PTG: an abstraction for unhindered parallelism

Anthony Danalis
Innovative Computing Laboratory
University of Tennessee
WOLFHPC 2014, New Orleans

George Bosilca
Aurelien Bouteiller
Mathieu Faverge
Thomas Herault
Jack Dongarra

Programming Paradigms

- PTG: Parameterized Task Graph
  - Key abstraction behind the PaRSEC runtime
- PTG revives an old programming paradigm:
  - Dataflow-based 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
```

Serial Code

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

We tell the computer what to do and exactly in what order to do it.

Vector Architectures

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

VLIW

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

Superscalar

```
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
```

OpenMP Code

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

Only for shared memory and all burden on compiler

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) + f(i)
enddo
// Exchange results
```

Data Distribution, parallelism and load balance are coupled.

MPI Code

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

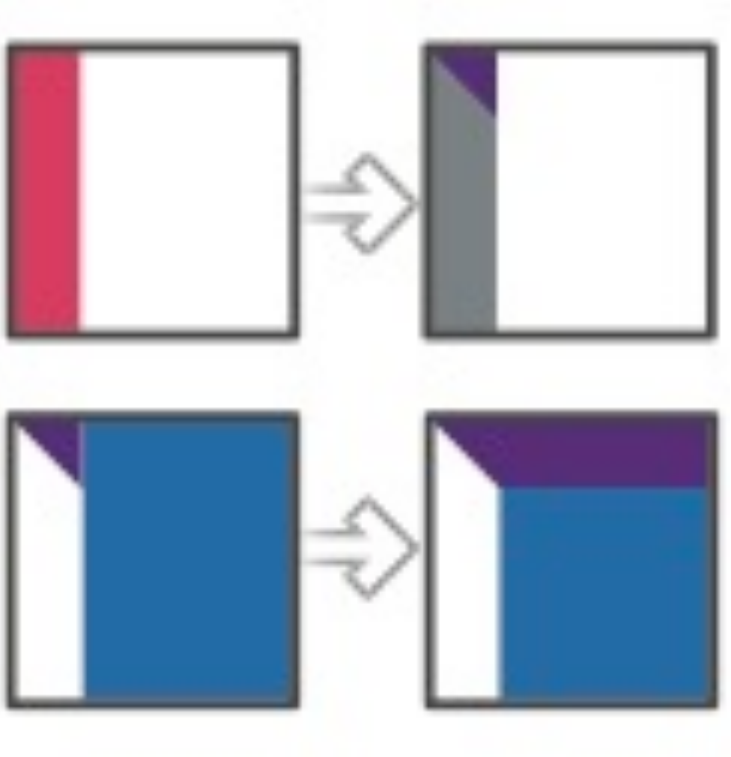
Current Programming Paradigm

Coarse Grain Parallelism with Explicit Message Passing (CGP)

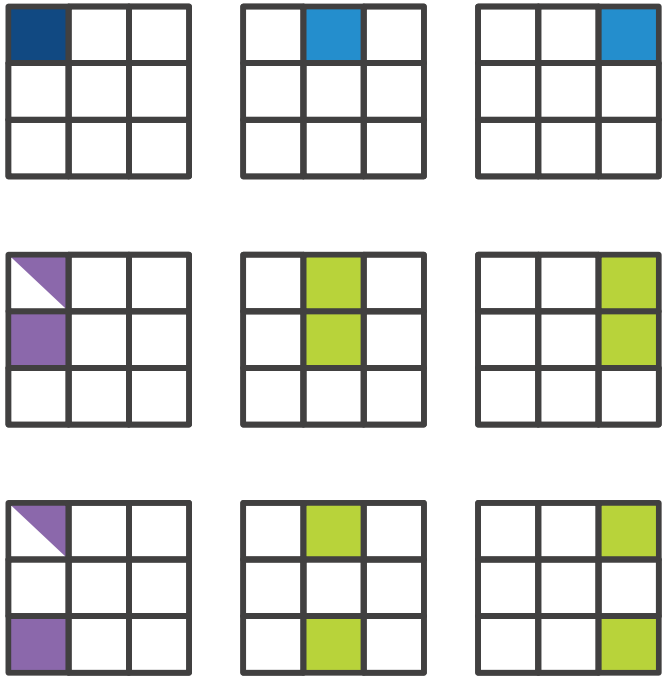
Most parallel programs are essentially sequential, with some code to coordinate.

