Fault Tolerance for Numerical Library Routines

Jack Dongarra

University of Tennessee
Oak Ridge National Laboratory
University of Manchester
Overview

• Can we easily generate LA library quality software that provides protection from faults?

• Can we protect against both hard and soft failures?

• Are the overheads reasonable?
Problem Definition

- **Dense Linear Algebra**
  - Direct methods: LU, Cholesky, QR, Singular values, and eigenvalue problems, ...

- Interested in understanding how we could enable this software to be fault tolerant without a total rewrite of the library.

- **System Errors**
  - Hard Error (single & multiple)
    - Fail stop model, process completely and definitely stops
  - Silent (soft) Error (single & multiple)
    - Detect and correct say bit flips

- **Platforms/Applications**
  - Distributed memory system
    - Eg. ScaLAPACK and DPLASMA, with MPI
## Last Generations of DLA Software

Software/Algorithms follow hardware evolution in time

<table>
<thead>
<tr>
<th>Period</th>
<th>Software/Algorithms</th>
<th>Dependencies</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>70's</td>
<td>LINPACK</td>
<td>Rely on Level-1 BLAS operations</td>
<td>Vector operations</td>
</tr>
<tr>
<td>80's</td>
<td>LAPACK</td>
<td>Rely on Level-3 BLAS operations</td>
<td>Blocking, cache friendly</td>
</tr>
<tr>
<td>90's</td>
<td>ScaLAPACK</td>
<td>Rely on PBLAS Mess Passing</td>
<td>Distributed Memory</td>
</tr>
</tbody>
</table>
Algorithm Based Fault Tolerance (ABFT)

- Matrix extended to contain additional information.
  - Extra column or row contains checksum.
- Algorithm designed to operate on the data and the encoded checksum.
- Checksum invariant during the course of the algorithm.
- No checkpoint needed.

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

\(G^T\) and \(G^T A\) and \(BG\) are the check sums.

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

- $C$ matrix contains a checksum (e.g. 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

Here the generate, $G$, is a column of all one’s.
$G^T = (1, 1, ..., 1)$

\[ Update(\sum_j A_{ij}) = \sum_j Update(A_{ij}) \]
ABFT Idea

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

\[ C_i = \sum_j A_{ij} \]
\[ A_{ij} = C_i - \sum_{k \neq j} A_{ik} \]

Here the generate, G, is a column of all one’s. \( G^T = (1, 1, ..., 1) \)

In case of failure, checksum inversion allows to restore the missing value
Detection & Local Correction through ABFT Strategy

- Before the factorization starts, a checksum is taken
- Column(s) appended to the matrix
- Factorization proceeds with the augmented matrix
- Check the results after completion or use checksum to recover missing information.

- Checksum invariance

\[ c_1 = Ae_1, \text{ where } e_1 = (1,1,...,1)^T \]

\[(A,c_1) \text{ Undergoes factorization, } QR \text{ in this case}\]

\[\tilde{(A,c_1)} \text{ Producing } (QR,\tilde{c_1})\]

\[QR e_1 = \tilde{c_1} \text{ (up to round off error)}\]
Checksum in ABFT

$A = ZU$

$A_c = (A, Ae, Aw)$

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

$c = Ue$

$v = Uw$

Generator, $G = (e, w)$

$e = \begin{pmatrix} 1 \\ 1 \\ \cdots \\ 1 \end{pmatrix}$

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

$w_i$: random number

We are using random numbers for the Generators.

Alan Edelman, 89: Condition number of random matrix $\log(n) + 1.5367$
With ScaLAPACK Data has a 2-D Block Cyclic Distribution

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

A $pxq$ process grid ($p = 2$, $q = 3$)

Logical grid of processes
For Scalability Reasons
2-D Block Cyclic Data Distribution

Matrix view

Process view

p x q process grid (2x3)
2-D Block Cyclic Data Distribution with Checksums

Matrix is extended with 2F columns for every q columns, here F = 1 and q = 3. Checksum blocks are doubled to allow recovery when data and checksum are lost together.
Failure Model
Make local copies of the group of q panels and the corresponding checksum
In case of Failure

- many blocks lost in data and checksums
- Checksum property not insured on trailing matrix
ABFT Overheads on ScaLAPACK Like Implementation

Matrix $M \times N$, Blocks $mb \times nb$, Process grid $p \times q$

F: maximum number of simultaneous failures tolerated

**Memory Overhead**

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

Matrix is extended with $2F$ columns every $q$ columns

**Computation Overhead**

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

flops for the checksum update, and

$$O(MN)$$

flops for the checksum creation.

Less than 5% computational overhead

*N.B.* Usually $F \ll q$

Relative overheads in $F/q$

e.g. 2 simultaneous faults on $192 \times 192$ process grid => 1% memory overhead
Kraken Cray XT5 system specifications:

- Cray Linux Environment (CLE) 3.1
- A peak performance of 1.17 PetaFLOP
- 112,896 compute cores
- 147 TB of compute memory
- A 3.3 PB raw parallel file system of disk storage for scratch space (2.4 PB available)
- 9,408 compute nodes

Each node has:

- Two 2.6 GHz six-core AMD Opteron processors (Istanbul)
- 12 cores
- 16 GB of memory
- Connection via Cray SeaStar2+ router
Performance for QR

![Graph showing performance and relative overhead over ScaLAPACK. The x-axis represents the number of processors (PxQ grid) and matrix size (N), with data points for 6x6; 20k, 12x12; 40k, 24x24; 80k, and 48x48; 160k. The y-axis represents performance (TFlop/s) and relative overhead over ScaLAPACK (%).]
Performance for QR

The graph shows the performance (in TFlop/s) of ScaLAPACK PDGEQRF, FT-PDGEQRF (no errors), and the overhead of FT-PDGEQRF (no errors) for various matrix sizes and processor grids. The x-axis represents the number of processors (PxQ grid) and the size of the matrix (N), while the y-axis shows the performance in TFlop/s. The relative overhead over ScaLAPACK is also plotted.
## Last Generations of DLA Software

<table>
<thead>
<tr>
<th>Software/Algorithms follow hardware evolution in time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LINPACK (70’s)</strong></td>
</tr>
<tr>
<td>(Vector operations)</td>
</tr>
<tr>
<td><img src="image1" alt="Diagram" /></td>
</tr>
<tr>
<td>Rely on</td>
</tr>
<tr>
<td>- Level-1 BLAS operations</td>
</tr>
<tr>
<td><strong>LAPACK (80’s)</strong></td>
</tr>
<tr>
<td>(Blocking, cache friendly)</td>
</tr>
<tr>
<td><img src="image2" alt="Diagram" /></td>
</tr>
<tr>
<td>Rely on</td>
</tr>
<tr>
<td>- Level-3 BLAS operations</td>
</tr>
<tr>
<td><strong>ScaLAPACK (90’s)</strong></td>
</tr>
<tr>
<td>(Distributed Memory)</td>
</tr>
<tr>
<td><img src="image3" alt="Diagram" /></td>
</tr>
<tr>
<td>Rely on</td>
</tr>
<tr>
<td>- PBLAS Mess Passing</td>
</tr>
</tbody>
</table>
Current Generation of DLA Software

<table>
<thead>
<tr>
<th>Software/Algorithms follow hardware evolution in time</th>
</tr>
</thead>
</table>
| **LINPACK (70’s)**
  (Vector operations) | ![Diagram](image) | Rely on
  - Level-1 BLAS operations |
| **LAPACK (80’s)**
  (Blocking, cache friendly) | ![Diagram](image) | Rely on
  - Level-3 BLAS operations |
| **ScalAPACK (90’s)**
  (Distributed Memory) | ![Diagram](image) | Rely on
  - PBLAS Mess Passing |
| **PLASMA**
New Algorithms
(many-core friendly) | ![Diagram](image) | Rely on
  - a DAG/scheduler
  - block data layout
  - some extra kernels |
Parallelization of LU and QR.

Parallelize the update:
- Easy and done in any reasonable software.
- This is the $2/3n^3$ term in the FLOPs count.
- Can be done efficiently with LAPACK+multithreaded BLAS

```
  LU   A^{(1)}
  
  dgetf2 --> lu()

  dtrsm (+ dswp)

  dgemm

  LU   A^{(2)}

  Fork - Join parallelism
  Bulk Sync Processing
```
PLASMA LU Factorization

Dataflow Driven

Numerical program generates tasks and run time system executes tasks respecting data dependences.

Sparse / Dense Matrix System

DAG-based factorization

Batched LA

- LU, QR, or Cholesky on small diagonal matrices
- TRSMs, QRs, or LUs
- TRSMs, TRMMs
- Updates (Schur complement) GEMMs, SYRKs, TRMMs

Each task, node in graph, is a matrix-matrix operations. Level 3 BLAS operation on tiles.
PLASMA - DAG Based Version

**Objectives**
- High utilization of each core
- Scaling to large number of cores
- Synchronization reducing algorithms

**Methodology**
- Dynamic DAG scheduling
- Explicit parallelism
- Implicit communication
- Fine granularity / block data layout

**Arbitrary DAG with dynamic scheduling**

Fork-join parallelism
Notice the synchronization penalty in the presence of heterogeneity.
PLASMA - DAG Based Version

**Objectives**
- High utilization of each core
- Scaling to large number of cores
- Synchronization reducing algorithms

**Methodology**
- Dynamic DAG scheduling
- Explicit parallelism
- Implicit communication
- Fine granularity / block data layout

**Arbitrary DAG with dynamic scheduling**

Notice the synchronization penalty in the presence of heterogeneity.
Motivation: Goal

- Failure model
  - **Soft error** (Silent Data Corruption): bit-flips, memory or processor registers
  - Here we focus on soft errors happening during computation

- Resilience / Fault Tolerance in dynamic task-based runtime
  - Implemented in PaRSEC, the runtime system for DPLASMA
  - Two levels of granularities and three mechanisms: Looking into DAG and task.
  - Case study on the Cholesky factorization
Introduction to PaRSEC

• Application representation

User’s view

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A₀₀</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A₁₀</td>
<td>A₁₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A₂₀</td>
<td>A₂₁</td>
<td>A₂₂</td>
<td></td>
</tr>
<tr>
<td>A₃₀</td>
<td>A₃₁</td>
<td>A₃₂</td>
<td>A₃₃</td>
</tr>
</tbody>
</table>

| Final result |
| POTRF |
| TRSM |
| SYRK |
| GEMM |

Runtime’s view

Silent error

FOR k = 0 .. SIZE - 1
A[k][k], T[k][k] <- GEQRT( A[k][k] )
FOR m = k+1 .. SIZE - 1
A[k][k][Up], A[m][k], T[m][k] <- TSQRT( A[k][k][Up], A[m][k], T[m][k] )
FOR n = k+1 .. SIZE - 1
A[k][n] <- UNMQR( A[k][k][Low], T[k][k], A[k][n] )
FOR m = k+1 .. SIZE - 1
A[k][n], A[m][n] <- TSMQR( A[m][k], T[m][k], A[k][n], A[m][n] )

Tasks:
PO: POTRF
TR: TRSM
GE: GEMM
SY: SYRKF
TR: TRSM
GE: GEMM
SY: SYRKF
Error Happening
Error Propagation
Node0
Node1
Node2
Node3
Detection & Local Correction through ABFT Strategy

• Implementation
  (1) Attaching 2 checksum vectors to original data
  (2) Provide recovery scheme inside task
  (3) Continue with the DAG execution
Detection & Local Correction through ABFT Strategy

- **Checksum invariance**

  - Matrix Operation (BLAS, LU, QR, etc)
  - Update
  - Checksum1 $C_1$
  - Checksum2 $C_2$

  \[
  \begin{align*}
  C_1 &= A g_1 \\
  C_2 &= A g_2 \\
  g_1 &= (1,1,...,1)^T \\
  g_2 &= (1,2,...,n)^T
  \end{align*}
  \]

  (*because of round-off errors, a small tolerance is allowed*)

- **Single bit Detection & correction**
  - After update, single bit error happens at $A(i,j)$
  \[
  \sum_{k=1}^{n} A(k,j) - C_1(j) = \gamma \quad \Rightarrow \text{error is in column } j
  \]
  \[
  \sum_{k=1}^{n} kA(k,j) - C_2(j) = i \gamma \quad \Rightarrow \text{error is in row } i
  \]

  \[
  A'(i,j) = A(i,j) - \gamma \quad \Rightarrow \text{adding the difference to recover}
  \]
2. A single bit flip in task

1) Applying ABFT method (avoid re-execution)

Attach 2 checksum vectors to every tile. Tile size is NB x NB.

① Overhead (time)
- Maintaining checksums: \( (1 + \frac{2}{NB})^3 - 1 \)
- Detecting & correcting error: \( \frac{1}{NB} \)
- Total: \( (1 + \frac{2}{NB})^3 - 1 + \frac{1}{NB} \)

② Overhead (Storage) => \( \frac{2}{NB} \)
Experiment Platform

- Machine: Titan in ORNL
- CPU: AMD Opteron™ 6274 (Interlagos)
  - 16 Cores, 8 FPUs
  - We use 8 cores per CPU, ensuring 1 FPU per core (no GPUs)
- Weak scaling experiments:
  - 6k x 6k matrix distributed on 1 node
  - Run up to 256 nodes
• Experiment 1: Single Bit Flip. ABFT Correction
• Experiment 1: Single Bit Flip. ABFT Correction
• Experiment 1: Single Bit Flip. ABFT Correction
Experiment Assumed Single Bit Flip, but...

- In a matrix computation an error may propagate.

- In this case need to save inputs and restart the task.


Single bit causes a rank-1 change to the factorization.
• Experiment 2: Save Task’s Inputs Locally and Restart Task
• Experiment 2: Save Task’s Inputs Locally and Restart Task
Experiment 2: Save Task’s Inputs Locally and Restart Task

![Graph showing performance and overhead for different matrix sizes and failure conditions.](image-url)
Sub-DAG & Periodic Checkpoint Strategy

• $\beta = 2$ example
  • Checkpointing intermediate data, limit the number of re-executions.
  • Checkpoint interval $\beta$, a process will save a copy of each data every $\beta$ updates.
  • Input of failed task:
    • The same tile checkpointed at most $\beta$ updates ago
    • Final output of another task (validated)
  • Max number of re-executions is $\beta$ for factorizations
• **Experiment 3: Checkpoint every 10 updates**

![Graph showing DPLASMA DPOTRF Performance](image)
• Experiment 3: Checkpoint every 10 updates

![Graph showing performance (Tflop/s) vs. matrix size (number of nodes)]
• Experiment 3: Checkpoint every 10 updates
Conclusion & Future work

• Conclusion
  • ABFT enables Software Verification / Validation of computation
  • Designing ABFT schemes for specific tasks is easier than designing ABFT schemes for the whole application
  • Fairly straightforward to port to other DPLASMA routines
    • Two application level mechanisms can be implemented in the runtime, PaRSEC, functions.
    • ABFT mechanism has similar idea to extend to other DLA routines.

• Future work
  • Integrate with accelerators
  • Investigate automatic Checkpointing and Rollback-Recovery
  • Extend to support fail stop model (hard error)

• Thanks to George Bosilca, Thomas Herault, Aurelien Bouteiller, Piotr Luszczek, Peng Du, Yulu Jai, and Chongxiao Cao
Dense Factorization

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

GETF2: factorize a column block

```
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
```
Performance for LU

![Graph showing performance and relative overhead over ScaLAPACK for different matrix sizes and processor counts. The x-axis represents the number of processors (P x Q grid) and matrix size (N), while the y-axis shows performance (TFlop/s) and relative overhead (ScaLAPACK PDGESV).]
Performance for LU

![Graph showing performance for LU with different grid sizes and matrix sizes, with overheads for ScaLAPACK PDGESV and FT-PDGESV (with and without errors).](image)
Techniques for Error Protection and Failure Recovery

- **Algorithm-Based Fault Tolerance**
    - 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.**
Reverse Neighboring Scheme

Elements are really blocks

The matrix of size $M \times N$, Blocked in blocks of $mb \times nb$
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