Refinement of Parallel Algorithms down to LLVM

A. Nonymous

No Institute Given

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 simple parallel sorting algorithm, and show that it performs on par with its C++ implementation, and is competitive to state-of-the-art parallel sorting algorithms.

1 Introduction

Modern hardware has become faster by getting more parallel, while the sequential execution speed has not significantly increased in the last years. To keep up with this development, and fully utilize the capabilities of modern hardware, parallel algorithms are required.

However, parallel algorithms come with new subtle classes of errors, like race conditions, which are particularly hard to find by testing. Thus, formal methods that can guarantee absence of all errors, including the hard-to-find ones, become even more important for parallel algorithms.

In this paper, 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), a powerful tool to verify efficient sequential software, such as model checkers [11,7,42], SAT solvers [26,27,12], or graph algorithms [24,31,32]. 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 sorting algorithm, re-using an existing verification of state-of-the-art sequential sorting algorithms [29]. Our verified parallel sorting algorithm is competitive to state-of-the-art parallel sorting algorithms, at least on moderately parallel hardware.

The contributions of this paper are as follows:

– a semantics for LLVM that supports parallel execution. Up to our knowledge, this is the first formalization of parallelism for a practically usable LLVM semantics.

– a framework that supports a stepwise refinement approach to verify total correctness of parallel programs.
– a verified parallel sorting algorithm. Up to our knowledge, this is the first verification of a parallel sorting algorithm, let alone a competitive implementation.

The formalization and benchmark scripts have been submitted as (anonymized) supplementary material.

1.1 Overview

This paper is based on the Isabelle Refinement Framework (IRF), a continuing effort to verify efficient implementations of complex algorithms, using stepwise refinement techniques. In this paragraph, we give a brief overview of the components and versions of the IRF, and explain how our work relates to them. Figure 1 displays the components of the Isabelle Refinement Framework.

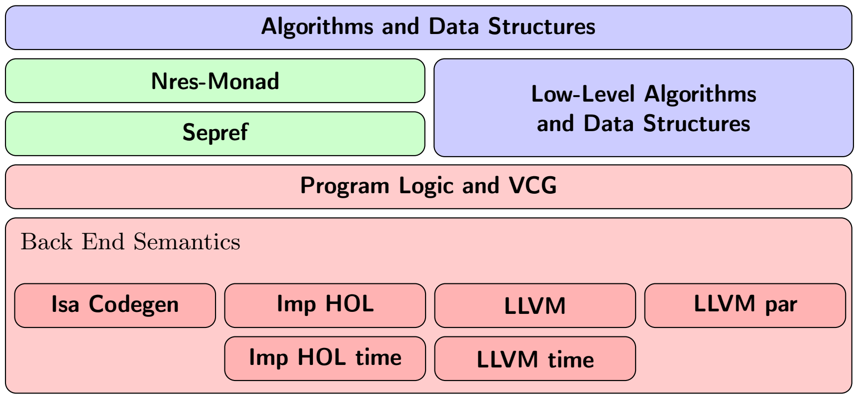


Fig. 1: Overview of the different components of the Isabelle Refinement Framework

At the basis, there is a semantics of a back end language. While the first version of the IRF used Isabelle's code generator to generate purely functional code [33,23], this has subsequently been extended to imperative functional code [25] using Imperative HOL [8], and purely imperative LLVM code [28]. For the imperative back ends, also versions that can reason about run-time complexity are available [15,17,16]. The work in this paper is based on the LLVM semantics without time [28], adding support for parallel programs.

On the next level, a program logic and verification condition generator is provided. While this is trivial when reasoning about purely functional code in Isabelle/HOL, the imperative versions use separation logic, and provide automation like frame-inference heuristics. While the Imperative HOL versions of the IRF are based on a separation logic formalization by Lammich and Meis [30], the LLVM versions are based on a more general framework by Klein et al. [20,21].

While this reasoning infrastructure can already be used to verify simple low-level algorithms and data structures, more complex developments are typically refined from a purely functional, monadic representation of nondeterministic programs (nres-monad), originally introduced in [33]. A typical development will do multiple refinement steps inside the nres-monad, and then use a semi-automatic tool (Sepref) to refine from the nres-monad into the back end semantics. While the nres-monad and associated tooling has largely stayed the same since it's first version [33], the Sepref tool has been continuously adapted and improved with each new back end that was added. Again, our work is based on the Sepref tool for LLVM without time [28].

We have also implemented parallelism for the Imperative HOL back end. This was done as a feasibility study, and uses the exactly same ideas and techniques as described in this paper for the more efficient LLVM back end. While a description of this initial implementation would not add any new ideas to this paper, it might still be useful in the context of older developments, that have not yet been ported to the LLVM back end.

The rest of this paper is roughly organized along the structure of the Isabelle Refinement Framework: We first describe our parallel LLVM semantics (Section 2) and separation logic 3. Then, we describe our changes to the refinement infrastructure (Section 4). Section 5 describes our parallel sorting algorithm case study and benchmark results, and Section 6 concludes the paper and discusses related and future work.

2 Shallow Embedding of Parallel Semantics

Isabelle LLVM [28] is a shallow embedding of a fragment of the LLVM semantics into the interactive theorem prover Isabelle/HOL [36]. It comes with a set of tools to facilitate algorithm development, and has been successfully used to verify efficient *sequential* algorithms [29]. The shallow embedding of Isabelle LLVM is key to its lightweightness and flexibility. In particular, the control flow is embedded as a state-error monad, and registers are mapped to bound variables. Only the memory model is deeply embedded.

We now describe an extension of Isabelle LLVM to support parallel algorithms, while keeping the shallow embedding. The resulting framework is backwards compatible to the original sequential Isabelle LLVM, such that porting existing verifications and tools is easy. The basic idea of our extension, which we detail in the following, is to support a parallel combinator $f \,||\, g$, that executes f and g in parallel, and fails if there is a possible data race.

2.1 Memory Model

We define a simple memory model[1], that supports arrays of values, where values can be pointers into arrays, integers, or tuples:

[1] The original formalization [28] also supports pointers into tuples. While complicated to implement, this feature has not been used in practice. To avoid additional compli-

datatype $addr = ADDR\ (block{:}\ nat)\ (idx{:}\ int)$
datatype $ptr = PTR\_NULL \mid PTR\_ADDR\ (the\_addr{:}\ addr)$
datatype $val = LL\_STRUCT\ val\ list \mid LL\_INT\ lint \mid LL\_PTR\ ptr$
typedef $memory = val\ list\ option\ list$

where $\alpha\ option = None \mid Some\ \alpha$ is Isabelle's option type.

Here, a memory is a list of blocks, where each block is either freed (*None*), or a list of values. Newly allocated blocks are simply appended to the end of the block list. An address consists of a block index and a value index, and a pointer is either null, or an address. 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 and deterministic allocation function, while being faithful wrt. any actual memory manager implementation.

The control flow is embedded into a state-error monad, which additionally records the memory addresses accessed by a computation[2]:

datatype $acc = MACC\ (reads{:}\ addr\ set)\ (writes{:}\ addr\ set)$
datatype $\alpha\ mres = NTERM \mid FAIL \mid SUCC\ \alpha\ acc\ memory$
datatype $\alpha\ M = M\ (run{:}\ memory \Rightarrow \alpha\ mres)$

Here, a memory access (*acc*) records the set of read and written addresses. A monadic result ($\alpha\ mres$) describes either a nonterminating (*NTERM*) or failed (*FAIL*) computation, or a successful computation ($SUCC\ x\ a\ m$), with result x, accessed addresses a, and new memory content m. A program ($\alpha\ M$) is a function from an initial memory to a monadic result.

