



Restoration of Legacy Parallelism: Transforming Pthreads into Farm and Pipeline Patterns

Vladimir Janjic¹ · Christopher Brown²  · Adam D. Barwell³

Received: 20 November 2020 / Accepted: 4 May 2021 / Published online: 10 June 2021
© The Author(s) 2021

Abstract

Parallel patterns are a high-level programming paradigm that enables non-experts in parallelism to develop structured parallel programs that are maintainable, adaptive, and portable whilst achieving good performance on a variety of parallel systems. However, there still exists a large base of *legacy-parallel code* developed using ad-hoc methods and incorporating low-level parallel/concurrency libraries such as *pthread*s without any parallel patterns in the fundamental design. This code would benefit from being restructured and rewritten into pattern-based code. However, the process of rewriting the code is laborious and error-prone, due to typical concurrency and pthreading code being closely intertwined throughout the business logic of the program. In this paper, we present a new software restoration methodology, to transform legacy-parallel programs implemented using *pthread*s into structured farm and pipeline patterned equivalents. We demonstrate our restoration technique on a number of benchmarks, allowing the introduction of patterned farm and pipeline parallelism in the resulting code; we record improvements in cyclomatic complexity and speedups on a number of representative benchmarks.

Keywords Parallel patterns · Restoration · Pthreads · Program transformation · Refactoring · Code analysis · Farm · Pipeline · TBB

✉ Christopher Brown
cmb21@st-andrews.ac.uk

Vladimir Janjic
vjanjic001@dundee.ac.uk

Adam D. Barwell
a.barwell@imperial.ac.uk

¹ School of Science and Engineering, University of Dundee, Dundee, UK

² School of Computer Science, University of St Andrews, St Andrews, UK

³ Department of Computing, Imperial College London, London, UK

1 Introduction

Parallel patterns are a well-established high-level parallel programming model for producing portable, maintainable, adaptive, and efficient parallel code. They have been endorsed by some of the biggest IT companies, such as Intel and Microsoft, who have developed their own parallel pattern libraries; e.g. Intel TBB [35] and Microsoft PPL. A standard way to use these libraries is to start with a sequential code base, identifying in it the portions of code that are amenable to parallelisation, together with the exact parallel pattern to be applied. Proceeding with instantiating the identified pattern at the identified location in the code, after possibly restructuring the code to accommodate the parallelism. *Sequential code therefore gives the cleanest starting point for introduction of parallel patterns.* There exists, however, a large base of *legacy* code that was parallelised using lower-level, mostly ad-hoc parallelisation methods and libraries, such as *pthreads* [12]. This code is usually very hard to read and understand, is tailored to a specific parallelisation, and optimised for a specific architecture, effectively preventing alternative (and possibly better) parallelisations and limiting portability and adaptivity of the code. An even bigger problem, from a software engineering perspective, is the maintainability of the legacy-parallel code: commonly, the programmer who wrote it is the only one who can understand and maintain the code. This is due to both complexity of low-level threading libraries and the need for custom-built data structures, synchronisation mechanisms, and sometimes even thread/task scheduling implemented in the code. The benefits of using parallel patterns lie in a clear separation between sequential and parallel parts of the code and a high-level description of the underlying parallelism, making the patterned applications much easier to maintain, change, and adapt to new architectures. In this paper, we deal with *farms* and *pipelines*. In a farm, a single computational worker is applied to a set of independent inputs. The parallelism arises from applying the worker to different input elements in parallel. In a parallel pipeline, a sequence of functions, f_1, f_2, \dots, f_m are applied to a stream of independent inputs, x_1, \dots, x_n where the output of f_i becomes the input to f_{i+1} ; the parallelism arises from executing $f_{i+1}(f_i(\dots f_1(x_k)\dots))$ in parallel with $f_i(f_{i-1}(\dots f_1(x_{k+1})\dots))$. In this paper, we present a *new methodology* for the restoration of legacy-parallel code into an equivalent *patterned* form, through the application of a number of identified program transformations; the ultimate goal of which is to provide a semi-automatic way of converting legacy-parallel code into an equivalent patterned code, therefore increasing its maintainability, adaptivity, and portability whilst either improving or maintaining performance. The transformations presented in this paper are intended as manual transformations. We envisage incorporating implementations of these refactorings into a semi-automated refactoring tool as future work.

This paper makes the following specific research contributions:

1. we present a novel software restoration methodology for converting legacy-parallel applications into their structured (patterned) parallel equivalents, via the farm and pipeline patterns;

- we present a new set of (manual) *restoration* transformations that attempt to *systematically*, (i) *eliminate* pthread operations from legacy C/C++ programs; (ii) perform *code repair*, fixing any bugs introduced in *i*; and, (iii) *reshape* code in preparation for parallel pattern introduction;
- we evaluate these transformations on a set of benchmarks, demonstrating that removal of parallelism can allow us to manually derive structured parallel code that is comparable to the original legacy-parallel version in terms of performance, while being more portable, adaptive, and maintainable.

2 Software Restoration

In this section, we propose a new *Software Restoration* methodology for improving the structure of legacy-parallel C++ code by applying a series of incremental program analysis and transformation steps to rewrite the code into its patterned equivalent. Software restoration is based on program transformation and code analysis and aims to:

- discover* the instances of common patterns in legacy-parallel code;
- eliminate* undesirable legacy parallel primitives from the same code; and
- replace* the removed parallel primitives with instances of parallel patterns.

The input to the Software Restoration process is a legacy-parallel C/C++ program that is based on some low-level parallelism library, such as pthreads, and the output is a semantically-equivalent C/C++ program based on parallel patterns. In this way, we obtain well-structured code based on a higher level of parallel abstraction, which is significantly more maintainable and adaptive while still preserving good performance of the original, highly-tuned parallel version. In this paper, we will

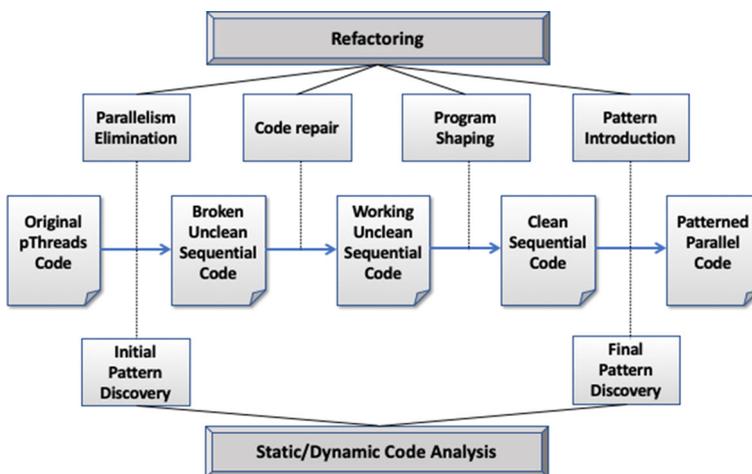


Fig. 1 Software restoration process

focus on the TBB library as our target code. It is important to note, however, that transforming the code into a *patterned* form also increases the portability of the code and gives a wider opportunity for parallelisation using different techniques, libraries and pattern approaches. In this paper, we target TBB as just one example of a typical and common pattern library but the patternisation step could easily be replaced with other equivalent and more general frameworks; e.g. the *Generic Reusable Parallel Pattern Interface* (GrPPI) [18], which allows multiple different pattern backends to be targetted from a single interface. Indeed, prior work on refactoring to introduce GrPPI [8] patterns could easily be deployed at this stage, further increasing portability of the patterned code.

