Leveraging a Task-based Asynchronous Dataflow Substrate for Efficient and Scalable Resiliency

Omer Subasi, Javier Arias, Jesus Labarta, Osman Unsal and Adrian Cristal

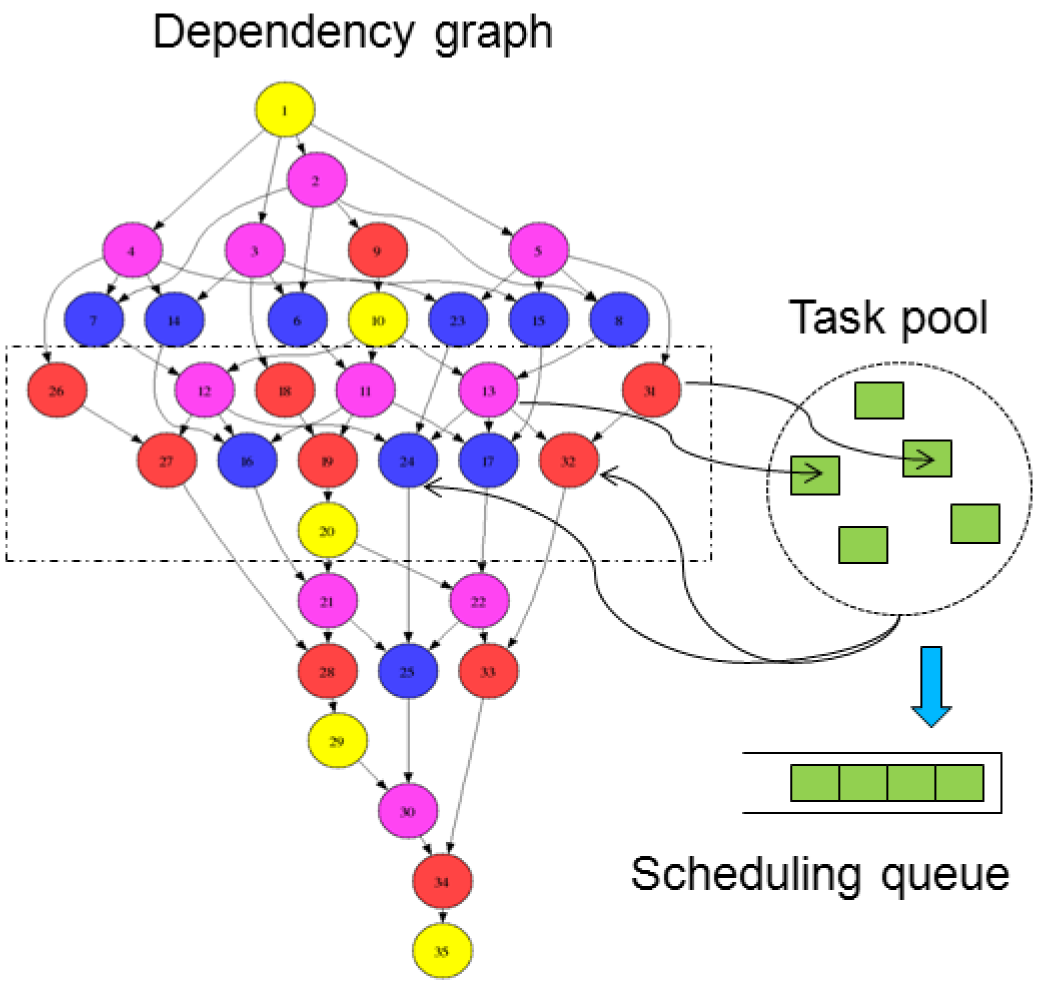
Barcelona Supercomputing Center

Outline

- Background
  - OmpSs and Nanos
  - Target fault models
- Advantage of our substrate for resilience
- Our proposed solutions
  - Checkpoint restart (detected uncorrected errors (DUE))
  - Task redundancy (silent data corruption (SDC + DUE))
  - Partial redundancy (SDC + DUE)

OmpSs and Nanos

- **OmpSs**– Task based programming model (OpenMP derivative)
  - Task - Once started can execute to completion independent of other tasks
  - Programmer supplies directionality annotations on tasks arguments
- **Nanos** – Runtime supporting OmpSs
  - Dataflow-based - if a task is “ready”, it will be scheduled to a processing element
  - Constructs dataflow graph dynamically from task dependencies



Our Fault Model From Hardware Failure Taxonomy (by Symptom)

- Undetected
  - Benign (masked faults)
  - Silent Data Corruption
- Hardware Detected
  - Hardware Corrected
  - Hardware Uncorrected
    - Detected Uncorrected Error (DUE)

- Background
  - OmpSs and Nanos
  - Target fault models
- Advantage of our substrate for resilience
- Our proposed solutions
  - Checkpoint restart (detected uncorrected errors (DUE))
  - Task redundancy (silent data corruption (SDC + DUE))
  - Partial redundancy (SDC + DUE)

Potential and Advantages of OmpSs and Nanos for resilience

- All task inputs and outputs known
  - It is relatively easy and efficient to checkpoint the inouts of tasks
    - facilitates recovery
  - It is relatively easy to replicate tasks and to check the outputs of the replicated tasks
    - facilitates fault detection
- Nanos tasks executed asynchronously and parallel
  - Inherently easy to implement asynchronous and parallel fault tolerance features
- Tasks deferred only because of their dependencies
  - Any redundancy (checkpointing, reissuing) defers only part of execution dependent on it
  - Thus, more efficient than mechanisms subject to fork/join parallelism and than synchronous approaches

