LACSS 2009: Workshop on HPC Resiliency
Santa Fe, New Mexico, USA
Oct 14th, 2009

Highly Scalable Fault Tolerance for Exascale HPC

Zizhong (Jeffrey) Chen
zchen@mines.edu

Colorado School Mines

Outline

- Motivation
- Fault Tolerance in Message Passing Interface (MPI)
- Highly scalable checkpointing for large scale computing
- Algorithm-based checkpoint-free fault tolerance
- Optimal floating-point error correction codes

Trends in HPC

(TOP500: U. of Tennessee, LBNL, and U. of Mannheim)

Rank	Site	Computer	Country	Cores	Rmax [Tflops]	Rmax/ Rpeak
1	DOE/NNSA/LANL	IBM / Roadrunner - BladeCenter QS22/LS21	USA	129600	1105.0	76%
2	DOE/Oak Ridge National Laboratory	Cray / Jaguar - Cray XT5 QC 2.3 GHz	USA	150152	1059.0	77%
3	NASA/Ames Research Center/NAS	SGI / Pleiades - SGI Altix ICE 8200EX	USA	51200	487.0	80%
4	DOE/NNSA/LLNL	IBM / eServer Blue Gene Solution	USA	212992	478.2	80%
5	DOE/Argonne National Laboratory	IBM / Blue Gene/P Solution	USA	163840	450.3	81%
6	NSF/Texas Advanced Computing Center/Univ. of Texas	Sun / Ranger - SunBlade x6420	USA	62976	433.2	75%
7	DOE/NERSC/LBNL	Cray / Franklin - Cray XT4	USA	38642	266.3	75%
8	DOE/Oak Ridge National Laboratory	Cray / Jaguar - Cray XT4	USA	30976	205.0	79%
9	DOE/NNSA/Sandia National Laboratories	Cray / Red Storm - XT3/4	USA	38208	204.2	72%
10	Shanghai Supercomputer Center	Dawning 5000A, Windows HPC 2008	China	30720	180.6	77%

- Parallel Systems with thousands of processors/cores are very common today
  - Largest (Blue Gene from IBM): more than 200K cores
  - Smallest (Dawning 5000A): more than 30K cores
  - Average (of top 10): more than 80K cores per system

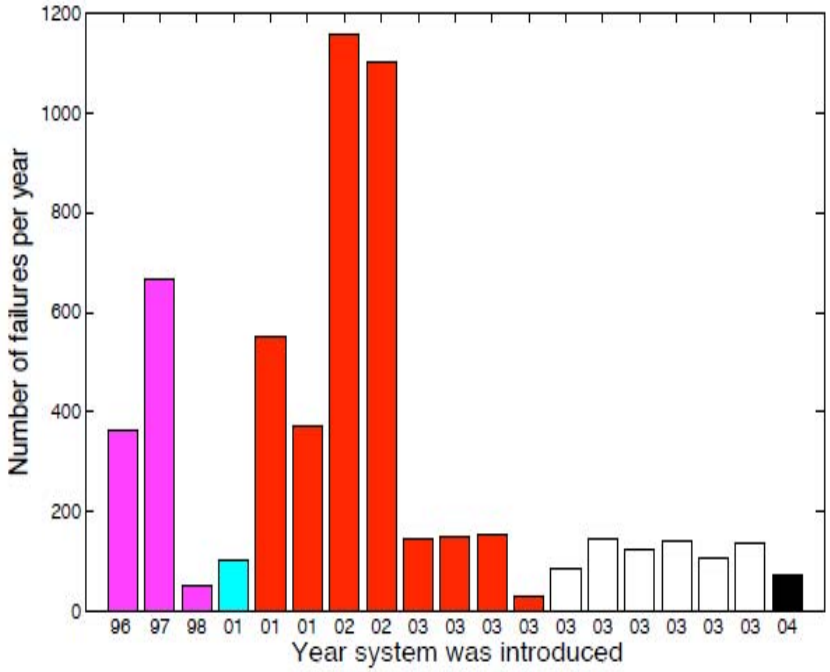
Reliability of Large Systems

- As HPC systems become larger and larger
  - Failure of some links or cores is becoming more and more frequent

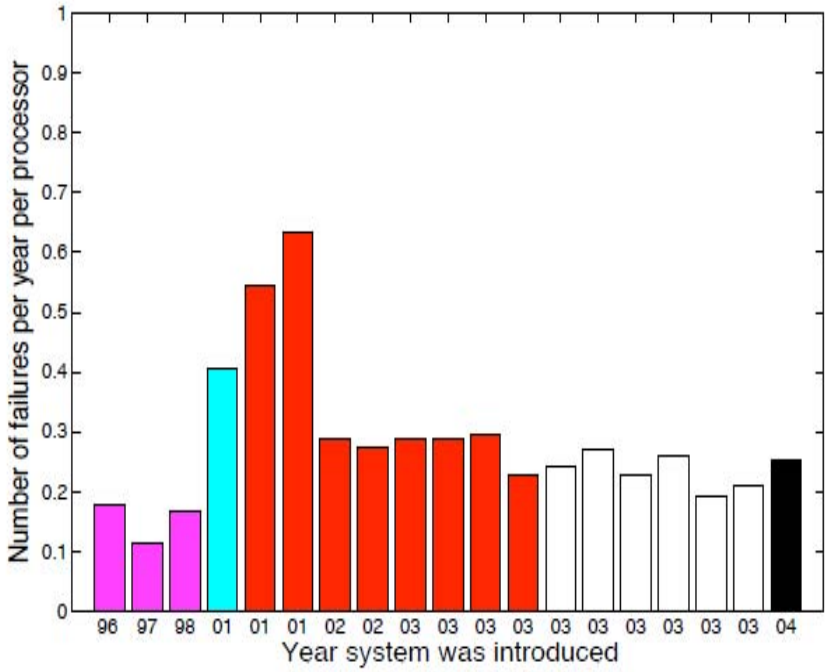
What Is the Failure Rate in HPC Practice?

Gibson (CMU): Fault Tolerance in Petascale Computers, CTWatchQuarterly, Nov, 2007

19 HPCs at LANL: The MTTF varies from about half a month to less than half a day



(a)



(b)

Figure 2. *(a) Average number of failures for each LANL system per year. (b) Average number of failures for each system per year normalized by number of processors in the system. Systems with the same hardware type have the same color.*

Expected Program Execution Time

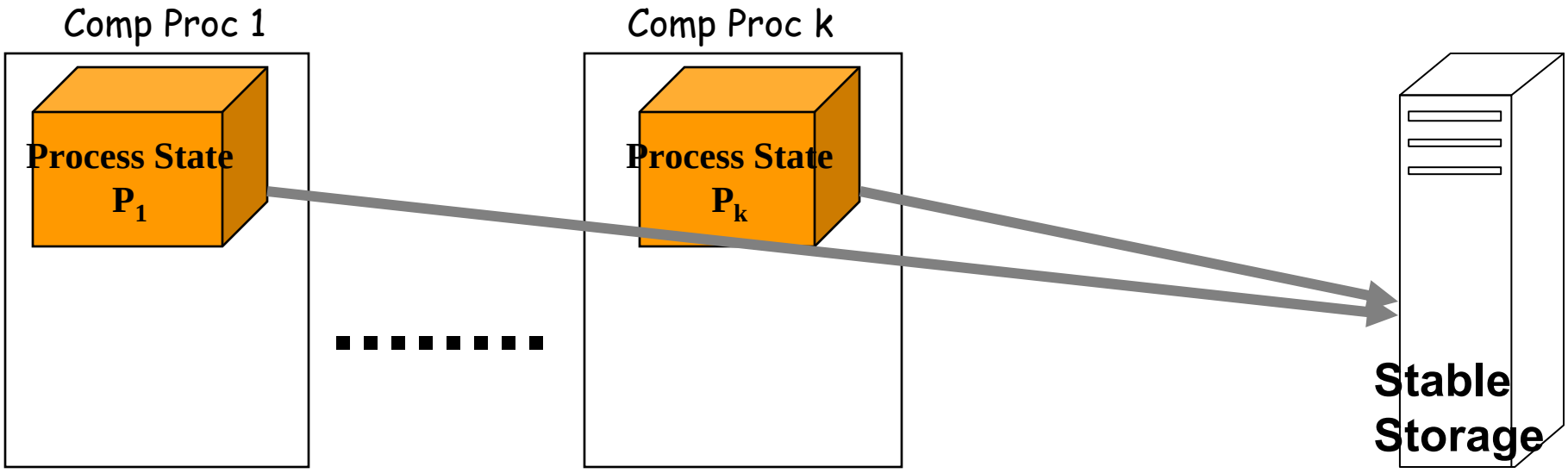
- Increase of the failure rate
  - Suppose the failure rate of a single node is λ
  - What is the failure rate of **n** nodes?
  - $n\lambda$

- Expected program execution time
  - T: program execution time in a failure free environment
  - E: expected program execution time in an unstable environment

$$E = T e^{-n\lambda T} + \int_0^T (\tau + E)\, n\lambda e^{-n\lambda\tau}\, d\tau$$

$$E = (e^{n\lambda T} - 1) / n\lambda$$

Checkpoint/Restart



- Checkpoint/restart is today's typical fault tolerance approach in HPC
  - Periodically write all process states into *stable-storage*
  - If one process fails, *abort all processes*
  - Good for tolerating the failure of the whole system
  - But the overhead is high : **T = # of procs * size_ckpt / io-bandwidth**

- Today's architectures are usually robust enough to survive partial failures without suffering the failure of the whole system
  - Can we tolerate partial failures with less overhead (and better scalability) than checkpoint/restart ?

