Scientific Libraries: MPI + Manycore Issues and Plans

Dense Numerical Linear Algebra Library at Scale

Jack Dongarra

University of Tennessee
Oak Ridge National Laboratory

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

parallelism

Threaded BLAS

PThreads

OpenMP

ScaLAPACK

PBLAS

Global

Local

LAPACK

parallelism

Threaded BLAS

BLACS

PThreads

OpenMP

Mess Passing (MPI , PVM, ...)

About 1 million lines of code

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.

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 / Auto Tuning
  - Block Data Layout
  - Mixed Precision in the Algorithm
  - Exploit Hybrid Architectures
  - Fault Tolerant Methods
  - Communication Avoiding Algorithms

Steps in the LAPACK LU

DGETF2 (Factor a panel)	LAPACK
DLSWP (Backward swap)	LAPACK
DLSWP (Forward swap)	LAPACK
DTRSM (Triangular solve)	**BLAS**
DGEMM (Matrix multiply)	**BLAS**

LU Timing Profile (4 core system)

Threads – no lookahead

Time for each component →

DGETF2	
DLASWP(L)	
DLASWP(R)	
DTRSM	
DGEMM	

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

Redesign

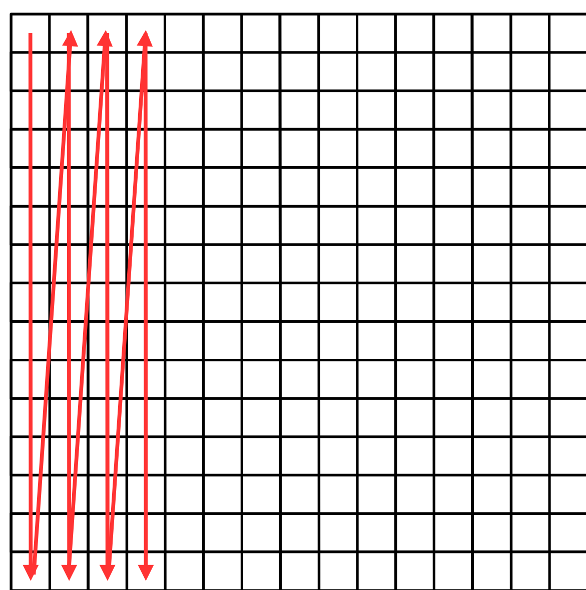
- Asychronicity
  - Avoid fork-join (Bulk sync design)
- Dynamic Scheduling
  - Out of order execution
- Fine Granularity
  - Independent block operations
- Locality of Reference
  - Data storage – Block Data Layout

33

Achieving Fine Granularity

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

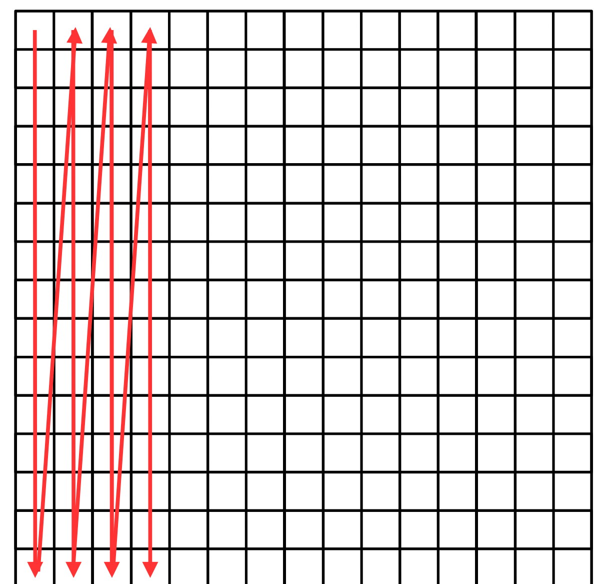
Column-Major



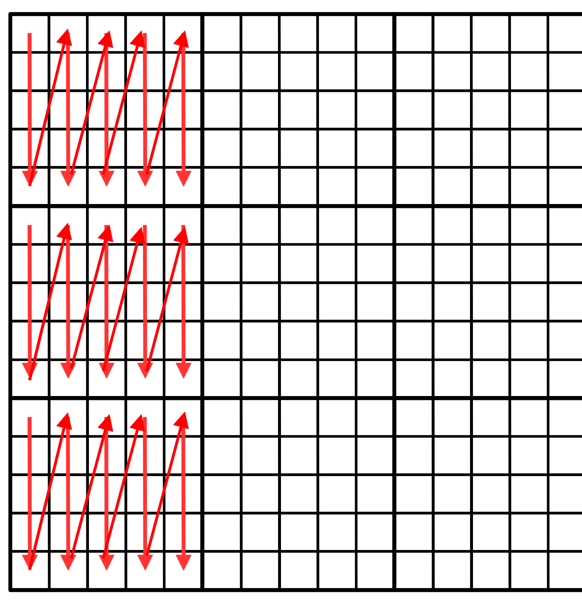
Achieving Fine Granularity

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

Column-Major



Blocked



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)

45000
40000
35000
30000
25000
20000
15000
10000
5000
0

Mflop/s

8 Core Experiments

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000 11000 12000 13000

Problems Size

P P P P
c c c c
BUS
L2 Cache

P P P P
c c c c
BUS
L2 Cache

LU -- 8-way dual Opteron -- MKL-9.1

LAPACK
MKL-9.1
Tiled+asynch.

