Identifying Use After Free Variables in Fire and Forget Tasks

Jyothi Krishna V S and Vassily Litvinov

jkrishna@cse.iitm.ac.in

IIT Madras

May 2, 2017

Overview

- Chapel
- `begin` tasks
- Use-After-Free Variables
- CCFG construction
- CCFG
- Parallel Program States: PPS
- Results
- Conclusion & Future works
- Choice of atomics

Chapel

- Chapel: programming language designed for productive parallel computing.
- Portable design: bridging gap between HPC architectures.
- Unifying HPC units (replace CUDA, MPI, OpenMP).
- Task parallelism

Philosophy

[a] *Good, top-down language design can tease system-specific implementation details away from an algorithm, permitting the compiler, run-time, applied scientist, and HPC expert to each focus on their strengths*

[a] SC '16, Salt Lake

Task parallelism

- Task Parallelism Constructs:
  - unstructured lifetime: `begin` tasks.
  - structured lifetime: `cobegin` tasks,`coforall` tasks.
- Data synchronization:
  - Non Blocking: `atomic` variables.
  - Blocking: `sync` variables.
  - `single` variables.

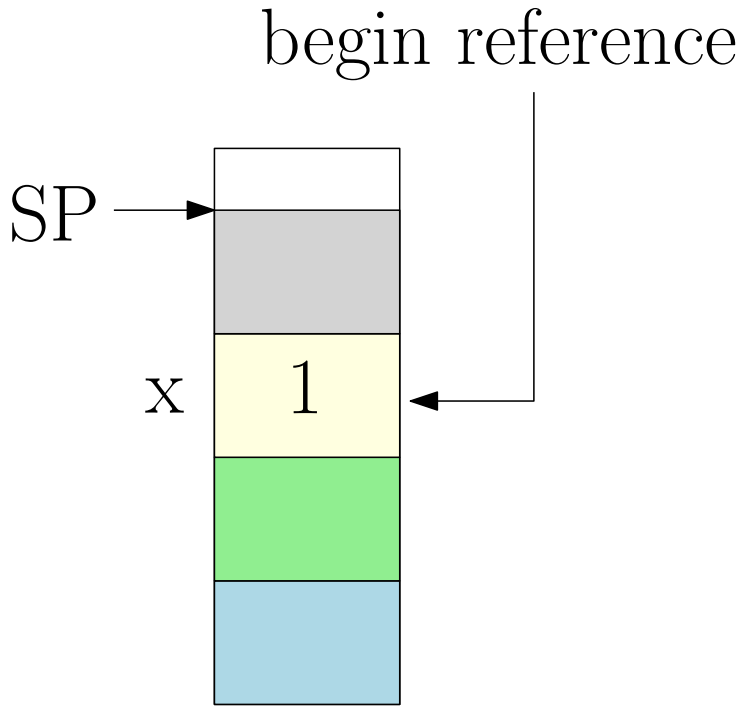
begin

- Creates a dynamic task with an unstructured lifetime.
- *fire and forget*
- Similar to `async` in x10.
- low synchronization an scheduling cost.

```
...
begin writeln("hello ");
writeln("world ");
...
```

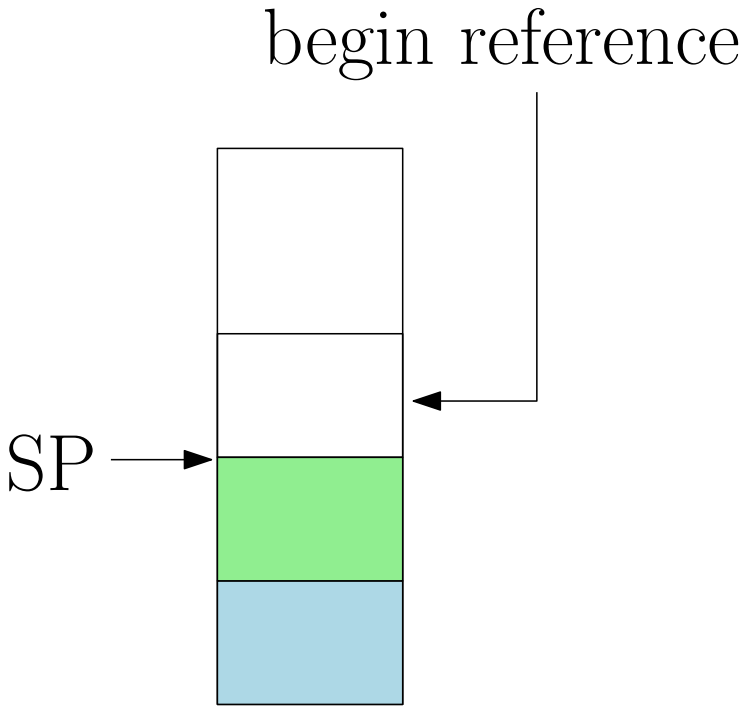
```
Expected outputs:
hello world
world hello
```

