Fault Tolerant Design for a Task-based Runtime

Chongxiao "Shawn" Cao, George Bosilca, Thomas Herault and Jack Dongarra

4/22/2016 – ICL Lunch Talk

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Introduction

- Towards Exascale computing, dynamic task-based runtimes can alleviate the disparity between hardware peak performance and application performance

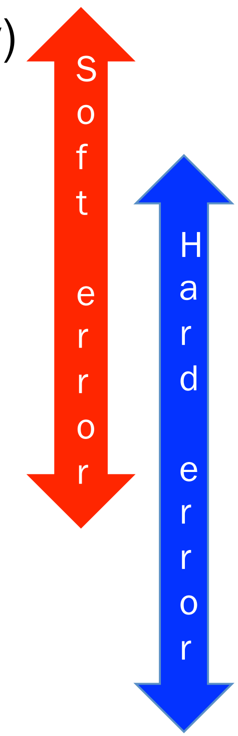
- As supercomputer grows larger, the rate of faults it represents will grow exponentially.
  - Cosmic rays: transistors get smaller, more prone to cosmic ray–induced errors.
  - Bad solder: radioactive lead can cause bit-flip in L1 cache
  - Reduced power: (1) smaller transistors & lower voltages increases the probability of circuits flipping state; (2) power cycling reduces a chip's lifetime

* "How To Kill A Supercomputer: Dirty Power, Cosmic Rays, and Bad Solder", Al Geist @ORNL, Feb 23, 2016

- Goal: Resilient support for a dynamic task-based runtime (PaRSEC)

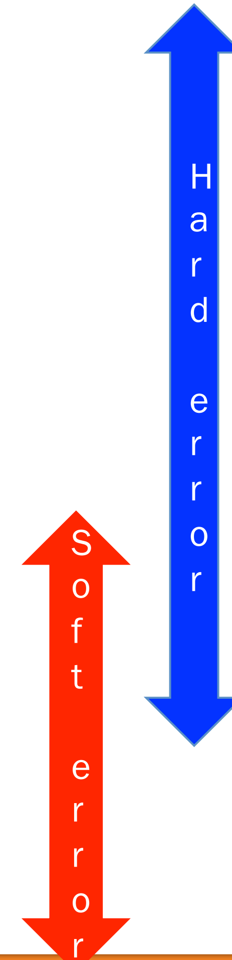
Related Works

- Error model
  - Soft error (silent data corruption, causes bit flips in disk, memory or registers)
  - Hard error (Fail-stop failure causes one node to crash and data resident on that node will be all gone)
- Error-correcting Code Memory (ECC memory)
  - Handle soft error
- Algorithm Based Fault Tolerance (@ICL)
  - Handle soft & hard error
  - Doesn't require disk accesses, low overhead
  - Widely adapted in dense linear algebra
- Checkpoint/Restart Technique
  - Handle soft & hard error
- Application Driven Fault Mitigation
  - Handle hard error
  - User Level Failure Mitigation (ULFM, generic, @ICL)



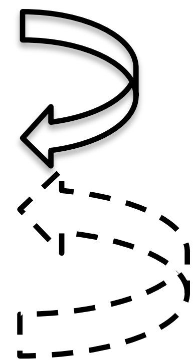
Related Works

- Fault Tolerance in Task-based Runtime
  - Static scheduling
    - Handle hard error
    - Tasks are duplicated and distributed on different processor before application starts
  - Dynamic scheduling
    - KAAPI framework (grid-based)
      - Handle hard failure
      - Coordinated checkpoint, global synchronization
    - Kepler-based distributed scientific workflows
      - Handle soft & hard error
      - Checkpointing & task re-execution
      - The goal is to manage scientific workflows
    - NABBIT (task graph scheduler)
      - Handle soft error, assuming error is reported by runtime
      - Task re-execution, shared-memory only



My contributions

- Towards “soft error”
  - Recovery based-on sub-DAG
  - Recovery based-on Algorithm based Fault Tolerance (Single bit flip)
  - Recovery based-on data logging
- FT layer in a Task-based Runtime
  - Automatic resilience for non-FT applications over PaRSEC

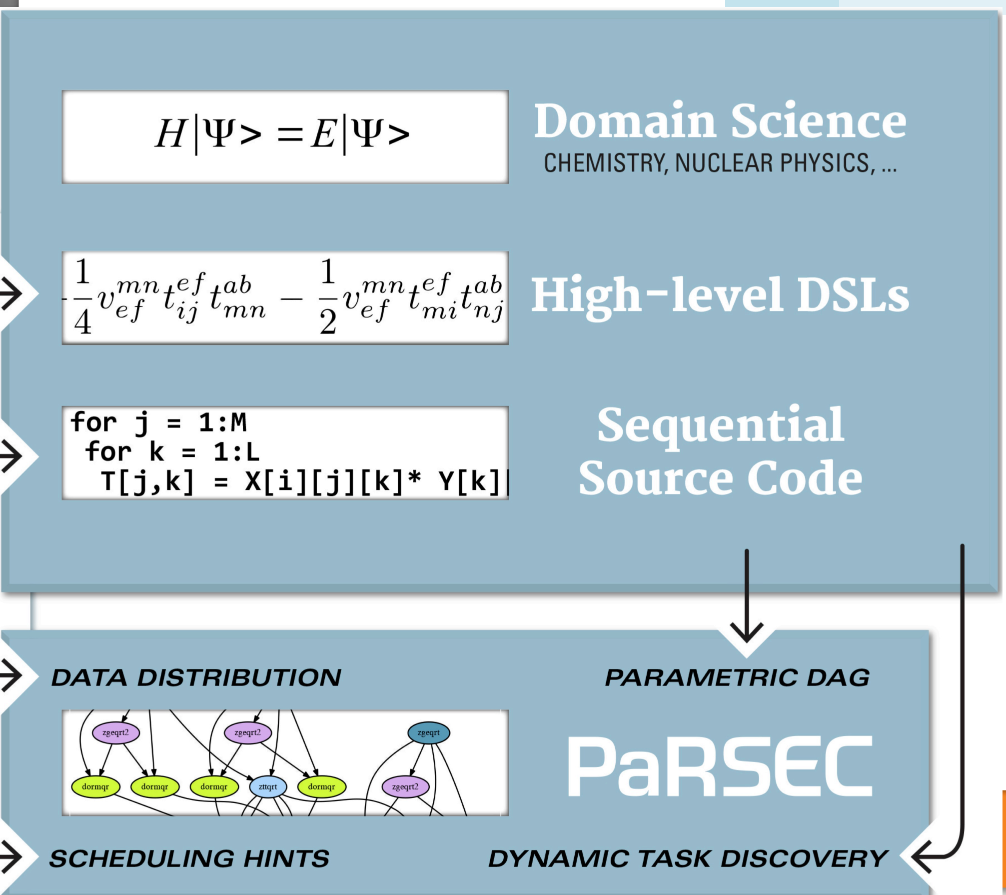


- Towards “hard error”
  - Recovery based-on data logging remotely (in progress)
- Provide optimal resilient scheme
  - Modeling protection and recovery cost (future work)
  - Efficient solution via dynamic programming algorithm (future work)

PaRSEC: a generic runtime system for asynchronous, architecture aware scheduling of fine-grained tasks on distributed many-core heterogeneous architectures

Concepts

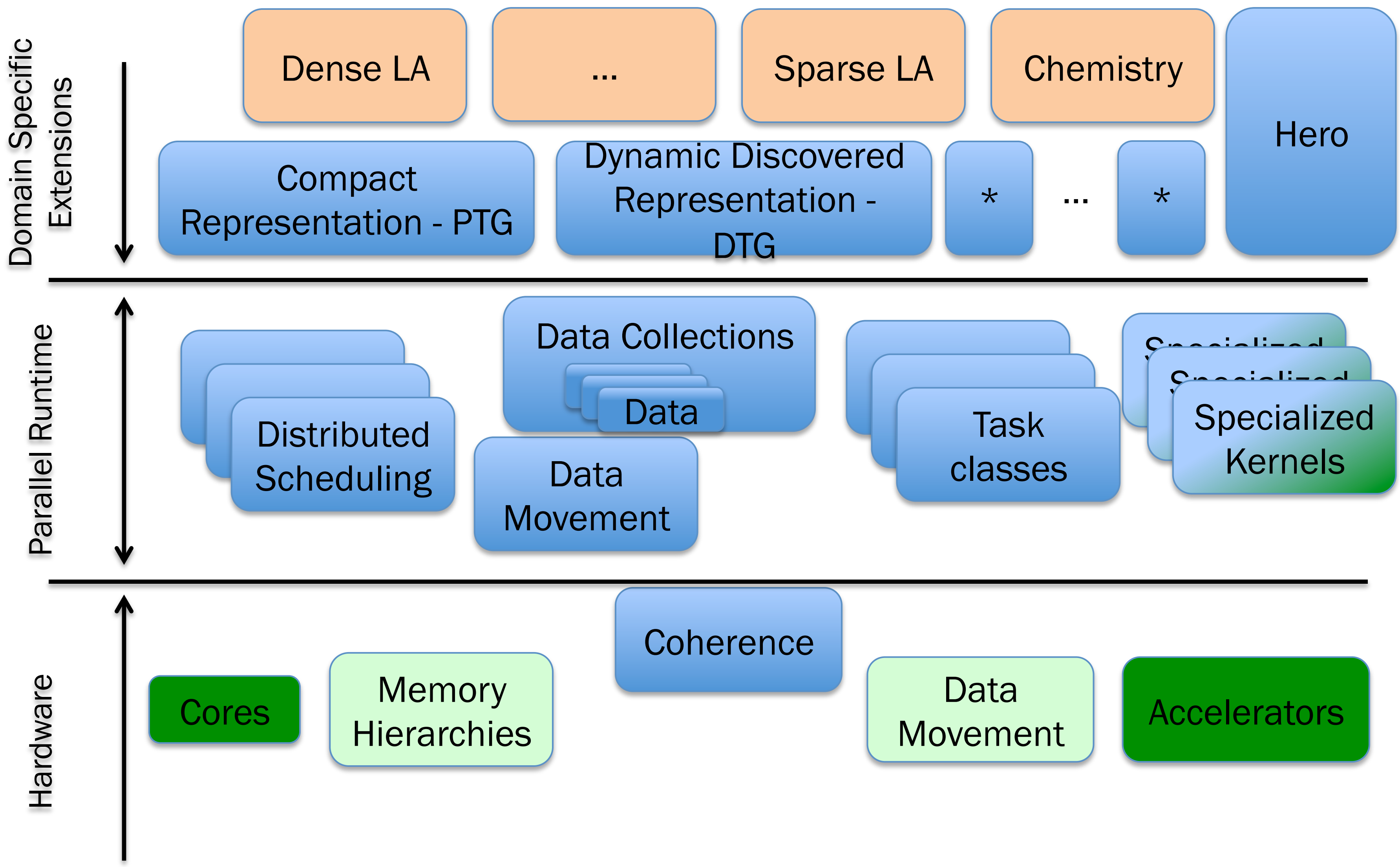
- Clear separation of concerns: compiler optimize each task class, developer describe dependencies between tasks, the runtime orchestrate the dynamic execution
- Interface with the application developers through specialized domain specific languages (PTG/JDF, Python, insert_task, fork/join, …)
- Separate algorithms from data distribution
- Make control flow executions a relic



