Design for a Soft Error Resilient Dynamic Task-based Runtime

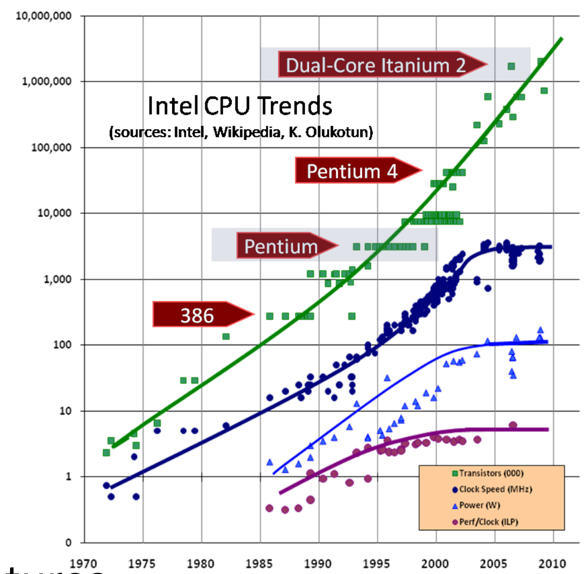
ICL "Friday Lunch", Nov. 14 2014

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

ICL INNOVATIVE COMPUTING LABORATORY
THE UNIVERSITY of TENNESSEE

Motivation: Programming Environment

- Today's HPC Programming Environment
  - Huge number of compute nodes (e.g., Titan in ORNL has 18,688 compute nodes)
  - Heterogeneity inside a compute node: Multicore CPU + Accelerators (e.g., Nvidia Kepler, Intel Xeon Phi, AMD Radeon)



- No free performance lunch anymore
- Efforts on performance portability
  - Load balancing
  - Memory distribution
  - Cache reuse and memory locality on NUMA architectures
  - Overlapping communications and computations

Motivation: Benefits of task-based runtime

- Solution: dynamic task-based runtime
  - User's role: Defining task and data dependencies between these tasks.
  - Runtime's role: Scheduling tasks, manipulating data and adapting execution on hardware.
- Exascale systems are expected to in 2018-2022
- Target of Runtime Systems in IESP roadmap

2014-15	Optimizing runtime: general-purpose runtime automatically achieving load balance, optimized network usage, and communication/computation overlap, minimization of memory consumption at large scale, maximization of performance to power ratio, malleability, and tolerance to performance noise/interference on heterogeneous systems. Why: Complexity of systems will require automatic tuning support to optimize the utilization of resources, which will not be feasible by static, user-specified schedules and partitionings.
2016-17	Fault-tolerant runtime: tolerating injection rates of 10 errors per hour (cooperating with application provided information and recovery mechanisms for some errors). Why: By then systems will have frequent failures, and it will be necessary to anticipate and react to them in order that the application delivers useful results.

Motivation: Goal

- Exascale systems == Vulnerability to failures

  (1) Increaseing number of components to reach the scale

  (2) Increasing Mean Time To Failure (MTTF) of each component is not enough to compensate (1)

- Failure model
  - **Soft error** (Silent Data Corruption)**:** bit-flips in disk, memory or processor registers
  - Fail-stop model: a process halts and does not execute any further operations
  - Here we focus on soft error.
- FT design for a dynamic task-based runtime
  - Implemented in PaRSEC
  - Two levels of granularities and three mechanisms: Looking into DAG and task.
  - Case study of Cholesky factorization