The Software Restoration methodology consists of a number of steps, each applying a class of code transformations, some of which are driven by the pattern discovery code analysis. The whole process is depicted in Fig. 1. In the below description, we will focus on the code transformation steps. We will use a synthetic, but representative, parallel pipeline as a running example in order to demonstrate the transformation. Listing 1 presents aspects of the original parallel code with pthreads that are pertinent to this demonstration.

Listing 1: Original Simple Pipeline Code

```

1  int main(int argc, char *argv[]) {
2  ...
3  // create the workers, then wait for them to finish
4  pthread_create(&workerid[0], &attr, Stage1, (void *)&stage_queues[0]);
5  pthread_create(&workerid[1], &attr, Stage2, (void *)&stage_queues[1]);
6  pthread_create(&workerid[2], &attr, Stage3, (void *)&stage_queues[2]);
7
8  for (i = 0; i < NRSTAGES; i++)
9      pthread_join(workerid[i], NULL);
10
11 ...
12 }
13
14 // Second stage reads an element from the input queue, adds 1 to it,
15 // and writes it to the output queue.
16 void *Stage2(void *arg) {
17     int my_input, my_output;
18     pipeline_stage_queues_t *myQueues = (pipeline_stage_queues_t *)arg;
19     queue_t *myOutputQueue = myQueues->outputQueue;
20     queue_t *myInputQueue = myQueues->inputQueue;
21
22     for (my_input = read_from_queue(myInputQueue);
23          my_input > 0 || my_input == EOS;
24          my_input = read_from_queue(myInputQueue)) {
25         if (my_input != EOS) {
26             my_output = my_input + 1;
27             add_to_queue(myOutputQueue, my_output);
28         } else { // EOS is a terminating token. Pass on if received.
29             add_to_queue(myOutputQueue, EOS);
30             break;
31         }
32     }
33     return NULL;
34 }
35
36 void add_to_queue(queue_t *queue, int elem)
37 {
38     pthread_mutex_lock(&queue->queue_lock);
39     // If the queue is full, wait until something reads from it before adding a new element
40     if (queue->nr_elements == queue->capacity)
41         pthread_cond_wait(&queue->queue_cond_read, &queue->queue_lock);
42     queue->elements[queue->addTo] = elem;
43     queue->addTo = (queue->addTo + 1) % queue->capacity;
44     queue->nr_elements++;
45     pthread_cond_signal(&queue->queue_cond_write);
46     pthread_mutex_unlock(&queue->queue_lock);
47 }

```

In the above `main` function (Lines 1–12), a pipeline of three stages is created using three threads. The stages are connected by queues such that all stages have an output queue, and stages two and three have an input queue. After creation, the `main` function waits for the threads to finish their work (Lines 8–9) before continuing. In Lines 14–34, we show the function for the middle stage of the pipeline, which reads an integer from the input queue, increments it by one, then puts it into the output queue. The first and third stages have a similar structure, where the first stage acts as a source of integers for the second stage, and the third stage doubles its inputs before adding them to the final output queue. All the relevant synchronisation code for the queues can be found in two functions: `add_to_queue` and `read_from_queue`. Only `add_to_queue` (Lines 36–47) is shown here; `read_from_queue` is defined similarly. Both functions use one mutex lock and two conditional variables. The latter are used for

synchronisation when threads are waiting to insert an element into a full queue or for reading from an empty queue (e.g. at the start of the program). When a thread needs to add to the queue, it first acquires the queue lock and checks if the queue is full (Lines 38–41). When the queue is full, the thread releases the lock and waits for a signal that some other thread has consumed an element of this queue (`queue->queue_cond_read` conditional variable, Line 41). After this conditional variable is signalled, the thread enqueues the element, updating the queue counter and pointer in the process (Lines 42–44). Finally, the thread signals that an element has been added to the queue (`queue->queue_cond_write` conditional variable, Line 45) and releases the queue lock (Line 46) before returning.

2.1 Parallelism Elimination

The initial step, *Initial Pattern Discovery*, analyses the original pthreaded code and discovers those parts of it, if any, that correspond to instances of parallel patterns. In our example, this stage identifies the linear pipeline created in Lines 4–6, with the pipeline stages being the functions: `Stage1`, `Stage2`, and `Stage3`. This process could be achieved by using a technique similar to the one described in [10] Following pattern discovery, the first code transformation step is applied, where pthread operations and primitives are either removed or transformed, eliminating parallelism. In Listing 1, this impacts `main` and `add_to_queue`. Listing 2 shows the resulting code.

Listing 2: Simple Pipeline Code with Parallelism Removed

```

1  int main(int argc, char *argv[]) {
2  ...
3  // Calls to pthread_create are converted to function calls.
4  Stage1((void *)&stage_queues[0]);
5  Stage2((void *)&stage_queues[1]);
6  Stage3((void *)&stage_queues[2]);
7
8  // The loop containing pthread_join is removed.
9
10 }
11
12 void add_to_queue(queue_t *queue, int elem) {
13 // All mutex and conditional variable operations are removed.
14 queue->elements[queue->addTo] = elem;
15 queue->addTo = (queue->addTo + 1) % queue->capacity;
16 queue->nr_elements++;
17 }

```

We note that the Parallelism Elimination stage *does not guarantee that a program's functional behaviour is preserved* and thus errors may be introduced. Here, `Stage1` contains a `for`-loop that enqueues elements in its output queue. Since the second stage, which reads from that queue, is no longer consuming those elements concurrently, and the queue is smaller than the total number of elements produced, the second stage will now consume and process only a subset of its inputs in the original pthreaded version after `Stage1` returns. Because the semantics of the program have changed following Parallelism Elimination, the code must be *repaired*.

2.2 Code Repair

Our example is just one of many in which merely removing pthread constructs introduces errors (see Sect. 5 for more examples). The next step in Software Restoration is, therefore, to repair the potentially broken code produced by Parallelism Elimination. In general, due to the potential complexity of this repair stage, multiple transformations may need to be applied. In order to effect repairs in our running example it is necessary to stop the first stage from overflowing its output queue. This can be achieved by merging the loops found in *Stage1*, *Stage2*, and *Stage3*, thereby resulting in a loop where the operations in stages two and three are applied to each integer produced by stage one in the same iteration that produces it. Listing 3 represents the result of this process, where *Stage1*, *Stage2*, and *Stage3* are first *lifted* into a new function, *Pipe*, and subsequently *unfolded* (i.e. unfolding in the transformational sense, à la Burstall and Darling-ton [11]). The *for*-loops exposed by this unfolding are then merged, allowing all three stages to be executed within a single iteration. This avoids the first stage overflowing its output queue, and consequently, results in a program that is sequential but semantically equivalent to the original pthreaded program.