Runtime

- Permeable portability layer for heterogeneous architectures
- Scheduling policies adapt every execution to the hardware & ongoing system status
- Data movements between consumers are inferred from dependencies. Communications/ computations overlap naturally unfold
- Coherency protocols minimize data movements
- Memory hierarchies (including NVRAM and disk) integral part of the scheduling decisions

The PaRSEC framework



A simpler example (POTRF)

- *The Cholesky Factorization*

User's view

Runtime's view

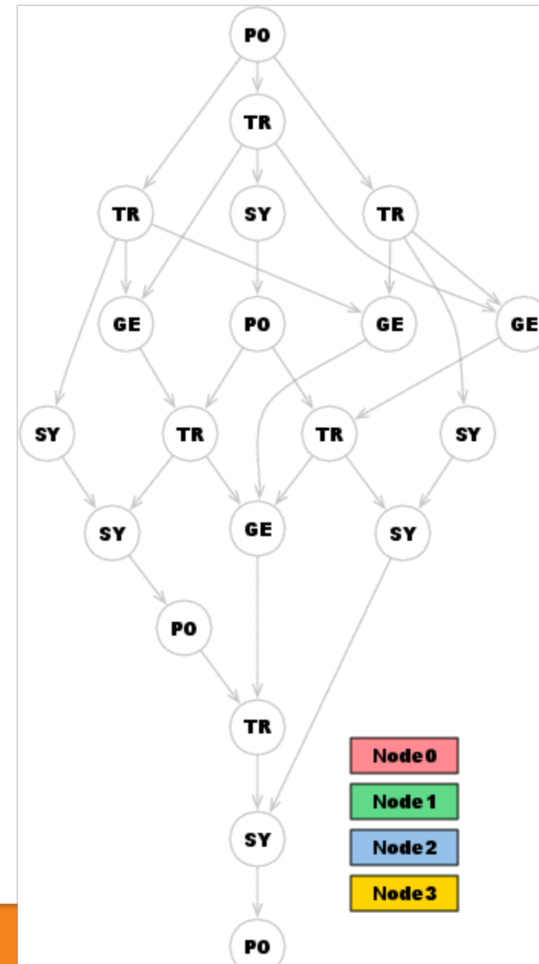
Algorithm:

for $k = 0 \ldots NT-1$ **do**
  $A[k][k] \leftarrow POTRF(A[k][k])$
  for $m = k+1 \ldots NT-1$ **do**
    $A[m][k] \leftarrow TRSM(A[k][k], A[m][k])$
  for $n = k+1 \ldots NT-1$ **do**
    $A[n][n] \leftarrow SYRK(A[n][k], A[n][n])$
  for $m = n+1 \ldots NT-1$ **do**
    $A[m][n] \leftarrow GEMM(A[m][k], A[n][k], A[m][n])$

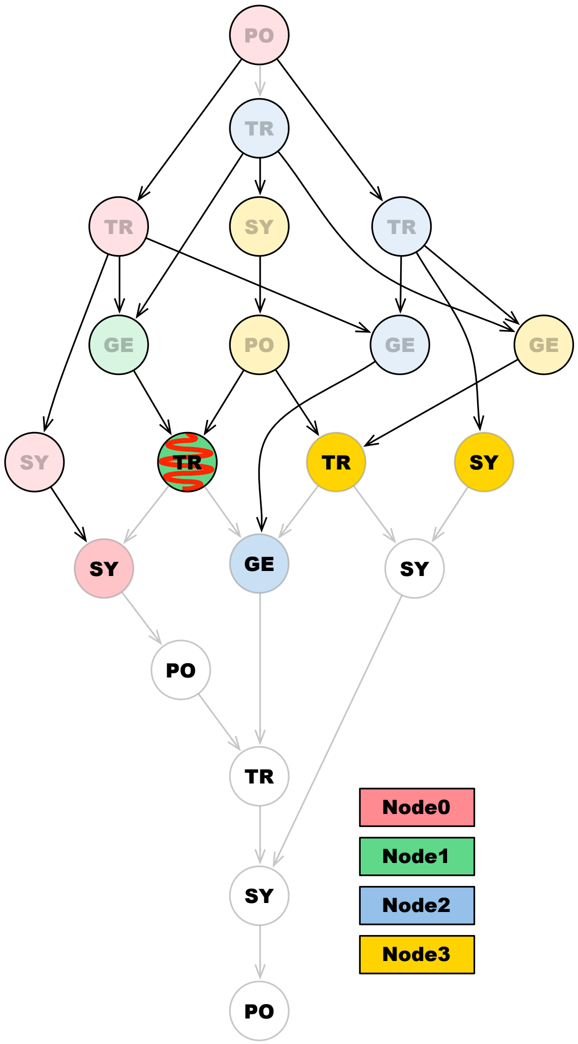
Data layout:

A_{00}			
A_{10}	A_{11}		
A_{20}	A_{21}	A_{22}	
A_{30}	A_{31}	A_{32}	A_{33}

- Final result
- POTRF
- TRSM
- SYRK
- GEMM

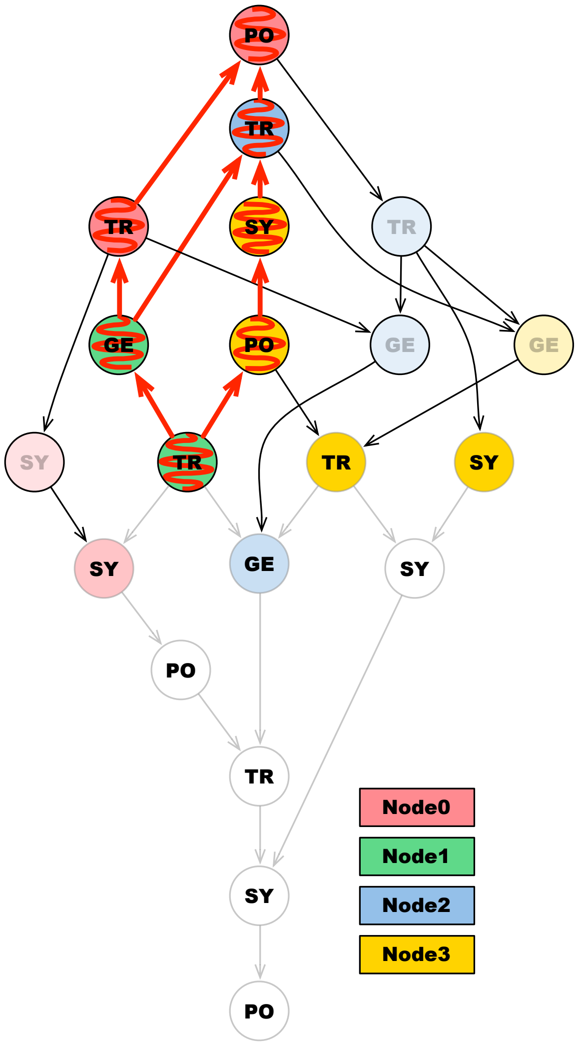


Approach 1: Sub-DAG Recovery



Approach 1: Sub-DAG Recovery

- For each predecessor of a failed task
  - Mark it as failed (if not already marked)
  - Repeat the algorithm



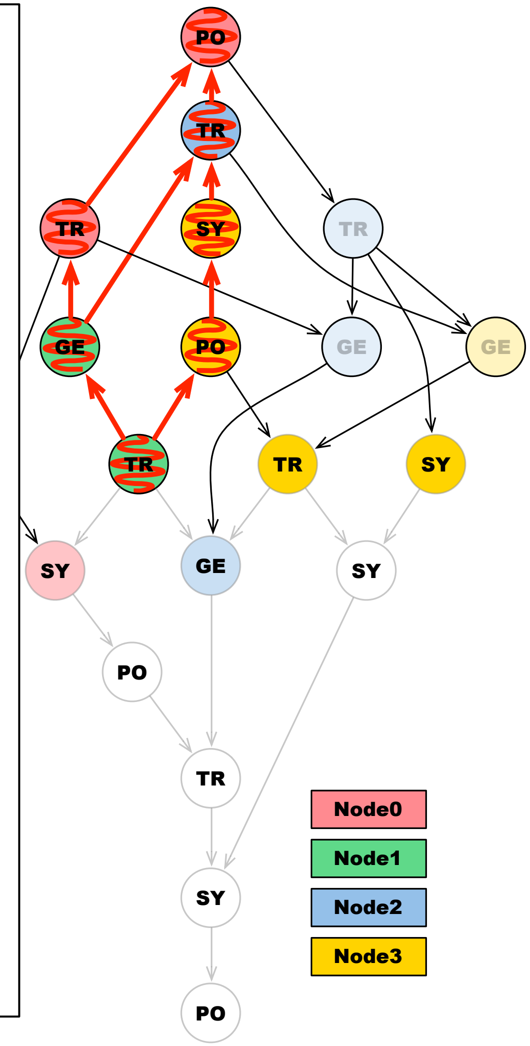
Approach 1: Sub-DAG Recovery

- F
  fa
  -
  -

We can move in both directions in the algorithms ?

