Refinement of Parallel Algorithms down to LLVM

Peter Lammich
University of Twente, Netherlands

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

2012 ACM Subject Classification Software and its engineering → Formal software verification; Theory of computation → Semantics and reasoning; Computing methodologies → Parallel algorithms

Keywords and phrases Isabelle,Concurrent Separation Logic,Parallel Sorting,LLVM

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.12

Supplementary Material *Software (Isabelle Formalization)*: `https://www21.in.tum.de/~lammich/isabelle_llvm_par/`

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), a powerful tool to verify efficient sequential software, such as model checkers [10, 7, 38], SAT solvers [24, 25, 11], or graph algorithms [22, 28, 29]. 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 [27]. Our verified parallel sorting algorithm is competitive to state-of-the-art parallel sorting algorithms.

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. 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 [30, 21], and sequential imperative code [23, 26], this paper describes a back end for parallel imperative LLVM code (Section 2).

13th International Conference on Interactive Theorem Proving (ITP 2022).
Editors: June Andronick and Leonardo de Moura; Article No. 12; pp. 12:1–12:20
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

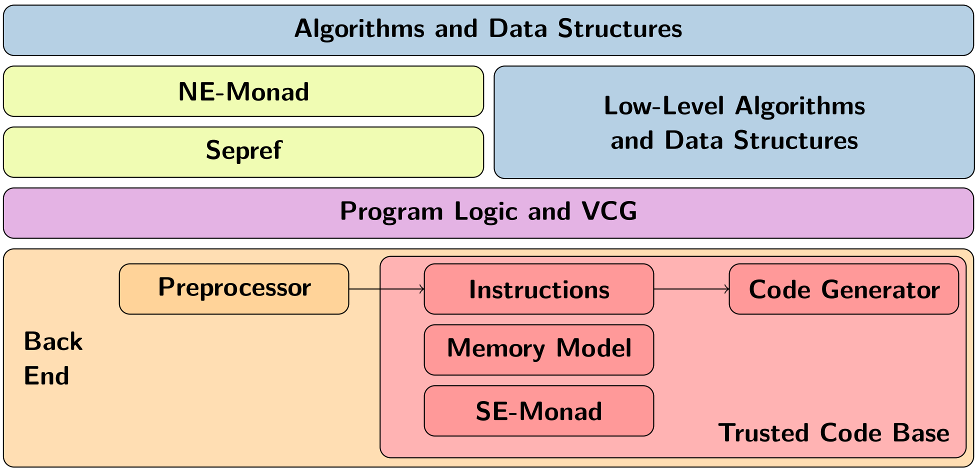

Figure 1 Components of the Isabelle Refinement Framework, with focus on the back end.

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 [33], and our VCG.

At the level of the program logic and VCG, our framework can already 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 [30]. A semi-automatic refinement procedure (Sepref [23, 26]) 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 [10, 7, 38, 24, 25, 11, 22, 28, 29, 27]. 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.

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 [23, 26], the shallow embedding is key to the flexibility and feasibility of the approach. The main idea is to make an execution report the memory that it accesses, 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:

$$
\begin{array}{l}
{}'a\ neM \equiv \texttt{spec}\ ({}'a \Rightarrow bool) \mid \texttt{fail}\\
\texttt{return}\ x \equiv \texttt{spec}\ (\lambda r.\ r{=}x)\\
\texttt{bind fail}\ f \equiv \texttt{fail}\\
\texttt{bind}\ (\texttt{spec}\ P)\ f \equiv \texttt{if}\ \exists x.\ P\ x \wedge f\ x = \texttt{fail then fail}\\
\qquad\qquad\qquad\quad \texttt{else spec}\ (\lambda r.\ \exists x\ Q.\ P\ x \wedge f\ x = \texttt{spec}\ Q \wedge Q\ r)
\end{array}
$$

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.

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:

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

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 $norace :: {}'\rho \Rightarrow {}'\rho \Rightarrow bool$ and $feasible :: {}'\rho \Rightarrow {}'\rho \Rightarrow bool$ be symmetric predicates, and let $combine :: ({}'\rho \times {}'\mu) \Rightarrow ({}'\rho \times {}'\mu) \Rightarrow ({}'\rho \times {}'\mu)$ be a commutative operator to compose two pairs of access reports and states. Then, we define a parallel composition operator for M:

