equivalent skeleton implementation of the same *task farm* in Erlang. Erlang [**?**] is a strict, impure, dynamically-typed functional programming language with support for higher-order functions, pattern matching, concurrency, communication, distribution, fault-tolerance and dynamic code loading. We begin with a sequential Erlang program, which simply maps a function Fun over a list, Parameters, as shown in Listing 3. Due to Erlang's functional style, functions are higher-order, meaning that they can take functions as arguments and return functions as results. We want to refactor this program into a task farm skeleton, as shown in Figure 1. The Erlang version is similar to the C version: we need to identify a number of components that will act as the *Workers* and the *Master*; a *Distribute* function is also required to merge the results of the workers. In Listing 3, the user has highlighted run as the *Master* component; Fun as the *Worker* and Parameters as the list of tasks.

The refactored version is shown in Listing 4. Here the refactoring has introduced a new function, do_run that takes three arguments: Fun, the computation to be performed by each worker; Parameter, the task sent to each worker; and Origin, the address of the master node on the network. Clearly, do_run acts as the *Worker* function in the task farm, computing the result by applying the computation Fun to the task, Parameter, and then sending the result back to the *Master*. Erlang uses the (!) primitive to send messages, and the receive primitive to receive messages. In the refactored example, the *Merge* function is expressed as the expression [receive {P, R} -> R end || P <- Procs]. which receives messages from the workers as they arrive. This *list comprehension* ensures that the merge only waits for the same number of results as there were original tasks. This merged list is then returned as the result of the program. It is also important to stress

Listing 4 Parallel Erlang Program *After* Task Farm Refactoring

```
1  -module(taskFarm).
2
3  -export([run/2, do_run/3]).
4
5  run(Fun, Parameters) ->
6      Procs = [spawn(?MODULE, do_run, [Fun, P, self()]) || P <- Parameters],
7      [receive {P, R} -> R end || P <- Procs].
8
9  do_run(Fun, Parameter, Origin) ->
10     Result = Fun(Parameter),
11     Origin ! {self(), Result}.
```

Listing 5 Parallel Erlang Program *After* a Renaming

```
1  -module(taskFarm).
2
3  -export([run/2, do_run/3]).
4
5  run(Fun, Tasks) ->
6      Procs = [spawn(?MODULE, do_run, [Fun, T, self()]) || T <- Tasks],
7      [receive {T, R} -> R end || T <- Procs].
8
9  do_run(Fun, Task, Origin) ->
10     Result = Fun(Task),
11     Origin ! {self(), Result}.
```

that the program in Listing 4 can undergo a further *renaming* refactoring by renaming Parameter in do_run to Task, P to T, and Parameters in run to Tasks. The completed code is shown in Listing 5.

3 The Design of the ParaPhrase Refactoring Tool

When constructing a refactoring tool, there are two main activities to consider: *program analysis* and *program transformation*. Program analysis checks whether certain side-conditions, which are necessary for the refactoring, are met and also collects any information that is needed during the program transformation phase. Program transformation performs the actual structural code changes that comprise a given refactoring. Both these steps are highly amenable to automation. The ParaPhrase refactorer will be *syntax independent*, initially working over C/C++ and Erlang. This demonstrates the generality of our approach, allowing patterns and rewrite rules to be expressed in terms of components rather than low-level language syntax. Targeting Erlang in addition to C/C++ also allows us to explore the advantages and limitations of both the imperative and functional paradigms, whilst also contributing to both user domains. Figure 2 shows

