Five Important Features to Consider When Computing at Scale

Jack Dongarra

INNOVATIVE COMPUTING LABORATORY

University of Tennessee
Oak Ridge National Laboratory
University of Manchester
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 1,000,000’s of cores things will fail
- **Communication Avoiding Algorithms**
  - For dense computations from $O(n \log p)$ to $O(\log p)$ communications
  - GMRES s-step compute ($x, Ax, A^2x, ... A^s x$)
Something’s Happening Here…

- In the “old days” it was: each year processors would become faster.
- Today the clock speed is fixed or getting slower.
- Things are still doubling every 18 - 24 months.
- Moore’s Law reinterpreted.

From K. Olukotun, L. Hammond, H. Sutter, and B. Smith
Moore’s Law Reinterpreted

- Number of cores per chip doubles every 2 years, while clock speed remains fixed or decreases
- Need to deal with systems with millions of concurrent threads
  - Future generation will have billions of threads!
    - Need to be able to easily replace inter-chip parallelism with intro-chip parallelism
- Number of threads of execution doubles every 2 years
Architecture of Interest

- Manycore chip
- Composed of hybrid cores
  - Some general purpose
  - Some graphics
  - Some floating point
Architecture of Interest

- Board composed of multiple chips sharing memory
Architecture of Interest

- Rack composed of multiple boards
Think millions of cores

A room full of these racks

Architecture of Interest
A New Generation of Software:

<table>
<thead>
<tr>
<th>Software/Algorithms follow hardware evolution in time</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINPACK (70’s)</td>
</tr>
<tr>
<td>(Vector operations)</td>
</tr>
<tr>
<td>Rely on</td>
</tr>
<tr>
<td>- Level-1 BLAS operations</td>
</tr>
</tbody>
</table>

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.
A New Generation of Software: A New Generation of Software:

Software/Algorithms follow hardware evolution in time

- **LINPACK (70's)** (Vector operations)
  - Level-1 BLAS operations
- **LAPACK (80's)** (Blocking, cache friendly)
  - Level-3 BLAS operations
- **ScaLAPACK (90's)** (Distributed Memory)
  - PBLAS Message Passing
- **PLASMA (00's)** New Algorithms (many-core friendly)
  - 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 lot 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.
### A New Generation of Software:

Software/Algorithms follow hardware evolution in time

<table>
<thead>
<tr>
<th>Software/Algorithms</th>
<th>Hardware Evolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINPACK (70’s) (Vector operations)</td>
<td>Rely on Level-1 BLAS operations</td>
</tr>
<tr>
<td>LAPACK (80’s) (Blocking, cache friendly)</td>
<td>Rely on Level-3 BLAS operations</td>
</tr>
<tr>
<td>ScaLAPACK (90’s) (Distributed Memory)</td>
<td>Rely on PBLAS Mess Passing</td>
</tr>
</tbody>
</table>

Those new algorithms have a very low granularity and scale very well (multicore, petascale computing, …) removing a lot of dependencies among the tasks (multicore, distributed computing) and avoiding latency (distributed computing, out-of-core). They rely on fast kernels.
## A New Generation of Software: Parallel Linear Algebra Software for Multicore Architectures (PLASMA)

Software/Algorithms follow hardware evolution in time

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

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.
Coding for an Abstract Multicore

Parallel software for multicores should have two characteristics:

- **Fine granularity:**
  - High level of parallelism is needed
  - Cores will probably be associated with relatively small local memories. This requires splitting an operation into tasks that operate on small portions of data in order to reduce bus traffic and improve data locality.

- **Asynchronicity:**
  - As the degree of thread level parallelism grows and granularity of the operations becomes smaller, the presence of synchronization points in a parallel execution seriously affects the efficiency of an algorithm.
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

LAPACK

ScaLAPACK

PBLAS

BLACS

Message Passing (MPI, PVM, ...)
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)

Time for each component

DGETF2  DLASWP(L)  DLASWP(R)  DTRSM  DGEMM

Threads – no lookahead

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.

