Wake Forest University

Five Important Features to Consider When Computing at Scale

Jack Dongarra

University of Tennessee
Oak Ridge National Laboratory
University of Manchester

10 Fastest Computers

Rank	Site	Computer	Country	Procs/Cores	Rmax [Tflops]	Rmax/Rpeak
1	DOE/NNSA/LANL	IBM / Roadrunner - BladeCenter QS22/LS21	USA	129600	1105.0	76%
2	DOE/Oak Ridge National Laboratory	Cray / Jaguar - Cray XT5 QC 2.3 GHz	USA	150152	1059.0	77%
3	NASA/Ames Research Center/NAS	SGI / Pleiades - SGI Altix ICE 8200EX	USA	51200	487.0	80%
4	DOE/NNSA/LLNL	IBM / eServer Blue Gene Solution	USA	212992	478.2	80%
5	DOE/Argonne National Laboratory	IBM / Blue Gene/P Solution	USA	163840	450.3	81%
6	NSF/Texas Advanced Computing Center/Univ. of Texas	Sun / Ranger - SunBlade x6420	USA	62976	433.2	75%
7	DOE/NERSC/LBNL	Cray / Franklin - Cray XT4	USA	38642	266.3	75%
8	DOE/Oak Ridge National Laboratory	Cray / Jaguar - Cray XT4	USA	30976	205.0	79%
9	DOE/NNSA/Sandia National Laboratories	Cray / Red Storm - XT3/4	USA	38208	204.2	72%
10	Shanghai Supercomputer Center	Dawning 5000A, Windows HPC 2008	China	30720	180.6	77%

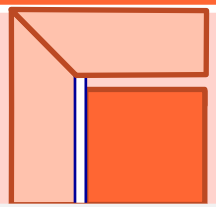
Numerical Linear Algebra Library

- Interested in developing numerical library for the fastest, largest computer platforms for scientific computing.
- Today we have machines with 100K of processors (cores) going to 1M in the next generation
- Many important issues must be addressed in the design of algorithms and software.

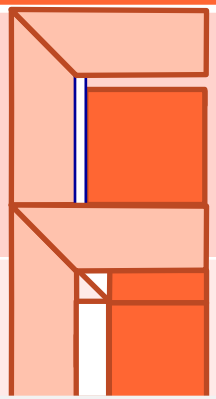
Five Important Features to Consider When Computing at Scale

- Effective Use of Many-Core and Hybrid architectures
  - Dynamic Data Driven Execution
  - Block Data Layout
- Exploiting Mixed Precision in the Algorithms
  - Single Precision is 2X faster than Double Precision
  - With GP-GPUs 10x
- Self Adapting / Auto Tuning of Software
  - Too hard to do by hand
- Fault Tolerant Algorithms
  - With 100K - 1M cores things will fail
- Communication Avoiding Algorithms
  - For dense computations from O(n *log* p) to O(*log* p) communications
  - GMRES s-step compute (x, Ax, A^2x, … A^sx)

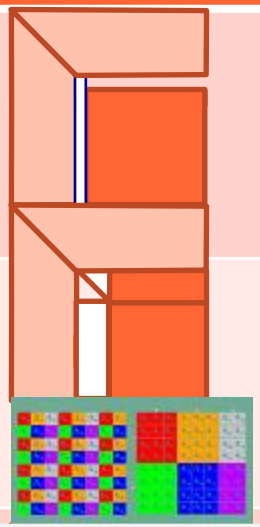
A New Generation of Software:

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations

A New Generation of Software:

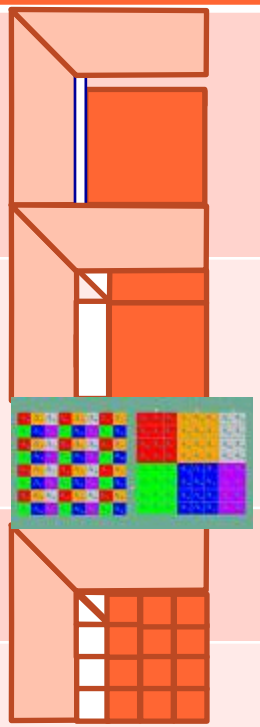
Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache		Rely on - Level-3 BLAS

A New Generation of Software:

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations

A New Generation of Software:

Parallel Linear Algebra Software for Multicore Architectures (PLASMA)

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations
ScaLAPACK (90's) (Distributed Memory)		Rely on - PBLAS Mess Passing
PLASMA (00's) New Algorithms (many-core friendly)		Rely on - a DAG/scheduler - block data layout - some extra kernels

Those new algorithms

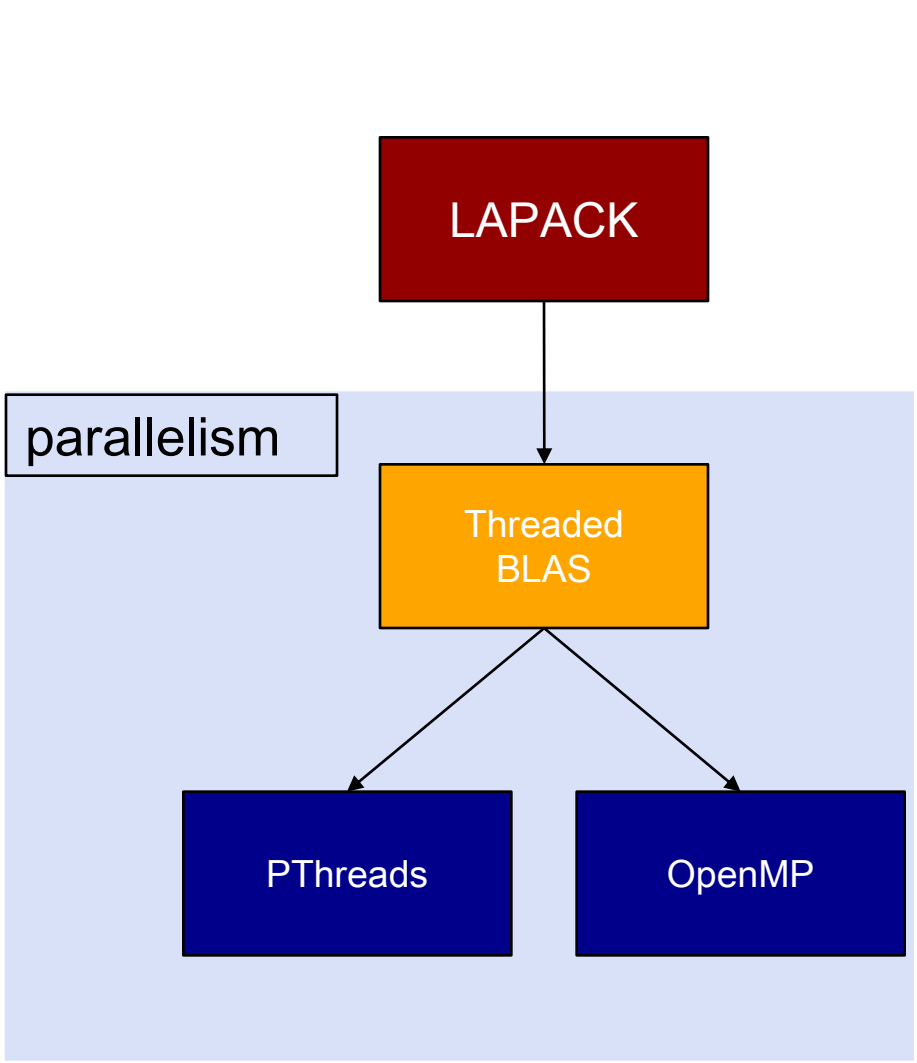
- have a very low granularity, they scale very well (multicore, petascale computing, …)
- removes a lots of dependencies among the tasks, (multicore, distributed computing)
- avoid latency (distributed computing, out-of-core)
- rely on fast kernels

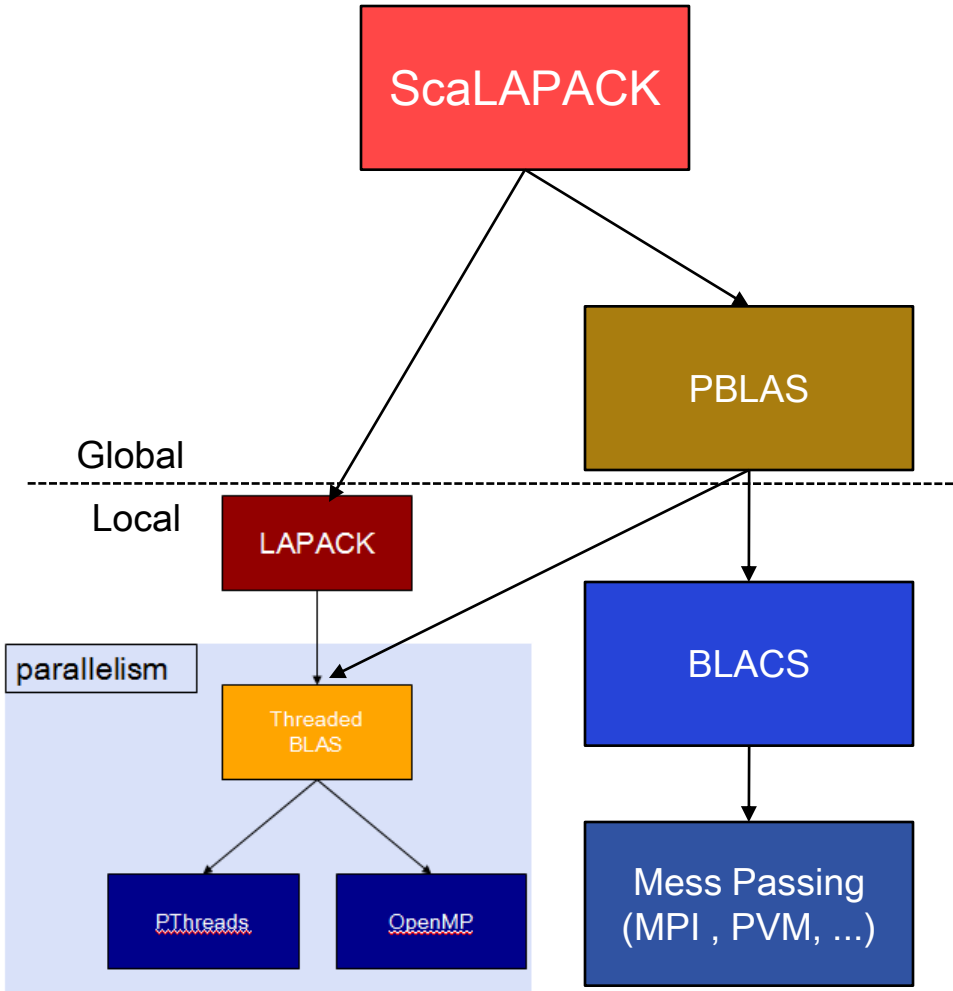
Those new algorithms need new kernels and rely on efficient scheduling algorithms.