Scalable Techniques for Fault Tolerant High Performance Computing

- Design scalable fault tolerance techniques for HPC to recover from partial process failures
  - Tolerate a small number of process failures
  - Highly scalable
- Assume a failure-stop model
  - Failed processes stop working
  - All data on failed processes are lost

Outline

- Motivation
- Fault Tolerance in Message Passing Interface (MPI)
- Scalable checkpointing for large scale computing
- Fault tolerant linear algebra algorithms and software
- Numerically stable floating-point numbers error correction codes

FT-MPI http://icl.cs.utk.edu/ft-mpi/

- MPI: A Message Passing Interface Standard for HPC
  - has been widely accepted in today's high performance computing
  - However, MPI standard does not address the fault tolerance issue
  - If one MPI process fails, most MPI implementations abort all

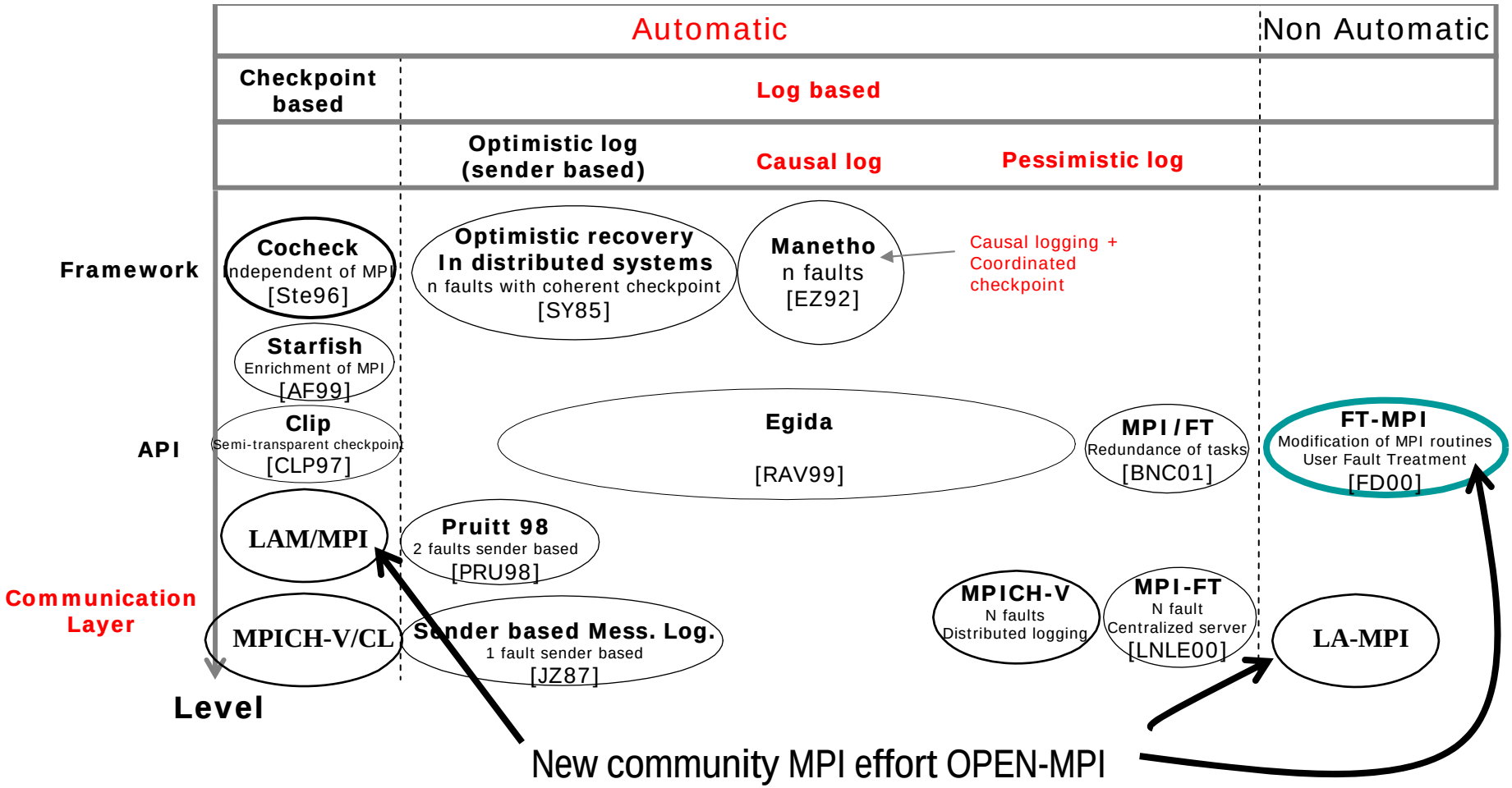
- FT-MPI is a fault tolerant MPI implementation developed at U of Tenn.
  - It detects failures and recovers the MPI runtime environment
  - Gives applications the possibility to survive partial process failures

- FT-MPI does NOT:
  - Automatically recover the application when a process failure occurs

- A fault tolerant application developer has to:
  - Design his own recovery schemes to recover his application

Open MPI http://icl.cs.utk.edu/open-mpi/

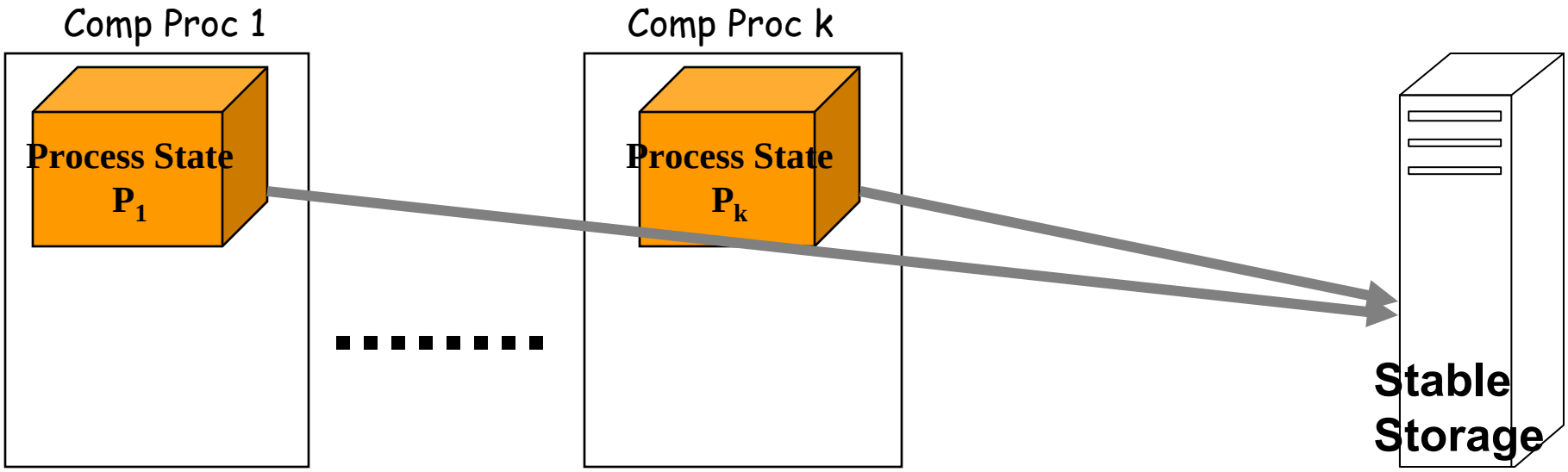
- **A classification of fault tolerant message passing environments considering**
  - A) level in the software stack where fault tolerance is managed and
  - B) fault tolerance techniques.



Outline

- Motivation
- Fault Tolerance in Message Passing Interface (MPI)
- Scalable checkpointing for large scale computing
- Fault tolerant linear algebra algorithms and software
- Numerically stable floating-point numbers error correction codes

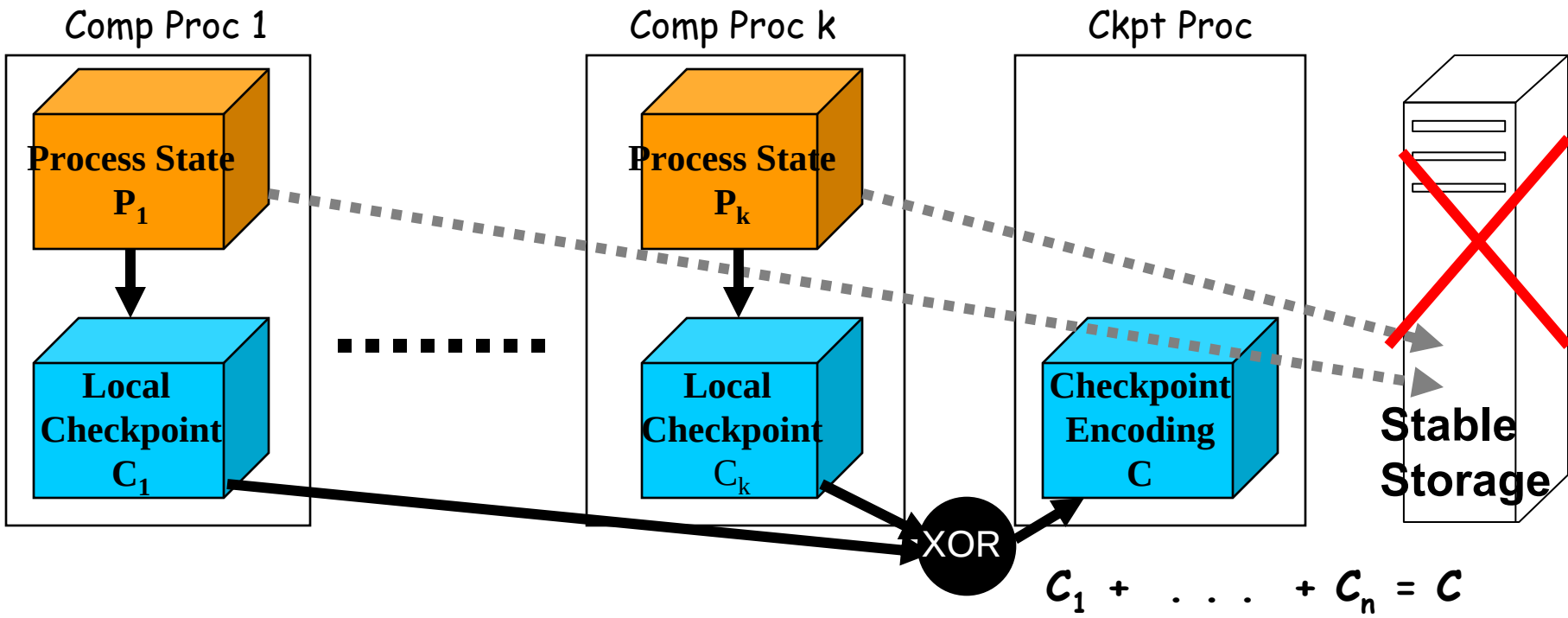
Checkpoint/Restart



- Checkpoint/restart is today's typical fault tolerance approach in HPC
  - Periodically write all process states into *stable-storage*
  - **T = # of procs * size_ckpt_per_procs / io-bandwidth**

Diskless Checkpointing

(J. S. Plank, et. el.)

