中国科学院大学
夏季强化课程
2022

Fast Solvers for Large Algebraic Systems

Lecture 7. Fault resilience and reliability

7

容错方法与可靠性

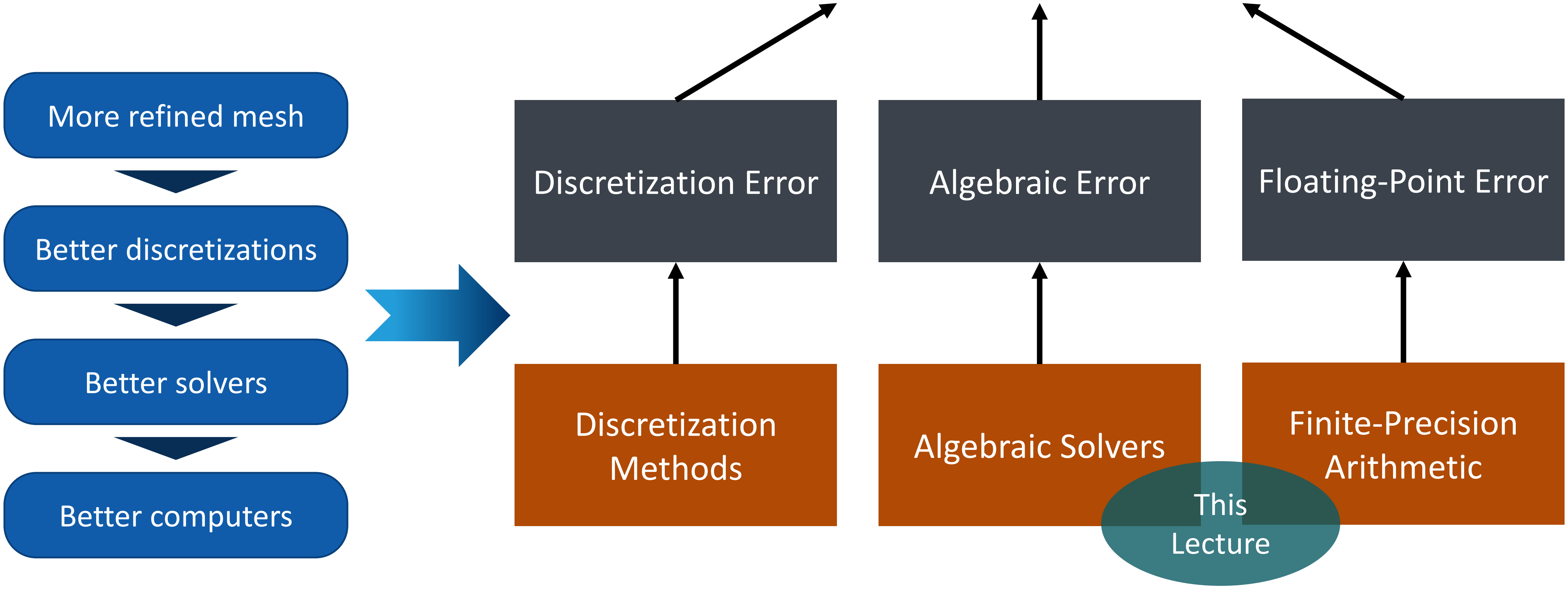
Chensong Zhang, AMSS
http://lsec.cc.ac.cn/~zhangcs

Table of Contents

- Lecture 1: Large-scale numerical simulation
- Lecture 2: Fast solvers for sparse linear systems
- Lecture 3: Methods for non-symmetric problems
- Lecture 4: Methods for nonlinear problems
- Lecture 5: Mixed-precision methods
- Lecture 6: Communication hiding and avoiding
- **Lecture 7: Fault resilience and reliability**
- Lecture 8: Robustness and adaptivity

Sources of Error in Simulation

Approximation: $$u(x) = U_h(x) + \mathcal{E}_{\text{dis}} + \mathcal{E}_{\text{alg}} + \mathcal{E}_{\text{fp}}$$



Introduction

Reliability and resilience of algebraic solvers

/01

Wake-Up Calls

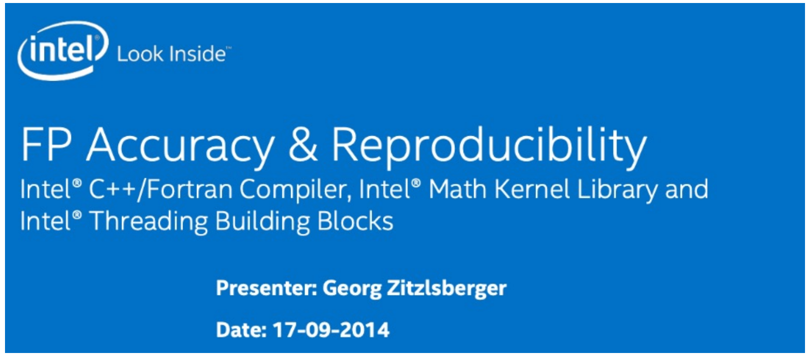
Intel® Xeon® Processor E5540

```
C:\Users\me>test.exe
4.012345678901111

C:\Users\me>test.exe
4.012345678901111
```

≠

Intel® Xeon® Processor E3-1275

```
C:\Users\me>test.exe
4.012345678902222

C:\Users\me>test.exe
4.012345678902222
```



LGPL v3 Free Software
Free as in Freedom

License

This software is free software distributed under the Lesser General Public License or LGPL, version 3.0 or any later versions. This software distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with FASP. If not, see http://www.gnu.org/licenses/.

Can you trust what was produced by your code, or by anyone's code?

Reproduce Baseline Results

- **Why we need reproducibility?**
- **How to enhance reproducibility?**

01

For upgrading and portability

In many cases, you have a trust-worthy existing code. Correctness of new code is determined by comparing to baseline results.

02

For debugging and optimization

When debugging or optimizing a code, run-to-run stability of the code is necessary in order to find potential problems.

03

To gain trust from users

End users from other teams or customers will be puzzled by the inconsistent results produced from your code.

Who Broke My Code

- When?
  - ✓ From one run to other run
  - ✓ From one compiler to another
  - ✓ From one option to another
  - ✓ From one computer to another

- Who?
  - ✓ Compiler
  - ✓ Compiler options
  - ✓ Libraries linked
  - ✓ Multi-threading

Reproducibility

- Produce the same result at any time
- Need deterministic behaviors
- Give up possible optimizations

VS

Performance

- Produce a result as fast as possible
- Do whatever best for speed
- Usually the results are close

Some Possibilities to Consider

01

Floating-Point Calculations

- FP numbers (IEEE754) have finite precisions
- Rounding happens for each (intermediate) result

02

Numerical Algorithms

- Some algorithms are less stable numerically
- Conditional computation with different input data

03

Task/Thread Scheduling

- Asynchronous task/thread scheduling may use different threads in each run
- Ordering is not preserved

A simple FP64 example with arithmetic ordering:

If you think the difference is small, think again!

- Exact: $2^{-53} + 1 - 1 = 2^{-53}$, $2^{-52} + 1 - 1 = 2^{-52}$
- FP64: $(2^{-53} + 1) - 1 = \mathbf{0.0}$, $2^{-53} + (1 - 1) = \mathbf{2^{-53}}$
- FP64: $(2^{-52} + 1) - 1 = \mathbf{2^{-52}}$, $2^{-52} + (1 - 1) = \mathbf{2^{-52}}$

Effects of Floating-Point Arithmetic

- We are doing floating-point calculations ➔ Nothing is "exact"

$$\mathrm{fl}(x \text{ op } y) = (x \text{ op } y)(1+\delta), \quad |\delta| \le u$$

- Review Lecture 5 for the FP precisions (e.g. Book by N. Higham 2002)
- Compilers might be doing something for you, under the hood!