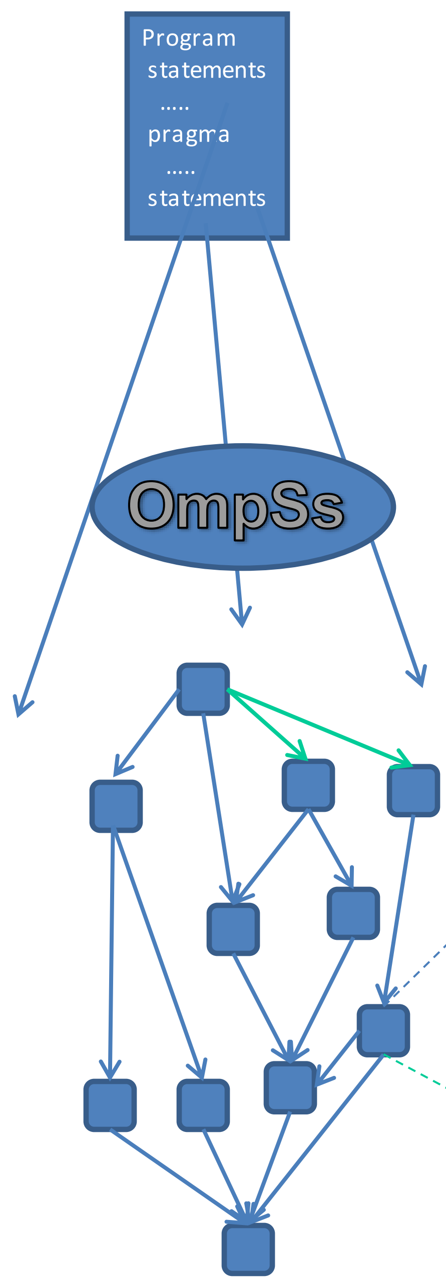
Advantages of OmpSs and Nanos for resilience (Cont.)

- Efficient incremental checkpointing schemes easily employed
  - since we only need to checkpoint the inputs of a task
- All dependencies among tasks are known, which
  - facilitates the development of runtime heuristics which can determine which tasks are more reliability-critical
  - facilitates partial redundancy
    - Both programmer specified & automated and adaptive
- The only state that propagates out of the task is through the outputs and inouts:
  - straightforward to limit error propagation, and to determine the source of an error

Outline

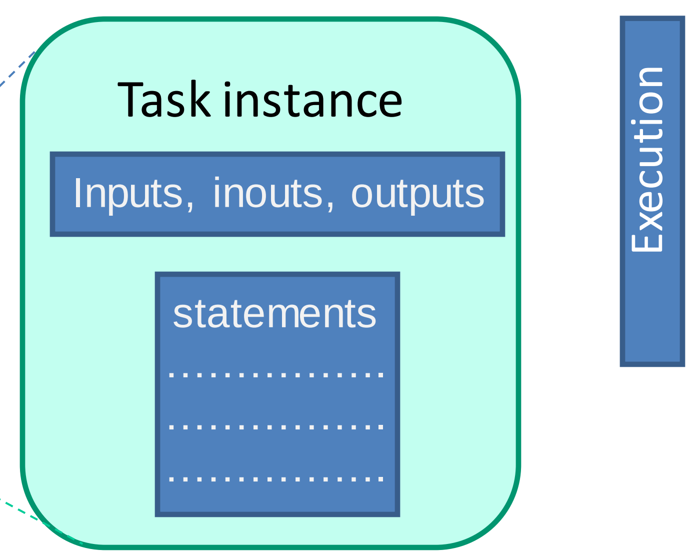
- Background
  - OmpSs and Nanos
  - Target fault models
- Advantage of our substrate for resilience
- Our proposed solutions
  - Checkpoint restart (detected uncorrected errors (DUE))
  - Task redundancy (silent data corruption (SDC + DUE))
  - Partial redundancy (SDC + DUE)