Listing 3: Simple Pipeline Code after Code Repair

```

1 void Pipe(void** a0, void* a1, void* a2, void* a3) {
2   int my_output1, i1;
3   pipeline_stage_queues_t *myQueues1 = (pipeline_stage_queues_t *)a1;
4   queue_t *myOutputQueue1 = myQueues1->outputQueue;
5   ...
6   for (i1 = MAXDATA ; i1>=0; i1--) {
7     if (i1 > 0) { ...
8       my_output1 = i1;
9     } else {
10      my_output1 = EOS;
11    }
12    add_to_queue(myOutputQueue1, my_output1);
13
14    my_input2 = read_from_queue(myInputQueue2);
15    if (my_input2 != EOS) {
16      my_output2 = my_input2 + 1;
17      add_to_queue(myOutputQueue2, my_output2);
18    } else {
19      add_to_queue(myOutputQueue2, EOS);
20    }
21
22    my_input3 = read_from_queue(myInputQueue3);
23    if (my_input3 != EOS) {
24      my_output3 = my_input3 * 2;
25      add_to_queue(myOutputQueue3, my_output3);
26    }
27  }
28 }

```

2.3 Program Shaping

The code produced by the Code Repair stage may still contain artefacts from the original legacy parallelisation. In our running example, this includes the EOS token and intermediate queues. In other examples, custom-built representations of flat data

structures, e.g. arrays, introduced for chunking purposes may also be present. These artefacts are redundant and could hinder alternative (and possibly better) parallelisations of the code. The next step is, therefore, to eliminate residual artefacts of legacy parallelism, and to improve structure where such improvements make the code more amenable to the introduction of patterned parallelism. As in Code Repair, due to the potential complexity of this task, multiple transformations may need to be applied. Each Program Shaping refactoring results in a program that is semantically equivalent to the one it transforms. The result of the Program Shaping stage on our running example can be found in Listing 4, where both the EOS token and intermediate queues have been removed (see Sect. 4.3 for details) and the individual stages lifted back into functions.

Listing 4: Clean Sequential Simple Pipeline Code

```

1  int S1(int i1) { ... }
2
3  int S2(int my_output1) { ... }
4
5  void S3(int my_output2, queue_t* myOutputQueue3) { ... }
6
7  void Pipe(void** a0, void* a1, void* a2, void* a3) {
8      int my_output1, i1;
9      ...
10     for (i1 = MAXDATA ; i1 > 0; i1--) {
11         S3(S2(S1(i1)), myOutputQueue3);
12     }
13 }
```

2.4 Pattern Introduction

After the final pattern discovery analysis is performed and the final patterns to be introduced are identified, together with the locations in the code where this will be done, the final step is to introduce instances of parallel patterns into the now-clean sequential code. The parts of the sequential code are replaced by calls to the functions from the high-level pattern libraries such as *Intel TBB* [35] or *OpenMP* [16]. This results in final, patterned parallel code that is semantically equivalent to the starting legacy-parallel code, but with much cleaner structure and simpler, higher-level code that allows easier maintainability, adaptivity and portability.

3 Pipeline Assumptions

In this paper, we demonstrate our methodology on a subset of pipelines defined using pthreads. Whilst the refactorings described below apply only to this subset, they can be extended to facilitate a more general application of the restoration process. We assume that a valid pipeline (i.e. a pipeline that can be restored using the below refactorings) is linear, that all tasks are generated by the first stage, and that no subsequent stages will create or destroy tasks. Moreover, we assume that the first stage of the pipeline will produce an end-of-stream (EOS) token, which is

propagated through the pipeline, and results in a stage halting when it is received as input. A valid pipeline is assumed to be set up in a single function containing a sequence (or loop) of `pthread_create` calls. For each `pthread_create` call there must be a corresponding `pthread_join` call in the same function. We assume that each stage of a pipeline is run on a single thread. However, no assumptions are made regarding thread attributes, arguments passed to the function upon thread creation, or the second argument passed to `pthread_join`. Tasks are sent between stages in pipelines via intermediate queues. Each stage of a pipeline is assumed to have an input queue, q_1 , and an output queue, q_2 , given that $q_1 \neq q_2$ and the output queue of stage i is the input queue of stage $i + 1$. Pipeline stages may only communicate via these intermediate queues; for simplicity, we assume that (intermediate) stages do not access global variables. Queues are assumed to be *cyclic*. If the queue is full, and the implementation does not wait for an element to be read before adding a new element, it is assumed that the queue overwrites (unread) elements. We assume that queues use `pthread_cond_wait`, `pthread_cond_signal`, `pthread_mutex_lock`, and `pthread_mutex_unlock` operations only. These restrictions are to ensure that, following Parallelism Elimination, the behaviour of the pipeline breaks in a consistent way. Any queue update functions should not be recursive. Whilst we make no assumption on the size of queues, the interesting case is when queues are smaller than the total number of tasks passing through the pipeline. Each stage of a valid pipeline is assumed to contain a loop that retrieves input and produces a modified version of it as output.

```

1 void *Stage2(void *arg) {
2   // SETUP
3   ...
4   for (input = read_from_queue(inputQueue);
5       valid(i) || input == EOS;
6       input = read_from_queue(inputQueue)) {
7     if (input != EOS) {
8       output = f(input);
9       add_to_queue(outputQueue, output);
10    } else {
11      add_to_queue(outputQueue, EOS);
12      break;
13    }
14  }
15 }

```

The `for`-loop is assumed to read input from the stage's input queue for each iteration, where the test expression is a disjunction permitting both `EOS` token and valid inputs (for some definition of `valid`). The test expression may be simplified by treating the `EOS` as a valid input. The body of the loop is assumed to comprise an `if`-statement that checks for the `EOS` token, represented here by a preprocessor macro. When the input is not the `EOS` token, it is modified (Line 8) and added to the output queue (Line 9). Conversely, when the input is the `EOS` token, it is propagated to the next stage (Line 11) and a `break` statement used in order to halt the stage (Line 12). Should the `EOS` token be handled incorrectly and not halt the stage, due to our assumptions on the nature of the intermediate queues, the `for`-loop will wait indefinitely for input that will never arrive. Whilst we permit the occurrence of

`break` statements only in the locations specified above, we assume that no part of the pipeline has `continue` or `goto` statements.

The first stage differs in that the input is not retrieved from an input queue, but tasks are generated in a `for`-loop. For example, in `Stage1`

```

1 void *Stage1(void *arg) {
2     t1 inputs[NINPUTS];
3     ...
4     for (i = 0; i<=NINPUTS; i++) {
5         if (i < NINPUTS) {
6             output = inputs[i];
7         } else {
8             output = EOS;
9         }
10        add_to_queue(outputQueue, output);
11    }
12 }
```

an array of inputs is iterated over and each element is sent to the second stage of the pipeline. We assume that the `for`-loop in the first stage is finite and that termination is controlled by the test expression and not through a conditional statement in the body of the loop. Once all tasks have been generated, the first stage will emit an `EOS` token and no further iterations of the loop occur.

4 Restoration Transformations

The following transformations are grouped according to the stages in Sect. 2 and all apply to C programs that adhere to the assumptions in Sect. 3. In this paper, we do not attempt to prove that our transformations preserve functional behaviour; indeed, some intentionally do not. Such proofs are left to future work, where they can be properly explored in depth. Instead, and where expected, we conjecture that our transformations preserve functional behaviour when they are applied to code that both adheres to the assumptions in Sect. 3 and that meets the pre-conditions of the individual refactoring. For example, commutativity of loops in *Merge for-loops* (Sect. 4.2). It is our intention that these assumptions and pre-conditions are sufficiently strong so as to render post-conditions and dynamic correctness checks unnecessary. Proof that this property holds for our refactorings is outside of the scope of this paper and will be considered as part of future work alongside proofs of general soundness.