Introduction of PaRSEC

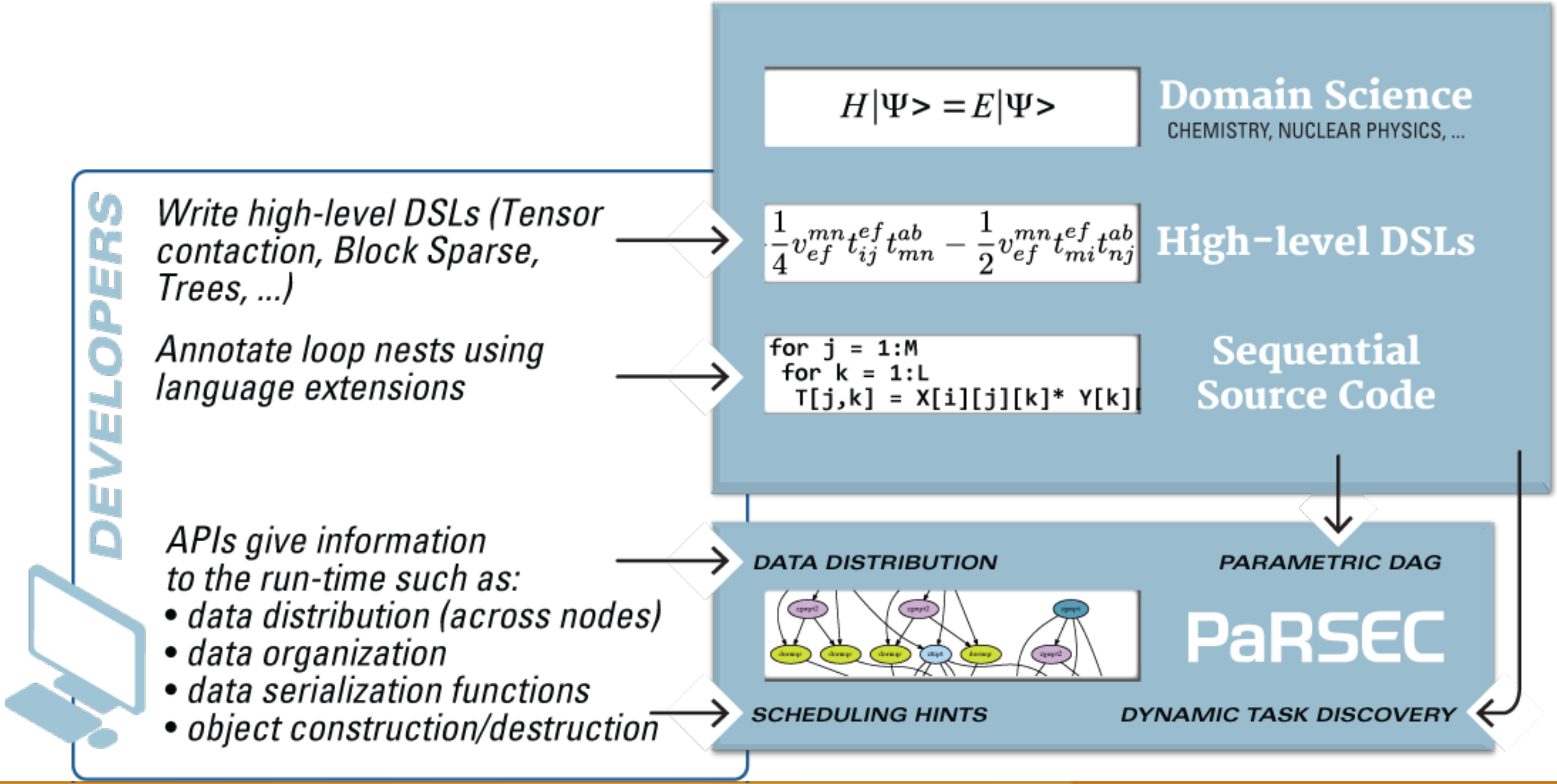
- *What's it ?*
  - Parallel Runtime Scheduling and Execution Controller

- Why choose it ?
  - Other task-based runtimes: e.g., Quark, Star Superscalar, StarPU
  - PaRSEC's unique feature: Using concise representation of DAG (Parameterized Task Graph) in memory, avoiding unrolling the complete DAG. ← (Potential benefits in DAG recovery)

Introduction of PaRSEC

- *How to use it ?*

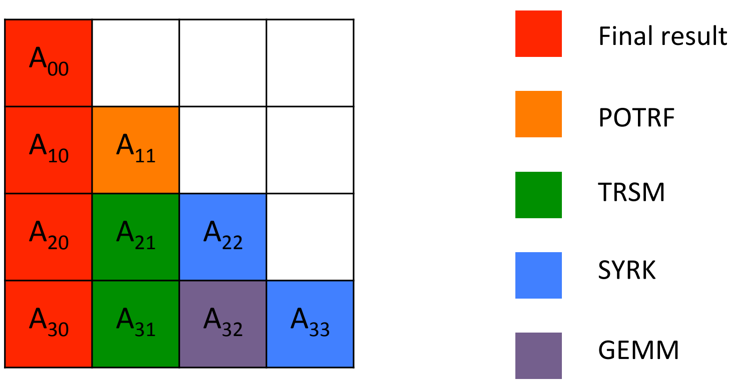
  sequential code in domain science: JDF in DPLASMA