Checkpoint Restart Implementation



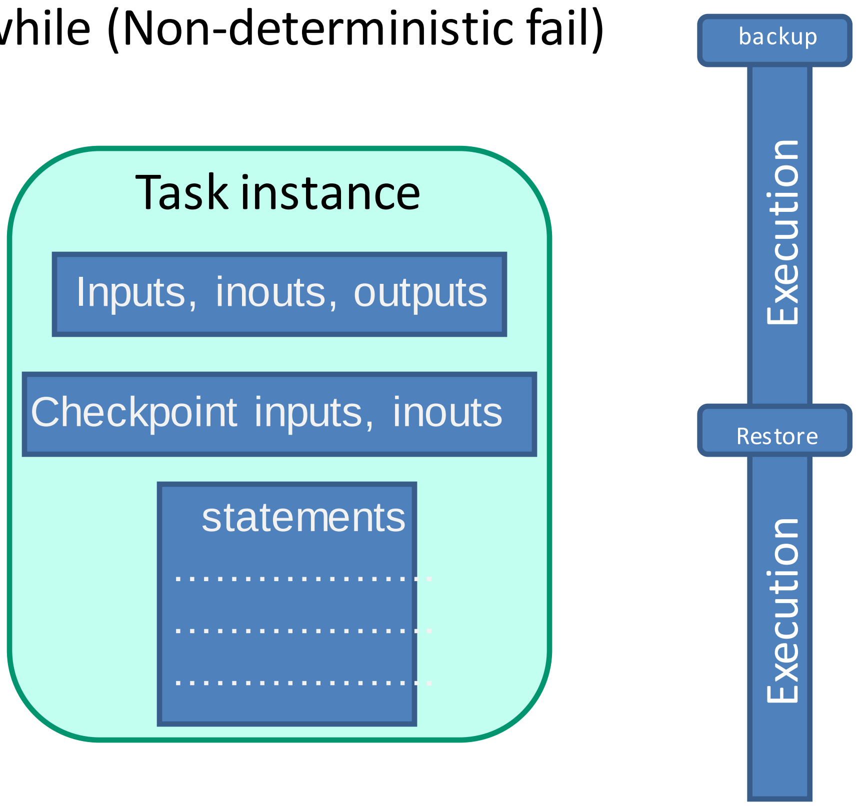
Regular version

-Satisfy dependences and wait for resources availability
- Run*



Checkpoint Restart version

- Satisfy dependences and wait for resources availability
- Checkpoint inputs, inouts to checkpoint structure
- do {
    if(fail) Restore checkpoint structure
    Run*;
  } while (Non-deterministic fail)

