

Distributed Dense Linear Algebra on Heterogeneous Architectures

George Bosilca
bosilca@eecs.utk.edu

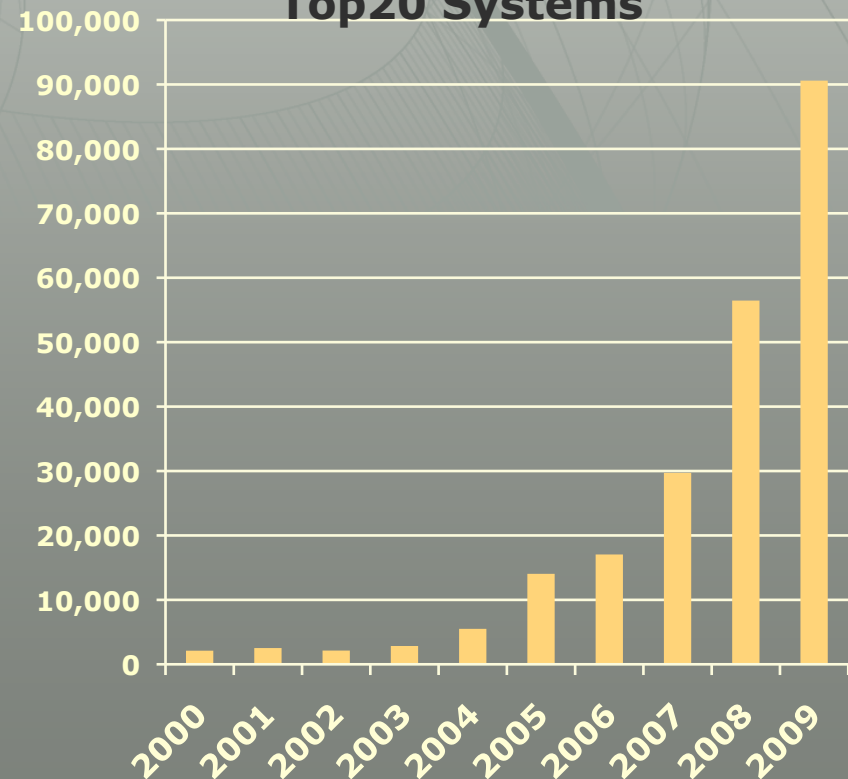


Innovative Computing Laboratory
COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF TENNESSEE

Factors that Necessitate to Redesign of Our Software

- » **Steepness of the ascent from terascale to petascale to exascale**
- » Extreme parallelism and hybrid design
 - » Preparing for million/billion way parallelism
- » Tightening memory/bandwidth bottleneck
 - » Limits on power/clock speed implication on multicore
 - » Reducing communication will become much more intense
 - » Memory per core changes, byte-to-flop ratio will change
- » Necessary Fault Tolerance
 - » MTTF will drop
 - » Checkpoint/restart has limitations

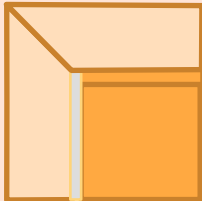
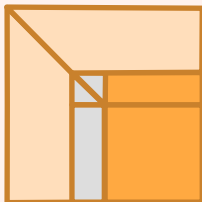
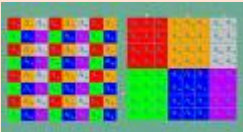
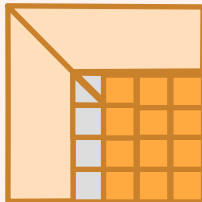
Average Number of Cores Per Supercomputer for Top20 Systems



Software infrastructure does not exist today

Linpack Evolution

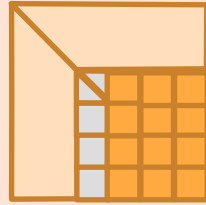
Software/Algorithms follow hardware evolution in time

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

Linpack Evolution

Software/Algorithms follow hardware evolution in time

PLASMA (00's)
New Algorithms
(many-core friendly)



Rely on

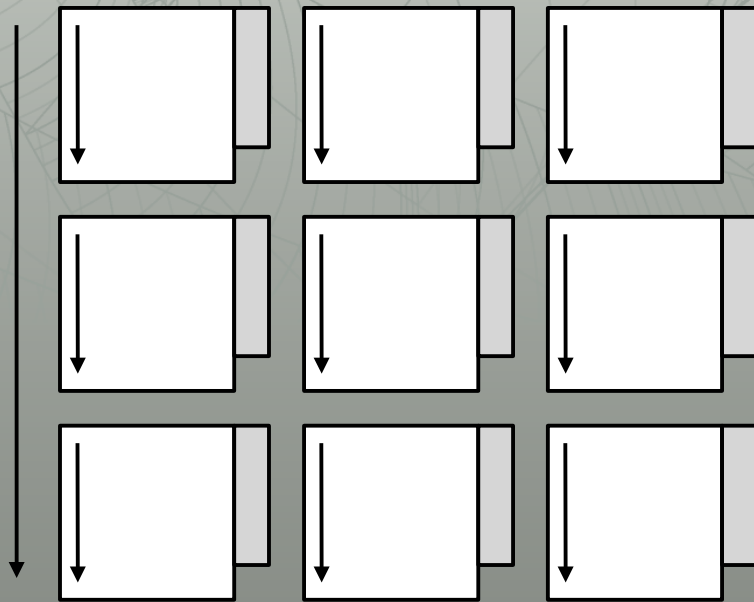
- a DAG/scheduler
- block data layout
- some extra kernels

Those new algorithms

- have a very **low granularity**, they scale very well (multicore, petascale computing, ...)
- **removes 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.