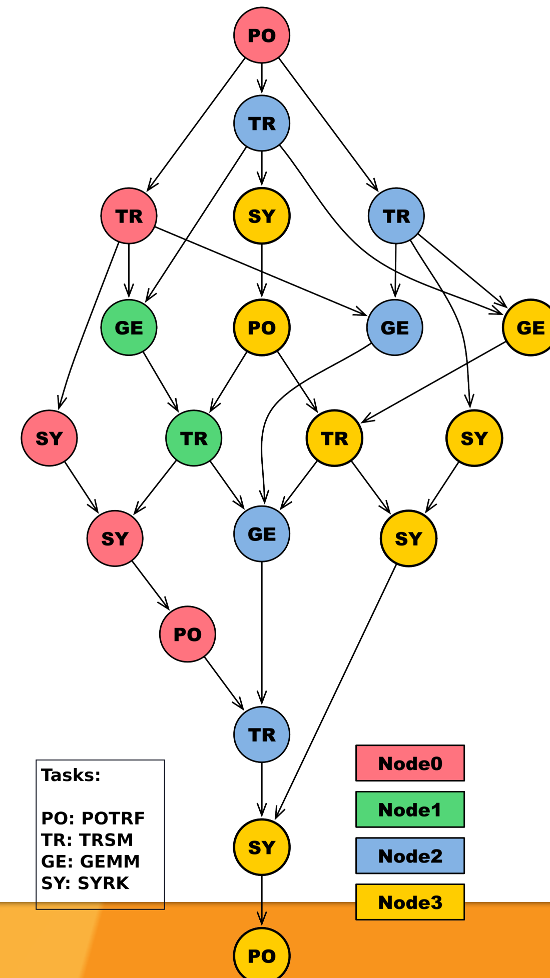
Introduction of PaRSEC

- *Application representation*

User's view



Runtime's view



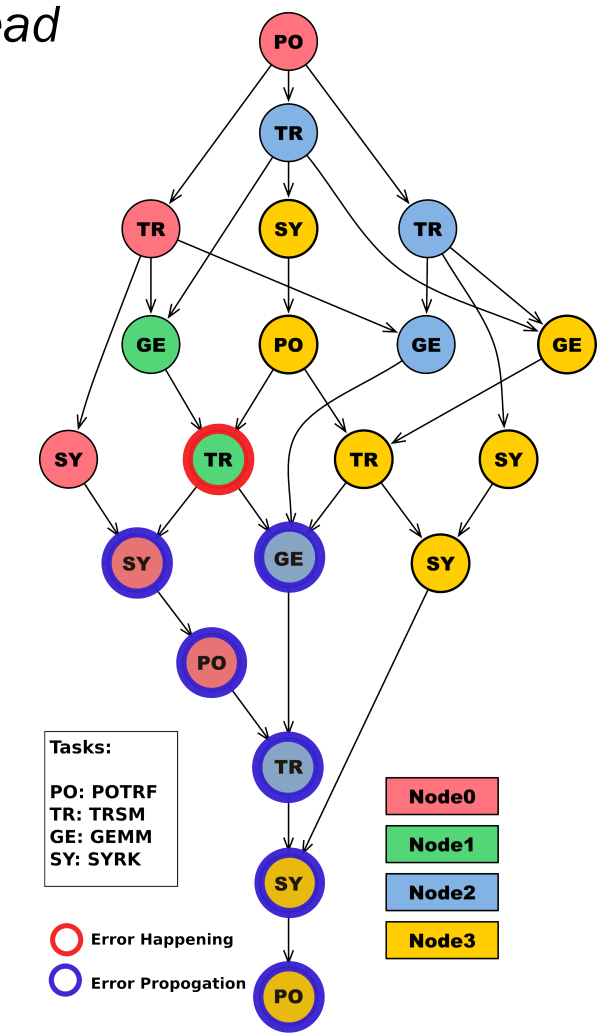
Problem statement

- *Recovery challenges, guaranteeing low-overhead*
  - Prevent failure from propagating to successors

    too many failures to be recovered
  - Recovery of a task in parallel with unaffected task

    original application is stalled, performance is slowed down
  - Avoid global synchronization

    synchronization in distributed-memory platforms is expensive



Failure Detection

- Hardware, e.g., ECC memory
- Algorithmic technique (ABFT)
  - Online (during computation):

    FT-ScaLAPACK: correcting soft errors on-line for ScaLAPACK Cholesky, QR, and LU factorization routines

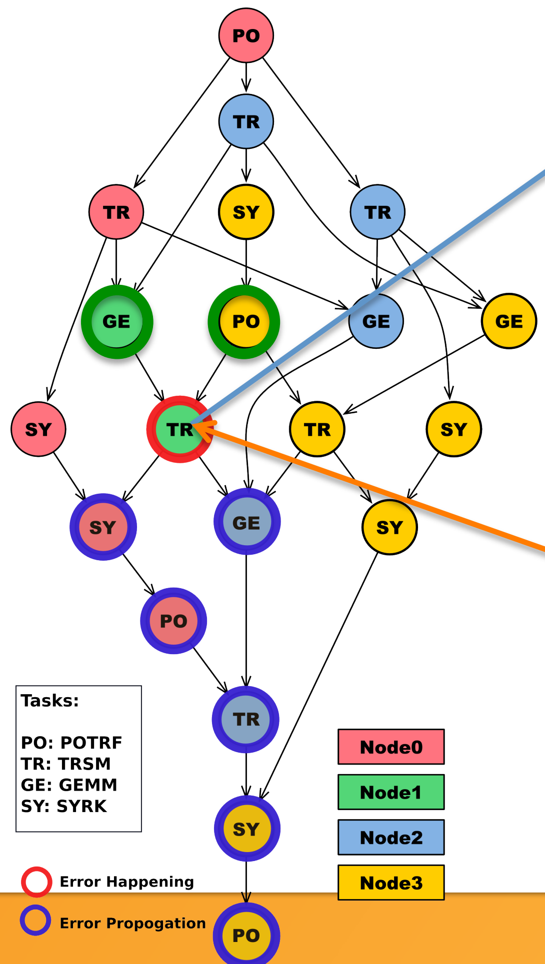
    (Panruo Wu, **Zizhong Chen**, HPDC'14)
  - Offline (end of computation):