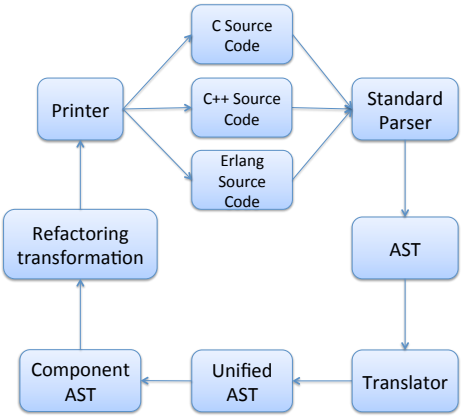

Fig. 2. The ParaPhrase Refactorer

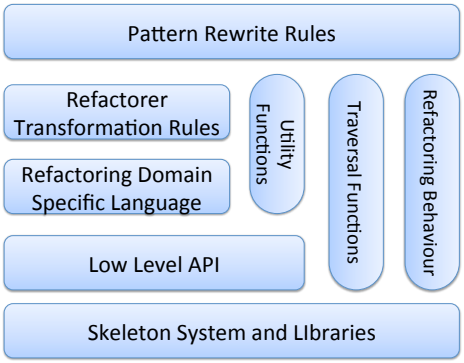

Fig. 3. The ParaPhrase Refactorer API

the design of the ParaPhrase refactorer, for C,C++ and Erlang. We note a number of components of the workflow in Figure 2:

1. The source syntax is parsed into a corresponding *Abstract Syntax Tree* (AST). Suitable static semantics must also be represented, such as use- and bind-locations for variables and the types of variables and functions. These static semantics are vital in order to correctly apply refactorings that make use of (and transform) the binding structure of a program, for example.
2. The AST is transformed into a *unified AST*, which must be general enough to express concepts from different paradigms and languages, yet still offer a sound representation of each syntax in order to transform and query it.
3. The unified AST is transformed into a *component AST* which can express the high level constructs in the program source that are required as arguments to the parallel patterns. Typically these constructs will be expressed as software components that might be identified automatically.
4. The component AST is refactored with respect to a set of well-defined transformation rules for the parallel patterns. The output of this refactoring will often be a modified version of the unified AST.
5. The refactored component AST is "pretty-printed" in the source syntax. Layout and comments should be preserved where possible, so that the programmer is presented with a refactored program that preserves their programming style and idiom.

The ParaPhrase refactoring tool will be made *user-extensible* through a number of layers of API abstractions, as shown in Figure 3. We predict that the following will be needed to support user-level pattern-based refactorings:

1. Patterns will be expressed in a high level abstract language that will allow users to write rules to *introduce* and *eliminate* patterns, together with their *composition*. This pattern language will be void of any syntax information and will be general enough to express pattern rewrites for all syntaxes. These rules are described in more detail in Section 3.1.

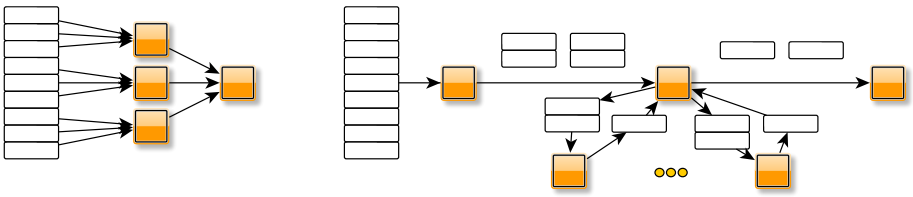

Fig. 4. Map-Reduce pattern (left) and an equivalent Pipeline-Farm pattern (right)

2. A language for expressing refactoring transformations will allow the refactorings themselves to be expressed in terms of a general syntax, including pre- and post-conditions and transformation rules.
3. A refactoring Domain Specific Language (DSL) framework will allow for the composition of the refactorings to form larger refactorings.
4. A collection of *utility* functions such as traversal functions for the Abstract Syntax Trees, retrieval of binding information, mapping an editor selection onto its Syntax Tree representation, etc.
5. A Skeleton library that the refactored source program can *import* to access low-level skeleton implementation details.
6. A concrete interface that will be integrated into a popular editing environment, such as Emacs or Eclipse.

