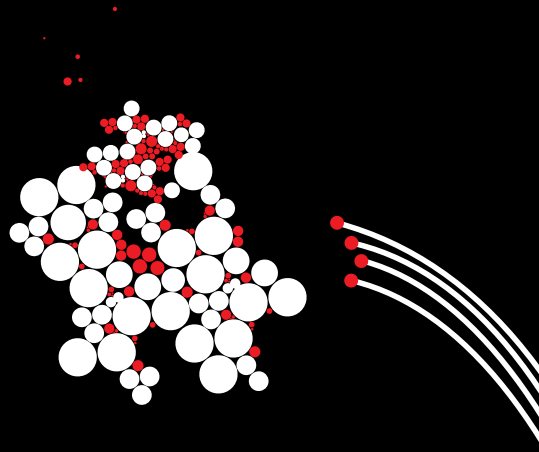
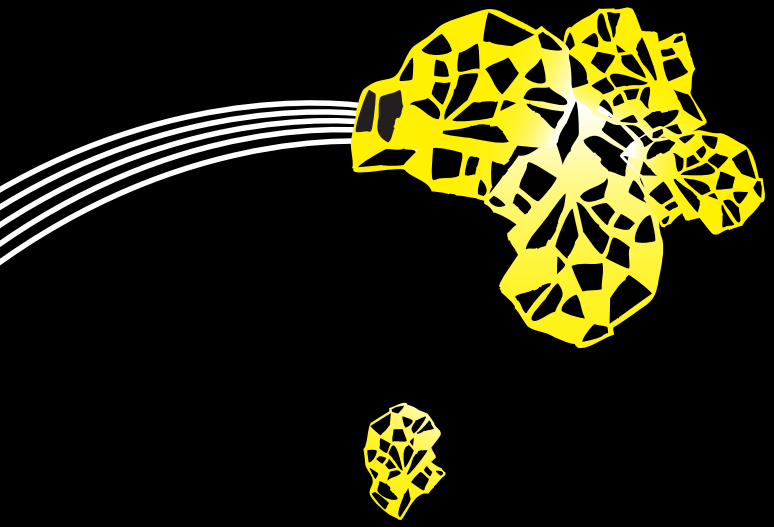
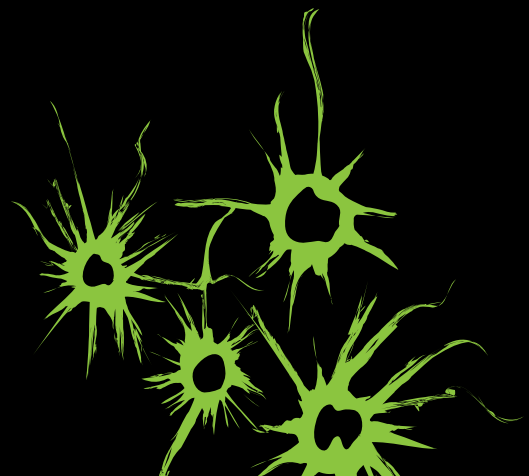
UNIVERSITY OF TWENTE.



VeyMont: Parallelising Verified Programs instead of Verifying Parallel Programs

Petra van den Bos and Sung-Shik Jongmans





Deductive verification

Deductive verification

- Hoare logic (pen-and-paper proof):

$$\{\mathtt{x} = v_1 \wedge \mathtt{y} = v_2\}$$
$$\mathtt{z} := \mathtt{x}\ ;\ \mathtt{x} := \mathtt{y}\ ;\ \mathtt{y} := \mathtt{z}$$
$$\{\mathtt{x} = v_2 \wedge \mathtt{y} = v_1\}$$

Deductive verification

- Hoare logic (pen-and-paper proof):

$$\{\mathtt{x} = v_1 \wedge \mathtt{y} = v_2\}$$
$$\mathtt{z} := \mathtt{x}\,;\ \mathtt{x} := \mathtt{y}\,;\ \mathtt{y} := \mathtt{z}$$
$$\{\mathtt{x} = v_2 \wedge \mathtt{y} = v_1\}$$