Reassociation: (a + b) + c ➔ a + (b + c)

Division to multiplication: a / b ➔ a * (1 / b)

Zero folding: a + 0 ➔ a, a * 0 ➔ 0 (But "a" might be **INF** or **NaN**)

Math functions: sin(a), log(a), … (No international standard on these functions)

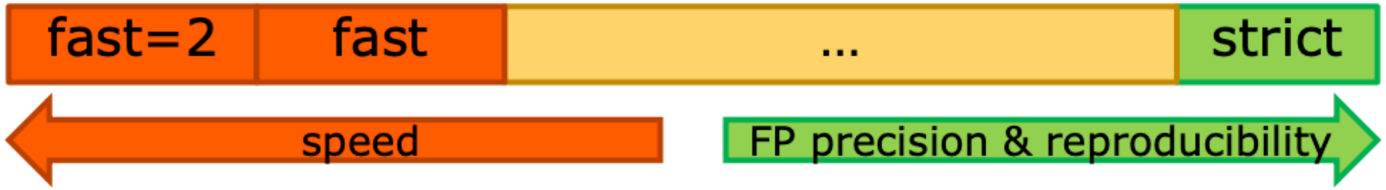
Subnormal FP numbers and underflow (flush to zero or not)

Use fused multiply-add (FMA) units or not

Different code paths (SIMD or non-SIMD, AVX2 v.s. SSE)

FP Model for Intel® Compilers

FP model	Description
precise	• Allow value-safe optimizations only
source	• Specify the intermediate precision used for FP expression evaluation
except	• Enable strict floating point exception semantics
strict	• Enable access to the FPU environment • Disable floating point contractions such as fused multiply-add instructions • Imply “precise” and “except”
fast[=1] (default)	• Allow value-unsafe optimizations compiler chooses precision for expression evaluation • Floating-point exception semantics not enforced • Access to the FPU environment not allowed • Floating-point contractions are allowed
fast=2	• Additional approximations allowed



Source: Georg Zitzlsberge, Intel, 2014

An Example in Reservoir Simulation

FP Model	Year = 0	Year = 1	Year = 5	Year = 10
Precise	3525.136841724400	3445.016272216141	3034.082337708471	2608.065832970213
Strict	3525.136841724400	3445.016272216141	3034.082337708471	2608.065832970213
Fast	**3525.136841724400**	**3445.0**50433013021	**3034**.789033440240	**260**9.016046486710

Pressure (psia) at one grid cell for SPE3 benchmark (compiled using Intel® oneAPI 2022)

- Although FP error is small, but not zero (review Lecture 5)
- Iterations might make it worse
- Time marching might make it even worse:

  Number of nonlinear iterations, number of linear iterations, time step sizes, …

An Example of FP Option

- A loop can be auto-vectorized (requires partial sum – reordering is done under the hood)

```
for (int i = 0; i < n; i++ ) sum = sum + x[i]*x[i];
```

- If compile with **–fp-model precise** or **–fp-model consistent**, the loop is not auto-vectorized

remark #15331: loop was not vectorized: precise FP model implied by the command line or a directive prevents vectorization.

- But you can do it by hand: adding OpenMP directives

```
#pragma omp simd reduction(+:sum)
    for (int i = 0; i < n; i++ ) sum = sum + x[i]*x[i];
```

remark #15301: OpenMP SIMD LOOP WAS VECTORIZED

HPC Top500 List, Revisited

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	1,110,144	151.90	214.35	2,942
4	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
5	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94.64	125.71	7,438
6	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93.01	125.44	15,371

SUPERCOMPUTER FUGAKU - SUPERCOMPUTER FUGAKU, A64FX 48C 2.2GHZ, TOFU INTERCONNECT D

Site:	RIKEN Center for Computational Science
System URL:	https://www.r-ccs.riken.jp/en/fugaku/project
Manufacturer:	Fujitsu
Cores:	7,630,848
Memory:	5,087,232 GB
Processor:	A64FX 48C 2.2GHz
Interconnect:	Tofu interconnect D
Performance	
Linpack Performance (Rmax)	442,010 TFlop/s
Theoretical Peak (Rpeak)	537,212 TFlop/s
Nmax	21,288,960
HPCG [TFlop/s]	16,004.5
Power Consumption	
Power:	29,899.23 kW (Optimized: **26248.36** kW)
Power Measurement Level:	2

Supercomputers are getting very complicated!

Bit Error Happens

Research based on large-scale data collected from the Google's servers (SIGMETRICS/Performance Conference 2009):

- The error rate is several orders of magnitude higher than the previous small-scale or laboratory studies
- Between 2.5×10^{-11} error/bit·h and 7×10^{-11} error/bit·h

  ➔➔➔ **How many bit errors per GB per hour?**
- How much memory does a supercomputer have?

  ➔➔➔ **Millions of GB!**
- Wow! A lot of errors!

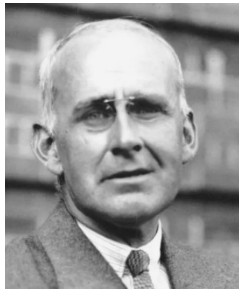
Can we trust what was produced by any computer code?

Reliability of Computation

Standard techniques used in practice for better reliability

/02

Reliability of Numerical Simulation



Never believe an experimental result until it has been confirmed by theory

Sir Arthur Stanley Eddington

01

Model Reliability

Mathematical model and its data describe the real physics with sufficient accuracy

02

Algorithm Reliability

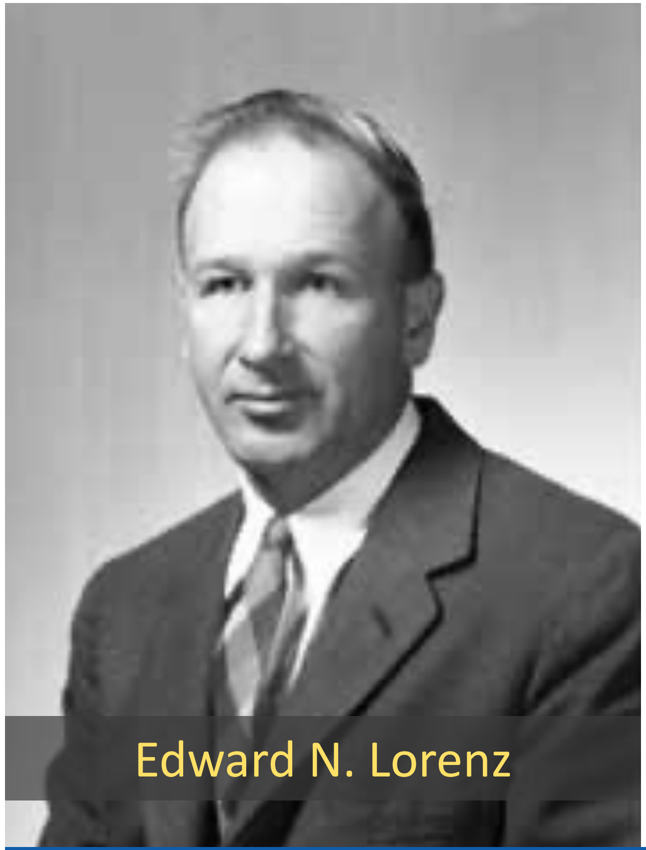
Underlying numerical algorithms should be stable, consistent, and convergent