Major Changes to Software

- Must rethink the design of our software
  - Another disruptive technology
    - Similar to what happened with cluster computing and message passing
  - Rethink and rewrite the applications, algorithms, and software
- Numerical libraries for example will change
  - For example, both LAPACK and ScaLAPACK will undergo major changes to accommodate this

LAPACK and ScaLAPACK





About 1 million lines of code

Steps in the LAPACK LU

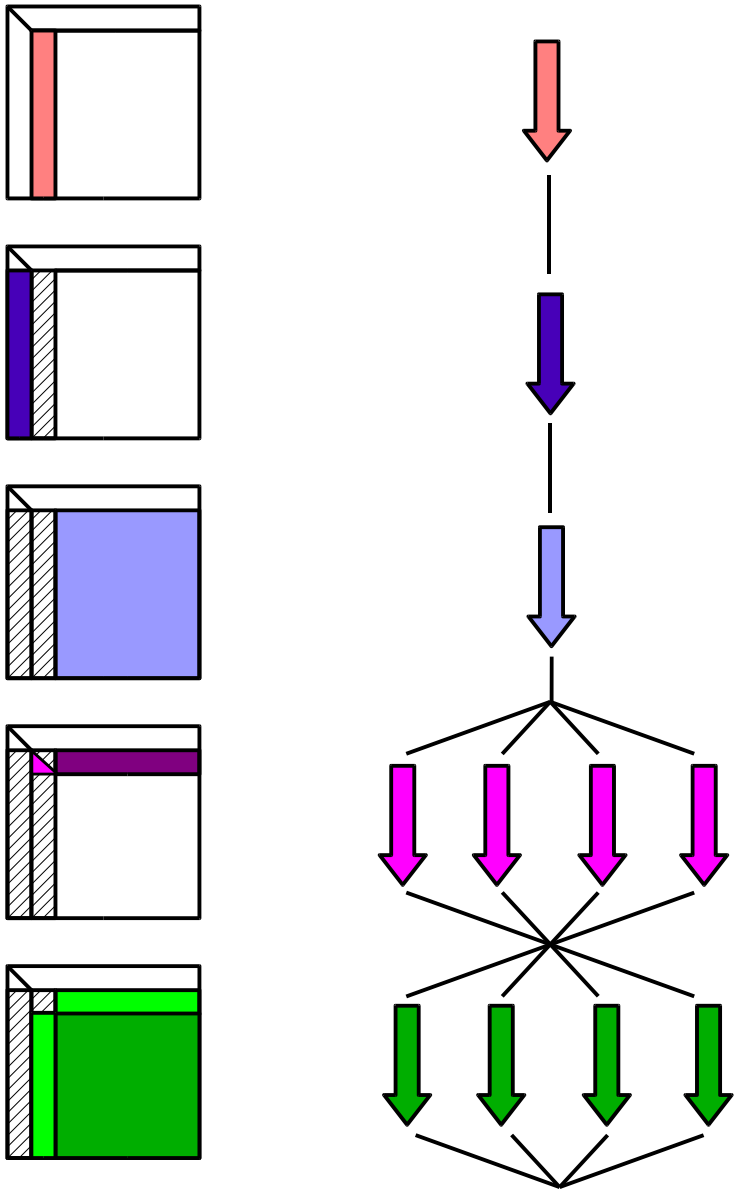
DGETF2
(Factor a panel)

DLSWP
(Backward swap)

DLSWP
(Forward swap)

DTRSM
(Triangular solve)

DGEMM
(Matrix multiply)



LAPACK

LAPACK

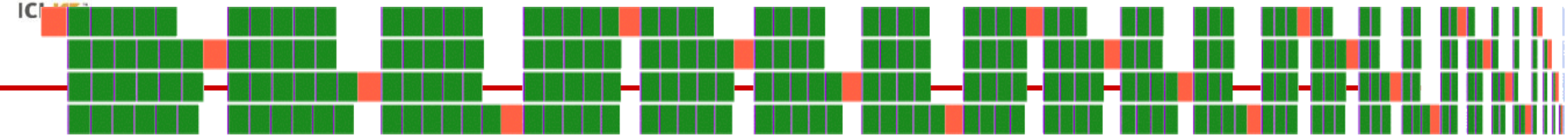
LAPACK

BLAS

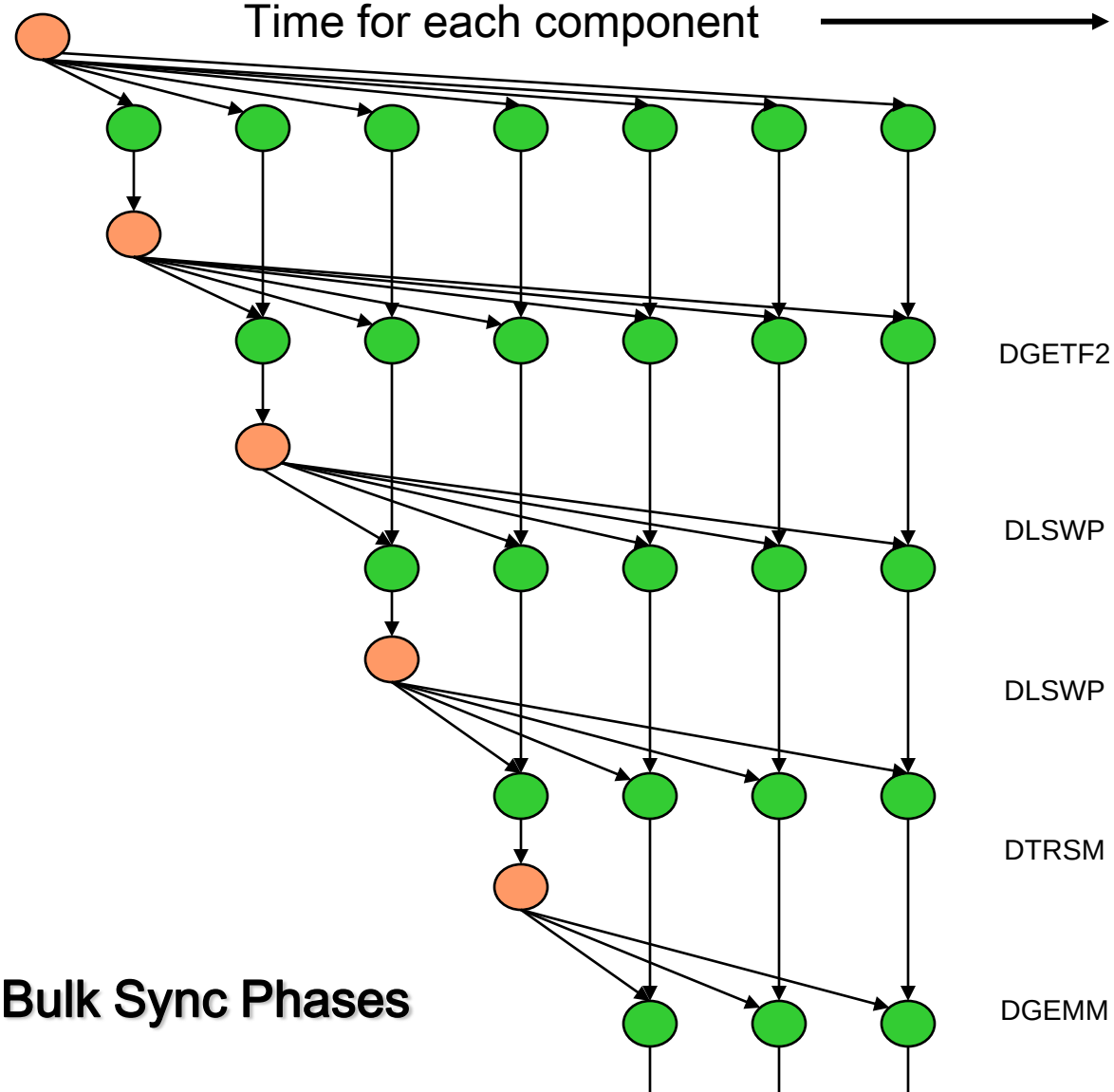
BLAS

LU Timing Profile (4 processor system)

Threads – no lookahead



Time for each component



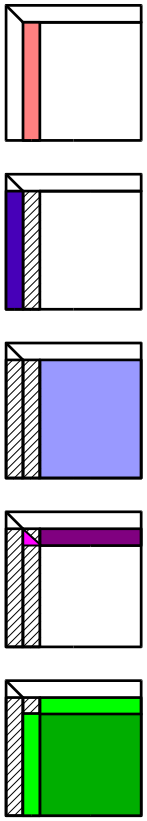
DGETF2

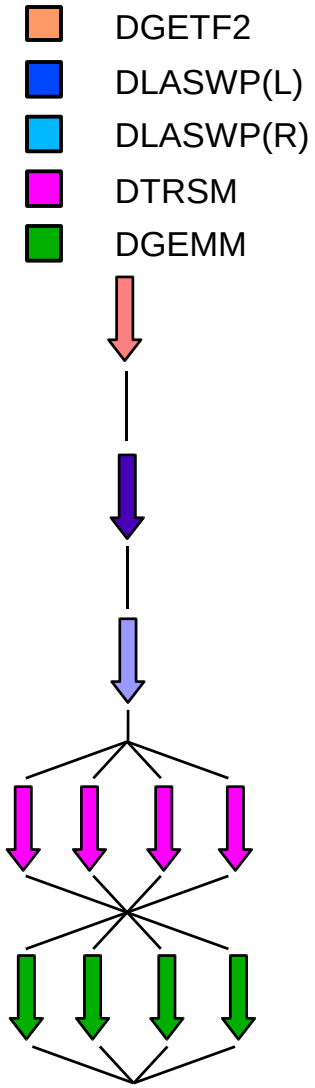
DLSWP

DLSWP

DTRSM

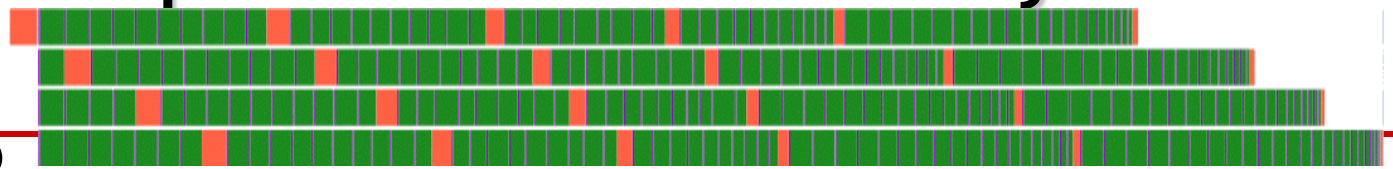
DGEMM

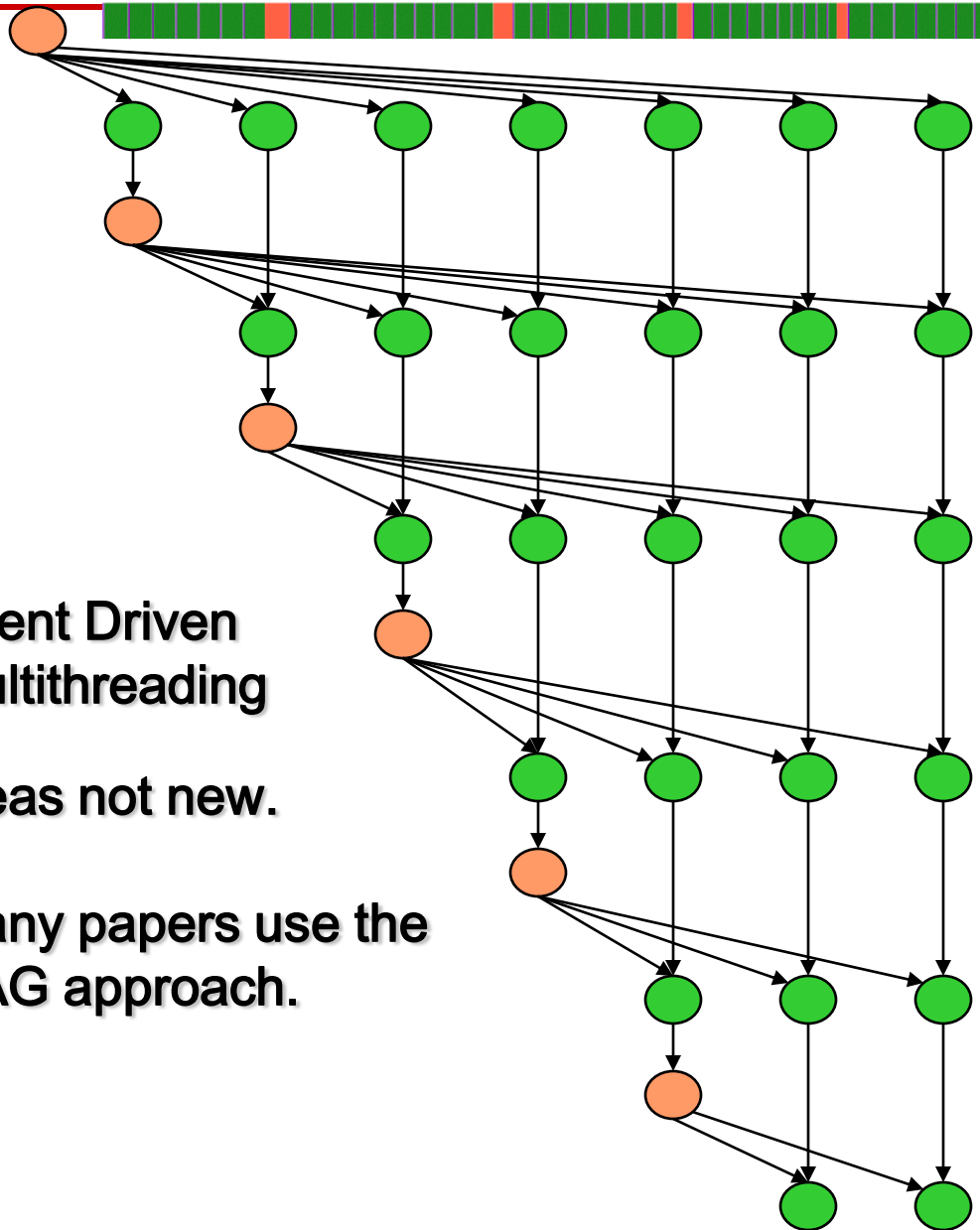




Bulk Sync Phases

Adaptive Lookahead - Dynamic





Event Driven Multithreading

Ideas not new.