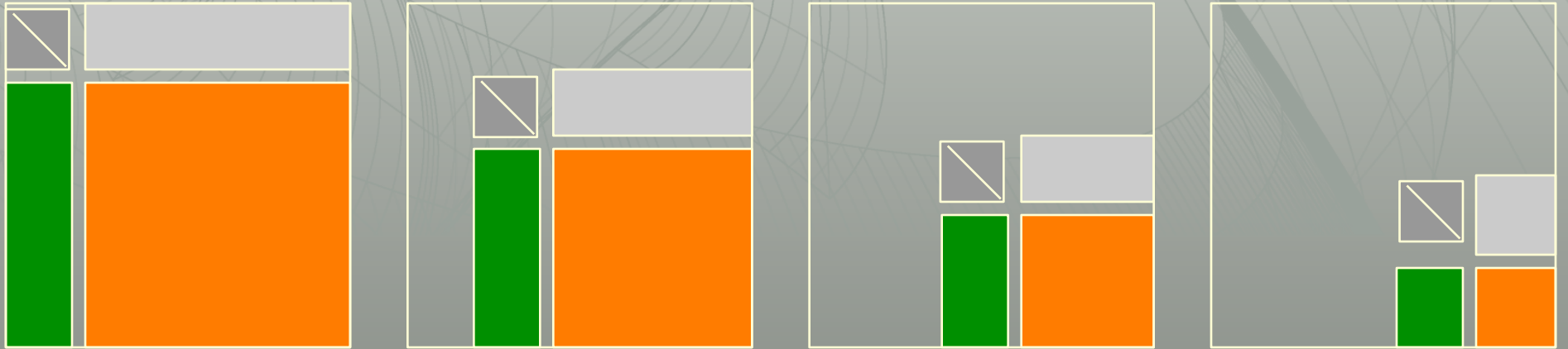
The algorithmic challenge



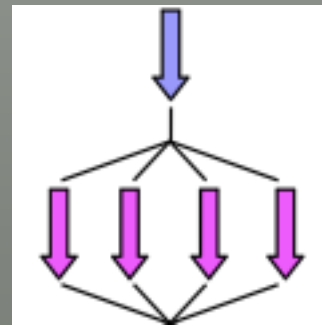
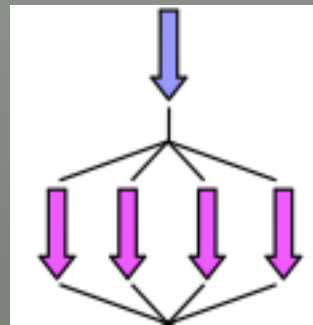
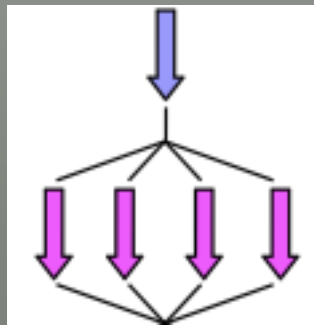
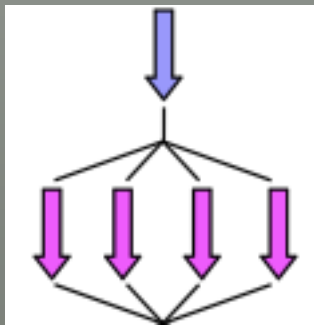
- ◆ Tiled
- ◆ Column-major / column-major
- ◆ Flat (no indirection)

- » Asynchronicity
 - » 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

LAPACK LU

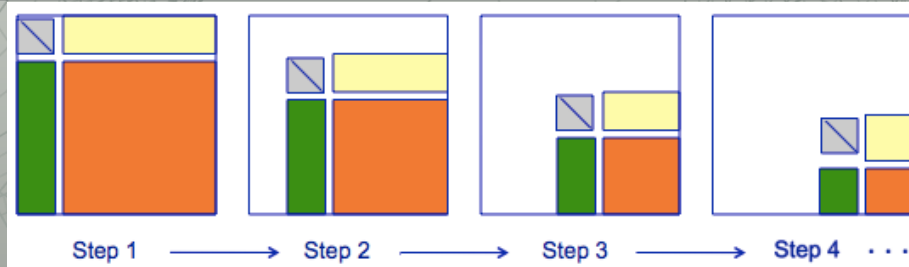


Step 1 → Step 2 → Step 3 → Step 4 . . .