$$
\begin{array}{ll}
(m_1 \mid\mid m_2)\ \mu \equiv & \\
\quad (x_1,r_1,\mu_1) \leftarrow m_1\ \mu;\ (x_2,r_2,\mu_2) \leftarrow m_2\ \mu; & \text{— \textit{execute both strands}}\\
\quad \texttt{assume}\ feasible\ \rho_1\ \rho_2; & \text{— \textit{ignore infeasible combinations}}\\
\quad \texttt{assert}\ norace\ \rho_1\ \rho_2; & \text{— \textit{fail on data race}}\\
\quad \texttt{return}_{ne}\ ((x_1,x_2),\ combine\ (\rho_1,\mu_1)\ (\rho_2,\mu_2)) & \text{— \textit{combine results}}\\
 & \\
\texttt{assume}\ P \equiv \texttt{if}\ P\ \texttt{then return}\ ()\ \texttt{else spec}\ (\lambda\_.\ False) & \\
\texttt{assert}\ P \equiv \texttt{if}\ P\ \texttt{then return}\ ()\ \texttt{else fail} &
\end{array}
$$

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` $(\lambda\_.\ False)$, 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:

$$m_1 \mid\mid m_2 = \mathit{mswap}\ (m_2 \mid\mid m_1) \quad \mathit{where} \quad \mathit{mswap}\ m \equiv (x_1,x_2) \leftarrow m;\ \mathtt{return}\ (x_2,x_1)$$

2.2 Memory Model

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

$$\begin{array}{l}
\textbf{datatype}\ \mathit{addr} \equiv \mathit{ADDR}\ (\mathit{bidx{:}\ nat})\ (\mathit{idx{:}\ nat}) \\
\textbf{datatype}\ \mathit{ptr} \equiv \mathit{PTR_NULL} \ \mid\ \mathit{PTR_ADDR}\ (\mathit{the_addr{:}\ addr}) \\
\textbf{datatype}\ \mathit{val} \equiv \mathit{LL_INT\ lint} \ \mid\ \mathit{LL_STRUCT\ val\ list} \ \mid\ \mathit{LL_PTR\ ptr} \\
\\
\textbf{datatype}\ \mathit{block} \equiv \mathit{FRESH} \ \mid\ \mathit{FREED} \ \mid\ \mathit{is_alloc{:}\ ALLOC}\ (\mathit{vals{:}\ val\ list}) \\
\textbf{typedef}\ \mathit{memory} \equiv \{\ \mu :: \mathit{nat} \Rightarrow \mathit{block}.\ \mathit{finite}\ \{b.\ \mu\ b \neq \mathit{FRESH}\}\ \}
\end{array}$$

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*:

$$\begin{array}{l}
\mathit{valid}\ \mu\ (\mathit{ADDR}\ b\ i) \equiv \mathit{is_alloc}\ (\mu\ b) \wedge i < |\mathit{vals}\ (\mu\ b)| \\
\mathit{get}\ \mu\ (\mathit{ADDR}\ b\ i) \equiv \mathit{vals}\ (\mu\ b)\ !\ i \\
\mathit{put}\ \mu\ (\mathit{ADDR}\ b\ i)\ x \equiv \mu(b := \mathit{ALLOC}\ ((\mathit{vals}\ (\mu\ b))[i{:=}x]))
\end{array}$$

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 sets of read and written addresses, as well as sets of allocated and freed blocks:

$$\begin{array}{l}
\mathit{acc} \equiv (\ r :: \mathit{addr\ set};\ w :: \mathit{addr\ set};\ a :: \mathit{nat\ set};\ f :: \mathit{nat\ set}\ ) \\
0 \equiv (\ \{\},\{\},\{\},\{\}\ ) \\
(r_1,w_1,a_1,f_1) + (r_2,w_2,a_2,f_2) \equiv (\ r_1 \cup r_2,\ w_1 \cup w_2,\ a_1 \cup a_2,\ f_1 \cup f_2\ )
\end{array}$$

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

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

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

$$
\begin{array}{l}
\textit{norace}\ (r_1, w_1, a_1, f_1)\ (r_2, w_2, a_2, f_2) \equiv \\
\quad \texttt{let}\ m_1 = w_1 \cup \{\ \textit{ADDR}\ b\ i.\ b \in a_1 \cup f_1\ \}\ \texttt{in} \\
\quad \texttt{let}\ m_2 = w_2 \cup \{\ \textit{ADDR}\ b\ i.\ b \in a_2 \cup f_2\ \}\ \texttt{in} \\
\qquad (r_1 \cup m_1) \cap m_2 = \{\} \quad \wedge \quad m_1 \cap (r_2 \cup m_2) = \{\}
\end{array}
$$