Many papers use the DAG approach.

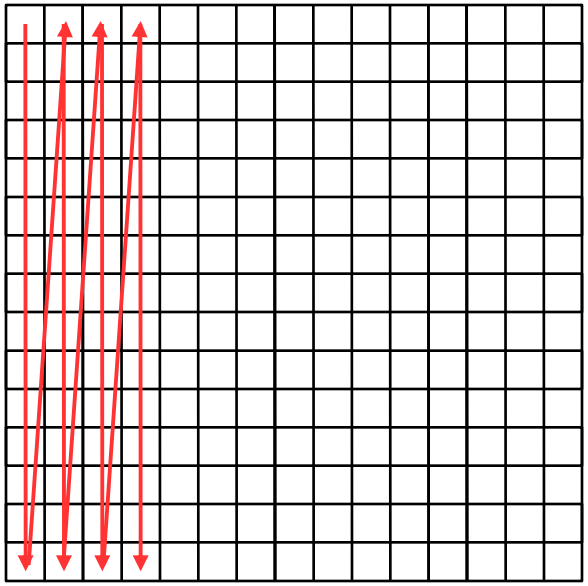
```
while(1)
    fetch_task();
    switch(task.type) {
        case PANEL:
            dgetf2();
            update_progress();
        case COLUMN:
            dlaswp();
            dtrsm();
            dgemm();
            update_progress();
        case END:
            for()
                dlaswp();
            return;
    }
}
```

Reorganizing algorithms to use this approach

Achieving Fine Granularity

Fine granularity may require novel data formats to overcome the limitations of BLAS on small chunks of data.

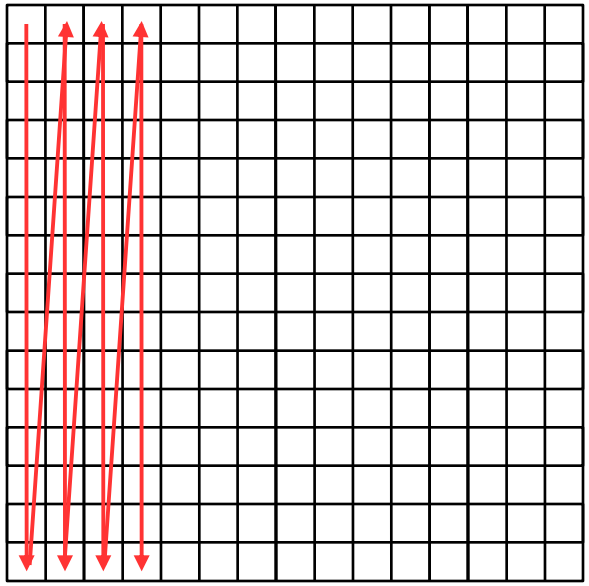
Column-Major



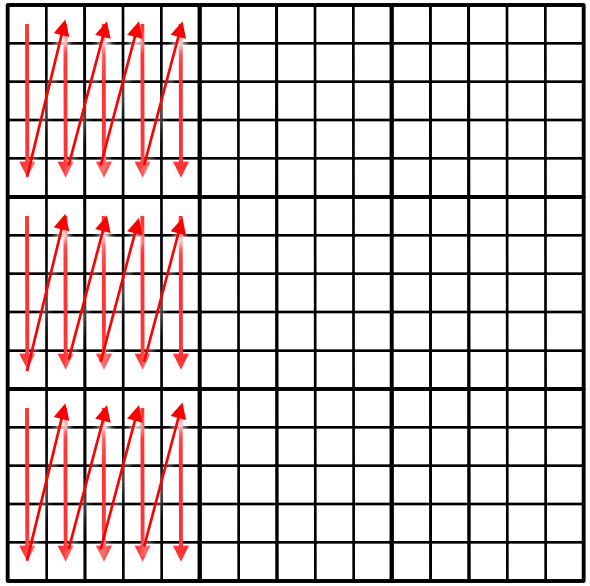
Achieving Fine Granularity

Fine granularity may require novel data formats to overcome the limitations of BLAS on small chunks of data.

Column-Major



Blocked



PLASMA (Redesign LAPACK/ScaLAPACK)

Parallel Linear Algebra Software for Multicore Architectures

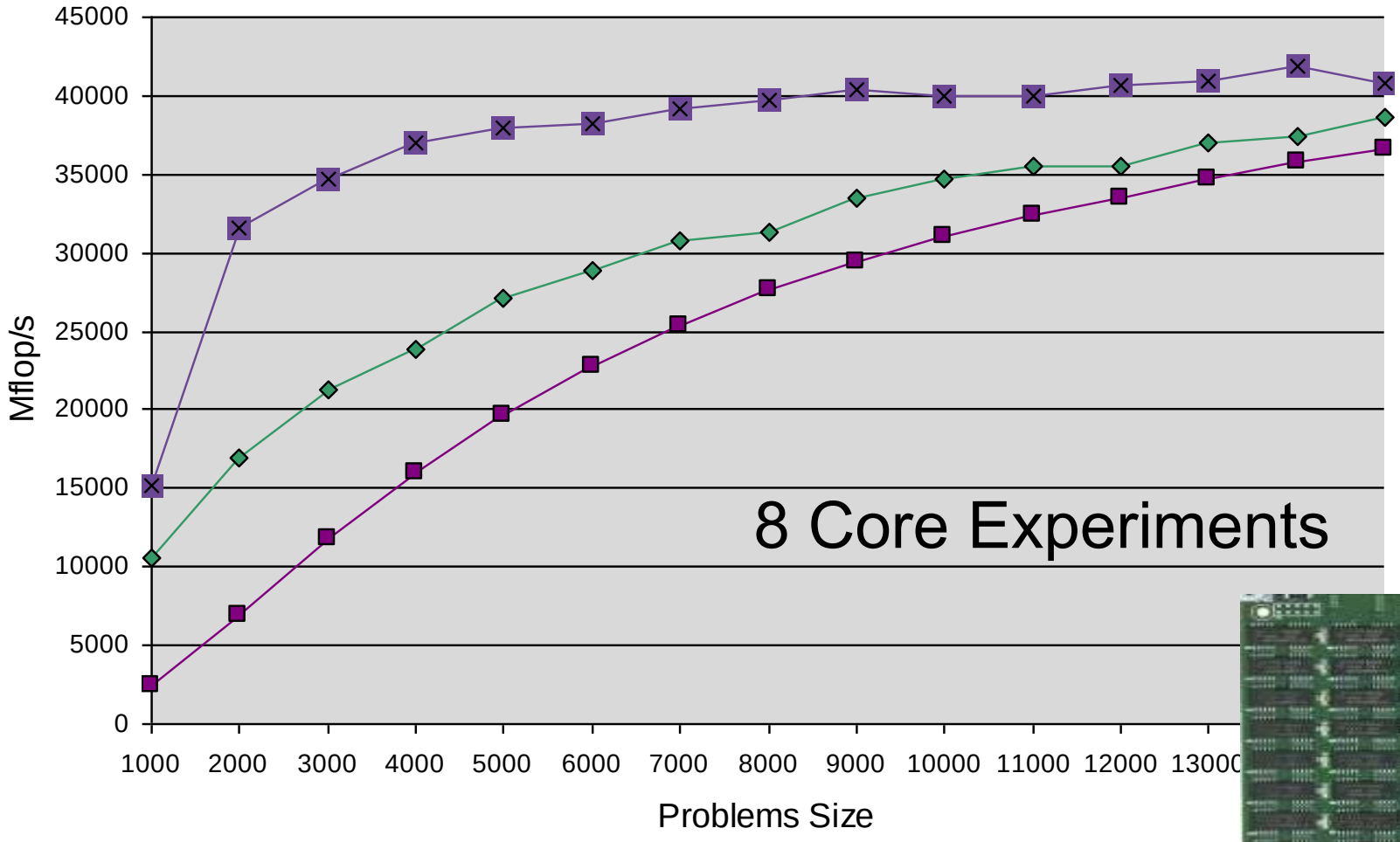
- Asychronicity
  - Avoid fork-join (Bulk sync design)
- Dynamic Scheduling
  - Out of order execution
- Fine Granularity
  - Independent block operations
- Locality of Reference
  - Data storage - Block Data Layout

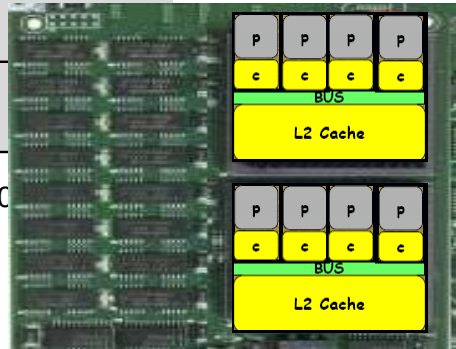
Lead by Tennessee and Berkeley similar to LAPACK/ScaLAPACK as a community effort

Intel's Clovertown Quad Core

3 Implementations of LU factorization
Quad core w/2 sockets per board, w/ 8 Treads

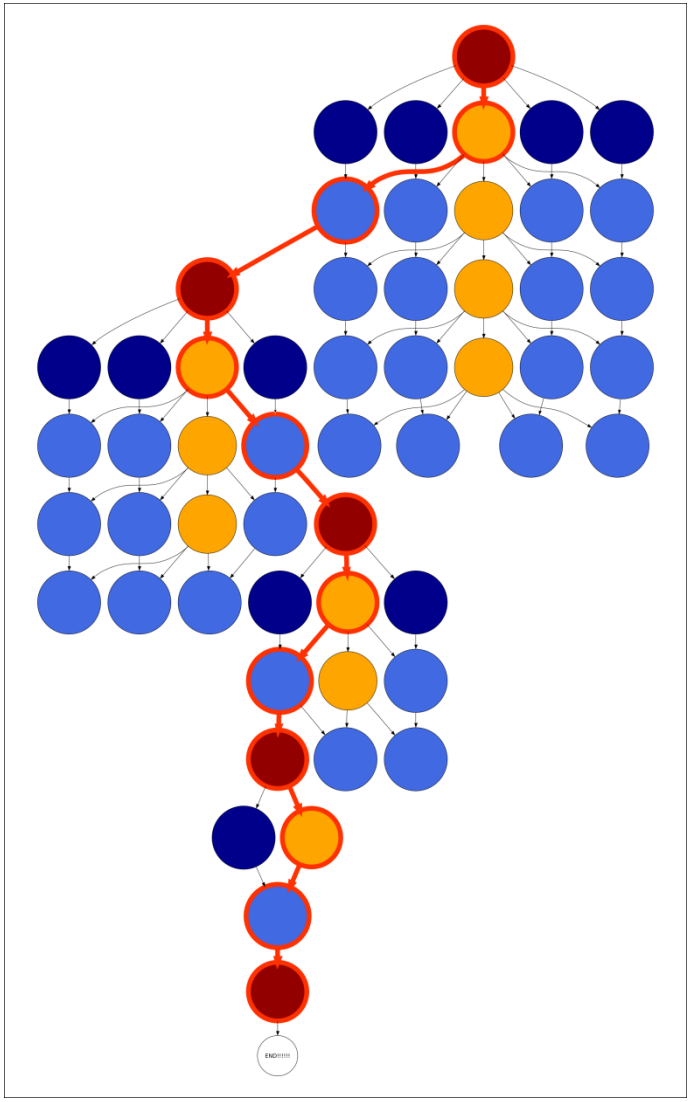
1. LAPACK (BLAS Fork-Join Parallelism)
2. ScaLAPACK (Mess Pass using mem copy)
3. DAG Based (Dynamic Scheduling)