Use-After-Free Variables



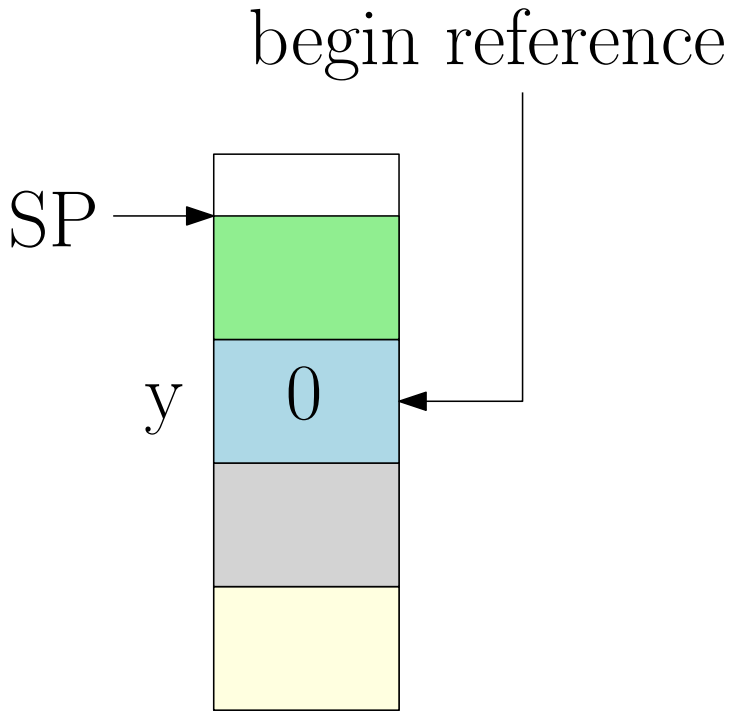

```
...
{
 x = 1;
 begin (ref x)
  {
   if(x == 0)
   print "chaos"
  }
}
....
y = 0;
```

Use-After-Free Variables




```
...
{
 x = 1;
 begin (ref x)
  {
   if(x == 0)
   print "chaos"
  }
}
....
y = 0;
```

Use-After-Free Variables




```
...
{
 x = 1;
 begin (ref x)
  {
   if(x == 0)
   print "chaos"
  }
}
....
y = 0;
```

`sync` and `single`

- Two properties: state and value.
- Two states: *empty*, *full*.
- Initialized with empty.
- Trailing $ to differentiate from rest of variables.
- `sync`
  - Always Blocking
  - On write: empty $\rightarrow$ full.
  - On read: full $\rightarrow$ empty.
  - one-to-one synchronization.
  - Reusable.
- `single`
  - Blocking if state is empty.
  - On write: empty $\rightarrow$ full.
  - one-to-many synchronization.

sync blocks

- Block synchronization.
- Similar to *finish* blocks in x10.
- All tasks declared inside the `sync` blocks should finish before the parent task proceeds.

```
...
int x;
sync {
   int y; //root task
   begin(ref x, ref y) {
    // x safe, y ?
   }
   . . .
}
...
```

General Algorithm

- CCFG: Concurrent Control Flow Graph.
- Extract out `begin` tasks and external variable (OV) uses into CCFG.
- Traverse through CCFG.
- PPS: Parallel Program States.
- Partial Inter-Procedural analysis.
- collect all `sync` scope details at call site.