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:

$$
\begin{array}{l}
\textit{invar}_M\ \ c \equiv \forall \mu\ P.\ c\ \mu = \texttt{spec}\ P \implies \\
\quad (\forall x\ \rho\ \mu'.\ P\ (x,\rho,\mu') \implies \textit{consistent}\ \mu\ \rho\ \mu') \\
\wedge\ (\forall B.\ \textit{finite}\ B \implies (\exists x\ \rho\ \mu'.\ P\ (x,\rho,\mu') \wedge \rho.a \cap B = \{\}\ ))
\end{array}
$$

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 $(\rho',\mu') = \textit{combine}\ (\rho_1,\mu_1)\ (\rho_2,\mu_2)$, and *addr* a valid address in μ'. Then

$$
\begin{array}{ll}
(1) & \mu'\ b = \textit{FRESH} \longleftrightarrow \mu\ b = \textit{FRESH} \wedge b \notin a_1 \cup a_2 \\
(2) & \textit{is\_alloc}\ (\mu'\ b) \longleftrightarrow (\textit{is\_alloc}\ (\mu\ b) \vee b \in a_1 \cup a_2) \wedge b \notin f_1 \cup f_2 \\
(3) & \mu'\ b = \textit{FREED} \longleftrightarrow \mu\ b = \textit{FREED} \vee b \in f_1 \cup f_2 \\
 & \\
(4) & a \in w_1 \vee b \in a_1 \implies \textit{get\_addr}\ \mu'\ a = \textit{get\_addr}\ \mu_1\ a \\
(5) & a \in w_2 \vee b \in a_2 \implies \textit{get\_addr}\ \mu'\ a = \textit{get\_addr}\ \mu_2\ a \\
(6) & a \notin w_1 \cup w_2 \vee b \notin a_1 \cup a_2 \implies \textit{get\_addr}\ \mu'\ a = \textit{get\_addr}\ \mu\ a
\end{array}
$$

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 or allocated at all keep their original content (6).

2.4 LLVM Instructions

Based on the M-monad, we define shallowly embedded LLVM instructions. For most instructions, this is analogous to the sequential case [26]. The exceptions are memory allocation, which nondeterministically allocates some available block (the original formalization deterministically counted up the block indexes), and an instruction for parallel function call:

$$
\textit{llc\_par}\ f\ g\ a\ b \equiv f\ a\ ||\ g\ b
$$

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 [36] 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 describe how to 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 [8]. 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 [8, 20]. We note that separation algebras naturally extend over functions and pairs, in a pointwise manner.

▶ **Example 1.** (Trivial Separation Algebra) The type α *option* = *None* | *Some* α 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:

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

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

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 ↑-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:

$$wp\ m\ Q\ \mu \equiv \texttt{case}\ m\ \mu\ \texttt{of spec}\ Q' \Rightarrow \forall x\ \rho\ \mu'.\ Q'\ (x,\rho,\mu') \implies Q\ x\ \rho\ \mu' \mid \texttt{fail} \Rightarrow False$$

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*:

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

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

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 a weakest precondition with access restrictions, 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:

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

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 [33]:

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

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

For memory operations, we prove:

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

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 [26], 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, focusing on our changes to support parallel algorithms.

4.1 Abstract Programs

Abstract programs are shallowly embedded into the nondeterminism error monad *'a neM* (cf. Section 2.1). They are purely functional, not modifying memory, or differentiating between sequential and parallel execution. We define a *refinement ordering* on *neM*:

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

Intuitively, $m_1 \le m_2$ means that m_1 returns fewer possible results than m_2, and may only fail if m_2 may fail. Note that $\le$ 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:

$$m \le \mathtt{assert}\ P;\ \mathtt{spec}\ x.\ Q\ x$$

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 [30].

4.2 The Sepref Tool

The Sepref tool [23, 26] 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 imperative 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:

$$\begin{aligned}&\mathit{hnr}\ \Gamma\ c_\dagger\ \Gamma'\ R\ \mathit{CP}\ c \equiv\\ &\quad c \neq \mathtt{fail} \implies \mathit{ht}\ \Gamma\ c_\dagger\ (\lambda x_\dagger.\ \exists x.\ \Gamma' * R\ x\ x_\dagger * \uparrow(\mathtt{return}\ x \leq c \wedge \mathit{CP}\ x_\dagger))\end{aligned}$$

That is, either the abstract program c fails, or for a memory described by assertion Γ, the LLVM program $c_\dagger$ succeeds with $x_\dagger$, such that the new memory is described by $\Gamma' * R\ x\ x_\dagger$, for a possible result x 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. Moreover it allows us to assume that assertions actually hold during the refinement proof:

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

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

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

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. On arrays, indexing and updating of arrays is implemented by:

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

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

$$\begin{aligned}&\mathit{hnr}\ (\mathit{arr}_A\ \mathit{xs}\ \mathit{xs}_\dagger * \mathit{idx}_A\ i\ i_\dagger)\ (\mathit{aget}\ \mathit{xs}_\dagger\ i_\dagger)\ (\mathit{arr}_A\ \mathit{xs}\ \mathit{xs}_\dagger * \mathit{idx}_A\ i\ i_\dagger)\ \mathit{id}_A\ (\lambda\_.\ \mathit{True})\ (\mathit{lget}\ \mathit{xs}\ i)\\ &\mathit{hnr}\ (\mathit{arr}_A\ \mathit{xs}\ \mathit{xs}_\dagger * \mathit{idx}_A\ i\ i_\dagger)\ (\mathit{aset}\ \mathit{xs}_\dagger\ i_\dagger\ x)\ (\mathit{idx}_A\ i\ i_\dagger)\ \mathit{arr}_A\ (\lambda r.\ r{=}\mathit{xs}_\dagger)\ (\mathit{lset}\ \mathit{xs}\ i\ x)\end{aligned}$$

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 $i_\dagger$ ($\mathit{idx}_A\ i\ i_\dagger$), the *aget* operation will return the same result as the *lget*

[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.

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 $xs_\dagger$. This information is important for refining to parallel programs on disjoint parts of an array (cf. Section 4.3).

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 *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 [26] 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 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:

$$\begin{array}{l}
\texttt{with_split}\ i\ xs\ f \equiv \\
\quad \texttt{assert}\ (i < |xs|); \\
\quad (xs_1, xs_2) \leftarrow f\ (take\ i\ xs)\ (drop\ i\ xs); \\
\quad \texttt{assert}\ (|xs_1| = i \wedge |xs_2| = |xs| - i); \\
\quad \texttt{return}\ (xs_1 @ xs_2)
\end{array}$$

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*:

$$\begin{array}{l}
awith_split\ i\ xs_\dagger\ f_\dagger \equiv xs_{\dagger 2} \leftarrow ll_ofs_ptr\ xs_\dagger\ i;\ f_\dagger\ xs_\dagger\ xs_{\dagger 2};\ \texttt{return}\ xs_\dagger \\
\\
hnr\ (arr_A\ xs_1\ xs_{\dagger 1} * arr_A\ xs_2\ xs_{\dagger 2})\ (f_\dagger\ xs_{\dagger 1}\ xs_{\dagger 2})\ \Box \\
\quad (arr_A \times arr_A)\ (\lambda(xs_{\dagger 1}', xs_{\dagger 2}').\ xs_{\dagger 1}' = xs_{\dagger 1} \wedge xs_{\dagger 2}' = xs_{\dagger 2}) \\
\quad (f\ xs_1\ xs_2) \\
\Longrightarrow \\
hnr\ (arr_A\ xs\ xs_\dagger * idx_A\ i\ i_\dagger)\ (awith_split\ i_\dagger\ xs_\dagger\ f_\dagger) \\
\quad (idx_A\ i\ i_\dagger)\ (\lambda xs\ xs_\dagger.\ arr_A\ xs\ xs_\dagger)\ (\lambda xs_\dagger'.\ xs_\dagger' = xs_\dagger) \\
\quad (\texttt{with_split}\ i\ xs\ f)
\end{array}$$

The refinement of the function f to $f_\dagger$ requires an additional proof that the returned pointers are equal to the argument pointers ($xs_{\dagger 1}' = xs_{\dagger 1} \wedge xs_{\dagger 2}' = xs_{\dagger 2}$). Sepref tries to prove that automatically, using a simple heuristics.

