Fault Tolerance,
Performance API,
and
MultiCore Optimization

Jack Dongarra
dongarra@cs.utk.edu
Innovative Computer Laboratory
University of Tennessee

http://lacsi.rice.edu/review/slides_2006

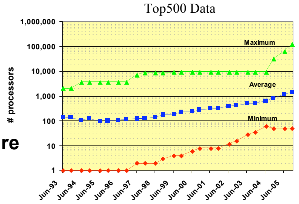
Participants:
Shirley Moore, Graham Fagg, George Bosilca, ICL Staff
Jeffery Chen, Jelena Pjesivac-Grbovic, Haihang You, Graduate Student

Outline

- Two parts with focus on tools and applications.
- Looking (near term and longer term) at:
  - —Open-MPI and using it in applications
  - —Parallel Tools effort with Eclipse / PAPI work
  - —MultiCore algorithms for future systems
- Focus on using enabling technology in applications to demonstrate effectiveness and usability in LANL applications.

Fault Tolerance: Motivation

- Trends in HPC:
  - —High end systems with thousand of processors
  - —Move to multicore chips

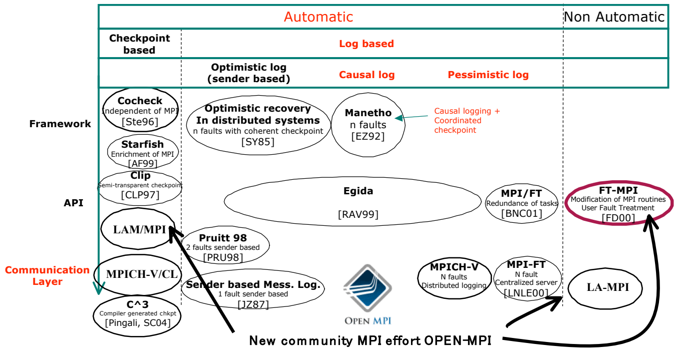


- Increased probability of a node failure
  - —Most systems nowadays are robust
- MPI widely accepted in scientific computing
  - —Process faults not tolerated in MPI standard

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.

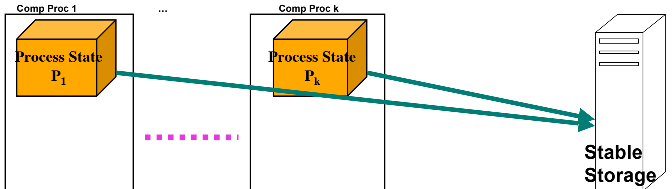


New community MPI effort OPEN-MPI
Joint with LANL, Indiana U, U Tennessee,and INRIA

Open-MPI Approach for Dealing with Faults

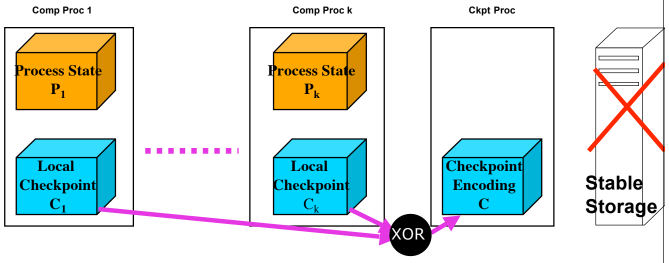
- **Application checkpointing, MP API+Fault management, automatic.**
  - Application ckpt: application store intermediate results and restart form them
  - MP API+FM: message passing API returns errors to be handled by the programmer
  - Automatic: runtime detects faults and handle recovery
- **Checkpoint coordination: no, coordinated, uncoordinated.**
  - Coordinated: all processes are synchronized, network is flushed before ckpt;
    - all processes rollback from the same snapshot
  - Uncoordinated: each process checkpoint independently of the others
    - each process is restarted independently of the other
- **Message logging: no, pessimistic, optimistic, causal.**
  - Pessimistic: all messages are logged on reliable media and used for replay
  - Optimistic: all messages are logged on non reliable media. If 1 node fails, replay is done according to other nodes logs. If >1 node fail, rollback to last coherent checkpoint
  - Causal: optimistic+Antecedence Graph, reduces the recovery time

Checkpoint/Restart



- **Checkpoint/restart is today's typical fault tolerance approach in HPC**
  - —Periodically write process states into ***stable-storage***
  - —If one process fails, ***abort all processes***
  - —Good to tolerate the failure of the whole system
  - —But the overhead is high : T = # of procs * size_ckpt / bandwidth
- **Today's architectures are usually robust enough to survive partial failures without suffering the failure of the whole system**
  - —Can we tolerate partial failures with less overhead (and higher scalability) than checkpoint/restart ?