    High Performance Dense Linear System Solver with Soft Error Resilience

    (Peng Du, Piotr Luszczek, Jack Dongarra, CLUSTER 2011)
- Assume failure is reported by runtime, focus on a generic recovery in DAG

Application Level Mechanism I

- *Correcting Sub-DAG Strategy*



Retrieve the missing data

Redo task

Where’s the input ?
From predecessors

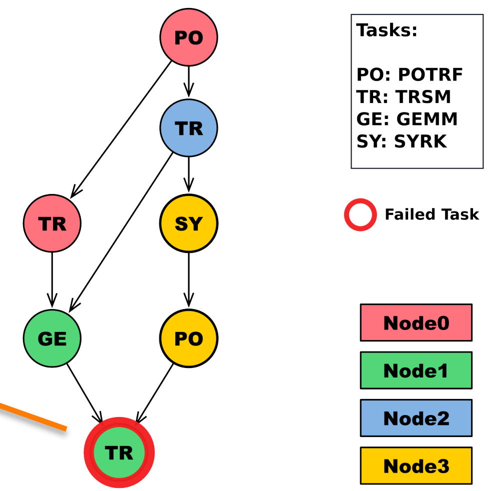
Redo predecessors

Backward traverse

Reach original input

Non-block recovery: only successors pending

PTG:
dynamically retrieve all the predecessors of a failed task



Application Level Mechanism I

- *Overhead*
  - Computing overhead:

  How far does the application go?

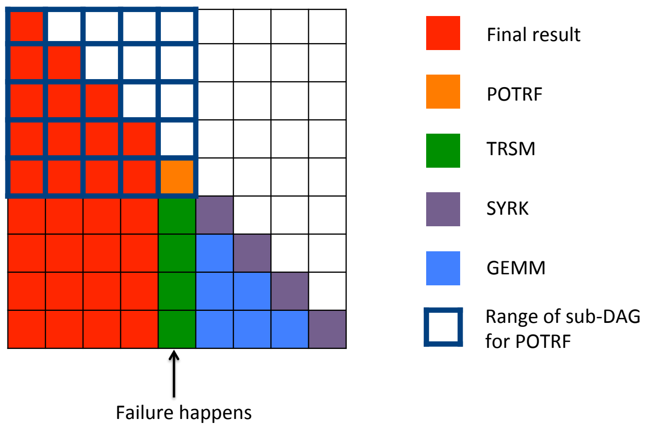
  The cost of recovering failed POTRF in *Kth* column

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

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

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

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

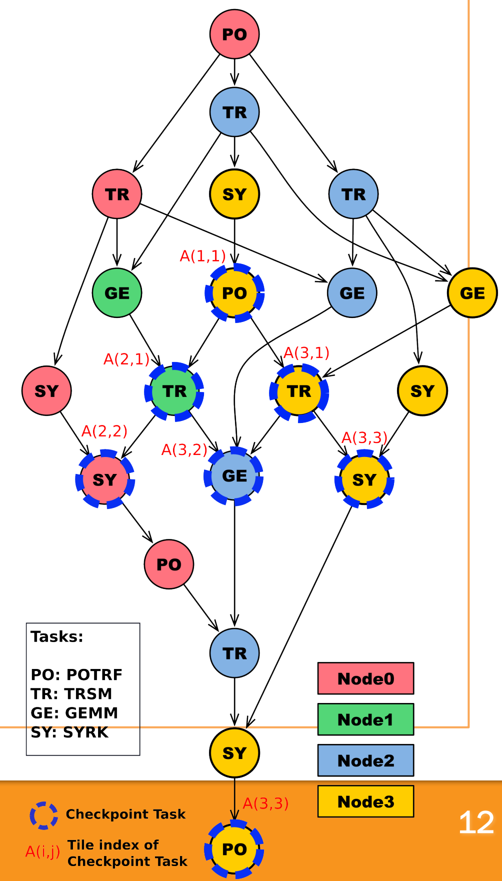


  - Storage overhead: up to 100%

Application Level Mechanism II

- *Sub-DAG & Periodic Checkpoint Composite Strategy*
  - Why the re-execution goes back to origin?
    Intermediate data is not saved.
  - Checkpointing intermediate data, limit the number of re-executions.
  - Checkpoint interval β, a process will save a copy of each data every β updates.
  - Input of failed task:
    - The same tile checkpointed at most β updates ago
    - Final output of another task (validated)
  - Max number of re-executions is β

β = 2 example



Application Level Mechanism II

- *Overhead*
  - Computing overhead:
    - 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

