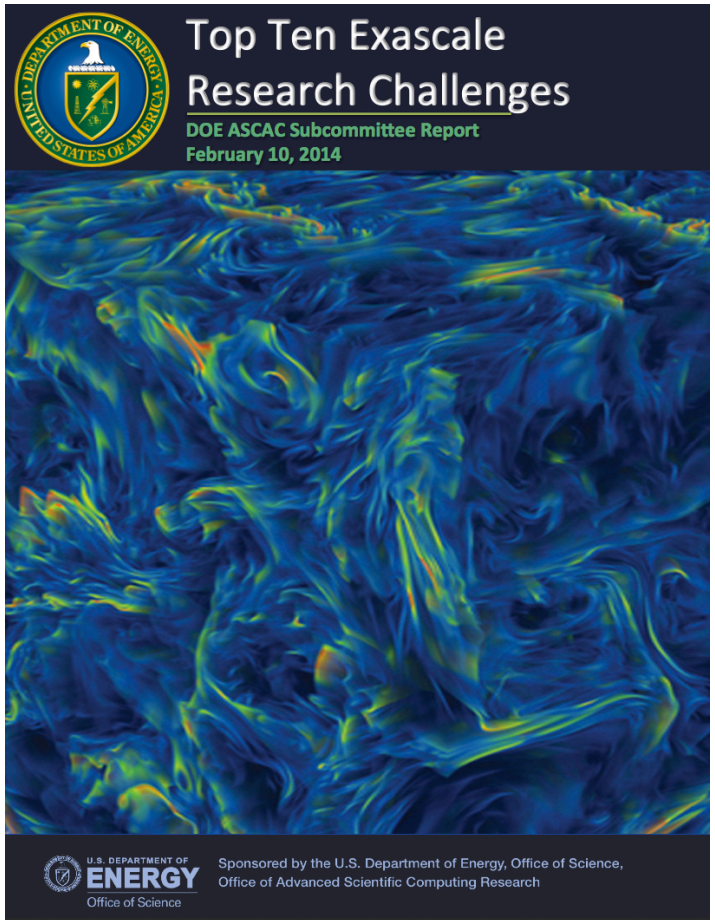
Fault Tolerance in Numerical Library Routines

Jack Dongarra

University of Tennessee
Oak Ridge National Laboratory
University of Manchester

Top 10 Challenges to Exascale

In a recent report U.S. Department of Energy identified ten research challenges (Google "Top 10 Challenges to Exascale")



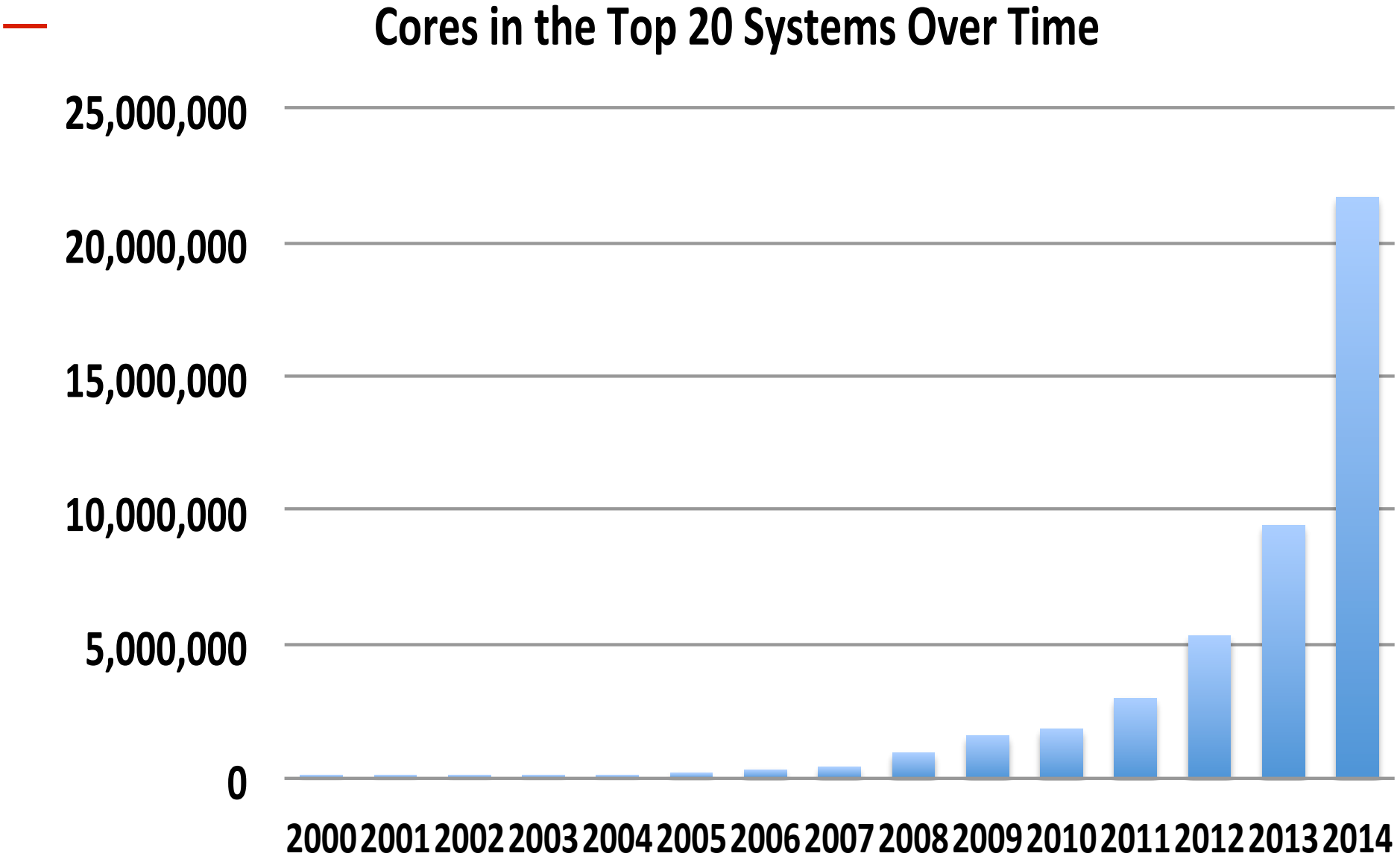
ASCAC Subcommittee for the Top Ten Exascale Research Challenges

Subcommittee Chair
Robert Lucas (University of Southern California, Information Sciences Institute)

Subcommittee Members
James Ang (Sandia National Laboratories)
Keren Bergman (Columbia University)
Shekhar Borkar (Intel)
William Carlson (Institute for Defense Analyses)
Laura Carrington (UC, San Diego)
George Chiu (IBM)
Robert Colwell (DARPA)
William Dally (NVIDIA)
Jack Dongarra (U. Tennessee)
Al Geist (ORNL)
Gary Grider (LANL)
Rud Haring (IBM)
Jeffrey Hittinger (LLNL)
Adolfy Hoisie (PNLL)
Dean Klein (Micron)
Peter Kogge (U. Notre Dame)
Richard Lethin (Reservoir Labs)
Vivek Sarkar (Rice U.)
Robert Schreiber (Hewlett Packard)
John Shalf (LBNL)
Thomas Sterling (Indiana U.)
Rick Stevens (ANL)

Top 10 Challenges to Exascale

1. **Energy efficiency:**
   - Creating more energy efficient circuit, power, and cooling technologies.
2. **Interconnect technology:**
   - Increasing the performance and energy efficiency of data movement.
3. **Memory Technology:**
   - Integrating advanced memory technologies to improve both capacity and bandwidth.
4. **Scalable System Software:**
   - Developing scalable system software that is power and resilience aware.
5. **Programming systems:**
   - Inventing new programming environments that express massive parallelism, data locality, and resilience
6. **Data management:**
   - Creating data management software that can handle the volume, velocity and diversity of data that is anticipated.
7. **Exascale Algorithms:**
   - Reformulating science problems and refactoring their solution algorithms for exascale systems.
8. **Algorithms for discovery, design, and decision:**
   - Facilitating mathematical optimization and uncertainty quantification for exascale discovery, design, and decision making.
9. **Resilience and correctness:**
   - Ensuring correct scientific computation in face of faults, reproducibility, and algorithm verification challenges.
10. **Scientific productivity:**
    - Increasing the productivity of computational scientists with new software engineering tools and environments.
    - Unless researcher productivity increases, the time to solution may be dominated by application development, not computation.

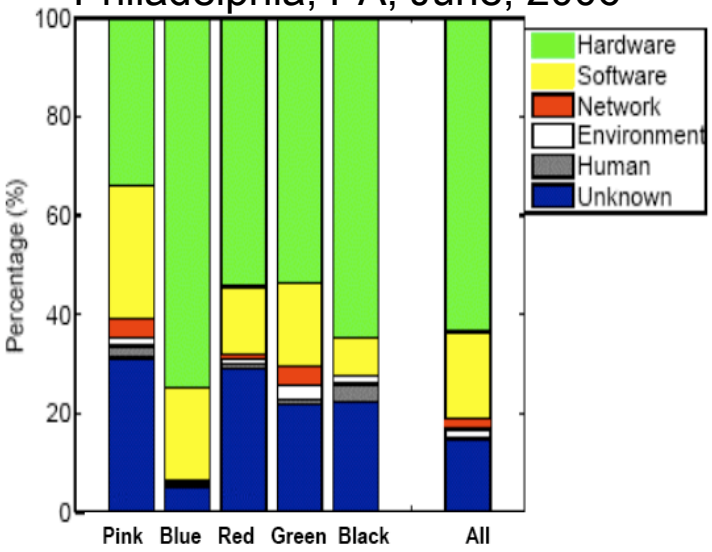

Cores in the Top 20 Systems Over Time
25,000,000
20,000,000
15,000,000
10,000,000
5,000,000
0
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014

Motivation

Types of faults:

- Power outage (voltage regulator failure, etc.)
- Hard errors (broken component: memory, network, core, disk, etc.)
- Soft errors (bit flip in memory, logic, bus)
- OS error (buffer overrun, deadlock, etc.)
- System Software error (service malfunction, tin
- Administrator error (Human)
- User errors (Human)

G. Gibson, Proc of the DSN2006, Philadelphia, PA, June, 2006



Relative frequency of root cause by system type.