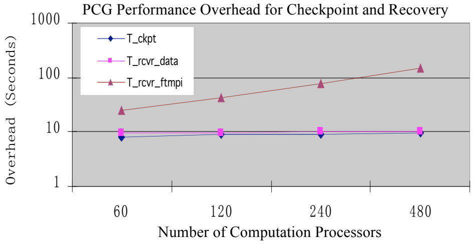
Diskless Checkpointing
(J. S. Plank, et. el.)



- Each computational processor saves a copy of its state locally in memory
- Dedicate an additional processor to save the encoding of these states
- The checkpoint overhead is (binary tree encoding):

T = log (# of procs) *size_ckpt / bandwidth + log (# of procs)*latency

PCG: Performance



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

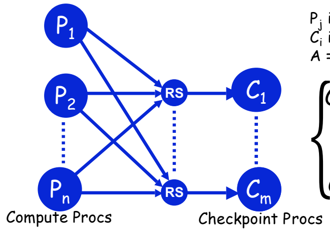
Run PCG for 5000 iterations and take checkpoint every 1000 iterations (about 5 minutes)

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)	Time w/o checkpoint	Checkpoint time	Data Recovery time	System Recovery time	Total time to recover from fault
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

Coding to Survive Multiple Failures: Basic Scheme (Reed-Solomon Encoding)



P_j is the checkpoint data on the j^{th} comp procs
C_i is the encoded data on the i^{th} ckpt procs
$A = (a_{ij})_{m \times n}$ is a encoding matrix

$$\begin{cases} C_1 = a_{11} * P_1 + \ ... \ + a_{1n} * P_n \\ \vdots \\ C_m = a_{m1} * P_1 + \ ... \ + a_{mn} * P_n \end{cases}$$

Key idea: establish *m* equalities by *m* encodings

If there are *k* (<= *m*) processes failed, then the *m* equalities become

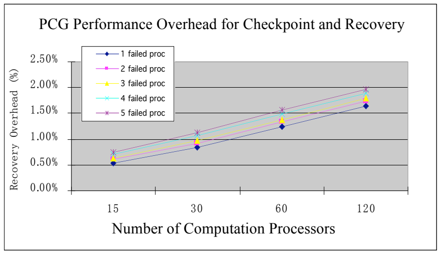
m equations with *k* unknowns

By appropriately choosing A, the lost data can be recovered by solving the *m* equations.

The checkpoint overhead (assume pipelined encoding):

T = m * { (1 + O(1/size_ckpt^0.5)) * size_ckpt / bandwidth + #of procs * latency }

PCG: Performance Overhead of Recovery



Run PCG for 20000 iterations and take checkpoint every 2000 iterations
Simulate a failure by exiting some processes at the 10000-th iteration

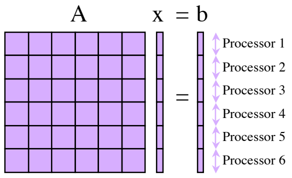
T (ckpt T)	0 failures	1 failures	2 failures	3 failures	4 failures	5 failures
15 comp	517.8	521.7 (2.8)	522.1 (3.2)	522.8 (3.3)	522.9 (3.7)	523.1 (3.9)
30 comp	532.2	537.5 (4.5)	537.7 (4.9)	538.1 (5.3)	538.5 (5.7)	538.6 (6.1)
60 comp	546.5	554.2 (6.9)	554.8 (7.4)	555.2 (7.6)	555.7 (8.2)	556.1 (8.7)
120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