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);
```

What's wrong with this code

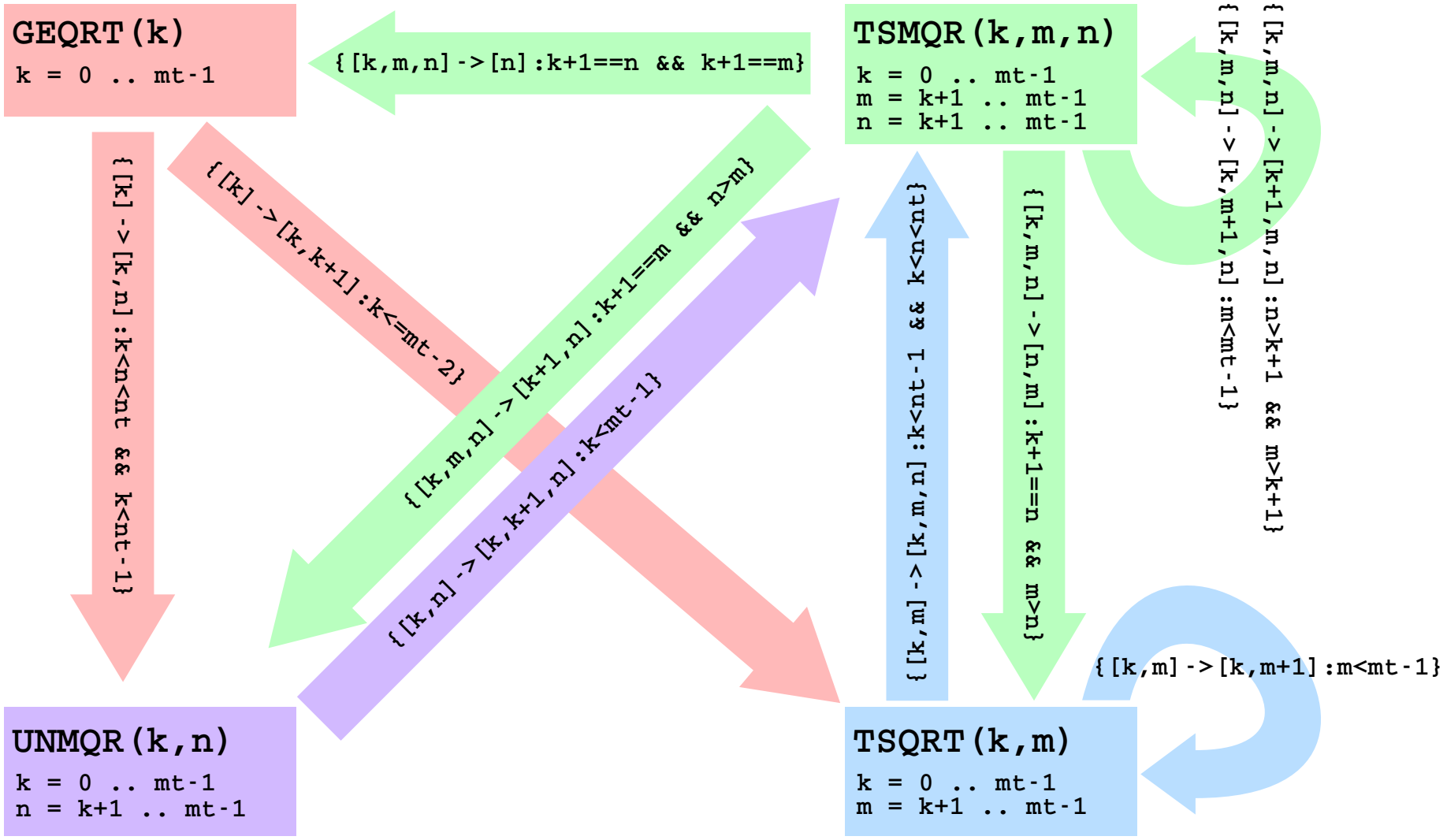
✗ It has:

- ✗ 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

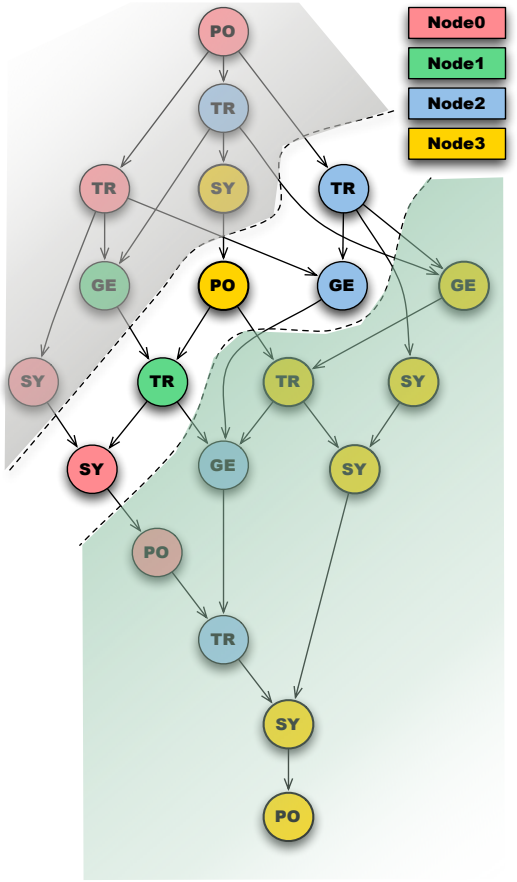