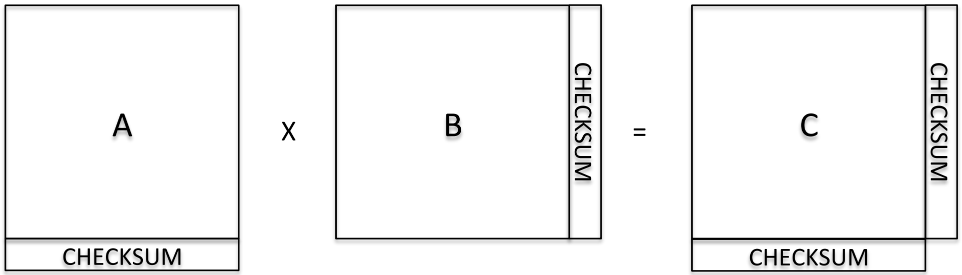
  - Storage overhead: up to 100%

Task Level Mechanism

- *Algorithm-Based Fault Tolerance*

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

- Avoid re-execution ?
  - Can Task be self-resilient ?
- Applying ABFT inside a task
  - Pros: avoid re-execution;
    error detection capability.
  - Cons: potentially less generic; ABFT limited linear algebra
- Example of ABFT matrix multiplication:



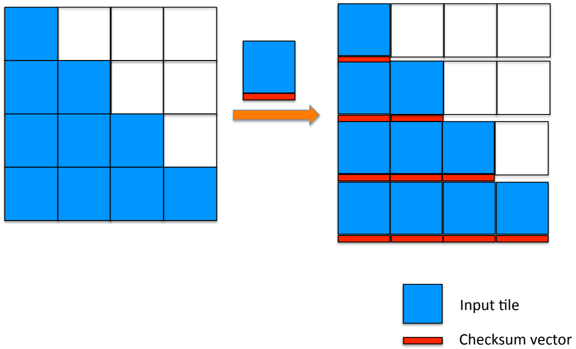
Task Level Mechanism

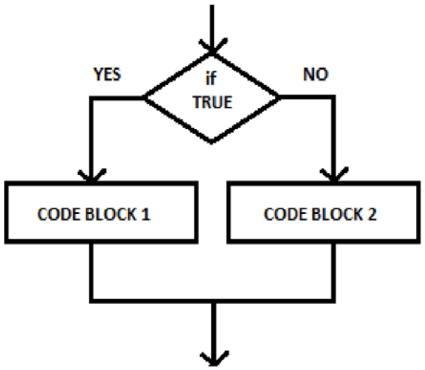
- Implementation

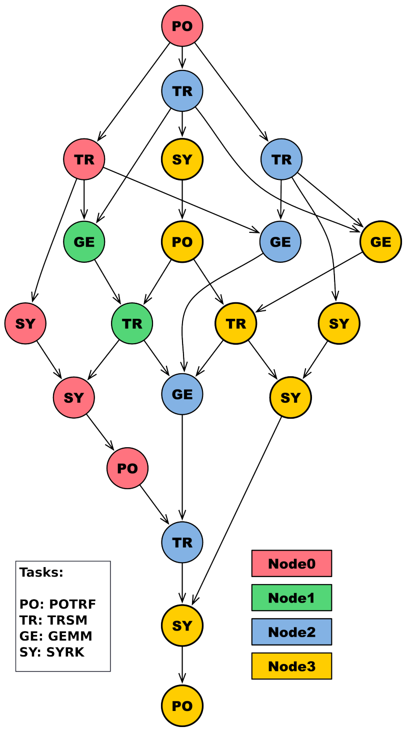
(1) Attaching 2 checksum vectors to original data

(2) Provide recovery scheme inside task

(3) Launch the same DAG







Task Level Mechanism

- *Overhead*
  - Computing overhead:

	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

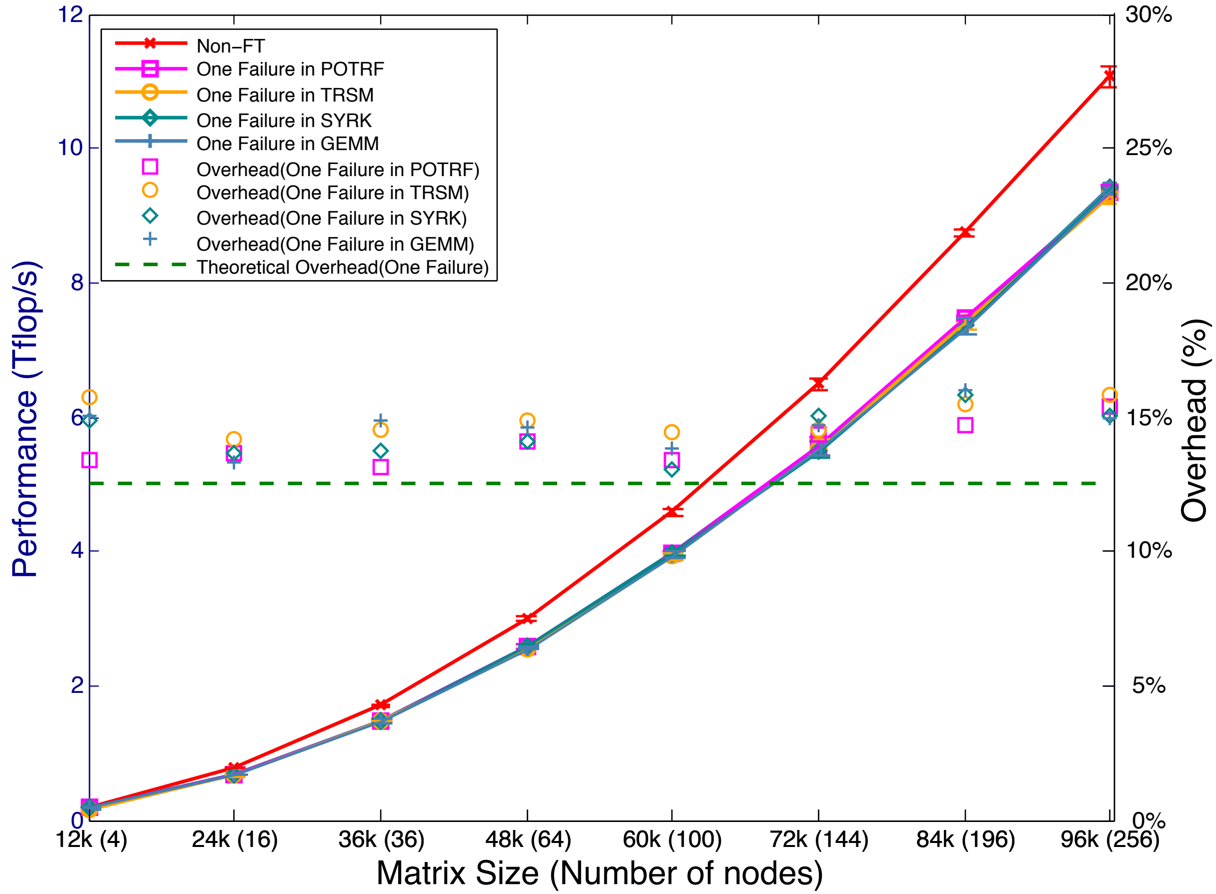
1 FLOP correcting error, negligible

  - Storage overhead: 2/NB

Results

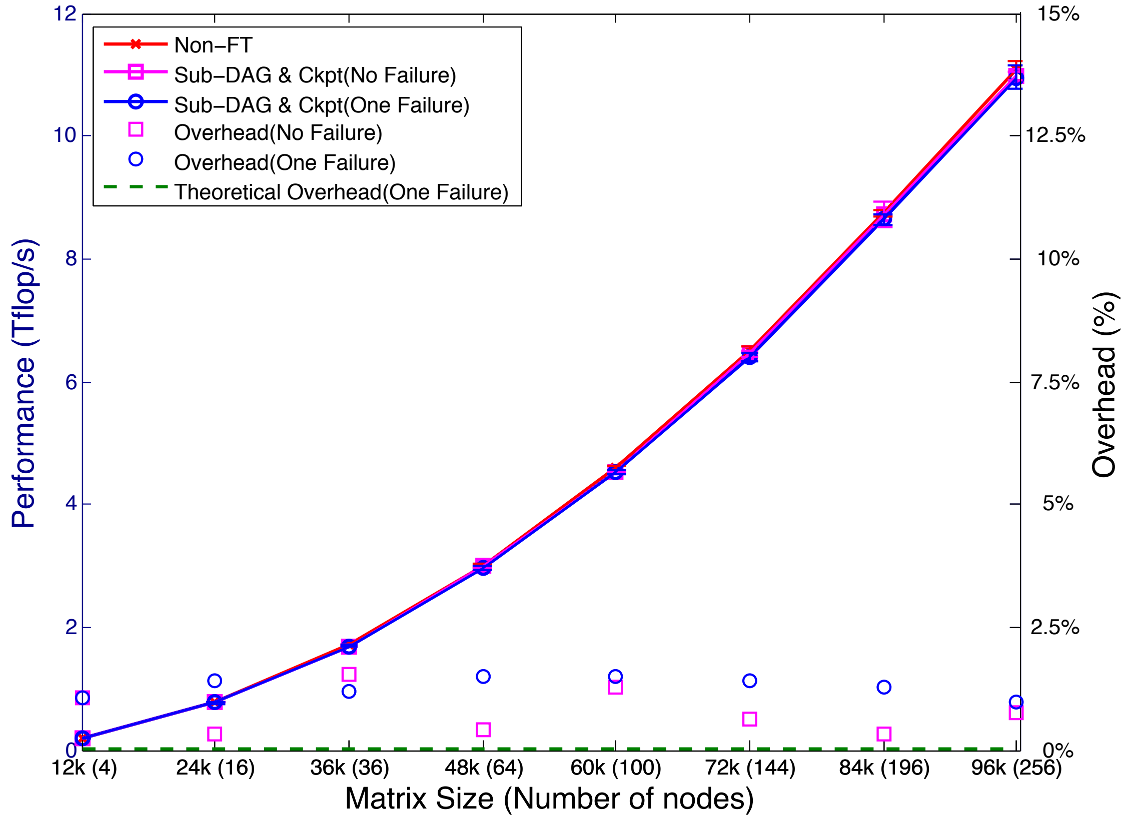
- System
  - Titan @ ORNL
  - Use 256 nodes (weak scalability)
  - Use CPU section of the system, every node has a 16-core AMD Opteron 6274 CPU; we use 8 core per node. (2 cores share 1 FPU in AMD Opteron 6274)
  - GCC 4.8.2, Cray LibSci
  - When to inject failure: factorization goes to the middle column