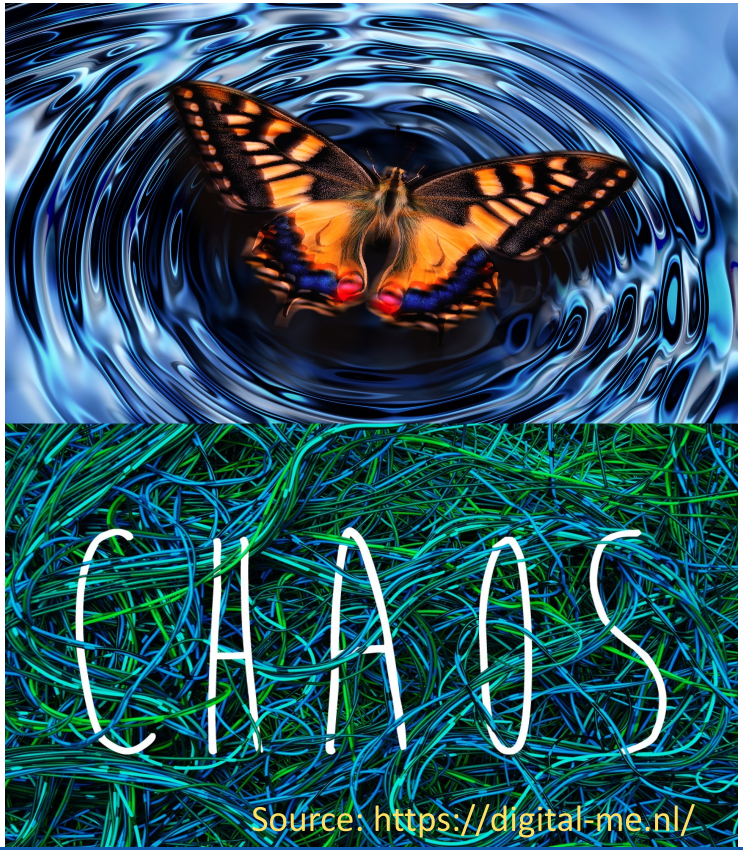
03

Software Reliability

The ability to perform its required functions under static conditions for a specific period

Why Results Might Not Be Reliable





In 1961, E. Lorenz was using a Royal McBee LGP-30 computer with 6-digit precision, to simulate weather. The printout rounded variables off to a 3-digit number, so a value like 0.506127 printed as 0.506. So he tried to input a rounded number, but …

Mean Time To Failure

MTTF (MTBF):

measures the average time a non-repairable asset operates before it fails

2 days

Ancient but stable

The mean time to failure of ENIAC (**1946**) [Alexander Randall 2006]

26.1

CPU failures per week

ASCI Q at LANL has 26.1 CPU failures per week [Michalak et al. **2005**]

4-6 hours

Soft error in L1 cache

BlueGene/L has one soft error in L1 cache every 4 to 6 hours [Bronevetsky, Supinski **2008**]

4.2 hours

Requires repair actions

Lessons learned from the analysis of system failures at petascale; see DSN **2014**

30 mins

Run without problems

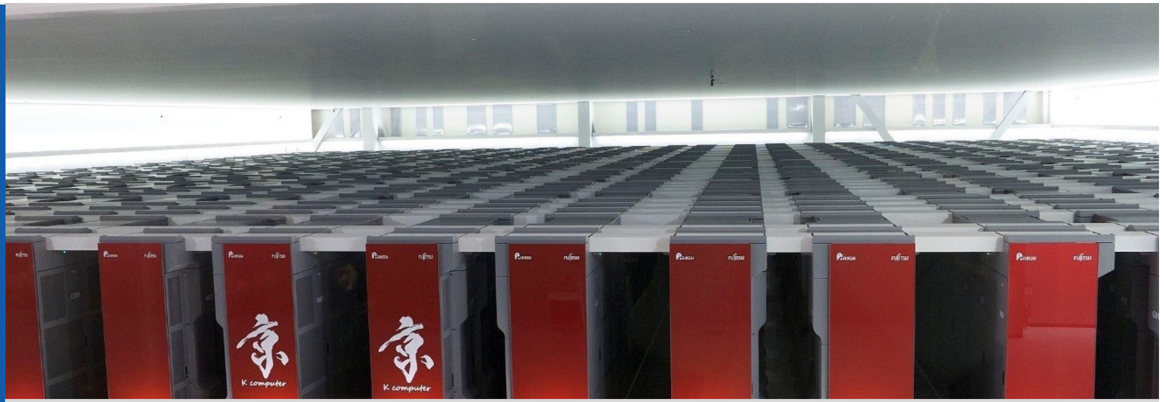
Researchers have predicted that large jobs may fail once every 30 minutes on **exascale** platforms

A Long Term Case Study

Measuring Reliability of HPC:

- MFR = Monthly Failure Rate
- AFR = Annualized Failure Rate = $1 - \exp(\frac{-1}{\text{MTBF}})$, Mean Time Between Failures measured in years
- FIT = Failure-In-Time = the amount of expected failures per one billion hours of operation
- Various studies have been carried out on Blue Waters, K-computer, Titan, Sunway, …

- K-computer: June 2019, Rmax = 10.510 PFLOPS
- 06/11--08/19 Peak: 864 cabinets, 88128 nodes, 705024 cores, 12.6 MW

