The National Center for Atmospheric Research

IBM Watson Research Center

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

Jack Dongarra
University of Tennessee
and
Oak Ridge National Laboratory

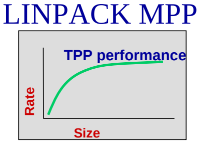
Overview

- **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**

TOP500

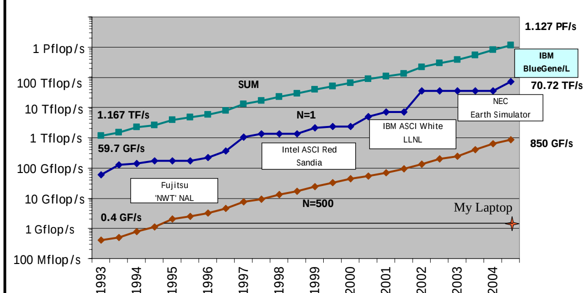
H. Meuer, H. Simon, E. Strohmaier, & JD

- Listing of the 500 most powerful Computers in the World
- Yardstick: Rmax from LINPACK MPP
  Ax=b, *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**

TOP500 Performance – November 2004

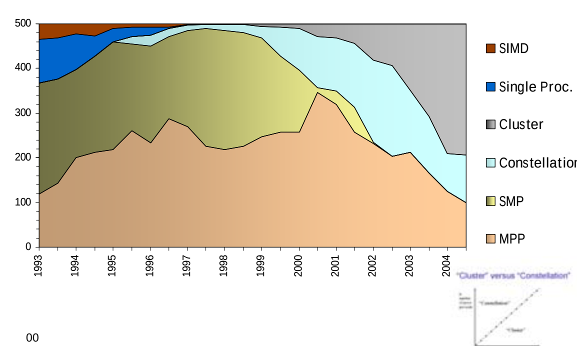


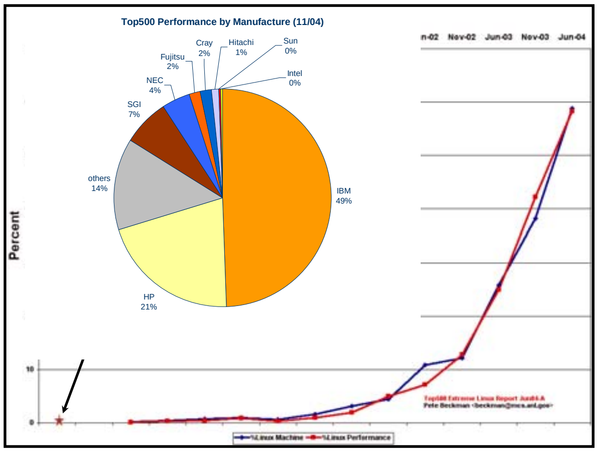
24th List: The TOP10

	Manufacturer	Computer	Rmax [TF/s]	Installation Site	Country	Year	#Proc
1	IBM	BlueGene/L β-System	70.72	DOE/IBM	USA	2004	32768
2	SGI	Columbia Altix, Infiniband	51.87	NASA Ames	USA	2004	10160
3	NEC	Earth-Simulator	35.86	Earth Simulator Center	Japan	2002	5120
4	IBM	MareNostrum BladeCenter JS20, Myrinet	20.53	Barcelona Supercomputer Center	Spain	2004	3564
5	CCD	Thunder Itanium2, Quadrics	19.94	Lawrence Livermore National Laboratory	USA	2004	4096
6	HP	ASCI Q AlphaServer SC, Quadrics	13.88	Los Alamos National Laboratory	USA	2002	8192
7	Self Made	X Apple XServe, Infiniband	12.25	Virginia Tech	USA	2004	2200
8	IBM/LLNL	BlueGene/L DD1 500 MHz	11.68	Lawrence Livermore National Laboratory	USA	2004	8192
9	IBM	pSeries 655	10.31	Naval Oceanographic Office	USA	2004	2944
10	Dell	Tungsten PowerEdge, Myrinet	9.82	NCSA	USA	2003	2500