Since there is already a refactoring tool for Erlang, Wrangler [**?**], that uses an expressive Domain Specific Language (DSL) to define refactorings in terms of their pre-conditions and transformation rules as Erlang macros, when dealing with Erlang it may be possible to plug our pattern-rewrite rules directly into the Wrangler DSL rather than using our own generic refactorer.

3.1 Patterns as Rewrite Rules

The ParaPhrase approach is based around the use of parallel patterns to drive the program transformations in the refactoring tool. Parallel patterns impose a clear and easily-recognised structure on the forms of parallelism that can be exploited in an application. As an example, if we use the well-known *map-reduce* pattern (Figure 4) to model parallel behaviour, then:

1. the signature of the function used to transform all the data collection items during the map phase is known;
2. the signature of the function used to "sum up" all the items in the result collection is known;
3. the data dependencies are known;
4. and it may, perhaps, be known whether the reduce operator is both associative and commutative.

All this information may be used to refactor the parallel computation in terms of other parallel patterns. The *map-reduce* computation could be expressed, for example, in terms of a *pipeline* pattern whose stages are:

– a stage splitting the input collection and delivering partitions of the collection to the next stage (*splitter* stage, sequential);

– a stage modelled after the "embarrassingly parallel" parallel pattern (the *partition map-reduce* stage, parallel), processing each partition by:
  • first applying the map operator to each partition item;
  • then applying the reduce operator to "sum up" all the computed results;
  • finally delivering the result to the stream leading to the next stage.
– a stage gathering the results from the partitions and summing them again, using the reduce operator.

The overall refactoring in this case transforms a composition of *map* and *reduce* patterns into a composition of *pipeline* and *farm* (the embarrassingly parallel pattern on streams) patterns, as shown in Figure 4. For a stream of input collections, the refactored program may exploit more parallelism (and therefore better performance). A number of "rewrite rules" can be used once parallel patterns are identified with all their functional and non-functional parameters. Assuming we have a pattern *palette* defined by the following BNF:

$$
\begin{array}{lll}
Pattern & ::= & Pipe \mid Farm \mid Comp \mid Map \mid Reduce \mid Seq \\
Pipe & ::= & pipe(Pattern, Pattern) \\
Farm & ::= & farm(Pattern) \\
Comp & ::= & comp(Pattern, Pattern) \\
Map & ::= & map(Pattern) \\
Reduce & ::= & reduce(Pattern) \\
Seq & ::= & \langle sequential\ code\ wrapping \rangle
\end{array}
$$

then the rules in Table 1 can be used to apply "parallel" refactoring. All these rules preserve the "functional" semantics, but not the "parallel" semantics. Both sides of each rule give the same results, but the computation may involve different parallel patterns and therefore different performance. The performance obviously depends on various parameters, including application-related and target-architecture related ones. These rules are used *de facto* by the *Pattern Rewrite Rules* box in Figure 3 to drive the *refactorer*. It is worth pointing out that:

– where all the parallel patterns are directly available as algorithmic skeletons (e.g. as entries of a skeleton library), the refactoring process may simply consist of substituting sequences of library calls;

P	→	farm(P)	farm introduction
farm(P)	→	P	farm elimination
pipe(pipe(P1,P2),P3)	≡	pipe(P1,pipe(P2,P3))	pipeline assoc
pipe(map(P1),map(P2))	≡	map(pipe(P1,P2))	pipe/map distrib
pipe(map(P1),reduce(P2))	→	pipe(comp(map(P1),reduce(P2)),reduce(P2))	map reduce promotion
map(P1)	≡	pipe(coll2singleton,farm(P1),singleton2coll)	map/farm equivalence
pipe(P1,P2)	→	comp(P1,P2)	stream par elimination
comp(P1,P2)	→	pipe(P1,P2)	stream par introduction

Table 1. Parallel pattern rewriting rules. The coll2singleton and singleton2coll functions transform a collection into a stream of collection item components and vice versa.

