

Scalable Fault Resilience with Task Replay and Restart

Rupanshu Soi

with Elliott Slaughter, Michael Bauer, and Alex Aiken



Fault resilience is important

- Expected number of faults \propto machine size \times duration
- Job-wall time limits
- Preemption

Checkpointing

- Program state → storage → new execution

Manual checkpointing desideratum

- Identify program data to checkpoint
- Ensure correctness of replay codepath given a checkpoint
- Overlap checkpointing with application work
- Not overload the filesystem with checkpointing I/O
- Be easy to integrate into an existing codebase

Can we do this automatically?

It depends on the programming model...

Automatic checkpointing in MPI

- Entire virtual address space is saved
- Network traffic is drained (blocking)
- Or, messages are logged (non-blocking)

**Task-based programming models enable
efficient automatic checkpointing**

Relight is an automatic checkpointing library for Legion

- Automatically identifies program data to checkpoint
- Uses the same user-facing codepath for replay
- Overlaps checkpointing with application work
- Distributes checkpointing work across the machine
- Generally only requires 2 LoC to be changed
- Completely user-level implementation, no changes to Legion required

Legion Primer

- Computation is organized as tasks
- Data is organized as regions
- Tasks are called in program order
 - But may execute in parallel
 - “Implicitly parallel”

```
data = region(...)  
a = reader1(data)  
b = reader2(data)  
c = writer(data)
```

reader1

reader2

writer

Program order

reader1

reader2

writer

Execution order

Several other systems have a similar programming model

- StarPU (INRIA, France)
- PaRSEC (UT Knoxville)
- Ray (UC Berkeley)
- Python Dask (also a startup)



Relight's insight...

Shimming the (Legion) API is all you need

- Want to observe task calls?
 - Shim the API.
- Want to track live regions in a program?
 - Shim the API.
- Want to insert additional tasks into a program?
 - Shim the API.
- Want to replay from a saved checkpoint?
 - You guessed it.

Scalar example

Replay execution

```
a = ←foo(...)
b = ←bar(...)
```

```
__ckpt()
```

```
c = baz(...)
```

Ckpt file

```
0. val
```

```
1. val
```

```
2. ckpt
```

Original execution

```
a = foo(...)
```

```
b = bar(...)
```

```
__ckpt()
```



```
c = baz(...)
```

Any tasks called before the ckpt immediately return their result during replay

Region example


Replay execution

```
R = region(...)  
writer(R)  
  
__checkpoint()  
  
read_from(R)
```

Ckpt file

```
0. region(...)  
  
1. ckpt
```

Original execution

```
R = region(...)  
writer(R)  
  
__ckpt()  
  
read_from(r)
```

While-loop example

```
R = region(...)
```

```
// 1st iter  
continue(t)  
f(R)  
t += 1  
__checkpoint()
```

```
// 2nd iter  
continue(t)  
f(R)  
...
```



```
R = region(...)
```

```
while continue(t) do  
  f(R)  
  t += 1  
  __ckpt()  
end
```


While-loop example

Replay execution

Ckpt file

Original execution

```
R = region(...)
```

```
// 1st iter
```

```
continue(t)
```

```
f(R)
```

```
t += 1
```

```
__ckpt()
```

```
// 2nd iter
```

```
continue(t)
```

```
f(R)
```

```
...
```

```
0. region(...)
```

```
1. true
```

```
2. ckpt
```

```
R = region(...)
```

```
// 1st iter
```

```
continue(t)
```

```
f(R)
```

```
t += 1
```

```
__ckpt()
```



```
// 2nd iter
```

```
continue(t)
```

```
f(R)
```

```
...
```

Local variables are recomputed during replay

For correctness, Relight requires...