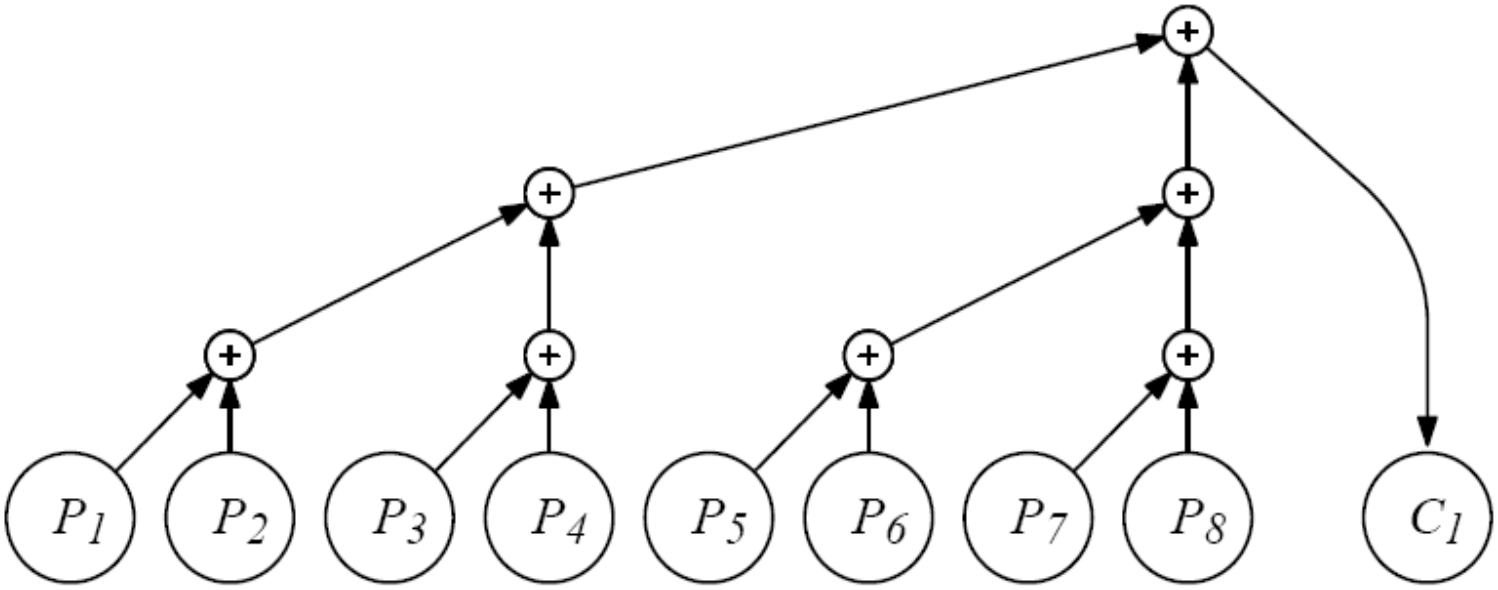


- Each computational processor saves a copy of its state locally in memory
- Dedicate an additional processor to save the XOR's of these states

- When a failure occurs
  - Surviving processor roll back to its last state. The failed processor's state can be calculated from local checkpoints and the encoding

Diskless Checkpointing

(J. S. Plank, et. el.)

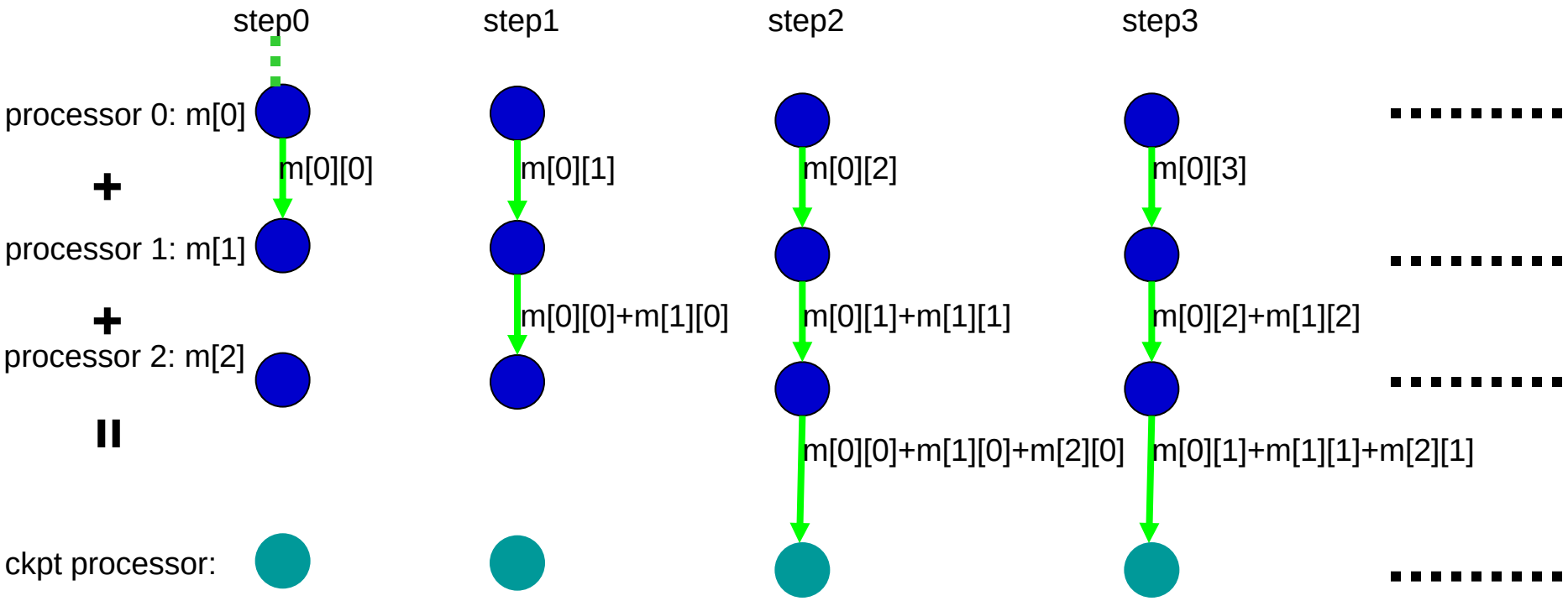


- Total number of steps: **log (# of procs)**
- The time to perform one checkpoint:

  T = log (# of procs) * (size_ckpt / bandwidth + latency)

  ≈ log (# of procs) * size_ckpt / bandwidth

A Scalable Algorithm for Checkpoint Encoding

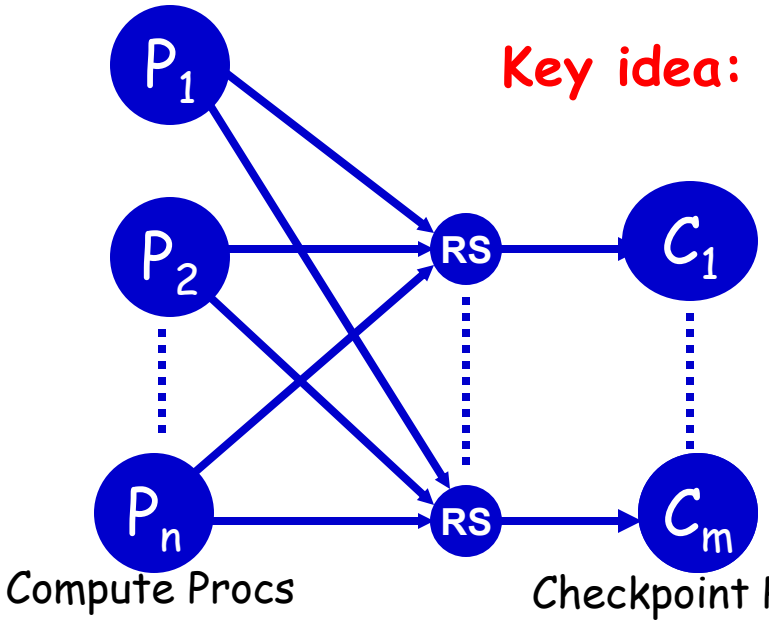


- Optimize *size_seg* to achieve optimal checkpoint performance

  - T = (p + N) * (size_seg / bandwidth + latency)
    = (p + size_ckpt / size_seg) * (size_seg / bandwidth + latency)
    >= size_ckpt / bandwidth * (1 + O (sqrt (latency * p / size_ckpt)))

    ≈ size_ckpt / bandwidth

Survive Multiple Failures by Reed-Solomon Encoding



Key idea: establish *m* equalities by *m* encodings

$$\begin{cases} C_1 = a_{11} * P_1 + \ldots + a_{1n} * P_n \\ \vdots \\ C_m = a_{m1} * P_1 + \ldots + a_{mn} * P_n \end{cases}$$

If there are *k* (<= *m*) processes failed, then the *m* equalities become

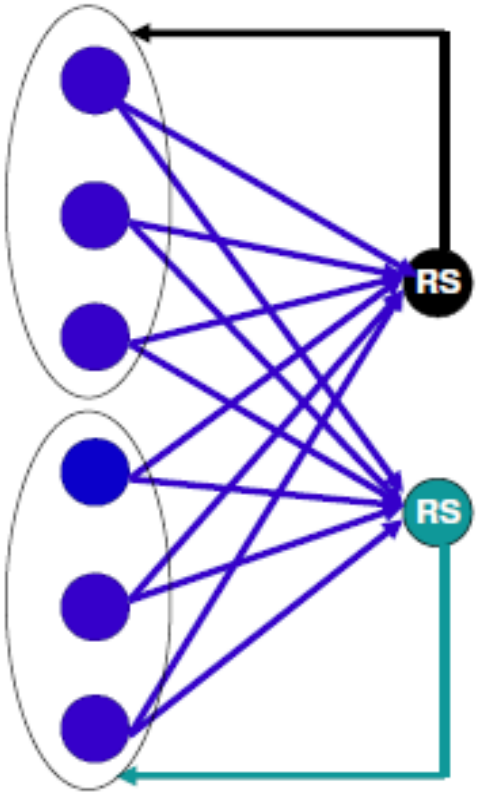
m equations with *k* unknowns

By appropriately choosing *A*, the lost data can be recovered by solving the *m* equations.

The checkpoint overhead (assuming pipelined encoding):

$T \approx m * size\_ckpt / bandwidth$

Reed-Solomon Encoding: Checkpoint into Neighbor Processes



In order to tolerate *m* failures, divide all processes into subgroup of size

$$m\,(\,m+1\,)$$

Within each subgroup, save each of the *m* encodings into (*m+1*) different processes.

The checkpoint overhead (assuming pipelined encoding):

$$T \approx (m+1) * size\_ckpt / bandwidth$$

How to Choose the Encoding Matrix *A* ?

$$\begin{cases} C_1 = a_{11} * P_1 + \cdots + a_{1j} * P_j + \cdots + a_{1(j+m)} * P_{j+m} + \cdots + a_{1n} * P_n \\ \vdots \\ C_m = a_{m1} * P_1 + \cdots + a_{mj} * P_j + \cdots + a_{m(j+m)} * P_{j+m} + \cdots + a_{mn} * P_n \end{cases}$$

- In order to be able to recover from any k (k <= m) failures, the checkpoint encoding matrix A has to satisfy

