13th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2020)
Porto, Portugal
July 9-10, 2020

Restoration of Legacy Parallelism in C and C++ Applications

Vladimir Janjic · Christopher Brown ·
Adam D. Barwell ·

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 *pthreads* 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 e.g. *pthreads* into structured patterned equivalents. We demonstrate our restoration technique on a number of benchmarks, allowing the introduction of patterned parallelism in the resulting code; we record improvements in cyclomatic complexity and speedups.

Keywords Parallel patterns, restoration, pthreads, program transformation, code analysis

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 (Intel

V. Janjic
School of Science and Engineering, University of Dundee, UK.
E-mail: vjanjic001@dundee.ac.uk

C. Brown, A. Barwell
School of Computer Science, University of St Andrews, UK.
E-mail: cmb21,adb23@st-andrews.ac.uk

TBB [1], Microsoft PPL, etc.) 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. Then instantiating the identified pattern at the identified location in the code, after possibly restructuring the code to accommodate the parallelism.

Sequential code 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* [10]. 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 the 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. Common examples include *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, $f_1, f_2, ..., f_m$ are applied to a stream of independent inputs, $x_1, ..., x_n$ where the output of f_i becomes the input to f_{i+1}; the parallelism arises from executing $f_{i+1}(f_i(...f_1(x_k)...))$ in parallel with $f_i(f_{i-1}(...f_1(x_{k+1})...))$.

In this paper, we present a new methodology for the restoration of legacy-parallel code into an equivalent *patterned* form, through 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. 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;
2. we present a new set of *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;
3. 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.

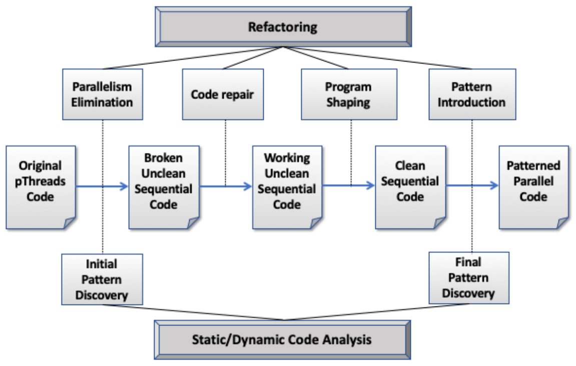

Fig. 1: Software Restoration Process

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:

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

The input to the Software Restoration process is a legacy-parallel C/C++ code that is based on some low-level parallelism library, such as pthreads, and the output is the semantically-equivalent code 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 focus on the TBB library as our target 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 Figure 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

```
int main(int argc, char *argv[]) {
  ...
  // create the workers, then wait for them to finish
  pthread_create(&workerid[0], &attr, Stage1, (void *)&stage_queues[0]);
  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
19    pipeline_stage_queues_t *myQueues = (pipeline_stage_queues_t *)arg;
20    queue_t *myOutputQueue = myQueues->outputQueue;
21    queue_t *myInputQueue = myQueues->inputQueue;
22
23    do {
24      my_input = read_from_queue(myInputQueue);
25      if (my_input > 0)
26        my_output = my_input + 1;
27      else // 0 is a terminating token. Pass on if received.
28        my_output = 0;
29      add_to_queue(myOutputQueue, my_output);
30    } while (my_input>0);
31
32    return NULL;
33  }
34
35  void add_to_queue(queue_t *queue, int elem)
36  {
37    pthread_mutex_lock(&queue->queue_lock);
38    // If the queue is full, wait until something reads from it before adding a new element
39    if (queue->nr_elements == queue->capacity)
40      pthread_cond_wait(&queue->queue_cond_read,&queue->queue_lock);
41    queue->elements[queue->addTo] = elem;
42    queue->addTo = (queue->addTo + 1) % queue->capacity;
43    queue->nr_elements++;
44    pthread_cond_signal(&queue->queue_cond_write);
45    pthread_mutex_unlock(&queue->queue_lock);
46  }
```

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 the first stage has an output queue, and stages two and three have both an input and an output queue. After creation, the `main` function waits for the threads to finish their work (Lines 8–9) before continuing. In Lines 14–32, 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 35–46) is shown here, since `read_from_queue` is similar. Both functions use one mutex lock and two conditional variables. The conditional variables 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 39–40). 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 at line 40). After this conditional variable is signalled, the thread adds the element to the queue, 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 in Line 44) and releases the queue lock (Line 45) before returning.