4.4 Refinement to Parallel Execution

The purely functional 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 ≡ x ← f a; y ← g b; return (x,y)

 hnr Ax (f† x†) Ax' Rx CP1 (f x) ∧ hnr Ay (g† y†) Ay' Ry CP2 (g y)
⟹
 hnr (Ax ∗ Ay) (llc_par f† g† x† y†) (Ax' ∗ Ay') (Rx × Ry)
   (λ(x†',y†'). CP1 x†' ∧ CP2 y†') (npar f g x y)
```

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:

```
sort_spec xs = spec xs'. mset xs'=mset xs ∧ sorted xs
```

That is, we return a sorted permutation of the original list. Note that this is a standard specification of sorting in Isabelle. Reusing the existing development of an abstract introsort algorithm [27], we easily prove with a few refinement steps that the following abstract algorithm implements *sort_spec*:

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

This algorithm is derived from the well-known quicksort and introsort algorithms [32]: like quicksort, it partitions the list (line 7), and then recursively sorts the partitions in parallel (l. 11). 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. 5). Similarly, when the partitioning is very unbalanced (l. 8), we sort the partitions sequentially (l. 10). 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].

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 *pdqsort* [27] with the in-place proofs: once we had amended the refinement statements, and bug-fixed the pointer equality proving heuristics that we added to Sepref, the proofs were automatic.

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

$$hnr\ (arr_A\ xs\ xs_\dagger\ *\ idx_A\ n\ n_\dagger)\ (psort_\dagger\ xs_\dagger\ n_\dagger)\ (idx_A\ n\ n_\dagger)\ arr_A\ (\lambda r.\ r = xs_\dagger)\ (psort\ xs\ n)$$

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:

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

That is, for a pointer $xs_\dagger$ to an array, whose contents are 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 contents are now 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, its correctness does not depend on the refinement steps, the Sepref tool, or the VCG.

5.2 A Sampling Partitioner

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. The highly optimized partitioning algorithms that we use swap the pivot to the front of the partition, such that we need to determine its index, rather than just its value. We simply use quicksort to find the median[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 [19], we parametrize over the comparison function. Moreover, we thread through an extra parameter for the data captured by the closure:

---

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

[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.

$$
\begin{array}{l}
\textbf{locale}\ \mathit{pcmp} = \\
\quad \textbf{fixes}\ \mathit{lt} :: {}'p \Rightarrow {}'e \Rightarrow {}'e \Rightarrow \mathit{bool}\ \textbf{and}\ \mathit{lt}_\dagger :: {}'p_\dagger \Rightarrow {}'e_\dagger \Rightarrow {}'e_\dagger \Rightarrow \mathit{bool} \\
\qquad \textbf{and}\ \mathit{par}_A :: {}'p \Rightarrow {}'p_\dagger \Rightarrow \mathit{assn}\ \textbf{and}\ \mathit{elem}_A :: {}'e \Rightarrow {}'e_\dagger \Rightarrow \mathit{assn} \\
\quad \textbf{assumes}\ \forall p.\ \mathit{weak\_ordering}\ (\mathit{lt}\ p) \\
\quad \textbf{assumes}\ \mathit{hnr}\ (\mathit{par}_A\ p\ \mathit{pi} * \mathit{elem}_A\ a\ \mathit{ai} * \mathit{elem}_A\ b\ \mathit{bi})\ (\mathit{lt}_\dagger\ \mathit{pi}\ \mathit{ai}\ \mathit{bi}) \\
\qquad (\mathit{par}_A\ p\ \mathit{pi} * \mathit{elem}_A\ a\ \mathit{ai} * \mathit{elem}_A\ b\ \mathit{bi})\ \ (\mathit{bool}_A)\ \ (\lambda\_.\ \mathit{True})\ (\mathit{lt}\ p\ a\ b)
\end{array}
$$