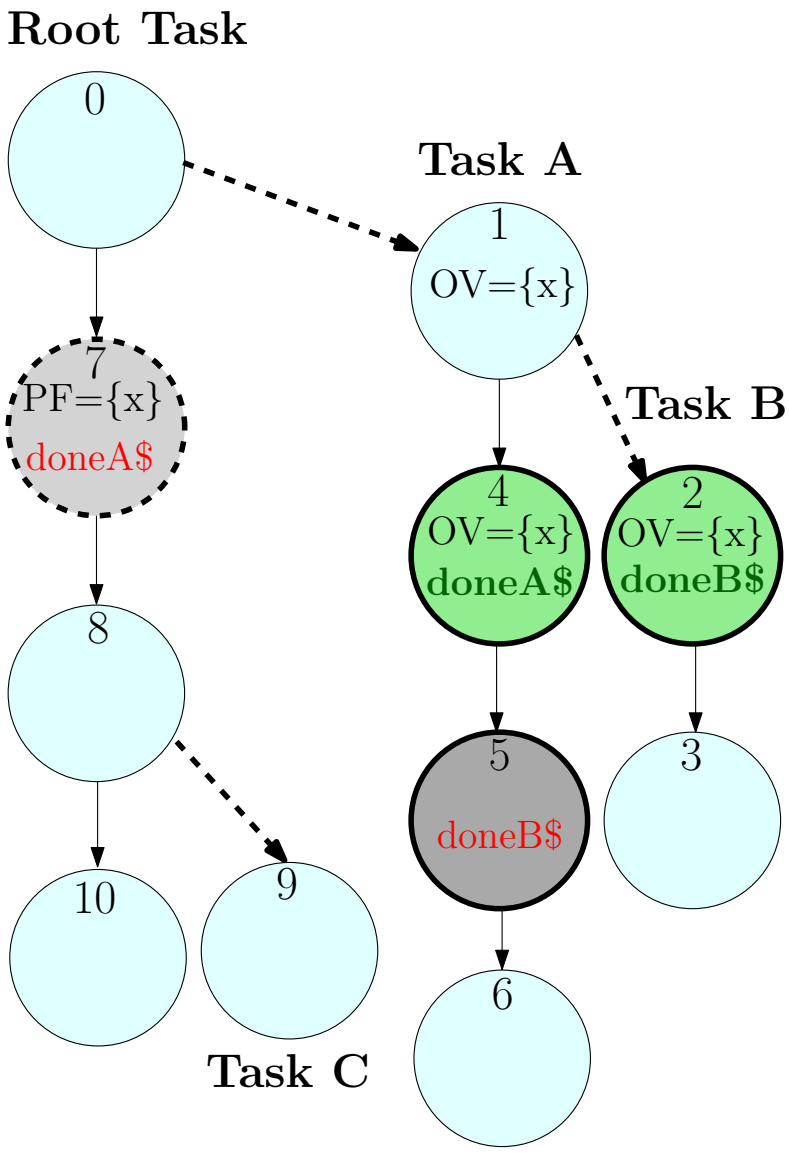
CCFG

- Bounded by a Concurrent Control Flow event.
  - Encounter `begin` statement.
  - Read/Write on a synchronization variable.
- nested function declaration.
- A CCFG Node
  - Outer Variable Set: OV
  - Synchronization type.
  - Synchronization variable.
- Sub graph of nested functions expanded at call site.
- A live set of `sync` block scope is maintained.
- Safe OV accesses are removed.

```
1   proc outerVarUse( ) {
2     var x: int = 10;
3     var doneA$: sync bool;
4     begin with (ref x) {
5       writeln(x++);
6       var doneB$:  sync bool;
7       begin with (ref x){
8         writeln(x);
9         doneB$ = true;
10      }
11      writeln(x);
12      doneA$ = true;
13      doneB$;
14    }
15    doneA$;
16    begin with (in x){
17      writeln(x);
18    }
19  }
```

- Task A Line 4.
- Task B: nested task, at Line 7.
- Task C: at Line 16. Pass by value.
- sync variables:
- doneA$: Task A and Root Task.
- doneB$: Task B and Task A.

```
proc outerVarUse( ) {
  var x: int = 10;
  var doneA$: sync bool;
  begin with (ref x) {
    writeln(x++);
    var doneB$: sync bool;
    begin with (ref x){
      writeln(x);
      doneB$ = true;
    }
    writeln(x);
    doneA$ = true;
    doneB$;
  }
}
```