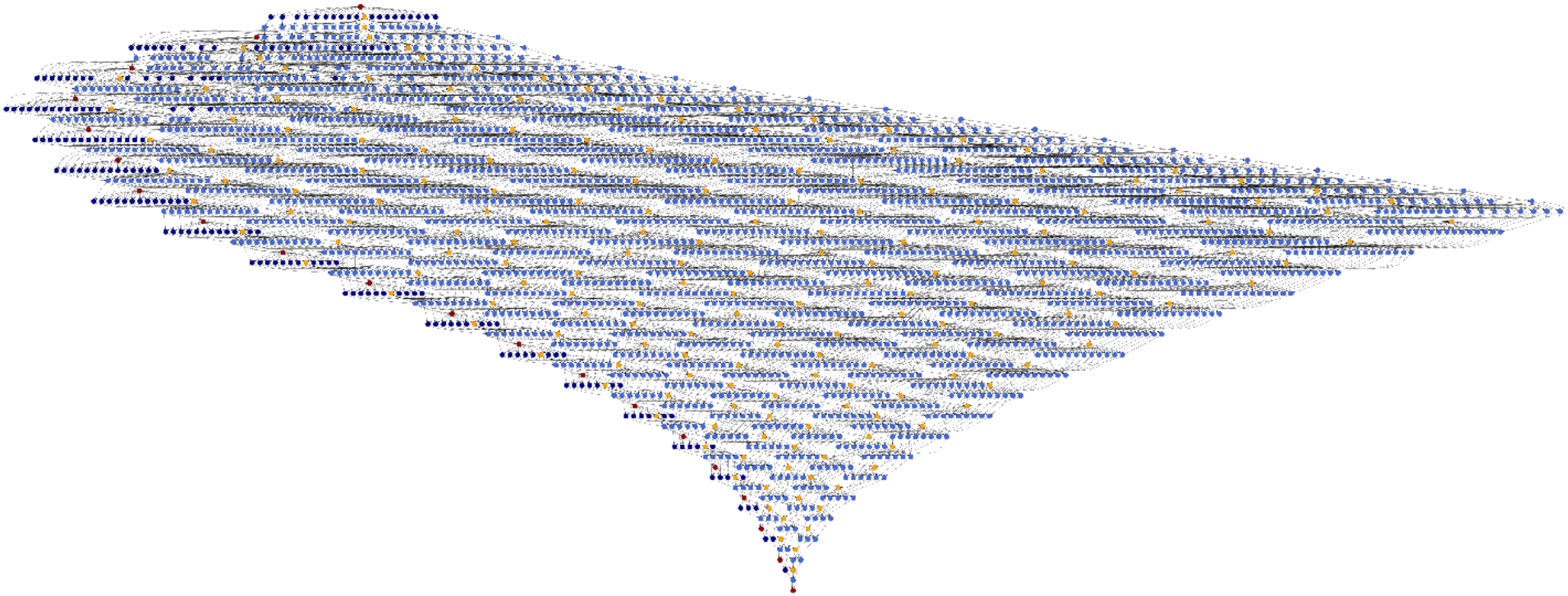
If We Had A Small Matrix Problem

- We would generate the DAG, find the critical path and execute it.
- DAG too large to generate ahead of time
  - Not explicitly generate
  - Dynamically generate the DAG as we go
- Machines will have large number of cores in a distributed fashion
  - Will have to engage in message passing
  - Distributed management
  - Locally have a run time system



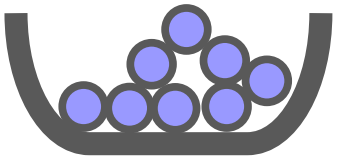
The DAGs are Large

- Here is the DAG for a factorization on a 20 x 20 matrix



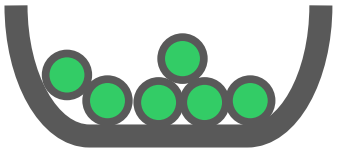
- For a large matrix say O(10^6) the DAG is huge
- Many challenges for the software

Each Node or Core Will Have A Run Time System



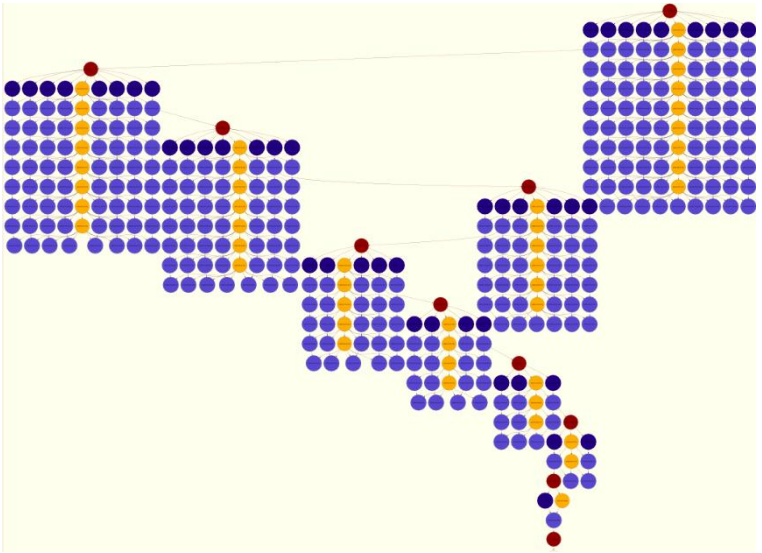
BIN 1

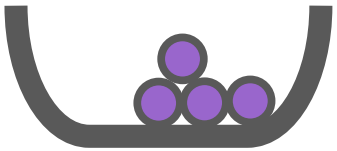
- some dependencies satisfied
- waiting for all dependencies



BIN 2

- all dependencies satisfied
- some data delivered
- waiting for all data





BIN 3

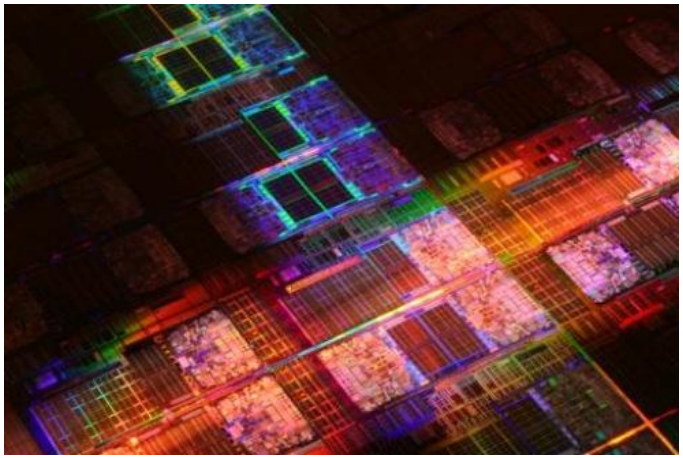
- all data delivered
- waiting for execution

Some Questions

- What's the best way to represent the DAG?
- What's the best approach to dynamically generating the DAG?
- What run time system should we use?
  - We will probably build something that we would target to the underlying system's RTS.
  - Per node or core?
- What about work stealing?
  - Can we do better than nearest neighbor work stealing?
- What does the program look like?
  - Experimenting with SMPss, Cilk, Charm++, UPC, Intel Threads
  - We would like to reuse as much of the existing software as possible
  - For software reuse, looking at a set of Task-BLAS with work with a RTS

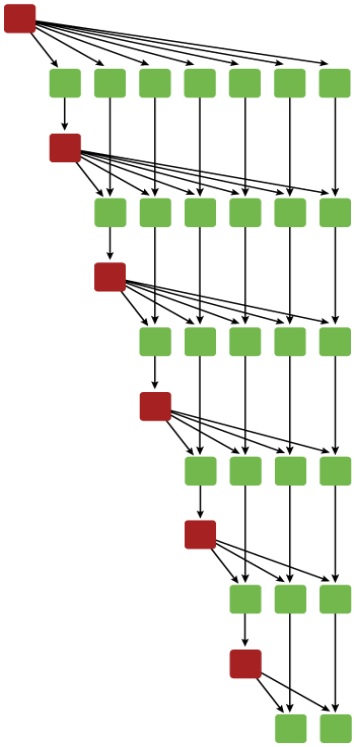
Future Computer Systems

- Most likely be a hybrid design
- Think standard multicore chips and GPUs

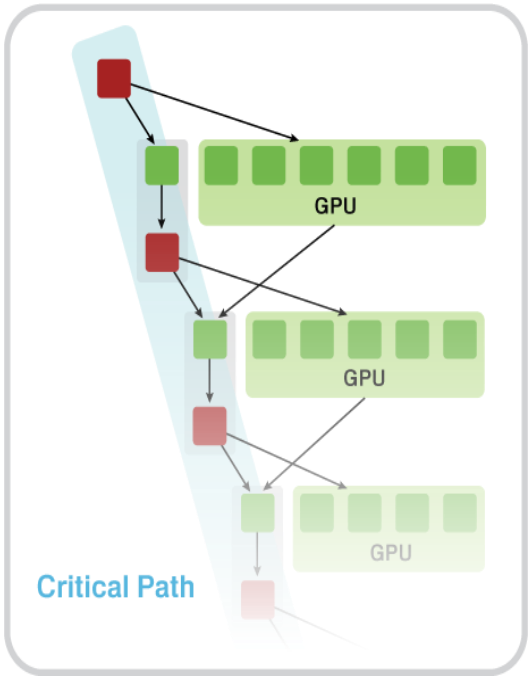


Extracting Parallelism

Algorithms as DAGs
(small tasks/tiles for multicore)



Current hybrid CPU+GPU algorithms
(small and large tasks)



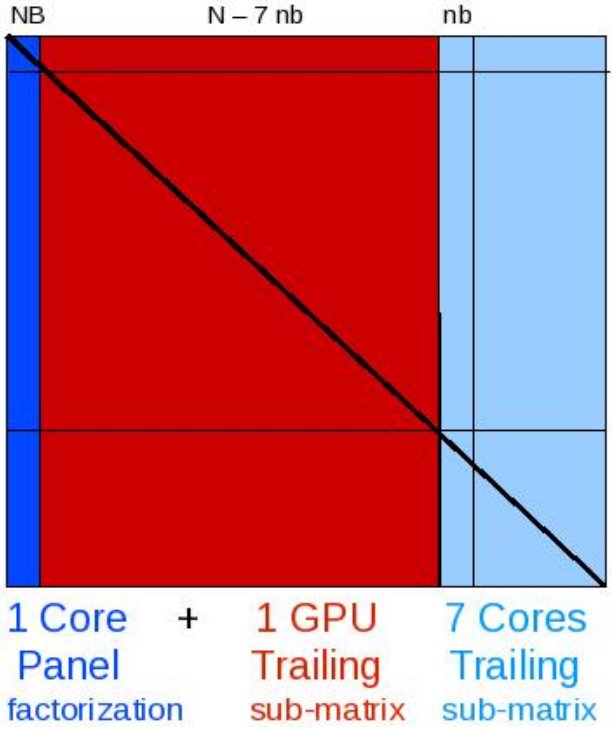
Approach:
- Critical path is done on CPU and overlapped with the GPU work whenever possible through proper task scheduling (e.g. look-ahead)
- Algorithmic changes (possibly less stable)

Challenges:
- Splitting algorithms into tasks
  e.g. has to be "Heterogeneity-aware", "Auto-tuned", etc.
- Scheduling task execution
- New algorithms and studies on associated with them numerical stability
- Reusing current Multicore results
  (although **Multicore ⊂ Hybrid computing**)

[Current Multicore efforts are on "tiled" algorithms and "uniform task splitting" with "block data layout";
Current Hybrid work leans more towards standard data layout, algorithms with variable block sizes, large GPU tasks and large CPU-GPU data transfers to minimize latency overheads, etc]
[Hybrid computing presents more opportunity to better match algorithmic requirements to underlaying architecture components; e.g. current main factorizations; How about Hessenberg reduction, that is hard (open problem) on Multicore?]