Parallelism Elimination. The initial code analysis 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 pipeline created in Lines 4–6, with the pipeline stages being the functions: `Stage1`, `Stage2`, and `Stage3`. Following pattern discovery, the first code transformation step is applied, where pthread operations and primitives are either removed or transformed so as to eliminate parallelism. In Listing 1, this impacts the `main` and `add_to_queue` functions; Listing 2 gives the results of parallelism elimination on both functions.

Listing 2: Simple Pipeline Code with Parallelism Removed

```c
int main(int argc, char *argv[]) {
  ...
  // Calls to pthread_create are converted to function calls.
  Stage1((void *)&stage_queues[0]);
  Stage2((void *)&stage_queues[1]);
  Stage3((void *)&stage_queues[2]);

  // The loop containing pthread_join is removed.
  ...
}

void add_to_queue(queue_t *queue, int elem) {
  // All mutex and conditional variable operations are removed.
  queue->elements[queue->addTo] = elem;
  queue->addTo = (queue->addTo + 1) % queue->capacity;
  queue->nr_elements++;
}
```

Whilst all pthread operations have been removed or transformed, and the program is now sequential, the Parallelism Elimination stage *does not guarantee that a program's semantics are preserved.* Accordingly, as in our running example, errors may be introduced. Here, `Stage1` contains a `do`-loop that adds items to 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. Ultimately, the semantics and output of the program produced by the Parallelism Elimination stage is not the same as the original pthreaded program; the code must therefore be *repaired.*

Code Repair. As observed in the previous step, Parallelism Elimination might result in code that is broken and, hence, not semantically equivalent to the original legacy-parallel code. Our example is just one of many instances in which merely removing pthread constructs actually breaks the code (see Section 4 for more examples). The next step in Software Restoration is, therefore, to repair the potentially broken code produced by Parallelism Elimination. Due to the potential complexity of this repair stage, multiple transformations may need to be applied.

In order to repair the broken pipeline 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 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. The result of this process can be found below in Listing 3.

Listing 3: Simple Pipeline Code after Code Repair

```c
void Pipe(void* a1, void* a2, void* a3) {
  // STAGE ONE
  int my_output_1, i_1 = MAXDATA;

  pipeline_stage_queues_t *myQueues_1 = (pipeline_stage_queues_t *)a1;
  queue_t *myOutputQueue_1 = myQueues_1->outputQueue;

  // STAGE TWO
  int my_input_2, my_output_2;
  ...
  do {
    // STAGE ONE
    if (i_1 >= -1) { ... }

    // STAGE TWO
    my_input_2 = read_from_queue(myInputQueue_2);
    if (my_input_2 >= 0) {
      if (my_input_2 > 0)
        my_output_2 = my_input_2 + 1;
      else
        my_output_2 = 0;
      add_to_queue(myOutputQueue_2, my_output_2);
    }

    // STAGE THREE
    my_input_3 = read_from_queue(myInputQueue_3);
    if (my_input_3 >= 0) { ... }
  } while (i_1 >= 0 || my_input_2 > 0 || my_input_3 > 0);
}
```

Here, the calls to `Stage1`, `Stage2`, and `Stage3` in `main` are first *lifted* into a new function, `Pipe`. Each of those calls are then *unfolded* in order to expose the `do`-loops that they contain. These loops 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.

Program Shaping. Despite correcting those errors introduced during Parallelism Removal, the code produced by the Code Repair stage may still contain artefacts from the original legacy parallelisation. In our running example, such artefacts include the queues between the stages. In other examples artefacts can include custom-built representations of flat data structures, such as arrays, perhaps introduced for chunking purposes. 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. In our running example, we remove the now redundant queues in between the stages, the result of which can be found in Listing 4.

