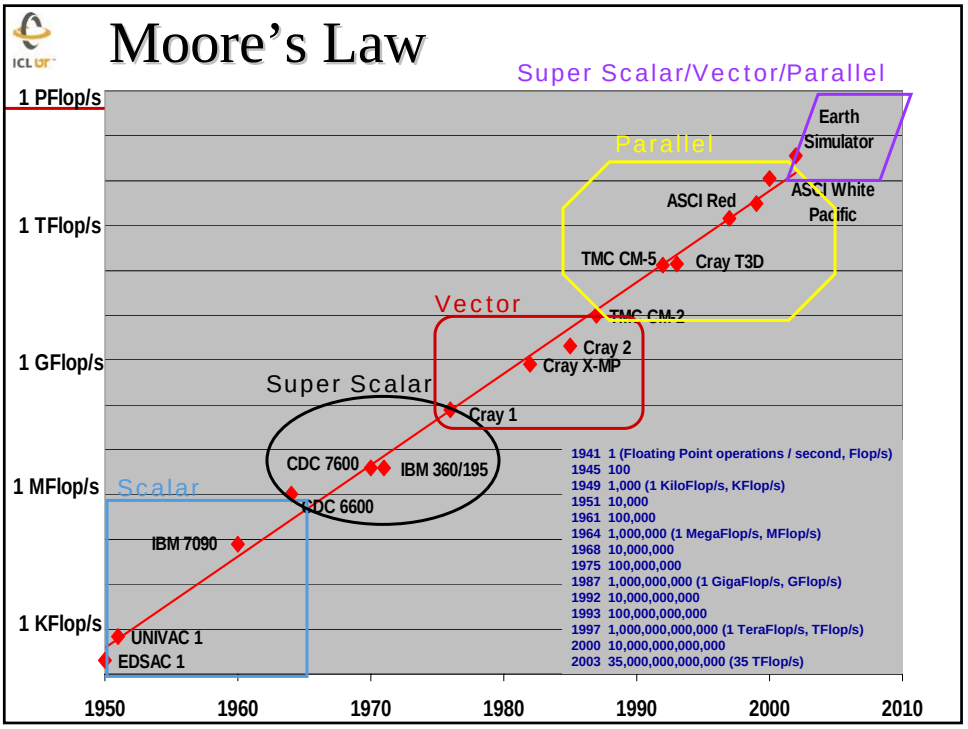
Self-Adapting Numerical Software (SANS-Effort)

Jack Dongarra,
Innovative Computing Laboratory
University of Tennessee
and
Computer Science and Mathematics Division
Oak Ridge National Laboratory

Moore's Law



Linpack (100x100) Analysis

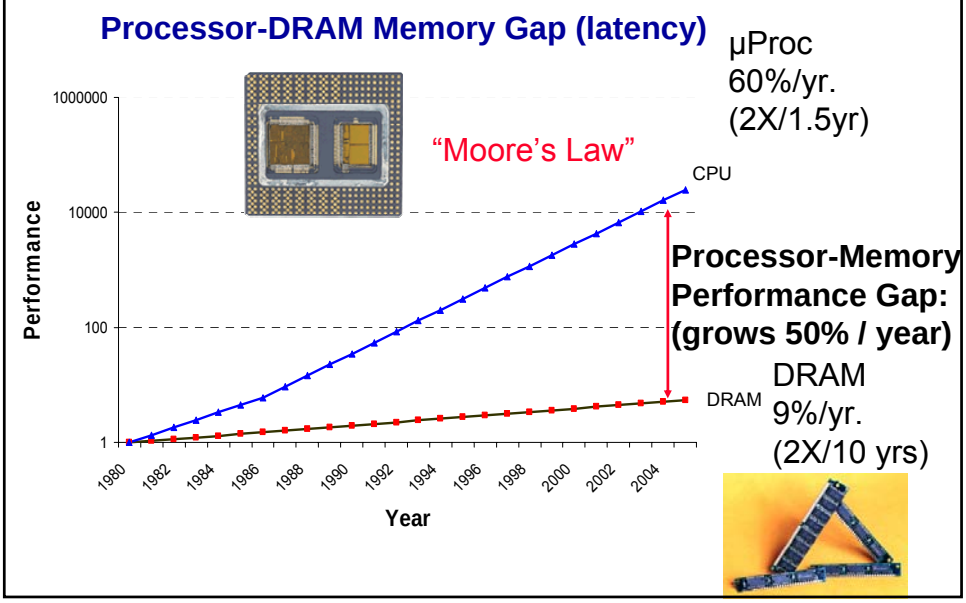
- Compaq 386/SX20 SX with FPA - .16 Mflop/s
- Pentium IV – 2.8 GHz – 1.32 Gflop/s
- 12 years ➔ we see a factor of ~ 8231
- Moore's Law says something about a factor of 2 every 18 months or a factor of 256 over 12 years