CCFG pruning

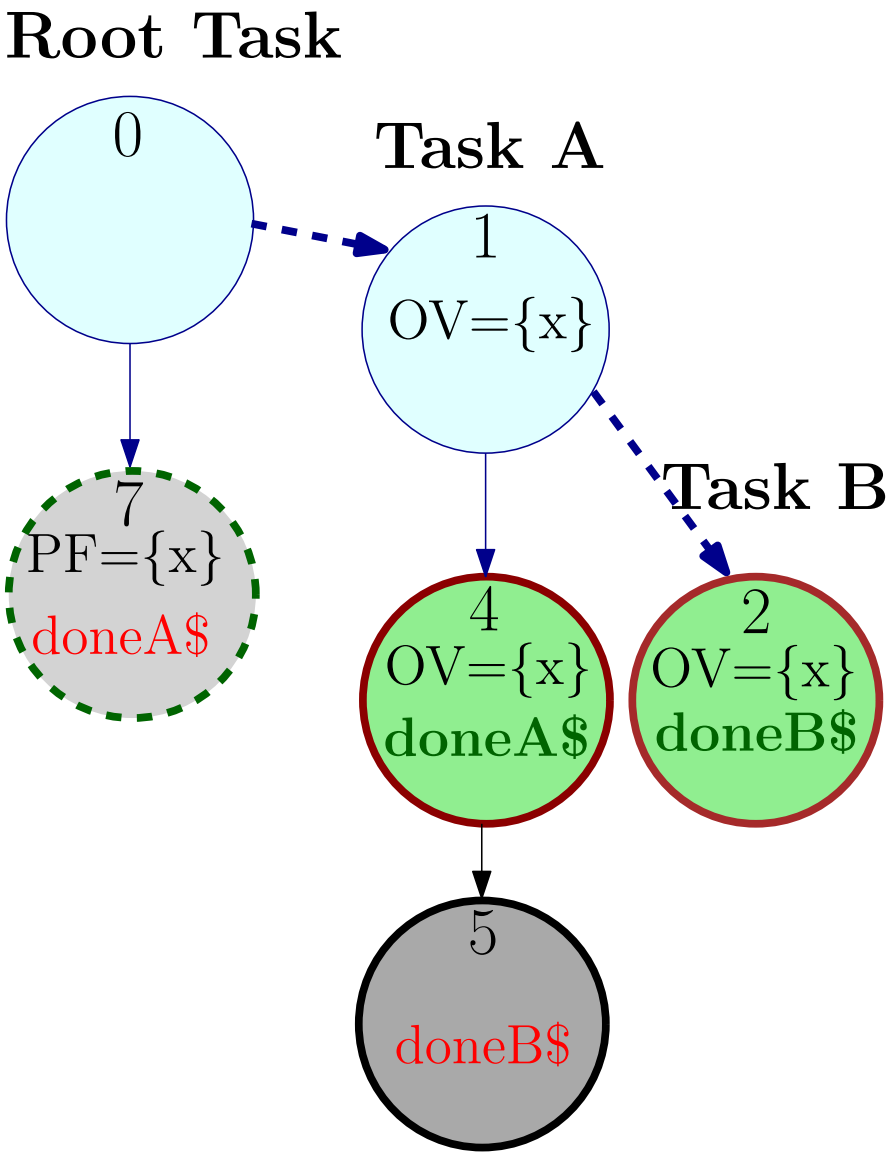
- Empty Nodes at end of each task.
- Safe Tasks:
  1. A `begin` task that does not contain any nested task or refers to any outer variable.
  2. A `begin` task, which is immediately encapsulated by a `sync` statement, provided all nested tasks are safe.
  3. A `begin` task, in which the scope of all external variables accessed by the task is protected by a `sync` block.
  4. A `begin` task, in which all nested tasks are safe and is by itself not referring to any outer variable.

PPS

A PPS:

- Active Sync Node (ASN): Nodes which are next in line to be executed.
- State Table (ST): State of all live synchronization variables.
- Safe access set(SV): A set of outer variable accesses which are safe.
- Live access set (OV): A set of OV accesses which *must have* happened before reaching the current PPS, excluding the set of outer variable accesses in SV.
- SV $\cap$ OV $= \phi$.

Pruned CCFG



PPS 0:

- ASN = {2, 4, **7**}
- State Table

var	state
doneA$	empty
doneB$	empty

- SV = ϕ
- OV = ϕ

Rules

Rule (SINGLE-READ)

A read on a `single` variable is visited if the current state of the variable is full.

Rule (READ)

A read of a `sync` variable can be visited if the current state of the variable is full. The state of the variable is changed to empty.

Rule (WRITE)

A write on `single` or `sync` variable can be visited if the current state of the variable is empty. The state of the variable is changed to full.

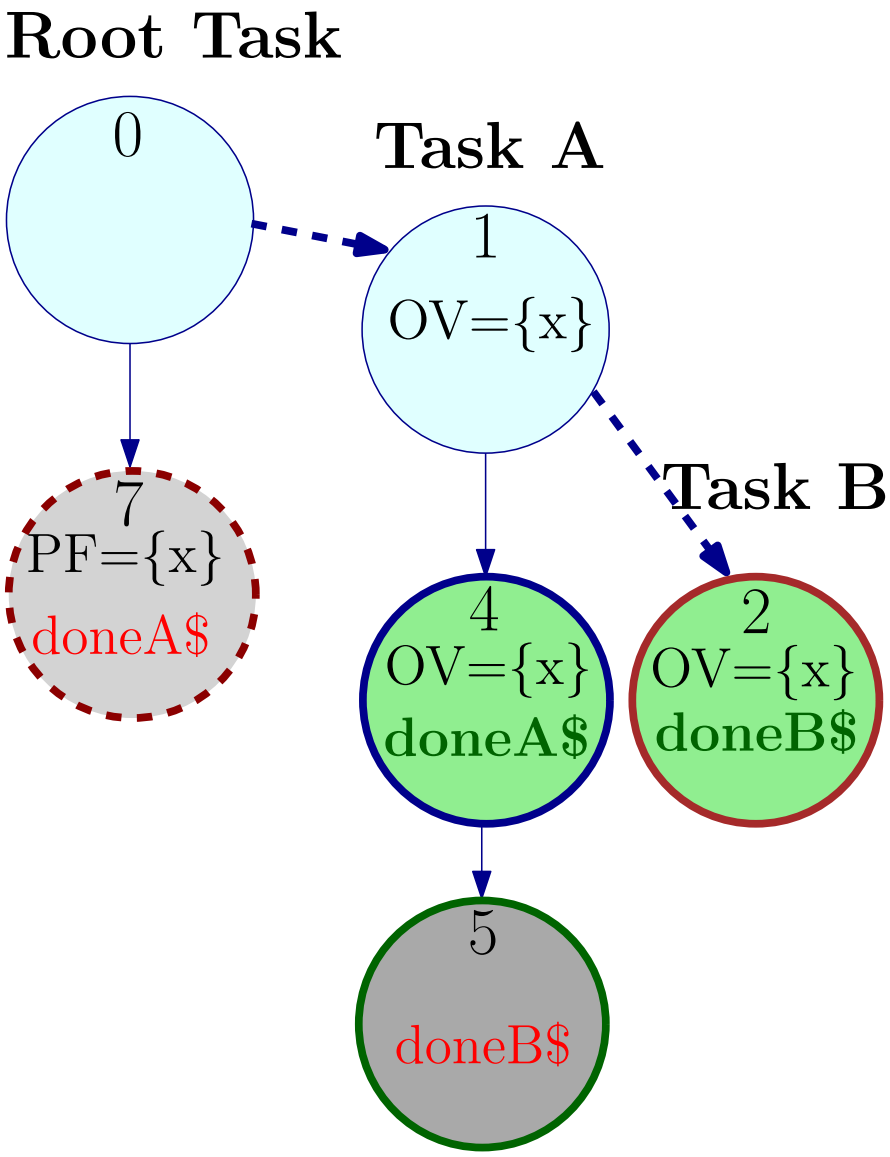
Parallel Frontier

- Defined for OV, `x` : PF(x).
- The last sync node encountered in a path in parent scope.
- Multiple paths could lead to Multiple PF.
- The safety checks limited to PF.