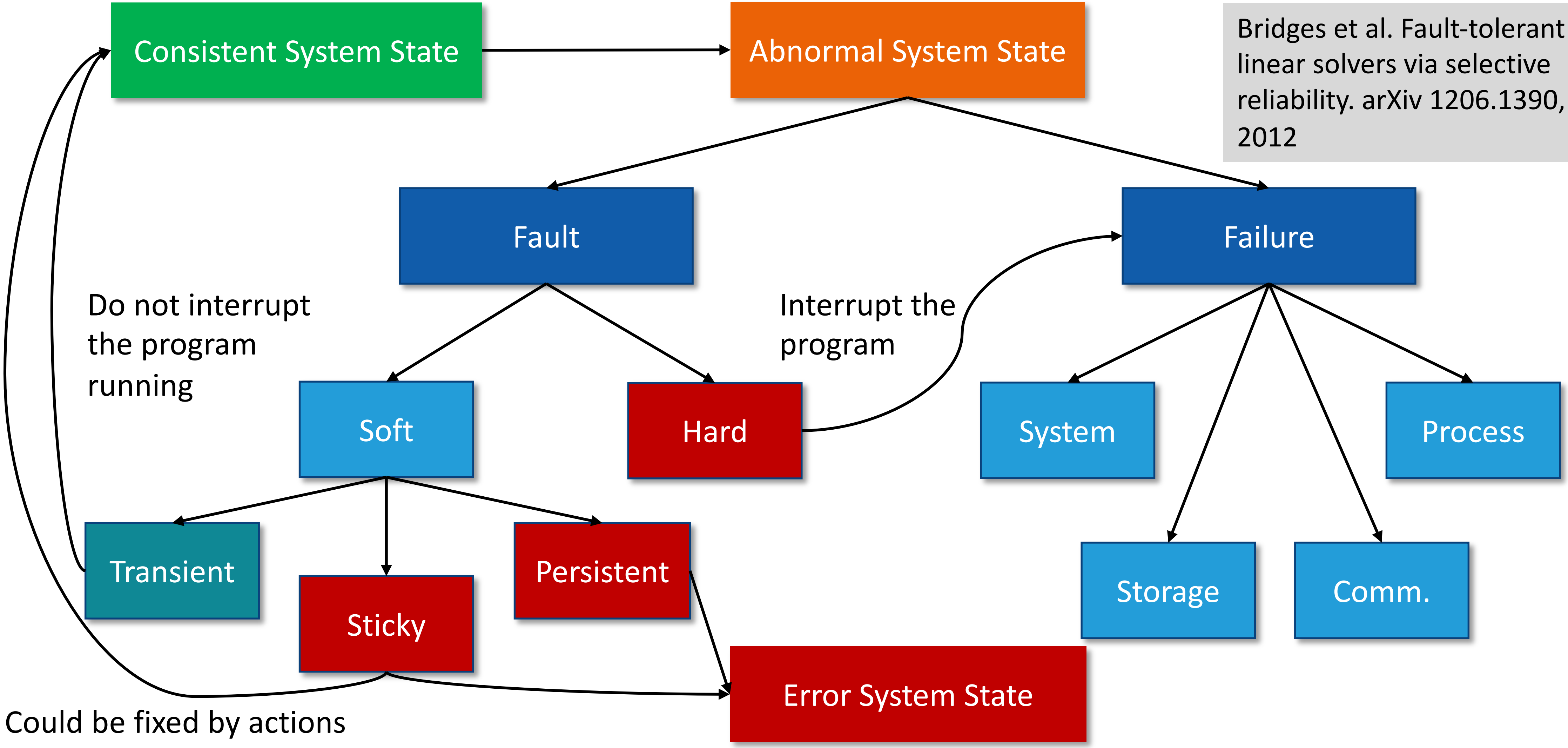


Ref: Fumiyoshi Shoji, Shuji Matsui, Mitsuo Okamoto, Fumichika Sueyasu, Toshiyuki Tsukamoto, Atsuya Uno and Keiji Yamamoto. “Long term failure analysis of 10 petascale supercomputer”. ISC’15

Parts	Number Parts	AFR	MTBF	FIT
CPU	82,944	0.06%	7.33 days	72.0
Memory	663,552	0.0016%	34.4 days	18.0

Analysis based on data from 2011.4 to 2015.4

Modeling Erroneous States



Bridges et al. Fault-tolerant linear solvers via selective reliability. arXiv 1206.1390, 2012

ECC-Enabled Devices

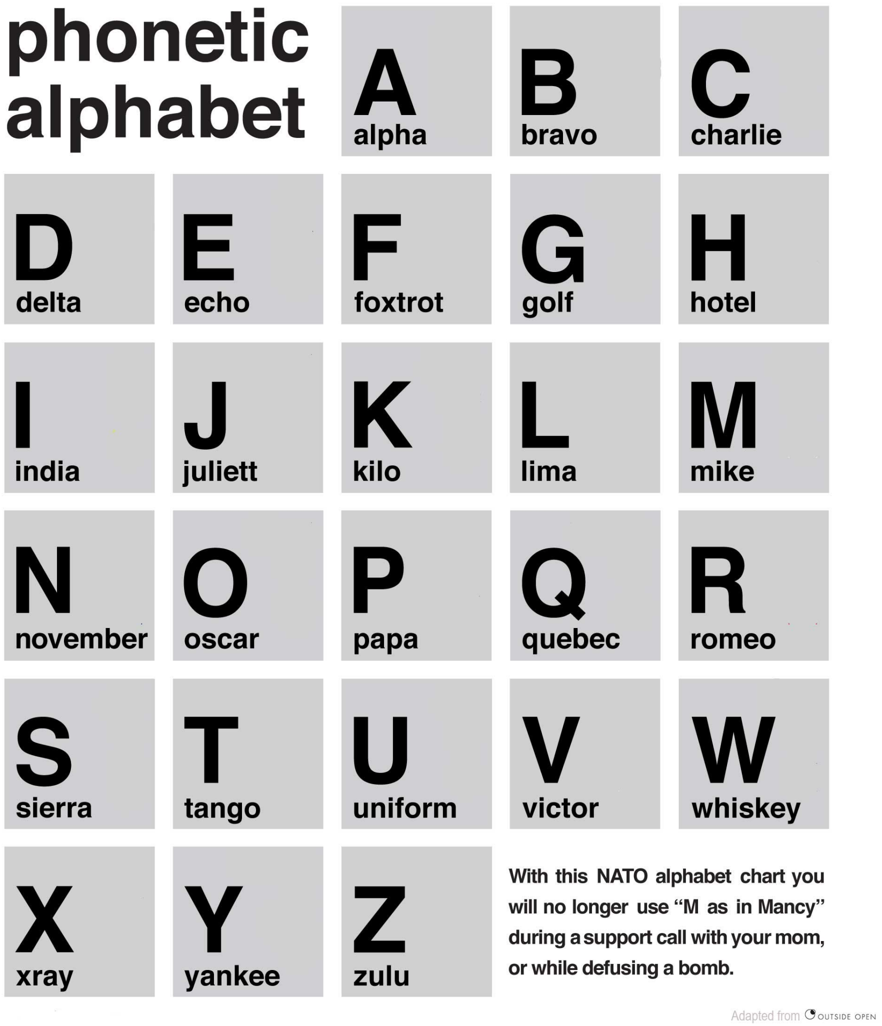
Various ECC technologies available:

- SECDED ECC: single error correction, double error detection; corrects single-bit faults and detects double-bit faults
- Chipkill ECC:
  - Chipkill-detect ECC detects any error from a single DRAM device
  - Chipkill-correct ECC corrects any error from a single DRAM device
  - Apparently, chipkill-detect and chipkill-correct ECC’s exhibit much lower undetected error rates than if using SEC-DED ECCs
  - Also cost more

Questions:

- Why does ECC work? How does ECC work? What’s the cost?

Simple Error Correction Code



If we have unlimited resources

Cost!

Reliability

Received Message	Interpreted bit
000	0
001	0
010	0
100	0
011	1
101	1
110	1
111	1

Examples of ECC

Some linear error correction codes:

- Repetition codes
- Parity codes
- Cyclic codes
- Hamming codes
- Golay code, both the binary and ternary versions
- Polynomial codes, of which BCH codes are an example
- Reed–Solomon codes
- Reed–Muller codes
- Goppa codes
- Low-density parity-check codes
- Expander codes
- Multidimensional parity-check codes
- Toric codes
- Turbo codes

