ManyCore Computing: The Impact on Numerical Software for Linear Algebra Libraries

Jack Dongarra
INNOVATIVE COMPUTING LABORATORY
University of Tennessee
Oak Ridge National Laboratory
University of Manchester

11/20/2007
Performance Projection
Top500 Data

- N=1
- N=500
### Possible Petascale System

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. # of cores per nodes</td>
<td>10 - 100 cores</td>
</tr>
<tr>
<td>2. Performance per nodes</td>
<td>100 - 1,000 GFlop/s</td>
</tr>
<tr>
<td>3. Number of nodes</td>
<td>1,000 - 10,000 nodes</td>
</tr>
<tr>
<td>4. Latency inter-nodes</td>
<td>1 μsec</td>
</tr>
<tr>
<td>5. Bandwidth inter-nodes</td>
<td>10 GB/s</td>
</tr>
<tr>
<td>6. Memory per nodes</td>
<td>10 GB</td>
</tr>
</tbody>
</table>

- **Part I:** First rule in linear algebra: Have an efficient DGEMM
  - Motivation in:
    2. performance per node 5. bandwidth inter-nodes 6. memory per nodes
- **Part II:** Algorithms for multicore and latency avoiding algorithms for LU, QR ...
  - Motivation in:
    1. Number of cores per node 2. performance per node 4. Latency inter-nodes
- **Part III:** Algorithms for fault tolerance
  - Motivation in:
    1. Number of cores per node 3. number of nodes
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
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 TLP grows and granularity of the operations becomes smaller, the presence of synchronization points in a parallel execution seriously affects the efficiency of an algorithm.
ManyCore - Parallelism for the Masses

- We are looking at the following concepts in designing the next numerical library implementation
  - Dynamic Data Driven Execution
  - Self Adapting
  - Block Data Layout
  - Mixed Precision in the Algorithm
  - Exploit Hybrid Architectures
  - Fault Tolerant Methods
## A New Generation of Software:

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

Those new algorithms have a very low granularity, they scale very well (multicore, distributed computing) and removes a lot of dependencies among the tasks (multicore, distributed computing) and rely on fast kernels.
# A New Generation of Software: Parallel Linear Algebra Software for Multicore Architectures (PLASMA)

Algorithms follow hardware evolution in time

<table>
<thead>
<tr>
<th>LINPACK (70’s)</th>
<th>Rely on</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Vector operations)</td>
<td>- Level-1 BLAS</td>
</tr>
<tr>
<td></td>
<td>operations</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LAPACK (80’s)</th>
<th>Rely on</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Blocking, cache</td>
<td>- Level-3 BLAS</td>
</tr>
<tr>
<td>friendly)</td>
<td>operations</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ScALAPACK (90’s)</th>
<th>Rely on</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Distributed Memory)</td>
<td>- PBLAS Mess Passing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PLASMA (00’s)</th>
<th>Rely on</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Algorithms</td>
<td>- a DAG/scheduler</td>
</tr>
<tr>
<td>(many-core friendly)</td>
<td>- block data layout</td>
</tr>
</tbody>
</table>

These 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.
Developing Parallel Algorithms

- LAPACK
  - Threaded BLAS
  - PThreads
  - OpenMP

parallelism

Parallel algorithms can be developed using various techniques, including:

- Threaded BLAS
- PThreads
- OpenMP

Sequential BLAS

LAPACK
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 core system)

Threads – no lookahead

Time for each component

Bulk Sync Phases

- DGETF2
- DLASWP(L)
- DLASWP(R)
- DTRSM
- DGEMM
Adaptive Lookahead - Dynamic

Event Driven Multithreading

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
Fork-Join vs. Dynamic Execution

Experiments on Intel’s Quad Core Clovertown with 2 Sockets w/ 8 Treads
Fork-Join vs. Dynamic Execution

Fork-Join – parallel BLAS

DAG-based – dynamic scheduling

Experiments on Intel’s Quad Core Clovertown with 2 Sockets w/ 8 Threads
Achieving Asynchronicity

The matrix factorization can be represented as a DAG:
  • **nodes**: tasks that operate on “tiles”
  • **edges**: dependencies among tasks

Tasks can be scheduled asynchronously and in any order as long as dependencies are not violated.
A critical path can be defined as the shortest path that connects all the nodes with the higher number of outgoing edges.

Priorities:
Achieving Asynchronicity

- Very fine granularity
- Few dependencies, i.e., high flexibility for the scheduling of tasks \(\rightarrow\) asynchronous scheduling
- No idle times
- Some degree of adaptativeness
- Better locality thanks to block data layout
Cholesky Factorization
DAG-based Dependency Tracking

Dependencies expressed by the DAG are enforced on a tile basis:
- fine-grained parallelization
- flexible scheduling
Cholesky on the IBM Cell

Pipelining:
- Between loop iterations.

Double Buffering:
- Within BLAS,
- Between BLAS,
- Between loop iterations.

Result:
- Minimum load imbalance,
- Minimum dependency stalls,
- Minimum memory stalls
  (no waiting for data).