- Where is the missing 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 getting 1.32 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

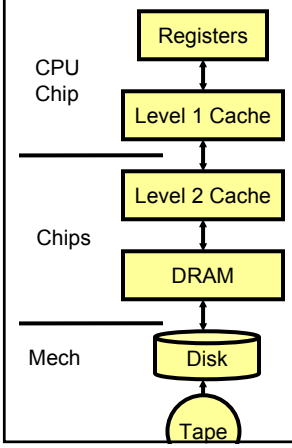
Many layers of translation from the application to the hardware
Changing with each generation

Where Does the Performance Go? or Why Should I Care About the Memory Hierarchy?



The Memory Hierarchy

- By taking advantage of the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.



1cy	3-10 words/cycle	compiler managed	
1-3cy	1-2 words/cycle	hardware managed	
5-10cy	1 word/cycle	hardware managed	
30-100cy	0.5 words/cycle	OS managed	
10^6-10^7cy	0.01 words/cycle	OS managed	

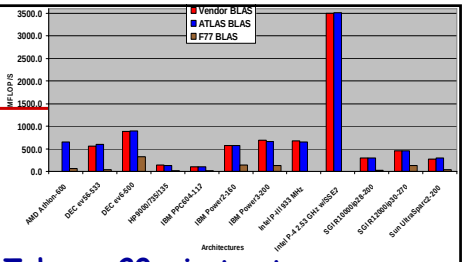
Motivation Self Adapting Numerical Software (SANS) Effort

- 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.

What is Self Adapting Performance Tuning of Software?

- Two steps:
  1. Identify and generate a space of algorithm/software, with various based on the architectural features
     - Instruction mixes and orders
     - Memory Access Patterns
     - Data structures
     - Mathematical Formulations
  2. Generate different versions and search for the fastest one, by running them
- When do we search?
  - Once per kernel and architecture
  - At compile time
  - At run time
  - All of the above
- Many examples
  - PHiPAC, ATLAS, Sparsity, FFTW, Spiral,…