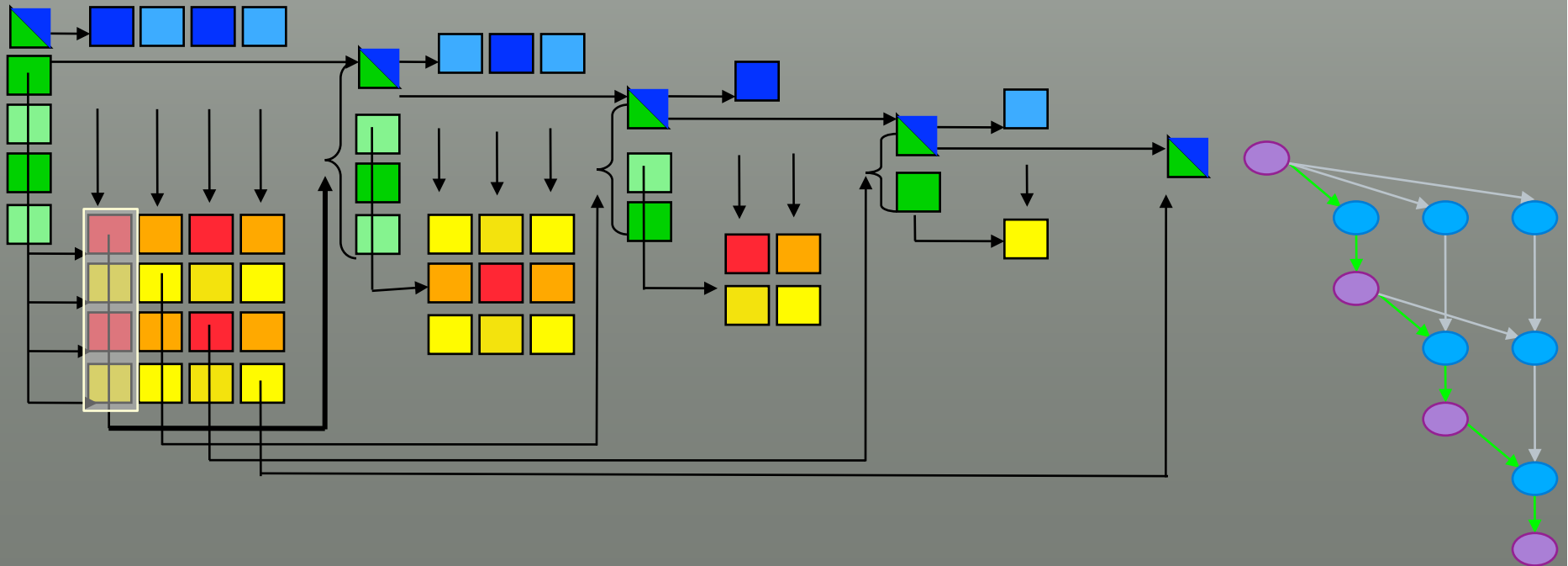


» Fork-join, bulk synchronous processing

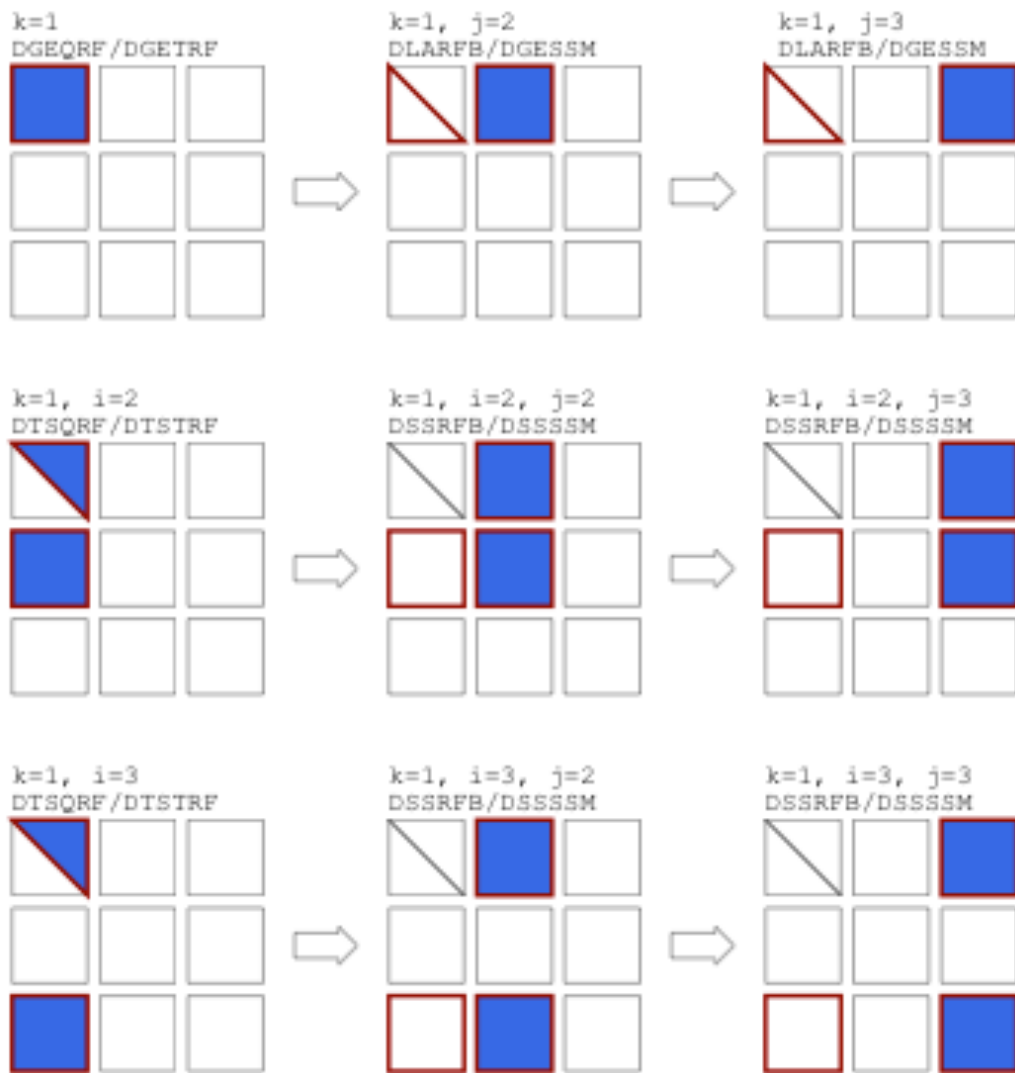
Parallel Tasks in LU



» Expose the intrinsic parallelism: break into smaller tasks and remove dependencies