- Program for deductive verification tool VerCors:

```
1 class SeqProgram {
2   int v1, v2, x, y, z;
3
4   context   ...
5   requires  x == v1 && y == v2;
6   ensures   x == v2 && y == v1;
7   void swap() {
8     z = x; x = y; y = z;
9 } }
```

Deductive verification of a parallel program

```
1  class Channel {
2    int s, buf; // state, buffer
3
4    resource lock_invariant() =
5      Perm(s, 1) **
6      (s == 1 || s == 2 || s == 3) **
7      (s == 1 ==> Perm(buf, 1\2)) **
8      (s == 2 ==> Perm(buf, 1\2));
9
10   context Perm(buf, 1\2);
11   ensures buf == v;
12   void writeValue(int v) {
13     lock this;
14     loop_invariant ...;
15     while (s != 1) { wait this; }
16     s = 2; buf = v;
17     unlock this;
18   }
19
20   ensures Perm(buf, 1\2);
21   ensures \result == buf;
22   int readValue() { ... }
23 }
24
25 class Thread {
26   Channel a, b; int v, v_old;
27   context  Perm(v, 1);
28   context  Perm(v_old, 1\2);
29   context  Perm(a, 1\2);
30   context  Perm(b, 1\2);
31   context  Perm(b.buf, 1\2);
32   requires v == v_old;
33   requires a != null;
34   requires b != null;
35   ensures  Perm(a.buf, 1\2);
36   ensures  v_old == b.buf
37   ensures  a.buf == v;
38   void run() {
39     b.writeValue(v);
40     v = a.readValue();
41 } }
42
43 class ParProgram {
44   int v1, v2; Thread t1, t2;
45
46   context  ...
47   requires t1.v == v1 && t2.v == v2;
48   ensures  t1.v == v2 && t2.v == v1;
49   void swap() {
50     fork t1; fork t2;
51     join t1; join t2;
52 } }
```

Applying deductive verification

- Sequential programs: relatively simple
  - i.e., relatively little effort to annotate

Applying deductive verification

- Sequential programs: relatively simple
  - i.e., relatively little effort to annotate
- Parallel programs: surprisingly hard
  - More code
  - More annotations

Deductive verification of parallel programs

- Standard approach:
  1. Parallelise a sequential program (or write from scratch).
  2. Verify it.

Deductive verification of parallel programs

- Standard approach:
  1. Parallelise a sequential program (or write from scratch).
  2. Verify it.
- Our approach (inverted workflow):
  1. Verify a sequential-ish program.
  2. Parallelise it.

Deductive verification of parallel programs

- Standard approach:
  1. Parallelise a sequential program (or write from scratch).
  2. Verify it.
- Our approach (inverted workflow):
  1. Verify a sequential-ish program.
  2. Parallelise it.
- Our tool: VeyMont

Why the name “VeyMont”?