In addition to the following transformations, standard refactorings may also facilitate the restoration process. For instance, the transformation to *unfold* a function definition [11] is used in both Code Repair and Shaping stages; e.g. in the former, it allows loops to be merged, and in the latter, it allows the elimination of intermediate queues. The *extract method* [21] transformation can be similarly used to lift a pipeline into a self-contained function, or to lift its individual stages (back) into separate functions.

4.1 Parallelism Elimination

Parallelism Elimination comprises a single composite transformation that either removes or transforms pthread operations. As noted in Sect. 2, Parallelism Elimination does *not* guarantee that the result of the transformation will be semantically equivalent to the transformed program. It is applied to the functions that are identified as part of the valid pipeline and effects the following transformations.

- Removes `#include <pthread>` when all pthread operations within the file are found within the functions identified as part of the valid pipeline.
- Removes all pthread operations within the pipeline functions aside from calls to both `pthread_join` and `pthread_create`.
- Removes all variable declarations whose types are defined as part of the pthread library, excepting `pthread_t`. This includes global declarations when those variables occur solely within the identified pipeline functions.
- Declarations in the form `pthread_t t;` are transformed into `void* t;`. As above, in the case where such declarations are global, the variables may occur only in the pipeline functions.
- Calls to `pthread_create` of the form,

```
1 pthread_create(t,a,f,x)
```

are transformed into the form:

```
1 t = f(x);
```

Recall that Parallelism Elimination converts the type of `pthread_t` variables to `void*` variables of the same name(s), and that `pthread_create` requires that `f` returns a value of type `void*`.

- Calls to `pthread_join` are transformed according to whether the second argument is `NULL`. When the second argument is *not* `NULL`, e.g. `pthread_join(t,x)`, the join operation is transformed into the form `x = t`. Otherwise, the call to `pthread_join` is removed.
- In cases where a call to `pthread_join` or `pthread_create` forms the right-hand-side of an assignment statement, e.g.

```
1 r = pthread_join(t,x);
```

in addition to the transformation of the pthread operation, an assignment statement is inserted where the variable being assigned, `r`, is assigned the value of a successful call to the original pthread operation, here `pthread_join` and `0`. The assignments resulting from the transformation is:

```
1 r = 0;  
2 x = t;
```

- Any `for`-loop whose body contains no statements following the removal of a `pthread` operation will itself be removed.
- Any `if`-statement with a branch whose body contains no statements following the removal of a `pthread` operation will be transformed to have only the other branch, or itself removed, if no such branch exists. For instance, given the `for`-loop from the synthetic pipeline example in Listing 1,

```
8 for (i = 0; i < NRSTAGES; i++)  
9   pthread_join(workerid[i], NULL);
```

since the second argument to `pthread_join` is `NULL`, the join operation result is itself a statement, and the body of the `for`-loop contains no other operations, this `for`-loop is removed.

4.2 Code Repair

In addition to *unfolding* and *extract method* refactorings, the merging of loops is a key transformation of the Code Repair stage when restoring valid pipelines. Parallelism Elimination can result in one or more intermediate queues to overwrite elements before they can be read. Merging the queues of all pipeline stages ensures that no queue overflows. Whilst we only describe the merging of `for`-loops, a similar approach can be used to merge equivalent loop kinds.

Merge for-loops. A sequence of n `for`-loops, in the same compound statement can be merged such that the result is a single loop containing the bodies of the original loops in the same order that they appeared in the original sequence. Any statements that appear in between loops in the original code, must be commutative with respect to all preceding loops; i.e. it must be possible to swap the ordering of the statements and preceding loops without changing the behaviour of the program.

Listing 5: Intermediate Code Repair Stage for Simple Pipeline Example

```

1 void Pipe(void* a1, void* a2, void* a3) {
2   int my_output1, i1;
3   ...
4   for (i1 = MAXDATA ; i1>=0; i1--) {
5     if (i1 > 0) {
6       my_output1 = ...;
7     } else {
8       my_output1 = EOS;
9     }
10    add_to_queue(myOutputQueue1, my_output1);
11  }
12
13  int my_input2, my_output2;
14  ...
15  for (my_input2 = read_from_queue(myInputQueue2);
16       my_input2>0 || my_input2 == EOS;
17       my_input2 = read_from_queue(myInputQueue2)) {
18    if (my_input2 != EOS) {
19      ...
20    } else {
21      add_to_queue(myOutputQueue2, EOS);
22      break;
23    }
24  }
25
26  int my_input3, my_output3;
27  ...
28  for (my_input3 = read_from_queue(myInputQueue3);
29       my_input3>0 || my_input3 == EOS;
30       my_input3 = read_from_queue(myInputQueue3)) {
31    if (my_input3 != EOS) {
32      ...
33    } else {
34      break;
35    }
36  }
37 }

```

Listing 5 builds on the example in Listing 2, where the calls to *Stage1*, *Stage2*, and *Stage3* have been lifted into the function *Pipe* using *extract method* and then *unfolded*. It is possible to merge all three loops since the statements in between the loops can be safely executed prior to the first and second loops.

Listing 6: Following Merging of loops in Listing 5

```

1 void Pipe(void** a0, void* a1, void* a2, void* a3) {
2   int my_output1, i1;
3   ...
4   for (i1 = MAXDATA ; i1>=0; i1--) {
5     if (i1 > 0) {
6       my_output1 = ...;
7     } else {
8       my_output1 = EOS;
9     }
10    add_to_queue(myOutputQueue1, my_output1);
11
12    my_input2 = read_from_queue(myInputQueue2);
13    if (my_input2 != EOS) {
14      ...
15    } else {
16      add_to_queue(myOutputQueue2, EOS);
17    }
18
19    my_input3 = read_from_queue(myInputQueue3);
20    if (my_input3 != EOS) {
21      ...
22    }
23  }
24 }

```

Since we assume that new tasks are only produced by the first stage and that all subsequent stages neither generate nor destroy tasks, it follows that the number of iterations for the first loop will be equal to the number of iterations for subsequent loops in the original pipeline. Consequently, the initialisation statement, test expression, and iteration expression of the merged loop will be those of the first loop; here, $i1 = \text{MAXDATA}$, $i1 \geq 0$, and $i1--$, respectively. The bodies of the individual loops are included in the same order as the original loops themselves. Since the merged loop uses its initialisation statement, test expression, and iteration expression, the body of the first loop is included unchanged. Bodies of subsequent loops, however, are preceded by their update statement. For example, the body of the second loop is preceded by the assignment to `my_input2` on Line 12 in Listing 6 which is taken from the update statement on Line 18 in Listing 5. This ensures that the input queue for each stage is read from only when a task has been added to that queue by the preceding stage. In addition to the inclusion of the update expression, the `break` statements from the original `for`-loops are removed, leaving the propagation of the `EOS` token in all but the final stage. In the final stage of our pipeline (Lines 19–22) we remove the entire `else` branch in Listing 6 since it contains only the `break` statement on Line 34 in Listing 5. Whilst the removal of these `break` statements is not strictly necessary, since they will only be evaluated when the first stage emits an `EOS` token once all other tasks have been processed, they are removed because they are redundant now that the termination of the merged loop is controlled by the first stage of the pipeline, which will terminate after generating the `EOS` token.