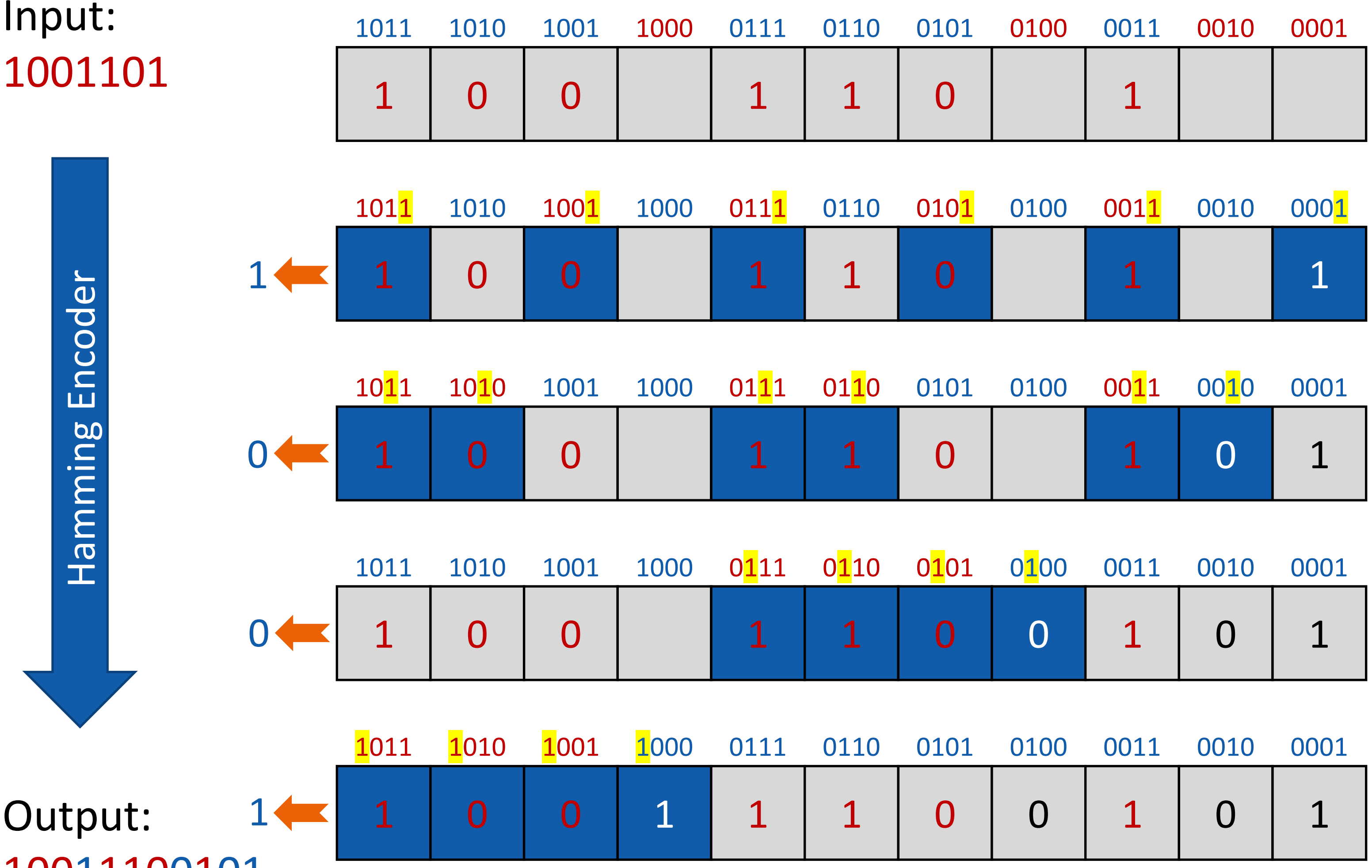
Source: https://en.wikipedia.org/wiki/Linear_code#Examples

- **Hamming codes** have a minimum distance of 3 -- the decoder can detect and correct a single error, but it cannot distinguish a double bit error from a single bit error -- SECDED
- **Extended Hamming code** is popular in computer memory systems (need an extra bit)

Ref: R. W. Hamming, "Error detecting and error correcting codes," in The Bell System Technical Journal, vol. 29, no. 2, pp. 147-160, April 1950, doi: 10.1002/j.1538-7305.1950.tb00463.x.

Hamming Code Encoder

Input:
1001101



Output:
10011100101

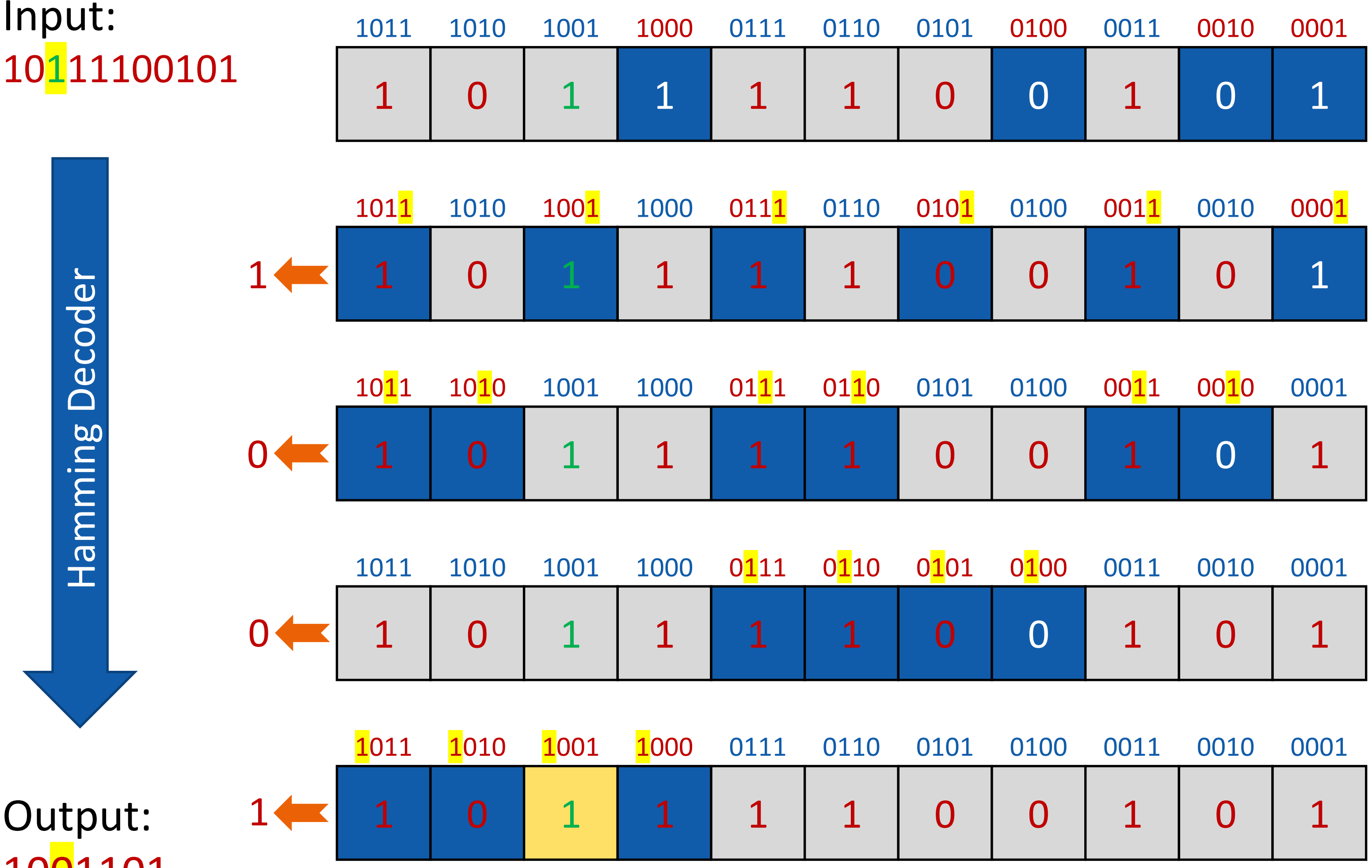
- Suppose we have n bits for coding
- Step 1 – Calculating the number of redundant bits r:

$$2^r \geq n + r + 1$$

- Step 2 – Positioning the redundant bits at locations $2^i, i = 0,1,2, \ldots$
- Step 3 – Calculating the values of each redundant bit by even (or odd) parity