Current work

- **Algorithms (in particular LU) for Multicore + GPU systems**
- **Challenges**
  - How to split the computation
  - Software development
  - Tuning



Work splitting
(for single GPU + 8 cores host)

Performance of Single Precision on Conventional Processors

- Realized have the similar situation on our commodity processors.
  - That is, SP is 2X as fast as DP on many systems
- The Intel Pentium and AMD Opteron have SSE2
  - 2 flops/cycle DP
  - 4 flops/cycle SP
- IBM PowerPC has AltiVec
  - 8 flops/cycle SP
  - 4 flops/cycle DP
    - No DP on AltiVec

	Size	SGEMM/ DGEMM	Size	SGEMV/ DGEMV
AMD Opteron 246	3000	2.00	5000	1.70
UltraSparc-IIe	3000	1.64	5000	1.66
Intel PIII Coppermine	3000	2.03	5000	2.09
PowerPC 970	3000	2.04	5000	1.44
Intel Woodcrest	3000	1.81	5000	2.18
Intel XEON	3000	2.04	5000	1.82
Intel Centrino Duo	3000	2.71	5000	2.21

Single precision is faster because:
- Operations are faster
- Reduced data motion
- Larger blocks gives higher locality in cache

Idea Goes Something Like This…

- Exploit 32 bit floating point as much as possible.
  - Especially for the bulk of the computation
- Correct or update the solution with selective use of 64 bit floating point to provide a refined results
- Intuitively:
  - Compute a 32 bit result,
  - Calculate a correction to 32 bit result using selected higher precision and,
  - Perform the update of the 32 bit results with the correction using high precision.

Mixed-Precision Iterative Refinement

- Iterative refinement for dense systems, $Ax = b$, can work this way.

L U = lu(A)	$O(n^3)$
x = L\(U\b)	$O(n^2)$
r = b – Ax	$O(n^2)$
WHILE || r || not small enough	
z = L\(U\r)	$O(n^2)$
x = x + z	$O(n^1)$
r = b – Ax	$O(n^2)$
END	

  - Wilkinson, Moler, Stewart, & Higham provide error bound for SP fl pt results when using DP fl pt.

Mixed-Precision Iterative Refinement

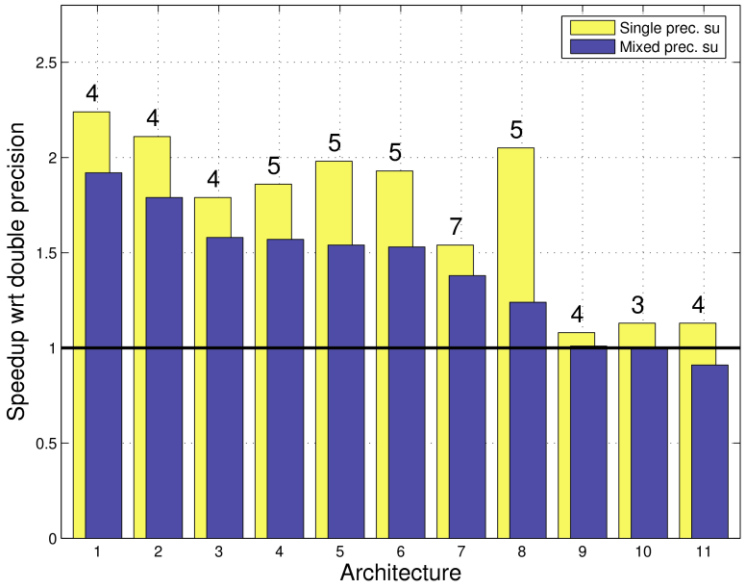
- Iterative refinement for dense systems, $Ax = b$, can work this way.

L U = lu(A)	SINGLE	$O(n^3)$
x = L\(U\b)	SINGLE	$O(n^2)$
r = b – Ax	DOUBLE	$O(n^2)$
WHILE || r || not small enough		
z = L\(U\r)	SINGLE	$O(n^2)$
x = x + z	DOUBLE	$O(n^1)$
r = b – Ax	DOUBLE	$O(n^2)$
END		

- Wilkinson, Moler, Stewart, & Higham provide error bound for SP fl pt results when using DP fl pt.
- It can be shown that using this approach we can compute the solution to 64-bit floating point precision.

- 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
- Problems if the matrix is ill-conditioned in sp; $O(10^8)$

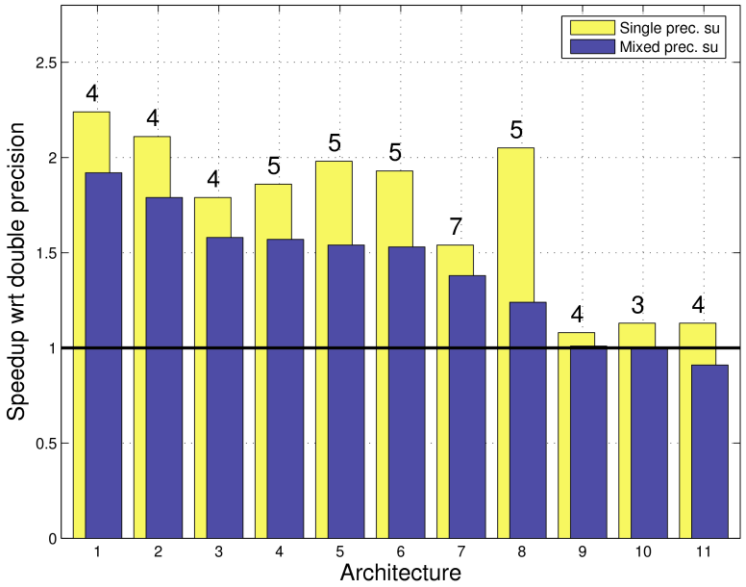
Results for Mixed Precision Iterative Refinement for Dense $Ax = b$



	Architecture (BLAS)
1	Intel Pentium III Coppermine (Goto)
2	Intel Pentium III Katmai (Goto)
3	Sun UltraSPARC IIe (Sunperf)
4	Intel Pentium IV Prescott (Goto)
5	Intel Pentium IV-M Northwood (Goto)
6	AMD Opteron (Goto)
7	Cray X1 (libsci)
8	IBM Power PC G5 (2.7 GHz) (VecLib)
9	Compaq Alpha EV6 (CXML)
10	IBM SP Power3 (ESSL)
11	SGI Octane (ATLAS)

- Single precision is faster than DP because:
  - **Higher parallelism within vector units**
    - **4 ops/cycle (usually) instead of 2 ops/cycle**
  - **Reduced data motion**
    - **32 bit data instead of 64 bit data**
  - **Higher locality in cache**
    - **More data items in cache**

Results for Mixed Precision Iterative Refinement for Dense *Ax = b*



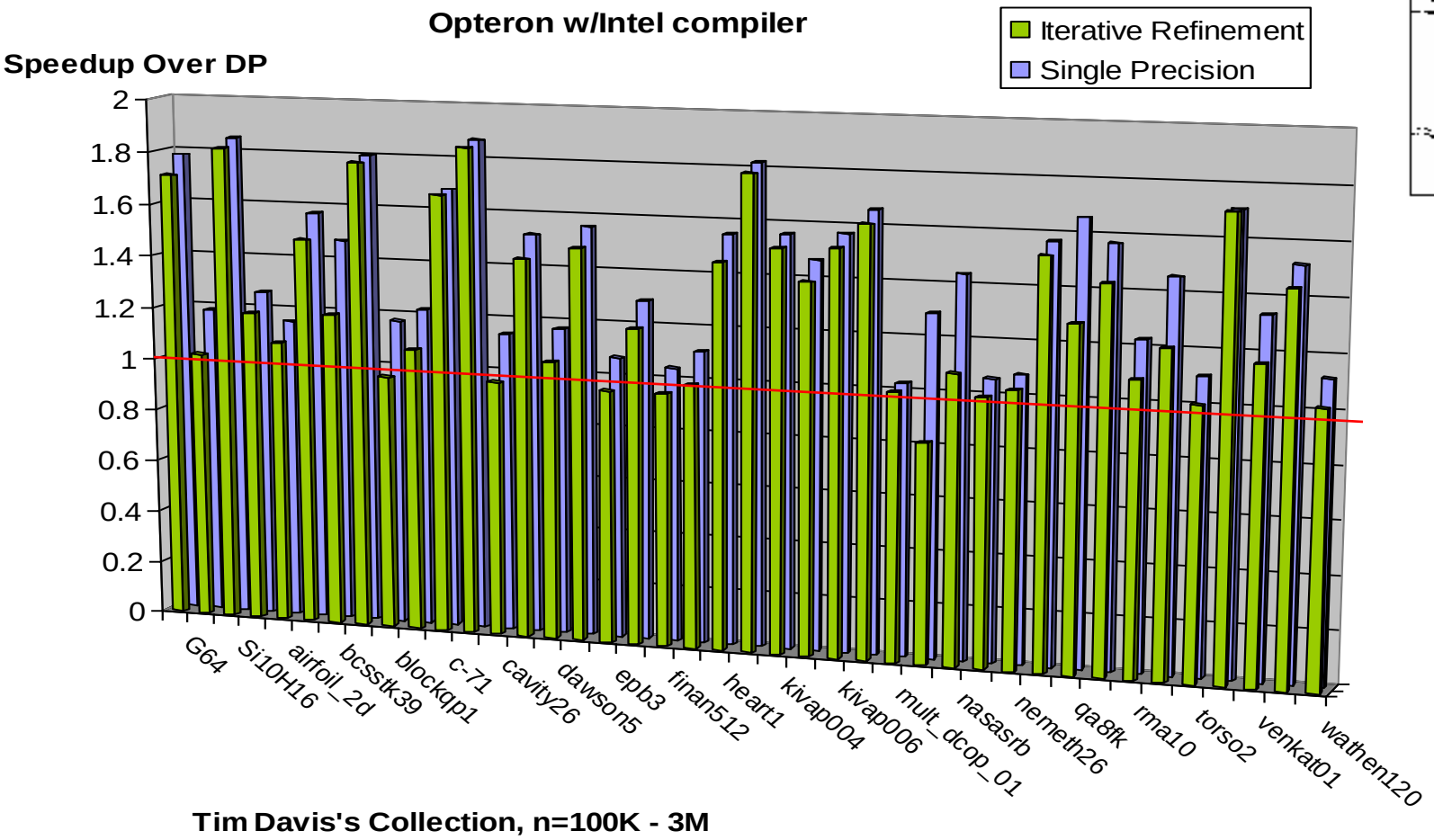
	Architecture (BLAS)
1	Intel Pentium III Coppermine (Goto)
2	Intel Pentium III Katmai (Goto)
3	Sun UltraSPARC IIe (Sunperf)
4	Intel Pentium IV Prescott (Goto)
5	Intel Pentium IV-M Northwood (Goto)
6	AMD Opteron (Goto)
7	Cray X1 (libsci)
8	IBM Power PC G5 (2.7 GHz) (VecLib)
9	Compaq Alpha EV6 (CXML)
10	IBM SP Power3 (ESSL)
11	SGI Octane (ATLAS)

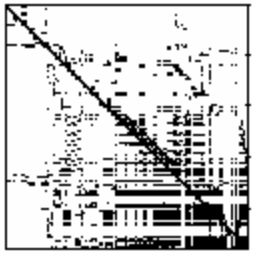
Architecture (BLAS-MPI)	# procs	*n*	DP Solve /SP Solve	DP Solve /Iter Ref	# iter
AMD Opteron (Goto – OpenMPI MX)	32	22627	1.85	1.79	6
AMD Opteron (Goto – OpenMPI MX)	64	32000	1.90	1.83	6

- Single precision is faster than DP because:
  - **Higher parallelism within vector units**
    - 4 ops/cycle (usually) instead of 2 ops/cycle
  - **Reduced data motion**
    - 32 bit data instead of 64 bit data
  - **Higher locality in cache**
    - More data items in cache

Sparse Direct Solver and Iterative Refinement

MUMPS package based on multifrontal approach which generates small dense matrix multiplies





Sparse Iterative Methods (PCG)