[Diagram showing differences between Blocked and Column-Major data access patterns]
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 Threads

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

8 Core Experiments
Cholesky on the CELL

- **1 CELL (8 SPEs)**
  - 186 Gflop/s
  - 91 % peak
  - 97 % SGEMM peak
- **2 CELLs (16 SPEs)**
  - 365 Gflop/s
  - 89 % peak
  - 95 % SGEMM peak

Single precision results on the Cell
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
DAG and Scheduling

- DAG is dynamically generated and implicit
- Look at executing a sliding window of the DAG.
- Everything designed for distributed memory systems
- Runtime system on each node or core

- Run time
  - Bin 1
    - See if new data has arrived
  - Bin 2
    - See if new dependences are satisfied
    - If so move task to Bin 3
  - Bin 3
    - Exec a task that’s ready
    - Notify children of completion
    - Send data to children
    - If no work do work stealing
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
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

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

<table>
<thead>
<tr>
<th>Processor</th>
<th>Size</th>
<th>SGEMM/ DGEMM</th>
<th>Size</th>
<th>SGEMV/ DGEMV</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Opteron 246</td>
<td>3000</td>
<td>2.00</td>
<td>5000</td>
<td>1.70</td>
</tr>
<tr>
<td>UltraSparc-Ile</td>
<td>3000</td>
<td>1.64</td>
<td>5000</td>
<td>1.66</td>
</tr>
<tr>
<td>Intel PIII Coppermine</td>
<td>3000</td>
<td>2.03</td>
<td>5000</td>
<td>2.09</td>
</tr>
<tr>
<td>PowerPC 970</td>
<td>3000</td>
<td>2.04</td>
<td>5000</td>
<td>1.44</td>
</tr>
<tr>
<td>Intel Woodcrest</td>
<td>3000</td>
<td>1.81</td>
<td>5000</td>
<td>2.18</td>
</tr>
<tr>
<td>Intel XEON</td>
<td>3000</td>
<td>2.04</td>
<td>5000</td>
<td>1.82</td>
</tr>
<tr>
<td>Intel Centrino Duo</td>
<td>3000</td>
<td>2.71</td>
<td>5000</td>
<td>2.21</td>
</tr>
</tbody>
</table>
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.
Iterative refinement for dense systems, $Ax = b$, can work this way.

\[ L U = lu(A) \quad O(n^3) \]
\[ x = L\backslash(U\backslash b) \quad O(n^2) \]
\[ r = b - Ax \quad O(n^2) \]
\[
\text{WHILE } ||r|| \text{ not small enough}
\]
\[ z = L\backslash(U\backslash r) \quad O(n^2) \]
\[ x = x + z \quad O(n^1) \]
\[ r = b - Ax \quad O(n^2) \]
\[
\text{END}
\]

Wilkinson, Moler, Stewart, & Higham provide error bound for SP fl pt results when using DP fl pt.
Iterative refinement for dense systems, $Ax = b$, can work this way.

\[
\begin{align*}
L U &= \text{lu}(A) & \text{SINGLE} & O(n^3) \\
x &= L\backslash(U\backslash b) & \text{SINGLE} & O(n^2) \\
r &= b - Ax & \text{DOUBLE} & O(n^2) \\
\text{WHILE } || r || \text{ not small enough} & \\
z &= L\backslash(U\backslash r) & \text{SINGLE} & O(n^2) \\
x &= x + z & \text{DOUBLE} & O(n^1) \\
r &= b - Ax & \text{DOUBLE} & O(n^2) \\
\text{END}
\end{align*}
\]

- 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 $Ax = b$

- 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

<table>
<thead>
<tr>
<th>Architecture (BLAS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Intel Pentium III Coppermine (Goto)</td>
</tr>
<tr>
<td>2  Intel Pentium III Katmai (Goto)</td>
</tr>
<tr>
<td>3  Sun UltraSPARC IIe (Sunperf)</td>
</tr>
<tr>
<td>4  Intel Pentium IV Prescott (Goto)</td>
</tr>
<tr>
<td>5  Intel Pentium IV-M Northwood (Goto)</td>
</tr>
<tr>
<td>6  AMD Opteron (Goto)</td>
</tr>
<tr>
<td>7  Cray X1 (libsci)</td>
</tr>
<tr>
<td>8  IBM Power PC G5 (2.7 GHz) (VecLib)</td>
</tr>
<tr>
<td>9  Compaq Alpha EV6 (CXML)</td>
</tr>
<tr>
<td>10 IBM SP Power3 (ESSL)</td>
</tr>
<tr>
<td>11 SGI Octane (ATLAS)</td>
</tr>
</tbody>
</table>
Results for Mixed Precision Iterative Refinement for Dense $Ax = b$