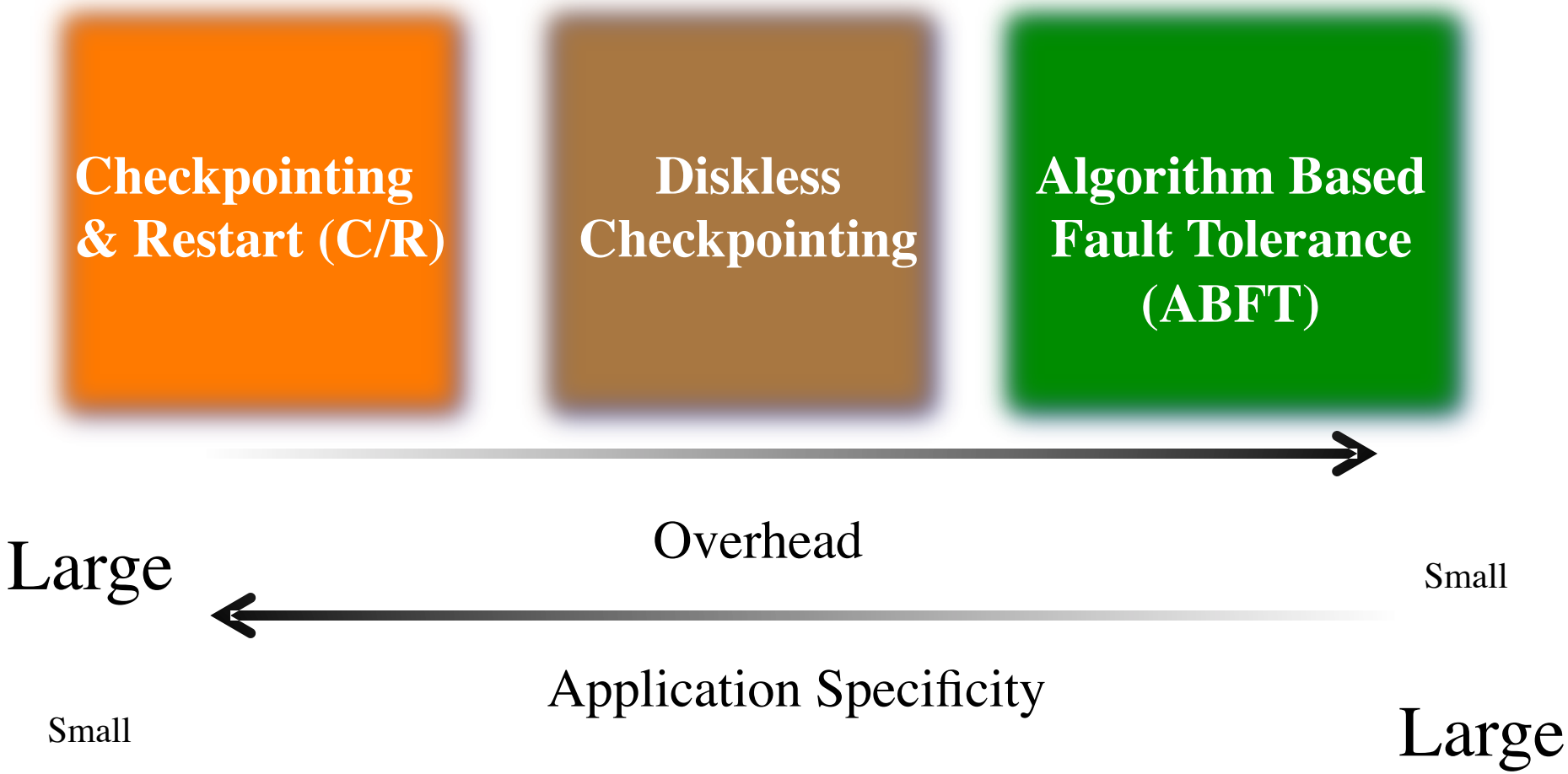
Classes of faults:

- Detected and corrected (by ECC, Replication, Re-execution)
- Detected and uncorrectable (leading to application crash: fail stop)
- Undetected (leading to data corruption, application hang, etc.)

Problem Definition

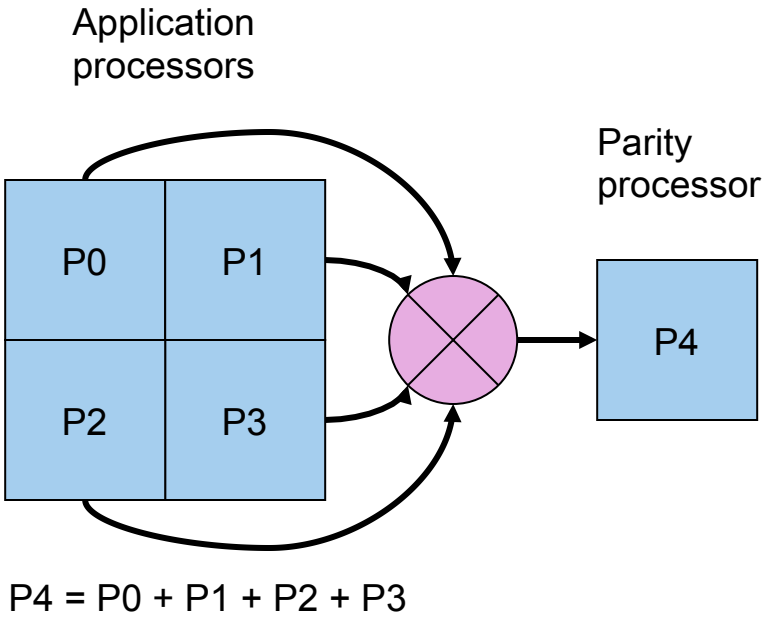
- System Errors
  - Hard Error (single, multiple)
  - Soft Error (single, multiple) (not discuss)
    - Bit flip effect
- Dense Linear Algebra
  - Direct methods: LU, Cholesky, QR, eigenvalue problems, …
  - Iterative methods
- Platforms/Applications
  - Distributed memory system
    - Eg. MPI, ScaLAPACK
  - Hybrid system with the GPGPU

Research Status Anatomy



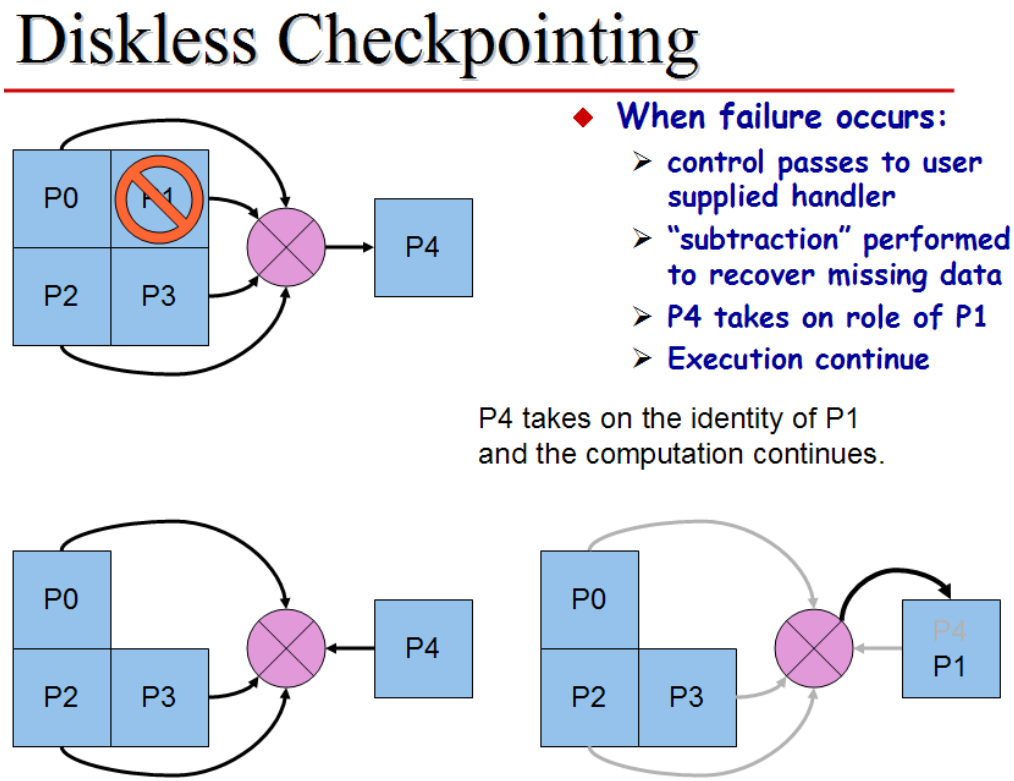
Diskless Checkpointing

- The p application processors (4 in this case) each maintain their own checkpoints locally.
- k extra processors maintain coding information so that if 1 or more processors fail, they can be replaced.
- Will describe for k=1 (parity)
- If a single processor fails, then its state may be restored from the remaining live processors



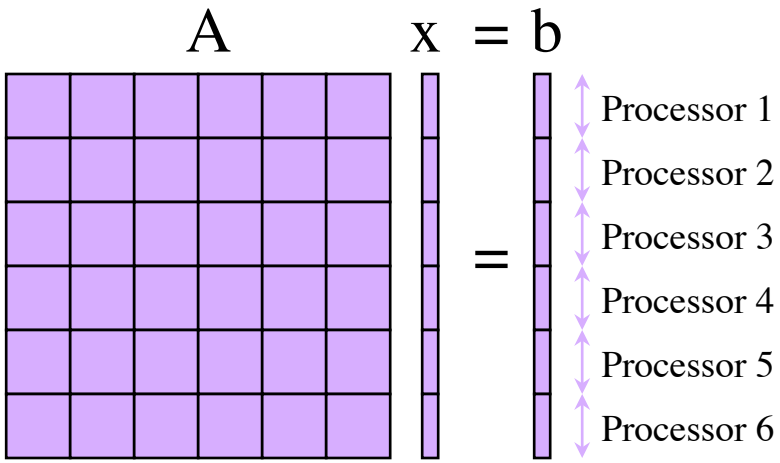
Three Ideas for Fault Tolerant Linear Algebra Algorithms

- **Lossless diskless check-pointing for iterative methods**
  - **Checksum maintained in active processors**
  - **On failure, roll back to checkpoint and continue**
  - **No lost data**



Lossy Algorithm : Basic Idea

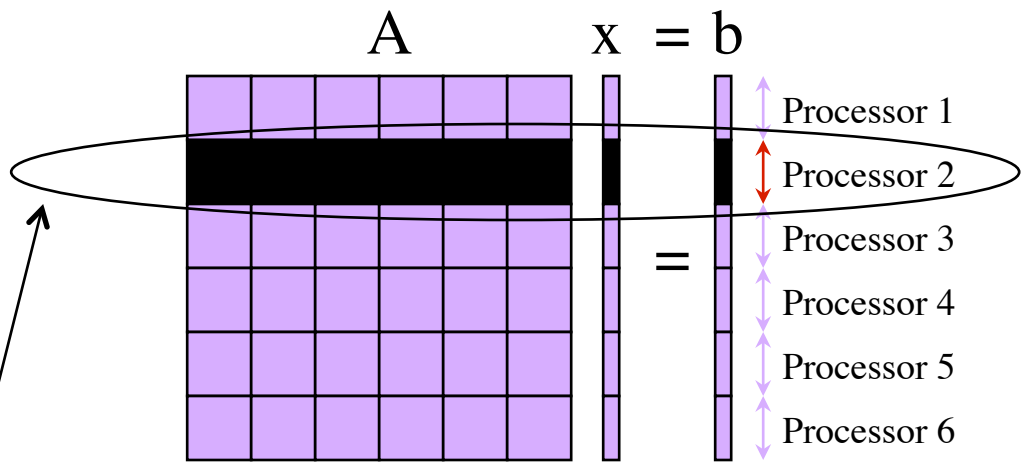
- Let us assume that the exact solution of the system Ax=b is stored on different processors by rows



- May work well for iterative methods

Lossy Algorithm : Basic Idea

- **Let us assume that the exact solution of the system Ax=b is stored on different processors by rows**

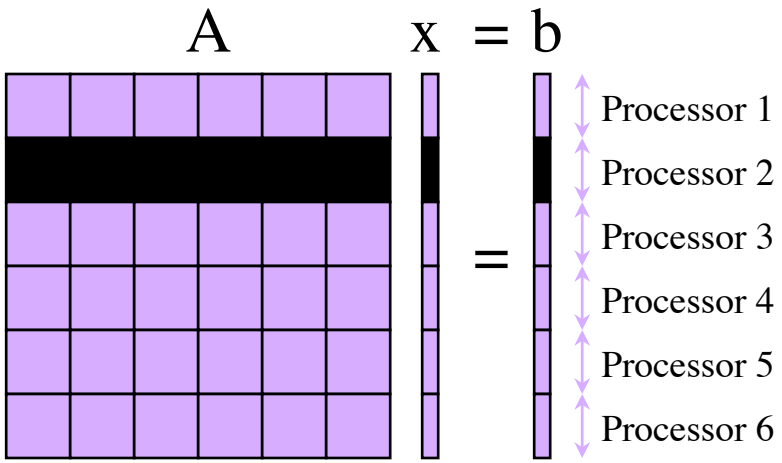


Processor 2 (e.g.) fails, all its data is lost.

