Northwestern

Rethinking Dependences for Compiler Parallelization

Federico Sossai | Simone Campanoni

Compilers still fail to extract the **parallelism** expert developers routinely exploit, but **why**?

1 Compilers analyze dependences between **low-level reads/writes**, missing the higher-level transformations that expose parallelism.

2 Dependence analysis **precision is not enough**: dependences must be transformed, not merely detected.

3 Parallelization often rely on algorithmic **properties impossible to recover** via static analysis, e.g., stale reads, order-insensitive updates, benign races..

What **programmers** write:

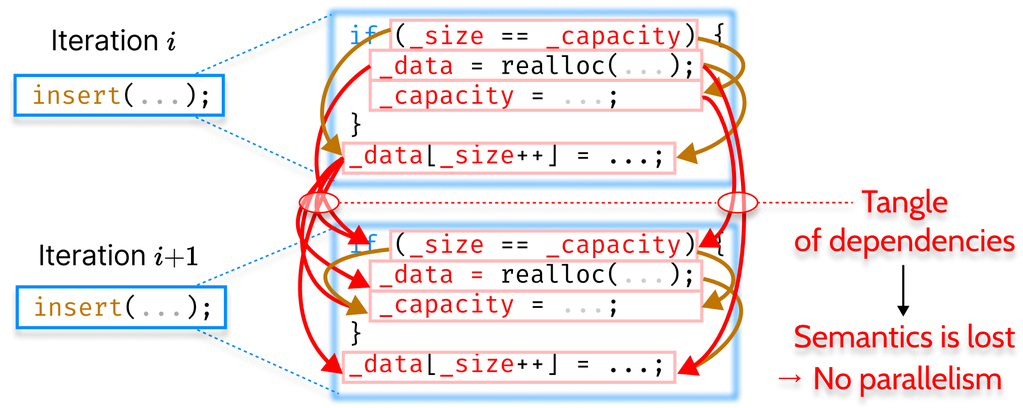
```
bag<int> bag;

for (int i = 0; i < N; i++) {
  ...
  insert(&bag, ...);
}
```

High-level operation

Well-know semantics

Programmers know how to parallelize insertions

What **compilers** see:



Because LLVM IR cannot represent the higher-level semantics required for parallelization, we introduce **two new concepts.**

E-function:

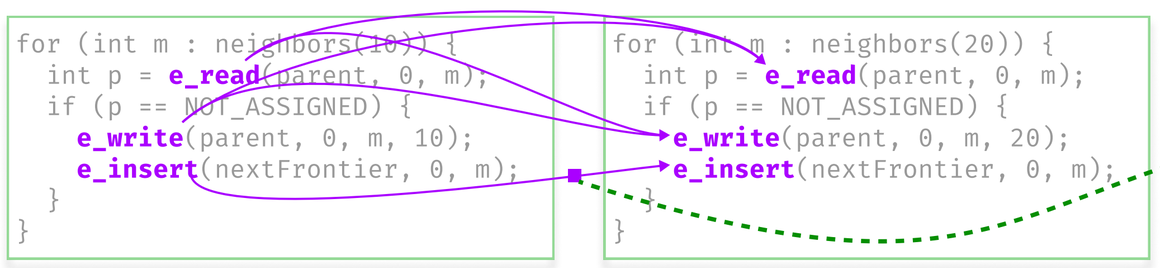
Function representing a **high-level operation** on a single object.

Clause:

Developer-provided rule stating that two E-functions can be **made independent,** together with the **transformation** needed to do so.

The Dependence Termination Graph

Extends traditional data dependence graph with terminable dependences.



Thanks to Clauses, terminable dependences carry the transformation that the compiler can use to eliminate it.

Expressing Opportunities

We designed a `#pragma` annotation **interface** to express E-functions and Clauses, e.g.:

```
#pragma clause(insert,insert) \
        pre(expand) comm
```

"Two calls to `insert` are commutative and can run in parallel once the function `expand` has been called."

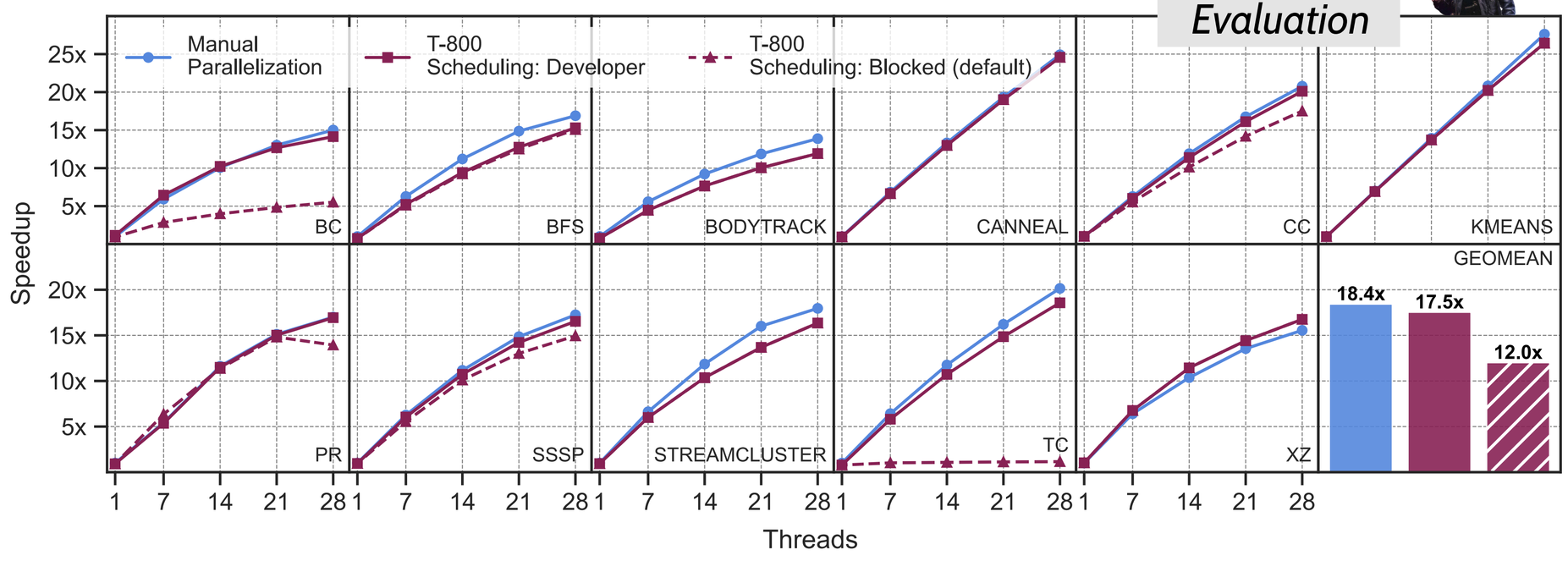
and implemented high-level **reusable** collections