For memory accesses, we define the operations 0 and $(+)$, and show that they form a commutative monoid:

$0 \equiv MACC\ \{\}\ \{\}$
$MACC\ r_1\ w_1 + MACC\ r_2\ w_2 \equiv MACC\ (r_1 \cup r_2)\ (w_1 \cup w_2)$

where $\equiv$ indicates defining equations. Moreover, we extend the $(+)$ operation to monadic results:

$(SUCC\ x\ a\ s) + a' \equiv SUCC\ x\ (a{+}a')\ s$
$FAIL + a \equiv FAIL$
$NTERM + a \equiv NTERM$

The basic monad combinators are `return` x, which returns a result without accessing any memory, and bind ($x{\leftarrow}m;\ f\ x$), which first executes m, binds its result to variable x, and then executes $f\ x$. They are defined as follows:

`return` $x \equiv M\ (SUCC\ x\ 0)$

cations with conflicting memory accesses from different but overlapping addresses, we dropped this feature. Now, each memory location is uniquely identified by an address, greatly simplifying the definition of data race.

[2] For technical reasons, we distinguish nontermination (*NTERM*) from errors (*FAIL*).

```
$x{\leftarrow}m;\ f\ x \equiv M\ (\lambda s.$ case $run\ m\ s$ of
  $SUCC\ x\ a\ s \Rightarrow (run\ (f\ x)\ s)\ +\ a$
| $NTERM \Rightarrow NTERM$
| $FAIL \Rightarrow FAIL)$
```

Here, the return operation makes no memory accesses (0), and the bind operation adds the memory accesses of both parts. If the second part does not depend on the result of the first part, we write $m_1;\ m_2$ as shortcut notation for $x{\leftarrow}m_1;\ m_2$.

Using the accessed memory, we can define a parallel combinator $m_1\ ||\ m_2$, which executes the programs m_1 and m_2 in parallel, and returns a pair of the results. When there is a data race, the combinator returns *FAIL*.

```
$m_1\ ||\ m_2 \equiv M\ (\lambda s.$ case $run\ m_1\ s$ of                      — $run\ m_1$
  $SUCC\ x_1\ a_1\ s \Rightarrow ($case $run\ m_2\ s$ of               — $m_1$ succeeds, run $m_2$
    $SUCC\ x_2\ a_2\ s \Rightarrow$ if $conflict\ a_1\ a_2$ then $FAIL$    — $m_2$ succeeds, check race
                       else $SUCC\ (x_1,x_2)\ (a_1{+}a_2)\ s$      — no race, add mem acc
  | $FAIL \Rightarrow FAIL$ | $NTERM \Rightarrow NTERM)$                 — $m_2$ fails or diverges
| $FAIL \Rightarrow FAIL$ | $NTERM \Rightarrow NTERM)$                   — $m_1$ fails or diverges
```

where

$conflict\ a_1\ a_2 \equiv writes\ a_1 \cap accs\ a_2 = \{\}\ \wedge\ writes\ a_2 \cap accs\ a_1 = \{\}$
$accs\ a \equiv reads\ a \cup writes\ a$

Note that we simply execute the programs sequentially on the given memory. As the programs only access disjoint addresses, the order of execution is irrelevant for the result — as long as the programs do not allocate new memory: in this case, the allocated addresses are different, depending on the order of execution. In particular, we cannot prove $par\ m_1\ m_2 = swap(par\ m_2\ m_1)$, where *swap* swaps the elements of the result tuple. However, we argue that our formalization is still correct, as the details of pointers, like block addresses, are not visible to the program.

Formally, this could be proved as follows: we define two states to be similar ($s_1 \approx s_2$), iff there exists a bijection f on block indexes, such that $f\ s_1 = s_2$. We lift $\approx$ to monadic results (*mres*), identifying *FAIL* and *NTERM*. We call a program *well-formed*, if it behaves similar on similar states, only depends on the addresses reported as read, and only modifies the addresses reported as written. Well-formedness syntactically extends over the basic monad operations. For well-formed programs, we can actually prove $par\ m_1\ m_2 \approx swap(par\ m_2\ m_1)$.

We leave a mechanised formalization of the above sketch to future work.

2.2 Extending the LLVM Backend

While the parallel operator gives us a semantics for parallel execution, it is difficult to map it to LLVM in its full generality, as this would mean to parallelly execute two parts of the same procedure, including all captured registers. Thus, our actual LLVM semantics only allows for a slightly more restricted parallel operator, which calls two single-argument functions in parallel, and returns the pair of their results:

$llc\_par\ f\ g\ a\ b \equiv par\ (f\ a)\ (g\ b)$

where f and g must be constants referring to LLVM functions.

We have extended the code generator to map *llc_par* to an external *parallel*() function with the following signature:

```
void parallel(void (*f1)(void*), void (*f2)(void*), void *x1, void *x2)
```

Its semantics is to perform the calls *f1*(*x1*) and *f2*(*x2*) in parallel. Here, the arguments *x1* and *x2* point to some stack-allocated memory containing the actual function arguments, and space for the results. The actual implementation of the *parallel* function has to be provided by a (trusted) support library. For our experiments, we use a very concise C++ implementation, based on the Threading Building Blocks [40] library:

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

3 Parallel Separation Logic

In order to reason about programs with a parallel operator, we define a parallel separation logic [37]. For our simple parallel combinator that does not allow for communication between parallel threads, a simple parallel rule is sufficient:

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \quad \{P_2\}\ c_2\ \{Q_2\}}{\{P_1 * P_2\}\ c_1 \parallel c_2\ \{Q_1 * Q_2\}}$$

That is, two threads c_1 and c_2 that run on disjoint memory can be executed in parallel. In this section, we briefly sketch the separation logic framework for Isabelle LLVM [28], and describe its extension to parallel separation logic.

3.1 Separation Algebra

In order to reason about memory with separation logic, a standard approach is to define an abstraction function from the memory into a separation algebra. The separation algebra can abstract away from details of the memory model (like our choice of using a list of blocks), and introduce the necessary formalism to support partial memories. This abstraction layer allows to use two different formalizations of the memory model, a concise and simple *concrete memory model* for the semantics, and an *abstract memory model* that forms a separation algebra. Note that the abstract memory model is only used for reasoning, and does not contribute to the trusted code base of our LLVM semantics.

In the following, we briefly sketch the concepts of separation algebra and our abstract memory model. For a more detailed explanation, we refer to [9,28].

A separation algebra [9] is a structure with a zero, a disjointness predicate $a\#b$, and a disjoint union $a+b$. Intuitively, elements describe parts of the memory.

Zero describes the empty memory, $a\#b$ means that a and b describe disjoint parts of the memory, and $a+b$ describes the memory described by the union of a and b. For the exact definition of a separation algebra, we refer to [9,20]. We note that separation algebras naturally extend over functions and pairs, in a pointwise manner.

Example 1. (Trivial Separation Algebra) We define the type $\alpha\ tsa = 0 \mid TRIV\ \alpha$. This forms a separation algebra where the operators are defined as follows:

$a \# b \equiv a{=}0 \lor b{=}0 \qquad a + 0 \equiv a \qquad 0 + b \equiv b$

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:

$\alpha :: memory \rightarrow (addr \rightarrow val\ tsa) \times (nat \rightarrow nat\ tsa)$
$\alpha\ s \equiv (\alpha_m\ s,\ \alpha_b\ s)$

where

$\alpha_m\ s\ a \equiv \texttt{if}\ valid\_addr\ a\ s\ \texttt{then}\ TRIV\ get\ a\ s\ \texttt{else}\ 0$
$\alpha_b\ s\ b \equiv \texttt{if}\ valid\_block\ b\ s\ \texttt{then}\ TRIV\ (block\_size\ b\ s)\ \texttt{else}\ 0$
$valid\_addr\ (ADDR\ b\ i)\ s \equiv b < |s| \land s!b \neq None \land i{<}|the\ (s!b)|$
$get\ (ADDR\ b\ i)\ s \equiv the\ (s!b)\ !\ i$
$valid\_block\ b\ s \equiv b < |s| \land s!b \neq None$
$block\_size\ b\ s \equiv |the\ (s!b)|$

where $|xs|$ is the length of list xs, $xs!i$ is the ith element, and $the\ (Some\ x) \equiv x$ is the selector function of the option type. That is, an abstract memory $\alpha\ s$ consists of two parts:

1. $\alpha_m\ s$ is a map from addresses to the values stored there. It is used to reason about load and store operations.
2. $\alpha_b\ s$ 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:

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

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 does not depend on the memory, and is used to lift plain logical statements into separation logic. When clear from the context, we omit the $\uparrow$-symbol, and just mix plain statements with separation logic assertions.