How to recover the lost part of x in this case?

Lossy Algorithm : Basic Idea

- **Let us assume that the exact solution of the system Ax=b is stored on different processors by rows**

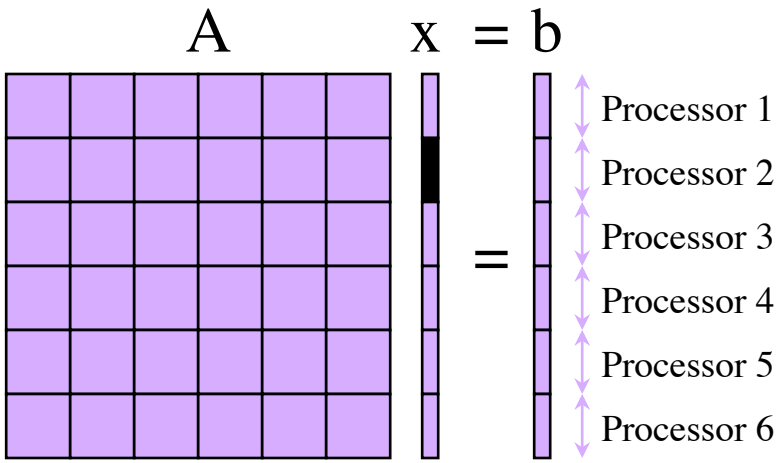


3 steps

Step 1: recover a processor and a running parallel environment (the job of the fault tolerant-MPI library)

Lossy Algorithm : Basic Idea

- **Let us assume that the exact solution of the system Ax=b is stored on different processors by rows**



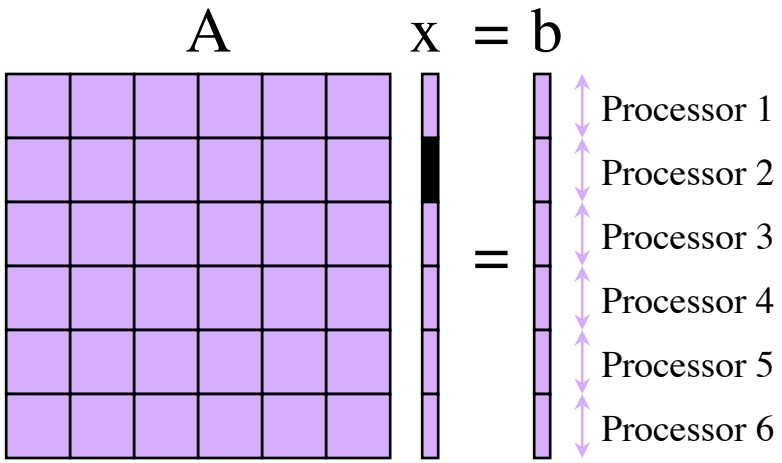
3 steps

Step 1: revover a processor and a running parallel environement (the job of the fault tolerant-MPI library)

Step 2: recover A_{21} A_{22}, …, A_{n2} and b_2 (the original data) on the failed processor

Lossy Algorithm : Basic Idea

- **Let us assume that the exact solution of the system Ax=b is stored on different processors by rows**



3 steps

Step 1: recover a processor and a running parallel environment (the job of the fault tolerant-MPI library)

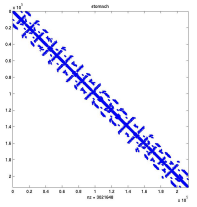
Step 2: recover A_{21} A_{22}, …, A_{n2} and b_2 (the original data) on the failed processor

Step 3: Notice that

$$A_{21} x_1 + A_{22} x_2 + \ldots + A_{2n} x_n = b_2)$$

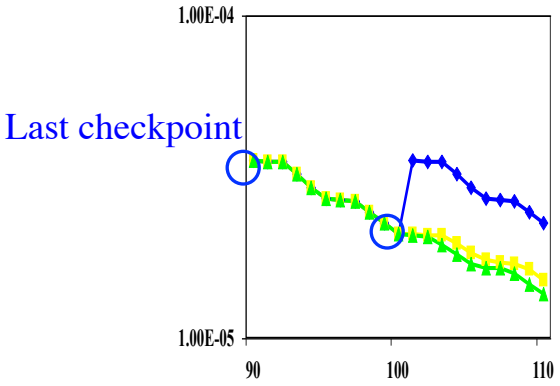
$$x_2 = A_{22}^{-1} (b_2 - \Sigma_{i \neq 2} A_{2i} x_i)$$

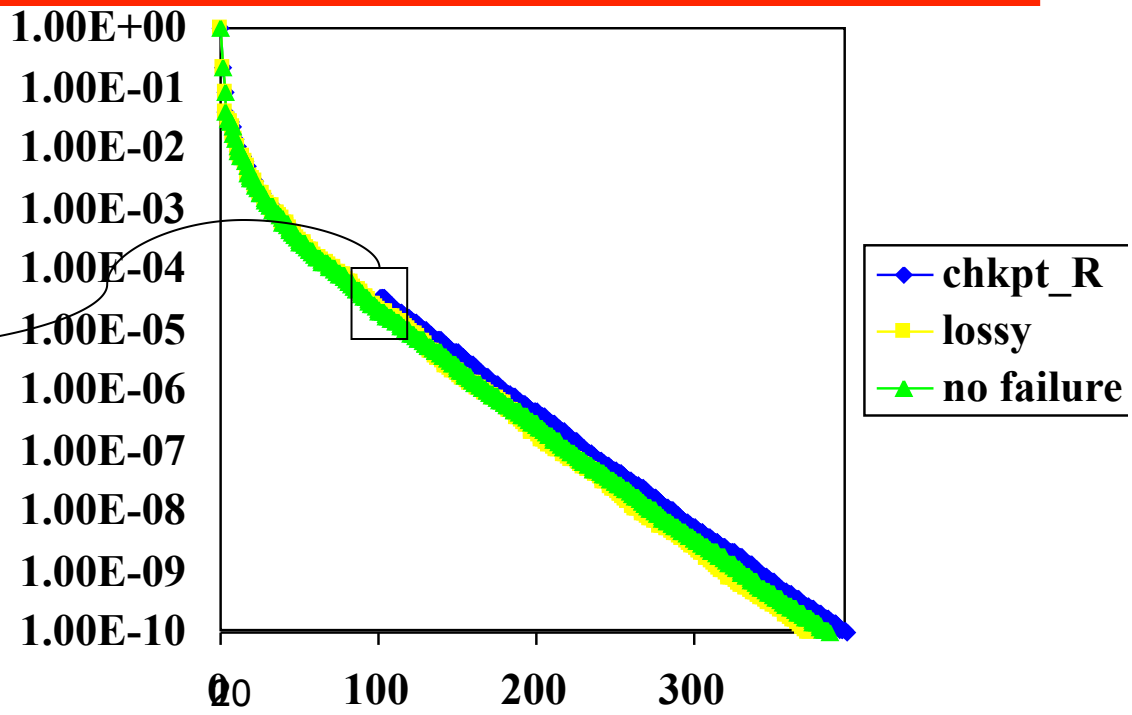
Using GMRES(30) Non Symetric Matrix



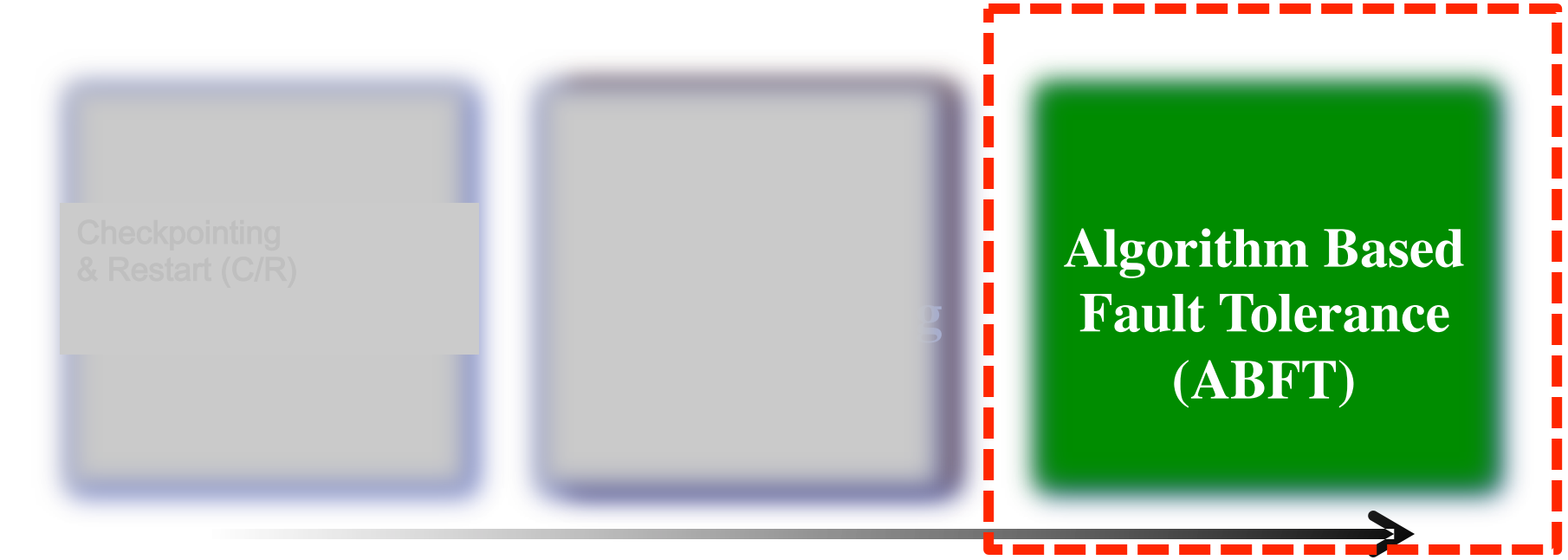
stomach; n=213,360; nnz=3,021,648; tol=10^{-10}; #procs=16; n_f=13,335; nnz=185,541									
recovery	$iter_f$	#iter	T_{Wall}	T_{Chkpt}	T_{Roll}	T_{Recov}	T_I	$T_{II,a,b}$	T_{III}
lossy	no	385	38.89						
$chkpt_R$	no	385	41.04	1.92					
lossy	100	372	42.38		1.56	5.38	1.03	0.33	3.91
$chkpt_R$	100	395	45.49	1.92	2.40	1.68	1.02	0.32	0.20