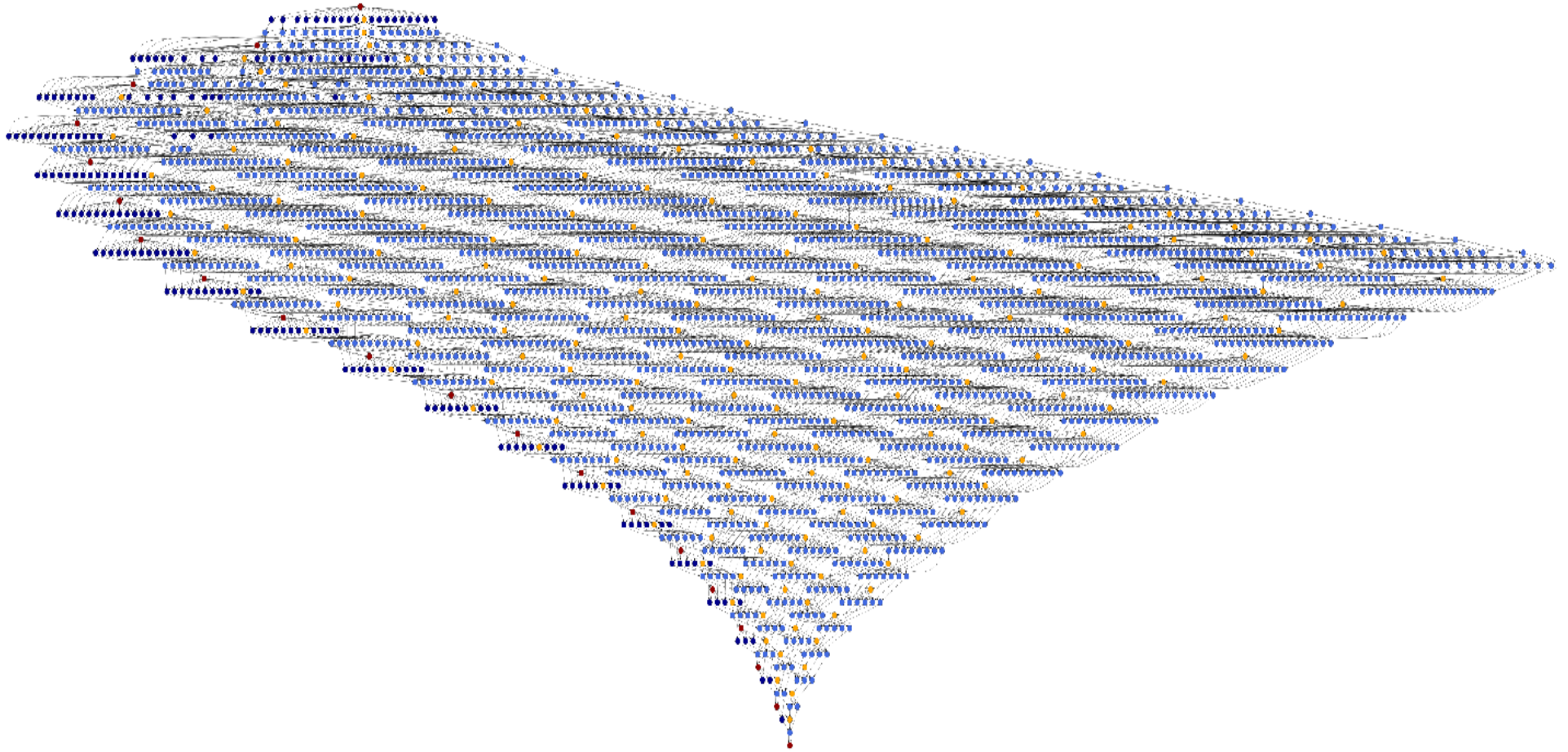


Expose the intrinsic parallelism



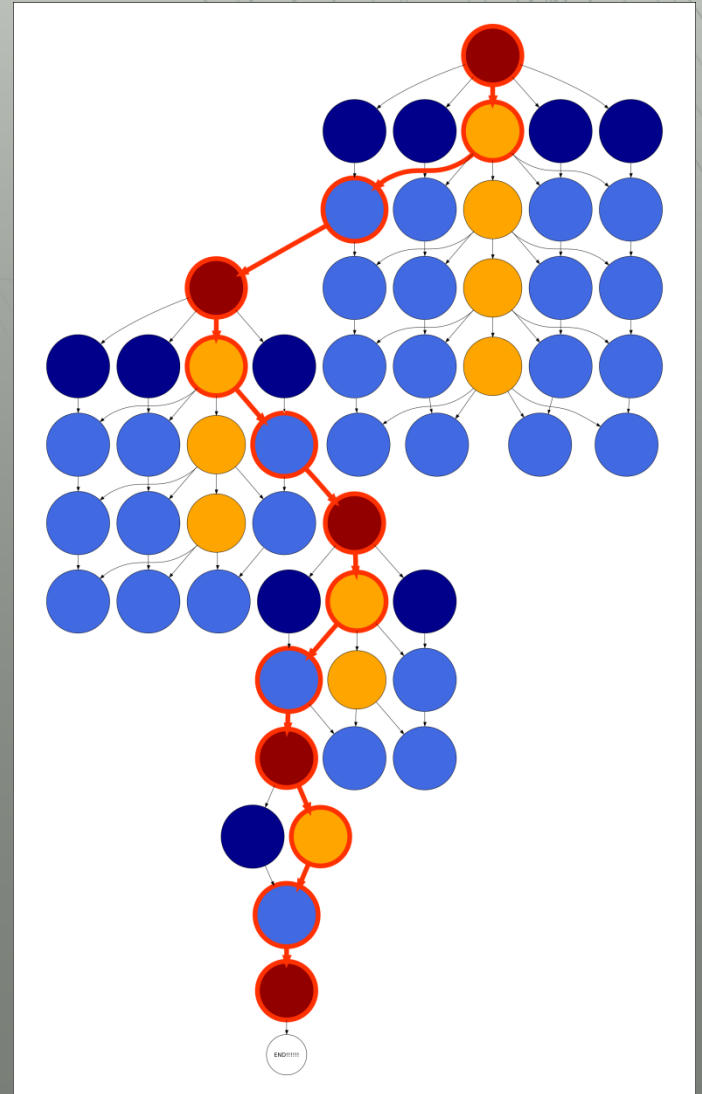
- » The three amigos
- » Example of QR factorization
 - » 4 basic kernels
- » LU identical to QR except using different kernels
- » Cholesky slightly different due to the matrix symmetry

LU DAG representation



A quite simple problem ...

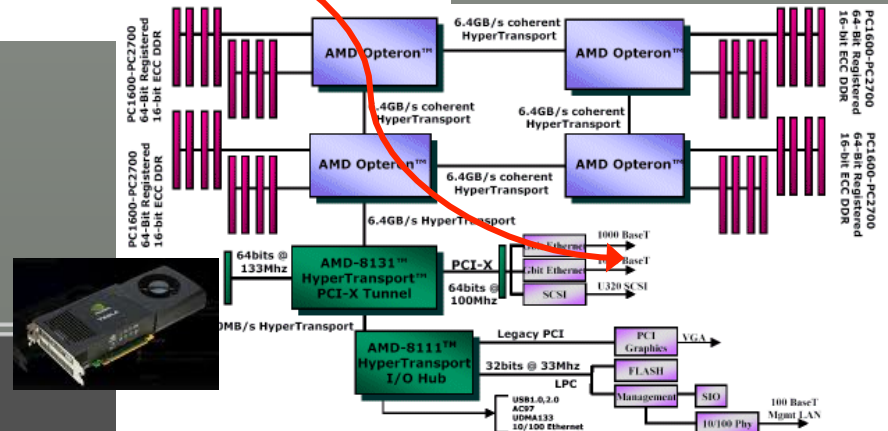
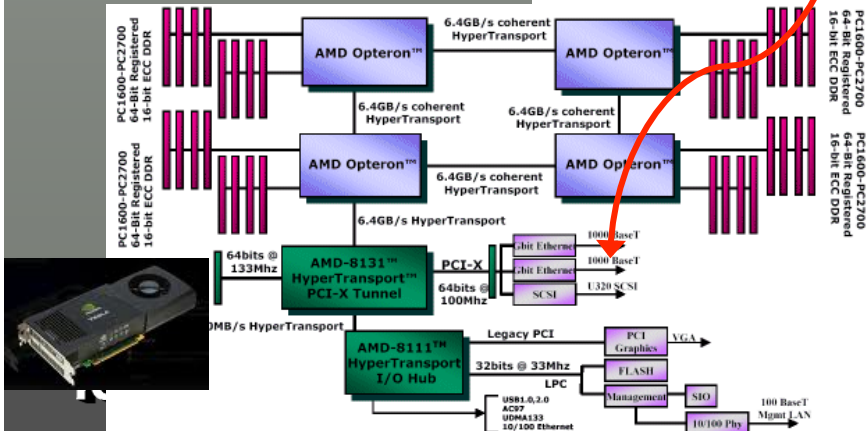
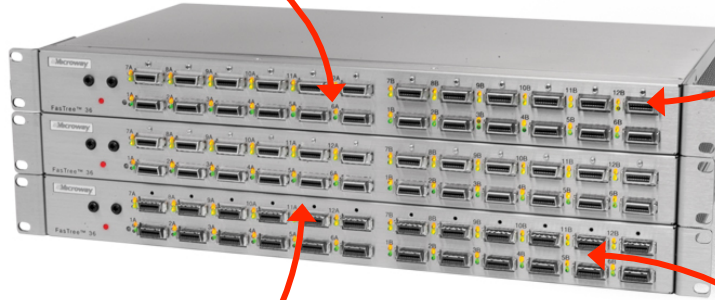
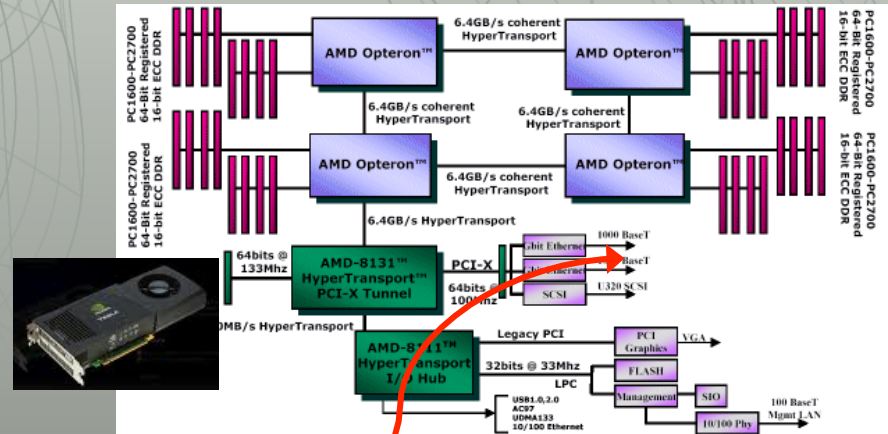
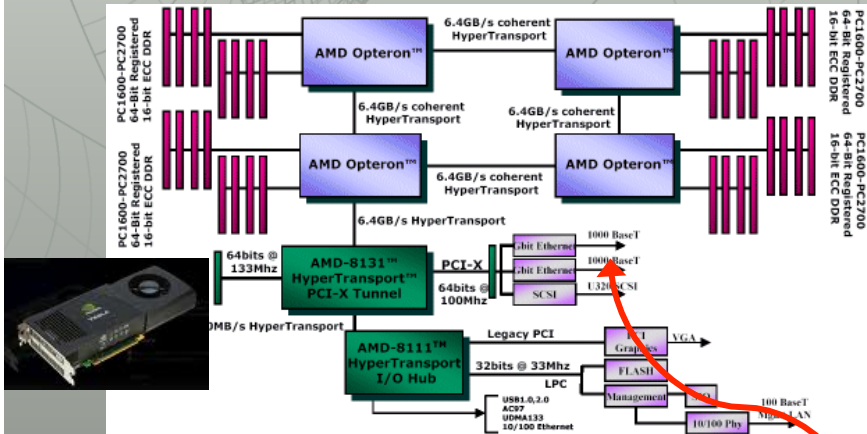
- » We would generate the DAG, find the critical path and execute it.
- » DAG too large to generate ahead of time
 - » Not explicitly generate
 - » Dynamically generate the DAG as we go
- » Machines will have large number of cores in a distributed fashion
 - » Will have to engage in message passing
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