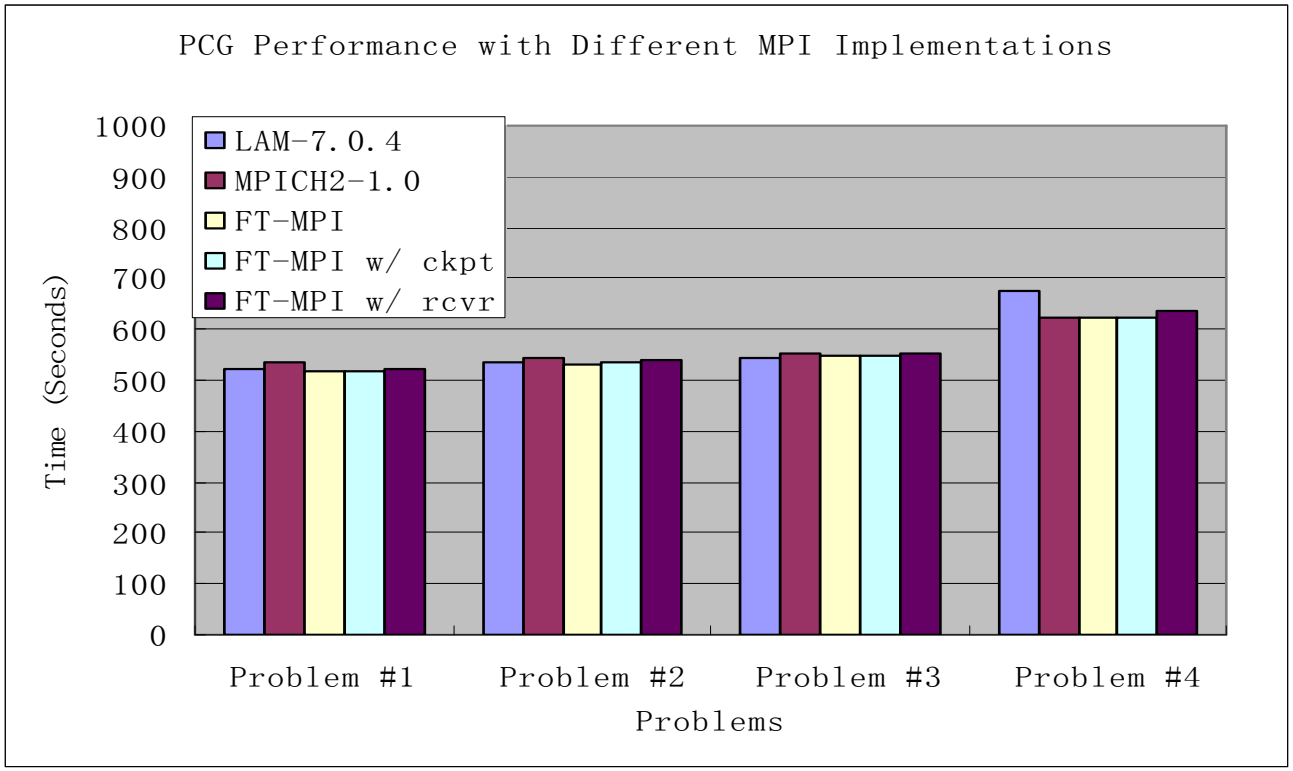
  Any square sub-matrix of *A* is non-singular

- How to find such an A ?
  - Cauchy matrix:

$$A = \begin{pmatrix} \frac{1}{x_1 + y_1} & \cdots & \frac{1}{x_1 + y_n} \\ \vdots & \cdots & \vdots \\ \frac{1}{x_m + y_1} & \cdots & \frac{1}{x_m + y_n} \end{pmatrix}$$

- **How to perform the multiplication of two process states ?**
  - **Process states have to be treated as bit-streams**
  - **Galois field arithmetic has to be used in the computation**

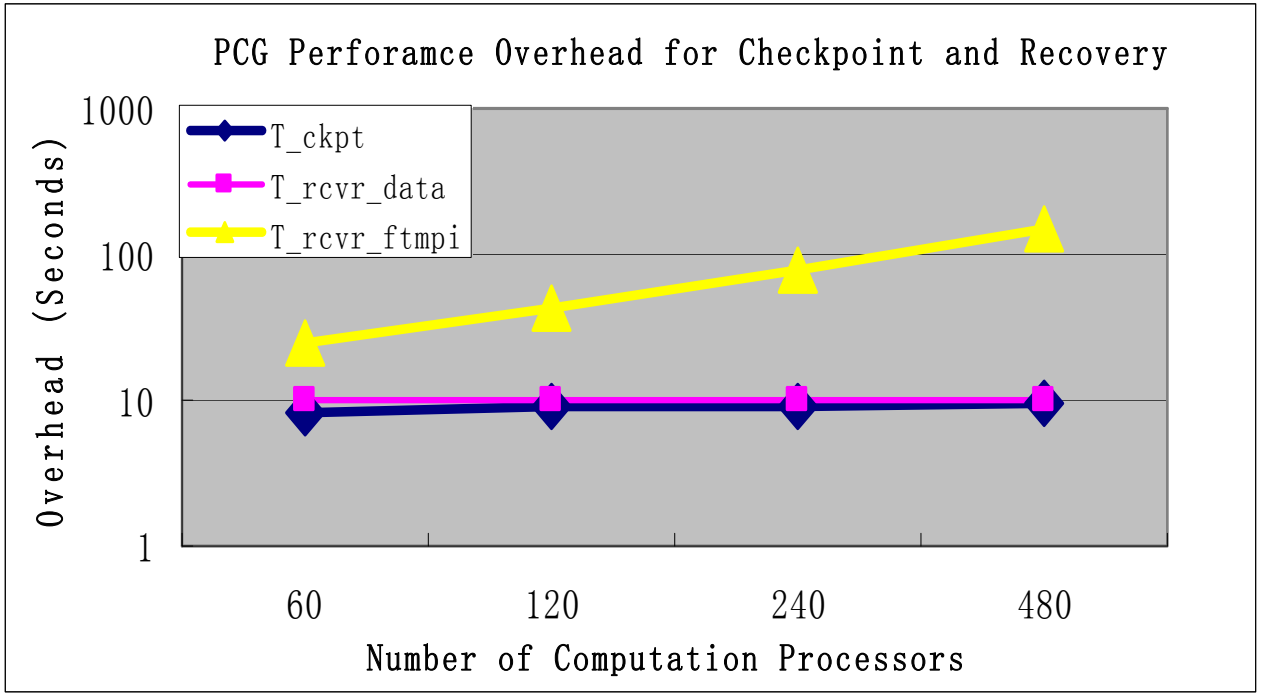
PCG: Performance with Different MPI Implementations



Platform: 64 dual-processor 2.4 GHz AMD Opteron nodes with Gigabit interconnect.
Size of checkpoint per processes: ~ 0.25 MBytes

T for 20000 iterations	LAM-7.0.4	MPICH2-1.0	FT-MPI	FT-MPI-1.2 ckpt/2000 iterations (≈ 1 minute)	FT-MPI-1.2 exit 1 proc @10000 iterations
15 comp. procs	522.5	536.3	517.8	518.9	521.7
30 comp. procs	532.9	542.9	532.2	533.3	537.5
60 comp. procs	545.5	553.0	546.5	547.8	554.2
120 comp. procs	674.3	624.4	622.9	624.4	637.1

PCG: the Overhead for Checkpoint and Recovery



Run PCG for 5000 iterations and take checkpoint every 1000 iterations (~5 minutes).
Simulate a failure of one node by exiting 4 processes at the 3000-th iteration.
Size of checkpoint per processes: ~ 25 Mbytes.

Time (Seconds)	T_tot	T_ckpt	T_rcvr_data	T_rcvr_ftmpi
60 procs	1441.7	8.0	9.8	24.8
120 procs	1490.5	9.2	9.9	42.1
240 procs	1557.5	9.2	10.0	77.2
480 procs	1697.0	9.7	10.1	146.1

Outline

- Motivation
- Fault Tolerance in Message Passing Interface (MPI)
- Scalable checkpointing for large scale computing
- Fault tolerant linear algebra algorithms and software
- Numerically stable floating-point numbers error correction codes

Checkpoint-Free Fault Tolerance

- Assume
  - we are running a parallel program where $P_i(t)$ denotes the data on the i^{th} processor at time t
  - $P_1(t)+ P_2(t)+ \ldots + P_n(t) = P_{n+1}(t)$ for any t
- If the first processor fails, how can we recover the lost data $P_1(t)$?
  - Answer: $P_1(t) = P_{n+1}(t) - P_2(t) - \ldots - P_n(t)$
- **Question 1: does this kind of special relationship exist in non-trivial applications ?**
  - YES: In many iterative methods, it does exist !
- **Question 2: If such a relationship does not exist, is it possible for us create this kind of special relationship on purpose ?**
  - Sometimes YES: For some programs doing dense linear algebra computations, such a relationship can be created by performing computations with encoded data

Checkpoint-Free Fault Tolerance for Iterative Methods
(SIAM: SISC 2007)

- Assume
  - we are solving $A * x = b$ by iterative methods
  - $r = b - A * x$ is maintained in memory

- In a parallel environment, $r = b - A * x$ can be rewrite as

$$\begin{cases} a_{11} * x_1 + \cdots + a_{1n} * x_n = b_1 - r_1 \\ \vdots \\ a_{n1} * x_1 + \cdots + a_{nn} * x_{1n} = b_1 - r_n \end{cases}$$

- If the i^{th} processor fails, then both $\mathbf{r}_i$ and $\mathbf{x}_i$ become unknown

  - $\mathbf{x}_i$ can be recovered accurately unless $\mathbf{a}_{ki}$ is singular for **ALL** $k \neq i$

  - If only $\mathbf{a}_{ii}$ is non-singular, set $\mathbf{r}_i = 0$

Checkpoint-Free Fault Tolerance for Dense Linear Algebra

The *column checksum* matrix A^c of the matrix A is defined by

$$A^c = \begin{pmatrix} a_{0\,0} & \cdots & a_{0\,n-1} \\ \vdots & \cdots & \vdots \\ a_{m-1\,0} & \cdots & a_{m-1\,n-1} \\ \sum_{i=0}^{m-1} a_{i\,0} & \cdots & \sum_{i=1}^{m-1} a_{i\,n-1} \end{pmatrix};$$

The *row checksum* matrix A^r of the matrix A is defined by

$$A^r = \begin{pmatrix} a_{0\,0} & \cdots & a_{0\,n-1} & \sum_{j=0}^{n-1} a_{0\,j} \\ \vdots & \cdots & \vdots & \vdots \\ a_{m-1\,0} & \cdots & a_{m-1\,n-1} & \sum_{j=0}^{n-1} a_{m-1\,j} \end{pmatrix};$$

The *full checksum* matrix A^f of the matrix A is defined by

$$A^f = \begin{pmatrix} a_{0\,0} & \cdots & a_{1n} & \sum_{j=0}^{n-1} a_{0\,j} \\ \vdots & \cdots & \vdots & \vdots \\ a_{m-1\,0} & \cdots & a_{m-1\,n-1} & \sum_{j=0}^{n-1} a_{m-1\,j} \\ \sum_{i=0}^{m-1} a_{i\,0} & \cdots & \sum_{i=0}^{m-1} a_{i\,n-1} & \sum_{j=0}^{n-1} \sum_{i=0}^{m-1} a_{i\,j} \end{pmatrix}.$$

Algorithm-based Fault Tolerance
Abraham (UTexas), 1984:

If $A * B = C$
Then $A^c * B^r = C^f$

Checksum is maintained in final computation results

Is the Checksum Maintained During Computation?

```
/* Calculate A^c * B^r by cannon's algorithm. */
initialize C = (c_ij) = 0;
for i = 0 to n - 1
    left-circular-shift row i of A^c by i
        so that a_{i,j} is overwritten by a_{i, (j+i) mod n};
end
for i = 0 to n - 1
    up-circular-shift column i of B^r by i
        so that b_{i,j} is overwritten by b_{(i+j) mod n, j};
end
for s = 0 to n - 1
    every processor (i, j) performs c_ij = c_ij + a_ij * b_ij
        locally in parallel;
    left-circular-shift each row of A^c by 1;
    up-circular-shift each column of B^r by 1;
    /* Here is the end of the s^th step. */
end
```

```
/* Calculate A^c * B^r by fox's algorithm. */
initialize C = (c_ij) = 0;
for s = 0 to n - 1
    for i = 0 to n - 1 in parallel
        processor (i, (i + s) mod n) broadcast local
        t = a_{i, (i+s) mod n} to other processors in row i;
    for i, j = 0 to n - 1 in parallel
        every processor (i, j)
        performs c_ij = c_ij + t * b_ij locally;
    up-circular-shift each column of B^r by 1;
    /* Here is the end of the s^th step. */
end
```