- **Outer/Inner Iteration**

  Outer iterations using 64 bit floating point

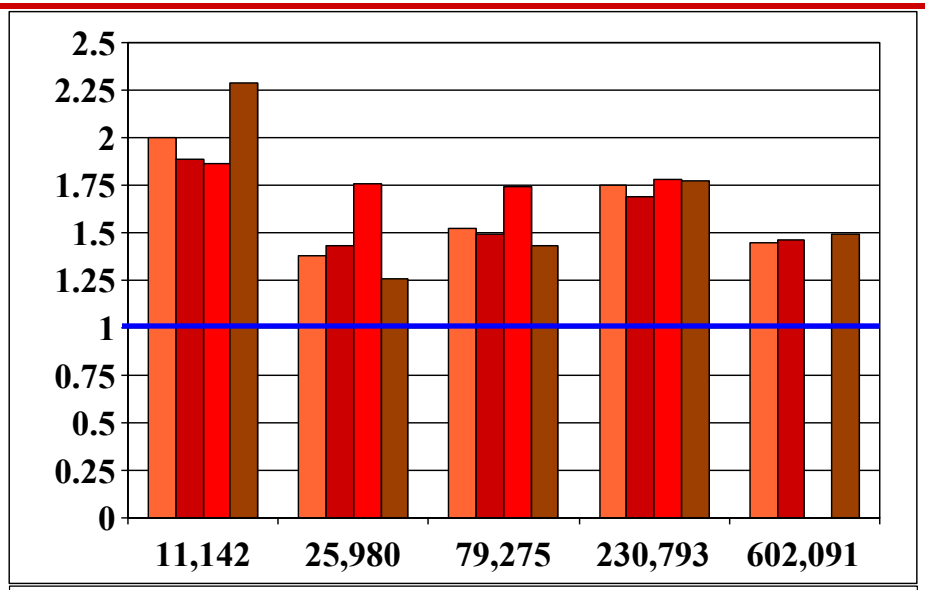
  Inner iteration:
  In 32 bit floating point

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

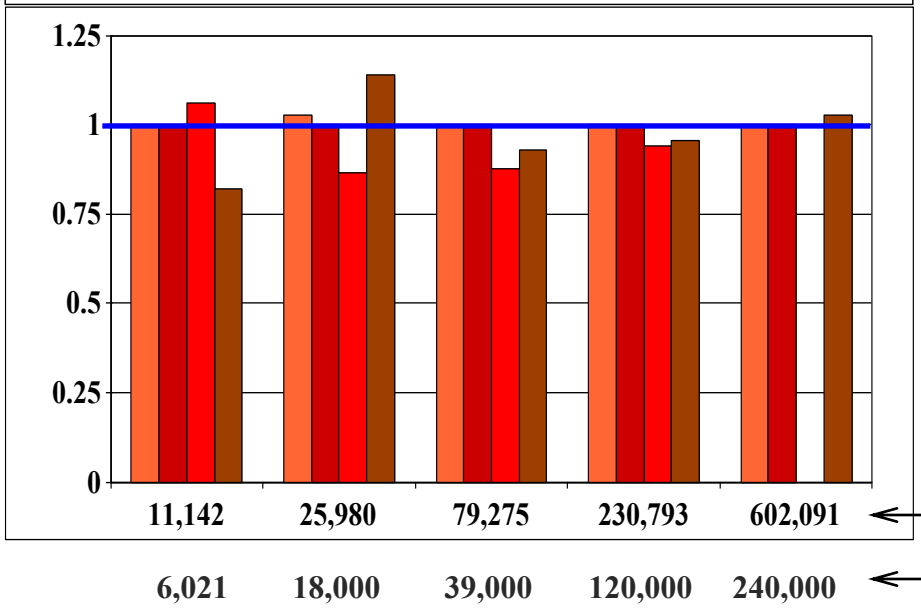
Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
for $i = 1, 2, \ldots$
  solve $Mz^{(i-1)} = r^{(i-1)}$
  $\rho_{i-1} = r^{(i-1)^T} z^{(i-1)}$
  if $i = 1$
    $p^{(1)} = z^{(0)}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
  endif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

- **Outer iteration in 64 bit floating point and inner iteration in 32 bit floating point**

Mixed Precision Computations for Sparse Inner/Outer-type Iterative Solvers



Speedups for mixed precision
Inner SP/Outer DP (SP/DP) iter. methods *vs* DP/DP
(CG^2, $GMRES^2$, PCG^2, and $PGMRES^2$ with diagonal prec.)
(***Higher is better***)



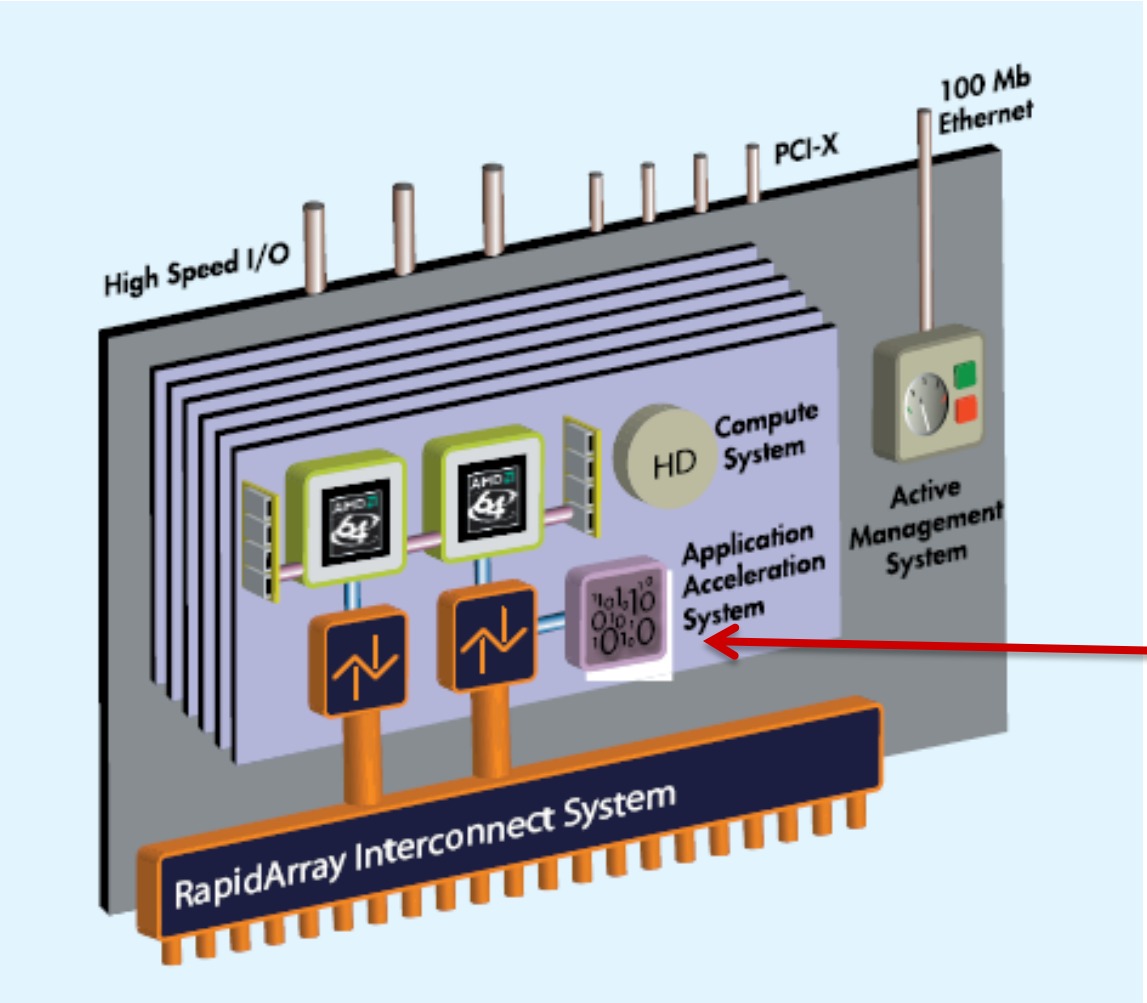
Iterations for mixed precision
SP/DP iterative methods *vs* DP/DP
(***Lower is better***)

Machine:
Intel Woodcrest (3GHz, 1333MHz bus)

Stopping criteria:
Relative to r_0 residual reduction (10^{-12})

Cray XD-1 (OctigaBay Systems)

Experiments with Field Programmable Gate Array
Specify arithmetic



Six Xilinx Virtex-4 Field Programmable Gate Arrays (FPGAs) per chassis

Mixed Precision Iterative Refinement

- FPGA Performance Test - Junqing Sun et al

Characteristics of multiplier on an FPGA* (using DSP48)

Data Formats	DSP48s	Frequency (MHz)	GFLOPs
s52e11 (double)	16/96	237	**1.42**
s51e11	16/96	238	1.43
s50e11	**9/96**	245	2.61
s34e8	9/96	289	3.08
s33e8	**4/96**	292	7.01
s23e8 (single)	4/96	339	8.14
s17e8	4/96	370	8.88
s16e8	**1/96**	331	31.78
s16e7	1/96	352	33.79
s13e7	1/96	336	**32.26**

* XC4LX160-10

Mixed Precision Iterative Refinement

- Random Matrix Test - Junqing Sun et al

Refinement iterations for customized formats (sXXe11).
Random matrices

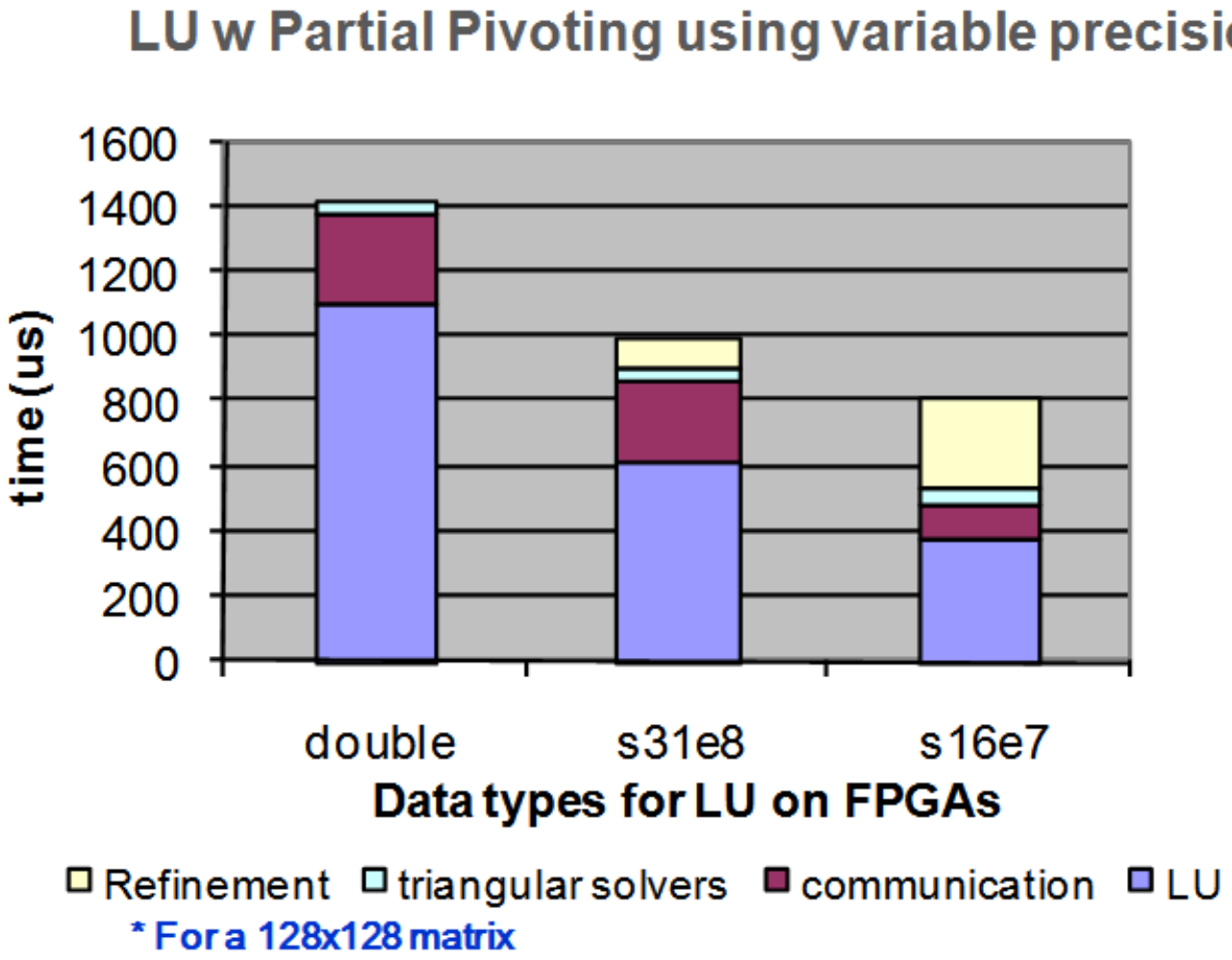
More Bits →

Problem Size \ Mantissa Bits	12	16	23	31	48	52
128	8.9	4	2	1	1	0
256	11.1	5.1	2.1	1	1	0
512	19.7	6.1	2.5	1	1	0
1024	28	6.3	2.6	1	1	0
2048	-	9.3	3	1.3	1	0
4096	-	13.3	3.1	1.43	1	0

