Proceedings Paper:
Botincan, Matko, Dodds, Michael David orcid.org/0000-0002-4439-0130 and Jagannathan, Suresh (2012) Resource-sensitive synchronization inference by abduction. In: POPL '12 Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, New York, pp. 309-322.

https://doi.org/10.1145/2103656.2103694

Resource-Sensitive Synchronization Inference by Abduction

Matko Botinčan
University of Cambridge
matko.botincan@cl.cam.ac.uk

Mike Dodds
University of Cambridge
mike.dodds@cl.cam.ac.uk

Suresh Jagannathan
Purdue University
suresh@cs.purdue.edu

Abstract

We present an analysis which takes as its input a sequential program, augmented with annotations indicating potential parallelization opportunities, and a sequential proof, written in separation logic, and produces a correctly-synchronized parallelized program and proof of that program. Unlike previous work, ours is not an independence analysis; we insert synchronization constructs to preserve relevant dependencies found in the sequential program that may otherwise be violated by a naïve translation. Separation logic allows us to parallelize fine-grained patterns of resource-usage, moving beyond straightforward points-to analysis.

Our analysis works by using the sequential proof to discover dependencies between different parts of the program. It leverages these discovered dependencies to guide the insertion of synchronization primitives into the parallelized program, and to ensure that the resulting parallelized program satisfies the same specification as the original sequential program, and exhibits the same sequential behaviour. Our analysis is built using frame inference and abduction, two techniques supported by an increasing number of separation logic tools.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Program Verification—Correctness proofs; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures

General Terms Languages, Theory, Verification

Keywords Separation Logic, Abduction, Frame Inference, Deterministic Parallelism

1. Introduction

Automatically verifying the safety properties of shared-memory concurrent programs remains a challenging problem. To be useful in practice, proof tools must (a) explore a potentially large number of interleavings, and (b) construct precise flow- and path-sensitive abstractions of a shared heap.

Just as significantly, verification complexity is often at odds with the straightforward intentions of the programmer. Low-level concurrency control abstractions such as locks obfuscate higher-level notions such as atomicity and linearizability, likely to be exploited by the programmer when writing programs. While attempts to incorporate these notions directly into programs have met with some success—for example, in software transactional memory [21], there remains a substantial burden on the programmer to ensure programs can be partitioned into easily-understood atomic and thread-local sections. Automated proof engines used to verify program correctness must also assimilate this knowledge to ensure proof-search focusses on relevant behaviour (e.g., serializability), and eschews irrelevant details (e.g., thread-local state) [18]; the alternative, expressing such distinctions in programmer-supplied specifications, is too often onerous to be acceptable in practice. In any case, the inherent complexity of dealing with concurrency, both for the programmer and for the verifier, unfortunately remains.

In many cases concurrency is an optimization, rather than intrinsic to the behaviour of the program. That is, a concurrent programs is often intended to achieve the same effect of a simpler sequential counterpart. Consequently, an attractive alternative to *constructing* a concurrent programming is to automatically *synthesise* one. In this approach, the programmer writes a sequential program; the program is automatically transformed into a concurrent one exhibiting the same behaviour [3–5, 17][1]. In this context, the verification problem becomes significantly more tractable. Understanding and verifying the concurrent program reduces to first verifying the sequential program, and second, verifying the parallelising transformation.

We propose a program analysis and transformation that yields a parallelized program, guided by a safety proof of the original sequential program. To guide the transformation, we assume that the programmer annotates points in the sequential program where concurrency can be profitably exploited, *without* supplying any additional concurrency control or synchronization. The result of the transformation is a concurrent program with corresponding behaviour, and a safety proof for the concurrent program; in this sense, parallelized programs are *verified by construction*. While our analysis is driven by a safety proof, parallelization protects all behaviours, not just those specified in the proof. Thus we can apply our approach to complex algorithms without needing to verify total functional correctness.

Our transformation ensures that behaviour is preserved by requiring that the concurrent program respects sequential data-dependencies. In other words, the way threads access and modify shared resources never results in behaviour that would not be possible under sequential evaluation. To enforce these dependencies, the transformation injects *barriers*, signalling operations that regulate when threads are allowed to read or write shared state. These barriers can be viewed as resource transfer operations which acquire and relinquish access to shared resources such as shared-memory data structures and regions when necessary.

Our analysis is built on separation logic [32]. By tracking how resources are demanded and consumed within a separation logic

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.

[1] This approach is commonly known as *deterministic parallelism*, although our approach does not, in fact, require that the input sequential program is deterministic.

proof, we can synthesize barriers to precisely control access to these resources. Our approach critically relies on frame inference and abduction [11], two techniques that generate fine-grained information on when resources are necessary and when they are redundant. This information enables optimizations that depend on deep structural properties of the resource—for example, we can split a linked list into dynamically-sized segments and transmit portions between threads piece-by-piece.

Our analysis thus enforces sequential order over visible behaviours, while allowing parallelization where it has no effect on behaviour. Insofar as our technique safely transforms a sequential program into a concurrent one, it can also be viewed as a kind of proof-directed compiler optimization. Notably, our transformation is *not* based on an independence analysis—our focus is not to identify when two potentially concurrent code regions do not interfere; instead, our analysis injects sufficient synchronization to ensure relevant sequential dependencies are preserved, even when regions share state and thus potentially interfere.

Contributions.

1. We present an automated technique to synthesize a parallel program given a partially-specified sequential program augmented with annotations indicating computations that are candidates for parallelization. The provided specifications are used to define relevant loop invariants.
2. We leverage abduction and frame inference to define a path- and context-sensitive program analysis capable of identifying per program-point resources that are both *redundant*—these are resources that would no longer be used by the thread executing this computation; and, *needed*—these are resources that would be required by any thread that executes this computation, but which is not known to have been released at this point.
3. We use information from the above analysis to inject *grant* and *allowed* barriers into the original program; their semantics enable resource transfer from redundant to needed resource points. The analysis also constructs a safety proof in separation logic which validates the correctness of the barrier injections.
4. We prove that our transformation is specification-preserving: the parallelized program is guaranteed to satisfy the same specification as the sequential program. Moreover, for terminating programs that do not dispose memory, we also show that the transformed program preserves the *behaviour* of the original. Complete proofs and other supporting material can be found in an associated technical report [8].

Paper structure. An overview of the approach and motivating examples are given in §2. Issues that arise in dealing with loops and recursive datatypes (such as lists) are discussed in §3. The analysis and transformation are formalized in §4 and §5. Behaviour preservation results are discussed in §6. Observations about the interplay between analysis precision and the predicates available in defining specifications are given in §7. §8 discusses related work.

2. Overview

The objective of our analysis is to take a sequential program (with annotations indicating where concurrency may be exploited), and to produce a *parallelized* program that conforms to the same specification. To do this, we insert *synchronization barriers* enforcing those sequential dependencies that can affect the observable behaviour of the program.

Our analysis uses *separation logic* to capture and manipulate these dependencies. The programmer must provide a proof of safety for the sequential program. This proof is used to drive the parallelization process, allowing our analysis to calculate precisely which resources can be accessed in parallel, and which must be accessed sequentially, in order to preserve salient sequential dependencies.

```
global *x, *y;
void main() {
  local i, n;
  n = nondet();
  x = alloc();
  y = alloc();
  pfor(i=1;i++;i<n){
    f(i);
  }
}

void f(i){
  local v=*x;
  if (v>=i) {
    g(y, v);
  }
  else {
    g(x, 0);
  }
}

void g(*p, v){
  *p = v;
}
```

Figure 1. A simple parallel program that operates over shared locations `x` and `y`. The number of concurrent iterations performed is chosen nondeterministically to represent the act of receiving values from some unknown process or user interaction.

2.1 Parallelization and Parallel-for

We assume that the portions of a program that can be parallelized will be written as *parallel-for* loops, `pfor`. The intended semantics of a parallel-for loop is that every iteration will run in parallel, but that the behaviour will be identical to running them in sequence; this semantics provides a simple but useful form of data parallelism in which concurrently executing computations may nonetheless have access to shared state.

For example, consider the program shown in Fig. 1. How can we parallelize this program without introducing any unwanted new behaviours, *i.e.*, behaviours not possible under sequential execution? Naïvely, we might simply run every iteration of the `pfor` in parallel, without synchronization. That is:

```
f(1) || f(2) || ...
```

In some cases, this parallelization is good, and introduces no new observable behaviours (e.g., if `v` is greater than `n`). But, under others, the second call to `f()` may write to a memory location that is subsequently read by the first call to `f()`, violating intended sequential ordering. For example, consider the case when `v` is initially `1` and `n` is 2; sequential execution would produce a final result in which the locations pointed to by `x` and `y` resp. are 0 and 1, while executing the second iteration before the first would yield a result in which the location pointed to by `y` remains undefined.

To be sure that no new observable behaviour is introduced, we must ensure that:

- no iteration can read from a location that was already written to by a later iteration.
- no iteration can write to a location that was already read from or written to by a later iteration.

In order to enforce this behaviour, we must introduce a mechanism to force later iterations to wait for earlier ones.

Note, crucially, that later iterations *need not always* wait for earlier ones—synchronization is only needed when reads and writes to a particular memory location could result in out-of-order behaviour. We want to enforce *salient* dependencies, while allowing beneficial race-free concurrency.

2.2 Dependency-Enforcing Barriers

Our analysis inserts barriers into the parallelized program requiring logically later iterations to wait for earlier ones. We introduce `grant()`, a barrier that signals to subsequent iterations that a resource can safely be accessed, and `wait()`, a barrier that blocks until the associated resource becomes available from preceding iterations [17].

```
f1(i){              ga1(*p, v){
 local v=*x;         *p=v;
 if (v>=i) {         grant(wy);
   grant(wx);       }
   ga1(y, v);
 }
 else {             gb1(*p, v){
   grant(wy);        *p=v;
   gb1(x, 0)         grant(wx);
 }                  }
}
```

```
f2(i){              ga2(*p, v){
 wait(wx);           wait(wy);
 local v=*x;         *p=v;
 if (v>=i) {        }
   ga2(y, v);
 }
 else {             gb2(*p, v){
   gb2(x, 0);        *p=v;
   wait(wy);        }
 }
}
```

Figure 2. Parallelization of `f()`. Only synchronization barriers between the first and second iterations are shown.

How can we use these barriers to enforce sequential dependencies in the example program? The exact pattern of parallelization depends on the granularity of our parallelization analysis. In the best case, there are no dependencies between iterations (e.g., each iteration operates on a different portion of a data structure). In this case, we need no barriers, and all iterations run independently. However, our example program shares the locations `x` and `y`, meaning barriers are required. In the worst case, each iteration of the `pfor` begins with a call to `wait()` and ends with a call to `grant()`—this would enforce sequential ordering on the iterations.

A better (although still relatively simple) parallelization is shown in Fig. 2. To simplify the presentation, we only show the synchronization barriers between the first and second iterations of `pfor`. We write $\texttt{f}_1$ for the first iteration, and $\texttt{f}_2$ for the second. (Our full analysis generates a single version of `f()` with sufficient barriers to run an arbitrary number of iterations in parallel. See §2.6 for a description.)

Two channels, `wx` and `wy`, are used to signal between $\texttt{f}_1$ and $\texttt{f}_2$—`wx` (resp. `wy`) is used to signal that the later thread can read and write to the heap location referred to by `x` (resp. `y`). The function `g()` has different resource requirements depending on its calling context. Consequently, we have specialized it to embed appropriate barriers depending on its context.

Our analysis inserts barriers so that an iteration of the parallel-for only calls `grant()` on a signal when the associated resource will no longer be accessed. Similarly, it ensures that an iteration calls `wait()` to acquire the resource before the resource is accessed.

How does our analysis generate these synchronization barriers? The example we have given here is simple, but in general our analysis must cope with complex dynamically-allocated resources such as linked lists. It must deal with the partial ownership (for example, read-access), and with manipulating portions of a dynamic structure (for example, just the head of a linked list). We therefore need a means to express complex (dynamic) patterns of resource management and transfer.

To achieve this, our analysis assumes a sequential *proof*, written in separation logic, rather than just an undecorated program. This proof need not capture full functional correctness—it is sufficient just to prove memory-safety. We exploit the dependencies expressed within this proof to determine the resources that are needed at a program point, and when they can be released. Our analysis inserts barriers to enforce the sequential dependancies represented in this proof. As our proof system is sound, these dependencies faithfully represent those in the original sequential program.