Software Generation Strategy - ATLAS BLAS

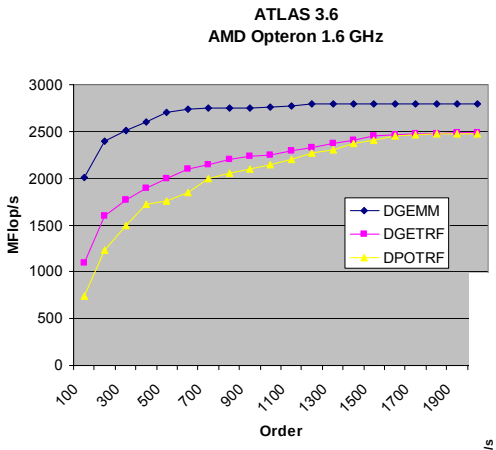


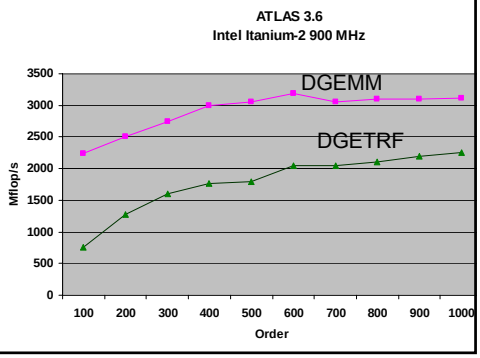
- Parameter study of the hw
- Generate multiple versions of code, w/difference values of key performance parameters
- Run and measure the performance for various versions
- Pick best and generate library
- Level 1 cache multiply optimizes for:
  - TLB access
  - L1 cache reuse
  - FP unit usage
  - Memory 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,…

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

ATLAS 3.6 (new release)





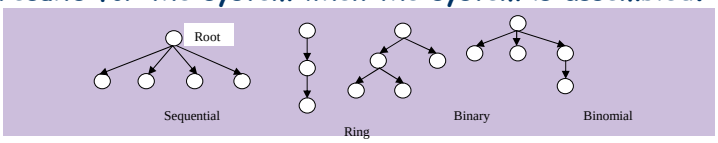
http://www.netlib.org/atlas/

Self Adapting Numerical Software - SANS Effort

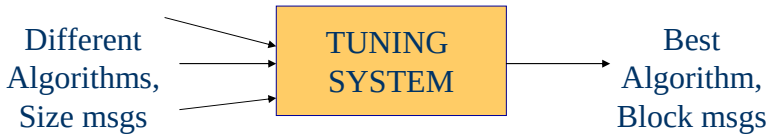
- Provide software technology to aid in high performance on commodity processors, clusters, and grids.
- Pre-run time (library building stage) and run time optimization.
- Integrated performance modeling and analysis
- Automatic algorithm selection – polyalgorithmic functions
- Automated installation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

Self Adapting for Message Passing

- **Communication libraries**
  - Optimize for the specifics of one's configuration.
  - A specific MPI collective communication algorithm implementation may not give best results on all platforms.
  - Choose collective communication parameters that give best results for the system when the system is assembled.



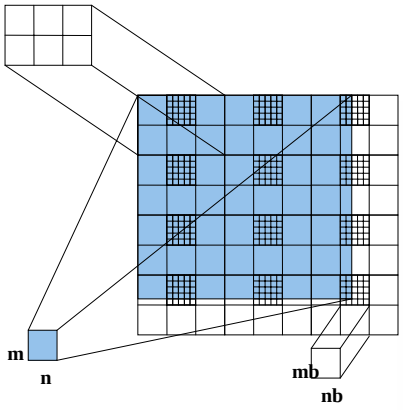
- **Algorithm layout and implementation**
  - Look at the different ways to express implementation



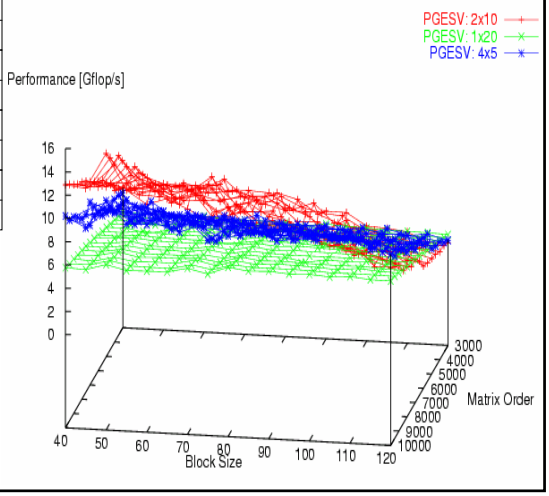
Self Adaptive Software

- **Software can adapt its workings to the environment in (at least) 3 ways**
  - Kernels, optimized for platform (Atlas, Sparsity): static determination
  - Scheduling, taking network conditions into account (LFC): dynamic, but data-independent
  - Algorithm choice (Salsa): dynamic, strongly dependent on user data.