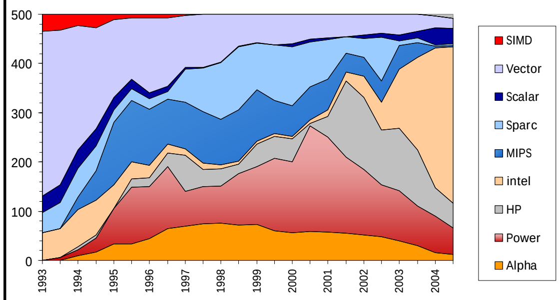
399 system > 1 TFlop/s; 294 machines are clusters, top10 average 8K proc
#39 NCAR

Architectures / Systems





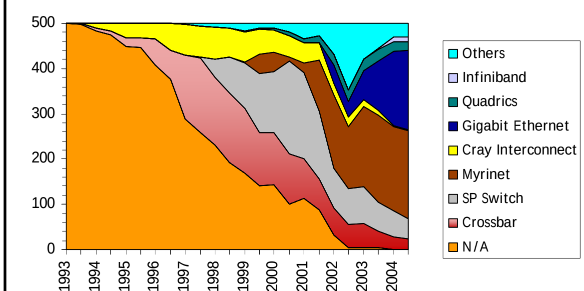
Processor Types



00

8

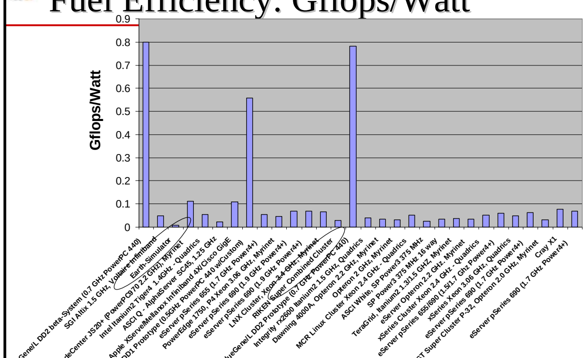
Interconnects / Systems



00

9

Fuel Efficiency: Gflops/Watt



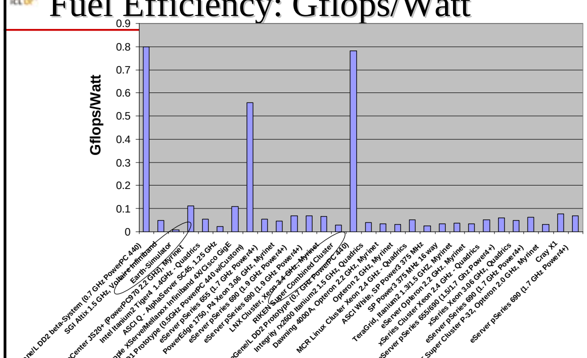
Top 20 systems

Based on processor power rating only

00

10

Fuel Efficiency: Gflops/Watt



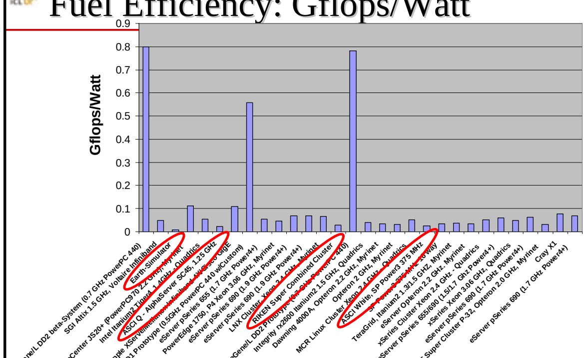
Top 20 systems

Based on processor power rating only

00

11

Fuel Efficiency: Gflops/Watt



Top 20 systems

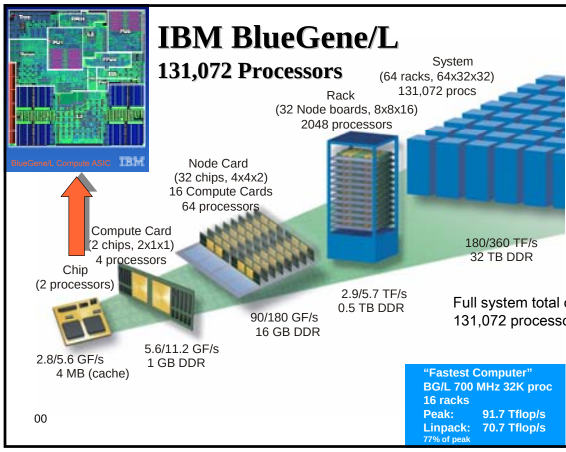
Based on processor power rating only

00

12

IBM BlueGene/L

131,072 Processors

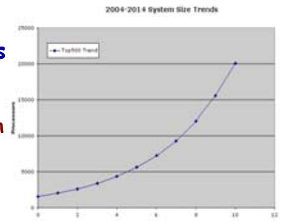


Full system total of 131,072 processors

"Fastest Computer"
BG/L 700 MHz 32K proc
16 racks
Peak: 91.7 Tflop/s
Linpack: 70.7 Tflop/s
77% of peak

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
  - 8096 processors is today's "large"
    - 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

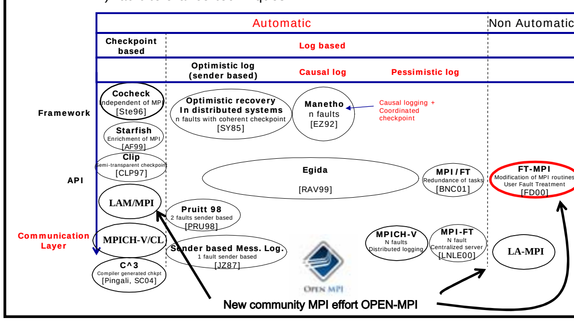
Fault Tolerance: Motivation