← More Iterations

Mixed Precision Hybrid Direct Solver

- Profiled Time* on Cray-XD1 - Junqing Sun et al



High Performance Mixed-Precision Linear Solver for FPGAs,
Junqing Sun, Gregory D. Peterson, Olaf Storaasli, IEEE TPDC, 2008

Intriguing Potential

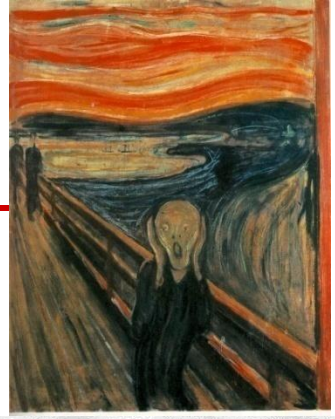
- Exploit lower precision as much as possible
  - Payoff in performance
    - Faster floating point
    - Less data to move
- Automatically switch between SP and DP to match the desired accuracy
  - Compute solution in SP and then a correction to the solution in DP
- Potential for GPU, FPGA, special purpose processors
  - Use as little you can get away with and improve the accuracy
- Applies to sparse direct and iterative linear systems and Eigenvalue, optimization problems, where Newton's method is used.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

$$x_{i+1} - x_i = -\frac{f(x_i)}{f'(x_i)}$$

Correction = - A\(b – Ax)

Fault Tolerance



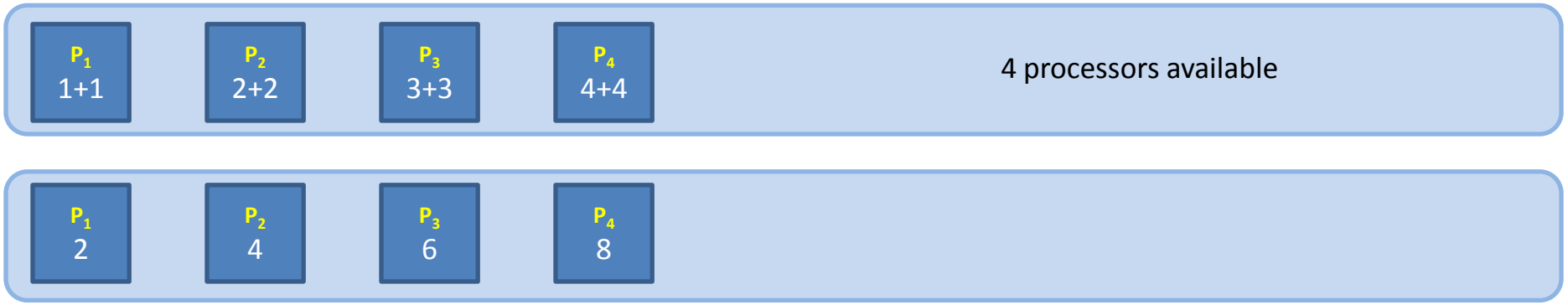
- Trends in HPC:
  - High end systems with thousand of processors.
- Increased probability of a system. failure
  - Most nodes today are robust, 3 year life.
  - Mean Time to Failure is growing shorter as systems grow and devices shrink.

- MPI widely accepted in scientific computing.
  - Process faults not tolerated in MPI model.

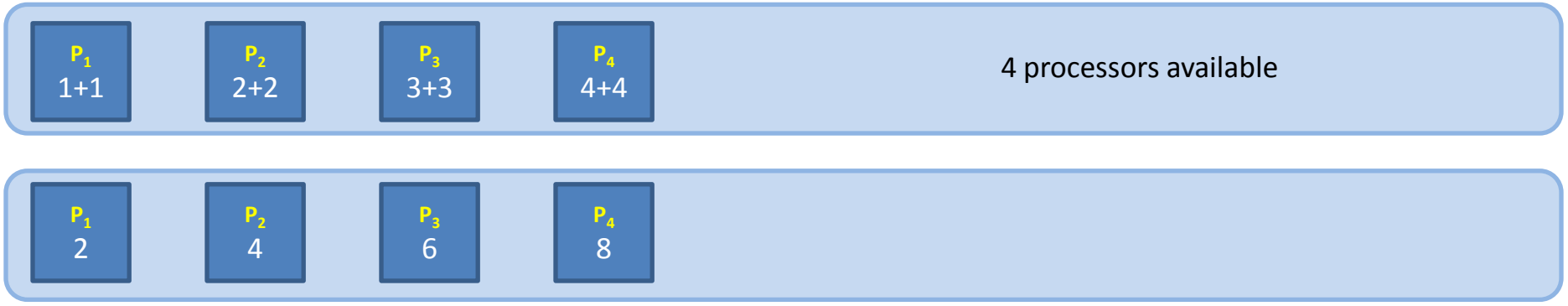
Interesting studies:

- The computer failure data repository (CFDR) http://cfdr.usenix.org/
- LANL Study: A Large-scale Study Of Failures In High-performance Computing Systems, B. Schroeder , & G. Gibson, *International Symposium on Dependable Systems and Networks (DSN 2006).*

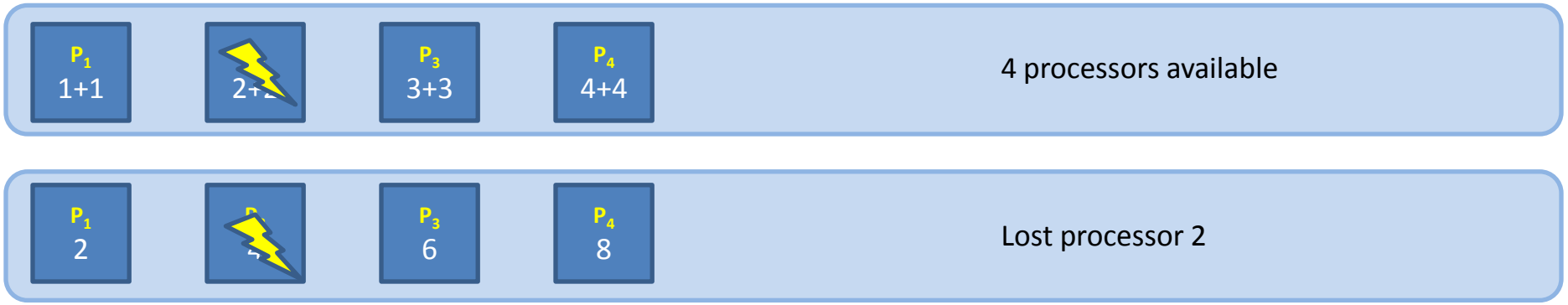
Erasure Problem



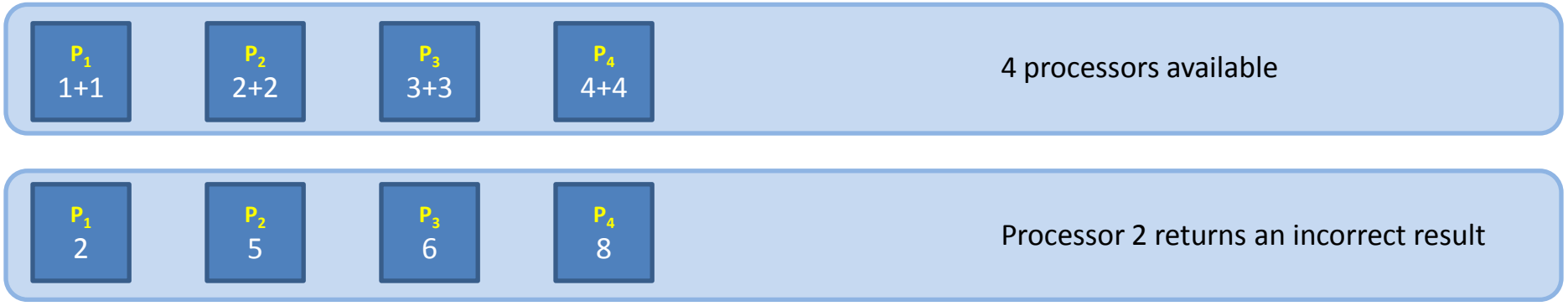
Error Problem



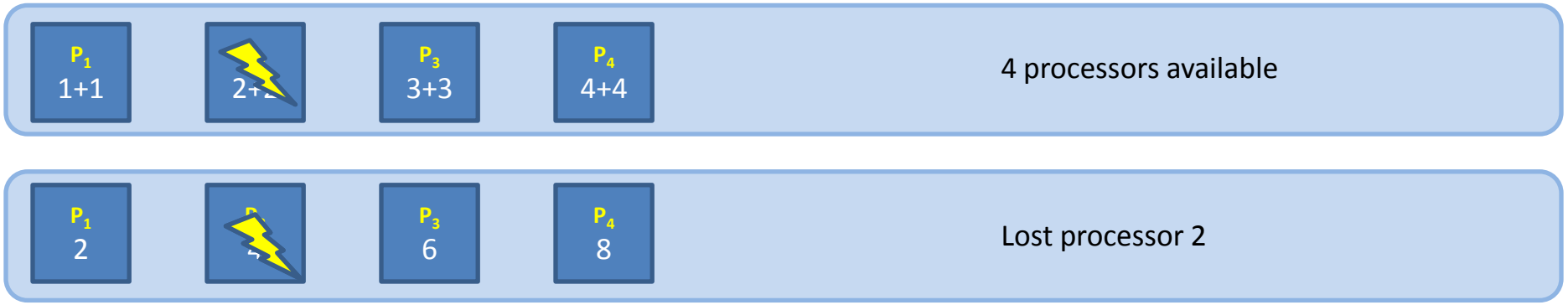
Erasure Problem



Error Problem

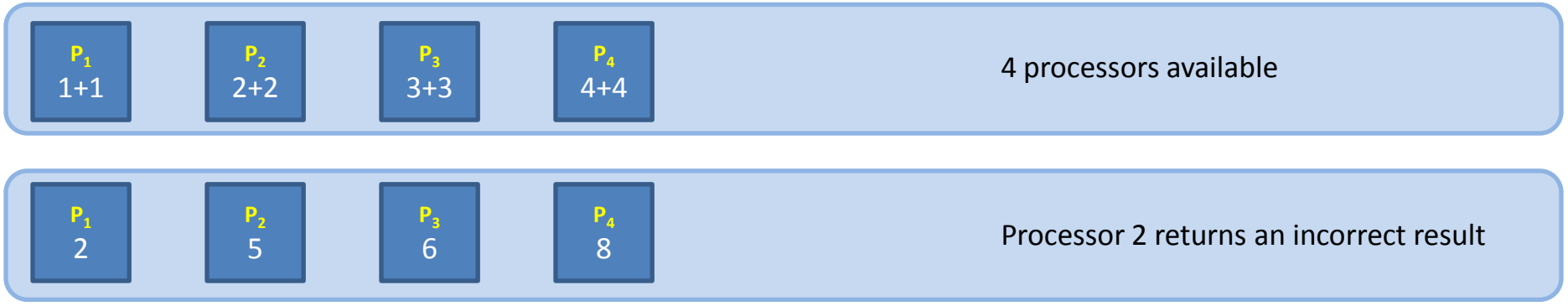


Erasure Problem



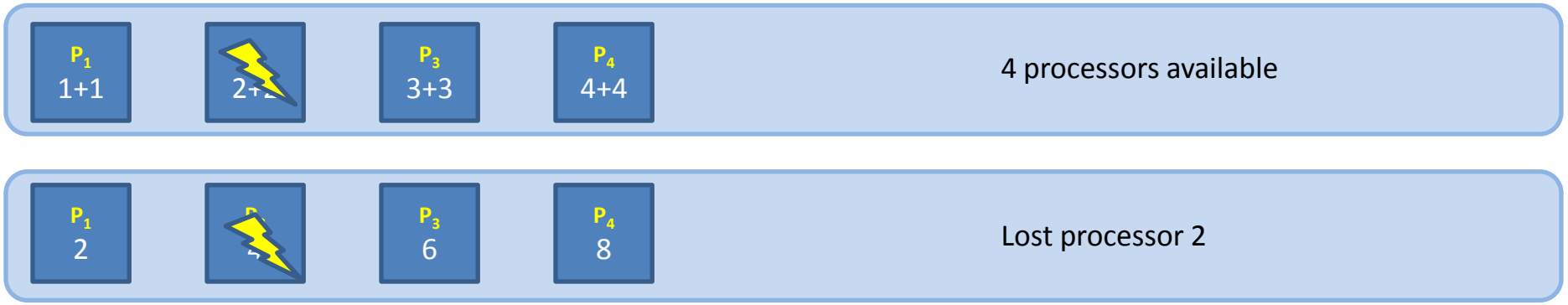
- we know whether there is an erasure or not,