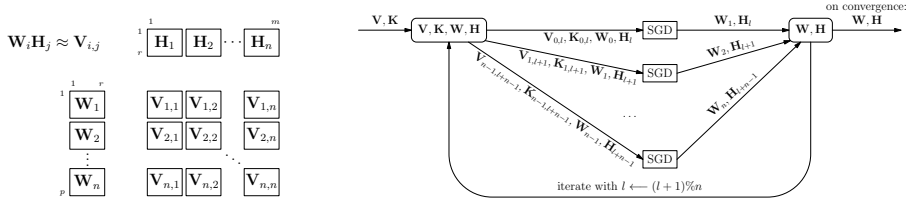

Fig. 5. Structure of coarse parallelization approach. By dividing matrix $\mathbf{V}$ into blocks, only parts i and j of matrices $\mathbf{W}$, $\mathbf{H}$, respectively, are used for the approximation of each block (i, j) of $\mathbf{V}$. n blocks can thus be processed simultaneously with Stochastic Gradient Descent (SGD) without conflicts of the parameter updates. After some SGD iterations, the parameters $\mathbf{W}$, $\mathbf{H}$ are collected again, and redistributed using a different set of n non-overlapping blocks of $\mathbf{V}$. As $\mathbf{V}$ has $n \times n$ different blocks, there are n iterations of n parallel SGD updates until one sweep through all values in $\mathbf{V}$ is completed (one epoch of an equivalent global SGD). $\mathbf{K}$ is a $k \times 2$ matrix containing in each row the index into V of one of the k known values of $\mathbf{V}$, needed to compute the SGD iterations.

– even where skeletons are available, the refactoring using the rules above may require new sequential code to be generated. For example, for the task farm refactoring above, the *worker* code for the second stage pipeline requires some specific code to iterate the map operator over all the elements of the partition. In most cases (e.g. in the rules of Table 1) the refactoring only requires changing the (sequence of) calls to the skeleton library.

4 Use Case: Large Scale Matrix Factorization

Matrix factorization for data modeling. Factorization of large matrices is a method common in applications like recommender systems, and user preference modeling [**?**]. An example is a problem setting in which partial data of the ratings of a large number of persons for a large number of movies are considered [**?**]. This problem setting provides a suitable vehicle for parallelization and refactoring, as the approach and its parallelization are simple enough for presentation here, while still being of real world significance. In the following development we omit problem specific sequential code (cf. [**?**]), and concentrate on the parallelization and refactoring.

The known and unknown ratings of users for movies are collected in a rating matrix $\mathbf{V}$ with size $p \times m$. The rows correspond to persons, and the columns to movies. Most of the values in this matrix will be unknown, as usually users rate only a limited number of movies. An auxiliary $k \times 2$ matrix $\mathbf{K}$ is used to keep track of the known values of $\mathbf{V}$: row $(s, t) \in \mathbf{K} \Leftrightarrow \mathbf{V}_{s,t}$ is known.

To obtain estimates for probable ratings of users for movies they have not rated (or even seen) so far, a factorization of $\mathbf{V}$ into the $p \times r$ row factor matrix $\mathbf{W}$ and the $r \times m$ matrix $\mathbf{H}$ is searched, with $\mathbf{V} \approx \mathbf{WH}$ (for the known entries in $\mathbf{V}$). Estimate of an unknown rating of person s for movie t is the dot product $\mathbf{W}_{s\cdot}\mathbf{H}_{\cdot t}$.

r is the number of factors, and $\mathbf{W}$ and $\mathbf{H}$ are the factor memberships of the persons and movies. Factors could correspond to groups of movies and persons, like action movies and action loving persons, or romantic movies/persons.