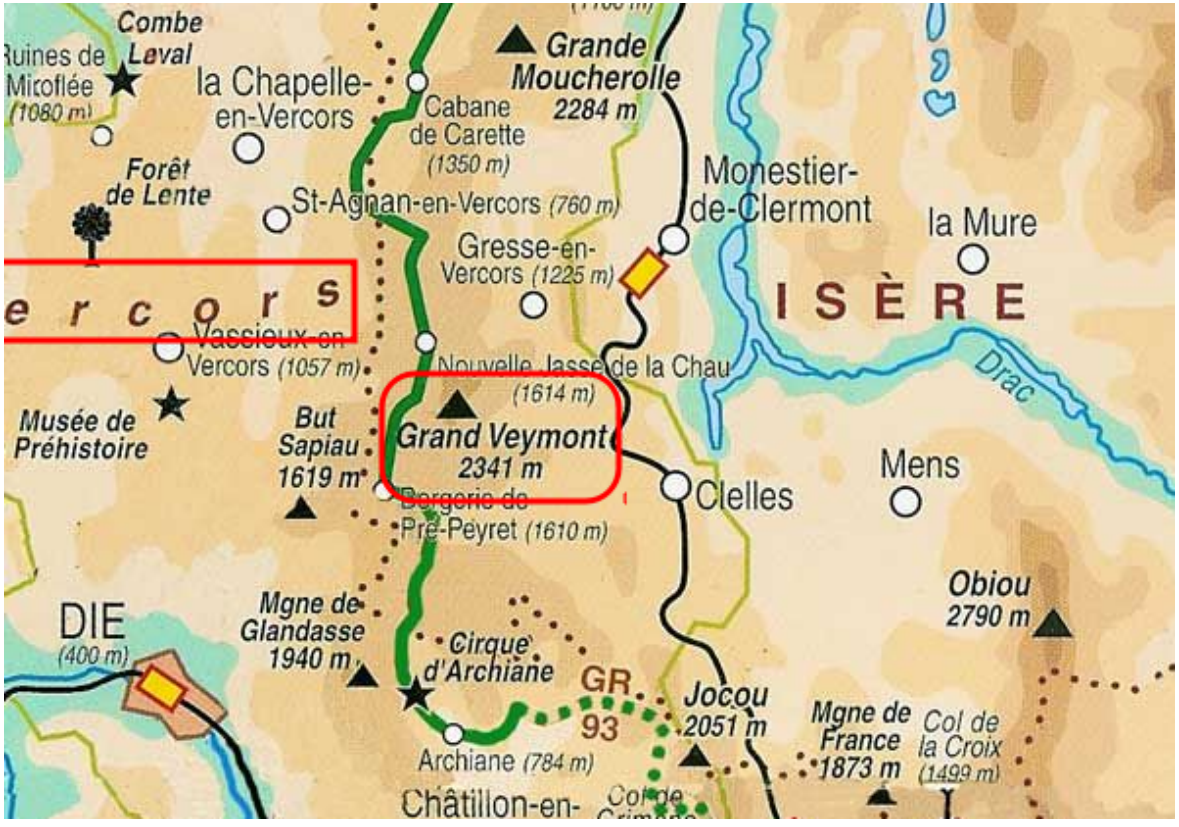
- VeyMont uses VerCors as back-end for deductive verification

Why the name “VeyMont”?

- VeyMont uses VerCors as back-end for deductive verification
- VeyMont is build on top of VerCors

Why the name “VeyMont”?

- VeyMont uses VerCors as back-end for deductive verification
- VeyMont is build on top of VerCors



Scope of research

- This research is not about:
  - fine-grained parallelism
    - e.g. loop parallelisation

Scope of research

- This research is not about:
  - fine-grained parallelism
    - e.g. loop parallelisation
- It is about:
  - coarse-grained parallelism among threads

Scope of research

- This research is not about:
  - fine-grained parallelism
    - e.g. loop parallelisation
- It is about:
  - coarse-grained parallelism among threads
    - without shared memory

Scope of research

- This research is not about:
  - fine-grained parallelism
    - e.g. loop parallelisation
- It is about:
  - coarse-grained parallelism among threads
    - without shared memory
    - with message passing