We generalized reductions beyond scalars and arrays!

..but our abstraction remains **collection-agnostic**!

	GAPBS						PARSEC			ROD.	SPEC
	BC	BF	CC	SP	TC	PR	BT	SC	CN	KM	XZ
Scalar	✓	✓			✓	✓	✓	✓	✓	✓	
StaleArray	✓	✓	✓	✓							
Bag/Sequence	✓	✓					✓				✓
Array	✓							✓		✓	
BitSet	✓	✓									
LocalState							✓		✓		
Multimap				✓							
RandNumGen							✓		✓		

Evaluation

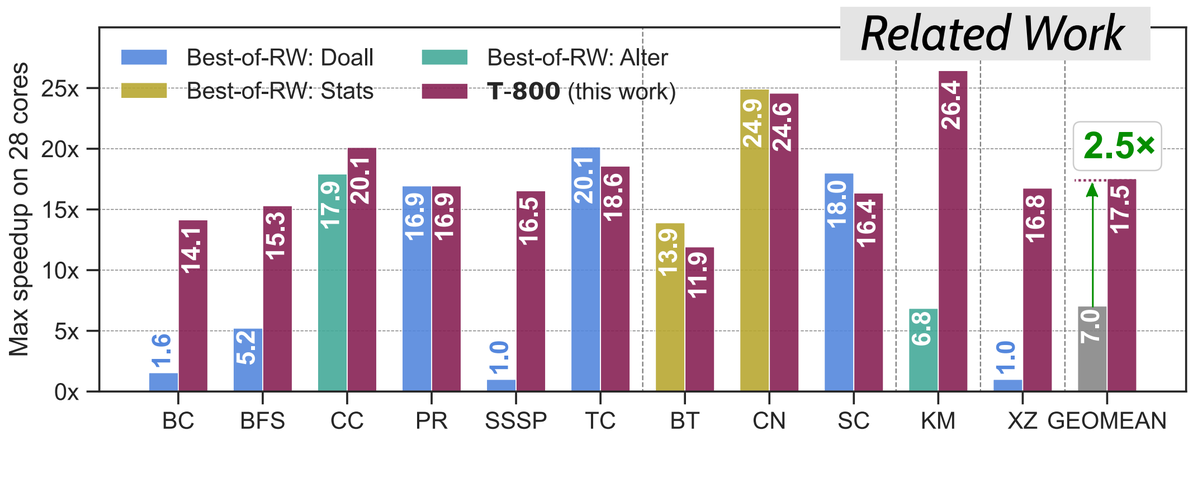


We implemented our compiler T-⊟⊘⊘ on LLVM 14 using NOELLE for memory analysis.

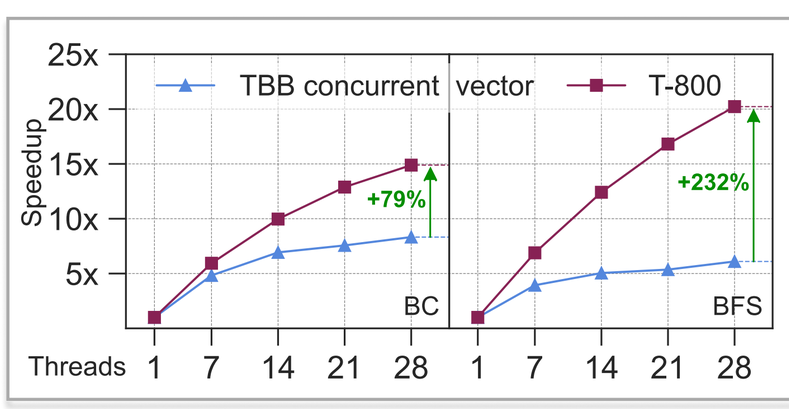
On 28 cores, binaries produced by T-⊟⊘⊘ reach **95%** of the speedup achievable via manual parallelization.

Our abstraction allows users to write **sequential** code that T-⊟⊘⊘ can efficiently parallelize.

Related Work



One might ask...



Are layout transformations really necessary? Why not concurrent collections?

Contention kills scaling!