Listing 4: Clean Sequential Simple Pipeline Code

```
 1  struct PipeStruct {
 2    // Input to pipeline
 3    int* i_1;
 4    // Output of pipeline
 5    queue_t* myOutputQueue_3;
 6    // Inter-stage temporary variables, used in loop-condition.
 7    int my_output_1;
 8    int my_output_2;
 9  };
10
11  PipeStruct S1(PipeStruct arg) { ... }
12
13  PipeStruct S2(PipeStruct arg) {
14    if (arg.my_output_1 >= 0) {
15      if (arg.my_output_1 > 0) arg.my_output_2 = arg.my_output_1 + 1;
16      else arg.my_output_2 = 0;
17    }
18    return arg;
19  }
20
21  PipeStruct S3(PipeStruct arg) { ... }
22
23  void Pipe(void* a1, void* a2, void* a3) {
24    // STAGE ONE
25    int my_output_1, i_1 = MAXDATA;
26    ...
27    do {
28      PipeStruct arg = PipeStruct {&i_1, myOutputQueue_3, &my_output_1, &my_output_2};
29      PipeStruct r = S3(S2(S1(arg)));
30      my_output_1 = r.my_output_1;
31      my_output_2 = r.my_output_2;
32    } while (i_1 >= 0 || my_output_1 > 0 || my_output_2 > 0);
33  }
```

Here, calls to `add_to_queue` and `read_from_queue` are first *unfolded*, allowing the specific read and write statements that represent the passing of data between stages to be matched and ultimately simplified to a single assignment statement between stages. The stages are then *lifted* into the functions `S1` (Line 11), `S2` (Lines 13–19), and `S3` (Line 21), and a `struct` (Lines 1–9) generated to ensure that each stage is a single-input single-output function. The function composition on Line 32 is now in a form where pattern-based parallelism can be simply introduced, perhaps again by refactoring.

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* [1] or *OpenMP* [14]. 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 Restoration Transformations

We propose a series of program transformations to facilitate the restoration of C programs that have been previously parallelised using low-level pthread parallelism techniques. The following transformations are grouped according to the stages in Section 2 in which they are used. In addition to the following, standard transformations may also facilitate the restoration process. For instance, the transformation to *unfold* a function definition [9] 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* [17] transformation can be similarly used to lift a pipeline into a self-contained function, or to lift its individual stages (back) into separate functions.

3.1 Parallelism Elimination

Parallelism Elimination comprises a single composite transformation that either removes or transforms pthread operations. As noted in Section 2, Parallelism Elimination, and by extension this transformation, does *not* guarantee that the result of the transformation will be semantically equivalent to the transformed program. Parallelism Elimination effects the following transformations.

– Removes `#include <pthread>`.
– Removes all pthread operations 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` declarations.
– Declarations in the form `pthread_t t;` are transformed into `void* t;`.
– Calls to `pthread_create` of the form,

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

are transformed into the form:

```
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, when the second argument is `NULL`, the call to `pthread_create` is removed.
- In cases where a call to `pthread_join` or `pthread_create` forms the right-hand-side of an assignment statement, e.g.

```
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` in the above example, is assigned the value of a successful call to the original pthread operation, here `pthread_join` and `0`. In the above example, the code resulting from the transformation is:

```
r = 0;
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.

3.2 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 pipelines. In order to avoid the overheads involved with thread creation, individual stages of a pipeline may loop until a termination token or condition is met. Merging the loops across pipeline stages from which pthreads have been eliminated using the transformations in Section 3.1 can be necessary to avoid overflowing any buffers or queues in between pipeline stages, thus restoring the original semantic behaviour of the code. Here, we describe only the merging of `do`-loops, but a similar approach can be used to merge, e.g., `for`-loops.

Merge `do`-loops. A sequence of n `do`-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 source code. We note that any statements that appear in between loops in the original code, must be commutative with respect to any 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.

To illustrate this transformation we use the below code, derived from the synthetic simple pipeline example in Section 2.

Listing 5: Intermediate Code Repair Stage for Simple Pipeline Example

```c
void Pipe(void* a1, void* a2, void* a3) {
  // STAGE ONE
  int my_output_1, i_1 = MAXDATA;
  ...
  do {
    ...
  } while(i_1>=0);

  // STAGE TWO
  int my_input_2, my_output_2;
  ...
  do {
    my_input_2 = read_from_queue(myInputQueue_2);
    ...
  } while (my_input_2>0);