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 algorithm is 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 a sorting algorithm for an actual compare function, we have to instantiate the locale, providing an abstract and concrete compare function, along with a proof that the abstract function is a weak ordering, and the concrete function refines the abstract one. For our example of sorting indexes into an array, where the array elements are, themselves, compared by a parametrized function *lt*, we get:

$$
\begin{array}{l}
\textbf{interpretation}\ \mathit{idx}\text{: }\mathit{pcmp}\ \mathit{lt\_idx}\ \mathit{lt\_idx}_\dagger\ (\mathit{par}_A \times \mathit{arr}_A)\ \mathit{idx}_A \quad \langle \mathit{proof} \rangle \\
\\
\mathit{lt\_idx}\ (p, \mathit{xs})\ i\ j \equiv \mathit{lt}\ p\ (\mathit{xs}!i)\ (\mathit{xs}!j) \\
\mathit{lt\_idx}_\dagger\ (p_\dagger, \mathit{xs}_\dagger)\ i_\dagger\ j_\dagger \equiv x_\dagger \leftarrow \mathit{aget}\ \mathit{xs}_\dagger\ i_\dagger;\ y_\dagger \leftarrow \mathit{aget}\ \mathit{xs}_\dagger\ j_\dagger;\ \mathit{lt}_\dagger\ p_\dagger\ x_\dagger\ y_\dagger
\end{array}
$$

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 Code Generation

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

$$
\begin{array}{l}
\textbf{interpretation}\ \mathit{unat}\text{: }\mathit{pcmp}\ (\lambda\_.\ <)\ (\lambda\_.\ \mathit{ll\_icmp\_ult})\ \mathit{unat}_A^{64}\ \langle \mathit{proof} \rangle \\
\textbf{interpretation}\ \mathit{str}\text{: }\mathit{pcmp}\ (\lambda\_.\ <)\ (\lambda\_.\ \mathit{strcmp})\ \mathit{str}_A^{64}\ \langle \mathit{proof} \rangle
\end{array}
$$

This yields implementations $unat.psort_\dagger$ and $str.psort_\dagger$, and automatically proves instantiated versions of the correctness theorems.

In a last step, we use our code generator to generate actual LLVM text, as well as a C header file with the signatures of the generated functions[6]:

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

[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++ [17].

[6] 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.

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.

5.4 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 [12], and *sample_sort* from the Boost C++ libraries [4, 5]. We have benchmarked the algorithm on two different machines, and various input distributions. The results are shown in Figure 2. While our verified 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 verified implementation is already useful in practice, and leave further optimizations to future work.

Finally, we measured the speedup that the implementations achieve 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 implementation achieves lower speedups than the state-of-the-art on the highly parallel server. Again, we leave further optimizations 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, Sepref tool, and Refinement Framework for abstract programs, ultimately prove a correctness theorem that only depends on our shallowly embedded semantics.

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, at least on moderately parallel hardware.

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 [26]. 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 [27].

6.1 Related Work

While there is extensive work on parallel sorting algorithm (e.g. [9, 1]), there seems to be almost no work on their formal verification. The only work we are aware of is a distributed merge sort algorithm [16], for which "no effort has been made to make it efficient"[16, Sec. 2], nor any executable code has been generated or benchmarked. Another verification [34] uses