- The top-level task be deterministic
 - “Replay determinism”
- All other tasks may be non-deterministic

FAQs

- What happens to data outside regions—local data on the stack or the heap?
- What if I have a lot of computation outside tasks?
- How is replay correct in a distributed setting?
- How fast is replay?

$$\begin{array}{c}
\text{[Int]} \quad \frac{}{D, E_0, S_0, n \vdash i \rightarrow_s i, E_0, S_0, n} \\
\\
\text{[New]} \quad \frac{l = \text{next_available}(S)}{D, E_0, S_0, n \vdash \text{new} \rightarrow_s i, E_0, S_0[(l, 0) \leftarrow 0, S(l, 1) \leftarrow 0], n} \\
\\
\text{[Rd]} \quad \frac{\begin{array}{c} D, E_0, S_0, n+1 \vdash e_1 \rightarrow_s l_1, E_1, S_1, m \\ D, E_1, S_1, m+1 \vdash e_2 \rightarrow_s v_2, E_2, S_2, p \end{array}}{D, E_0, S_0, n \vdash e_1[e_2] \rightarrow_s S(l_1, v_2), E_2, S_2, p} \\
\\
\text{[Wrt]} \quad \frac{\begin{array}{c} D, E_0, S_0, n+1 \vdash e_1 \rightarrow_s l_1, E_1, S_1, m \\ D, E_1, S_1, m+1 \vdash e_2 \rightarrow_s v_2, E_2, S_2, p \\ D, E_2, S_2, p+1 \vdash e_3 \rightarrow_s v_3, E_3, S_3, q \end{array}}{D, E_0, S_0, n \vdash e_1[e_2] := e_3 \rightarrow_s v_3, E_2, S_3[(l_1, v_2) \leftarrow v_3], q} \\
\\
\text{[Let]} \quad \frac{D, E_0[x \leftarrow 0], S_0, n+1 \vdash e \rightarrow_s v_1, E_1, S_1, m}{D, E_0, S_0, n \vdash \text{let } x \text{ in } e \rightarrow_s v_1, E_1, S_1, m} \\
\\
\text{[Asn]} \quad \frac{D, E_0, S_0, n+1 \vdash e \rightarrow_s v_1, E_1, S_1, m}{D, E_0, S_0, n \vdash x := e \rightarrow_s v_1, E_1[x \leftarrow v_1], S_1, m} \\
\\
\text{[IfT]} \quad \frac{\begin{array}{c} D, E_0, S_0, n+1 \vdash e_1 \rightarrow_s i_1, E_1, S_1, m \\ i_1 \neq 0 \\ D, E_1, S_1, m+1 \vdash e_2 \rightarrow_s v_2, E_2, S_2, p \end{array}}{D, E_0, S_0, n \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightarrow_s v_2, E_2, S_2, p} \\
\\
\text{[IfF]} \quad \frac{\begin{array}{c} D, E_0, S_0, n+1 \vdash e_1 \rightarrow_s i_1, E_1, S_1, m \\ i_1 = 0 \\ D, E_1, S_1, m+1 \vdash e_3 \rightarrow_s v_2, E_2, S_2, p \end{array}}{D, E_0, S_0, n \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightarrow_s v_2, E_2, S_2, p} \\
\\
\text{[WhT]} \quad \frac{\begin{array}{c} D, E_0, S_0, n+1 \vdash e_1 \rightarrow_s i_1, E_1, S_1, m \\ i_1 \neq 0 \\ D, E_1, S_1, m+1 \vdash e_2 \rightarrow_s v_2, E_2, S_2, p \\ D, E_2, S_2, p+1 \vdash \text{while } e_1 \text{ do } e_2 \rightarrow_s v_3, E_3, S_3, q \end{array}}{D, E_0, S_0, n \vdash \text{while } e_1 \text{ do } e_2 \rightarrow_s v_3, E_3, S_3, q} \\
\\
\text{[WhF]} \quad \frac{\begin{array}{c} D, E_0, S_0, n+1 \vdash e_1 \rightarrow_s i_1, E_1, S_1, m \\ i_1 = 0 \end{array}}{D, E_0, S_0, n \vdash \text{while } e_1 \text{ do } e_2 \rightarrow_s 0, E_1, S_1, m} \\
\\
\text{[Seq]} \quad \frac{\begin{array}{c} D, E_0, S_0, n+1 \vdash e_1 \rightarrow_s v_1, E_1, S_1, m \\ D, E_1, S_1, m+1 \vdash e_2 \rightarrow_s v_2, E_2, S_2, p \end{array}}{D, E_0, S_0, n \vdash e_1; e_2 \rightarrow_s v_2, E_2, S_2, p}
\end{array}$$