The factorization is performed by optimization of a cost function w.r.t. the learned parameters $\mathbf{W}$, $\mathbf{H}$. The cost function is the mean of some loss over the known rating values, e.g. the squared difference of the rating estimates to the real ratings, for all known values $(s,t) \in \mathbf{K}$. The optimization is performed by gradient descent, which iterates epochs (one sweep through all known values) of learning until convergence of the cost value.

Parallelization Approach. This problem can be parallelized on two levels. The general approach for a high level parallelization is sketched in Figure 5. The matrix $\mathbf{V}$ is split into blocks, the parameter matrices into stripes. The matrix factorization can now be performed for each block of $\mathbf{V}$ separately, and all blocks corresponding to independent stripes of $\mathbf{W}$ and $\mathbf{H}$ can be optimized in parallel. As blocks of $\mathbf{V}$ in the same row or column share parameters and cannot be optimized in parallel, one epoch of learning needs an additional loop around these parallel optimizations, until all blocks are optimized once. Afterwards, the whole process is repeated until convergence of the parameters. The square root n of the number of blocks $n \times n$ of matrix $\mathbf{V}$ is the degree of parallelization, and can be tuned to the number of processing elements, and to the problem size.

A sequential version of this approach, comprising the starting point of the following refactoring, is shown in the C++-like pseudo code in Listing 6. The details of functions stochasticGradientDescent, partitionMatrices and randomMatrix are not important for the high level parallelization, and the implementation is just sketched. The second level of parallelization can be performed inside the stochastic gradient descent. The computations of stochastic gradient descent are mainly linear algebra. They are well parallelizable using a data-parallel approach. ParaPhrase will also provide appropriate patterns, e.g. Map-Reduce, and support for heterogeneous parallel architectures, such as distributed multicore systems with GPGPUs on each node. The matrix factorization example could then be mapped onto such a cluster by distributing the matrix blocks to the available computation nodes, and parallelizing the stochastic gradient descent on their respective GPGPUs.

Parallelization by Refactoring. Listing 6 is the starting point of refactoring for parallelization. We wish to use the *Farm* pattern to distribute the n parallel SGD computations. Before the *Farm* pattern can be applied, the worker function performing the SGD needs to be wrapped inside a *Seq* pattern, as other refactoring rules can only be applied to existing patterns (cf. Section 3.1). This is achieved by using the refactorer to mark the stochasticGradientDescent function and wrap it in a *Seq* pattern. Afterwards, the "P $\rightarrow$ farm(P)" rewriting rule can be used, to make the sequential loop parallel. The code which results from those refactorings is given in Listing 7, with the changes highlighted. Similar refactorings are also possible to introduce *map-reduce* parallelism in the linear algebra operations performed by the stochastic gradient. The convenience of code rewriting is one advantage of automatic refactoring; in addition, the following considerations make it a valuable tool for applications such as the one above:

Listing 6 Sequential version of matrix factorization

```
1  (Matrix, Matrix) matrixFactorization(Matrix V, Matrix K, int r, int n) {
2    // initializations ...
3    List<List<Matrix>> blocksV, filteredK;
4    List<Matrix> rowsW, colsH;
5    // assume that blocksV, ... are views of parts of V, ...,
6    // such that updates to rowsW, colsH also update W, H
7    (blocksV, filteredK, rowsW, colsH) = partitionMatrices(n, V, K, W, H);
8    while (!converged(loss, V, K, W, H)) {
9      loss = 0;
10     for (l ∈ {0, ..., n − 1}) {
11       for (i ∈ {0, ..., n − 1}) {
12         int j = (l+i) % n;
13         (loss_part, rowsW[i], colsH[j]) = stochasticGradientDescent(
14           blocksV[i,j], filteredK[i,j], rowsW[i], colsH[j]);
15         loss += loss_part;
16       }
17     }
18     loss /= n;
19   }
20   return (W, H);
21 }
22
23 // Performs one epoch of Stochastic gradient descent, returning loss and new W, H.
24 // An epoch corresponds to one sweep over all (known) elements of the matrix
25 (double, Matrix, Matrix) stochasticGradientDescent(Matrix V, Matrix K,
26     Matrix W, Matrix H) {
27   // ...
28 }
29
30 // split V, K, W, H into n partitions (n × n for V and K).
31 (List<List<Matrix>>, List<List<Matrix>>, List<Matrix>, List<Matrix>)
32     partitionMatrices(int n, Matrix V, Matrix K, Matrix W, Matrix H) {
33   // ...
34   // filteredK[i][j] contains all rows u of K, for which K[u] = (s, t) is
35   // an index into block (i, j) of V, recomputed to be the
36   // equivalent index into blocksV[i][j]
37   return (blocksV, filteredK, rowsW, colsH);
38 }
```