NO

```
/* Calculate A_r * B_c by outer product algorithm.*/
initialize C = (c_ij) = 0;
for s = 0 to n - 1
    row broadcast the s^th row of A_c;
    column broadcast the s^th column of B_r;
    every processor (i,j) performs c_ij = c_ij + a_is * b_sj
        locally in parallel;
    /* Here is the end of the s^th step. */
end
```

YES

- **NOT** maintained
  - Cannon's algorithm
  - Fox's algorithm
- Maintained
  - Outer product version algorithm
- **Outer product version is usually used in today's HPC practice**

Checkpoint-Free Fault Tolerance for ScaLAPACK
(IEEE: TPDS to appear)

- Define the *row distributed checksum* matrix of M as

$$M^{r} = \begin{pmatrix} M_{11} & \cdots & M_{1q} \\ \vdots & \cdots & \vdots \\ M_{p1} & \cdots & M_{pq} \\ \sum_{i=1}^{p} M_{i1} & \cdots & \sum_{i=1}^{p} M_{iq} \end{pmatrix}$$

- Define the *column distributed checksum* matrix of M as

$$M^{c} = \begin{pmatrix} M_{11} & \cdots & M_{1q} & \sum_{j=1}^{q} M_{1j} \\ \vdots & \cdots & \vdots & \vdots \\ M_{p1} & \cdots & M_{pq} & \sum_{j=1}^{q} M_{pj} \end{pmatrix}$$

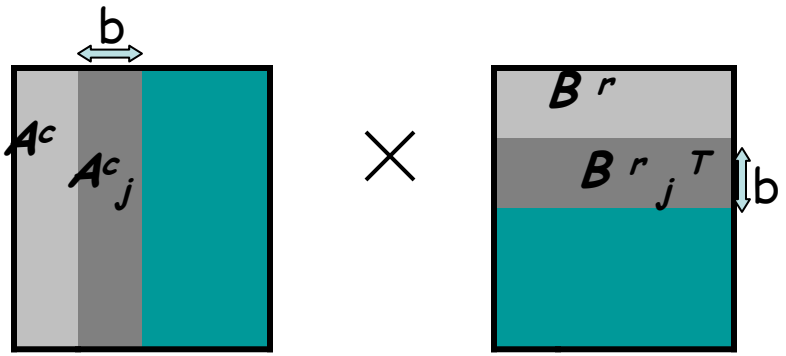
- Define the *full distributed checksum* matrix of M as

$$M^{f} = \begin{pmatrix} M_{11} & \cdots & M_{1q} & \sum_{j=1}^{q} M_{1j} \\ \vdots & \cdots & \vdots & \vdots \\ M_{p1} & \cdots & M_{pq} & \sum_{j=1}^{q} M_{pj} \\ \sum_{i=1}^{p} M_{i1} & \cdots & \sum_{k=1}^{p} M_{iq} & \sum_{i=1}^{p}\sum_{j=1}^{q} M_{ij} \end{pmatrix}$$

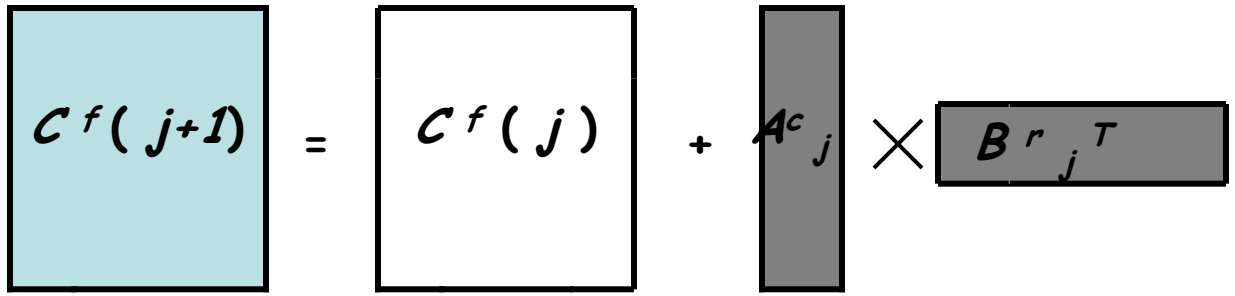
An Example: ScaLAPACK Matrix-Matrix Multiplication

PDGEMM: C = C + A * B

FT-PDGEMM operates on A^r, B^c and C^f



At the j^{th} iteration:



- **Theorem:**

  At the end of each iteration, the checksum relationships in A^r, B^c, and C^f are still maintained

- **Conclusion**
  - **Single failure during computation can be recovered from the checksum**
  - **By using a floating-point version Reed-Solomon code, multiple failures can be tolerated**

Matrix Multiplication: Overhead and Scalability Analysis

- The overhead (%) for constructing Checksum at the beginning

$$R_{total\_encode} = \frac{T_{total\_encode}}{T_{matrix\_mult}} = O(\frac{1}{Pm})$$

- The overhead (%) for computation due to larger size of matices

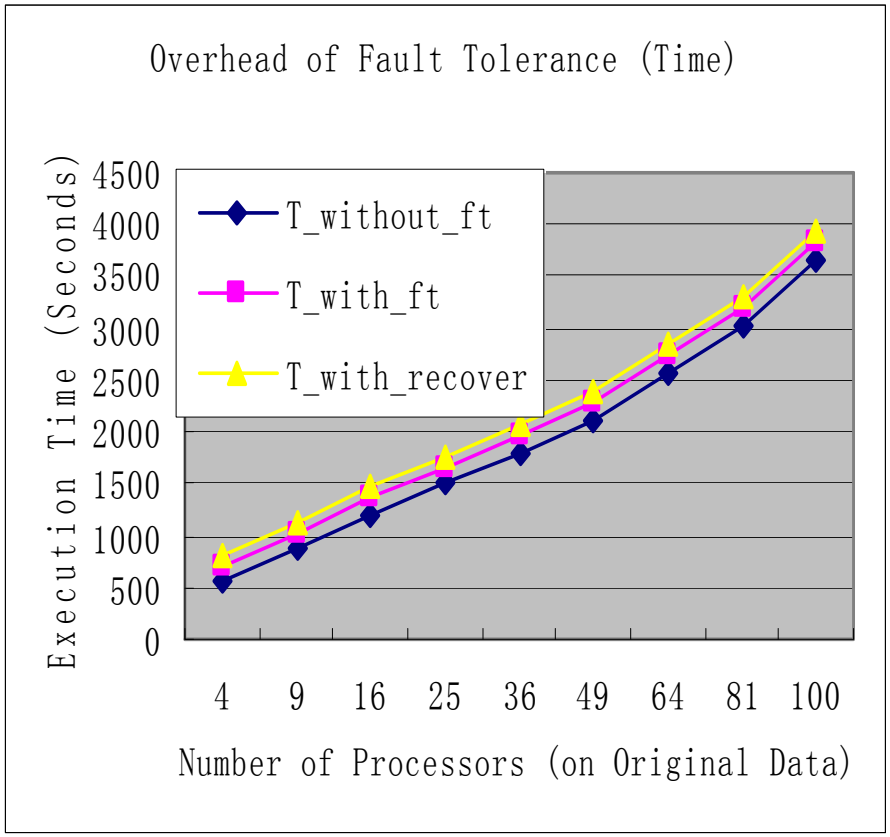
$$R_{overhead\_matrix\_mult} = \frac{T_{overhead\_matrix\_mult}}{T_{matrix\_mult}} = O(\frac{1}{Pm})$$

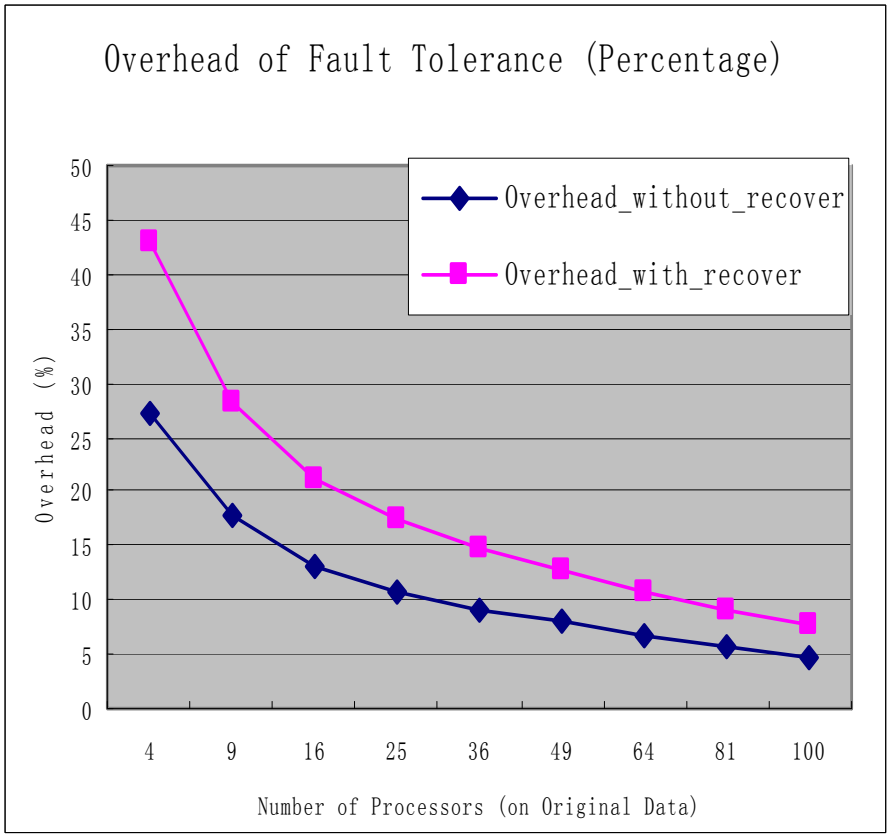
- The overhead (%) for recovery after a failure

$$R_{recover\_data} = \frac{T_{recover\_data}}{T_{matrix\_mult}} = O(\frac{1}{Pm})$$