VeyMont: what the tool does

- Input: sequential-ish program with annotations

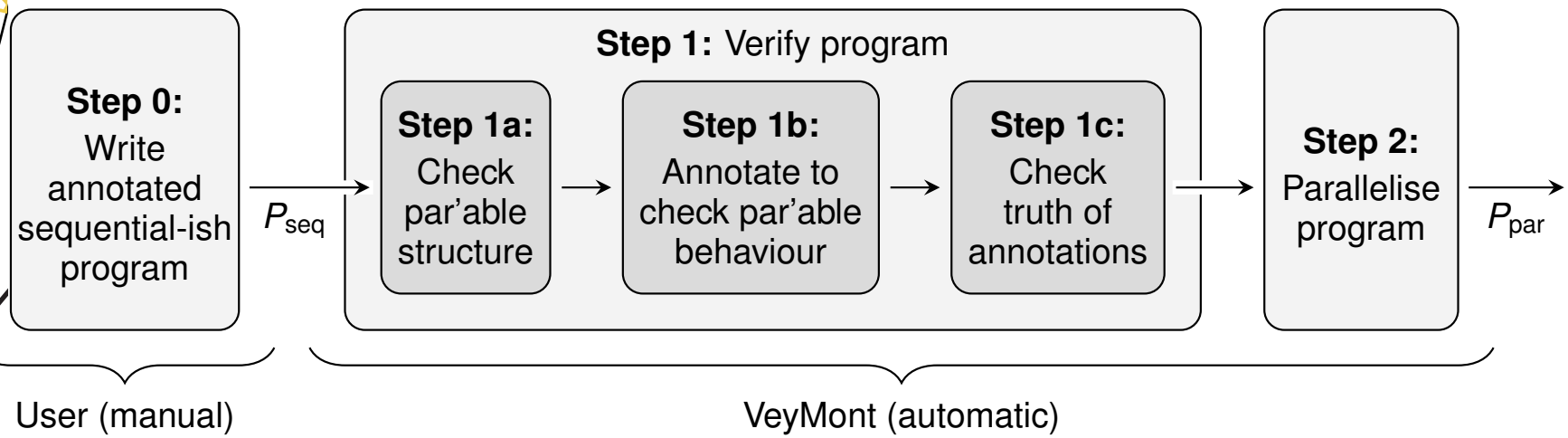
VeyMont: what the tool does

- Input: sequential-ish program with annotations
- Output: parallel program in Java
  - Executable program
  - Functionally correct w.r.t given annotations
  - Deadlock-free by construction

VeyMont & theory

- VeyMont theory is inspired by choreographic programming
  - Standard: deadlock-freedom-by-construction
  - Our extension: functional correctness
- ESOP paper: A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming

VeyMont: internal workflow



Verifying swap with VeyMont

Sequential-ish program:

```
1  class Storage {
2    int v, temp;
3    Storage(int v) {
4      this.v = v;
5  } }
6
```

```
7  class SeqProgram {
8    Storage s1, s2;
9    SeqProgram(int x, int y) {
10     s1 = new Storage(x);
11     s2 = new Storage(y);
12   }
13   requires true;
14   ensures s1.v == \old(s2.v);
15   ensures s2.v == \old(s1.v);
16   void run() {
17     s1.temp = s1.v;
18     s2.temp = s2.v;
19     s1.v = s2.temp;
20     s2.v = s1.temp;
21 } }
```

Verifying swap with VeyMont

Parallel program:

```
1  class s1Thread extends Thread {
2    Storage s1;
3    IntegerChannel s1_s2;
4    IntegerChannel s2_s1;
5
6    s1Thread(int x,
7      IntegerChannel s1_s2,
8      IntegerChannel s2_s1) {
9
10     s1 = new Storage(x);
11     s1_s2 = s1_s2;
12     s2_s1 = s2_s1;
13   }
14
15   public void run() {
16     s1.temp = s1.v;
17     s1.v = s2_s1.read();
18     s1_s2.write(s1.temp);
19 } }
```

```
20 class s2Thread extends Thread {
21
22   ... // similar to lines 2-13
23
24   public void run() {
25     s2.temp = s2.v;
26     s2_s1.write(s2.temp);
27     s2.v = s1_s2.read();
28 } }
29
30 class ParProgram {
31   ParProgram(int x, int y) {
32     IntegerChannel s1_s2 = ...;
33     IntegerChannel s2_s1 = ...;
34     new s1Thread(
35       x, s1_s2, s2_s1).start();
36     new s2Thread(
37       y, s1_s2, s2_s1).start();
38 } }
```

Implementation of parallelisation (step 2)

	sequential-ish	→	**parallel**: `s1Thread`	**parallel**: `s2Thread`
store	`s1.x = 5;`	→	`s1.x = 5;`	`/* skip */`
transfer	`s2.y = s1.x + 4;`	→	`s1_s2.write(s1.x + 4);`	`s2.y = s1_s2.read();`
if/while	`if (s1.x == 5 &&`	→	`if (s1.x == 5`	`if (/* skip */`
	`s2.y == 9) {`	→	`/* skip */) {`	`s2.y == 9) {`
call	`s1.foo(s1.x); }`	→	`s1.foo(s1.x); }`	`/* skip */ }`

Evaluation

1. Applicability: VeyMont is:
   - applicable on various distributed algorithms, and
   - significantly reduces the annotation burden

Evaluation

1. Applicability: VeyMont is:
   - applicable on various distributed algorithms, and
   - significantly reduces the annotation burden
2. Efficiency: VeyMont produces parallel programs that are:
   - competitive with existing third-party parallel programs.