  // STAGE THREE
  int my_input_3, my_output_3;
  ...
  do {
    my_input_3 = read_from_queue(myInputQueue_3);
    ...
  } while (my_input_3>0);
}
```

This represents the example following the Parallelism Elimination stage (Listing 2), and where the calls to `Stage1`, `Stage2`, and `Stage3` have been lifted into the function `Pipe` using *extract method* and then *unfolded*. Since the statements in between the above loops consist solely of declarations and assignment statements and can be safely executed prior to the first and second loops, it is possible to merge these loops.

Listing 6: Following Merging of loops in Listing 5)

```c
void Pipe(void* a1, void* a2, void* a3) {
  int my_output_1, i_1 = MAXDATA;
  ...
  do {
    // STAGE ONE
    if (i_1 >= -1) {
      ...
    }

    // STAGE TWO
    my_input_2 = read_from_queue(myInputQueue_2);
    if (my_input_2 >= 0) {
      ...
    }

    // STAGE THREE
    my_input_3 = read_from_queue(myInputQueue_3);
    if (my_input_3 >= 0) {
      ...
    }
  } while (i_1 >=0 || my_input_2 > 0 || my_input_3 > 0);
}
```

The bounding condition of the merged loop is formed of the disjunction of the conditions of the original loops. Similarly, the body of the merged loop comprises the bodies of the original loops wrapped in `if`-statements. The condition of one of these `if`-statements is the *weakened* condition of the respective original `do`-loop; e.g. the condition `my_input_2>0` above is weakened to `my_input_2>=0`. This weakening is necessary, since the body of a `do`-loop is executed before the bounding condition is checked. Moreover, because the loop body is guaranteed to execute once, it is possible that a variable used in the bounding condition may be declared outside of the loop, but only initialised within it. For example, in the second and third stages above, neither `my_input_2` nor `my_input_3` are initialised before the assignment *inside* the body of their respective loops (Lines 15 & 23, Listing 5). In order to merge these loops such that the second and third stages will execute, we move the aforementioned assignment statements outside of the introduced `if`-statement around the loop body (Lines 12 & 18, Listing 6). Such assignment statements can only be lifted out of the body if they themselves depend upon variables already assigned outside of the loop. Variables are renamed in the bodies of the loops as necessary.

3.3 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* [23], 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 tupling (and potential localisation) of arguments to present transformations that introduce algebraic skeletons with a simple composition of single-parameter functions. In line with our running example, we propose transformations to remove intermediate queues, and to merge arguments.

Remove Intermediate Queues. In a pthreaded pipeline, passing the result of a stage to the next can involve intermediary queues. Once the parallelism from the pipeline has been eliminated and the code repaired, so too can these queues be removed. We remove these intermediate queues by inspecting, matching, and transforming *read*, *write*, and *update* operations pertaining to those queues. In our recurring example we begin this process following the Code Repair stage, and having *unfolded* `add_to_queue` and `read_from_queue` operations for *intermediate queues only*; note that the output queue operation on Line 35 has not been unfolded.