- **Note that**
  - **O (1 / (p*m)) ⟶ 0, as p, m ⟶ ∞**

PDGEMM: the Overhead for Fault Tolerance

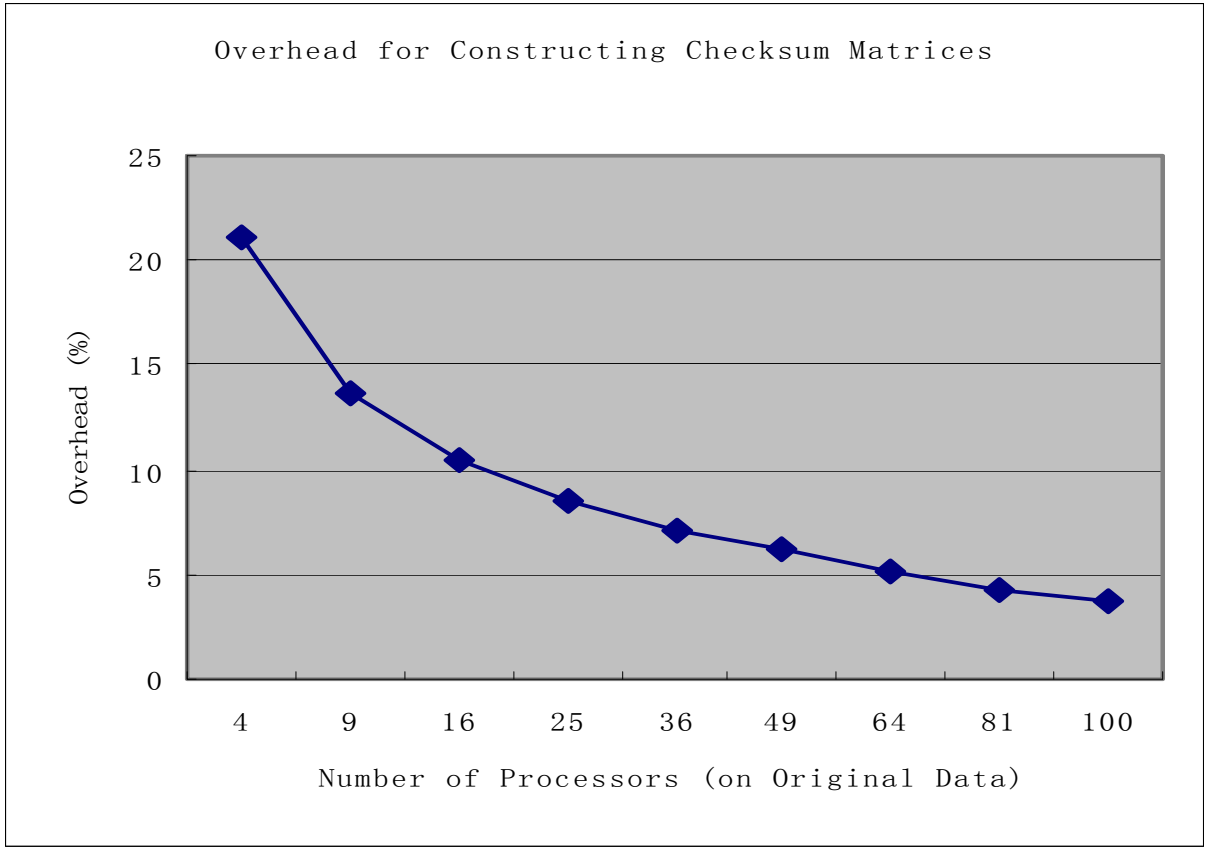




- Remember that the percentage of overhead for fault tolerance is

  - $O(1/(p*n)) \longrightarrow 0$, as $p \longrightarrow \infty$

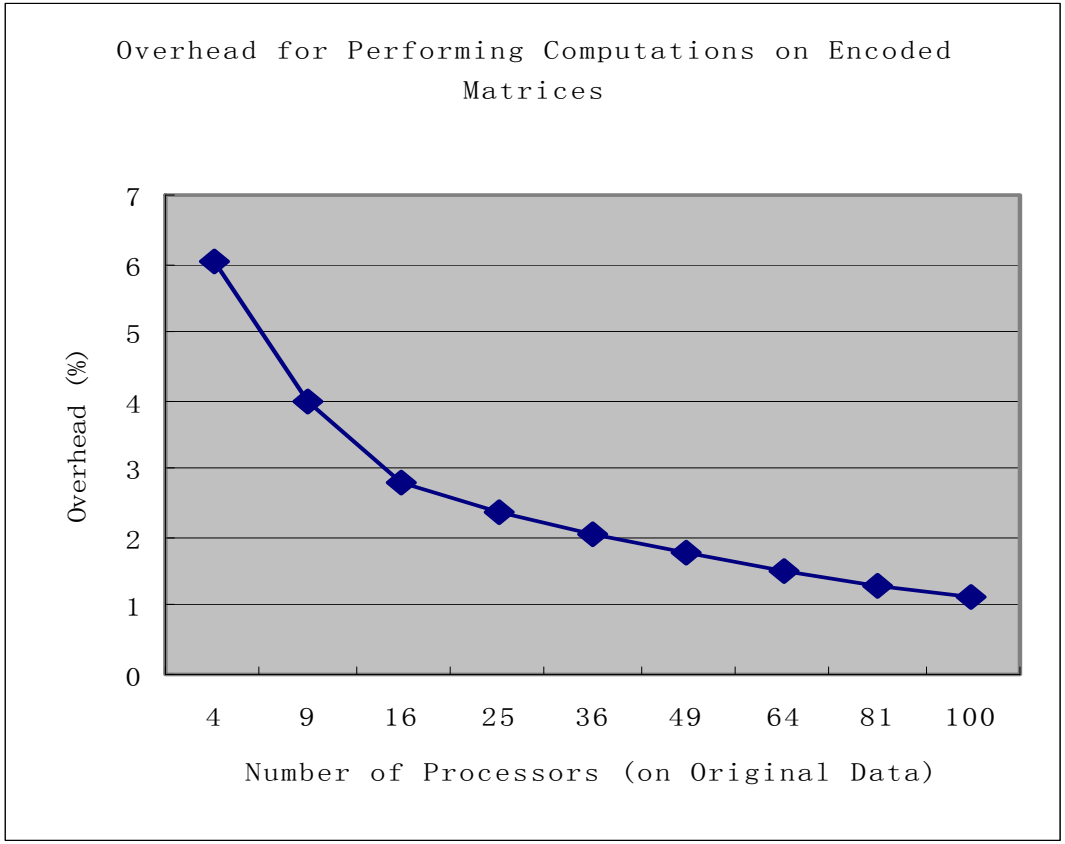
PDGEMM: The Overhead (%) for Calculating Encodings



- Note that the overhead for encoding is

  - $O(1/(p*n)) \longrightarrow 0$, as $p \longrightarrow \infty$

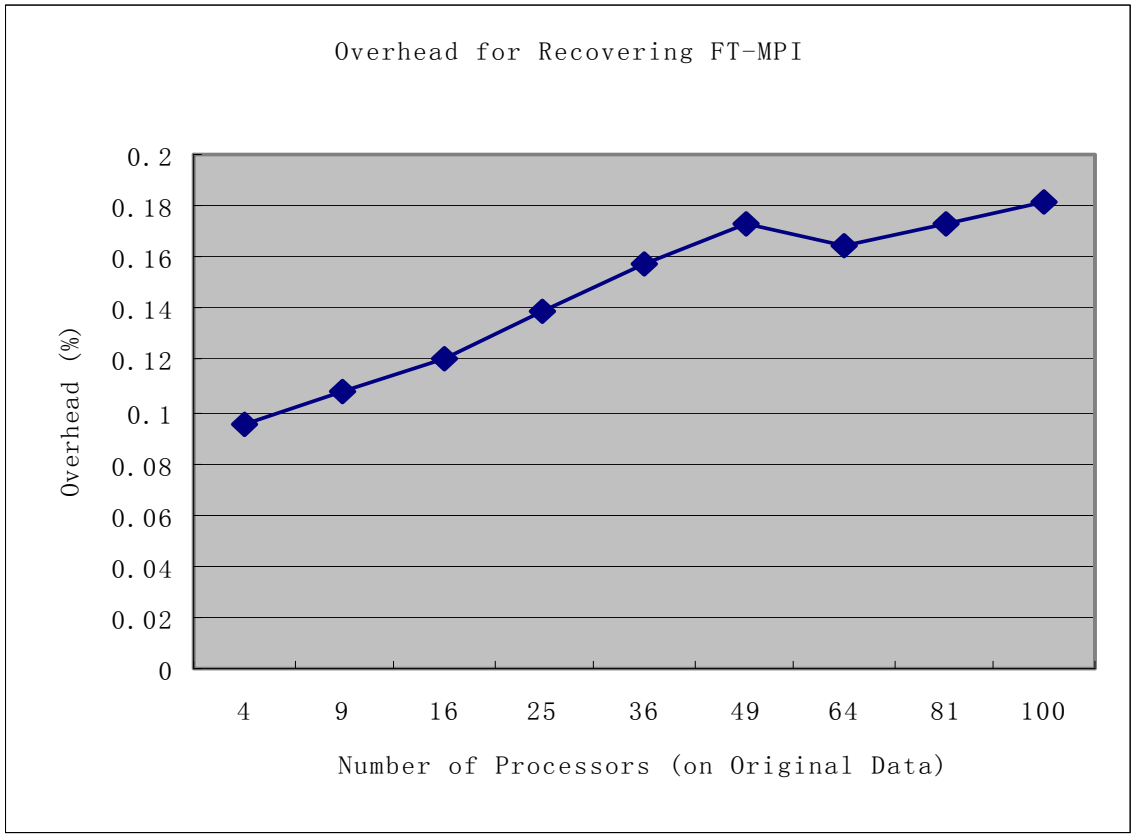
PDGEMM: The Overhead for Performing Computation with Encoded Matrices



- Note that the overhead for performing computation with encoded matrices is

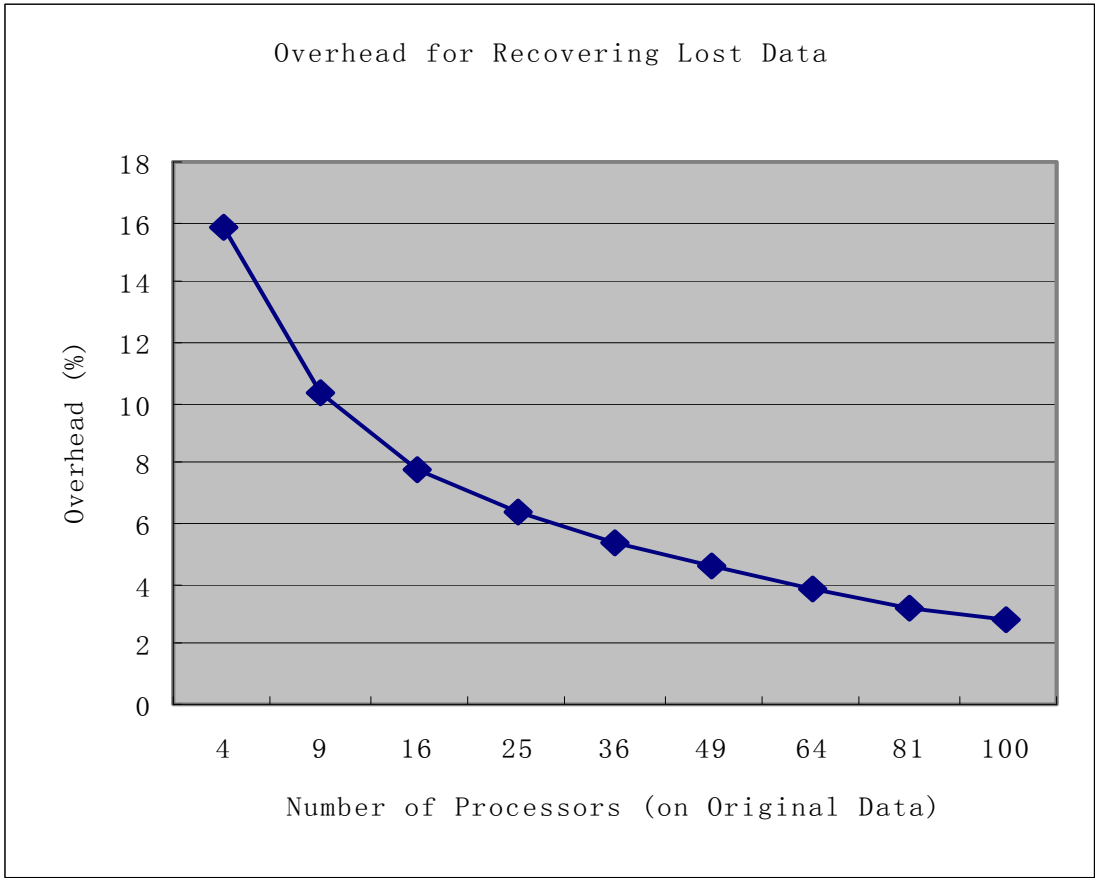
  - $O(1/(p*n)) \longrightarrow 0$, as $p \longrightarrow \infty$