Hamming Code Decoder

Input:
10111100101



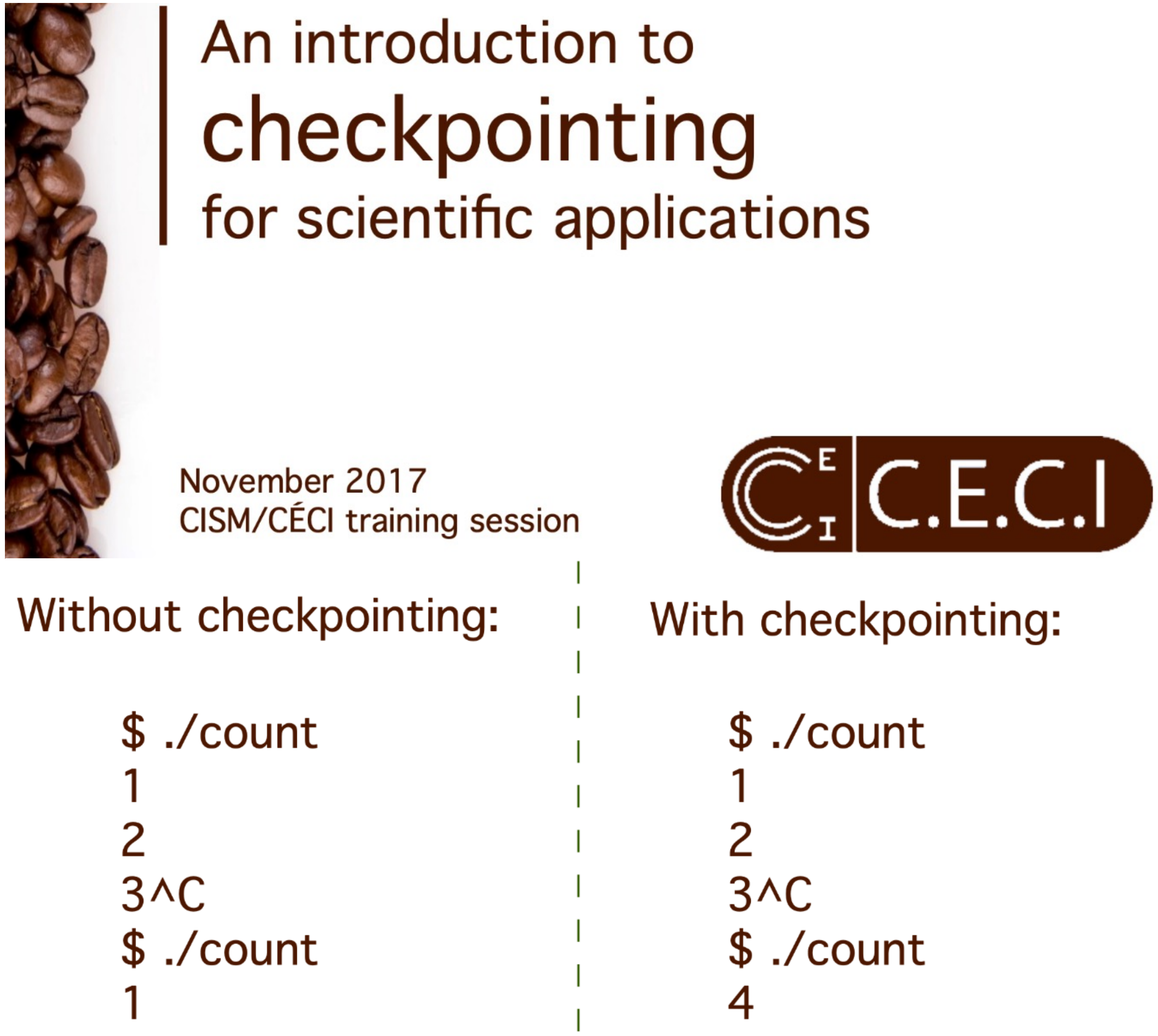
Output:
1001101

- Step 1 – Calculating the number of redundant bits r:

$$2^r \geq n + r + 1$$

- Step 2 – Positioning the redundant bits at locations $2^i, i = 0,1, \ldots$
- Step 3 – Checking parity and give an index of wrong bit
- Step 4 – Error detection and correction: if the index is 0 then correct, otherwise it is the wrong bit

Checkpointing



- Checkpointing (CP) is the action of saving the state of a running process to a checkpoint image
- The simulation can later be restarted (rollback or R) from the checkpoint, continuing from where it left off from any computer
- CP/R can be done in application-level or in system-level
- CP/R has been widely used in commercial and open-source software

A Simple CP/R Algorithm

Step 1. Look for a state file or image;

Step 2. If found, then restore the state (initialize all variables from the file or image);

else, create an initial state;

Step 3. Start and save state at suitable occasions

- Creating a transparent-to-users CP/R tool for HPC applications (system-level checkpointing) is very challenging, requiring extensive development and maintenance effort
  - Coordinated checkpointing: Processes coordinate checkpoints to form a system-wide consistent state
  - Independent or uncoordinated checkpointing: Each process has autonomy in deciding when to take checkpoints
- In distributed systems, rolling back one process can cause the roll back of others (Domino effect)
- The User-Level Failure Mitigation (ULFM) proposal to support the continued operation of MPI programs after crash (node failures)

Fault Tolerant Iterative Solvers

Iterative methods and preconditioners with built-in resilience

/03

Enhancing Fault-Tolerance

Goal: Maximize performance while resilient to moderate soft faults

- Failure of a process or node in a distributed system causes inconsistencies in the state of the system
- There are many type of errors and failures
- How to detect errors and failures? How to correct errors? How to recover from failures?
- This is still mainly computer science? What about numerical methods?
- FT algorithm: The algorithm is able to survive a failure at one or more processes or nodes

Algorithm-Based Fault Tolerance

- ABFT: Algorithm-based fault tolerance
- Q: Can a numerical simulator tell whether its result is correct or not?
- It is very unlikely! How can a simulator handle errors then? We must assume something …

Sandbox Model: The execution of untrusted "guest" code in a partition of the computer's state that protects the rest of the computer (or "host") from the guest's possibly bad behaviors.

- Many simulation spent most of the CPU time in (iterative) solvers
- Maintain convergence when error occurs assuming it is detectable
- Introduce low computational overhead when no error occurs
- Require only small amount of point-to-point communication (locality) while maintain good load balance (granularity)

Fault-Tolerant GMRES Method

Algorithm 21: FT-GMRES method

1 %% Given a nonsingular matrix $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$, and an initial guess $x \in \mathbb{R}^n$;
2 $r \leftarrow b - Ax$, $\beta \leftarrow \|r\|$, $q_1 \leftarrow r/\beta$;
3 for $j = 1, 2, \ldots, m$
4   Solve $Az_j = q_j$ in a sandbox;
5     $w_j \leftarrow Az_j$;
6     $h_{ij} \leftarrow (w_j, q_i)$, $w_j \leftarrow w_j - h_{ij}q_i$, $i = 1, 2, \ldots, j$;
7     $h_{j+1,j} \leftarrow \|w_j\|$;
8   if $h_{j+1,j} < \varepsilon$
9   if $H(1:j, 1:j)$ is full rank
10   Done at this iteration;
11   else
12   Recover from error;
13   end
14   else
15         $q_{j+1} \leftarrow w_j / h_{j+1,j}$;
16   end
17 end
18 $\bar{H}_m \leftarrow \{h_{ij}\}_{1\leqslant i\leqslant m+1, 1\leqslant j\leqslant m}$;
19 $y_m \leftarrow \mathrm{argmin}_y \|\beta e_1 - \bar{H}_m y\|$;
20 Update: $x \leftarrow x + Z_m y_m$;

Only one line is running in a sandbox, not reliable; and it is the most time-consuming part