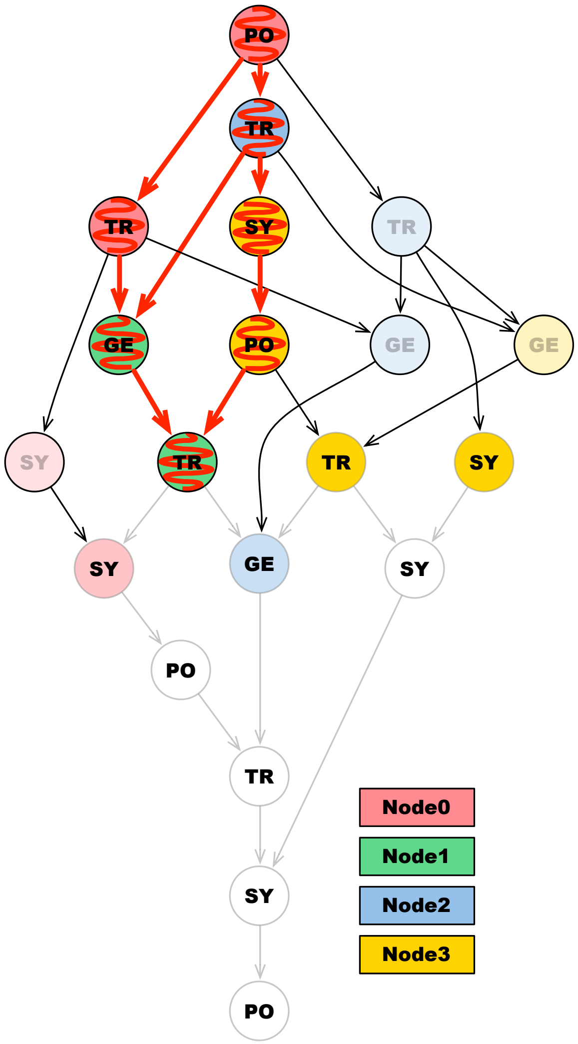
In PTG the DAG does not exists, only a bi-directional parameterized representation. This symbolic representation allows the exploration of the algorithm in any needed way without additional storage for the entire DAG.

```
POTRF(k)
 /* Execution space */
 k = 0 .. NT-1
 /* Locality */
 : A(k, k)
 RW    A <- (k == 0)    ? A(k, k)
                       : A1 HERK(k-1, k)
         ->              A TRSM(k+1 .. NT-1, k)   [type = LOWER]
         ->              A(k, k)                [type = LOWER]
  /* Priority */
 ;(NT-k)*(NT-k)*(NT-k)
```



Approach 1: Sub-DAG Recovery

- For each predecessor of a failed task
  - Mark it as failed (if not already marked)
  - Repeat the algorithm
- Execute the identified sub-DAG in parallel with the tasks of the original DAG
- As long as we have access to the original data (which should be protected) we can guarantee the completion of the algorithm with the correct result
- Burst of errors are supported, multiple sub-DAGs will be executed in parallel with the original



Approach 1: Overheads for POTRF

- How far the application went?
- Computing overhead:
  - Depends on the failure position

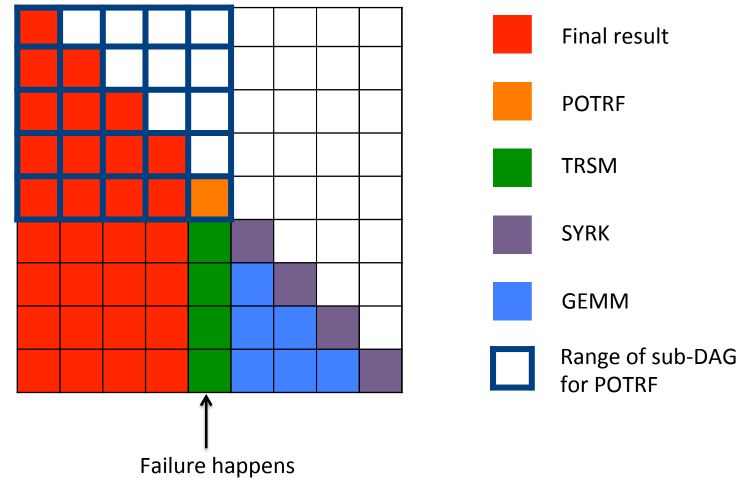
The cost of recovering for a soft error in *Kth* step

$FLOP_{orig} = 1/3(N)^3$

$FLOP_{extra} = 1/3(K*NB)^3$

$Overhead_{comp} = (K*NB/N)^3$

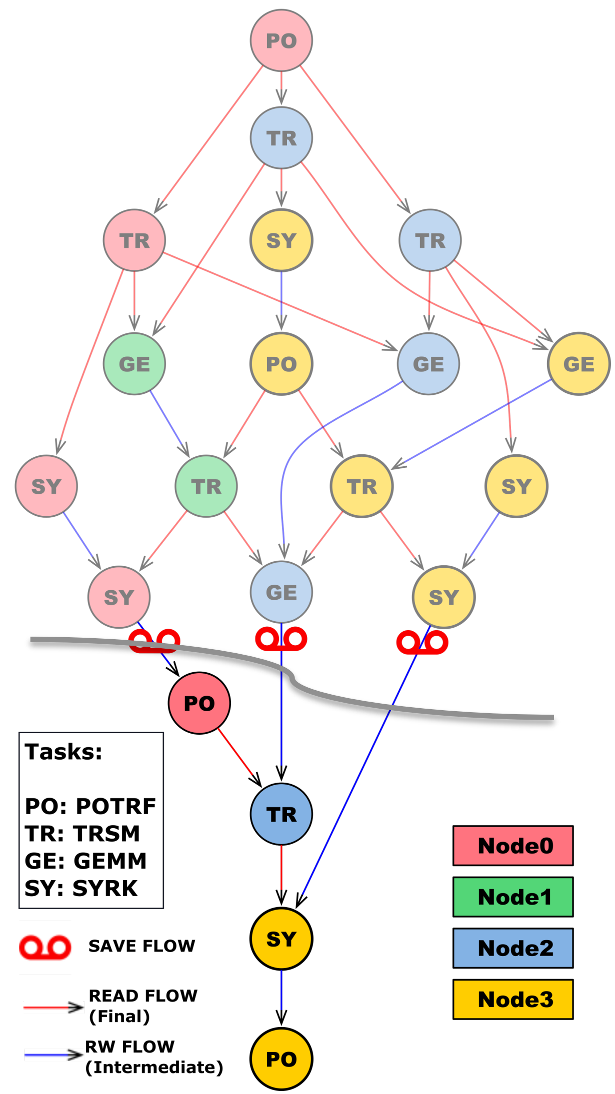
Beginning	Middle	End	No Failure
$(NB/N)^3$	12.5%	100%	0



- Storage overhead: up to 100% (the stable storage for the original input)

Approach 2: Data Logging

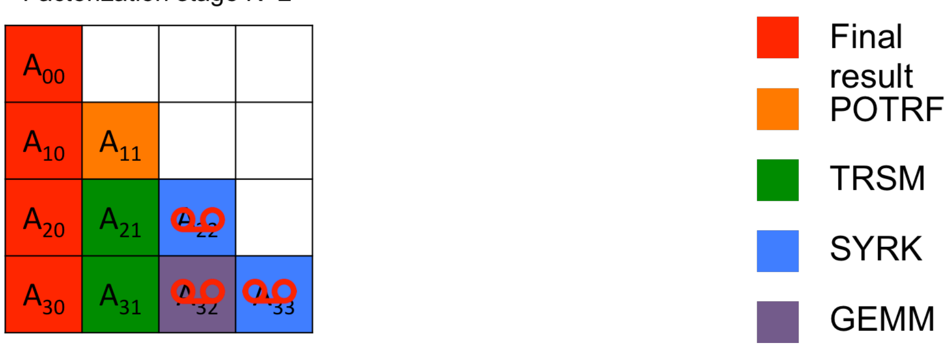
- Minimize the re-execution by logging intermediary data (RW flow)
  - Tasks above the log wave will never be re-executed
- For each predecessor of a failed task
  - If (task has RW flow ! behind the log wave)
    Mark it as failed (if not already marked)
  - Repeat the algorithm

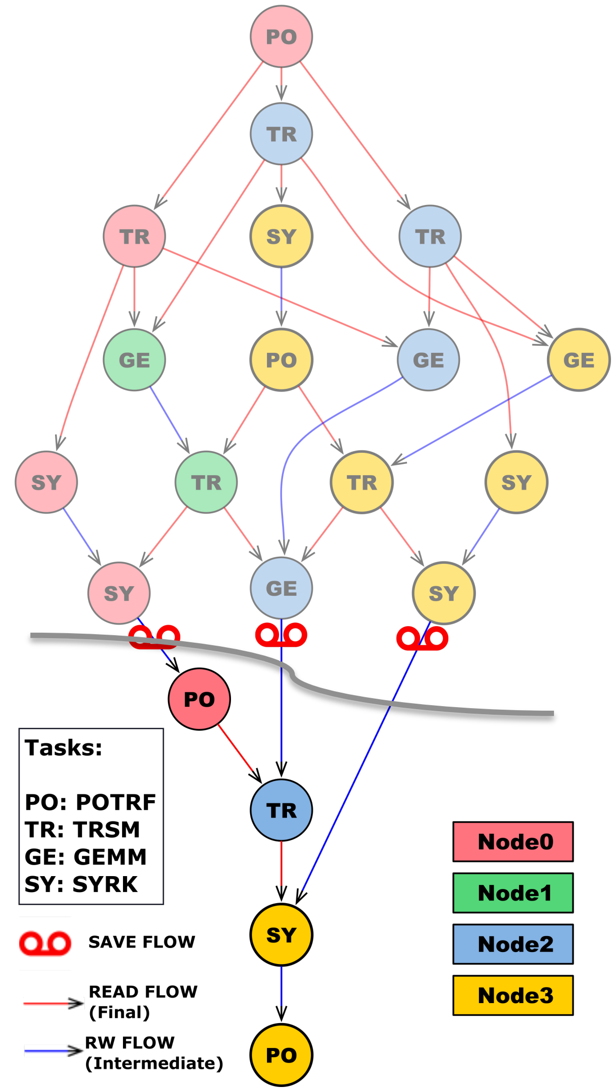


Approach 2: Data Logging

- Minimize the re-execution by logging intermediary data (RW flow)
  - Tasks above the log wave will never be re-executed
- For each predecessor of a failed task
  - If (task has RW flow ! behind the log wave)
    Mark it as failed (if not already marked)
  - Repeat the algorithm