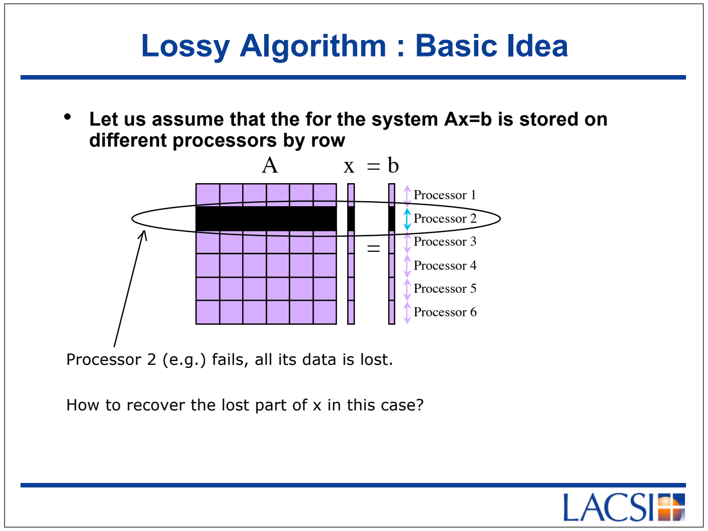
Second Approach

- **Lossy approach for iterative methods**
  - **Here there is only a checkpoint of the primary data**
    - **Continuous checkpointing is not done during the iteration.**
  - **When the failure occurs we will approximate the missing data and continue**
    - **No guarantee here; may or may not work**

Lossy Algorithm : Basic Idea

- **Let us assume that the data for the system Ax=b is stored on different processors by rows and original data, A and b, can be retrieved.**





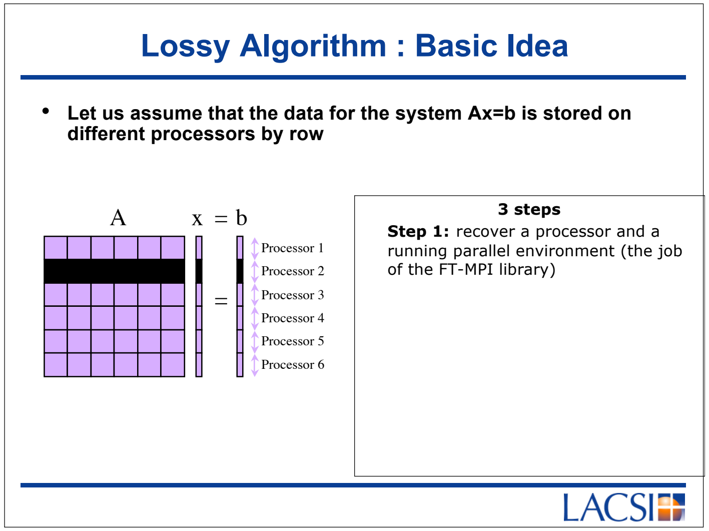
Lossy Algorithm : Basic Idea

- **Let us assume that the for the system Ax=b is stored on different processors by row**

A
x = b
Processor 1
Processor 2
Processor 3
Processor 4
Processor 5
Processor 6

Processor 2 (e.g.) fails, all its data is lost.

How to recover the lost part of x in this case?



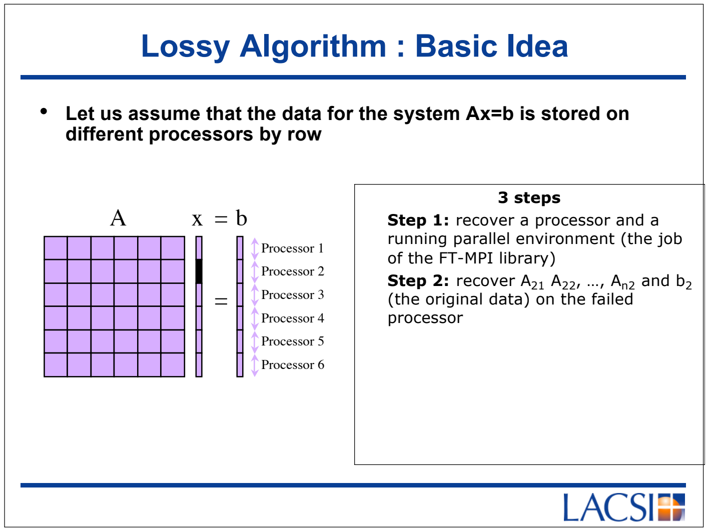
Lossy Algorithm : Basic Idea

- **Let us assume that the data for the system Ax=b is stored on different processors by row**