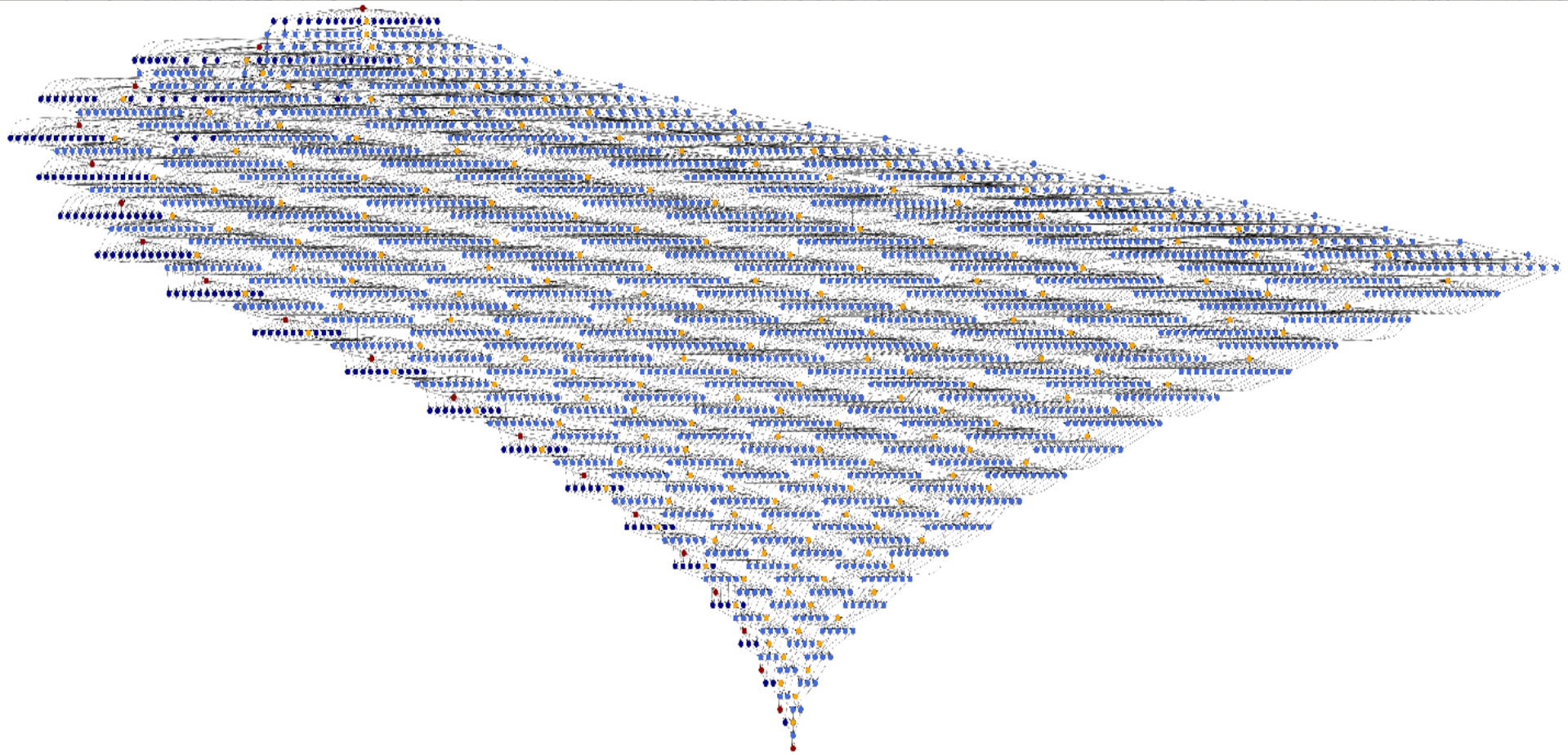


What is PLASMA?

- Dense linear algebra software library
 - Linear systems & least squares (LU, Cholesky, QR/LQ)
 - Eigenvalues & singular values
- Multicore processors
 - Multi-socket multi-core / Shared memory systems
 - NUMA systems
- What is next:
 - GPU acceleration – MAGMA
 - Distributed Memory – DAGuE / DPLASMA

Current Architectures





DPLASMA / DAGuE

- » DAGuE: the runtime
 - » Deploy a DAG on a heterogeneous distributed environment
 - » Architecture aware
 - » Minimize data movements (in and out the node)
 - » Enforce data locality (cache / NUMA / GPU)
 - » Move the data across nodes
- » DPLASMA: a algebraic description of a DAG
 - » Define the data distribution
 - » Describe the algorithm at a high level
 - » A powerful description language (?)