Lemma III.1 (Fast Forward). Consider e where D where e does not contain new, reads or writes of pairs, or checkpoint. Let P be a proof that $D, E_0, S_0, n \vdash e \rightarrow_s v, E_1, S_1, m$, and let C be P 's checkpoint environment. Then:

$$D, C, E_0, _, n \vdash e \rightarrow_r v, E_1, _, m$$

Theorem III.2 (Replay). Without loss of generality, consider a program e ; checkpoint where D where e does not contain new, reads or writes of pairs, or checkpoint. Let P be a proof that $D, E_0, S_0, n \vdash e; \text{checkpoint} \rightarrow_s 0, E_1, S_1, m$, and let C be the checkpoint environment for P . Then

$$D, C, E_0, S_0, n \vdash e; \text{checkpoint} \rightarrow_r 0, E_1, S_1, m$$

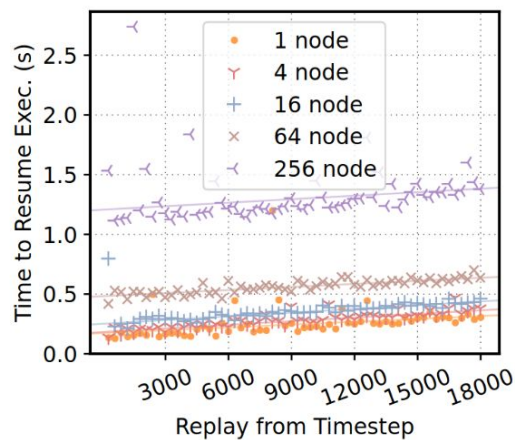


Performance results

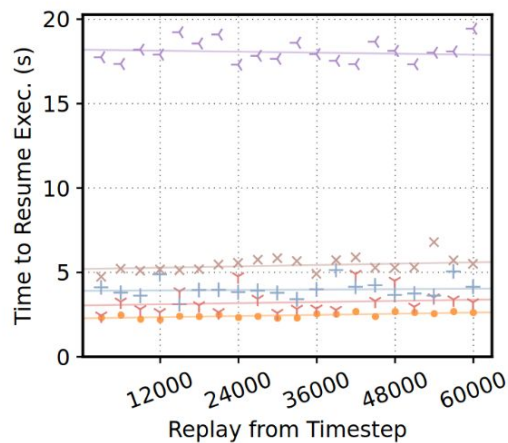
Three benchmarks

- Circuit
- Stencil
- Pennant

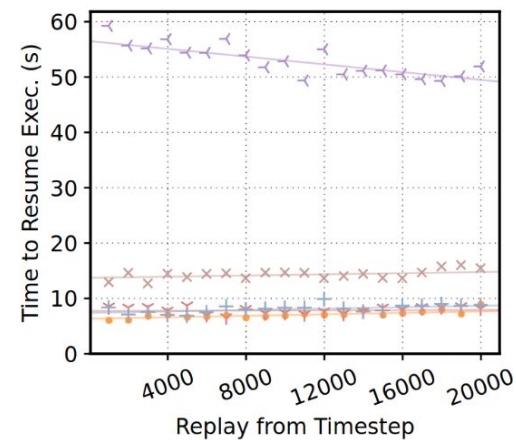
Relight can replay 10k timesteps/sec



(a) Circuit

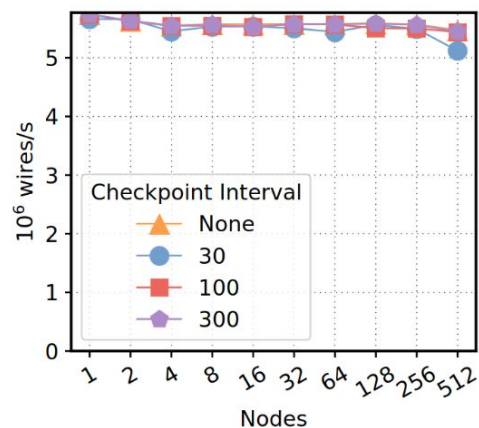


(b) Stencil

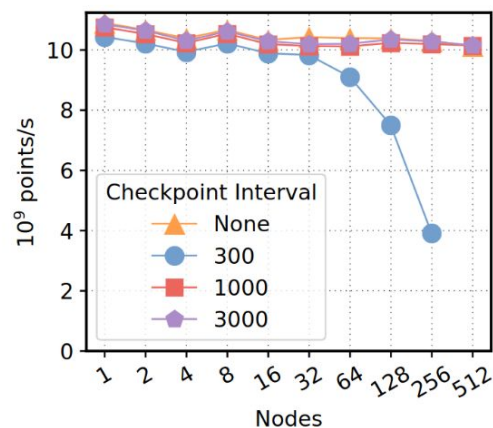


(c) Pennant (Regent)