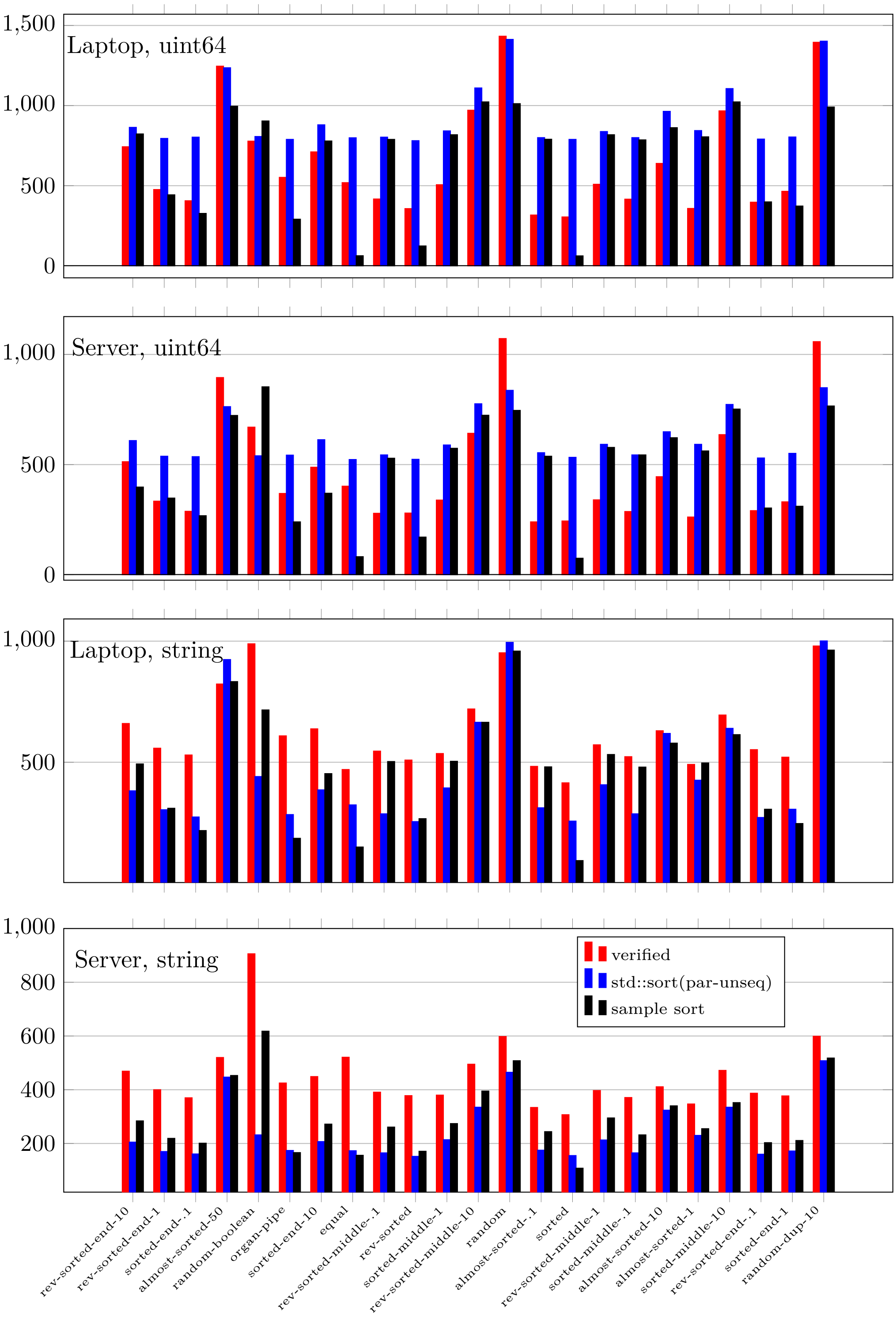

Figure 2 Runtimes in milliseconds for sorting various distributions of unsigned 64 bit integers and strings with our verified parallel sorting algorithm, C++'s standard parallel sorting algorithm, and Boost's parallel sample sort algorithm. The experiments were performed on a server machine with 22 AMD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

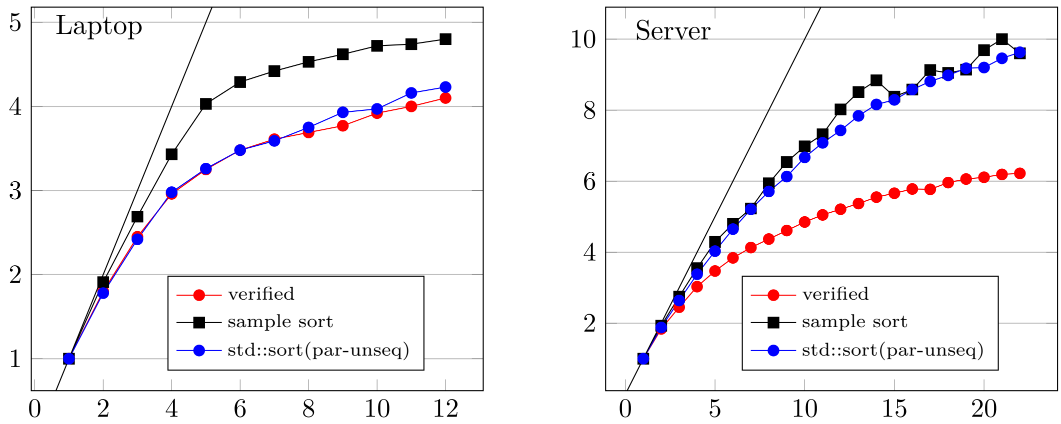

Figure 3 Speedup of the various implementations, for sorting unsigned 64 bit integers with a random distribution, on a server with 22 AMD Opteron 6176 cores and 128GiB of RAM, and a laptop with a 6 core (12 threads) i7-10750H CPU and 32GiB of RAM. The x axis ranges over the number of cores, and the y-axis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

the VerCors deductive verifier to prove the permutation property ($\mathit{mset}\ xs' = \mathit{mset}\ xs$) of odd-even transposition sort [13], but neither the sortedness property nor termination.

Concurrent separation logic is used by many verification tools such as VerCors [3], and also formalized in proof assistants, for example in the VST [37] 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 partial correctness of subtle concurrent algorithms (e.g. [31]). Reasoning about total correctness is more complicated in the step-indexed separation logic provided by IRIS, and currently only supported for sequential programs [35]. 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 [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.

Another obvious next step is to verify a state-of-the-art parallel sorting algorithm, like Boost's sample sort. Like our current 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 [14, 15]. 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 report a set of possible traces, which are checked for race-freedom and then combined.

Finally, 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 barrier synchronization, which is important in the GPU context, can be integrated into our approach.

References

1 Mikhail Asiatici, Damian Maiorano, and Paolo 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 Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.

3 Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The vercors tool set: Verification of parallel and concurrent software. In Nadia Polikarpova and Steve 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 Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew 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. URL: `http://doi.acm.org/10.1145/1040305.1040327`, `doi:10.1145/1040305.1040327`.

7 Julian Brunner and Peter Lammich. Formal verification of an executable LTL model checker with partial order reduction. *J. Autom. Reasoning*, 60(1):3–21, 2018. `doi:10.1007/s10817-017-9418-4`.

8 C. Calcagno, P.W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS 2007*, pages 366–378, July 2007.

9 Jatin Chhugani, Anthony D Nguyen, Victor W Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324, 2008.

10 Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A fully verified executable LTL model checker. In *CAV*, volume 8044 of *LNCS*, pages 463–478. Springer, 2013.

11 Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. A verified SAT solver with watched literals using Imperative HOL. In *Proc. of CPP*, pages 158–171, 2018.

12 The GNU C++ library 3.4.28. `https://gcc.gnu.org/onlinedocs/libstdc++/`.

13 A. Nico Habermann. Parallel neighbor-sort, Jun 1972. URL: `https://kilthub.cmu.edu/articles/journal_contribution/Parallel_neighbor-sort_or_the_glory_of_the_induction_principle_/6608258/1`, `doi:10.1184/R1/6608258.v1`.