Kernels description

```
POTRF(k) (high_priority)
RW T <- type = [tile]
    -> type = [tile]
BODY [core]
    CORE(potrf, (uplo, NB, T, NB,
                &INFO))
END
```

```
SYRK(k,n) (high_priority)
R A <- type = [tile]
RW T <- type = [tile]
    -> type = [tile]
BODY [core]
    CORE(syrk, (Lower, NoTrans, NB, NB, -1.0,
                A, NB, 1.0, T, NB))
END
```

```
TRSM(k,n) (high_priority)
R T <- type = [tile]
RW C <- type = [tile]
    -> type = [tile]
BODY [core]
    CORE(trsm, (Right, Lower, Trans, NonUnit,
                NB, NB, 1.0, T, NB, C, NB))
END
```

```
GEMM(k,m,n)
R A <- type = [tile]
R B <- type = [tile]
RW C <- type = [tile]
    -> type = [tile]
BODY [core]
    CORE(gemm, (NoTrans, Trans, NB, NB, NB,
                -1.0, B, NB, A, NB, 1.0, C, NB))
END
BODY [cuda]
    ...
END
```


Algebraic dependencies description

```
POTRF(k)
k = [0 .. SIZE-1]
: (k / rtileSIZE) % GRIDrows == rowRANK
: (k / ctileSIZE) % GRIDcols == colRANK
T <- (k == 0) ? A(k, k) : T SYRK(k-1, k)
  -> T TRSM(k, k+1..SIZE-1)
  -> A(k, k)

; (k >= (SIZE - PRI_CHANGE)) ? 10 * (SIZE - k)
* (SIZE - k) * (SIZE - k) : 1000000000
```

```
SYRK(k,n)
k = [0 .. SIZE-1]
n = [k+1 .. SIZE-1]
: (n / rtileSIZE) % GRIDrows == rowRANK
: (n / ctileSIZE) % GRIDcols == colRANK
A <- C TRSM(k, n)
T <- (k == 0) ? A(n,n) : T SYRK(k-1, n)
  -> (n == k+1) ? T POTRF(k+1) : T SYRK(k+1,n)

; (n >= (SIZE - PRI_CHANGE)) ? 10 * (SIZE - n) *
(SIZE - n) * (SIZE - n) + 1 : 1000000000
```

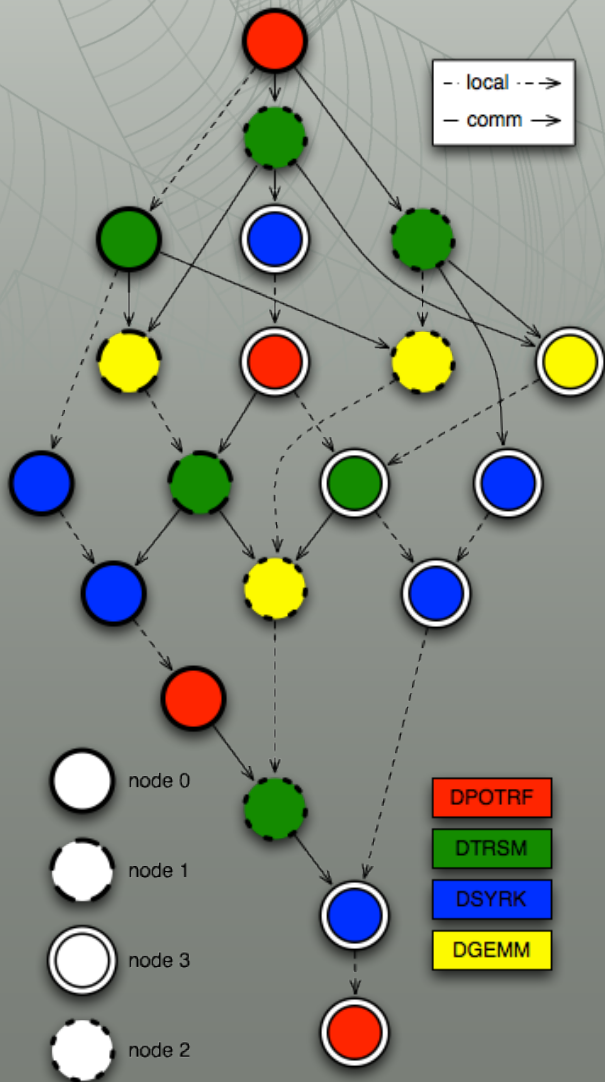
```
TRSM(k,n)
k = [0 .. SIZE-1]
n = [k+1 .. SIZE-1]
: (n / rtileSIZE) % GRIDrows == rowRANK
: (k / ctileSIZE) % GRIDcols == colRANK
T <- T POTRF(k)
C <- (k == 0) ? A(n, k) : C GEMM(k-1, n, k)
  -> A SYRK(k, n)
  -> A GEMM(k, n+1..SIZE-1, n)
  -> B GEMM(k, n, k+1..n-1)
  -> A(n, k)

; (n >= (SIZE - PRI_CHANGE)) ? 10 * (SIZE - n)
* (SIZE - n) * (SIZE - n) + 2 : 1000000000
```

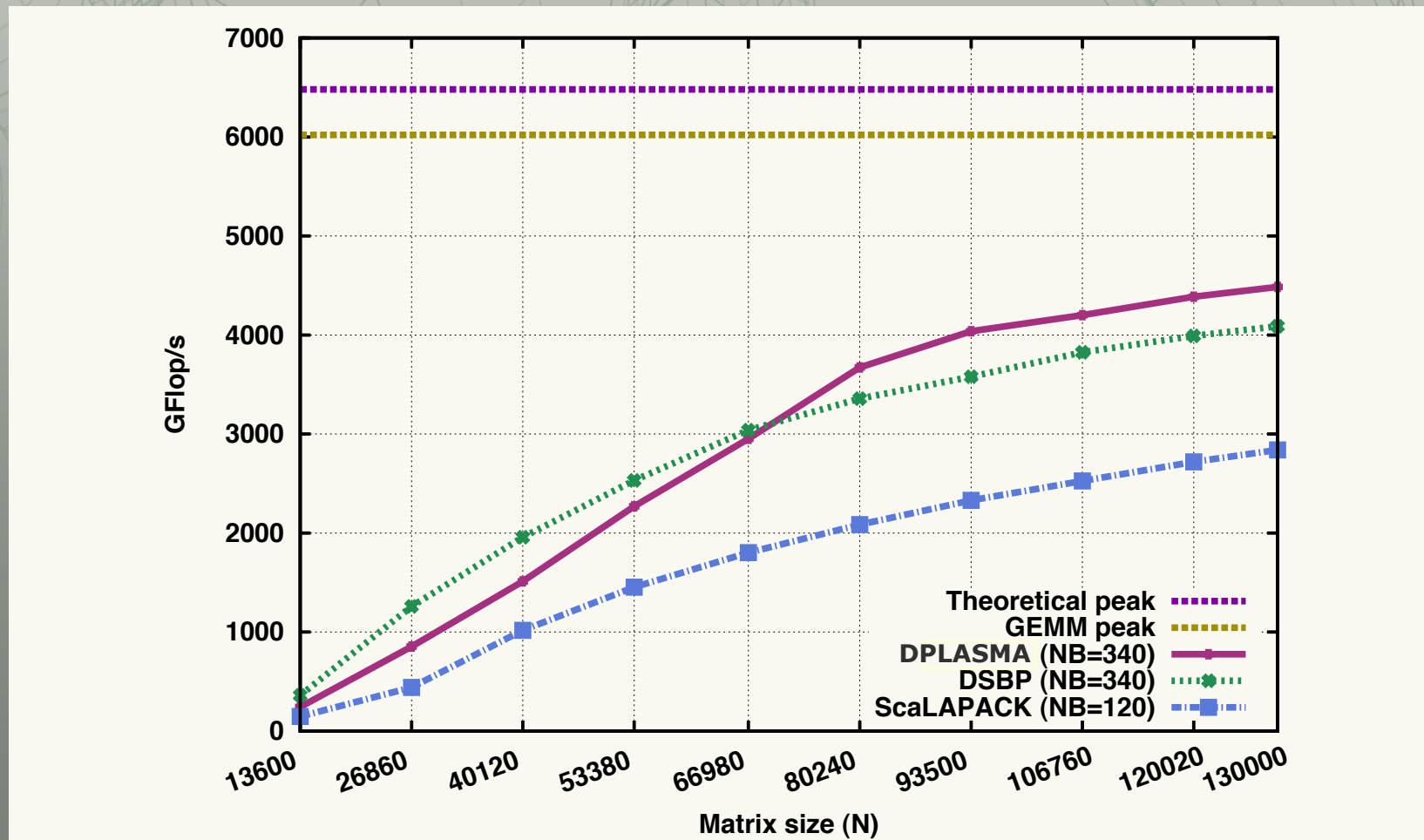
```
GEMM(k,m,n) (assoc(k))
k = [0 .. SIZE]
m = [k+2 .. SIZE-1]
n = [k+1 .. m-1]
: (m / rtileSIZE) % GRIDrows == rowRANK
: (n / ctileSIZE) % GRIDcols == colRANK
A <- C TRSM(k, n)
B <- C TRSM(k, m)
C <- (k == 0) ? A(m, n) : C GEMM(k-1, m, n)
  -> (n == k+1) ? C TRSM(k+1, m) :
      C GEMM(k+1, m, n)

; (m >= (SIZE - PRI_CHANGE)) ? 10 * (SIZE - m)
* (SIZE - m) * (SIZE - m) + 1 : 1000000000
```