- checks will be performed to determine whether the intended refactoring is possible, and whether there are conflicts on the new identifiers;
- switches between different kinds of parallel patterns will be easier, facilitating the optimization of the patterns used for the application and for the available hardware; and
- guidance might be available regarding an optimal choice of patterns given non-functional criteria such as run-time considerations.

Listing 7 *Farm* version of matrix factorization

```
1  (Matrix, Matrix) matrixFactorization(Matrix V, Matrix K, int r, int n) {
2    // ...
3    (blocksV, filteredK, rowsW, colsH) = partitionMatrices(n, V, K, W, H);
4    Pattern seqSGD = SequentialPattern(stochasticGradientDescent);
5    Pattern farmSGDEpoch = FarmPattern(seqSGD);
6    while (!converged(loss, V, K, W, H)) {
7      loss = 0;
8      for (l ∈ {0, ..., n − 1}) {
9        for (i ∈ {0, ..., n − 1}) {
10         int j = (l+i) % n;
11         farmSGDEpoch.execute(blocksV[i,j], filterdK[i,j], rowsW[i], colsH[j]);
12       }
13       List<(double,Matrix,Matrix)> resList = farmSGDEpoch.waitforall();
14       for (i ∈ {0, ..., n − 1}) {
15         int j = (l+i) % n;
16         loss_part = resList[i][0]; rowsW[i] = resList[i][1]; colsH[j] = resList[i][2];
17         loss += loss_part;
18       }
19     }
20     loss /= n;
21   }
22   return (W, H);
23 }
```

Pattern Refactoring. To exploit the ease of refactoring parallel patterns, a refactoring to use a parallel *Map* pattern instead of a *Farm* pattern is now considered. Such refactoring could be useful for clarifying the program structure, or it might be more appropriate for a given parallel architecture. Automatic Refactoring allows easy switching between such patterns, making it straightforward to explore the available alternatives and find the optimal one for a given situation. To apply the *Map* pattern, the input arguments for the *Farm* workers have to be collected in a list, automatized as much as possible by the refactoring, to which the already existing *Seq* pattern can then be applied by the *Map*, as shown in Listing 8 with refactoring changes highlighted.

Refactoring Analysis. Comparing the three versions of the matrixFactorization function, it is obvious that large parts are very similar, a prerequisite for automatic refactoring. Still, it is also obvious that refactoring does not mean just a simple exchange of a FarmPattern by a MapPattern, or similar. Parts of the surrounding code have to be reorganized as well. In this example, the following issues arise:

Listing 8 *Map* version of matrix factorization

```
1  (Matrix, Matrix) matrixFactorization(Matrix V, Matrix K, int r, int n) {
2    // ...
3    (blocksV, filteredK, rowsW, colsH) = partitionMatrices(n, V, K, W, H);
4    Pattern seqSGD = SequentialPattern(stochasticGradientDescent);
5    Pattern mapSGDEpoch = MapPattern(seqSGD);
6    while (!converged(loss, V, K, W, H)) {
7      loss = 0;
8      for (l ∈ {0, ..., n − 1}) {
9        List<(Matrix,Matrix,Matrix,Matrix)> mapList;
10       for (i ∈ {0, ..., n − 1}) {
11         int j = (l+i) % n;
12         mapList.append( (blocksV[i,j], filteredK[i,j], rowsW[i], colsH[j]) );
13       }
14       List<(double,Matrix,Matrix)> resList = mapSGDEpoch.execute(mapList);
15       for (i ∈ {0, ..., n − 1}) {
16         int j = (l+i) % n;
17         loss_part = resList[i][0];  rowsW[i] = resList[i][1]; colsH[j] = resList[i][2];
18         loss += loss_part;
19       }
20     }
21     loss /= n;
22   }
23   return (W, H);
24 }
```