Applicability case studies

Three classical distributed algorithms:

- Two-phase commit “2PC”:
  - One client and N-1 servers fulfil joint query in distributed database
- Anonymous election:
  - N threads probabilistically elect a unique leader among them
- Consensus:
  - N threads agree on a common value via a majority vote

Applicability case studies: results

	$\frac{\text{spec}}{\text{impl}}$	1c	total
2PC (n=3)	32/75	6.8	8.0
2PC (n=5)	42/91	8.9	10.1
2PC (n=8)	57/115	13.9	15.2
2PC (n=12)	77/147	26.4	27.8
2PC (n=17)	102/187	62.2	63.9

	$\frac{\text{spec}}{\text{impl}}$	1c	total
election (n=3)	22/49	7.2	8.5
election (n=5)	28/69	14.3	16.2
election (n=8)	37/99	61.5	65.4
consensus (n=3)	72/73	9.8	11.3
consensus (n=5)	90/104	41.0	42.5

Applicability case studies: results

	$\frac{\text{spec}}{\text{impl}}$	1c	total
2PC (n=3)	32/75	6.8	8.0
2PC (n=5)	42/91	8.9	10.1
2PC (n=8)	57/115	13.9	15.2
2PC (n=12)	77/147	26.4	27.8
2PC (n=17)	102/187	62.2	63.9

	$\frac{\text{spec}}{\text{impl}}$	1c	total
election (n=3)	22/49	7.2	8.5
election (n=5)	28/69	14.3	16.2
election (n=8)	37/99	61.5	65.4
consensus (n=3)	72/73	9.8	11.3
consensus (n=5)	90/104	41.0	42.5

Wolf et al. [52]: spec/impl of 2.69–3.16 with traditional methods

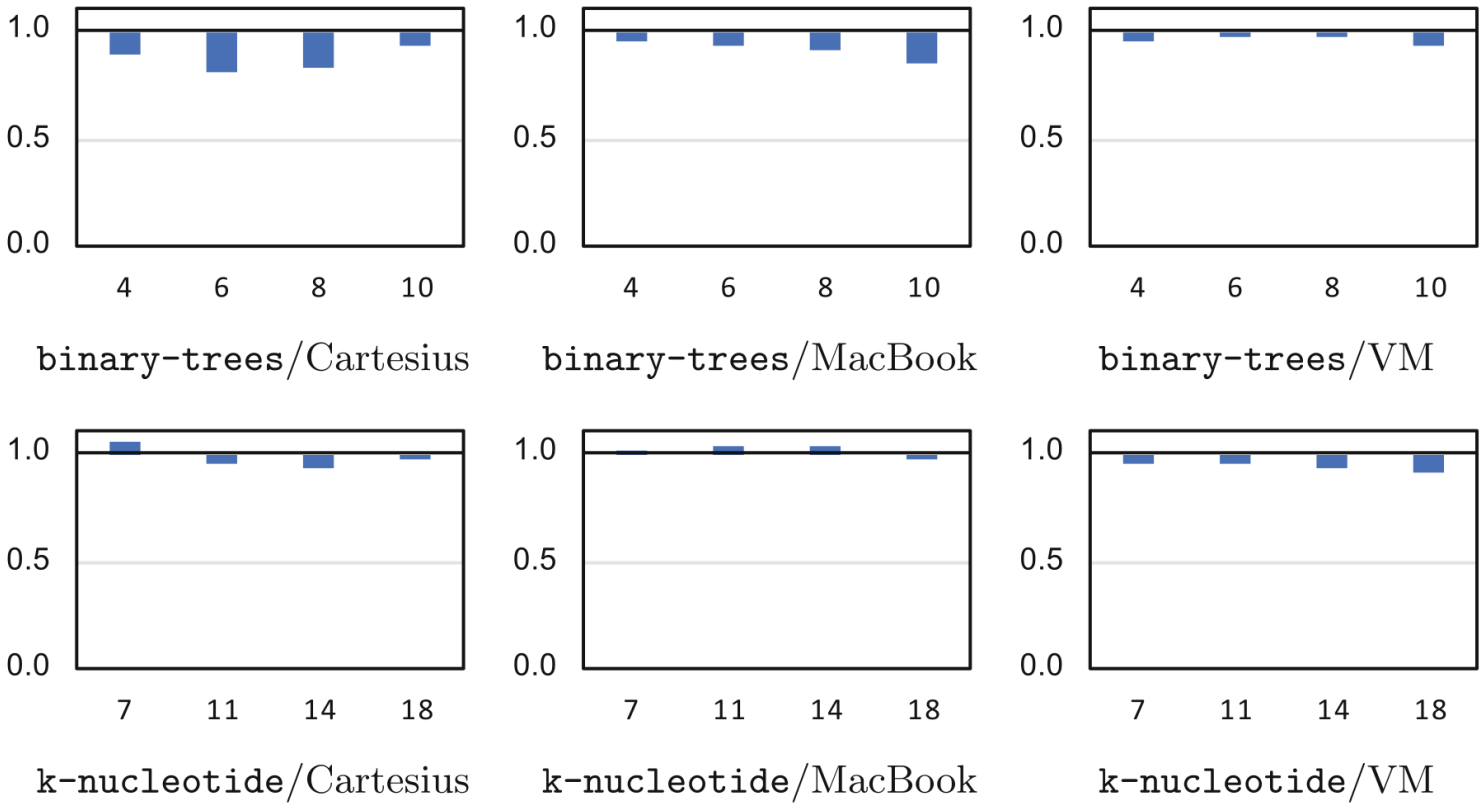
Efficiency case studies

Two third-party parallel programs from CLBG database [1]:

- binary-trees:
  - Parallel tree walk
- k-nucleotide
  - Parallel pattern matching of molecule sequences vs. a DNA string