Performance of Correcting Sub-DAG Mechanism



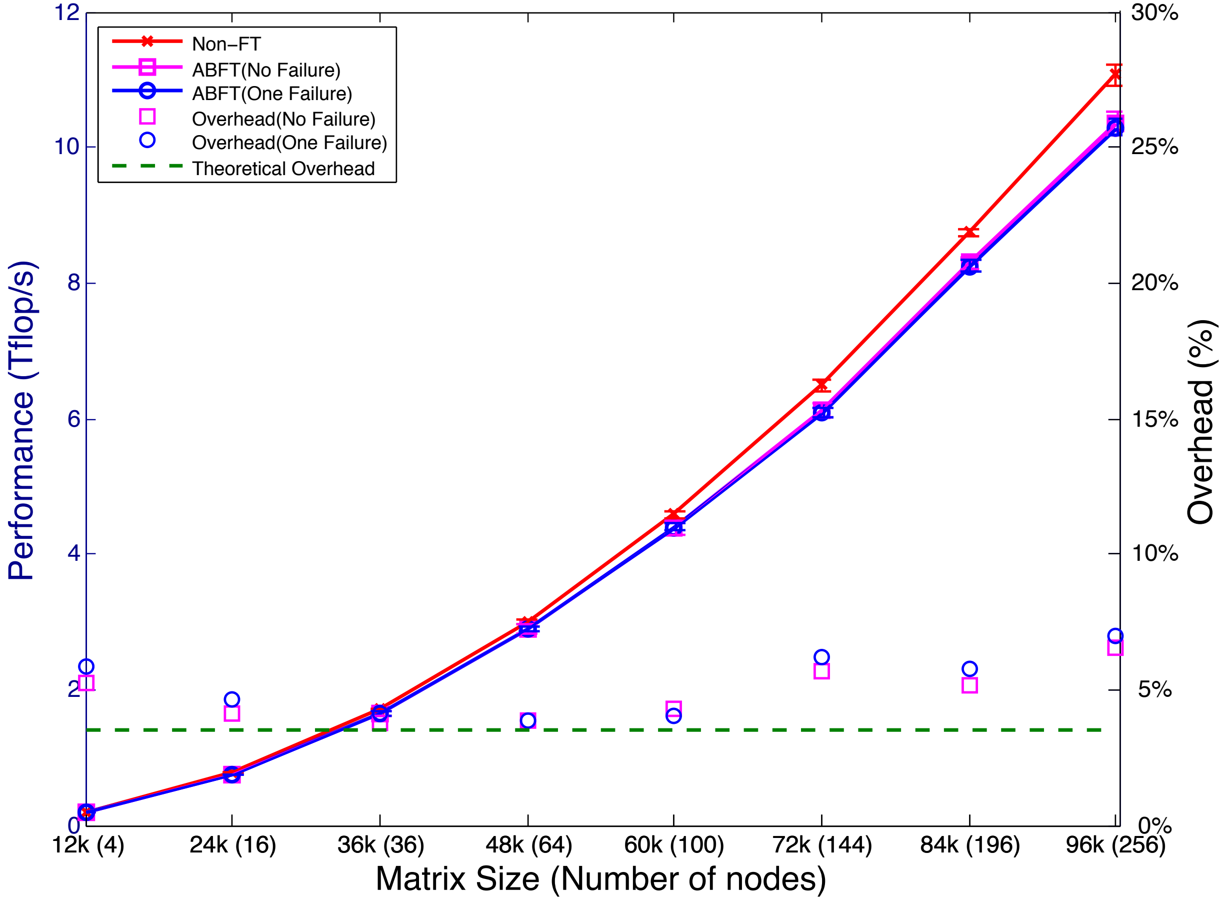
Perf. of Sub-DAG & Periodic Chpkt Mechanism