Achieves 174 Gflop/s; 85% of peak in SP.
Cholesky - Using 2 Cell Chips

SPOTRF - QS20 - 2 CELL BEs

Gflop/s

Size

QS20 Peak
SGEMM Peak
SPOTRF
Parallelism in LAPACK: Blocked Storage

Column-Major
Parallelism in LAPACK: Blocked Storage
Parallelism in LAPACK: blocked storage

Column-Major

Blocked
Parallelism in LAPACK: Blocked Storage

The use of blocked storage can significantly improve performance.

Blocking Speedup

<table>
<thead>
<tr>
<th>block size</th>
<th>DGEMM speedup</th>
<th>DTRSM speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>1.8</td>
<td>1.6</td>
</tr>
<tr>
<td>128</td>
<td>1.6</td>
<td>1.4</td>
</tr>
<tr>
<td>256</td>
<td>1.2</td>
<td>1.2</td>
</tr>
</tbody>
</table>
Multicore Friendly Algorithms

QR Factorization -- 2-socket Clovertown
(Peak 85.12 Gflop/s)

![Graph showing performance of different algorithms for QR Factorization on 2-socket Clovertown.](image-url)
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)

8 Core Experiments

Problems Size vs. Mflop/s
With the Hype on Cell & PS3 We Became Interested

- The PlayStation 3's CPU based on a "Cell" processor
- Each Cell contains a Power PC processor and 8 SPEs. (SPE is processing unit, SPE: SPU + DMA engine)
  - An SPE is a self contained vector processor which acts independently from the others.
    - 4 way SIMD floating point units capable of a total of 25.6 Gflop/s @ 3.2 GHZ
  - 204.8 Gflop/s peak!
- The catch is that this is for 32 bit floating point; (Single Precision SP)
- And 64 bit floating point runs at 14.6 Gflop/s total for all 8 SPEs!!
  - Divide SP peak by 14; factor of 2 because of DP and 7 because of latency issues

SPE ~ 25 Gflop/s peak
Moving Data Around on the Cell

Worst case memory bound operations (no reuse of data)
3 data movements (2 in and 1 out) with 2 ops (SAXPY)
For the cell would be 4.6 Gflop/s (25.6 GB/s*2ops/12B)
32 or 64 bit Floating Point Precision?

- A long time ago 32 bit floating point was used
  - Still used in scientific apps but limited
- Most apps use 64 bit floating point
  - Accumulation of round off error
    - A 10 TFlop/s computer running for 4 hours performs > 1 Exaflop (10^{18}) ops.
  - Ill conditioned problems
  - IEEE SP exponent bits too few (8 bits, 10^{±38})
  - Critical sections need higher precision
    - Sometimes need extended precision (128 bit fl pt)
  - However some can get by with 32 bit fl pt in some parts
- Mixed precision a possibility
  - Approximate in lower precision and then refine or improve solution to high precision.
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.

\[
\begin{align*}
L U &= lu(A) \\
x &= L\backslash(U\backslash b) \\
r &= b - Ax
\end{align*}
\]

\[
\text{WHILE } || r || \text{ not small enough} \\
z &= L\backslash(U\backslash r) \\
x &= x + z \\
r &= b - Ax
\]

\[
\text{END}
\]

- 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.

\[
\begin{align*}
L U &= lu(A) \\
x &= L^{-1}(U^{-1}b) \\
r &= b - Ax \\
\text{WHILE} \quad ||r|| \text{ not small enough} \\
z &= L^{-1}(U^{-1}r) \\
x &= x + z \\
r &= b - Ax
\end{align*}
\]

- 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 $Ax = b$

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

- 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
IBM Cell 3.2 GHz, $Ax = b$

8 SGEMM (Embarrassingly Parallel)

- SP Peak (204 Gflop/s)
- SP $Ax=b$ IBM
- DP Peak (15 Gflop/s)
- DP $Ax=b$ IBM

Matrix Size

GFlop/s

0 50 100 150 200 250

0 500 1000 1500 2000 2500 3000 3500 4000 4500

.30 secs

3.9 secs
IBM Cell 3.2 GHz, $Ax = b$

- SP Peak (204 Gflop/s)
- DP Peak (15 Gflop/s)
- SP $Ax=b$ IBM
- DSGESV
- DP $Ax=b$ IBM

Matrix Size vs. GFlop/s

- 8 SGEMM (Embarrassingly Parallel)
- 8.3X

- 0.30 secs
- 0.47 secs
- 3.9 secs
Cholesky on the Cell, $Ax=b$, $A=AT$, $x^TAx > 0$

- Single precision performance
- Mixed precision performance using iterative refinement
  Method achieving 64 bit accuracy

For the SPE's standard C code and C language SIMD extensions (intrinsics)
Sparse Linear Algebra

- **Computational speed doesn't matter**
  - Peak 204 Gflop/s
- **Memory bus matters**
  - 25 GB/s = 12 Gflop/s
    - Assuming matrix read from memory
  - In practice ~6 Gflop/s
    - In SP using 8 SPEs
What About That PS3?