```
void Pipe(void* a1, void* a2, void* a3) {
 2   int my_output_1, i_1 = MAXDATA;
 3
 4   pipeline_stage_queues_t *myQueues_1 = (pipeline_stage_queues_t *)a1;
 5   queue_t *myOutputQueue_1 = myQueues_1->outputQueue;
 6   ...
 7   do {
 8     // STAGE ONE
 9     if (i_1 >= 0) {
10       ...
11       myOutputQueue_1->elements[myOutputQueue_1->addTo] = my_output_1;
12       myOutputQueue_1->addTo = (myOutputQueue_1->addTo + 1) % myOutputQueue_1->capacity;
13       myOutputQueue_1->nr_elements++;
14     }
15
16     // STAGE TWO
17     my_input_2 = myInputQueue_2->elements[myInputQueue_2->readFrom];
18     myInputQueue_2->nr_elements--;
19     myInputQueue_2->readFrom = (myInputQueue_2->readFrom + 1) % myInputQueue_2->capacity;
20
21     if (my_input_2 >= 0) {
22       ...
23       myOutputQueue_2->elements[myOutputQueue_2->addTo] = my_output_2;
24       myOutputQueue_2->addTo = (myOutputQueue_2->addTo + 1) % myOutputQueue_2->capacity;
25       myOutputQueue_2->nr_elements++;
26     }
27
28     // STAGE THREE
29     my_input_3 = myInputQueue_3->elements[myInputQueue_3->readFrom];
30     myInputQueue_3->nr_elements--;
31     myInputQueue_3->readFrom = (myInputQueue_3->readFrom + 1) % myInputQueue_3->capacity;
32
33     if (my_input_3 >= 0) {
34       ...
35       add_to_queue(myOutputQueue_3, my_output_3);
36     }
37   } while (i_1 >= 0 || my_input_2 > 0 || my_input_3 > 0);
38 }
```

A variable is *read* from when that variable occurs in a statement and that variable is not being *updated*; e.g. `capacity` on Line 12 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 13. 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 12 above. Basic increment operators, e.g. `nr_elements++` on Line 13, 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. `myOutputQueue_1` and `myInputQueue_2` refer to the same intermediate queue, thus `myOutputQueue_1->elements` and `myInputQueue_2->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 12, 13, 18, 19, 24, 25, 30, and 31 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 11 and the *read* from `elements` on Line 17 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_output_1` on Line 11 to `my_input_2` on Line 17, it is possible to remove Line 17 and transform Line 11 into the form `my_input_2 = my_output_1`.

An unpaired *read* that is part of an *update*, e.g. `capacity` on Line 12, or a paired *write*, e.g. `addTo` on Line 11, 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 17, is also transformed according to the paired *read* statement. When applied, the above transformations result in the removal of the two intermediate queues.

```
void Pipe(void* a1, void* a2, void* a3) {
  int my_output_1, i_1 = MAXDATA;

  pipeline_stage_queues_t *myQueues_1 = (pipeline_stage_queues_t *)a1;
  queue_t *myOutputQueue_1 = myQueues_1->outputQueue;
  ...
  do {
    // STAGE ONE
    if (i_1 >= 0) {
      ...
      my_input_2 = my_output_1;
    }
    // STAGE TWO
    if (my_input_2 >= 0) {
      ...
      my_input_3 = my_output_2;
    }
    // STAGE THREE
    if (my_input_3 >= 0) {
      ...
      add_to_queue(myOutputQueue_3, my_output_3);
    }
  } while (i_1 >= 0 || my_input_2 > 0 || my_input_3 > 0);
}
```

Merge Arguments. During restoration, the stages of a pipeline may each be represented by a single function. Since the majority of patterned pipeline implementations expect pipeline stages to take a single argument, i.e. usually the result of the preceding stage, it may be necessary to tuple the parameters of each stage. This can be done (semi-)automatically. A `struct` can be synthesised across each of the functions representing stages in the pipeline, with each function transformed to both return and take the synthesised `struct` as its argument. In our running pipeline example, following the removal of the intermediate queues, we have three stages represented by the functions `S1`, `S2`, and `S3`, respectively. `S1` takes a pointer to an integer and returns an integer value; `S2` takes and returns an integer value; and `S3` takes an integer value and a pointer to its output queue and returns nothing.

```
int S1(int* i_1) {
  int my_output_1;
  if (*i_1 >= 0) {
    my_output_1 = *i_1;
    *i_1 = *i_1-1;
  }
  return my_output_1;
}

int S2(int my_output_1) {
  ...
  return my_output_2
}

void S3(int my_output_2, queue_t* myOutputQueue_3) {
  ...
}

void Pipe(void* a1, void* a2, void* a3) {
  ...
  do {
    my_output_1 = S1(&i_1);
    my_output_2 = S2(my_output_1);
    S3(my_output_2, myOutputQueue_3);
  } while (i_1 >= 0 || my_output_1 > 0 || my_output_2 > 0);
}
```

Here, we observe that the result of the first two stages are used in the merged loop bounding condition and must therefore be propagated through the stages when converted to composition form. This is achieved by synthesising a new `struct`, `PipeStruct`, that contains all those variables used in the condition statement, and both the input and output to the pipeline.

