Refinement of Parallel Algorithms down to LLVM

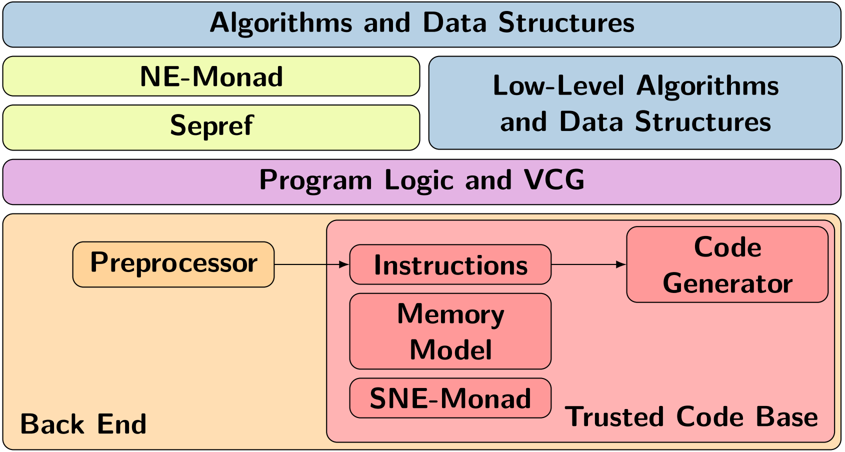
Peter Lammich

University of Twente

August 2022 @ FLOC $\in$ Haifa

The Isabelle Refinement Framework

Stepwise Refinement approach to verified algorithms in Isabelle/HOL



Isabelle LLVM Backend

- Shallowly embedded LLVM semantics (fragment just big enough)
- Structured control flow (compiled by code generator)
- Features: int+float, recursive struct, C header file generation, ...

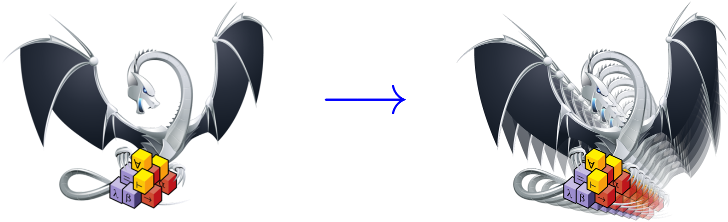
```
fib:: 64 word ⇒ 64 word llM
fib n = do {
  t ← ll_icmp_ule n 1;
  llc_if t
    (return n)
    (do {
      n1 ← ll_sub n 1;
      a  ← fib n1;
      n2 ← ll_sub n 2;
      b  ← fib n2;
      c  ← ll_add a b;
      return c
    }) }
```

```
export_llvm
  fib is uint64_t fib(uint64_t)
```

Contribution

Add parallelism to Isabelle Refinement Framework

- Amend LLVM backend, VCG, Sepref
- Verified, competitive parallel sorting algorithm



Isabelle LLVM Back End

- Shallow embedding into monad

  $\alpha\ M =$

Isabelle LLVM Back End

- Shallow embedding into error-monad

  $\alpha\ M = \alpha\ \text{option}$

None — undefined behaviour, nontermination

Isabelle LLVM Back End

- Shallow embedding into ndet-error-monad

  $\alpha\ M = \alpha\ \text{set option}$

None — undefined behaviour, nontermination
α set — set of possible results

Isabelle LLVM Back End

- Shallow embedding into state-ndet-error-monad
  $\alpha\ M = \mu \to (\alpha \times \mu)\ \text{set option}$

None — undefined behaviour, nontermination
α set — set of possible results
μ — memory

Isabelle LLVM Back End

- Shallow embedding into state-ndet-error-monad with access reports
  $\alpha\ M = \mu \to (\alpha \times \rho \times \mu)$ set option

None — undefined behaviour, nontermination

α set — set of possible results

μ — memory

ρ — access report: read/written/allocated/freed addresses

Isabelle LLVM Back End

- Shallow embedding into state-ndet-error-monad with access reports
  $\alpha\ M = \mu \rightarrow (\alpha \times \rho \times \mu)\ \text{set option}$

None — undefined behaviour, nontermination
α set — set of possible results
μ — memory
ρ — access report: read/written/allocated/freed addresses

Basic block: $x_1 \leftarrow op_1; \ldots; return \ldots$

Isabelle LLVM Back End

- Shallow embedding into state-ndet-error-monad with access reports
  $\alpha\ M = \mu \to (\alpha \times \rho \times \mu)$ set option

None — undefined behaviour, nontermination
α set — set of possible results
μ — memory
ρ — access report: read/written/allocated/freed addresses

Basic block: $x_1 \leftarrow op_1; \ldots; return \ldots$
if-then-else, while — structured control flow (compiled by code-gen)

Parallel Operator

Parallel Operator

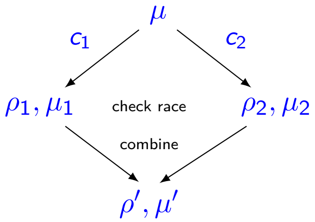
- $c_1 \parallel c_2$ — execute in parallel, fail on data race

Parallel Operator

- $c_1 \parallel c_2$ — execute in parallel, fail on data race
- Use access reports to detect data races

Parallel Operator

- $c_1 \parallel c_2$ — execute in parallel, fail on data race
- Use access reports to detect data races



Parallel Operator

- $c_1 \parallel c_2$ — execute in parallel, fail on data race
- Use access reports to detect data races

$$
\begin{array}{ll}
(\mathsf{c}_1 \parallel \mathsf{c}_2)\ \mu \equiv & \\
\quad (\mathsf{r}_1,\rho_1,\mu_1) \leftarrow \mathsf{c}_1\ \mu & \text{— execute first strand} \\
\quad (\mathsf{r}_2,\rho_2,\mu_2) \leftarrow \mathsf{c}_2\ \mu & \text{— execute second strand} \\
\quad \texttt{assume}\ \rho_1.\mathsf{alloc} \cap \rho_2.\mathsf{alloc} = \emptyset & \text{— ignore infeasible combinations} \\
\quad \texttt{assert}\ \mathsf{no\_race}\ \rho_1\ \rho_2 & \text{— fail on data race} \\
\quad (\rho',\mu') = \mathsf{combine}\ \rho_1\ \mu_1\quad \rho_2\ \mu_2 & \text{— combine states} \\
\quad \texttt{return}\ ((\mathsf{r}_1,\mathsf{r}_2),\ \rho',\ \mu') &
\end{array}
$$

Parallel Operator

- $c_1 \parallel c_2$ — execute in parallel, fail on data race
- Use access reports to detect data races

```
(c₁ || c₂) μ ≡
  (r₁,ρ₁,μ₁) ← c₁ μ                          — execute first strand
  (r₂,ρ₂,μ₂) ← c₂ μ                          — execute second strand
  assume ρ₁.alloc ∩ ρ₂.alloc = ∅             — ignore infeasible combinations
  assert no_race ρ₁ ρ₂                       — fail on data race
  (ρ′,μ′) = combine ρ₁ μ₁   ρ₂ μ₂            — combine states
  return ((r₁,r₂), ρ′, μ′)
```

Sanity checks: prove (as type invariant):

- access reports match actually modified addresses
- there is at least one execution.

Parallel Operator

- $c_1 \parallel c_2$ — execute in parallel, fail on data race
- Use access reports to detect data races
- Code-gen: external function + some glue code

Parallel Operator

- $c_1 \parallel c_2$ — execute in parallel, fail on data race
- Use access reports to detect data races
- Code-gen: external function + some glue code

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

Separation Logic

{P} c {Q} iff

$\forall\mu$ a af. $\alpha\ \mu$ = a + af $\wedge$ P a — for all memories that satisfy precond
$\Longrightarrow$ $\exists$S. c μ = Some S — program does not fail
$\wedge\ \forall$(r,ρ,μ') $\in$ S. — and all possible results
$\exists$ a'. $\alpha\ \mu'$ = a'+af $\wedge$ Q r a' — satisfy postcond
$\wedge$ disjoint ρ af — and accessed memory not in frame

α: abstracts memory into separation algebra
Baked-in frame rule

Separation Logic

$\{P\}$ c $\{Q\}$ iff

$$\forall \mu\ a\ af.\ \alpha\ \mu = a + af \wedge P\ a$$ — for all memories that satisfy precond

$$\implies \exists S.\ c\ \mu = \text{Some } S$$ — program does not fail

$$\wedge\ \forall (r,\rho,\mu') \in S.$$ — and all possible results

$$\exists\ a'.\ \alpha\ \mu' = a' + af \wedge Q\ r\ a'$$ — satisfy postcond

$$\wedge\ \text{disjoint}\ \rho\ af$$ — and accessed memory not in frame

α: abstracts memory into separation algebra

Baked-in frame rule

We prove the standard Hoare-rules, e.g. dj-conc rule:

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

Separation Logic

$\{P\}$ c $\{Q\}$ iff

$$
\begin{array}{ll}
\forall \mu \text{ a af. } \alpha\ \mu = \text{a} + \text{af} \wedge \text{P a} & \text{— for all memories that satisfy precond} \\
\Longrightarrow\ \exists \text{S. c } \mu = \text{Some S} & \text{— program does not fail} \\
\quad \wedge\ \forall (\text{r},\rho,\mu') \in \text{S.} & \text{— and all possible results} \\
\qquad \exists\ \text{a}'.\ \alpha\ \mu' = \text{a}'+\text{af} \wedge \text{Q r a}' & \text{— satisfy postcond} \\
\qquad \wedge \text{ disjoint } \rho \text{ af} & \text{— and accessed memory not in frame}
\end{array}
$$

α: abstracts memory into separation algebra
Baked-in frame rule
We prove the standard Hoare-rules, e.g. dj-conc rule:

$$
\{P_1\}\ c_1\ \{Q_1\} \quad \wedge \quad \{P_2\}\ c_2\ \{Q_2\}
$$
$$
\Longrightarrow
$$
$$
\{P_1 * P_2\}\ c_1 \mid\mid c_2\ \{\lambda(r_1,r_2).\ Q_1\ r_1 * Q_2\ r_2\}
$$

VCG helps with proof automation

Sepref

- Semi-automatic data refinement.
  - from purely functional nres-error monad
  - to (shallowly embedded) LLVM semantics
  - place pure data on heap (eg. lists $\rightarrow$ arrays)

Refinement Relation

hnr Γ $c_\dagger$ Γ' R CP c

iff

$$c = \text{Some } S \implies \{\Gamma\}\ c_\dagger\ \{\lambda r_\dagger.\ \exists r.\ R\ r\ r_\dagger * \Gamma' * r \in S * CP\ r_\dagger\}$$

$c_\dagger / c$ concrete/abstract programs

Γ / Γ' refinements for variables in $c_\dagger$ and c, before/after execution

R refinement for result

CP concrete (pointer) equalities

Refinement Relation

$\text{hnr } \Gamma \ c_\dagger \ \Gamma' \ R \ CP \ c$

iff

$c = \text{Some } S \implies \{\Gamma\} \ c_\dagger \ \{\lambda r_\dagger. \ \exists r. \ R \ r \ r_\dagger * \Gamma' * r \in S * CP \ r_\dagger\}$

$c_\dagger / c$ concrete/abstract programs

Γ / Γ' refinements for variables in $c_\dagger$ and c, before/after execution

R refinement for result

CP concrete (pointer) equalities

Sepref: syntactically guided heuristics
synthesize $c_\dagger$, Γ', R, CP from Γ and c + annotations

Example

```
hnr
  ( arr xs p ∗ idx n i )                     — argument refinements
  ( store x (p+i); return p )                — concrete program: store, return pointer
  ( idx n i )                                — original refinement for array is gone
  ( arr )                                    — result refinement
  ( λr. r=p )                                — concrete result is same as argument p
  ( return xs[n:=x] )                        — abstract program: functional list update
```

arr refines list to array

idx refines nat to size_t

Refinement Building Blocks

- Patterns and strategies for refinement

Refinement Building Blocks

- Patterns and strategies for refinement
- Sequential: e.g., nat $\rightarrow$ size_t, list $\rightarrow$ array, fold $\rightarrow$ loop

Refinement Building Blocks

- Patterns and strategies for refinement
- Sequential: e.g., nat $\rightarrow$ size_t, list $\rightarrow$ array, fold $\rightarrow$ loop
- Here: parallelization and array-splitting

Parallelization

- Refine sequential (independent) execution to parallel execution

  $\mathsf{hnr}\ \Gamma_1\ c_{\dagger 1}\ \Gamma_1'\ R_1\ CP_1\ c_1 \quad \wedge \quad \mathsf{hnr}\ \Gamma_2\ c_{\dagger 2}\ \Gamma_2'\ R_2\ CP_2\ c_2$
  $\Longrightarrow$
  $\mathsf{hnr}\ (\Gamma_1 * \Gamma_2)\ (c_{\dagger 1} \parallel c_{\dagger 2})\ (\Gamma_1' * \Gamma_2')\ (R_1 \times R_2)\ (CP_1 \wedge CP_2)\ (\mathsf{fpar}\ c_1\ c_2)$

where $\mathsf{fpar}\ c_1\ c_2 \equiv r_1 \leftarrow c_1;\ r_2 \leftarrow c_2;\ \mathtt{return}\ (r_1,r_2)$
fpar is annotation for Sepref to request parallelization

Array Splitting

- Work on two separate parts of same array (e.g. in parallel)
- Functionally:

```
with_split n xs f =
  (xs₁,xs₂) ← f (take n xs) (drop n xs)
  return xs₁ @ xs₂
```

- Imperative with arrays

```
with_split_arr i p f† =
  p₂ ← ofs_ptr p i
  f† p p₂
  return p
```

- Refinement rule uses *CP*-predicates to ensure that $f_\dagger$ is in-place

Parallel Quicksort

(Simplified) functional algorithm:

$$
\begin{array}{l}
\mathsf{qsort}\ \mathsf{xs} \equiv \\
\quad \texttt{if}\ |\mathsf{xs}| < 1\ \texttt{then}\ \texttt{return}\ \mathsf{xs} \\
\quad \texttt{else} \\
\qquad (\mathsf{xs},\mathsf{m}) \leftarrow \mathsf{partition}\ \mathsf{xs} \\
\qquad \mathsf{with\_split}\ \mathsf{m}\ \mathsf{xs}\ (\lambda \mathsf{xs}_1\ \mathsf{xs}_2. \\
\qquad\quad \mathsf{fpar}\ (\mathsf{qsort}\ \mathsf{xs}_1)\ (\mathsf{qsort}\ \mathsf{xs}_2) \\
\qquad)
\end{array}
$$

Correctness statement:

$$
\begin{array}{l}
\mathsf{qsort}\ \mathsf{xs} \le \texttt{spec}\ \mathsf{xs}'.\ \mathsf{sorted}\ \mathsf{xs}' \\
\qquad\qquad\qquad \wedge\ \mathsf{mset}\ \mathsf{xs}' = \mathsf{mset}\ \mathsf{xs}
\end{array}
$$

Parallel Quicksort

(Simplified) functional algorithm:

```
qsort xs ≡
  if |xs| < 1 then return xs
  else
    (xs,m) ← partition xs
    with_split m xs (λxs₁ xs₂.
      fpar (qsort xs₁) (qsort xs₂)
    )
```

Correctness statement:

$$\text{qsort xs} \leq \texttt{spec}\ \text{xs}'.\ \text{sorted xs}' \wedge \text{mset xs}' = \text{mset xs}$$

we have actually verified some 'extras':

- use sequential sorting for small, unbalanced, or deep partitions
- partitioning uses c=64 equidistant samples
- sequential sorting: using verified pdq-sort (competitive with std::sort)

Correctness theorem and TCB

Sepref generates $\mathit{qsort}_\dagger$ and theorem

$\text{hnr (arr xs p} * \text{idx } |\text{xs}| \text{ n) (qsort}_\dagger \text{ p n) (idx } |\text{xs}| \text{ n) arr (=p) (qsort xs)}$

Correctness theorem and TCB

Sepref generates $\mathit{qsort}_\dagger$ and theorem

$\mathsf{hnr}\ (\mathsf{arr}\ \mathsf{xs}\ \mathsf{p} * \mathsf{idx}\ |\mathsf{xs}|\ \mathsf{n})\ (\mathsf{qsort}_\dagger\ \mathsf{p}\ \mathsf{n})\ (\mathsf{idx}\ |\mathsf{xs}|\ \mathsf{n})\ \mathsf{arr}\ (=\mathsf{p})\ (\mathsf{qsort}\ \mathsf{xs})$

Combination with correctness theorem of *qsort* yields

$$
\begin{array}{l}
\{\mathsf{arr}\ \mathsf{xs}\ \mathsf{p} * \mathsf{idx}\ |\mathsf{xs}|\ \mathsf{n}\} \\
\quad \mathsf{qsort}_\dagger\ \mathsf{p}\ \mathsf{n} \\
\{\lambda \mathsf{r}.\ \exists \mathsf{xs}'.\ \mathsf{r}{=}\mathsf{p} * \mathsf{arr}\ \mathsf{xs}'\ \mathsf{p} * \mathsf{sorted}\ \mathsf{xs}' * \mathsf{mset}\ \mathsf{xs}' = \mathsf{mset}\ \mathsf{xs}\}
\end{array}
$$

Correctness theorem and TCB

Sepref generates $\mathit{qsort}_\dagger$ and theorem

$\mathsf{hnr}\ (\mathsf{arr}\ \mathsf{xs}\ \mathsf{p} * \mathsf{idx}\ |\mathsf{xs}|\ \mathsf{n})\ (\mathsf{qsort}_\dagger\ \mathsf{p}\ \mathsf{n})\ (\mathsf{idx}\ |\mathsf{xs}|\ \mathsf{n})\ \mathsf{arr}\ (=\mathsf{p})\ (\mathsf{qsort}\ \mathsf{xs})$

Combination with correctness theorem of *qsort* yields

$\{\mathsf{arr}\ \mathsf{xs}\ \mathsf{p} * \mathsf{idx}\ |\mathsf{xs}|\ \mathsf{n}\}$
$\quad \mathsf{qsort}_\dagger\ \mathsf{p}\ \mathsf{n}$
$\{\lambda \mathsf{r}.\ \exists \mathsf{xs}'.\ \mathsf{r}{=}\mathsf{p} * \mathsf{arr}\ \mathsf{xs}'\ \mathsf{p} * \mathsf{sorted}\ \mathsf{xs}' * \mathsf{mset}\ \mathsf{xs}' = \mathsf{mset}\ \mathsf{xs}\}$

Code generator generates LLVM text from $\mathit{qsort}_\dagger$.

export_llvm $\mathsf{qsort}_\dagger$ **is** uint64* qsort_uint64(uint64*, size_t)
(and, similar but more complicated for strings, ...)

Correctness theorem and TCB

Sepref generates $qsort_\dagger$ and theorem

$\mathsf{hnr}\ (\mathsf{arr}\ \mathsf{xs}\ \mathsf{p} * \mathsf{idx}\ |\mathsf{xs}|\ \mathsf{n})\ (\mathsf{qsort}_\dagger\ \mathsf{p}\ \mathsf{n})\ (\mathsf{idx}\ |\mathsf{xs}|\ \mathsf{n})\ \mathsf{arr}\ (=\mathsf{p})\ (\mathsf{qsort}\ \mathsf{xs})$

Combination with correctness theorem of *qsort* yields

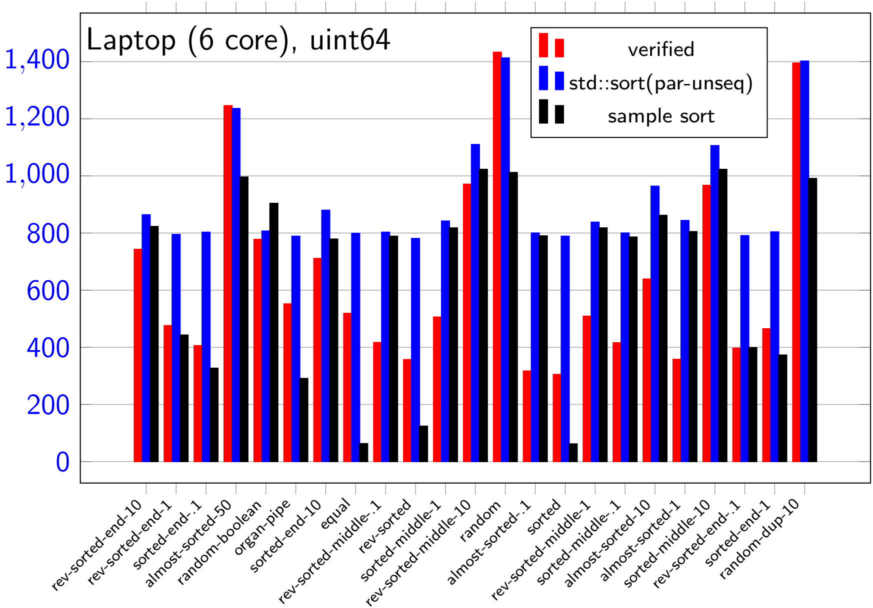
$$\{\mathsf{arr}\ \mathsf{xs}\ \mathsf{p} * \mathsf{idx}\ |\mathsf{xs}|\ \mathsf{n}\}$$
$$\quad \mathsf{qsort}_\dagger\ \mathsf{p}\ \mathsf{n}$$
$$\{\lambda \mathsf{r}.\ \exists \mathsf{xs}'.\ \mathsf{r}{=}\mathsf{p} * \mathsf{arr}\ \mathsf{xs}'\ \mathsf{p} * \mathsf{sorted}\ \mathsf{xs}' * \mathsf{mset}\ \mathsf{xs}' = \mathsf{mset}\ \mathsf{xs}\}$$

Code generator generates LLVM text from $qsort_\dagger$.

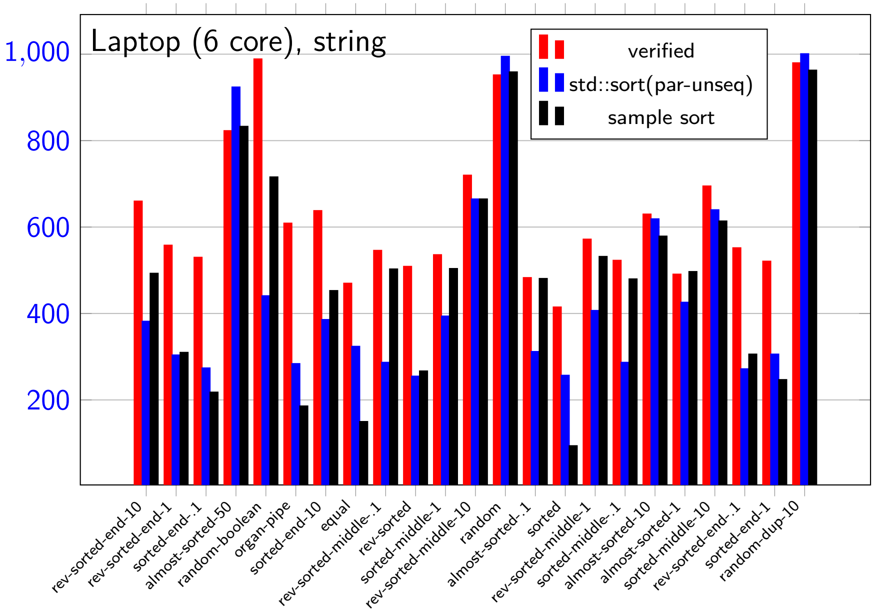
export_llvm $\mathsf{qsort}_\dagger$ **is** uint64* qsort_uint64(uint64*, size_t)
(and, similar but more complicated for strings, ...)

This can be compiled and linked against, e.g., benchmark suite

Benchmarks

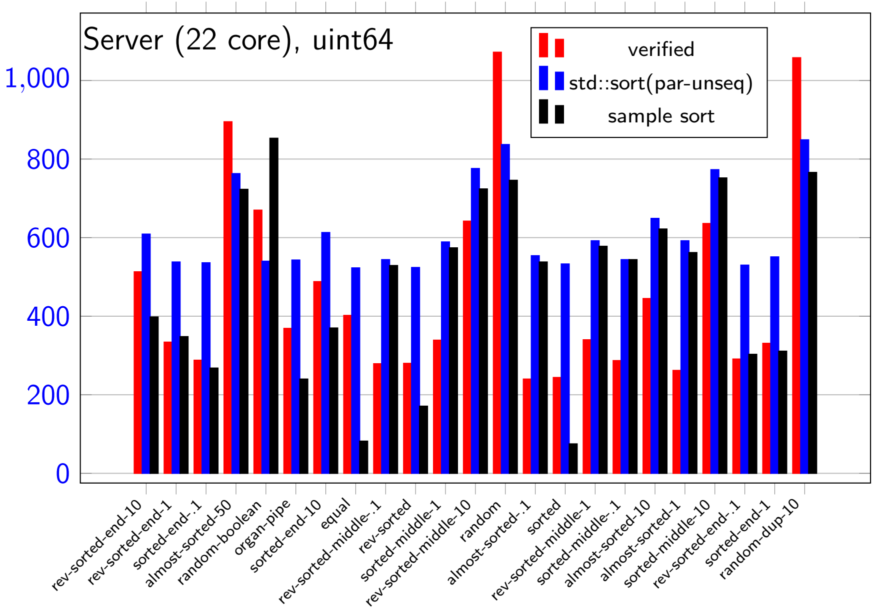

Laptop (6 core), uint64
verified
std::sort(par-unseq)
sample sort
1,400
1,200
1,000
800
600
400
200
0
rev-sorted-end-10
rev-sorted-end-1
sorted-end-.1
almost-sorted-50
random-boolean
organ-pipe
sorted-end-10
equal
rev-sorted-middle-.1
rev-sorted
sorted-middle-1
rev-sorted-middle-10
random
almost-sorted-.1
sorted
rev-sorted-middle-1
sorted-middle-.1
almost-sorted-10
almost-sorted-1
sorted-middle-10
rev-sorted-end-.1
sorted-end-1
random-dup-10

Benchmarks

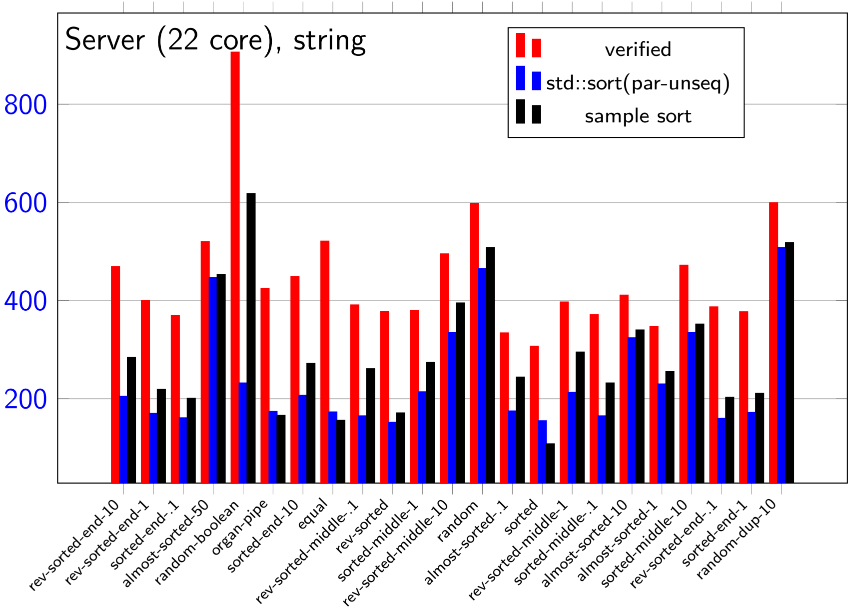


Laptop (6 core), string
verified
std::sort(par-unseq)
sample sort
1,000
800
600
400
200
rev-sorted-end-10
rev-sorted-end-1
sorted-end-.1
almost-sorted-50
random-boolean
organ-pipe
sorted-end-10
equal
rev-sorted-middle-.1
rev-sorted
sorted-middle-1
rev-sorted-middle-10
random
almost-sorted-.1
sorted
rev-sorted-middle-1
sorted-middle-.1
almost-sorted-10
almost-sorted-1
sorted-middle-10
rev-sorted-end-.1
sorted-end-1
random-dup-10

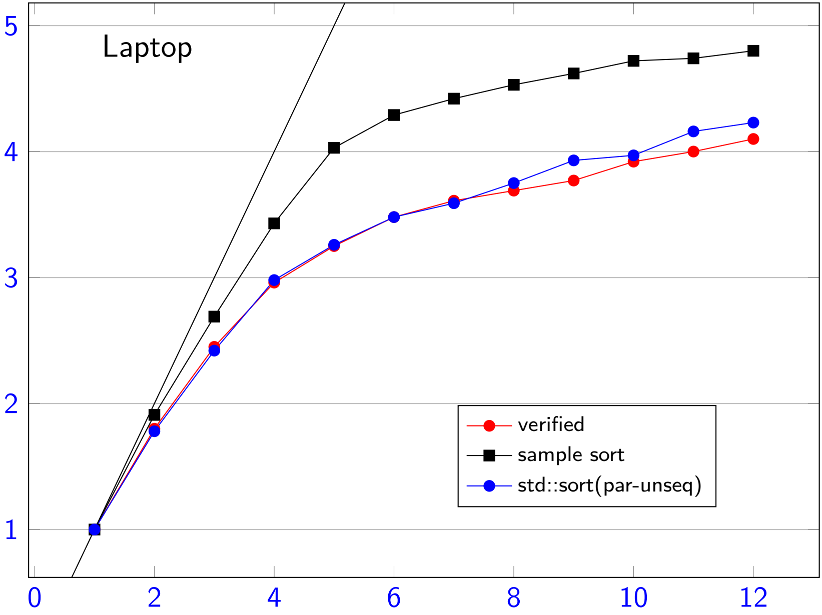
Benchmarks


Server (22 core), uint64
verified
std::sort(par-unseq)
sample sort
0
200
400
600
800
1,000
rev-sorted-end-10
rev-sorted-end-1
sorted-end-.1
almost-sorted-50
random-boolean
organ-pipe
sorted-end-10
equal
rev-sorted-middle-.1
rev-sorted
sorted-middle-1
rev-sorted-middle-10
random
almost-sorted-.1
sorted
rev-sorted-middle-1
sorted-middle-.1
almost-sorted-10
almost-sorted-1
sorted-middle-10
rev-sorted-end-.1
sorted-end-1
random-dup-10

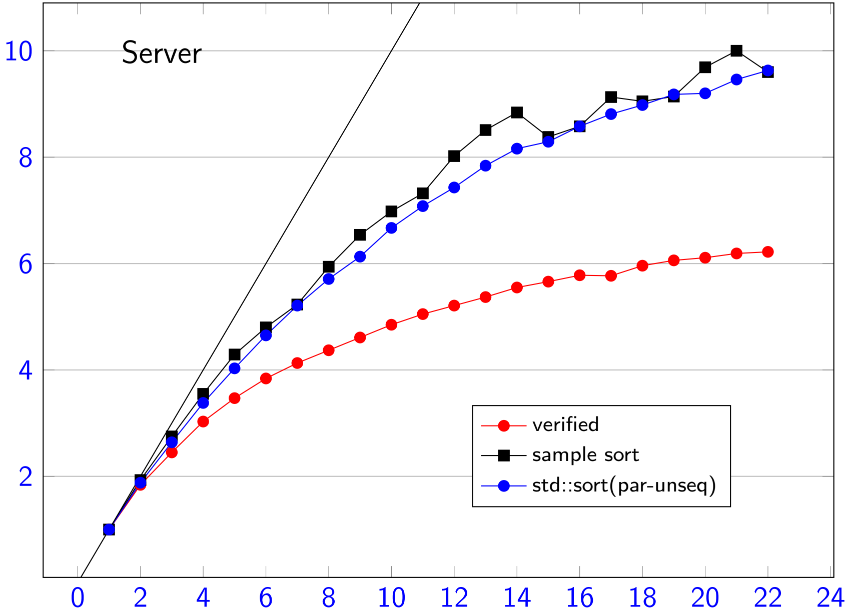
Benchmarks


Server (22 core), string
verified
std::sort(par-unseq)
sample sort
800
600
400
200
rev-sorted-end-10
rev-sorted-end-1
sorted-end-.1
almost-sorted-50
random-boolean
organ-pipe
sorted-end-10
equal
rev-sorted-middle-.1
rev-sorted
sorted-middle-1
rev-sorted-middle-10
random
almost-sorted-.1
sorted
rev-sorted-middle-1
sorted-middle-.1
almost-sorted-10
almost-sorted-1
sorted-middle-10
rev-sorted-end-.1
sorted-end-1
random-dup-10

Speedup


Laptop
verified
sample sort
std::sort(par-unseq)
0
2
4
6
8
10
12
1
2
3
4
5

Speedup



Server
10
8
6
4
2
0
2
4
6
8
10
12
14
16
18
20
22
24
verified
sample sort
std::sort(par-unseq)

Benchmark Interpretation

- our algorithm is competitive for integers
- still some problems for strings
- could scale better to larger number of cores

Conclusion

- Verification of parallel programs
  - stepwise refinement to tackle complexity
  - down to LLVM, small TCB
  - fast verified programs
- Idea: shallow embedding, using access reports
  - backwards compatible with sequential IRF
- Future work
  - state-of-the-art parallel sorting
  - fractional separation logic (for shared read-only)
  - more concurrency (synchronization, atomic, ...)
  - complexity of parallel algorithms
  - GP-GPUs

https://www21.in.tum.de/~lammich/isabelle_llvm_par/
https://github.com/lammich/isabelle_llvm/tree/2021-1