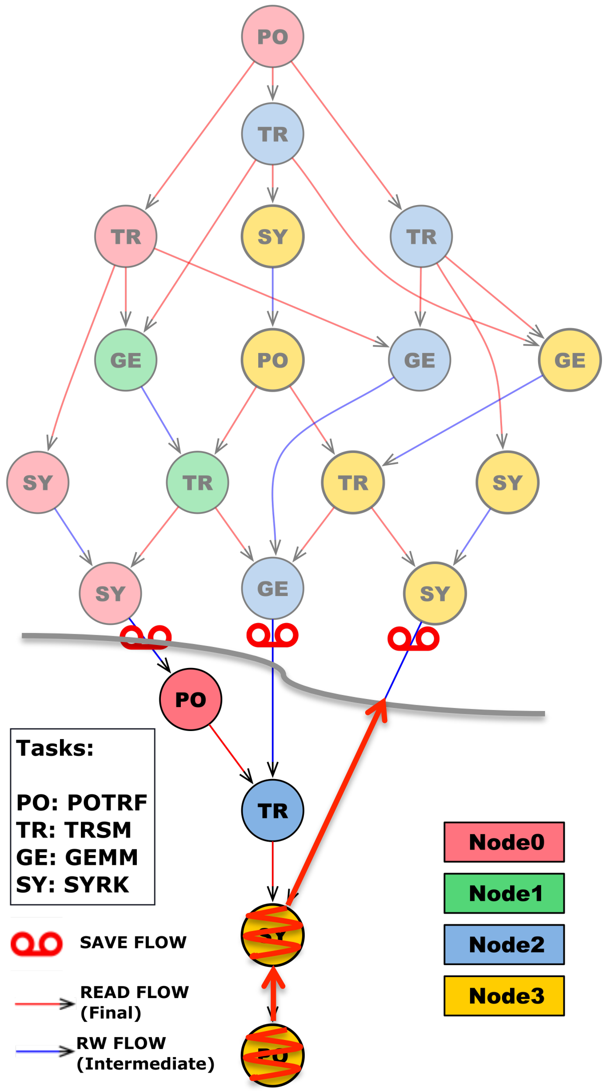
Factorization stage K=2





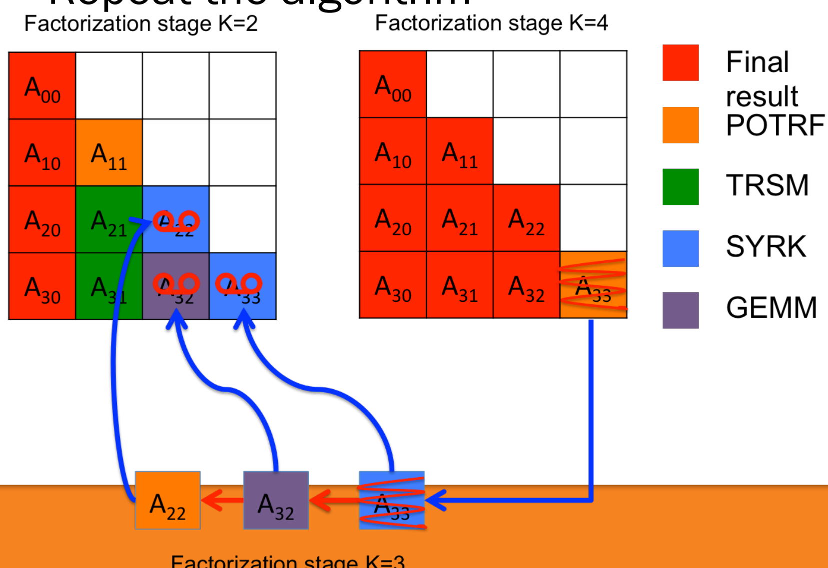
Approach 2: Data Logging

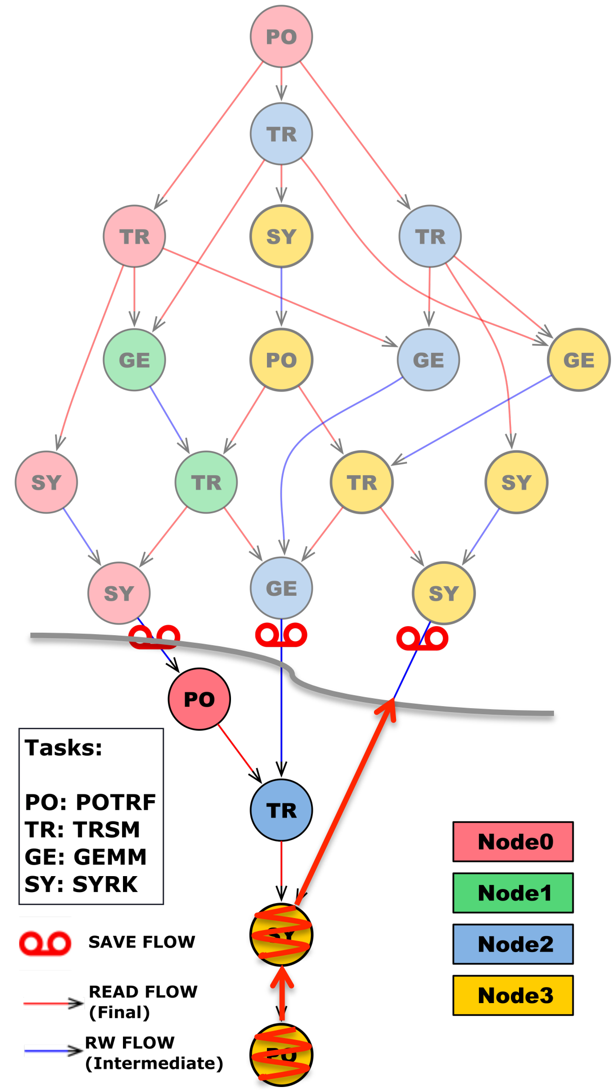
- Minimize the re-execution by logging intermediary data (RW flow)
  - Tasks above the log wave will never be re-executed
- For each predecessor of a failed task
  - If (task has RW flow ! behind the log wave)
    Mark it as failed (if not already marked)
  - Repeat the algorithm



Approach 2: Data Logging

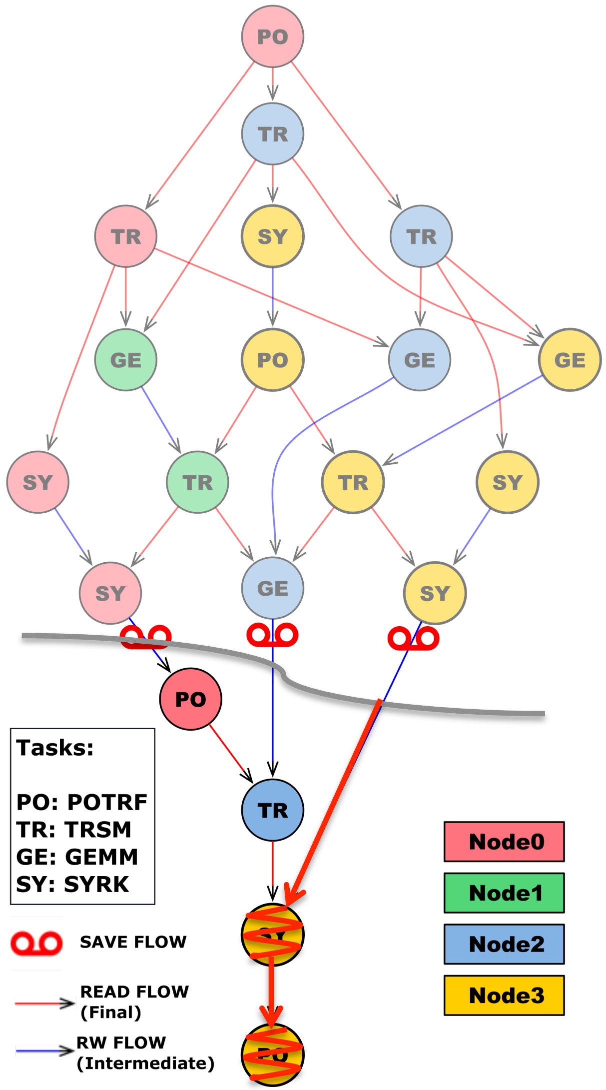
- Minimize the re-execution by logging intermediary data (RW flow)
  - Tasks above the log wave will never be re-executed
- For each predecessor of a failed task
  - If (task has RW flow ! behind the log wave)
    Mark it as failed (if not already marked)
  - Repeat the algorithm





Approach 2: Data Logging

- Minimize the re-execution by logging intermediary data (RW flow)
  - Tasks above the log wave will never be re-executed
- For each predecessor of a failed task
  - If (task has RW flow ! behind the log wave)
    Mark it as failed (if not already marked)
  - Repeat the algorithm
- Execute the identified sub-DAG in parallel with the tasks of the original DAG
- As long as we have access to the logged data (which should be protected) we guarantee the completion of the algorithm with the correct result
- Burst of errors are supported, multiple sub-DAGs will be executed in parallel with the original



Approach 2: Overhead for POTRF

- Saving interval β, a copy of each dataflow is reserved every β updates
  - How to decide the value of β? cost modeling (future work)
- Failure position: any step in the algorithm
- Computing overhead:
  - Almost independent with the failure position
  - The number of FLOPs of a task is $C \cdot nb^3$, where C is 1/3 for POTRF, 1 for TRSM, 1 for SYRK and 2 for GEMM. We set C to 2.
  - $FLOP_{extra} = \beta 2NB^3$