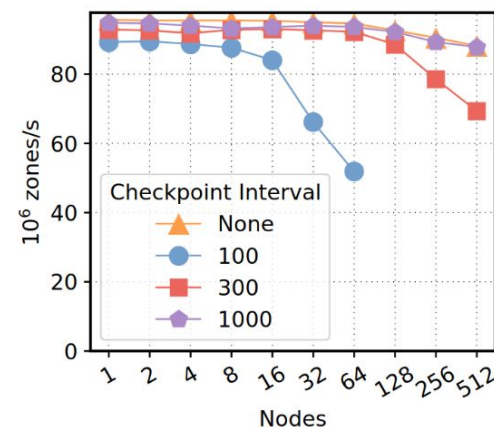
Relight weak scales to 512 nodes



(a) Circuit



(b) Stencil



(c) Pennant (Regent)

 **Switching gears...**

Hard vs. soft errors

- Hard errors tend to kill your program
 - CPU, memory, NIC faults
 - Power outages
- Soft errors make your program incorrect, but do not kill it
 - Silent data corruptions

Cores that don't count

Peter H. Hochschild
Paul Turner
Jeffrey C. Mogul
Google
Sunnyvale, CA, US

Rama Govindaraju
Parthasarathy
Ranganathan
Google
Sunnyvale, CA, US

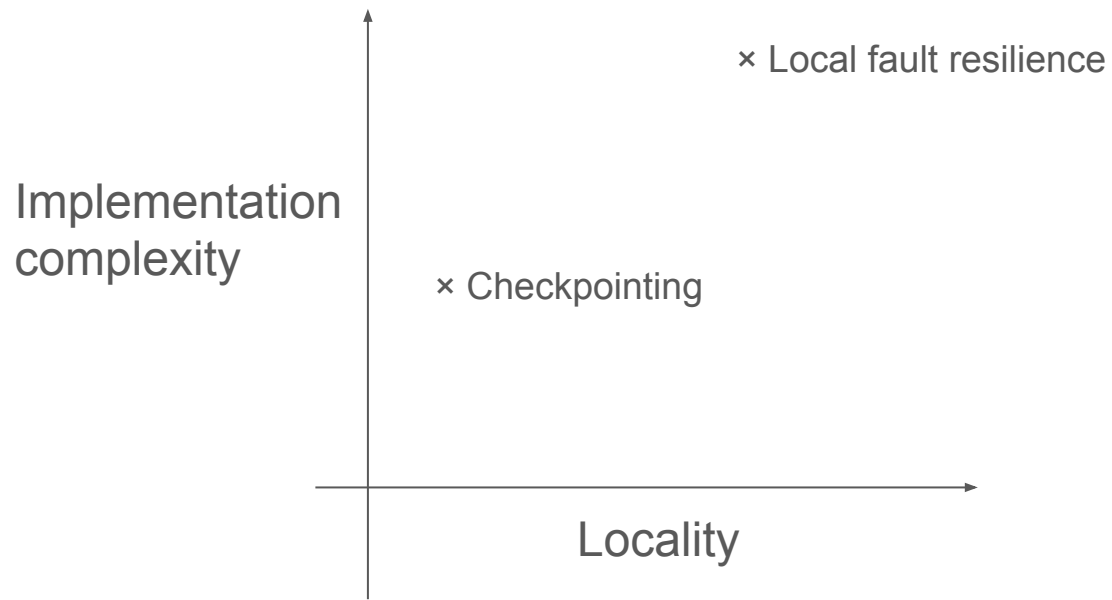
David E. Culler
Amin Vahdat
Google
Sunnyvale, CA, US

Global vs. local fault resilience

- If a point failure affects the whole (distributed) application, your fault resilience mechanism is global.
 - Otherwise, local.
- Relight is global.

