Dataflow-based Task Execution through PaRSEC for HPC.

Anthony Danalis

Innovative Computing Laboratory

University of Tennessee

MCSoC'14, Japan

George Bosilca
Aurelien Bouteiller
Mathieu Faverge
Thomas Herault
Jack Dongarra

Programming Paradigms

- ✧PaRSEC offers a new programming paradigm
    - ✧Dataflow-based Task Execution
- ✧What is wrong with current prog. paradigms?
- ✧What is the current programming paradigm?

Serial Code

```
do i = 1, len
    y(i) = y(i) + a*x(i)
enddo
```

OpenMP Code

```
#pragma omp parallel
do i = 1, len
    y(i) = y(i) + a*x(i)
enddo
```

MPI Code

```
// Exchange initial data
do i = my_start, my_len
    y(i) = y(i) + a*x(i)
enddo
// Exchange results
```

MPI Code

```
// Exchange initial data
do i = my_start, my_le
    y(i) = y(i) + (i)
enddo
// Exchange ults
```

Data Distribution, parallelism and load balance are coupled.

MPI Code

```
// Exchange initial data
do     my_start, my_le
    y(      (i) +     (i)
enddo
// Exchange
```

No mechanism to deal with jitter.

Data Distribution, parallel   nd load balance are coupled.

MPI+X Code

- Works well (maybe) only for isomorphic systems
- Plagued by limitations of MPI and X
  - Dynamic Load balancing
  - Handling Jitter
  - Data distribution
  - Difficult to develop
  - Difficult to debug
- Replaces "big hammer" by two big hammers!

Current Programming Paradigm

Coarse Grain Parallelism with Explicit Message Passing (CGP)

MPI programs are essentially sequential, with some code to coordinate.

Tile Algorithms

PARALLEL LINEAR ALGEBRA SOFTWARE FOR MULTICORE ARCHITECTURES

PLASMA

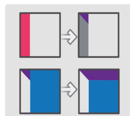
INNOVATIVE COMPUTING LABORATORY
THE UNIVERSITY of TENNESSEE UT

THE PARALLEL LINEAR ALGEBRA SOFTWARE FOR MULTICORE ARCHITECTURES (PLASMA) PROJECT aims to address the critical and highly disruptive situation that is facing the Linear Algebra and High Performance Computing community due to the introduction of multicore architectures. PLASMA's ultimate goal is to create software frameworks that enable programmers to simplify the process of developing applications that can achieve both high performance and portability across a range of new architectures. PLASMA uses a programming model that allows asynchronous, out-of-order scheduling of operations in order to achieve a scalable yet highly efficient software framework for Computational Linear Algebra applications.

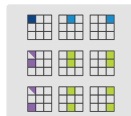
TILE ALGORITHMS

Unlike LAPACK, which uses block algorithms, PLASMA relies on tile algorithms to enable the use of fine grained parallelism.

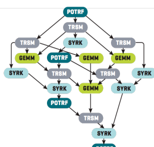
LAPACK
Block algorithms



PLASMA
Tile algorithms



Tile algorithms of Linear Algebra operations can be represented as Directed Acyclic Graphs (DAG) where nodes represent the tasks in which the operation can be decomposed and the edges represent the dependencies among them. As long as the task execution order does not violate the dependencies, the result will be correct.



Example of a DAG for a Cholesky Factorization

PLASMA 2.1.0

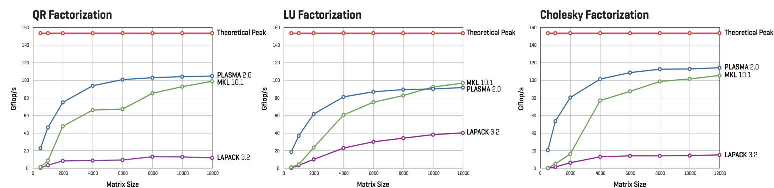
- Solution of Linear Equations
- Linear Least Squares Problems
- Multiple Precision Support
- Mixed-Precision Iterative Solver
- Static Scheduling
- LAPACK Interface / Native Interface
- LAPACK-Compliant Error Handling
- LAPACK-Derived Testing Suite
- Thread Safety
- Windows, Linux, AIX, Mac OS
- PLASMA Users' Guide

CURRENT RESEARCH

- Singular Value Decomposition
- Symmetric and Non Symmetric Eigenvalue Problems
- Dynamic Scheduling
- Communication Avoiding Algorithms
- Autotuning
- Distributed Memory Machines
- Hardware Accelerators

PERFORMANCE RESULTS DOUBLE PRECISION

CPU Intel Xeon 2.4 GHz Quad-socket Quad cores (16 cores total)



DOWNLOAD THE LIBRARY AT http://icl.eecs.utk.edu/plasma/

A COLLABORATION OF THE UNIVERSITY of TENNESSEE, Berkeley UNIVERSITY OF CALIFORNIA, University of Colorado Denver

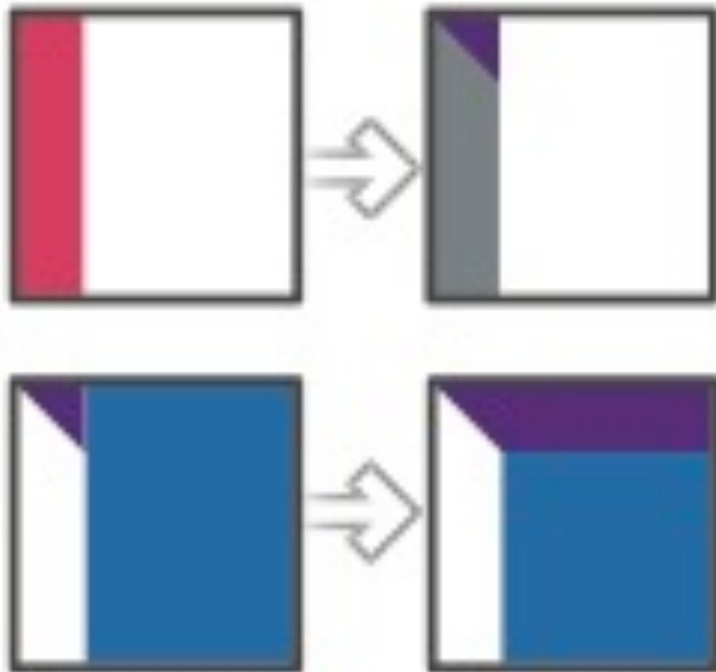
WITH SUPPORT FROM The MathWorks, Microsoft

SPONSORED BY CITR CENTER for INFORMATION TECHNOLOGY RESEARCH

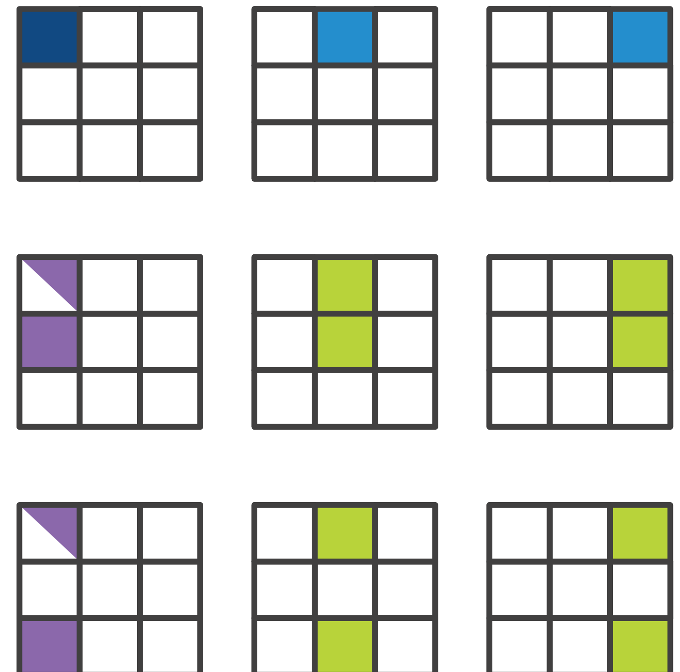
ICL UT

Panel vs Tile Algorithms

LAPACK / Panel



PLASMA / Tile



What does the code look like?

```
for (k = 0; k < MT; k++) {
    Insert_Task( geqrt,  A[k][k], INOUT,  T[k][k], OUTPUT);
    for (m = k+1; m < MT; m++) {
        Insert_Task( tsqrt,  A[k][k], INOUT | REGION_D|REGION_U,
                              A[m][k], INOUT | LOCALITY,
                              T[m][k], OUTPUT);
    }
    for (n = k+1; n < NT; n++) {
        Insert_Task( unmqr,  A[k][k], INPUT | REGION_L,
                              T[k][k], INPUT,
                              A[k][m], INOUT);
        for (m = k+1; m < MT; m++) {
            Insert_Task( tsmqr, A[k][n], INOUT,
                                  A[m][n], INOUT | LOCALITY,
                                  A[m][k], INPUT,
                                  T[m][k], INPUT);
```

ICL UT

What's wrong with this code

- ✗ It has:
  - ✗ Control Flow
  - ✗ Hints for runtime to infer Data Flow
  - ✗ High memory requirements or reduced parallelism

- ✓ It should have:
  - ✓ No (or minimal, user defined) Control Flow
  - ✓ Explicit Data Flow
  - ✓ Unhindered parallelism

What captures the semantics?