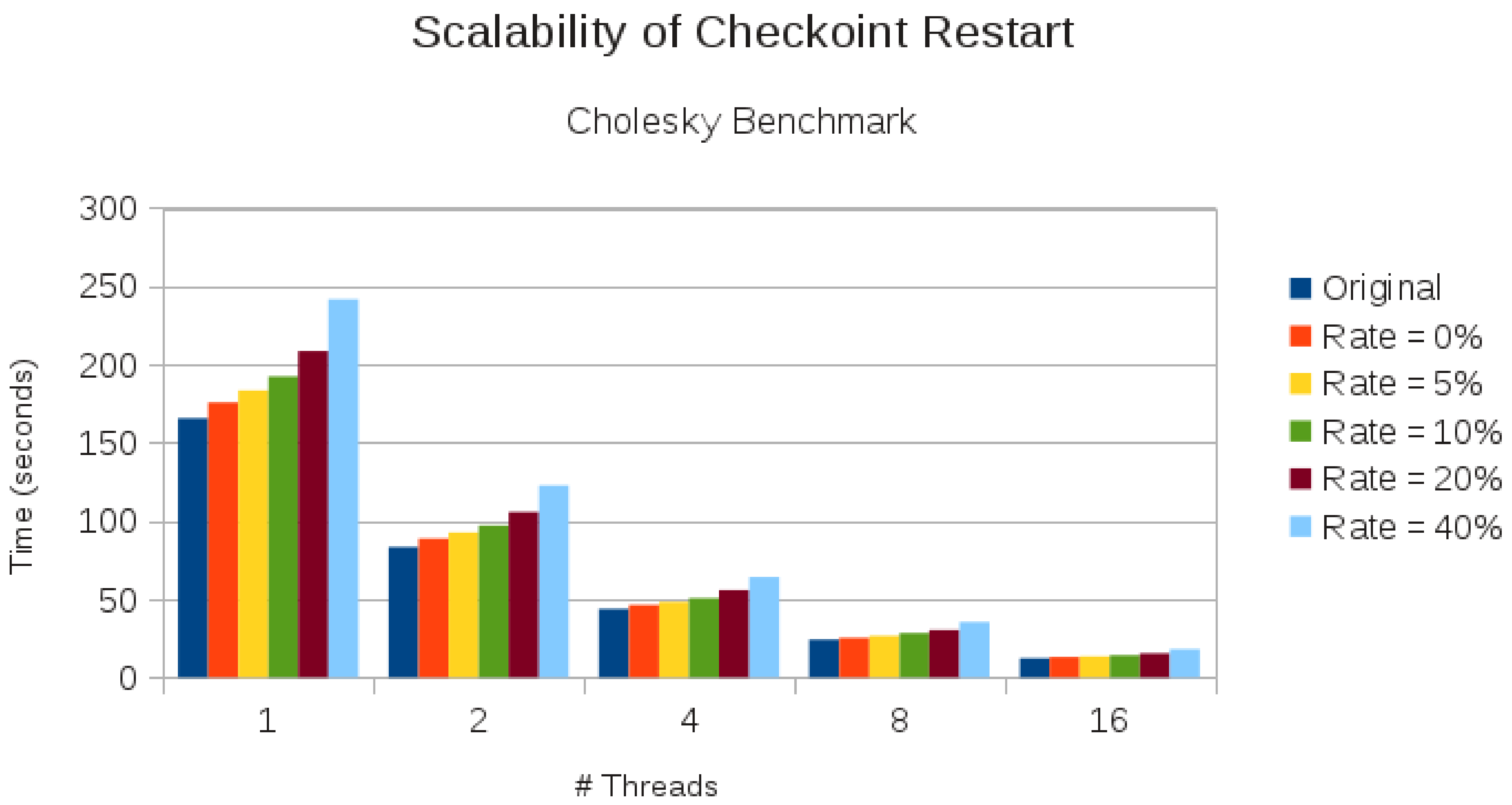


*Instance of task is run in parallel within the rest of the task instances

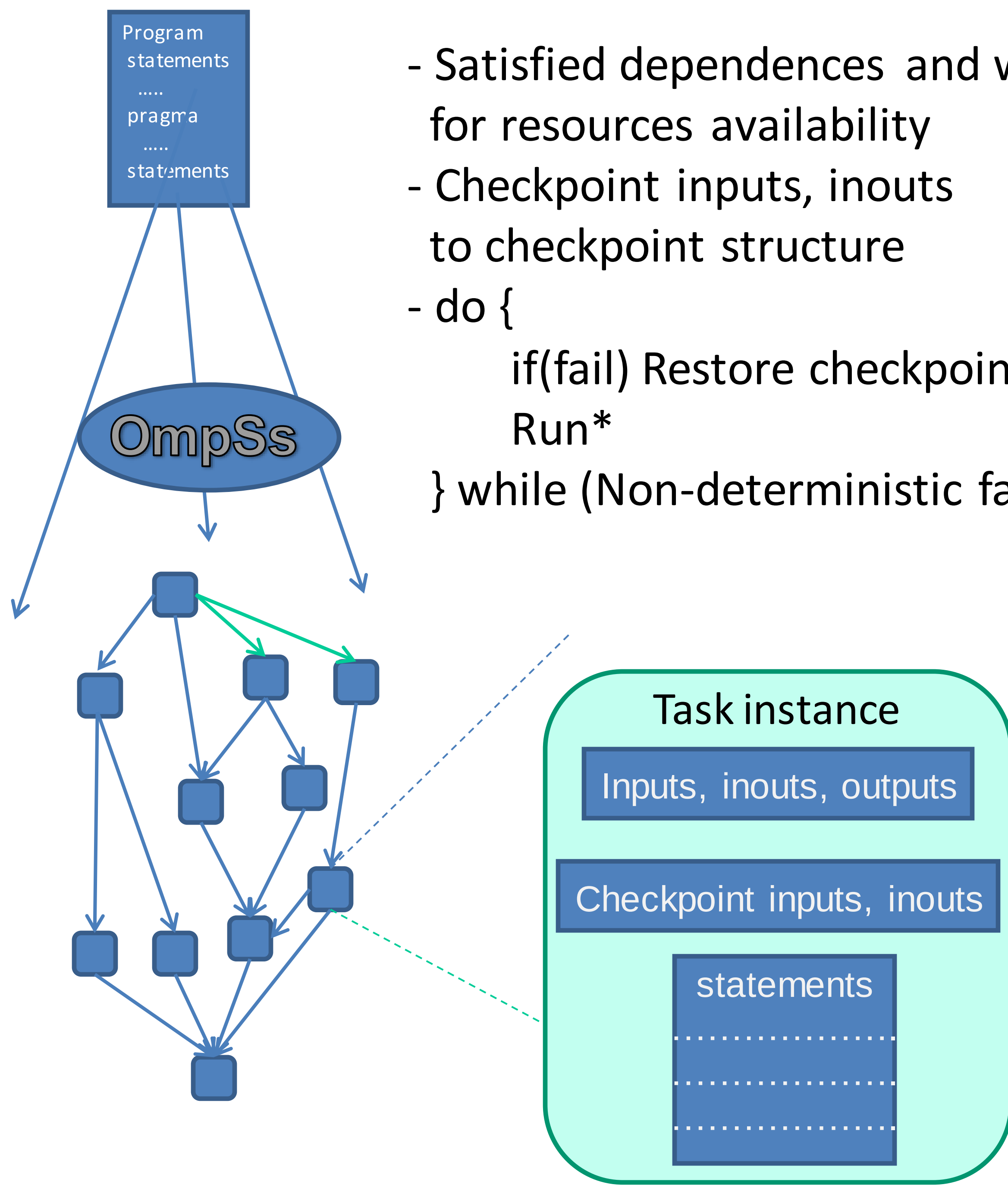
Experimental Setup and Benchmarks

- Experiments run in MareNostrum III (Sandy Bridge)
- Benchmarks
  - Cholesky
    - Matrix size 16384x16384 and block size 512x512
  - Sparse LU
    - Matrix size 6400x6400, block size 100x100
  - Fast Fourier Transform
    - Array size 16384x16384, block size 128

Checkpoint Restart: Cholesky Scalability