4.3 Program Shaping

Program Shaping represents the broadest stage in the process and presents the programmer with the largest range of choices in terms of transformations that may

be effected. In addition to *unfolding* definitions and creating new functions via *extract method*, other standard transformations may be applied, e.g. *dead-code elimination* [28], in order to improve or simplify the structure of the code. In order to remove aspects of the code that represent optimisations enacted for the legacy parallelisation, both existing and novel transformations may be necessary. Novel transformations may include the unchunking of data, the removal intermediate, and now redundant, queues between stages, and a removal of the **EOS** token. In line with our running example, we propose transformations to remove **EOS** tokens and to remove intermediate queues.

4.4 Remove **EOS** Token from Merged Loops

Intuitively, we assume that Remove **EOS** Token from Merged Loops applies to the result of Merge Loops. Since termination of all stages is controlled solely by the merged loop, the **EOS** token is redundant and can be removed so that the resulting restored pipeline doesn't perform unnecessary work. By our assumption, at the beginning of the **for**-loop, there is an **if**-statement that determines whether the iterator being generated by the first stage is genuine output or the **EOS** token. This **if**-statement is replaced by the branch of the **if**-statement that produces genuine output. Additionally, the test expression of the merged loop is replaced by the condition of the **if**-statement being removed. This results in the merged loop executing one fewer iteration and the first stage of the (now-removed) pipeline no longer adding the **EOS** token to its output queue. For all other stages of the pipeline, we replace each stage's **if**-statement with their output branches, thus removing the **EOS** token behaviour.

Listing 7: Following application of Remove **EOS** Token from Merged Loops to Listing 6

```

1 void Pipe(void** a0, void* a1, void* a2, void* a3) {
2   int my_output1, i1;
3   ...
4   for (i1 = MAXDATA ; i1 > 0; i1--) {
5     my_output1 = ...;
6     add_to_queue(myOutputQueue1, my_output1);
7
8     my_input2 = read_from_queue(myInputQueue2);
9     my_output2 = ...;
10    add_to_queue(myOutputQueue2, my_output2);
11
12    my_input3 = read_from_queue(myInputQueue3);
13    my_output3 = ...;
14    add_to_queue(myOutputQueue3, my_output3);
15  }
16 }
```

Listing 7 gives the result of applying Remove **EOS** Token from Merged Loops to the code in Listing 6. Here, the original test expression of the merged loop, $i1 \geq 0$ (Line 4 Listing 7), has been replaced with the condition of the **if**-statement from the first stage of the pipeline, $i1 > 0$ (Line 5, Listing 7). That **if**-statement has itself been replaced by its true branch. The **if**-statements for the other two stages (Lines

13–17 and 20–22, Listing 7) have similarly been replaced by their true branches, since their false branches handle the **EOS** token.

4.5 Remove Intermediate Queues

Following the application of Merge Loops the intermediate queues become redundant. They can be removed by inspecting, matching, and transforming *read*, *write*, and *update* operations pertaining to those queues. In our recurring example we begin this process having removed the **EOS** token, and having *unfolded* both *add_to_queue* and *read_from_queue* operations. Note that the *add_to_queue* operation in the third stage is not unfolded since this is the output of the pipeline itself.

```

1 void Pipe(void** a0, void* a1, void* a2, void* a3) {
2   int my_output1, i1;
3   ...
4   for (i1 = MAXDATA ; i1 > 0; i1--) {
5     ...
6     myOutputQueue1->elements[myOutputQueue1->addTo] = my_output1;
7     myOutputQueue1->addTo = (myOutputQueue1->addTo + 1) % myOutputQueue1->capacity;
8     myOutputQueue1->nr_elements++;
9
10    my_input2 = myInputQueue2->elements[myInputQueue2->readFrom];
11    myInputQueue2->nr_elements--;
12    myInputQueue2->readFrom = (myInputQueue2->readFrom + 1) % myInputQueue2->capacity;
13    ...
14    myOutputQueue2->elements[myOutputQueue2->addTo] = my_output2;
15    myOutputQueue2->addTo = (myOutputQueue2->addTo + 1) % myOutputQueue2->capacity;
16    myOutputQueue2->nr_elements++;
17
18    my_input3 = myInputQueue3->elements[myInputQueue3->readFrom];
19    myInputQueue3->nr_elements--;
20    myInputQueue3->readFrom = (myInputQueue3->readFrom + 1) % myInputQueue3->capacity;
21    ...
22    add_to_queue(myOutputQueue3, my_output3);
23  }

```

A variable is *read* from when that variable occurs in a statement and that variable is not being *updated*; e.g. *capacity* on Line 7 above. Similarly, a variable undergoes a *write* when it is being assigned to and is not being *updated*; e.g. *elements* in the first output queue is written to on Line 6. Finally, a variable is *updated* when it occurs in a statement that is both *reading* from and *writing* to that variable; e.g. *addTo* in Line 7 above. Basic increment operators, e.g. *nr_elements++* on Line 8, are similarly considered *updates* due to their semantics. In order to transform these *read*, *write*, and *update* operations, we pair the operations in the order that they appear in the code and according to the variables they *read*, *write*, or *update*, and transform those pairs according to their composition. If two queues are semantically the same but referred to by different variables then they themselves will be considered the same during pairing; e.g. *myOutputQueue1* and *myInputQueue2* refer to the same intermediate queue, thus *myOutputQueue1->elements* and *myInputQueue2->elements* are similarly considered to be the same variable for pairing. In the above example, two cases arise:

1. *Updates* to variables that do not occur elsewhere in the code pertain to queue housekeeping operations are therefore removed. In the above code, Lines 7, 8, 11, 12, 15, 16, 19, and 20 are all removed.
2. A *write* followed by a *read* is merged into a single assignment statement s.t. the RHS of the *read* is replaced with the RHS of the *write*, and where the original *write* statement is removed. For example, in the above code, the *write* to *elements* on Line 6 and the *read* from *elements* on Line 10 can be paired (due in part to the behaviour of the queue reading the element that has just been added). Since this represents passing *my_output1* on Line 6 to *my_input2* on Line 10, it is possible to remove Line 10 and transform Line 6 into the form `my_input_2 = my_output_1`.