– introducing new variables; e.g. the variables for the pattern instances, but also those collecting the *Farm* or *Map* results;
– handling arguments of the original worker function and its results, e.g. by wrapping and unwrapping to and from single list items;
– splitting of loops, such that results of several iterations can be collected outside the loop, while unwrapping the results might necessitate replicating the loop a second time and repeating parts of the loop (here, "**int** j=(l+i)%n" is necessary a second time);
– here, the sequential version already contained the code necessary for splitting the four distributed matrices into blocks; it would be desirable, although maybe not realistic, to also have the refactoring introduce such helper functions as necessary;
– the sequential version has already ensured that the worker function has no side effects; it might be desirable to introduce refactorings transforming functions (and calls to them) which are not yet side-effect free.

How many of these tasks can be performed automatically, and what other tasks might be necessary for other use cases, remains to be investigated during the ParaPhrase project.

5 Related Work

Program transformation has a long history, with early work in the field being described by Partsch and Steinbruggen in 1983 [**?**] and Mens and Tourwé producing an extensive survey of refactoring tools and techniques in 2004 [**?**]. The first refactoring tool system was the *fold/unfold* system of Burstall and Darlington [**?**] which was intended to transform recursively defined functions. The overall aim of the *fold/unfold* system was to help programmers to write correct programs which are easy to modify. There are six basic transformation rules that the system is based on: unfolding; folding; instantiation; abstraction; definition and laws. The advantage of using this methodology was that it deployed a number of simple, and yet effective, structural program transformations that aimed to develop more efficient definitions; the disadvantage was that the use of the fold rule sometimes resulted in non-terminating definitions.

The Haskell Refactorer, HaRe, is a semi-automated refactoring tool for sequential Haskell programs, developed at the University of Kent by Thompson, Li and Brown [**?**]. HaRe works over the full Haskell 98 standard, and contains a large catalog of refactorings that concentrate on small structural changes in sequential Haskell programs, such as *renaming*, *lambda lifting* and type-based refactorings. HaRe was recently extended by us to deal with a limited number of parallel refactorings. This technique is known as *paraforming* [**?**], and allows Haskell programmers to construct data and task parallelism using small structural refactoring steps, although it does not use pattern-based rewriting, as in ParaPhrase. Wrangler [**?**], also developed at the University of Kent by Thompson and Li, is similar to HaRe, but works over sequential Erlang instead. Wrangler also contains a large database of refactorings for Erlang programs and includes a Domain Specific Language for expressing transformations (together with their conditions) [**?**] and a further language for composing refactorings [**?**]. Unlike our requirements for the ParaPhrase project, Wrangler does not deal with parallel refactorings in any way. Cocinelle and its DSL framework, SmPL [**?**], is a program matching and transformation engine for specifying desired matches and transformations in C code. Stratego/XT [**?**] provides a language-independent framework for expressing refactorings over arbitrarily defined syntaxes. It may be possible to use Stratego in the context of Paraphrase for expressing the parallel refactorings.

There has been a limited amount of work on parallel refactoring in general, mostly with loop parallelisation in Fortran [**?**] and Java [**?**]. However these approaches are limited to concrete structural changes (such as loop unrolling) rather than applying high-level pattern-based rewrites. A companion paper contains a much more detailed survey of refactoring tools for parallelisation [**?**].