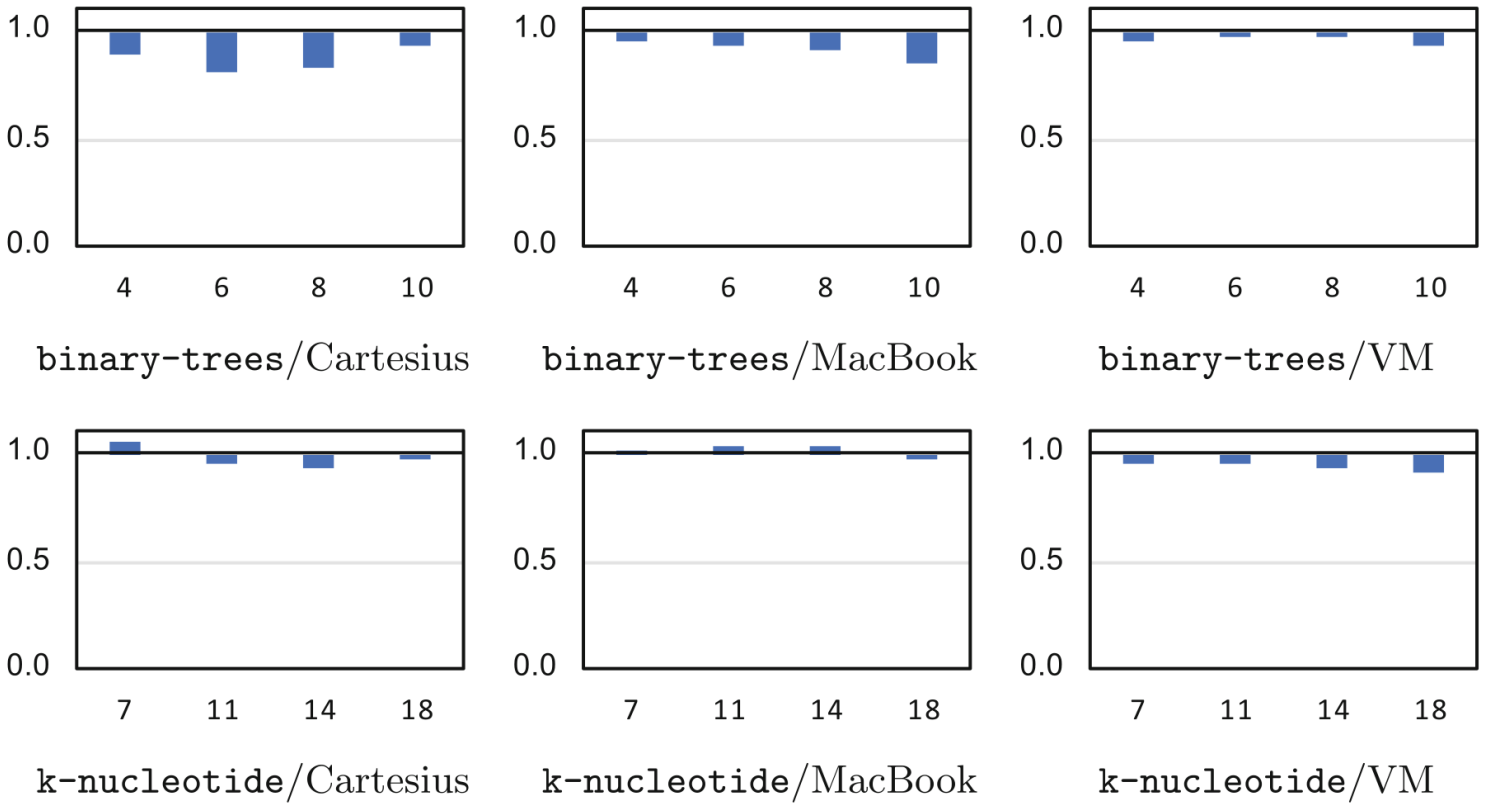
Efficiency case studies: results

Below graphs show mean run time ratio $\frac{\mu_{\text{CLBG}}}{\mu_{\text{VeyMont}}}$



Efficiency case studies: results

Below graphs show mean run time ratio $\frac{\mu_{\text{CLBG}}}{\mu_{\text{VeyMont}}}$



Slow-down is generally less than 10%

Conclusions

With VeyMont:

- verifying functional correctness & deadlock freedom of parallel programs
- is as easy as verifying sequential programs

Conclusions

With VeyMont:

- verifying functional correctness & deadlock freedom of parallel programs
- is as easy as verifying sequential programs

Future work:

- Parametrisation: programs for N threads

Conclusions

With VeyMont:

- verifying functional correctness & deadlock freedom of parallel programs
- is as easy as verifying sequential programs

Future work:

- Parametrisation: programs for N threads
- Relaxation of restrictions (checked in step 1a, 1b), to e.g.:
  - Relax conditions for if statements and while loops
  - Support read-only shared arrays

Conclusions

With VeyMont:

- verifying functional correctness & deadlock freedom of parallel programs
- is as easy as verifying sequential programs

Future work:

- Parametrisation: programs for N threads
- Relaxation of restrictions (checked in step 1a, 1b), to e.g.:
  - Relax conditions for if statements and while loops
  - Support read-only shared arrays
- Combine with other parallelisation techniques

Conclusions

With VeyMont:

- verifying functional correctness & deadlock freedom of parallel programs
- is as easy as verifying sequential programs

Future work:

- Parametrisation: programs for N threads
- Relaxation of restrictions (checked in step 1a, 1b), to e.g.:
  - Relax conditions for if statements and while loops
  - Support read-only shared arrays
- Combine with other parallelisation techniques
- Engineering: use other back-end than VerCors

Thank you for listening

Read the paper?