$chkpt_R$ every GMRES restart, 30 iterations





Research Status Anatomy



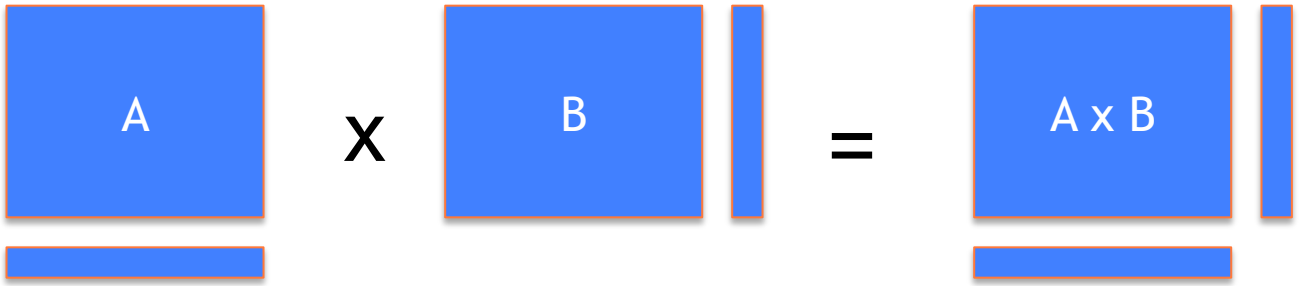
Motivation

- Algorithm Based Fault Tolerance
  - Handle failures at the application level
  - User knows more, can do better
- Mix of different techniques
  - Checkpointing (few data)
  - Transactions
  - Replication
  - Classical ABFT checksumming

Techniques for Error Protection and Failure Recovery

- **Algorithm-Based Fault Tolerance**
  - **KH Huang & Jacob Abraham, ABFT for Matrix Operations, IEEE Trans. Computers. 01/1984;**
    - **Implementation on systolic arrays**
  - **Takes advantage of additional mathematical relationship(s)**
    - **Already present in algorithm**
    - **Introduced (cheaply, if possible) by ABFT**
  - **Goal of this work is to do an implementation that could be carried out on a complete numerical library such as ScaLAPACK.**

Algorithm Based Fault Tolerance (ABFT)



$$\begin{bmatrix} A \\ G^T A \end{bmatrix} \times \begin{bmatrix} B & BG \end{bmatrix} = \begin{bmatrix} AB & ABG \\ G^T AB & G^T ABG \end{bmatrix}$$

G^T

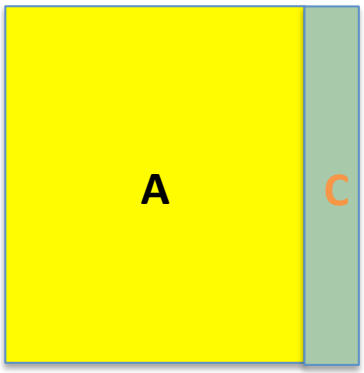
Algorithm-based fault tolerance for matrix operations, Huang, K.H. and Abraham, 1984

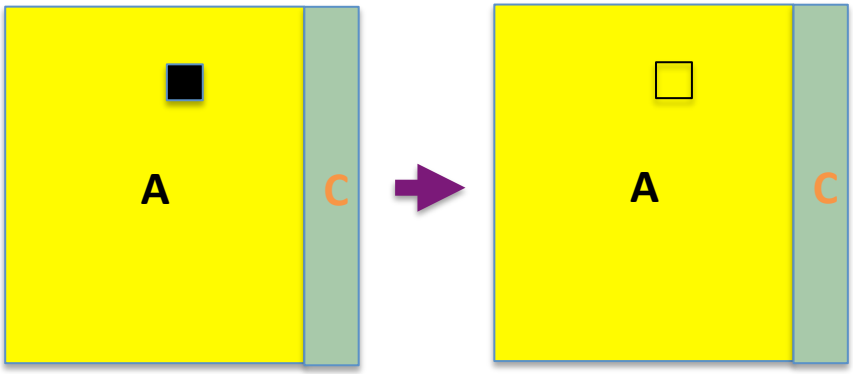
ABFT Idea

- C matrix contains a checksum (row summations) of A
- Checksums introduce redundancy (resilience) to the algorithm

$$C_i = \sum_j A_{ij}$$

$$A_{ij} = C_i - \sum_{k \neq j} A_{ik}$$





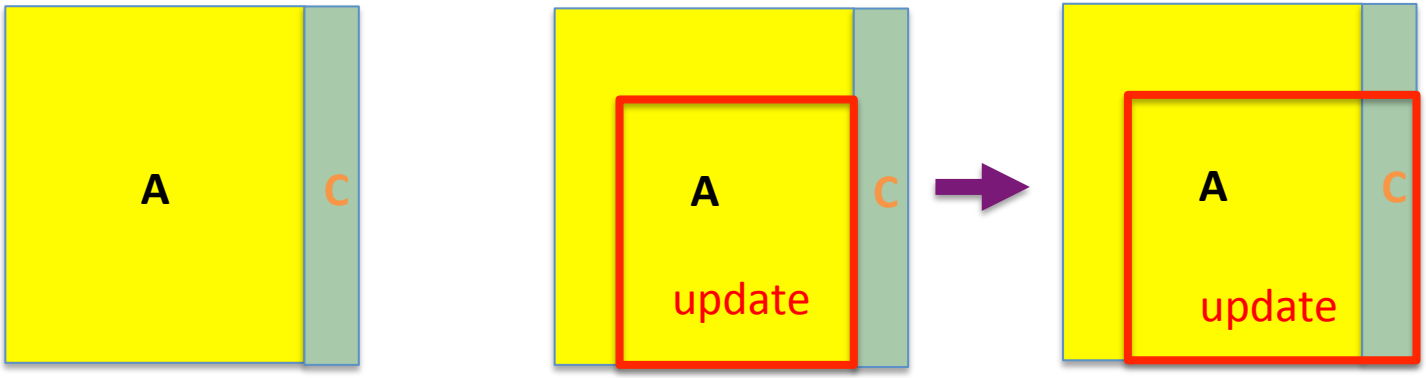
In case of failure, checksum inversion allows to restore the missing value

ABFT Idea

- C matrix contains a checksum (row summations) of A
- Checksums remain mathematically invariant!

$$C_i = \sum_j A_{ij}$$

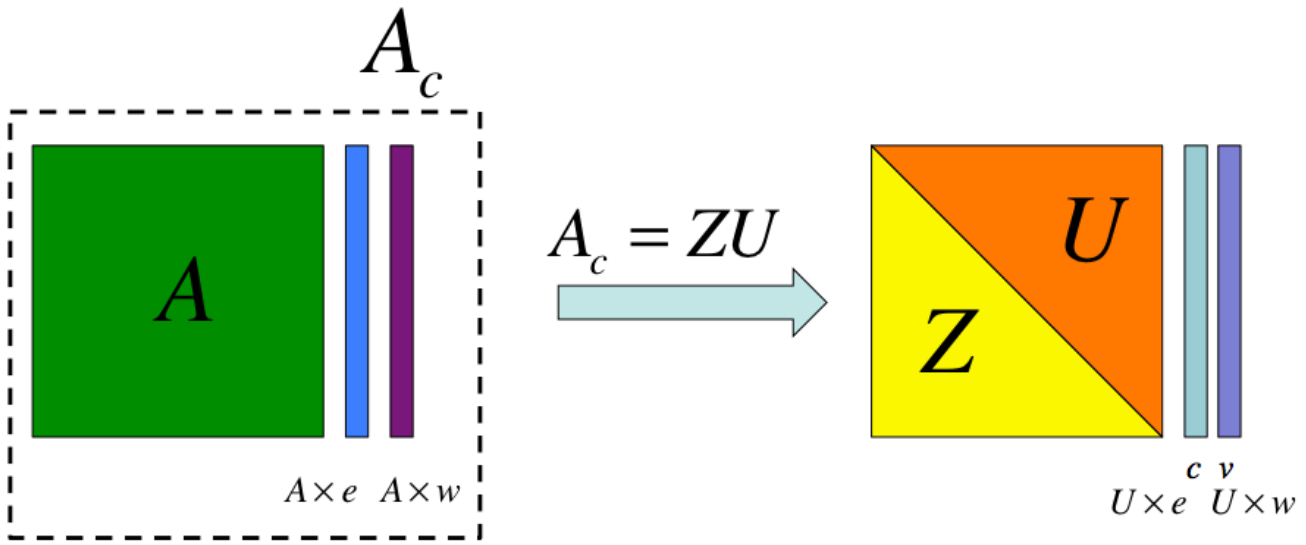
*The **Same algorithm** updates both the **trailing matrix AND** the **checksums***



$$Update(\sum_j A_{ij}) = \sum_j Update(A_{ij})$$

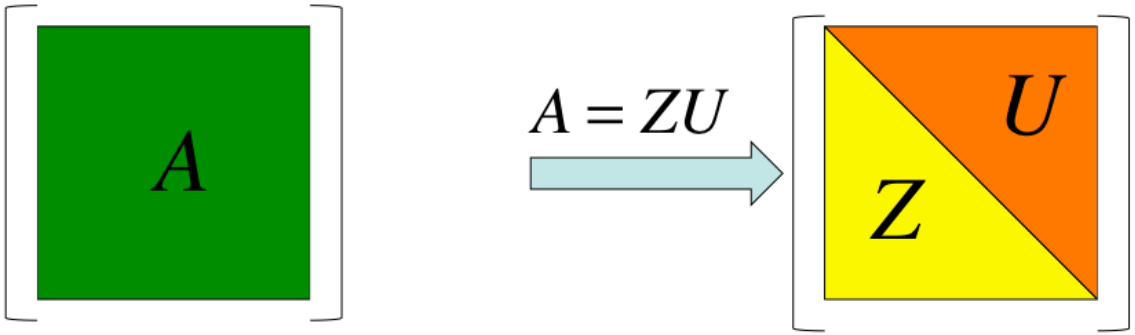
Algorithm Based Fault Tolerance

- Before the factorization starts, a checksum is taken and Algorithm Based Fault Tolerance (ABFT) is used to carry the checksum along with the computation.