A
x = b
Processor 1
Processor 2
Processor 3
Processor 4
Processor 5
Processor 6

3 steps

Step 1: recover a processor and a running parallel environment (the job of the FT-MPI library)



Lossy Algorithm : Basic Idea

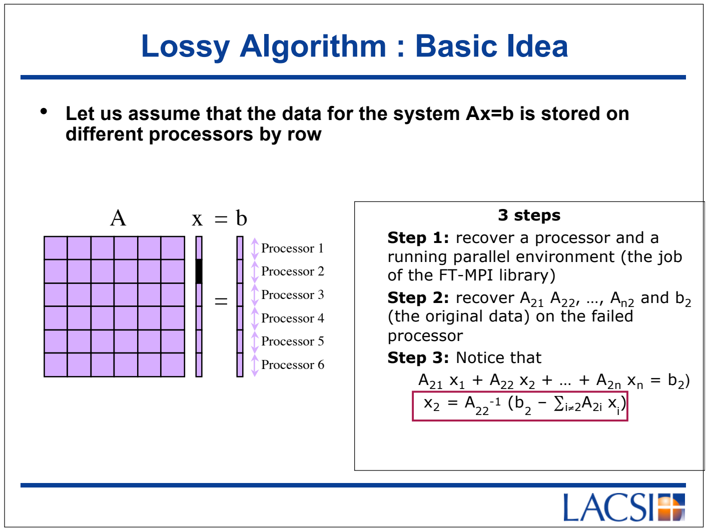
- **Let us assume that the data for the system Ax=b is stored on different processors by row**

A
x = b
Processor 1
Processor 2
Processor 3
Processor 4
Processor 5
Processor 6

3 steps

Step 1: recover a processor and a running parallel environment (the job of the FT-MPI library)

Step 2: recover A_{21} A_{22}, …, A_{n2} and b_2 (the original data) on the failed processor



Lossy Algorithm : Basic Idea

- **Let us assume that the data for the system Ax=b is stored on different processors by row**

A
x = b
Processor 1
Processor 2
Processor 3
Processor 4
Processor 5
Processor 6

3 steps

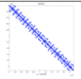
Step 1: recover a processor and a running parallel environment (the job of the FT-MPI library)

Step 2: recover A_{21} A_{22}, …, A_{n2} and b_2 (the original data) on the failed processor

Step 3: Notice that

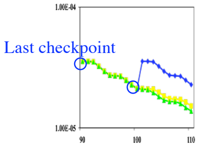
$A_{21} x_1 + A_{22} x_2 + \dots + A_{2n} x_n = b_2)$

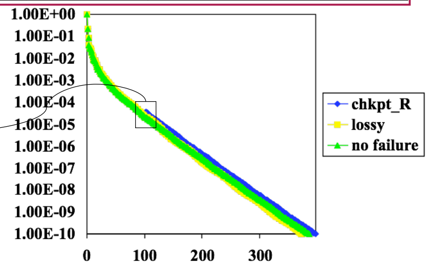
$\boxed{x_2 = A_{22}^{-1} (b_2 - \sum_{i\neq 2} A_{2i} x_i)}$



Using GMRES(30) Non Symmetric Matrix

stomach; n=213,360; nnz=3,021,648; tol=10^{-10}; #procs=16; n_f=13,335; nnz=185,541									
recovery	$iter_f$	#iter	T_{Wall}	T_{Chkpt}	T_{Roll}	T_{Recov}	T_I	$T_{II,a,b}$	T_{III}
lossy	no	385	38.89						
$chkpt_R$	no	385	41.04	1.92					
lossy	100	372	42.38		1.56	5.38	1.03	0.33	3.91
$chkpt_R$	100	395	45.49	1.92	2.40	1.68	1.02	0.32	0.20