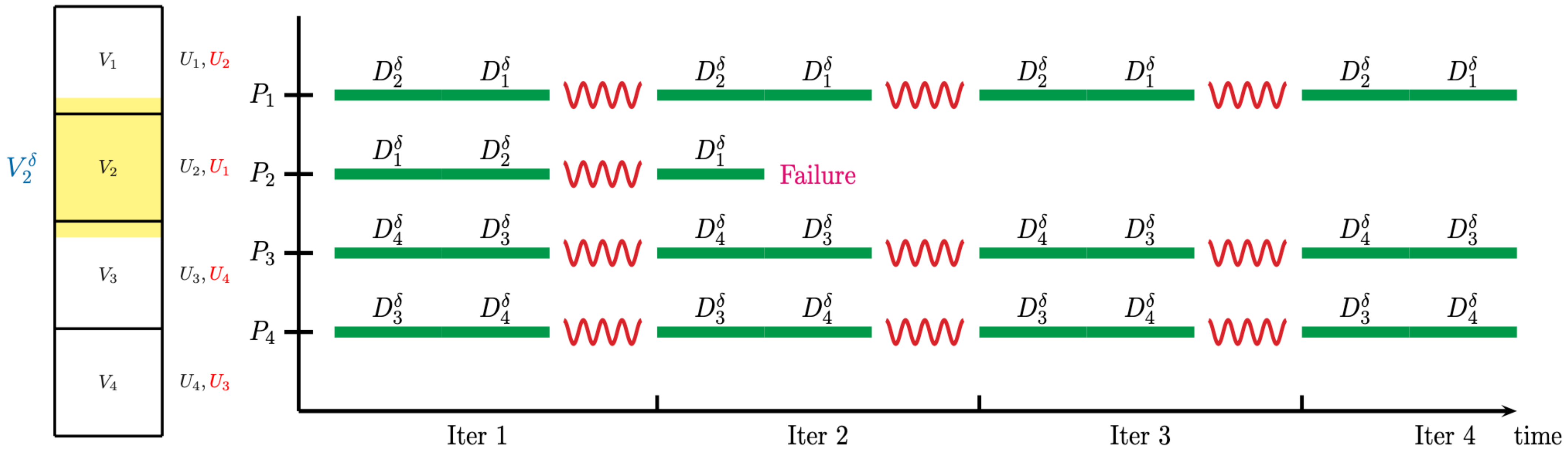
We had lucky breakdown before!

But how?
Retry from Line 4 or just return the last iteration

Does it look familiar?

Ref: Patrick G. Bridges, Kurt B. Ferreira, Michael A. Heroux, Mark Hoemmen. Fault-tolerant linear solvers via selective reliability. 2012 https://arxiv.org/abs/1206.1390

Redundant Subspace Correction



- Light overhead: non-global communication needed
- Resilient to temporary or permanent hardware failures
- Redundancy to maintain convergence when components fail
- Globally parallel and locally successive subspace correction

Divide-and-Conquer

Detecting Runtime Failures

Runtime Level Failure Detection and Propagation in HPC Systems

Dong Zhong
The University of Tennessee
Knoxville, TN, USA

Aurelien Bouteiller
The University of Tennessee
Knoxville, TN, USA

Xi Luo
The University of Tennessee
Knoxville, TN, USA

George Bosilca
The University of Tennessee
Knoxville, TN, USA

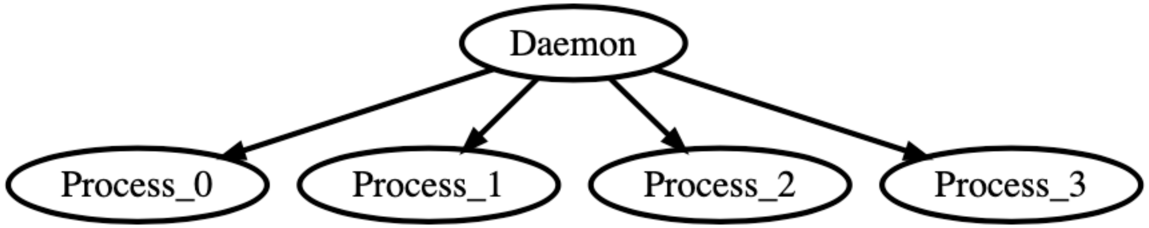


Figure 1: Hierarchical notification of hosted processes through PMIx notification routines. The PRRTE daemon is in charge of observing, and forward notifications to the node-local managed application processes. The detection and reliable broadcast topology operates at the node level between daemons.

PRSC for Poisson: Weak Scaling

DOFs	#Cores	Error-Free			With Error		
		#Iter	Time	Efficiency	#Iter	Time	Efficiency
1,335,489	16	12	8.09	—	13	8.13	—
2,146,689	32	13	8.64	75.25%	15	8.99	72.68%
4,243,841	64	14	8.91	72.13%	16	9.37	68.93%
10,584,449	128	19	12.87	62.27%	20	13.95	57.73%
16,974,593	256	23	18.01	35.68%	25	19.13	33.76%
33,751,809	512	25	20.90	30.57%	27	26.11	24.59%

Table: Weak-scaling of the PRSC preconditioner for the Poisson equation

Source: Tao Cui, Jinchao Xu and Chen-song Zhang. “An error-resilient redundant subspace correction method.” Computing and Visualization in Science 18 (2017): 65-77.

PRSC for Poisson: Error Effect

#Cores	Standard B_{PSC}			B_{PRSC} Error-Free			B_{PRSC} With Error		
	#Iter	Time	Speedup	#Iter	Time	Speedup	#Iter	Time	Speedup
16	42	162.9	—	21	163.4	—	24	165.3	—
32	48	82.71	1.97	25	82.79	1.97	27	83.35	1.98
64	48	41.14	3.96	26	42.13	3.88	28	43.62	3.79
128	50	20.95	7.78	27	22.66	7.21	29	23.95	6.91
256	51	11.84	13.8	27	13.46	12.1	29	14.03	11.8
512	52	6.91	23.6	28	7.43	21.9	29	7.79	21.2

Table: Strong-scaling of the PRSC for the Poisson equation (16,974,593 DOFs)

Source: Tao Cui, Jinchao Xu and Chen-song Zhang. “An error-resilient redundant subspace correction method.” Computing and Visualization in Science 18 (2017): 65-77.

PRSC for More Problems

Example	DOFs	Standard B_{PSC}		B_{PRSC} Error-Free		B_{PRSC} With Error	
		#Iter	Time	#Iter	Time	#Iter	Time
Poisson	1,335,489	23	7.92	12	8.09	13	8.13
Maxwell	468,064	42	4.09	21	4.23	24	4.48
Elasticity	436,515	16	10.18	9	11.01	10	11.35

Table: Convergence of the PRSC preconditioner (16 cores)

Source: Tao Cui, Jinchao Xu and Chen-song Zhang. "An error-resilient redundant subspace correction method." Computing and Visualization in Science 18 (2017): 65-77.