Error Problem



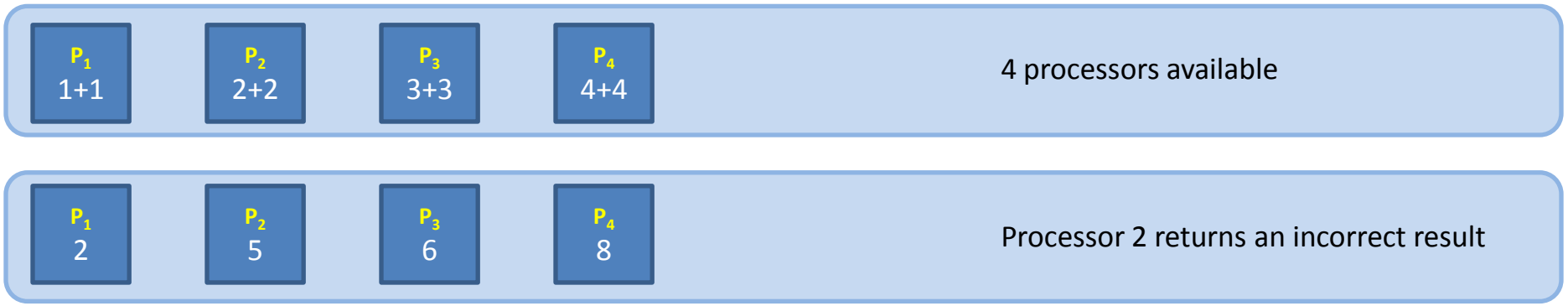
- we do not know if there is an error,

Erasure Problem



- we know whether there is an erasure or not,
- we know where the erasure is,

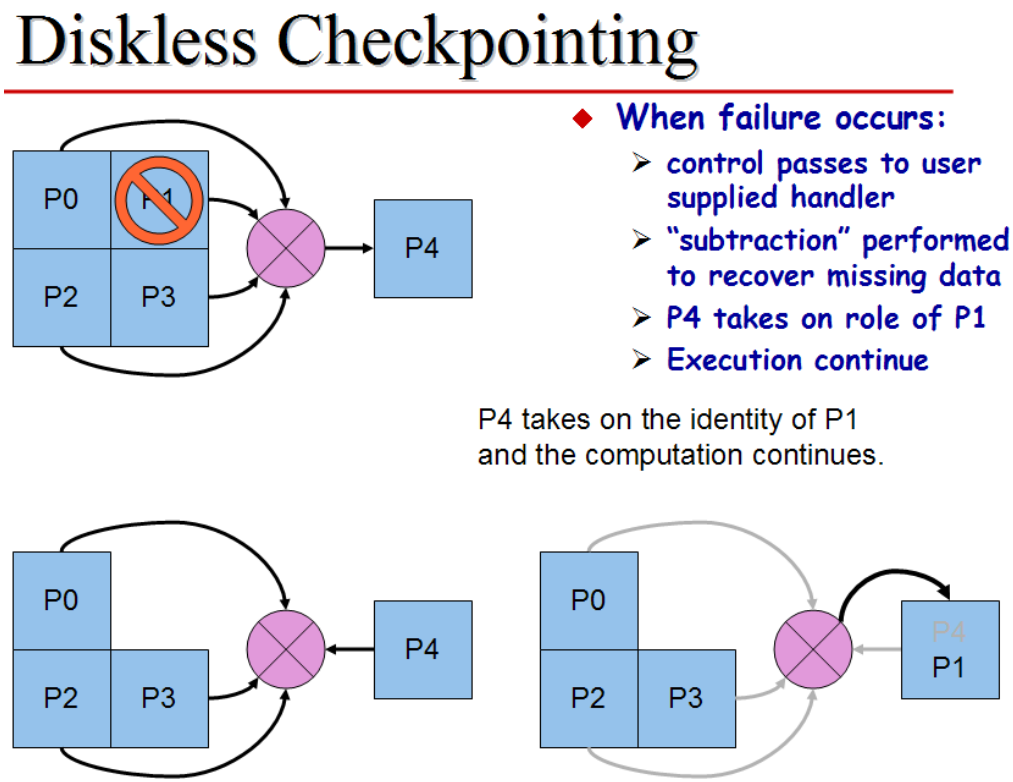
Error Problem



- we do not know if there is an error,
- assuming we know that an error occurs, we do not know where it is

Three Ideas for Fault Tolerant Linear Algebra Algorithms

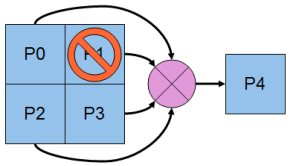
- Lossless diskless check-pointing for iterative methods
  - Checksum maintained in active processors
  - On failure, roll back to checkpoint and continue
  - No lost data



Three Ideas for Fault To Linear Algebra Algori

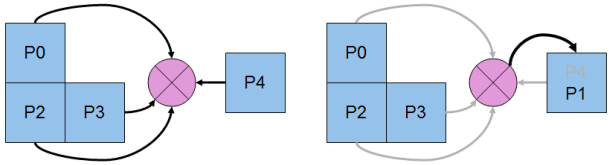
- Lossless diskless check-pointing for iterative methods
  - Checksum maintained in active processors
  - On failure, roll back to checkpoint and continue
  - No lost data
- Lossy approach for iterative methods
  - No checkpoint for computed data maintained
  - On failure, approximate missing data carry on
  - Lost data but use approximation to recover

Diskless Checkpointing



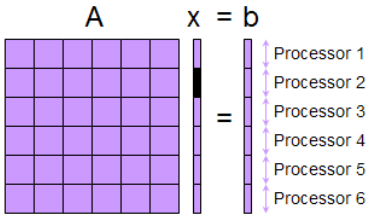
- When failure occurs:
  - control passes to user supplied handler
  - "subtraction" performed to recover missing data
  - P4 takes on role of P1
  - Execution continue

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



Lossy Algorithm : Basic Idea

- Let us assume that the exact solution of the system Ax=b is stored on different processors by rows



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

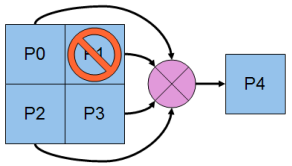
$$A_{21} x_1 + A_{22} x_2 + \ldots + A_{2n} x_n = b_2 \Rightarrow$$

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

Three Ideas for Fault To Linear Algebra Algori

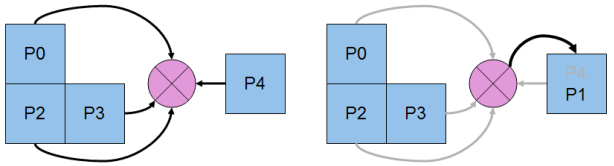
- Lossless diskless check-pointing for iterative methods
  - Checksum maintained in active processors
  - On failure, roll back to checkpoint and continue
  - No lost data
- Lossy approach for iterative methods
  - No checkpoint maintained
  - On failure, approximate missing data and carry on
  - Lost data but use approximation to recover
- Check-pointless methods for dense algorithms
  - Checksum maintained as part of computation
  - No roll back needed; No lost data

Diskless Checkpointing



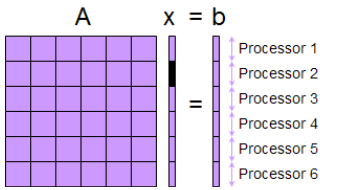
- When failure occurs:
  - control passes to user supplied handler
  - "subtraction" performed to recover missing data
  - P4 takes on role of P1
  - Execution continue

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



Lossy Algorithm : Basic Idea

- Let us assume that the exact solution of the system Ax=b is stored on different processors by rows



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

$A_{21} x_1 + A_{22} x_2 + \ldots + A_{2n} x_n = b_2 \Rightarrow$

An Example: ScaLAPACK/PBLAS Matrix Multiplication

$$\begin{pmatrix} A_{11} & \cdots & A_{1q} \\ \vdots & \cdots & \vdots \\ A_{p1} & \cdots & A_{pq} \\ \sum_{i=1}^{p} A_{i1} & \cdots & \sum_{i=1}^{p} A_{iq} \end{pmatrix} * \begin{pmatrix} B_{11} & \cdots & B_{1p} & \sum_{j=1}^{p} B_{1j} \\ \vdots & \cdots & \vdots & \vdots \\ B_{q1} & \cdots & B_{qp} & \sum_{j=1}^{p} B_{qj} \end{pmatrix}$$

$$= \begin{pmatrix} C_{11} & \cdots & C_{1p} & \sum_{j=1}^{p} C_{1j} \\ \vdots & \cdots & \vdots & \vdots \\ C_{p1} & \cdots & C_{pp} & \sum_{j=1}^{p} C_{pj} \\ \sum_{i=1}^{p} C_{i1} & \cdots & \sum_{k=1}^{p} C_{ip} & \sum_{i=1}^{p} \sum_{j=1}^{p} C_{ij} \end{pmatrix}$$

- Single failure during computation can be recovered from the checksum relationship
- By using a floating-point version Reed-Solomon code, multiple failures can be tolerated

Conclusions

- For the last decade or more, the research investment strategy has been overwhelmingly biased in favor of hardware.
- This strategy needs to be rebalanced - barriers to progress are increasingly on the software side.
- Moreover, the return on investment is more favorable to software.
  - Hardware has a half-life measured in years, while software has a half-life measured in decades.
- High Performance Ecosystem out of balance
  - Hardware, OS, Compilers, Software, Algorithms, Applications
    - No Moore's Law for software, algorithms and applications

Conclusions

- Parallelism is exploding
  - Number of cores will double every ~2 years
  - Petaflop (million processor) machines will be common in HPC by 2015
- Performance will become a software problem
  - Parallelism and locality are fundamental; can save power by pushing these to software
- Locality will continue to be important
  - On-chip to off-chip as well as node to node
  - Need to design algorithms for what counts (communication not computation)
- Massive parallelism required (including pipelining and overlap)

PLASMA Collaborators

- U Tennessee, Knoxville
  - Jack Dongarra, Julie Langou, Stan Tomov, Jakub Kurzak, Hatem Ltaief, Alfredo Buttari, Julien Langou, Piotr Luszczek, Marc Baboulin
- UC Berkeley
  - Jim Demmel, Ming Gu, W. Kahan, Beresford Parlett, Xiaoye Li, Osni Marques, Yozo Hida, Jason Riedy, Vasily Volkov, Christof Voemel, David Bindel
- Other Academic Institutions
  - UC Davis, CU Denver, Florida IT, Georgia Tech, U Maryland, North Carolina SU, UC Santa Barbara, UT Austin, LBNL
  - TU Berlin, ETH, U Electrocomm. (Japan), FU Hagen, U Carlos III Madrid, U Manchester, U Umeå, U Wuppertal, U Zagreb, UPC Barcelona, ENS Lyon, INRIA
- Industrial Partners
  - Cray, HP, Intel, Interactive Supercomputing, MathWorks, NAG, NVIDIA, Microsoft