Cholesky (4x4)

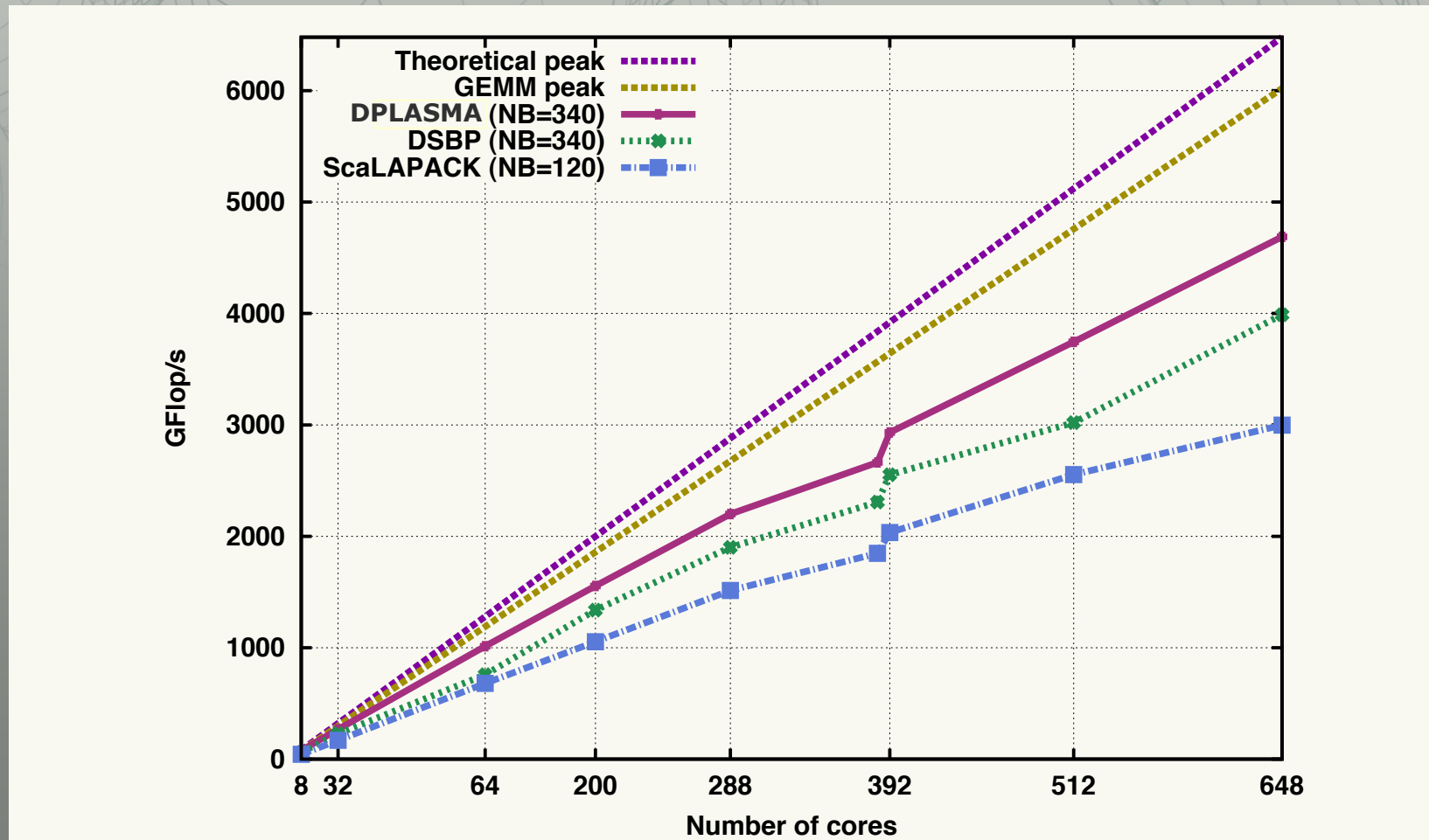


Cholesky (problem size)