Gflop/s

0
10
20
30
40
50
60
70

2000
4000
6000
8000
10000
12000

problem size

L2 Cache
BUS

Cholesky -- quad-socket, dual-core Opteron

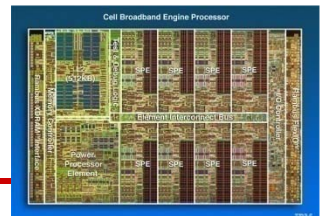
PLASMA & ACML BLAS
ACML Cholesky 4.0.0
MKL Cholesky 9.1
LAPACK & ACML BLAS
DGEMM

Gflop/s
70
60
50
40
30
20
10
0

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000
problem size

QR -- quad-socket, dual-core Opteron

Cholesky on the CELL



Cholesky -- CELL Processor

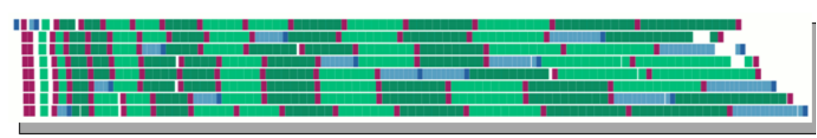
UT Tile Cholesky -- one CELL (8 SPEs)
UT Tile Cholesky -- two CELLs (16 SPEs)
LAPACK & IBM CELL BLAS

Gflop/s: 0, 50, 100, 150, 200, 250, 300, 350, 400

problem size: 0, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000

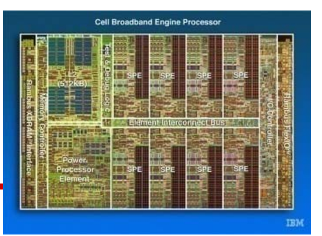
- 1 CELL (8 SPEs)
  - 186 Gflop/s
  - 91 % peak
  - 97 % SGEMM peak
- 2 CELLs (16 SPEs)
  - 365 Gflop/s
  - 89 % peak
  - 95 % SGEMM peak

- CELL Cholesky – 16 cores
- CELL Cholesky – 8 cores



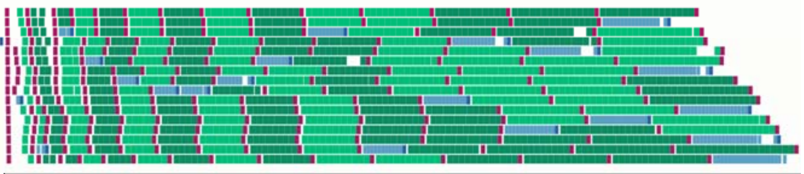
Single precision results on the Cell

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

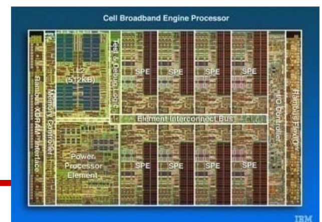
◆ CELL Cholesky – 16 cores



◆ CELL Cholesky – 8 cores

Single precision results on the Cell

Cholesky on the CELL



Cholesky -- CELL Processor

- UT Tile Cholesky -- two CELLs (16 SPEs)
- UT Tile Cholesky -- one CELL (8 SPEs)
- LAPACK & IBM CELL BLAS