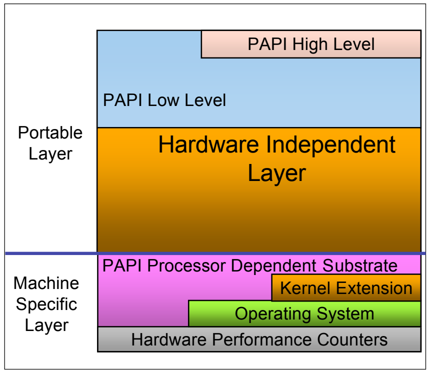


Time are given in seconds
Intel Xeon at 2.40 GHz with
Myrinet interconnect

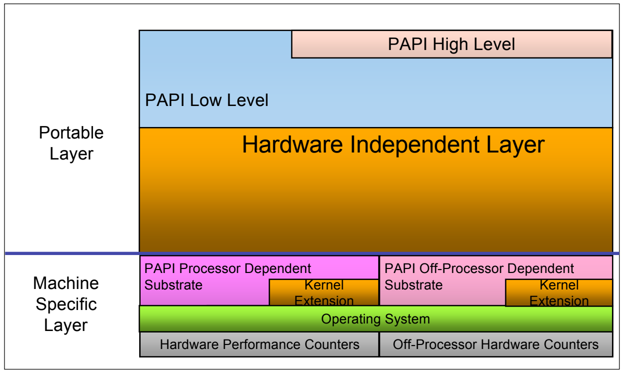
Extending PAPI Beyond the Processor

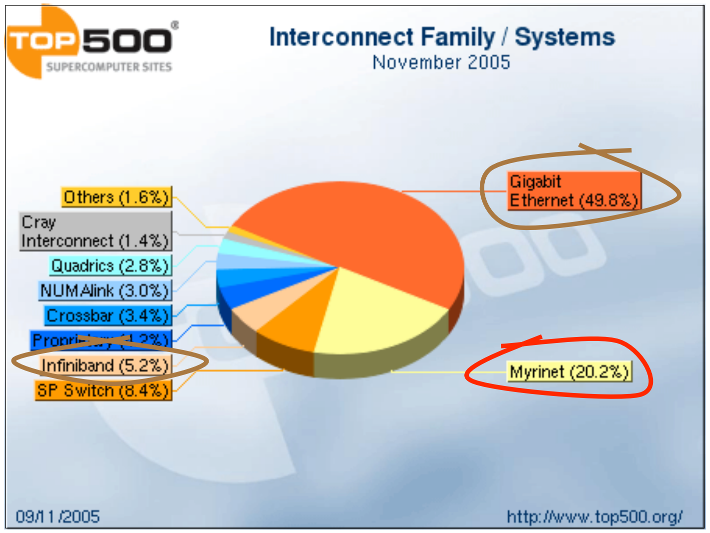
- PAPI is software layer that aims to provide the tool designer and application engineer with a consistent interface and methodology for use of the performance counter hardware found in most major microprocessors.
- PAPI has historically targeted on on-processor performance counters
  - —Ops, cycles, memory traffic,
- Several categories of off-processor counters exist
  - —network interfaces: Myrinet, Infiniband, GigE
  - —memory interfaces: Cray X1
  - —thermal and power interfaces: ACPI
    - – Anticipate processor problems
- CHALLENGE:
  - —Extend the PAPI interface to address multiple counter domains
  - —Preserve the PAPI calling sematics, ease of use, and platform independence for existing applications

PAPI 3.0 Design



PAPI 4.0 Multiple Substrate Design


PAPI High Level
PAPI Low Level
Portable Layer
Hardware Independent Layer
Machine Specific Layer
PAPI Processor Dependent Substrate
Kernel Extension
PAPI Off-Processor Dependent Substrate
Kernel Extension
Operating System
Hardware Performance Counters
Off-Processor Hardware Counters

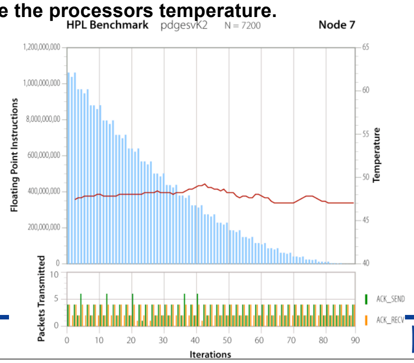


PAPI 4.0

- Multi-substrate work complete
- Substrates available for
  - —ACPI (Advanced Configuration and Power Interface)
  - —Myrinet MX
- Substrates under development for
  - —Infiniband
  - —GigE
- Friendly User release available now for CVS checkout
- PAPI 4.0 Beta release expected Q2, 2006