Beginning	Middle	End	No Failure
$(NB/N)^3$	$\beta 6(NB/N)^3$	$\beta 6(NB/N)^3$	≈ 0

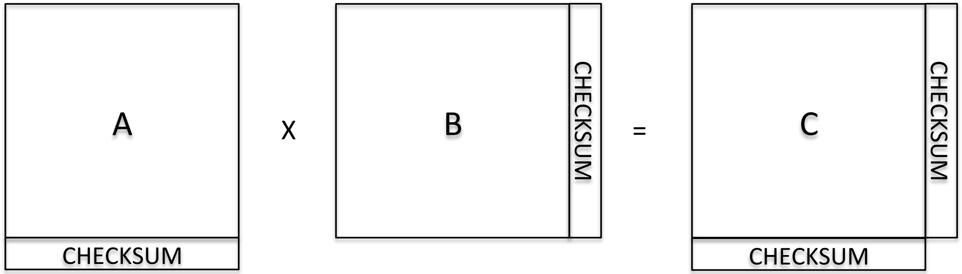
- Storage overhead: up to 100%

Approach 3: ABFT

- Algorithm-Based Fault Tolerance (ABFT)

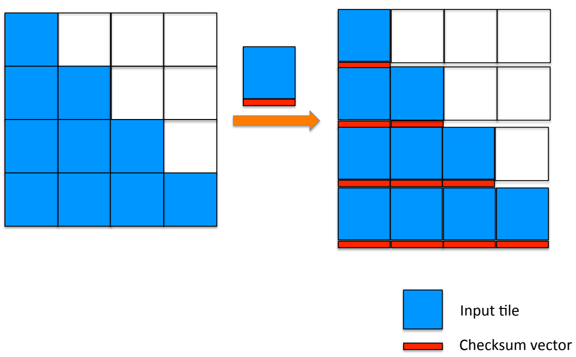
	Application Level	Task Level
Minimum recovery unit	Task in DAG	Operation in Task

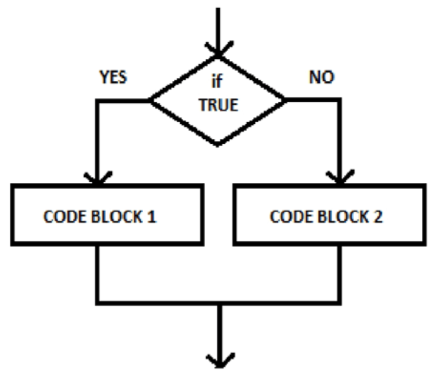
- No re-execution each task become self-sufficient
- Applying ABFT inside a task
  - Pros: avoid re-execution;
    error detection capability.
  - Cons: potentially less generic; ABFT limited in linear algebra
- Example of ABFT matrix multiplication:

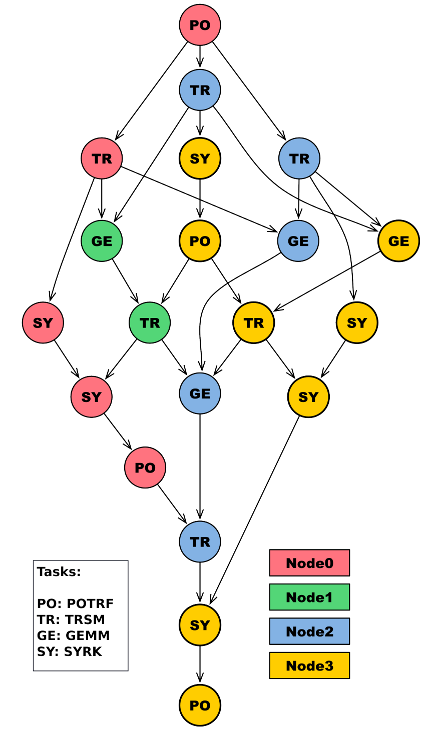


Approach 3: ABFT

- Extend the data collections to attach the 2 checksum vectors to the original data
- Provide recovery scheme inside the task
- Same algorithm (same DAG)







Approach 3: Overhead for POTRF

- Computing overhead:
  - Independent with failure position

	One Failure	No Failure
$Overhead_{Comp}$	$(1 + \frac{2}{nb})^3 - 1 + \frac{1}{nb}$	$(1 + \frac{2}{nb})^3 - 1 + \frac{1}{nb}$

Maintain Checksum

Detecting and recovering inside a task

1 FLOP correcting error, negligible

- Storage overhead: 2/nb (2 checksum vectors are attached to every nb x nb tile)

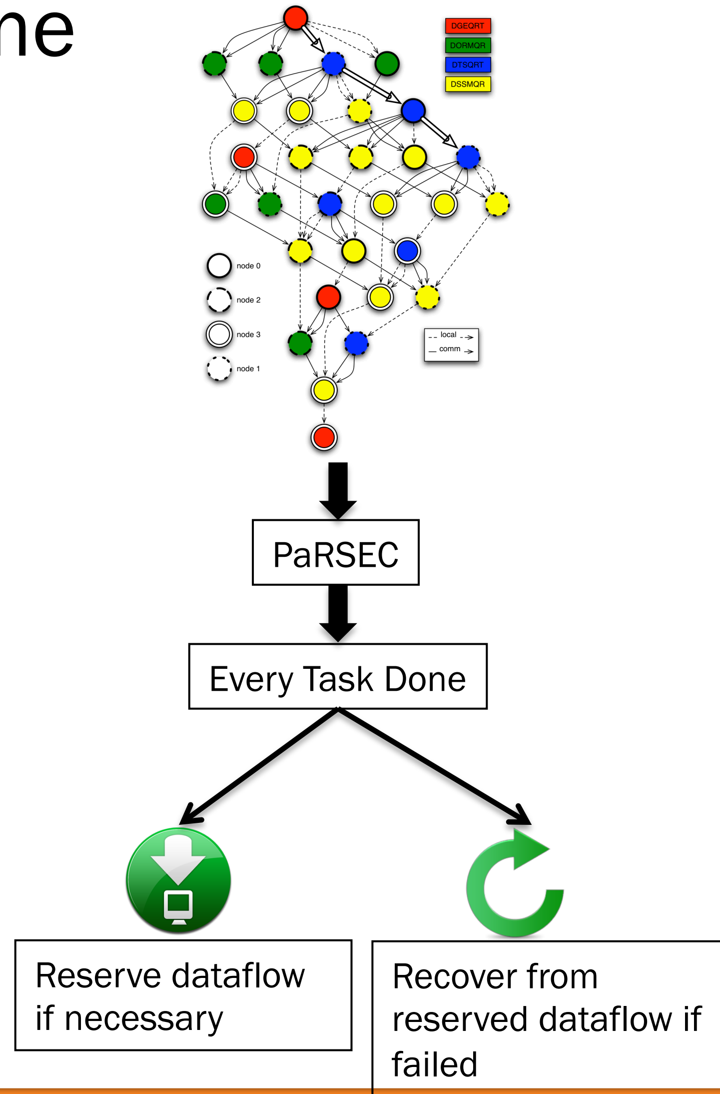
Comparison of Three Mechanisms

	Overhead depends on failure position	Failure detector included	Failure detector possible
Mechanism I: Stateless Runtime	Yes	No	Yes
Mechanism II: Data Logging	Minimally	No	Yes
Mechanism III: Algorithm Based Fault Tolerance	No	Yes	Yes

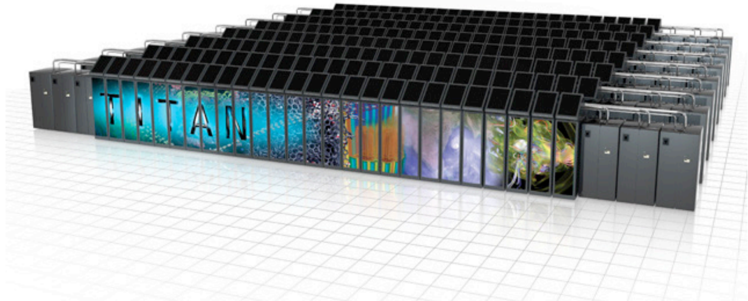
FT layer in a Task-based Runtime

- Resilient support from runtime
  - Recovery based-on data logging (generic & low-overhead)
  - Merge resilient features into runtime:
    - Reserve minimum dataflow for protection
    - Minimize task re-execution
    - Minimize extra memory
  - Export interface for user/tool –configurable data logging scheme
  - Automatic resilience for non-FT applications over PaRSEC

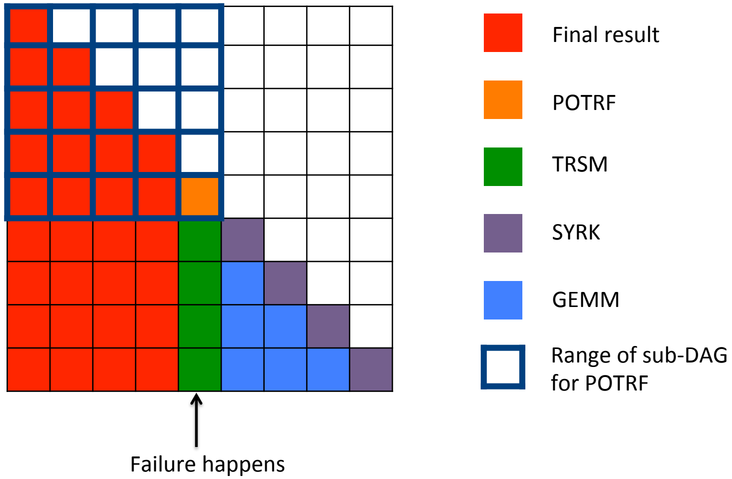
Original Non-FT DAG (jdf)