- 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

<table>
<thead>
<tr>
<th>Architecture (BLAS)</th>
<th># procs</th>
<th>$n$</th>
<th>DP Solve/SP Solve</th>
<th>DP Solve/Iter Ref</th>
<th># iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Opteron (Goto – OpenMPI MX)</td>
<td>32</td>
<td>22627</td>
<td>1.85</td>
<td>1.79</td>
<td>6</td>
</tr>
<tr>
<td>AMD Opteron (Goto – OpenMPI MX)</td>
<td>64</td>
<td>32000</td>
<td>1.90</td>
<td>1.83</td>
<td>6</td>
</tr>
</tbody>
</table>
Sparse Direct Solver and Iterative Refinement

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

Tim Davis's Collection, n=100K - 3M
Sparse Iterative Methods (PCG)

- **Outer/Inner Iteration**
  
  Outer iterations using 64 bit floating point
  
  Compute \( r^{(0)} = b - A x^{(0)} \) for some initial guess \( x^{(0)} \)

  for \( i = 1, 2, \ldots \)
  
  solve \( M z^{(i-1)} = r^{(i-1)} \)
  
  \( \rho_{i-1} = r^{(i-1)^T} z^{(i-1)} \)
  
  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
  
  \( q^{(i)} = A p^{(i)} \)
  
  \( \alpha_i = \rho_{i-1}/p^{(i)^T} q^{(i)} \)
  
  \( x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)} \)
  
  \( r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)} \)
  
  check convergence; continue if necessary

end

- **Inner iteration**
  
  In 32 bit floating point

  Compute \( r^{(0)} = b - A x^{(0)} \) for some initial guess \( x^{(0)} \)

  for \( i = 1, 2, \ldots \)
  
  solve \( M z^{(i-1)} = r^{(i-1)} \)
  
  \( \hat{p}_{i-1} = r^{(i-1)^T} z^{(i-1)} \)
  
  if \( i = 1 \)
  
  \( p^{(1)} = z^{(0)} \)
  
  else
  
  \( \beta_{i-1} = \hat{p}_{i-1}/\hat{p}_{i-2} \)
  
  \( p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)} \)
  
  endif
  
  \( q^{(i)} = A p^{(i)} \)
  
  \( \alpha_i = \hat{p}_{i-1}/p^{(i)^T} q^{(i)} \)
  
  \( x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)} \)
  
  \( r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)} \)
  
  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\textsuperscript{2}, GMRES\textsuperscript{2}, PCG\textsuperscript{2}, and PGMRES\textsuperscript{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
## Characteristics of multiplier on an FPGA* (using DSP48)

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

* XC4LX160-10
Mixed Precision Iterative Refinement

- Random Matrix Test - Junqing Sun et al

Refinement iterations for customized formats (sXXe11).

Random matrices

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Mantissa Bits</th>
<th>12</th>
<th>16</th>
<th>23</th>
<th>31</th>
<th>48</th>
<th>52</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td></td>
<td>8.9</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td></td>
<td>11.1</td>
<td>5.1</td>
<td>2.1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td></td>
<td>19.7</td>
<td>6.1</td>
<td>2.5</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td></td>
<td>28</td>
<td>6.3</td>
<td>2.6</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td></td>
<td>-</td>
<td>9.3</td>
<td>3</td>
<td>1.3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td></td>
<td>-</td>
<td>13.3</td>
<td>3.1</td>
<td>1.43</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

More Bits

More Iterations
Mixed Precision Hybrid Direct Solver

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

LU w Partial Pivoting using variable precision on an FPGA

Data types for LU on FPGAs

High Performance Mixed-Precision Linear Solver for FPGAs

Junqing Sun, Gregory D. Peterson, Olaf Storaasli, To appear IEEE JPDC

* For a 128x128 matrix
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.