2.3 Resources, Separation and Dependency

At the heart of our analysis is automated reasoning using concurrent separation logic [28, 32]. Separation logic is a Hoare-style program logic for reasoning about mutable, allocatable resources. A separation logic proof of a program C establishes a specification

$$\{P\}\ C\ \{Q\}$$

This can be be read as saying: "*the program C, if run in a resource satisfying P, will not fault, and will give a resource satisfying Q if it terminates*". In general, a *resource* can be anything for which ownership can be partitioned ('separated') between different threads [10]. In practice, resources are most often heap-allocated structures such as lists, trees, locks, etc, where separation corresponds to disjointness between underlying heap addresses.

An essential feature of separation logic is that specifications are *tight*. This means that all of the resources accessed by a program C must be described in its precondition P, or acquired through explicit resource transfer. No other resources will be accessed or affected by C when executed from a resource satisfying P. The tight interpretation is essential for our analysis. Suppose we prove a specification for a program (or portion thereof); resources outside of the precondition cannot affect the program's behaviour, and consequently can be safely transferred to other threads.

One result of this tight interpretation is the *frame rule*, which allows a small specification to be embedded into a larger context.

$$\frac{\{P\}\ C\ \{Q\}}{\{P * F\}\ C\ \{Q * F\}}\ \textsc{Frame}$$

Here $*$ is the *separating conjunction*. An assertion $P * F$ is satisfied if P and F hold, and the portions of the state they denote do not overlap.

The concurrent counterpart of the frame rule is the *parallel rule*. This allows two threads that access non-overlapping resources to run in parallel, without affecting each other's behaviour.

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \| C_2\ \{Q_1 * Q_2\}}\ \textsc{Parallel}$$

The parallel rule enforces the absence of data-races between C_1 and C_2. The two threads can consequently only affect each other's behaviour by communicating through race-free synchronization mechanisms, such as locks or barriers.

Automatic inference. Automated reasoning in separation logic revolves around two kinds of inference questions. The first, *frame inference*, calculates the portion of a formula S which is 'left over', once another formula P has been satisfied. We call this remainder $F_?$ the *frame*, and write the inference question as follows:

$$S \ \vdash\ P * [F_?]$$

(Throughout the paper, we use square brackets to indicate the portion of the entailment that is to be computed.)

The second kind of inference question, *abduction*, calculates the 'missing' formula that must be combined with a formula S in order to satisfy some other formula P. We call this formula $A_?$ the *antiframe*, and write the inference question as follows:

$$S * [A_?] \ \vdash\ P$$

Frame inference and abduction form the basis of symbolic execution in separation logic [2, 11]. Intuitively, frame inference lets us reason forwards, while abduction lets us reason backwards. Frame inference can work out which portions of the state will (and will not) be affected by the command, while abduction can work out what extra resources are necessary to execute the command safely.

Suppose we have a symbolic state represented by an assertion S, and a command c with specification $\{P\}\ c\ \{Q\}$. If we calculate the frame in $S \ \vdash\ P * [F_?]$, the symbolic state after executing c must be $Q * F_?$. The tight interpretation of triples is necessary for this kind of reasoning. Because a specification must describe all the resources affected by a thread, any resources in the frame must be unaffected. Conversely, if we calculate the antiframe $S * [A_?] \vdash P$,

it must be the case that before executing c, we must first acquire the additional resource $A_?$ (as well as S).

Redundant and needed resources. The tight interpretation means that a proof in separation logic expresses all resources modified by each individual command. Because separation logic ensures race-free behaviour, a proof also expresses all the resources that can affect the observable behaviour of each individual command in the program. Our parallelization analysis uses this to calculate resources that are redundant and those that are needed during the execution of the program.

We use frame inference to determine *redundant resources*—resources that will not be accessed in a particular portion of the program, and which can thus be safely transferred to other threads. Conversely, we use abduction to determine *needed resources*—resources that must be held for a particular portion of the program to complete, and which must thus be acquired before the program can proceed safely.

In our analysis, we generally calculate the redundant resource from the current program point to the start of the subsequent iteration. This resource can be transferred to the subsequent iteration using a `grant()` barrier. We generally calculate the needed resource from the end of the previous iteration to the current program point. This resource must be acquired from the previous iteration using a `wait()` barrier before execution can proceed further.

Note that these two resources need not be disjoint. A resource may be used early in an iteration, in which case it will be both needed from the previous iteration to the current point, and redundant from the current point up to the next iteration. Note also that redundant and needed resources need not cover the entire state—some resource described in the proof may never be accessed or modified by the program.

2.4 Algorithm Overview

The user supplies a sequential program which makes use of the `pfor` parallel-for construct, as well as a sequential proof written in separation logic establishing the memory-safety of the program. The high-level structure of our algorithm is then as follows:

1. The *resource usage analysis* phase uses abduction and frame inference to discover redundant and needed resources for different portions of the program.
2. The *parallelising transformation* phase consists of two parts:
   (a) The *resource matching* phase matches redundant resources in one iteration of the parallel-`for` with needed resources in the subsequent iteration.
   (b) The *barrier insertion* phase converts the sequential program into a concurrent program and inserts `grant()` and `wait()` barriers consistent with the discovered resource-transfer.

The result is a parallelized program, and its separation logic proof.

2.5 Resource Usage Analysis

The resource usage analysis takes as its input the program, and a separation logic proof for the program for some specification. Consider once again the functions `f()` and `g()`, which we introduced at the start of this section. Let us assume the programmer proves the following specification:

$$\{\mathtt{x} \mapsto x * \mathtt{y} \mapsto y\}$$
$$\mathtt{f(i)}$$
$$\{(x \geq \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto x) \vee (\mathtt{x} \mapsto 0 * \mathtt{y} \mapsto y)\}$$

Fig. 3 shows an outline proof of this specification.

For each program point, the resource usage analysis computes a pair of assertions: the *needed* resource and the *redundant* resource

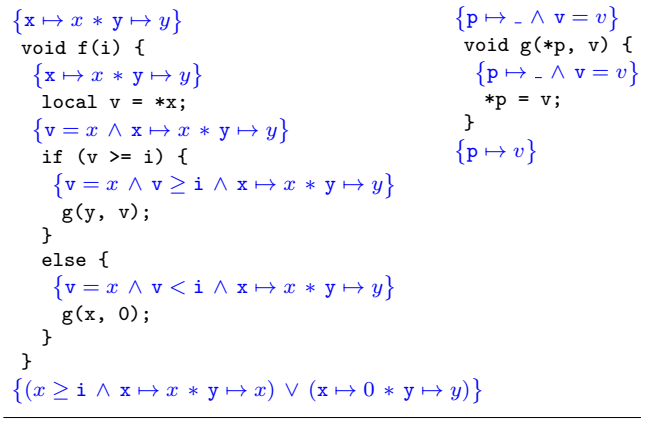

Figure 3. Proofs of functions `f()` and `g()`.

(see Fig. 4). The needed resource is the portion of the sequential precondition required to reach the program point without faulting; the redundant resource is the portion of the resource held in the sequential proof that is unnecessary for reaching the end of the function. In other words, needed assertions at program point p denote resources that are needed from the beginning of the function up to (but not including) p; redundant assertions denote resources that are not required from p to the end of the function.

For example, the redundant resource established within the false branch of the conditional in `f` asserts that a thread executing this function will no longer require access to `y` at this point whenever `v < i`. The needed resource here asserts that the thread reaching this point still requires access to `x`, provided `v < i`.

2.6 Generalizing to n Threads

Our analysis can handle an unbounded number of loop iterations running in parallel. Fig. 5 shows the final parallelization of our example, generalized to deal with an arbitrary number of iterations.

A `pfor` is translated to a sequential `for` loop, in which each iteration forks a new copy of the parallelized loop body. Resources are transferred between the threads in order of thread-creation. That is, the nth iteration of the `pfor` acquires resources from the logically-preceding $(n-1)$th iteration, and releases resources to the logically $(n+1)$th iteration. This ordering is implemented through shared channels—a thread shares with its predecessor a set of channels for receiving resources, and with its successor a set of channels for sending resources.

The number of threads—and so, the required number of channels—is potentially decided at run-time. Consequently, channels are dynamically allocated in the main `for`-loop using the `newchan()` operation [17]. Each iteration creates a set of new channels, and passes the prior and new set to the forked thread. The parallelized version of `f` (Fig. 5) now takes four channel arguments, a pair of each for `x` and `y`. The prior channels are used for resource transfer with the logically-preceding thread (here `wx`, `wy`), and the new channels are used to communicate resource transfer with the logically-following thread (here, `wx'`, `wy'`).

Our analysis generates two versions of function `g`, specialized to the two contexts in which it is invoked. Function `ga` executes when `v >= i`. Here, the executing thread needs write access to `y`, which it acquires by calling `wait(wyp)`. Function `gb` needs write access to `x`, but it runs in a context where the resource `x` has already been acquired. Consequently, it need not call `wait`. Both versions release a resource to the following thread using `grant` before returning.

```
void f(i) {
 n:{emp}
 r:{x < i ∧ y ↦ y}
  local v = *x;
 n:{x ↦ x}
 r:{x ≥ i ∧ x ↦ x) ∨ (x < i ∧ y ↦ y)}
  if (v >= i) {
   n:{x ≥ i ∧ x ↦ x}
   r:{x ≥ i ∧ x ↦ x}
    g(y, v); ⟶ void g(*p, v) {
                 n:{x ≥ i ∧ x ↦ x}
                 r:{x ≥ i ∧ x ↦ x}
                  *p = v;
                 n:{x ≥ i ∧ x ↦ x ∗ y ↦ y}
                 r:{x ↦ x ∗ y ↦ y}
                }
  }
  else {
   n:{x < i ∧ x ↦ x}
   r:{x < i ∧ y ↦ y}
    g(x, 0); ⟶ void g(*p, v) {
                 n:{x < i ∧ x ↦ x}
                 r:{x < i ∧ y ↦ y}
                  *p = v;
                 n:{x < i ∧ x ↦ x}
                 r:{x ↦ x ∗ y ↦ y}
                }
  }
 n:{(x ≥ i ∧ x ↦ x ∗ y ↦ y) ∨ (x < i ∧ x ↦ x)}
 r:{x ↦ x ∗ y ↦ y}
}
```

Figure 4. Redundant and needed resources in the function `f()`. Function `g()` is inlined, as its resource usage depends on the calling context.

3. Loops and Recursive Data Structures

Up to this point, we have dealt with channels transferring single heap locations injected into straight-line code. Our analysis can in fact handle more complicated heap-allocated data-structures such as linked lists, and control-flow constructs such as loops. To illustrate, consider the example program `sum_head(n)` shown (and verified) in Fig. 6. `sum_head(n)` sums the values stored in the first `n` elements of the list, then zeros the rest of the list.

An important feature of our approach is that the input safety proof need not specify all the relevant sequential properties; all sequential dependencies are enforced. Thus we can see `sum_head` as representative of the class of algorithms that traverse the head of a list, then mutate the tail. With minor modifications, the same proof pattern would cover insertion of a node into a list, or sorting the tail of the list.

Consider two calls to this function within a parallel-`for`:

```
void main(){ pfor(i=1;i++;i<3){ n=nondet(); sum_head(n) }
```

As above, for the sake of exposition we specialize the first and second iterations of the `pfor` loop, calling them $\texttt{sum\_head}_1$ and $\texttt{sum\_head}_2$. Our analysis generalizes to n iterations exactly as described in §2.6. The barriers injected into `sum_head` must enforce the following properties:

- $\texttt{sum\_head}_2$ must not write to a list node until $\texttt{sum\_head}_1$ has finished both writing to and reading from it. Consequently, if `n2` $<$ `n1`, $\texttt{sum\_head}_2$ must wait for $\texttt{sum\_head}_1$ to finish summing the values stored in the first `n1` nodes before writing to them.
- $\texttt{sum\_head}_2$ must not read from a list node until $\texttt{sum\_head}_1$ has finished writing to them. Consequently, $\texttt{sum\_head}_2$ must wait for $\texttt{sum\_head}_1$ to finish zeroing the value stored in a node before reading from the node.

```
void main() {
  local i, n = nondet();
  x = alloc(); y = alloc();
  chan wx', wy';
  chan wx = newchan(); chan wy = newchan();
  grant(wx); grant(wy);
  for (i = 0; i++; i < n) {
    wx' = newchan(); wy' = newchan();
    fork(f(i, wx, wy, wx', wy'));
    wx = wx'; wy = wy';
  }
  wait(wx); wait(wy);
}

f(i, wxp, wyp, wx, wy){
  wait(wxp);
  local v = *x;
  if (v >= i) {
    grant(wx);
    ga(y, v, wy);
  }
  else {
    wait(wyp);
    grant(wy);
    gb(x, 0, wx);
  } }

ga(*p, v, wyp, wy){
  wait(wyp);
  *p = v;
  grant(wy);
}

gb(*p, v, wx){
  *p = v;
  grant(wx);
}
```

Figure 5. Parallelization of our running example generalized to deal with n threads. Channels are used to signal availability of resources from one thread to another.