Towards Soft Errors



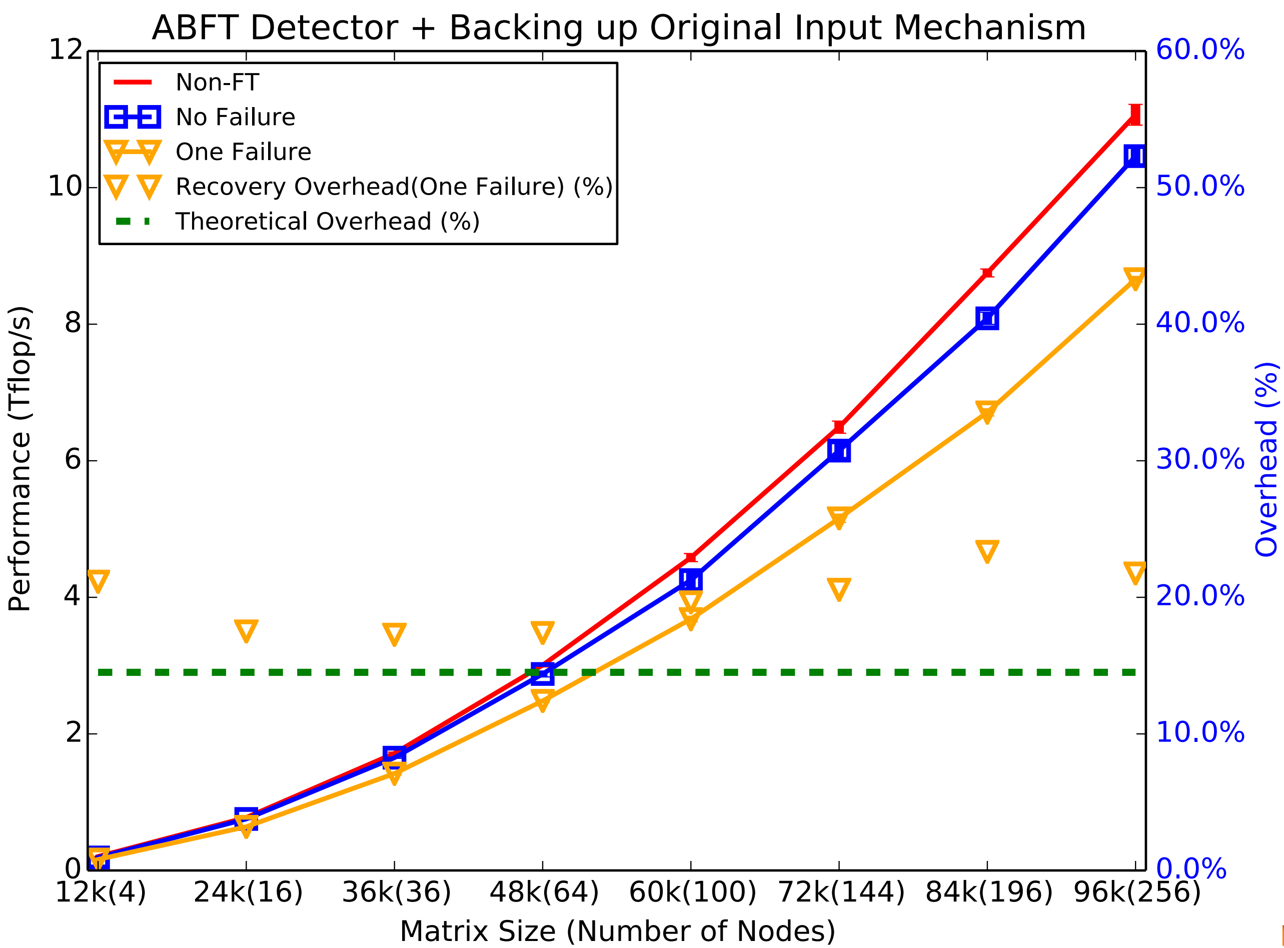
- Experiment System
  - Titan @ ORNL
  - Use 256 nodes (weak scalability)
  - Use CPU section of the system, every node has a 16-core AMD Opteron 6274 CPU; we use 8 core per node. (2 cores share 1 FPU)
  - GCC 4.8.2, Cray LibSci
  - DPLASMA tile size = 200
  - Failure injected during the factorization of the middle column



The worst case: In practice there is no accurate/portable mechanism to report bitflips fast. Because we are looking specifically at dense linear algebra kernels, we forced PaRSEC to provide a failure detector (derived from the Algorithm Based Fault Tolerance techniques), by 1) maintaining a checksum during all operations; and 2) validating each operation once completed. Thus, an overhead due to the failure detector is visible on all the performance graph.

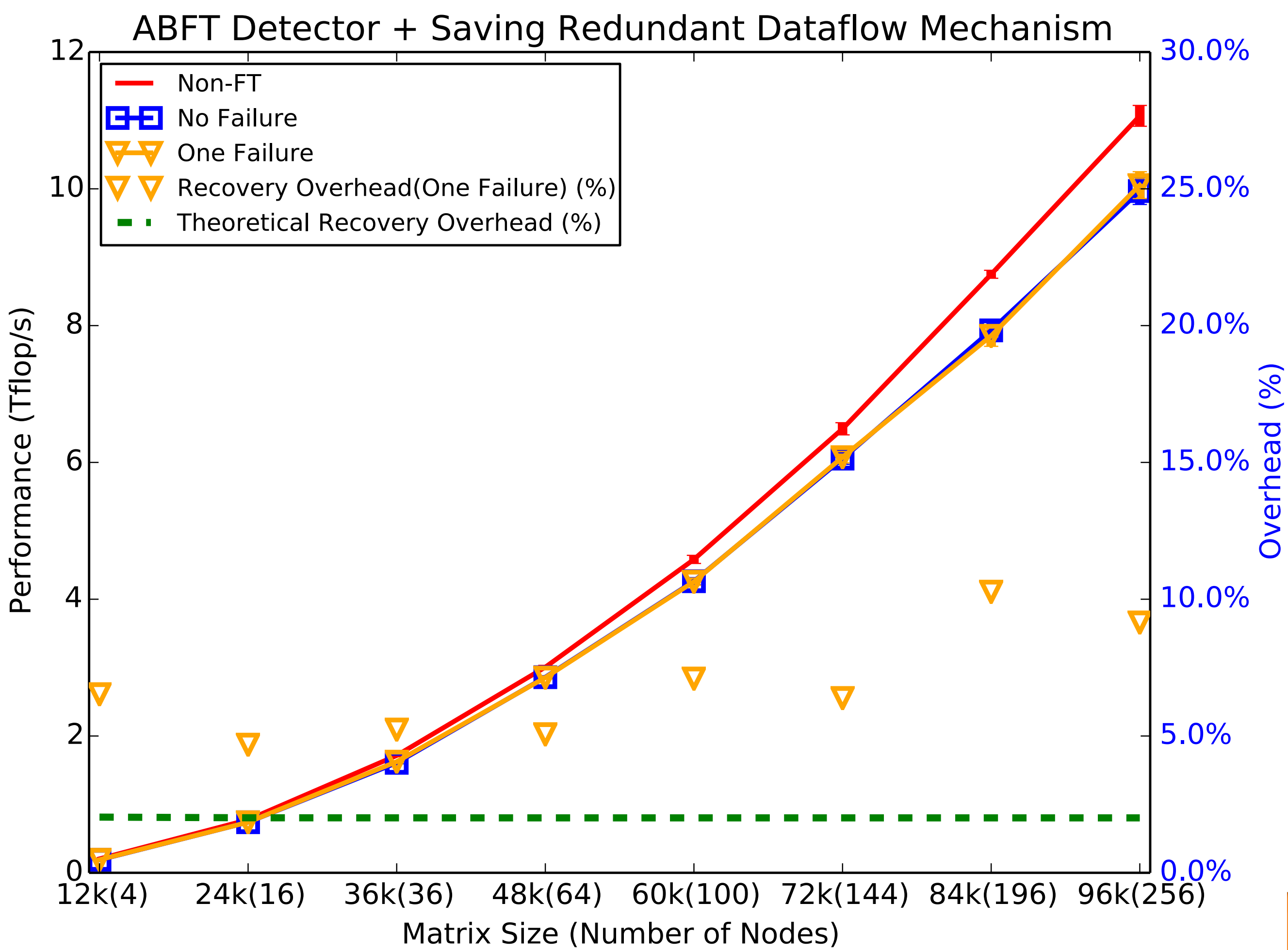
Approach 1: Sub-DAG Recovery

- Original input backup to stable storage
- Failure detector integrated in the algorithm