- 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.

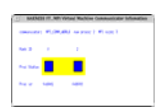


FT-MPI http://icl.cs.utk.edu/ft-mpi/

- Define the behavior of MPI in case an error occurs.
- FT-MPI based on MPI 1.3 (plus some MPI 2 features) with a fault tolerant model similar to what was done in PVM.
  - Complete reimplementation, not based on other implementations.
- Gives the application the possibility to recover from a process-failure.
- A regular, non fault-tolerant MPI program will run using FT-MPI.
- What FT-MPI does not do:
  - Recover user data (e.g. automatic check-pointing)
  - Provide transparent fault-tolerance

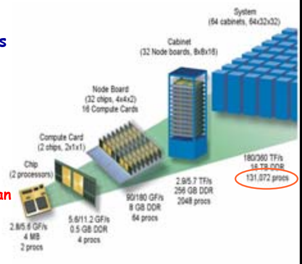
FT-MPI Failure Recovery Modes

- 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 10^5 - 10^6 hours for component.
  - sounds like a lot until you divide by 10^5!
  - 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.



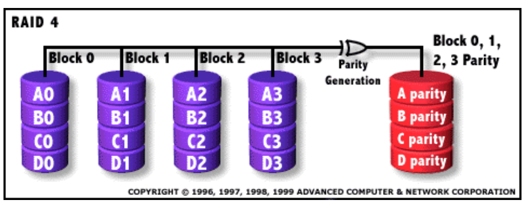
- Many cluster based on commodity parts don't have error correcting primary memory.

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.

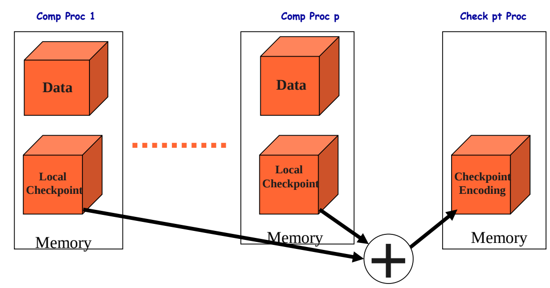
How Raid for a Disk System Works

- Similar to RAID for disks.



- If X = A XOR B then this is true:
  X XOR B = A
  A XOR X = B

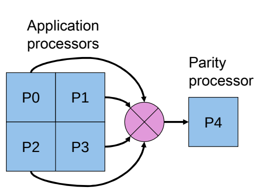
How Diskless Checkpointing Works



The encoding establishes an equality: $C_1 + C_2 + \dots C_p = C_{p+1}$.
If one of the processor failed, the above equality becomes a linear equation with only one unknown, therefore, lost data can be solved from the equation.

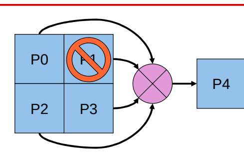
Diskless Checkpointing

- 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.



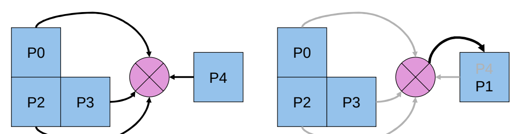
P4 = P0 ⊗ P1 ⊗ P2 ⊗ P3

Diskless Checkpointing



- When failure occurs:
  - control passes to user supplied handler
  - "XOR" performed to recover missing data
  - P4 takes on role of P1
  - Execution continue

P4 takes on the identity of P1 and the computation continues.

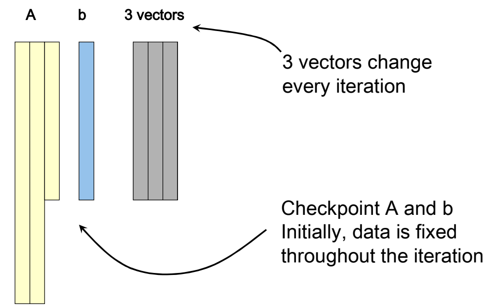


A Fault-Tolerant Parallel Conjugate Gradient Solver

- 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 ($k << p$) you need k additional processes (second example).

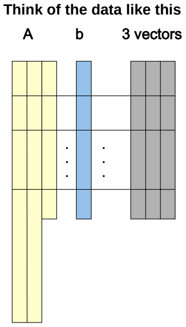
CG Data Storage

Think of the data like this

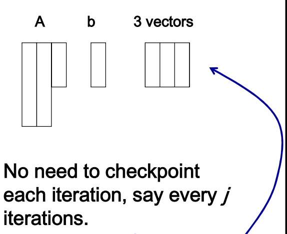


Parallel Version

Think of the data like this



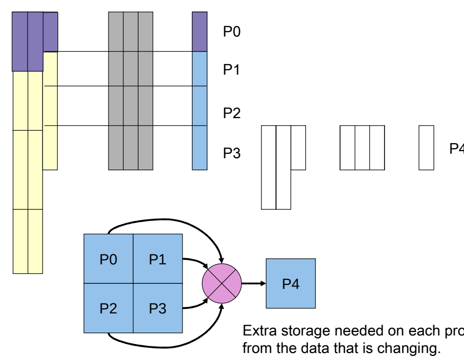
Think of the data like this on each processor



No need to checkpoint each iteration, say every j iterations.

Need a copy of the 3 vectors from checkpt in each processor.

Diskless Version



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

Actually don't do XOR, add the information.

FT PCG Algorithm Analysis

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
for $i = 1, 2, \ldots$
  solve $Mz^{(i-1)} = r^{(i-1)}$
  $\rho_{i-1} = r^{(i-1)^T} z^{(i-1)}$
  if $i = 1$
    $p^{(1)} = z^{(0)}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
  endif
  $q^{(i)} = Ap^{(i)}$
  $\alpha_i = \rho_{i-1}/p^{(i)^T} q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
  check convergence; continue if necessary
end

Global Operations

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

Global operation in Checkpoint: encoding the local checkpoint.

FT PCG Algorithm Analysis

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
for $i = 1, 2, \ldots$
  solve $Mz^{(i-1)} = r^{(i-1)}$
  $\rho_{i-1} = r^{(i-1)^T} z^{(i-1)}$
  if $i = 1$
    $p^{(1)} = z^{(0)}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
  endif
  $q^{(i)} = Ap^{(i)}$
  $\alpha_i = \rho_{i-1}/p^{(i)^T} q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
  check convergence; continue if necessary
end

Checkpoint x, r, and p every k iterations

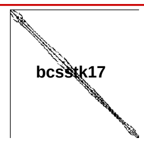
Global Operations

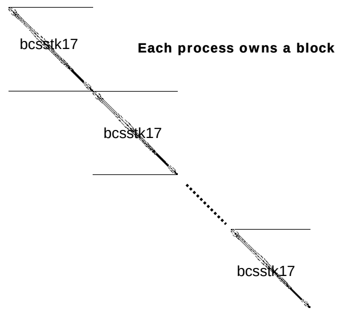
Global operation in PCG: three dot product, one preconditioning, and one matrix vector multiplication.

Global operation in Checkpoint: encoding the local checkpoint.

Global operation in checkpoint can be localized by sub-group.

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

	Size of the Problem	No. of Comp. Procs
Problem #1	164,610	15
Problem #2	329,220	30
Problem #3	658,440	60
Problem #4	1,316,880	120

All experiment are performed on:

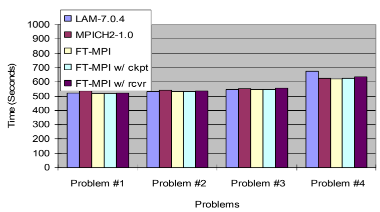
64 dual-processor 2.4 GHz AMD Opteron nodes

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

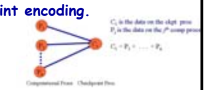


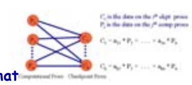
T for 20000 iters	LAM-7.0.4	MPICH2-1.0	FT-MPI	FT-MPI ckpt /2000 iters	FT-MPI exit 1 proc @10000 iters
Problem #1	522.5	536.3	517.8	518.9	521.7
Problem #2	532.9	542.9	532.2	533.3	537.5
Problem #3	545.5	553.0	546.5	547.8	554.2
Problem #4	674.3	624.4	622.9	624.4	637.1

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

Protecting for More Than One Failure: Reed-Solomon (Checkpoint Encoding Matrices)

- In order to be able to recover from any k (≤ number of checkpoint processes) failures, need a checkpoint encoding.
- With one checkpoint process we had:
  - P sets of data and a function A such that
  - $C=A*P$ where $P=(P_1,P_2,\ldots P_p)^T$;
    - C: Checkpoint data (C and P_i same size)
    - With A = (1, 1, …, 1)
    - $C = a_1P_1 + a_2P_2 + \ldots + a_p P_p$; $C = A*P$
    - To recover P_k;
      solve $P_k = (C-a_1P_1-a_{k-1}P_{k-1}-a_{k+1}P_{k+1}-a_pP_p)/a_k$





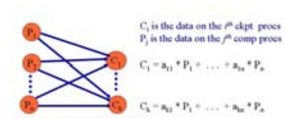
- With k checkpoints we need a function A such that
  $C=A*P$ where $P=(P_1,P_2,\ldots P_p)^T$;
    - C: Checkpoint data $C = (C_1,C_2,\ldots C_k)^T$ (C_i and P_i same size).
    - A: Checkpoint-Encoding matrix A is k x p (k << p).
- When h failures occur, recover the data by taking the h x h submatrix of A, call it A', corresponding to the failed processes and solving A'P' = C'; to recover the h "lost" P's.
  - A' is the h x h submatrix.
  - C' is made up of the surviving h checkpoints.

Reed-Solomon Approach

$A*P = C$, *where A is k x p made up of random numbers, P is p x n, C is k x n*

Here using 4 processors and 3 Ckpt processors:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{pmatrix} = \begin{pmatrix} C_1 \\ C_2 \\ C_3 \end{pmatrix}$$



Reed-Solomon Approach

$A*P = C$, *where A is k x p made up of random numbers, P is p x n, C is k x n*

Here using 4 processors and 3 Ckpt processors:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix} \begin{pmatrix} P_1 \\ \cancel{P_2} \\ \cancel{P_3} \\ P_4 \end{pmatrix} = \begin{pmatrix} C_1 \\ C_2 \\ C_3 \end{pmatrix}$$

Say 2 processors fail, P_2 and P_3.

Reed-Solomon Approach

$A*P = C$, *where A is k x p made up of random numbers, P is p x n, C is k x n*

Here using 4 processors and 3 Ckpt processors:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix} \begin{pmatrix} P_1 \\ \cancel{P_2} \\ \cancel{P_3} \\ P_4 \end{pmatrix} = \begin{pmatrix} C_1 \\ C_2 \\ C_3 \end{pmatrix}$$

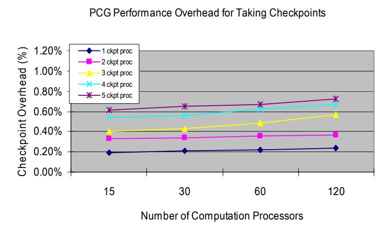
Say 2 processors fail, P_2 and P_3.

Take a subset of A's (colunm 2 and 3) and solve for P_2 and P_3.

$$\begin{pmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{pmatrix} \begin{pmatrix} P_2 \\ P_3 \end{pmatrix} = \begin{pmatrix} C_1 \\ C_2 \end{pmatrix}$$

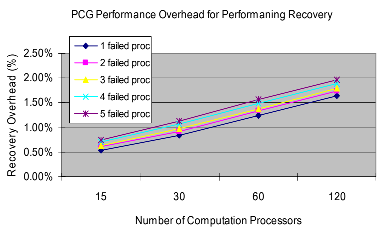
Could use GF(2). Signal processing aps do this. In that case, A is Vandermonde or Cauchy matrix. (Need to have any subset of A be non singular.) We use A as a random matrix.

PCG: Performance Overhead of Taking Checkpoints



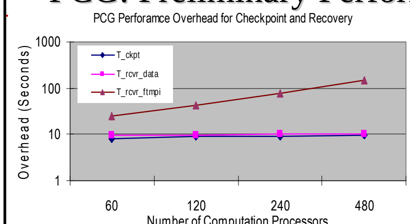
T (ckpt T)	0 ckpt	1 ckpt	2 ckpt	3 ckpt	4 ckpt	5 ckpt
15 comp	517.8	518.9 (1.0)	519.6 (1.7)	519.8 (2.1)	520.4 (2.8)	521.0 (3.2)
30 comp	532.2	533.3 (1.1)	533.7 (1.8)	534.5 (2.3)	535.1 (3.0)	535.6 (3.5)
60 comp	546.5	547.8 (1.2)	548.0 (2.0)	548.8 (2.7)	549.7 (3.2)	550.1 (3.7)
120 comp	622.9	624.4 (1.5)	625.5 (2.3)	626.7 (3.6)	627.5 (4.2)	628.6 (4.5)

PCG: Performance Overhead of Performing Recovery



T (ckpt T)	0 proc	1 proc	2 proc	3 proc	4 proc	5 proc
15 comp	517.8	521.7 (2.8)	522.1 (3.2)	522.8 (3.3)	522.9 (3.7)	523.1 (3.9)
30 comp	532.2	537.5 (4.5)	537.7 (4.9)	538.1 (5.3)	538.5 (5.7)	538.6 (6.1)
60 comp	546.5	554.2 (6.9)	554.8 (7.4)	555.2 (7.6)	555.7 (8.2)	556.1 (8.7)
120 comp	622.9	637.1 (10.5)	637.2 (11.1)	637.7 (11.5)	638.0 (12.0)	638.5 (12.5)

PCG: Preliminary Performance



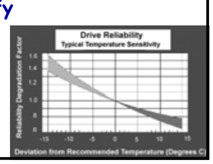
IBM RS/6000 SP w/176 Winterhawk II thin nodes (each with four 375 MHz Power3-II processors)

