ICTAC 2018
15th International Colloquium on Theoretical Aspects of Computing
Stellenbosch, South Africa, Oct 19, 2018

Finding Rare Concurrent Programming Bugs

*An **Automatic**, **Symbolic**, **Randomized**, and **Parallelizable** Approach*

Gennaro Parlato
gennaro@ecs.soton.ac.uk

UNIVERSITY OF Southampton

Concurrent programs

Concurrency is everywhere in computing

- Embedded systems
- multi-core architectures
- worldwide networks

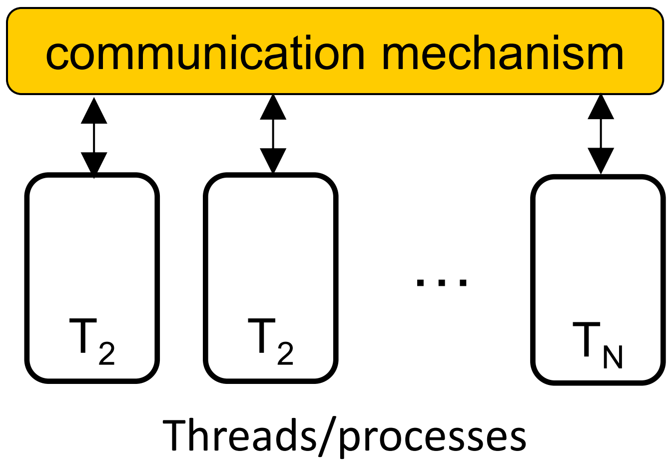
Large concurrent computing resources are available

- clusters
- cloud computing

There is a big demand for concurrent software

- enterprise customer services (e.g, telecom companies)
- government services (e.g., tax payment services)
- social networks, cloud services, …

Developing concurrent programs is difficult



Programmers have to guarantee

- correctness of sequential execution of each individual thread
- under nondeterministic interferences from other threads (**interleavings**)

Developing concurrent programs is difficult

What happens here...???

```
int n=0; //atomic shared variable

int P(void) {
   int tmp, i=1;
   while (i<=10) {
        tmp = n;
        n = tmp + 1;
        i++;
   }
}

int main (void)
   id1 = thread_create(P);
   id2 = thread_create(P);
   join( id1 );
   join( id2 );
   assert(n == 20);
}
```

Can the `assert` fail?

Developing concurrent programs is difficult

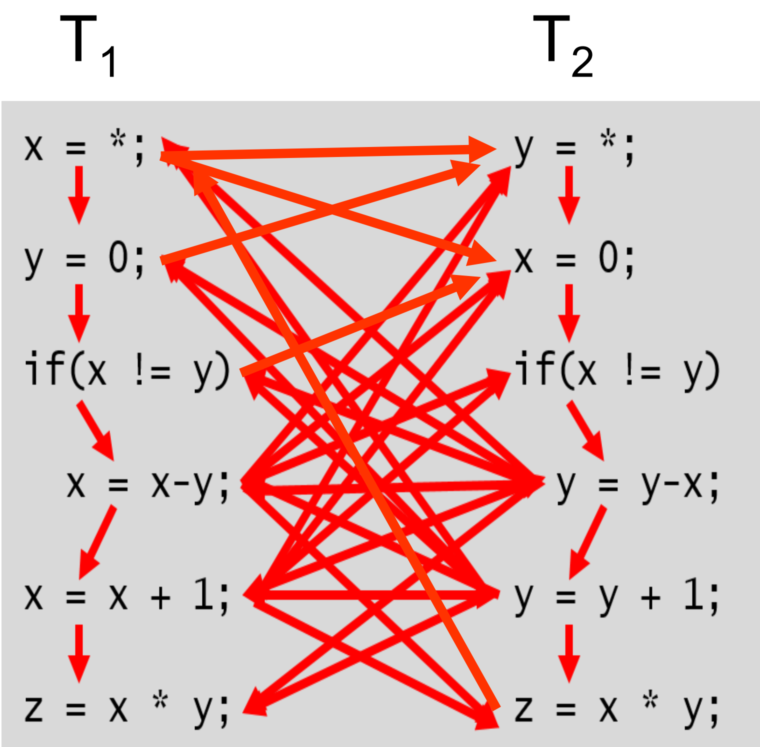
What happens here...???

```
int n=0; //atomic shared variable

int P(void) {
   int tmp, i=1;
   while (i<=10) {
        tmp = n;
        n = tmp + 1;
        i++;
   }
}

int main (void)
   id1 = thread_create(P);
   id2 = thread_create(P);
   join( id1 );
   join( id2 );
   assert(n > 2);
}
```

Scale of the challenge: #interleavings

2 threads with N LOC

#interleavings: $\binom{2N}{N}$



Scenario 1:

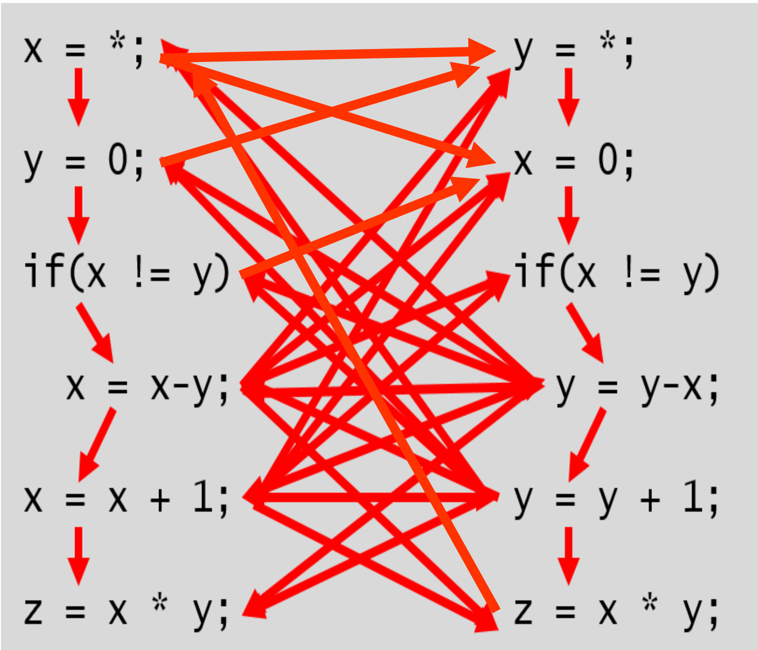
- N=40
- If 1 billion interleavings are simulated per second
  - ➢ 3.4 million years

Scenario 2:

- N=150
  - ➢ # interleavings > estimated # atoms in the known universe! >= 10^{80}

Bug-finding: finding needles in a haystack

Set of interleavings

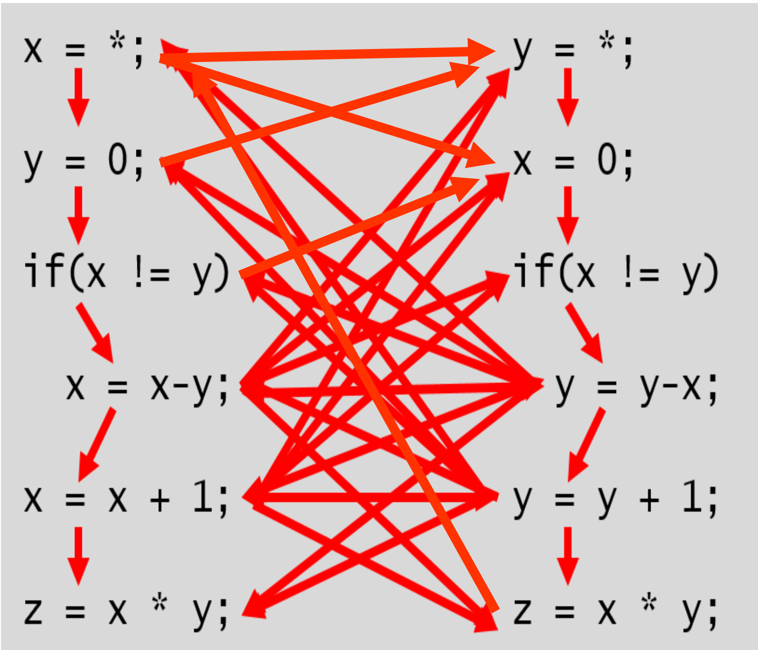


Haystack

Testing is easy when many interleavings are buggy

Bug-finding: finding A needle in a haystack

Set of interleavings



Haystack

… but is hard when buggy

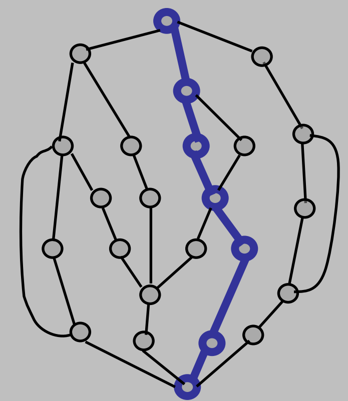
interleavings are rare

⇒ … needs to be ***complemented*** by automated ***analyses*** that handle ***interleavings symbolically***

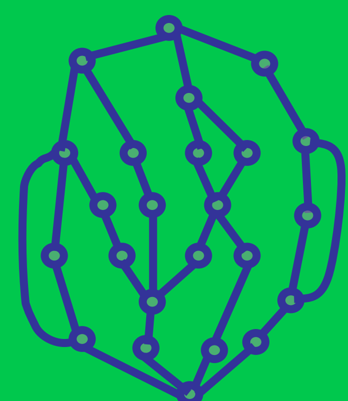
Bounded Model Checking (BMC) of concurrent programs

Testing vs Bounded Model Checking

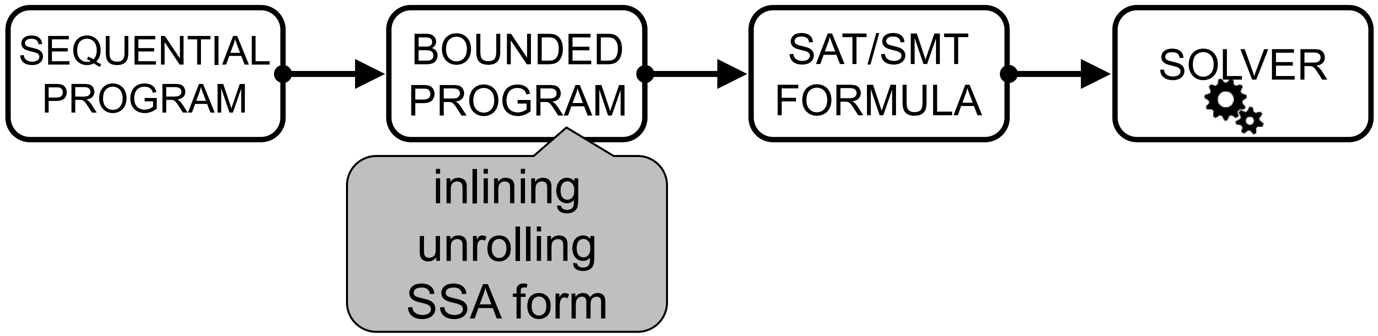
- Testing:
  - checks some executions
  - may miss errors
  - fast



- Bounded Model Checking (BMC)
  - Exhaustively explores all executions
    - ▹ bounding loop iterations
    - ▹ bounding context-switches, etc.
  - Can be extremely resource-hungry



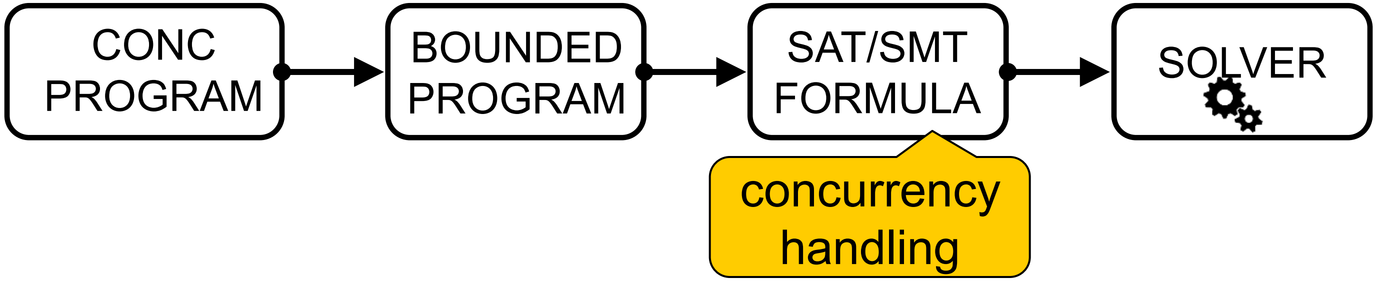
BMC for sequential C programs



tools

- BLITZ [Cho, D'Silva, Song – ASE'13]
- CBMC [Clarke, Kroening, Lerda – TACAS'04]
- LLBMC [Falke, Merz, Sinz – ASE'13]
- ESBMC [Cordeiro, Fischer, Marques-Silva – ASE'09]

BMC for concurrent C programs



SAT/SMT approach

- encode each thread as in the sequential case
- add a conjunct for shared memory operations
- all possible interleavings in the bounded program

$$\varphi_{threads} \land \varphi_{concurrency}$$

papers

- **[Sinha, Wang – POPL'11]**
- **[Alglave, Kroening, Tautschnig – CAV'13]** **CBMC**

Sequentialization targeting BMC

Sequentialization: motivations

Building verification tools for full-fledged concurrent languages is difficult and expensive...