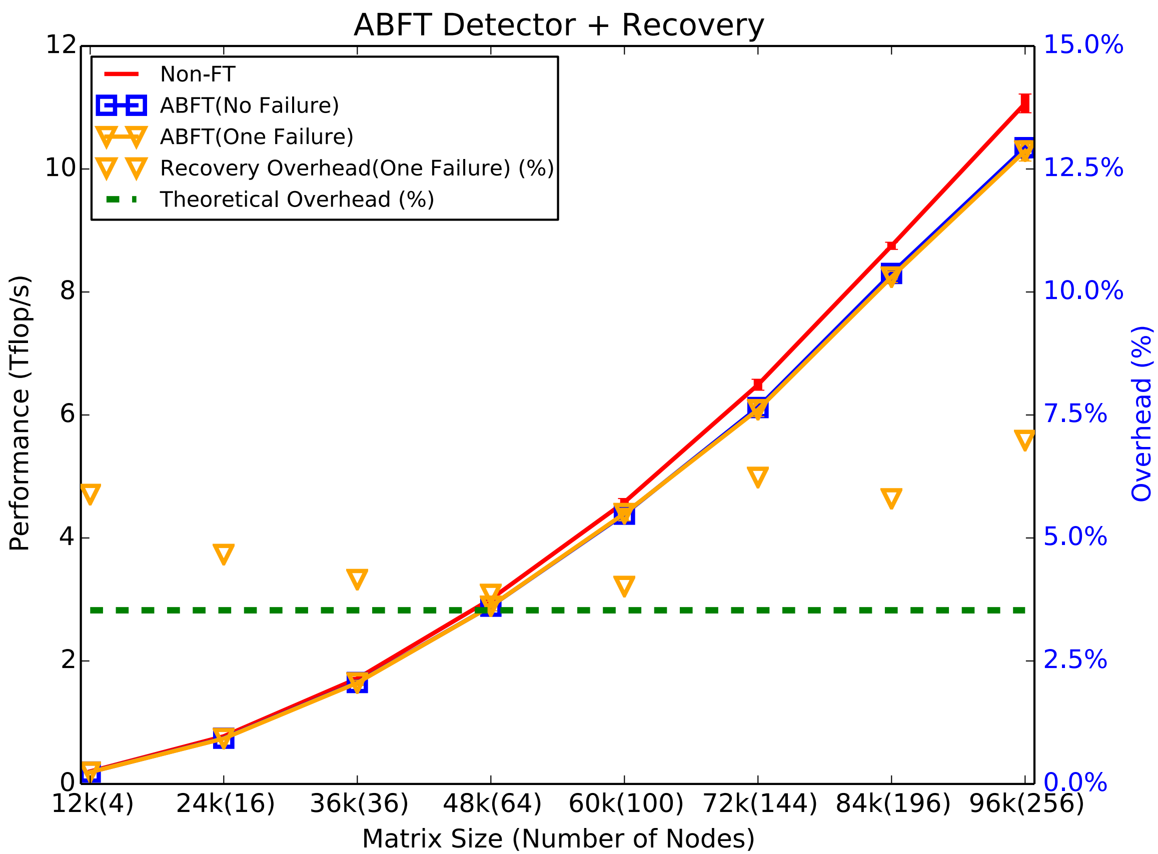
Approach 2: Data Logging

- Save dataflow every 10 updates
- Failure detector integrated in the algorithm

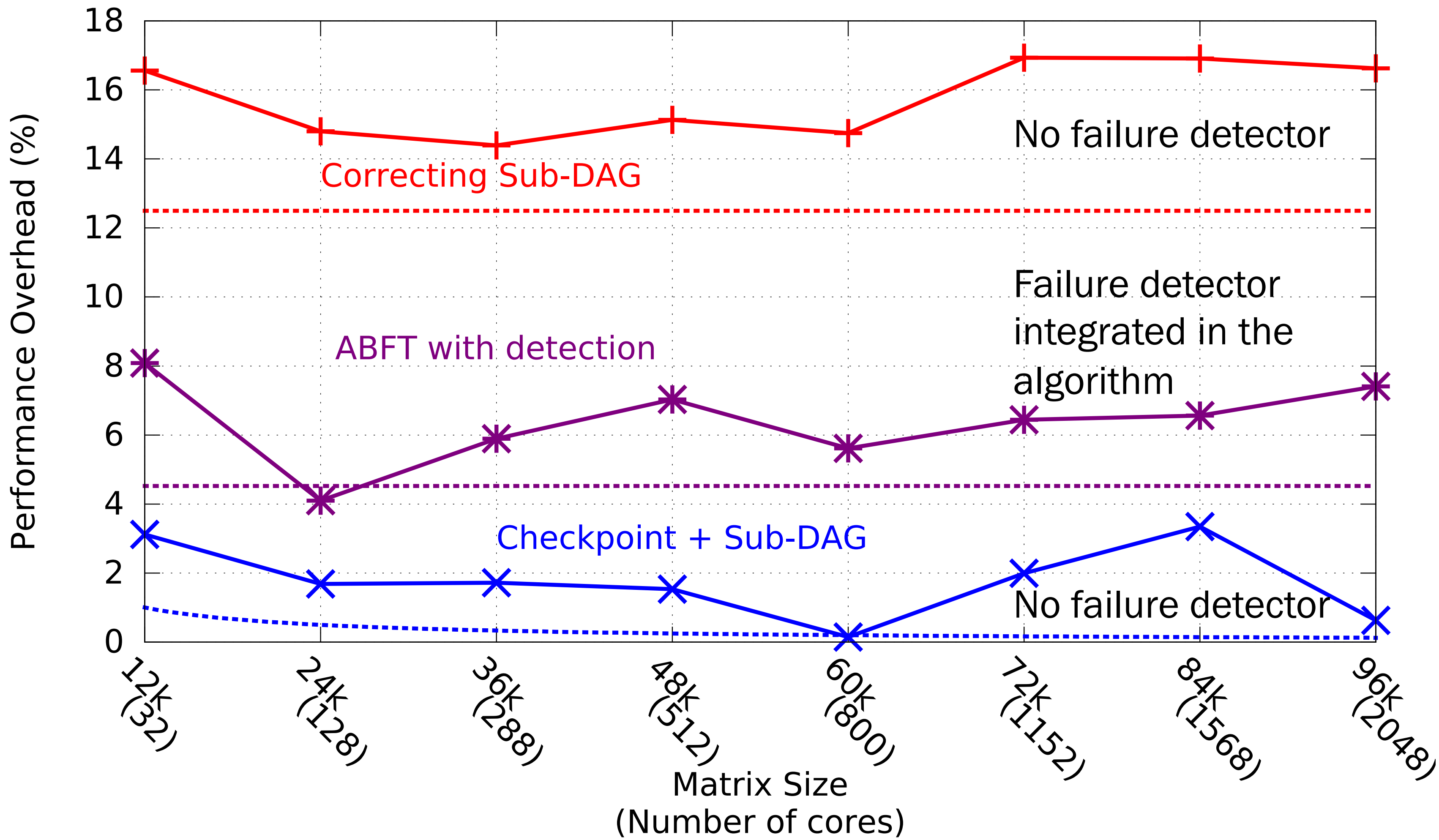


Approach 3: ABFT

- Failure detector integrated in the algorithm
- Single bit-flip



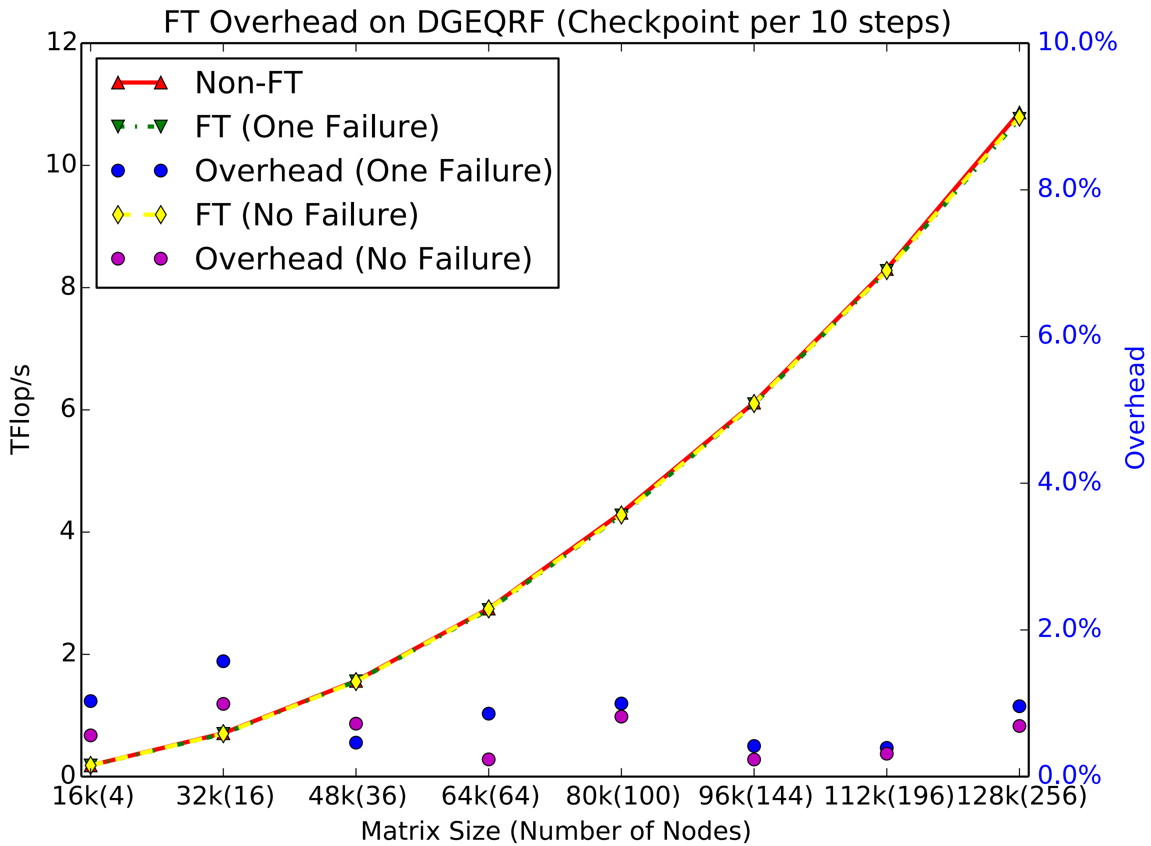
When a hardware failure detector is available



At this block size the overhead of the check-summing and validating each operation accounts for about 6%

Automatic Resilient Support for QR

- Apply Non-FT QR on FT layer
  - FT layer provides data logging (save dataflow every 10 updates)
  - FT layer re-executes tasks