14 Maximilian P. L. Haslbeck and Peter Lammich. For a few dollars more - verified fine-grained algorithm analysis down to LLVM. *TOPLAS, S.I. ESOP'21.* to appear.

15 Maximilian P. L. Haslbeck and Peter Lammich. For a few dollars more - verified fine-grained algorithm analysis down to LLVM. In Nobuko 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. `doi:10.1007/978-3-030-72019-3\_11`.

16 Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: Session-type based reasoning in separation logic. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. `doi:10.1145/3371074`.

17 Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley Professional, 2nd edition, 2012.

18 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018. doi:10.1017/S0956796818000151.

19 Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales a sectioning concept for isabelle. In Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin, editors, *Theorem Proving in Higher Order Logics*, pages 149–165, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

20 Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Mechanised separation algebra. In *ITP*, pages 332–337. Springer, Aug 2012.

21 Peter Lammich. Automatic data refinement. In *ITP*, volume 7998 of *LNCS*, pages 84–99. Springer, 2013.

22 Peter Lammich. Verified efficient implementation of gabow's strongly connected component algorithm. In *International Conference on Interactive Theorem Proving*, pages 325–340. Springer, 2014.

23 Peter Lammich. Refinement to Imperative/HOL. In *ITP*, volume 9236 of *LNCS*, pages 253–269. Springer, 2015.

24 Peter Lammich. Efficient verified (UN)SAT certificate checking. In *Proc. of CADE*. Springer, 2017.

25 Peter Lammich. The GRAT tool chain - efficient (UN)SAT certificate checking with formal correctness guarantees. In *SAT*, pages 457–463, 2017.

26 Peter Lammich. Generating Verified LLVM from Isabelle/HOL. In John Harrison, John O'Leary, and Andrew 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. URL: http://drops.dagstuhl.de/opus/volltexte/2019/11077, doi:10.4230/LIPIcs.ITP.2019.22.

27 Peter Lammich. Efficient verified implementation of introsort and pdqsort. In Nicolas Peltier and Viorica 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. doi:10.1007/978-3-030-51054-1_18.

28 Peter Lammich and S. Reza Sefidgar. Formalizing the Edmonds-Karp algorithm. In *Proc. of ITP*, pages 219–234, 2016.

29 Peter Lammich and S. Reza Sefidgar. Formalizing network flow algorithms: A refinement approach in Isabelle/HOL. *J. Autom. Reasoning*, 62(2):261–280, 2019. doi:10.1007/s10817-017-9442-4.

30 Peter Lammich and Thomas Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. In Lennart Beringer and Amy P. Felty, editors, *ITP 2012*, volume 7406 of *LNCS*, pages 166–182. Springer, 2012.

31 Glen Mével and Jacques-Henri Jourdan. Formal verification of a concurrent bounded queue in a weak memory model. *Proc. ACM Program. Lang.*, 5(ICFP), August 2021. doi:10.1145/3473571.

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

33 Peter W. O'Hearn. Resources, concurrency and local reasoning. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, pages 49–67, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

34 Mohsen Safari and Marieke 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.

35 Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars 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.

36 Intel oneapi threading building blocks. `https://software.intel.com/en-us/intel-tbb`.

37 Verified software toolchain project web page. `https://vst.cs.princeton.edu/`.

38 Simon Wimmer and Peter Lammich. Verified model checking of timed automata. In *TACAS 2018*, pages 61–78, 2018.