Checkpoint Restart Implementation: Singleton backups



Checkpoint Restart version

- Satisfied dependences and wait for resources availability
- Checkpoint inputs, inouts to checkpoint structure
- do {

    if(fail) Restore checkpoint structure

    Run*

  } while (Non-deterministic fail)

CR version and singleton backups

-Satisfy dependences and wait for resources availability

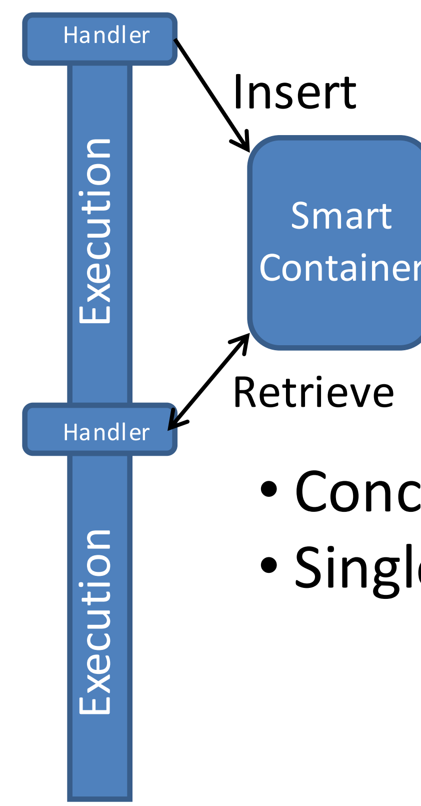
-Checkpoint inputs, inouts using the concurrent backup handler