Data Layout Critical for Performance

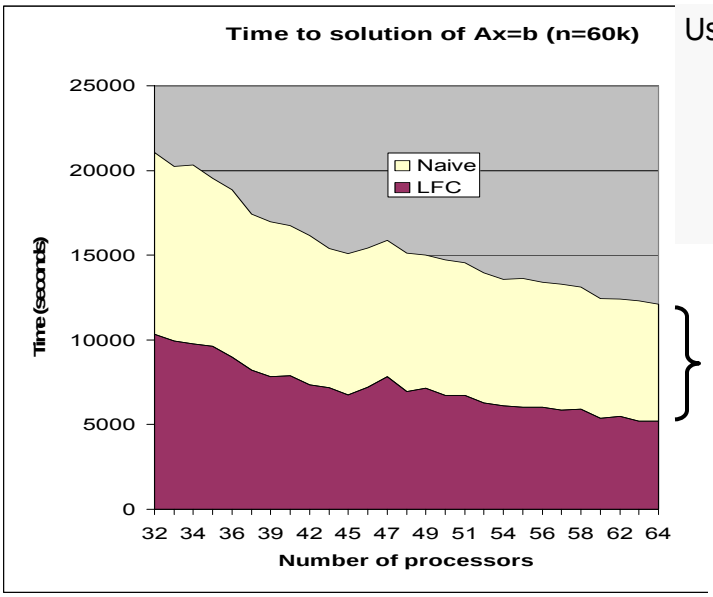


Number of processors
Aspect ratio of processes
Block size



Needs An "Expert" To Do The Tuning

LFC Performance Results

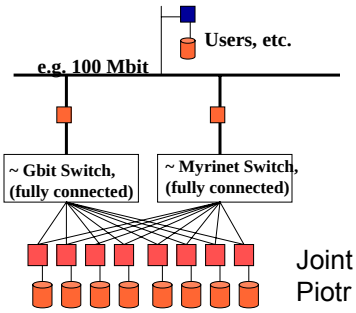


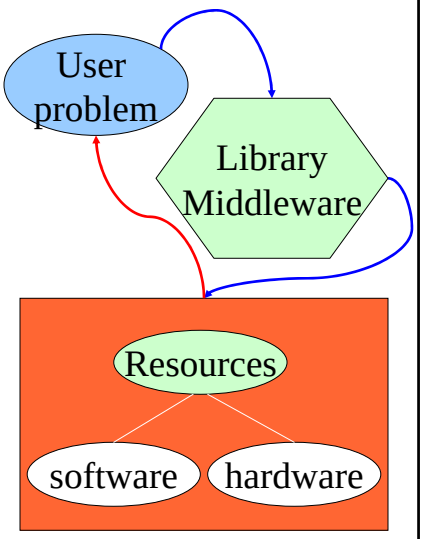
Using up to 64 of AMD 1.4 GHz processors at Ohio Supercomputer Center

Increasing margin of potential user error

LFC: LAPACK For Clusters

- Want to relieve the user of some of the tasks via Cluster Middleware
- Make decisions on the number of processors to use based on the user's problem and the state of the system
  - Optimize for the best time to solution
  - Distribute the data on the processors and collections of results
  - Start the SPMD library routine on all the platforms