This example is substantially more subtle than our earlier one, requiring more than a simple points-to analysis, because the list is not divided into statically-apparent reachable segments. In the worst case, a `wait()` at the start of $\texttt{sum\_head}_2$`()` and a `grant()` at the end of $\texttt{sum\_head}_1$`()` enforces sequential order. However, by reasoning about the structure of the manipulated list using the safety proof given in Fig. 6, our approach can do considerably better.

The parallelization synthesized by our analysis is shown in Fig. 7 (the general n-thread version is given in the technical report [8]). This parallelization divides the list into two segments, consisting of the portions read and written to by $\texttt{sum\_head}_1$`()`. A shared heap-location `xpr` stores the starting address of the portion written by $\texttt{sum\_head}_1$`()`. The thread $\texttt{sum\_head}_2$ uses `xpr` to control when to access the second segment of the list[2]. We discuss how the analysis materializes `xpr` below.

Handling dynamic structures means dealing with allocation and disposal. Fortunately, separation logic handles both straightforwardly. Updates to the data-structure and object allocation are by assumption reflected in the invariants of the original sequential proof. Thus updates and allocations are also reflected in the invariants which our analysis constructs to represent the contents of channels. However, introducing allocation and disposal affects the behaviour-preservation result discussed in §6; this result ensures

[2] For simplicity, here we write $\texttt{sum\_head}_2$ with `wait` controlled by conditionals. The actual transformation performs syntactic loop-splitting to avoid the need to modify the loop invariant. Details are given in §5.2.

```
lnode *hd;
{hd ↦ h * lseg(h, nil)}
sum_head(n) {
  local i, sum, x;
  int i = 1;
  int sum = 0;
  x = *hd;
  {hd ↦ x * lseg(x, nil)}
  // entailment on lseg predicate
  {hd ↦ h * lseg(h, x) * lseg(x, nil)}
  while(x!=nil && i!=n) {
    {hd ↦ h * lseg(h, x) * ∃v, y. x.val ↦ v * x.nxt ↦ y * lseg(y, nil)}
    sum += x.val;
    i++;
    x = x.nxt;
  }
  {hd ↦ h * lseg(h, x) * lseg(x, nil)}
  while(x!=nil) {
    {hd ↦ h * lseg(h, x) * ∃v, y. x.val ↦ v * x.nxt ↦ y * lseg(y, nil)}
    x.val = 0;
    x = x.nxt;
  }
}
{hd ↦ h * lseg(h, nil)}
```

Figure 6. Separation logic proof of `sum_head`, a list-manipulating program whose automated parallelization requires reasoning over complex assertions and predicates.

```
sum_head1(n){
  i = 1;
  sum = 0;
  x = *hd;
  while(x!=nil && i!=n){
    sum+=x.val;
    i++;
    x = x.nxt;
  }
  *xpr = x;
  grant(i1);
  while(x!=nil){
    x.val = 0;
    x = x.nxt;
  }
  grant(i2);
}
```

```
sum_head2(n){
  i = 1;
  sum = 0;
  wait(i1);
  x = *hd;
  while(x!=nil && i!=n){
    if(x==*xpr) wait(i2);
    sum +=x.val;
    i++;
    x = x.nxt;
  }
  while(x!=nil){
    if(x==*xpr) wait(i2);
    x.val = 0;
    x = x.nxt;
  }
}
```

Figure 7. Parallelization of `sum_head` for two threads.

the program behaviour is unaffected by the translation (i.e. the translation enforces deterministic parallelism).

Reasoning about list segments. We assume that our separation logic domain includes the predicate $\mathsf{lseg}(x, t)$, which asserts that a segment of a linked list exists with head x and tail-pointer t. We define lseg as the least separation logic predicate satisfying the following recursive equation[3]:

$$\mathsf{lseg}(x,t) \triangleq x = t \wedge \mathsf{emp} \vee \exists v, y.\, x.\mathrm{val} \mapsto v * x.\mathrm{nxt} \mapsto y * \mathsf{lseg}(y,t)$$

We assume that the programmer proves the following specification for the sequential version of `sum_head`:

$$\{\mathtt{hd} \mapsto h * \mathsf{lseg}(h, \mathsf{nil})\} \quad \mathtt{sum\_head(n)} \quad \{\mathtt{hd} \mapsto h * \mathsf{lseg}(h, \mathsf{nil})\}$$

This specification is trivial: all it says is that executing `sum_head` beginning with a list segment, results in a list segment. Fig. 6 shows a proof of this specification.

We run our resource-usage analysis over the program to determine redundant and needed resources. Consider the following loop from the end of $\mathtt{sum\_head}_1$:

```
...
while(x!=nil) {
  x.val = 0; x = x.nxt;
}
```

Our analysis reveals that only the resource defined by $\mathsf{lseg}(\mathtt{x}, \mathsf{nil})$ is needed from the start of this loop to the end of the iteration. Comparing this resource to the corresponding invariant in the sequential proof reveals that the resource $\exists h.\, \mathtt{hd} \mapsto h * \mathsf{lseg}(h, \mathtt{x})$ is redundant at this point. This assertion represents the segment of the list that has already been traversed, from the head of the list to `x`.

Materialization and barrier injection. Notice that the assertions generated by our analysis are partly expressed in terms of the local variable `x`, which may change during execution. In order to safely transfer these assertions to subsequent iterations of the `pfor`, we need them to be *globally accessible* and *invariant*. To satisfy this goal, we could simply existentially quantify the offending variable, `x`, giving a redundant invariant

$$\exists h, y.\, \mathtt{hd} \mapsto h * \mathsf{lseg}(h, y)$$

However, such a weakening loses important information, in particular the relationship between the necessary resource, the list segment from `x` and the tail of the list. To retain such dependency relationships, our analysis *materializes* the current value of `x` into a location `xpr` shared between $\mathtt{sum\_head}_1$ and $\mathtt{sum\_head}_2$. An assignment is injected into $\mathtt{sum\_head}_1$ at the start of the second loop:

```
...
*xpr = x;
while(x!=nil) {
...
```

After the assignment to `xpr`, the redundant state can now be described as follows:

$$\exists h, y.\, \mathtt{hd} \mapsto h * \mathsf{lseg}(h, y) * \mathtt{xpr} \overset{1/2}{\longmapsto} y$$

Here we use fractional permissions in the style of [6] to allow a location to be shared between threads. The assertion $\mathtt{xpr} \overset{1/2}{\longmapsto} y$ represents fractional, read-only permission on the shared location `xpr`. This helps in binding together the head of the list and the remainder of the list when they are recombined.

When traversing the list, $\mathtt{sum\_head}_2$ compares its current position with `xpr`. If it reaches the pointer stored in `xpr`, it must wait to receive the second, remainder segment of the list from $\mathtt{sum\_head}_1$.

4. Technical Background

4.1 Programming Language and Representation

We assume the following heap-manipulating language[4]:

e	::=	$\mathtt{x} \mid \mathtt{nil} \mid t(\bar{e}) \mid \ldots$	*(expressions)*
b	::=	$\mathtt{true} \mid \mathtt{false} \mid e = e \mid e \neq e \mid \ldots$	*(booleans)*
a	::=	$\ldots$	*(atomic commands)*
C	::=	$C; C \mid \ell\colon \mathtt{skip} \mid \ell\colon a \mid \ell\colon \mathtt{x} := \mathtt{f}(\bar{e}) \mid \ell\colon \mathtt{return}\ e$	
	$\mid$	$\ell\colon \mathtt{if}(b)\,\{C\}\,\mathtt{else}\,\{C\} \mid \ell\colon \mathtt{while}(b)\,\{C\}$	
$\mathbb{P}$	::=	$\mathtt{global}\ \bar{\mathtt{r}};\ (\mathtt{f}(\bar{\mathtt{x}})\,\{\mathtt{local}\ \bar{\mathtt{y}};\ C\})^+$	

[3] Our analysis depends strongly on the choice of these basic predicates. See §7 for a discussion of alternatives to lseg.

[4] To avoid introducing an extra construct, we define $\mathtt{for}(C_1; C_2; b)\{C_3\}$ as $C_1; \mathtt{while}(b)\{C_2; C_3\}$.

```
void f(int i){
ℓ1:  int v = *x;
ℓ2:  if (v >= i){
ℓ3:    g(y, v);
     }
     else{
ℓ4:    g(x, 0);
     }
}

void g(int* p,
       int v){
ℓ5:  *p = v;
}
```

$\mathtt{f_s}\colon\ \mathtt{x} \mapsto x * \mathtt{y} \mapsto y$
$\ell_1\colon\ \mathtt{x} \mapsto x * \mathtt{y} \mapsto y$
$\ell_2\colon\ \mathtt{v} = x \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y$
$\ell_3\colon\ \mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y$
$\ell_4\colon\ \mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y$
$\mathtt{f_e}\colon\ (x \geq \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto x) \vee (x < \mathtt{i} \wedge \mathtt{x} \mapsto 0 * \mathtt{y} \mapsto y)$
$\mathtt{g_s}\colon\ \mathtt{p} \mapsto \_ \wedge \mathtt{v} = v$
$\ell_5\colon\ \mathtt{p} \mapsto \_ \wedge \mathtt{v} = v$
$\mathtt{g_e}\colon\ \mathtt{p} \mapsto v$

Figure 8. Left: labels for commands in `f` and `g`. Right: associated assertions in the sequential proof.

To simplify the exposition slightly, we assume a fixed input program $\mathbb{P}$ of the following form:

$$\mathtt{main}_{\mathbb{P}}()\,\{\,\mathtt{pfor}(C_1; C_2; b)\,\{\,\mathtt{work}();\,\}\,\}$$

Our parallelization approach generates a new program $\mathbb{P}_{par}$ that allows safe execution of (a transformed version of) `work` in separate threads. We denote by $\mathsf{Func} \ni \{\mathtt{main}, \mathtt{work}\}$ the set of functions declared in a program $\mathbb{P}$.

Labelling of commands. Every command $C \in \mathsf{Cmd}$ in the program is indexed by a unique label $\ell \in \mathsf{Label}$; function identifiers are also treated as labels. We identify a particular command by its label, and when needed denote by $\mathsf{cmd}(\ell)$ the command at the label ℓ. The set of all labels occurring in C is given by $\mathsf{Lbs}(C)$. The left-hand side of Fig. 8 shows a labelling of the functions `f` and `g`.

Commands within a block (function, `if-else` or `while`) form a sequence identified by the sequence of the corresponding labels. Functions $\mathsf{pred}, \mathsf{succ}\colon \mathsf{Label} \rightharpoonup \mathsf{Label}$ return the label of the previous (the next, resp.) command in the sequence. For ℓ corresponding to a function name, a function call or a `while` loop, the predecessor of the first command and successor of the last command in the block are denoted by $\ell_{\mathbf{s}}$ and $\ell_{\mathbf{e}}$, respectively. For ℓ corresponding to `if-else`, $\ell^{\mathsf{t}}_{\mathbf{s}}$ ($\ell^{\mathsf{t}}_{\mathbf{e}}$) and $\ell^{\mathsf{f}}_{\mathbf{s}}$ ($\ell^{\mathsf{f}}_{\mathbf{e}}$) are labels of the predecessor of the first (the successor of the last) commands in the `if` and `else` branches, respectively.