Note: when ZU is QR factorization, Z is not lower triangular

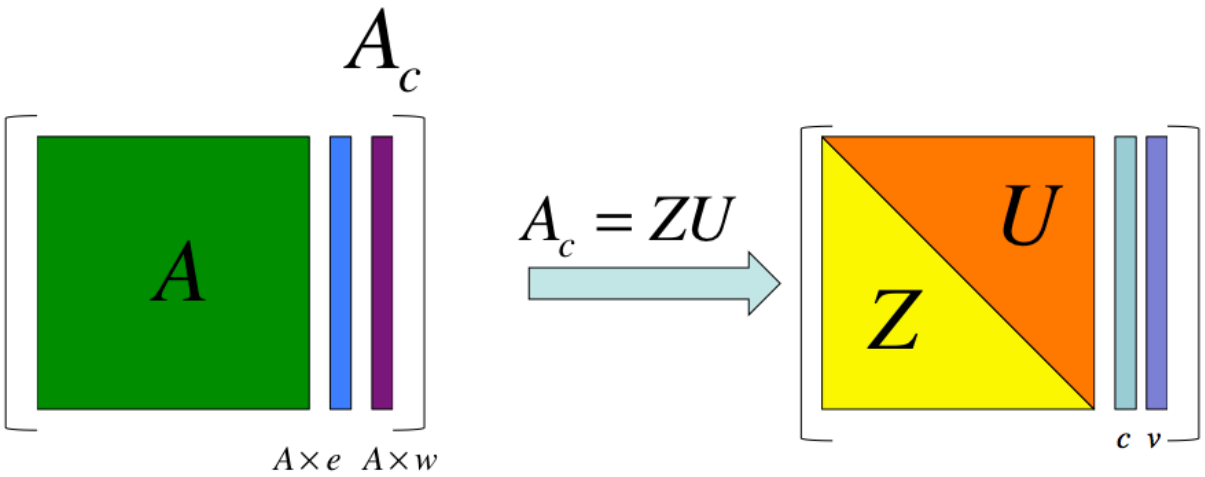
Checksum in ABFT



Generator, *G =(e, w)*

$$e = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \quad w = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix}$$

w_i random number



$$A_c = (A, Ae, Aw)$$

$$A_c = Z(U, c, v)$$

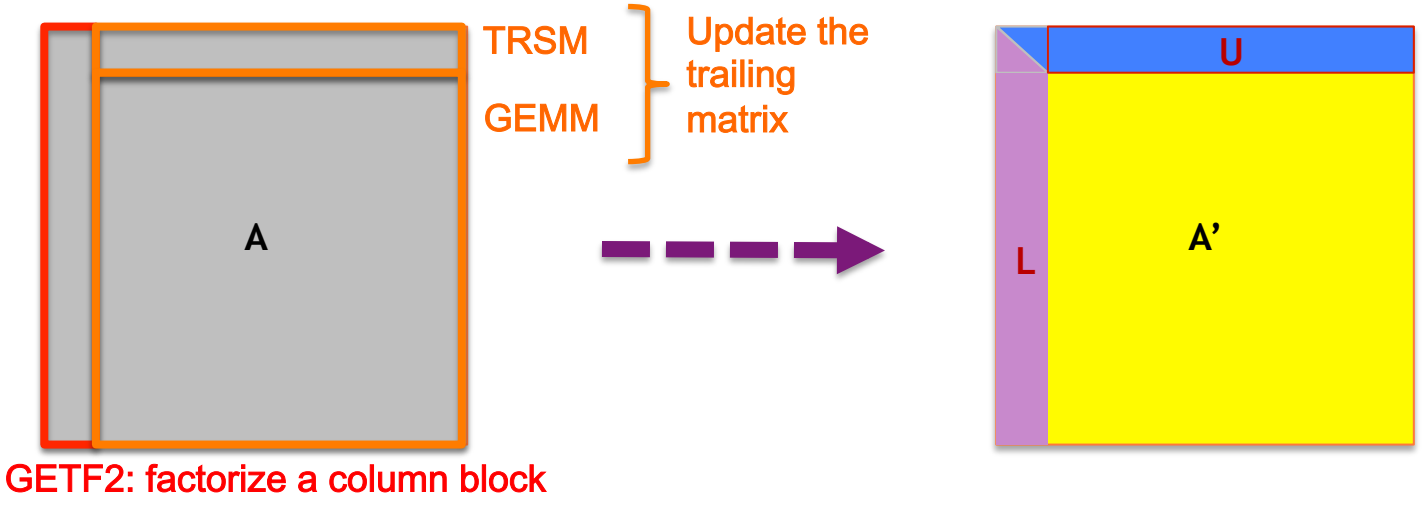
$$c = Ue$$

$$v = Uw$$

Dense Factorization

$$\begin{bmatrix} L & A & P & A & C & K \\ L & -A & P & -A & C & -K \\ L & A & P & A & -C & -K \\ L & -A & P & -A & -C & K \\ L & A & -P & -A & C & K \\ L & -A & -P & A & C & -K \end{bmatrix}$$

- **Recursive block LU (ScaLapack)**
  - Want to solve Ax=b (hard)
  - Transform A into LU factorization
  - Solve Ly=Pb, then Ux=y (easy)

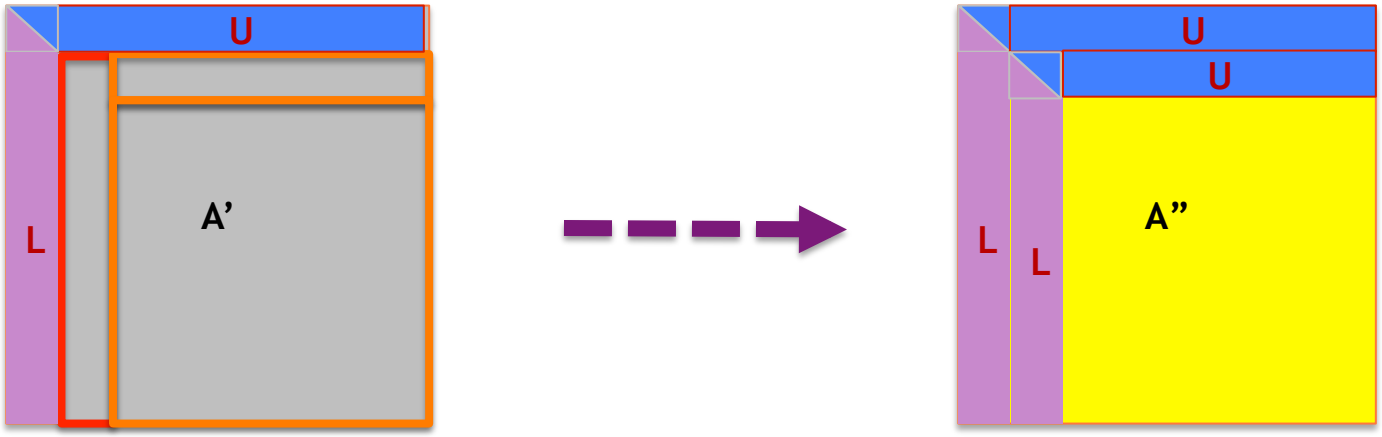


GETF2: factorize a column block

Dense Factorization

L	A	P	A	C	K
L	-A	P	-A	C	-K
L	A	P	A	-C	-K
L	-A	P	-A	-C	K
L	A	-P	-A	C	K
L	-A	-P	A	C	-K

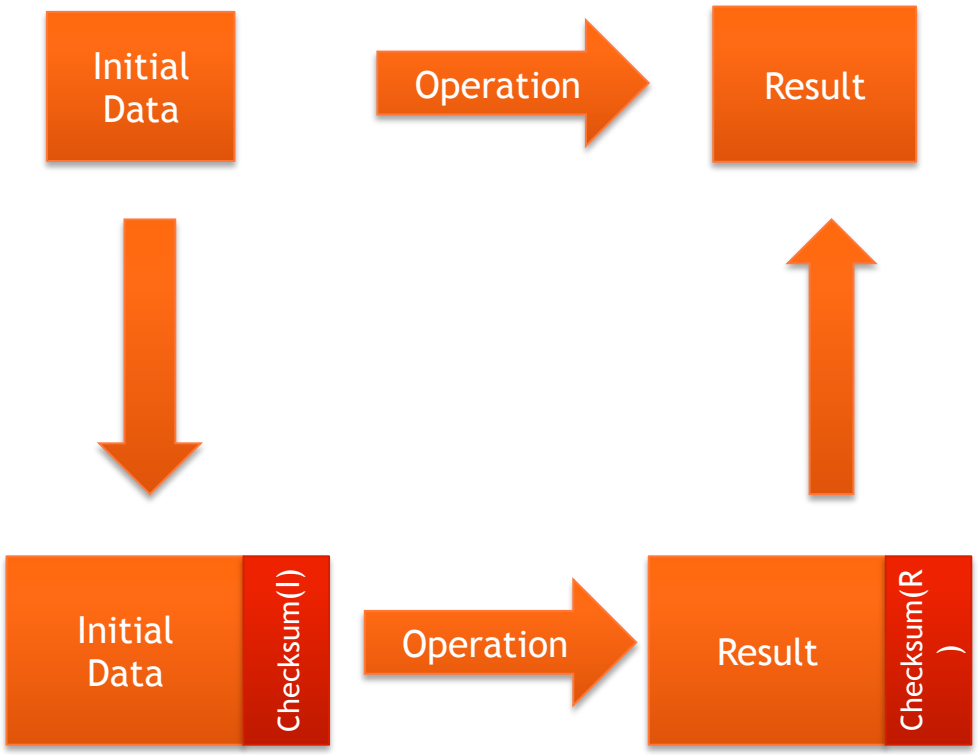
- **Recursive block LU (ScaLapack)**
  - Want to solve Ax=b (hard)
  - Transform A into LU factorization
  - Solve Ly=Pb, then Ux=y (easy)



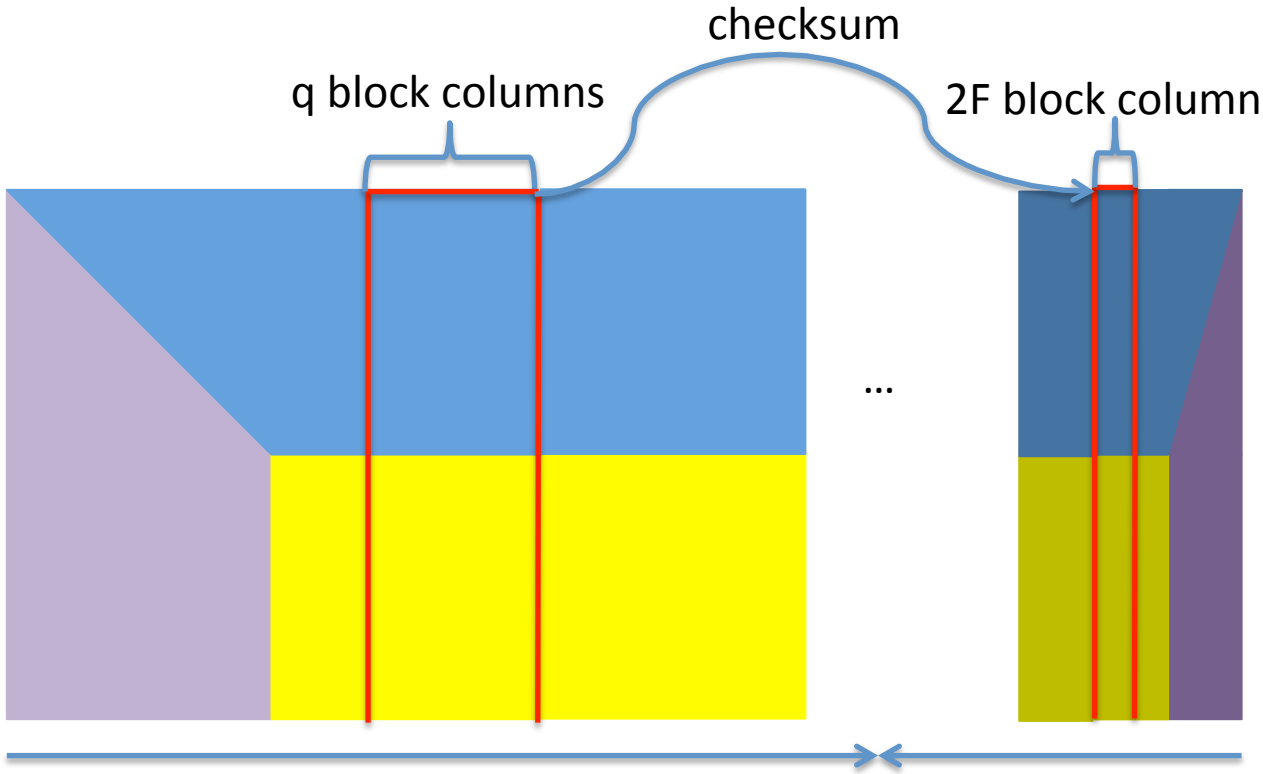
Principle of ABFT

- Data is distributed
- Result is distributed
- The operation preserves the checksum properties