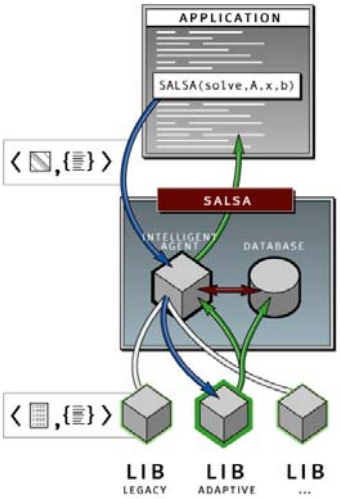


Joint with
Piotr Łuszczek & Kenny Roche
http://icl.cs.utk.edu/lfc/

SALSA: Self-Adaptive Linear Solver Architecture

Run-time adaptation to user data for linear system solving

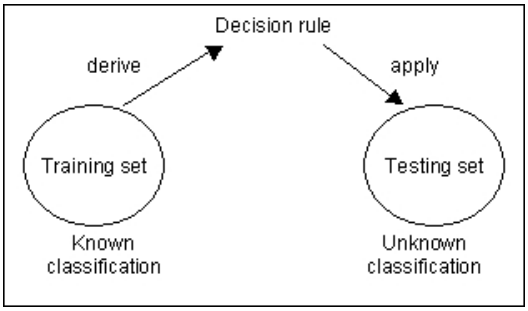
- Choice between direct/iterative solver
  - Space and runtime considerations
  - Numerical properties of system
- Choice of preconditioner, scaling, ordering, decomposition
- User steering of decision process
- Insertion of performance data in database
- Metadata on both numerical data and algorithms
- Heuristics-driven automated analysis
- Self-adaptivity: tuning of heuristics over time through experience gained from production runs



Joint work with
Victor Eijkhout, Bill Gropp, & David Keyes

Finding Heuristics By Statistical Pattern Recognition

- Use a training set to arrive at a *Decision Rule* and *Features* on which to base it.



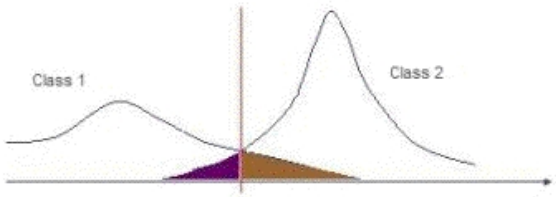
- Pick the method for best chance of converging based on the properties of this matrix.
- The training process gathers the data to construct these density functions (probability of converging with these features).

Statistical Approach for Numerical Algorithms

- The strategy in determining numerical algorithms by the Bayesian statistical technique is globally as follows:
  1. We solve a large collection of test problems by every available method, that is, every choice of algorithm, and a suitable `binning' of algorithm parameters.
  2. Each problem is assigned to a class corresponding to the method that gave the fastest solution.
  3. Draw up a list of characteristics of each problem.
  4. Compute a probability density function for each class.
- As a result of this process we find a function $p_i(x)$ where i ranges over all classes, that is, all methods, and x is in the space of the vectors of features of the input problems.
- Given a new problem and its feature vector x, we then decide to solve the problem with the method i for which $p_i(x)$ is maximized

Statistical Pattern Recognition

- Build probability density function for each method
- Use Maximum Likelihood rule to predict best method for the test set

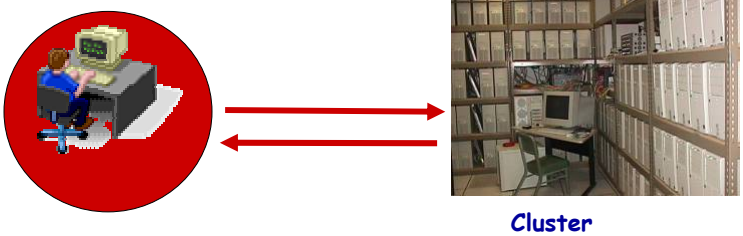


- Classes correspond to different methods
- density function states how likely a feature set is successfully solved by that method.
  - Shape of the spectrum: ratio of the x/y size of the enclosing ellipse, and ratio of positive to negative eigenvalues.
  - Element variability in rows and columns (ratio between smallest and largest element).
- For a value of the feature for a matrix, this is how likely that this method is the best

CONIE – Cluster Oriented Numerical Intensive Execution (Executing Matlab Programs on a Cluster)