Given labels ℓ and ℓ' within the same block, we say that $\ell < \ell'$ if $\exists n \geq 1$ such that $\ell' = \mathsf{succ}^n(\ell)$. Let $\ell_\uparrow$ and $\ell_\downarrow$ denote the smallest (resp. largest) label in the block containing ℓ. Let $[\ell, \ell'\rangle$ denote the sequence of labels between ℓ inclusively and ℓ' exclusively, and let $\mathbb{P}_{[\ell,\ell'\rangle}$ stand for the corresponding program fragment. Analogously, we define $\langle \ell, \ell'\rangle$ and $\mathbb{P}_{\langle \ell,\ell'\rangle}$.

Program representation. A program is represented via a control-flow graph with Label as the set of nodes and as the set of edges all (ℓ, ℓ') such that ℓ corresponds to a label of a function call and $\ell' \in \langle \ell_{\mathbf{s}}, \ell_{\mathbf{e}}\rangle$, or ℓ corresponds to `if-else` and $\ell' \in \langle \ell^{\mathsf{t}}_{\mathbf{s}}, \ell^{\mathsf{t}}_{\mathbf{e}}\rangle \cup \langle \ell^{\mathsf{f}}_{\mathbf{s}}, \ell^{\mathsf{f}}_{\mathbf{e}}\rangle$. A *path delimiter* is a finite sequence of nodes in the control-flow graph. Intuitively, a path delimiter represents the sequence of function calls and conditionals that need to be traversed in order to reach a particular assertion. Due to the structure of our analysis, we are only interested in the set of path delimiters in the function `work`, denoted by AP.

We often want to manipulate and compare path delimiters. For $\gamma = \ell_1 \ldots \ell_n \in \mathsf{AP}$, we write $\gamma[i]$ to denote ℓ_i, $\gamma[i..j]$ to denote $\ell_i \ldots \ell_j$ and $|\gamma|$ to denote the length n. For path delimiters γ and γ', $\gamma \curlywedge \gamma'$ denotes their longest common prefix, i.e., for $k = |\gamma \curlywedge \gamma'|$ we have $\forall j \leq k,\ \gamma \curlywedge \gamma' = \gamma[j] = \gamma'[j]$ and if $|\gamma|, |\gamma'| > k$ then $\gamma[k+1] \neq \gamma'[k+1]$. We define a partial order $\prec$ on AP as follows. We say that $\gamma \prec \gamma'$ iff $\gamma \curlywedge \gamma' = \gamma$ and $|\gamma| < |\gamma'|$, or for $k = |\gamma \curlywedge \gamma'|$ we have $|\gamma|, |\gamma'| > k$ and $\gamma[k+1] < \gamma'[k+1]$. We say that $\gamma \preceq \gamma'$ iff $\gamma \prec \gamma'$ or $\gamma = \gamma'$.

Lemma 1. *$(\mathsf{AP}, \preceq)$ is a lattice with the least element $\mathtt{work_s}$ and the greatest element $\mathtt{work_e}$.*

For $\Gamma \subseteq_{fin} \mathsf{AP}$ we define $\max(\Gamma)$ as $\max(\Gamma) := \{\gamma \in \Gamma \mid \neg\exists \gamma' \in \Gamma\,.\,\gamma \prec \gamma'\}$. We define $\min(\Gamma)$ analogously.

4.2 Assertion Language and Theorem Prover

Assertions in our approach are expressed using a class of separation logic formulae called *symbolic heaps*. A symbolic heap Δ is a formula of the form $\exists \bar{x}\,.\,\Pi \wedge \Sigma$ where Π (the pure part) and Σ (the spatial part) are defined by:

$$\begin{array}{rcl} \Pi & ::= & \mathsf{true} \mid e = e \mid e \neq e \mid p(\bar{e}) \mid \Pi \wedge \Pi \\ \Sigma & ::= & \mathsf{emp} \mid x \mapsto e \mid \mathsf{lseg}(e, e) \mid s(\bar{e}) \mid \Sigma * \Sigma \end{array}$$

Here $\bar{x}$ are logical variables, e ranges over expressions, $p(\bar{e})$ is a family of pure (first-order) predicates (such as e.g., arithmetic inequalities, etc), and $s(\bar{e})$ a family of other spatial predicates (such as e.g., doubly-linked lists, trees, etc) besides points-to and linked-list predicates previously discussed. We refer to the pure and the spatial part of Δ as Δ^{Π} and Δ^{Σ}. We denote set of all quantifier-free first-order formulae built in the same way as Π but also allowing the $\vee$ connective by $\Pi_\vee$.

During our analysis, we often need to substitute variables, for example when recasting an assertion into a different calling context. If $\varrho = \bar{x} \mapsto \bar{e}$ is a mapping from variables in $\bar{x}$ to $\bar{e}$ then by $\Delta[\varrho]$ we denote the formula obtained by simultaneously substituting every occurrence of x_i in Δ with the corresponding e_i. We denote by δ^{-1} the inverse variable mapping, if δ is injective[5].

The set of all symbolic heaps is denoted by SH. We represent a disjunction of symbolic heaps as a set and interchangeably use the $\cup$ and $\vee$ operators. The set of all disjunctive symbolic heaps is $\mathcal{P}(\mathsf{SH})$. We overload the $\wedge$ and $*$ operators in a natural way: for $\Delta_i = \Pi_i \wedge \Sigma_i, i = 1, 2$, we define $\Delta_1 * \Delta_2 = (\Pi_1 \wedge \Pi_2) \wedge (\Sigma_1 * \Sigma_2)$, $\Pi \wedge \Delta_i = (\Pi \wedge \Pi_i) \wedge \Sigma_i$ and $\Sigma * \Delta_i = \Pi_i \wedge (\Sigma * \Sigma_i)$. Operators $\wedge$ and $*$ distribute over $\vee$, thus we allow these operations on disjunctive heaps just as if they were on symbolic heaps and furthermore use the same notation Δ to refer to both symbolic and disjunctive symbolic heaps.

We assume a sufficiently powerful (automated) prover for separation logic that can deal with three types of inference queries:

- $\Delta_1 \vdash \Delta_2 * [\Delta_F]$ (frame inference): given Δ_1 and Δ_2 find the frame Δ_F such that $\Delta_1 \vdash \Delta_2 * \Delta_F$ holds;
- $\Delta_1 * [\Delta_A] \vdash \Delta_2$ (abduction): given Δ_1 and Δ_2 find the "missing" assumption Δ_A such that $\Delta_1 * \Delta_A \vdash \Delta_2$ holds;
- $\Delta_1 * [\Delta_A] \vdash \Delta_2 * [\Delta_F]$ (bi-abduction): given Δ_1 and Δ_2 find Δ_A and Δ_F such that $\Delta_1 * \Delta_A \vdash \Delta_2 * \Delta_F$ holds.

As before, square brackets denote the portion of the entailment that should be computed. We sometimes write $[\_]$ for a computed assertion that is existentially quantified and will not be reused.

None of these queries has a unique answer in general. However, for our analysis any answer is acceptable (though some will give rise to a better parallelization than the others). Existing separation logic tools generally provide only one answer.

[5] In our framework, substitutions are always guaranteed to be injective because the variables being substituted correspond to heap locations and channel resources whose denotations are guaranteed to be distinct; if the substitution involves values, then they must be implicitly existentially quantified, and can therefore be assumed to be distinct.

4.3 Sequential Proof

We assume a separation logic proof of the program $\mathbb{P}$, represented as a map $\mathfrak{P}\colon \mathsf{Label} \to \mathcal{P}(\mathsf{SH})$. Intuitively, assertions in the proof have the property that for any label ℓ executing the program from a state satisfying $\mathfrak{P}(\ell)$ up to some subsequent label ℓ' will result in a state satisfying $\mathfrak{P}(\ell')$, and will not fault.

More formally, we assume functions $\mathsf{Pre}, \mathsf{Post}\colon \mathsf{Label} \rightharpoonup \mathcal{P}(\mathsf{SH})$ associating labels of atomic commands and function calls with their pre- and post-condition, respectively, and a function $\mathsf{Inv}\colon \mathsf{Label} \rightharpoonup \mathcal{P}(\mathsf{SH})$ associating `while` labels with loop invariants. We also assume Ω, a mapping from labels to variable substitutions such that $\Omega(l) = \bar{x} \mapsto \bar{x'}$ maps formal variables $\bar{x}$ in the specification assertion to actual variables $\bar{x'}$ in the proof assertion. Finally, we assume a function $\mathfrak{F}\colon \mathsf{Label} \rightharpoonup \mathcal{P}(\mathsf{SH})$ giving the framed portion of the state. We write $\Delta[\Omega(l)]$ to represent the heap constructed by applying the substitutions defined by $\Omega(l)$ to the assertion Δ.

Then at each label ℓ we have $\mathfrak{P}(\ell) \vdash \mathsf{Pre}(\ell)[\Omega(\ell)] * \mathfrak{F}(\ell)$ and $\mathsf{Post}(\ell)[\Omega(\ell)] * \mathfrak{F}(\ell) \vdash \mathfrak{P}(\mathsf{succ}(\ell))$. If ℓ is a `while` label then $\mathsf{Pre}(\ell)$ and $\mathsf{Post}(\ell)$ are replaced by $\mathsf{Inv}(\ell)$.

This structure means that the proof is *modular*, i.e., each atomic command, function and loop is given a specification "in isolation", without a reference to the particular context in which the specification is being used. The right-hand side of Fig. 8 shows a proof for the functions `f` and `g` which is in this form. Our approach is agnostic to the method by which the proof $\mathfrak{P}$ is created: it can be discovered automatically (e.g., by a tool such as Abductor [11]), prepared by a proof assistant, or written manually.

Proof assertions inlining. The proof $\mathfrak{P}$ assumes use of a modular specification for each function. However, our analysis needs to refer to the assertions in the local function with respect to the variables of the caller, all the way up to the top-most `work` function. We therefore define a process for lifting local assertions with respect to their global contexts.

For $\gamma \in \mathsf{AP}$ such that $|\gamma| = m$ and $\ell_1, \ldots, \ell_n$ are the labels corresponding to the function calls (in the order of occurrence as in γ) we define the lifted ("inlined") proof assertion $\mathfrak{P}(\gamma)$ as

$$\mathfrak{F}(\ell_1) * (\mathfrak{F}(\ell_2) * \ldots (\mathfrak{F}(\ell_n) * \mathfrak{P}(\gamma[m])[\Omega(\ell_n)])[\Omega(\ell_{n-1})] \ldots)[\Omega(\ell_1)]$$

Intuitively, the assertion $\mathfrak{P}(\gamma)$ represents the "global" proof state at γ (including the framed portions) in terms of `work`'s variables.

Finally, let $\mathsf{pc}\colon \mathsf{AP} \rightharpoonup \Pi_\vee$ represent the path constraint associated with each path delimiter. The path constraint at γ comprises the pure part of $\mathfrak{P}(\gamma)$ corresponding to the assumptions from the conditionals encountered on the way from $\texttt{work}_\mathbf{s}$ to γ. The path constraint may be extracted directly from the proof or computed by some other means.

5. Parallelization Algorithm

We now formally define our parallelization algorithm. The goal of our approach is to construct a parallelized version of the input program $\mathbb{P}$. In particular, our approach generates a new function `work`′ in $\mathbb{P}_{par}$ (invoking possibly transformed callees) such that

```
main_{P_par}() {
  // initial channel creation.
  for(C1; C2; b) {
    // channel creation.
    fork(work', ...);
  }
  // channel finalization.
}
```

has the same behaviour as the original $\texttt{main}_{\mathbb{P}}$.

Algorithm 1 Computing locally needed resources using backward symbolic execution.

function NEEDED-LOC($(\ell', \ell'')\colon \mathsf{Label} \times \mathsf{Label}, \Delta\colon \mathcal{P}(\mathsf{SH})$)
  $\ell := \ell''$
  while $\ell \neq \ell'$ **do**
    $\ell := \mathsf{pred}(\ell)$
    if $\mathsf{cmd}(\ell)$ matches $\texttt{if}(b)\,\{C\}\,\texttt{else}\,\{C'\}$ **then**
      $\Delta := \mathfrak{P}(\ell)^{\Pi} \wedge \begin{pmatrix}\text{NEEDED-LOC}((\ell^{\mathbf{t}}_{\mathbf{s}}, \ell^{\mathbf{t}}_{\mathbf{e}}), \Delta) \cup \\ \text{NEEDED-LOC}((\ell^{\mathbf{f}}_{\mathbf{s}}, \ell^{\mathbf{f}}_{\mathbf{e}}), \Delta)\end{pmatrix}$
    if $\mathsf{cmd}(\ell)$ matches $\texttt{while}(b)\,\{C\}$ **then**
      $\mathsf{Inv}(\ell)[\Omega(\ell)] * [\Delta_A] \vdash \Delta * [\_]$
      $\Delta := \mathfrak{P}(\ell)^{\Pi} \wedge (\mathsf{Inv}(\ell)[\Omega(\ell)] * \Delta_A)$
    else
      $\mathsf{Post}(\ell)[\Omega(\ell)] * [\Delta_A] \vdash \Delta * [\_]$
      $\Delta := \mathfrak{P}(\ell)^{\Pi} \wedge (\mathsf{Pre}(\ell)[\Omega(\ell)] * \Delta_A)$
  return Δ

5.1 Resource Usage Analysis

Our approach to parallelization traverses the program by referring to a finite prefix- and $<$-closed subset of path delimiters $\mathsf{P} \subseteq_{fin} \mathsf{AP}$. This subset reflects the portions of the program that are path- and context-sensitive targets of parallelization. The set P provides precise control over which functions the analysis should address, but how this set is chosen is not considered here. Function invocations whose successors are not in P are treated as a single operation with effects defined by their specification.

The goal of resource usage analysis is to compute the maps:

$$\mathsf{redundant}\colon \mathsf{P} \times \mathsf{P} \rightharpoonup \mathcal{P}(\mathsf{SH})$$
$$\mathsf{needed}\colon \mathsf{P} \times \mathsf{P} \rightharpoonup \mathcal{P}(\mathsf{SH})$$

For $p, q \in \mathsf{P}$ such that $p \prec q$, $\mathsf{redundant}(p, q)$ gives the resources that are guaranteed to not be accessed by the program between p and q. In parallelization, these are resources that can safely be transferred to other parallel threads. For $p \prec q$, $\mathsf{needed}(p, q)$ gives the resources that might be accessed during execution from p to q. In parallelization, these are the resources that must be acquired before execution of the current thread can proceed.

The function NEEDED-LOC (Alg. 1) uses backward symbolic execution to compute needed resources between pairs of path delimiters (ℓ', ℓ'') in the same function block. It uses the pure parts of the sequential proof to guide the abductive inference. Since we already have function summaries and loop invariants in the sequential proof, NEEDED-LOC gives a symbolic heap sufficient to execute the fragment $\mathbb{P}_{[\ell', \ell'')}$.

Lemma 2. NEEDED-LOC(ℓ', ℓ'') *is a sufficient precondition for* $\mathbb{P}_{[\ell', \ell'')}$.

Proof. Follows from the disjunctive version of the frame rule:

$$\frac{\{P\}\, C\, \{\bigvee_{i \in \mathcal{I}} Q_i\} \qquad \Delta * \bigvee_{j \in \mathcal{J}} \Delta^A_j \vdash P * \bigvee_{k \in \mathcal{K}} \Delta^F_k}{\{\bigvee_{j \in \mathcal{J}} (\Delta * \Delta^A_j)\}\, C\, \{\bigvee_{i \in \mathcal{I}, k \in \mathcal{K}} (Q_i * \Delta^F_k)\}}$$

and Hoare's rule of composition. □

The function NEEDED(γ_e, γ_s) (Alg. 2) lifts NEEDED-LOC to the context-sensitive interprocedural level. Given two assertion points represented as path delimiters γ_s and γ_e, NEEDED(γ_e, γ_s) works by successively pushing backwards an assertion Δ from γ_e to γ_s. The algorithm operates in two phases. In phase A, it steps backwards from γ_e towards the outermost calling context in the function-invocation hierarchy. This context, represented as the longest common prefix of γ_s and γ_e, is the dominator of the two

Algorithm 2 Computing needed resources.

function NEEDED($\gamma_s : \mathsf{P}, \gamma_e : \mathsf{P}$)
  $k := |\gamma_e|$
  $\Delta := \mathfrak{P}(\gamma_e)$
  while $k > |\gamma_s \curlywedge \gamma_e| + 1$ **do** } A
    $\varrho := \Omega(\gamma_e[1..k])$
    $\Delta :=$ NEEDED-LOC$((\gamma_e[k]_\uparrow, \gamma_e[k]), \Delta[\varrho^{-1}])[\varrho]$
    $k := k - 1$
    if $\mathsf{cmd}(\gamma_e[k])$ is function call **then**
      $\Delta := \Delta[\Omega(\gamma_e[k])^{-1}]$
  $\varrho := \Omega(\gamma_s[1..k])$
  $\Delta :=$ NEEDED-LOC$((\gamma_e[k], \gamma_s[k]), \Delta[\varrho^{-1}])[\varrho]$
  while $k < |\gamma_s|$ **do** } B
    if $\mathsf{cmd}(\gamma_s[k])$ is function call **then**
      $\Delta := \Delta[\Omega(\gamma_s[k])]$
    $k := k + 1$
    $\varrho := \Omega(\gamma_s[1..k])$
    $\Delta :=$ NEEDED-LOC$((\gamma_s[k], \gamma_s[k]_\downarrow), \Delta[\varrho^{-1}])[\varrho]$
  return Δ

Algorithm 3 Computing redundant resources

function REDUNDANT($\gamma_s : \mathsf{P}, \gamma_e : \mathsf{P}$)
  $\mathfrak{P}(\gamma_s) \vdash$ NEEDED$(\gamma_s, \gamma_e) * [\Delta_R]$
  return Δ_R

functions in which γ_s and γ_e are found in the function call graph. Phase B of the algorithm keeps stepping backwards, but proceeds inwards into the function-invocation hierarchy towards γ_s. Both phases of the algorithm use Alg. 1 to compute the needed resources in-between function call boundaries: in phase A we establish the needed assertions from the dominating point to γ_e, and in phase B from γ_s to the dominating point.

Since the invariants of the input proof are written in terms of the outermost calling context, comparing locally-computed specifications with these invariants requires the local specifications to be recast in terms of the outer context. In the first line of phase A we construct a variable substitution ϱ that recasts the assertion in terms of the calling context at the start of γ_e. The second line constructs $\Delta[\varrho^{-1}]$—the running state recast in terms of γ_e's starting context; this is typically the context defined by the `work()` function used in a `pfor` command. NEEDED-LOC constructs a new needed state up to the start of the current block. Finally, ϱ recasts the resulting state back into the current context. When a function call is reached, we unwind the variable substitution by one call since we now have moved from the callee's context to a caller's. Operations in phase B are similar.

The results computed by NEEDED are tabulated as follows. If $\mathfrak{P}(\gamma_s) \vdash$ NEEDED$(\gamma_s, \gamma_e) * [_]$, then $\mathsf{needed}(\gamma_s, \gamma_e) := \mathfrak{P}(\gamma_s)^{\text{II}} \wedge$ NEEDED(γ_s, γ_e); otherwise, $\mathsf{needed}(\gamma_s, \gamma_e) := \mathfrak{P}(\gamma_s)$.

Lemma 3. $\mathsf{needed}(\gamma_s, \gamma_e)$ *is a sufficient precondition to execute* $\mathbb{P}$ *from* γ_s *to* γ_e.

In order to be able to use the needed map further in the algorithm we must ensure that it grows monotonically, i.e., that $\forall \gamma, \gamma', \gamma'' \in \mathsf{P}$ such that $\gamma \prec \gamma' \prec \gamma''$ we have $\mathsf{needed}(\gamma, \gamma'') \vdash \mathsf{needed}(\gamma, \gamma') * [\_]$. If the underlying theorem prover behaves consistently with respect to failing and precision this property always holds. However, we can also make an additional check and insert assertions from the sequential proof as needed.

The redundant resource between two path delimiters is the portion of the inlined proof-state in $\mathfrak{P}$ that is not required by the needed map. In Alg. 3 we calculate this by frame inference.

	$\mathtt{f_e}$
ℓ_1	$x < \mathtt{i} \wedge \mathtt{y} \mapsto y$
ℓ_2	$(\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto x)$ $\vee (\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{y} \mapsto y)$
$\ell_2\ell_3$	$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto x$
$\ell_2\ell_3\mathtt{g_e}$	$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y$
$\ell_2\ell_4$	$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{y} \mapsto y$
$\ell_2\ell_4\mathtt{g_e}$	$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y$

Figure 10. redundant map with respect to $\mathtt{f_e}$.

Lemma 4. *If* $\mathfrak{P}(\gamma_s) \vdash \mathsf{redundant}(\gamma_s, \gamma_e) * [\Delta]$ *then* Δ *is a sufficient precondition to execute* $\mathbb{P}$ *from* γ_s *to* γ_e.

Consider the functions `f` and `g` from §2. Fig. 8 shows the labels and sequential proof for these functions. The map obtained by NEEDED is shown in Fig. 9. Fig. 10 shows the map computed by REDUNDANT with respect to path delimiter $\mathtt{f_e}$.

5.2 Parallelising Transformation

We now describe a parallelising transformation based on our resource-usage analysis. The construction proceeds in two phases. First, we compute an idealized resource transfer between threads. Then we inject `grant` and `wait` barriers to realise this resource transfer. The resource transfer mechanism transfers a resource from one invocation of the `work` function to another.

This parallelising transformation can be viewed as just one application of the resource-usage analysis; more optimized proof-preserving parallelising transformations are certainly possible. Our overall goal is to describe a framework for a resource-sensitive dependency-preserving analysis.

Conditions on released and acquired resources. In the first phase we determine resources that should be released and acquired at particular points in the parallelized program. Released resources cannot be revoked, i.e., each released resource should be included in the redundant map from the point of the release to the end of the `work` function—this way we know the resource will not be needed further. Acquired resources are held by the executing thread until released. Resources that are acquired along a sequence of path delimiters should contain what is prescribed by the needed map between each of the path delimiters.

We represent the result of this phase of the algorithm via the following maps:

- $\mathsf{resource} : \mathsf{ResId} \to \mathcal{P}(\mathsf{SH})$, denoting resource identifiers that identify released and acquired resources;
- $\mathsf{released} : \mathsf{P} \rightharpoonup \mathsf{ResId} \times \mathsf{Subst}$, representing resources that are going to be released at a path delimiter together with the variable substitution applied at that point;
- $\mathsf{acquired} : \mathsf{P} \rightharpoonup \mathsf{ResId}$, representing resources that are going to be acquired at a path delimiter.

We require the following well-formedness properties:

1. $\forall \gamma \in \mathrm{dom}(\mathsf{released}) . \forall \gamma' \succ \gamma . \mathsf{released}(\gamma) = (r, \rho) \to (\mathsf{redundant}(\gamma, \gamma') \vdash \mathsf{resource}(r)[\rho] * [\_])$;
2. $\forall \gamma \in \mathsf{P} . \circledast_{r \in \mathrm{dom}(\mathsf{resource})} \{\mathsf{resource}(r) \mid \exists \gamma' \prec \gamma \wedge \mathsf{acquired}(\gamma') = r\} \vdash \mathsf{needed}(\gamma, \mathtt{work_e}) * [\_]$;
3. $\forall \gamma \in \mathrm{dom}(\mathsf{released}) . \circledast_{r \in \mathrm{dom}(\mathsf{resource})} \{\mathsf{resource}(r) \mid \exists \gamma' \prec \gamma \wedge \mathsf{acquired}(\gamma') = r\} \vdash \mathsf{released}(\gamma) * [\_]$.

The first property states we can release only resources that are not needed between the given path delimiter and any subsequent one. The second property states that the resources needed at a

	ℓ_2	$\ell_2\ell_3$	$\ell_2\ell_3\mathtt{g_e}$	$\ell_2\ell_4$	$\ell_2\ell_4\mathtt{g_e}$	$\mathtt{f_e}$
ℓ_1	$\mathtt{x} \mapsto x$	$x \geq \mathtt{i} \wedge \mathtt{x} \mapsto x$	$x \geq \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y$	$x < \mathtt{i} \wedge \mathtt{x} \mapsto x$	$x < \mathtt{i} \wedge \mathtt{x} \mapsto x$	$(x \geq \mathtt{i} \wedge \mathtt{x} \mapsto x * \mathtt{y} \mapsto y) \vee (x < \mathtt{i} \wedge \mathtt{x} \mapsto x)$
ℓ_2		$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i}$	$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} * \mathtt{y} \mapsto y$	$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i}$	$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto x$	$(\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} * \mathtt{y} \mapsto y) \vee (\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto x)$
$\ell_2\ell_3$			$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} * \mathtt{y} \mapsto y$			$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i} * \mathtt{y} \mapsto y$
$\ell_2\ell_3\mathtt{g_e}$						$\mathtt{v} = x \wedge \mathtt{v} \geq \mathtt{i}$
$\ell_2\ell_4$					$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} * \mathtt{x} \mapsto x$	$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i} * \mathtt{x} \mapsto x$
$\ell_2\ell_4\mathtt{g_e}$						$\mathtt{v} = x \wedge \mathtt{v} < \mathtt{i}$

Figure 9. needed map (some entries omitted).

Algorithm 4 Computing released and acquired resources.

1: $\mathcal{N} := \mathsf{needed}$; $\mathcal{R} := \mathsf{redundant}$
2: $\mathcal{C} := \max\{\gamma \in \mathsf{P} \mid \mathcal{N}(\mathtt{work_s}, \gamma) = \mathsf{emp}\}$
3: **while** $\mathcal{C} \neq \{\mathtt{work_e}\}$ **do**
4: $\quad \Sigma_R := \mathcal{N}(\mathit{choose}(\{(\gamma, \gamma') \mid \gamma \in \mathcal{C}, \gamma' \in \mathsf{P}\}))^{\Sigma}$
5: $\quad \varrho := \bar{\mathtt{x}} \mapsto \bar{\mathtt{x}}'$, where $\bar{\mathtt{x}}'$ fresh
6: $\quad \Sigma'_R := \Sigma_R[\varrho]$
7: $\quad \mathcal{C}_r := \min\left\{\gamma \in \mathsf{P} \;\middle|\; \begin{array}{r}\mathcal{R}(\gamma, \mathtt{work_e})^{\Sigma} \vdash \Sigma'_R * [\,\_\,] \\ \wedge\, \exists \gamma' \in \mathcal{C}.\, \gamma' \preceq \gamma\end{array}\right\}$
8: $\quad$ **if** $\bigvee_{\gamma \in \mathcal{C}_r} \mathsf{pc}(\gamma) \Leftrightarrow \mathsf{true}$ **then**
9: $\quad\quad r :=$ fresh resource id
10: $\quad\quad \mathsf{resource}(r) := \Sigma'_R$
11: $\quad\quad$ **for all** $\gamma \in \mathcal{C}_r$ **do**
12: $\quad\quad\quad \mathsf{released}(\gamma) := (r, \varrho)$
13: $\quad\quad\quad$ **for all** γ' s.t. $\gamma \preceq \gamma'$ **do**
14: $\quad\quad\quad\quad \mathcal{R}(\gamma', \mathtt{work_e})^{\Sigma} \vdash \Sigma_R * [\Delta]$
15: $\quad\quad\quad\quad \mathcal{R}(\gamma', \mathtt{work_e}) := \Delta$
16: $\quad\quad$ **for all** $\gamma \in \mathcal{C}$ **do**
17: $\quad\quad\quad \mathsf{acquired}(\gamma) := r$
18: $\quad\quad\quad$ **for all** γ', γ'' s.t. $\gamma \preceq \gamma' \preceq \gamma''$ **do**
19: $\quad\quad\quad\quad \mathcal{N}(\gamma', \gamma'') * [\,\_\,] \vdash \Sigma'_R * [\Delta]$
20: $\quad\quad\quad\quad \mathcal{N}(\gamma', \gamma'') := \Delta$
21: $\quad\quad \mathcal{C} := \max\{\gamma' \mid \exists \gamma \in \mathcal{C}\,.\, \mathcal{N}(\gamma, \gamma') = \mathsf{emp}\}$

path delimiter must have already been acquired. The third property states that only the resources that have been previously acquired can be released.[6]

In general, there are many solutions satisfying properties 1–3. For instance, there is always a trivial solution that acquires $\mathsf{needed}(\mathtt{work_s}, \mathtt{work_e})$ before the first command and releases it after the last, causing each invocation of `work` to be blocked until the preceding invocation finishes the last command. Of course, some solutions are better than others.

Computing released and acquired maps. Algorithm 4 constructs released, acquired and resource maps satisfying properties 1–3. Each iteration of the algorithm heuristically picks a needed resource, and then iteratively searches for matching redundant resource along all paths. The algorithm maintains a set $\mathcal{C}$ of all path delimiters up to which no more resources are needed. It terminates once no unsatisfied needed resources remain (line 3).

At the start of the main loop (line 4) the algorithm picks a still-needed resource between a path delimiter in $\mathcal{C}$ and some further path delimiter. The picking of the needed resource is governed by a heuristic function *choose*, for which we make no assumption. The *choose* function serves as a proxy for external knowledge about likely points for parallelization.

The key step of the algorithm is performed on line 7:

$$\mathcal{C}_r := \min\left\{\gamma \in \mathsf{P} \;\middle|\; \mathcal{R}(\gamma, \mathtt{work_e})^{\Sigma} \vdash \Sigma'_R * [\,\_\,] \wedge \exists \gamma' \in \mathcal{C}.\, \gamma' \preceq \gamma\right\}$$

Here $\mathcal{R}(\gamma, \mathtt{work_e})$ is the redundant resource from γ to the end of the `work` function and Σ'_R is the candidate resource that we want to acquire. The constructed set $\mathcal{C}_r$ is a set of path delimiters along which we can satisfy the candidate needed resource. In line 8, the algorithm checks that $\mathcal{C}_r$ covers all paths by checking the conjunction of path constraints is tautologous.

Resources stored in needed contain path constraints (and other conditions on local variables) embedded in the pure part of the symbolic heap. Since we can transfer resources between different path delimiters, we only take the spatial part of the resource into consideration when asking entailment questions; this is denoted by a superscript Σ. Moreover, since the acquired resource is being sent to a different function invocation, we substitute a fresh set of variables (line 5).

The remainder of the algorithm is devoted to constructing the new resource (line 10), and with updating released (lines 11–15), acquired (lines 16–20), and $\mathcal{C}$ (line 21).

Lemma 5. *Maps* resource, released *and* acquired *computed by Algorithm 4 satisfy properties 1–3.*

Consider our running example. If *choose* picks (ℓ_1, ℓ_2) in the first iteration and $(\ell_2\ell_3, \ell_2\ell_3\mathtt{g_e})$ in the second iteration, then the end result of Alg. 4 is $\mathsf{resource} = \{r_1 \mapsto (\mathtt{x} \mapsto x), r_2 \mapsto (\mathtt{y} \mapsto y)\}$, $\mathsf{released} = \{\ell_2\ell_3 \mapsto (r_1, \emptyset), \ell_2\ell_4\mathtt{g_e} \mapsto (r_1, \emptyset), \ell_2\ell_3\mathtt{g_e} \mapsto (r_2, \emptyset), \ell_2\ell_4 \mapsto (r_2, \emptyset)\}$ and $\mathsf{acquired} = \{\ell_1 \mapsto r_1, \ell_2\ell_3\ell_5 \mapsto r_2, \ell_2\ell_4 \mapsto r_2\}$.

Inserting grant and wait barriers. In this phase we transform the sequential program $\mathbb{P}$ into a parallel program $\mathbb{P}_{par}$ by inserting `grant` and `wait` barriers. The inserted barriers realise resource transfer defined by the maps released and acquired.

We generate the parallel function $\mathtt{work}'(\bar{\mathtt{i}}_r^{(p)}, \bar{\mathtt{i}}_r, \mathtt{env}^{(p)}, \mathtt{env})$ in $\mathbb{P}_{par}$ as follows:

1. To each $r \in \mathsf{ResId}$ we assign a unique channel name $\mathtt{i}_r$. Denote by $\mathtt{i}_r^{(p)}$ the corresponding channel of the previous thread in the sequence.
2. Let `env` be an associative array that for each channel maps (escaped) local variable names to values. Let $\mathtt{env}^{(p)}$ be such map from the previous thread in the sequence. `env` and $\mathtt{env}^{(p)}$ are used for *materialization*.
3. For each $\gamma = \ell_1 \dots \ell_n \in \mathrm{dom}(\mathsf{released}) \cup \mathrm{dom}(\mathsf{acquired})$ let $\ell_{k_1}, \dots \ell_{k_m}$ be the labels in γ corresponding to function calls. Then for each $\gamma_j := \ell_1 \dots \ell_{k_j}$ we create in $\mathbb{P}_{par}$ an identical copy $\mathtt{f}'$ of the function $\mathtt{f}$ called at ℓ_{k_j} and replace the call to $\mathtt{f}$ with the call to $\mathtt{f}'$. Let us denote by $\mathtt{tr}(\gamma')$ the path delimiter

[6] We could relax the third requirement if we extended our barriers to support *renunciation* [17], the ability to release a resource without first acquiring it. Renunciation allows a resource to 'skip' iterations, giving limited out-of-order signalling. We believe it would be straightforward to fold such techniques into the analysis, although such extensions are outside the focus of this paper.

in $\mathbb{P}_{par}$ corresponding to γ' after this transformation has been applied for all $\gamma \in \mathrm{dom}(\mathsf{released}) \cup \mathrm{dom}(\mathsf{acquired})$.

4. For each $\gamma \in \mathrm{dom}(\mathsf{acquired})$ such that $\mathsf{acquired}(\gamma) = r$ we insert a wait barrier $\mathtt{wait}(\mathtt{i}_r^{(p)})$ between path delimiters $\mathsf{tr}(\mathsf{pred}(\gamma))$ and $\mathsf{tr}(\gamma)$.
5. For each $\gamma \in \mathrm{dom}(\mathsf{released})$ such that $\mathsf{released}(\gamma) = (r, \_)$, between path delimiters $\mathsf{tr}(\mathsf{pred}(\gamma))$ and $\mathsf{tr}(\gamma)$ we insert a sequence of assignments of the form $\mathtt{env}(\mathtt{i}_r)[\text{"}\mathtt{y}\text{"}] := \mathtt{y}$ for every local variable $\mathtt{y}$, followed by a grant barrier $\mathtt{grant}(\mathtt{i}_r)$.

Each invocation of `work` creates a fresh set of local variables that are bound to the scope of the function. However, some resources must be expressed in terms of function-local variables. Parallelization must take account of this. If the structure of a resource depends on local variables from a previous invocation, this must be encoded explicitly by *materialising* the variables of the previous invocation.

The main function $\mathtt{main}_{\mathbb{P}_{par}}$ in $\mathbb{P}_{par}$ first creates the set of "dummy" channels; then in the `while` loop repeatedly creates a set of new channels for the current iteration, forks a new thread with $\mathtt{work}'$ taking the channels from the previous iteration as $\bar{\mathtt{i}}_r^{(p)}$ and from the current iteration as $\bar{\mathtt{i}}_r$, and at the end of the loop body assigns the new channels to the previous channels; and, after the `while` loop completes waits on the channels in the last set.

We generate the parallel proof $\mathfrak{P}_{par}$ from the sequential proof $\mathfrak{P}$ using the following specifications from [17]:

$$\begin{array}{rcl} \{\mathsf{emp}\} & \mathtt{i} := \mathtt{newchan}() & \{\mathsf{req}(\mathtt{i}, R) * \mathsf{fut}(\mathtt{i}, R)\} \\ \{\mathsf{req}(\mathtt{i}, R) * R\} & \mathtt{grant}(\mathtt{i}) & \{\mathsf{emp}\} \\ \{\mathsf{fut}(\mathtt{i}, R)\} & \mathtt{wait}(\mathtt{i}) & \{R\} \end{array}$$

Each variable R in $\mathfrak{P}_{par}$ associated with channel $\mathtt{i}_r$ is instantiated with the corresponding resource $\mathsf{resource}(r)$. The predicates req and fut track the ownership of the input and output ends of the channel. To reason about threads, we use the standard separation logic rules for fork-join disjoint concurrency (e.g., as in [19]).

Theorem 6. *$\mathfrak{P}_{par}$ is a proof of the parallel program $\mathbb{P}_{par}$, and defines the same specification for $\mathtt{main}_{\mathbb{P}_{par}}$ as $\mathfrak{P}$ does for $\mathtt{main}_{\mathbb{P}}$.*

Loop-splitting. The approach presented so far treats a loop as a single command with a specification derived from its invariant. Acquiring or releasing resources within a loop is subtle, as it changes the sequential loop invariant. It is not clear how to handle this in full generality, so we take a pragmatic approach that performs heuristic loop-splitting.

The example in §3 uses two channels to transfer the segment of the list traversed after the first and the second `while` loop, respectively. The resource released via channel `i1` in Fig. 7 is $\mathtt{hd} \mapsto h * \mathsf{lseg}(h, \mathtt{xpr})$. In the following iteration, the needed resource for the whole loop is $\mathtt{hd} \mapsto h * \mathsf{lseg}(h, \mathtt{x})$. If we try to match released against needed, the entailment $\mathcal{R}(\gamma, \mathtt{work}_\mathtt{e})^\Sigma \vdash \Sigma'_R * [\_]$ in Algorithm 4 will fail. This is because the value of `x` is unknown at the start of the loop, meaning we cannot establish whether the released resource will cover the needed resource.

One way to resolve this would be to acquire the entire list before the first loop, but this would result in a very poor parallelization. Instead, we modify the structure of the loop to expose the point at which the second list segment becomes necessary:

1. We split the spatial portion of the resource needed by the whole loop into a "dead" (already traversed) and a "live" (still to be traversed) part. In our example, $\mathtt{hd} \mapsto h * \mathsf{lseg}(h, \mathtt{x})$ would be the "dead" and $\mathsf{lseg}(\mathtt{x}, \mathsf{nil})$ the live part. This kind of splitting is specific to some classes of programs, e.g., linked list programs that do not traverse a node twice.
2. We match the resource against the "dead" part of the loop invariant and infer the condition under which the two resources are the same. In our example, the entailment between the two resources holds if $\mathtt{x} = \mathtt{xpr}$. This condition can be inferred by asking a bi-abduction question $\mathcal{R}(\gamma, \mathtt{work}_\mathtt{e})^\Sigma \wedge [c] \vdash \Sigma'_R * [\_]$ with the pure abducted fact c.

```
...
xpr:=envp(i1)["x"];
while(x!=nil&&i!=n&&x!=xpr){
  sum+=x.val;
  i++;
  x:=x.nxt;
}
if(x!=nil&&i!=n&&x==xpr){
  sum+=x.val;
  i++;
  x:=x.nxt;
  while(x!=nil&&i!=n){
    sum+=x.val;
    i++;
    x:=x.nxt;
  }
  // remainder skipped.
  ...
else{
...
```

Figure 11. Loop splitting for the `sum_head` example.

Now we can syntactically split the loop against the inferred condition $c = (\mathtt{x} = \mathtt{xpr})$ and obtain a transformed version that ensures that after entering the true branch of the `if` statement the condition c holds. The transformation of our example is shown in Fig 11.

Formally, we define splitting of a command (comprising possibly multiple loops) against a condition c as follows (to simplify, we assume here that every command ends with `skip`):

function SPLIT(C: Cmd, c: $\Pi_\vee$)
  match C **with**
    | $\mathtt{while}(b)\,\{\,C'\,\};C'' \rightarrow$
      $\mathtt{while}(b \wedge \neg c')\,\{\,C'\,\};$
      $\mathtt{if}(b \wedge c')\,\{\,C';\mathtt{while}(b)\,\{\,C'\,\};C''\,\}$
      $\mathtt{else}\,\{\,\mathrm{SPLIT}(C'', c)\,\}$
    | $C';C'' \rightarrow C';\mathrm{SPLIT}(C'', c)$
    | $\mathtt{skip} \rightarrow \mathtt{skip}$

The condition c' is obtained from c by replacing a reference to every primed variable $\mathtt{y}'$ by a reference to $\mathtt{env}^{(p)}(\mathtt{i}_r^{(p)})[\text{"}\mathtt{y}\text{"}]$, where $\mathtt{i}_r$ is the channel name associated to the resource. It is not difficult to see that the accompanying proof of C can be split in a proof preserving way against c. This transformation can either be applied to $\mathbb{P}$ between the resource-usage analysis and parallelization, or embedded within Alg. 4.

5.3 Implementation

We have validated our parallelization algorithm by crafting a prototype implementation on top of the existing separation logic tool, coreStar [7]. While our implementation is not intended to provide full end-to-end automated translation, it is capable of validating the algorithms on the examples given in the paper, and automatically answering the underlying theorem proving queries.

Our parallelization algorithm does not assume a shape invariant generator, except possibly to help construct the sequential proof. Soundness is independent of the "cleanliness" of the invariants (the analysis will always give a correct result, in the worst case defaulting to sequential behaviour). Our examples in coreStar [7] have been validated using automatically-generated invariants. Other efforts [11] indicate that bi-abduction works well with automatically-generated invariants produced by shape analysis, even over very large code bases.

6. Behaviour Preservation

A distinctive property of our parallelization analysis is that it enforces sequential data-dependencies in the parallelized program even if the safety proof does not explicitly reason about such dependencies. The result is that our analysis preserves the sequential

behaviour of the program: any behaviour exhibited by the parallelized program is also a behaviour that could have occurred in the original sequential program. However, there are important caveats relating to *termination* and *allocation*.

Termination. If the original sequential program does not terminate, our analysis may introduce new behaviours simply by virtue of running segments of the program that would be unreachable under a sequential schedule. To see this, suppose we have a `pfor` such that the first iteration of the loop will never terminate. Sequentially, the second iteration of the loop will never execute. However, our parallelization analysis will execute all iterations of the loop in parallel. This permits witnessing behaviours from the second (and subsequent) iterations. These behaviours were latent in the original program, and become visible only as a result of parallelization.

Allocation and disposal. If the program both allocates and disposes memory, the parallelized program may exhibit aliasing that could not occur in the original program. To see this, consider the following sequential program:

```
x=alloc(); y=alloc(); dispose(x).
```

For simplicity, we have avoided re-structuring the program to use data parallelism via `pfor`—this example could easily be encoded as such, however. Parallelization might give us the following program:

```
(x=alloc(); grant(wx); y=alloc())
       || (wait(wx); dispose(x))
```

This parallelized version of the program is race-free and obeys the required sequential ordering on data. Depending upon the implementation of the underlying memory allocator, however, `x` and `y` may be aliased if the `dispose` operation was interleaved between the two allocations. Such aliasing could not happen in the original non-parallelized version.

Either kind of new behaviour might result in further new behaviours—for example, we might have an if-statement conditional on `x==y` in the second example above. These caveats are common to our analysis and others based on separation logic—for example, see the similar discussion in [13].

Proving behaviour preservation. We now sketch a behaviour preservation result (a detailed proof is given in [8]). The theorem defines preservation in terms of an interleaved operational semantics for a core language, similar to the one described in §4.1, additionally equipped with threads, and operations on channels (such as `grant` and `wait`). Data-races are interpreted as faults in this semantics. We prove our result for a sequential thread t possibly executing concurrently with other sequential threads—this degenerates into the purely sequential case when there are no other runnable threads. Because t is sequential, it does not call `fork`; we also assume that it is guaranteed to terminate, and that it never disposes memory based on the caveats discussed above.

Theorem 7. *Let t be a sequential thread, and let t_{par} a corresponding parallelized version, equipped with thread creation operations, and synchronization actions that enforce sequential dependencies among the child threads it creates. Let $\mathcal{K}_{\mathrm{par}}$ be a terminating non-faulting (i.e., data-race free) trace of an execution of t_{par}. There exists a corresponding trace $\mathcal{K}$ derived by substituting corresponding operations in t for t_{par} such that: (1) $\mathcal{K}$ and $\mathcal{K}_{\mathrm{par}}$ begin in the same state; (2) for every thread $t' \neq t$ in $\mathcal{K}$, $\mathcal{K}$ and $\mathcal{K}_{\mathrm{par}}$ exhibit identical thread-local behaviour; and (3) the terminal state of thread t in $\mathcal{K}_{\mathrm{par}}$ is a substate of the corresponding state in $\mathcal{K}$.*

Proof sketch. We show that, under the assumption that forked child threads never wait for channels granted by their parent or later-forked child threads, any terminating non-faulting trace $\mathcal{K}_{\mathrm{par}}$ can be reordered into a sequentialized trace $\mathcal{K}_{\mathrm{seq}}$ with the same thread-local behaviour. By *sequentialized*, we mean that forked children must execute to completion before their parents can be scheduled.

We establish a simulation invariant between executions of t_{par} decorated with arbitrary calls to `newch`, `wait`, `grant` and `fork`, and executions of $t_{\mathrm{par}}^{\downarrow}$ identical to t_{par} except with all such constructs erased. The invariant establishes that every non-faulting sequentialized trace $\mathcal{K}_{\mathrm{seq}}$ of t_{par} simulates some trace $\mathcal{K}$ of $t_{\mathrm{par}}^{\downarrow}$.

We tie this result to our analysis by observing that parallelization only inserts the four constructs `newch`, `wait`, `grant` and `fork` (leaving aside issues of loop-splitting and materialization). As a result, $t_{\mathrm{par}}^{\downarrow} = t$. We also show that the way we insert signals ensures that they are ordered with respect to thread creation, as assumed in the previous step.

This establishes a behaviour-preservation result for non-faulting traces, but parallelization might introduce faults. However, parallelized programs are verified using separation logic, meaning we can assume that they do not fault when executed in a state satisfying the precondition. This completes the proof. □

One attractive property is that this proof does not place any explicit requirements on the positioning of barriers—it is sufficient that we can provide a separation-logic proof for t_{par}. The caveat on memory disposition manifests in the way we establish our simulation invariant, which necessarily assumes newly allocated data is always "fresh" and thus does not alias with any accessible heap-allocated structure. The caveat on termination is necessary to ensure we can suitably reorder forked threads to yield a sequentialized trace.

In addition to inserting barriers, our analysis mutates the program by materialising thread-local variables and splitting loops. Both of these can be performed as mutations on the initial sequential program, and neither affect visible behaviour. Loop-splitting is straightforwardly semantics-preserving, while materialized variables are only used to control barrier calls.

Theorem 7 guarantees that parallelisation does not introduce deadlocks; otherwise the simulation relation would not exist. The ordering on barriers ensures that termination in the sequential program is preserved in the resulting parallel program.

7. Refining the Predicate Domain

The structure of the sequential proof affects the success of parallelization in two ways. First, the loop invariants may allow the analysis to verify a parallelized program when it would otherwise have failed. Second, the choice of predicate domain controls how much information about resource-usage is available to the analysis. Intuitively, enriching the domain may permit a finer-grained splitting of resources, allowing redundant resources to be identified earlier or missing resources later.

To see how the choice of abstract domain influences precision and effectiveness of the analysis, consider the example discussed in §3; there, we parallelized `sum_head` using the list segment predicate lseg. An alternative predicate is $\mathsf{lsegni}(h, t, n, i)$, which extends lseg with a parameter $n \in \mathbb{Z}$ recording the *length* of the list segment, and with a parameter $i \in (0, 1]$ recording the *permission* on the list segment. If $i = 1$ the thread has read-write access; otherwise it has read access only. We define lsegni as the least predicate satisfying the following equation:

$$\mathsf{lsegni}(x,t,n,i) \triangleq (x = t \wedge n = 0 \wedge \mathsf{emp}) \vee \left(\exists v, y.\, x.\mathrm{val} \overset{i}{\mapsto} v * x.\mathrm{nxt} \overset{i}{\mapsto} y * \mathsf{lsegni}(y,t,n-1,i)\right)$$

The equivalence $\mathsf{lseg}(h,t) \iff \exists n.\, \mathsf{lsegni}(h,t,n,1)$ would allow the analysis to exploit this finer-grained predicate even if the sequential proof is written using the coarse-grained lseg predicate.

```
sum_head1(n){
  *npr = n;
  grant(i1);
  i = 1;
  sum = 0;
  x = *hd;
  while(x!=nil && i!=n){
    sum += x.val;
    i++;
    x = x.nxt;
  }
  *xpr = x;
  grant(i2);
  while(x!=nil){
    x.val = 0;
    x = x.nxt;
  }
  grant(i3);
}
```

```
sum_head2(n){
  i = 1;
  sum = 0;
  wait(i1);
  x = *hd;
  while(x!=nil && i!=n){
    if(n==*npr) wait(i2);
    sum += x.val;
    i++;
    x = x.nxt;
  }
  wait(i2);
  while(x!=nil){
    if(x==*xpr) wait(i3);
    x.val = 0;
    x = x.nxt;
  }
}
```

Figure 12. Improved parallelization of `sum_head`.

Function `sum_head` only writes to the list after traversing `n` nodes. Consequently, it needs only non-exclusive, read-only access to the first `n` nodes. (Or the entire list, if the list is shorter than `n` nodes long.) We can extend our resource usage analysis to deal with this richer predicate such that it identifies the following as a sufficient precondition for the *entire* `sum_head` function:

$$\mathtt{hd} \overset{j}{\mapsto} h * \begin{pmatrix} (\exists t.\, \mathsf{lsegni}(h,t,\mathtt{n},i) * \mathsf{lseg}(t,\mathsf{nil})) \vee \\ (\exists n'.\, \mathsf{lsegni}(h,\mathsf{nil},n',i) \wedge n' < \mathtt{n}) \end{pmatrix}$$

Without the lsegni predicate, this precondition could not be expressed. The availability of lsegni would also allow the analysis to discover that this resource is redundant at the start of `sum_head`:

$$\exists h, i, j.\ \mathtt{hd} \overset{j}{\mapsto} h * \\ \mathsf{lsegni}(h,t,\mathtt{n},i) \vee (\exists n'.\, \mathsf{lsegni}(h,\mathsf{nil},n',i) \wedge n' < \mathtt{n})$$

When `sum_head(n)` is called in a parallel-`for`, the subsequent call to `sum_head` can *immediately* read from the first `n` nodes of the list. Applying our analysis using the lsegni predicate yields the the two-thread parallelization shown in Fig. 12. `i1` signals that the first `n` nodes can be read by the subsequent iteration of `sum_head`. Similar to the transformation discussed in §3, `grant` calls can be inserted on `i2` to signal that the subsequent iteration can write to the first `n` nodes; and on `i3` to signal that the entire list can be written to. Because this parallelization calls `grant` earlier than the parallelization discussed earlier, it can consequently extract more parallelism from the original sequential program.

As this discussion reveals, our analysis is generic in the choice of abstract domain; any separation logic predicate could be used in place of lseg, for example. However, the success of *automated* parallelization is highly dependent on the power of the entailment prover in the chosen domain. The lseg domain is one of the best-developed in separation logic, and consequently automated parallelization is feasible using tools such as coreStar [7]. Other domains (such as trees) are far less developed.

8. Related Work

Resource-usage inference by abduction. We have defined an interprocedural, control-flow-sensitive analysis capable of determining the resource that will (and will not) be accessed between particular points in the program. At its core, our analysis uses abductive reasoning [11] to discover *redundancies*—that is, state used earlier in the program that will not be accessed subsequent to the current program point. Using abduction in this way was first proposed in [15], where it is used to discover memory leaks, albeit without conditionals, procedures, loops, or code specialization.

In [12], abduction is used to infer resource invariants for synchronization, using a process of counterexample-driven refinement. Our approach similarly infers resource invariants, but using a very different technique: invariants are derived from a sequential proof, and we also infer synchronization points and specialise the program to reveal synchronization opportunities.

Behaviour-preserving parallelization. We expect our resource-usage analysis can be used in other synchronization-related optimizations, but in this paper, we have used it as the basis for a parallelising transformation. This transformation is in the style of *deterministic parallelism* [3–5, 9]—although our approach does not, in fact, require determinacy of the source program. In this vein, our transformation ensures that every behaviour of the parallelized program is a behaviour of the source sequential program (modulo the caveats about allocation and termination discussed in §6).

Previous approaches to deterministic parallelism operate without the benefit of a high-level specification. This places a substantial burden on the analysis and runtime to safely extract information on resource usage and transfer—information that is readily available in a proof. As a result, these analyses tend to be much more conservative in their treatment of mutable data. Our proof-based technique gives us a general approach to splitting mutable resources; for example, by allowing the analysis to perform ad-hoc list splitting, as we do with `sum_head`.

Our approach transforms a sequential `for`-loop by running all the iterations in parallel and signalling between them. This idea has been proposed previously, for example in numerical computation [33]. The novelty in our approach lies in inferring synchronization over (portions of) complex mutable data-structures. Alternatively, we could have used a more irregular concurrency annotation, for example safe futures [27], or an unordered parallel-`for`, as in the Galois system [30]. In the former case, our resource-usage analysis would be mostly unchanged, but our parallelized program would construct a set of syntactically-distinct threads, rather than a pipeline of identical threads. Removing ordering between iterations, as in the latter case, would mean replacing ordered `grant`-`wait` pairs with conventional locks, and would introduce an obligation to show that locks were always acquired together, as a set.

Proof-driven parallelization. A insight central to our approach is that a separation logic proof expresses data dependencies for *parts* of a program, as well as for the whole program. These internal dependencies can be used to inject safe parallelism. This insight is due to [13, 31] and [24], both of which propose parallelization analyses based on separation logic. The analyses proposed in these papers are much more conservative than ours, in that they discover independence which already exists between commands of the program. They do not insert synchronization constructs, and consequently cannot enforce sequential dependencies among concurrent computations that share and modify state. Indeed, [31] does not consider any program transformations, since the goal of that work is to identify memory separation of different commands, while [24] expresses optimizations as reordering rewrites on proof trees.

Bell *et. al* [1] construct a proof of an *already-transformed* multithreaded program parallelized by the DSWP transformation [29]. This approach assumes a specific pattern of (linear) dependencies in the while-loop consistent with DSWP, a specific pattern of sequential proof, and a fixed number of threads. In our `sum_head` example, the outermost (parallelising) loop contains two successive inner loops, while the example in Fig. 2 illustrates how the technique can deal with interprocedural and control-flow sensitive dependencies. In both cases, the resulting parallelization is specialized to inject synchronization primitives to enforce sequential de-

pendencies. We believe examples like these do not fall within the scope of either DSWP or the proof techniques supported by [1].

Outside separation logic, Deshmukh *et. al* [14] propose an analysis which augments a sequential library with synchronization. This approach takes as input a sequential proof expressing the correctness criteria for the library, and generates synchronization ensuring this proof is preserved if methods run concurrently. A basic assumption is that the sequential proof represents all the properties that must be preserved. In contrast, we also preserve sequential order on access to resources. Consequently, Deshmukh et al. permit parallelizations that we would prohibit, and can introduce new behaviours into the parallelized program. Another difference is that [14] derives a linearizable implementation given a sequential specification in the form of input-output assertions; because they do not consider *specialization* of multiple instances of the library running concurrently, it is unclear how their approach would deal with transformations of the kind we use for `sum_head`.

Separation logic and concurrency. Separation logic is essential to our approach. It allows us to localise the behaviour of a program fragment to precisely the resources it accesses. Our proofs are written using concurrent separation logic [28]. CSL has been extended to deal with dynamically-allocated locks [19, 23, 25], re-entrant locks [20], and primitive channels [1, 22, 26, 34]. Sequential tools for separation logic have achieved impressive scalability—for example [11] has verified a large proportion of the Linux kernel. Our work can be seen as an attempt to leverage the success of such sequential tools. Our experiments are built on coreStar [7, 16], a language-independent proof tool for separation logic.

The parallelization phase of our analysis makes use of the specifications for parallelization barriers proposed in [17]. That paper defined high-level specifications representing the abstract behaviour of barriers, and verified those specifications against the barriers' low-level implementations. However, it assumed that barriers were already placed in the program, and made no attempt to infer barrier positions. In contrast, we assume the high-level specification, and define an analysis to insert barriers. The semantics of barriers used in that paper and here was initially proposed in [27].

Acknowledgments

This work was supported by the Gates trust, by EPSRC grant EP/H010815/1, and by NSF grant CCF-0811631. Thanks to Dino Distefano, Matthew Parkinson, Mohammad Raza, John Wickerson and the anonymous reviewers for comments and suggestions.

References

[1] C. J. Bell, A. Appel, and D. Walker. Concurrent Separation Logic for Pipelined Parallelization. In *SAS*, pages 151–166, 2009.

[2] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. *SIGPLAN Not.*, 45(3):53–64, 2010.

[4] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*, pages 81–96, 2010.

[5] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, pages 91–116, 2009.

[6] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission Accounting in Separation Logic. In *POPL*, pages 259–270, 2005.

[7] M. Botinčan, D. Distefano, M. Dodds, R. Griore, Naudžiūnienė, and M. Parkinson. coreStar: The Core of jStar. In *Boogie*, pages 65–77, 2011.

[8] M. Botinčan, M. Dodds, and S. Jagannathan. Resource-Sensitive Synchronization Inference by Abduction. Technical Report 808, University of Cambridge Computer Laboratory, 2011.

[9] J. Burnim and K. Sen. Asserting and Checking Determinism for Multithreaded Programs. *Commun. ACM*, 53:97–105, June 2010.

[10] C. Calcagno, P. W. O'Hearn, and H. Yang. Local Action and Abstract Separation Logic. In *LICS*, pages 366–378, 2007.

[11] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-Abduction. In *POPL*, pages 289–300, 2009.

[12] C. Calcagno, D. Distefano, and V. Vafeiadis. Bi-abductive Resource Invariant Synthesis. In *APLAS*, pages 259–274, 2009.

[13] B. Cook, S. Magill, M. Raza, J. Simsa, and S. Singh. Making Fast Hardware with Separation Logic, 2010. Unpublished, `http://cs.cmu.edu/~smagill/papers/fast-hardware.pdf`.

[14] J. V. Deshmukh, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Logical Concurrency Control from Sequential Proofs. In *ESOP*, pages 226–245, 2010.

[15] D. Distefano and I. Filipović. Memory Leaks Detection in Java by Bi-abductive Inference. In *FASE*, pages 278–292, 2010.

[16] D. Distefano and M. J. Parkinson J. jStar: Towards Practical Verification for Java. In *OOPSLA*, pages 213–226, 2008.

[17] M. Dodds, S. Jagannathan, and M. J. Parkinson. Modular Reasoning for Deterministic Parallelism. In *POPL*, pages 259–270, 2011.

[18] T. Elmas, S. Qadeer, and S. Tasiran. A Calculus of Atomic Actions. In *POPL*, pages 2–15, 2009.

[19] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local Reasoning for Storable Locks and Threads. In *APLAS*, pages 19–37, 2007.

[20] C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's Reentrant Locks. In *APLAS*, pages 171–187, 2008.

[21] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Morgan-Claypool, 2010.

[22] C. A. R. Hoare and P. W. O'Hearn. Separation Logic Semantics for Communicating Processes. *ENTCS*, 212:3–25, 2008.

[23] A. Hobor, A. W. Appel, and F. Zappa Nardelli. Oracle semantics for concurrent separation logic. In *ESOP*, 2008.

[24] C. Hurlin. Automatic Parallelization and Optimization of Programs by Proof Rewriting. In *SAS*, pages 52–68, 2009.

[25] B. Jacobs and F. Piessens. Modular full functional specification and verification of lock-free data structures. Technical Report CW 551, Katholieke Universiteit Leuven, Dept. of Computer Science, 2009.

[26] K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free Channels and Locks. In *ESOP*, pages 407–426, 2010.

[27] A. Navabi, X. Zhang, and S. Jagannathan. Quasi-static Scheduling for Safe Futures. In *PPoPP*, pages 23–32. ACM, 2008.

[28] P. W. O'Hearn. Resources, Concurrency and Local Reasoning. *TCS*, 375:271–307, 2007.

[29] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *MICRO*, pages 105–118, 2005.

[30] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The Tao of Parallelism in Algorithms. In *PLDI*, pages 12–25, 2011.

[31] M. Raza, C. Calcagno, and P. Gardner. Automatic Parallelization with Separation Logic. In *ESOP*, pages 348–362, 2009.

[32] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, pages 55–74, 2002.

[33] P. Tang, P. Tang, J. N. Zigman, and J. N. Zigman. Reducing Data Communication Overhead for DOACROSS Loop Nests. In *International Conference on Supercomputing*, pages 44–53, 1993.

[34] J. Villard, É. Lozes, and C. Calcagno. Tracking Heaps That Hop with Heap-Hop. In *TACAS*, pages 275–279, 2010.