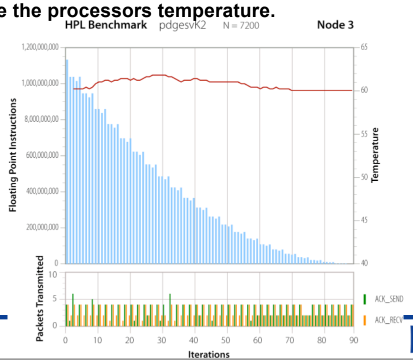
Temperature Sensor

- AMD Opteron provides an on-die thermal diode with anode and cathode brought out to processor pins.
- This diode can be read by an external temperature sensor to determine the processors temperature.

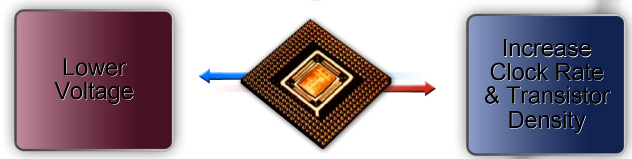


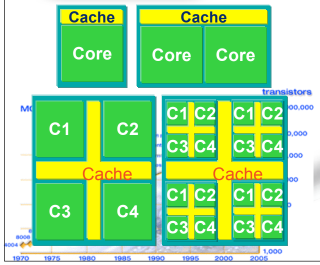
Temperature Sensor

- AMD Opteron provides an on-die thermal diode with anode and cathode brought out to processor pins.
- This diode can be read by an external temperature sensor to determine the processors temperature.



Increasing CPU Performance: A Delicate Balancing Act





We have seen increasing number of gates on a chip and increasing clock speed.

Heat becoming an unmanageable problem, Intel Processors > 100 Watts

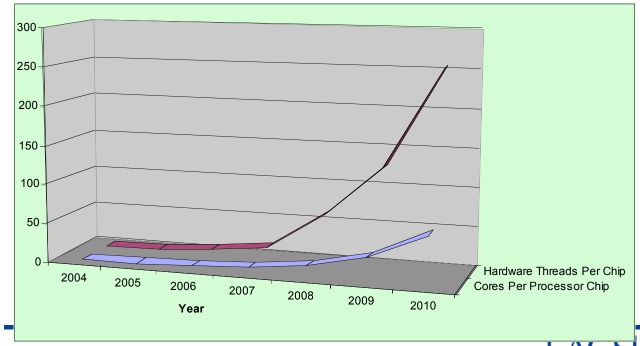
We will not see the dramatic increases in clock speeds in the future.

However, the number of gates on a chip will continue to increase.

Intel Yonah will double the processing power on a per watt basis. AMD, SUN, IBM, … have it.

CPU Desktop Trends 2004-2010

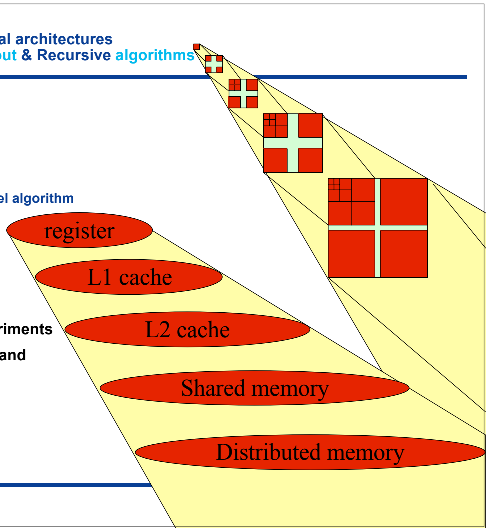
- Relative processing power will continue to double every 18 months
- 256 logical processors per chip in late 2010



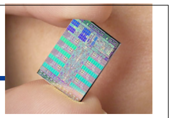
LACSI

Recursive/Fractal architectures
Recursive/Fractal data layout & Recursive algorithms

