Self Adapting Numerical Software (SANS) – Effort and Fault Tolerance in Linear Algebra Algorithms

Jack Dongarra
University of Tennessee
and
Oak Ridge National Laboratory

Overview

♦ Quick look at fastest computers
  ➢ From the November Top500
♦ Techniques for fault tolerant computations for iterative methods
  ➢ Strategies when we start to using 10’s of thousands of processors

H. Meuer, H. Simon, E. Strohmaier, & JD

- Listing of the 500 most powerful Computers in the World
- Yardstick: Rmax from LINPACK MPP
  \[ Ax = b, \text{ dense problem} \]
- Updated twice a year
  SC’xy in the States in November
  Meeting in Mannheim, Germany in June

All data available from www.top500.org

24th List: The TOP10

<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Computer</th>
<th>Rmax (TF/s)</th>
<th>Installation Site</th>
<th>Country</th>
<th>Year</th>
<th>#Proc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IBM</td>
<td>70.72</td>
<td>DOE/IBM</td>
<td>USA</td>
<td>2004</td>
<td>32768</td>
</tr>
<tr>
<td>2</td>
<td>NEC</td>
<td>51.87</td>
<td>NASA Ames</td>
<td>USA</td>
<td>2004</td>
<td>00150</td>
</tr>
<tr>
<td>3</td>
<td>IBM</td>
<td>57.86</td>
<td>Japan</td>
<td>2002</td>
<td>5120</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>IBM</td>
<td>50.93</td>
<td>ESVS</td>
<td>Spain</td>
<td>2004</td>
<td>00150</td>
</tr>
<tr>
<td>5</td>
<td>IBM</td>
<td>19.94</td>
<td>Lawrence Livermore</td>
<td>USA</td>
<td>2004</td>
<td>4096</td>
</tr>
<tr>
<td>6</td>
<td>IBM</td>
<td>12.88</td>
<td>Lawrence Livermore</td>
<td>USA</td>
<td>2002</td>
<td>6192</td>
</tr>
<tr>
<td>7</td>
<td>HP</td>
<td>12.25</td>
<td>Virginia Tech</td>
<td>USA</td>
<td>2004</td>
<td>2200</td>
</tr>
<tr>
<td>8</td>
<td>IBM</td>
<td>11.68</td>
<td>Lawrence Livermore</td>
<td>USA</td>
<td>2004</td>
<td>6192</td>
</tr>
<tr>
<td>9</td>
<td>IBM</td>
<td>10.31</td>
<td>NCSA</td>
<td>USA</td>
<td>2004</td>
<td>2944</td>
</tr>
<tr>
<td>10</td>
<td>Dell</td>
<td>9.82</td>
<td>NCSA</td>
<td>USA</td>
<td>2003</td>
<td>2500</td>
</tr>
</tbody>
</table>

399 system > 1 TFlop/s; 294 machines are clusters, top10 average 8K proc

Architectures / Systems

- SIMD
- Single Proc.
- Cluster
- Constellation
- SMP
- MPP
Interconnects / Systems

Processor Types

Fuel Efficiency: Gflops/Watt

Top 20 systems
Based on processor power rating only
IBM BlueGene/L
131,072 Processors

System
(64 racks, 64x32x32)

131,072 procex

Node Card
(32 Node boards, 8x8x1)

16 Proc cards
64 processors

8096 processors is today’s “large”
we’re struggling even here

How Big Is Big?

♦ Every 10X brings new challenges
  ➢ 64 processors was once considered large
  ➢ it hasn’t been “large” for quite a while
  ➢ 1024 processors is today’s “medium” size
  ➢ we’ve been “large” for quite a while
  ➢ we’re struggling even here

♦ 100K processor systems
  ➢ are in construction
  ➢ we have fundamental challenges in dealing with
    machines of this size
  ➢ ... and little in the way of programming support

Fault Tolerance: Motivation

♦ Trends in HPC:
  ➢ High end systems with thousand of processors
  ➢ Increased probability of a node failure
    ➢ Most systems nowadays are robust
  ➢ MPI widely accepted in scientific computing
    ➢ Process faults not tolerated in MPI model

Mismatch between hardware and (non fault-tolerant) programming paradigm of MPI.

Related work

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.