Set β=10



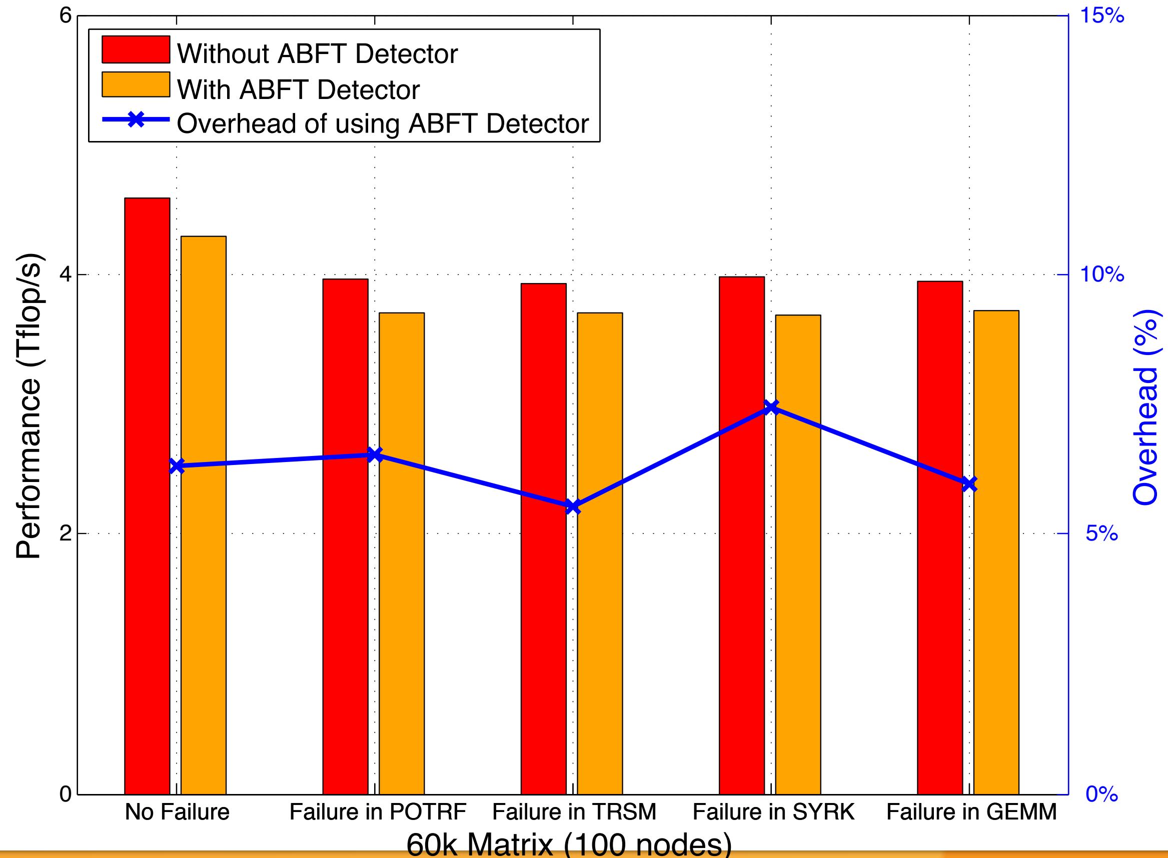
Performance of ABFT Mechanism

(*Detector included)



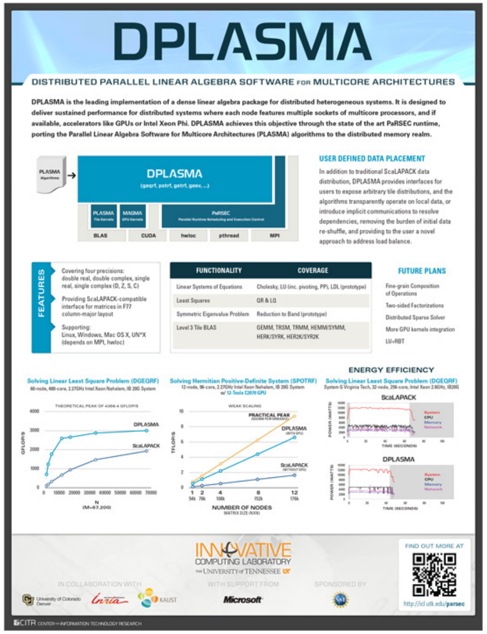
Overhead of Detection Mechanism

- Using ABFT as a detection mechanism for the correcting sub-DAG approach without failures and with one failure.



DPLASMA VS ScaLAPACK

- Similar overhead of adding FT support
  - ScaLAPACK: FT overhead is around 5%
    (*FT-ScaLAPACK: correcting soft errors on-line for ScaLAPACK cholesky, QR, and LU factorization routines*, Panruo Wu, Zizhong Chen, HPDC'14)
- DPLASMA more attractive for users
  - FT feature
  - Higher performance
    - Dynamic Scheduling
    - Accelerators Support



Conclusion & Future work

- *Conclusion*
  - *Two levels of granularity design, three mechanisms*
  - *Straightforward to port to support other DPLASMA routines*
    - *Two application level mechanisms can be implemented in PaRSEC functions.*
    - *ABFT mechanism has similar idea to extend to other DLA routines.*

- *Future work*
    - Integrate with accelerators
    - Extend to support fail stop model (hard error)

- *Questions ?*