An unpaired *read* that is part of an *update*, e.g. *capacity* on Line 7, or a paired *write*, e.g. *addTo* on Line 6, is removed or otherwise transformed along with the *update* or paired *write* statement. Similarly, an unpaired *read* that is part of a *paired read* statement, e.g. *readFrom* on Line 10, is also transformed according to the paired *read* statement. When applied, the above transformations result in the removal of the two intermediate queues.

```

1 void Pipe(void** a0, void* a1, void* a2, void* a3) {
2   ...
3   for (i1 = MAXDATA ; i1 > 0; i1--) {
4     my_output1 = ...;
5
6     my_input2 = my_output1;
7     my_output2 = ...;
8
9     my_input3 = my_output2;
10    my_output3 = ...;
11    add_to_queue(myOutputQueue3, my_output3);
12  }

```

5 Evaluation

In this section, we present an evaluation of our restoration methodology on a number of examples of pthreaded C and C++ applications taken from a variety of domains, including image convolution, nqueens, cholesky decomposition, blacksholes, pgpry, mandelbrot and matrix multiplication¹. For each benchmark we evaluate the effectiveness of our technique using standard metrics, such as McCabe's Cyclomatic Complexity [31], lines of code and difference in runtimes between the original pthread version and the restored TBB version, using the maximum number of available cores; these results are summarised in Table 1, which also labels if each benchmark is a standard task from implementation (F) or a pipeline (P), where each stage can also be farmed. All of our execution experiments are executed 5 times and conducted on a server with Intel Xeon E5-2690 CPU with

¹ Repository of examples available at <https://github.com/Paraformance/restoration>

Table 1 Metrics for each benchmark, where F = Farm, and P = Pipeline; performance times are in seconds on a 28-core machine

Benchmark		McCabe		Lines		Performance (std dev)	
		Before	After	Before	After	Before	After
Blackscholes	F	29	29	366	393	38.5 (0.07)	39 (0.42)
MatMult	F	9	15	176	146	918.7 (24.6)	922.24 (30.42)
Mandelbrot	F	12	11	145	142	2.21 (0.01)	2.27 (0.04)
Cholesky	F	31	19	321	226	16.97 (0.07)	17.08 (0.02)
NQueens	P (2)	41	24	421	337	8.63 (0.04)	8.622 (0.27)
PGPry	P (2)	23	19	210	243	138.1 (0.23)	131 (0.10)
ImageConv	P (1)	71	29	714	280	12.85 (6.08)	5.2 (0.02)

The number of tokens for TBB pipelines are shown in parentheses

28 cores, running at 2.6 GHz with 256 GB of RAM, with the Scientific Linux 6.2 operating system.

5.1 Image Convolution

Image Convolution is a technique widely used in image processing applications for blurring, smoothing, and edge detection. We consider an instance of the image convolution from video processing applications, where we are given a list of images that are processed by applying a filter. Applying a filter to an image consists of computing a scalar product of the filter weights with the input pixels within a window surrounding each of the output pixels:

$$out(i, j) = \sum_m \sum_n in(i - n, j - m) \times filt(n, m) \quad (1)$$

Listing 8: Original Convolution with PThreads

```

1 void add_to_queue(queue_t *queue, task_t elem)
2 {
3     /* Same as in Listing 1 */
4 }
5
6 task_t read_from_queue(queue_t *queue)
7 {
8     ...
9 }
10
11 void* stage1() {
12     ..
13     while(1) {
14         t = read_from_queue(tq1);
15
16         r = workerStage1(t); /* Reads in pixels from a file into an array */
17         add_to_queue(tq2, r);
18     }
19     return NULL;
20 }
21 void* stage2() {
22     ..
23     while(1) {
24         t = read_from_queue(tq2);
25         r = workerStage2(t); /* Applies transformation to each pixel in a received array */
26         add_to_queue(tq3, r);
27     }
28     return NULL;
29 }
30
31 int main (int argc, char **argv)
32 {
33     ...
34     /* Reading in the images in the task queue tq1 */
35     ...
36     /* Create the pipeline */
37     for (int i=0; i<nw1; i++)
38         pthread_create(&workers1[i], NULL, stage1, NULL);
39     for (int i=0; i<nw2; i++)
40         pthread_create(&workers2[i], NULL, stage2, NULL);
41     ...
42     /* Wait for threads to finish execution and output results to files */
43 }

```

For the convolution example, we start off with a pthreaded version in Listing 8, with a similar structure as the other pipelined examples in this paper, and outlined in Sect. 2. After setting up the task queue for the first stage of the pipeline (e.g. by reading a list of names of files with images), the example creates the pipeline in Lines 37–40, spawning a number of worker threads for each stage of the pipeline. The pipeline stages are shown at Lines 11 and 21, respectively; each stage has a similar structure: a non-terminating `while` loop that retrieves a task from the stage’s input queue (`tq1` and `tq2` for `stage1` and `stage2`, respectively), computes the unit of work on the task item (Lines 15 and 25) and then places the result on an output queue (Lines 16 and 26). Functions `add_to_queue` and `read_from_queue` put a task in an output queue and read a task from an input queue, respectively, in a thread safe manner. The code for `add_to_queue` was shown in Listing 1.

The *first step* to restoration is to remove the threading code; this is a fairly straightforward process, but results in an executable that no-longer terminates. This is due to the fact that there is no termination condition of the `while` loops within the stages. A simple repair for this step is to add a termination token, `EOS`. When no more tasks are available on the original input queue, `EOS` is sent through the pipeline, terminating the stages (Listing 9).

Listing 9: Convolution, Repaired with a Termination Token

```

1  if ((int)(task_t)t == EOS) {
2      puttask(tq2, (task_t2 *)EOS);
3      break;
4  }
```

The next step is to perform *program shaping* which goes through various steps, including *unfolding* the various calls to `gettask` and `puttask` in the stages, *merging* the stages together, and finally *removing* the intermediate queue between the two stages (leaving the input and output queue; see Listing 10).

Listing 10: Stages merged, unfolded and intermediate queue removed

```

1  /* Unfolded gettask function, reutrning t1 as an input task to stage 1 */
2  . . .
3  r1 = workerStage1(t1);
4  r2 = workerStage2(r1);
5  /* Unfolded puttask function that puts r2 into queue tq2 */
6  tq2->elements[tq2->addTo] = r2;
7  tq2->addTo = (tq2->addTo + 1) % tq2->capacity;
8  tq2->nr_elements ++
```

The final step in the shaping process is to arrive at the code shown in Listing 11 where we remove the input and output queues completely, and transform the program into a simple function composition; the function composition has been *unfolded* into the original `for` loop (Line 37–40 from Listing 8), and the loops merged into a single loop.

Listing 11: Convolution Shaped

```

1  for (int i=0; i<NIMGS; i++) {
2      workerStage2(workerStage1(i));
3  }
```

Finally, the fully shaped program from Listing 11 can be parallelised using a structured pattern approach. Here we use TBB, to define a pipeline, using C++ classes, as shown in Listing 12.

Listing 12: Convolution Restored with TBB

```

1  tbb::parallel_pipeline(
2      ntoken, tbb::make_filter<void, task_t2*>(tbb::filter::serial, Stage1(NIMGS) )
3      & tbb::make_filter<task_t2*, int>(tbb::filter::parallel, Stage2() )
```

5.2 Discussion

Table 1 shows the summary of our results for all the benchmarks. For all benchmarks we see comparable results in the McCabe metrics, where the TBB version gives a better result, apart from Blackscholes, where the complexity is equal, and Matrix Multiplication, where the complexity actually increases. This is most likely because both of these benchmarks are simple farms, and the TBB logic actually introduces some complexity over simply calling `pthread_create` multiple times. The number of lines of code for the TBB version is mostly comparable, with most benchmarks showing a decrease in lines of code. Blackscholes shows a slight increase in LOC, most likely, again, due to the slight increase in code logic for TBB versus the pthread version. In terms of performance, again, the TBB versions are mostly comparable, with the exception of a few cases. For convolution, the TBB version performs $2.4\times$ faster, due to the pthreading version introducing extra overheads in the locking code; Blackscholes also performs very slightly worse, by 1.28%.