Local fault resilience is desirable for two reasons:

- Better utilizes hardware cycles
- Reduces IO required to recover from failures



Local fault resilience in MPI

- Via ULFM, a set of extensions
- Developers must write their own local fault resilience schemes

Implementing local fault resilience in MPI is challenging

- Recovery must be specialized for each application
- Lack of guarantees
- Applications must manage MPI's state after a failure

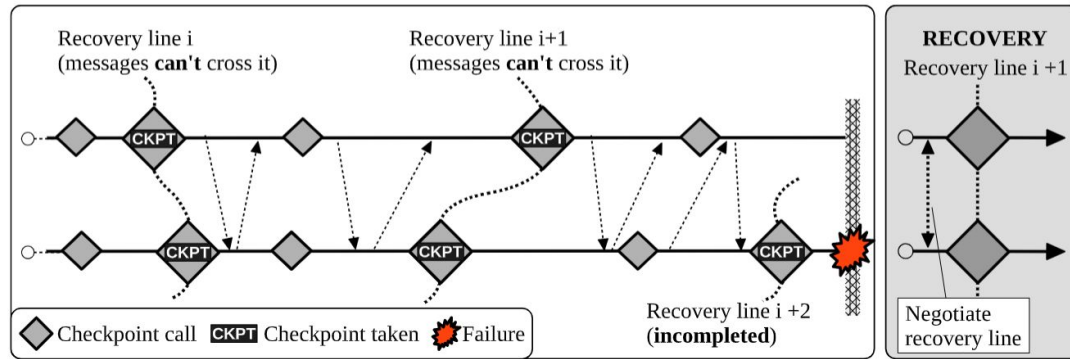


Fig. 1. Spatial coordination protocol.

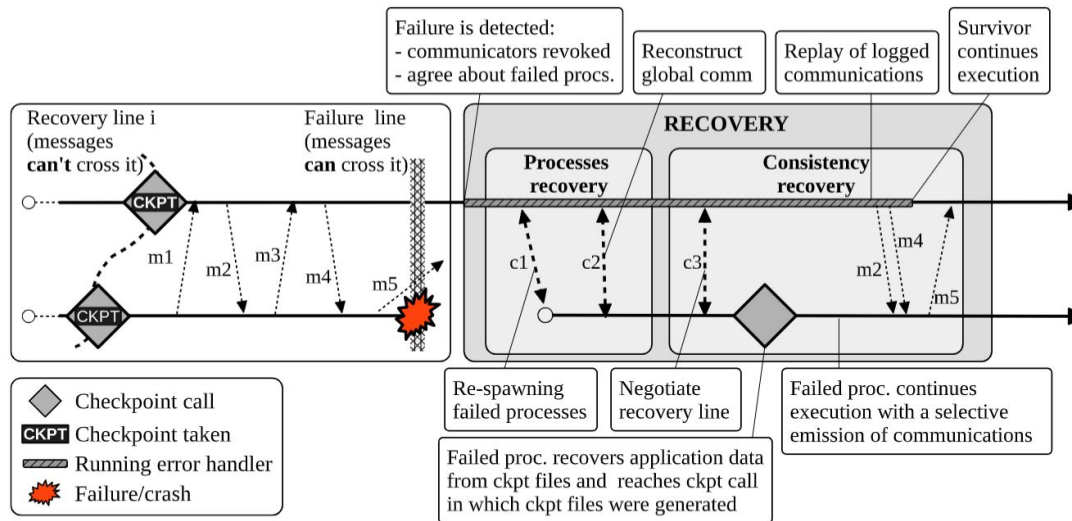


Fig. 2. Local rollback.

**Local resilience against hard errors is
fundamentally difficult**

Can we do something better for soft errors?

Restartable tasks

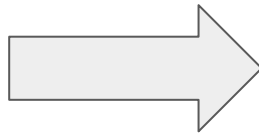
- Simple idea: if a soft error occurs, re-execute the task until it succeeds
- Easily expressible in Legion

To make a task restartable...

1. Outline it into a wrapper.
2. Save a copy of its arguments.
3. Run it.
4. Check for errors. If there were no errors, goto 7.
5. Restore its arguments.
6. Goto 3.
7. Return its result.