```
struct PipeStruct {
  // INPUTS
  int* i_1;
  // OUTPUTS
  queue_t* myOutputQueue_3;
  // USED IN LOOP CONDITION
  int my_output_1;
  int my_output_2;
};

PipeStruct S1(PipeStruct arg) {
  ...
  return arg;
}

PipeStruct S2(PipeStruct arg) {
  ...
  return arg;
}

PipeStruct S3(PipeStruct arg) {
  ...
  return arg;
}

void Pipe(void* a1, void* a2, void* a3) {
  ...
  do {
    PipeStruct arg = PipeStruct {&i_1, myOutputQueue_3};
    PipeStruct r = S3(S2(S1(arg)));
    my_output_1 = r.my_output_1;
    my_output_2 = r.my_output_2;
  } while (i_1 >= 0 || my_output_1 > 0 || my_output_2 > 0);
}
```

Here, in order to introduce a TBB pipeline, we keep the input and output to the pipeline as pointers. The variables my_output_1 and my_output_2 are only used as part of the condition and are not outputs of the pipeline, as such they are stored by their value. In this particular example, it is possible to remove the second and third disjunctions and therefore remove my_ouptut_1 and my_ouptut_2 from PipeStruct, and thus further simplify the pipeline, but this is left to another Shaping transformation or sequence of transformations.

4 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, blackscholes, pgpry, mandelbrot and matrix multiplication. For each benchmark we evaluate the effectiveness of our technique using standard metrics, such as McCabe's Cyclomatic Complexity [26], 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, where each stage can also be farmed (P). All of our execution experiments are conducted on a server with Intel Xeon E5-2690 CPU with 28 cores, running at 2.6 GHz with 256 GB of RAM, with the Scientific Linux 6.2 operating system.

4.1 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 rocessed 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 7: Original Convolution with PThreads

```
void add_to_queue(queue_t *queue, task_t elem)
{
  /* Same as in Listing 1 */
}

task_t read_from_queue(queue_t *queue)
{
  ...
}

```

```
11  void* stage1() {
12    ..
13    while(1) {
14      t = read_from_queue(tq1);
15      r = workerStage1(t); /* Reads in pixels from a file into an array */
16      add_to_queue(tq2, r);
17    }
18    return NULL;
19  }
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 7, with a similar structure as the other pipelined examples in this paper, and outlined in Section 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, respectivelly, 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`, which threads through the pipeline computation when no more tasks are on the original input queue amd terminates the stages when received (Listing 8).

Listing 8: Convolution, Repaired with a Termination Token

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

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 9).

Listing 9: Stages merged, unfolded and intermediate queue removed

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

The final step in the shaping process is to arrive at the code shown in Listing 10, 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 7), and the loops merged into a single loop.

Listing 10: Convolution Shaped

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

Finally, the fully shaped program from Listing 10 can be parallelised using a structured pattern approach. Here we use TBB, to define a pipeline, using C++ classes, as shown in Listing 11.

Listing 11: Convolution Restored with TBB

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

4.2 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,

Benchmark		McCabe		Lines		Performance	
		Before	After	Before	After	Before	After
Blackscholes	F	29	29	366	393	38.5	39
Matrix Multiplication	F	9	15	176	146	909.4	896.5
Mandelbrot	F	12	11	145	142	2.21	2.28
NQueens	P	41	24	421	337	8.63	8.622
Cholesky Decomposition	P	31	19	321	226	16.97	17.08
PGPry	P	23	19	210	243	138.1	131
Image Convolution	P	71	29	714	280	12.85	5.2

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

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 2.4x faster, due to the pthreading version introducing extra overheads in the locking code; Blackscholes also performs very slightly worse, by 0.5 seconds.