5.3 Limitations

The methodology presented in this paper is still preliminary and intended to be a foundational step to a more advanced (semi-automated) restoration technique that applies to many different kinds of patterns and applications. As such, we have noted a number of general limitations to the approach, which we document here.

As the transformations are currently applied manually, with the lack of a (semi-automated) tool, mistakes can easily happen as part of that manual transformation. The functionality of the transformed program has to be tested and checked with the original version at each step to ensure validity. However, if a (non-specialised) developer was undertaking this task, without any real direction or structure to parallelise their code, the manual process would undoubtedly take much longer and be a much more error-prone process. Our methodology aims to give a guide to programmers so that they can restore their programs, and act as a basis for future implementation efforts.

Furthermore, many code bases (including PARSEC) are large and diffuse, containing many files and many hundreds or thousands of lines of code. Applying such a manual technique to large and complex code is a very time consuming process, limiting our choice in use-cases and examples. Knowledge of the use-case is also needed in order to transform it correctly, taking into account subtle structural and algorithmic properties of the underlying source code. Often the most time consuming part of the process is understanding the underlying algorithmic properties, and not in the transformations themselves.

Our methodology only currently applies to farms and pipelines. A preliminary study of other applications with instances of different types of patterns, such as BSP, reduce, and map, seem to indicate that many of the restoration steps will be different, resulting in different transformation steps depending on the pattern instance to be restored. However, many of the transformations that we present in this paper to eliminate pthreaded code can still be applied. We intend to pursue this direction as an avenue for future work.

6 Related Work

The concept of a *systematic*, or *structured* approach to *software restoration* has, to our knowledge, been largely previously unexplored. A concept that is probably most related to *software restoration* is that of *reverse engineering*, which is a technique used to retrieve high-level requirements from existing sequential code [14, 15]. Yu et al. [37] proposed a technique that attempts to use refactoring to try and recover requirements goal models from legacy code. However, the work only targets sequential code and only capture high-level information that is not useful for parallel restoration. Refactoring has roots in Burstall and Darlington's fold/unfold system [11], and has been applied to a wide range of applications as an approach to program transformation [32], with refactoring tools a feature of popular IDEs including, *i.a.*, Eclipse [36] and Visual Studio [33]. Previous work on parallelisation *via* refactoring has primarily focussed on the introduction and manipulation of parallel pattern libraries in C++ [9, 27] and Erlang [5, 7]. Another approach has been the automated introduction of annotations in the form of C++ attributes [19]. Dig proposed an approach to parallelise loops in Java [20], but did not use high-level algorithmic skeletons. Aldinucci and Danelutto proposed an approach to convert between skeleton configurations and could be used to introduce parallelism, but where the sequential program must also be defined using (sequential) skeletons [1]. Thompson et al. [29] proposed an approach to refactor sequential Erlang programs into concurrent versions, using program slicing to guide the refactoring process. However, their approach was not focussed on parallel performance, and did not use restoration or parallel patterns. High-level parallel patterns, sometimes known as *algorithmic skeletons* offer high-level abstraction over low-level concurrency methods [3, 22]. A range of pattern/skeleton implementations have been developed for a number of programming languages; these include: RPL [27]; Feldspar [4]; FastFlow [2]; Microsoft's Pattern Parallel Library [13]; and Intel's Threading Building Blocks (TBB) library [35]. Since patterns are well-defined, rewrites can be used to automatically explore the space of equivalent patterns, e.g. optimising for performance [24, 30] or generating optimised code as part of a domain-specific language (DSL) [23]. Moreover, since patterns are architecture-agnostic, patterns have been similarly implemented for multiple architectures [26, 34]. SPar [25] is a C++ internal domain-specific-language (DSL) for supporting the development of classic stream parallel applications targeting a FastFlow [2] backend. SPar allows the programmer to annotate C++ code with high-level annotations, relating to the streaming and staging properties of the underlying algorithm. A compiler then transforms the SPar annotations into FastFlow code which can then be executed. We believe SPar would be a viable framework to support the introduction of patterns in our methodology. The P3ARSEC benchmark suite [17], offer patterned implementations of the Parsec benchmark suite using pattern-based parallel programming, in contrast to the standard Parsec implementations of pthreads, OpenMP and TBB. P3ARSEC instead uses a FastFlow backend, and makes use of several parallel patterns, including pipeline, farm, map and reduce.

7 Conclusions

In this paper, we have introduced a software restoration methodology for converting legacy-parallel applications into structured parallel code using parallel patterns. This ensures portability, maintainability and adaptivity of parallel code while maintaining, and sometimes even increasing, performance. We also presented transformations to eliminate ad-hoc pthread parallelism from legacy-parallel code, transformations that repair the code from bugs introduced by the elimination step, and , shape the code in order to patternise it. Furthermore, we evaluated our software restoration methodology on a number of realistic benchmarks and use-cases, demonstrating benefit in terms of gained performance, increased adaptivity, portability and maintainability. One of the limitations of the work is the fact that the transformations must be applied manually in their current form. It would be possible to take the transformations presented here and implement them into a semi-automatic refactoring tool, such as ParaFormance², which is a semi-automatic tool for transforming sequential C and C++ applications into parallel patterned versions. The refactorings presented in this paper would be implemented in terms of the pre-conditions and transformation rules of an Abstract Syntax Tree, both of which are described in this paper. Other similar refactorings to introduce patterns into C and C++ applications have previously been described in [7, 8, 9, 27] and the restoration refactorings would take a similar direction. We will extend our methodology to take into account many other types of parallel patterns, including reduce, maps, stencil, etc. It's possible that we will see similarities and overlapping ideas in the restoration of different types of patterns. We will catalogue these commonalities in a future paper. We also intend on giving proofs of general soundness that our refactorings conform to their specification and do not change the program's functional behaviour. Proving soundness of refactorings is a complicated and challenging issue, but recent work in the use of Dependent Types [6] has allowed us to capture the soundness properties as part of the refactoring implementations. We envisage extending this to prove soundness of concurrency refactorings in a similar way. Lastly, we would like to explore optimisation of the restored application in other domains, such as energy optimisation. Once the code has been refactored into a structured, maintainable code base, it is possible to then apply different kinds of optimisations for energy, memory usage, etc. in a similar way to parallelisation.