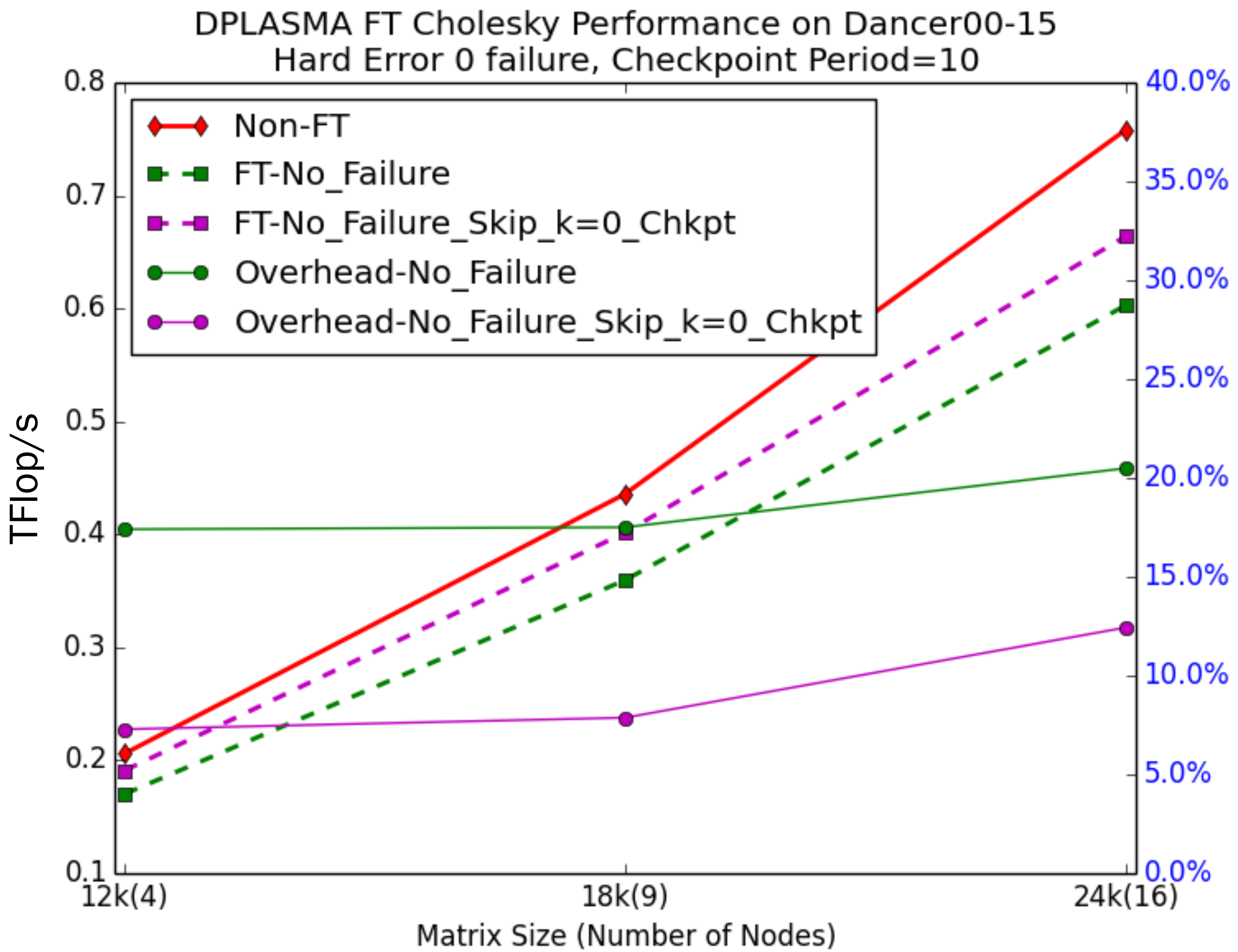


Conclusions & Future Work

- Conclusions
  - Low-overhead fault tolerant support for task-based runtime
  - Resilient feature integrated into runtime
- Future work
  - Support hard error (in progress)
  - Efficient fault-tolerant scheme via protection and recovery cost modeling

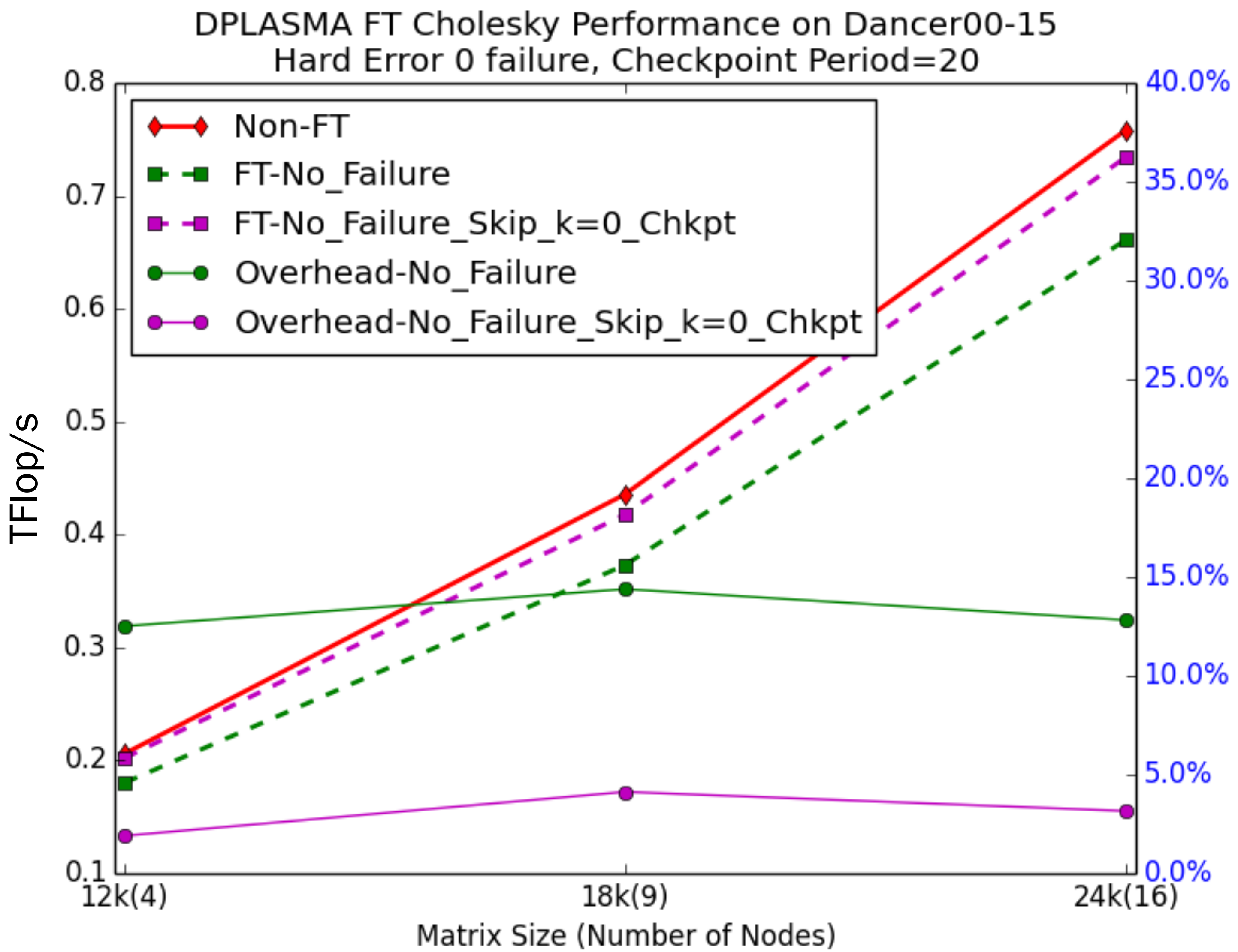
First step toward hard error support

- Log the data remotely

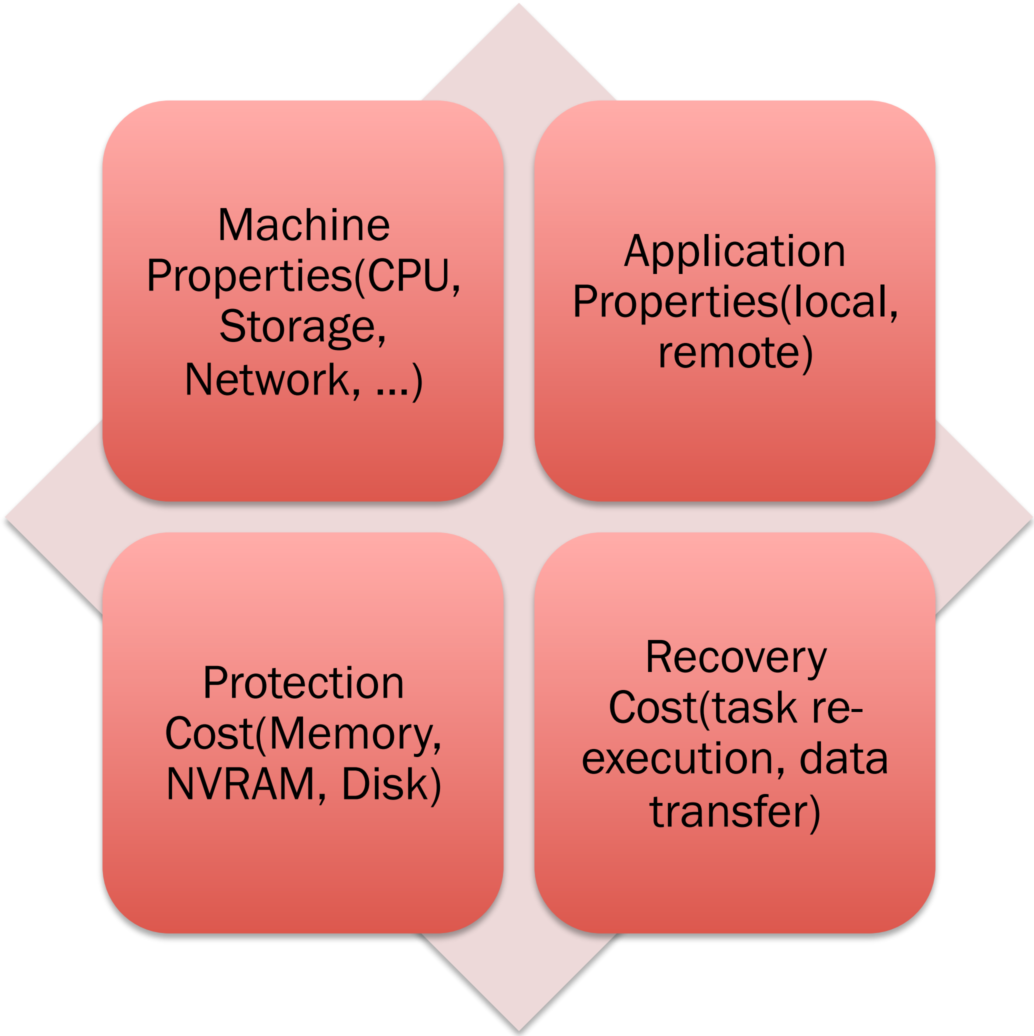


First step toward hard error support

- Log the data remotely



Cost Modeling



Questions?