Theorem

A statement that accesses an outer variable x, is potentially unsafe, if there exists an execution path serialization where the corresponding Parallel Frontier node is executed before the statement.

Execute Node 4



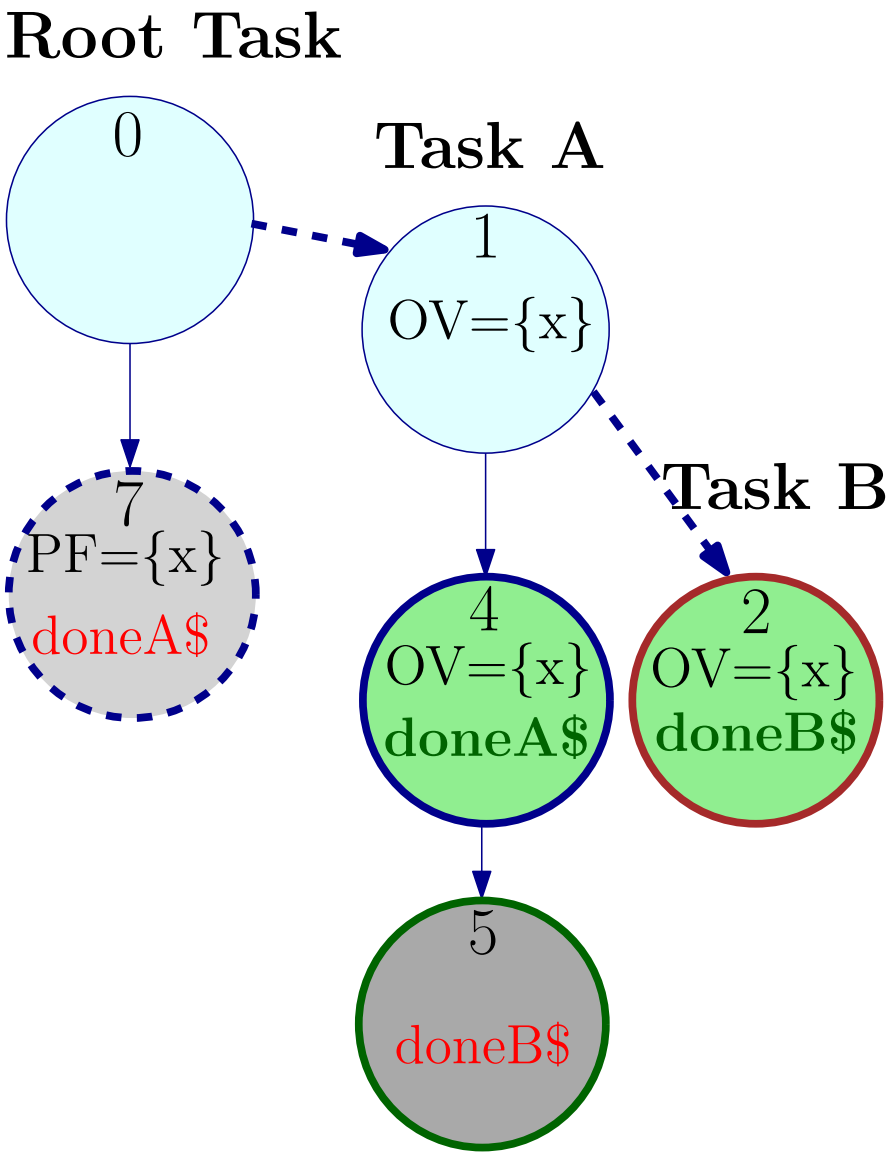
PPS 1:

- ASN = {2, **5**, 7}
- State Table

var	state
doneA$	`full`
doneB$	`empty`

- SV = ϕ
- OV = $\{x_1, x_4\}$

Execute Node 7



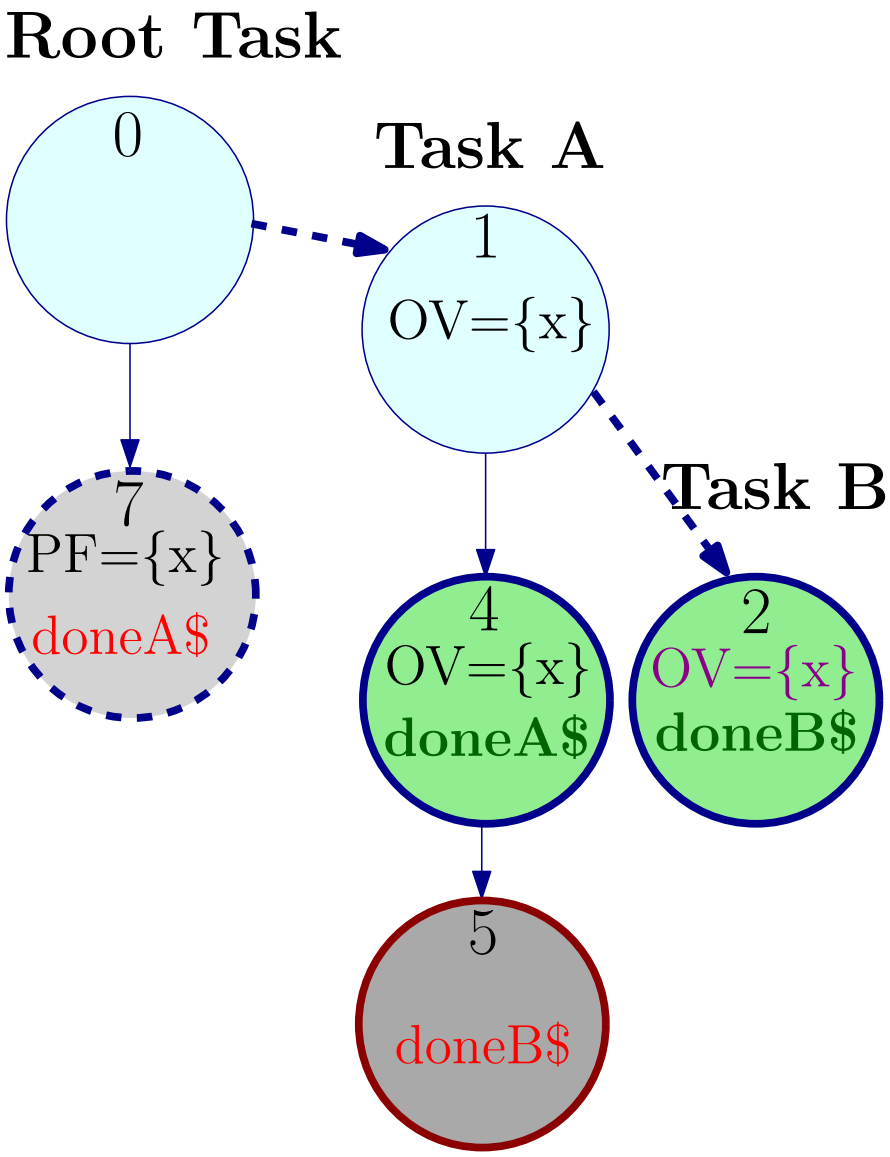
PPS 2:

- ASN = {2, **5**}
- State Table

var	state
doneA$	empty
doneB$	empty

- SV = $\{x_1, x_4\}$
- OV = ϕ

Execute Node 2



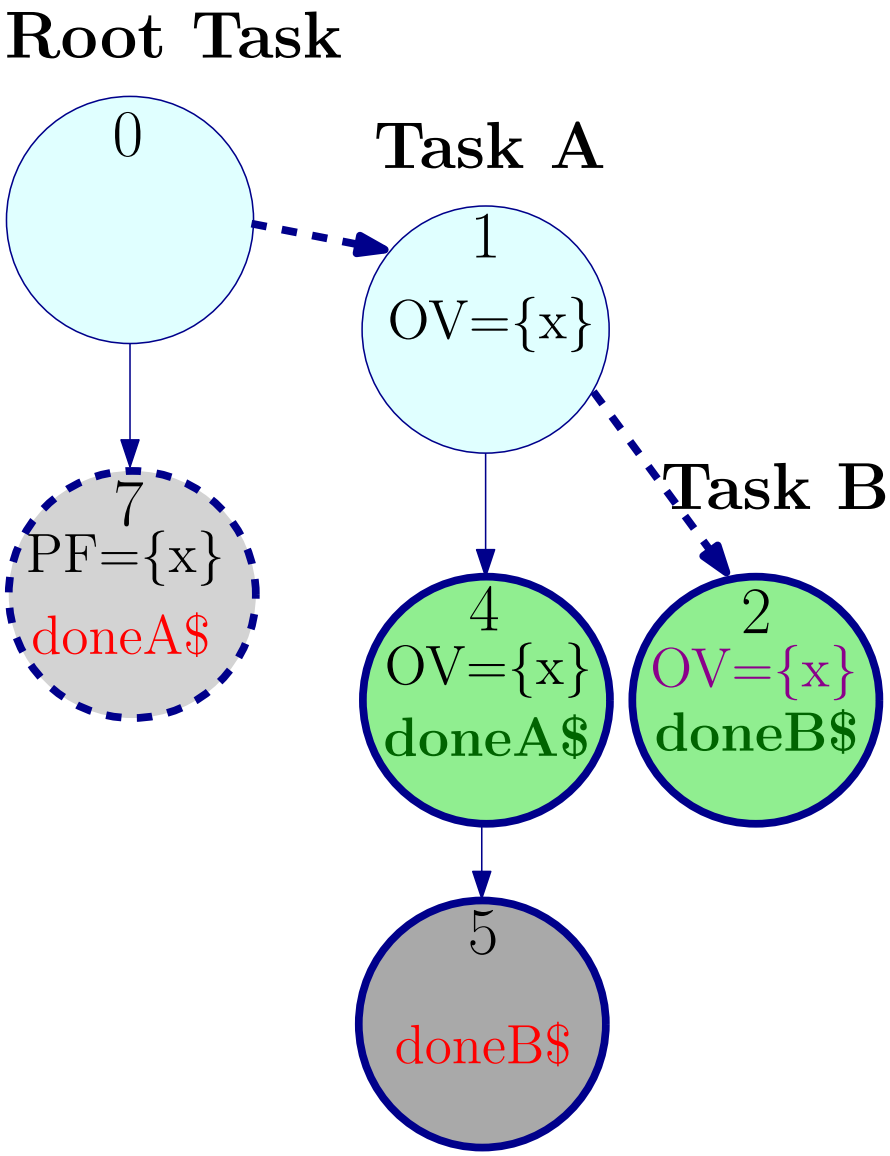
PPS 3:

- ASN = { 5 }
- State Table

var	state
doneA$	empty
doneB$	full

- SV = $\{x_1, x_4\}$
- OV = $\{x_2\}$

Execute Node 5



PPS 4:

- ASN = ϕ
- State Table

var	state
doneA$	`empty`
doneB$	`empty`

- SV = $\{x_1, x_4\}$
- OV = $\{ x_2 \}$
- Report x_2.

Condition Nodes

- Static analysis.
- Both branches are explored separately.
- Loops:
  - Just OV accesses: treated as single node with OV access
  - Not handled:Loops containing `begin` or synchronization node.

Optimization & Limitations

- Merging PPS
  - Identical State table.
  - Equivalent ASN set.
  - SV : $SV_i \cap SV_j$.
  - OV : $OV_i \cup OV_j$.
- Mark already reported accesses.
- Clubbing variable accesses.
- Unsafe $\cup$ safe.
- Not Handled: Non blocking sync events: atomic

Results

Table: Results of running use-after-free check in Chapel version 1.11 test suite.

Total test cases	5127
Test cases with `begin` tasks	218
Test cases with Use-After-Free warnings	38
Number of warnings reported	437
True positives	63
Percentage of true positives	14.4%

Conclusions

- Identify and report potentially dangerous OV accesses to the user.
- Future: Inter procedural
- Future: Loops & recursion.
- Choice of synchronization.
- Child to parent task:
  - `sync`, `single`, atomic integers.
- Broadcast:
  - `single`, atomic integers
- Multiple child tasks:
  - `sync` block, atomic integers.