- Apply the operation on Data + Checksum
- In case of failure, recover the missing data by inverting the checksum



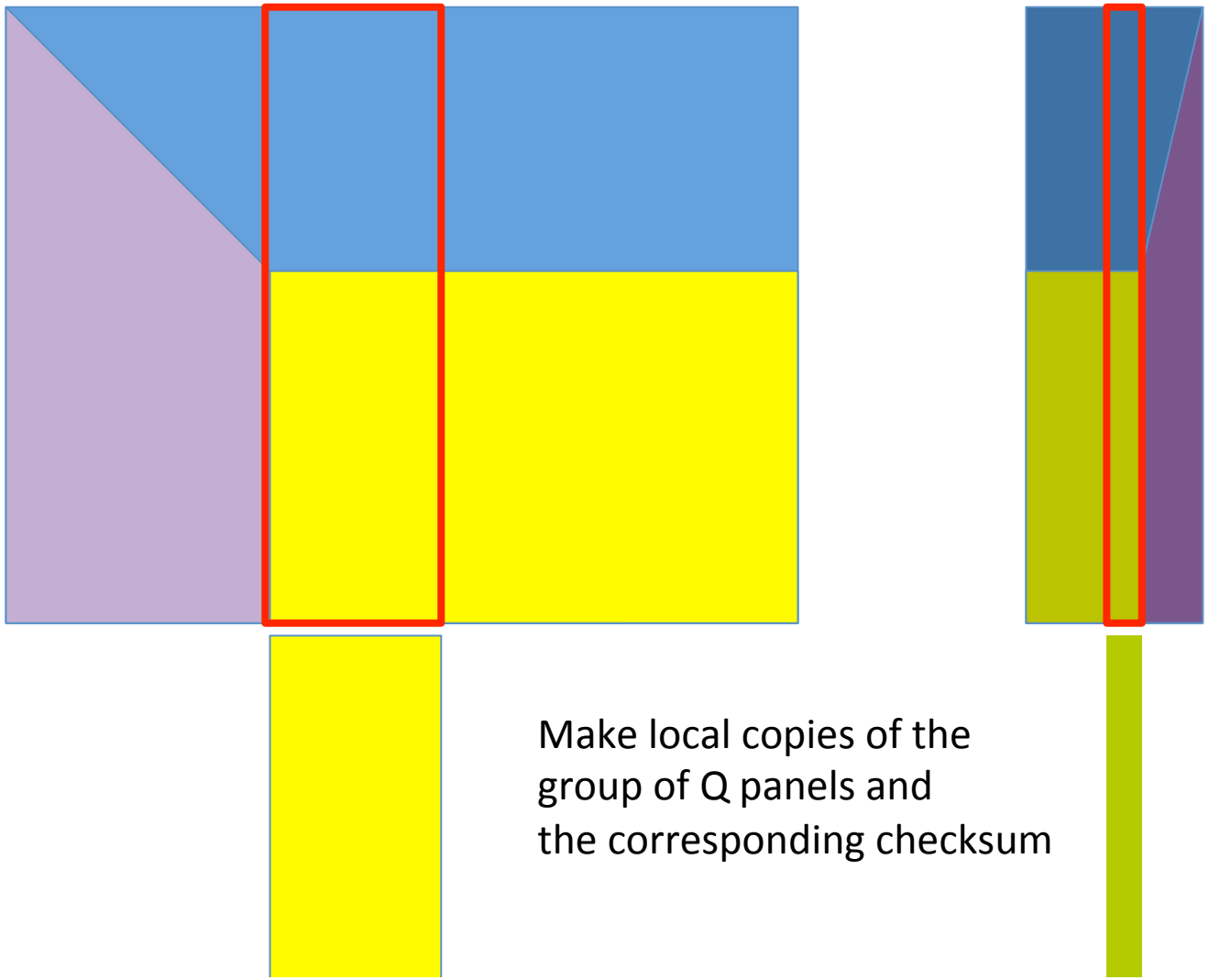
Reverse Neighboring Scheme



The matrix of size M x N,
Blocked in blocks of mb x nb
And distributed over a 2D process grid of pxq
is extended with $\frac{2F \times N}{q \times nb}$ block columns to store a checksum
that will allow to tolerate up to F simultaneous faults on the same row

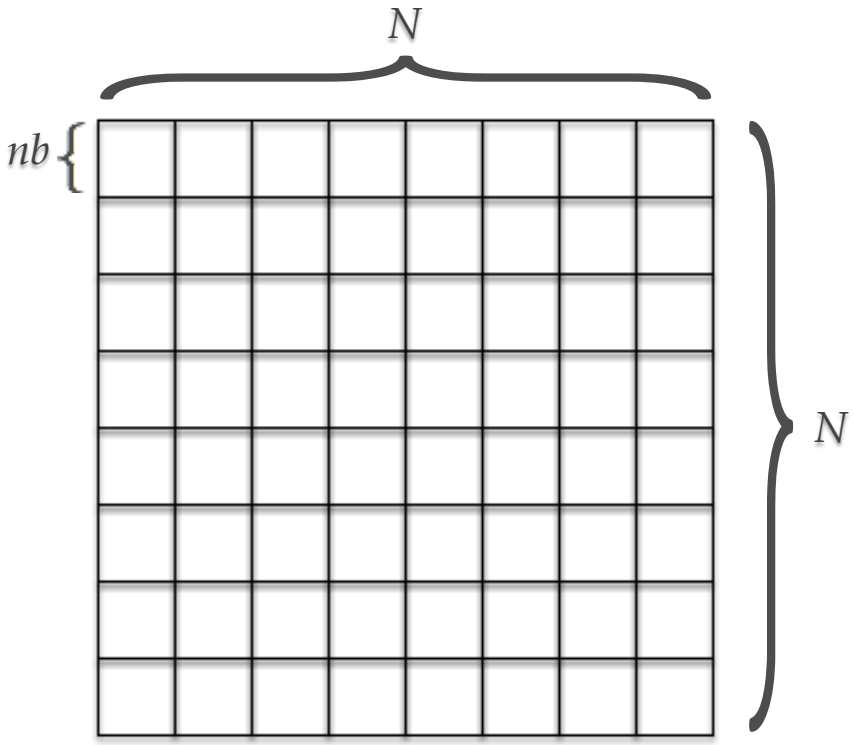
Group of Q panels

Corresponding Checksum

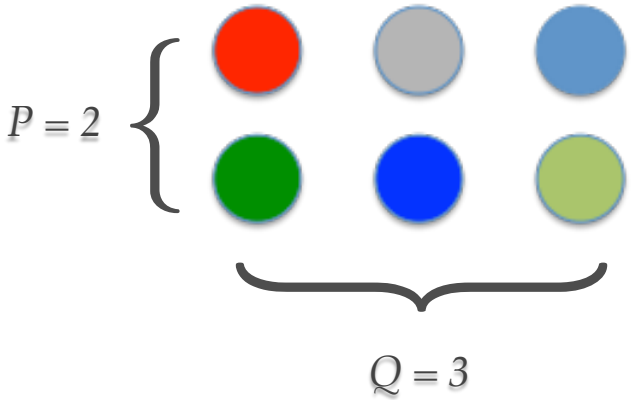


Make local copies of the
group of Q panels and
the corresponding checksum

2-D Block Cyclic Data Distribution

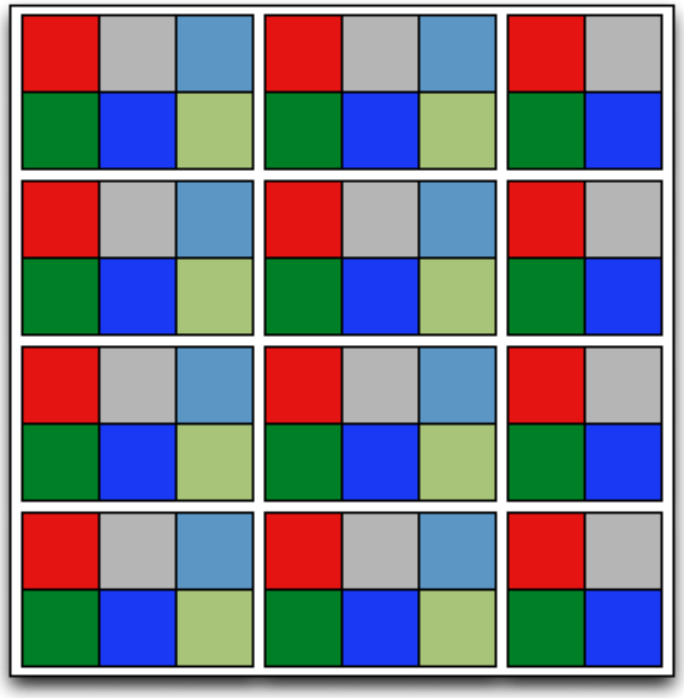


An NxN matrix partitioned into $nb \times nb$ blocks

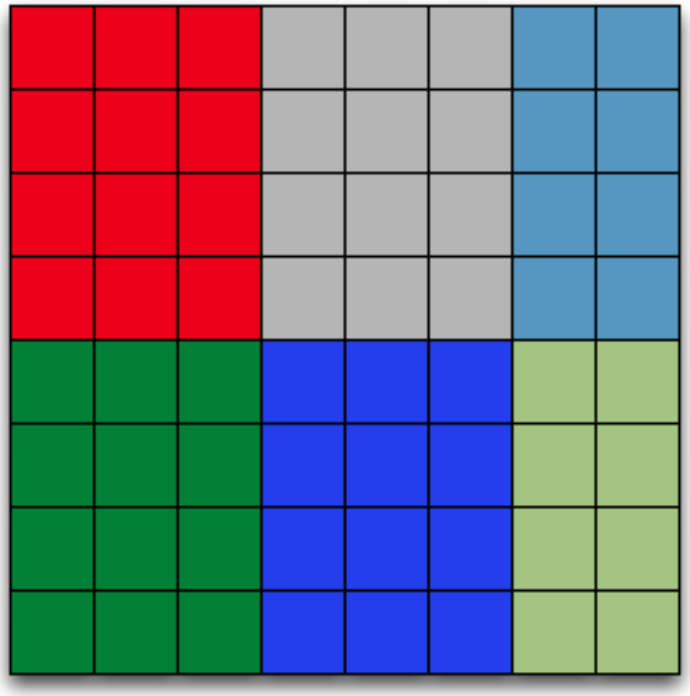


A PxQ processor gird (P = 2, Q = 3)