Transformation for a single task

```
task toplevel()  
  r = region(...)  
  compute(r)  
  ...  
end
```

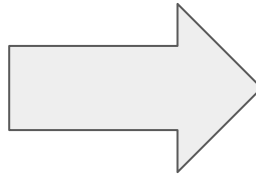


```
task toplevel()  
  r = region(...)  
  wrapper(r)  
  ...  
end
```

```
task wrapper(r)  
  rc = region(...)  
  copy(r, rc)  
  while true do  
    compute(r)  
    -- block and check for errors  
    if not error then return end  
    copy(rc, r)  
  end  
end
```

Transformation for a timestep loop

```
for t = 0, T do  
  compute(t, r)  
end
```



```
task wrapper(r)  
  ...  
  for t = 0, T/2 do  
    compute(t, r)  
  end  
  ...  
end
```

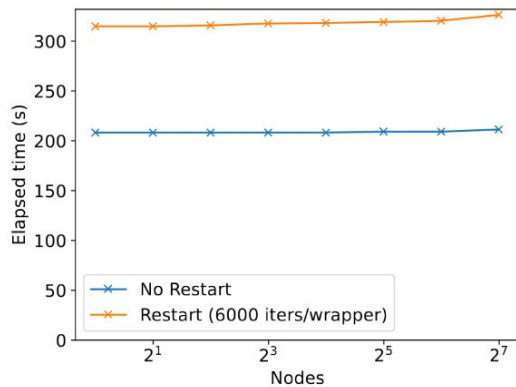
```
task wrapper(r)  
  ...  
  for t = T/2, T do  
    compute(t, r)  
  end  
  ...  
end
```

 **Performance results (encore)**

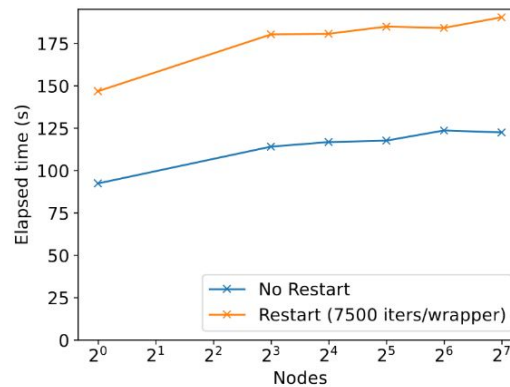
Experimental setup

- Same three apps
- Split every timestep loop into two wrapper tasks
- Simulate exactly one failure