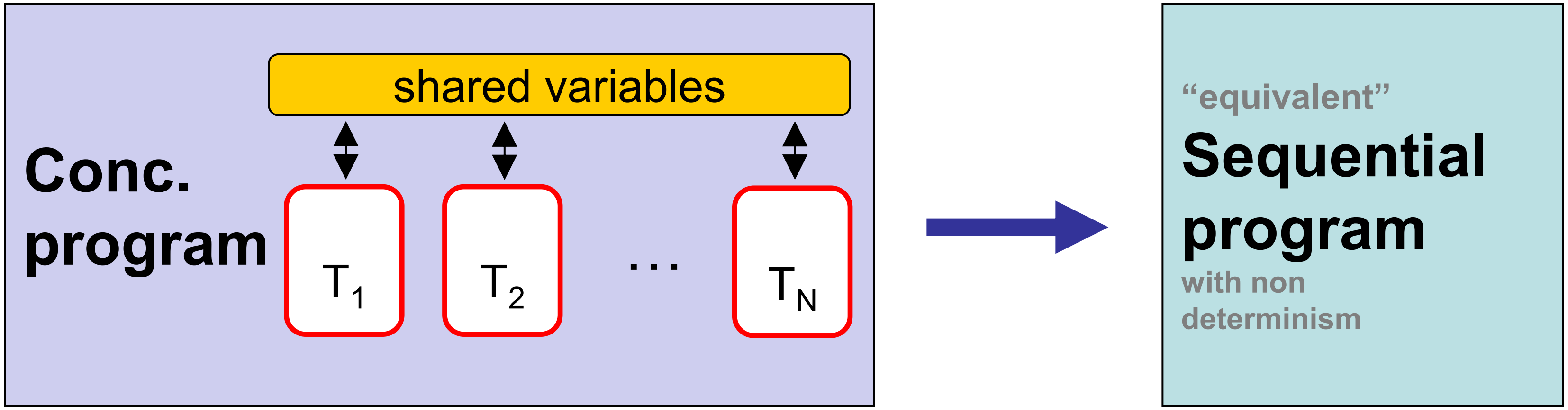
… but scalable verification techniques exist for sequential languages

- Abstraction
- SAT/SMT techniques (i.e., bounded model checking)
- *…*

⇒ **Can we leverage these?**

Sequentialization as a code-to-code translation

Code-to-code translation from multithreaded recursive programs to sequential programs that preserves reachability



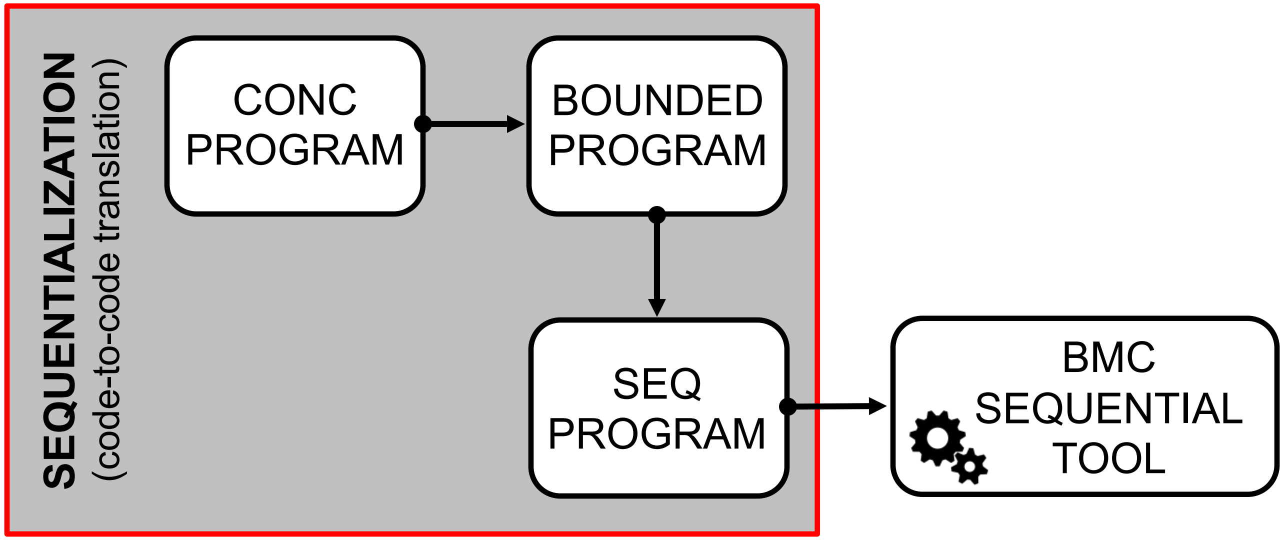
Use existing automatic verification techniques designed for sequential programs to analyze concurrent programs

[Inverso–Tomasco–Fischer–La Torre–Parlato, CAV'14]

Lazy-CSeq: Schema Overview

(a sequentialization for BMC)

Lazy-CSeq approach



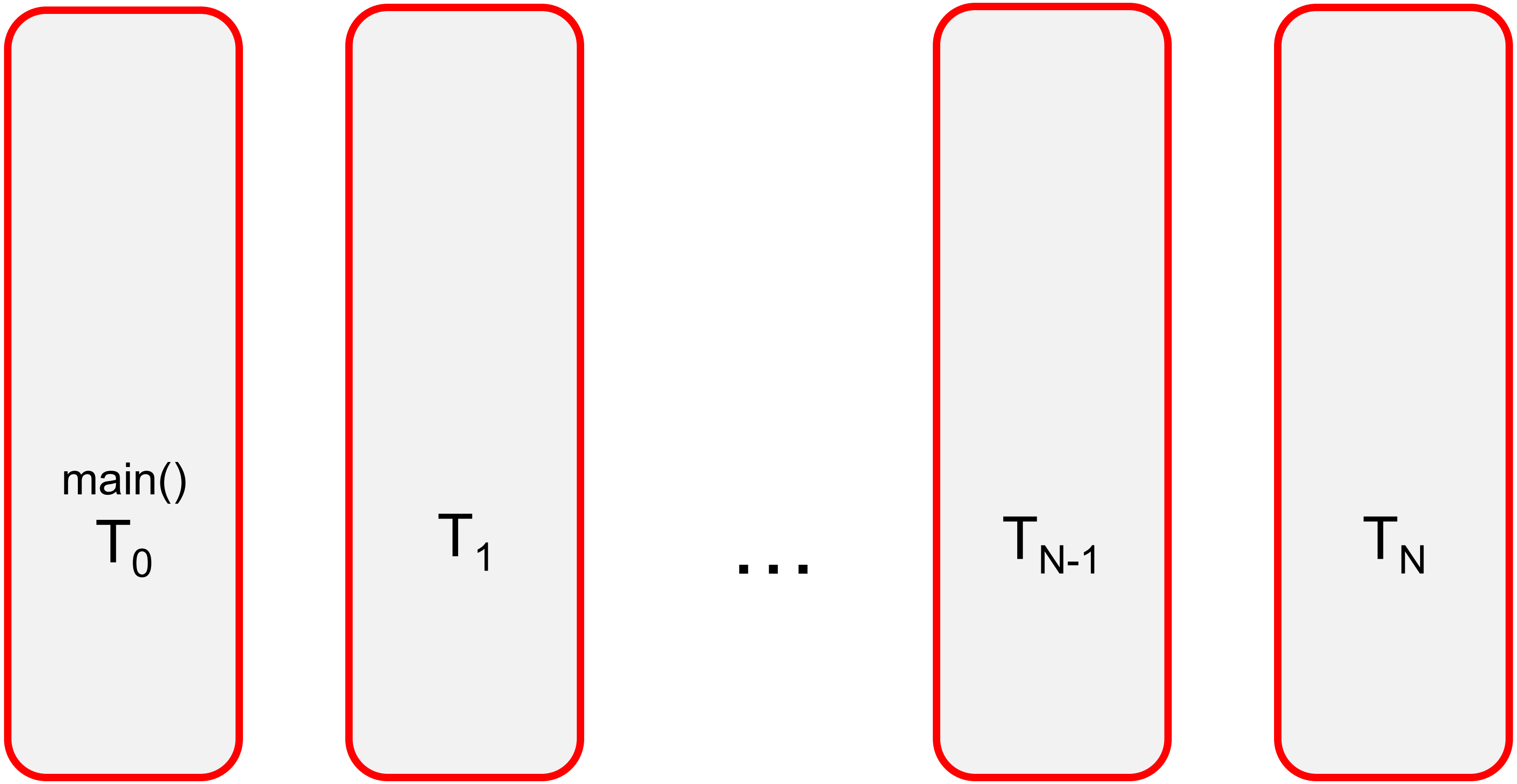
We have designed

new sequentializations targeting BMC

scalable analyses + surprisingly simple

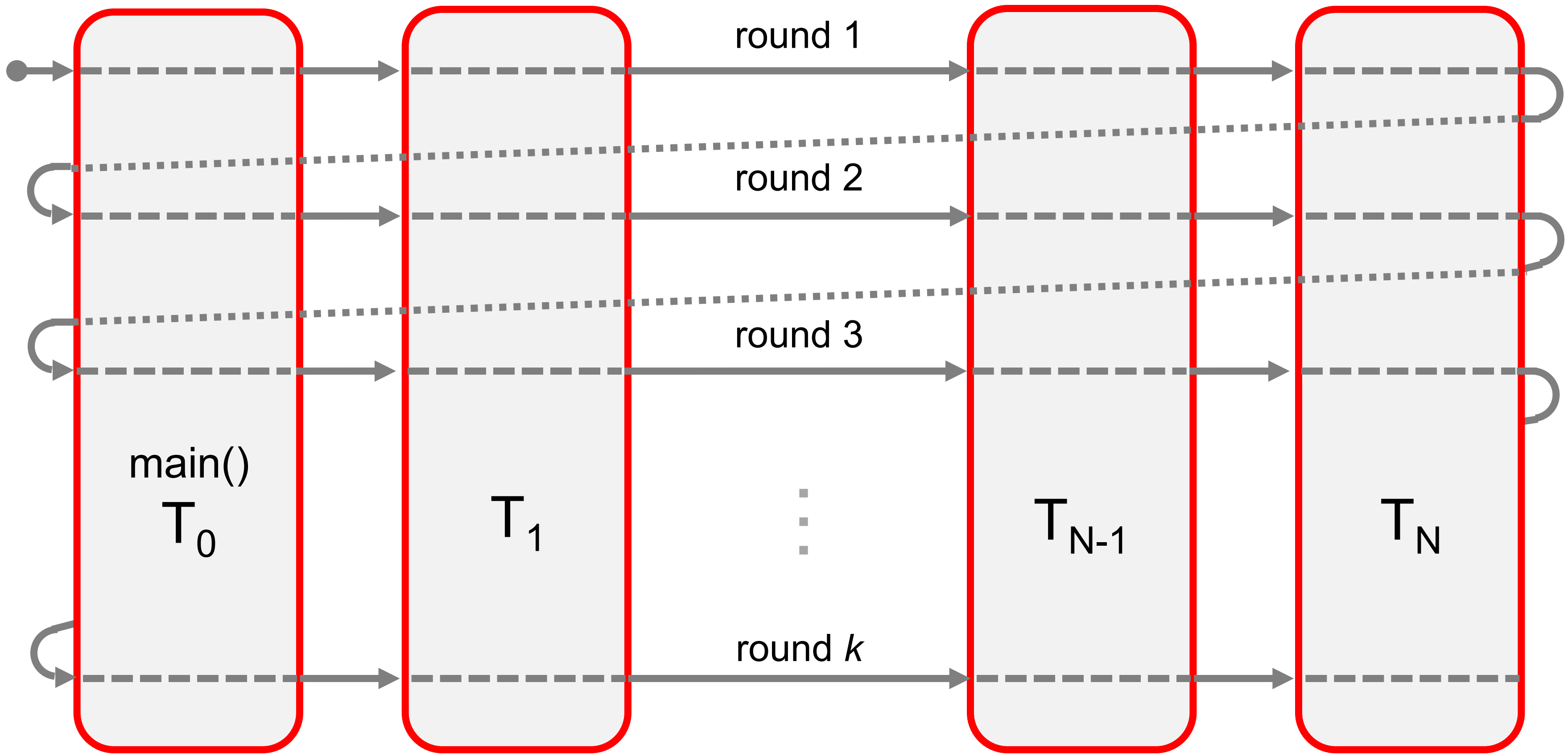
Lazy-CSeq

Bounded Concurrent Programs



- **no loops**
- **no function calls**
- **control flow only forward**
- **one procedure for each thread**

Round Robin Schedule

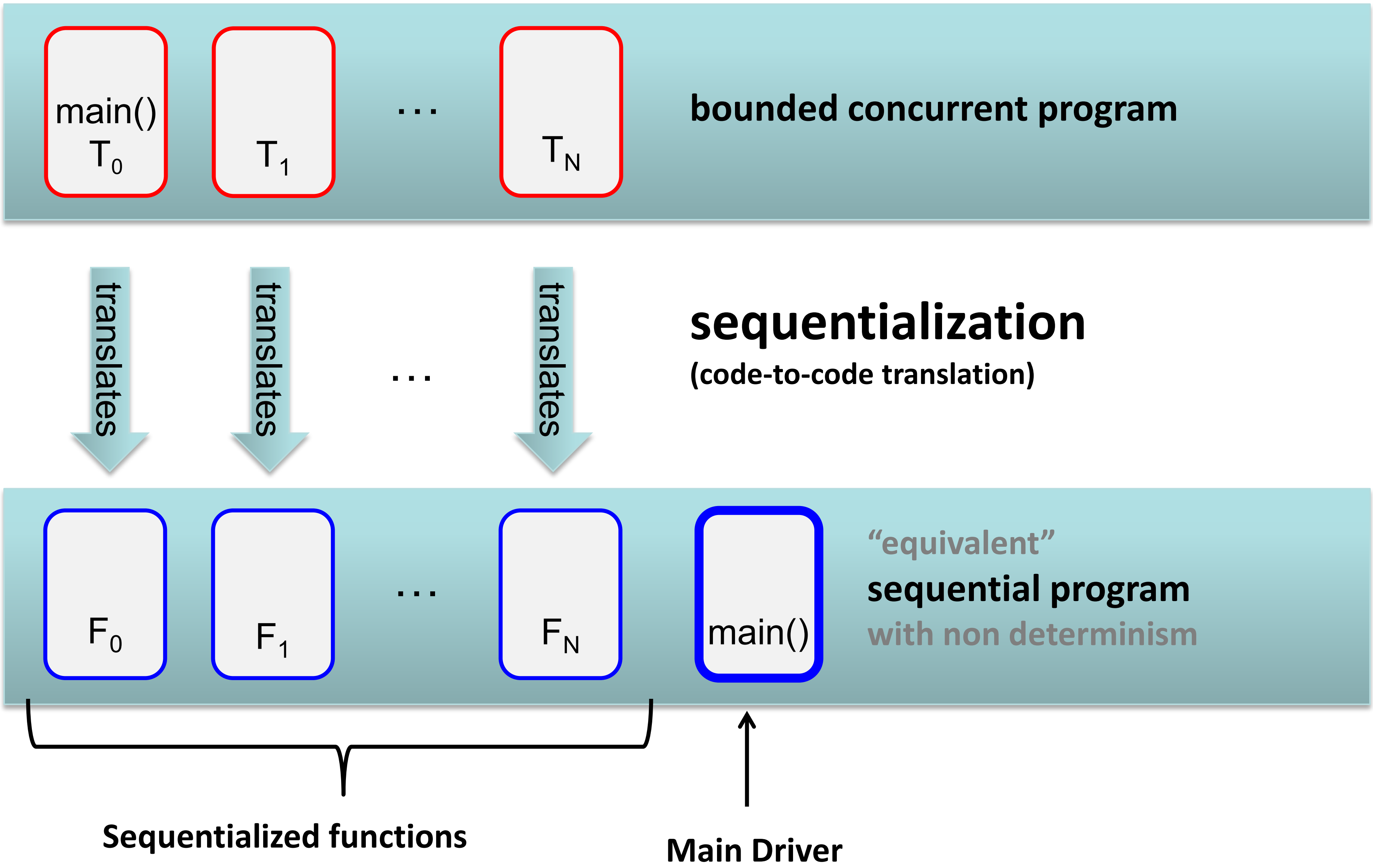


Lazy-Cseq sequentialization:

- captures all bounded Round-Robin computations for a given bound
- error manifest themselves within very few rounds

[Musuvathi, Qadeer – PLDI'07]

Schema Overview



Naïve Lazy Sequentialization

main driver

```
pc0=0;    ... pcN=0;
local0;   ... localk;

main() {
  for (r=0; r<K; r++)
    for (i=0; i<N; i++)
      // simulate Ti
      if (activei)
         Fi();
}
```

- Add a global pc for each thread
- thread locals → thread global

Naïve Lazy Sequentialization

main driver

```
pc0=0;   ... pcN=0;
local0;  ... localk;

main() {
  for (r=0; r<K; r++)
    for (i=0; i<N; i++)
      // simulate Ti
      if (activei)
         Fi();
}
```

for each round

for each thread T_i

simulate T_i

Naïve Lazy Sequentialization

main driver

```
pc0=0;    ... pcN=0;
local0;   ... localk;

main() {
  for (r=0; r<K; r++)
    for (i=0; i<N; i++)
      // simulate Ti
      if (activei)
        Fi();
}
```

Fi()

```
0:          stmt0;
1:          stmt1;
2:          stmt2;
             .
             .
             .
M:          stmtM;
```

Naïve Lazy Sequentialization

main driver

```
pc0=0;    ... pcN=0;
local0;   ... localk;

main() {
  for (r=0; r<K; r++)
    for (i=0; i<N; i++)
      // simulate Ti
      if (activei)
        Fi();
}
```

Fi()

```
switch(pci) {
   case 0: goto 0;
   case 1: goto 1;
   case 2: goto 2;
           ...
   case M: goto M;
}

0:          stmt0;
1:          stmt1;
2:          stmt2;
              .
              .
              .
M:          stmtM;
```

resume mechanism

Naïve Lazy Sequentialization

main driver

```
pc0=0;    ... pcN=0;
local0;   ... localk;

main() {
  for (r=0; r<K; r++)
    for (i=0; i<N; i++)
      // simulate Ti
      if (activei)
        Fi();
}
```

Fi()

```
switch(pci) {
  case 0: goto 0;
  case 1: goto 1;
  case 2: goto 2;
         ...
  case M: goto M;
}

0: CS(0); stmt0;
1: CS(1); stmt1;
2: CS(2); stmt2;
   .      .
   .      .
   .      .
M: CS(M); stmtM;
```

Context-switch mechanism:

```
#define CS(j)
  if (*) { pci=j; return; }
```

Naïve Lazy Sequentialization

Formula encoding:

goto statement to formula

add a guard for each crossing control-flow edge

= O(M^2) guards

```
switch(pc_i) {
   case 0: goto 0;
   case 1: goto 1;
   case 2: goto 2;
        ...
   case M: goto M;
}

0: CS(0); stmt0;
1: CS(1); stmt1;
2: CS(2); stmt2;
      .        .
      .        .
      .        .
M: CS(M); stmt_M;
```

Context-switch mechanism:

```
#define CS(j)
  if (*) { pc_i=j; return; }
```

Lazy-CSeq sequentialization

main driver

```
pc0=0;   ... pcN=0;
local0;  ... localk;
nextCS;
main()
  for (r=0; r<K; r++)
    for (i=0; i<N; i++)
      // simulate Ti
      if (activei)
        nextCS = nondet;
        assume(nextCS>=pci)
        Fi();
        pci = nextCS;
```

Guess next context-switch point

Lazy-CSeq sequentialization

main driver

```
pc_0=0;    ... pc_N=0;
local_0;   ... local_k;
nextCS;
main()
  for (r=0; r<K; r++)
    for (i=0; i<N; i++)
      // simulate T_i
      if (active_i)
        nextCS = nondet;
        assume(nextCS>=pc_i)
        F_i();
        pc_i = nextCS;
```

$F_i()$

skip

skip

```
0: J(0); stmt0;
1: J(1); stmt1;
2: J(2); stmt2;
   .        .
   .        .
   .        .
   .        .
   .        .
   .        .
M: J(M); stmt_M;
```

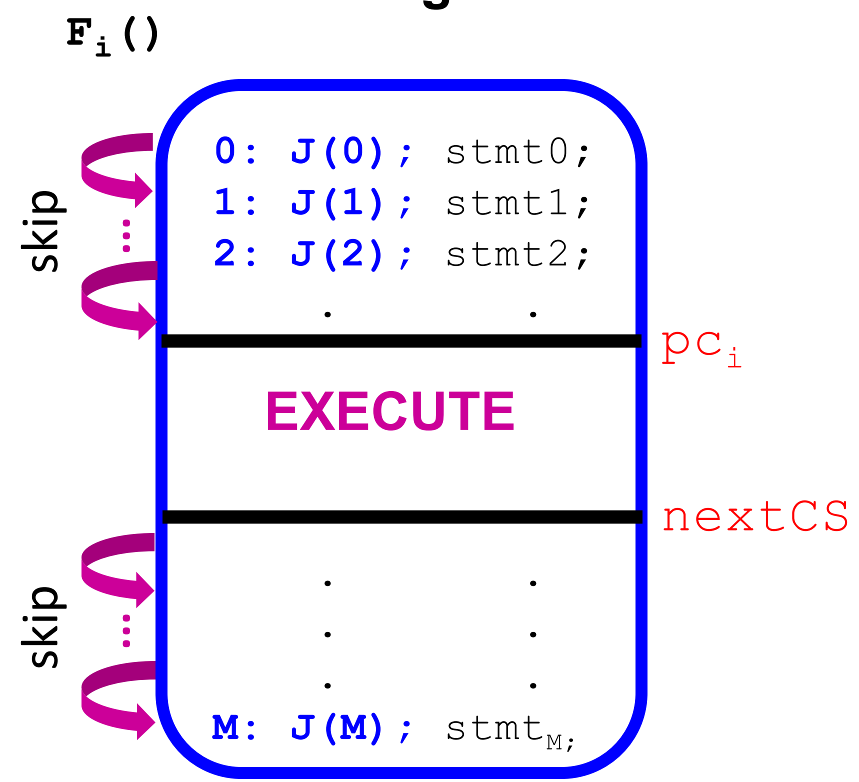
```
#define J(j)
 if (j<pc_i || j>=nextCS) goto j+1;
```

Lazy-CSeq sequentialization

main driver

```
pc0=0;    ... pcN=0;
local0;   ... localk;
nextCS;
main()
   for (r=0; r<K; r++)
      for (i=0; i<N; i++)
         // simulate Ti
         if (activei)
            nextCS = nondet;
            assume(nextCS>=pci)
            Fi();
            pci = nextCS;
```

resuming + context-switch




```
#define J(j)
 if (j<pci || j>=nextCS) goto j+1;
```

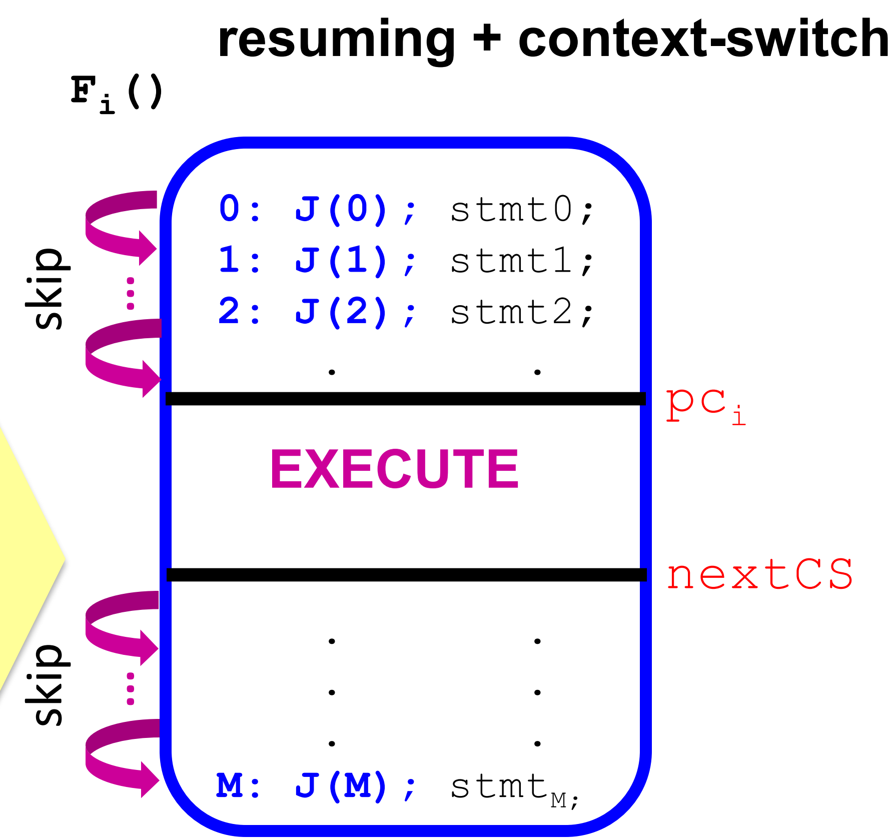
Lazy-CSeq sequentialization

Formula encoding:

goto statement to formula

add a guard for each crossing control-flow edge

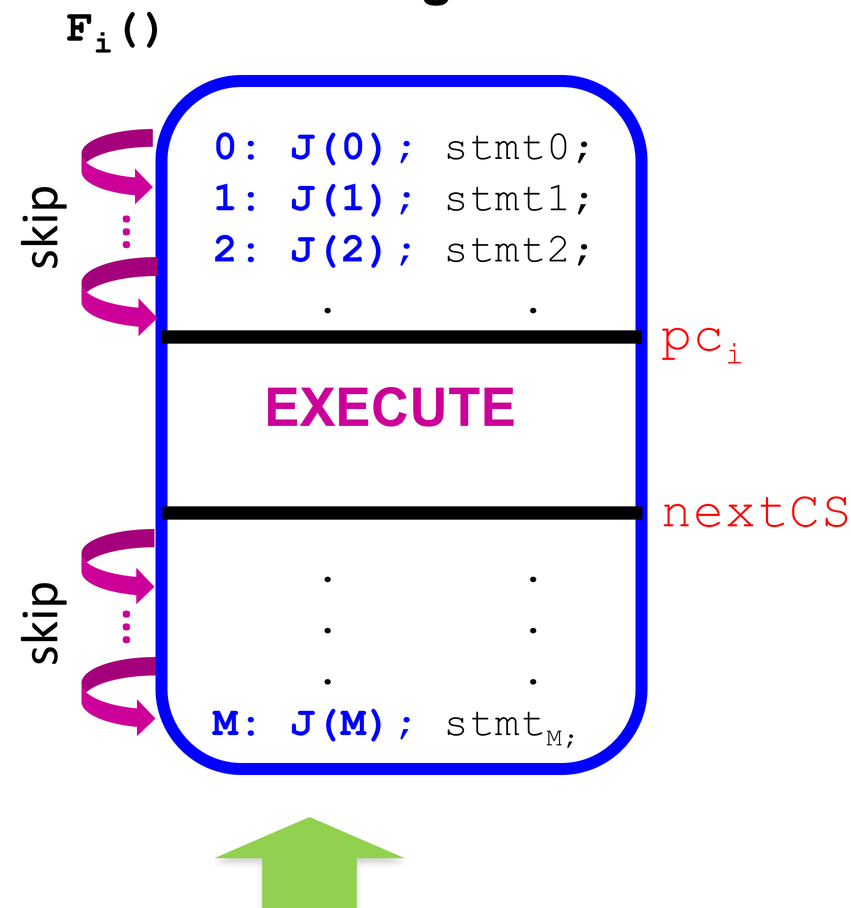
= O(M) guards




```
#define J(j)
  if (j<pc_i || j>=nextCS) goto j+1;
```

Lazy-CSeq sequentialization

resuming + context-switch



inject light-weight, non-invasive control code

- no non-determinism
- no pc assignments
- no return

```
#define J(j)
 if (j<pc_i || j>=nextCS) goto j+1;
```

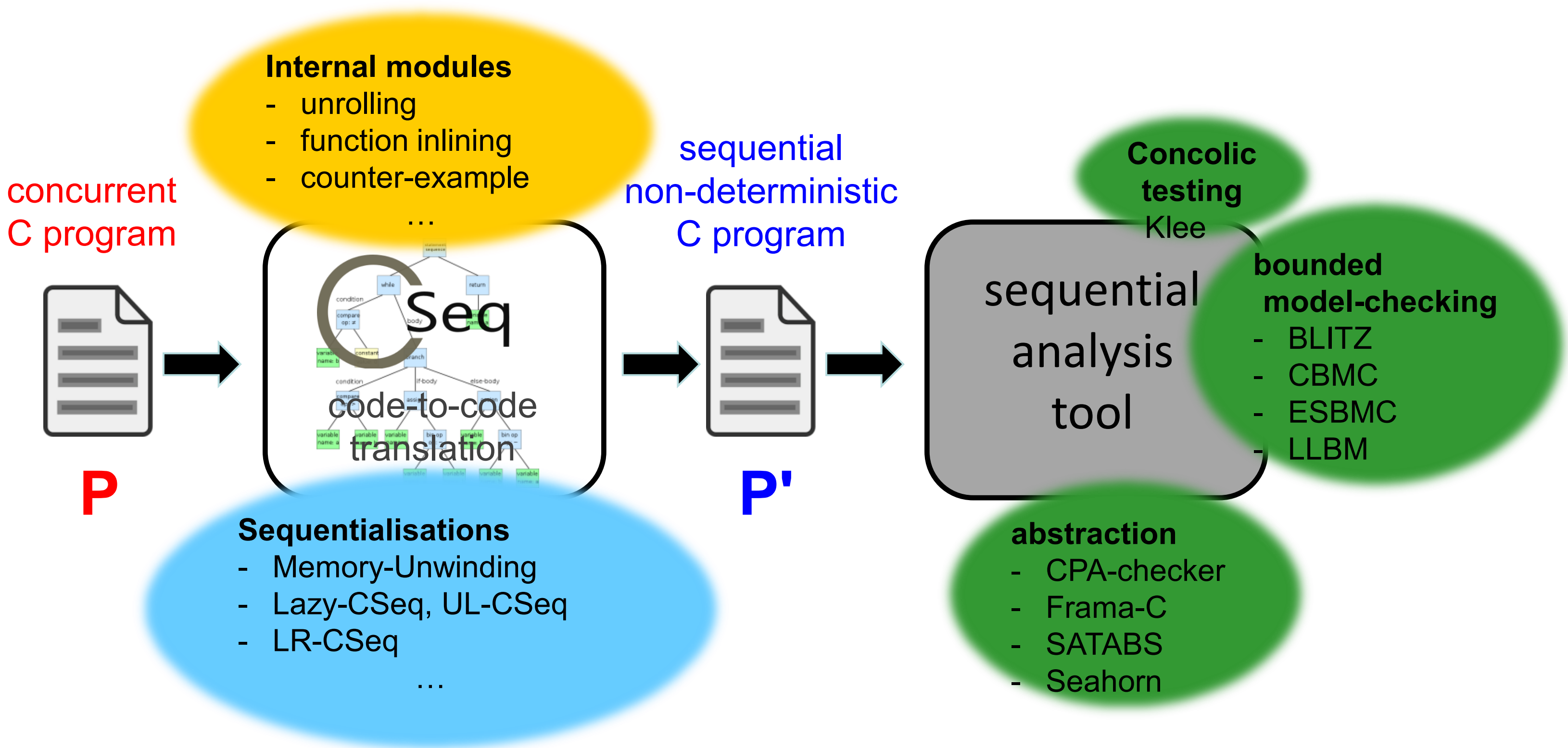
(lazy-cseq-example.pdf)

Lazy-CSeq tool

[Inverso-Nguyen-Fischer-La Torre-Parlato, ASE'15]

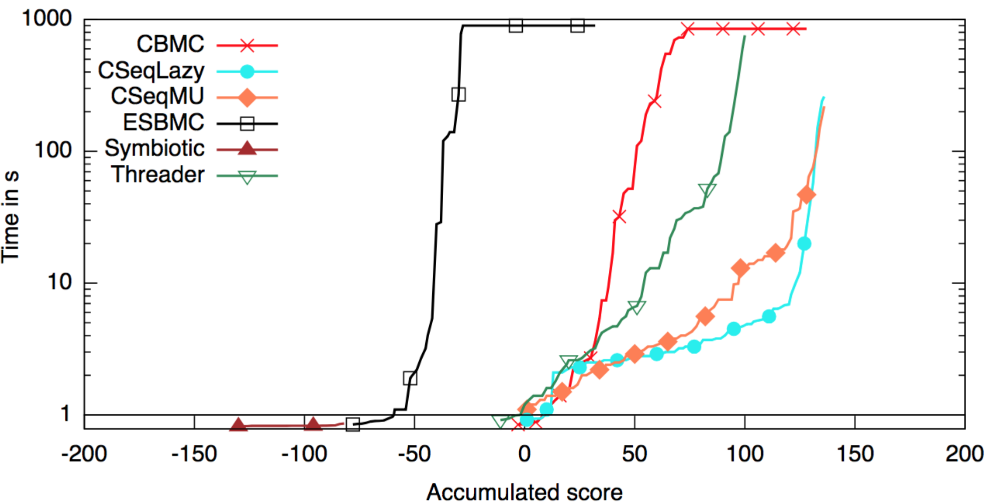
CSeq is a framework that simplifies code-to-code translations

- for C programs + Pthread
- comprises several code-to-code translation modules
- supports several sequential analysis back-end tools

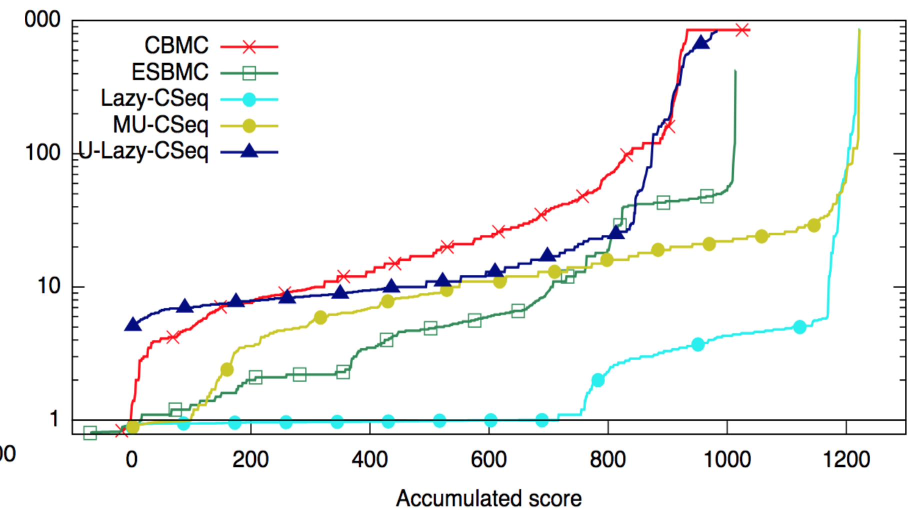


SV-COMP concurrency (2014-17)

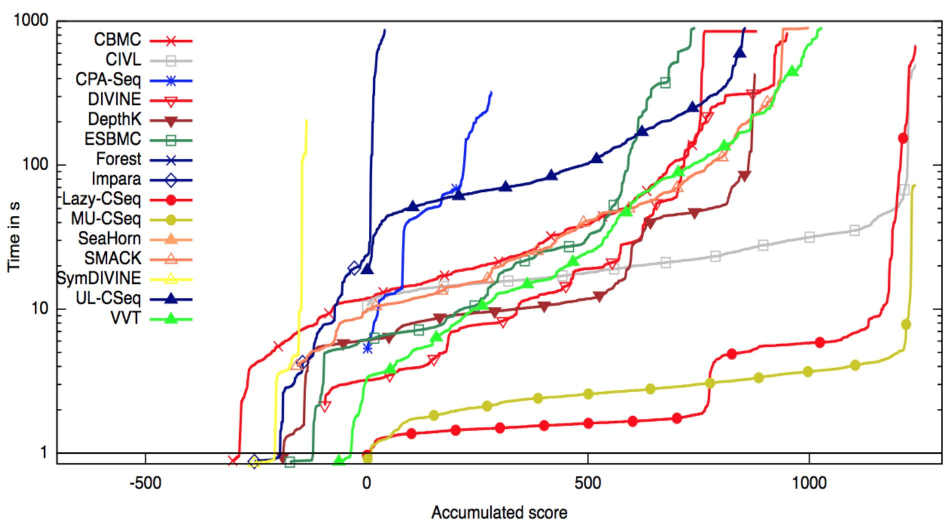
2014



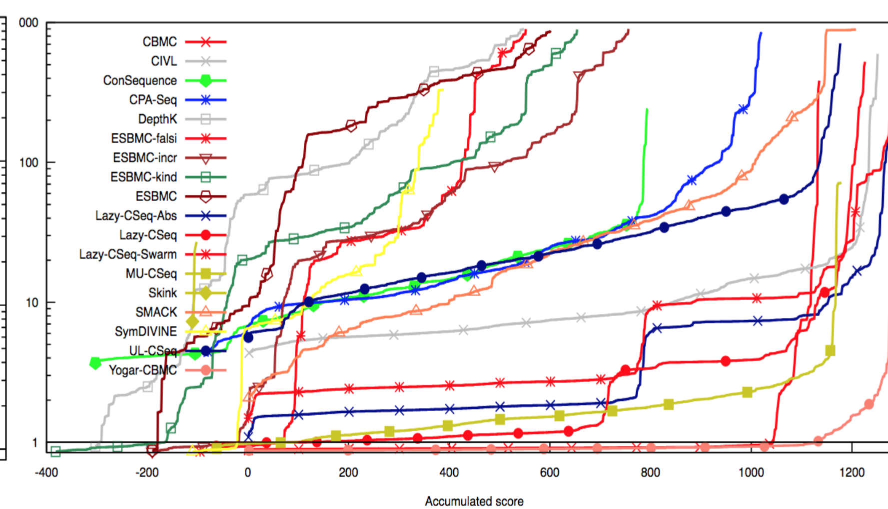
2015



2016



2017



Experiments on lock-free data structures

(hard benchmarks)

Safestack **[Concurrency Testing Using Controlled Schedulers: An Empirical Study, Thomson, Donaldson, Betts, PPoPP'14, TOPC'16]**

– ABA problem: requires context bound of 5 for exposure

– **Lazy-CSeq** can find bug in **~7h** and **6.5GB**

  ▹ #unwind=3, #rounds=4, #threads=4, size=152 visible stmts

– **all other tools fail**

Eliminationstack **[Bouajjani, Emmi, Enea, Hamza--POPL'15]**

– ABA problem: requires 7 threads for exposure

– **Lazy-CSeq** can find bug in **~13h** and **4GB**

  ▹ #unwind=1, #rounds=2, #threads=8, size=52 visible stmts

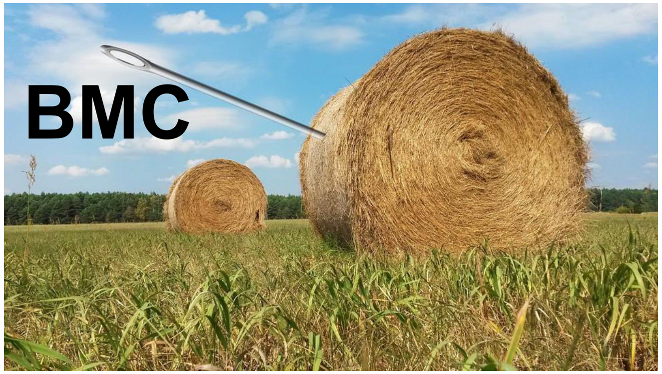
– **all other tools fail**

State of affairs

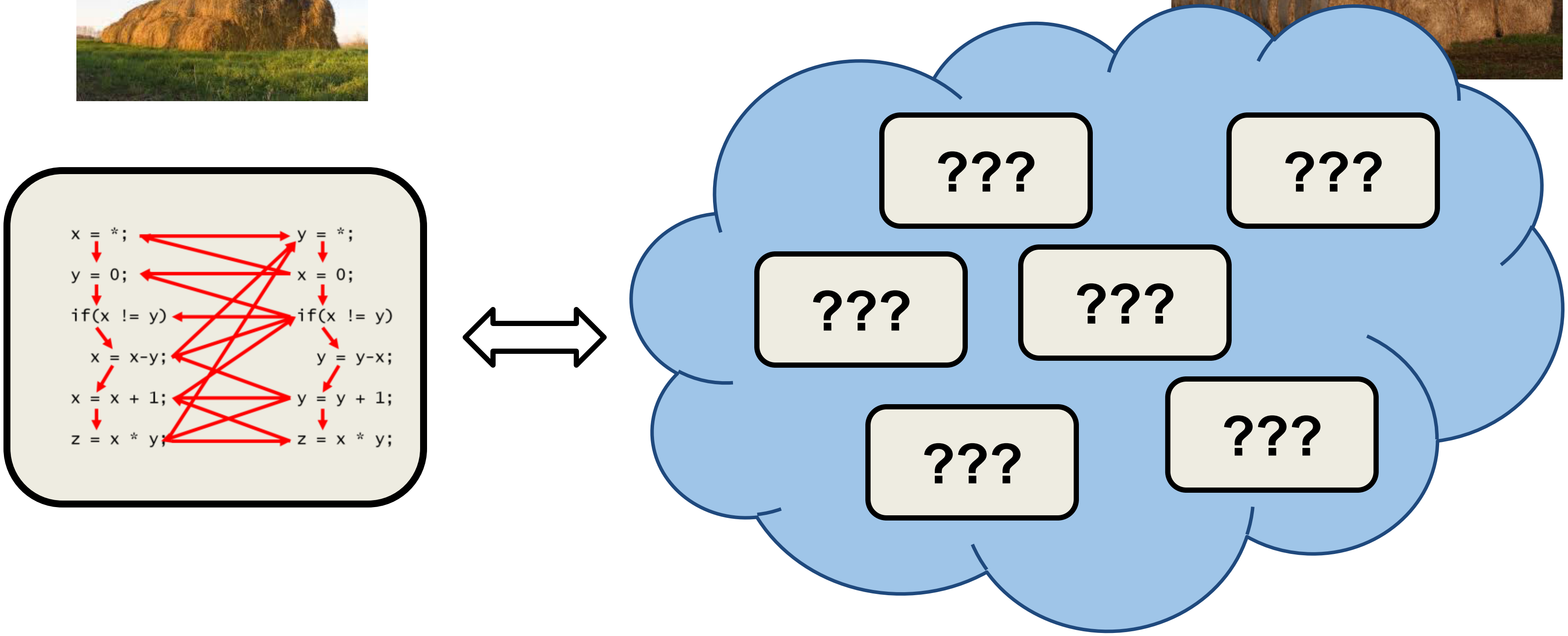
[Nguyen –Schrammel–Fischer–La Torre–Parlato, ASE'17]

VERISMART

Intuition



How can we get the bales?

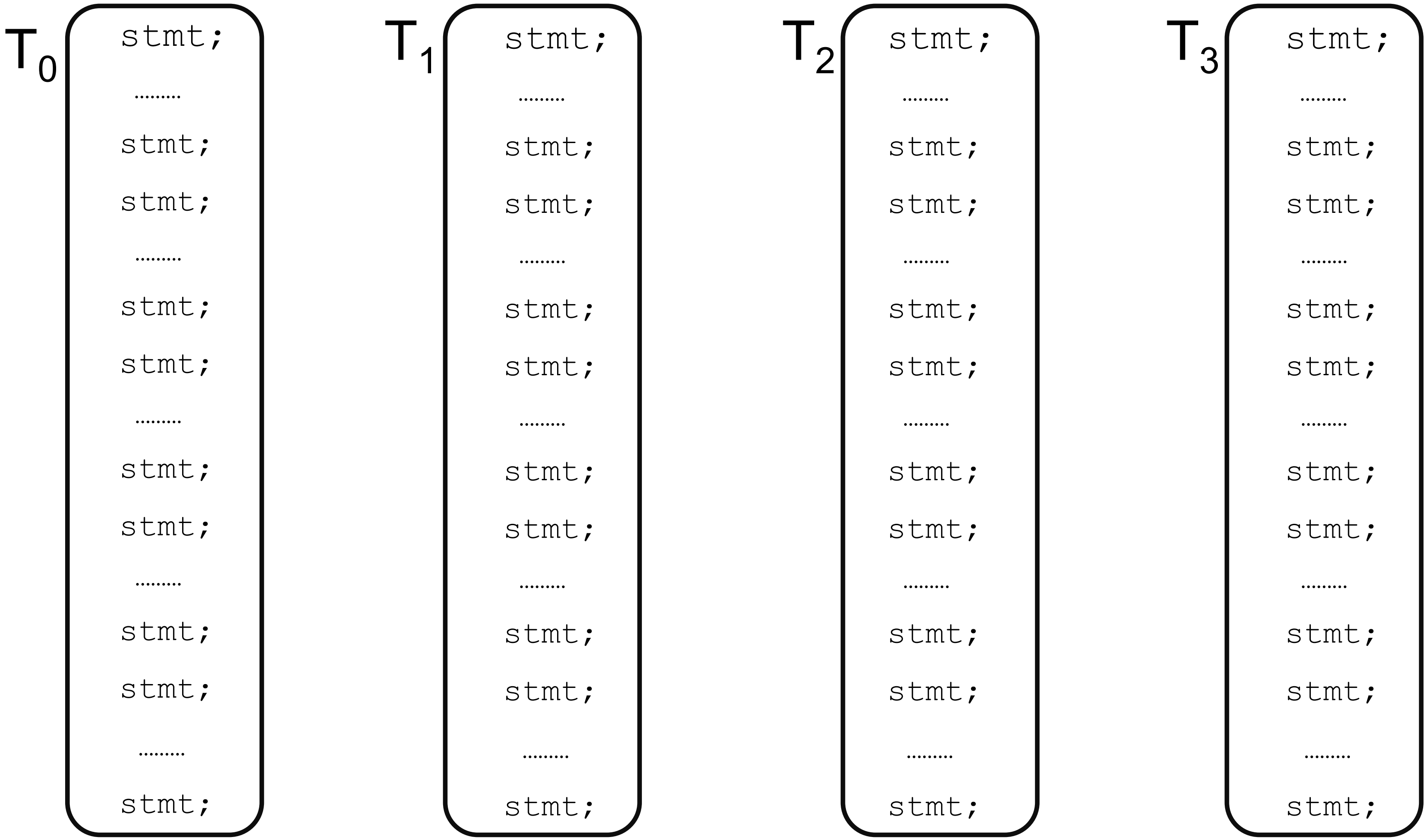


How can we partition a task into **independent smaller** tasks?

Tiling threads

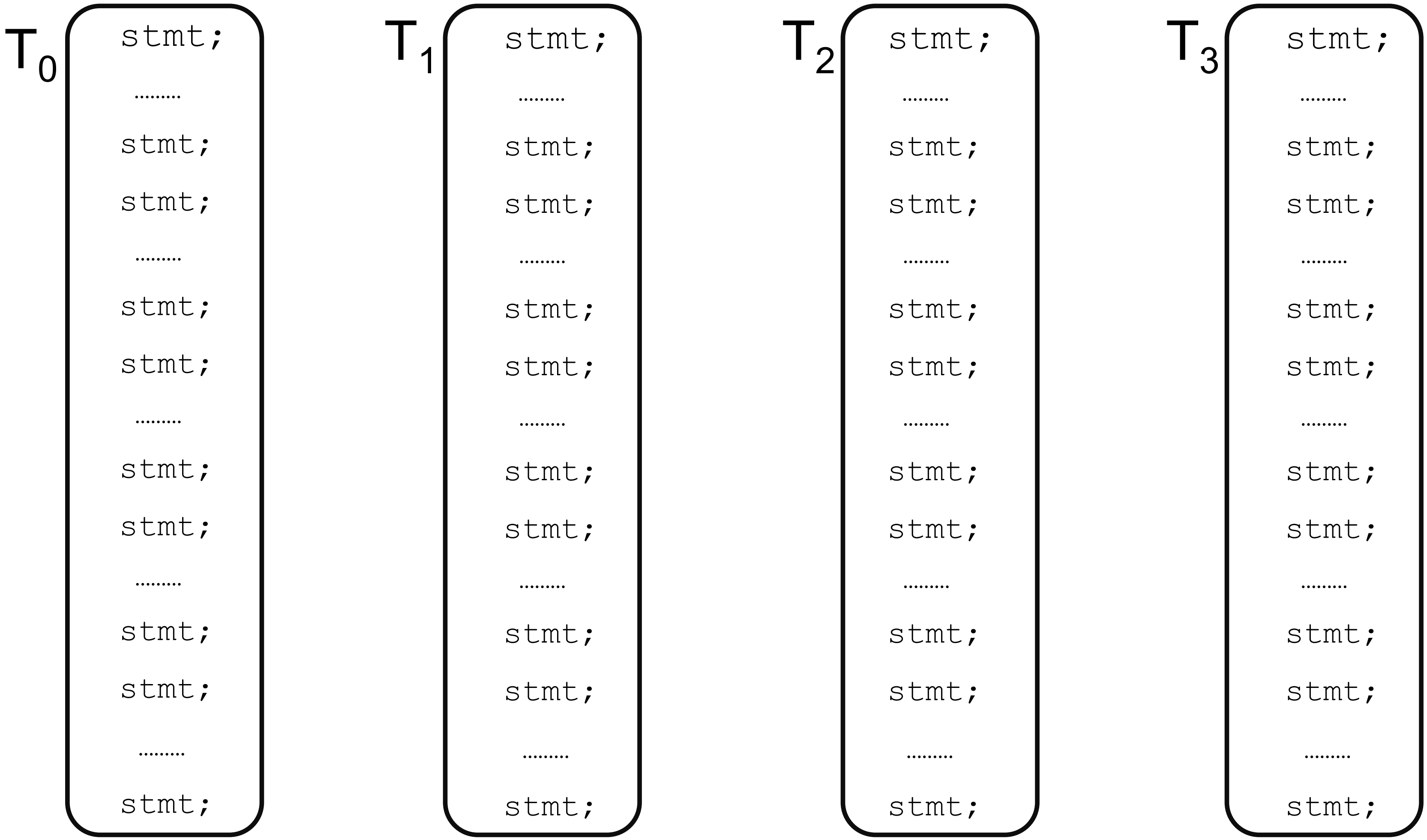
Assumption: bounded concurrent programs

- control can only go forward
- same # of stmts, e.g.1000



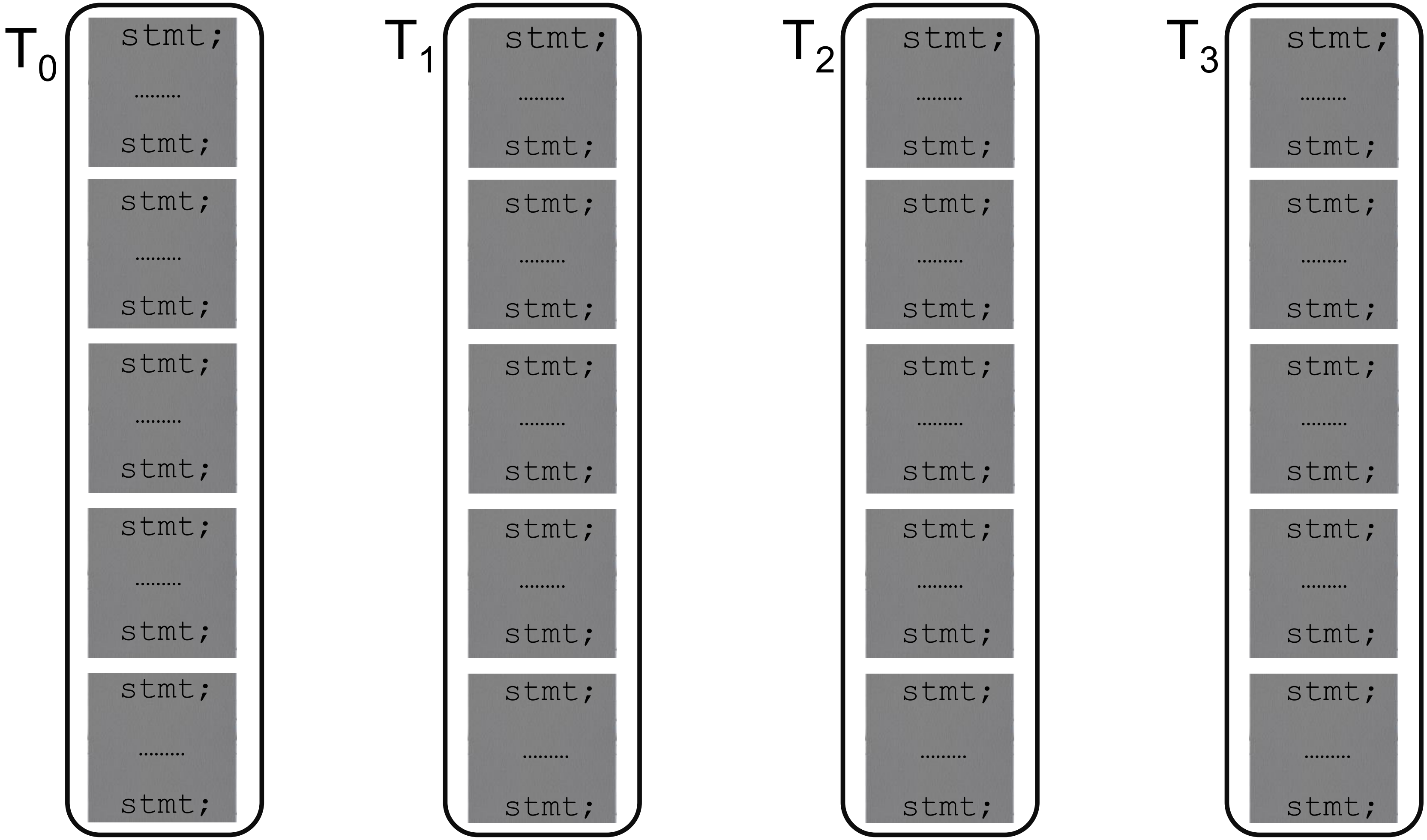
Tiling threads

Tasks as variants of the original program by splitting the code of each thread **into** fragments (**tiles**) and allowing **context-switches only in some of them**



Tiling threads

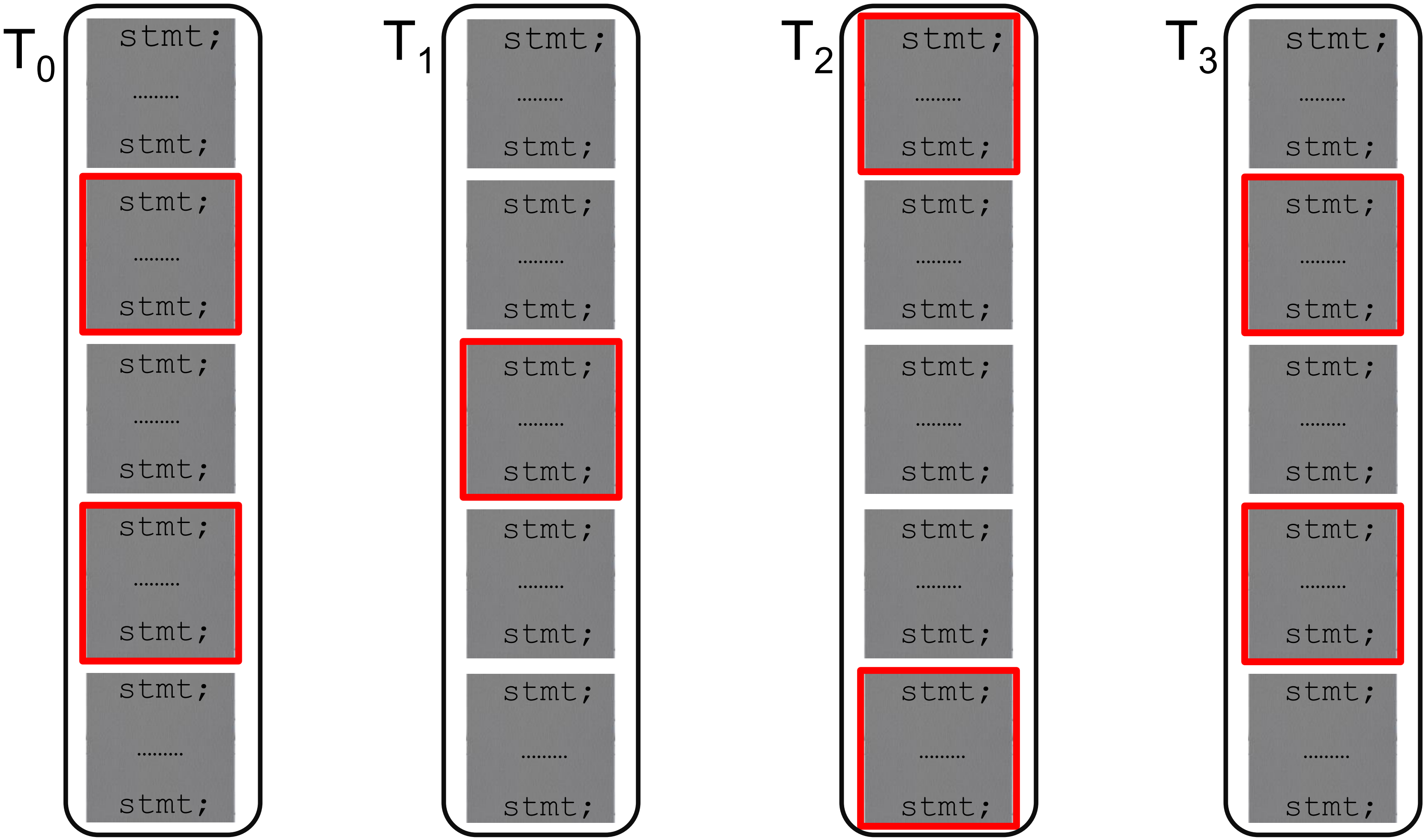
- **tile**: (contiguous) subset of visible statements
- **tiling**: partition of program into tiles
- **uniform window tiling**: all tiles have same size



Tiling threads

Observation: For a *k*-round execution at most *k* tiles per thread are involved in context-switching!

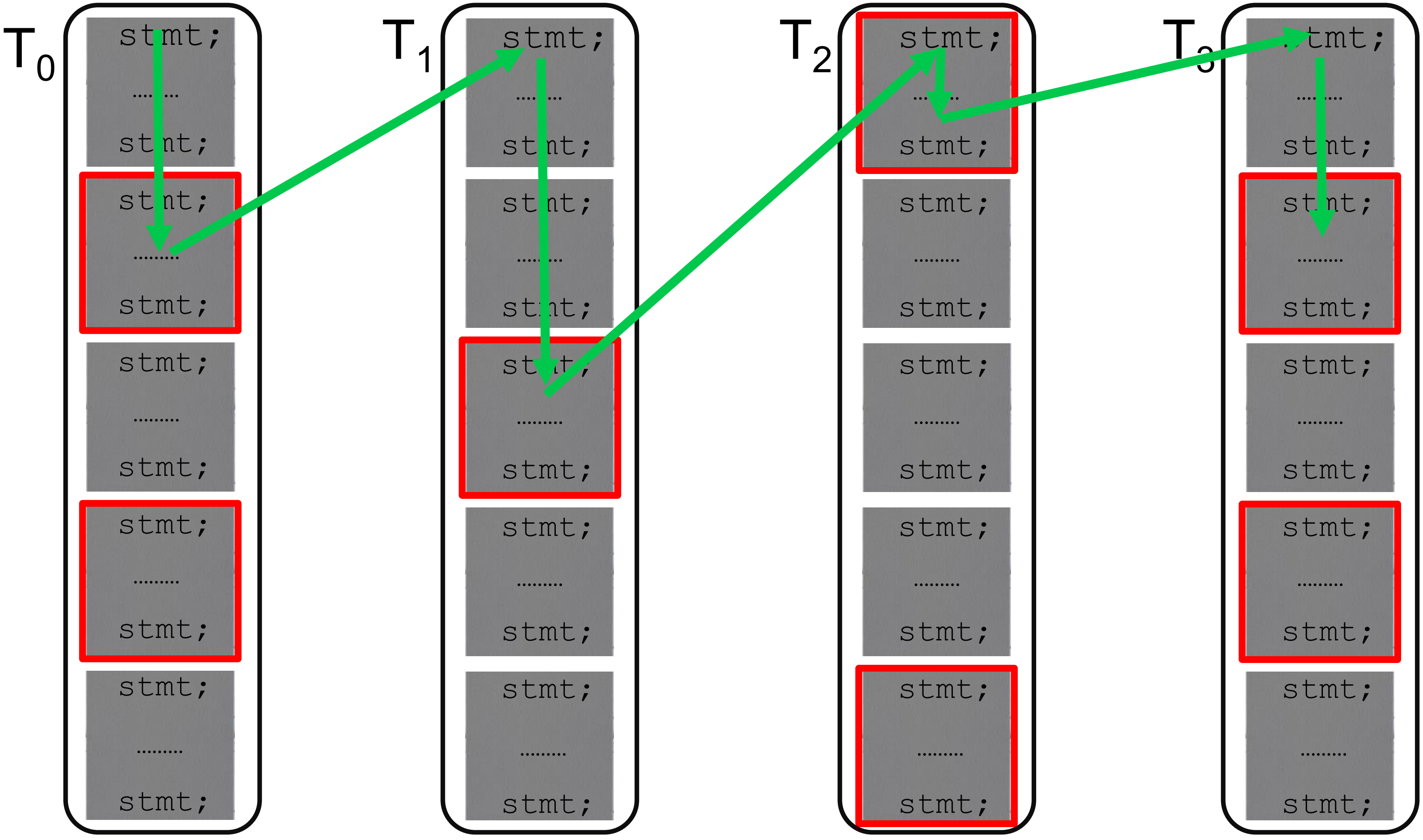
Example: *k*=2



Tiling threads

Observation: For a *k*-round execution at most *k* tiles per thread are involved in context-switching!

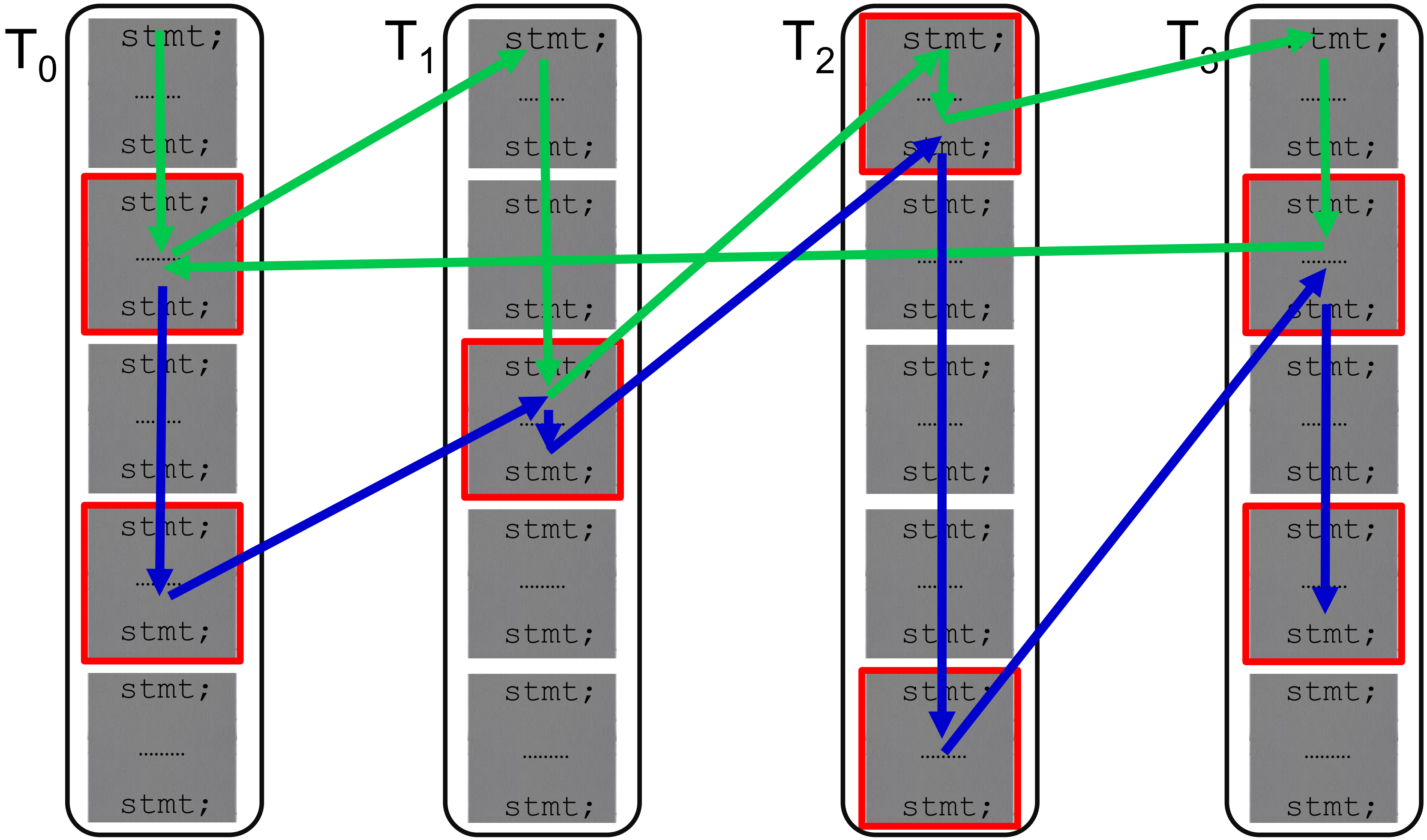
Example: k=2



Tiling threads

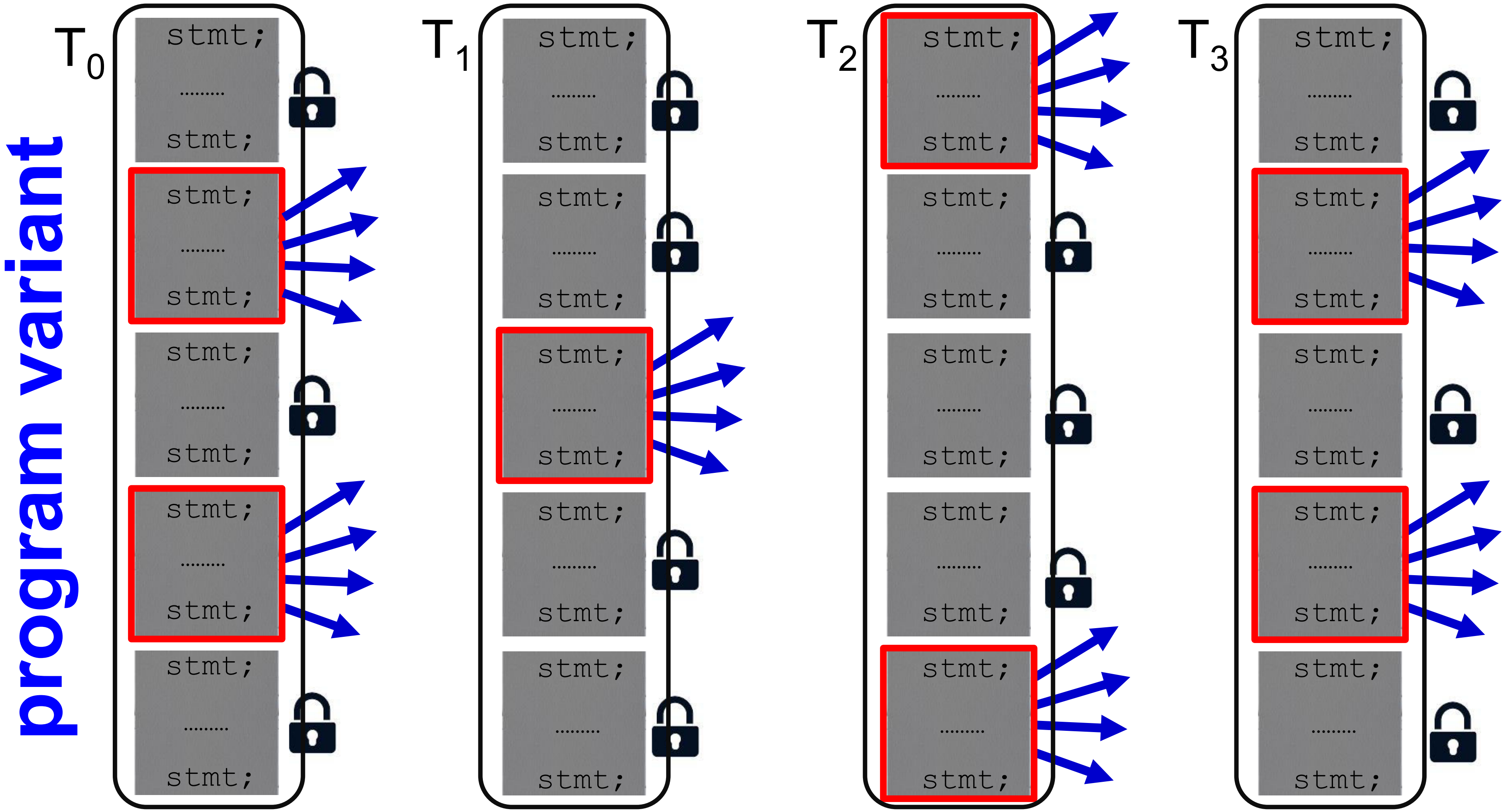
Observation: For a *k*-round execution at most *k* tiles per thread are involved in context-switching!

Example: k=2



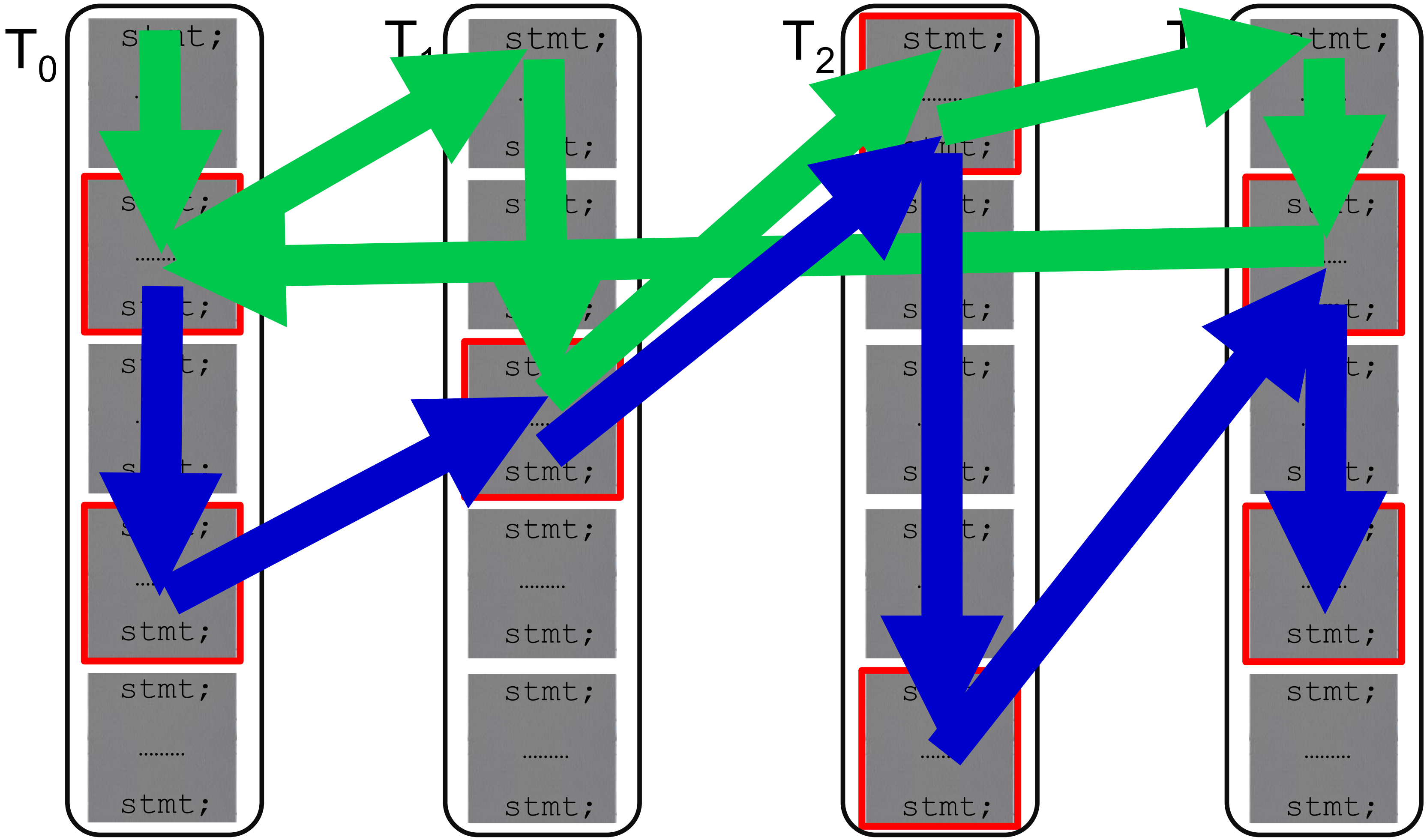
k-selections & program variants

- ***k*-selection**: subset of *k* tiles for each thread
  - **context switches** are **only** allowed from **selected tiles**
- each *k*-selection specifies a **reduced interleaving instance**

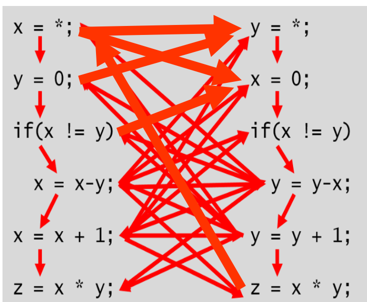


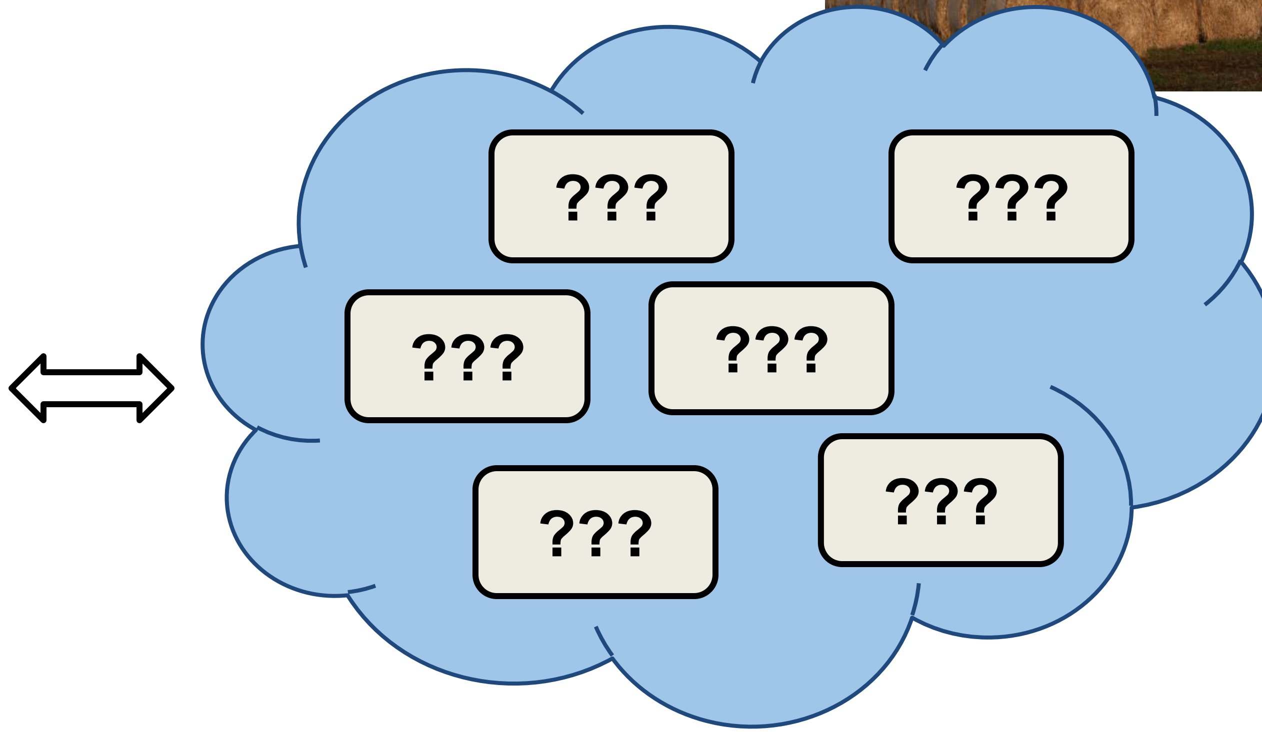
Tiling threads

- ***k*-selection**: subset of *k* tiles for each thread
  - **context switches** are **only** allowed from **selected tiles**
- each *k*-selection specifies a **reduced interleaving instance**



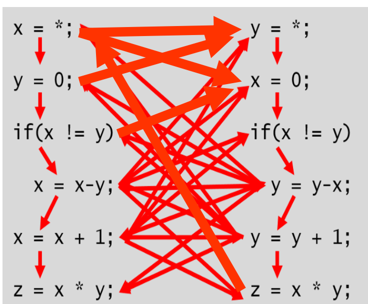
How can we get the bales?



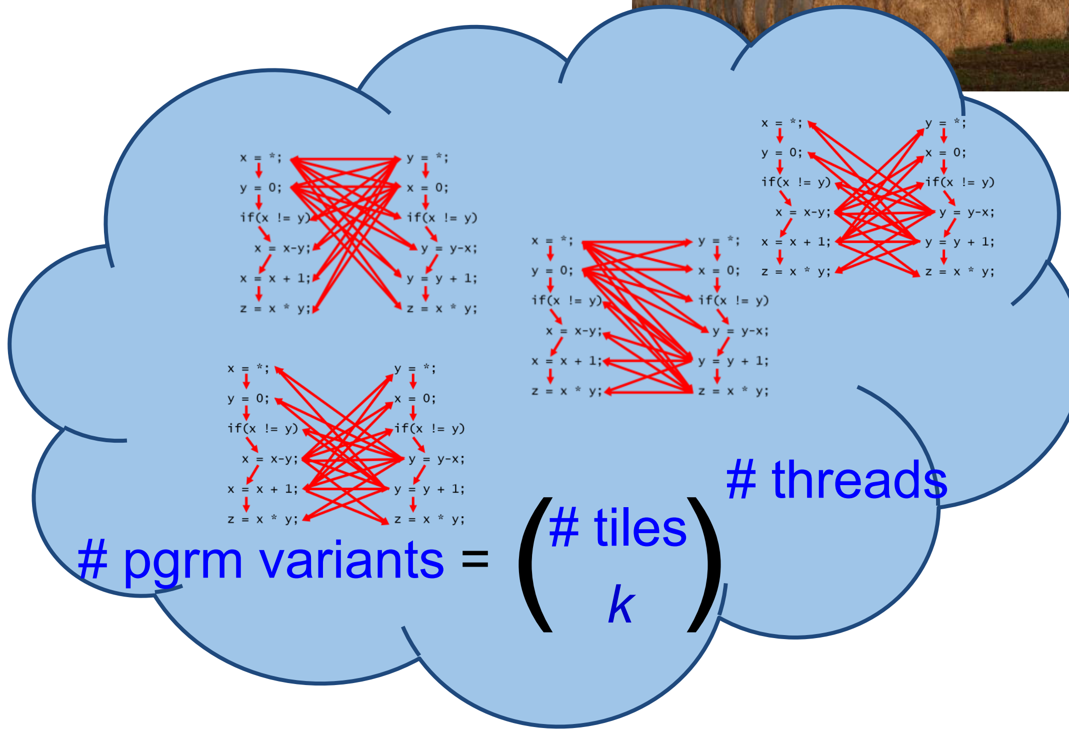


How can we partition a task into **independent smaller** tasks?

How can we get the bales?



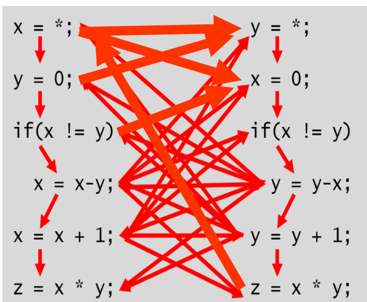
⟺



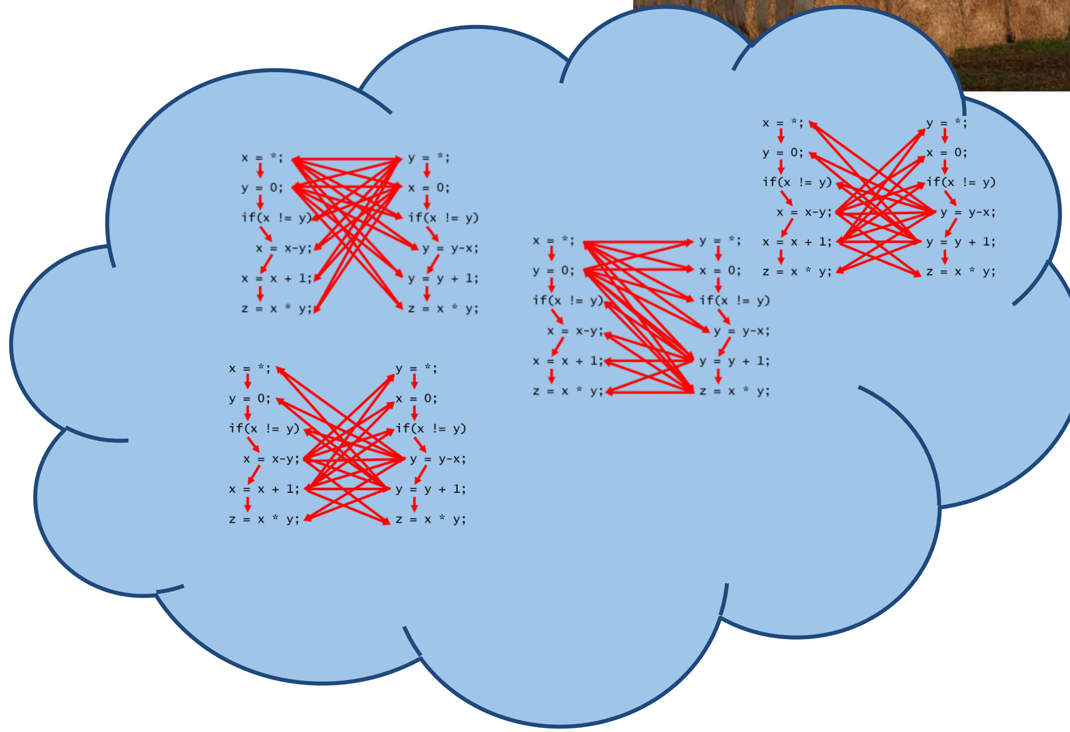
Answer:

- fix a tiling and *k*
- generate the program variants for all *k*-selections

How can we get the bales?



⟺



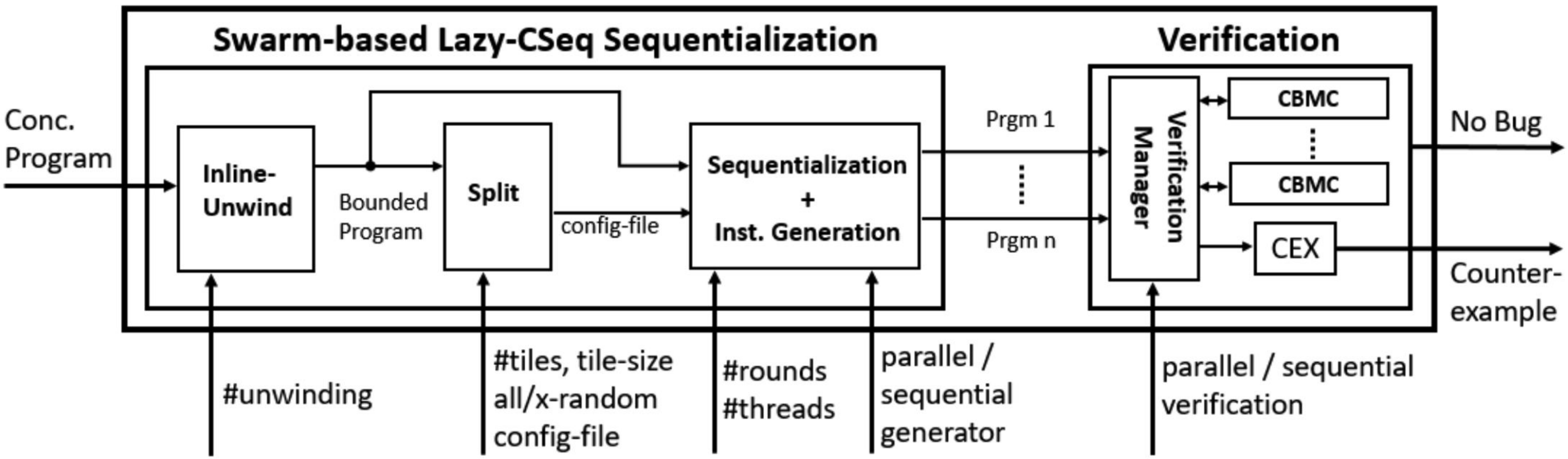
Answer:

- fix a tiling
- generate the program variants for all *k*-selections

Why does this work?

- each prgm variant captures a subset of k-round executions of P
- each execution is captured by a prgm variant

VERISMART architecture



Eliminationstack: results

Eliminationstack

- ABA problem: requires 7 threads for exposure
- **Lazy-CSeq** can find bug in **~13h** and **4GB**
  - #unwind=1, #rounds=2, #threads=8, size=52 visible stmts

– each experiment: 8,000 instances chosen randomly

VeriSMART: 2 tiles								
#1: tile size 12, t_max 1.5hrs			#2: tile size 14, t_max 2hrs			#3: tile size 18, t_max 3hrs		
Verification	Time	Memory	Verification	Time	Memory	Verification	Time	Memory
Min	34.9	945.2	Min	39.7	979.84	Min	37.1	999.8
Max	4753.6	1199.1	Max	7195.2	1281.3	Max	10762.0	1785.5
Average	1116.3	1017.8	Average	2169.5	1096.3	Average	3162.41	1156.91
instances with bug: 38.33%			instances with bug: 61.38%			instances with bug: 69.01%		

fastest instances very fast – 1000x

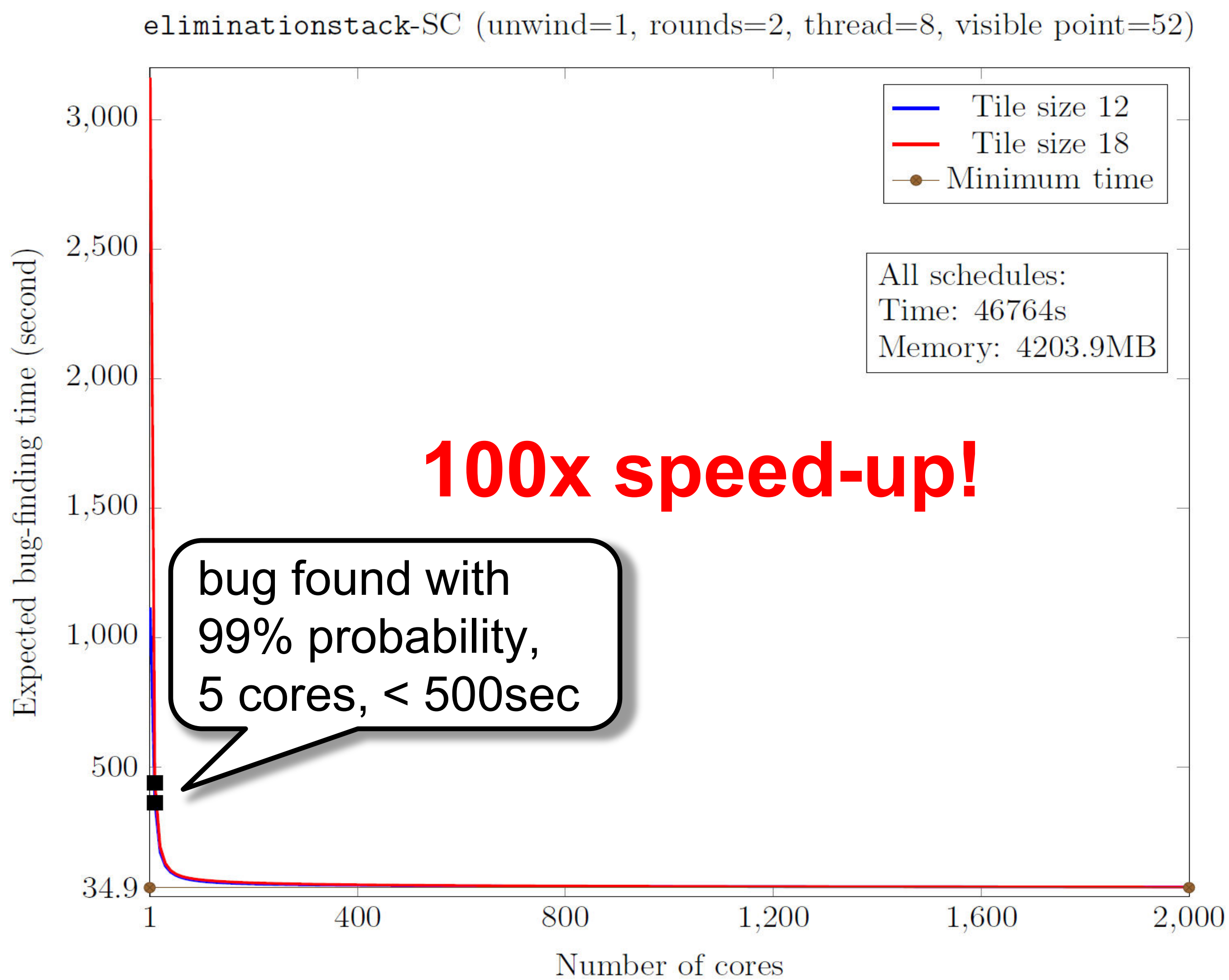
reduced memory consumption – 4x

average still very fast – 40x

high fraction of bug-exposing instances

some slowdown for larger tile sizes – 10x

Eliminationstack: expected bug-finding time



Safestack: experiments

Safestack

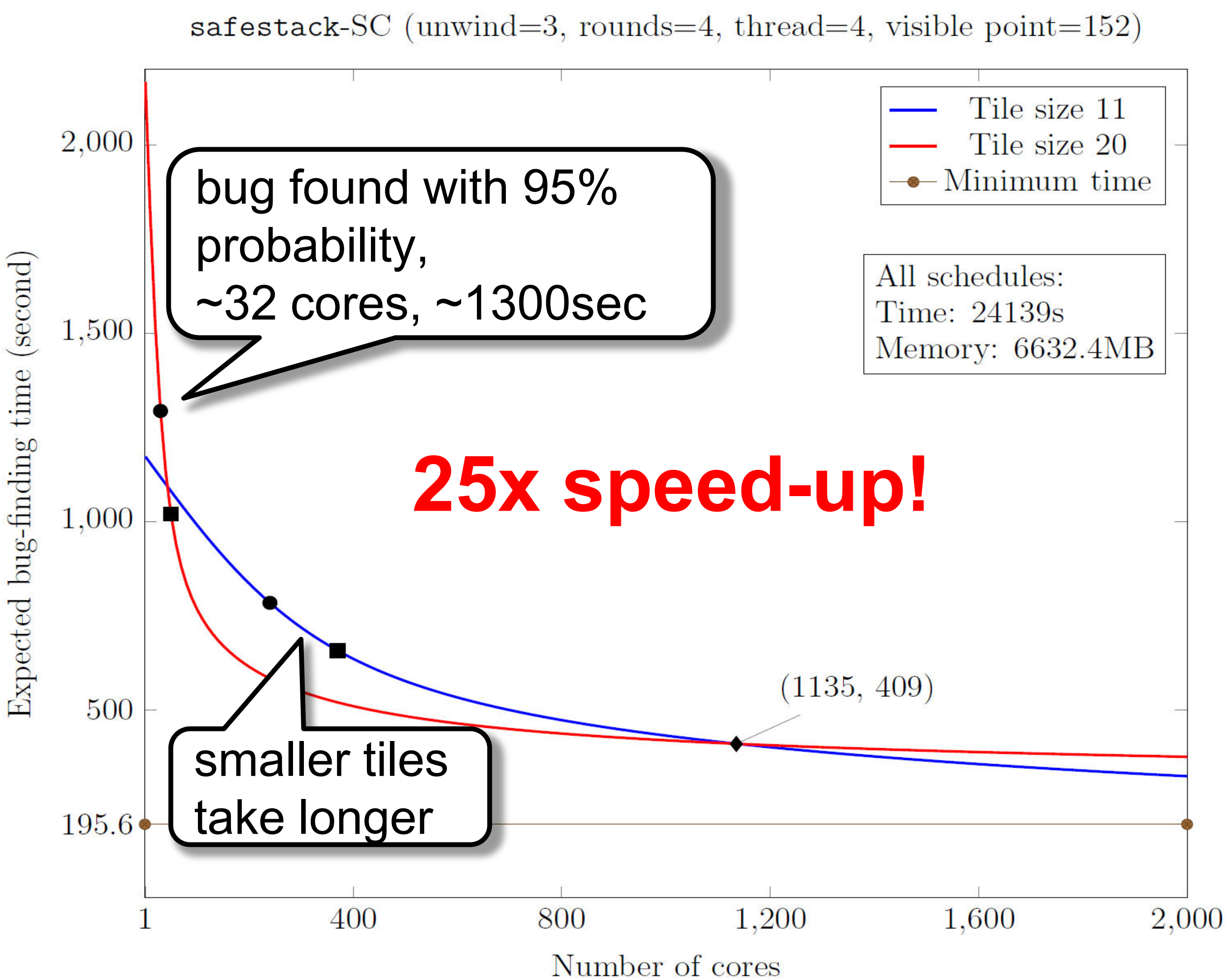
- ABA problem: requires context bound of 5 for exposure
- **Lazy-CSeq** can find bug in **~7h** and **6.5GB**
  - ▹ #unwind=3, #rounds=4, #threads=4, size=152 visible stmts

VeriSMART: 4 tiles per thread								
#1: tile size 11, t_max 1hr			#2: tile size 14, t_max 1hr			#3: tile size 20, t_max 4hrs		
Verification	Time	Memory	Verification	Time	Memory	Verification	Time	Memory
Min	195.6	774.5	Min	574.8	846.6	Min	313.0	850.3
Max	2662.6	1265.7	Max	3521.8	1450.4	Max	10315.8	3830.8
Average	1172.2	928.8	Average	1851.1	1147.3	Average	2167.5	1230.1
instances with bug: 1.26%			instances with bug: 2.14%			instances with bug: 10.20%		

lower fraction of bug-exposing instances than eliminationstack

…but boosted with larger tile sizes

Safestack: expected bug-finding time



Conclusions

Testing

VERISMART

Lazy-CSeq

BMC: fully symbolic

PROBABILITY →

← PERFORMANCE

Current & Future Work

- Fast over-approximations to filter out safe instances
  - abstract interpretation based on BMC?

- BBD-based analysis + VERISMART
  - Safestack: bug found < 1 min

- Weak Memory Models
  - Efficient encoding / Lazy-CSeq
    - ▹ Memory shadowing
  - VERISMART

People

Omar Inverso
PhD U. Southampton

Ermenegildo Tomasco
PhD U. Southampton

Truc L Nguyen
PhD U. Southampton

Salvatore La Torre
U. Salerno

Bernd Fischer
U. Stellenbosch

Peter Schrammel
U. Sussex, diffblue

Thank You

CSeq

users.ecs.soton.ac.uk/gp4/cseq