Making Linear Algebra Scale on Hybrid Distributed Systems: DPLASMA over PaRSEC

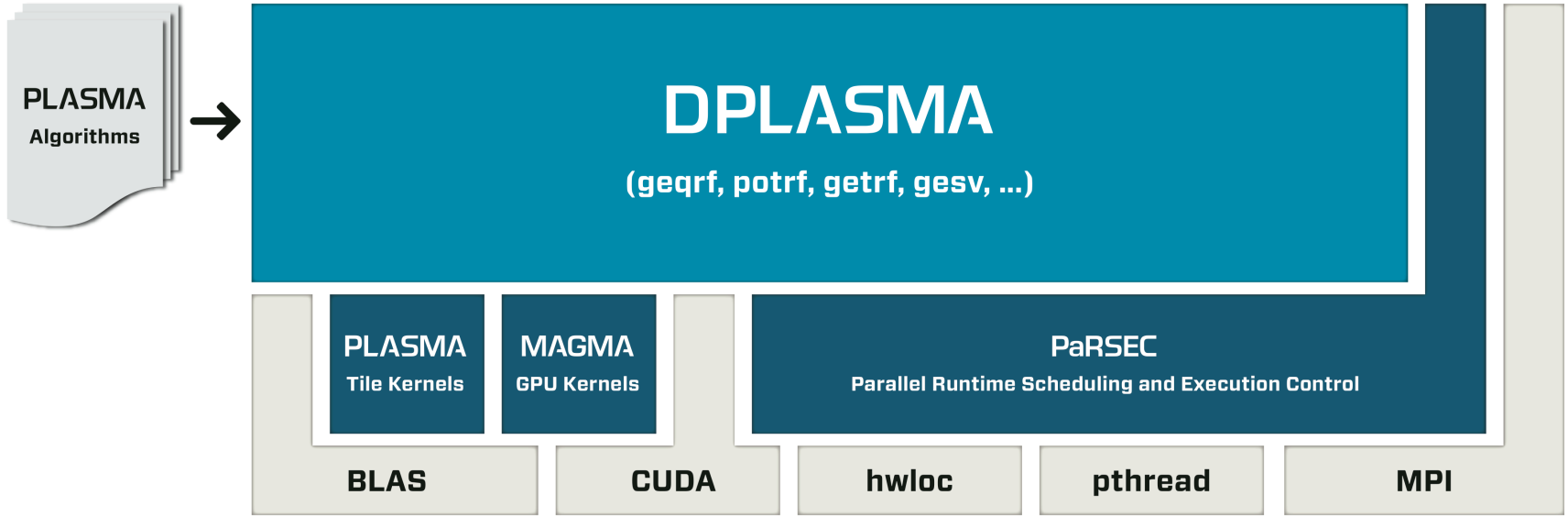
Aurelien Bouteiller

Innovative Computing Laboratory

The University of Tennessee

INNOVATIVE COMPUTING LABORATORY
THE UNIVERSITY of TENNESSEE

Dplasma


PLASMA
Algorithms
DPLASMA
(geqrf, potrf, getrf, gesv, ...)
PLASMA
Tile Kernels
MAGMA
GPU Kernels
PaRSEC
Parallel Runtime Scheduling and Execution Control
BLAS
CUDA
hwloc
pthread
MPI

Dplasma Coverage

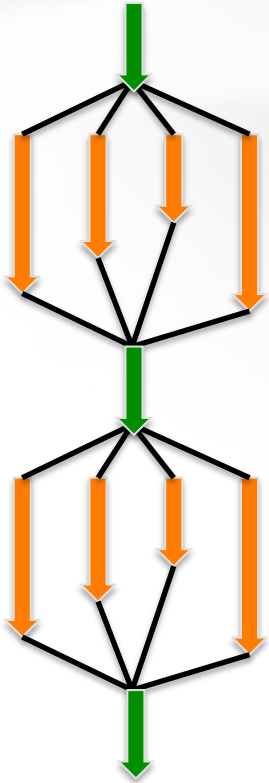
FEATURES

- Covering four precisions: double real, double complex, single real, single complex (D, Z, S, C)
- Providing ScaLAPACK-compatible interface for matrices in F77 column-major layout
- Supporting: Linux, Windows, Mac OS X, UN*X (depends on MPI, hwloc)

FUNCTIONALITY	COVERAGE
Linear Systems of Equations	Cholesky, LU (inc. pivoting, PP), LDL (prototype)
Least Squares	QR & LQ
Symmetric Eigenvalue Problem	Reduction to Band (prototype)
Level 3 Tile BLAS	GEMM, TRSM, TRMM, HEMM/SYMM, HERK/SYRK, HER2K/SYR2K

Motivation (for dataflow)

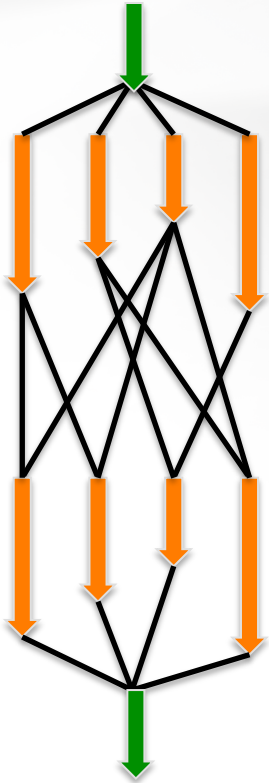
- Today software developers face
  - ~1 TFLOP of compute power per node
  - 32+ of cores, 100+ hardware threads
  - Highly heterogeneous architectures (cores + specialized cores + accelerators/coprocessors)
  - Deep memory hierarchies
  - ➔ systemic load imbalance / decreasing use of the resources
- How to harness these devices productively?
  - SPMD/BSP produces choke points, wasted wait times
  - We need to improve efficiency, power and reliability



Distributed Dataflow

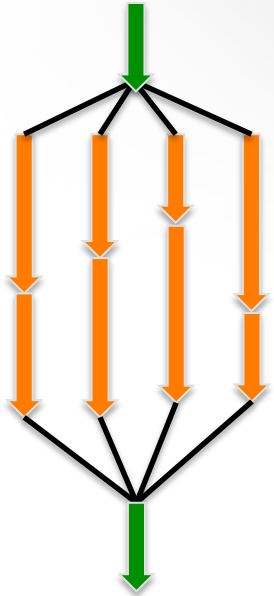
- Threads & synchronization | Processes & Messages
  - Hand written Pthreads, compiler-based OpenMP, Chapel, UPC, MPI, hybrid
- Mixed-model Hybrid programming
  - very challenging to find parallelism, to debug, to maintain and to get good performance
  - *Portably*
  - *With reasonable development efforts*

  When is it time to redesign a software?
- Increasing gaps between the capabilities of today's programming environments, the requirements of emerging applications, and the challenges of future parallel architectures



Dataflow with a unified runtime scheduler

- Algorithms need help to unleash their power
  - *Hardware specificities*: a runtime can provide portability, performance, scheduling heuristics, heterogeneity management, data movement, …
  - *Scalability*: maximize parallelism extraction, but no centralized scheduling or entire DAG unpacking: dynamic and independent discovery of the relevant portions during the execution
  - *Jitter resilience*: Do not support explicit communications, instead make them implicit and schedule to maximize overlap and load balance
- The need to express the algorithms differently
- A single model removes all BSP choke points



What is ParSEC ?

- **Par**allel **R**untime **S**cheduler & **E**xecution **C**ontrol
  - Executes a **dataflow** representation of a program
  - **Scheduler** provides
    - Automatic **load-balance between cores**
    - Harness the power of accelerators (GPU, Mic, etc)
  - Works on large scale distributed memory machines
    - **Communications** are **implicit**, **overlapped**
    - user defined Communication pattern and data-distribution

Prominent feature: *Parameterized Task Graph (PTG)*

Related Work

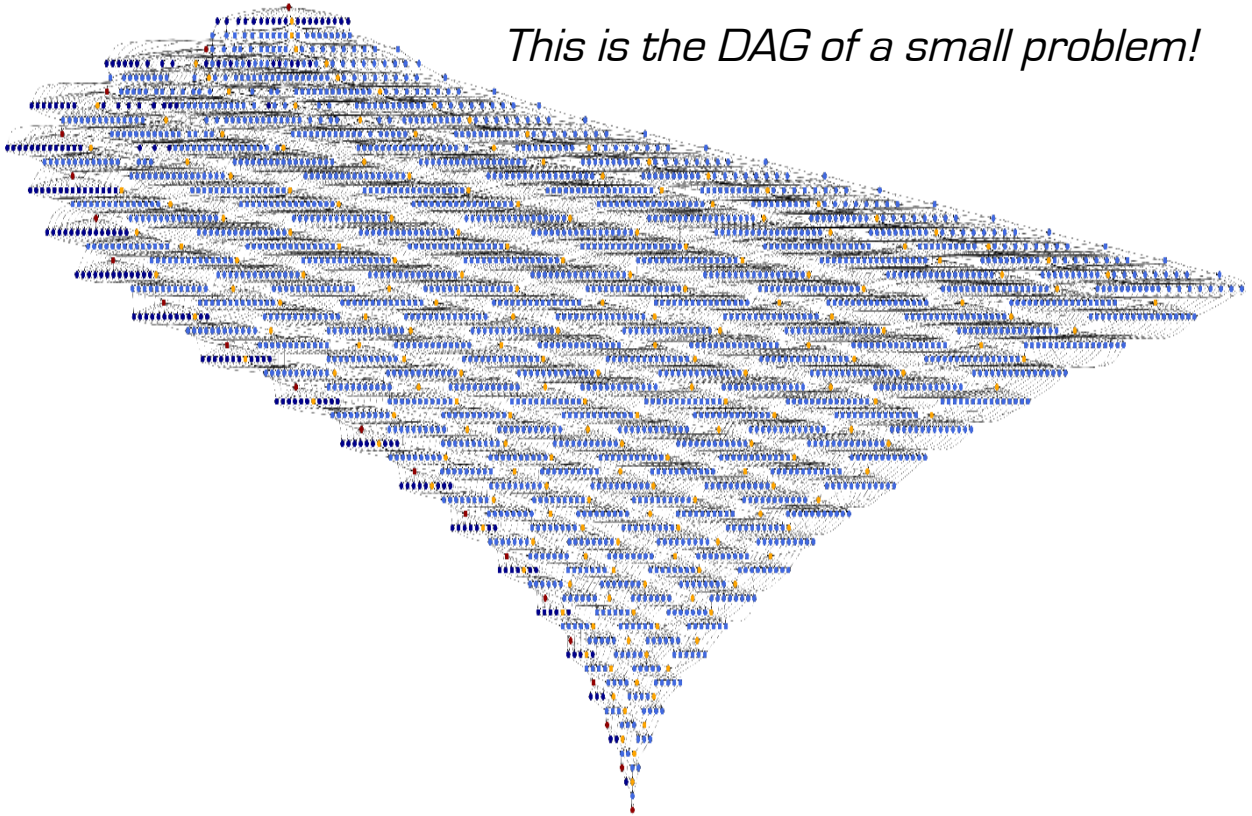
	PaRSEC	SMPss	StarPU	Charm ++	FLAME	QUARK	Tblas	PTG
Scheduling	Distr. (1/core)	Repl (1/node)	Repl (1/node)	Distr. (Actors)	w/ SuperMatrix	Repl (1/node)	Centr.	Centr.
Language	Internal or Seq. w/ Affine Loops or w/ add_task	Seq. w/ add_task	Seq. w/ add_task	Msg-Driven Objects	Internal (LA DSL)	Seq. w/ add_task	Seq. w/ add_task	Internal
Accelerator	GPU/Phi	GPU	GPU		GPU	GPU		
Availability	Public	Public	Public	Public	Public	Public	Not Avail.	Not Avail.

Early stage: ParalleX
Non-academic: Swarm, MadLINQ, CnC

All projects support Distributed and Shared Memory
(QUARK with QUARKd; FLAME with Elemental)

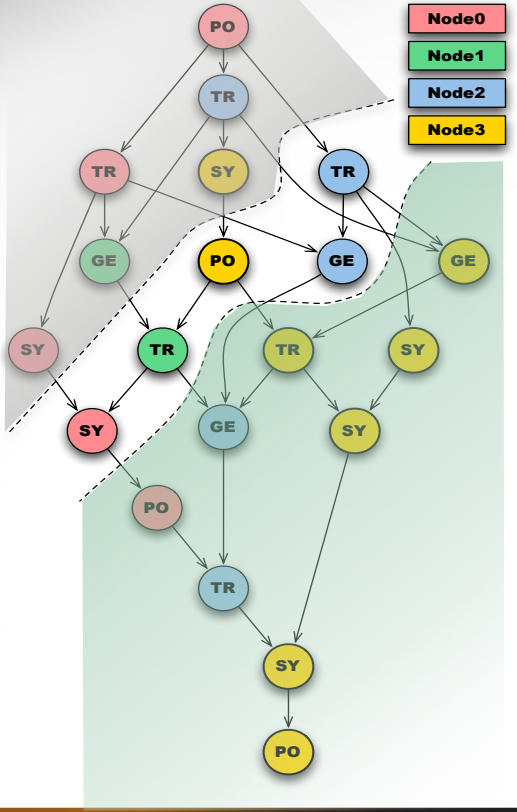
Advantages of PTG for scalable scheduling

- Typical Dataflow scheduler
  - Run the **serial code to discover the DAG** as it goes
  - Millions of vertices in the DAG, scheduler has to **visit all**, including for tasks it does not execute
- **PTG Scheduler**
  - Explores **only the neighborhood** of tasks it actually executes



This is the DAG of a small problem!

Runtime DAG scheduling



- Every process has the symbolic DAG representation
  - Only the (node local) frontier of the DAG is considered
  - Distributed Scheduling based on remote completion notifications
- Background remote data transfer automatic with overlap
- NUMA / Cache aware Scheduling
  - Work Stealing and sharing based on memory hierarchies

PTG vs DTD

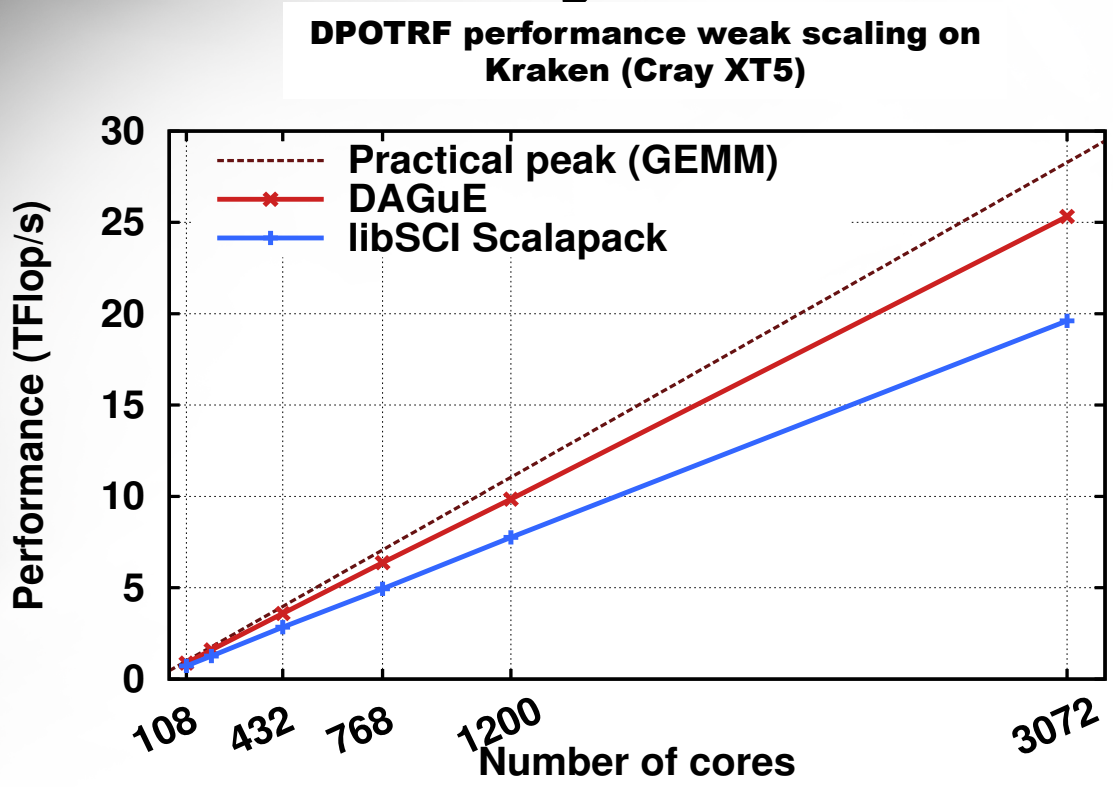
Dynamic Task Discovery

- Discover the DAG while unrolling a sequential code
- StarPU, SMP*, PLASMA, PaRSEC… popular approach
- Window-Based (the DAG is huge)
- Similar to sequential loops
- Runtime cannot do much optimization for tasks in the future

Parameterized Task Graph

- PaRSEC, PTG approach
- Problem-size independent object represents the whole DAG
- Given a task (the parameters of a terminated task), can compute the successors in O(d) (O(1))
- From these successors, can keep only the local & ready ones

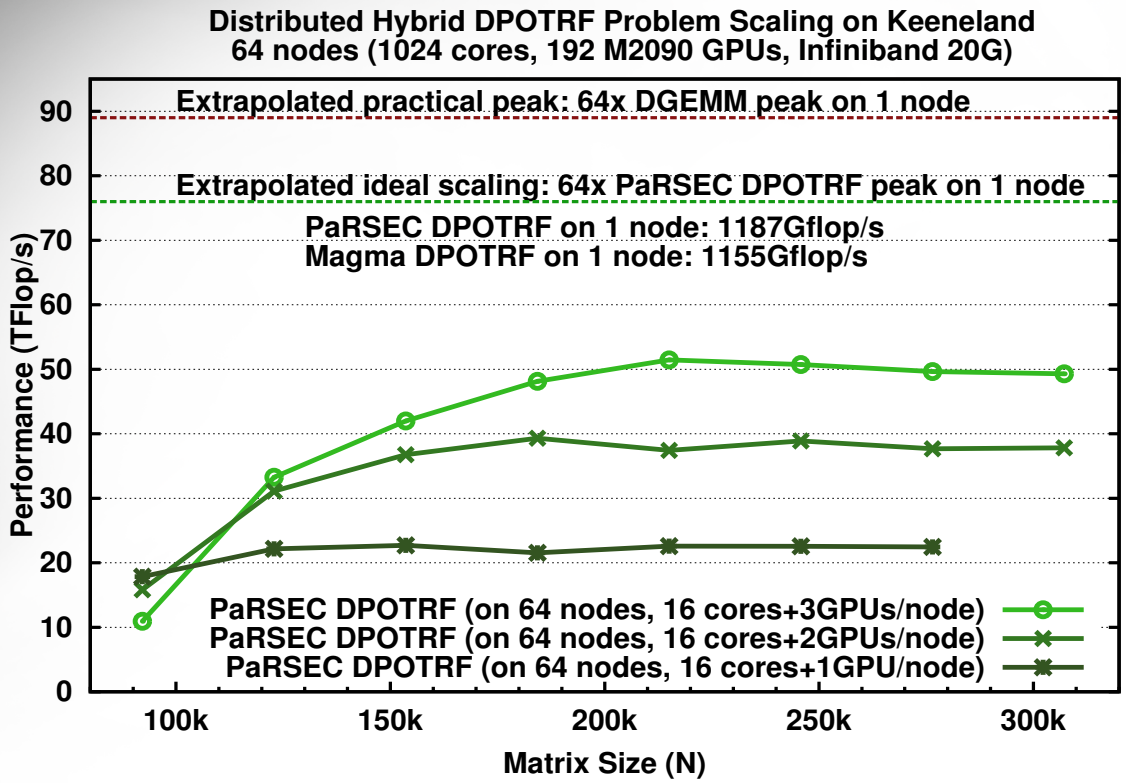
Scalability in Distributed Memory



- Parameterized Task Graph representation
- Independent distributed scheduling

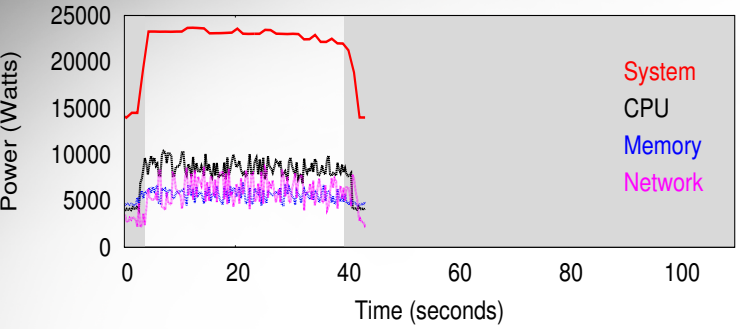
⇒ Scales well

Hybrid Clusters of Accelerators

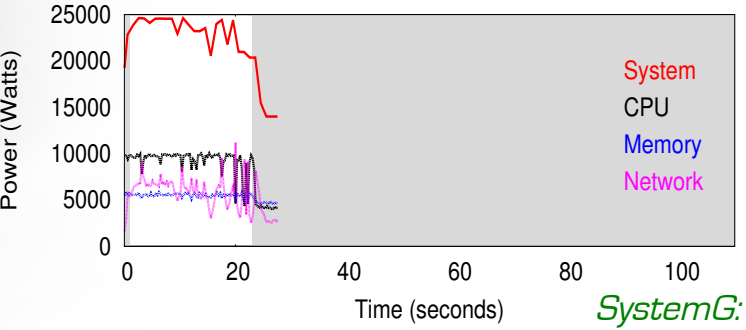


- Keeneland system: 3 GPU accelerators per node, ib20g only
  - severe computation/ bandwidth provision imbalance
- 75% of ideal scaling
- 60% of GEMM peak

Energy efficiency



(a) ScaLAPACK.



(b) DPLASMA.

Total energy consumption

# Cores	Library	Cholesky	QR
128	ScaLAPACK	192000	672000
	DPLASMA	128000	540000
256	ScaLAPACK	240000	816000
	DPLASMA	96000	540000
512	ScaLAPACK	325000	1000000
	DPLASMA	125000	576000

Work in progress with Hatem Ltaief

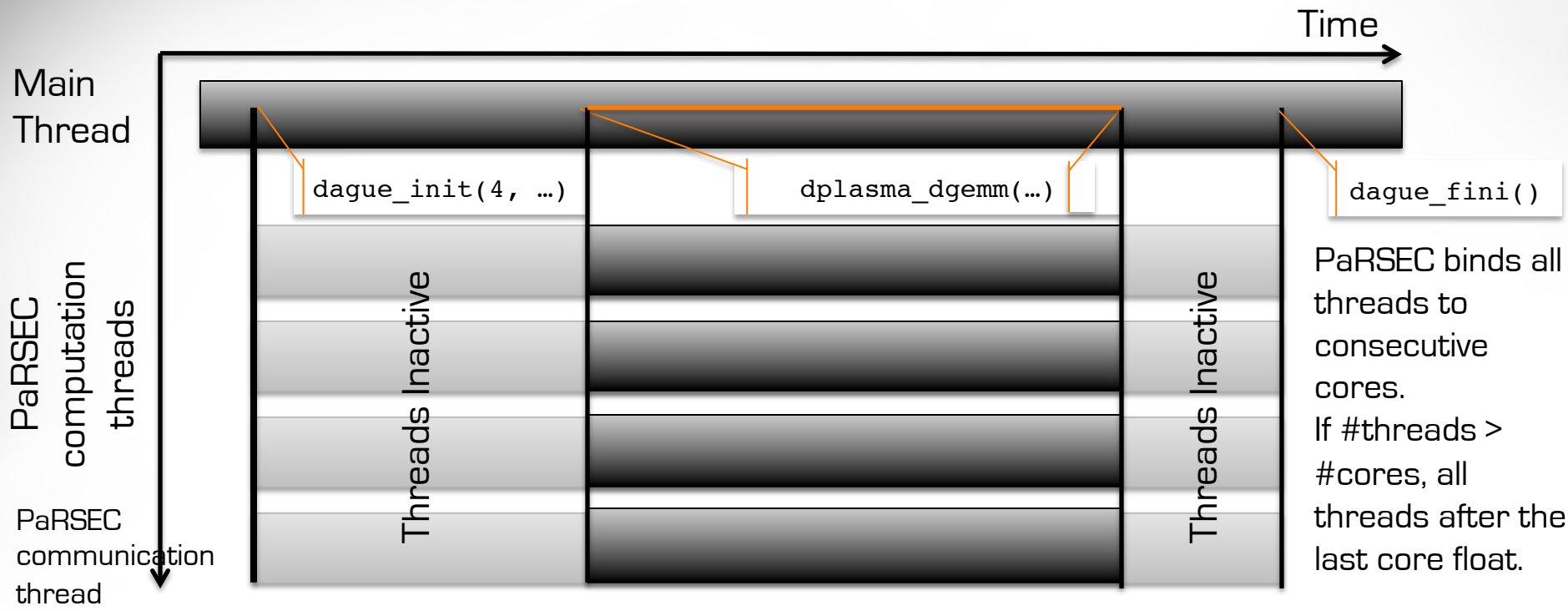
- Energy used depending on the number of cores
- Up to 62% more energy efficient while using a high performance tuned scheduling
  - Power efficient scheduling

SystemG: Virginia Tech Energy Monitored cluster
(4 cores/socket, 2 sockets/node, 324 nodes, ib40g, intel 2.8GHz,8GB/node)

The end-user perspective on Dplasma

Using Dplasma

PaRSEC / dplasma threads



PaRSEC binds all threads to consecutive cores.
If #threads > #cores, all threads after the last core float.

Using dplasma

```
#include <mpi.h>
#include <dplasma.h>
int main(int argc, char *argv[])
{
    dague_context_t *dague;
    MPI_Init(&argc, &argv);
    dague = dague_init(NBCORES_PER_NODE, &argc, &argv);
    <… distribute data A,B,C …>
    dplasma_dgemm(dague, 'N', 'T', 1.0, A, B, -1.0, C);
    <… output data …>
    dague_fini(&dague)
    MPI_Finalize();
}
```

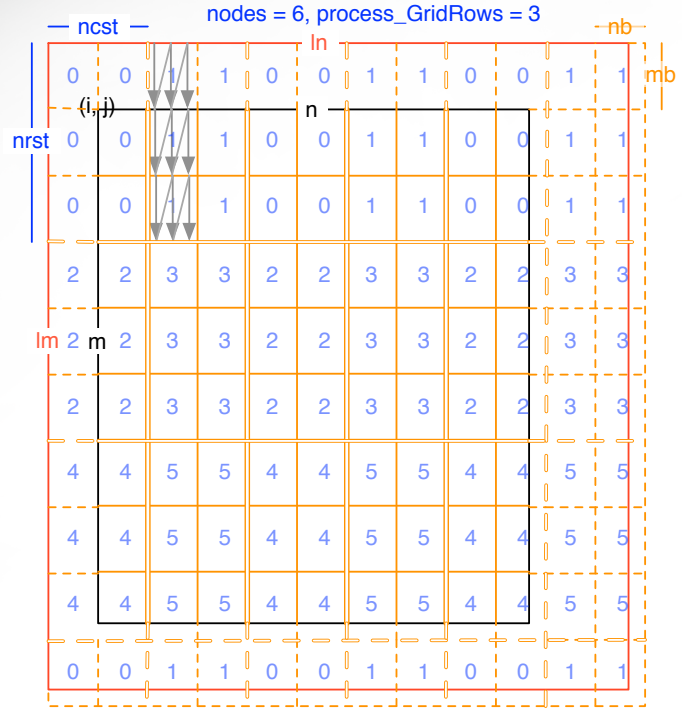
Set the number of threads PaRSEC creates

DAGuE opaque obect to be passed to all dplasma / PaRSEC calls

Data Distribution

- Similar to PLASMA, the data distribution is Tile-Based.
  - Data within tiles are stored Column Major
  - Tiles are stored consecutively, in Column Major
- The DAGuE engine is responsible to move data for the dplasma operation
  - It needs to know the data distribution.
  - This distribution is described using a dague_ddesc_t structure. This structure provides to the engine two functions to determine the node that holds a data, and on this node, the address of the data.
  - DAGuE comes with a set of builtin data distributions:
    - dplasma uses two_dim_block_cylcic and sym_two_dim_block_cyclic.

Data Distribution (2D Block Cyclic)



- Matrix of lm x ln elements of type <precision>
  - (z = double complex; c = single precision complex; d = double precision real; f = single precision real)
- Submatrix of the operation starts at element (i,j)
- Submatrix of the operation is of size m x n
- It is tiled in tiles of mb x nb elements.
- In each tile, elements are stored **Column Major**
- Tiles are distributed among processes following a 2D-Block Cyclic distribution, with blocks of nrst x ncst tiles.
- Border tiles (on the right and bottom) must be entirely allocated with padding, even if the operation applies only on the m x n submatrix.
- i must be a multiple of mb
- j must be a multiple of nb
- process_GridRows must divide nodes
- *On NUMA architectures, users are strongly encouraged to distribute the locally allocated memory among all memory banks, per units of tiles.*

Data Distribution (example)

```
two_dim_block_cyclic_t ddescA;
two_dim_block_cyclic_init(
      &ddescA, matrix_ComplexDouble,
      nbnodes, nbcores, myrank, /* For Process Grid */
      MB, NB,  /* Tile Size */
      LDA, N,  /* Matrix Size */
      0, 0,    /* Starting Point in the global matrix */
      M, N,    /* Submatrix Size */
      1, 1,    /* nrst x ncst : supertiling */
      P);      /* For Process Grid of P x (nbnodes / P) */
/* If possible, allocate memory using dague_data_allocate, to allow data placement
optimization, and DMA operations with GPUs and NIC */
ddescA.mat = dague_data_allocate(
      (size_t)ddescA.super.nb_local_tiles *
      (size_t)ddescA.super.bsiz *
      (size_t)dague_datadist_getsizeoftype(ddescA.super.mtype));
```

Calling dplasma routines non BSP mode

```
dplasma_zplrnt( dague, (tiled_matrix_desc_t *)&ddescA, SEED);
dplasma_zlaset( dague, PlasmaUpperLower, 0., 0.,
                (tiled_matrix_desc_t *)&ddescT);
```

High Level API: blocking calls to dplasma

```
dague_object_t* zgeqrt = dplasma_zgeqrt_New(
                (tiled_matrix_desc_t*)&ddescA,
                (tiled_matrix_desc_t*)&ddescT);
dague_enqueue(dague, zgeqrt);
dague_progress(dague);
dplasma_zgeqrf_Destruct( zgeqrt );
```

Low Level API:
- Create a dague object for the dplasma routine
- Enqueue the object
- Progress on it until completion
- Destruct it

Hybridization and GPU Support

```
gpu_kernel_init_dgemm(dague);

dague_gpu_data_register(dague,
          A, MT*NT, MB*NB*sizeof(double));

dplasma_dpotrf(dague, 'L', A, &info);
```

Enable GPU acceleration for the DGEMM kernel

Prepares PaRSEC GPU coherency protocol to handle matrix A

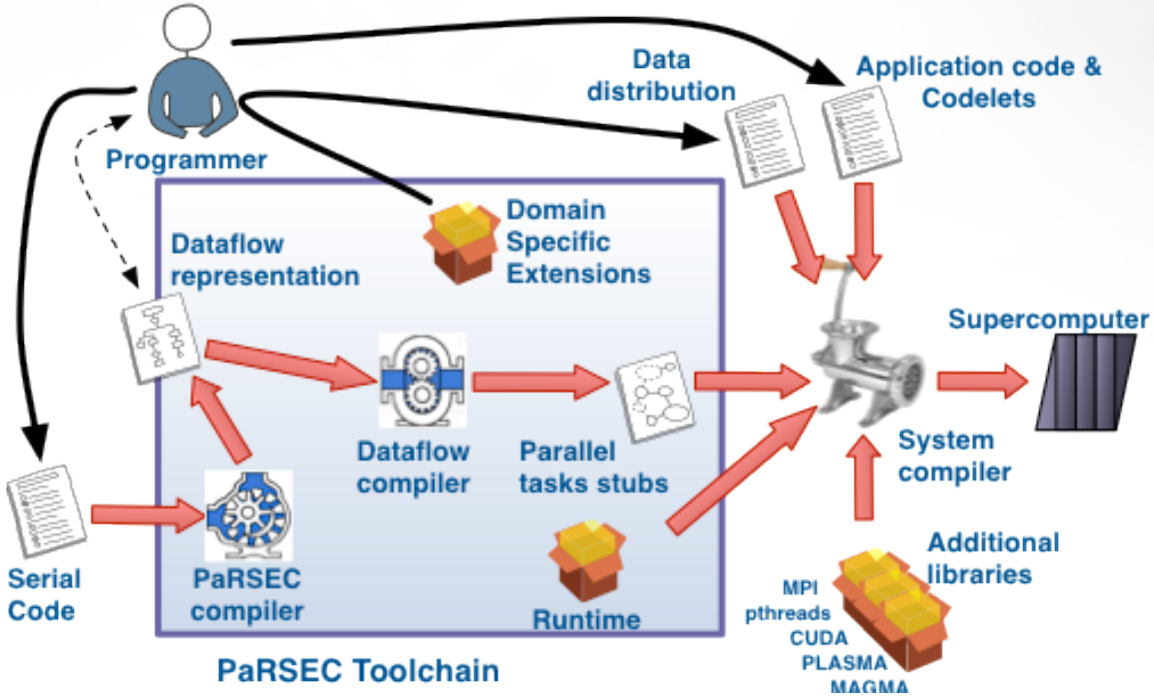
DGEMM kernels are now automatically offloaded to GPU during DPOTRF, depending on runtime load

Dare to become a power user

Creating your own Linear Algebra routine with PaRSEC

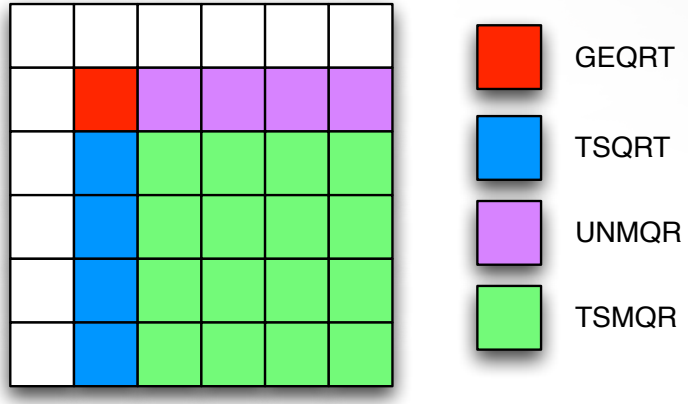
Toolchain process

1. User inputs **Serial code** (.c, affine loops, Quark like syntax)
2. Q2j extracts dependencies (generates .jdf, PTG dependencies in text)
3. .jdf can be worked on by humans (or created from scratch, it is *practical*)
4. **daguepp** transforms .jdf into C source code, injects scheduler into the task stubs
5. User picks from **supporting libs** a data distribution, common LA datatypes, may derive his own
6. Everything is fed to the system compiler, generates final supercomputer executable



Example: QR Factorization

```
FOR k = 0 .. SIZE - 1

    A[k][k], T[k][k]  <-  GEQRT( A[k][k] )

    FOR m = k+1 .. SIZE - 1

        A[k][k]|Up, A[m][k], T[m][k]  <-
            TSQRT( A[k][k]|Up, A[m][k], T[m][k] )

    FOR n = k+1 .. SIZE - 1

        A[k][n] <- UNMQR( A[k][k]|Low, T[k][k], A[k][n] )

        FOR m = k+1 .. SIZE - 1

            A[k][n], A[m][n] <-
                TSMQR( A[m][k], T[m][k], A[k][n], A[m][n] )
```



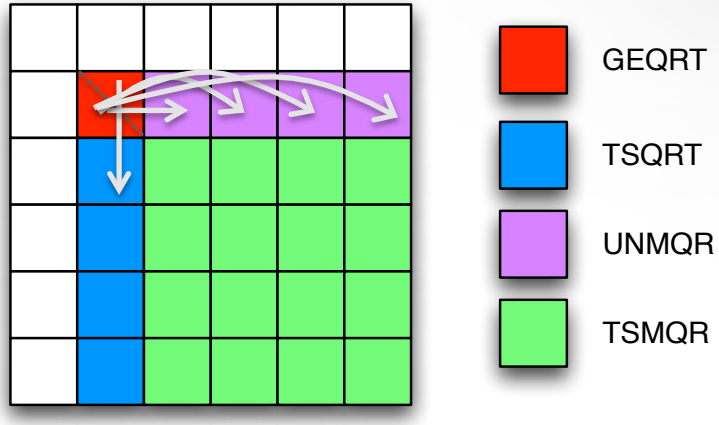
In Quark form

```
for (k = 0; k < A.mt; k++) {
  Insert_Task( zgeqrt,  A[k][k], INOUT,
                        T[k][k], OUTPUT);
  for (m = k+1; m < A.mt; m++) {
    Insert_Task( ztsqrt,  A[k][k], INOUT | REGION_D|REGION_U,
                          A[m][k], INOUT | LOCALITY,
                          T[m][k], OUTPUT);
  }
  for (n = k+1; n < A.nt; n++) {
    Insert_Task( zunmqr,  A[k][k], INPUT | REGION_L,
                          T[k][k], INPUT,
                          A[k][m], INOUT);
    for (m = k+1; m < A.mt; m++) {
      Insert_Task( ztsmqr, A[k][n], INOUT,
                           A[m][n], INOUT | LOCALITY,
                           A[m][k], INPUT,
                           T[m][k], INPUT);
    }
  }
}
```

- Sequential C code
- Annotated through specific syntax
  - Insert_Task
  - INOUT, OUTPUT, INPUT
  - REGION_L, REGION_U, REGION_D, …
  - LOCALITY

Parameterized Task Graph

```
GEQRT(k)
 /* Execution space */
 k = 0..( MT < NT ) ? MT-1 : NT-1 )
 /* Locality */
 : A(k, k)
 RW    A <- (k == 0)     ? A(k, k)
                         : A1 TSMQR(k-1, k, k)
         -> (k < NT-1)   ? A UNMQR(k, k+1 .. NT-1)   [type = LOWER]
         -> (k < MT-1)   ? A1 TSQRT(k, k+1)          [type = UPPER]
         -> (k == MT-1)  ? A(k, k)                   [type = UPPER]
 WRITE T <- T(k, k)
         -> T(k, k)
         -> (k <  NT-1) ? T UNMQR(k, k+1 .. NT-1)
 /* Priority */
 ;(NT-k)*(NT-k)*(NT-k)

BODY
   zgeqrt( A, T )
END
```



Control flow is eliminated, therefore maximum parallelism is possible

Sidenote on automatic conversion, entering the realm of dragons

The PaRSEC Frontend Compiler

Automatic conversion Example: Tile-QR

```
for k = 0 .. N-1 {
    GEQRT( RW: A[k][k] )
    for m = k+1 .. N-1 {
        TSQRT( RW: A[k][k] | U, RW: A[m][k])
    }
    for n = k+1 .. N-1 {
        UNMQR( R: A[k][k] | L, RW: A[k][n] )
        for m = k+1 .. N-1 {
            TSMQR( R: A[m][k], RW: A[k][n], RW: A[m][n] )
        }
    }
}
```

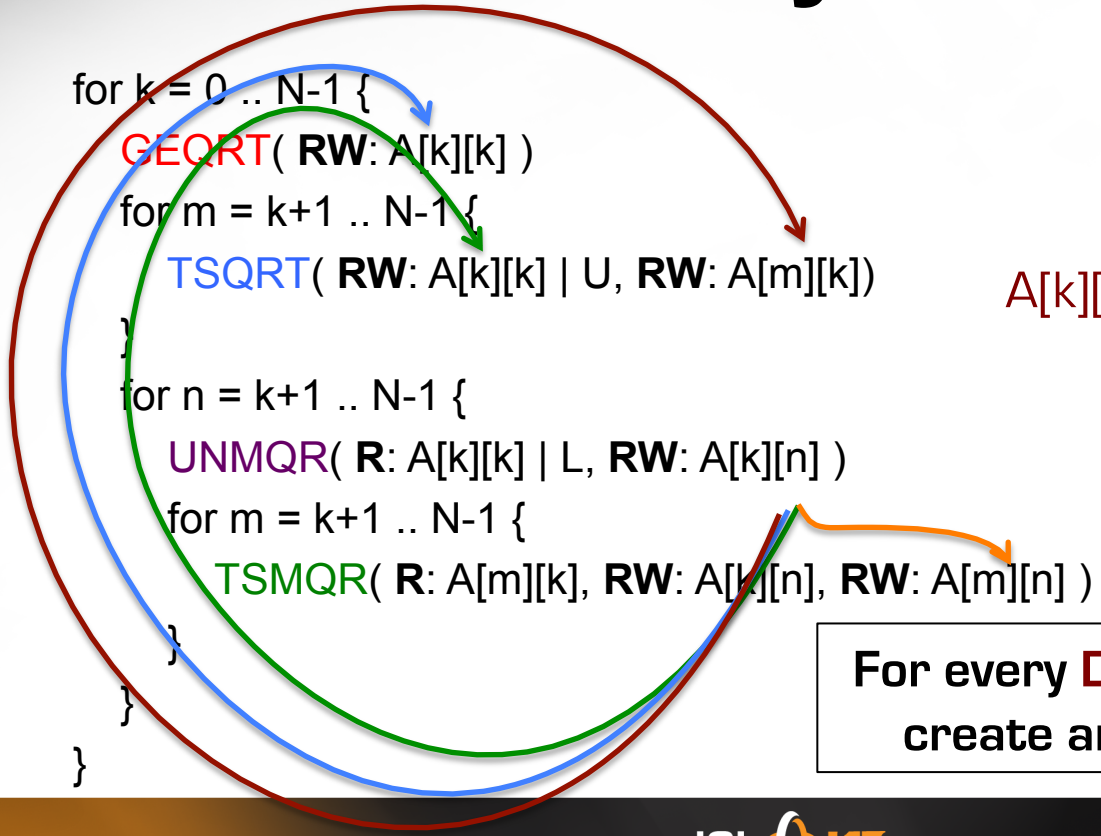
Data-Flow Analysis

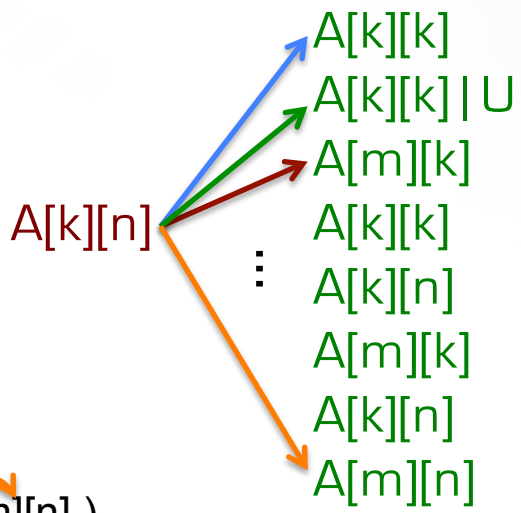
```
for k = 0 .. N-1 {
    GEQRT( RW: A[k][k] )
    for m = k+1 .. N-1 {
        TSQRT( RW: A[k][k] | U, RW: A[m][k])
    }
    for n = k+1 .. N-1 {
        UNMQR( R: A[k][k] | L, RW: A[k][n] )
        for m = k+1 .. N-1 {
            TSMQR( R: A[m][k], RW: A[k][n], RW: A[m][n] )
        }
    }
}
```

Recording DEFs and USEs

DEFs	USEs
A[k][k]	A[k][k]
A[k][k] | U	A[k][k] | U
A[m][k]	A[m][k]
A[k][n]	A[k][k]
A[k][n]	A[k][n]
A[m][n]	A[m][k]
	A[k][n]
	A[m][n]

Data-Flow Analysis





For every DEF-USE pair we will create an *Omega* Relation

Data-Flow Analysis

```
for k = 0 .. N-1 {
   GEQRT( RW: A[k][k] )
   for m = k+1 .. N-1 {
      TSQRT( RW: A[k][k] | U, RW: A[m][k])
   }
   for n = k+1 .. N-1 {
      UNMQR( R: A[k][k] | L, RW: A[k][n] )
      for m = k+1 .. N-1 {
         TSMQR( R: A[m][k], RW: A[k][n], RW: A[m][n] )
      }
   }
}
```

R := {[k,n]->[k',m] }

A[k][n] →

A[k][k]
A[k][k] | U
A[m][k]
A[k][k]
A[k][n]
A[m][k]
A[k][n]
A[m][n]

input and output parameters come from the loops enclosing the DEF/USE

Data-Flow Analysis

```
for k = 0 .. N-1 {
    GEQRT( RW: A[k][k] )
    for m = k+1 .. N-1 {
        TSQRT( RW: A[k][k] | U, RW: A[m][k])
    }
    for n = k+1 .. N-1 {
        UNMQR( R: A[k][k] | L, RW: A[k][n] )
        for m = k+1 .. N-1 {
            TSMQR( R: A[m][k], RW: A[k][n], RW: A[m][n] )
        }
    }
}
```

R := {[k,n]->[k',m] : k=m && n=k' }

A[k][n] →

A[k][k]
A[k][k] | U
A[m][k]
A[k][k]
A[k][n]
A[m][k]
A[k][n]
A[m][n]

Add conditions demanded by matrix indices (A[k][n] -> A[m][k'])

Data-Flow Analysis

```
for k = 0 .. N-1 {
    GEQRT( RW: A[k][k] )
    for m = k+1 .. N-1 {
        TSQRT( RW: A[k][k] | U, RW: A[m][k])
    }
    for n = k+1 .. N-1 {
        UNMQR( R: A[k][k] | L, RW: A[k][n] )
        for m = k+1 .. N-1 {
            TSMQR( R: A[m][k], RW: A[k][n], RW: A[m][n] )
        }
    }
}
```

A[k][n] →

A[k][k]
A[k][k] | U
A[m][k]
A[k][k]
A[k][n]
A[m][k]
A[k][n]
A[m][n]

R := {[k,n]->[k',m] : k=m && n=k' &&
0<=k<=N-1 && k+1<=n<=N-1 &&
0<=k'<=N-1 && k'+1<=m<=N-1 }

Add conditions demanded by loop bounds

Data-Flow Analysis

```
for k = 0 .. N-1 {
    GEQRT( RW: A[k][k] )
    for m = k+1 .. N-1 {
        TSQRT( RW: A[k][k] | U, RW: A[m][k])
    }
    for n = k+1 .. N-1 {
        UNMQR( R: A[k][k] | L, RW: A[k][n] )
        for m = k+1 .. N-1 {
            TSMQR( R: A[m][k], RW: A[k][n], RW: A[m][n] )
        }
    }
}
```

A[k][n] →
A[k][k]
A[k][k] | U
A[m][k]
A[k][k]
A[k][n]
A[m][k]
A[k][n]
A[m][n]

R := {[k,n]->[k',m] : k=m && n=k' &&
0<=k<=N-1 && k+1<=n<=N-1 &&
0<=k'<=N-1 && k'+1<=m<=N-1 &&
k<k' }

Add conditions demanded by control flow (k' must execute after k)

Data-Flow Analysis

```
for k = 0 .. N-1 {
   GEQRT( RW: A[k][k] )
   for m = k+1 .. N-1 {
      TSQRT( RW: A[k][k] | U, RW: A[m][k])
   }
   for n = k+1 .. N-1 {
      UNMQR( R: A[k][k] | L, RW: A[k][n] )
      for m = k+1 .. N-1 {
         TSMQR( R: A[m][k], RW: A[k][n], RW: A[m][n] )
      }
   }
}
```

A[k][n] ⋮

A[k][k]
A[k][k] | U
A[m][k]
A[k][k]
A[k][n]
A[m][k]
A[k][n]
A[m][n]

R := {[k,n]->[k',m] : k=m && n=k' &&
0<=k<=N-1 && k+1<=n<=N-1 &&
0<=k'<=N-1 && k'+1<=m<=N-1 &&
k<k' }

Check if the Relation is satisfiable (this one is not)

From Omega Relations to PTG textual representation

1. Keep satisfiable **flow** dependencies (Read-after-Write)
2. Keep satisfiable **output** dependencies (Write-after-Write)
3. Keep satisfiable **anti**-dependencies (Write-after-Read)
4. Subtract output from flow dependencies
   - Convert resulting Relations into outgoing communication
   - Invert the Relations for incoming communication
5. Subtract transitive edges from anti-dependencies
   - Convert resulting Relations into control messages

Using the brain, when automation fails

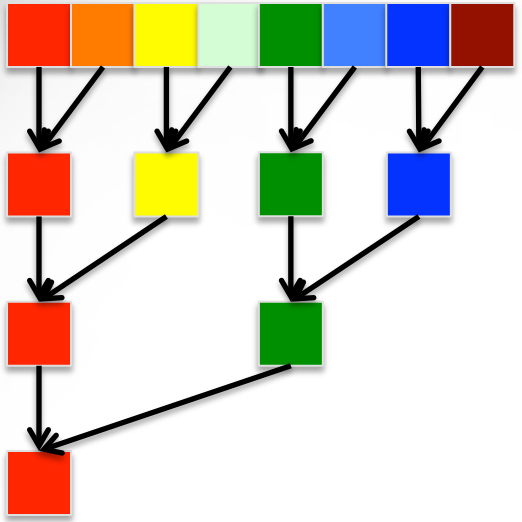
Writing PTG directly

Example: Reduction Operation

- Reduction: 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



- Reduction: apply a user defined operator on each data and store the result in a single location.

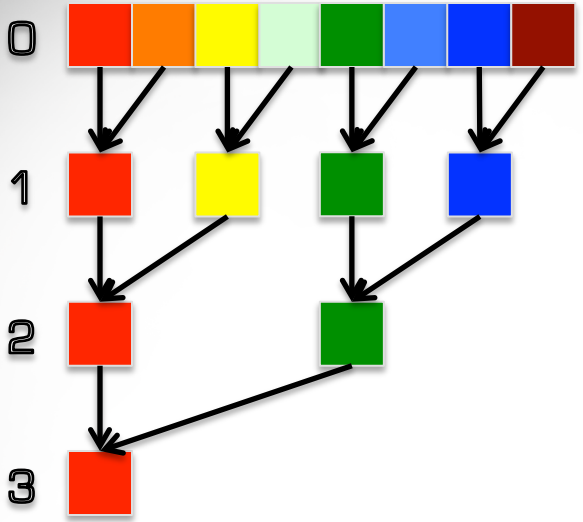
  (Suppose the operator is associative and commutative)

```
for(s = 1; s < N/2; s = 2*s)
  for(i = 0; i < N-s; i += s)
    operator(V[i], V[i+s])
```

Issue: Non-affine loops lead to non-polyhedral array accessing

Omega cannot solve the dependencies

Example: Reduction Operation

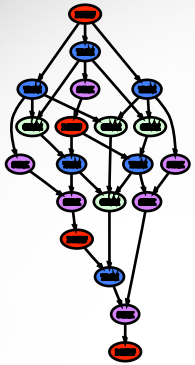


Solution: Hand-writing of the data dependency using the intermediate Data Flow representation

```
reduce(l, p)
: V(p)
l = 1 .. depth+1
p = 0 .. (MT / (1<<l))

RW   A <- (1 == l) ? V(2*p)
                   : A reduce( l-1, 2*p )
       -> ((depth+1) == l) ? V(0)
       -> (0 == (p%2))? A reduce(l+1, p/2)
                        : B reduce(l+1, p/2)
READ B <- ((p*(1<<l) + (1<<(l-1))) > MT) ? V(0)
       <- (1 == l) ? V(2*p+1)
       <- (1 != l) ? A reduce( l-1, p*2+1 )
BODY
  operator(A, B);
END
```

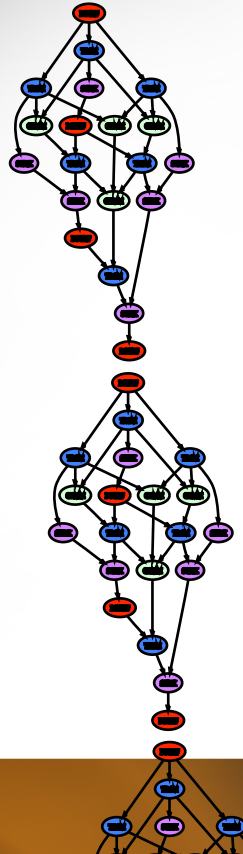
Hand written, automatically generated and BSP routines can coexist in the same application: ***incremental upgrade path***

Composition



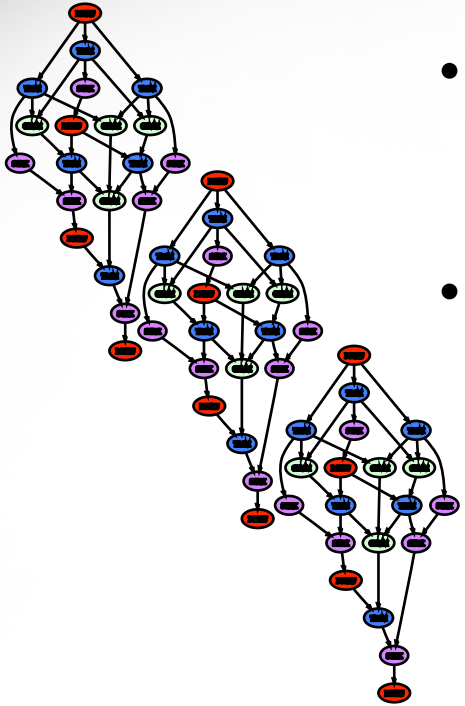
- An algorithm is a series of PTG operations with data dependencies inbetween
- A sequential composition limits the parallelism due to strict synchronizations
  - Connecting Outputs and Inputs of differnet PTGs, we can loosen the synchronizations and transform them in data dependencies (at runtime)

Composition



- An algorithm is a series of PTG operations with data dependencies inbetween
- A sequential composition limits the parallelism due to strict synchronizations
  - Connecting Outputs and Inputs of differnet PTGs, we can loosen the synchronizations and transform them in data dependencies (at runtime)

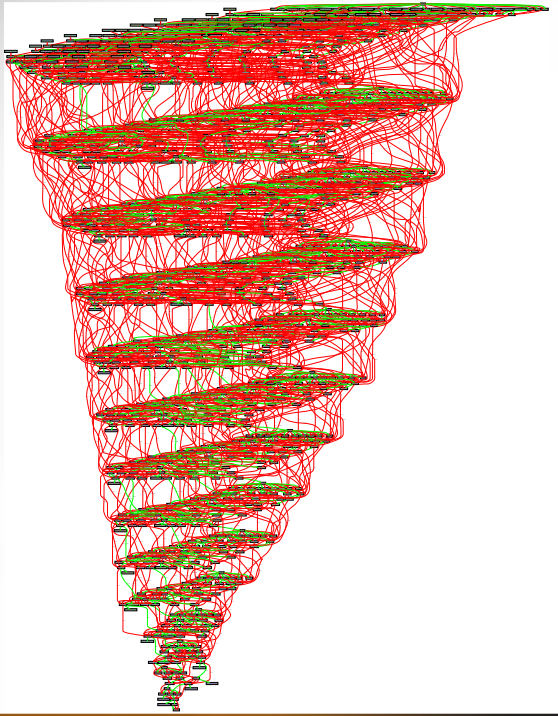
Composition

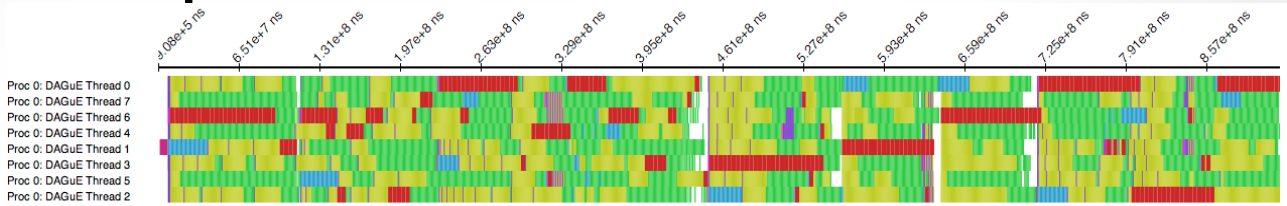


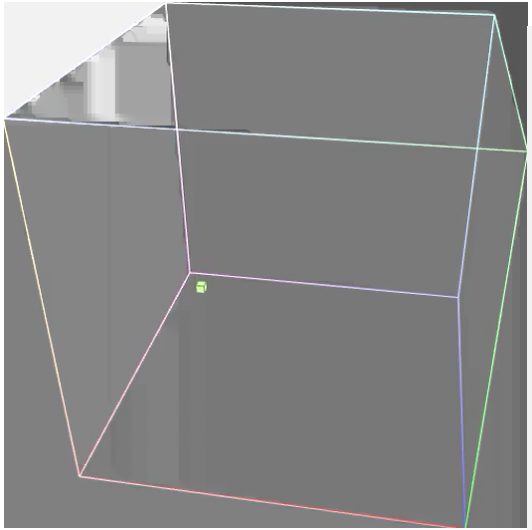
- An algorithm is a series of PTG operations with data dependencies inbetween
- A sequential composition limits the parallelism due to strict synchronizations
  - Connecting Outputs and Inputs of differnet PTGs, we can loosen the synchronizations and transform them in data dependencies (at runtime)

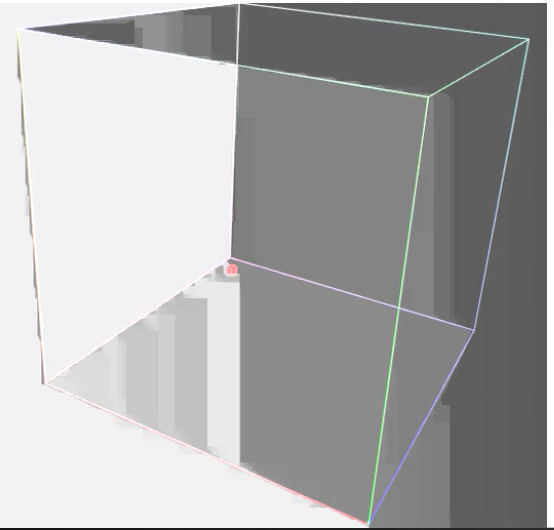
Analysis Tools

benefitting from structured parallelism







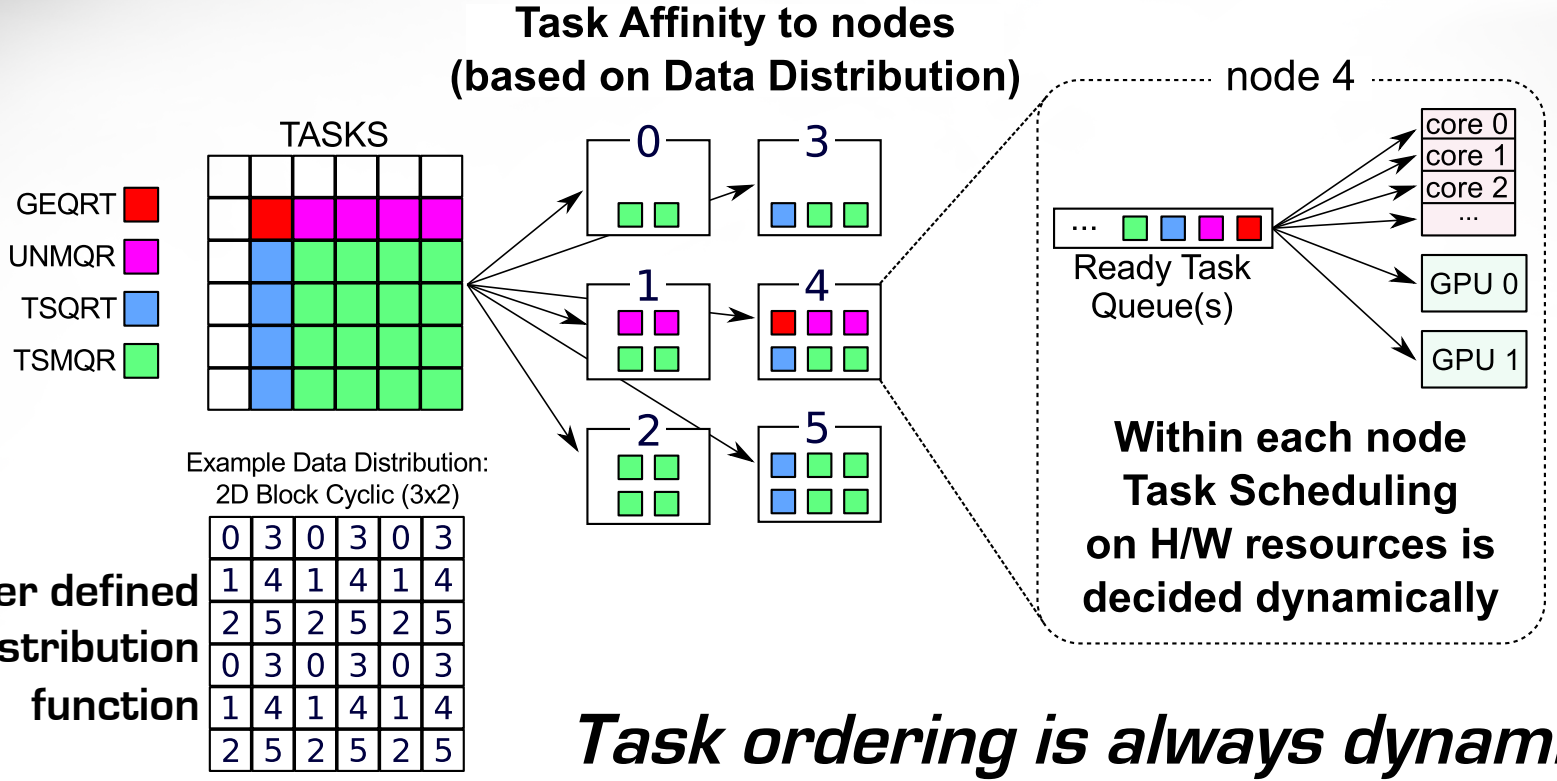


Hermitian Band Diagonal; 16x16 tiles

Opening the blackbox

PaRSEC Runtime System

Dynamic / Static Task Placement

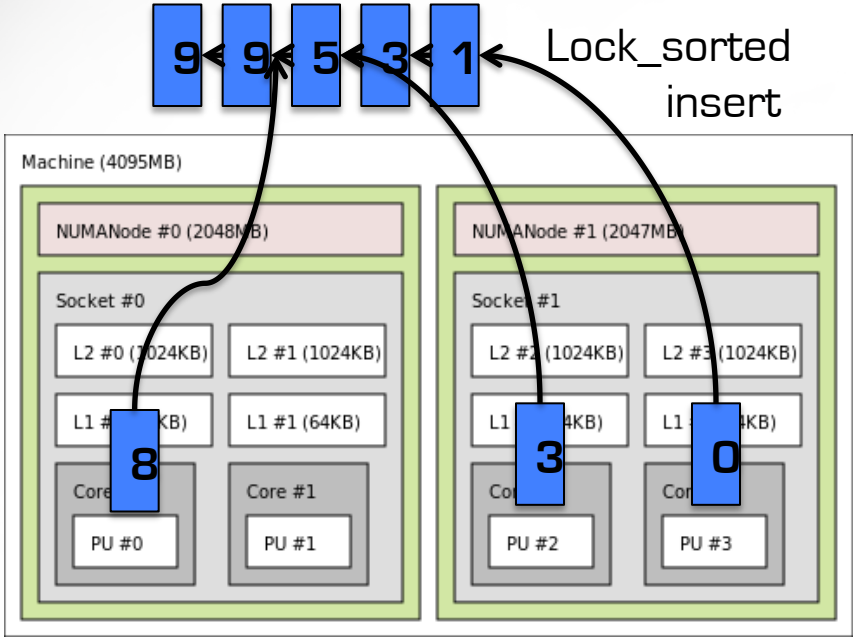


Task ordering is always dynamic

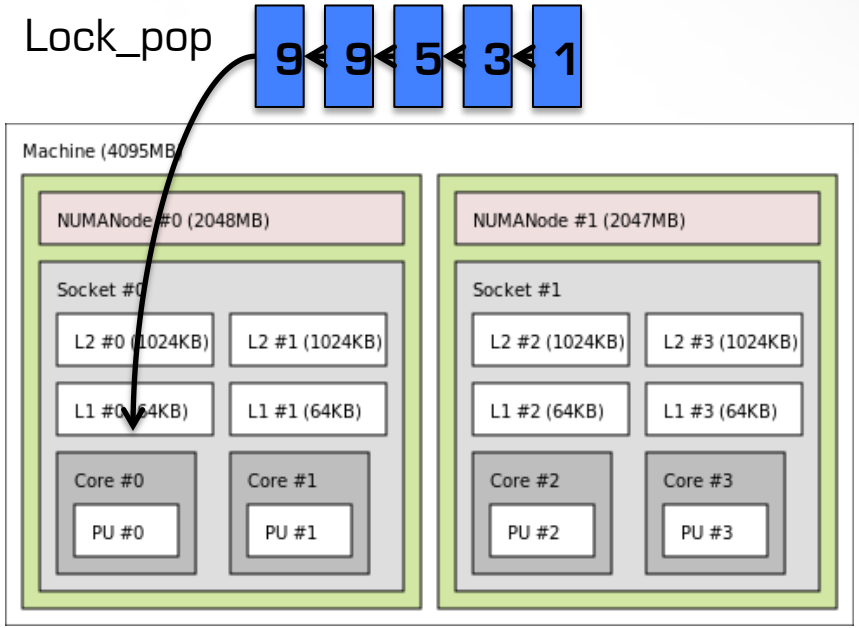
Absolute Priorities

Honor critical path order, expensive lock, poor locality

Scheduling (push task)



Selecting (pop task)

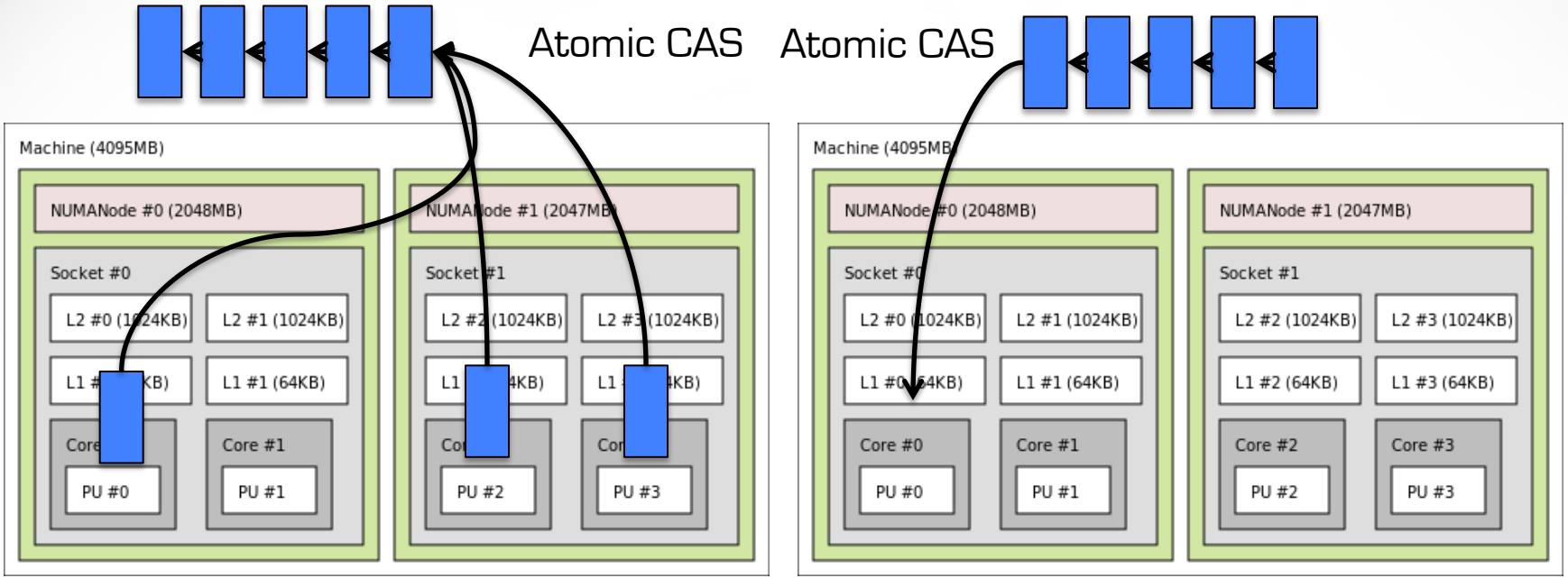


Global Dequeue

locality for one immediate successor, efficient CAS atomics

Scheduling (push task)

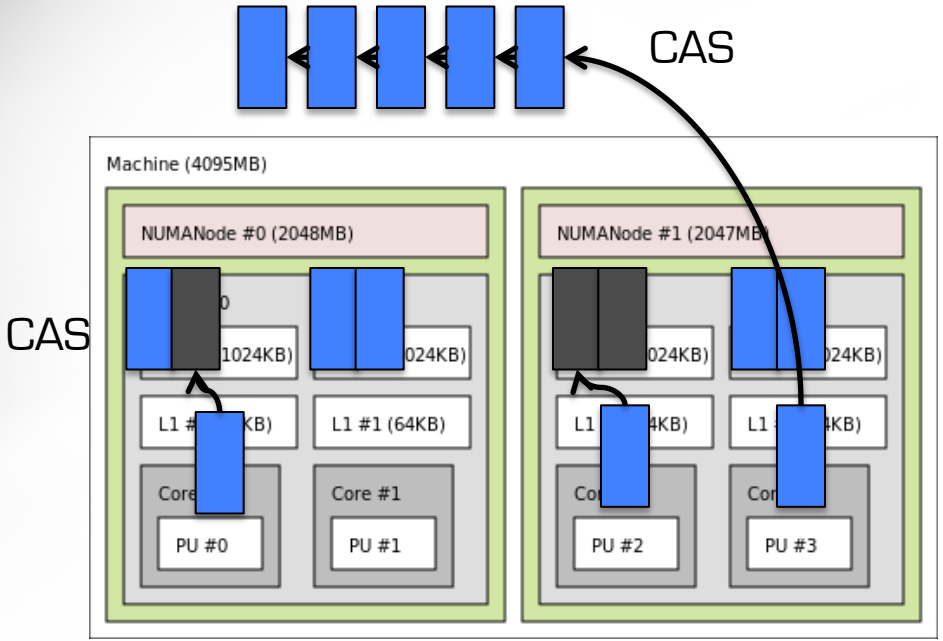
Selecting (pop task)



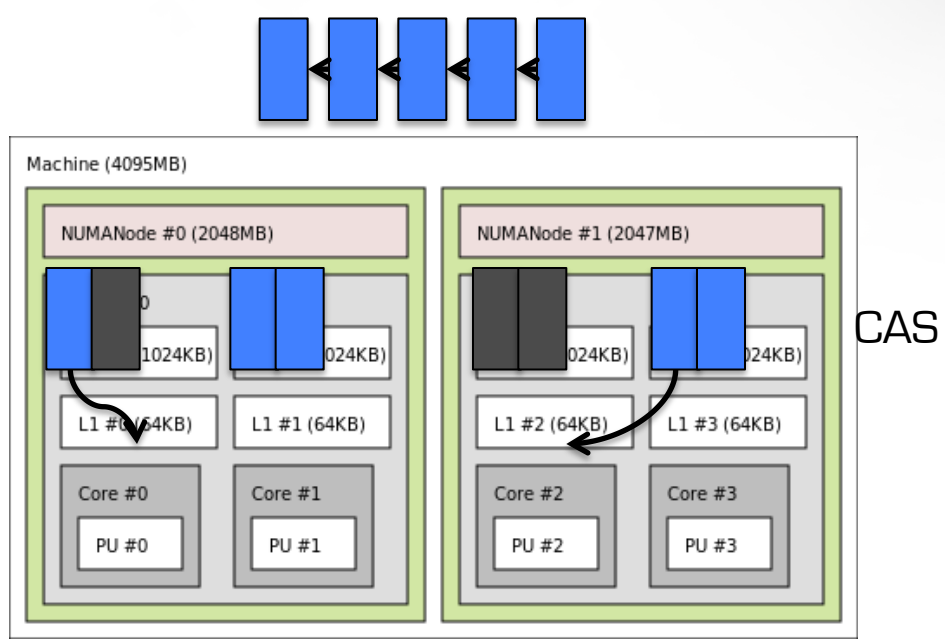
Local Flat Queues (1)

Enqueue localy, dequeue if full

Scheduling (push task)



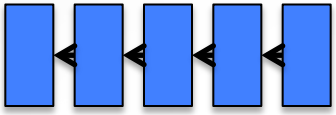
Selecting (pop task)



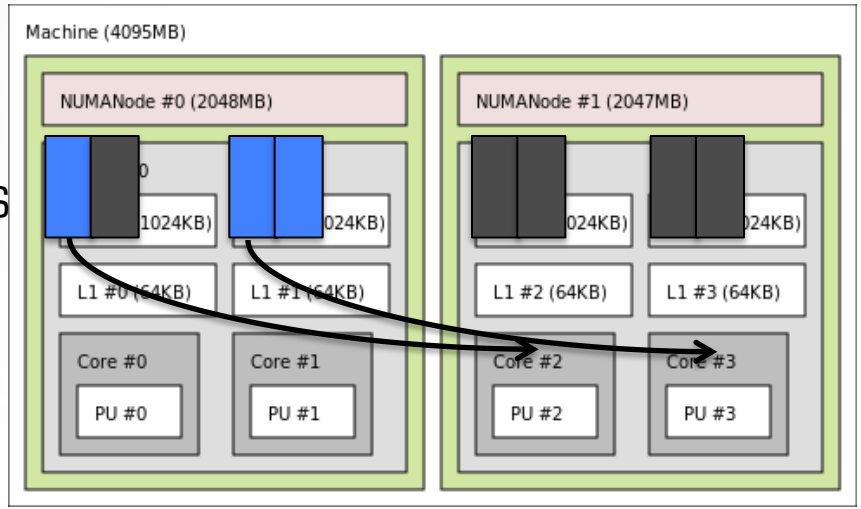
Local Flat Queues (2)

Steal from closest neighbor queue, dequeue if nothing found

Selecting (pop task)

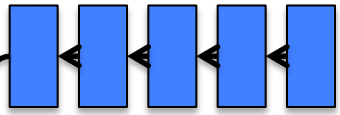


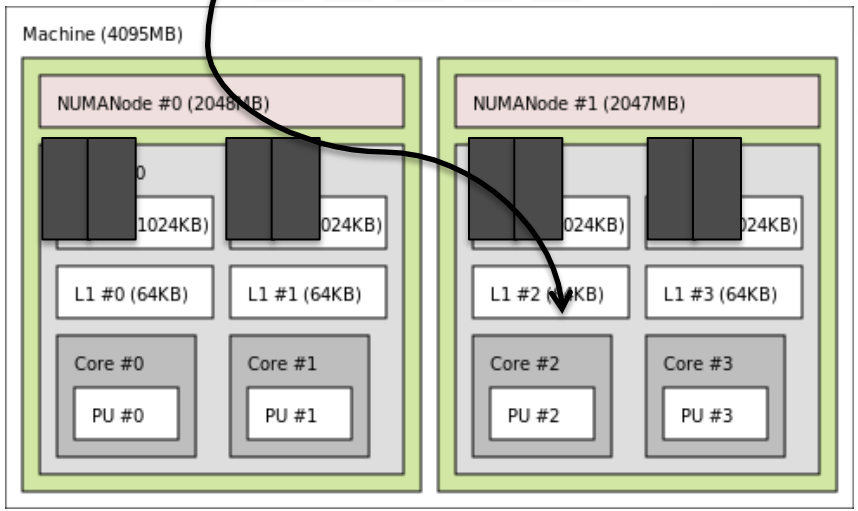
CAS



Selecting (pop task)

CAS

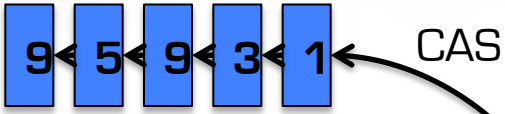




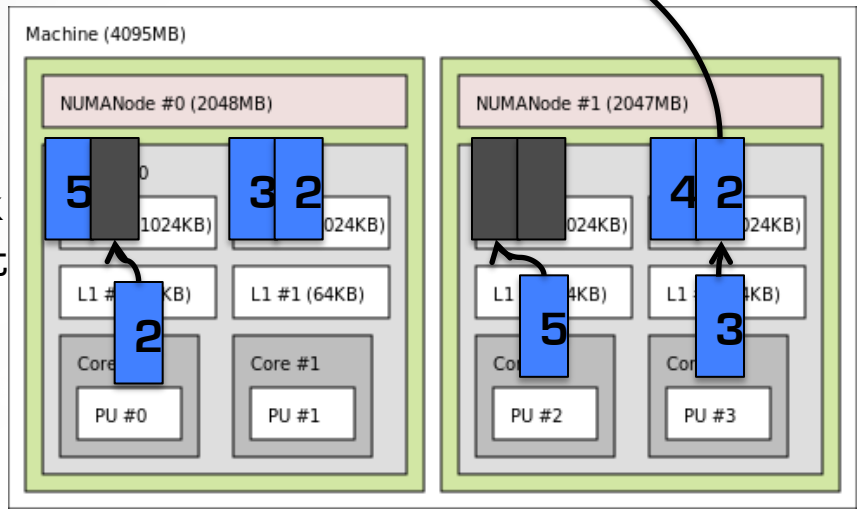
Priority Based Queues

Same as Local Flat queues, but local queues are sorted

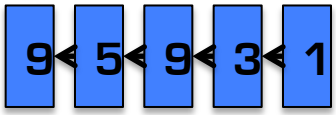
Scheduling (push task)

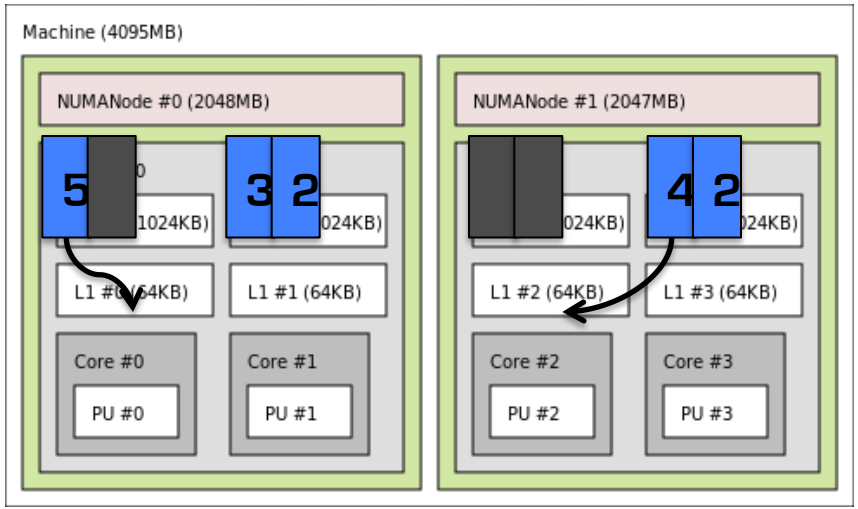


lock
sort



Selecting (pop task)

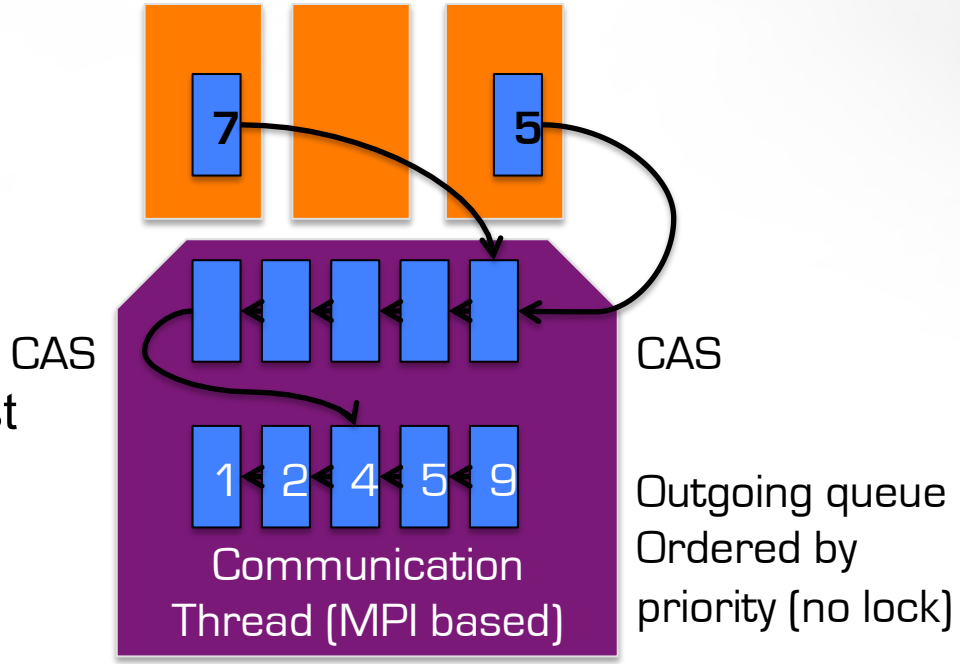




lock
sort

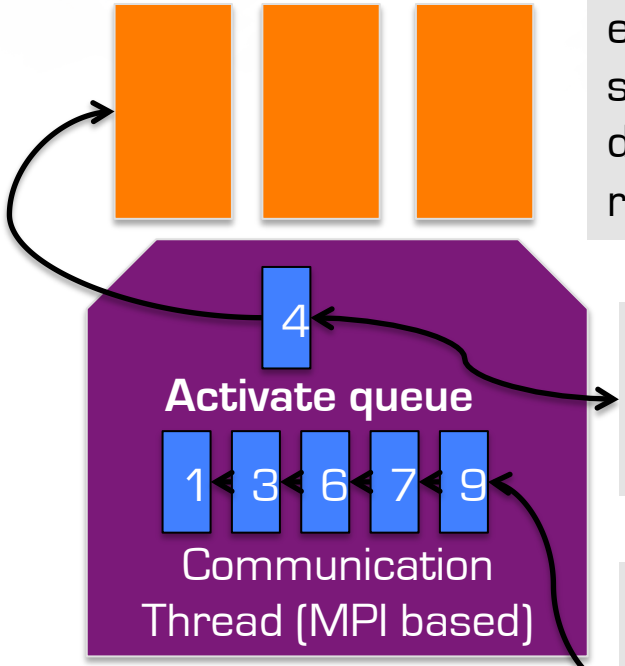
Communication Engine

- Communications asynchornous (delegated to a supplementary thread)
- MPI implementation, emulates Active Messages on pt-2-pt
- Threads post task completions to the communication engine
- Automatic detection of broadcast pattern
- Communication engine has (outgoing) flow control, reorders sends according to target tasks priority



Communication Engine

- Communications asynchornous (delegated to a supplementary thread)
- MPI implementation, emulates Active Messages on pt-2-pt
- Communication engine executes completion hook of task indicated by ACTIVATE message
- RDMA Get the necessary data
- Each Get completion triggers the normal scheduling hooks (thus schedules dependent tasks)



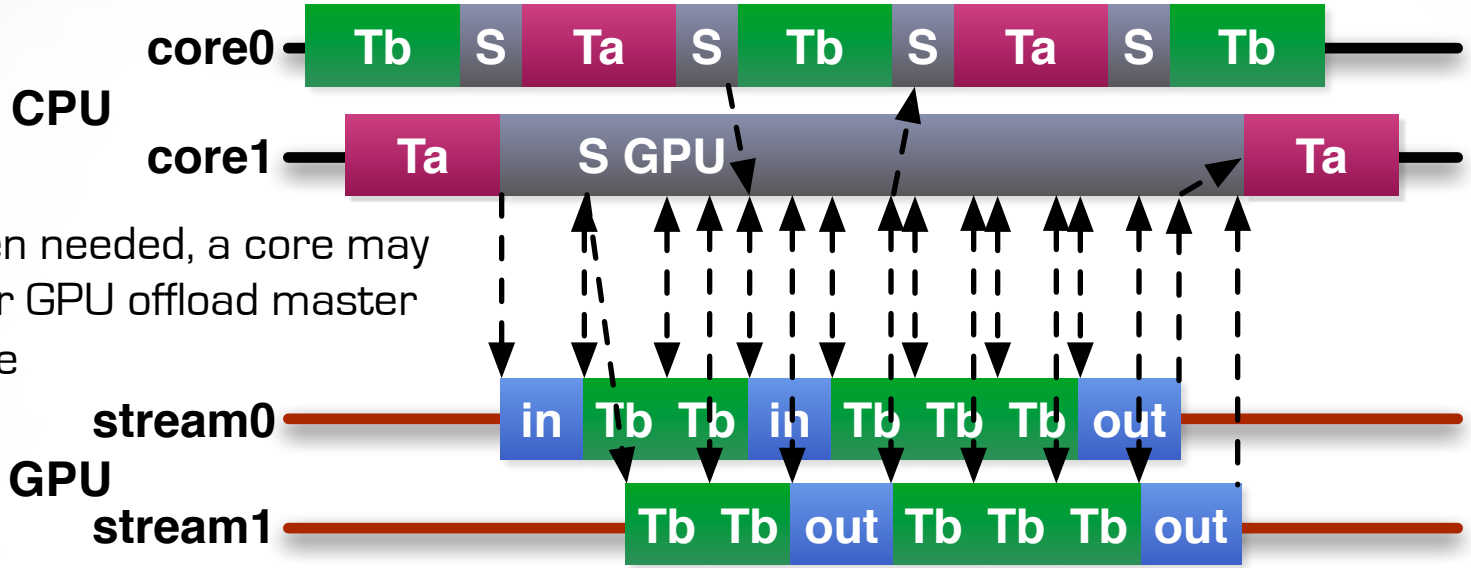
Completion of each GET schedules dependents of received data