2-D Block Cyclic Data Distribution



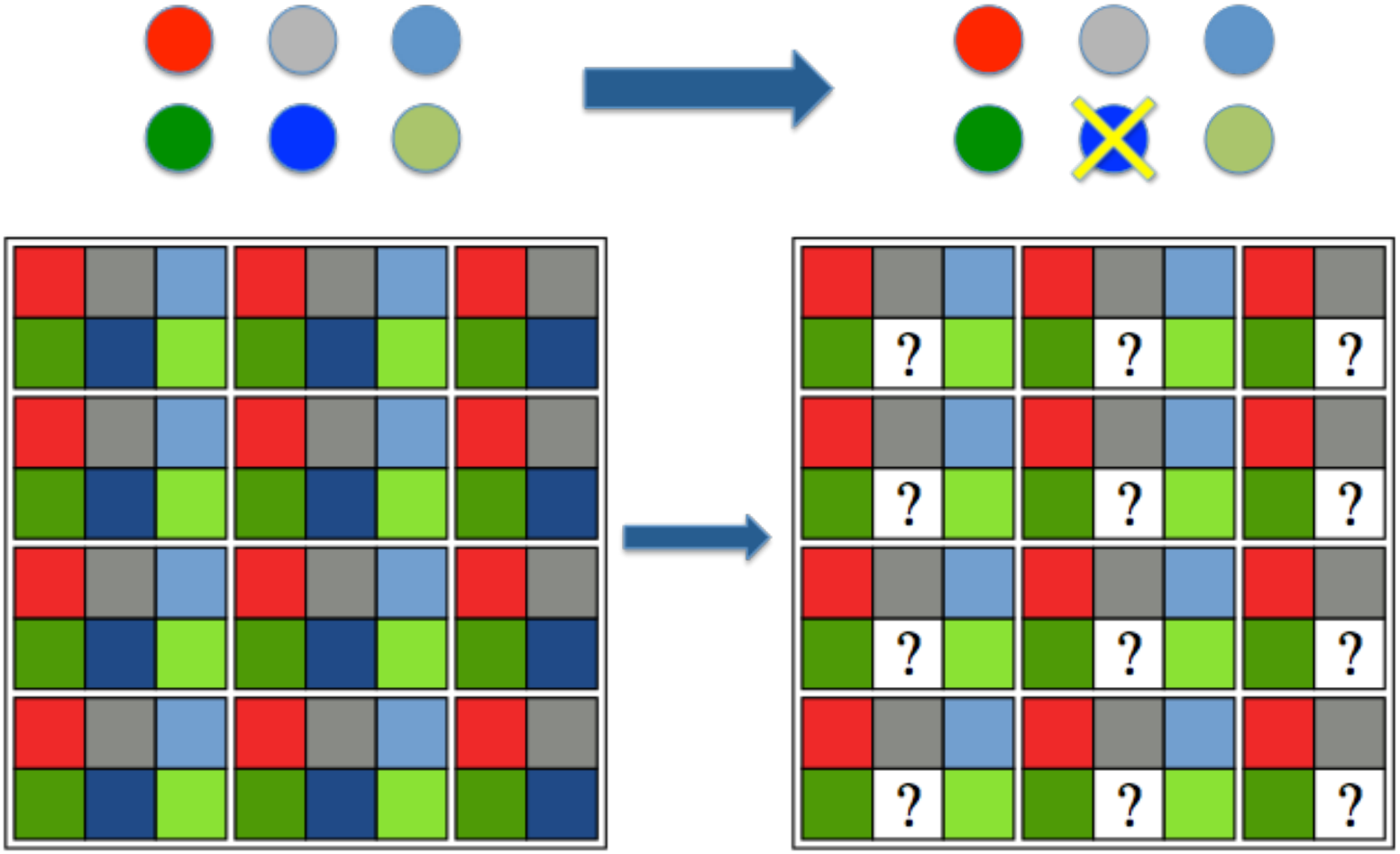
Matrix view



Process view

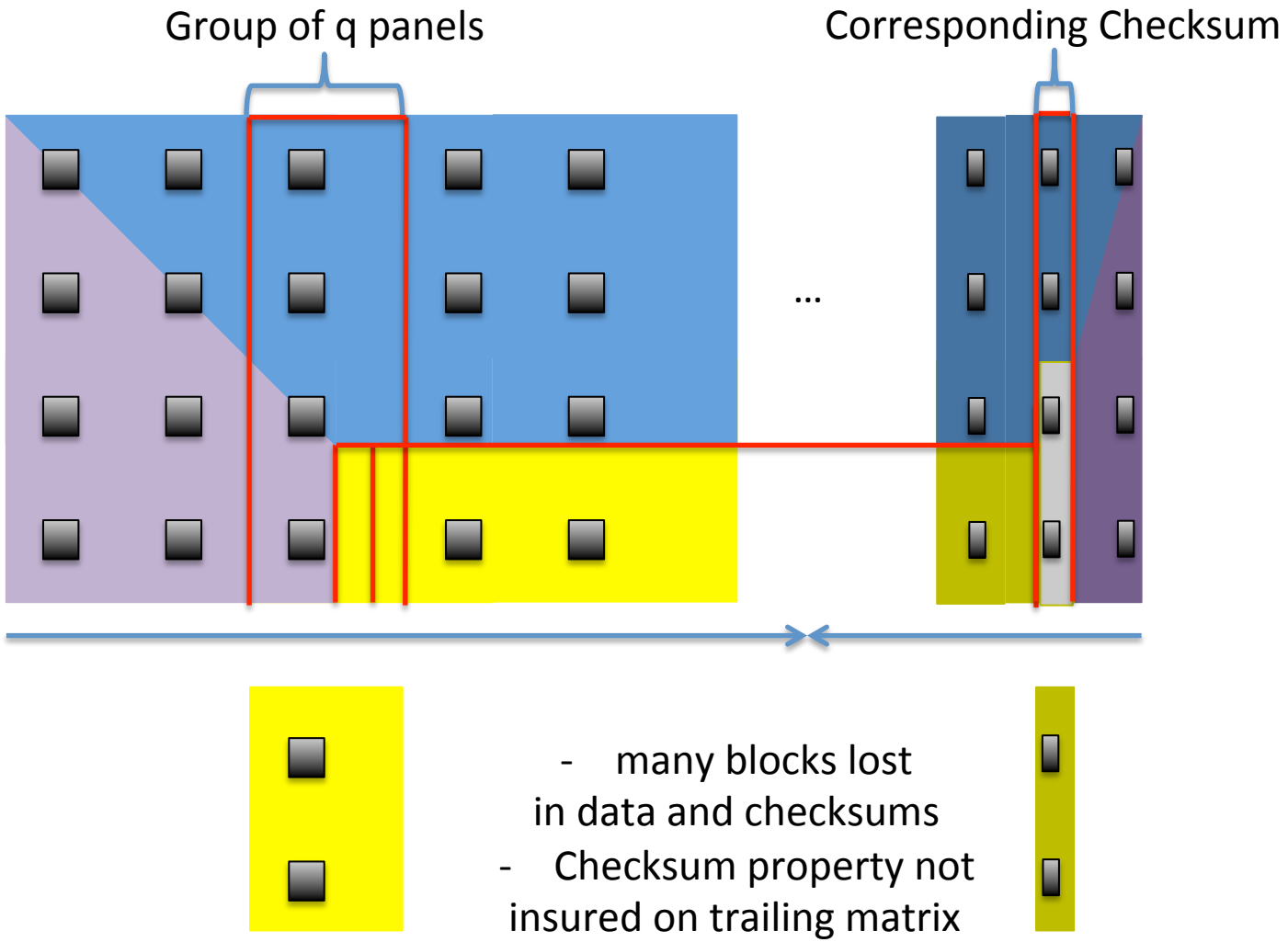
P x Q process grid (2x3)