Gflop/s: 0, 50, 100, 150, 200, 250, 300, 350, 400

problem size: 0, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000

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

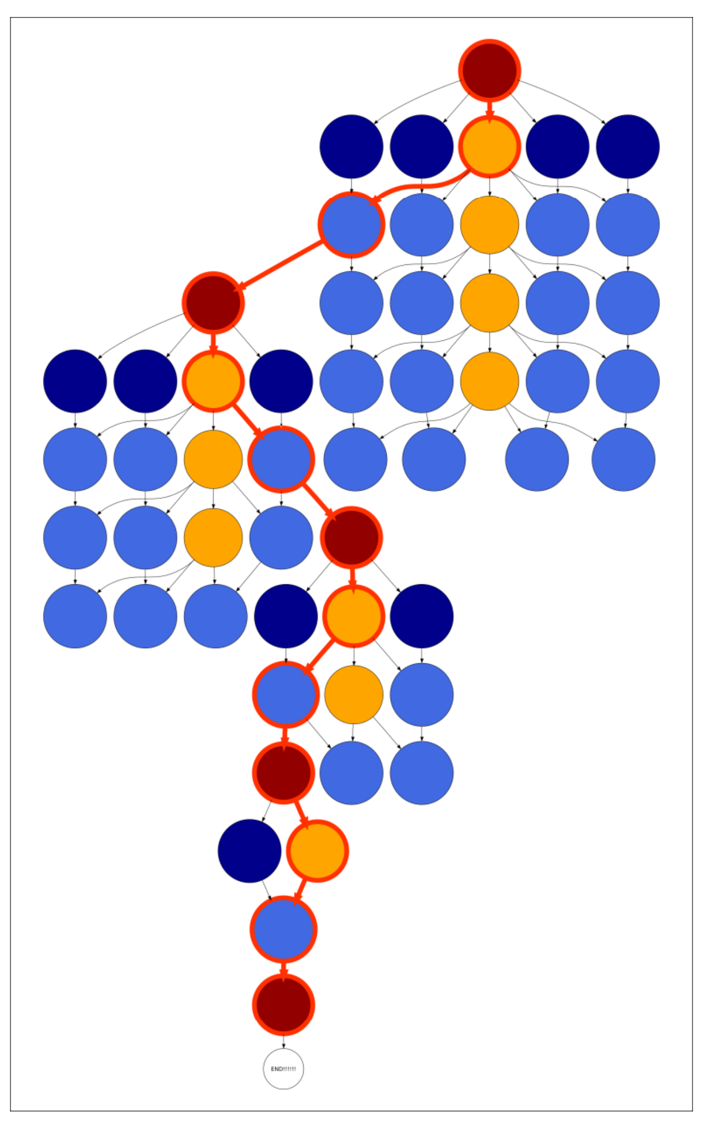
- CELL Cholesky – 16 cores

- CELL Cholesky – 8 cores

Single precision results on the Cell

If We Had A Small Matrix Problem

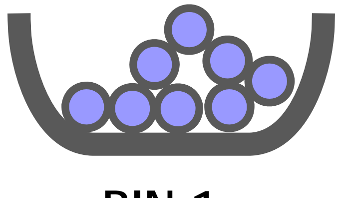
- We would generate the DAG, find the critical path and execute it.
- DAG too large to generate ahead of time
  - Not explicitly generate
  - Dynamically generate the DAG as we go
- Machines will have large number of cores in a distributed fashion
  - Will have to engage in message passing
  - Distributed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



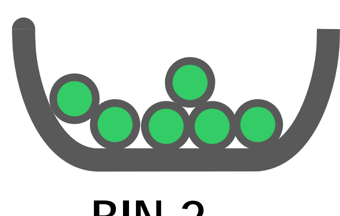
The DAGs are Large

- 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

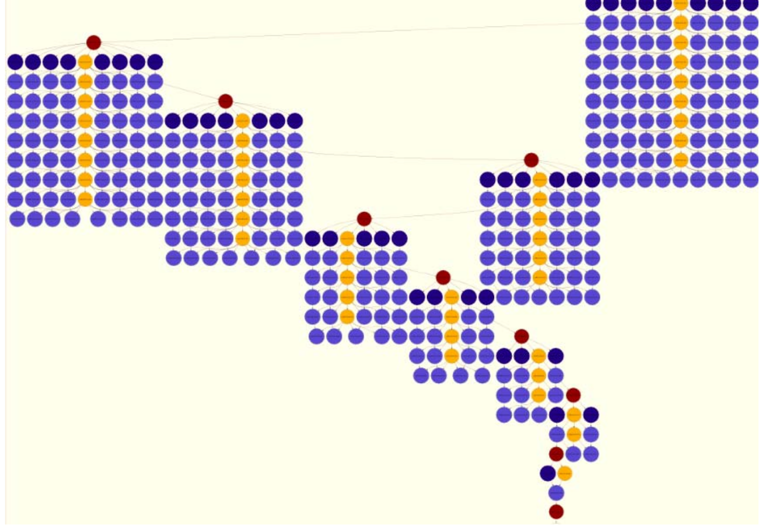
Each Node or Core Will Have A Run Time System

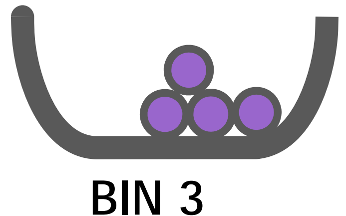


- some dependencies satisfied
- waiting for all dependencies



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





- all data delivered
- waiting for execution

DAG and Scheduling

- DAG is dynamically generated and implicit
- 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.
- What about work stealing?
  - Can we do better than nearest neighbor work stealing?
- What does the program look like?
  - Experimenting with Cilk, Charm++, UPC, Intel Threads
  - I would like to reuse as much of the existing software as possible

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