PDGEMM: The Overhead for Recovering FT-MPI Environment



- Note that the time to recover FT-MPI
  - is currently O (p^2), but will be improved to O(log p) soon
  - is negligible (< 0.2%) compared with the time to recover application data

PDGEMM: The Overhead for Recovering Application Data



- Note that the overhead for recovering the application data is
  - $O(1/(p*n)) \longrightarrow 0$, as $p \longrightarrow \infty$

Outline

- Motivation
- Fault Tolerance in Message Passing Interface (MPI)
- Scalable checkpointing for large scale computing
- Fault tolerant linear algebra algorithms and software
- Numerically stable floating-point numbers error correction codes

Floating-Point Codes to Tolerate Multiple Failures

$$\begin{cases} C_1 = a_{11} * P_1 + \cdots + a_{1j} * P_j + \cdots + a_{1(j+m)} * P_{j+m} + \cdots + a_{1n} * P_n \\ \qquad\qquad\qquad\qquad \vdots \\ C_m = a_{m1} * P_1 + \cdots + a_{mj} * P_j + \cdots + a_{m(j+m)} * P_{j+m} + \cdots + a_{mn} * P_n \end{cases}$$

- In order to be able to recover from any k (k <= m) failures, the weight matrix A has to satisfy

  Any square sub-matrix of A is non-singular

- **To maintain the checksum relationship**
  - **Floating-point arithmetic** has to be used when calculating encodings
- **Due to round-off errors in floating-point computations**

  Require any square sub-matrix of A is well-conditioned

- **Can we use Cauchy, Vantermonde, DFT matrices ?** **No!**

Floating-Point Codes Based on Random Matrices

- It is well-known Gaussian random matrices are well-conditioned with high probability.
- Any sub-matrix of a Gaussian random matrix *G* is still a Gaussian random matrix
  - Therefore, *any square sub-matrix of G is well conditioned with high probability*
- How well-conditioned a Gaussian random matrix is?

$$\frac{1}{\sqrt{2\pi}}\left(\frac{0.245\,\frac{n}{|n-m|+1}}{x}\right)^{|n-m|+1} < \Pr\left(\kappa_2(G_{m\times n}) > x\right) < \frac{1}{\sqrt{2\pi}}\left(\frac{6.414\,\frac{n}{|n-m|+1}}{x}\right)^{|n-m|+1}$$

$$E\left(\log_{10}\kappa_2(G_{m\times n})\right) < \log_{10}\frac{n}{|n-m|+1} + 0.981.$$

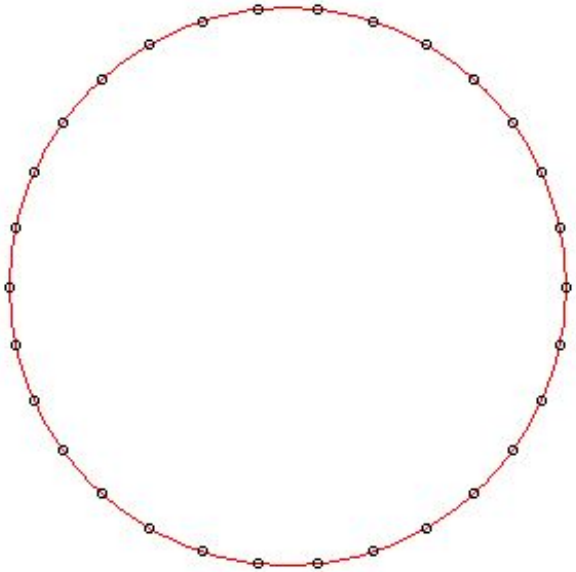
(SIAM: SIMAX 2005)

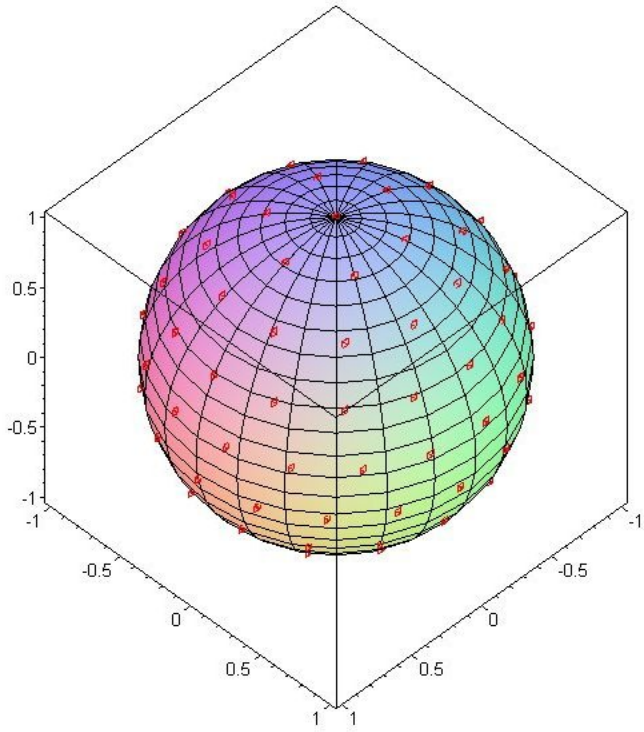
- In our fault tolerant applications:

  Assume we are running an application on a 100K-processor system, and tolerating 20 concurrent failures. If there are 10 concurrent failures actually occur, then

  (1). On average, we will loss about 1 decimal digit in recovery
  (2). The probability to lose 2 decimal digits is less than 10^{-10}

Why Gaussian Random Matrices are so Good ?





- The condition number of a matrix can be treated as a measure of the degree of linear independence of the columns of the matrix
  - Dependent: $k = \infty$; Orthogonal: $k = 1$;
- Uniformly-distributed points on spheres produce good degree of linear independence
- Gaussian distribution in R^n is equivalent to Uniform distribution in S^{n-1} (Knuth 69)

Real Number Codes from Grassmannian Frames

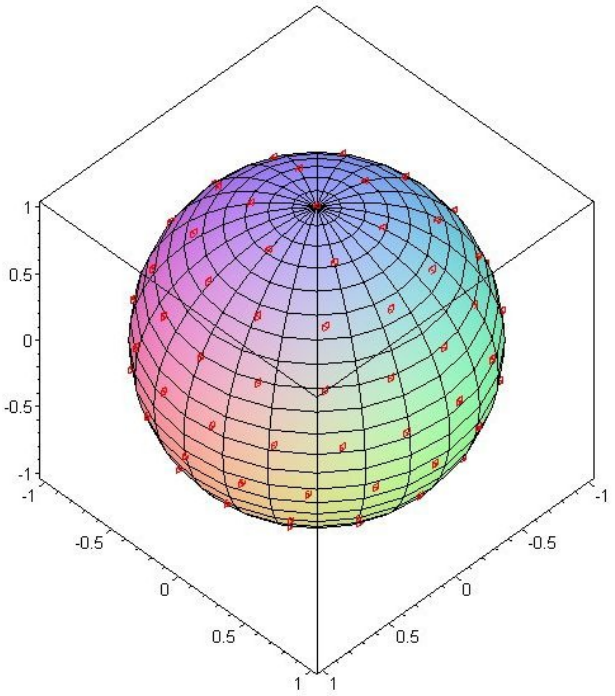
Maximize the minimum angle
between columns of the matrix

Distributing points on sphere appropriately is usually difficult

Stephen Smale @ Berkeley: 18 unsolved math problems in 21st century

Question #7: distribution of points on the 2-sphere

No analytical solution available except for very few combination of N and d



Best Line Packing in Grassmannian Space

- Maximize the minimum angle between pair of vectors
  - Packing lines in Grassmannian space

- Neil J. A. Sloane @ AT &T Research
  - Known optimal line packing for selected m,and n
  - For most m and n: it is unknown
  - Computational approximations
  - http://www.research.att.com/~njas/grass/grassTab.html

- **Grassmannian Codes**
  - **Does NOT guarantee sub-matrices are well conditioned**
  - **It is even possible to contain singular sub-matices**

Real Number Codes with Optimal Numerical Stability

$$\begin{cases} C_1 = g_{11} * P_1 + \cdots + g_{1j} * P_{j+1} + \cdots + g_{1(j+m)} * P_{j+m} + \cdots + g_{1n} * P_n \\ \vdots \\ C_m = g_{m1} * P_1 + \cdots + g_{mj} * P_{j+1} + \cdots + g_{m(j+m)} * P_{j+m} + \cdots + g_{mn} * P_n \end{cases}$$

- **Recovery accuracy**
  - NOT directly related to the *correlation of pair of vectors*
  - BUT more directly related to the *condition number* of the coefficient matrix
  - The coefficient matrix may be any square sub-matrix of G
- **Real number codes with optimal numerical stability**
  - Optimize the recovery accuracy for the worst case scenario
  - Minimize maximum condition number of the coefficient matrix
  - Defined optimal real number codes as G that satisfies

$$f(m,n) = \min_{G} \max_{i} \kappa(G_i)$$

Optimal codes for 2 erasures

$$G = \begin{pmatrix} \cos\frac{\pi}{2n} & \cos\frac{3\pi}{2n} & \cdots & \frac{(2n-1)\pi}{2n} \\ \sin\frac{\pi}{2n} & \cos\frac{3\pi}{2n} & \cdots & \frac{(2n-1)\pi}{2n} \end{pmatrix}$$

$$f(2,n) = \sqrt{\frac{1+\cos\frac{\pi}{n}}{1-\cos\frac{\pi}{n}}}$$

$$\approx \frac{2n}{\pi} \quad \to \infty \qquad (n \to \infty)$$

- There is NO arbitrarily large matrix that satisfys

  Any sub-matrix of the matrix is well conditioned

- It is impossible even for the best 2-erasure codes to correct ALL possible 2-erasures when the number of data items (processors) is large
  - The introduced numerical errors can be arbitrarily large