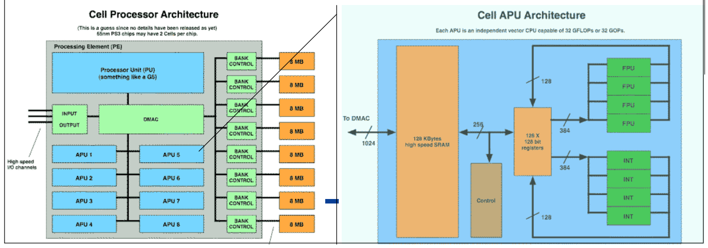
- Enable:
  - — Register blocking
  - — L1cache bocking
  - — L2 cache blocking
  - — Natural layout for parallel algorithm
- Close to the 2D block cyclic distribution
- Proven efficient by experiments on recursive algorithms and recursive data layout (see Gustavson et al.)



Things to Watch: Cell Processor - PlayStation 3



- The PlayStation 3's chip based on the "Cell" processor (IBM, Sony, & Toshiba)
- Each Cell chip contains 9 processors – 1 PowerPC & 8 APUs.
  - — An APU is a self contained vector processor which acts independently from the others.
  - — 4 floating point units capable of a total of 32 Gflop/s (8 Gflop/s each)
  - — 256 Gflop/s peak! 32 bit floating point; 64 bit floating point at 25 Gflop/s.
  - — IEEE format, but only rounds toward zero
  - — Datapaths "lite"


Cell Processor Architecture

(This is a guess since no details have been released as yet)
65nm PS3 chips may have 2 Cells per chip.

Cell APU Architecture

Each APU is an independent vector CPU capable of 32 GFLOPs or 32 GOPs.

32 and 64 Bit Floating Point Arithmetic

- Use 32 bit floating point whenever possible and resort to 64 bit floating point when needed to refine solution.
- Have done this for years with iterative refinement for dense systems of linear equations.

Solve Ax = b in lower precision, keeping the factorization (L*U = A*P)

—Computer in higher precision r = b – A*x;

– Requires the original data A (stored in high precision)

—Solve Az = r; using the lower precision factorization;

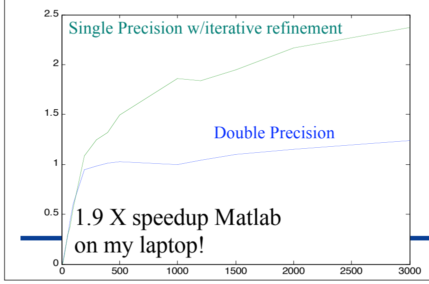
—Update solution $x_+ = x + z$ using high precision

Iterate until converged.

Requires extra storage, total is 1.5 times normal;
O(n^3) work is done in lower precision
O(n^2) work is done in high precision
Doubles number of digits per iteration
Issue if the matrix is ill-conditioned.

Another Look at Iterative Refinement

- On Cell processor, single precision is at 256 Gflop/s and double precision is at 25 Gflop/s.
- On a Pentium; using SSE2, single precision can perform 4 floating point operations per cycle and in double precision 2 floating point operations per cycle.
- Reduced memory traffic



1.9 X speedup Matlab on my laptop!

Cluster w/3.2 GHz Xeons

#procs	n	Speedup	#steps
2	2000	1.52	4
2	4000	1.60	5
2	6000	1.66	4
2	8000	1.65	5
4	4000	1.66	4
4	8000	1.78	6
4	12000	1.69	6
4	16000	1.69	5
8	8000	1.64	5
8	16000	1.78	6
8	24000	1.83	5
16	16000	1.92	5
16	32000	1.77	18
32	32000	1.84	12

Acknowledgements

- Shirley Moore, Staff, ICL
- Graham Fagg,
  Staff, ICL
- George Bosilca, Post Doc, ICL
- Jeffery Chen, Graduate Student
- Jelena Pjesivac-Grbovic,
  Graduate Student
- Haihang You, Graduate Student

- Interactions at LANL with:

—Open-MPI group at LANL

– FT-MPI and Open MPI effort

—Eclipse effort at LANL

– PAPI and Eclipse effort

—Performance group at LANL

– Multicore and performance work