Refinement of Parallel Algorithms down to LLVM

applied to practically efficient parallel sorting

Peter Lammich[1*]

[1*]Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente Enschede Netherlands.

Corresponding author(s). E-mail(s): p.lammich@utwente.nl;

Abstract

We present a stepwise refinement approach to develop verified parallel algorithms, down to efficient LLVM code. The resulting algorithms' performance is competitive with their counterparts implemented in C/C++. Our approach is backwards compatible with the Isabelle Refinement Framework, such that existing sequential formalizations can easily be adapted or re-used. As case study, we verify a parallel quicksort algorithm that is competitive to unverified state-of-the-art algorithms.

Keywords: Isabelle,Concurrent Separation Logic,Parallel Sorting,LLVM

1 Introduction

We present a stepwise refinement approach to develop verified and efficient parallel algorithms. Our method can verify total correctness down to LLVM intermediate code. The resulting verified implementations are competitive with state-of-the-art unverified implementations. Our approach is backwards compatible to the Isabelle Refinement Framework (IRF) [1], a powerful tool to verify efficient sequential software, such as model checkers [2–4], SAT solvers [5–7], or graph algorithms [8–10]. This paper adds parallel execution to the IRF's toolbox, without invalidating the existing formalizations, which can now be used as sequential building blocks for parallel algorithms, or be modified to add parallelization.

As a case study, we verify total correctness of a parallel quicksort algorithm, re-using an existing verification of state-of-the-art sequential sorting algorithms [11]. Our verified parallel sorting algorithm is competitive to state-of-the-art parallel sorting algorithms from GNU's C++ standard library and the Boost C++ Libraries.

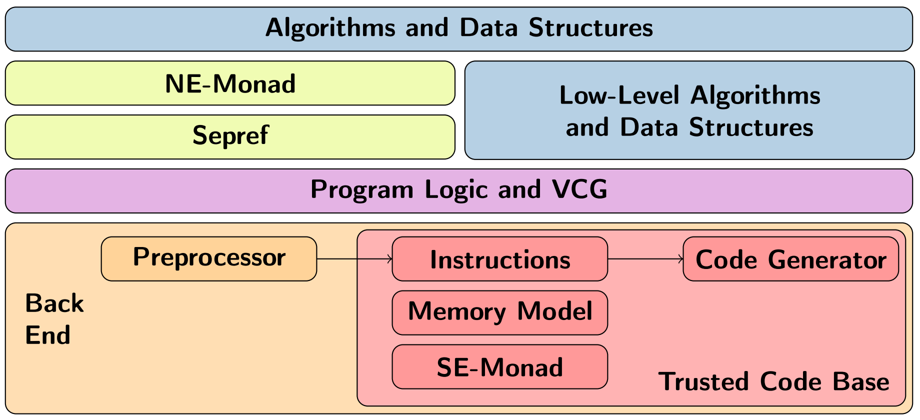

Fig. 1: Components of the IRF, with focus on the back end.

This paper is an extended version of our ITP 2023 paper [12]. The main new contribution is a verified parallel partitioning algorithm, which significantly improves the efficiency and scalability of our sorting algorithm from [12]. To this end, we added a description of the interval list data structure (Section 4.3) and the `with_idxs` combinator (Section 4.4), which are required by the parallel partitioner. The parallel partitioning algorithm itself is described in Section 5.3, and Section 5.5 contains updated and more extensive benchmarks.
The formalization is available at www21.in.tum.de/~lammich/isabelle_llvm_par.

1.1 Overview

This paper is based on the Isabelle Refinement Framework, a continuing effort to verify efficient implementations of complex algorithms, using stepwise refinement techniques [1, 13–17]. Figure 1 displays the components of the Isabelle Refinement Framework. The back end layer handles the translation from Isabelle/HOL to the actual target language. The instructions of the target language are shallowly embedded into Isabelle/HOL, using a state-error (SE) monad. An instruction with undefined behaviour, or behaviour outside our supported fragment, raises an error. The state of the monad is the memory, represented via a memory model. The code generator translates the instructions to actual code. These components form the trusted code base, while all the remaining components of the Isabelle Refinement Framework generate proofs. In the back-end, the preprocessor transforms expressions to the syntactically restricted format required by the code generator, proving semantic equality of the original and transformed expression. While there exist back ends for purely functional code [14, 15], and sequential imperative code [1, 16], this paper describes a back end for parallel imperative LLVM code (Section 2).

On top of the back-end, a program logic is used to prove programs correct. It uses separation logic, and provides automation like a verification condition generator (VCG). In Section 3, we describe our formalization of concurrent separation logic [18], and our VCG.

At the level of the program logic and VCG, our framework can be used to verify simple low-level algorithms and data structures, like dynamic arrays and linked lists. More complex developments typically use a stepwise refinement approach, starting at purely functional programs modelled in a nondeterminism-error (NE) monad [14]. A semi-automatic refinement procedure (Sepref [1, 16]) translates from the purely functional code to imperative code, refining abstract functional data types to concrete imperative ones. In Section 4, we describe our extensions to support refinement to parallel executions, and a fine-grained tracking of pointer equalities, required to parallelize computations that work on disjoint parts of the same array.

Using our approach, complex algorithms and data structures can be developed and refined to optimized efficient code. The stepwise refinement ensures a separation of concerns between high-level algorithmic ideas and low-level optimizations. We have used this approach to verify a wide range of practically efficient algorithms [2–11]. In Section 5, we use our techniques to verify a parallel sorting algorithm, with competitive performance wrt. unverified state-of-the-art algorithms.

Section 6 concludes the paper and discusses related and future work.

1.2 Notation

We mainly use Isabelle/HOL notation, with some shortcuts and adaptations for presentation in a paper. In this section, we give examples of the more unusual notations: for implication, we always write $\Longrightarrow$, e.g., $\forall x.\ x<4 \Longrightarrow x\leq 3$ (in Isabelle, one must use $\longrightarrow$ here).

Type variables are $'a, 'b, \ldots$ and a function types are written curried as $'a \Rightarrow 'b \Rightarrow 'c$. Types can be annotated to any symbol or term, e.g., $(x::nat) < 4 \Longrightarrow x\leq 3$. Function application is written as $f\ x$, and function update is $f(x:=y)$. For definitions, we use $\equiv$, e.g., $f\ x \equiv x+1$.

Algebraic datatypes are written as $'a\ option \equiv Some\ (the{:}\ 'a) \mid None$. This also defines the selector function $the :: 'a\ option \Rightarrow 'a$. Values of the tuple type $'a_1 \times \ldots \times 'a\_n$ are written as $(a_1,\ldots,a\_n)$. Lists have type $'a\ list$, and we write $[]$ for the empty list, $x\#xs$ for the list with head x and tail xs, and $xs_1 @ xs_2$ for list concatenation. We also use the list notation $[1,2,3]$. Then length of list xs is $|xs|$.

The empty set is $\{\} :: 'a\ set$. The set $\{l..<h\}$ contains all elements between l inclusive and h exclusive, and the set $\{l..\}$ contains all elements greater than or equal to l. Disjoint union is written as $\dot\cup$. We only write $\dot\cup$ if it is clear from the context where the disjointness constraint belongs to. Otherwise, we will explicitly write, e.g. $s = s_1 \cup s_2 \ \wedge\ s_1 \cap s_2 = \{\}$. The cardinality of the finite set s is $|s|$.

Lambda abstraction is written as $\lambda x.\ x+1$. When clear from the context, we omit the λ, e.g. `spec` $x.\ x>42$. We also use Haskell-like sections for infix operators, e.g. $(+1)$ for $\lambda x.\ x+1$. Also, we can use underscores to indicate the parameter positions, e.g., $\{\_..<\_\}$ for $\lambda l\ h.\ \{l..<h\}$.

2 A Back End for LLVM with Parallel Execution

We formalize a semantics for parallel execution, shallowly embedded into Isabelle/HOL. As for the existing sequential back ends [1, 16], the shallow embedding is key to the flexibility and feasibility of the approach. The main idea is to make an execution report its accessed memory, and use this information to raise an error when joining executions that would have exhibited a data race. We use this to model an instruction that calls two functions in parallel, and waits until both have returned.

2.1 State-Nondeterminism-Error Monad with Access Reports

We define the underlying monad in two steps. We start with a nondeterminism-error monad, and then lift it to a state monad and add access reports. Defining a nondeterminism-error monad is straightforward in Isabelle/HOL:

$'a\ neM \equiv$ `spec` $('a \Rightarrow bool)\ |$ `fail`
`return` $x \equiv$ `spec` $(\lambda r.\ r{=}x)$
`bind fail` $f \equiv$ `fail`
`bind` (`spec` P) $f \equiv$ `if` $\exists x.\ P\ x \wedge f\ x =$ `fail then fail`
    `else spec` $(\lambda r.\ \exists x\ Q.\ P\ x \wedge f\ x =$ `spec` $Q \wedge Q\ r)$

A program either fails, or yields a possible set of results (`spec` P), described by its characteristic function P. The `return` operation yields exactly one result, and `bind` combines all possible results, failing if there is a possibility to fail. We use the notation $x{\leftarrow}m;\ f\ x$ for `bind` $m\ (\lambda x.\ f\ x)$, and $m_1;\ m_2$ for `bind` $m_1\ (\lambda\_.\ m_2)$.

Now assume that we have a state (memory) type $'\mu$, and an access report type $'\rho$, which forms a monoid $(0,+)$. With this, we define our state-nondeterminism-error monad with access reports, just called M for brevity:

$'x\ M \equiv\ '\mu \Rightarrow ('x \times\ '\rho \times\ '\mu)\ neM$
$\texttt{return}_M\ x\ \mu \equiv \texttt{return}_{ne}\ (x,0,\mu)$
$\texttt{bind}_M\ m\ f\ \mu \equiv (x_1,r_1,\mu) \leftarrow m\ \mu;\ (x_2,r_2,\mu) \leftarrow f\ x_1\ \mu;\ \texttt{return}_{ne}\ (x_2,r_1{+}r_2,\mu)$

Here, `return` does not change the state, and reports no accesses (0), and `bind` sequentially composes the executions, threading through the state μ and adding up the access reports r_1 and r_2.

Typically, the access report will contain read and written addresses, such that data races can be detected. Moreover, if parallel executions can allocate memory, we must detect those executions where the memory manager allocated the same block in both parallel strands. As we assume a thread safe memory manager, those *infeasible* executions can safely be ignored. Let $feasible :: '\rho \Rightarrow '\rho \Rightarrow bool$ and $norace :: '\rho \Rightarrow '\rho \Rightarrow bool$ be symmetric predicates, and let $combine :: ('\rho \times '\mu) \Rightarrow ('\rho \times '\mu) \Rightarrow ('\rho \times '\mu)$ be a commutative operator to combine two pairs of access reports and states. Then, we define a parallel composition operator for M:

$(m_1\ ||\ m_2)\ \mu \equiv$
  $(x_1,r_1,\mu_1) \leftarrow m_1\ \mu;\ (x_2,r_2,\mu_2) \leftarrow m_2\ \mu;$ — *execute both strands*
  `assume` $feasible\ \rho_1\ \rho_2;$ — *ignore infeasible combinations*
  `assert` $norace\ \rho_1\ \rho_2;$ — *fail on data race*
  $\texttt{return}_{ne}\ ((x_1,x_2),\ combine\ (\rho_1,\mu_1)\ (\rho_2,\mu_2))$ — *combine results*

`assume` $P \equiv$ `if` P `then return` $()$ `else spec` $(\lambda\_.\ False)$
`assert` $P \equiv$ `if` P `then return` $()$ `else fail`

Here, we use `assume` to ignore infeasible executions, and `assert` to fail on data races. Note that, if one parallel strand fails, and the other parallel strand has no possible results (`spec` (λ_. *False*)), the behaviour of the parallel composition is not clear. For this reason, we fix an invariant $invar_M$ :: ($'\mu \Rightarrow ('x \times '\rho \times '\mu)$ *neM*) $\Rightarrow$ *bool*, which implies that every non-failing execution has at least one possible result. We define the actual type M as the subtype satisfying $invar_M$. Thus, we have to prove that every combinator and instruction of our semantics preserves the invariant, which is an important sanity check. As additional sanity check, we prove symmetry of parallel composition:

$m_1 \mid\mid m_2 = mswap\ (m_2 \mid\mid m_1)$ *where* $mswap\ m \equiv (x_1,x_2) \leftarrow m;$ `return` (x_2,x_1)

2.2 Memory Model

Our memory model supports blocks of values, where values can be integers, structures, or pointers into a block:

datatype *addr* ≡ *ADDR* (*bidx: nat*) (*idx: nat*)
datatype *ptr* ≡ *PTR_NULL* | *PTR_ADDR* (*the_addr: addr*)
datatype *val* ≡ *LL_INT lint* | *LL_STRUCT val list* | *LL_PTR ptr*

datatype *block* ≡ *FRESH* | *FREED* | *is_alloc: ALLOC* (*vals: val list*)
typedef *memory* ≡ { μ :: *nat* ⇒ *block*. *finite* {*b*. μ *b* ≠ *FRESH*} }

A block is either fresh, freed, or allocated, and a memory is a mapping from block indexes to blocks, such that only finitely many blocks are not fresh. Every block's state transitions from fresh to allocated to freed. This avoids ever reusing the same block, and thus allows us to semantically detect use after free errors. Every program execution can only allocate finitely many blocks, such that we will never run out of fresh blocks[1]. An allocated block contains an array of values, modelled as a list. Thus, an address consists of a block number, and an index into the array.

To access and modify memory, we define the functions *valid*, *get*, and *put*:

$valid\ \mu\ (ADDR\ b\ i) \equiv is\_alloc\ (\mu\ b) \wedge i<|vals\ (\mu\ b)|$
$get\ \mu\ (ADDR\ b\ i) \equiv vals\ (\mu\ b)\ !\ i$
$put\ \mu\ (ADDR\ b\ i)\ x \equiv \mu(b := ALLOC\ ((vals\ (\mu\ b))[i{:=}x]))$

where $|xs|$ is the length of list xs, $xs!i$ returns the ith element of list xs, and $xs[i{:=}x]$ replaces the ith element of xs by x.

Note that our LLVM semantics does not support conversion of pointers to integers, nor comparison or difference of pointers to different blocks. This way, a program cannot see the internal representation of a pointer, and we can choose a simple abstract representation, while being faithful wrt. any actual memory manager implementation.

2.3 Access Reports

We now fix the state of the M-monad to be memory, and the access reports to be tuples (r,w,a,f) of read and written addresses, as well as sets of allocated and freed blocks:

[1]If the actual system does run out of memory, we will terminate the program in a defined way.

$acc \equiv addr\ set \times addr\ set \times nat\ set \times nat\ set$
$0 \equiv (\ \{\},\{\},\{\},\{\}\ )$
$(r_1,w_1,a_1,f_1) + (r_2,w_2,a_2,f_2) \equiv (\ r_1 \cup r_2,\ w_1 \cup w_2,\ a_1 \cup a_2,\ f_1 \cup f_2\ )$

Two parallel executions are feasible if they did not allocate the same block. They have a data race if one execution accesses addresses or blocks modified by the other:

$feasible\ (r_1,w_1,a_1,f_1)\ (r_2,w_2,a_2,f_2) \equiv a_1 \cap a_2 = \{\}$

```
norace $(r_1,w_1,a_1,f_1)\ (r_2,w_2,a_2,f_2) \equiv$
  let $m_1 = w_1 \cup \{\ ADDR\ b\ i.\ b \in a_1 \cup f_1\ \}$ in
  let $m_2 = w_2 \cup \{\ ADDR\ b\ i.\ b \in a_2 \cup f_2\ \}$ in
    $(r_1 \cup m_1) \cap m_2 = \{\} \quad \wedge \quad m_1 \cap (r_2 \cup m_2) = \{\}$
```

The combine function joins the access reports and memories, preferring allocated over fresh, and freed over allocated memory. When joining two allocated blocks, the written addresses from the access report are used to join the blocks. We skip the rather technical definition of combine, and just state the relevant properties: Let $\rho_1=(r_1,w_1,a_1,f_1)$ and $\rho_2=(r_2,w_2,a_2,f_2)$ be feasible and race free access reports, and $\mu_1,\ \mu_2$ be memories that have evolved from a common memory μ, consistently with the access reports $\rho_1,\ \rho_2$. Let $(\rho',\mu') = combine\ (\rho_1,\mu_1)\ (\rho_2,\mu_2)$. Then

(1) $\mu'\ b = FRESH \longleftrightarrow \mu\ b = FRESH \wedge b \notin a_1 \cup a_2$
(2) $is\_alloc\ (\mu'\ b) \longleftrightarrow (is\_alloc\ (\mu\ b) \vee b \in a_1 \cup a_2) \wedge b \notin f_1 \cup f_2$
(3) $\mu'\ b = FREED \longleftrightarrow \mu\ b = FREED \vee b \in f_1 \cup f_2$

Moreover, for all addresses $addr = ADDR\ b\ i$ with $valid\ \mu'\ addr$:

(4) $addr \in w_1 \vee b \in a_1 \implies get\ \mu'\ addr = get\ \mu_1\ addr$
(5) $addr \in w_2 \vee b \in a_2 \implies get\ \mu'\ addr = get\ \mu_2\ addr$
(6) $addr \notin w_1 \cup w_2 \wedge b \notin a_1 \cup a_2 \implies get\ \mu'\ addr = get\ \mu\ addr$

The properties (1)–(3) define the state of blocks in the combined memory: a fresh block in μ' was fresh already in μ, and has not been allocated (1); an allocated block was already allocated or has been allocated, but has not been freed (2); and a freed block was already freed, or has been freed (3). The properties (4)–(6) define the content: addresses written or allocated in the first or second execution get their content from μ_1 (4) or μ_2 (5) respectively. Addresses not written nor allocated at all keep their original content (6).

2.4 The Interface of the M-Monad

The invariant for M states that blocks transition only from fresh to allocated to free, allocated blocks never change their size, and the access report matches the observable state change (*consistent*). It also states, that for each finite set of blocks B, there is an execution that does not allocate blocks from B. The latter is required to show that we always find feasible parallel executions:

$invar_M\ c \equiv \forall \mu\ P.\ c\ \mu = \mathtt{spec}\ P \implies$
$\quad (\forall x\ \rho\ \mu'.\ P\ (x,\rho,\mu') \implies consistent\ \mu\ \rho\ \mu')$
$\wedge\ (\forall B.\ finite\ B \implies (\exists x\ \rho\ \mu'.\ P\ (x,\rho,\mu') \wedge \rho.a \cap B = \{\}\ ))$

To define functions in the M-monad, we have to show that they satisfy this invariant. For `return` and `bind`, this is straightforward. The proof for the parallel operator is slightly more involved, using the properties of *combine*, and the invariant for the operands to obtain a feasible parallel execution.

Moreover, the M monad provides the memory management functions *Mmalloc*, *Mfree*, *Mload*, *Mstore*, and *Mvalid_addr*. The latter function checks if a given address is valid, and is used to check if pointer arithmetic can be performed on that address. Currently, it behaves like loading from that address, in particular it does not support pointers one past the end of an allocated block. We leave integration of such pointers to future work.

Example 1 (Memory Allocation)**.** To define memory allocation in *M*, we first define the allocation function in the underlying nondeterminism-error monad:

```
malloc vs μ ≡
  b ← spec b. is_FRESH μ b;
  return (b, ({},{},{b},{}), μ(b:=ALLOC vs))
```

This function selects an arbitrary fresh block *b*, and initializes it with the given list *vs* of values. It returns the allocated block, an access report for the allocation, and the updated memory.

We then show that *malloc* satisfies the invariant of M: we correctly report the allocated block, and we can select any fresh block. As our memory model guarantees an infinite supply of fresh blocks, any finite set of blocks can be avoided:

invarM_malloc: invarM (*malloc vs*)

Finally, we define the corresponding function in the M monad, using Isabelle's lifting and transfer package [19]:

lift_definition *Mmalloc :: val list ⇒ M* **is** *malloc* **by** (*rule invarM_malloc*)

The other memory management functions are defined analogously.

2.5 LLVM Instructions

Based on the M-monad, we define shallowly embedded LLVM instructions. For most instructions, this is analogous to the sequential case [1]. Additionally, we define an instruction for parallel function call:

llc_par f g a b ≡ *f a* || *g b*

The code generator only accepts this, if *f* and *g* are constants (i.e., function names). It then generates some type-casting boilerplate, and a call to an external *parallel* function, which we implement using the Threading Building Blocks [20] library:

```
void parallel(void (*f1)(void*), void (*f2)(void*), void *x1, void *x2) {
  tbb::parallel_invoke([=]{f1(x1);}, [=]{f2(x2);}); }
```

I.e., the two functions *f1*(*x1*) and *f2*(*x2*) are called in parallel. The generated boilerplate code sets up *x1* and *x2* to point to both, the actual arguments and space for the results.

3 Parallel Separation Logic

In the previous section, we have defined a shallow embedding of LLVM programs into Isabelle/HOL. We now reason about these programs, using separation logic.

3.1 Separation Algebra

In order to reason about memory with separation logic, we define an abstraction function from the memory into a separation algebra [21]. Separation algebras formalize the intuition of combining disjoint parts of memory. They come with a *zero* (0) that describes the empty part, a *disjointness predicate* $a\#b$ describing that the parts a and b do not overlap, and a *disjoint union* $a+b$ that combines two disjoint parts. For the exact definition of a separation algebra, we refer to [21, 22]. We note that separation algebras naturally extend over functions and pairs in a pointwise manner.
Example 2. (Trivial Separation Algebra) The type $'a\ option = None \mid Some\ 'a$ forms a separation algebra with:

$$0 \equiv None \qquad a \# b \equiv a{=}0 \vee b{=}0 \qquad a + 0 \equiv a \qquad 0 + b \equiv b$$

Intuitively, this separation algebra does not allow for combination of contents, except if one side is zero. While it is not very useful on its own, the trivial separation algebra is a useful building block for more complex separation algebras.
For our memory model, we define the following abstraction function:

$$\alpha :: memory \Rightarrow (addr \Rightarrow val\ option) \times (nat \Rightarrow nat\ option)$$
$$\alpha\ \mu \equiv (\alpha_m\ \mu,\ \alpha_b\ \mu)$$

$$\alpha_m\ \mu\ addr \equiv \texttt{if}\ valid\ \mu\ addr\ \texttt{then}\ Some\ (get\ \mu\ addr)\ \texttt{else}\ 0$$
$$\alpha_b\ \mu\ b \equiv \texttt{if}\ is\_alloc\ (\mu\ b)\ \texttt{then}\ Some\ (|vals\ (\mu\ b)|)\ \texttt{else}\ 0$$

An abstract memory $\alpha\ \mu$ consists of two parts: $\alpha_m\ \mu$ is a map from addresses to the values stored there. It is used to reason about load and store operations. $\alpha_b\ \mu$ is a map from block indexes to the sizes of the corresponding blocks. It is used to ensure that one owns all addresses of a block when freeing it.
We continue to define a separation logic: assertions are predicates over separation algebra elements. The basic connectives are defined as follows:

$$false\ a \equiv False \qquad true\ a \equiv True \qquad \square\ a \equiv a{=}0$$
$$(P{*}Q)\ a \equiv \exists a_1\ a_2.\ a_1 \# a_2 \wedge a = a_1 + a_2 \wedge P\ a_1 \wedge Q\ a_2$$

That is, the assertion *false* never holds and the assertion *true* holds for all abstract memories. The empty assertion $\square$ holds for the zero memory, and the separating conjunction $P{*}Q$ holds if the memory can be split into two disjoint parts, such that P holds for one, and Q holds for the other part. The lifting assertion $\uparrow\phi$ holds iff the Boolean value ϕ is true:

$$\uparrow\phi \equiv \texttt{if}\ \phi\ \texttt{then}\ \square\ \texttt{else}\ false$$

It is used to lift plain logical statements into separation logic assertions owning no memory. When clear from the context, we omit the $\uparrow$-symbol, and just mix plain statements with separation logic assertions.

3.2 Weakest Preconditions and Hoare Triples

We define a *weakest precondition* predicate directly via the semantics:

$$\begin{aligned} &wp\ m\ Q\ \mu \equiv \texttt{case}\ m\ \mu\ \texttt{of spec}\ Q' \Rightarrow \forall x\ \rho\ \mu'.\ Q'\ (x,\rho,\mu') \implies Q\ x\ \rho\ \mu' \\ &\qquad\qquad\qquad\qquad\quad \mid \texttt{fail} \Rightarrow \mathit{False} \end{aligned}$$

That is, $wp\ m\ Q\ \mu$ holds, iff program m run on memory μ does not fail, and all possible results (return value x, access report ρ, new memory μ') satisfy the *postcondition* Q.

To set up a verification condition generator based on separation logic, we standardize the postcondition: the reported memory accesses must be disjoint from some abstract memory *amf*, called the *frame*. We define the *weakest precondition with frame*:

$$wpf\ amf\ c\ Q\ \mu \equiv wp\ c\ (\lambda x\ \rho\ \mu'.\ Q\ x\ \mu' \wedge disjoint\ \rho\ amf)\ \mu$$

$$\begin{aligned} &disjoint\ (r,w,a,f)\ (m,b) \equiv (\forall addr.\ m\ addr \neq 0 \implies addr \notin r \cup w \wedge addr.bidx \notin f) \\ &\qquad\qquad\qquad\qquad\qquad \wedge (\forall i.\ b\ i \neq 0 \implies i \notin f) \end{aligned}$$

that is, when executed on memory μ, the program c does not fail, every return value x and new memory μ' satisfies Q, and no memory described by the frame *amf* is accessed.

Equipped with *wpf*, we define a Hoare-triple:

$$ABS\ amf\ P\ \mu \equiv \exists am.\ am\ \#\ amf\ \wedge\ \alpha\ \mu = am{+}amf\ \wedge\ P\ am$$

$$ht\ P\ c\ Q \equiv \forall \mu\ amf.\ ABS\ amf\ P\ \mu \implies wpf\ amf\ c\ (\lambda x\ \mu'.\ ABS\ amf\ (Q\ x)\ \mu')\ \mu$$

The predicate $ABS\ amf\ P\ \mu$ specifies that the abstract memory $\alpha\ \mu$ can be split into a part *am* and the given frame *amf*, such that *am* satisfies the precondition P. A Hoare-triple $ht\ P\ c\ Q$ specifies that for all memories and frames for which the precondition holds ($ABS\ amf\ P\ \mu$), the program will succeed, not using any memory of the frame, and every result will satisfy the postcondition wrt. the original frame ($ABS\ amf\ (Q\ x)\ \mu'$).

3.3 Verification Condition Generator

The verification condition generator is implemented as a proof tactic that works on subgoals of the form:

$$ABS\ amf\ P\ \mu \wedge \ldots \implies wpf\ amf\ c\ Q\ \mu$$

The tactic is guided by the syntax of the command c. Basic monad combinators are broken down using the following rules:

$$\begin{aligned} &Q\ r\ \mu \implies wpf\ amf\ (\texttt{return}\ r)\ Q\ \mu \\ &wpf\ amf\ m\ (\lambda x.\ wpf\ amf\ (f\ x)\ Q)\ \mu \implies wpf\ amf\ (\{x \leftarrow m;\ f\ x\})\ Q\ \mu \end{aligned}$$

For other instructions and user defined functions, the VCG expects a Hoare-triple to be already proved. It then uses the following rule:

$$\begin{aligned} &ht\ P\ c\ Q \wedge ABS\ amf\ P'\ \mu && \text{— \textit{match Hoare triple and current state}} \\ &\wedge\ P' \vdash P{*}F && \text{— \textit{infer frame}} \\ &\wedge\ (\textstyle\bigwedge r\ \mu.\ ABS\ amf\ (Q\ r * F)\ \mu \implies Q'\ r\ \mu) && \text{— \textit{continue with postcondition}} \\ &\implies wpf\ amf\ c\ Q'\ \mu \end{aligned}$$

To process a command c, the first assumption is instantiated with the Hoare-triple for c, and the second assumption with the assertion P' for the current state. Then, a simple syntactic heuristics infers a frame F and proves that the current assertion P' entails the required precondition P and the frame. Finally, verification condition generation continues with the postcondition Q and the frame as current assertion.

3.4 Hoare-Triples for Instructions

To use the VCG to verify LLVM programs, we have to prove Hoare triples for the LLVM instructions. For parallel calls, we prove the well-known disjoint concurrency rule [18]:

$$ht\ P_1\ c_1\ Q_1 \wedge ht\ P_2\ c_2\ Q_2 \implies ht\ (P_1 * P_2)\ (par\ c_1\ c_2)\ (\lambda(r_1,r_2).\ Q_1\ r_1 * Q_2\ r_2)$$

That is, commands with disjoint preconditions can be executed in parallel.

For memory operations, we prove:

$$\models \{n{\neq}0\}\ ll\_malloc\ TYPE('a)\ n\ \{\lambda p.\ range\ \{0..{<}n\}\ p\ (\lambda\_.\ init) * b\_tag\ p\ n\}$$
$$\models \{range\ \{0..{<}n\}\ p\ f * b\_tag\ p\ n\}\ ll\_free\ p\ \{\lambda\_.\ \square\}$$
$$\models \{pto\ p\ x\}\ ll\_load\ p\ \{\lambda r.\ r{=}x * pto\ p\ x\}$$
$$\models \{pto\ p\ y\}\ ll\_store\ x\ p\ \{\lambda\_.\ pto\ p\ x\}$$

Here $b\_tag\ p\ n$ asserts that p points to the beginning of a block of size n, and $range\ I\ p\ f$ describes that for all $i \in I$, $p + i$ points to value $f\ i$. Intuitively, ll_malloc creates a block of size n, initialized with the default $init$ value, and a tag. If one possesses both, the whole block and the tag, it can be deallocated by free. The rules for load and store are straightforward, where $pto\ p\ x$ describes that p points to value x.

4 Refinement for (Parallel) Programs

At this point, we have described a separation logic framework for parallel programs in LLVM. It is largely backwards compatible with the framework for sequential programs described in [1], such that we could easily port the algorithms formalized there to our new framework. The next step towards verifying complex programs is to set up a stepwise refinement framework. In this section we describe the refinement infrastructure of the Isabelle Refinement Framework.

4.1 Abstract Programs

Abstract programs are shallowly embedded into the nondeterminism error monad $'a\ neM$ (cf. Section 2.1). They are purely functional and have no notion of parallel execution. We define a *refinement ordering* on neM:

$$\texttt{spec}\ P \leq \texttt{spec}\ Q \equiv \forall x.\ P\ x \implies Q\ x \qquad \texttt{fail} \not\leq \texttt{spec}\ Q \qquad m \leq \texttt{fail}$$

Intuitively, $m_1 \leq m_2$ means that m_1 returns fewer possible results than m_2, and may only fail if m_2 may fail. Note that $\leq$ is a complete lattice, with top element `fail`.

We use refinement and assertions to specify that a program m satisfies a specification with precondition P and postcondition Q:

$m \leq \texttt{assert}\ P;\ \texttt{spec}\ x.\ Q\ x$

If the precondition is false, the right hand side is `fail`, and the statement trivially holds. Otherwise, m cannot fail, and every possible result x of m must satisfy Q.

For a detailed description on using the *ne*-monad for stepwise refinement based program verification, we refer the reader to [14].

Example 3 (Swapping multiple elements)**.** We specify an operation to perform multiple swaps. It takes two disjoint sets of indices s_1 and s_2, and a list xs. It then swaps each index in s_1 with some index in s_2. The precondition of this operation assumes that the index sets are in range, disjoint, and have the same cardinality:

$swap\_spec\_pre\ s_1\ s_2\ xs \equiv s_1 \cap s_2 = \{\} \wedge s_1 \cup s_2 \subseteq \{0..<|xs|\} \wedge |s_1| = |s_2|$

The postcondition ensures that the resulting list is a permutation of the original list, the elements at indexes outside $s_1 \cup s_2$ are unchanged, and that each element in s_1 is swapped with one in s_2:

$swap\_spec\_post\ s_1\ s_2\ xs\ xs' \equiv$
$\quad mset\ xs' = mset\ xs \wedge (\forall i{\notin}s_1 \cup s_2.\ i < |xs| \implies xs'!i{=}xs!i)$
$\wedge\ (\forall i{\in}s_1.\ \exists j{\in}s_2.\ xs'!i{=}xs!j) \wedge (\forall j{\in}s_2.\ \exists i{\in}s_1.\ xs'!j{=}xs!i)$

As a sanity check, we prove that our specification is not vacuous, i.e., that for every input that satisfies the precondition, there exists an output that satisfies the postcondition:

$swap\_spec\_pre\ s_1\ s_2\ xs \implies \exists xs'.\ swap\_spec\_post\ s_1\ s_2\ xs\ xs'$

Note that this is only a sanity check lemma to detect problems early. Should we accidentally insert a vacuous specification here, we won't be able to prove refinement to an M-monad program later, which cannot be vacuous due to $invar_M$.

In the *ne* monad, we then specify:

$swap\_spec\ s_1\ s_2\ xs \equiv$
$\quad \texttt{assert}\ (swap\_spec\_pre\ s_1\ s_2\ xs);\ \texttt{spec}\ xs'.\ swap\_spec\_post\ s_1\ s_2\ xs\ xs'$

In Section 5.3 we will refine this specification to a parallel implementation in LLVM.

4.2 The Sepref Tool

The Sepref tool [1, 16] symbolically executes an abstract program in the *ne*-monad, keeping track of refinements for every abstract variable to a concrete representation, which may use pointers to dynamically allocated memory. During the symbolic execution, the tool synthesizes an Isabelle-LLVM program, together with a refinement proof. The synthesis is automatic, but requires annotations to the abstract program.

The main concept of the Sepref tool is refinement between an abstract program c in the *ne*-monad, and a concrete program $c_\dagger$ in the M monad, as expressed by the *hnr*-predicate:

$hnr\ \Gamma\ c_\dagger\ \Gamma'\ R\ CP\ c \equiv$
$\quad c \neq \texttt{fail} \implies ht\ \Gamma\ c_\dagger\ (\lambda r_\dagger.\ \exists r.\ \Gamma' * R\ r_\dagger\ r * {\uparrow}(\texttt{return}\ r \leq c \wedge CP\ r_\dagger))$

That is, either the abstract program c fails, or for a memory described by assertion Γ, the LLVM program $c_\dagger$ succeeds with result $r_\dagger$, such that the new memory is described by $\Gamma' * R\ r_\dagger\ r$, for a possible result r of the abstract program c. Moreover, the predicate CP holds for the concrete result. Note that hnr trivially holds for a failing abstract program. This makes sense, as we prove that the abstract program does not fail anyway. It allows us to assume abstract assertions during the refinement proof:

$$(\ \phi \implies hnr\ \Gamma\ c_\dagger\ \Gamma'\ R\ CP\ c\ ) \implies hnr\ \Gamma\ c_\dagger\ \Gamma'\ R\ CP\ (\texttt{assert}\ \phi;\ c)$$

Example 4. (Refinement of lists to arrays) We define abstract programs for indexing and updating a list:

$lget\ xs\ i \equiv \texttt{assert}\ (i{<}|xs|);\ \texttt{return}\ xs!i$
$lset\ xs\ i\ x \equiv \texttt{assert}\ (i{<}|xs|);\ \texttt{return}\ xs[i{:=}x]$

These programs assert that the index is in bounds, and then return the accessed element ($xs!i$) or the updated list ($xs[i{:=}x]$) respectively. The following assertion links a pointer to a list of elements stored at the pointed-to location:

$arr_A\ p\ xs = range\ \{0..{<}|xs|\}\ p\ (\lambda i.\ xs!i)$

That is, for every $i < |xs|$, $p + i$ points to the ith element of xs. Assertions like arr_A, that relate concrete to abstract values, are called *refinement relations*. If we want to emphasize that they depend on the heap, we also call them refinement *assertions*.

Indexing and updating of arrays is implemented by:

$aget\ p\ i \equiv ll\_ofs\_ptr\ p\ i;\ ll\_load\ p$
$aset\ p\ i\ x \equiv ll\_ofs\_ptr\ p\ i;\ ll\_store\ x\ p;\ \texttt{return}\ p$

The abstract and concrete programs are linked by the following refinement theorems:

$hnr\ (arr_A\ xs_\dagger\ xs * idx_A\ i_\dagger\ i)\ (aget\ xs_\dagger\ i_\dagger)\ (arr_A\ xs_\dagger\ xs * idx_A\ i_\dagger\ i)\ id_A\ \top\ (lget\ xs\ i)$
$hnr\ (arr_A\ xs_\dagger\ xs * idx_A\ i_\dagger\ i)\ (aset\ xs_\dagger\ i_\dagger\ x)\ (idx_A\ i_\dagger\ i)\ arr_A\ (\lambda r.\ r{=}xs_\dagger)\ (lset\ xs\ i\ x)$

That is, if the list xs is refined by array $xs_\dagger$, and the natural number i is refined by the fixed-width[2] word $i_\dagger$ ($idx_A\ i_\dagger\ i$), the $aget$ operation will return the same result as the $lget$ operation (id_A). The resulting memory will still contain the original array. Note that there is no explicit precondition ($\top$) that the array access is in bounds, as this follows already from the assertion in the abstract $lget$ operation. The $aset$ operation will return a pointer to an array that refines the updated list returned by $lset$. As the array is updated in place, the original refinement of the array is no longer valid. Moreover, the returned pointer r will be the same as the argument pointer $xs_\dagger$. This information is important for refining to parallel programs on disjoint parts of an array (cf. Section 4.4).

To increase readability, we introduce an (almost) point-free notation for refinement theorems. The theorems for the array operations above can also be written as:

$aget,\ lget : arr_A \to idx_A \to id_A$
$aset,\ lset : {arr_A}^d{:}xs_\dagger \to idx_A \to id_A \to arr_A{:}r\ [r{=}xs_\dagger]$

[2]We use Isabelle's word library here, which encodes the actual width as a type variable, such that our functions work with any bit width. For code generation, we will fix the width to 64 bit.

The first theorem simply states that the first argument is refined by arr_A, the second argument by idx_A, and the result by id_A. The second theorem adds the annotation $\cdot^d$ to the refinement for the array argument, indicating that this argument will be destroyed, i.e., the refinement is no longer valid when the function returns. Moreover, it binds the array argument to the name $xs_\dagger$ and the result to r. These names are used in the pointer equality predicate $[r{=}xs_\dagger]$ at the end, indicating that the result will be the same pointer as the array argument.

Given refinement relations for the parameters, and refinement theorems for all operations in a program, the Sepref tool automatically synthesizes an LLVM program from an abstract *neM* program. The tool tries to automatically discharge additional proof obligations, typically arising from translating arithmetic operations from unbounded numbers to fixed width numbers. Where automatic proof fails, the user has to add assertions to the abstract program to help the proof. The main difference of our tool wrt. the existing Sepref tool [1] is the additional condition (*CP*) on the concrete result, which is used to track pointer equalities. We have added a heuristics to automatically synthesize and discharge these equalities.

4.3 Modular Data Structure Development

The Refinement Framework allows us to build more complex data structures, using already existing ones as building blocks, and chaining together several refinements. We describe the development of an interval list data structure, which we need for our parallel partitioning algorithm (cf. 5.3).

A pair of natural numbers (l,h) can be used to represent the set $\{l..<h\}$. We define iv_R to be the refinement relation between intervals (pairs) and sets. Moreover, we define operations for constructing an interval, testing if an interval is empty, intersection, and cardinality. We show that these operations refine the corresponding operations on sets:

$iv_R :: (nat \times nat) \Rightarrow nat\ set \Rightarrow bool$ $\qquad$ $iv_R\ (l,h)\ s \equiv s = \{l..<h\}$

$iv\ l\ h \equiv (l,h)$ $\qquad$ $iv,\ \{\_..<\_\} : id_R \rightarrow id_R \rightarrow iv_R$

$iv\_is\_empty\ (l,h) \equiv h{\le}l$ $\qquad$ $iv\_is\_empty,\ (=\{\}) : iv_R \rightarrow id_R$

$iv\_inter\ (l_1,h_1)\ (l_2,h_2) \equiv (\ max\ l_1\ l_2,\ min\ h_1\ h_2\ )$ $\qquad$ $iv\_inter,\ (\cap) : iv_R \rightarrow iv_R \rightarrow iv_R$

$iv\_card\ (l,h) \equiv \texttt{if}\ h{<}l\ \texttt{then}\ 0\ \texttt{else}\ h{-}l$ $\qquad$ $iv\_card,\ |\_| : iv_R \rightarrow id_R$

Note that $h{<}l \implies \{l..<h\} = \{\}$, and we do not enforce $l \le h$ for our representation. Thus, no checks are needed for construction and intersection. However, we use a check to avoid underflow when computing the cardinality.

Analogously, we implement open intervals with a single number, and define operations to construct an open interval, and to intersect a closed and an open interval:

$ivo_R :: nat \Rightarrow nat\ set \Rightarrow bool$ $\qquad$ $ivo_R\ l\ s \equiv s = \{l..\}$

$ivo\ l \equiv l$ $\qquad$ $ivo,\ \{\_..\} : id_R \rightarrow iv_R$

$iv\_inter\_ivo\ (l_1,h_1)\ l_2 \equiv (max\ l_1\ l_2)\ h_1$ $\qquad$ $iv\_inter\_ivo,\ (\cap) : iv_R \rightarrow ivo_R \rightarrow iv_R$

Next, we use Sepref to implement the natural numbers by fixed-sized words (idx_A). For example, given the definition of *iv_inter*, and an annotation to implement natural numbers by 64 bit words, Sepref synthesizes the Isabelle LLVM program $\mathit{iv\_inter}_\dagger$ and proves the refinement theorem:

$\mathit{iv\_inter}_\dagger,\ \mathit{iv\_inter} :\ \ \mathit{idx}_A \times \mathit{idx}_A \to \mathit{idx}_A \times \mathit{idx}_A \to \mathit{idx}_A \times \mathit{idx}_A$

We then define $\mathit{iv}_A \equiv (\mathit{idx}_A \times \mathit{idx}_A)\ O\ \mathit{iv}_R$ as the composition of the two refinements (word to nat to set). With the help of Sepref's FCOMP tool, we can automatically compose the refinement lemmas. For example, composing the refinement lemmas for $\mathit{iv\_inter}_\dagger,\ \mathit{iv\_inter}$ and $\mathit{iv\_inter},\ (\cap)$ yields:

$\mathit{iv\_inter}_\dagger,\ (\cap) :\ \ \mathit{iv}_A \to \mathit{iv}_A \to \mathit{iv}_A$

Thus, we obtain imperative implementations of the set operations. We proceed analogously for open intervals.

Next, we implement a set as the union of a list of non-empty, pairwise disjoint, and finite sets. While that seems to make little sense at first glance, we will later implement the sets in the list by intervals, and the list itself by dynamic arrays, to obtain an imperative interval list data structure. We define operations for constructing an empty set, emptiness test, disjoint union with a single set, cardinality, and a more specialized operation *split*, which splits off a non-empty set from the list:

$\mathit{ivl}_R :: \mathit{nat\ set\ list} \Rightarrow \mathit{nat\ set} \Rightarrow \mathit{bool}$
$\mathit{ivl}_R\ \mathit{ls}\ s \equiv s = \bigcup\ \mathit{set}\ s \wedge \mathit{pw\_disjoint}\ \mathit{ls} \wedge \forall x{:}\mathit{set}\ \mathit{ls}.\ \mathit{finite}\ x \wedge x \neq \{\}$

$\mathit{ivl\_empty} \equiv []$	$\mathit{ivl\_empty},\ \{\} : \mathit{ivl}_R$
$\mathit{ivl\_is\_empty}\ \mathit{ls} \equiv \mathit{ls}{=}[]$	$\mathit{ivl\_is\_empty},\ ({=}\{\}) : \mathit{ivl}_R \to \mathit{id}_R$
$\mathit{ivl\_dj\_un}\ s\ \mathit{ls} \equiv s\#\mathit{ls}$	$\mathit{ivl\_dj\_un},\ (\dot\cup) : \mathit{id}_R \to \mathit{ivl}_R \to \mathit{ivl}_R$
$\mathit{ivl\_card}\ \mathit{ls} \equiv \mathit{fold}\ (\lambda s\ c.\ c{=}c{+}\lvert s\rvert)\ \mathit{ls}\ 0$	$\mathit{ivl\_card},\ \mathit{card} : \mathit{ivl}_R \to \mathit{id}_R$
$\mathit{ivl\_split}\ (s\#\mathit{ls}) \equiv (s,\mathit{ls})$	$\mathit{ivl\_split},\ \mathit{split} : \mathit{ivl}_R \to \mathit{id}_R \times \mathit{ivl}_R$

$\mathit{split}\ s \equiv \mathtt{assert}\ (s{\neq}\{\});\ \mathtt{spec}\ (s_1,s_2).\ s = s_1 \mathbin{\dot\cup} s_2 \wedge s_1{\neq}\{\}$

We then refine the lists of sets to array lists (dynamic arrays) of intervals: $\mathit{al}_A\ \mathit{iv}_A$. Here, al_A is the refinement assertion from lists to the array list data structure from the IRF collections library. As argument, it takes a refinement assertion for the list elements. Again, Sepref automatically generates imperative implementations of the *ivl* functions and proves the corresponding refinement lemmas. Combining them with the refinements to sets yields the desired imperative interval list data structure, with the refinement relation $\mathit{ivl}_A \equiv \mathit{al}_A\ \mathit{iv}_A\ \ O\ \mathit{ivl}_R$. For example, for joining a single interval to the list, and for splitting off an interval, we get:

$\mathit{ivl\_dj\_un}_\dagger,\ (\dot\cup) : \mathit{iv}_A \to \mathit{ivl}_A{}^d \to \mathit{ivl}_A$
$\mathit{ivl\_split}_\dagger,\ \mathit{split} : \mathit{ivl}_A{}^d \to \mathit{iv}_A \times \mathit{ivl}_A$

Note that we update the underlying dynamic array destructively, hence the $\cdot^d$ annotation to the argument refinements.

In a last step, we define some operations on (finite) sets, and use Sepref to directly refine them to arrays, without any explicit intermediate steps. For example, intersecting two finite sets can be expressed as:

$\mathit{set\_inter\_it}\ s_1\ \ s_2 \equiv$

```
  assert(finite $s_2$);
  $r$={}; while ($s_2 \neq \{\}$) ( $(ss,s_2) \leftarrow split\ s_2$; $r = (s_1 \cap ss) \dot\cup r$ );
  return $r$
```

It is straightforward to prove that this algorithm returns $s_1 \cap s_2$. Also, Sepref can implement s_1 with a closed or open interval, and s_2 with an interval array, yielding:

$iv\_inter\_ivl_\dagger,\ (\cap) : iv_A \to {ivl_A}^d \to ivl_A$
$ivo\_inter\_ivl_\dagger,\ (\cap) : ivo_A \to {ivl_A}^d \to ivl_A$

We have demonstrated one way of modularly developing an interval list data structure based on a dynamic array. By separating the actual intervals from the list data structure, the proofs about the interval list where independent of the interval implementation. This is a design choice, and a more direct design, using $(nat \times nat)$ *list* as intermediate data structure, is certainly possible.

4.4 Array Splitting

An important concept for parallel programs is to concurrently operate on disjoint parts of the memory, e.g., different slices of the same array. However, abstractly, arrays are just lists. They are updated by returning a new list, and there is no way to express that the new list is stored at the same address as the old list. Nevertheless, in order to refine a program that updates two disjoint slices of a list to one that updates disjoint parts of the array in place, we need to know that the result is stored in the same array as the input. This is handled by the *CP* argument to *hnr*. To indicate that operations shall be refined to disjoint parts of the same array, we introduce the combinator `with_split` for abstract programs:

```
with_split $i\ xs\ f \equiv$
  assert ($i < |xs|$);
  $(xs_1,xs_2) \leftarrow f\ (take\ i\ xs)\ (drop\ i\ xs)$;
  assert ($|xs_1| = i \wedge |xs_2| = |xs| - i$);
  return ($xs_1 @ xs_2$)
```

Abstractly, this is an annotation that is inlined when proving the abstract program correct. However, Sepref will translate it to the concrete combinator *awith_split*:

$awith\_split\ i\ p\ f_\dagger \equiv p_2 \leftarrow ll\_ofs\_ptr\ p\ i;\ f_\dagger\ p\ p_2;$ `return` p

The corresponding refinement theorem is:

awith_split, `with_split` *:*
$\quad {arr_A}^d{:}p \to idx_A \to ({arr_A}^d{:}p_1 \to {arr_A}^d{:}p_2 \to arr_A{:}r_1 \times arr_A{:}r_2\ [r_1{=}p_1 \wedge r_2{=}p_2])$
$\quad \to arr_A{:}r\ [r{=}p]$

or, equivalently, in pointwise notation:

$hnr\ (arr_A\ p_1\ xs_1 * arr_A\ p_2\ xs_2)\ (f_\dagger\ p_1\ p_2)\ \Box$
$\quad (arr_A \times arr_A)\ (\lambda(r_1,r_2).\ r_1{=}p_1 \wedge r_2 = p_2)$
$\quad (f\ xs_1\ xs_2)$
$\Longrightarrow$

$hnr\ (arr_A\ p\ xs * idx_A\ i_\dagger\ i)\ (awith\_split\ i_\dagger\ p\ f_\dagger)$
$\quad (idx_A\ i_\dagger\ i)\ arr_A\ (\lambda r.\ r{=}p)$
$\quad (\texttt{with\_split}\ i\ xs\ f)$

The refinement of the function argument (f to $f_\dagger$) requires an additional proof that the returned pointers are equal to the argument pointers ($r_1 = p_1 \wedge r_2 = p_2$). Sepref tries to prove that automatically, using its pointer equality heuristics.

Splitting an array into two halves allows us to abstractly treat the array and its two halves just as lists, which simplifies the abstract proofs: the fact that the two halves come from the same array is only visible at a later refinement stage. However, while splitting an array in halves is adequate for many operations, it is not a workable abstraction for swapping multiple elements in parallel (cf. Sec. 3): while, in theory, we could split the array element-wise, this would incur a considerable proof burden.

A more elegant solution is to keep track of which elements of a list can be accessed already on the abstract level. To this end, we model a list of option values, *None* meaning that we cannot access this element. We start by defining functions to abstractly handle lists of option values. These functions work on the actual list, and the *structure* of the list, which is a list of Booleans indicating which elements we do *not* own. Using the structure of the list as an explicit concept simplifies abstract proofs, as, typically, the values in the list will change, while the structure is preserved. The following functions obtain the structure of a list, and determine if two structures are compatible, i.e., have the same lengths and own disjoint indexes:

$sl\_struct :: {}'a\ option\ list \Rightarrow bool\ list$	$sl\_struct\ xs \equiv map\ ({=}None)\ xs$
$sl\_compat :: bool\ list \Rightarrow bool\ list \Rightarrow bool$	$sl\_compat\ s_1\ s_2 \equiv list\_all2\ (\vee)\ s_1\ s_2$

We also define functions to split and join lists:

$sl\_split\ s\ xs \equiv map\ (\lambda i.\ \texttt{if}\ i{\in}s \wedge i < |xs|\ \texttt{then}\ xs!i\ \texttt{else}\ None)\ [0..{<}|xs|]$
$join\_option\ None\ y \equiv y \qquad join\_option\ x\ None \equiv x \qquad join\_option\ \_\ \_ \equiv None$
$sl\_join\ xs_1\ xs_2 \equiv map2\ join\_option\ xs_1\ xs_2$

Here, *sl_split s xs* returns a list that owns the indexes that are in s and owned by xs, and $sl\_join\ xs_1\ xs_2$ joins the elements of two (compatible) lists.

Analogously to `with_split`, we define a combinator `with_idxs` *s xs f*, that splits the list xs into the lists with the indexes s, and without the indexes s, executes f on these lists, and joins the resulting lists:

```
with_idxs $s\ xs\ m \equiv$
  assert $(\forall i{\in}s.\ i{<}|xs| \wedge xs!i \neq None)$;
  let $(xs_1, xs_2) = (sl\_split\ s\ xs,\ sl\_split\ ({-}s)\ xs)$;
  $(xs_1', xs_2') \leftarrow f\ xs_1\ xs_2$;
  assert $(sl\_struct\ xs_1' = sl\_struct\ xs_1 \wedge sl\_struct\ xs_2' = sl\_struct\ xs_2)$;
  return $(sl\_join\ xs_1'\ xs_2')$
```

On the concrete side, we define the refinement assertion $oarr_A$ between arrays and lists of options. It only owns the indexes of the array that are not None:

$oarr_A\ p\ xs \equiv range\ \{i \mid i.\ i{<}length\ xs \wedge xs!i{\neq}None\}\ p\ (\lambda i.\ the\ (xs!i))$

We implement abstract operations for accessing the list, and show the corresponding refinement lemmas:

$olget\ xs\ i \equiv$ `assert` $(i{<}|xs| \wedge xs!i \neq None)$; `return` $the\ (xs!i)$
$olset\ xs\ i\ x \equiv$ `assert` $(i{<}|xs| \wedge xs!i \neq None)$; `return` $the\ (xs[i{:=}x])$

$aget,\ olget : oarr_A \rightarrow idx_A \rightarrow id_A$
$aset,\ olset : {oarr_A}^d{:}p \rightarrow idx_A \rightarrow id_A \rightarrow oarr_A{:}r\ [r{=}p]$

We also define conversion operations between plain lists and lists of option values:

$l2o\ xs = map\ Some\ xs$	`return`, $l2o : arr_A \rightarrow oarr_A$
$o2l\ xs =$ `assert` $(None \notin set\ xs)$; $map\ the\ xs$	`return`, $o2l : oarr_A \rightarrow arr_A$

These conversion operations are important to limit the proof overhead when using lists of option values: only where fine-grained ownership control is needed, we use option values. When we are done, and have reassembled all parts of the list, we convert it back to a plain list.

Finally, we define *awith_idxs* and prove its refinement theorem:

$awith\_idxs\ p\ f \equiv f\ p\ p$; `return` p

$awith\_idxs$, `with_idxs` s :
$\quad {oarr_A}^d{:}p \rightarrow ({oarr_A}^d{:}p_1 \rightarrow {oarr_A}^d{:}p_2 \rightarrow oarr_A{:}r_1 \times oarr_A{:}r_2\ [r_1{=}p_1 \wedge r_2{=}p_2])$
$\quad \rightarrow oarr_A{:}r\ [r{=}p]$

Note that the set s of indexes does not have a concrete counterpart. It is a ghost variable that controls the split on the abstract level.

4.5 Refinement to Parallel Execution

Our abstract programs have no notion of parallel execution. To indicate that refinement to parallel execution is desired, we define an abstract annotation `npar`:

`npar` $f\ g\ a\ b \equiv x \leftarrow f\ a;\ y \leftarrow g\ b;$ `return` (x,y)

Its refinement rule is:

$$\begin{array}{l} hnr\ Ax\ (f_\dagger\ x_\dagger)\ Ax'\ Rx\ CP_1\ (f\ x) \wedge hnr\ Ay\ (g_\dagger\ y_\dagger)\ Ay'\ Ry\ CP_2\ (g\ y) \\ \Longrightarrow \\ hnr\ (Ax * Ay)\ (llc\_par\ f_\dagger\ g_\dagger\ x_\dagger\ y_\dagger)\ (Ax' * Ay')\ (Rx \times Ry) \\ \quad (\lambda(x_\dagger', y_\dagger').\ CP_1\ x_\dagger' \wedge CP_2\ y_\dagger')\ (\texttt{npar}\ f\ g\ x\ y) \end{array}$$

This rule can be used to automatically parallelize any (independent) abstract computations. For convenience, we also define `nseq`. Abstractly, it's the same as `npar`, but Sepref translates it to sequential execution.

5 A Parallel Sorting Algorithm

To test the usability of our framework, we verify a parallel sorting algorithm. We start with the abstract specification of an algorithm that sorts a list:

```
 1  psort xs n ≡
 2    assert n=|xs|; if n≤1 then return xs else psort_aux xs n (log2 n * 2)
 3
 4  psort_aux xs n d ≡
 5    assert n=|xs|
 6    if d=0 ∨ n<100000 then sort_spec xs
 7    else
 8      (xs,m) ← partition_spec xs;
 9      let bad = m<n div 8 ∨ (n−m < n div 8)
10      (_,xs) ← with_split m xs (λxs₁ xs₂.
11        if bad then nseq psort_aux psort_aux (xs₁,m,d−1) (xs₂,n−m,d−1)
12        else npar psort_aux psort_aux (xs₁,m,d−1) (xs₂,n−m,d−1)
13      );
14      return xs
```

Fig. 2: Abstract version of our parallel quicksort algorithm.

$\mathit{sort\_spec}\ xs \equiv \texttt{spec}\ xs'.\ \mathit{mset}\ xs'{=}\mathit{mset}\ xs \wedge \mathit{sorted}\ xs$

I.e., we return a sorted permutation of the original list. This is a standard specification of sorting in Isabelle, and easily proved equivalent to other, more explicit specifications:

$\mathit{sort\_spec}\ xs = \texttt{spec}\ xs'.$
$\quad(\forall x.\ \mathit{count\_list}\ xs'\ x = \mathit{count\_list}\ xs\ x) \wedge (\forall i\ j.\ i{<}j \wedge j{<}\mathit{length}\ xs' \implies xs'!j \not< xs'!i)$

Figure 2 show our abstract parallel sorting algorithm *psort*. This algorithm is derived from the well-known quicksort and introsort algorithms [23]: like quicksort, it partitions the list (line 8), and then recursively sorts the partitions in parallel (l. 12). Like introsort, when the recursion gets too deep, or the list too short, we fall back to some (not yet specified) sequential sorting algorithm (l. 6). Similarly, when the partitioning is very unbalanced (l. 9), we sort the partitions sequentially (l. 11). These optimizations aim at not spawning threads for small sorting tasks, where the overhead of thread creation outweighs the advantages of parallel execution. A more technical aspect is the extra parameter n that we introduced for the list length. Thus, we can refine the list to just a pointer to an array, and still access its length[3].

Reusing our existing development of an abstract introsort algorithm [11], we prove with a few refinement steps that *psort* implements *sort_spec*:

$\mathit{psort}\ xs\ |xs| \le \mathit{sort\_spec}\ xs$

5.1 Implementation and Correctness Theorem

Next, we have to provide implementations for the fallback *sort_spec*, and for *partition_spec*. These implementations must be proved to be in-place, i.e., return a pointer to the same array. It was straightforward to amend our existing formalization of

[3]Alternatively, we could refine a list to a pair of array pointer and length.

pdqsort [11] with the in-place proofs: once we had amended the refinement statements and bug-fixed the pointer equality proving heuristics, the proofs were automatic.

Given implementations of *sort_spec* and *partition_spec*, Sepref generates an LLVM program $psort_\dagger$ from the abstract *psort*, and proves a corresponding refinement lemma:

$$psort_\dagger,\ psort : {arr_A}^d{:}xs_\dagger \to idx_A \to arr_A{:}r\ [r{=}xs_\dagger]$$

Combining this with the correctness lemma of the abstract *psort* algorithm, and unfolding the definition of *hnr*, we prove the following Hoare-triple for our final implementation:

$$\begin{array}{l} ht\ (arr_A\ xs_\dagger\ xs\ \ *\ \ idx_A\ n_\dagger\ n\ \ *\ \ n = |xs|) \\ \quad (psort_\dagger\ xs_\dagger\ n_\dagger) \\ \quad (\lambda r.\ r{=}xs_\dagger\ \ *\ \exists\ xs'.\ arr_A\ xs_\dagger\ xs'\ \ *\ \ sorted\ xs'\ \ *\ \ mset\ xs' = mset\ xs) \end{array}$$

That is, for a pointer $xs_\dagger$ to an array, whose content is described by list xs (arr_A), and a fixed-size word $n_\dagger$ representing the natural number n (idx_A), which must be the number of elements in the list xs, our sorting algorithm returns the original pointer $xs_\dagger$, and the array content now is xs', which is sorted and a permutation of xs. Note that this statement uses our semantically defined Hoare triples (cf. Section 3.2). In particular, it does not depend on the refinement steps, the Sepref tool, or the VCG.

5.2 Sampling Pivot Selection

While we could simply re-use the existing partitioning algorithm from the pdqsort formalization, which uses a pseudomedian of nine pivot selection, we observe that the quality of the pivot is particularly important for a balanced parallelization. Moreover, the partitioning in the *psort_aux* procedure is only done for arrays above a quite big size threshold. Thus, we can invest a little more work to find a good pivot, which is still negligible compared to the cost of sorting the resulting partitions. We choose a sampling approach, using the median of 64 equidistant samples as pivot. We simply use quicksort to find the index of the pivot[4]:

$$sample\ xs \equiv is \leftarrow equidist\ |xs|\ 64;\ is \leftarrow sort\_wrt\ (\lambda i\ j.\ xs!i < xs!j)\ is;\ \texttt{return}\ (is!32)$$

Proving that this algorithm finds a valid pivot index is straightforward. More challenging is to refine it to purely imperative LLVM code, which does not support closures like $\lambda i\ j.\ xs!i < xs!j$.

We resolve such closures over the comparison function manually: using Isabelle's locale mechanism [24], we parametrize over the comparison function. Moreover, we thread through an extra parameter for the data captured by the closure:

locale $pcmp =$
  fixes $lt :: 'p \Rightarrow 'e \Rightarrow 'e \Rightarrow bool$ **and** $lt_\dagger :: 'p_\dagger \Rightarrow 'e_\dagger \Rightarrow 'e_\dagger \Rightarrow bool$
    and $par_A :: 'p_\dagger \Rightarrow 'p \Rightarrow assn$ **and** $elem_A :: 'e_\dagger \Rightarrow 'e \Rightarrow assn$
  assumes $\forall p.\ weak\_ordering\ (lt\ p)$
  assumes $lt_\dagger,\ lt : par_A \to elem_A \to elem_A \to bool_A$

[4]We leave verification of efficient median algorithms, e.g., quickselect, to future work. Note that the overhead of sorting 64 elements is negligible compared to the large partition that has to be sorted.

This defines a context in which we have an abstract compare function *lt* for the abstract elements of type *'e*. It takes an extra parameter of type *'p* (e.g. the list *xs*), and forms a weak ordering[5]. Note that the strict compare function *lt* also induces a non-strict version $le\ p\ a\ b \equiv \neg lt\ p\ b\ a$. Moreover, we have a concrete implementation $lt_\dagger$ of the compare function, wrt. the refinement assertions par_A for the parameter and $elem_A$ for the elements.

Our sorting algorithms are developed and verified in the context of this locale (to avoid confusion, our presentation has, up to now, just used $<$, $\leq$, and *sorted* instead of *lt p*, *le p*, and *sorted_wrt* (*le p*)). To get an actual sorting algorithm, we instantiate the locale with an abstract and concrete compare function, proving that the abstract function is a weak ordering, and that the concrete function refines the abstract one. For our example of sorting indexes into an array, where the array elements themselves are compared by a function *lt*, we get:

interpretation $idx\text{: } pcmp\ lt\_idx\ lt\_idx_\dagger\ arr_A\ idx_A$ $\quad\langle proof\rangle$

$lt\_idx\ xs\ i\ j \equiv lt\ (xs!i)\ (xs!j)$
$lt\_idx_\dagger\ xs_\dagger\ i_\dagger\ j_\dagger \equiv x_\dagger \leftarrow aget\ xs_\dagger\ i_\dagger;\ y_\dagger \leftarrow aget\ xs_\dagger\ j_\dagger;\ lt_\dagger\ x_\dagger\ y_\dagger$

This yields sorting algorithms for sorting indexes, taking an extra parameter for the array to index into. For our sampling application, we use *idx.introsort xs*.

5.3 Parallel Partitioning

While our parallel quicksort scheme parallelizes the sorting, partitioning is still a bottleneck: before the first thread is even spawned, the whole array needs to be partitioned. On the next recursion level, only two partitionings can run in parallel, and so on. That is, initially, most processors will be idle. To this end, the partitioning itself can be parallelized. The parallel partitioning algorithms used in the latest research on practically efficient sorting algorithms [26] are branchless k-way algorithms, which use atomic operations to orchestrate the parallel threads. In contrast, we *only* verify a 2-way partitioning algorithm that uses parallel calls as its only synchronization mechanism. This is a compromise between verification effort and efficiency, taking into account the features currently supported by Isabelle-LLVM. The idea of our parallel partitioning algorithm is sketched in Figure 3.

We specify our algorithm on sets of indexes, and then refine it to intervals and interval arrays (cf. Section 4.3). First, we specify Steps 1 and 2, i.e., returning a permutation of the list, along with the sets *ls* of indexes that belong to a left partition, and *rs* of indexes that belong to a right partition. In Figure 3, *ls* corresponds to the set of blue ($\leq p$) intervals, and *rs* to the set of red ($\geq p$) intervals:

$ppart\_spec\ p\ xs \equiv \texttt{spec}\ (xs',ls,rs).$
$\quad mset\ xs' = mset\ xs$
$\wedge\ ls \cup rs = \{0..<|xs'|\} \wedge ls \cap rs = \{\}$
$\wedge\ (\forall i{\in}ls.\ xs'!i \leq p) \wedge (\forall i{\in}rs.\ xs'!i \leq p)$

[5]A weak ordering is induced by a mapping of the elements into a total ordering. It is the standard prerequisite for sorting algorithms in C++ [25].

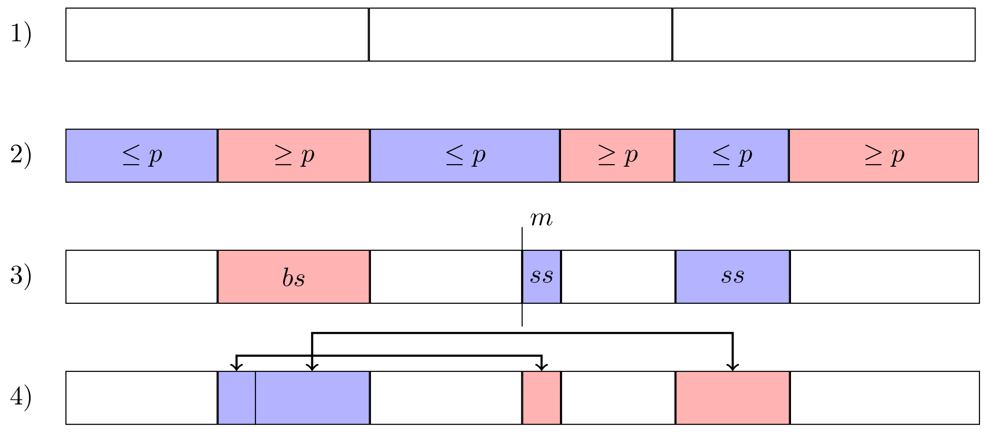

Fig. 3: Illustration of the phases of our parallel partitioning algorithm, after a pivot p has been picked. Step 1 splits the array into slices. Step 2 partitions the slices in parallel. Step 3 computes the start index m of the right partition as the sum of the sizes of all left partitions. It then determines the indexes of misplaced elements: the set bs contains right-partition elements that are left of m, and the set ss contains left partition elements that are right of m. Step 4 then swaps the misplaced elements in parallel. While Steps 1 and 3 are computationally cheap, Steps 2 and 4 do the main part of the work in parallel.

The whole partitioning algorithm is specified as follows:

```
$ppart1\ p\ xs \equiv$
  $(xs,ls,rs) \leftarrow ppart\_spec\ p\ xs;$                                — Step 1 and 2
  let $m = |ls|;\ ss = \{i{\in}ls.\ m{\leq}i\};\ bs = \{i{\in}rs.\ i{<}m\};$   — Step 3
  $xs \leftarrow swap\_spec\ ss\ bs\ xs;$                                    — Step 4
  return $(m,xs)$
```

A straightforward proof, only using arguments on lists and sets of indexes, shows that this algorithm partitions the list:

$$ppart1\ p\ xs \leq \texttt{spec}\ (m,\ xs').\quad mset\ xs' = mset\ xs \wedge m \leq |xs'| \wedge (\forall i{<}m.\ xs'!i \leq p) \wedge (\forall i{\in}\{m..{<}|xs'|\}.\ p \leq xs'!i)$$

The set operations in Step 3 are implemented by the operations $ivo\_inter\_ivl_\dagger$ and $iv\_inter\_ivl_\dagger$ of our interval arrays (cf. Section 4.3), using the equalities: $\{i{\in}ls.\ m{\leq}i\} = \{m..\} \cap ls$ and $\{i{\in}rs.\ i{<}m\} = \{0..{<}m\} \cap rs$. Thus, we are only missing implementations for *ppart_spec* and *swap_spec*.

The refinement of the parallel partitioning *ppart_spec* is similar to that of the parallel sorting algorithm *psort*: using `with_split` and `npar`, we split the list into slices that we then partition with a sequential algorithm.

The parallel swapping algorithm is refined as follows:

```
$par\_swap\ ss\ bs\ xs \equiv$
  if $(ss{=}\{\})$ then return $xs$
  else $xs \leftarrow l2o\ xs;\ xs \leftarrow par\_swap\_aux\ ss\ bs\ xs;\ xs \leftarrow o2l\ xs;$ return $xs$

$par\_swap\_aux\ ss\ bs\ xs \equiv$
  $((ss_1,ss_2),(bs_1,bs_2))$
    $\leftarrow$ spec $((ss_1,ss_2),(bs_1,bs_2)).\ ss{=}ss_1\dot{\cup}ss_2 \wedge bs{=}bs_1\dot{\cup}bs_2 \wedge |ss_1|{=}|bs_1| \wedge ss_1{\neq}\{\}$

  if $(ss_2{=}\{\})$ then $swap\_opt\_spec\ ss_1\ bs_1\ xs$
  else
    with_idxs $(ss_1 \cup bs_1)\ xs\ (\lambda xs_1\ xs_2.$
      $(xs_1,xs_2) \leftarrow$ npar $swap\_opt\_spec\ par\_swap\_aux\ (ss_1,bs_1,xs_1)\ (ss_2,bs_2,xs_2);$
      return $(xs_1,xs_2)$
```

The main procedure *par_swap* first checks if there are any indexes to swap. Then, it converts the plain list to a list of option values (*l2o*), invokes the actual parallel swapping procedure *par_swap_aux*, and converts the result back to a plain list (*o2l*).

The *par_swap_aux* procedure first splits off equally sized, non-empty sets ss_1 and bs_1 from the index sets, and then swaps these in parallel with the rest. Here, *swap_opt_spec* is the lifting of *swap_spec* to lists of option values, and `with_idxs` ensures that the swap operation will own the necessary elements.

We prove that our algorithm is correct ($par\_swap\ ss\ bs\ xs \leq swap\_spec\ ss\ bs\ xs$), and use Sepref, a straightforward implementation of sequential swapping, and our interval array implementation to refine it to efficient imperative Isabelle-LLVM code.

Combining all refinements in this section gives us a parallel partitioning algorithm. When we wanted to show that it satisfies the specification *partition_spec* as required by our parallel sorting algorithm, we discovered that we also need to prove that neither partition can be empty. While this is certainly possible along the same lines as it is proved for sequential partitioning, we chose a pragmatic solution here: we dynamically check for the extreme case of one partition being empty, and fix that with an additional swap. The runtime impact of this check is negligible, but it greatly simplifies the correctness proof.

5.4 Code Generation

Finally, we instantiate the sorting algorithms to sort unsigned integers and strings:

interpretation *unat: cmp* $(\lambda\_.\ <)$ $(\lambda\_.\ ll\_icmp\_ult)$ $unat_A^{64}$ ⟨*proof*⟩
interpretation *str: cmp* $(\lambda\_.\ <)$ $(\lambda\_.\ strcmp)$ str_A^{64} ⟨*proof*⟩

Here, the locale *cmp* is the version of *pcmp* without an extra parameter to the compare function[6]. This yields implementations $unat.psort_\dagger$ and $str.psort_\dagger$, and instantiated versions of the correctness theorem.

We then use our code generator to generate actual LLVM text, as well as a C header file with the signatures of the generated functions[7]:

[6]Parameters to the compare function are currently not supported for parallel sorting algorithms, as we cannot efficiently share the parameter between multiple threads. Integrating fractional separation logic into Sepref, which would enable such a sharing, is left to future work.

[7]For technical reasons, we represent the array size as non-negative signed integer, thus the C signature uses *int64_t*. Moreover, we use a string implementation based on dynamic arrays, rather than C's zero terminated strings.

export_llvm
  unat.psort$_\dagger$ **is** *uint64_t* psort*(*uint64_t*, int64_t*)
  str.psort$_\dagger$ **is** *llstring* str_psort*(*llstring*, int64_t*)
  defines
    `typedef struct` {*int64_t size;* `struct` {*int64_t capacity; char *data;*};} *llstring;*
  file *psort.ll*

This checks that the specified C signatures are compatible with the actual types, and then generates *psort.ll* and *psort.h*, which can be used in a standard C/C++ toolchain.

5.5 Benchmarks

We have benchmarked our verified sorting algorithm against a direct implementation of the same algorithm in C++. The result was that both implementations have the same runtime, up to some minor noise. This indicates that there is no systemic slowdown: algorithms verified with our framework run as fast as their unverified counterparts implemented in C++.

We also benchmarked against the state-of-the-art implementations *std::sort* with execution policy *par_unseq* from the GNU C++ standard library [27], and *sample_sort* from the Boost C++ libraries [28, 29]. We have benchmarked the algorithm on two different machines, and various input distributions. The results are shown in Figure 4: our verified algorithm is clearly competitive with the unverified state-of-the-art implementations. Only for a few string-sorting benchmarks, it is slightly slower. We leave improving on this for future work.

We also measured the speedup that the implementations achieve for a certain number of cores. The results are displayed in Figure 5: again, our verified implementation is clearly competitive.

The previous two benchmarks used relatively large input sizes. Figure 6 displays a benchmark for smaller input sizes. While we are still competitive with *sample_sort*, *std::sort* is clearly faster for small arrays. This result is expected: our parallelization uses hard-coded thresholds to switch to sequential algorithms, which are independent of the number of available processors or the input size. We leave fine-tuning of the parallelization scheme to future work.

6 Conclusions

We have presented a stepwise refinement approach to verify total correctness of efficient parallel algorithms. Our approach targets LLVM as back end, and there is no systemic efficiency loss in our approach when compared to unverified algorithms implemented in C++.

The trusted code base of our approach is relatively small: apart from Isabelle's inference kernel, it contains our shallow embedding of a small fragment of the LLVM semantics, and the code generator. All other tools that we used, e.g., our Hoare logic, the Sepref tool, and the Refinement Framework for abstract programs, ultimately prove a correctness theorem that only depends on our shallowly embedded semantics.

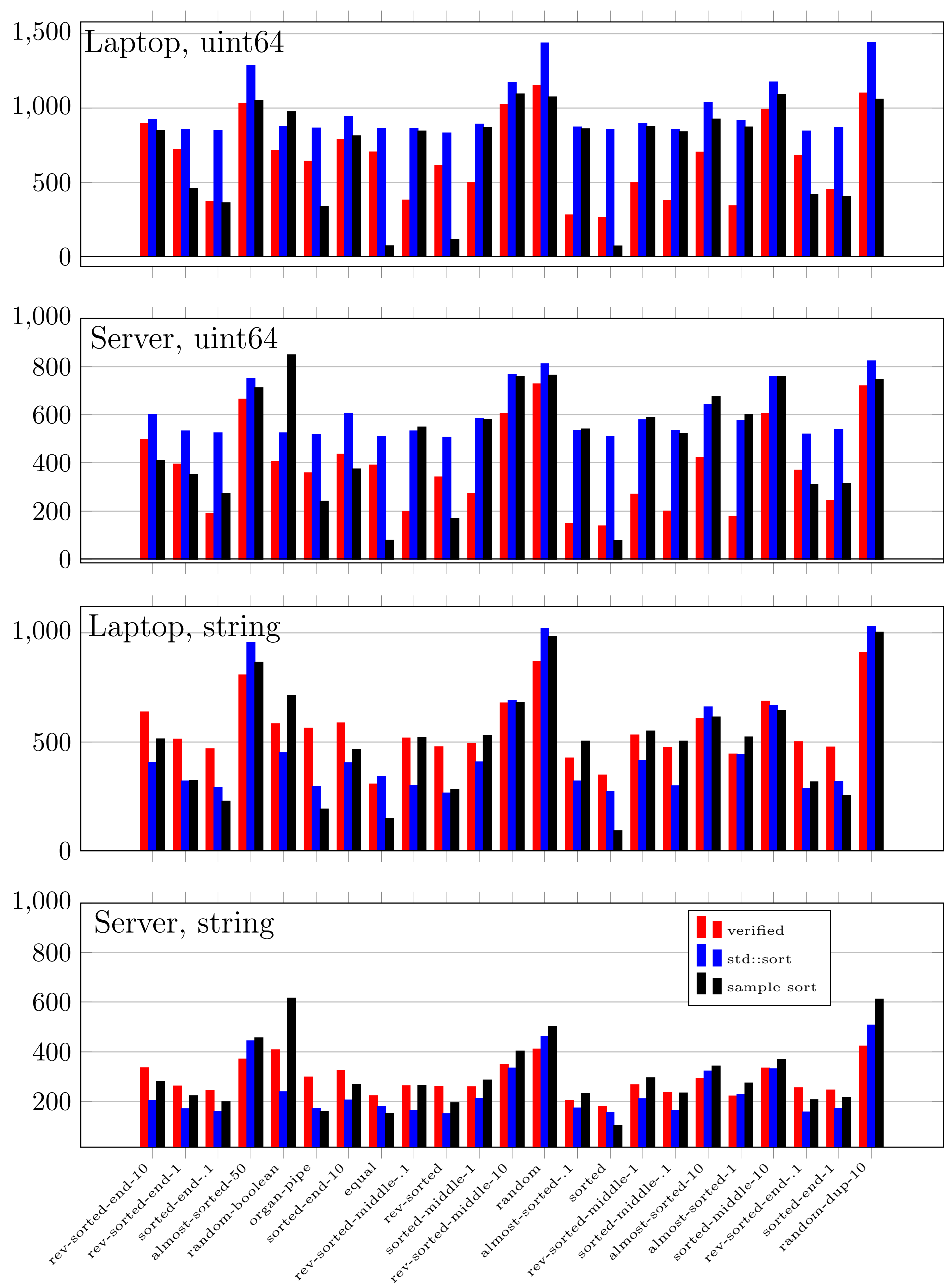

Fig. 4: Runtimes in milliseconds for sorting various distributions of 10^8 unsigned 64 bit integers and 10^7 strings with our verified parallel sorting algorithm, C++'s standard parallel sorting algorithm, and Boost's parallel sample sort algorithm. The experiments were performed on a server machine with 22 AMD Opteron 6176 cores and 128GiB of RAM, and a laptop with a 6 core (12 threads) i7-10750H CPU and 32GiB of RAM.

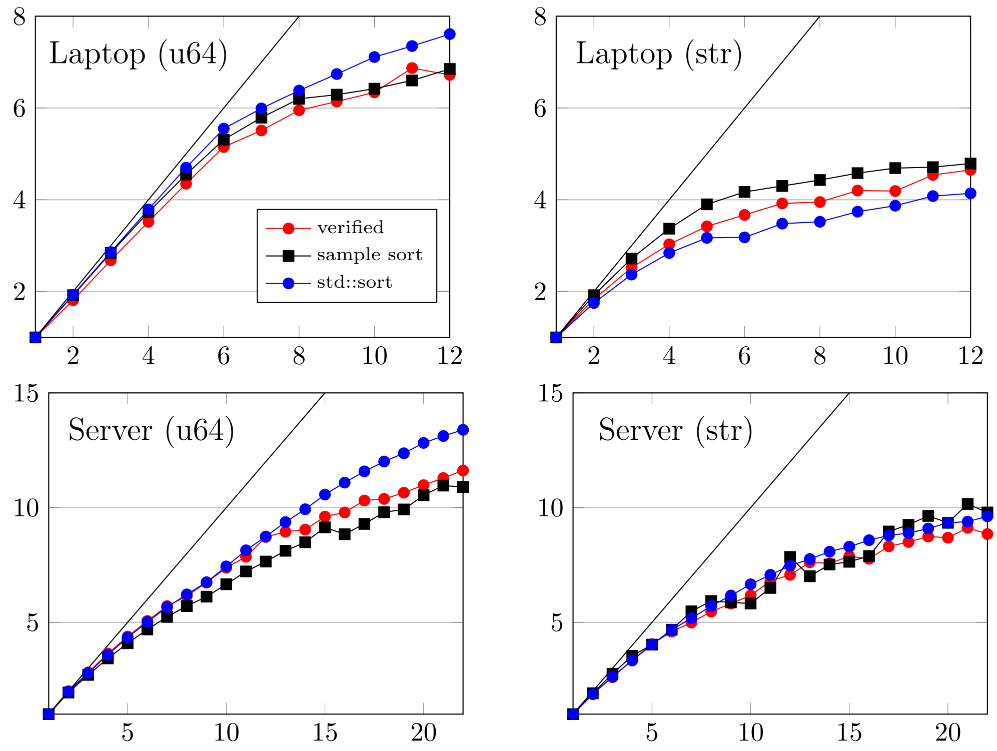

Fig. 5: Speedup of the various implementations for sorting 10^8 integers and 10^7 strings with a random distribution. The x axis ranges over the number of cores, and the y-axis gives the speedup wrt. the same implementation run on only one core. The thin black lines indicate linear speedup.

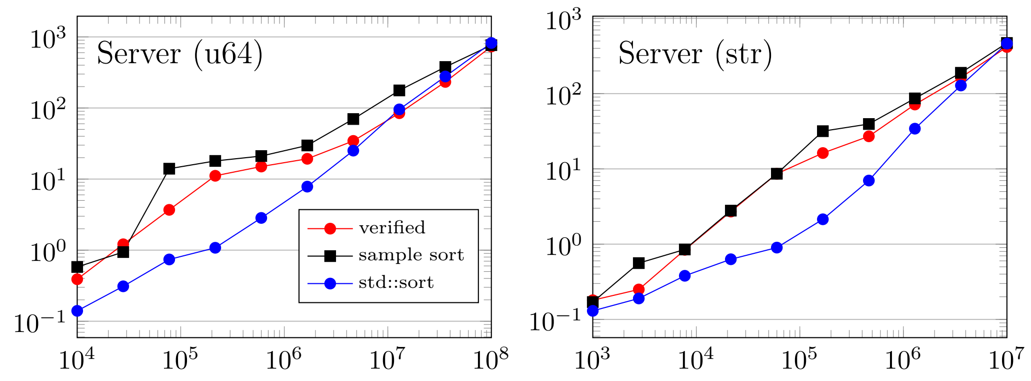

Fig. 6: Performance for sorting small arrays of randomly distributed integers and strings. The y-axis shows the runtime in milliseconds, and the x-axis the array size. Note that both axis are logarithmic.