3.2 Weakest Preconditions

We define a *weakest precondition* predicate as usual:

$$\begin{array}{l} \mathit{wp}\ m\ Q\ s \equiv \texttt{case}\ \mathit{run}\ m\ s\ \texttt{of} \\ \quad \mathit{SUCC}\ x\ a\ s' \Rightarrow Q\ x\ a\ s' \mid \_ \Rightarrow \mathit{False} \end{array}$$

That is, $\mathit{wp}\ m\ Q\ s$ holds, iff program m run on state s succeeds with a result x, memory accesses a, and new state s' that satisfy *postcondition* $Q\ x\ a\ s'$.

To set up a verification condition generator based on separation logic, the postcondition should be a separation logic assertion over the corresponding abstract state, i.e., we want $Q\ x\ a\ s$ to have the form $(Q'\ x)\ (\alpha\ s)$, where Q' is a separation logic assertion that depends on the result. However, it is not clear how to incorporate the accessed addresses a into this form, as they have no meaning on the level of assertions.

We observe that, for reasoning about parallel programs, it suffices to ensure that one thread does not access memory that is used by another, parallel thread. Moreover, we require the program to be well-formed in the sense that it only reports accessed memory that was allocated at the start of the program, or got newly allocated during the program's execution, and that an address can only transition from unallocated to allocated to freed. In particular, once freed, an address will never be re-used[3].

For our memory model, we define the set of allocated and freed addresses, and the newly allocated memory between two states:

$$\begin{array}{l} \mathit{valid\_addrs}\ s \equiv \{\ a \mid \mathit{valid\_addr}\ a\ s\ \} \\ \mathit{freed\_addrs}\ s \equiv \{\ \mathit{ADDR}\ b\ i \mid b{<}|s| \wedge s!b = \mathit{None}\ \} \\ \mathit{used\_mem}\ s \equiv \mathit{valid\_addrs}\ s \cup \mathit{freed\_addrs}\ s \\ \mathit{new\_mem}\ s\ s' \equiv \mathit{used\_mem}\ s' - \mathit{used\_mem}\ s \end{array}$$

Note that, for *freed_addrs*, we do not keep track of the block size of a freed block, and simply report all addresses with this block index as freed. We also define a predicate to state that memory only transitions from fresh to allocated to freed:

$$\begin{array}{l} \mathit{valid\_trans}\ s\ s' \equiv \mathit{valid\_addrs}\ s \subseteq \mathit{valid\_addrs}\ s' \cup \mathit{freed\_addrs}\ s' \\ \qquad\qquad\qquad \wedge\ \mathit{freed\_addrs}\ s \subseteq \mathit{freed\_addrs}\ s' \end{array}$$

Finally, we define the *weakest precondition with access restrictions* as follows:

$$\begin{array}{ll} \mathit{wpa}\ A\ c\ Q\ s \equiv \mathit{wp}\ c\ (\lambda r\ a\ s'. & \\ \quad Q\ r\ s' & \textit{— postcondition holds} \\ \quad \wedge\ \mathit{accs}\ a \subseteq \mathit{allocated}\ s \cup \mathit{new\_mem}\ s\ s' \wedge \mathit{valid\_trans}\ s\ s' & \textit{— well formed} \\ \quad \wedge\ \mathit{accs}\ a \cap A = \{\})\ s & \textit{— no addresses in A accessed} \end{array}$$

[3] Note that this property is already required for a (deterministic) sequential semantics. Otherwise, internals of the memory model implementation would be visible to the program via pointer comparisons. Giving newly allocated memory a unique block and restricting pointer comparison to pointers to the same block ensures that our memory model is sound wrt. any actual implementation.

That is, $wpa\ A\ c\ Q\ s$ states that execution of program c in state s yields a result r and a new state s' that satisfies the postcondition $Q\ r\ s'$. Moreover, the execution is well-formed, and does not access any addresses in A.

3.3 Hoare-Triples

Equipped with a weakest precondition with access restrictions, we define a Hoare-triple as follows:

$$STATE\ asf\ P\ s \equiv \exists as.\ as\ \#\#\ asf \wedge \alpha\ s = as{+}asf \wedge P\ as$$

$$\begin{aligned} ht\ P\ c\ Q \equiv{} & \forall s\ asf.\ STATE\ asf\ P\ s \\ & \implies wpa\ (addrs\ asf)\ c\ (\lambda r\ s'.\ STATE\ asf\ (Q\ r)\ s')\ s) \end{aligned}$$

Intuitively, the predicate $STATE\ asf\ P\ s$ specifies that the abstract memory $\alpha\ s$ can be split into a part as and the given frame asf, such that as satisfies assertion P. A Hoare-triple specifies that for all states s and frames asf, such that P describes s wrt. the frame ($STATE\ asf\ P\ s$), the program c will succeed on state s, not using any addresses of the frame, and the result and new state will satisfy assertion Q wrt. the original frame ($STATE\ asf\ (Q\ r)\ s'$).

Using this notion of Hoare-triple, we can prove the standard Hoare rules, in particular the frame rule and parallel rule:

$$ht\ P\ c\ Q \implies ht\ (P * F)\ c\ (\lambda\ r.\ Q\ r * F)$$

$$\begin{aligned} & ht\ P_1\ c_1\ Q_1 \wedge ht\ P_2\ c_2\ Q_2 \\ & \implies ht\ (P_1 * P_2)\ (par\ c_1\ c_2)\ (\lambda(r_1,r_2).\ Q_1\ r_1 * Q_2\ r_2) \end{aligned}$$

3.4 Verification Condition Generator

Based on our notion of weakest precondition and Hoare-triples, we implement a verification condition generator (VCG). Like the semantics and separation logic, also the VCG is shallowly embedded into HOL. For technical reasons[4], our VCG does not work on Hoare-triples, but on subgoals of the form:

$$STATE\ asf\ P\ s \wedge \ldots \implies wpa\ A\ c\ Q\ s$$

It repeatedly normalizes the state, and then applies rules that decompose the command c. Normalization of the state includes moving lifted assertions to HOL assumptions, and eliminating existential quantifiers. For example, the goal

$$STATE\ asf\ (\exists x.\ P_1\ a\ x * P_2\ x\ b * \uparrow(\ i < n))\ s \implies wpa\ A\ c\ Q\ s$$

becomes

[4] This form simplifies reasoning using Isabelle's standard tactics. In particular, for the Hoare-triple form, an application of a consequence rule may introduce a unification variable for the postcondition. This cannot depend on bound variables introduced later, rendering the goal unprovable.

$\bigwedge x.\ i{<}n \wedge \mathit{STATE}\ \mathit{asf}\ (P_1\ a\ x * P_2\ x\ b)\ s \implies \mathit{wpa}\ A\ c\ Q\ s$

There are rules for decomposing the basic monad combinators:

$Q\ r\ s \implies \mathit{wpa}\ A\ (\texttt{return}\ r)\ s$

$\mathit{wpa}\ A\ m\ (\lambda x.\ \mathit{wpa}\ A\ (f\ x)\ Q)\ s \implies \mathit{wpa}\ A\ (\{x \leftarrow m;\ f\ x\})\ Q\ s$

Moreover, any Hoare-triple can be turned into a rule for the VCG. Using the frame rule, it also generates a frame inference proof obligation:

$\mathit{ht}\ P\ c\ Q$
$\wedge\ \mathit{STATE}\ \mathit{asf}\ P'\ s \wedge P' \vdash P{*}F$
$\wedge\ (\bigwedge r\ s.\ \mathit{STATE}\ \mathit{asf}\ (Q\ r\ {*}{*}\ F)\ s \implies Q'\ r\ s)$
$\implies \mathit{wpa}\ (\mathit{addrs}\ \mathit{asf})\ c\ Q'\ s$

That is, if the VCG encounters a proof obligation for a command c, for which there is a Hoare-triple $\mathit{ht}\ P\ c\ Q$, it will first generate a frame inference proof obligation from the current state P' to the precondition P, use a simple heuristics to automatically infer the frame, and then continue to derive the original post-condition Q' from a state described by the postcondition Q of the Hoare-triple and the inferred frame F.