What captures the semantics?



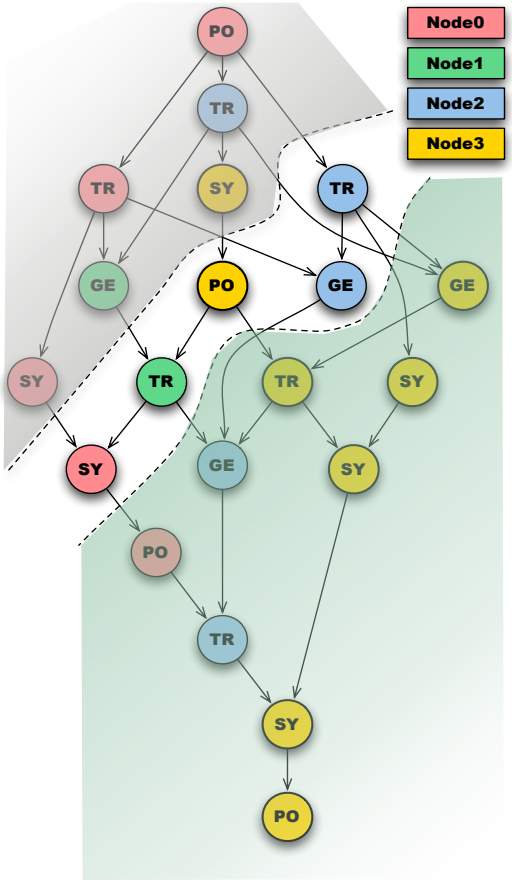
Parameterized Task Graph (PTG)

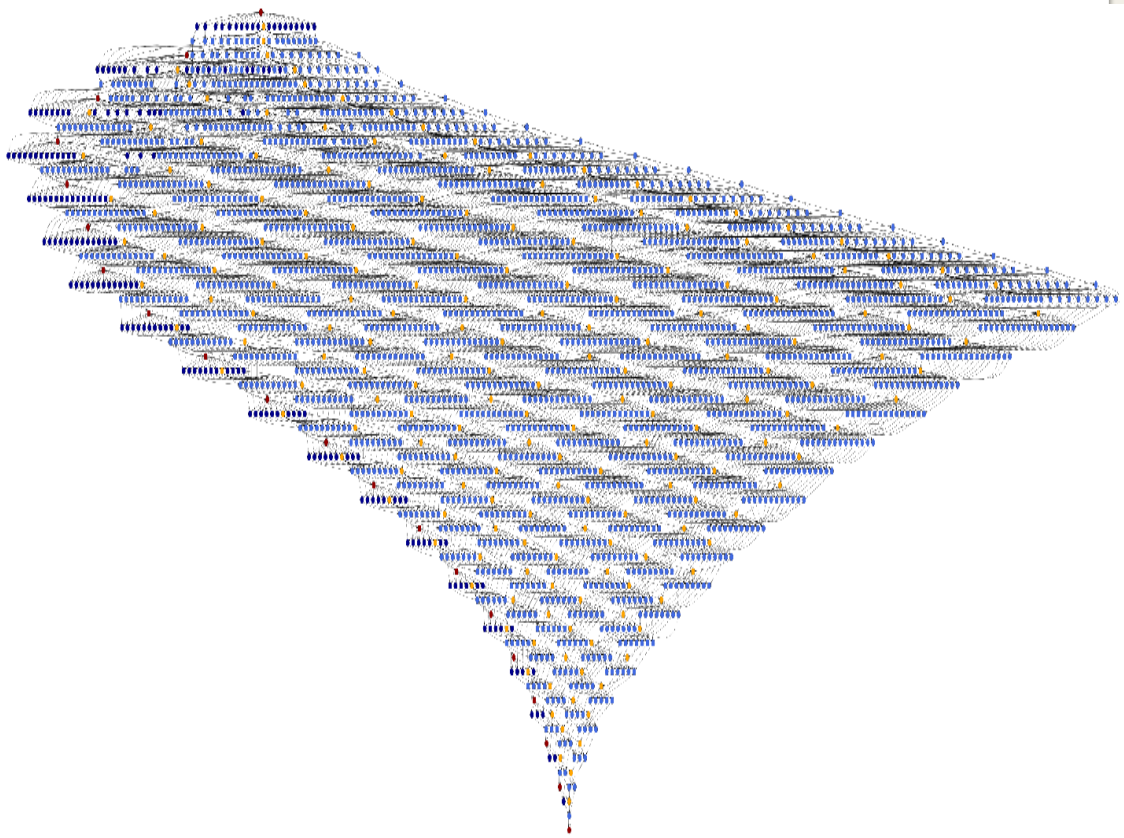
- ✓ Task Classes w/ parameters
  - ✓ geqrt(k), tsqrt(k,m), unmqr(k,n), tsmqr(k,n,m)
- ✓ Precedence constraints between Tasks
- ✓ Compressed form of the Execution DAG
- ✓ Fixed size (problem size independent)

Why not discover the whole DAG?



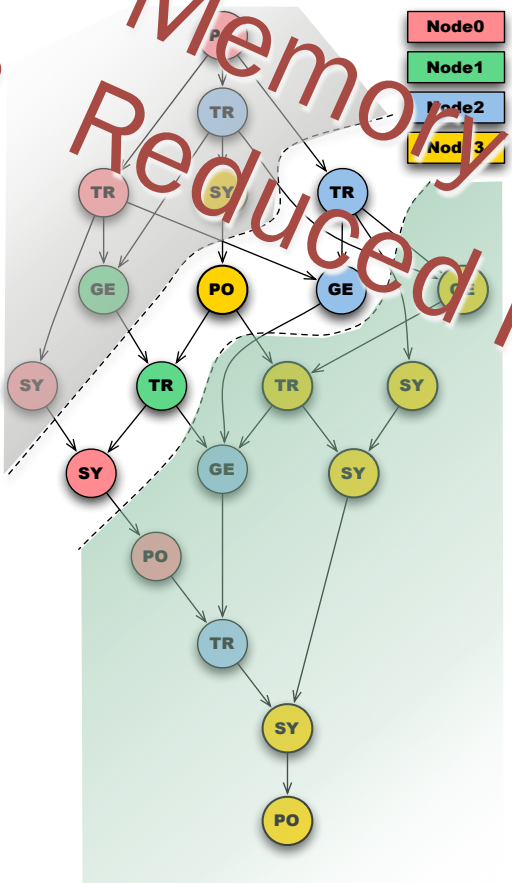
Why not discover the whole DAG?