FT-MPI Failure Recovery Modes

♦ ABORT: Just do as other MPI implementations.
♦ BLANK: Leave hole in communicator.
♦ SHRINK: Re-order processes to make a contiguous communicator.
  ➢ Some ranks change
♦ REBUILD: Re-spawn lost processes and add them to
  MPI_COMM_WORLD.
**Fault Tolerance in the Computation**

- Some next generation systems are being designed with > 100K processors (IBM Blue Gene L).
- MTTF 10^9 - 10^6 hours for component.
- Sounds like a lot until you divide by 10^6!
- Failures for such a system can be just a few hours, perhaps minutes away.
- Problem with the MPI standard, no recovery from faults.
- FT-MPI allows user to provide recovery.
- Application checkpoint / restart is today’s typical fault tolerance method.
- Many cluster based on commodity parts don’t have error correcting primary memory.

**Fault Tolerance - Diskless Checkpointing Built into Software**

- Checkpointing to disk is slow.
  - May not have any disks on the system.
  - Have extra checkpointing processors.
  - Use “RAID like” checkpointing to processor.
- Maintain a system checkpoint in memory.
  - All processors may be rolled back if necessary.
  - Use k extra processors to encode checkpoints so that if up to k processors fail, their checkpoints may be restored (Reed-Solomon encoding).
- Idea to build into library routines.
  - We are looking at iterative solvers.
  - Not transparent, has to be built into the algorithm.

**How Raid for a Disk System Works**

- Similar to RAID for disks.
- If \( X = A \ XOR B \) then this is true:
  - \( X \ XOR B = A \)
  - \( A \ XOR X = B \)

**How Diskless Checkpointing Works**

- The N 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.

- When failure occurs:
  - Control passes to user supplied handler.
  - “XOR” performed to recover missing data.
  - P4 takes on role of P1.
  - Execution continues.

- P4 takes on the identity of P1 and the computation continues.
A Fault-Tolerant Parallel Conjugate Gradient Solver

- Tightly coupled computation.
- Do a “backup” (checkpoint) every \( j \) iterations for changing data.
  - Requires each process to keep copy of iteration changing data from checkpoint.
- First example can survive the failure of a single process.
- Dedicate an additional process for holding data, which can be used during the recovery operation.

- For surviving \( k \) process failures \((k \ll p)\) you need \( k \) additional processes (second example).

CG Data Storage

Think of the data like this

- 3 vectors change every iteration

Initial data is fixed throughout the iteration.

Checkpoint A and b

Parallel Version

Think of the data like this

- No need to checkpoint each iteration, say every \( j \) iterations.
- Need a copy of the 3 vectors from checkpoint in each processor.

Diskless Version

Extra storage needed on each processor from the data that is changing.

Actually don’t do XOR, add the information.

FT PCG Algorithm Analysis

Compute \( r(0) = \beta - \lambda \alpha \) for some initial guess \( x(0) \)

for \( i = 1, 2, \ldots \)

solve \( A x(i-1) = y(i) \)

if \( i = 1 \)

\( \rho(i) = \frac{\| r(i-1) \|^2}{\| r(i) \|^2} \)

\( p(i) = \alpha(i) \)

\( x(i) = x(i-1) + \alpha(i) p(i) \)

\( p(i) = p(i-1) \)

\( \rho(i) = \frac{\| r(i) \|^2}{\| r(i-1) \|^2} \)

\( x(i) = x(i-1) + \alpha(i) p(i) \)

Global operation: three dot product, one preconditioning, and one matrix vector multiplication.

Global operation in PCG: three dot product, one preconditioning, and one matrix vector multiplication.

FT PCG Algorithm Analysis

Compute \( r(0) = \beta - \lambda \alpha \) for some initial guess \( x(0) \)

for \( i = 1, 2, \ldots \)

solve \( A x(i-1) = y(i) \)

if \( i = 1 \)

\( \rho(i) = \frac{\| r(i-1) \|^2}{\| r(i) \|^2} \)

\( p(i) = \alpha(i) \)

\( x(i) = x(i-1) + \alpha(i) p(i) \)

\( p(i) = p(i-1) \)

\( \rho(i) = \frac{\| r(i) \|^2}{\| r(i-1) \|^2} \)

\( x(i) = x(i-1) + \alpha(i) p(i) \)

Global operation: three dot product, one preconditioning, and one matrix vector multiplication.