GEQRT(k)
k = 0 .. mt-1

TSMQR(k,m,n)
k = 0 .. mt-1
m = k+1 .. mt-1
n = k+1 .. mt-1

UNMQR(k,n)
k = 0 .. mt-1
n = k+1 .. mt-1

TSQRT(k,m)
k = 0 .. mt-1
m = k+1 .. mt-1

{[k,m,n]->[n]:k+1==n && k+1==m}
{[k]->[k,n]:k<n<nt && k<nt-1}
{[k]->[k,k+1]:k<=mt-2}
{[k,m,n]->[k+1,n]:k+1==m && n>m}
{[k,n]->[k,k+1,n]:k<mt-1}
{[k,m]->[k,m,n]:k<nt-1 && k<n<nt}
{[k,m,n]->[n,m]:k+1==n && m>n}
{[k,m,n]->[k,m+1,n]:m<mt-1}
{[k,m,n]->[k+1,m,n]:n>k+1 && m>k+1}
{[k,m]->[k,m+1]:m<mt-1}

Parameterized Task Graph (PTG)

- ✓ Compressed form of the Execution DAG
- ✓ Fixed size (problem size independent)
- ✓ Task Classes w/ parameters
  - ✓ geqrt(k), tsqrt(k,m), unmqr(k,n), tsmqr(k,n,m)
- ✓ Precedence constraints between Tasks

PTG: PING-PONG

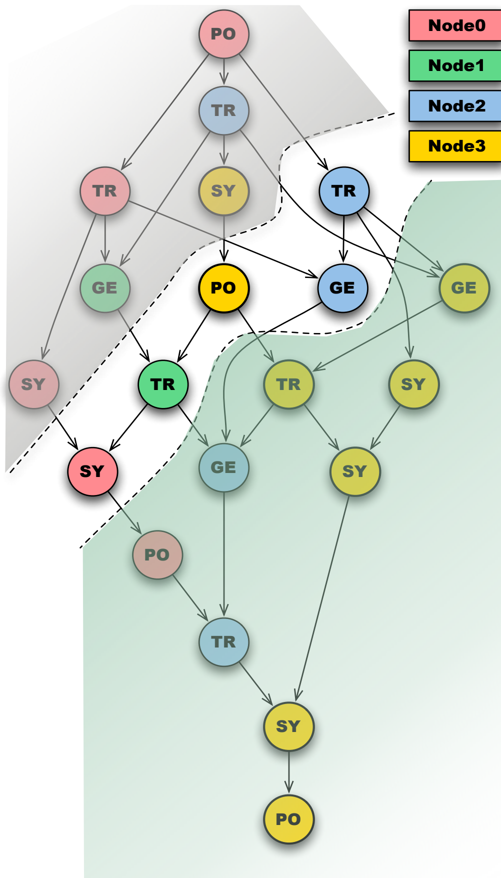
```
PING(s)
  s = 0..max_steps-1
  : A(s)
  RW    A0 <- A(s)
           -> A0 PONG(s)
  READ A1 <- (s != 0) ? PONG(s-1)
BODY verify_response(A0, A1); END

PONG(s)
  s = 0..max_steps-2
  : A(s+1)
  RW    A0 <- A0 PING(s)
           -> A1 PING(s+1)
BODY /* do nothing on data */ END
```

PTG: Binary Tree Reduction

```
BT_REDUC(tree, step, i)
  tree_count = count_bits(NT)
  tree   = 1 .. tree_count
  max_step = log_of_tree_size(NT, tree)
  step   = 1 .. max_step
  i   = 0 ..(1<<(max_step-step))-1
  offset = compute_offset(NT, tree)

  : dataA(offset+i*2,0)

  READ A   <- (1==step) ? A REDUCTION(offset+i*2)
           <- (1!=step) ? B BT_REDUC(tree,step-1,i*2)
  RW   B   <- (1==step)                        ? A REDUCTION(offset+i*2+1)
           <- (1!=step)                        ? B BT_REDUC(tree, step-1, i*2+1)
           -> ((max_step!=step) && (0==i%2)) ? A BT_REDUC(tree, step+1, i/2)
           -> ((max_step!=step) && (0!=i%2)) ? B BT_REDUC(tree, step+1, i/2)
           ->  (max_step==step)                ? C LINEAR_REDUC(tree)
BODY int j; for(j=0; j<NB; j++){ REDUCE( A, B, j ); } END
```

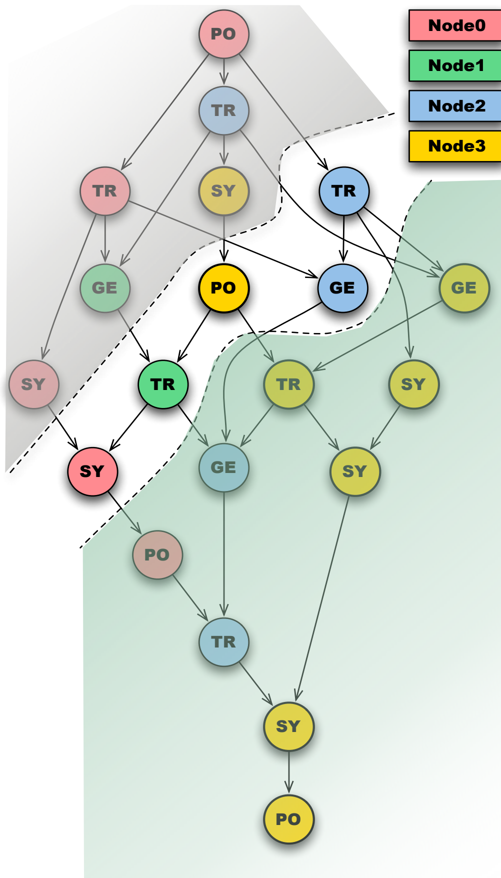
Why not discover the whole DAG?

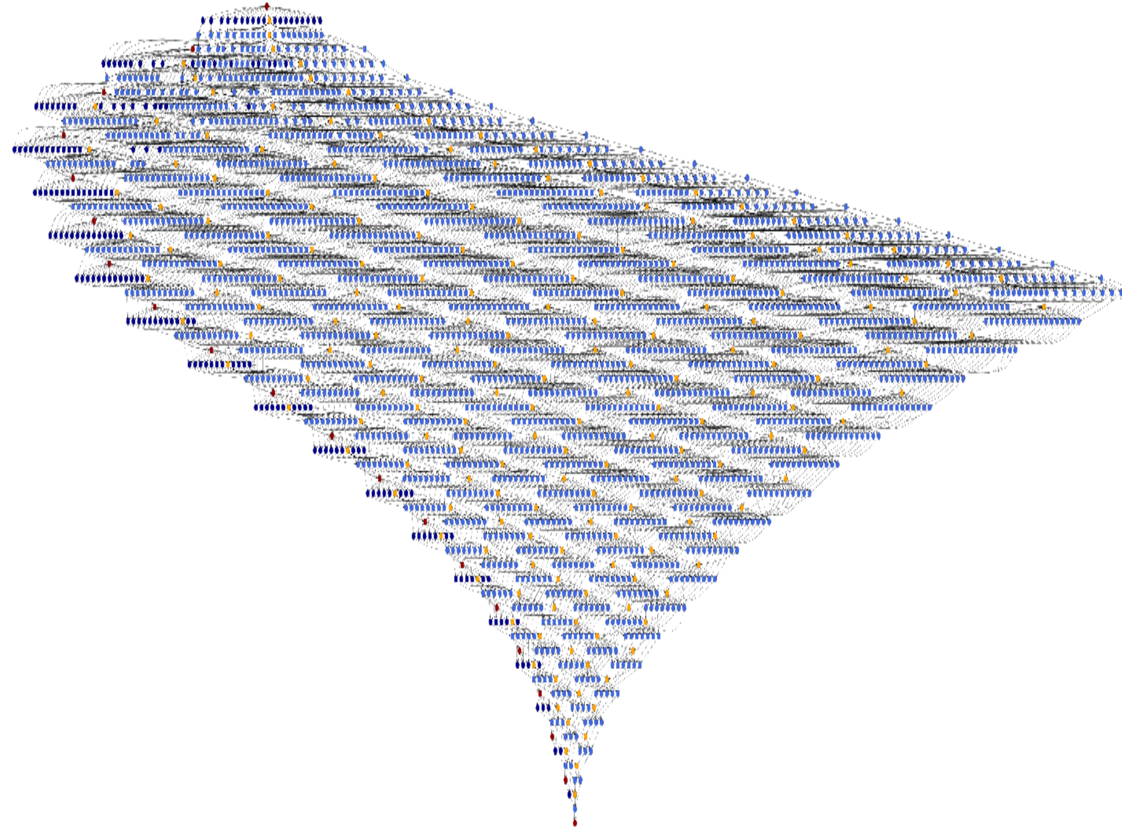
1. Memory Overhead



Why not discover the whole DAG?

1. Memory Overhead





Why not discover the whole DAG?

2. Reduced Parallelism (if using window)

Why not discover the whole DAG?

2. Reduced Parallelism (if using window)

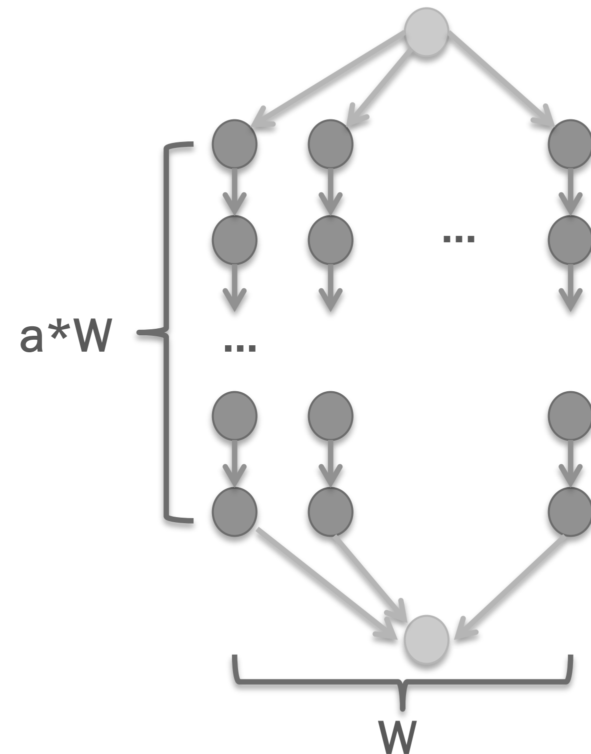
Question:

Given window size **W** and **P** processors, what is the highest level of **parallelism** that can be **missed** because of control flow limitations?

Assuming P<W

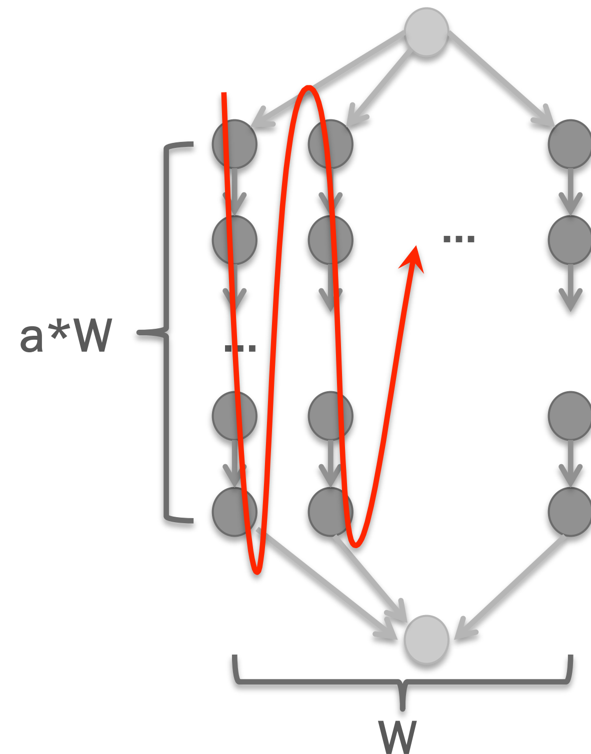
PTG vs DTG

```
for(i=0;i<W;i++){
    Task(RW:A[i][0]);
    for(j=1; j<a*W; j++){
        Task(R:A[i][j-1], W:A[i][j]);
    }
}
```



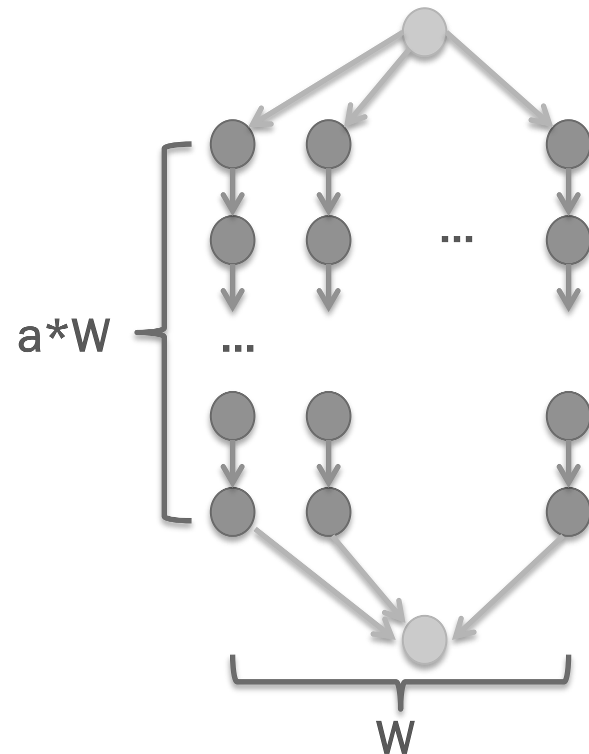
PTG vs DTG

```
for(i=0;i<W;i++){
    Task(RW:A[i][0]);
    for(j=1; j<a*W; j++){
        Task(R:A[i][j-1], W:A[i][j]);
    }
}
```



PTG vs DTG

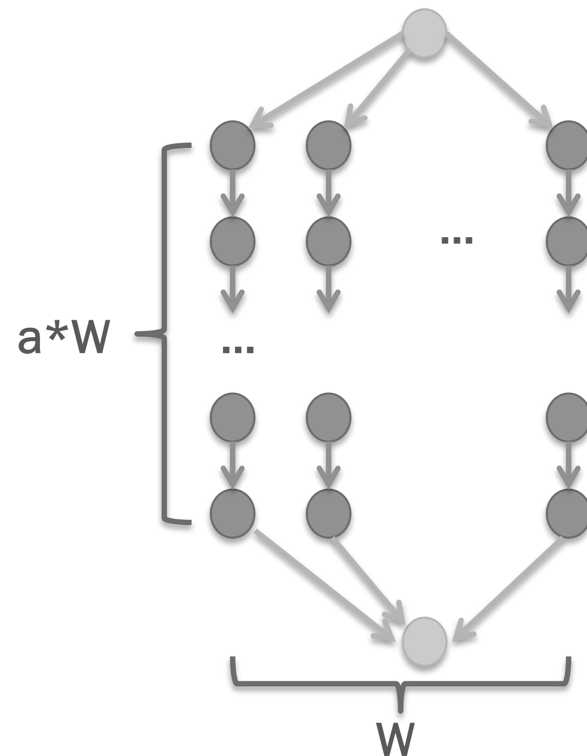
Dynamic Task Graph (DTG):
a*W+(W-1)*(a-1)*W = O(a*W^2)



PTG vs DTG

Dynamic Task Graph (DTG):
a*W+(W-1)*(a-1)*W = O(a*W^2)

Parameterized Task Graph (PTG):
O(a*W^2/P)

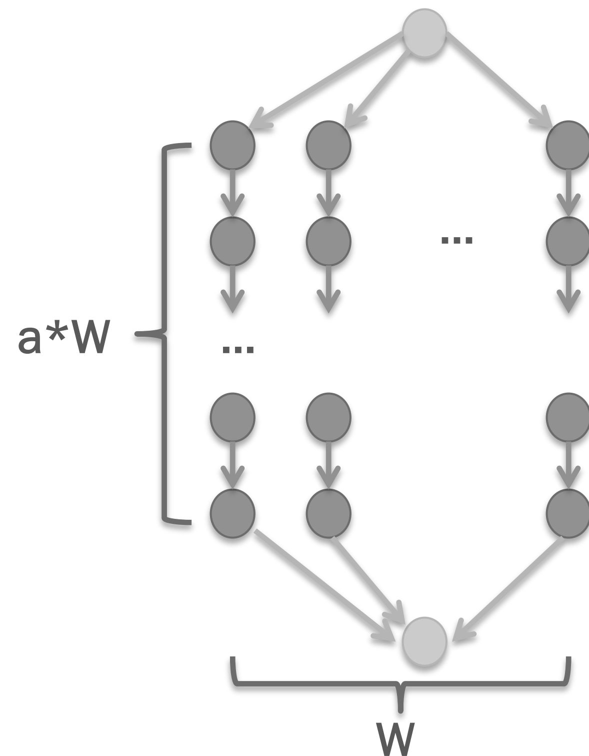


PTG vs DTG

Dynamic Task Graph (DTG):
a*W+(W-1)*(a-1)*W = O(a*W^2)

Parameterized Task Graph (PTG):
O(a*W^2/P)

O(DTG)/O(PTG) = P



PaRSEC: focus on the algorithm

Concepts

- Separation of roles: compiler optimizes each task, developer describes dependencies between tasks, runtime orchestrates dynamic execution
- Separate algorithms from data distribution
- Avoid limitations of control flow execution

$H|\Psi> = E|\Psi>$

Domain Science
CHEMISTRY, NUCLEAR PHYSICS, ...

$\frac{1}{4} v_{ef}^{mn} t_{ij}^{ef} t_{mn}^{ab} - \frac{1}{2} v_{ef}^{mn} t_{mi}^{ef} t_{nj}^{ab}$

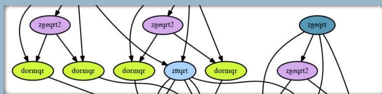
High-level DSLs

```
for j = 1:M
 for k = 1:L
  T[j,k] = X[i][j][k]* Y[k]
```

Sequential Source Code

DATA DISTRIBUTION

PARAMETRIC DAG



PaRSEC

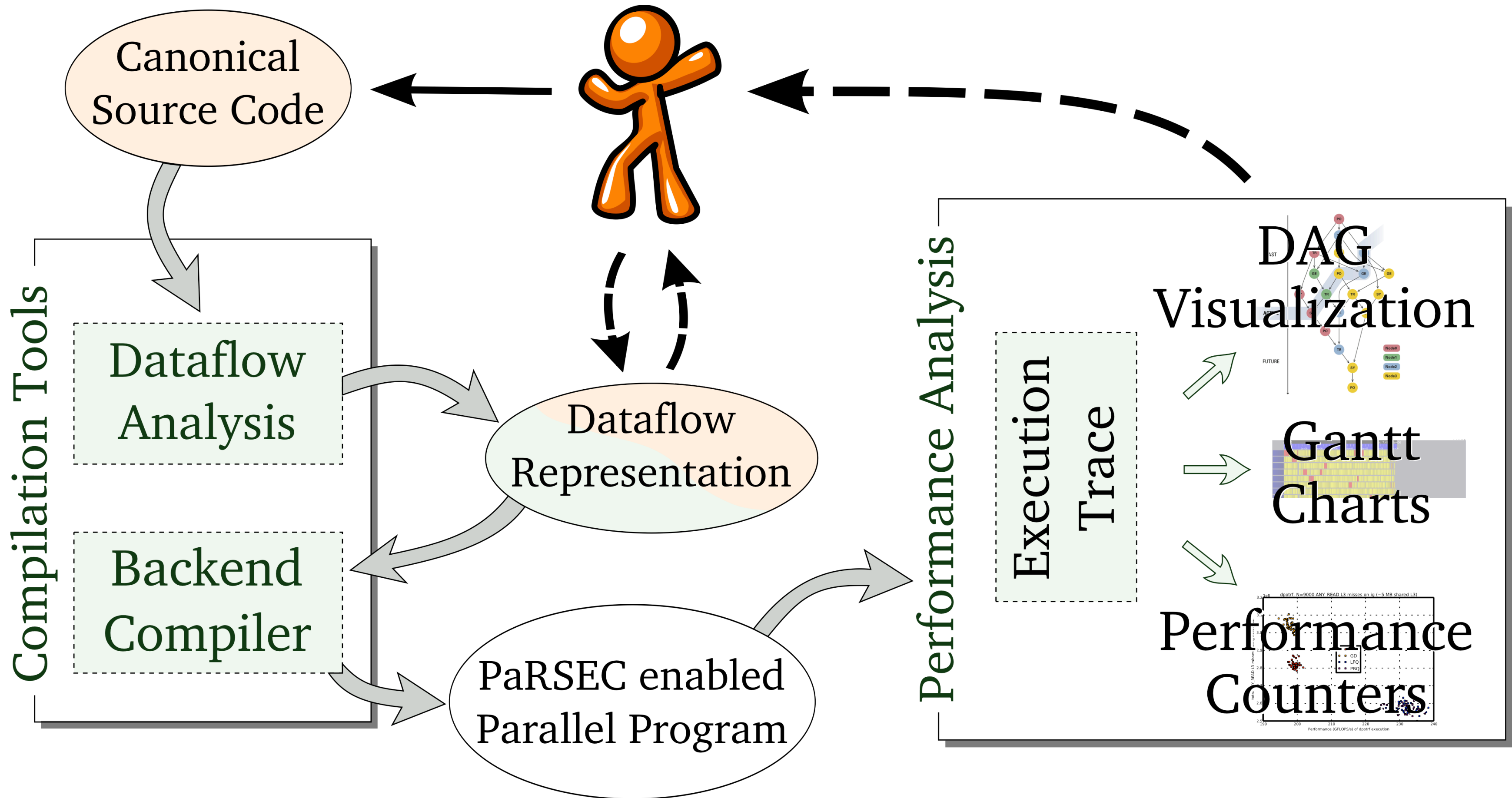
SCHEDULING HINTS

DYNAMIC TASK DISCOVERY

Runtime

- Portability layer for heterogeneous architectures
- Scheduling policies adapt execution to the hardware & ongoing system status
- Data movements between consumers are inferred from dependencies. Communication/computation overlap
- Coherency protocols minimize data movements
- Memory hierarchy (including NVRAM and disk) integral part of the scheduling decisions

Developer's view of PaRSEC



Application Complexity & PaRSEC

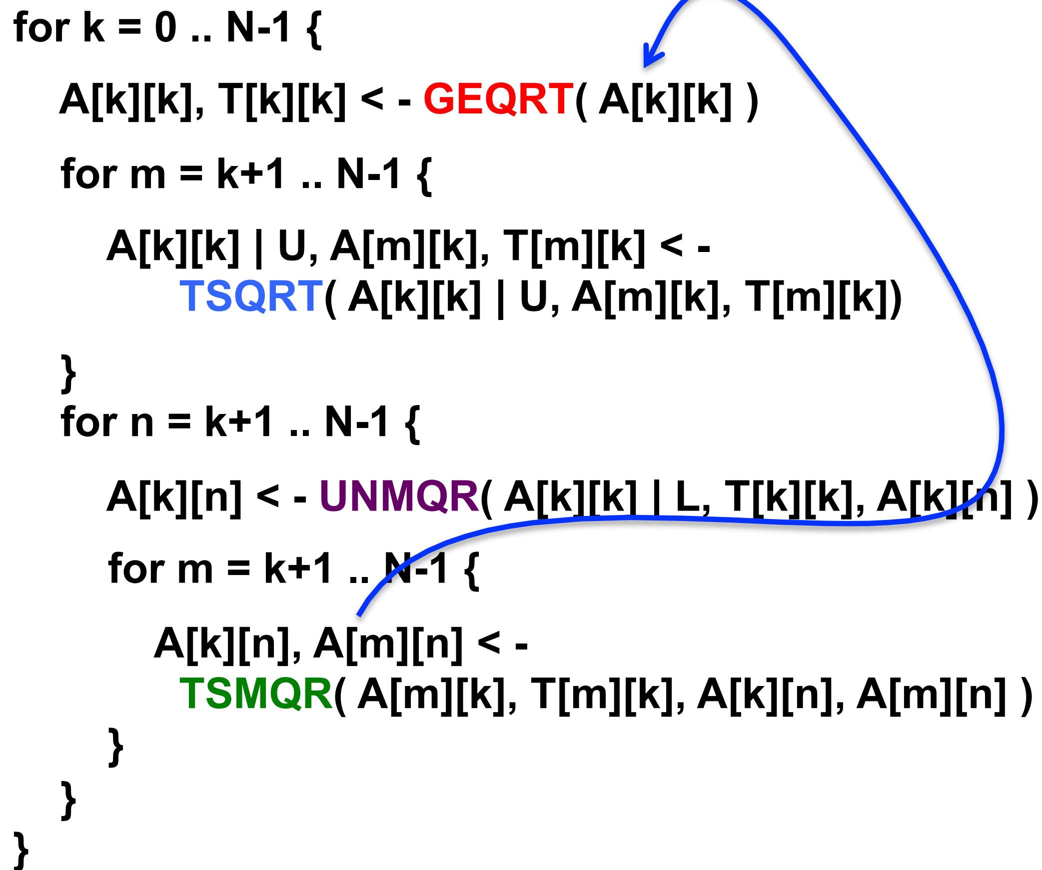
Generality
Expressivity

Simplicity
Regularity

Data dependent Data-flow	Dynamic but Fixed Pattern	Statically Decidable
Mesh Refinement Image flooding	Sparce Linear Algebra Sorting Computational Chemistry	Dense Linear Algebra

ICL UT

Dataflow Analysis



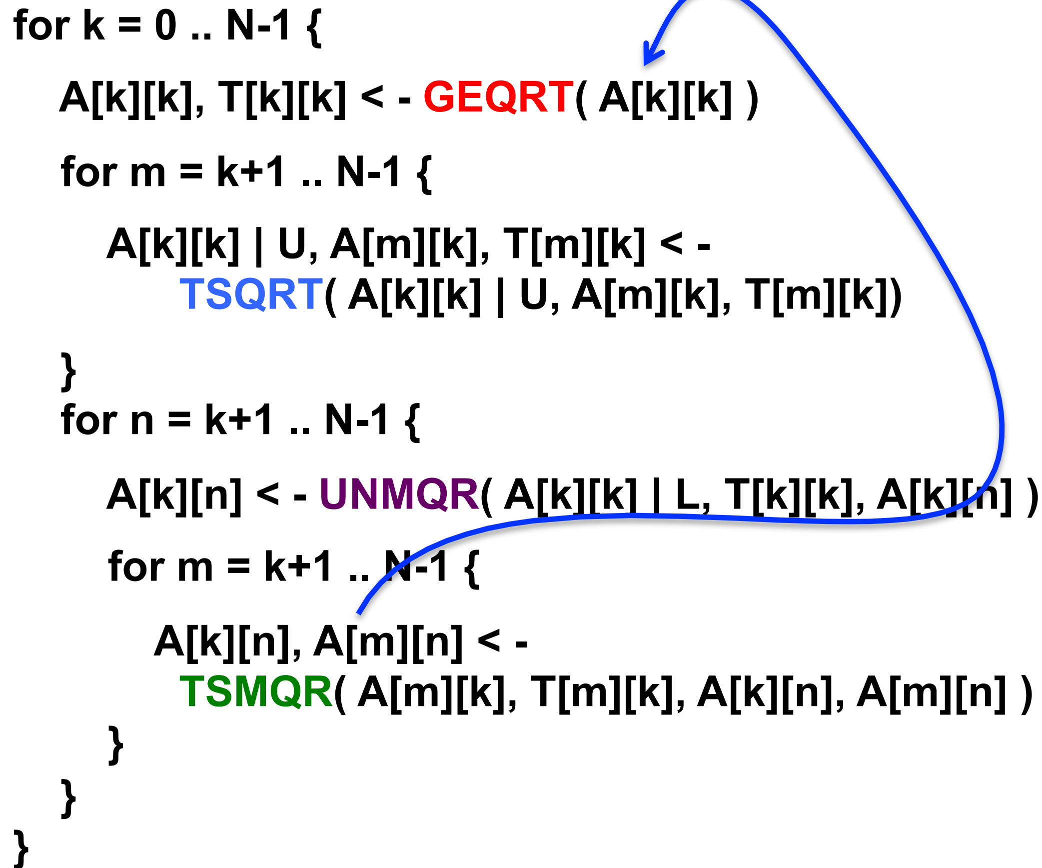
USE

A[k'][k'] :
0 <= k' < N-1

DEF

A[m][n] :
k+1<= m < N-1
k+1<= n < N-1
0 <= k < N-1

Dataflow Analysis



USE

A[k’][k’] :
0 <= k’ <N-1

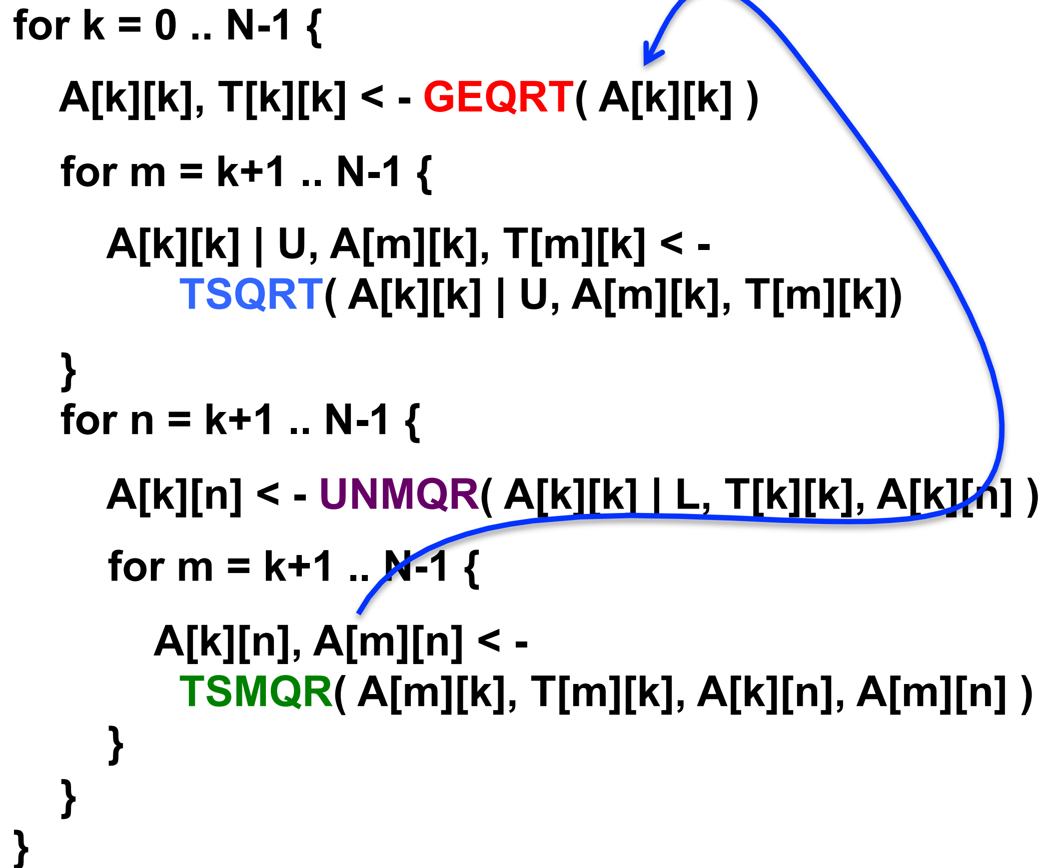
Ctrl Flow

k’ > k

DEF

A[m][n] :
k+1<= m < N-1
k+1<= n < N-1
0 <= k < N-1

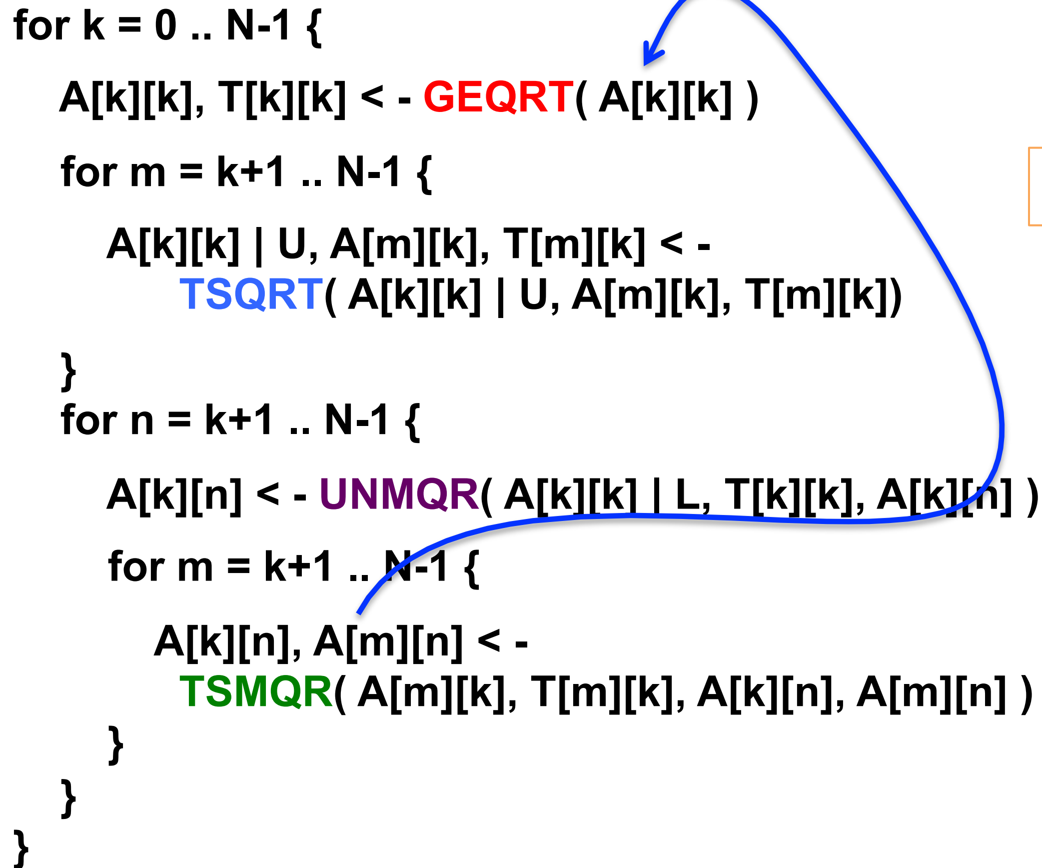
Dataflow Analysis



Flow Dependency (RAW) Relation

[k,m,n] -> [k’] :
k+1<= m < N-1
k+1<= n < N-1
0 <= k < N-1
0 <= k’ < N-1
k < k’
m = k’
n = k’

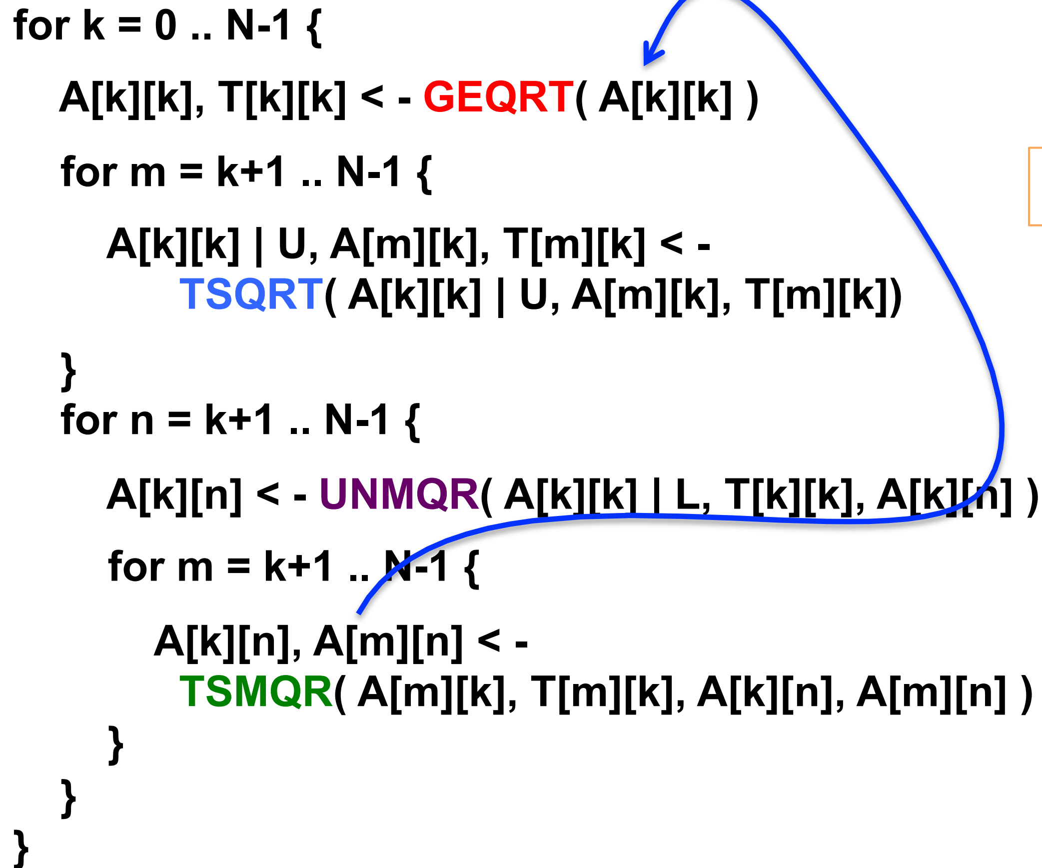
Dataflow Analysis



Omega Simplified

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

Dataflow Analysis



Omega Simplified

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

Question:

At **every** step (k), **all** TSMQR tasks will generate data that flows to future GEQRT tasks. Is this correct?

Dataflow Analysis

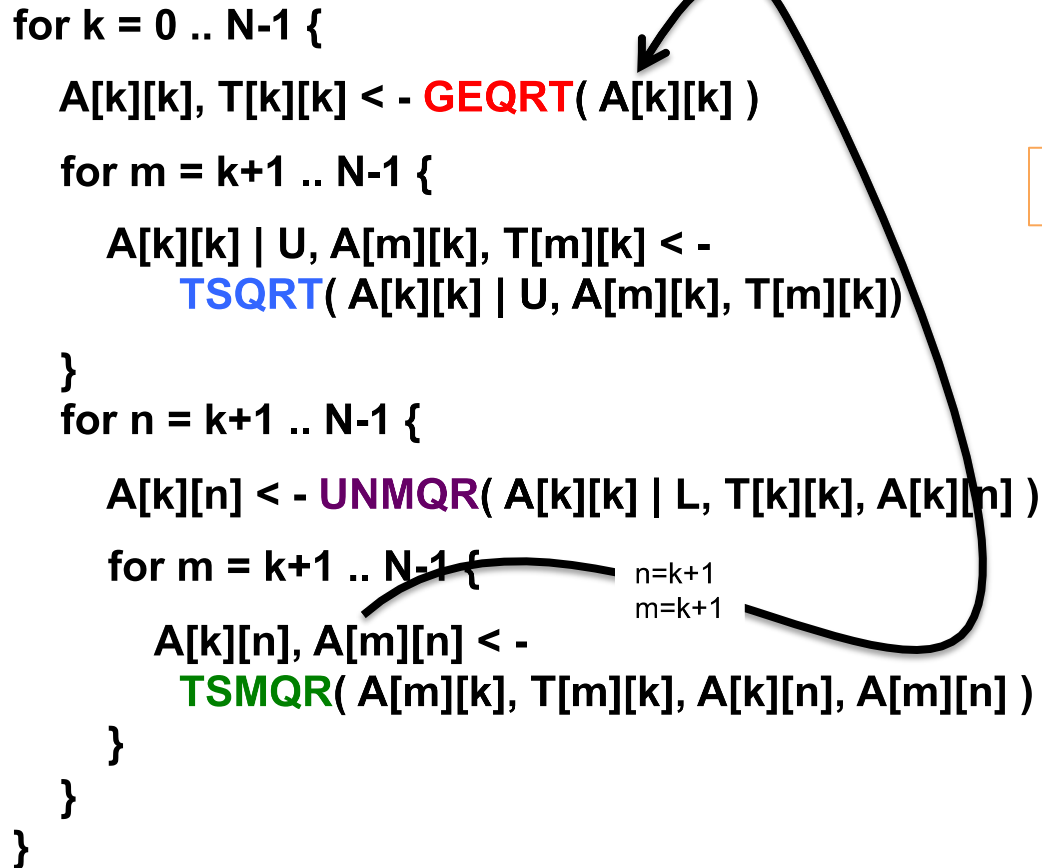
```
for k = 0 .. N-1 {
   A[k][k], T[k][k] < - GEQRT( A[k][k] )
   for m = k+1 .. N-1 {
      A[k][k] | U, A[m][k], T[m][k] < -
          TSQRT( A[k][k] | U, A[m][k], T[m][k])
   }
   for n = k+1 .. N-1 {
      A[k][n] < - UNMQR( A[k][k] | L, T[k][k], A[k][n] )
      for m = k+1 .. N-1 {
         A[k][n],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

Omega Simplified

```
{[k,m,m] -> [m] : k+1, 0 <= m < N}
```

Output Dependency (WAW)

Dataflow Analysis



Real Edge: Flow - Output

{[k,k+1,k+1] -> [k+1] : 0 <= k <= N-2}

Dataflow Analysis

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

n=k+1
m=k+1

Real Edge: Flow - Output

{[k,k+1,k+1] -> [k+1] : 0 <= k <= N-2}

GEQRT's incoming edge

{[k-1,k, k] -> [k] : 0 < k <= N-1}
(k>0) ? TSMQR(k-1,k,k)

Discovering Collectives

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

GEQRT - > UNMQR

{[k] -> [k, n] : 0 <= k <= N-2 &&
k+1 <= n <=N-1 }

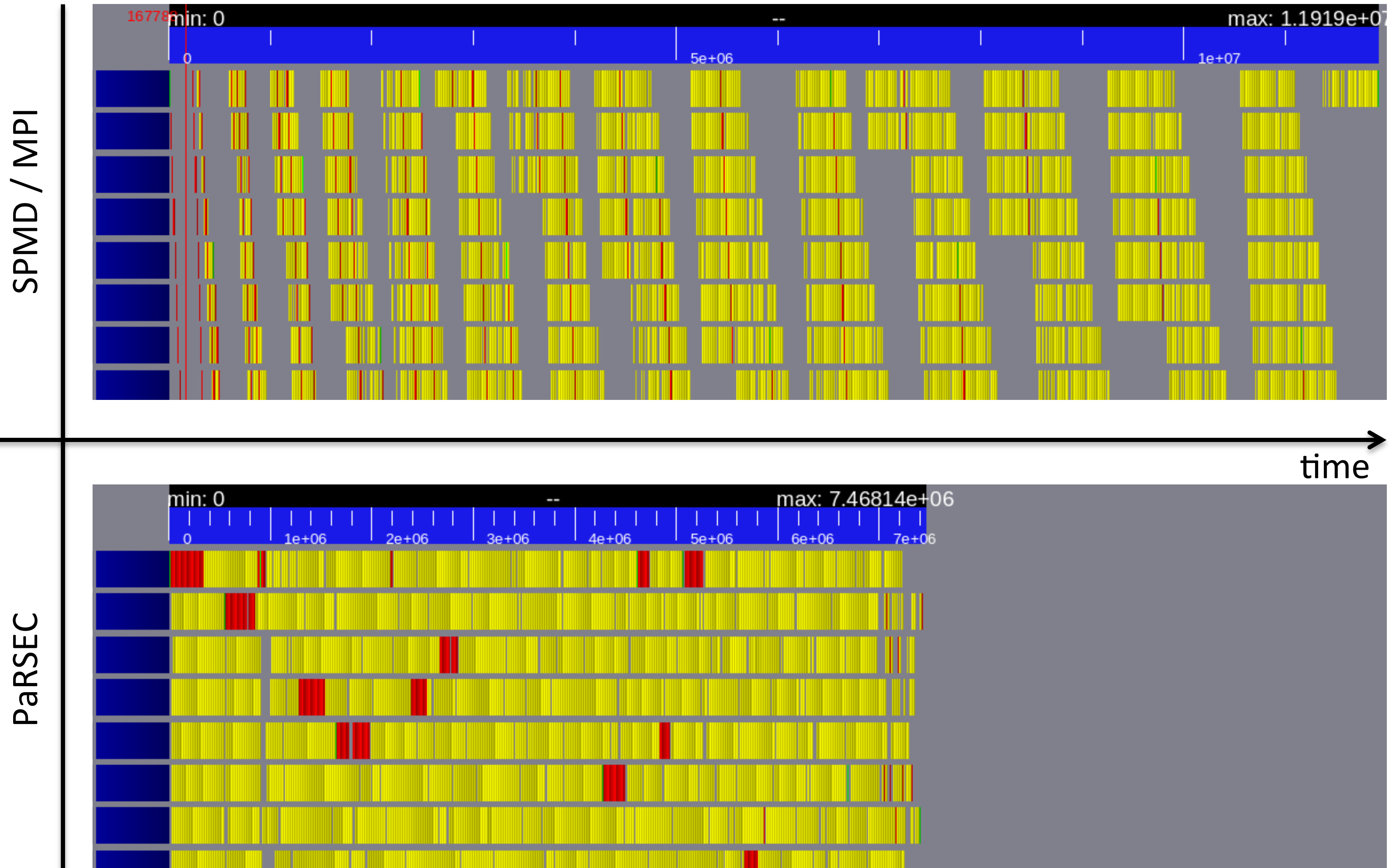
Discovering Collectives

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

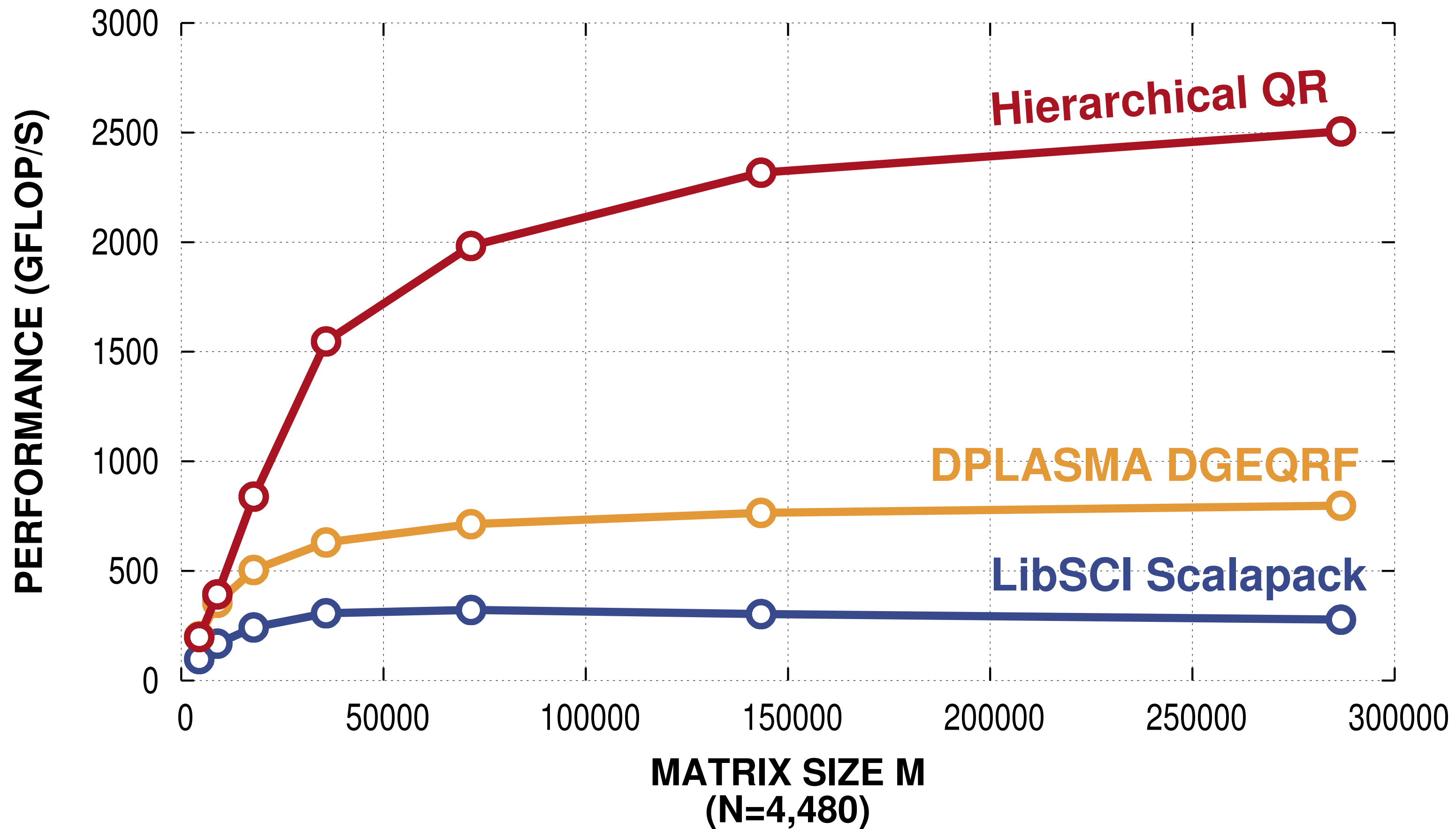
GEQRT - > UNMQR

```
{[k] -> [k, n] :    0 <= k <= N-2 &&
                    k+1 <= n <=N-1 }

-> (k<N-1) ? UNMQR(k, k+1..N-1)
```

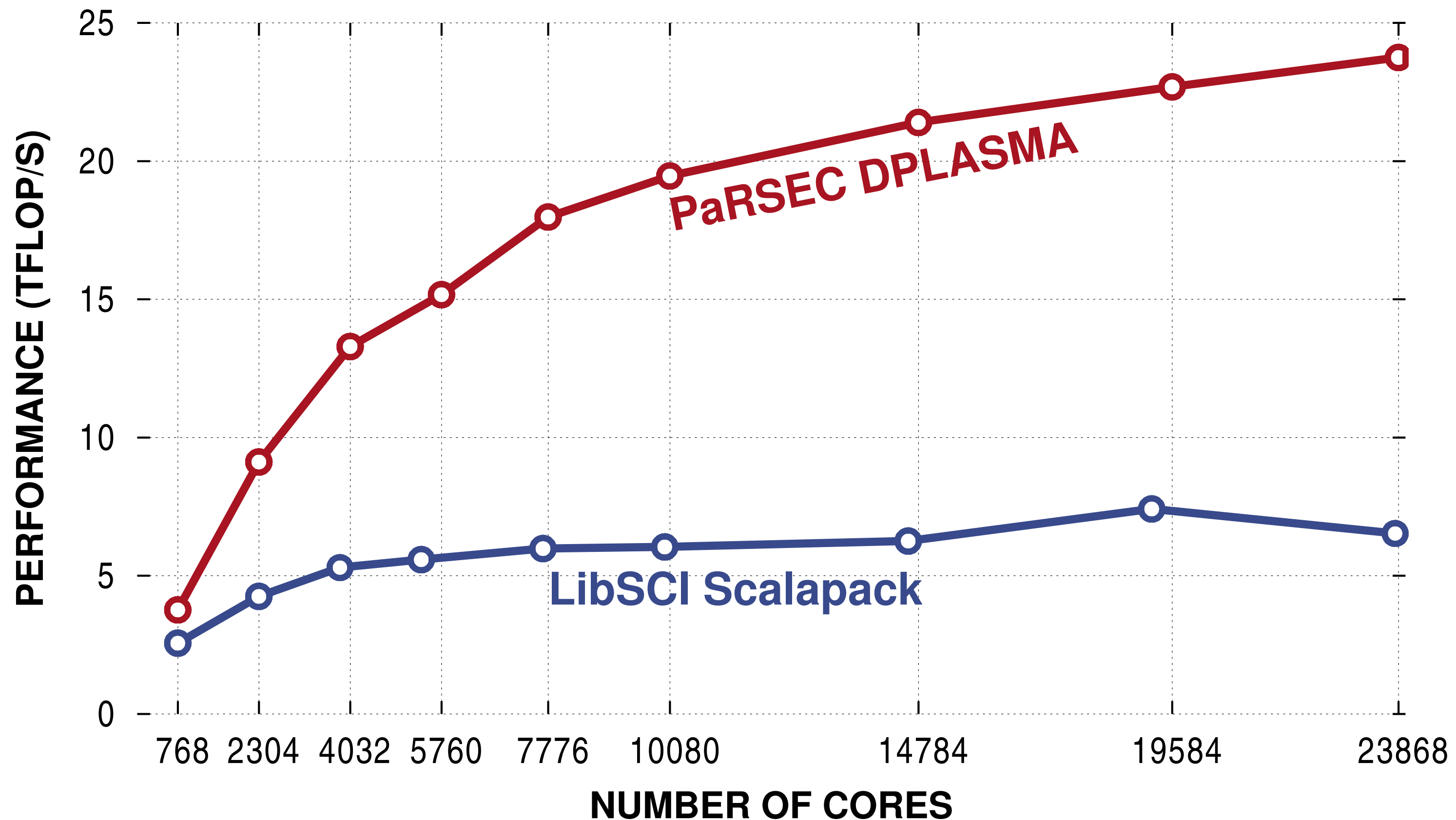
Load Balance, Idle time & Jitter



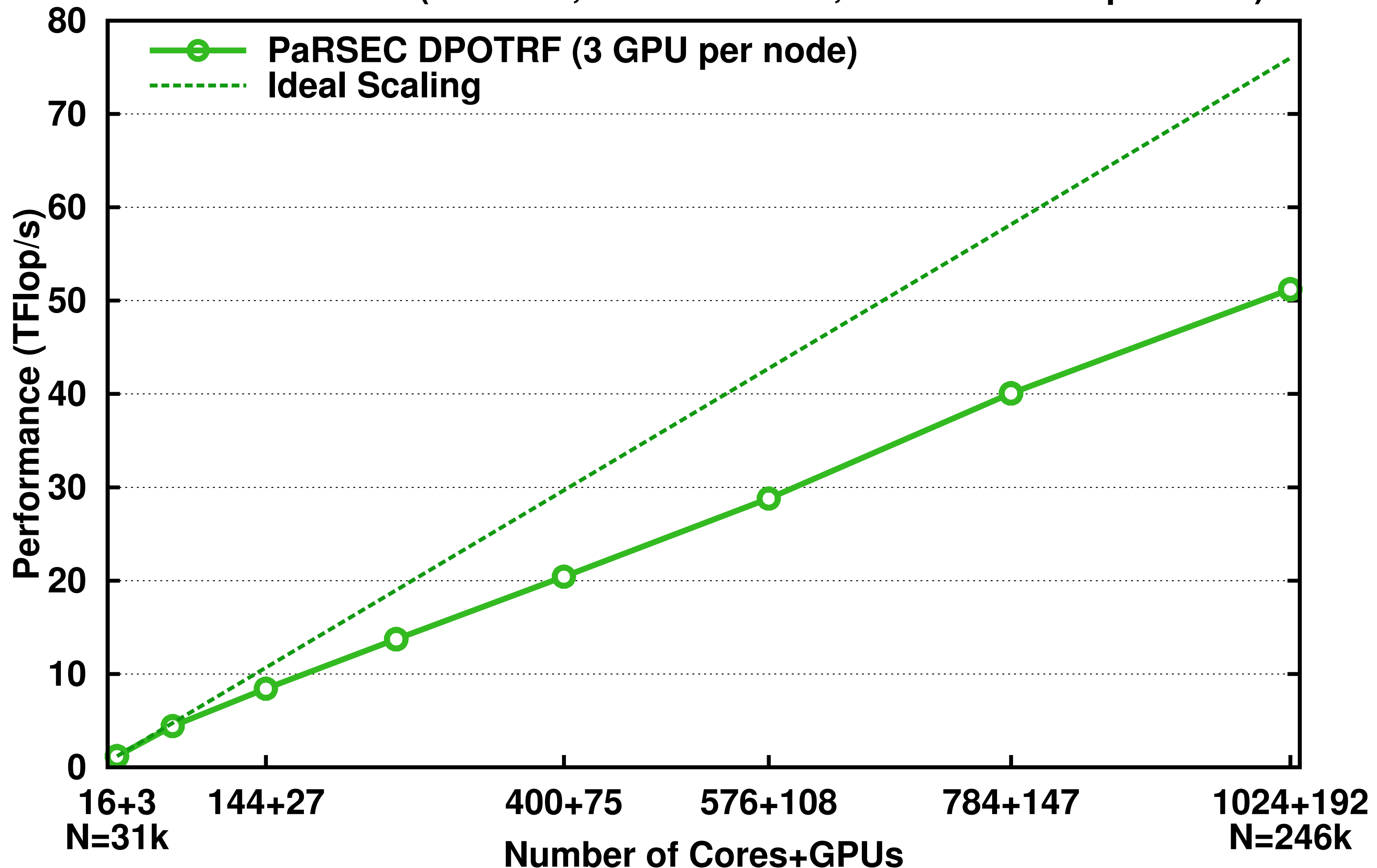
Performance: (H)QR



Performance: Systolic QR



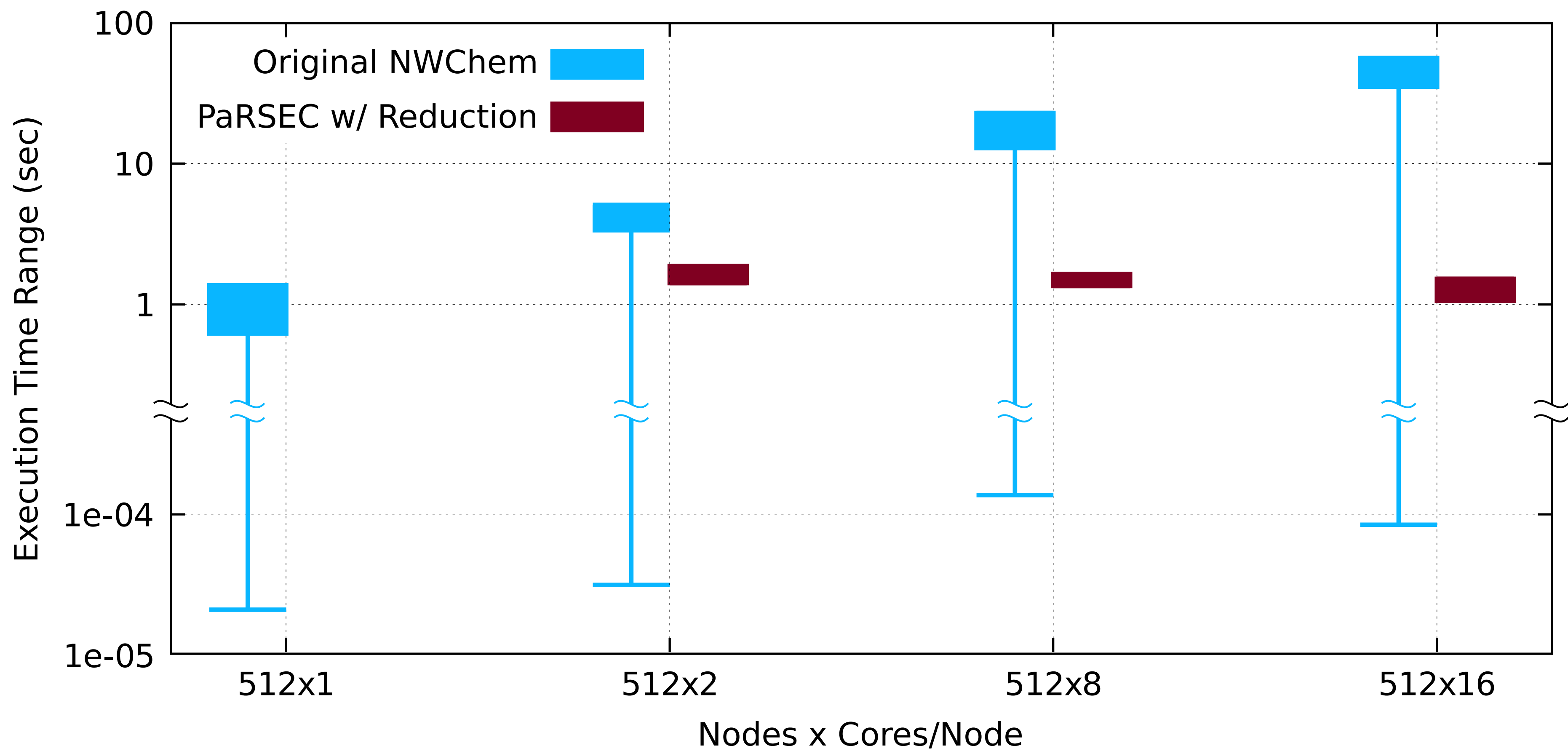
Distributed CPUs + GPUs



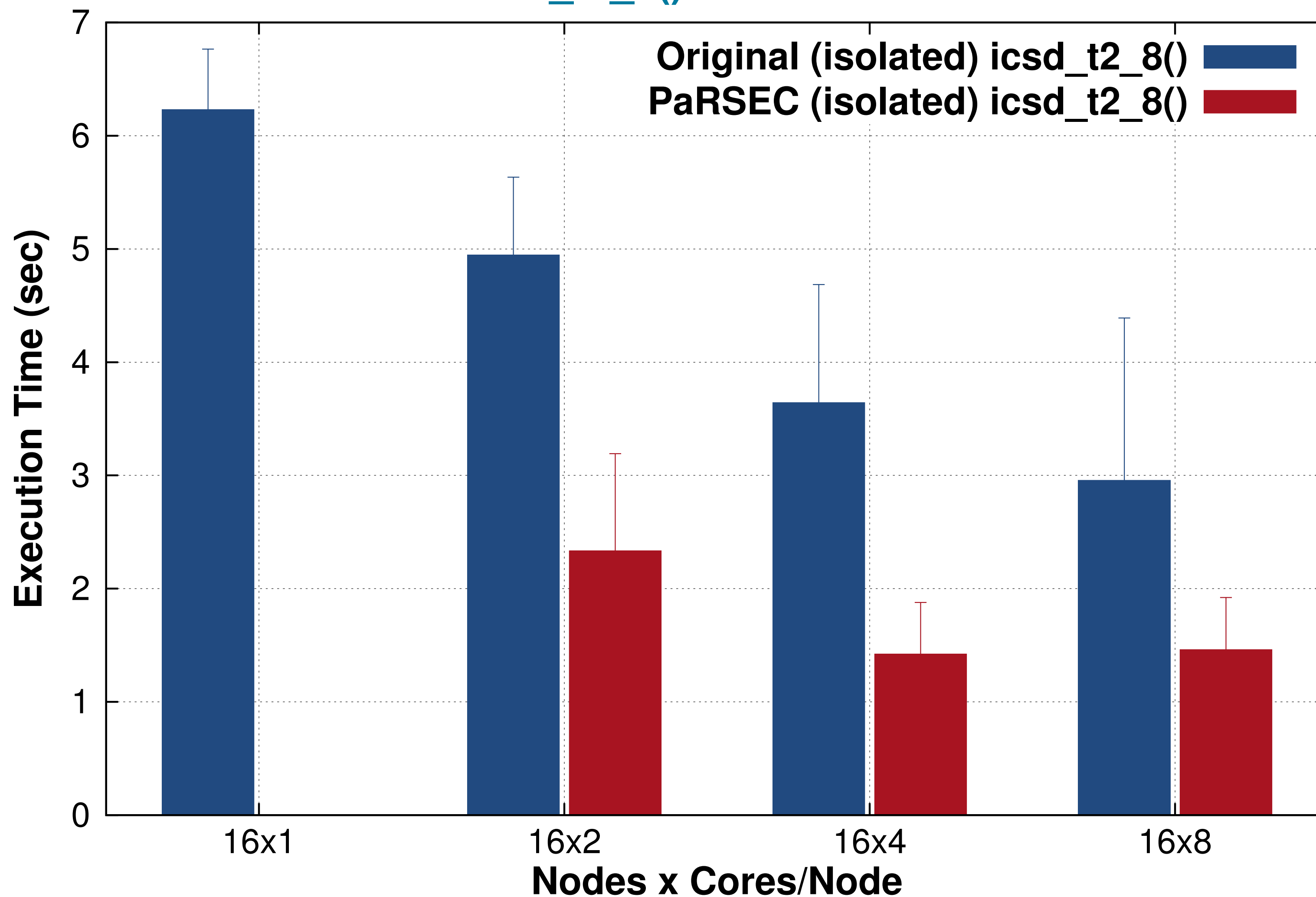
NWChem Coupled Cluster (CC)

- Computational Chemistry
- TCE Coupled Cluster
- Machine generated Fortran 77 code
- Behavior depends on dynamic, **immutable** data
- Long sequential chains of DGEMMs
- Work (chain) stealing through GA atomics

CC t1_2_2_2



CC t2_8



Breaking the chains

```
DO p3b, p4b, h1b, h2b
      CALL DFILL(dimc,0.0d0,dbl_mb(k_c_sort),1)

      DO p5b, p6b
          IF (int_mb(k_spin+p5b-1) .eq. ...) THEN
              CALL DGEMM( ... )
          END IF
      END DO

      CALL TCE_SORT_4(dbl_mb(k_c_sort),dbl_mb(k_c), ... )
      CALL ADD_HASH_BLOCK(d_c,dbl_mb(k_c),dimc, expr )
END DO
```

Breaking the chains

```
DO p3b, p4b, h1b, h2b
     CALL DFILL(dimc,0.0d0,dbl_mb(k_c_sort),1)

     DO p5b, p6b
         IF (int_mb(k_spin+p5b-1) .eq. ...) THEN
             CALL DGEMM( ... )
         END IF
     END DO

     CALL TCE_SORT_4(dbl_mb(k_c_sort),dbl_mb(k_c), ... )
     CALL ADD_HASH_BLOCK(d_c,dbl_mb(k_c),dimc, expr )
END DO
```

DO

C = C + A*B

ENDDO

Breaking the chains

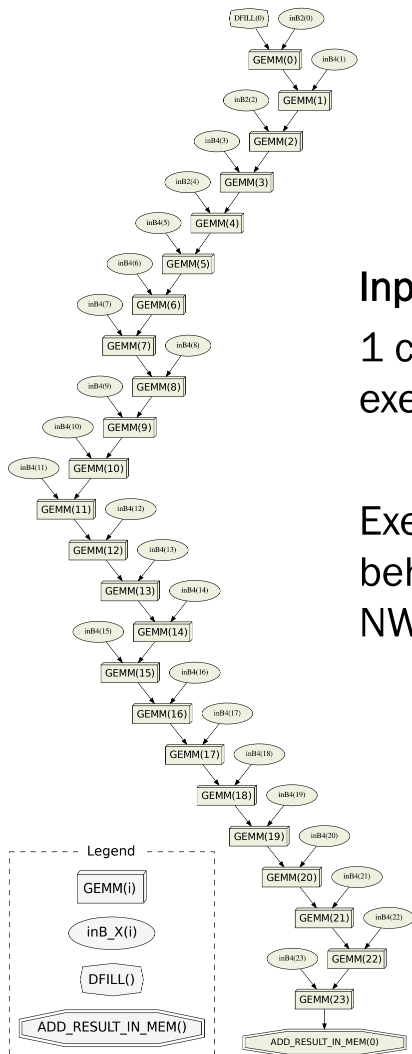
```
DO p3b, p4b, h1b, h2b
      CALL DFILL(dimc,0.0d0,dbl_mb(k_c_sort),1)

      DO p5b, p6b
         IF (int_mb(k_spin+p5b-1) .eq. ...) THEN
            CALL DGEMM( ... )
         END IF
      END DO

      CALL TCE_SORT_4(dbl_mb(k_c_sort),dbl_mb(k_c), ... )
      CALL ADD_HASH_BLOCK(d_c,dbl_mb(k_c),dimc, expr )
END DO
```

DOANY

C = C + A*B

ENDDO

Chain to reduction tree

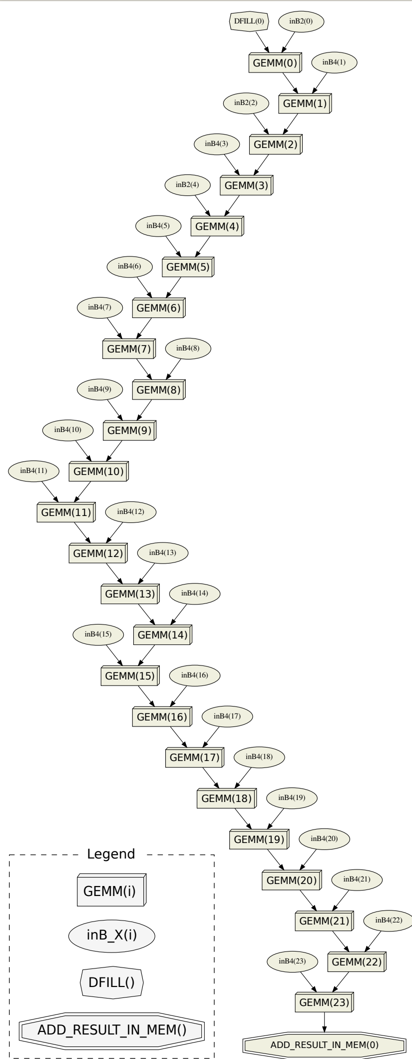


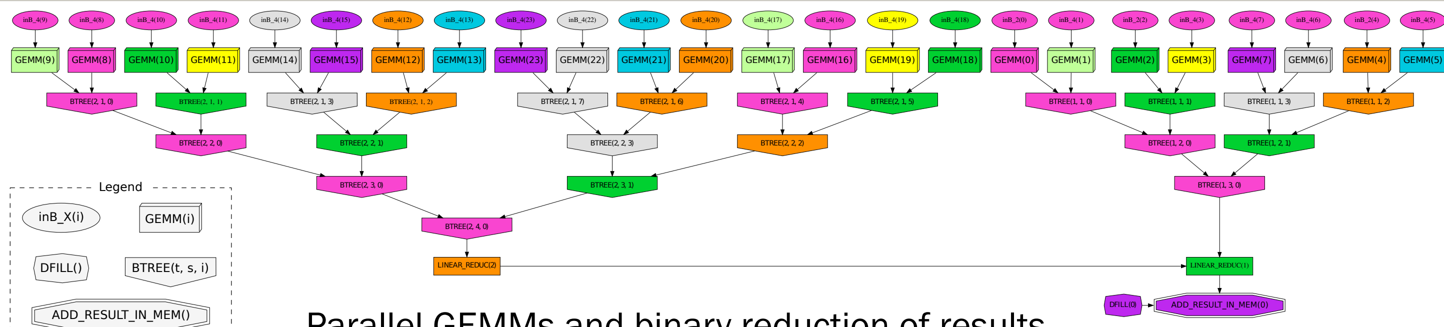
Input: uracil-dimer

1 chain of 24 GEMMs executed sequentially

Execution matches the behavior of original NWChem CCSD

Chain to reduction tree





Parallel GEMMs and binary reduction of results
(each color corresponds to 1 out of 8 nodes)

Conclusions

✧PTG offers a Data-Flow based Prog. Paradigm

✧PaRSEC offers state-of-the-art performance

✧Data distribution is decoupled from Algorithm

✧Control Flow limitations are avoided

✧CGP != highest performance

✧CGP != easiest to program (?)