Syntactical restrictions of step 1a

$$
\begin{array}{rcl}
P & ::= & \tilde{C} \\
C & ::= & \texttt{class}\ n\ \{\ \tilde{F}\ \tilde{M}\ \} \\
F & ::= & f \\
M & ::= & \tilde{A}\ m(\tilde{x})\ \{\ \tilde{S}\ \} \mid \\
 & & \tilde{A}\ m(\tilde{x}) \\
A & ::= & \texttt{requires}\ B; \mid \\
 & & \texttt{ensures}\ B; \mid \\
 & & \texttt{context}\ B;
\end{array}
\qquad
\begin{array}{rcl}
S & ::= & \texttt{assert}\ B; \mid X = E; \mid E.m(\tilde{E}); \mid \\
 & & \texttt{if}\ (B)\ \{\ \tilde{S}_1\ \}\ \texttt{else}\ \{\ \tilde{S}_2\ \} \mid \\
 & & \texttt{loop\_invariant}\ B_1; \texttt{while}\ (B_2)\ \{\ \tilde{S}\ \} \\
X & ::= & x \mid f \mid \texttt{this}.f \mid X.f \mid X[E] \\
E & ::= & X \mid \texttt{this} \mid \texttt{null} \mid B \mid \texttt{0} \mid \texttt{1} \mid E_1 + E_2 \mid \cdots \mid \\
 & & \texttt{new}\ n(\tilde{E}) \mid E.f \mid E.m(\tilde{E}) \mid \texttt{new}\ [E] \mid E_1[E_2] \\
B & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{!}B \mid B_1\ \texttt{\&\&}\ B_2 \mid B_1\ \texttt{==>}\ B_2 \mid \\
 & & (\texttt{\textbackslash forall}\ x = E_1 .. E_2;\ B) \mid E_1 == E_2 \mid \cdots
\end{array}
$$

- Format of `SeqProgram` (fields, constructor, run method, other methods)
- Assignments refer to at at most one field at each side of =
- If/while conditions are of format E_1 && ... && E_k for k threads
- Method calls only refer to one thread (e.g. `s1.foo(s1.x)`)

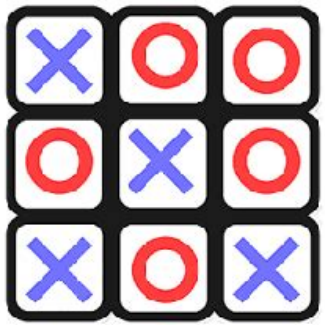
Semantical conditions of step 1b

- Alias-freedom: variables in sequential program cannot be aliased
  - Threads in parallel program operate on disjoint memory fragments
- Branch unanimity: equivalent conditions for each thread
  - If/while condition E_1 && ... && E_k requires $E_1 == E_2$ && ... && $E_k - 1 == E_k$

Features of VeyMont

- Turing complete
  - while-loops!
- Support for data structures, e.g.
  - multidimensional arrays
  - nesting of classes
- Trusted code: abstract methods with
  - a specification (precondition and postcondition)
  - but no implementation (method body)

Verifying Tic-Tac-Toe with VeyMont



Verifying Tic-Tac-Toe with VeyMont

```
1  class Player {
2    int m, n;
3    int[][] grid;
4    Move move;
5
6    ...
7
8    ensures (\forall int i = 0..m;
9               (\forall int j = 0..n;
10                grid[i][j] == 0));
11   ensures ...
12   Player(int m, int n, ...) {
13     this.grid = new int[m][n];
14     ...
15   }
16
17  requires ...
18   ensures 0 <= move.x && move.x < m;
19   ensures 0 <= move.y && move.y < n;
20   ensures grid[move.x][move.y] == 0;
21   ensures ...
22   void think();
23
24  requires ...
25  requires 0 <= move.x && move.x < m;
26  requires 0 <= move.y && move.y < n;
27  requires grid[move.x][move.y] == 0;
28   ensures ...
29   void play();
30 }
```

```
31 class Move {
32   int x, y, t;
33   ...
34 }
```

```
35 class SeqProgram {
36   Player p1, p2;
37
38   ...
39
40   ensures eq_grids(p1, p2);
41   SeqProgram(int m, int n) {
42     p1 = new Player(m, n, ...);
43     p2 = new Player(m, n, ...);
44   }
45
46   context eq_grids(p1, p2);
47   void turn1() {
48     p1.think();
49     p1.play();
50     p2.think(); // in the background
51     p2.move = p1.move.clone();
52     p2.play(); // to update
53   }
54
55   ...
56
57   context eq_grids(p1, p2);
58   void run() {
59     loop_invariant eq_grids(p1, p2);
60     while (p1.inPlay && p2.inPlay) {
61       turn1();
62       if (p1.inPlay && p2.inPlay) {
63         turn2();
64 } } } }
```