> mpirun -np 128 lfc_server port=35000 &

> Matlab



Cluster

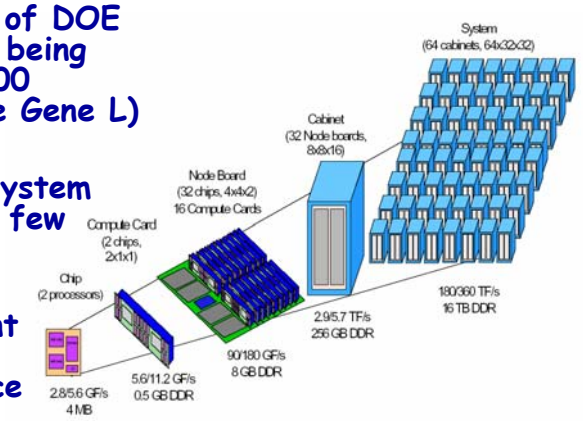
```
server_connect(35000);
A = lfc_fread(…);
b = lfc_fread(…);
x = A \ b;  % copy A; save factors
r = b – A * x;
z = A \ r; % use factors from above
x = x + z;
norm(b-A*x)/(norm(A)*norm(x))
    % results printed on laptop
```

- **Arrays will live on the server and execution takes place there via LFC / ScaLAPACK / SALSA.**
- **Debug on laptop, run on cluster**

Plans for Python, Mathematica, Maple … as well

Fault Tolerance in the Computation

- The next generation of DOE ASCI computers are being designed with 131,000 processors (IBM Blue Gene L)
- Failures for such a system is likely to be just a few minutes away.
- Application checkpoint /restart is today's typical fault tolerance method.
- Problem with MPI, no recovery from faults in the standard



MPI Implementations with Fault Tolerance

	Automatic			Semi-automatic
	Checkpoint based	Log based		Other
		Optimistic	Casual / Pessimistic	
Framework	CoCheck Starfish	Manetho		
API	Clip	Egida	MPI/FT	FT-MPI
Comms layer	LAM/MPI MPICH-V/CL	Pruitt98 Send based Mesg. logging	MPI-FT MPICH-V2	LA-MPI

Algorithm Based Fault Tolerance Using Diskless Check Pointing

- Not "automagic", recovery has to be built into the algorithm
- N processors will be executing the computation.
  - Each processor maintains their own checkpoint locally
- M (M << N) extra processors maintain coding information so that if 1 or more processors die, they can be replaced
- Look at M = 1 (parity processor)

- FT-MPI based on MPI 1.3 but with Fault Tolerance available to the programmer.
  - Similar to what was done in PVM.
  - http://icl.cs.utk.edu/ft-mpi/

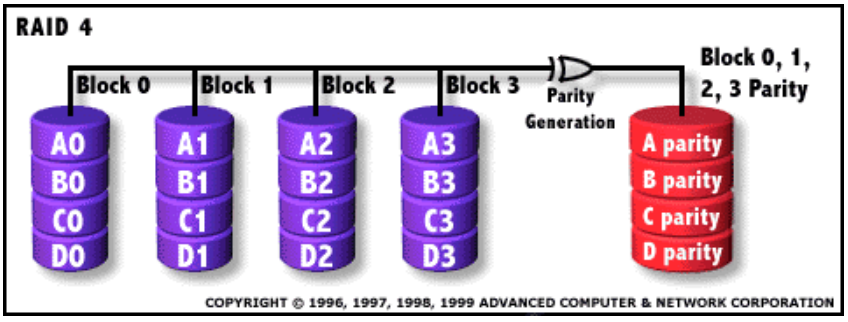
Fault Tolerance - Diskless (RAID) Checkpointing - Built into software

(J. Plank, Y. Kim, J. Dongarra)

- Maintain a system checkpoint in memory
  - All processors may be roll back if necessary
  - Use m extra processors to encode checkpoints so that if up to m processors fail, their checkpoints may be restored
  - No reliance on disk
- Checksum and reverse communication
  - Checkpoint less frequently
  - Reverse the computation of the non-failed processors back to previous checkpoint
- Idea to build into library routines
  - System or user can dial it up
  - Working prototype for MM, LU, LL^T, QR, sparse solvers (built on PVM)