Global operation in Checkpoint: encoding the local checkpoint.

Global operation in Checkpoint: encoding the local checkpoint.
PCG: Test Problems (Matrices)

- **bcsstk17**
  - The size is: 10974 x 10974
  - Non-zeros: 428650
  - Sparsity: 39 non-zeros per row on average
  - Source: Linear equation from elevated pressure vessel

PCG: Experiment Configurations

<table>
<thead>
<tr>
<th>Problem</th>
<th>Size of the Problem</th>
<th>No. of Comp. Procs</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>1,316,880</td>
<td>16</td>
</tr>
<tr>
<td>#2</td>
<td>858,440</td>
<td>16</td>
</tr>
<tr>
<td>#3</td>
<td>529,220</td>
<td>30</td>
</tr>
<tr>
<td>#4</td>
<td>229,360</td>
<td>60</td>
</tr>
</tbody>
</table>

All experiments are performed on:
- 64 dual-processor 2.4 GHz AMD Opteron nodes
- Each node of the cluster has 2 GB of memory
- Each node runs the Linux operation system
- Nodes are connected with a Gigabit Ethernet.

PCG: Performance with Different MPI Implementations

- **Problem #1**: 658,440
- **Problem #2**: 329,220
- **Problem #3**: 164,610
- **Problem #4**: Size of the Problem

All experiments are performed on:
- 64 dual-processor 2.4 GHz AMD Opteron nodes
- Each node of the cluster has 2 GB of memory
- Each node runs the Linux operation system
- Nodes are connected with a Gigabit Ethernet.

Reed-Solomon Approach

- \( A \cdot P = C \) where \( A \) is \( k \times p \) made up of random numbers, \( P \) is \( p \times n \), \( C \) is \( k \times n \)
- Here using 4 processors and 3 Checkpoint processors:
  - \( \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \end{pmatrix} = \begin{pmatrix} C_1 \\ C_2 \\ C_3 \end{pmatrix} \)

Reed-Solomon Approach

- \( A \cdot P = C \) where \( A \) is \( k \times p \) made up of random numbers, \( P \) is \( p \times n \), \( C \) is \( k \times n \)
- Here using 4 processors and 3 Checkpoint processors:
  - \( \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \end{pmatrix} = \begin{pmatrix} C_1 \\ C_2 \\ C_3 \end{pmatrix} \)

- Say 2 processors fail, \( P_2 \) and \( P_3 \)
Reed-Solomon Approach

\[ A^*P = C, \]  
where \( A \) is \( k \times p \) made up of random numbers, \( P \) is \( p \times n \), \( C \) is \( k \times n \)

Here using 4 processors and 3 Ckpt processors:

\[
\begin{pmatrix}
  a_{11} & a_{12} & a_{13} & a_{14} \\
  a_{21} & a_{22} & a_{23} & a_{24} \\
  a_{31} & a_{32} & a_{33} & a_{34}
\end{pmatrix}
\begin{pmatrix}
  P_1 \\
  P_2 \\
  P_3
\end{pmatrix}
= 
\begin{pmatrix}
  C_1 \\
  C_2 \\
  C_3
\end{pmatrix}
\]

Say 2 processors fail, \( P_2 \) and \( P_3 \).
Take a subset of \( A \)'s (column 2 and 33) and solve for \( P_1 \) and \( P_2 \).
Could use GF(2), signal processing apps.

\[
\begin{pmatrix}
  a_{12} & a_{13} \\
  a_{22} & a_{23} \\
  a_{32} & a_{33}
\end{pmatrix}
\begin{pmatrix}
  P_1 \\
  P_2 \\
  P_3
\end{pmatrix}
= 
\begin{pmatrix}
  C_1 \\
  C_2 \\
  C_3
\end{pmatrix}
\]

\( P_{\text{erc}} = \frac{\text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}} \)

Next Steps

- **Large-scale fault tolerance**
  - adaptation: resilience and recovery
  - predictive techniques for probability of failure
  - resource classes and capabilities
  - resilience implementation mechanisms
  - adaptive checkpoint frequency
  - in memory checkpoints
- **By monitoring, one can identify**
  - performance problems
  - failure probability