PowerPC

- 25.6 Gflop/s
- 25.6 Gflop/s
- 25.6 Gflop/s
- 25.6 Gflop/s
- 200 GB/s
- 25 GB/s
- 512 MiB

- 3.2 GHz
- 25 GB/s injection bandwidth
- 200 GB/s between SPEs
- 32 bit peak perf 8*25.6 Gflop/s
- 204.8 Gflop/s peak
- 64 bit peak perf 8*1.8 Gflop/s
- 14.6 Gflop/s peak
- 512 MiB memory
**PS3 Hardware Overview**

- **PowerPC**
- **256 MiB memory**
- **25 GB/s injection bandwidth**
- **200 GB/s between SPEs**
- **32 bit peak perf 6*25.6 Gflop/s**
- **64 bit peak perf 6*1.8 Gflop/s**
- **1 Gb/s NIC**

---

**PE**

- **25.6 Gflop/s**
- **25.6 Gflop/s**
- **25.6 Gflop/s**

---

**GameOS Hypervisor**

- **Disabled/Broken: Yield issues**

---

**SIT CELL**

- **200 GB/s**

---

**3.2 GHz**

- **25 GB/s injection bandwidth**
- **200 GB/s between SPEs**
- **64 bit peak perf 6*1.8 Gflop/s**
- **10.8 Gflop/s peak**

---

**nVIDIA**

- **256 MiB**

---

[Images and diagrams related to hardware components and performance metrics]
HPC in the Living Room
Matrix Multiple on a 4 Node PlayStation3 Cluster

What's good
- Very cheap: ~4$ per Gflop/s (with 32 bit fl pt theoretical peak)
- Fast local computations between SPEs
- Perfect overlap between communications and computations is possible (Open-MPI running):
  - PPE does communication via MPI
  - SPEs do computation via SGEMMs

What's bad
- Gigabit network card. 1 Gb/s is too little for such computational power (150 Gflop/s per node)
- Linux can only run on top of GameOS (hypervisor)
  - Extremely high network access latencies (120 usec)
  - Low bandwidth (600 Mb/s)
- Only 256 MB local memory
- Only 6 SPEs

33 Time
Gold: Computation: 8 ms
Blue: Communication: 20 ms
SUMMA on a 2x2 PlayStation3 cluster

SUMMA -- Model vs Measured 6 SPEs

Problem size vs Gflop/s graph showing the comparison between model predictions and measured values for 6 SPEs.
SUMMA on a 2x2 PlayStation3 cluster

SUMMA -- Model vs Measured 6 SPEs

- Model 6 SPEs
- Measures 6 SPEs

problem size vs. Gflop/s

memory limit
• SCOP3: A Rough Guide to Scientific Computing on the PlayStation 3
• See webpage for details
How to Deal with Complexity?

• Adaptivity is the key for applications to effectively use available resources whose complexity is exponentially increasing.

• Goal:
  ▪ Automatically bridge the gap between the application and computers that are rapidly changing and getting more and more complex.
Self-Adapting Software

• Variation
  ▪ Many different algorithm implementation are generated automatically and tested for performance

• Selection
  ▪ The best performing implementation is sought by optimization
Self-Adapting Software

Huge search space (algorithms, parameters, ...)

Generate + Adapt (once per target) → Use (often)

Variation    Selection

Automatic Performance Tuning
Self-Adapting Software

- Automatically generated HW adapted libraries
- Large sections of straight-line code produced

Examples

- Numerical linear algebra: ATLAS, OSKI
- Discrete Fourier transforms: FFTW
- Digital signal processing: SPIRAL
- MPI Collectives (UCB, UTK): FT-MPI
Generic Code Optimization

- Can ATLAS-like techniques be applied to arbitrary code?
- What do we mean by ATLAS-like techniques?
  - Blocking
  - Loop unrolling
  - Data prefetch
  - Functional unit scheduling
  - etc.
- Referred to as empirical optimization
  - Generate many variations
  - Pick the best implementation by measuring the performance
Applying Self Adapting Software

- **Numerical and Non-numerical applications**
  - BLAS like ops / message passing collectives
- **Static or Dynamic determine code to be used**
  - Perform at make time / every time invoked
- **Independent or dependent on data presented**
  - Same on each data set / depends on properties of data
Multi, Many, ..., Many-More

- **Parallelism for the masses**
- **Multi, Many, Many-MoreCore** are here and coming fast
- **Our approach for numerical libraries:**
  - Use Dynamic DAG based scheduling
  - Minimize sync - Non-blocking communication
  - Maximize locality - Block data layout
- **Autotuners should take on a larger, or at least complementary, role to compilers in translating parallel programs.**
- **What’s needed is a long-term, balanced investment in hardware, software, algorithms and applications in the HPC Ecosystem.**
Collaborators / Support

Alfredo Buttari, UTK
Julien Langou, UColorado
Julie Langou, UTK
Piotr Luszczek, MathWorks
Jakub Kurzak, UTK
Stan Tomov, UTK