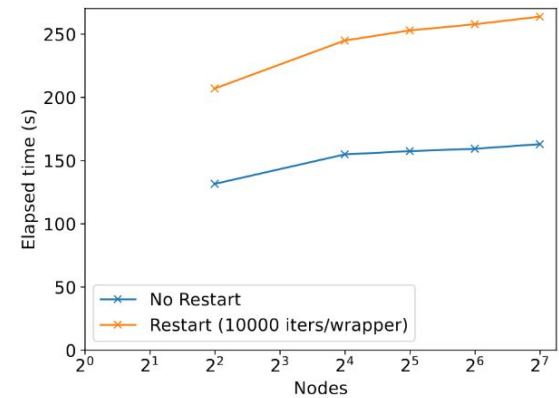
Restartable tasks are scalable



(a) Circuit

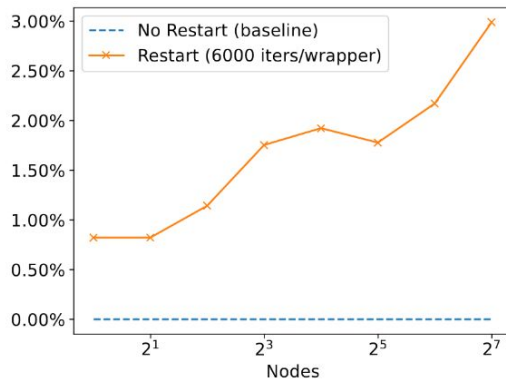


(b) Stencil

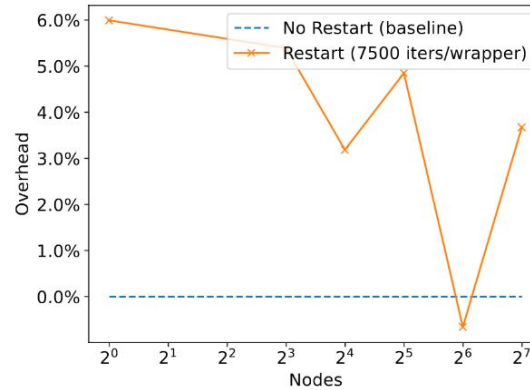


(c) Pennant

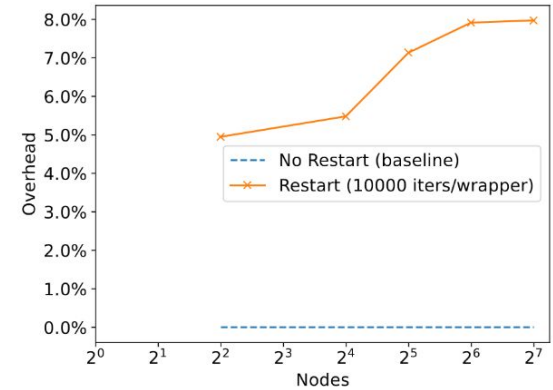
Restartable tasks are low overhead



(a) Circuit

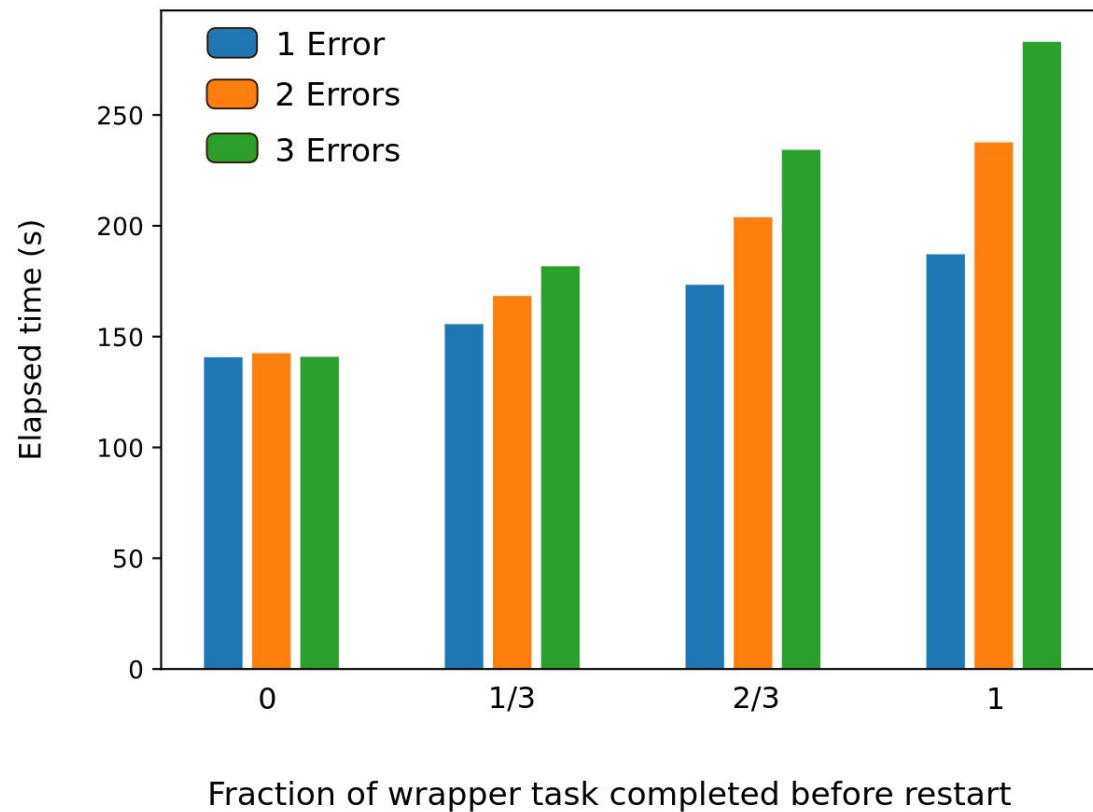


(b) Stencil



(c) Pennant

Restartable tasks can handle multiple failures with low overhead



Recap

- Fault resilience is an important and challenging problem for distributed applications
- Fault resilience can be simplified in a task-based programming model
- Relight provides global fault resilience via checkpointing
- Restartable tasks provide local fault resilience

Thanks! Learn more at:

legion.stanford.edu