3.5 Hoare-Triples for Memory Operations

Finally, we prove Hoare-rules for all basic LLVM instructions, and register them with our VCG infrastructure. We display the rules for memory operations:

$\models \{n{\neq}0\}\ \mathit{ll\_malloc}\ \mathit{TYPE}(\alpha)\ n\ \{\lambda p.\ \mathit{range}\ \{0..{<}n\}\ (\lambda\_.\ \mathit{init})\ p * \mathit{b\_tag}\ n\ p\}$
$\models \{\mathit{range}\ \{0..{<}n\}\ \mathit{xs}\ p * \mathit{b\_tag}\ n\ p\}\ \mathit{ll\_free}\ p\ \{\lambda\_.\ \Box\}$
$\models \{\mathit{pto}\ x\ p\}\ \mathit{ll\_load}\ p\ \{\lambda r.\ r{=}x * \mathit{pto}\ x\ p\}$
$\models \{\mathit{pto}\ \mathit{xx}\ p\}\ \mathit{ll\_store}\ x\ p\ \{\lambda\_.\ \mathit{pto}\ x\ p\}$

Here $\mathit{b\_tag}\ n\ p$ asserts that p points to the beginning of a block of size n, and $\mathit{range}\ I\ f\ p$ 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 $\mathit{pto}\ x\ p$ 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 [28], such that we could easily port the data structures formalized there to our new framework. We also can verify simple parallel programs, using our VCG.

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 (cf. Section 1.1), focusing on our changes to support parallel algorithms.

4.1 Abstract Programs

Abstract programs are shallowly embedded. They are terms of type α *nres*, representing the result of a nondeterministic stateless computation:

$\alpha\ \mathit{nres} \equiv \texttt{fail} \mid \texttt{spec}\ (\alpha \Rightarrow \mathit{bool})$

Here, `fail` represents possible non-termination or assertion violation, and `spec` P represents a result nondeterministically chosen to satisfy predicate P. The *refinement ordering* on *nres* is defined by:

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

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`. The *monad combinators* are defined as:

$\texttt{return}\ x\ \equiv\ \texttt{spec}\ y.\ y{=}x$
$x \leftarrow \texttt{spec}\ P;\ f\ x\ \equiv\ \bigsqcup\{f\ x \mid P\ x\} \qquad x \leftarrow \texttt{fail};\ f\ \equiv\ \texttt{fail}$

Here, `return` x deterministically returns x, and $x{\leftarrow}m;\ f\ x$ chooses a result of m, binds it to x, and then executes $f\ x$. If m may fail, then the bind may also fail.

Arbitrary recursive programs can be defined via a fixed-point construction [22]. An *assertion* fails if its condition is not met, otherwise it returns the unit value:

$\texttt{assert}\ \phi \equiv \texttt{if}\ \phi\ \texttt{then return}\ ()\ \texttt{else fail};$

Assertions are used to express 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.

4.2 Specification of Sorting Algorithms

The first step to verify a sorting algorithm is to specify the desired result. We specify an algorithm that sorts a list *xs* as follows:

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

where *mset xs* is the multiset of the elements in the list *xs*. That is, the resulting list xs' is sorted, and a permutation of the original list *xs*.

For quicksort-like sorting algorithms, the main function is partitioning, which reshuffles a given list into two partitions, such that any element in the first partition is less than or equal to any element in the second partition. We specify a partitioning function that assumes at least four elements, and is guaranteed to return non-empty partitions. The result is returned as a reshuffled list xs', and the index m of the first element of the second partition:

$\mathit{partition\_spec}\ xs = \texttt{assert}\ (4 \leq \mathit{length}\ xs);$
$\quad \texttt{spec}\ (xs',m).\ \mathit{mset}\ xs' = \mathit{mset}\ xs \wedge 0{<}m \wedge m{<}|xs|$
$\quad \wedge\ (\forall i{\in}\{0..{<}m\}.\ \forall j{\in}\{m..{<}|xs|\}.\ xs'!i \leq xs'!j)$

Using the above specifications as building blocks, we can define a simple abstract sorting algorithm:

```
sort_aux xs d ≡
  if $d=0 \lor |xs|<100000$ then {              — check depth bound or size threshold
    sort_spec xs                                 — sort with some (sequential) algorithm
  } else {
    $(xs,m) \leftarrow$ partition_spec xs;                                  — partition
    with_split m xs ($\lambda xs_1\ xs_2$. {          — work on partitions $xs_1$ and $xs_2$
      npar sort_aux sort_aux $(xs_1,d-1)$ $(xs_2,d-1)$  — recursively sort in parallel
    })
  }
```

where

```
with_split m xs f ≡
  assert $(m < |xs|)$;
  $(xs_1,xs_2) \leftarrow f$ (take m xs) (drop m xs);
  assert $(|xs_1| = m \land |xs_2| = |xs| - m)$;
  return $(xs_1 @ xs_2)$
```

```
npar f g x y ≡ r1 ← f x; r2 ← g y; return (r1,r2)
```

where @ appends two lists. This algorithm is derived from Introsort [35], a quicksort with a depth bound to ensure $O(n \log n)$ worst-case complexity. If the recursion depth bound d is reached, or the list size is below a certain threshold, we sort the list using some yet unspecified sorting algorithm. Otherwise, we partition the list and recursively sort the two partitions.

Here, we have introduced two new combinators, which are hints for the following refinement steps: (1) the `with_split` combinator splits a given list into two parts, and invokes a function that can change the content of these two parts, but not their length. It then returns the concatenation of the two changed parts. It is a hint for the further refinement steps to implement the list as an array that is modified in-place. (2) the `npar` combinator simply executes two blocks sequentially. However, it is a hint to the subsequent refinement steps to refine to parallel execution.

Using the tools of the Isabelle Refinement Framework, and some existing theories copied from [28], it is straightforward to prove that this sorting algorithm is correct, i.e.:

$$\mathit{sort\_aux}\ xs\ d \leq \mathit{sort\_spec}\ xs$$

In the next sections, we will explain how to refine this two an actual parallel sorting algorithm in LLVM, re-using the existing sequential pdqsort algorithm [28] for small or too deep partitions, and a sampling algorithm for finding good pivots for partitioning.

4.3 The Sepref Tool

The Sepref tool [25,28] symbolically executes an abstract program in the *nres*-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 imperative Isabelle-LLVM program, together with a refinement proof. The synthesis is automatic, but usually requires some program-specific setup and boilerplate. In this section we briefly introduce the Sepref tool, focusing on our extensions. For a more detailed discussion, we refer the reader to [25,28].

The main concept of the Sepref tool is refinement between a functional program m in the nres monad, and an LLVM program c, as expressed by the *hnr*-predicate[5]:

$$\begin{array}{l} \mathit{hnr}\ \Gamma\ \ c\ \Gamma'\ R\ \ \mathit{CP}\ m \equiv \\ m \neq \texttt{fail}\ \implies\ \mathit{ht}\ \Gamma\ \ c\ (\lambda r.\ \exists x.\ \Gamma' * R\ x\ r * \uparrow(\texttt{return}\ x \leq\ m \wedge\ \mathit{CP}\ r)) \end{array}$$

That is, either m fails, or for a memory described by Γ, the LLVM program c succeeds with concrete result r, such that there is an abstract result x, the new memory is described by $\Gamma' * R\ x\ r$, and x is a possible result of m. Moreover, $\mathit{CP}\ r$ holds for the concrete result r. Note that the refinement trivially holds for a failing abstract program. This makes sense, as we prove that the abstract program does not fail anyway. Moreover, during the refinement proof, we can assume that assertions actually hold, as stated by the following rule:

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

Example 2. (Refinement of lists to arrays) We define abstract programs for indexing and updating a list:

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

That is, these programs assert that the index is in bounds, and then return the accessed element ($\mathit{xs}!i$) or the updated list ($\mathit{xs}[i{:=}x]$) respectively.