Rewriting of structured parallel programs has been studied in different contexts. Backus' Turing award lecture note [**?**], although not explicitly dealing with parallel patterns or design patterns, sets up a scenario that allows to "compute" program transformations using an algebra of programs. In particular, several transformations related to map and reduce second order functions are already present in this work. As an example the "pipe/map" rule described in Table 1 was already present in that seminal work described as

$$\alpha f \circ \alpha g \equiv \alpha(f \circ g)$$

with the functional composition operator $\circ$ representing pipeline, as usual (stream parallel), and the *apply-to-all* α representing the map skeleton. More recently more rewriting rules have been designed for algorithmic skeletons. In [**?**] a concept of "normal form" for stream parallel skeleton compositions is introduced that maximizes the service time by applying systematic rewriting of arbitrary stream parallel skeleton compositions to farms with sequential workers. Other authors consider usage of different skeleton rewriting rules to target different architectures, in particular those including GPUs [**?**]. The full SkeTo skeleton based programming framework (see `http://sketo.ipl-lab.org/`) uses the results from Bird-Merteens theory [**?**,**?**] to optimize data parallel skeleton compositions [**?**]. The potential refactoring for C++ code by introducing algorithmic skeletons has been previously discussed in the context of FastFlow (see `http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about`). In particular, in [**?**] the refactoring of sequential code to introduce a task farm skeleton is discussed. Finally, PEPPHER: *Performance Portability and Programmability for Heterogeneous Many-core Architectures* [**?**], is an EU FP7 funded project that started in January 2010. The aim of PEPPHER is to devise a unified framework for programming and optimizing applications for a diverse range of architectures such as heterogeneous many-core processes in order to ensure performance portability. PEPPHER uses direct compilation to the target architectures, so portability is supported by powerful composition methods with a toolbox of adaptive algorithms. However, the PEPPHER project does not use refactoring and pattern rewriting to increase the programmability of parallel systems as we do here, relying instead on adaptive algorithms and architecture-directed compilation.

6 Conclusions

This paper has described a new design methodology for the ParaPhrase refactoring tool, a radically new system that will refactor systems of software components into efficient parallel implementations. The tool will refactor programs written in C, C++ and Erlang (although we also expect to extend our tool to deal with other languages such as Haskell and Python), by applying high-level abstract pattern rewrite rules that can either:

- Introduce new patterns into an existing sequential program; or,
- Modify an existing parallel program by changing the pattern already specified, or introducing a new pattern, therefore composing patterns together.

A companion paper by Hammond *et al.* [**?**] gives an overview of the ParaPhrase project encompassing many key technologies and techniques that we will employ in addition to refactoring. Among these is the need to identify means of specifying non-functional properties of systems in such a way that it becomes possible to verify that a refactoring achieves its intended purpose. We have demonstrated the effectiveness of having such a refactoring tool by motivating our design with a number of key examples in C, Erlang and C++, where we showed that having a refactoring tool can effectively *automate* the majority of the boilerplate implementation detail of introducing a new skeleton,

such as adding MPI code in C, or adding a master/worker skeleton in Erlang. This is potentially an *enormous* saving in effort, allowing the programmer to focus on designing algorithms rather than worrying about the details of parallel implementation. We believe this is the correct way to improve substantially the *programmability* of such parallel systems. A refactoring tool such as the one described here would be a key component in increasing the productivity of parallel programming in general.

Acknowledgments

This work has been supported by the European Union grants RII3-CT-2005-026133 SCIEnce: Symbolic Computing Infrastructure in Europe, IST-2010- 248828 ADVANCE: Asynchronous and Dynamic Virtualisation through performance ANalysis to support Concurrency Engineering, and IST-2011-288570 ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems, and by the UK's Engineering and Physical Sciences Research Council grant EP/G055181/1 HPC-GAP: High Performance Computational Algebra.