How Diskless Check Pointing Works

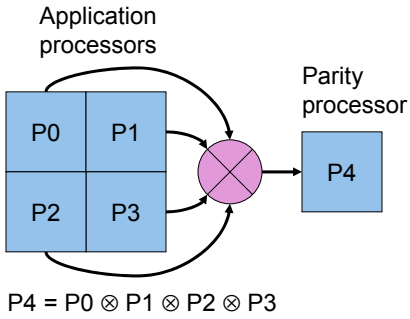
- Similar to RAID for disks.



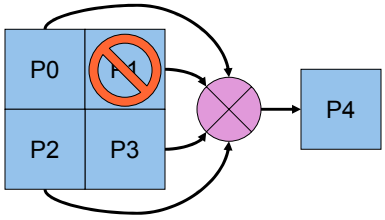
- If X = A XOR B then this is true:
  X XOR B = A
  A XOR X = B

Diskless Checkpointing

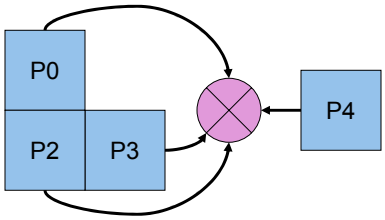
- The N application processors (4 in this case) each maintain their own checkpoints locally.
- M extra processors maintain coding information so that if 1 or more processors die, they can be replaced.
- Will describe for m=1 (parity)
- If a single processor fails, then its state may be restored from the remaining live processors



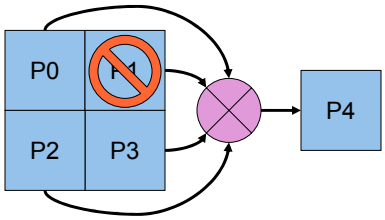
Diskless Checkpointing



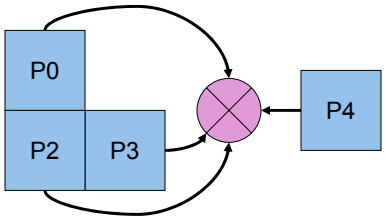
P1 = P0 ⊗ P2 ⊗ P3 ⊗ P4

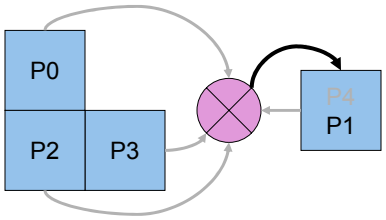


Diskless Checkpointing



P4 takes on the identity of P1
and the computation continues





A Fault-Tolerant Parallel CG Solver

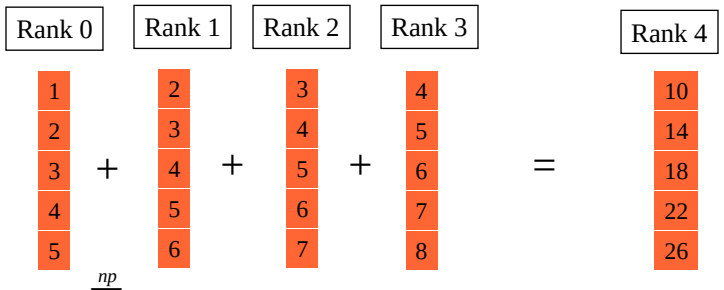
- Tightly coupled computation
- Do a "backup" (checkpoint) every *k* iterations

- Can survive the failure of a single process
- Dedicate an additional process for holding data, which can be used during the recovery operation
- Work-communicator excludes the backup process

- For surviving *m* process failures (*m* < *np*) you need *m* additional processes

The Checkpoint Procedure

- 4 processes participating in the computation, one for checkpointing and recovery
- If your application can survive one process failure at a time