Optimal codes for 2 erasures

$$G = \begin{pmatrix} \cos\frac{\pi}{2n} & \cos\frac{3\pi}{2n} & \cdots & \frac{(2n-1)\pi}{2n} \\ \sin\frac{\pi}{2n} & \cos\frac{3\pi}{2n} & \cdots & \frac{(2n-1)\pi}{2n} \end{pmatrix}$$

$$f(2,n) = \sqrt{\frac{1+\cos\frac{\pi}{n}}{1-\cos\frac{\pi}{n}}}$$

$$\approx \frac{2n}{\pi} \quad \to \infty \qquad (n \to \infty)$$

- In order to guarantee to correct ALL possible 2-erasures in IEEE standard 754 floating point numbers (16 digits of accuracy) with k digits of accuracy
  - The total number of data items (processors) *n* must satisfy

$$n \leq 10^{16-k} \cdot \frac{\pi}{2}$$

- When the number of data items (or pocessors) is larger than 10^(16-k)
  - 100% recovery can be guaranteed by dividing data items (or processors) into subgroup of less than 10^(16-k) items

Experimental Results: Worst Case Comparison

Tolerate 3 failures in 10 numbers: 120 possible combinations.
Compare worst 5 condition numbers of all possible 3X3 sub-matrices

Random	**47.2541**	**96.3024**	**111.9643**	**161.5512**	**217.1479**
Grassmannian	**0.5334*10^17**	**0.7204*10^17**	**2.3102* 0^17**	**2.3102*10^17**	**Inf**
Optimal	**14.7503**	**15.6580**	**16.1104**	**16.1609**	**16.5355**

Random:

```
0.2143 -0.2966 0.2635  0.0381 -0.0060 0.0777 -0.2428 -0.0233  0.2121 -0.2288
-0.1691 0.0577 -0.2430 0.1148  0.4060 0.0591 -0.1413 -0.3526 -0.3728 -0.1803
0.3306  0.2845 -0.3741 0.2420  0.4698 -0.4113 0.4380 -0.1750 -0.2025 -0.0038
```

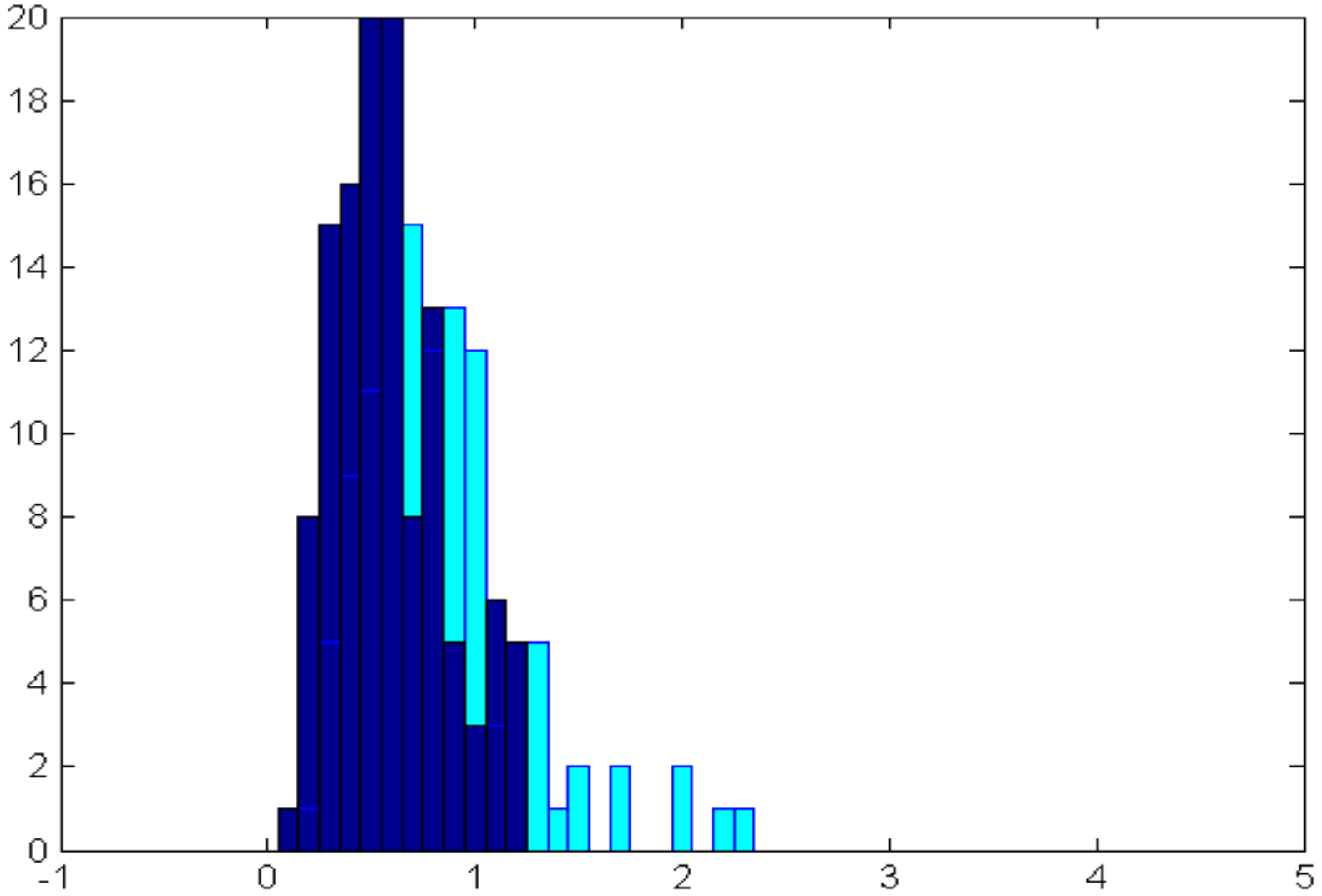
Grassmannian (Sloane@A&TT)

```
1      0.6101 0.6101 0.6101  0.6101  0.6101  0.6101  0       0       0
0      0.7923 0.3961 -0.3961 -0.7923 -0.3961 0.3961  0.8660 -0.8660 0
0      0      0.6861 0.6861  0       -0.6861 -0.6861 0.5000 0.5000  -1
```

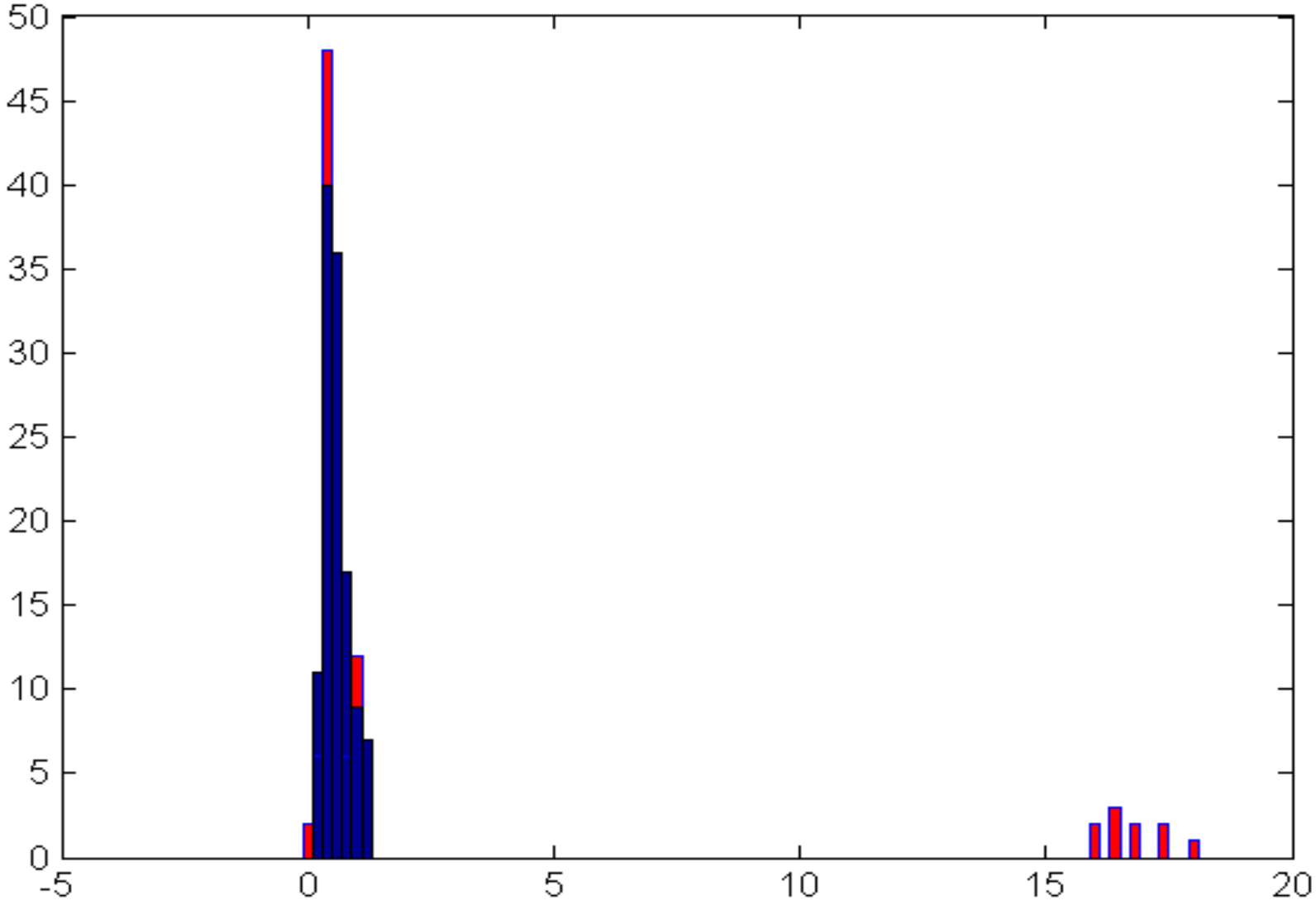
Optimal

```
-0.5566 0.1467 0.7247 0.9919 0.4631 -0.6691 0.5614  -0.2353 0.0686 -0.6749
0.8095  0.7985 0.4905 0.1217 -0.1332 0.2351 -0.6914 -0.0325 0.9466 -0.5804
0.1871  0.5839 0.4839 0.0365 0.8763  0.7050 0.4547  0.9714  -0.3149 0.4556
```

Distribution of all 120 condition numbers: Random vs Optimal



Distribution of all 120 condition numbers: Grassmannian vs Optimal



Conclusions and Future Work

- Several scalable fault tolerance techniques have been developed to survive a small number of process failures in large parallel computing.
  - Extended the existing diskless checkpointing technique to improve the scalability in large scale computing
  - Designed checkpoint-free fault tolerance technique for parallel matrix computations
  - Developed a class of numerically stable floating-point erasure codes to help to survive multiple simultaneous processes failures
- Experiment results demonstrate that these techniques are highly scalable.
- Future work: relieve the burden of FT from the application programmer
  - Incorporate the scalable checkpointing technique into the Open MPI library
  - Build the checkpoint-free technique into ScaLAPACK and PETSc (from ANL)

Questions?