GET necessary data from source rank

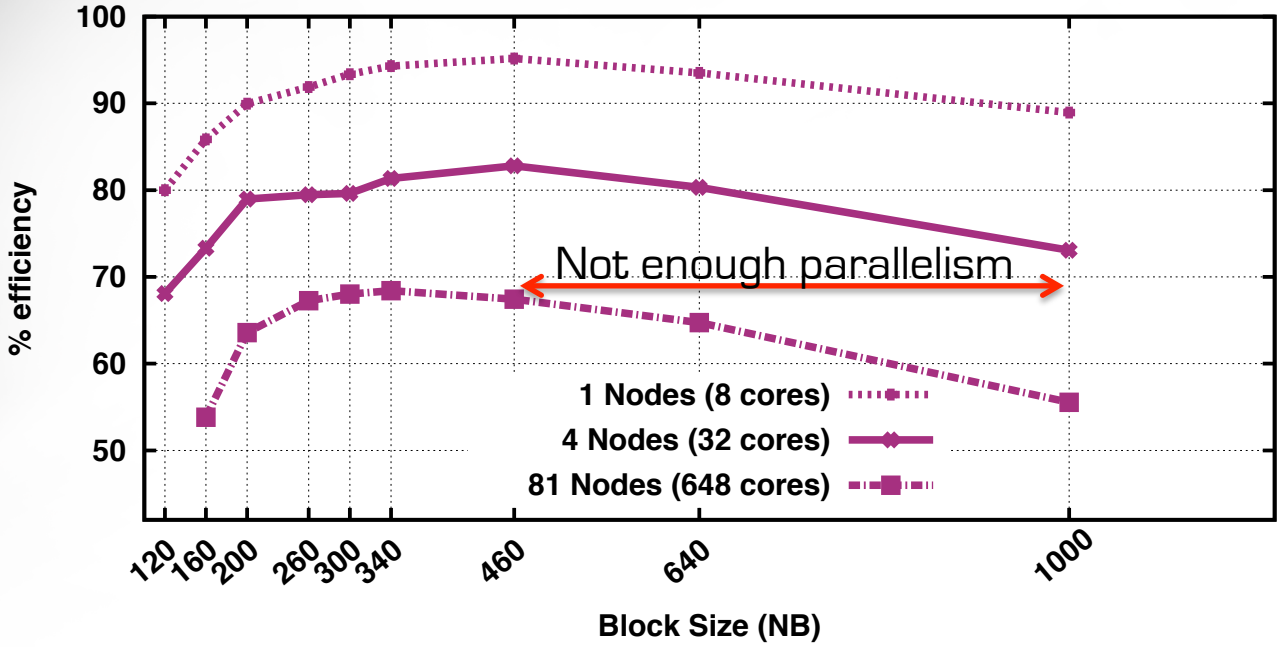
Completion at another rank generates ACTIVATE MSG

Hybrid Scheduling

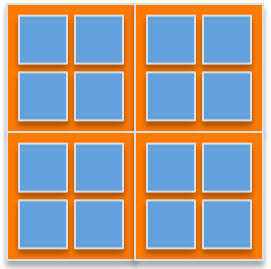
Cores alternate between task execution and scheduling for completed task



Task Granularity in Hybrid systems



- Multi-level tuning
  - Tune the kernels based on local architecture
  - Then tune the algorithm
- Depends on the network, type and number of cores
- For a fixed size matrix increasing the task duration (or the tile size) decrease parallelism
- For best performance: auto-tune per system



Package content

Compiling and Running Dplasma

Obtaining Dplasma

- PaRSEC website: http://icl.cs.utk.edu/parsec/
- Software package release available (see above)
  - The bleeding edge release is on **bitbucket**
    - *https://bitbucket.org/bosilca/dplasma*
    - Use your favorite Mercurial client and hg clone !

- Help Wiki:
  - https://bitbucket.org/bosilca/dplasma/wiki/Home
- User support Mailing list:
  - https://lists.eecs.utk.edu/mailman/listinfo/dplasma-users

Building

- tar xvzf dplasma.tgz
- mkdir build && cd build
- ../dplasma-src/contrib/platform/config.linux
- make –j 8
- (mpirun -np 1) dplama/testing/testing_dpotrf 4000 -v -x

It's a script that invokes cmake, look inside and tweak as needed

Notable options:

`Debug`
gdb friendly build

`-DDAGUE_GPU_WITH_CUDA=OFF`
do not use CUDA (even if available)

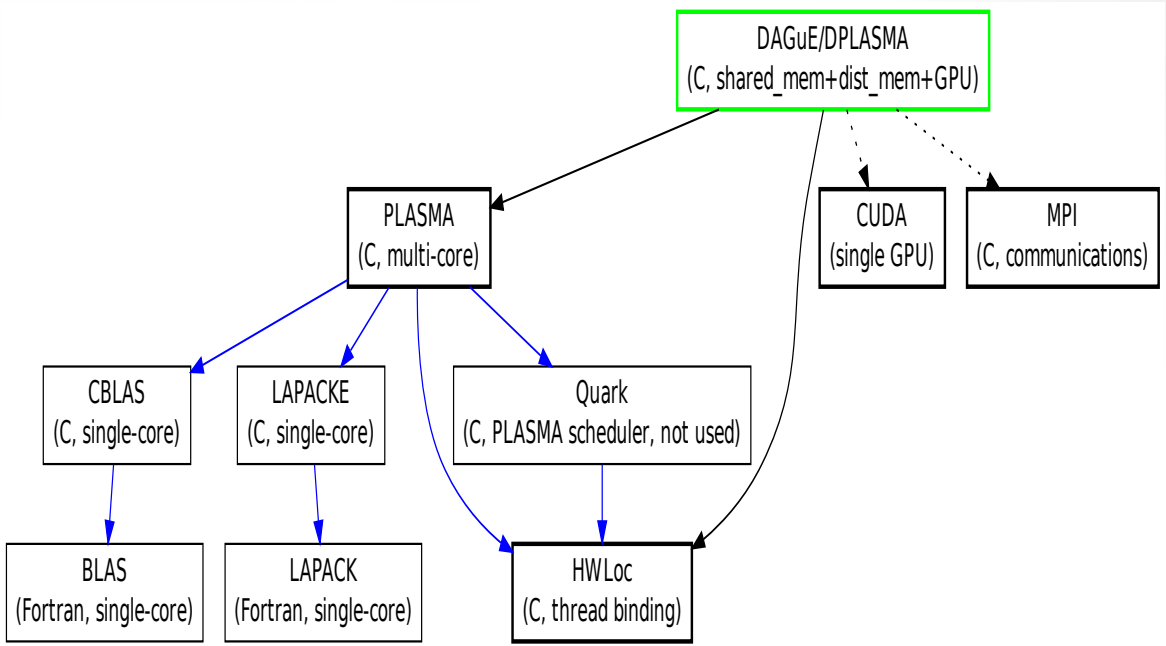
`-DDAGUE_DIST_WITH_MPI=OFF`
do not use MPI (even if available)

`-DDAGUE_PROF_TRACE=ON`
generate XML profile trace

*ccmake . (**t** to toggle advanced mode) lists all possible options*

Dependencies

- Use cmake version 2.8.4 or higher
  - http://www.cmake.org/
- Supported compilers: **gcc, llvm, Intel**, PGI, XLC,… (requires C99)
- PLASMA version 2.4.5 (and associated BLAS)
  - http://icl.cs.utk.edu/plasma/
- CUDA >= 3.0 (optional), MPI (optional)
- HWLOC version 1.3 (optional)
  - http://www.open-mpi.org/projects/hwloc/
- Omega test (optional, for Q2J compiler)
  - https://github.com/davewathaverford/the-omega-project/



After Compilation

- After a successful compilation, the following files are created:
  - `/path/to/dague-bin/libdague.a`
    - The static library of the DAGuE runtime engine
  - `/path/to/dague-bin/dplasma/lib/libdplasma.a`
    - The static library of DPLASMA
  - `/path/to/dague-bin/dplasma/testing/testing_*`
    - Executables of example drivers and checks for basic routines available in dplasma.

Full directory structure

- `contrib/platforms/config.xxx`
  - Build process uses **Cmake**
  - **config.xxx sets sensible cmake options** for xxx architecture
  - For Kraken (Cray XT), Linux, Keeneland, BlueGenes, etc.
- `include/, src/`
  - The PaRSEC Runtime
- `tools/dague-compiler`
  - source to source compiler: PTG (.jdf) to C stubs (.c)
- `tools/q2j`
  - Dataflow extractor: Quark code (.c) to PTG (.jdf)
- `tools/profiling`
  - Debugging & profiling & trace tools
- `data_dist/`
  - Common block cyclic distributions
- `dplasma/lib`
  - PTG versions of Plasma tile algorithms
- `dplasma/include`
  - C interface (fortran interface coming soon)
- `dplasma/testing/`
  - Exemple dplasma programs (and stress/ performance tests)

Running your program, tuning

- **mpirun –x OMP_NUM_THREADS=1 –x LD_LIBRARY_PATH –bynode –np 128 dplasma/testing/testing_dfoobar –N 30000 –t 180 –p 8 –q 16**
  - **1 process per node**! (-bynode in Open MPI)
  - OMP_NUM_THREAD=1 (**no MT in BLAS, Plasma**!) (-x forwards env in Open MPI)
  - **PxQ process grid**, Scalapack rule of thumb still apply (P~=Q for LLT, Q~=2P for LU, QR, etc)
  - -c: forces the number of PaRSEC execution units (default uses all cores)
    - On some Cray XT machines, setting –c NCORES-1 improves communication speed, most other Linux machines do not need this
    - Hwloc custom binding available for complex hardware (-V option)
  - -g: number of GPUs to use (default is 0!); not all test are GPU accelerated (gemm, potrf are)
  - -t: tune NB, -i: tune IB (if applicable)
    - NB=~180-200 is typical for cores (defaults are provided, but each machine is different)
    - On Nvidia GPUs, NB should be a multiple of 64: 320, 384 or 640 work well on C2050
  - -s: supertiling, reorders tiles to increase intra-node locality (at the expense of temporal load balance)
  - -x: verify result correctness (backward error in most cases, depends on algorithm)

Conclusion

- Programming made easy(ier)
  - Portability: inherently take advantage of all hardware capabilities
  - Efficiency: deliver the best performance on several families of algorithms
  - Formalism helps debugging and verification
- Let different people focus on different problems
  - Application developers on their algorithms
  - System developers on system issues

Questions?

Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J. **"DAGuE: A generic distributed DAG Engine for High Performance Computing.,"** *Parallel Computing*, T. Hoefler eds. Elsevier, Vol. 38, No 1-2, 27-51, 2012.

Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaeif, H., Luszczek, P., YarKhan, A., Dongarra, J. **"Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA,"** *Proceedings of the Workshops of the 25th IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2011 Workshops)*, IEEE, Anchorage, Alaska, USA, 1432-1441, 16-20 May, 2011.

History: Beginnings of Data Flow

- "*Design of a separable transition-diagram compiler*", M.E. Conway, Comm. ACM, 1963
  - Coroutines, flow of data between process
- J.B. Dennis, 60's
  - Data Flow representation of programs
  - Reasoning about parallelism, equivalence of programs, …
- "The semantics of a simple language for parallel programming", G. Kahn
  - Kahn Networks