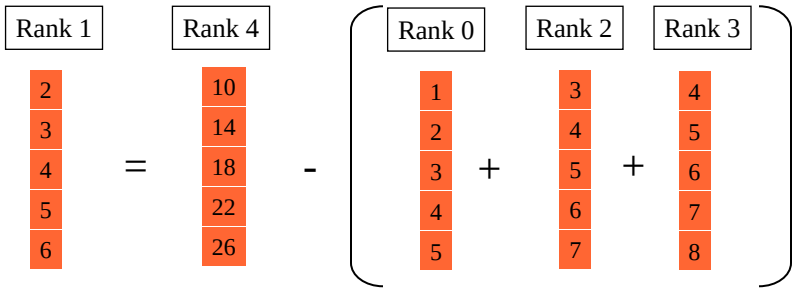


or $b_i = \sum_{j=1}^{np} v_i(j)$

- Implementation: a single reduce operation for a vector
- Keep a copy of the vector *v* which you used for the backup

The Recovery Procedure

- Rebuild work-communicator and Recover data
- Say lose process w/rank 1, checkpoint in process 4, then use remain processes 0, 2, and 3 along with checkpoint in 4 to recover data from process 1.



- Reset iteration counter
- On each process: copy backup of vector *v* into the current version

Preconditioned Conjugate Grad Performance

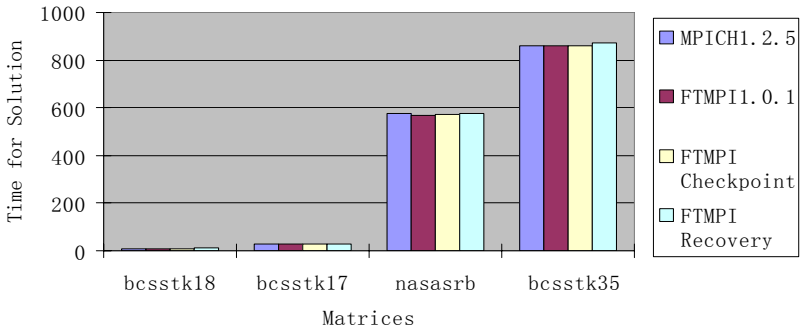


Table 1: PCG performance on 25 nodes of the Boba cluster at UTK. 24 nodes are used for computation. 1 node is used for checkpoint Checkpoint every 100 iterations, with diagonal preconditioning

Matrix (Size)	Mpich1.2.5 (sec)	FT-MPI (sec)	FT-MPI w/ ckpoint (sec)	FT-MPI w/ recovery (sec)	Recovery (sec)	Ckpoint Ohead (%)	Recovery Overhead (%)
bcsstk18.rsa (11948)	9.81	9.78	10.0	12.9	2.31	2.4	23.7
bcsstk17.rsa (10974)	27.5	27.2	27.5	30.5	2.48	1.1	9.1
nasasrb.rsa (54870)	577.	569.	570.	577.	4.09	0.23	0.72
bcsstk35.rsa (30237)	860.	858.	859.	872.	3.17	0.12	0.37

Futures for Numerical Algorithms and Software

- Numerical software will be adaptive, exploratory, and intelligent
- Determinism in numerical computing will be gone.
  - After all, its not reasonable to ask for exactness in numerical computations.
  - Auditability of the computation, reproducibility at a cost
- Importance of floating point arithmetic will be undiminished.
  - 16, 32, 64, 128 bits and beyond.
- Fault tolerance a critical feature of future software and hardware systems
- Adaptivity is a key so applications can effectively use the resources.

Collaborators / Support

- LFC/SALSA/BeBOP
  - Victor Eijkhout, UTK
  - Erika Fuentes, UTK
  - Kenny Roche, UTK
  - Piotr Luszczek, UTK
  - David Keyes, CU
  - Bill Gropp, ANL
  - Jim Demmel, UCB
  - Kathy Yelick, UCB
- Python/Matlab Clusters
  - Piotr Luszczek, UTK

LACSI SciDAC Scientific Discovery through Advanced Computing DARPA

- For more information: