# DAGuE: A Generic Distributed DAG Engine for HPC

Flexible Development of Dense Linear Algebra Algorithms on Heterogeneous Parallel Architectures with DAGuE

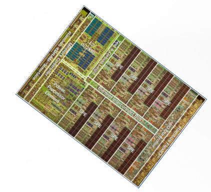
INNOVATIVE COMPUTING LABORATORY
THE UNIVERSITY of TENNESSEE UT

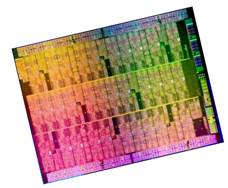
## Hardware Complexity

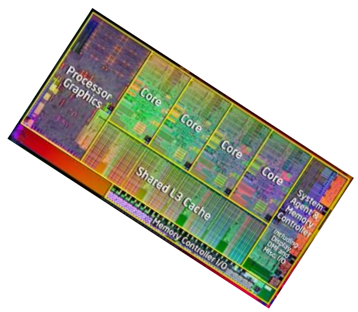
- Hierarchies of Multi-Cores
- Non Uniform Memory Access
- Accelerators
- Networks with deep hierarchies

## Portability

- Programming Portability
- Performance Portability

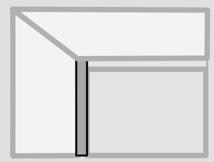
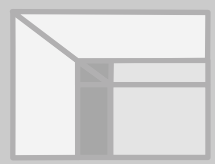
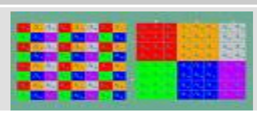



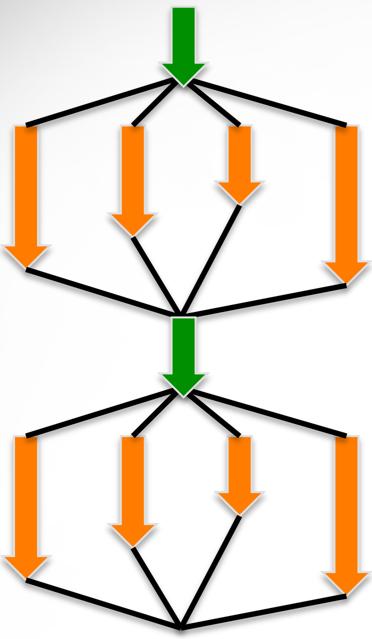




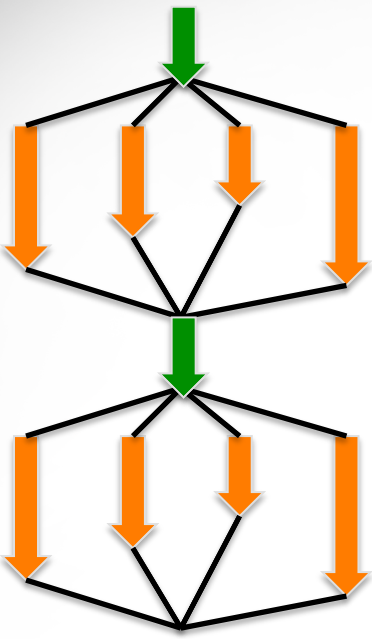

**Calls for Dynamic / Asynchronous Programming Model**

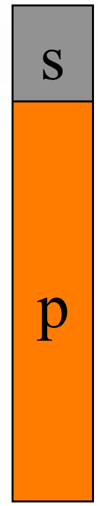
# Software Evolution

| Software/Algorithms follow hardware evolution in time | | |
|---|---|---|
| LINPACK (70's)<br>(Vector operations) |  | Rely on<br>- Level-1 BLAS operations |
| LAPACK (80's)<br>(Blocking, cache friendly) |  | Rely on<br>- Level-3 BLAS operations |
| ScaLAPACK (90's)<br>(Distributed Memory) |  | Rely on<br>- PBLAS Mess Passing |



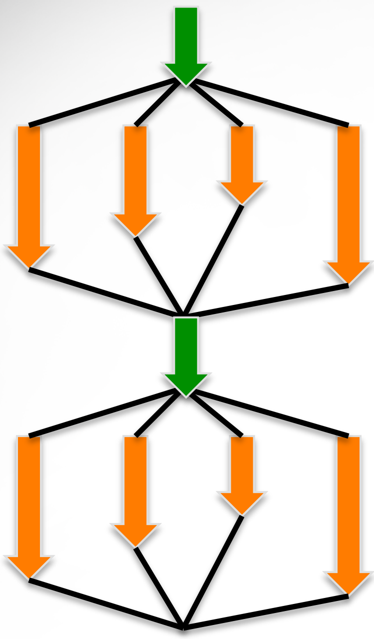
# Amdahl's Law

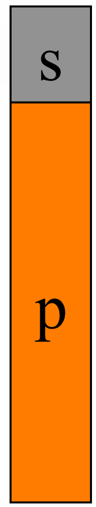





$$speedup = \frac{s+p}{s+{}^{p}\!/_{N}}$$

# Amdahl's Law






$$speedup = \frac{s+p}{s+p/N}$$

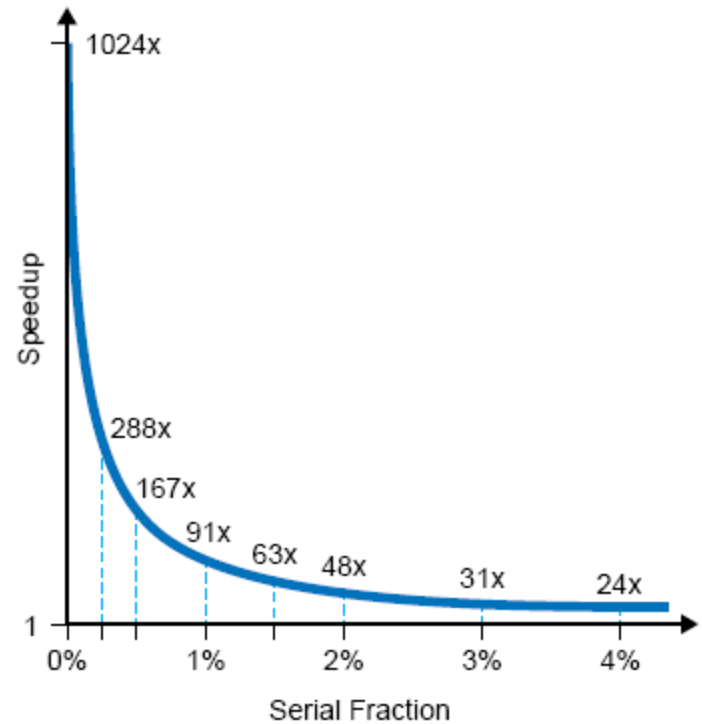
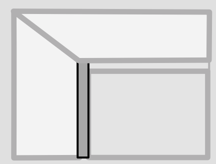
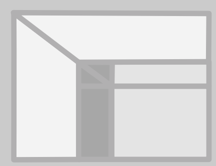
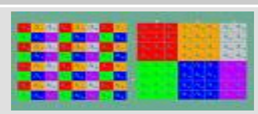



FIGURE 1. Speedup under Amdahl's Law

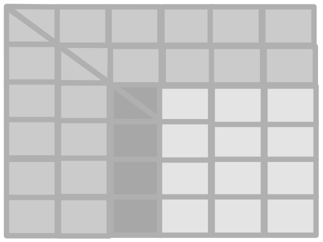
# Software Evolution

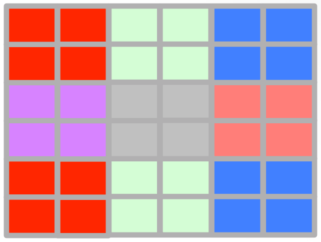
| Software/Algorithms follow hardware evolution in time | | |
|---|---|---|
| LINPACK (70's)<br>(Vector operations) |  | Rely on<br>- Level-1 BLAS operations |
| LAPACK (80's)<br>(Blocking, cache friendly) |  | Rely on<br>- Level-3 BLAS operations |
| ScaLAPACK (90's)<br>(Distributed Memory) |  | Rely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

# Software Evolution (10's)

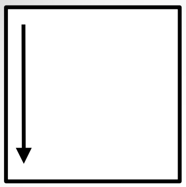
Those new algorithms
- have a very **low granularity**, they scale very well (multicore, *scale computing, … )
- **removes of dependencies** among the tasks, (multicore, distributed computing)
- **avoid latency** (distributed computing, out-of-core)
- **rely on fast kernels**

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

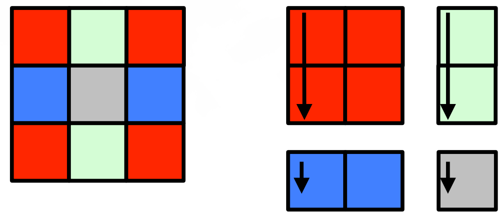




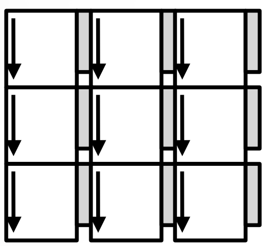
# Data Layout



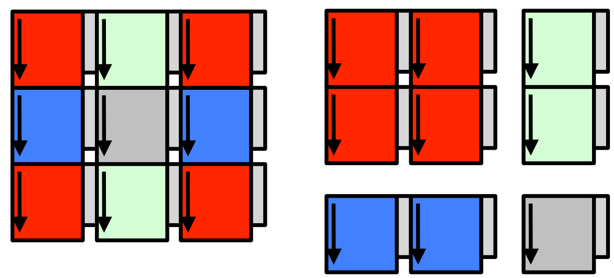
LAPACK



SCALAPACK



PLASMA

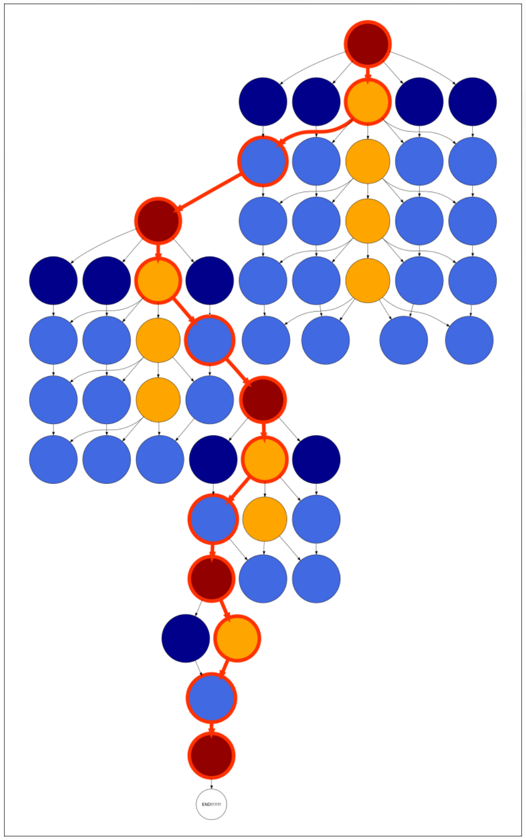


DPLASMA

4K pages – 2.4% memory overhead

# PLASMA

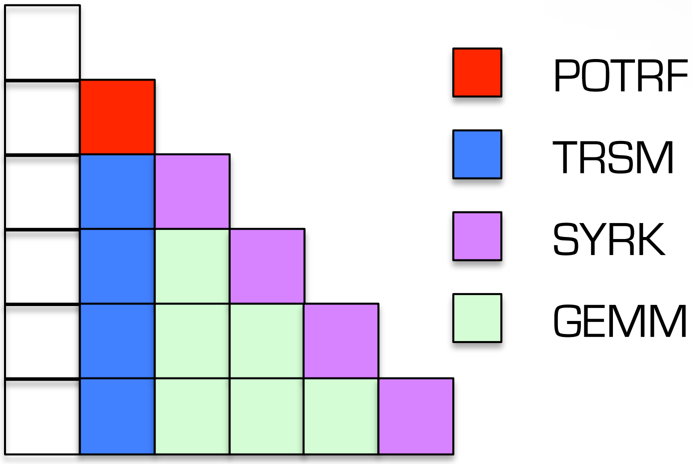
- Asychronicity
  - Avoid fork-join (Bulk sync design)
- Dynamic Scheduling
  - Out of order execution
- Fine Granularity
  - Independent block operations
- Locality of Reference
  - Data storage – Block Data Layout



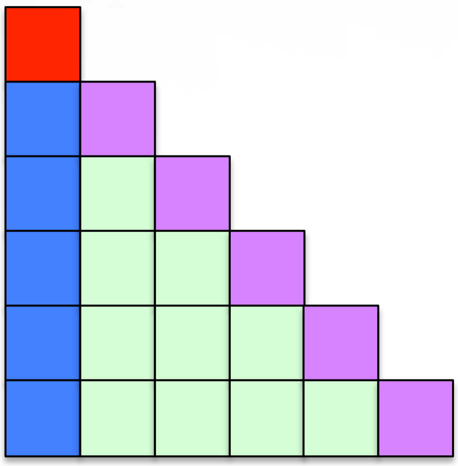
# Example: Cholesky Factorization

- Cholesky Decomposition
  - Let A be a real symmetric positive definite matrix
  - Find L such that $A = LL^T$

Tiled Algorithm in A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, A class of parallel tiled linear algebra algorithms for multicore architectures, Parallel Computing, 2008

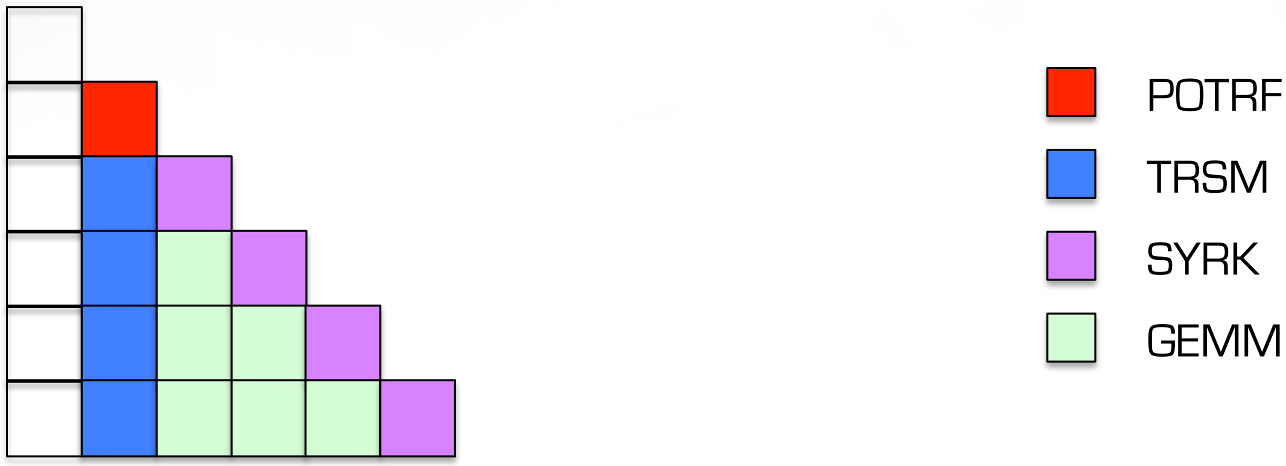


# Cholesky Factorization

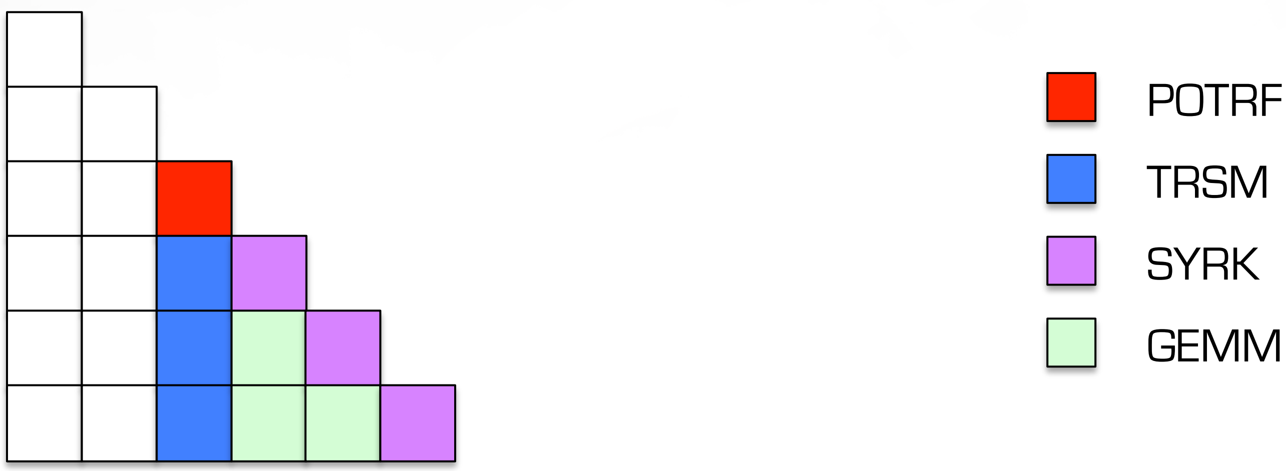


- POTRF
- TRSM
- SYRK
- GEMM

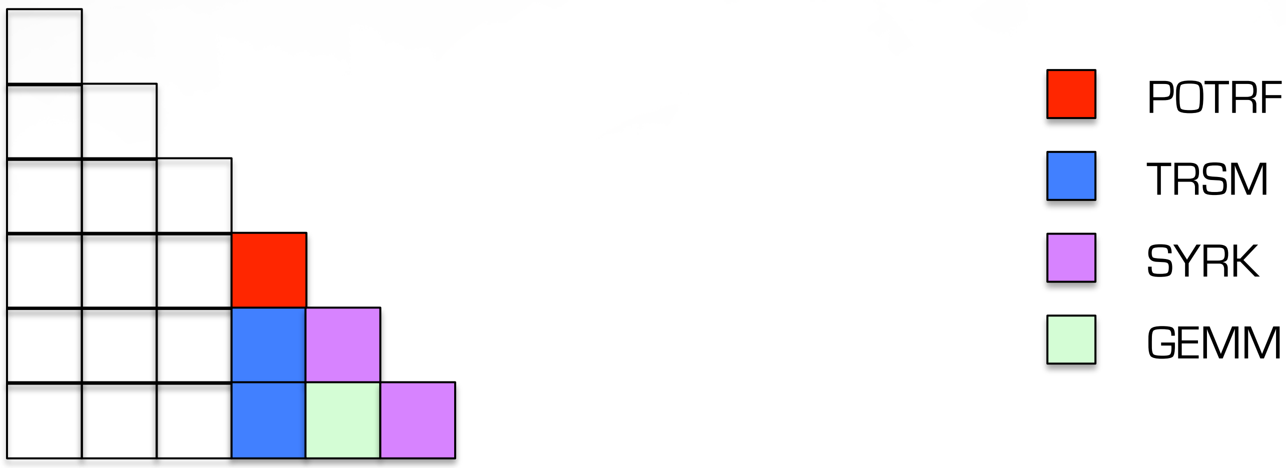
# Cholesky Factorization



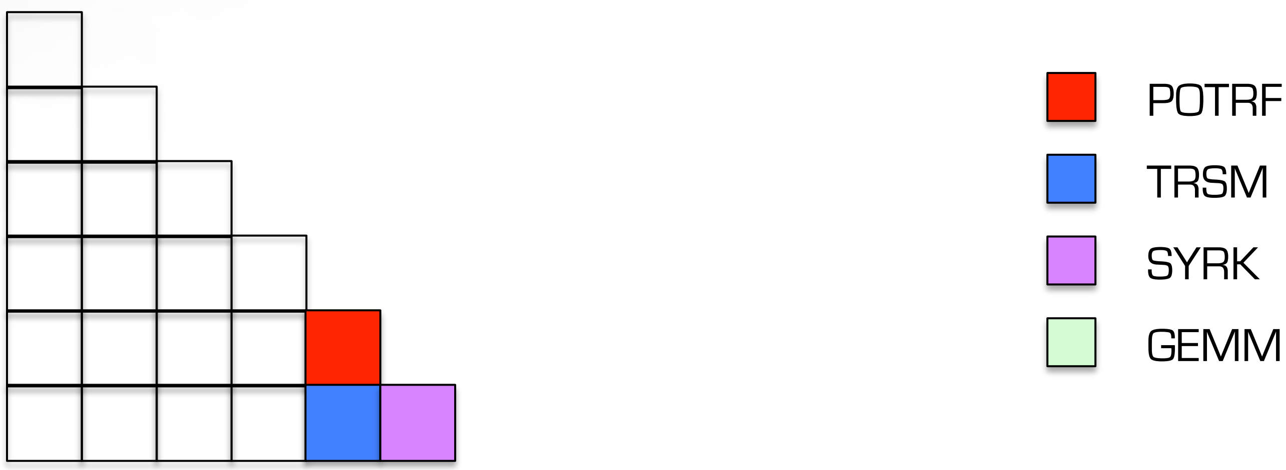
# Cholesky Factorization



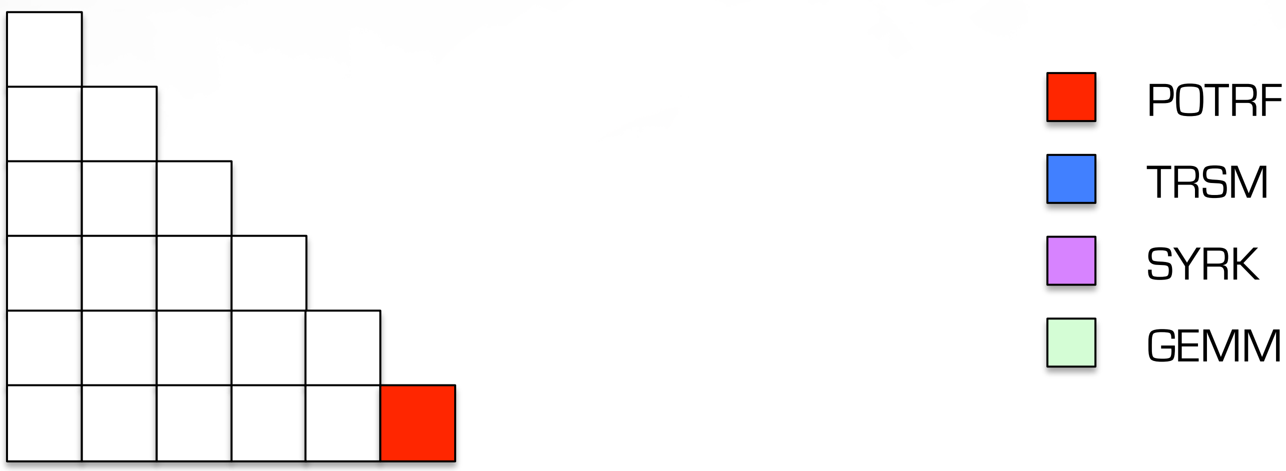
# Cholesky Factorization



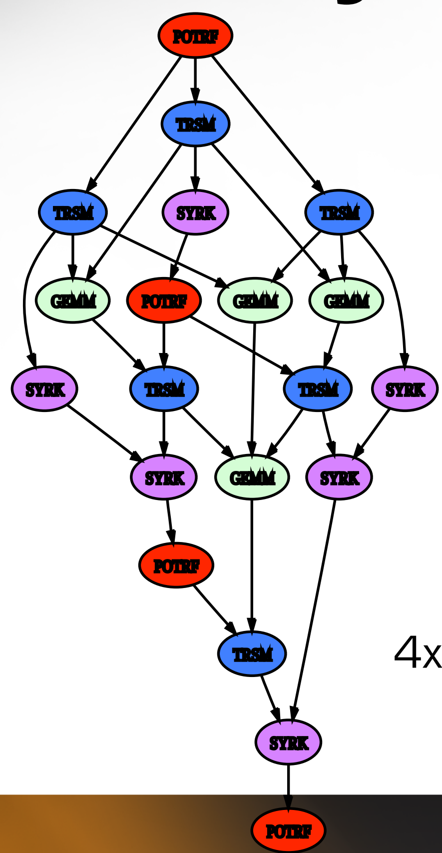
# Cholesky Factorization



# Cholesky Factorization



# Cholesky Factorization




4x4 Cholesky

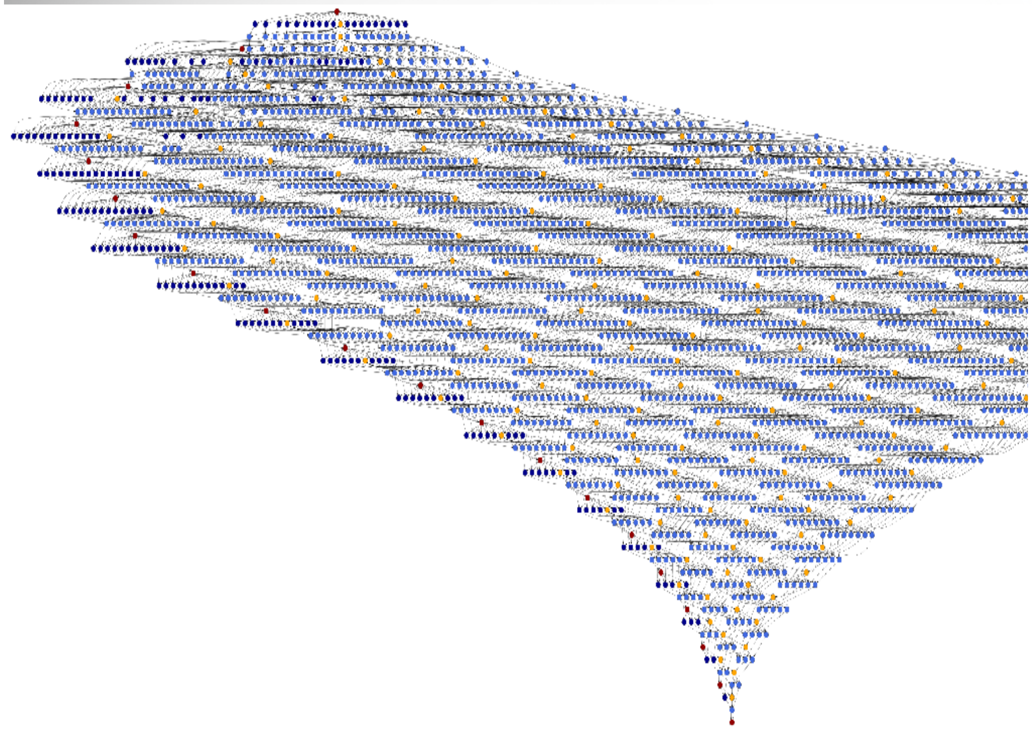
POTRF

TRSM

SYRK

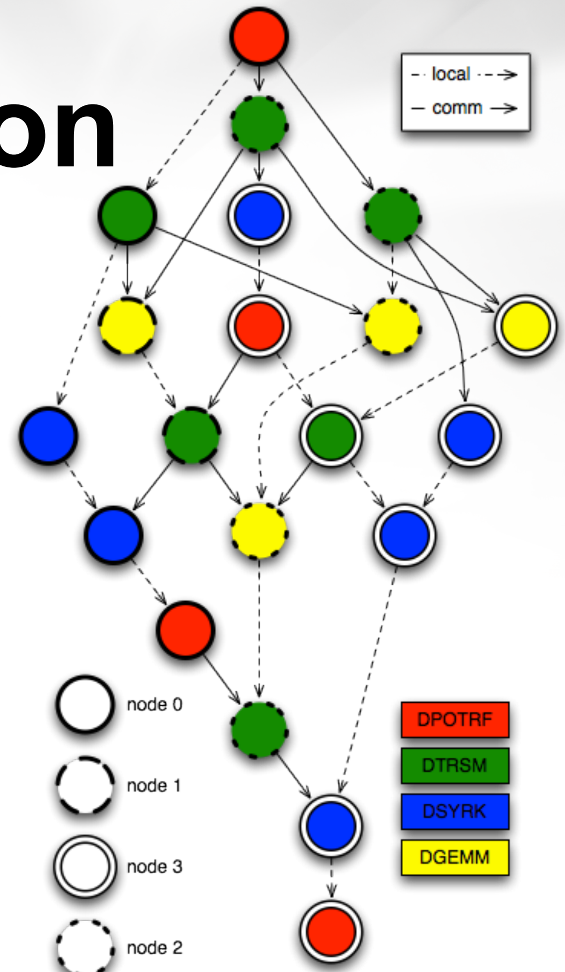
GEMM

# DPLASMA



- Observations
  - DAG too large to be generated ahead of time
    - Generate it dynamically
  - HPC is about distributed heterogeneous resources
    - Have to get involved in message passing
    - Distributed management of the scheduling
    - Dynamically deal with heterogeneity

# Cholesky Factorization




4x4 Cholesky

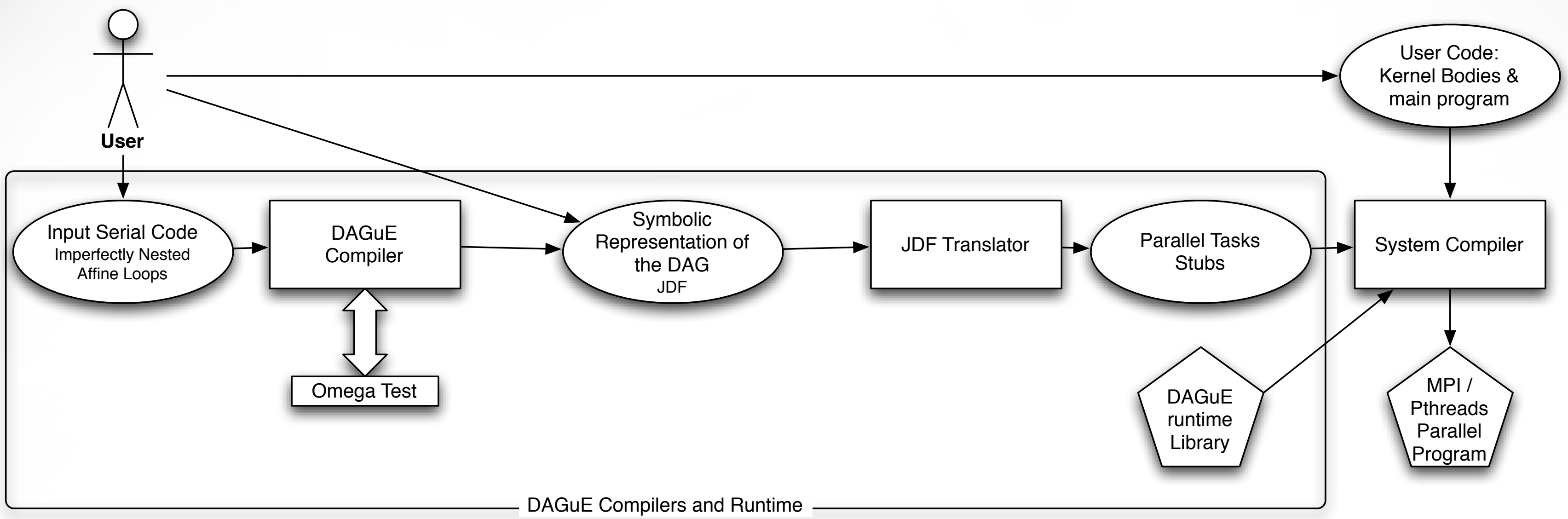
# Runtime

- Algorithms need help to unleash their power
  - The runtime can provide portability, performance, scheduling heuristics, heterogeneity management, data movement, …
  - Do not unroll/unpack the DAG, instead discover it during the execution
  - Do not support explicit communications, instead make them implicit and schedule them based on …
- The need to express the algorithms differently

# DAGuE Goals

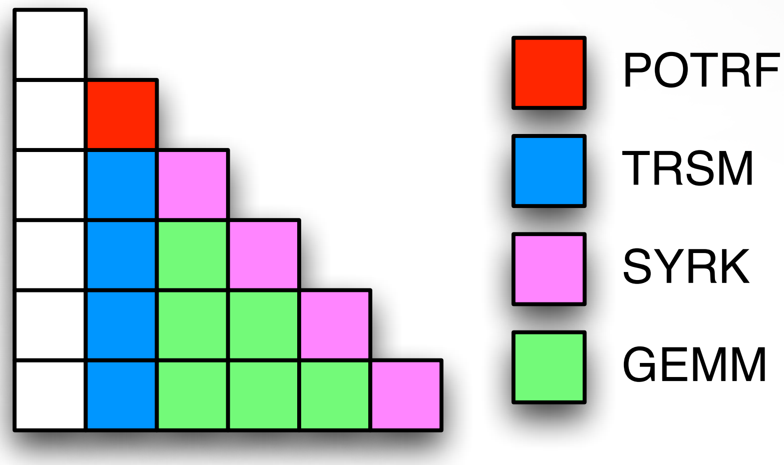
- Keep the algorithm as simple as possible
  - Depict only the flow of data between tasks
  - *Distributed Dataflow Environment based on Dynamic Scheduling of (Micro) Tasks*
- Programmability: layered approach
  - Algorithm / Data Distribution
- Portability / Efficiency
  - Use all available hardware; overlap comm / comp
- Decouple "System issues" from Algorithm

# DAGuE toolchain



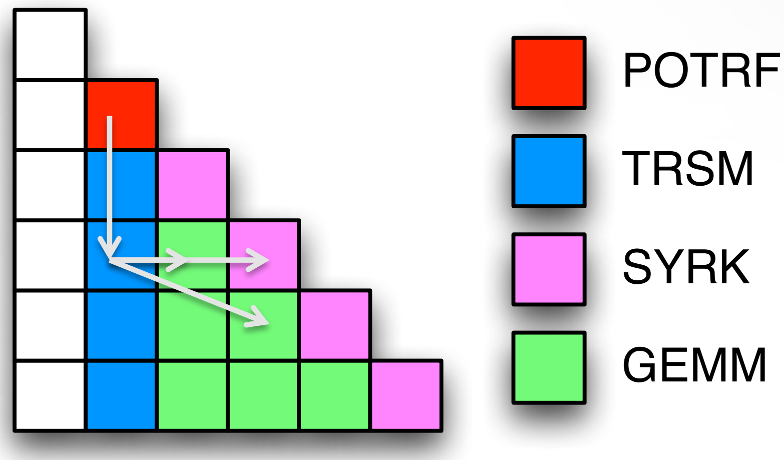
# Input Format: SMPSS-Like

```
FOR k = 0..TILES-1
  A[k][k] ← DPOTRF(A[k][k])
  FOR m = k+1..TILES-1
    A[m][k] ← DTRSM(A[k][k], A[m][k])
  FOR n = k+1..TILES-1
    A[n][n] ← DSYRK(A[n][k], A[n][n])
    FOR m = n+1..TILES-1
      A[m][n] ← DGEMM(A[m][k], A[n][k], A[m][n])
```



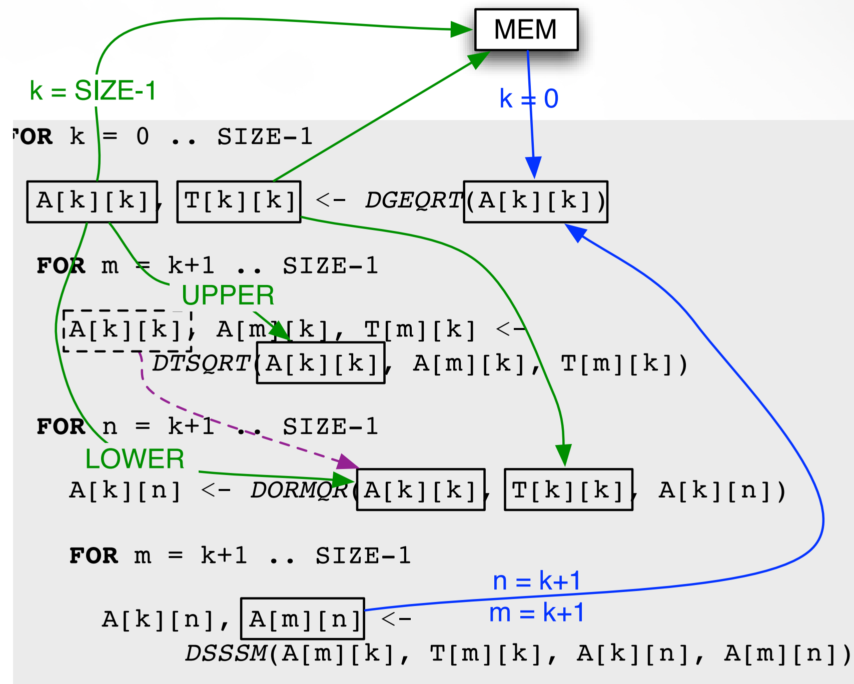
# Input Format: Job Data Flow

```
TRSM(k, n)
// Execution space
k = 0..SIZE-1
n = k+1..SIZE-1
: A(n, k) // Parallel Partitionning
READ  T <- T POTRF(k)
RW    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)
```

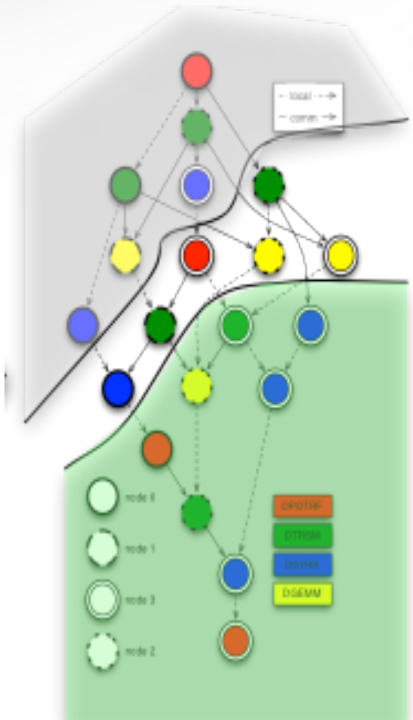


# From Seq. to JDF

- DAGuE Compiler
  - Analysis the data flow using algebraic expressions
  - Omega Test used to compute algebraic relations between edges
    - Imperfectly nested affine loop tests
  - Anti-Dependencies may introduce additional control edges

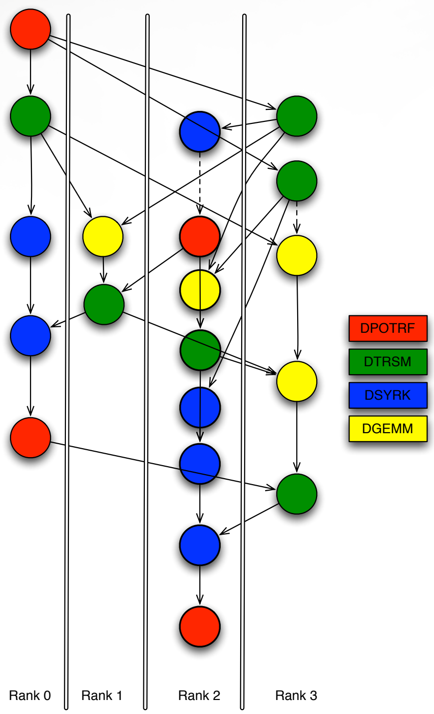


# Runtime DAG Representation




- Every process has the algebraic DAG rep.
- Dist. Scheduling based on remote completion notifications
- NUMA / Cache aware Scheduling
- Work Stealing and sharing based on memory hierarchies

# Runtime DAGuE Engine




- Data Distribution (and data/task affinity) imposes a task location
- On each node, the full DAG algebraic representation is available
- Each computing unit (core, GPU, etc.) runs its own instance of the DAGuE scheduler
- An additional communication thread sends completion notifications and data when necessary

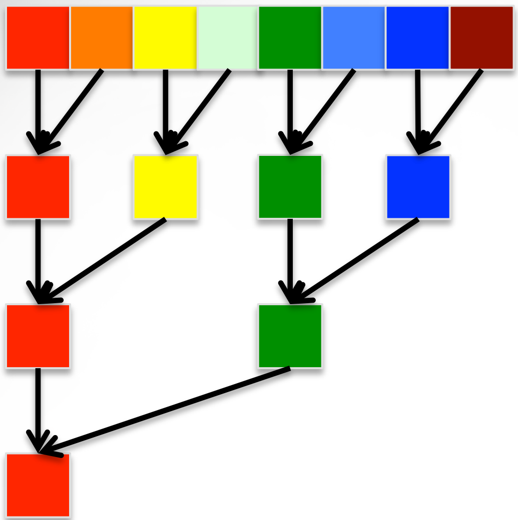
# Scheduling in DAGuE

- Based on Work Stealing
  - Shared data structures with atomic access operations
  - Uniform scheduler: all scheduler run with the global view of the DAG and the local view of progress (plus remote notifications)
  - Fully Distributed scheduler: all threads alternate between scheduling and work
- Main heuristic: data locality
  - DAGuE engine tracks data usage, and targets to improve data reuse
  - NUMA aware hierarchical bounded buffers to implement work stealing
- Users hints: tasks with “high priority”; Algebraic expressions for priorities
  - Insertion in waiting queue abides to priority, but work stealing can alter this ordering
- Communications heuristics
  - Communications inherits priority of destination task

# Example: Reduction Operation

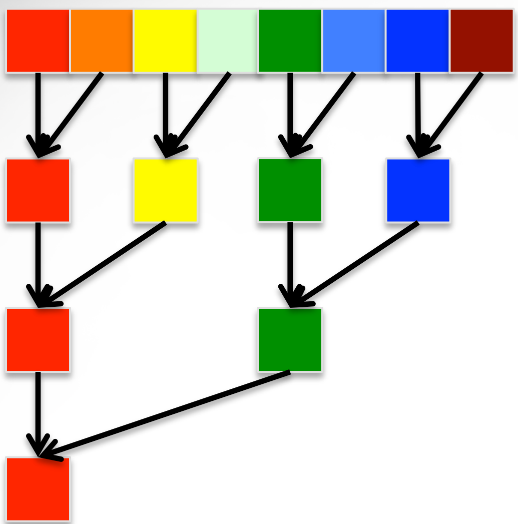
- Apply a user defined operator on each data and store the result in a single location.
- Suppose the operator is associative and commutative.

# Example: Reduction Operation



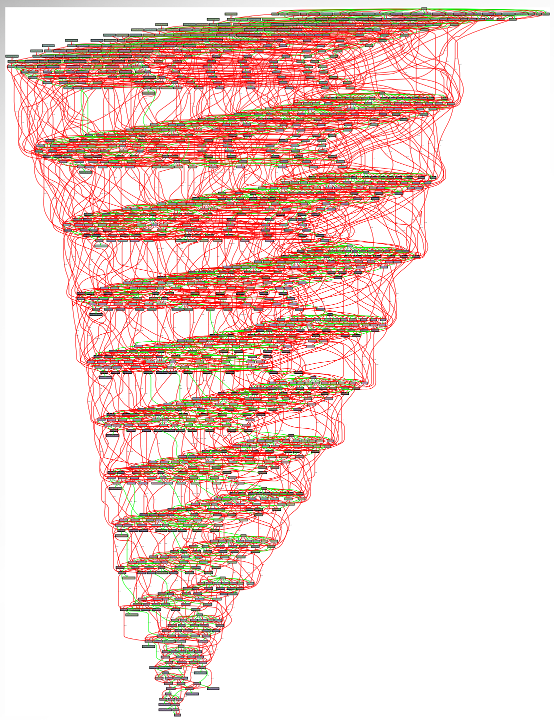
- Apply a user defined operator on each data and store the result in a single location.
- Suppose the operator is associative and commutative.

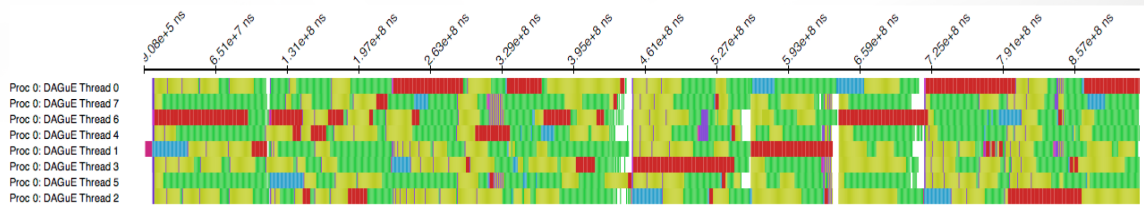
# Example: Reduction Operation

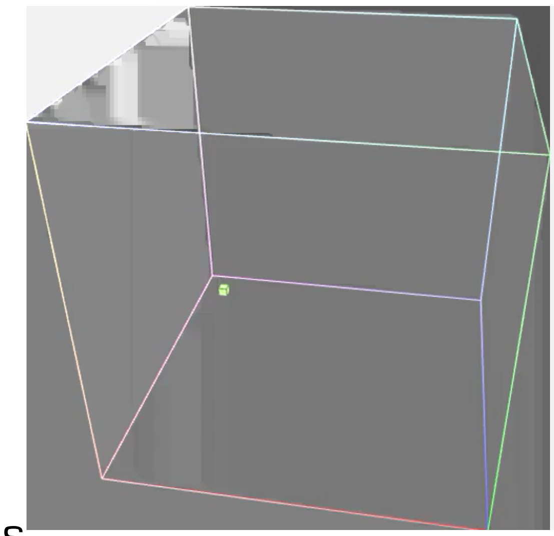


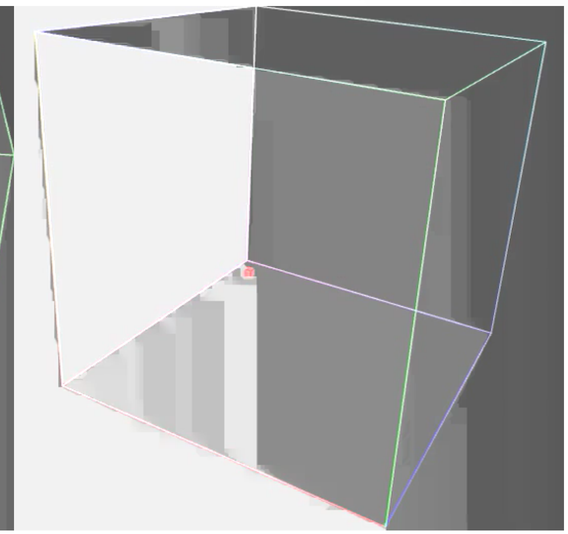
```
0  reduce(l, p)
     l = 1 .. depth+1
1    p = 0 .. (MT / (1<<l))
     : A( p )
    READ A <- (1 == l) ? A(2*p) : C reduce( l - 1, 2 * p )
2   READ B <- ((p * (1 << l) + (1 << (l-1))) >  MT)  ? A(0)
              <- (1 == l) ? A(2*p+1)
3             <- (1 != l)  ? C reduce(l - 1, p * 2 + 1)
     WRITE C -> ((depth+1) == l) ? R(p)
          -> (0 == (p%2)) ? A reduce(l+1, p/2)
                              :B reduce(l+1, p/2)
```

# DAGuE: Analysis Tools

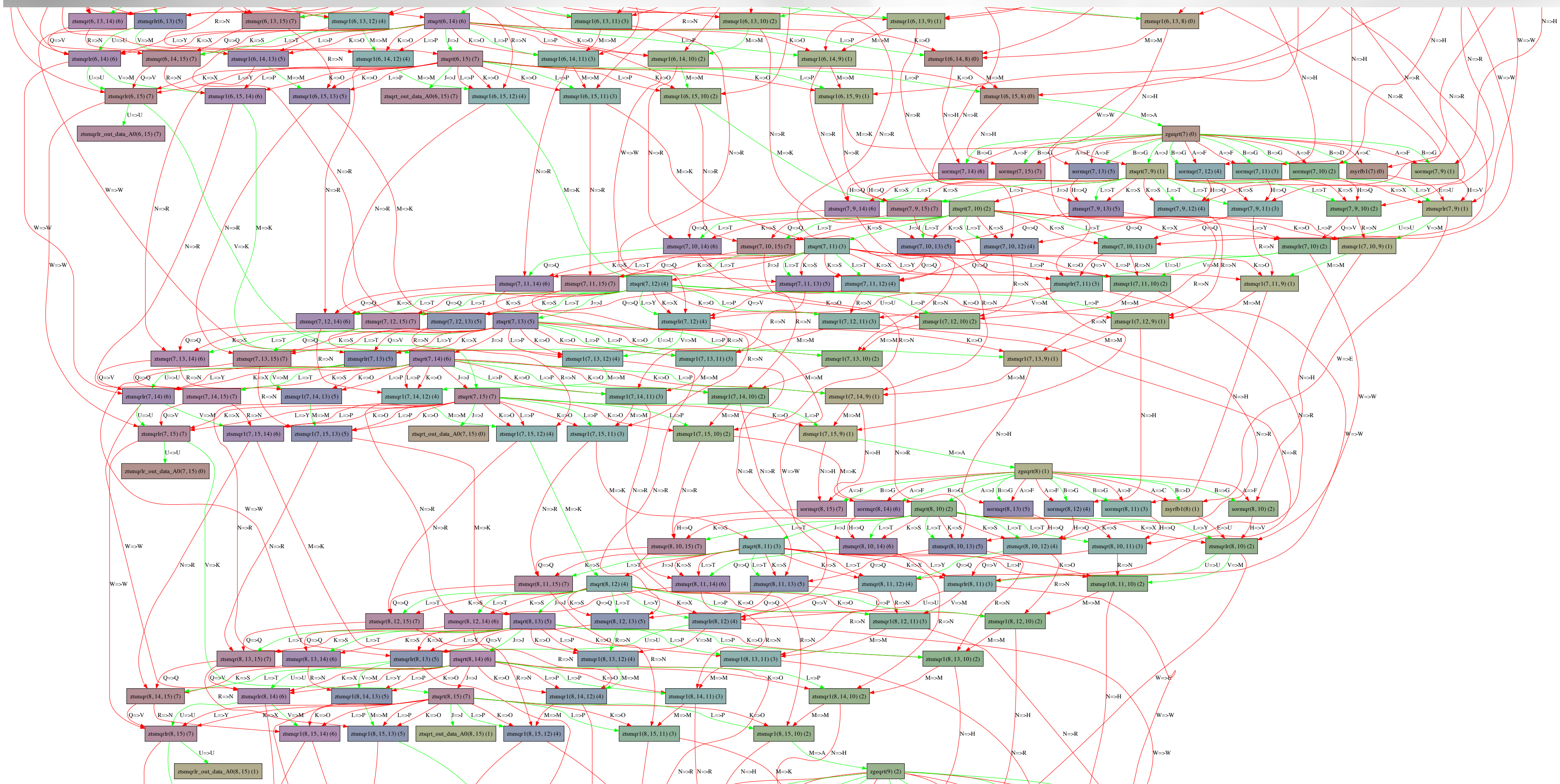






Hermitian Band Diagonal; 16x16 tiles



# Experimental Platform

**Dancer @ UTK**

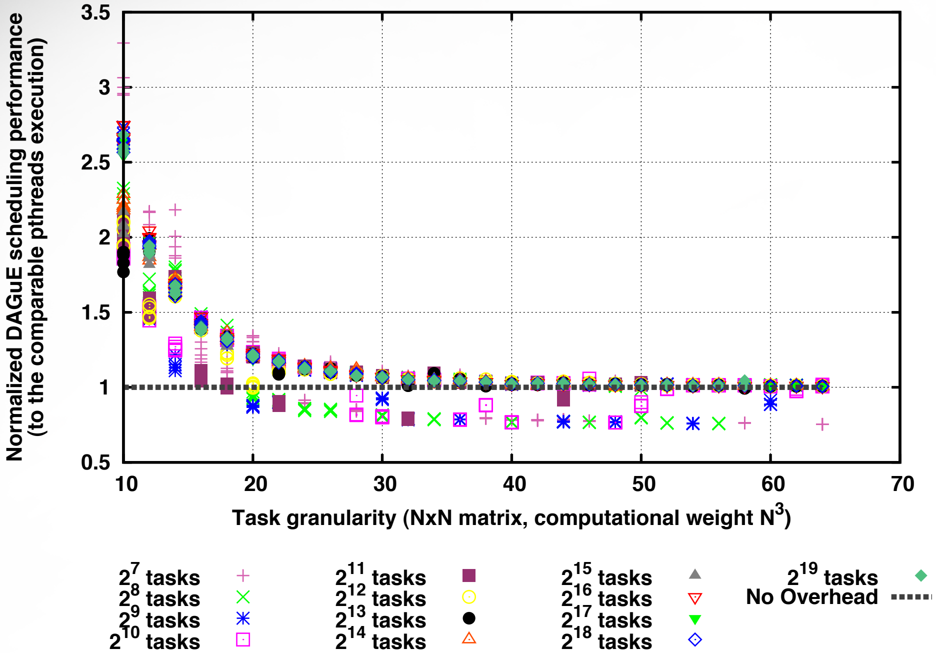
- 32 Cores (8 sockets)
- Intel Q9400 quad cores @ 2.5GHz
- 4GB RAM
- 2x 1GB/s ethernet
- 4 nodes with Fermi GPU
- 4 nodes with Tesla GPU

**Griffon@ Grid 5000**

- 648 Cores (8 sockets)
- Intel Q9400 octo cores @ 2.5GHz
- 4GB RAM / core
- Infiniband 20Gbs
- no GPU

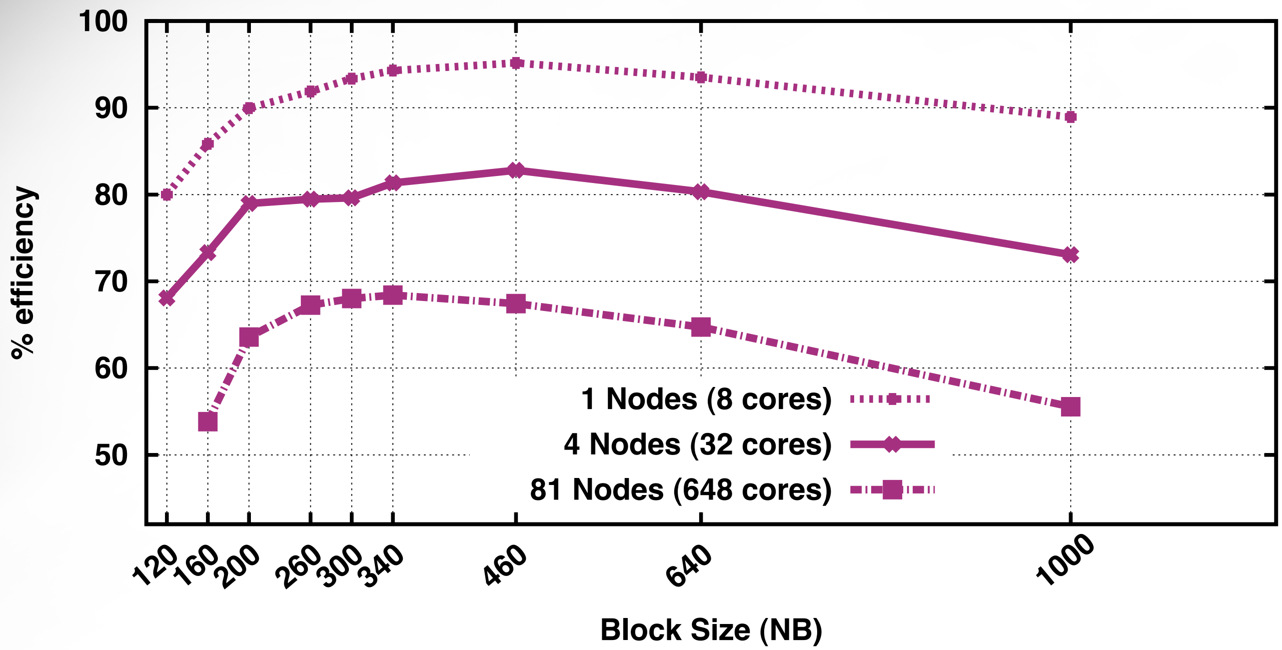
MKL-10.1.0.015 / gcc 4.4 / gfortran 4.4

# Scheduling overhead



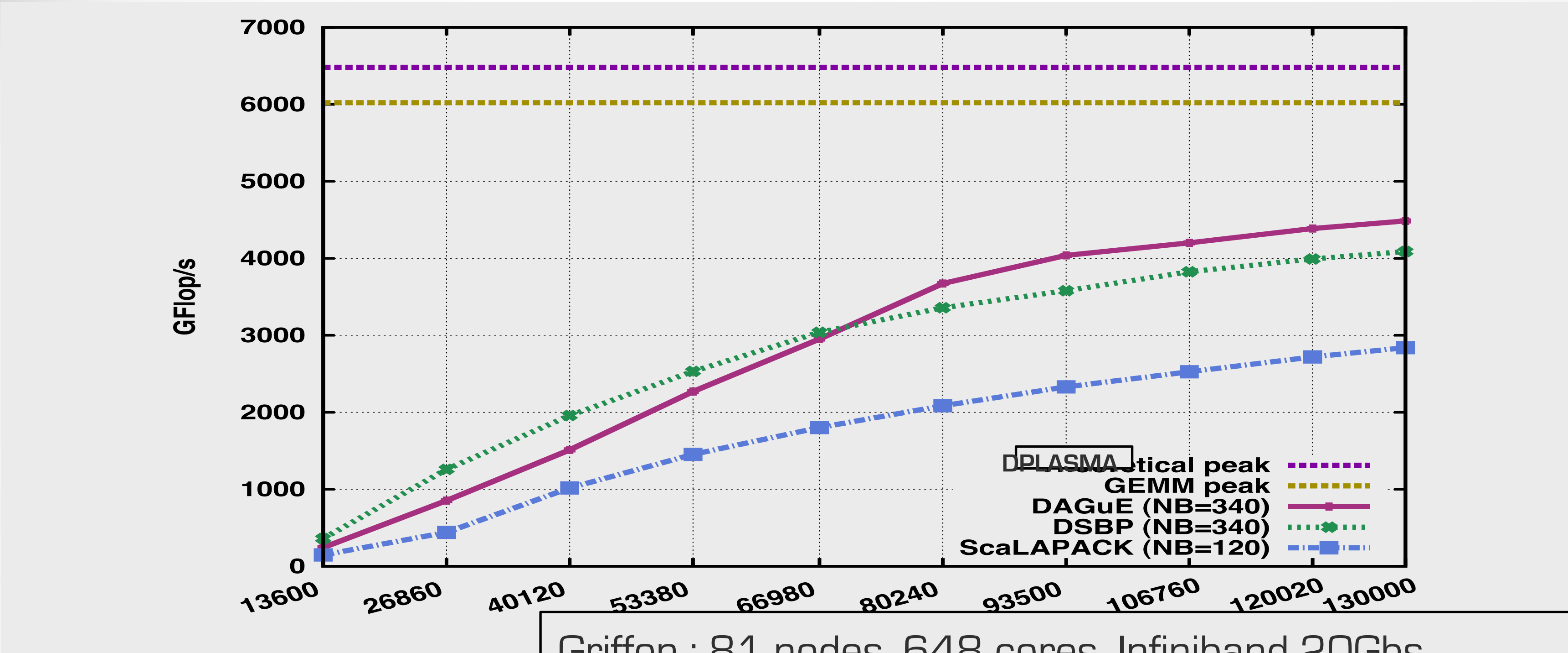

- Scheduler capable of handling fine grain tasks – 1 microsec

# Task granularity



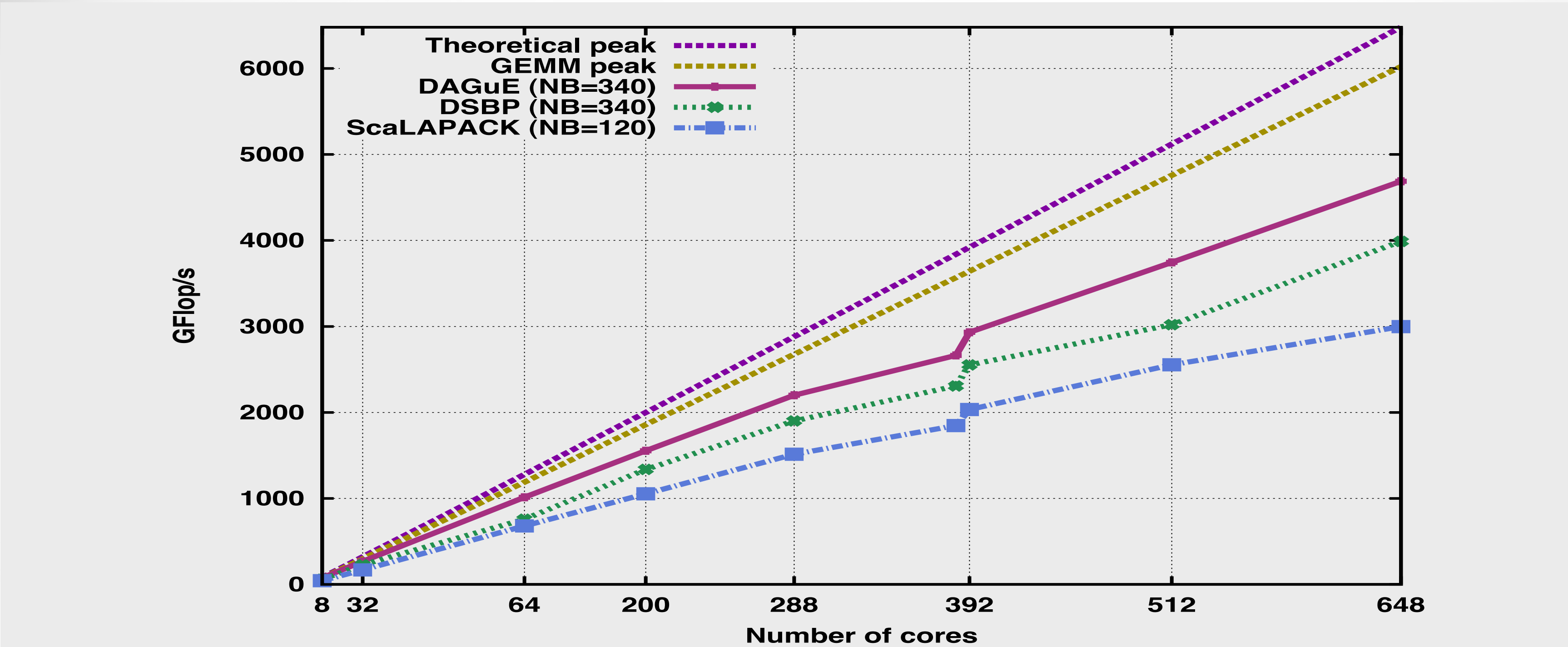

- Depends on the network, available resources.
- For best performance: auto-tune per system

# Cholesky (problem size)



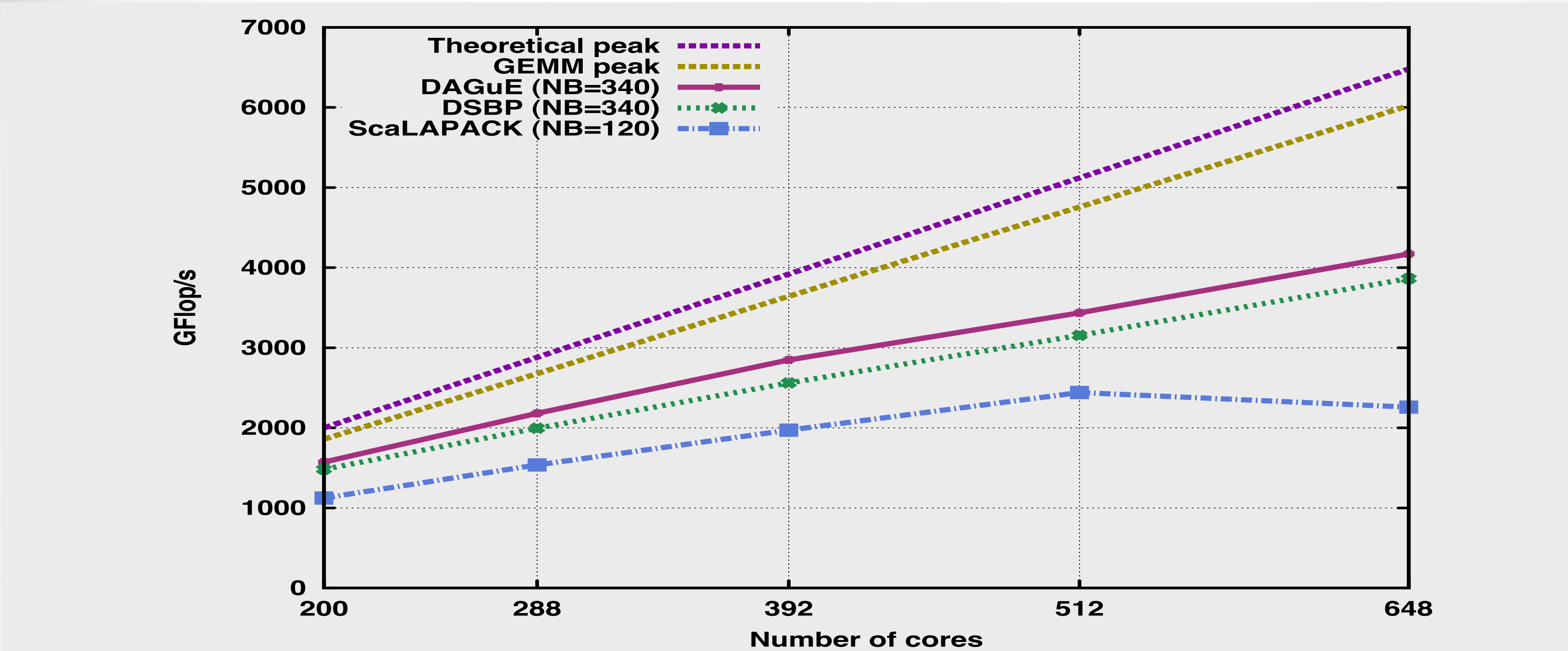

Griffon : 81 nodes, 648 cores, Infiniband 20Gbs

# Cholesky (weak scalability)

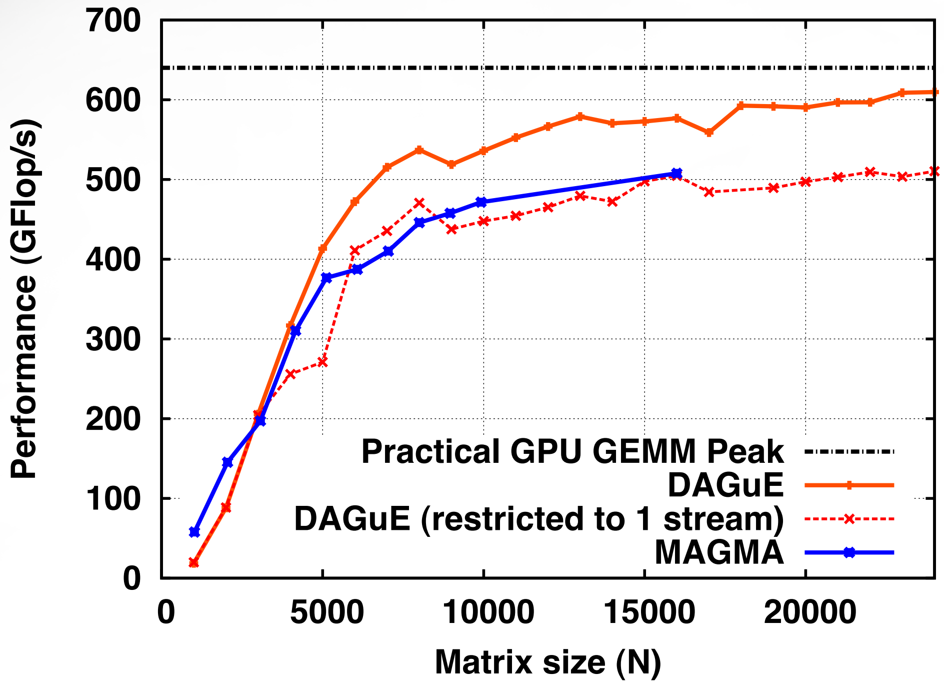



ICL UT

# Cholesky (strong scalability)

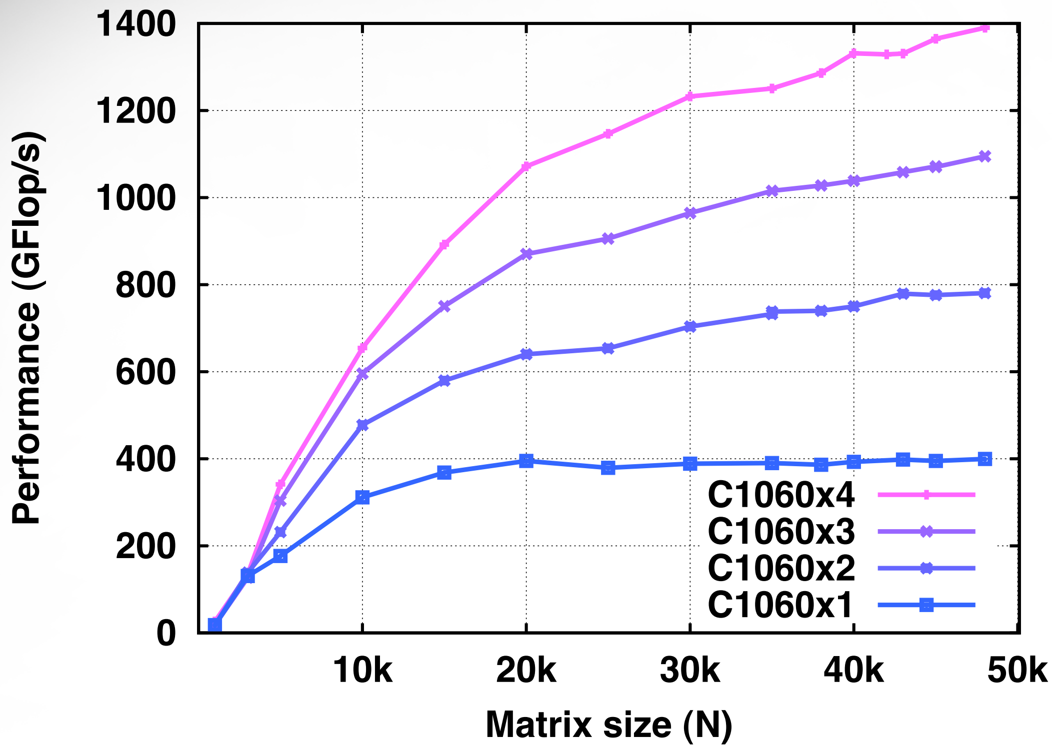


# Single GPU – single node




- Fermi (C2050)
- MAGMA 1.0

# Multiple GPU – single node



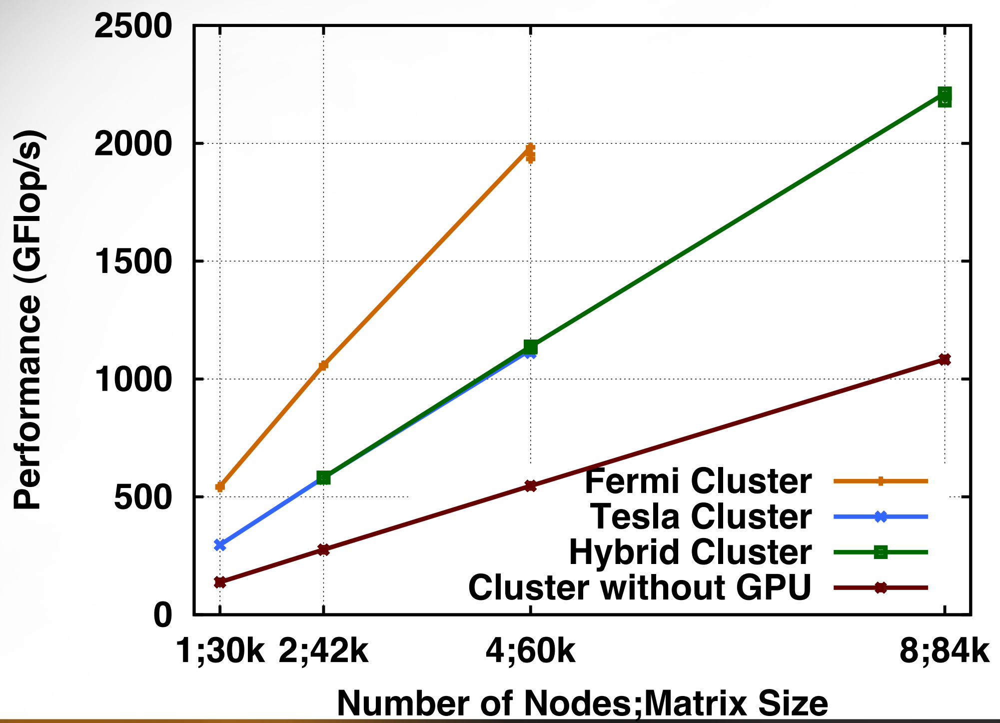

- 4xTesla (C1060)
- 8 cores

# GPU vs. Network

| Perturbation | none | remote die | same die | interleave |
|---|---|---|---|---|
| Network | - | 11.533 | 11.363 | 11.001 |
| GPU 0 push | 29.250 | 26.497 | 12.897 | 25.919 |
| GPU 1 push | 21.509 | 21.580 | 11.457 | 21.553 |
| GPU 0 pull | 13.746 | 12.897 | 11.366 | 12.060 |
| GPU 1 pull | 13.089 | 11.457 | 9.636 | 10.767 |

- The PCI bus is a critical resource shared between different components
- Scheduling cannot be done independently

# Distributed GPUs




- 4xTesla (C1060)
- 4xFermi (C2050)
- 8 cores / node

- Weak scaling

# Conclusion

- Hybrid programming (of dense LA) made easy(ier)
  - Portability: inherently take advantage of all hardware capabilities
  - Efficiency: deliver the best performance on tested algorithms
- Works well with Dense Linear Algebra with Direct Method
  - Sparse?
  - Branch and Bound?
  - Iterative Method?
- Let different people focus on different problems
  - Application developers on their algorithms
  - System developers on system issues