Run PCG for 5000 iterations and take checkpoint every 1000 iterations (about 5 minutes)
Simulate a failure of one node by exiting 4 processes at the 3000-th iteration.
Matrix size scales with the processors used, i.e. 60 procs: n=658,440; 480 procs: n=5.2M

Time (Sec)	T_pcg_comp	T_ckpt	T_rcvr_data	T_rcvr_ftmpi	T_tot
60 procs	1399.1	8.0	9.8	24.8	1441.7
120 procs	1429.3	9.2	9.9	42.1	1490.5
240 procs	1461.1	9.2	10.0	77.2	1557.5
480 procs	1531.1	9.7	10.1	146.1	1697.0

Predictive Adaptive Fault Tolerence

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



Next Steps

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 checkpoint.
- Generalize the ideas to provide a library of routines to do the diskless check pointing.
- 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
- 12 years ➔ we see a factor of ~ 8125
- 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 that 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

Complex set of interaction between
Users' applications
Algorithm
Programming language
Compiler
Machine instruction
Hardware
Many layers of translation from the application to the hardware
Changing with each generation

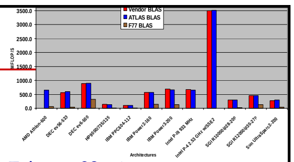
00 43

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 redone 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.

00 44

Software Generation Strategy - ATLAS BLAS

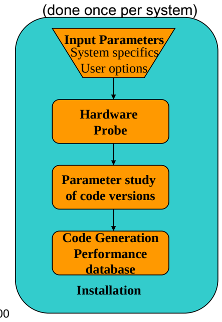


- Parameter study of the hw
- Generate multiple versions of code, w/difference values of key performance parameters
- Run and measure the performance for various versions
- Pick best and generate library
- Level 1 cache multiply optimizes for:
  - TLB access
  - L1 cache reuse
  - FP unit usage
  - Memory fetch
  - Register reuse
  - Loop overhead minimization
- Similar to FFTW and Johnsson, UH
- Takes ~ 20 minutes to run, generates Level 1,2, & 3 BLAS
- "New" model of high performance programming where critical code is machine generated using parameter optimization.
- Designed for modern architectures
  - Need reasonable C compiler
- Today ATLAS in used within various ASCI and SciDAC activities and by Matlab, Mathematica, Octave, Maple, Debian, Scyld Beowulf, SuSE,…

00 See: http://icl.cs.utk.edu/atlas/ joint with Clint Whaley & Antoine Petitet 45

Performance Tuning Methodology

Software Installation
(done once per system)



Software Generation
Strategy - ATLAS BLAS
http://www.netlib.org/atlas/

- Parameter study of the hw
- Generate multiple versions of code, w/difference values of key performance parameters
- Run and measure the performance for various versions
- Pick best and generate library
- Optimize over 8 parameters
  - Cache blocking
  - Register blocking (2)
  - FP unit latency
  - Memory fetch
  - Interleaving loads & computation
  - Loop unrolling
  - Loop overhead minimization
- Similar to FFTW

00 46

Self Adapting Numerical Software - SANS Effort

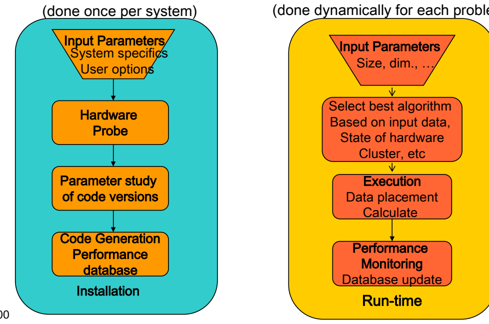
- Provide software technology to aid in high performance on commodity processors, clusters, and grids.
- Pre-run time (library building stage) and run time optimization.
- Integrated performance modeling and analysis
- Automatic algorithm selection – polyalgorithmic functions
- Automated installation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

00 47

Performance Tuning Methodology

Software Installation
(done once per system)

Software Execution
(done dynamically for each problem)



00 48

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 1000 to 10000 to 100000 of processors!!
    - multidisciplinary 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.

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 Pjesivac-Grbovic, UTK

LACSI

SciDAC