Failure Model



31

In case of Failure



ABFT Overheads

Matrix M x N, Blocks mb x nb, Process grid p x q

F: maximum number of ***simultaneous*** failures tolerated

Memory

$$O(\frac{F}{q} \times M \times N)$$

Matrix is extended with 2F columns every q columns

N.B. Usually F << q

Relative overheads in F/q

e.g. 2 simultaneous faults on 192x192 process grid => 1% overhead

Computation

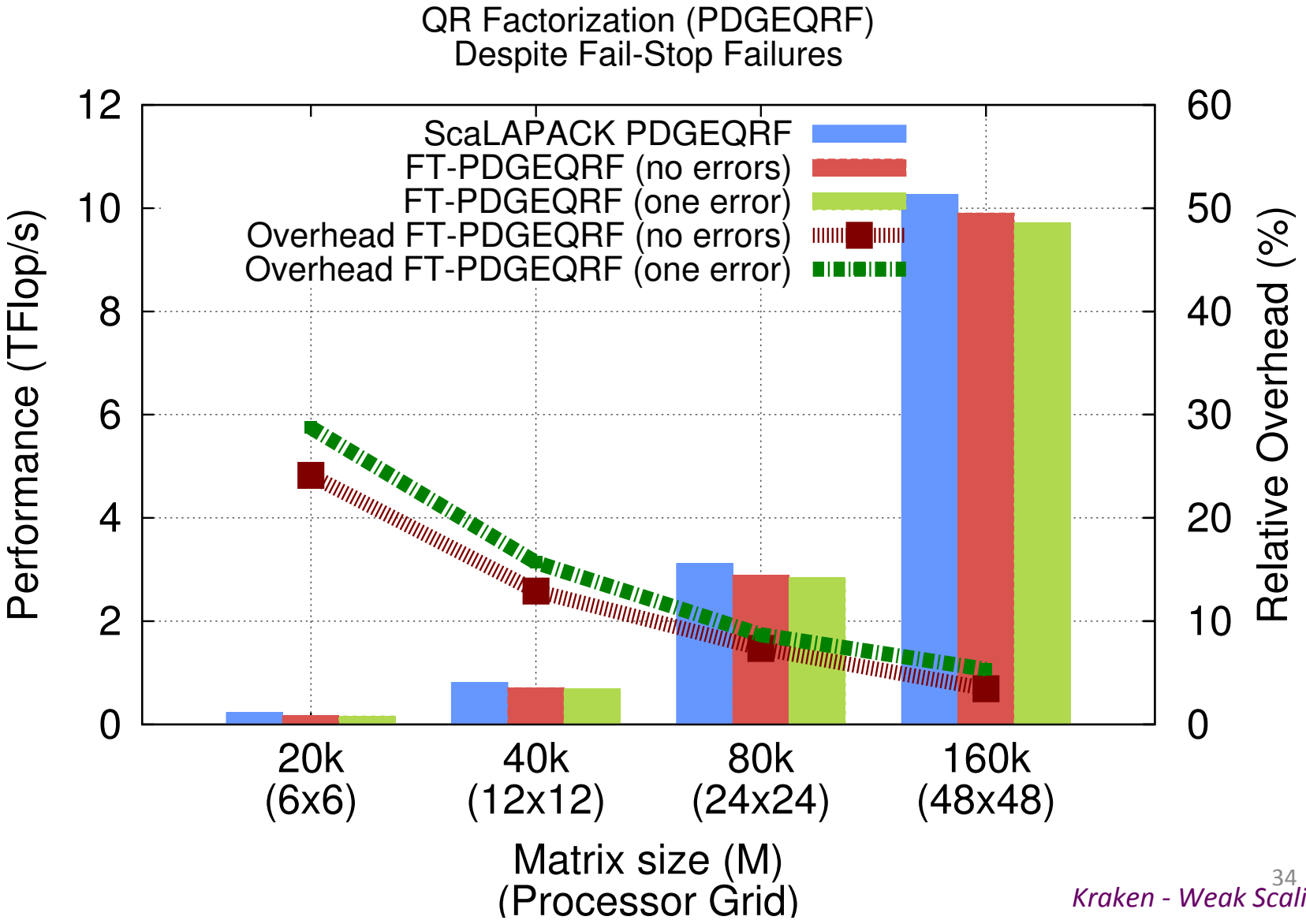
$$O(\frac{F}{q} \times M^3)$$

flops for the checksum update, and

$$O(MN)$$

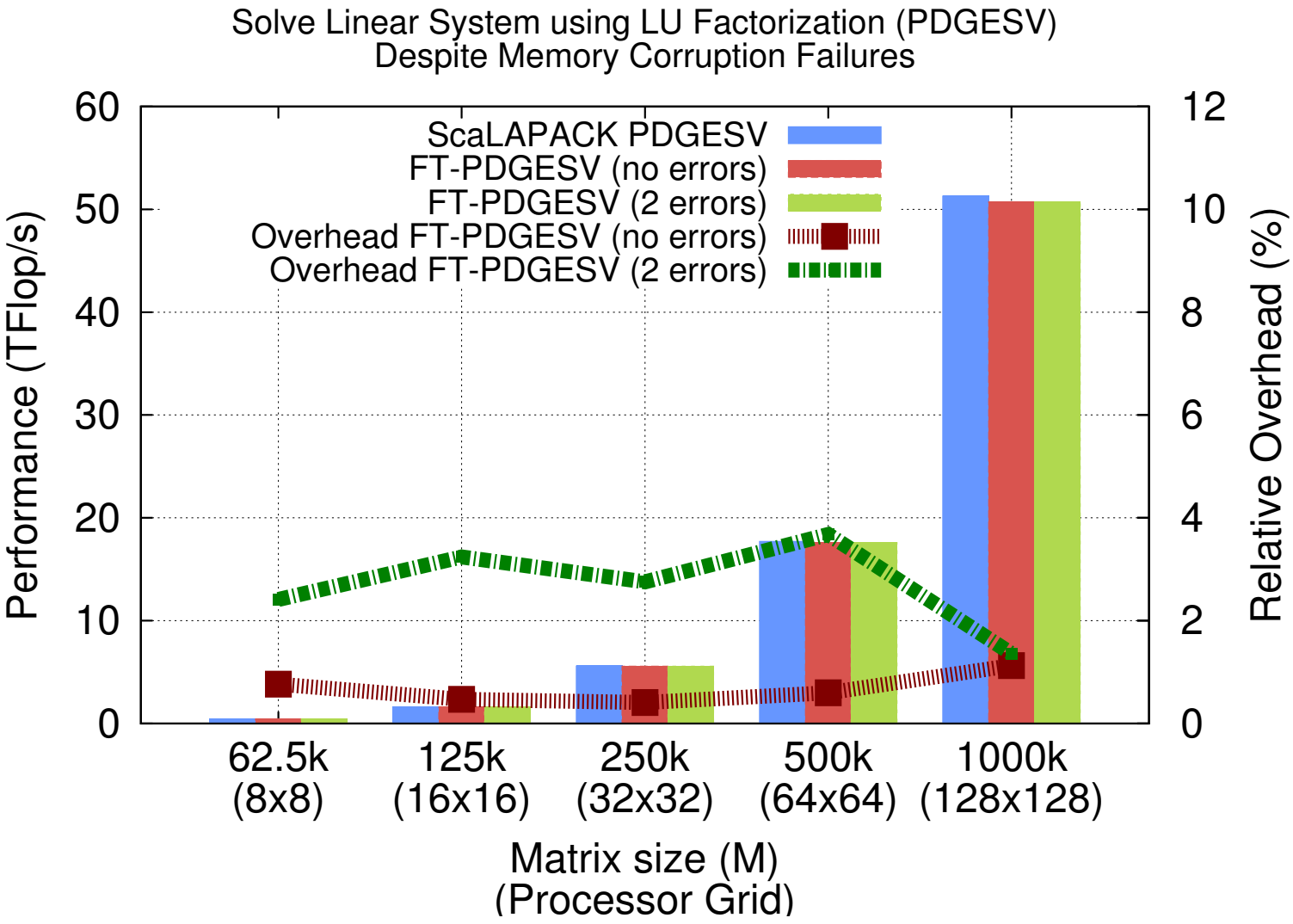
flops for the checksum creation

Performance



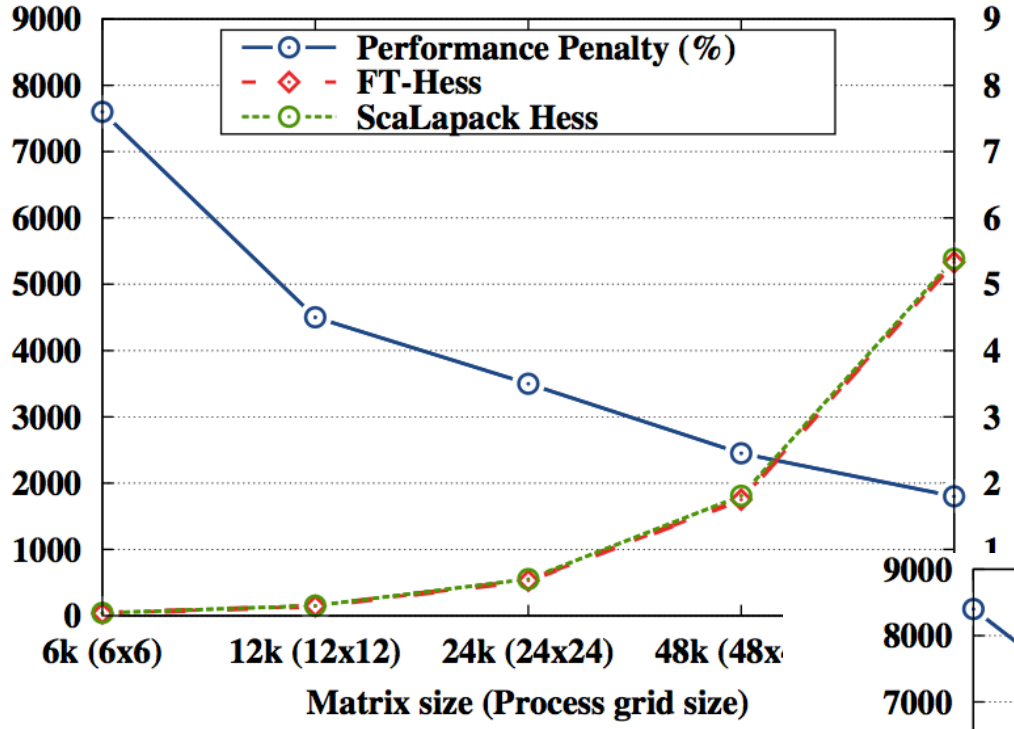
Kraken - Weak Scaling

Performance



Kraken - Weak Scaling

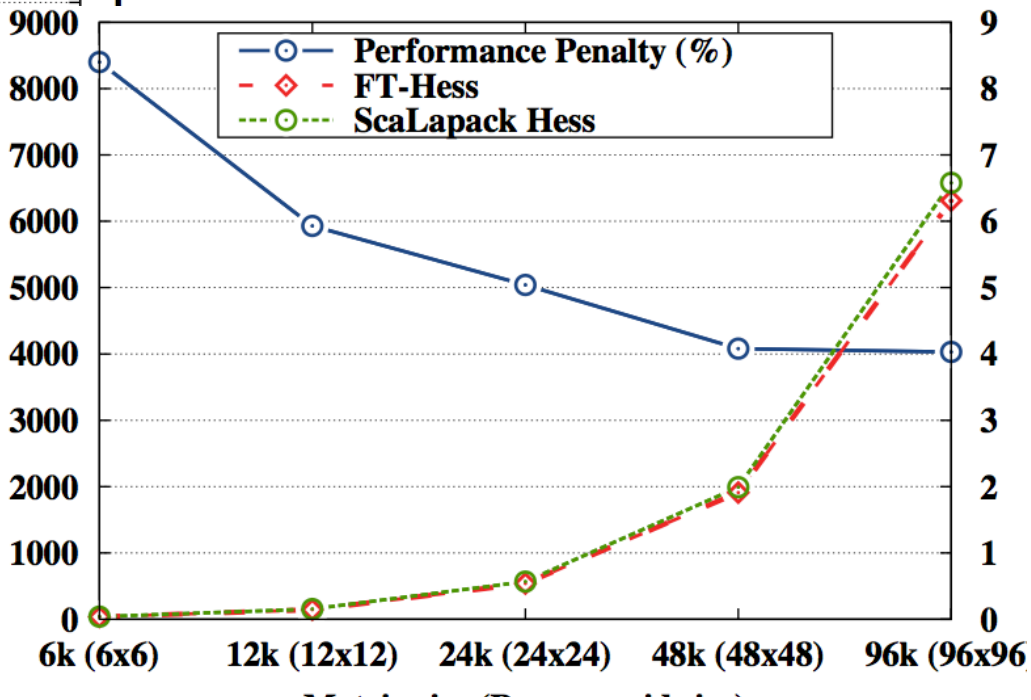
Experiment: Hessenberg Reduction



- ORNL Titan: Cray XK7
- AMD Opteron 6274
- 299,008 core
- Cray Gemini 3D Torus

No errors

1 error



Hessenberg Reduction: Recovery Error

Cores	Process grid	Fault-Tolerant	Classic (ScaLAPACK)
36	6 x 6	5.20×10^{-3}	5.01×10^{-3}
144	12 x 12	3.09×10^{-3}	2.34×10^{-3}
576	24 x 24	2.16×10^{-3}	1.17×10^{-3}
2304	48 x 48	1.36×10^{-3}	6.35×10^{-4}
9216	96 x 96	1.03×10^{-3}	3.37×10^{-4}

ULFM & FT-LA

- User Level Failure Mitigation,
  - An extension to the MPI Standard, currently under discussion by the MPI Forum, and a fault tolerant MPI implementation that let the application space be in control of failure recovery.
  - Implemented in Open-MPI and MPITCH
- FT-LA supports the following operations (in all sdcz precisions):
  - QR factorization (protection against fail-stop failures, failure of at most 1 proc at a time, both Q and R factors protected)
  - LU factorization (protection against fail-stop failures, failure of at most 1 proc at a time, both L and U factors protected)
  - See: http://icl.cs.utk.edu/ft-la/software/index.html

Collaborators / Software / Support

- **PLASMA**
  http://icl.cs.utk.edu/plasma/
- **MAGMA**
  http://icl.cs.utk.edu/magma/
- **Quark (RT for Shared Memory)**
  http://icl.cs.utk.edu/quark/
- **PaRSEC(**Parallel Runtime Scheduling and Execution Control**)**
  http://icl.cs.utk.edu/parsec/

- Collaborating partners
  University of Tennessee, Knoxville
  University of California, Berkeley
  University of Colorado, Denver

  INRIA, France
  KAUST, Saudi Arabia

These tools are being applied to a range of applications beyond dense LA:
Sparse direct, Sparse iterations methods and Fast Multipole Methods