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

Correction = \(-A\backslash(b - Ax)\)
Fault Tolerance

- **Trends in HPC:**
  - High end systems with thousands 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/](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

1+1
2+2
3+3
4+4

Error Problem

4 processors available

1+1
2+2
3+3
4+4

4 processors available
Erasure Problem

1+1
P1

3+3
P2

2+2
P3

4+4
P4

4 processors available
Lost processor 2

Error Problem

4 processors available
Processor 2 returns an incorrect result

1+1
P1

2+2
P2

3+3
P3

4+4
P4

1+1
P1

2
P2

6
P3

8
P4

2
P1

5
P2

6
P3

8
P4
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

**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.
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

- 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

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 $A_{21}, A_{22}, ..., A_{2k}$ and $b_2$ (the original data) on the failed processor

Step 3: Notice that

$$A_{21} x_1 - A_{22} x_2 + ... + A_{2k} x_k = b_2 = x_2 = A_{21}^{-1} (b_2 - \sum_{i=2}^{k} A_{2i} x_i)$$
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

- **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 continues

**Lossy Algorithm: Basic Idea**

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

**An Example: ScalAPACK/PBLAS Matrix Multiplication**

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

\[
A_F = \begin{pmatrix}
A & A_{CR} \\
C'^T A & C'^T A C_R
\end{pmatrix}
\quad \text{and} \quad
B_F = \begin{pmatrix}
B & B_{CR} \\
C'^T B & C'^T B C_R
\end{pmatrix}
\]

\[
\begin{pmatrix}
A \\
C'^T A
\end{pmatrix}
\begin{pmatrix}
B & B_{CR}
\end{pmatrix} = \begin{pmatrix}
A B & A B C_R \\
C'^T A B & C'^T A B C_R
\end{pmatrix} = (A B)_F
\]

ABFT-PDGEMM

The overhead:
- 2p-1 extra processes for $p^2$
- one extra process need to receive the data for the rows and columns

Conclusion: a very scalable approach, more processors means less overhead

$$A_F = \begin{pmatrix} A & AC_R \\ C_T^TA & C_T^TAC_R \end{pmatrix}$$ and $$B_F = \begin{pmatrix} B & BC_R \\ C_T^T B & C_T^T BC_R \end{pmatrix}$$

$$\begin{pmatrix} A \\ C_T^TA \end{pmatrix}(B \ BC_R) = \begin{pmatrix} AB \\ C_T^T AB \end{pmatrix} = (AB)_F$$
• Processor type Opteron 2.2 GHz
• Processor theoretical peak 4.4 GFlops/sec
• Number of application processors 712
• System theoretical peak (computational nodes) 3.13 TFlops/sec
• Number of shared-memory application nodes 356
• Processors per node 2
• Physical memory per node 6 GBytes
• Usable memory per node 3-5 GBytes
• Switch Interconnect InfiniBand
• Switch MPI Unidirectional Latency 4.5 μsec
• Switch MPI Unidirectional Bandwidth (peak) 620 MB/s
• Global shared disk GPFS Usable disk space 30 TBytes
• Batch system PBS Pro
FT-PDGEMM -- nloc=4,000
Weak Scaling
Performance modeling

![Graph showing the performance of two models, Model SUMMA and Model ABFT, across varying numbers of processors. The graph includes measured data points for SUMMA and ABFT.](image-url)
Strong scalability

![Graph showing strong scalability with different nloc values and GFLOPs/sec/proc performance across different numbers of processors. The graph illustrates the decline in performance as the number of processors increases.]
Strong scalability

ABFT represents the only known alternative to address fault tolerance in strong scalability.
ABFT represents the only known alternative to address fault tolerance in strong scalability
ABFT summary

- Relies on floating-point arithmetic
- Exploit the checksum processor
- Stable algorithms exist for many linear algebra operation:
  - AXPY, SCAL (BLAS1)
  - GEMV (BLAS2)
  - GEMM (BLAS3)
  - LU, QR, Cholesky (LAPACK)
  - FFT
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
MPI Dominates Petascale Communication Survey top HPC open science applications

<table>
<thead>
<tr>
<th>Language/Prog Models</th>
<th>RT Dev</th>
<th>I/O</th>
<th>Math</th>
<th>Sys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran77</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Fortran90</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Fortran95</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C++</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CAF</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>OpenMP</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MPI</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MPI2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Global Arrays</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Python</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Abinit</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>BLACS</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>boost</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CHIMERA</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>POP/CICE</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SD3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>GTC</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MADNESS</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AORS3A</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LAMMPS</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FLAPW</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Mil/Chroma</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CPFLOTRAH</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>QOBOX</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CAM</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CCSD</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>T3P</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>VASP</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>NEWTRIX</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>HWChem</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>OReTran</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CASINO</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Must have
- Can use