The following assertion links a pointer to a list of elements stored at the pointed-to location:

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

That is, for every $i < |xs|$, $p + i$ points to the ith element of xs.

The following Isabelle-LLVM programs index and update an array:

$$\begin{array}{l} \mathit{aget}\ p\ i \equiv \mathit{ll\_ofs\_ptr}\ p\ i;\ \mathit{ll\_load}\ p \\ \mathit{aset}\ p\ i\ x \equiv \mathit{ll\_ofs\_ptr}\ p\ i;\ \mathit{ll\_store}\ x\ p;\ \texttt{return}\ p \end{array}$$

We can link the abstract and concrete programs by the following theorems:

[5] This definition is the same as in [28], except that we added the CP parameter.

$hnr\ (arr_A\ xs\ p * idx_A\ i\ ii)\ (aget\ p\ ii)$
$\quad (arr_A\ xs\ p * idx_A\ i\ ii)\ id_A\ (\lambda\_.\ True)\ (lget\ xs\ i)$
$hnr\ (arr_A\ xs\ p * idx_A\ i\ ii)\ (aset\ p\ ii\ x)$
$\quad (idx_A\ i\ ii)\ arr_A\ (\lambda r.\ r{=}p)\ (lset\ xs\ i\ x)$

That is, if the list xs is refined by array p, and the natural number i is refined by the fixed-width[6] word ii ($idx_A\ i\ ii$), 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 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 p. This information is important for refining to parallel programs on disjoint parts of an array (cf. Section 4.4).

Given refinement assertions for the parameters, and *hnr*-rules for all operations in a program, the Sepref tool automatically synthesizes an LLVM program from an abstract *nres* 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. We have only slightly changed the existing Sepref tool, mainly adding a heuristics to keep track of concrete pointer equalities.

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*, as illustrated in Example 2.

We now refine the abstract `with_split` combinator from Section 4.2 to arrays. The corresponding concrete LLVM operation is[7]:

```
$awith\_split\ i\ a\ m \equiv$
  $a_2 \leftarrow ll\_ofs\_ptr\ a\ i;$
  $m\ a\ a_2;$
  return $a$
```

[6] 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.

[7] LLVM does not support higher-order functions, such as *awith_split*. However, the preprocessor of our code generator can inline such functions, to obtain valid LLVM code for non-recursive functions.

The abstract and concrete functions are connected by the following rule:

$$\begin{array}{l} hnr\ (arr_A\ xs_1\ xsi_1\ *\ arr_A\ xs_2\ xsi_2)\ (mi\ xsi_1\ xsi_2)\ \square \\ \quad (arr_A\ \times\ arr_A)\ (\lambda(xsi_1',xsi_2').\ xsi_1'{=}xsi_1 \wedge xsi_2' = xsi_2) \\ \quad (m\ xs_1\ xs_2) \\ \Longrightarrow \\ hnr\ (arr_A\ xs\ xsi\ *\ idx_A\ n\ ni)\ (awith\_split\ ni\ xsi\ mi) \\ \quad (idx_A\ n\ ni)\ (\lambda xs\ xsi.\ arr_A\ xs\ xsi)\ (\lambda xsi'.\ xsi'{=}xsi) \\ \quad (\texttt{with\_split}\ n\ xs\ m) \end{array}$$

That is, in order to refine an abstract `with_split` n xs m combinator, we have to refine the inner program m that operates on the split lists, in a way that the concrete pointers for the result (xsi_1',xsi_2') are the same as the arguments $(xsi_1,\ xsi_2)$. Thus, when the Sepref tool encounters a `with_split` combinator for a list that is refined by an array, it will apply the above rule, i.e., synthesize a refinement for the inner program, and try to prove that the return values are the same as the arguments.

4.5 Refinement to Parallel Execution

In Section 4.2, we have used the nres-combinator `npar` $f\ g\ x\ y$ as a mere abbreviation for executing $f\ x$ and $g\ y$ sequentially. In fact, for purely functional programs, there is no (semantic) difference between sequential or parallel execution, as long as one execution does not depend on the result of the other. For concrete programs that share memory, we have to additionally ensure that the two programs operate on disjoint parts of the memory. As disjointness is given by the separation conjunction, we get the following rule to refine the abstract `npar` to actual parallel execution:

$$\begin{array}{l} \ \ hnr\ Ax\ (fi\ xi)\ Ax'\ Rx\ CP_1\ (f\ x) \wedge hnr\ Ay\ (gi\ yi)\ Ay'\ Ry\ CP_2\ (g\ y) \\ \Longrightarrow \\ \ \ hnr\ (Ax * Ay)\ (llc\_par\ fi\ gi\ xi\ yi)\ (Ax' * Ay')\ (Rx \times Ry) \\ \quad (\lambda(rxi,ryi).\ CP_1\ rxi \wedge CP_2\ ryi)\ (\texttt{npar}\ f\ g\ x\ y) \end{array}$$

That is, the Sepref tool will try to refine the argument functions first, and then synthesize a program using *llc_par* to execute them in parallel.

5 A Parallel Sorting Algorithm

Our version of the Sepref tool is mostly backwards compatible with the original Sepref tool for Isabelle-LLVM [28]. The only difference is the handling of concrete pointer equalities (cf. Section 4.4). Thus, it was a mainly straightforward task to port the efficient verified Introsort and pdqsort algorithms [29] to the new tool, additionally proving that the concrete algorithm is in-place, i.e., the returned array pointer equals the argument pointer. Equipped with these refinements, the Sepref tool can synthesize a parallel sorting algorithm from *sort_aux* (Section 4.2), using the already existing partitioning algorithm and the pdqsort algorithm for the abstract *partition_spec* and *sort_spec* specifications.

Actually, we used refinement steps on the abstract level to add two optimizations: first, we add an explicit parameter for the length of the list. This allows us to refine the list to just an array pointer[8]. Second, to prevent creation of parallel threads for small partitions, we proceed sequentially if the created partitions are too unbalanced. The following is the algorithm that we input into Sepref[9]:

```
$par\_sort\_aux\ xs\ n\ d \equiv$
  assert $n{=}|xs|$
  if $d{=}0 \vee n{<}100000$ then {
    $slice\_sort\_spec\ xs\ 0\ n$
  } else {
    $(xs,m) \leftarrow slice\_partition\_spec\ xs\ 0\ n;$
    let $bad = m{<}n\ div\ 8 \vee (n{-}m < n\ div\ 8)$
    $(\_,xs) \leftarrow$ with_split $m\ xs\ (\lambda xs_1\ xs_2.$ {
      if $bad$ then nseq $par\_sort\_aux\ par\_sort\_aux\ (xs_1,m,d{-}1)\ (xs_2,n{-}m,d{-}1)$
      else npar $par\_sort\_aux\ par\_sort\_aux\ (xs_1,m,d{-}1)\ (xs_2,n{-}m,d{-}1)$
    $\});$
    return $xs$
  }

nseq $f_1\ f_2\ x_1\ x_2 \equiv r_1 \leftarrow f_1\ x_1;\ r_2 \leftarrow f_2\ x_2;$ return $(x_1,x_2)$

$par\_sort\ xs\ n \equiv$
  assert $n{=}|xs|$
  if $n{\le}1$ then return $xs$
  else $par\_sort\_aux\ xs\ n\ (log2\ n * 2)$
```

where `nseq` is abstractly the same as `npar`, but is translated to sequential execution by the Sepref tool. Moreover, *slice_sort_spec* and *slice_partition_spec* are the specifications for sorting and partitioning only a slice of the array, leaving the rest unchanged. These match the interface of the original sequential algorithms proved correct in [29]. Finally, *par_sort* wraps the algorithm to run with an initial depth limit of $2\lfloor\log_2 n\rfloor$.