As a case study, we have implemented a parallel sorting algorithm. It uses an existing verified sequential pdqsort algorithm as a building block, and is competitive with state-of-the-art parallel sorting algorithms.

The main idea of our parallel extension is to shallowly embed the semantics of a parallel combinator into a sequential semantics, by making the semantics report the accessed memory locations, and fail if there is a potential data race. We only needed to change the lower levels of our existing framework for sequential LLVM [1]. Higher-level tools like the VCG and Sepref remained largely unchanged and backwards compatible. This greatly simplified reusing of existing verification projects, like the sequential pdqsort algorithm [11].

While the verification of our sorting algorithms uses a top-down approach, we actually started with implementing, benchmarking, and fine-tuning the algorithms in C++. This gave us a quick option to find an algorithm that is efficient, and, at the same time not too complex for verification, without having to verify each intermediate step towards this algorithm. Only then we used our top-down approach to first formalize the abstract ideas behind the algorithm, and then refine it to an efficient implementation close to what we had written in C++. At this point, one may ask why not directly verify the C++ implementation: while this might be possible, the required work and steps would be similar: to manage the complexity of such a verification, several bottom-up refinement steps would be necessary, ultimately arriving at something similarly abstract as our initial abstract algorithm.

6.1 Related Work

While there is extensive work on parallel sorting algorithm (e.g. [26, 30, 31]), there seems to be almost no work on their formal verification. The only work we are aware of is a distributed merge sort algorithm [32], for which "no effort has been made to make it efficient"[32, Sec. 2], nor any executable code has been generated or benchmarked. Another verification [33] uses the VerCors deductive verifier to prove the permutation property ($\mathit{mset}\ \mathit{xs}' = \mathit{mset}\ \mathit{xs}$) of odd-even transposition sort [34], but neither the sortedness property nor termination.

Concurrent separation logic is used by many verification tools such as VerCors [35], and also formalized in proof assistants, for example in the VST [36] and IRIS [37] projects for Coq [38]. These formalizations contain elaborate concepts to reason about communication between threads via shared memory, and are typically used to verify partial correctness of subtle concurrent algorithms (e.g. [39]). Reasoning about total correctness is more complicated in the step-indexed separation logic provided by IRIS, and currently only supported for sequential programs [40]. Our approach is less expressive, but naturally supports total correctness, and is already sufficient for many practically relevant parallel algorithms like sorting, matrix-multiplication, or parallel algorithms from the C++ STL.

6.2 Future Work

An obvious next step is to implement a fractional separation logic [41], to reason about parallel threads that share read-only memory. While our semantics already supports shared read-only memory, our separation logic does not. We believe that implementing a fractional separation logic will be straightforward, and mainly pose technical issues for automatic frame inference.

Extending our approach towards more advanced synchronization like locks or atomic operations may be possible: instead of accessed memory addresses, a thread could report a set of possible traces, which are checked for race-freedom and then combined. Moreover, our framework currently targets multicore CPUs. Another important architecture are general purpose GPUs. As LLVM is also available for GPUs, porting our framework to this architecture should be possible. We even expect that we can model barrier synchronization, which is important in the GPU context.

Finally, the Sepref framework has recently been extended to reason about complexity of (sequential) LLVM programs [17, 42]. This could be combined with our parallel extension, to verify the complexity (e.g. work and span) of parallel algorithms.

Another direction for future work is to further optimize our verified sorting algorithm. We expect that tuning our parallelization scheme will improve the speedup for smaller input sizes. Also, while we are competitive with standard library implementations, recent research indicates that there is still some room for improvement, for example with the IPS^{4}o algorithm [26]. While this algorithm uses atomic operations in one place, many other of its optimizations, for example branchless decision trees for multi-way partitioning, only require features already supported by our framework.

References

[1] Lammich, P.: Generating Verified LLVM from Isabelle/HOL. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) 10th International Conference on Interactive Theorem Proving (ITP 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 141, pp. 22–12219. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.22 . http://drops.dagstuhl.de/opus/volltexte/2019/11077

[2] Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.-G.: A fully verified executable LTL model checker. In: CAV. LNCS, vol. 8044, pp. 463–478. Springer, ??? (2013)

[3] Brunner, J., Lammich, P.: Formal verification of an executable LTL model checker with partial order reduction. J. Autom. Reasoning **60**(1), 3–21 (2018) https://doi.org/10.1007/s10817-017-9418-4

[4] Wimmer, S., Lammich, P.: Verified model checking of timed automata. In: TACAS 2018, pp. 61–78 (2018)

[5] Lammich, P.: Efficient verified (UN)SAT certificate checking. In: Proc. of CADE. Springer, ??? (2017)

[6] Lammich, P.: The GRAT tool chain - efficient (UN)SAT certificate checking with formal correctness guarantees. In: SAT, pp. 457–463 (2017)

[7] Fleury, M., Blanchette, J.C., Lammich, P.: A verified SAT solver with watched literals using Imperative HOL. In: Proc. of CPP, pp. 158–171 (2018)

[8] Lammich, P.: Verified efficient implementation of gabow's strongly connected component algorithm. In: International Conference on Interactive Theorem Proving, pp. 325–340 (2014). Springer

[9] Lammich, P., Sefidgar, S.R.: Formalizing the Edmonds-Karp algorithm. In: Proc. of ITP, pp. 219–234 (2016)

[10] Lammich, P., Sefidgar, S.R.: Formalizing network flow algorithms: A refinement approach in Isabelle/HOL. J. Autom. Reasoning **62**(2), 261–280 (2019) https://doi.org/10.1007/s10817-017-9442-4

[11] Lammich, P.: Efficient verified implementation of introsort and pdqsort. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12167, pp. 307–323. Springer, ??? (2020). https://doi.org/10.1007/978-3-030-51054-1_18 . https://doi.org/10.1007/978-3-030-51054-1_18

[12] Lammich, P.: Refinement of parallel algorithms down to LLVM. In: Andronick, J., Moura, L. (eds.) 13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel. LIPIcs, vol. 237, pp. 24–12418. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, ??? (2022). https://doi.org/10.4230/LIPIcs.ITP.2022.24 . https://doi.org/10.4230/LIPIcs.ITP.2022.24

[13] Lammich, P., Lochbihler, A.: The Isabelle Collections Framework. In: ITP 2010. LNCS, vol. 6172, pp. 339–354. Springer, ??? (2010)

[14] Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Beringer, L., Felty, A.P. (eds.) ITP 2012. LNCS, vol. 7406, pp. 166–182. Springer, ??? (2012)

[15] Lammich, P.: Automatic data refinement. In: ITP. LNCS, vol. 7998, pp. 84–99. Springer, ??? (2013)

[16] Lammich, P.: Refinement to Imperative/HOL. In: ITP. LNCS, vol. 9236, pp. 253–269. Springer, ??? (2015)

[17] Haslbeck, M.P.L., Lammich, P.: For a few dollars more - verified fine-grained algorithm analysis down to LLVM. In: Yoshida, N. (ed.) Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12648, pp. 292–319. Springer, ??? (2021). https://doi.org/10.1007/978-3-030-72019-3_11 . https:

//doi.org/10.1007/978-3-030-72019-3_11

[18] O'Hearn, P.W.: Resources, concurrency and local reasoning. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004 - Concurrency Theory, pp. 49–67. Springer, Berlin, Heidelberg (2004)

[19] Huffman, B., Kuncar, O.: Lifting and transfer: A modular design for quotients in isabelle/hol. In: Gonthier, G., Norrish, M. (eds.) Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings. Lecture Notes in Computer Science, vol. 8307, pp. 131–146. Springer, ??? (2013). https://doi.org/10.1007/978-3-319-03545-1_9 . https://doi.org/10.1007/978-3-319-03545-1_9

[20] Intel oneAPI Threading Building Blocks. https://software.intel.com/en-us/intel-tbb

[21] Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS 2007, pp. 366–378 (2007)

[22] Klein, G., Kolanski, R., Boyton, A.: Mechanised separation algebra. In: ITP, pp. 332–337. Springer, ??? (2012)

[23] MUSSER, D.R.: Introspective sorting and selection algorithms. Software: Practice and Experience **27**(8), 983–993 (1997) https://doi.org/10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-#

[24] Kammüller, F., Wenzel, M., Paulson, L.C.: Locales a sectioning concept for isabelle. In: Bertot, Y., Dowek, G., Théry, L., Hirschowitz, A., Paulin, C. (eds.) Theorem Proving in Higher Order Logics, pp. 149–165. Springer, Berlin, Heidelberg (1999)

[25] Josuttis, N.M.: The C++ Standard Library: A Tutorial and Reference, 2nd edn. Addison-Wesley Professional, ??? (2012)

[26] Axtmann, M., Witt, S., Ferizovic, D., Sanders, P.: Engineering in-place (shared-memory) sorting algorithms. ACM Trans. Parallel Comput. **9**(1), 2–1262 (2022) https://doi.org/10.1145/3505286

[27] The GNU C++ Library 3.4.28. https://gcc.gnu.org/onlinedocs/libstdc++/

[28] Boost C++ Libraries. https://www.boost.org/

[29] Boost C++ Libraries Sorting Algorithms. https://www.boost.org/doc/libs/1_77_0/libs/sort/doc/html/index.html

[30] Chhugani, J., Nguyen, A.D., Lee, V.W., Macy, W., Hagog, M., Chen, Y.-K.,

Baransi, A., Kumar, S., Dubey, P.: Efficient implementation of sorting on multicore simd cpu architecture. Proceedings of the VLDB Endowment **1**(2), 1313–1324 (2008)

[31] Asiatici, M., Maiorano, D., Ienne, P.: How many cpu cores is an fpga worth? lessons learned from accelerating string sorting on a cpu-fpga system. Journal of Signal Processing Systems, 1–13 (2021)

[32] Hinrichsen, J.K., Bengtson, J., Krebbers, R.: Actris: Session-type based reasoning in separation logic. Proc. ACM Program. Lang. **4**(POPL) (2019) https://doi.org/10.1145/3371074

[33] Safari, M., Huisman, M.: A generic approach to the verification of the permutation property of sequential and parallel swap-based sorting algorithms. In: International Conference on Integrated Formal Methods, pp. 257–275 (2020). Springer

[34] Habermann, A.N.: Parallel neighbor-sort. Carnegie Mellon University (1972). https://doi.org/10.1184/R1/6608258.v1 . https://kilthub.cmu.edu/articles/journal_contribution/Parallel_neighbor-sort_or_the_glory_of_the_induction_principle_/6608258/1

[35] Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The vercors tool set: Verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (eds.) Integrated Formal Methods, pp. 102–110. Springer, Cham (2017)

[36] Verified Software Toolchain Project Web Page. https://vst.cs.princeton.edu/

[37] Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, 20 (2018) https://doi.org/10.1017/S0956796818000151

[38] Bertot, Y., Castran, P.: Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions, 1st edn. Springer, ??? (2010)

[39] Mével, G., Jourdan, J.-H.: Formal verification of a concurrent bounded queue in a weak memory model. Proc. ACM Program. Lang. **5**(ICFP) (2021) https://doi.org/10.1145/3473571

[40] Spies, S., Gäher, L., Gratzer, D., Tassarotti, J., Krebbers, R., Dreyer, D., Birkedal, L.: Transfinite iris: Resolving an existential dilemma of step-indexed separation logic. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 80–95 (2021)

[41] Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '05, pp. 259–270.

ACM, New York, NY, USA (2005). https://doi.org/10.1145/1040305.1040327 . http://doi.acm.org/10.1145/1040305.1040327

[42] Haslbeck, M.P.L., Lammich, P.: For a few dollars more - verified fine-grained algorithm analysis down to LLVM. TOPLAS, S.I. ESOP'21. to appear