Investigate ideas for 1K to 10K processors, then to BG/L.
- Software to determine the checkpointing interval and number of checkpoint processors from the machine characteristics.
  - Perhaps use historical information.
- Local checkpoint and restart algorithm.
  - Coordination of local checkpoints.
  - Processors hold backups of neighbors.
  - Have the checkpoint processes participate in the computation and do data rearrangement when a failure occurs.
  - Use \( p \) processors for the computation and have \( k \) of them hold checkpoints.
  - Generalize the ideas to provide a library of routines to do the diskless checking.
  - Look at “real applications” and investigate “Lossy” algorithms.
  - FT-MPI available today and one of the contributions to Open MPI.
Linpack (100x100) Analysis

- Compaq 386/SX20 SX with FPA, 16 Mflop/s
- Pentium IV, 2.8 GHz, 1.3 Gflop/s
- Moore’s Law says something about a factor of 2 every 18 months or a factor of 256 over 12 years
- Seem to be missing a factor of 32...
  - Clock speed increase = 128x
  - External Bus Width & Caching:
    - 16 vs. 64 bits = 4x
  - Floating Point:
    - 4/8 bits multi vs. 64 bits (1 clock) = 8x
  - Compiler Technology = 2x
- However, the theoretical peak for Pentium 4 is 5.6 Gflop/s and here we are only getting 1.3 Gflop/s
  - Still a factor of 4.25 off of peak

Performance Tuning Methodology

Software Installation (done once per system)
- Input Parameters
- System specifics
- User options
- Hardware Probe
- Parameter study of code versions
- Code Generation
- Performance Database
- Installation

Software Execution (done dynamically for each problem)
- Input Parameters
- System specifics
- User options
- Hardware Probe
- Parameter study of code versions
- Code Generation
- Performance Database
- Installation
- Run-time

Motivation Self Adapting Numerical Software (SANS) Effort

- Optimizing software to exploit the features of a given system has historically been an exercise in hand customization.
  - Time consuming and tedious
  - Hard to predict performance from source code
  - Must be re-done for every architecture and compiler
- Software technology often lags architecture
  - Best algorithm may depend on input, so some tuning may be needed at run-time.
- There is a need for quick/dynamic deployment of optimized routines.

Self Adapting Numerical Software - SANS Effort

- Provide software technology to aid in high performance on commodity processors, clusters, and grids.
- Pre-run time (library building stage) and run time optimization.
- Integrated performance modeling and analysis
- Automatic algorithm selection - polyalgorithmic functions
- Automated installation process
- Can be expanded to areas such as communication software and selection of numerical algorithms
Some Current Unmet Needs

- Performance / Portability
- Fault tolerance
- Better programming models
  - Global shared address space
  - Visible locality
- Maybe coming soon (incremental, yet offering real benefits):
  - Global Address Space (GAS) languages: UPC, Co-Array Fortran, Titanium)
  - "Minor" extensions to existing languages
  - More convenient than MPI
  - Have performance transparency via explicit remote memory references
- The critical cycle of prototyping, assessment, and commercialization must be a long-term, sustaining investment, not a one time, crash program.

Collaborators / Support

- Top500 Team
  - Erich Strohmaier, NERSC
  - Hans Meuer, Mannheim
  - Horst Simon, NERSC

- Fault Tolerant Work
  - Julien Langou, UTK
  - Jeffery Chen, UTK
- FT-MPI
  - http://icl.cs.utk.edu/ft-mpi/
  - Graham Fagg, UTK
  - Edgar Gabriel, HLRS
  - Thara Angskun, UTK
  - George Bosilca, UTK
  - Jelena Fjesinac-Grbovic, UTK

Real Crisis With HPC Is With The Software

- Programming is stuck
  - Arguably hasn’t changed since the 60’s
- It’s time for a change
  - Complexity is rising dramatically
  - Highly parallel and distributed systems
    - From 10 to 100 to 10000 to 100000 of processors!!
  - Multi-disciplinary applications
- A supercomputer application and software are usually much more long-lived than a hardware
  - Hardware life typically five years at most.
  - Fortran and C are the main programming models
- Software is a major cost component of modern technologies.
  - The tradition in HPC system procurement is to assume that the software is free.
- We don’t have many great ideas about how to solve this problem.