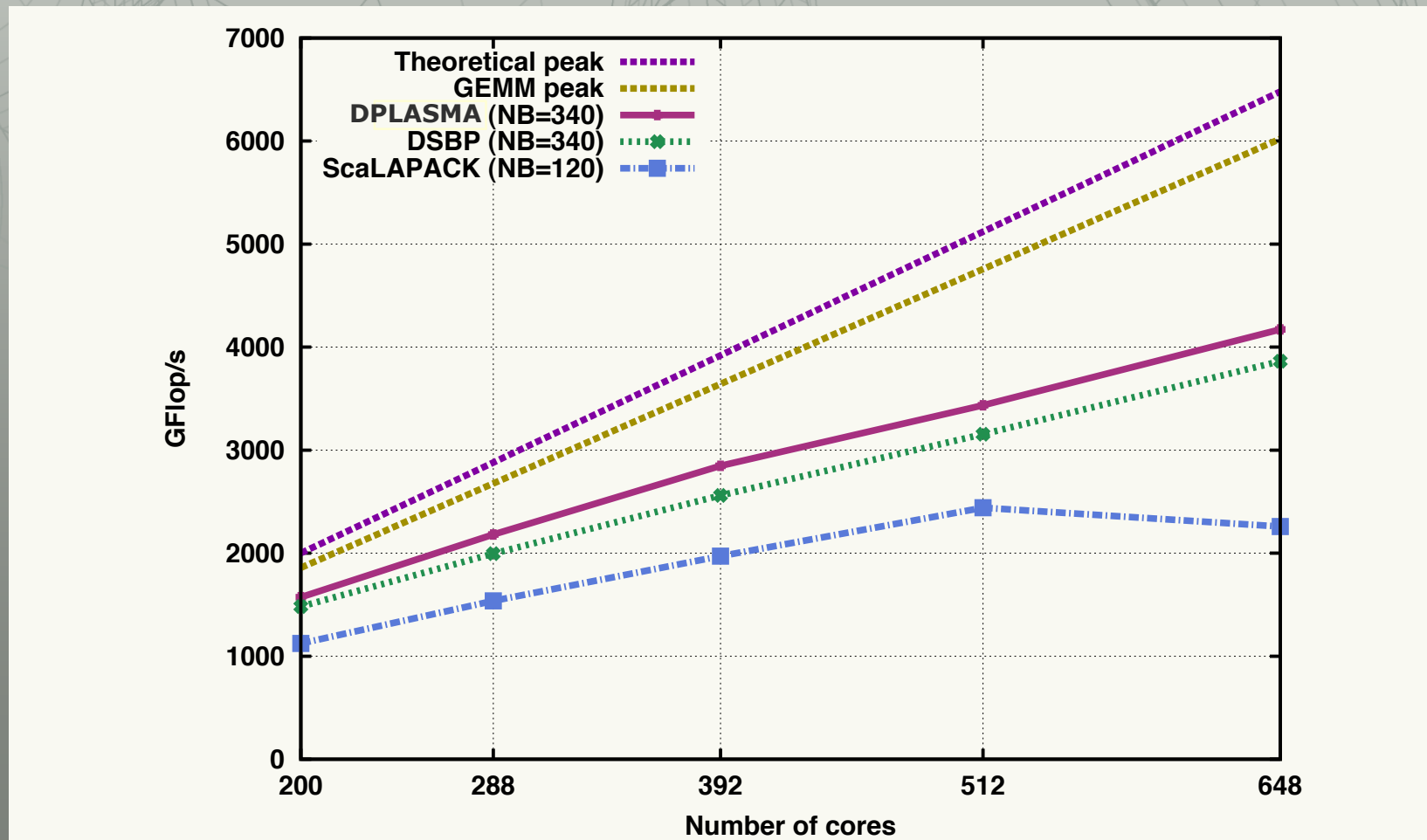


Griffon : 81 nodes, 648 cores, Infiniband 20Gbs

Cholesky (weak scalability)

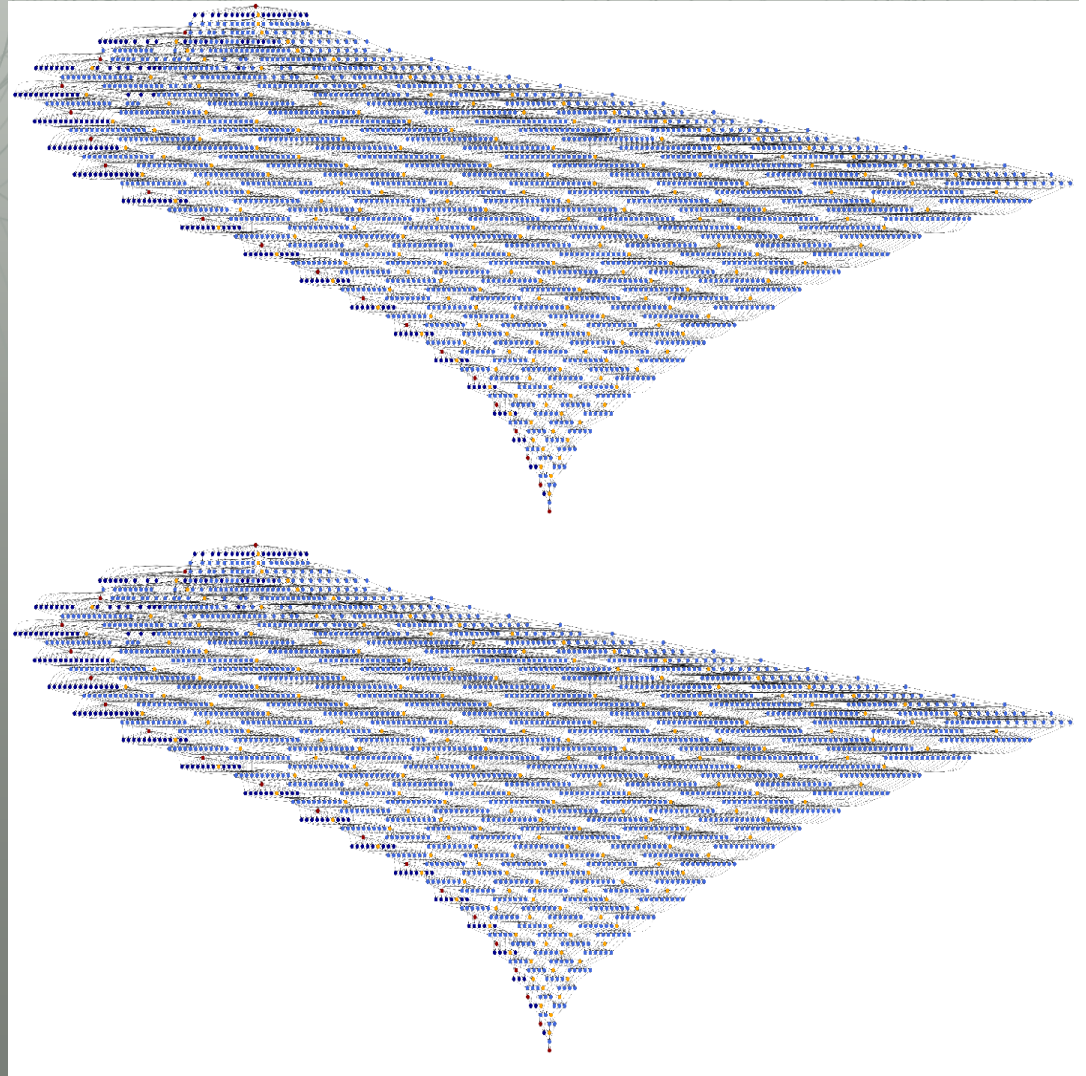


Cholesky (strong scalability)



Composability

- » Totally remove the need for synchronization
- » Follow the data flow between algebraic description of DAGs
- » Shorten the time to completion



Composability