-do {

    if(fail) Restore checkpoint using the handler

    Run*

  } while (Non-deterministic fail)



- Concurrent HashMap
- Single Copies

*Instance of task is run in parallel within the rest of the task instances

Singleton Mechanism Results

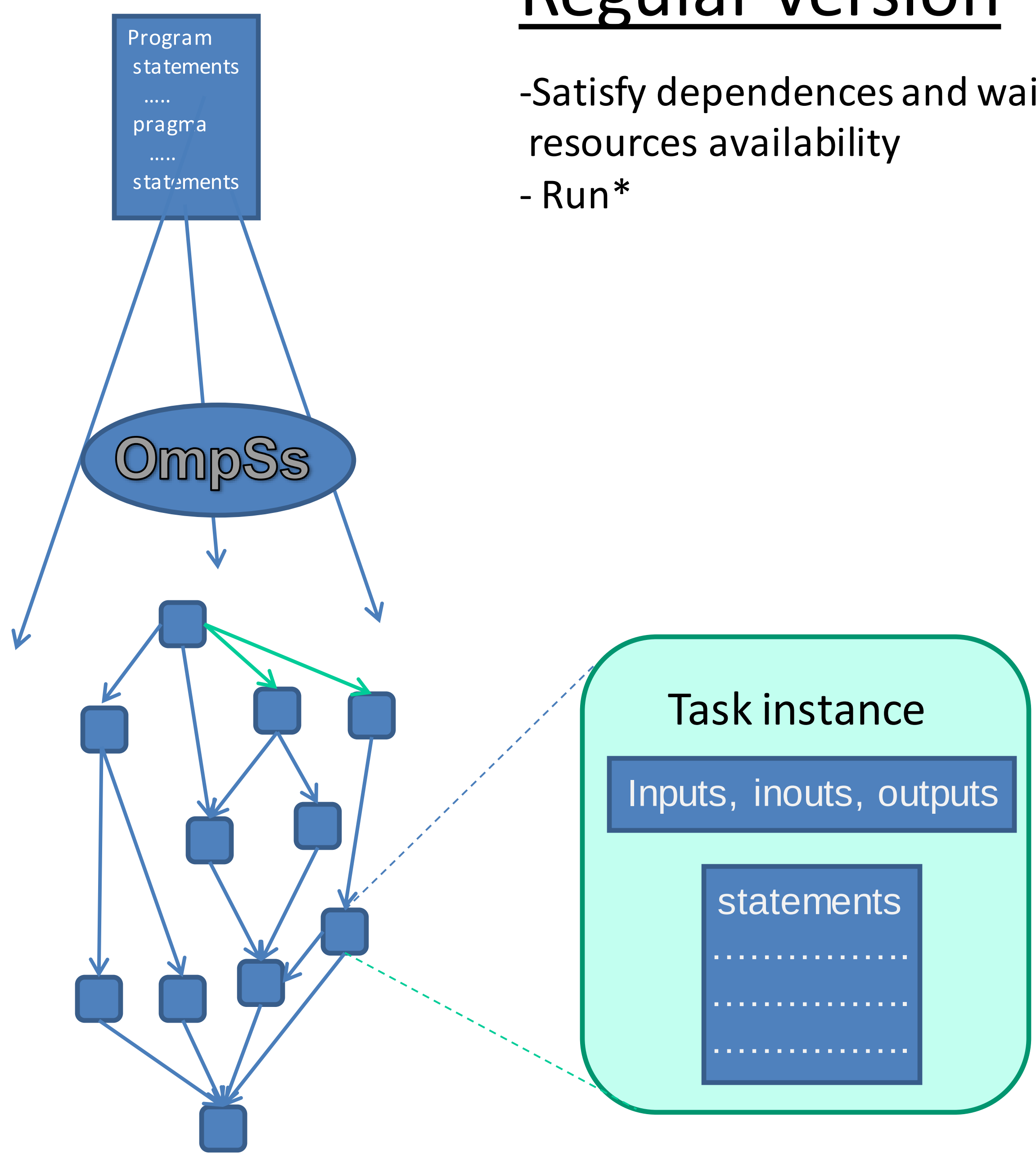
	SparseLU	Cholesky	FFT
Checkpoint/Restart: Checkpoint Overhead to Fault-free Exe.Time	2%	6%	9%
Singleton: Checkpoint Overhead to Fault-free Exe.Time	0.2%	1%	7%

	SparseLU	Cholesky	FFT
Gain in X in memory usage	31x	32x	2x

Outline

- Background
  - OmpSs and Nanos
  - Target fault models
- Advantage of our substrate for resilience
- Our proposed solutions
  - Checkpoint restart (detected uncorrected errors (DUE))
  - Task redundancy (silent data corruption (SDC + DUE))
  - Partial redundancy (SDC + DUE)