5 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 [12,13]. Yu et al. [31] 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 [9], and has been applied to a wide range of applications as an approach to program transformation [27], with refactoring tools a feature of popular IDEs including, *i.a.*, Eclipse [16] and Visual Studio [28]. Previous work on parallelisation *via* refactoring has primarily focussed on the introduction and manipulation of parallel pattern libraries in C++ [8,22] and Erlang [7,6]. Another approach has been the automated introduction of annotations in the form of C++ attributes [30]. Dig proposed an approach to parallel loops in Java [15], 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 [2]. Thompson et al. [24] 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 [4,18]. A range of pattern/skeleton implementations have been developed for a number of programming languages; these include: RPL [22]; Feldspar [5]; FastFlow [3]; Microsoft's Pattern Parallel Library [11]; and Intel's Threading Building Blocks (TBB) library [1]. Since patterns are well-defined, rewrites can be used to automatically explore the space of equivalent patterns, e.g. optimising for performance [25,20] or generating optimised code as part of a DSL [19]. Moreover, since patterns are architecture-agnostic, patterns have been similarly implemented for multiple architectures [21,29].

6 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 out software restoration methodology on a number of realistic benchmarks and use-cases, demonstrating benefit in terms of gained performance, increased adaptivity, portability and maintainability.

References

1. TBB (intel threading building blocks). In: Encyclopedia of Parallel Computing, p. 2029. Springer (2011)
2. Aldinucci, M., Danelutto, M.: Stream parallel skeleton optimization. In: PDCS, pp. 955–962 (1999)
3. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: High-Level and Efficient Streaming on Multicore, chap. 13, pp. 261–280 (2017). DOI 10.1002/9781119332015.ch13
4. Asanovic, K., Bodík, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, 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)
5. 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: IFL, *Lecture Notes in Computer Science*, vol. 6647, pp. 121–136. Springer (2010)
6. 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. Computing and Informatics **35**(4), 792–818 (2016)
7. Brown, C., Danelutto, M., Hammond, K., Kilpatrick, P., Elliott, A.: Cost-directed refactoring for parallel erlang programs. International Journal of Parallel Programming **42**(4), 564–582 (2014)
8. Brown, C., Janjic, V., Hammond, K., Schöner, H., Idrees, K., Glass, C.W.: Agricultural reform: More efficient farming using advanced parallel refactoring tools. In: PDP, pp. 36–43. IEEE Computer Society (2014)
9. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. J. ACM **24**(1), 44–67 (1977)
10. Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley Longman Publishing Co., Inc., USA (1997)

11. 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)
12. 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)
13. 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: ICSE, pp. 439–448. ACM (2000)
14. Dagum, L., Menon, R.: Openmp: An industry-standard api for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 4655 (1998)
15. Dig, D.: A refactoring approach to parallelism. IEEE Software **28**(1), 17–22 (2011)
16. Foundation, E.: Eclipse - an Open Development Platform (2009). `http://www.eclipse.org`
17. Fowler, M.: Refactoring - Improving the Design of Existing Code. Addison Wesley object technology series. Addison-Wesley (1999)
18. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. Softw., Pract. Exper. **40**(12), 1135–1160 (2010)
19. 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)
20. Gorlatch, S., Wedler, C., Lengauer, C.: Optimization rules for programming with collective operations. In: IPPS/SPDP, pp. 492–499. IEEE Computer Society (1999)
21. Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., Dubach, C.: High performance stencil code generation with lift. In: CGO, pp. 100–112. ACM (2018)
22. 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: PDP, pp. 288–295. IEEE Computer Society (2016)
23. Kennedy, K.: A Survey of Data Flow Analysis Techniques, p. 554 (1981)
24. Li, H., Thompson, S.J.: Safe concurrency introduction through slicing. In: PEPM, pp. 103–113. ACM (2015)
25. 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)
26. McCabe, T.J.: A complexity measure. IEEE Trans. Software Eng. **2**(4), 308–320 (1976)
27. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. Software Eng. **30**(2), 126–139 (2004)
28. Microsoft: Visual Studio IDE (2019). `https://visualstudio.microsoft.com/vs/`
29. Reyes, R., Lomüller, V.: SYCL: single-source C++ accelerator programming. In: PARCO, *Advances in Parallel Computing*, vol. 27, pp. 673–682. IOS Press (2015)
30. 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. IJHPCA **32**(6) (2018)
31. Yu, Y., Wang, Y., Mylopoulos, J., Liaskos, S., Lapouchnian, A., do Prado Leite, J.C.S.: Reverse engineering goal models from legacy code. In: RE, pp. 363–372. IEEE Computer Society (2005)