Acknowledgements This work was generously supported by the EU Horizon 2020 project, *TeamPlay* (<https://www.teamplay-h2020.eu>), Grant Number 779882, and UK EPSRC *Discovery*, Grant Number EP/P020631/1.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative

² <http://www.paraformance.com/>

Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Aldinucci, M., Danelutto, M.: Stream Parallel Skeleton Optimization. In: PDCS, pp. 955–962 (1999)
2. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: High-level and efficient streaming on multicore, chap. 13, pp. 261–280 (2017). [10.1002/9781119332015.ch13](https://doi.org/10.1002/9781119332015.ch13)
3. Asanovic, K., Bodík, R., Demmel, J., Keaveny, T., Keutzer, K., Kubitowicz, J., Morgan, N., Patterson, D.A., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.A.: A view of the parallel computing landscape. *Commun. ACM* **52**(10), 56–67 (2009)
4. Axelsson, E., Claessen, K., Sheeran, M., Svenningsson, J., Engdal, D., Persson, A.: The design and implementation of feldspar—an embedded language for digital signal processing. In: The 22nd International Symposium on Implementation and Application of Functional Languages, IFL 2010, *Lecture Notes in Computer Science*, vol. 6647, pp. 121–136. Springer (2010)
5. Barwell, A.D., Brown, C., Hammond, K., Turek, W., Byrski, A.: Using program shaping and algorithmic skeletons to parallelise an evolutionary multi-agent system in Erlang. *J. Comput. Inform.* **35**(4), 792–818 (2016)
6. Barwell, A.D., Brown, C., Sarkar, S.: Proof-carrying refactorings: sound refactoring for haskell via dependent types. In: International Conference on Functional Programming (ICFP), pp. 1–25. *In Submission* (2021)
7. Brown, C., Danelutto, M., Hammond, K., Kilpatrick, P., Elliott, A.: Cost-directed refactoring for parallel Erlang programs. *Int. J. Parallel Programm.* **42**(4), 564–582 (2014)
8. Brown, C., Janjic, V., Barwell, A., García, J.D., MacKenzie, K.: Refactoring GrPPI: generic refactoring for generic parallelism in C++. *Int. J. Parallel Programm.* **48**(4), 603–625 (2020). <https://doi.org/10.1007/s10766-020-00667-x>
9. Brown, C., Janjic, V., Hammond, K., Schöner, H., Idrees, K., Glass, C.W.: Agricultural reform: more efficient farming using advanced parallel refactoring tools. In: Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP, pp. 36–43. IEEE Computer Society (2014)
10. Brown, C., Janjic, V., Barwell, A., Thomson, J, Lozano, R. C., Cole, M, Franke, B, Garcia-Sanchez, J.D., Astorga, D. D. R., MacKenzie, K.: A hybrid approach to parallel pattern discovery in C++. In: 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 187–191 (2020)
11. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. *J. ACM* **24**(1), 44–67 (1977)
12. Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley Longman Publishing Co. Inc., USA (1997)
13. Campbell, C., Miller, A.: A parallel programming with microsoft visual C++: design patterns for decomposition and coordination on multicore architectures, 1st edn. Microsoft Press (2011)
14. Cook, J.E., Wolf, A.L.: Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol.* **7**(3), 215–249 (1998)
15. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from java source code. In: Proceedings of the 22nd International Conference on Software Engineering ICSE, pp. 439–448. ACM (2000)
16. Dagum, L., Menon, R.: Openmp: an industry-standard API for shared-memory programming. *IEEE J. Comput. Sci. Eng.* **5**(1), 46–55 (1998)

17. De Sensi, D., De Matteis, T., Torquati, M., Mencagli, G., Danelutto, M.: Bringing parallel patterns out of the corner: the P3ARSEC benchmark suite. *ACM Trans. Architect. Code Optim.* (2017). <https://doi.org/10.1145/3132710>
18. del Rio Astorga, D., Dolz, M.F., Fernández, J., García, J.D.: A generic parallel pattern interface for stream and data processing. *Concurr. Comput. Practice Exp.* **29**(24), e4175 (2017). <https://doi.org/10.1002/cpe.4175>
19. del Rio Astorga, D., Dolz, M.F., Sánchez, L.M., García, J.D., Danelutto, M., Torquati, M.: Finding parallel patterns through static analysis in C++ applications. *Int. J. High-Performance Comput. (IJHPCA)* **32**(6), 779–88 (2018)
20. Dig, D.: A refactoring approach to parallelism. *IEEE Softw.* **28**(1), 17–22 (2011)
21. Fowler, M.: Refactoring—improving the design of existing code. Addison Wesley Object Technology Series. Addison-Wesley, Boston (1999)
22. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Softw. Pract. Exp.* **40**(12), 1135–1160 (2010)
23. Gorlatch, S.: Domain-specific optimizations of composed parallel components. In: Domain-specific program generation, *Lecture Notes in Computer Science*, vol. 3016, pp. 274–290. Springer (2003)
24. Gorlatch, S., Wedler, C., Lengauer, C.: Optimization rules for programming with collective operations. In: Proceedings of the 13th International Parallel Processing Symposium on Parallel and Distributed Computing IPPS/SPDP, pp. 492–499. IEEE Computer Society (1999)
25. Griebler, D., Danelutto, M., Torquati, M., Fernandes, L.G.: Spar: A DSL for high-level and productive stream parallelism. *Parallel Processing Letters* **27**(1), 1740005:1–1740005:20 (2017). <https://doi.org/10.1142/S0129626417400059>
26. Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., Dubach, C.: High performance stencil code generation with lift. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO, pp. 100–112. ACM (2018)
27. Janjic, V., Brown, C., Mackenzie, K., Hammond, K., Danelutto, M., Aldinucci, M., García, J.D.: RPL: A domain-specific language for designing and implementing parallel C++ applications. In: Proceedings of the 24th International Conference on Parallel, Distributed and Network-based Processing, PDP, pp. 288–295. IEEE Computer Society (2016)
28. Kennedy, K.: A survey of data flow analysis techniques. In: Muchnick, S.S., Jones, N.D. (eds.) *Program Flow Analysis*, pp. 5–54. Prentice-Hall, Englewood Cliffs (1981)
29. Li, H., Thompson, S.J.: Safe concurrency introduction through slicing. In: Proceedings of the Workshop on Partial Evaluation and Program Manipulation, PEPM, pp. 103–113. ACM (2015)
30. Matsuzaki, K., Kakehi, K., Iwasaki, H., Hu, Z., Akashi, Y.: A fusion-embedded skeleton library. In: Euro-Par, *Lecture Notes in Computer Science*, vol. 3149, pp. 644–653. Springer (2004)
31. McCabe, T.J.: A complexity measure. *IEEE Trans. Softw. Eng.* **2**(4), 308–320 (1976)
32. Mens, T., Tourwé, T.: A survey of software refactoring. *IEEE Trans. Softw. Eng.* **30**(2), 126–139 (2004)
33. Microsoft: Visual Studio IDE: <https://visualstudio.microsoft.com/vs/>. (2019)
34. Reyes, R., Lomüller, V.: SYCL: Single-source C++ accelerator programming. In: Proceedings of the International Conference on Parallel Computing (PARCO), *Advances in Parallel Computing*, vol. 27, pp. 673–682. IOS Press (2015)
35. TBB (Intel Threading Building Blocks). In: *Encyclopedia of Parallel Computing*, p. 2029. Springer (2011)
36. The Eclipse Foundation.: Eclipse—an open development platform (2009). <http://www.eclipse.org>
37. Yu, Y., Wang, Y., Mylopoulos, J., Liaskos, S., Lapouchnian, A., do Prado Leite, J.C.S.: Reverse engineering goal models from legacy code. In: 13th IEEE International Conference on Requirements Engineering (RE 2005), 29 August–2 September 2005, Paris, France, pp. 363–372. IEEE Computer Society (2005)