Task Redundancy: Current Implementation

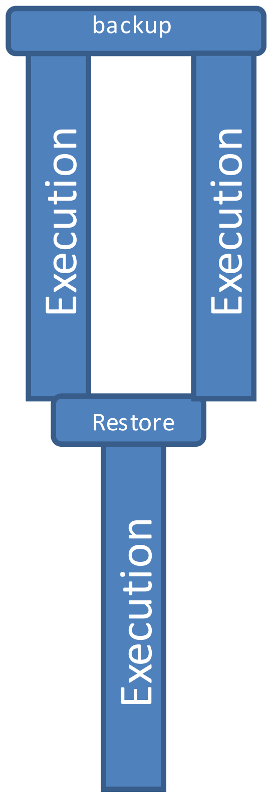


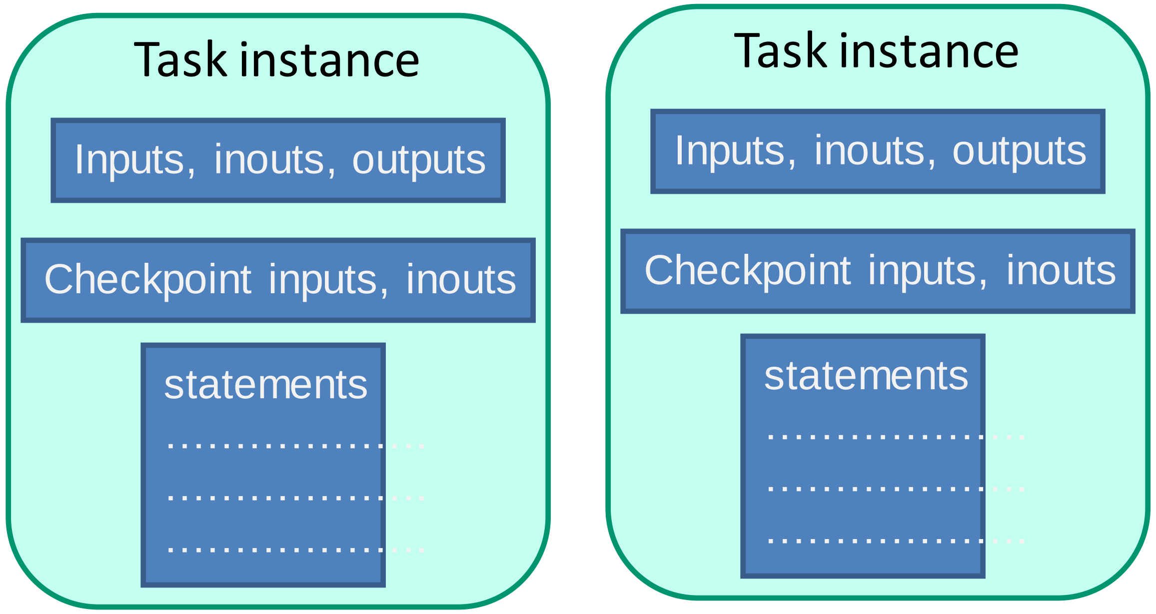
Regular version

-Satisfy dependences and wait for resources availability
- Run*

TR version

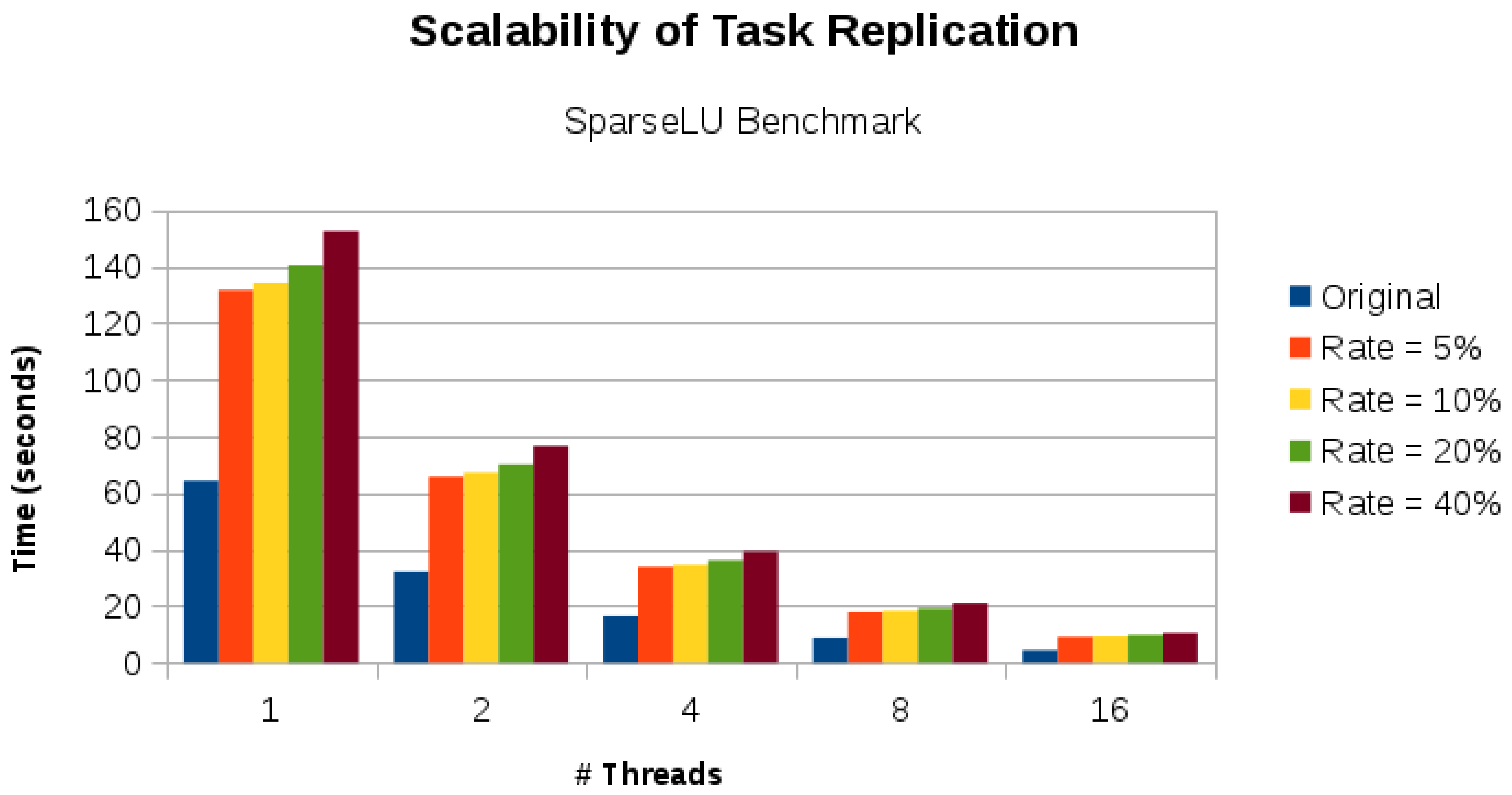
- Satisfy dependences and wait for resources availability
- Checkpoint inputs, inouts using the concurrent backup handler
- Run* and Run parallel duplicated-run;
- If(different results)
  Restore checkpoint using the handler
  Rerun one instance
}