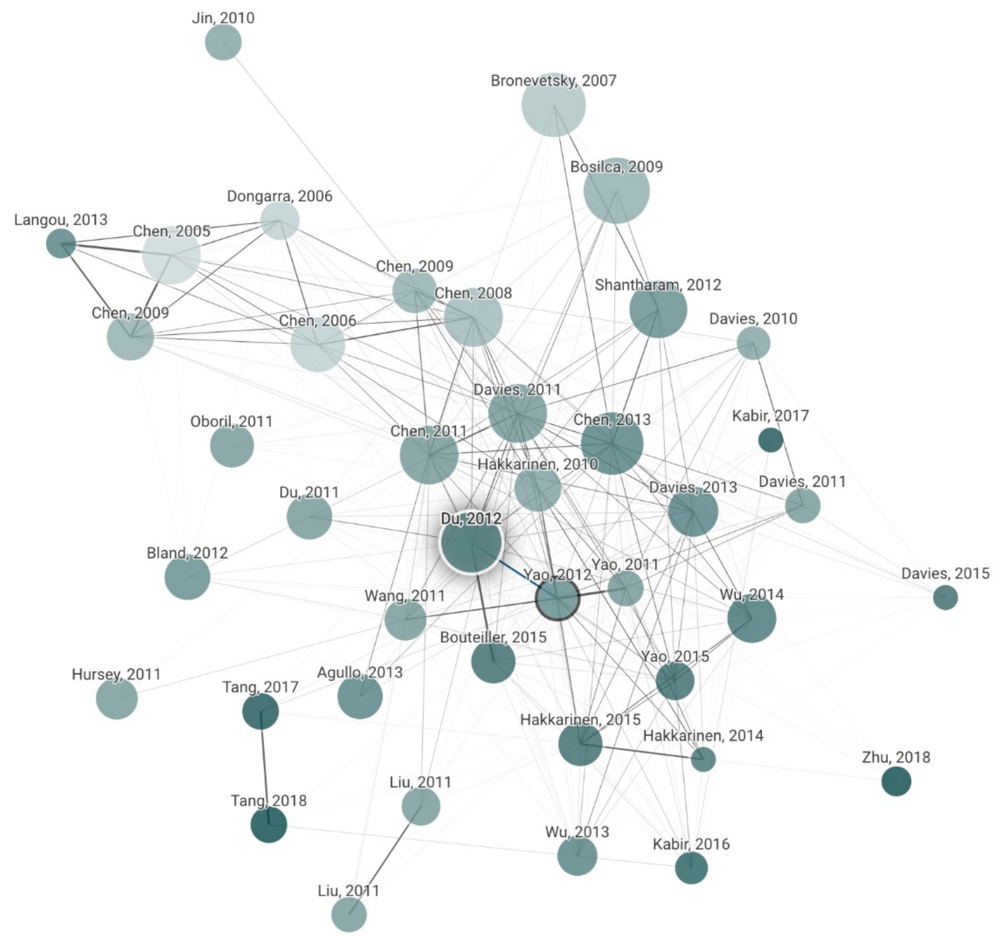
More Failures

#Cores	1 failure		2 failures		4 failures		8 failures		16 failures	
	#Iter	Time	#Iter	Time	#Iter	Time	#Iter	Time	#Iter	Time
16	24	165.3	32	192.5	38	225.8	46	261.2	—	—
32	27	83.35	37	107.1	42	119.5	45	122.1	50	129.8
64	28	43.62	35	50.4	36	52.1	42	57.9	49	65.5
128	29	23.95	31	24.2	36	27.1	41	30.3	47	34.6

Table: Convergence of the PRSC with failures for Poisson (16,974,593 DOFs)

Source: Tao Cui, Jinchao Xu and Chen-song Zhang. “An error-resilient redundant subspace correction method.” Computing and Visualization in Science 18 (2017): 65-77.

ABFT Papers



Algorithm-based fault tolerance for dense matrix factorizations

Peng Du, Aurélien Bouteiller, G. Bosilca, T. Hérault, J. Dongarra

2012, PPoPP '12

136 Citations

Save

Open in:

Dense matrix factorizations, such as LU, Cholesky and QR, are widely used for scientific applications that require solving systems of linear equations, eigenvalues and linear least squares problems. Such computations are normally carried out on supercomputers, whose ever-growing scale induces a fast decline of the Mean Time To Failure (MTTF). This paper proposes a new hybrid approach, based on Algorithm-Based Fault Tolerance (ABFT), to help matrix factorizations algorithms survive fail-stop failures. We consider extreme conditions, such as the absence of any reliable component and the possibility of loosing both data and checksum from a single failure. We will present a generic solution for protecting the right factor, where the updates are applied, of all above mentioned factorizations. For the left factor, where the panel has been applied, we propose a scalable checkpointing algorithm. This algorithm features high degree of checkpointing parallelism and cooperatively utilizes the checksum storage leftover from the right factor protection. The fault-tolerant algorithms derived from this hybrid solution is applicable to a wide range of dense matrix factorizations, with minor modifications. Theoretical analysis shows that the fault tolerance overhead sharply decreases with the scaling in the number of computing units and the problem size. Experimental results of LU and QR factorization on the Kraken (Cray XT5) supercomputer validate the theoretical evaluation and confirm negligible overhead, with- and without-errors.

Source: https://www.connectedpapers.com/

Stopping Criteria and Reliability

Algorithm 2: Conjugate gradient method

%% Given an initial guess u and a tolerance ε;

$r \leftarrow f - Au, \; p \leftarrow r;$

while $\|r\| > \varepsilon$

  $\alpha \leftarrow (r,r)/(Ap,p);$

  $\tilde{u} \leftarrow u + \alpha p;$

  $\tilde{r} \leftarrow r - \alpha Ap;$ ← Iterative updating

  $\beta \leftarrow (\tilde{r},\tilde{r})/(r,r);$

  $\tilde{p} \leftarrow \tilde{r} + \beta p;$

  Update: $u \leftarrow \tilde{u}, \; r \leftarrow \tilde{r}, \; p \leftarrow \tilde{p};$

end

- ✓ In some (maybe most) code for iterative methods, we check L2-norm of residual
- ✓ To save time, we update the residual using iteration instead of computing it
- ✓ $\|r\|_{A^{-1}} = \|e\|_A$

Ref: Hestenes and Stiefel. "Methods of Conjugate Gradients for Solving Linear Systems". Journal of Research of the National Bureau of Standards 49, 1952

Reading and Thinking

SIAM J. Sci. Comput.
Vol. 39, No. 2, pp. C116–C143

IS THE MULTIGRID METHOD FAULT TOLERANT? THE TWO-GRID CASE*

MARK AINSWORTH† AND CHRISTIAN GLUSA‡

This paper is dedicated to Professor Ivo Babuška on the occasion of his 90th birthday

Abstract. The predicted reduced resiliency of next-generation high performance computers means that it will become necessary to take into account the effects of randomly occurring faults on numerical methods. Further, in the event of a hard fault occurring, a decision has to be made as to what remedial action should be taken in order to resume the execution of the algorithm. The action that is chosen can have a dramatic effect on the performance and characteristics of the scheme. Ideally, the resulting algorithm should be subjected to the same kind of mathematical analysis that was applied to the original, deterministic variant. The purpose of this work is to provide the first rigorous analysis of the behavior of the multigrid algorithm in the presence of faults. Specifically, we prove estimates on the behavior of the Two Grid Method similar to the classical asymptotic results. Multigrid is arguably the method of choice for the solution of large-scale linear algebra problems arising from discretization of partial differential equations, and it is of considerable importance to anticipate its behavior on an exascale machine. The analysis of resilience of algorithms is in its infancy, and the current work is perhaps the first to provide a mathematical model for faults and analyze the behavior of a state-of-the-art algorithm under the model. It is shown that the Two Grid Method fails to be resilient to faults. Attention is then turned to identifying the minimal necessary remedial action required to restore the rate of convergence to that enjoyed by the ideal fault-free method.

Key words. multigrid, fault tolerance, resilience, random matrices, convergence analysis

AMS subject classifications. 65F10, 65N22, 65N55, 68M15

DOI. 10.1137/16M1100691

- Have you experienced errors when you program? Of course, you did.
- What types of errors you have made?
- Any of the errors not made by yourselves? What are they?
- How can you make your code more reliable?
- How can you make your algorithms more fault-tolerant? Any plans?

Contact Me

- Office hours: Mon 14:00—15:00
- Walk-in or online with appointment
- zhangcs@lsec.cc.ac.cn
- http://lsec.cc.ac.cn/~zhangcs

My sincere gratitude to:

Tao Cui, Shizhe Li, Bin Dai, Yan Xie

中国科学院大学
夏季强化课程
2022

Fast Solvers for Large Algebraic Systems

THANKS

Chensong Zhang, AMSS

http://lsec.cc.ac.cn/~zhangcs

Release version 2022.07.10