5.1 Final Implementation and Correctness Theorem

From this algorithm, the Sepref tool generates an imperative implementation *par_sort_impl* and proves the corresponding *hnr* refinement lemma. Combining this with the correctness proof of the abstract *par_sort* algorithm, and unfolding the definition of *hnr*, we get the following Hoare-triple, that states correctness of our final algorithm:

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

[9] We have omitted some of the assertions required to prove that the arithmetic operations do not overflow.

$$\begin{array}{l} \mathit{ht}\ (\mathit{arr}_A\ \mathit{xs}\ \mathit{xsi} * \mathit{idx}_A\ n\ \mathit{ni} * n = |\mathit{xs}|) \\ \quad (\mathit{par\_sort\_impl}\ \mathit{xsi}\ \mathit{ni}) \\ \quad (\lambda r.\ r{=}\mathit{xsi} * \exists\ \mathit{xs}'.\ \mathit{arr}_A\ \mathit{xs}'\ \mathit{xsi} * \mathit{sorted}\ \mathit{xs}' * \mathit{mset}\ \mathit{xs}' = \mathit{mset}\ \mathit{xs}) \end{array}$$

That is, for a pointer *xsi* to an array, whose contents are described by list *xs* (arr_A), and a fixed-size word *ni* 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 *xsi*, and the array contents are now xs', which is sorted and a permutation of *xs*. The trusted code base (TCB) of this statement is our formalization of separation logic, Hoare-triples, arr_A, idx_A, the *sorted* and *mset* functions, as well as the LLVM semantics, the code generator, Isabelle's inference kernel, and, finally, the LLVM compiler. While this may seem a lot, we emphasize that it, in particular, does not include the nres-monad, the Sepref tool, the VCG, the Hoare-logic rules, or any high-level proof tactics or automated theorem provers. While all these components are actually used to create the proof, Isabelle's LCF style kernel ensures that they are not part of the TCB.

5.2 A Sampling Partitioner

While we could simply re-use the existing partitioning algorithm from the Introsort formalization, which uses a median-of-three pivot selection, we observe that the quality of the pivot is particularly important for a balanced parallelization. Moreover, the partitioning in the *par_sort_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 rather simple sampling approach: for an array of size n, we take $\min(n, 64)$ equidistant samples, and use the median of these samples as pivot.

Formalization of this sampling algorithm in the Refinement Framework is straightforward. To easily integrate with the existing (highly optimized) implementation of the partitioning algorithm, we have to determine the *index* of the pivot element, swap it with the first element of the array, and then use the existing partitioning algorithm.

This suggests an algorithm to find the median of a list of sample indexes, comparing each index by looking it up in the list to be sorted, and comparing the corresponding elements. We briefly sketch our approach to formalize such parametrized algorithms using locales, and an extension to Isabelle-LLVM's preprocessor to allow for painless code generation. Note that, for the sake of readability, we have slightly idealized the presentation[10].

A *locale* [19] is a named context that fixes some parameters and makes assumptions about these parameters. Inside a locale, definitions and theorems can depend on the parameters and assumptions. Later, a locale can be instantiated

[10] We have omitted some of the fixed parameters, and only describe a parametrized locale. In practice, we have two versions of the locale, one with a parameter, and one without. While they are still based on the same abstract algorithms, they duplicate some boilerplate code.

with actual terms for the parameters, proving that they satisfy the assumptions. This makes available instantiated versions of the definitions and theorems proved inside the locale. Instantiations can also be nested, that is, a locale can be instantiated in the context of another locale.

We define a locale to set up a parametrized compare function as follows:

locale *pcmp* =
  fixes *lt lti* par_A $elem_A$
  assumes *weak_ordering* (*lt p*)
  assumes *hnr*
    (par_A *p pi* ∗ $elem_A$ *a ai* ∗ $elem_A$ *b bi*)
    (*lti pi ai bi*)
    (par_A *p pi* ∗ $elem_A$ *a ai* ∗ $elem_A$ *b bi*) ($bool_A$) (λ_. *True*)
    (*lt p a b*)

that is, we fix an abstract compare function *lt*. It takes one extra parameter *p*, and is a weak ordering[11] for each such *p*. We also fix a concrete compare function *lti*, that refines the abstract compare function, with refinement assertion par_A for the parameter, and $elem_A$ for the elements to be sorted.

Inside the *pcmp* locale, we define our sorting and median functions, threading through the explicit parameter *p*. Among others, this will yield a function *median p xs*, which returns the median index of the elements in list *xs*, comparing wrt. *lt p*. We also get a corresponding implementation *median_impl*[12].

To implement the sampling algorithm, *inside* the *pcmp* locale, we define a comparison function for indexes, parametrized with the original parameter and the array to be sorted. For this, we instantiate the *pcmp* locale again[13]:

context *pcmp* **begin**

  definition *lt_idx* (*p*,*xs*) *i j* ≡ *lt p* (*xs*!*i*) (*xs*!*j*)
  definition *lt_idx_impl* ≡ ...

  sublocale *IDX: pcmp lt_idx lt_idx_impl* (par_A × arr_A) idx_A ⟨*proof*⟩

This makes available a function *IDX.median* and its implementation, which compares indexes into arrays as desired. This is then used in the partitioning function.

[11] A weak ordering can be defined by mapping the elements into a total ordering, and is the standard prerequisite for sorting algorithms.

[12] We simply reused a sequential sorting algorithm to completely sort the list and then obtain the middle element. While this is not the most efficient way of finding a median, the overhead when sorting 64 element arrays is negligible compared to the time spent for sorting the partitions that are at least 10^5 elements large. Thus, a more efficient algorithm like quickselect would not have any measurable effect on the overall performance, and we leave its verification to future work.

[13] In the actual formalization, we use a slightly more complex locale hierarchy to carefully avoid infinite recursive instantiations.

Finally, the *pcmp* locale is instantiated globally, for example to sort unsigned integers, and the resulting implementation is exported to LLVM text:

global_interpretation *unat: pcmp* $(<)$ *ll_icmp_ult* $unat_A^{64}$ ⟨*proof*⟩

export_llvm *unat.par_sort_impl*

The **global_interpretation** command instantiates the locale for comparing natural numbers implemented by 64 bit words ($unat_A^{64}$). Note that the compare function is not parametrized. Finally, the **export_llvm** command generates LLVM text. It first uses a preprocessor to generate and prove correct a set of code equations, which are then pretty printed to actual LLVM text. The preprocessor does not contribute to the trusted code base, as it proves correct its transformations. At this point, we have added a transformation to automatically instantiate the higher-order parameters generated by Isabelle's locale mechanism. This saves a lot of boilerplate code that was necessary in the original sorting algorithm formalization [29], and became unmanageable once we added the sublocale for comparing indexes into an array.

5.3 Benchmarks

The resulting parallel sorting algorithm is quite simplistic, ignoring a lot of research on efficient parallel sorting algorithms [10], as well as practically well-tested implementations [5]. In this section, we report on benchmarking our verified algorithm against unverified algorithms.

In a first experiment, we implemented the same simplistic algorithm directly in C++, and benchmarked it against its verified counterpart on various sets of distributions. 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++.

In a second experiment, we benchmark our verified algorithm against state-of-the-art implementations: we use the parallel *std::sort*(*execution_policy::par_unseq*) from the GNU C++ standard library [13], and the *sample_sort* implementation from the Boost C++ libraries [4]. We have benchmarked the algorithm on two different machines, and various input distributions. The results are shown in Figure 2. While our simple algorithm is clearly competitive for integer sorting on the less parallel laptop machine, it's slightly less efficient for sorting strings on the highly parallel server machine. Nevertheless, we believe that our simple verified algorithm is already useful in practice, and leave further optimizations to future work.

In a third experiment, we measure the speedup that the algorithm achieves for a certain number of cores. The results are displayed in Figure 3. While the speedup on the moderately parallel laptop is comparable to the one of the C++ standard library, our simple algorithm achieves lower speedups than the state-of-the-art implementations on the highly parallel server. Again, we leave further optimization of our algorithm to future work.

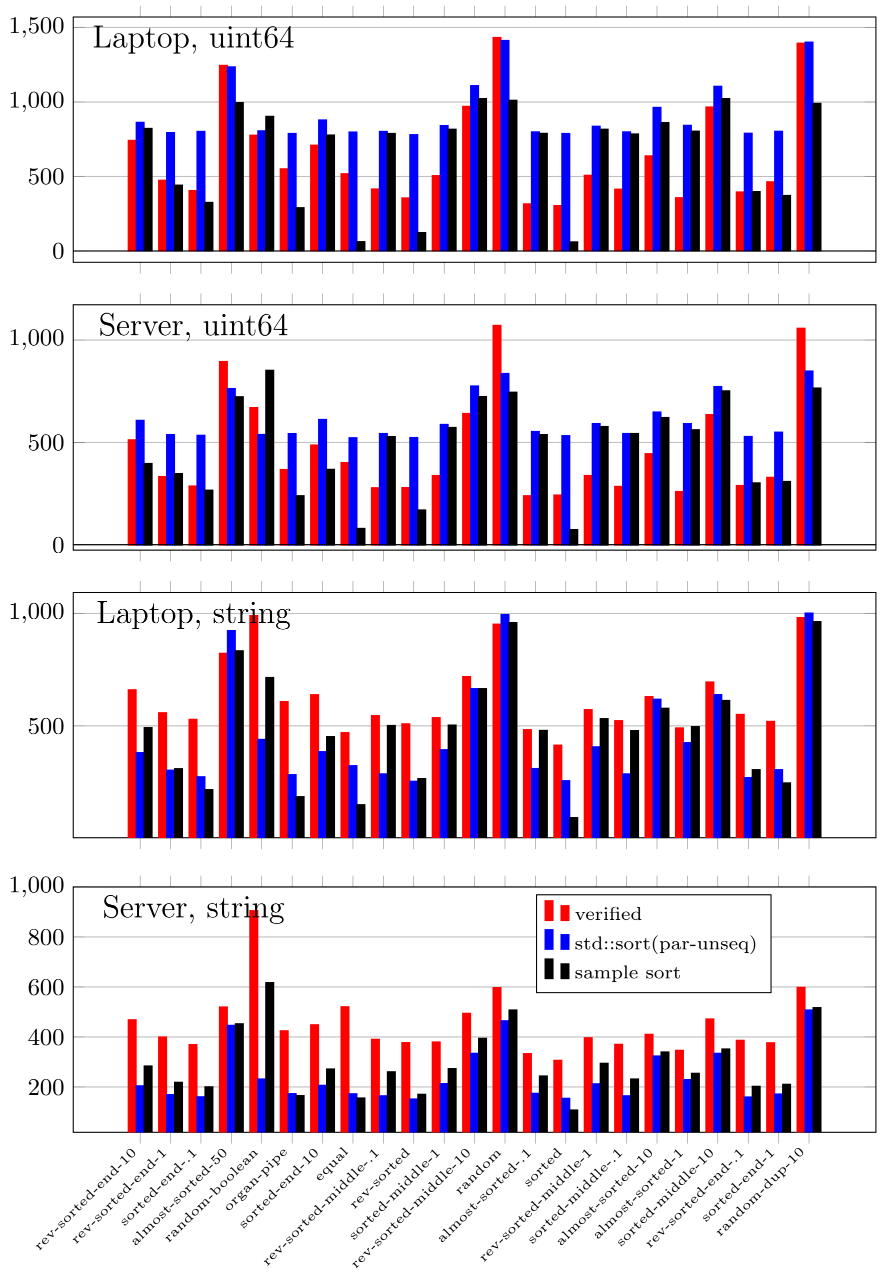
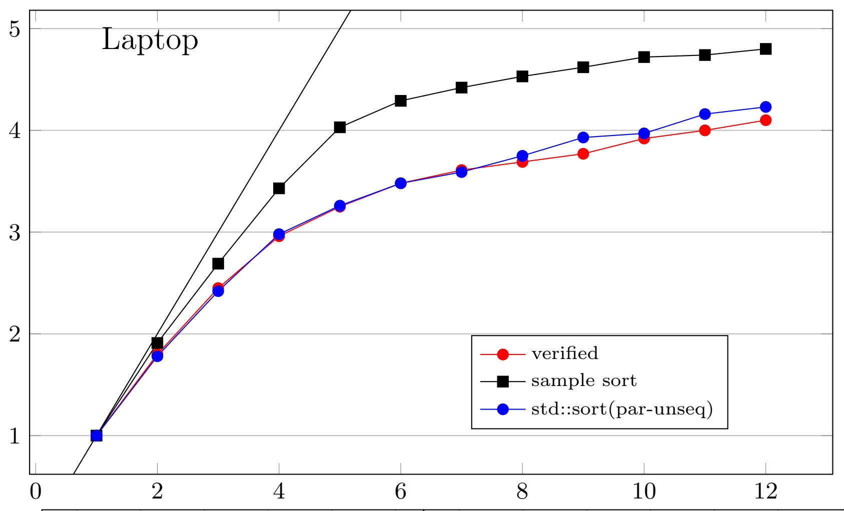


Fig. 2: Runtimes in milliseconds for sorting various distributions of unsigned 64 bit integers and strings with C++ standard sorting algorithm, our verified parallel sorting algorithm and Boosts sample sort algorithm. The experiments are performed on a server machine with 22 AMD Opteron 6176 cores and 128GiB of RAM, and a laptop with a 6 core (12 threads) i7-10750H CPU and 32GiB of RAM.



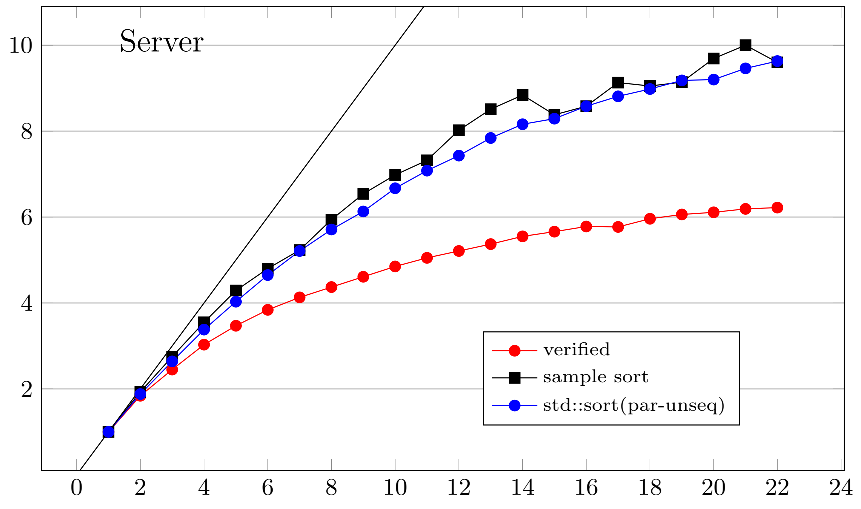


Fig. 3: Speedup of the various implementations, for sorting unsigned 64 bit integers with a random distribution, on a server with 22 AMD Opteron 6176 cores and 128GiB of RAM, and a laptop with a 6 core (12 threads) i7-10750H CPU and 32GiB of RAM. The x axis ranges over the number of cores, and the y-axis gives the speedup wrt. the same implementation run on only one core. The thin black lines indicate linear speedup.

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++. As a case study, we have implemented a simple parallel sorting algorithm. It uses an existing verified pdqsort algorithm as a building block, and is competitive with state-of-the-art parallel sorting algorithms, at least on moderately parallel hardware.

The main idea of our parallel extension is to shallowly embed the semantics of a parallel combinator into an existing sequential semantics, by making the existing semantics report the accessed memory locations, and fail if there is a potential data race. This only changed the lower levels of the existing tooling, while higher-level tools like VCG and Sepref remained largely unchanged and mostly backwards compatible. This allowed us to easily port the verification of already existing sequential algorithms to our parallel framework, where they could be used as building blocks for parallel algorithms.

In our approach, the high-level algorithms are phrased as purely functional and sequential algorithms, that, however, already contain hints how they should be refined to imperative parallel algorithms. Only during the last refinement step, performed by Sepref, actual imperative and parallel code is generated.

The trusted code base of our approach is relatively small. To the trusted code base of the original Isabelle-LLVM tool, we only added the semantics and code generation for the parallel combinator.

6.1 Related Work

While there is extensive work on parallel sorting algorithm (e.g. [10,1]), there seems to be almost no work on their formal verification. The only work we are aware of [38] uses the VerCors deductive verifier to prove the permutation property ($\mathit{mset}\ \mathit{xs}' = \mathit{mset}\ \mathit{xs}$), but neither the sortedness property nor termination, of odd-even transposition sort [14].

Concurrent separation logic is used by many verification tools such as VerCors [3], and also formalized in proof assistants, for example in the VST [41] and IRIS [18] projects for Coq [2]. These formalizations contain elaborate concepts to reason about communication between threads via shared memory, and are typically used to verify subtle concurrent algorithms (e.g. [34]). However, they can only reason about partial correctness, and extending them to total correctness is a non-trivial endeavour that is subject of active research [39]. On the other hand, our (less expressive) separation logic naturally supports total correctness, and is sufficient for many parallel algorithms.

6.2 Future Work

An obvious next step for our work is to implement a fractional separation logic [6], 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 in our VCG and Sepref tool.

Another obvious next step is to verify a state-of-the-art parallel sorting algorithm, like Boost's sample sort. As our simple algorithm, sample sort does not require advanced synchronization concepts, and can be implemented only with a parallel combinator.

Finally, the Sepref framework has recently been extended to reason about complexity of (sequential) LLVM programs [16,17]. This line of work could be combined with our parallel extension, to verify the complexity (e.g. work and span) of parallel algorithms.

Extending our approach towards more advanced synchronization like locks or atomic operations may be possible: instead of accessed memory addresses, a thread could return a set of possible traces, which are checked for race-freedom and then combined. However, this would require a nondeterministic LLVM semantics instead of the current deterministic one.

Finally, our framework currently targets multicore CPUs. Another emerging 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 barrier synchronization, which is important in the GPU context, can be easily integrated into our approach.

References

1. M. Asiatici, D. Maiorano, and P. Ienne. How many cpu cores is an fpga worth? lessons learned from accelerating string sorting on a cpu-fpga system. *Journal of Signal Processing Systems*, pages 1–13, 2021.
2. Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.
3. S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The vercors tool set: Verification of parallel and concurrent software. In N. Polikarpova and S. Schneider, editors, *Integrated Formal Methods*, pages 102–110, Cham, 2017. Springer International Publishing.
4. Boost C++ libraries. `https://www.boost.org/`.
5. Boost C++ libraries sorting algorithms. `https://www.boost.org/doc/libs/1_77_0/libs/sort/doc/html/index.html`.
6. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 259–270, New York, NY, USA, 2005. ACM.
7. J. Brunner and P. Lammich. Formal verification of an executable LTL model checker with partial order reduction. *J. Autom. Reasoning*, 60(1):3–21, 2018.
8. L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative functional programming with Isabelle/HOL. In O. A. Mohamed, C. A. Muñoz, and S. Tahar, editors, *TPHOLs 2008*, volume 5170 of *LNCS*, pages 134–149. Springer, 2008.
9. C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS 2007*, pages 366–378, July 2007.
10. J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324, 2008.
11. J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J.-G. Smaus. A fully verified executable LTL model checker. In *CAV*, volume 8044 of *LNCS*, pages 463–478. Springer, 2013.
12. M. Fleury, J. C. Blanchette, and P. Lammich. A verified SAT solver with watched literals using Imperative HOL. In *Proc. of CPP*, pages 158–171, 2018.
13. The GNU C++ library 3.4.28. `https://gcc.gnu.org/onlinedocs/libstdc++/`.
14. A. N. Habermann. Parallel neighbor-sort, Jun 1972.
15. M. Haslbeck and P. Lammich. Refinement with time – refining the run-time of algorithms in isabelle/hol. In *ITP2019: Interactive Theorem Proving*, 6 2019.
16. M. P. L. Haslbeck and P. Lammich. For a few dollars more - verified fine-grained algorithm analysis down to LLVM. *TOPLAS, S.I. ESOP'21*. to appear.
17. M. P. L. Haslbeck and P. Lammich. For a few dollars more - verified fine-grained algorithm analysis down to LLVM. In N. Yoshida, editor, *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*, volume 12648 of *Lecture Notes in Computer Science*, pages 292–319. Springer, 2021.
18. R. Jung, R. Krebbers, J. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018.

19. F. Kammüller, M. Wenzel, and L. C. Paulson. Locales a sectioning concept for isabelle. In Y. Bertot, G. Dowek, L. Théry, A. Hirschowitz, and C. Paulin, editors, *Theorem Proving in Higher Order Logics*, pages 149–165, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
20. G. Klein, R. Kolanski, and A. Boyton. Mechanised separation algebra. In *ITP*, pages 332–337. Springer, Aug 2012.
21. G. Klein, R. Kolanski, and A. Boyton. Separation algebra. *Archive of Formal Proofs*, May 2012. `http://isa-afp.org/entries/Separation_Algebra.html`, Formal proof development.
22. A. Krauss. Recursive definitions of monadic functions. In *Proc. of PAR*, volume 43, pages 1–13, 2010.
23. P. Lammich. Automatic data refinement. In *ITP*, volume 7998 of *LNCS*, pages 84–99. Springer, 2013.
24. P. Lammich. Verified efficient implementation of gabow's strongly connected component algorithm. In *International Conference on Interactive Theorem Proving*, pages 325–340. Springer, 2014.
25. P. Lammich. Refinement to Imperative/HOL. In *ITP*, volume 9236 of *LNCS*, pages 253–269. Springer, 2015.
26. P. Lammich. Efficient verified (UN)SAT certificate checking. In *Proc. of CADE*. Springer, 2017.
27. P. Lammich. The GRAT tool chain - efficient (UN)SAT certificate checking with formal correctness guarantees. In *SAT*, pages 457–463, 2017.
28. P. Lammich. Generating Verified LLVM from Isabelle/HOL. In J. Harrison, J. O'Leary, and A. Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
29. P. Lammich. Efficient verified implementation of introsort and pdqsort. In N. Peltier and V. Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 307–323. Springer, 2020.
30. P. Lammich and R. Meis. A separation logic framework for imperative hol. *Archive of Formal Proofs*, Nov. 2012. `https://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html`, Formal proof development.
31. P. Lammich and S. R. Sefidgar. Formalizing the Edmonds-Karp algorithm. In *Proc. of ITP*, pages 219–234, 2016.
32. P. Lammich and S. R. Sefidgar. Formalizing network flow algorithms: A refinement approach in Isabelle/HOL. *J. Autom. Reasoning*, 62(2):261–280, 2019.
33. P. Lammich and T. Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. In L. Beringer and A. P. Felty, editors, *ITP 2012*, volume 7406 of *LNCS*, pages 166–182. Springer, 2012.
34. G. Mével and J.-H. Jourdan. Formal verification of a concurrent bounded queue in a weak memory model. *Proc. ACM Program. Lang.*, 5(ICFP), Aug. 2021.
35. D. R. MUSSER. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993, 1997.
36. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
37. P. W. O'Hearn. Resources, concurrency and local reasoning. In P. Gardner and N. Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, pages 49–67, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

38. M. Safari and M. Huisman. 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*, pages 257–275. Springer, 2020.
39. S. Spies, L. Gäher, D. Gratzer, J. Tassarotti, R. Krebbers, D. Dreyer, and L. Birkedal. 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*, pages 80–95, 2021.
40. Intel oneapi threading building blocks. `https://software.intel.com/en-us/intel-tbb`.
41. Verified software toolchain project web page. `https://vst.cs.princeton.edu/`.
42. S. Wimmer and P. Lammich. Verified model checking of timed automata. In *TACAS 2018*, pages 61–78, 2018.