*Instance of task is run in parallel within the rest of the task instances

Task Redundancy: SparseLU Scalability



Outline

- Background
  - OmpSs and Nanos
  - Target fault models
- Advantage of our substrate for resilience
- Our proposed solutions
  - Checkpoint restart (detected uncorrected errors (DUE))
  - Task redundancy (silent data corruption (SDC + DUE))
  - Partial redundancy (SDC + DUE)

Partial Redundancy

- Partial task replication in Nanos Runtime
  - Automated replication
  - User-specified replication
  - Comparison to random task selection

Automated Partial Redundancy

- Simple Runtime Heuristic:
  - Only replicate reliability-critical tasks
  - Select t for replication if risk(t) > ***global task risk***
  - Global task risk = 0.7*global task risk + 0.3*risk(t)
  - risk(t) = (i + o)^2 + s

- i: # inputs of the task t
- o: # outputs of the task t
- s: # successors of the task t

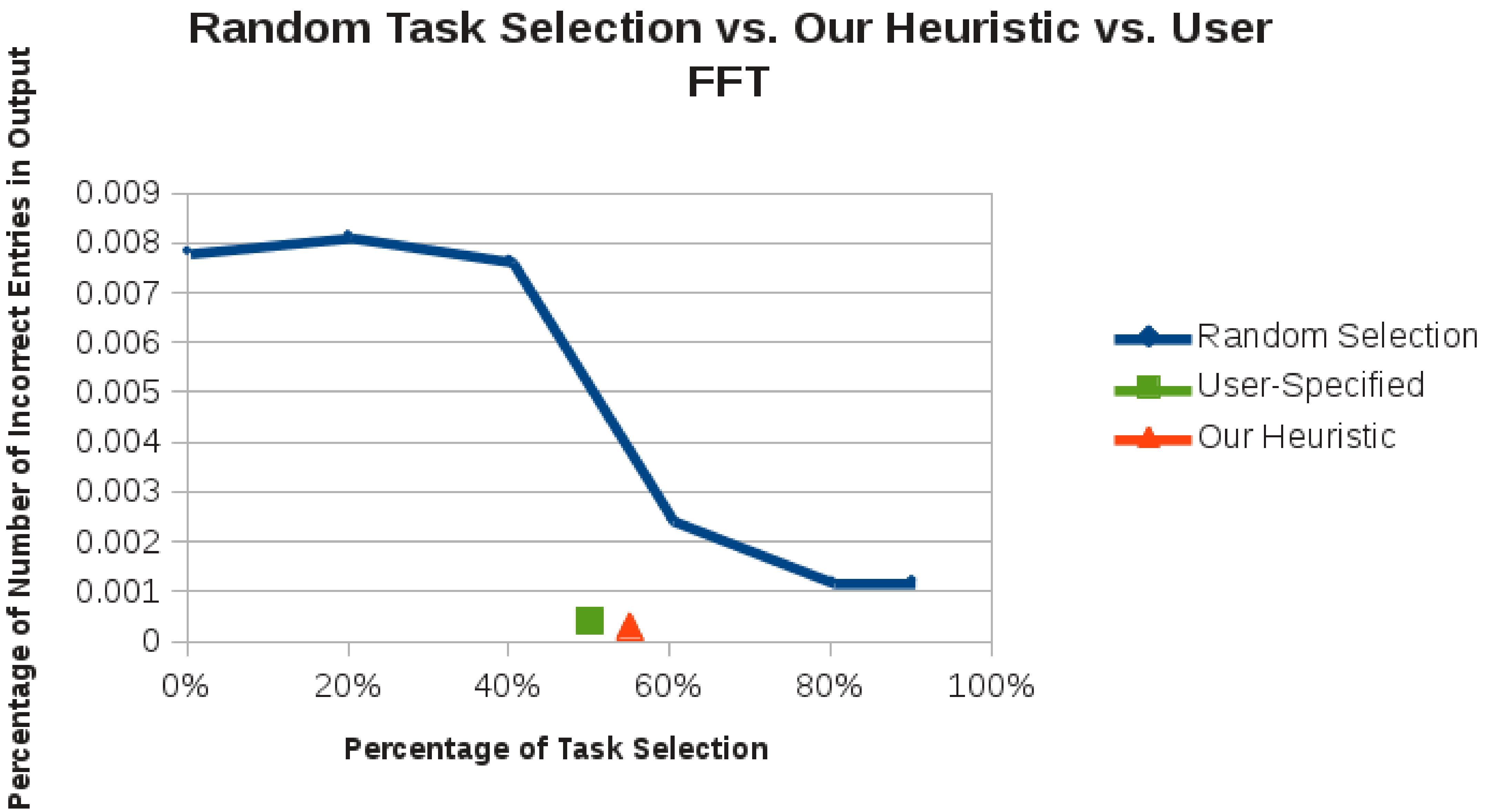
Rationale for Risk Definition

- To capture the memory space used by the tasks as well as dependency among tasks
- Number of inputs/outputs is good hint for memory space usage
- The more a task has successors, the more the severe effect of not protecting the task in terms of error propagation to the successors

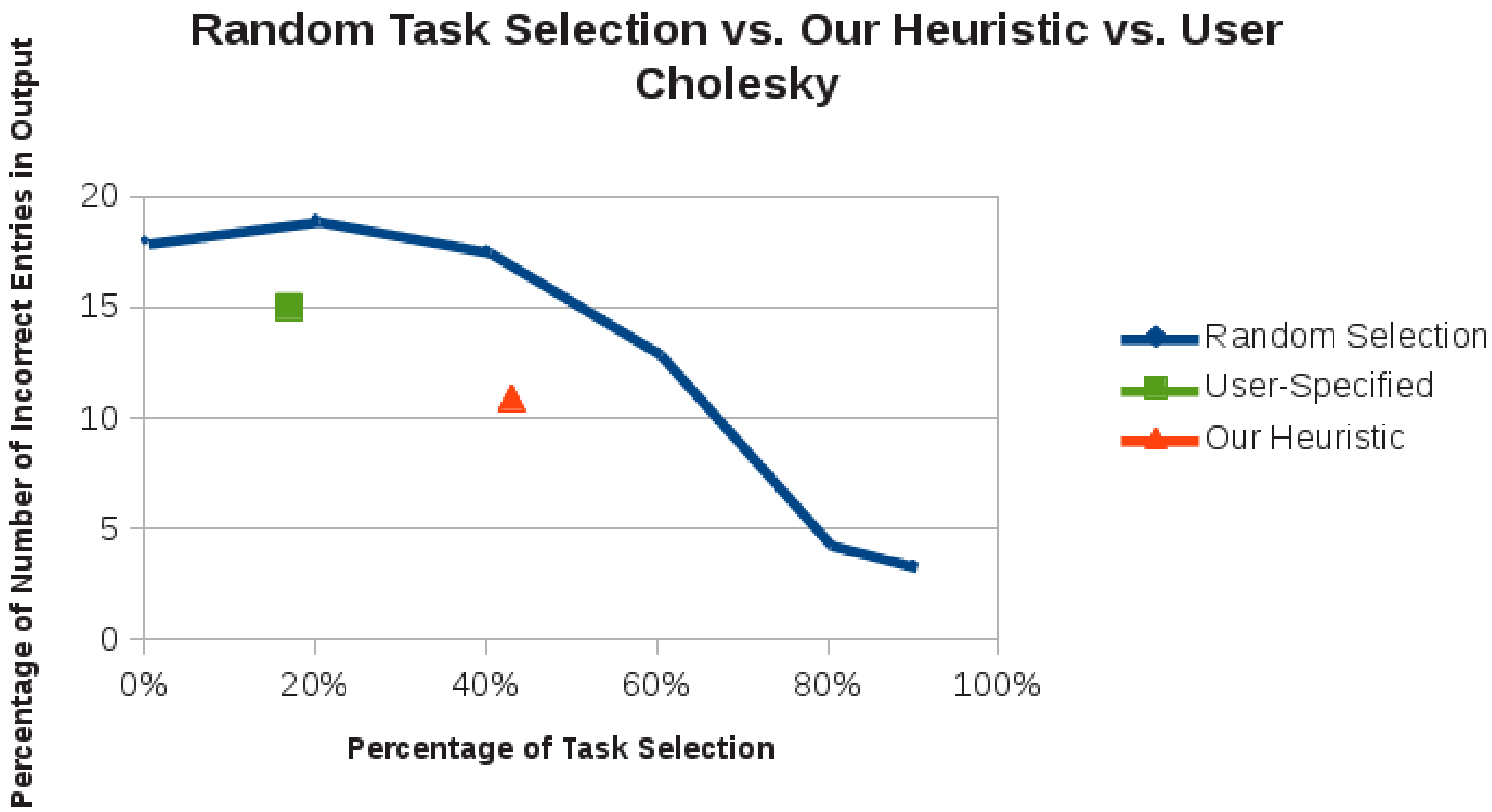
User-specified Selective Task Replication

- User specifies which tasks to protect for runtime
- FTT (early tasks)
  - As being a iterative refinement algorithm, early stages likely to be more reliability-critical
- Cholesky (diagonal tasks)
  - As these blocks are utilized during all subsequent phases of the algorithm, directly or indirectly
- SparseLU (no clear distinction between tasks but can protect early tasks processing diagonal elements)

Selective Task Replication Results: FFT



Selective Task Replication Results: Cholesky



Conclusions

- OmpSs and Nanos can be leveraged to develop efficient fault-tolerance mechanisms
- Current results seem promising
  - Scalable
  - Low overhead for checkpointing
  - Parallel and asynchronous