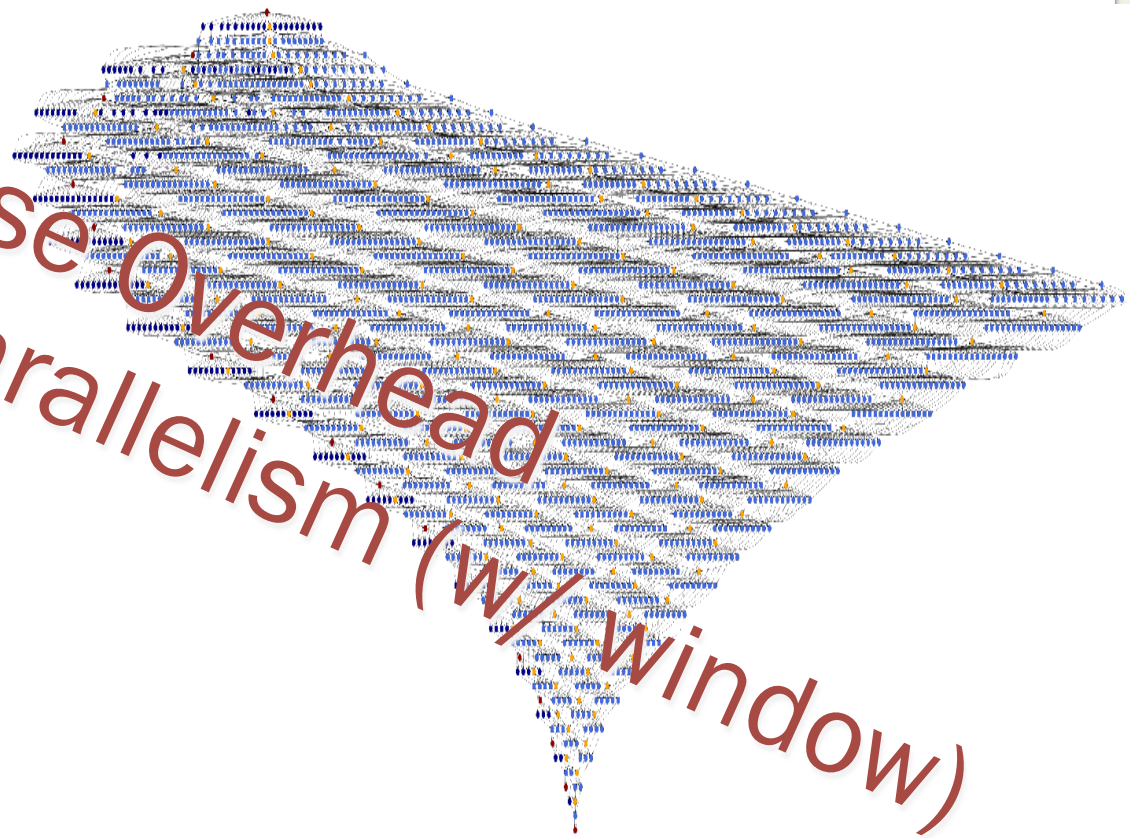




Why not discover the whole DAG?

- Memory Use Overhead
- Reduced Parallelism (w/ window)





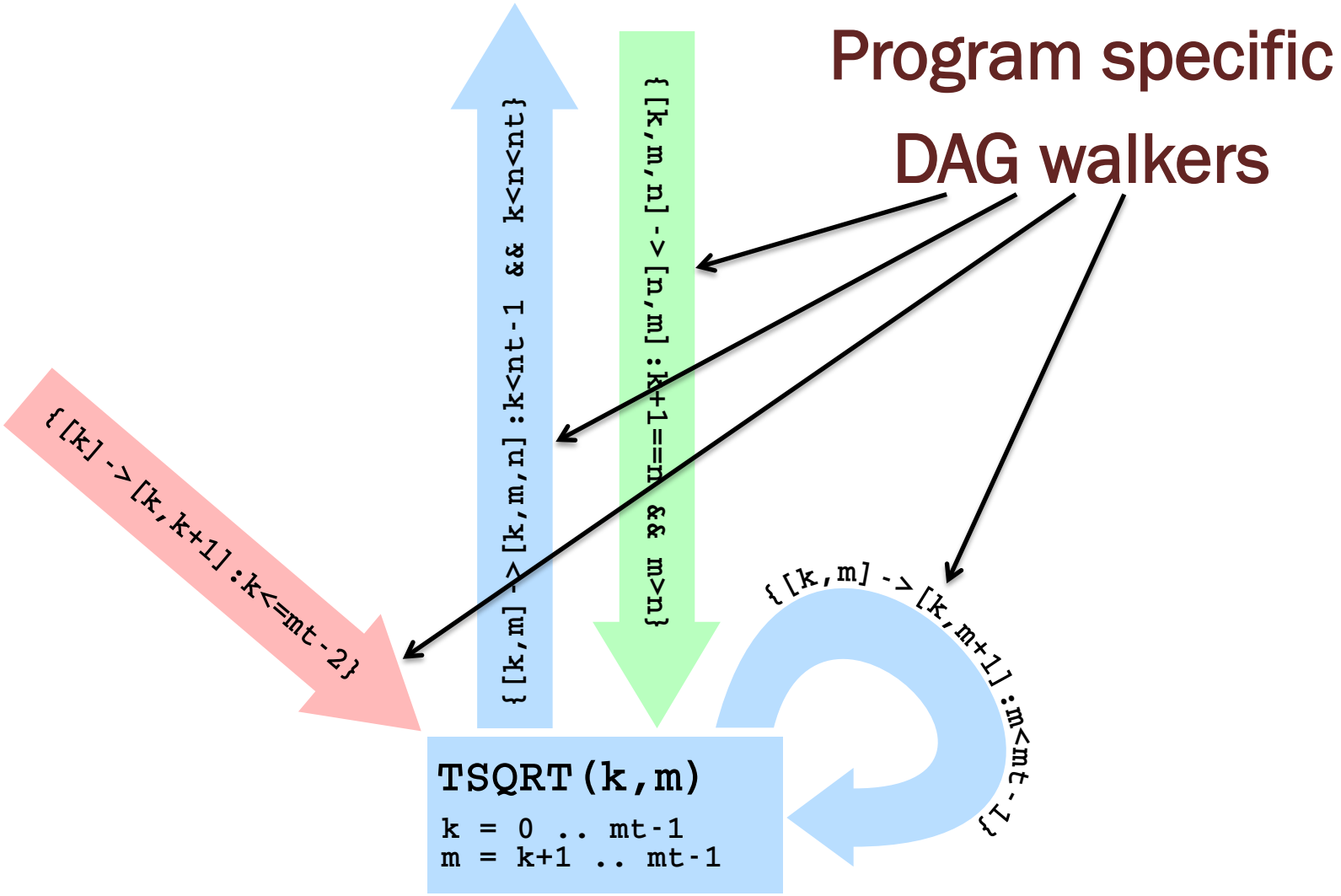
When building the DTG:

The runtime uses a **single:**

- Insert_Task() function
- Structure for the elements of the DAG
- Function for traversing a task's neighborhood

The Dynamic Task Graph is not program specific

When using a PTG:



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
```

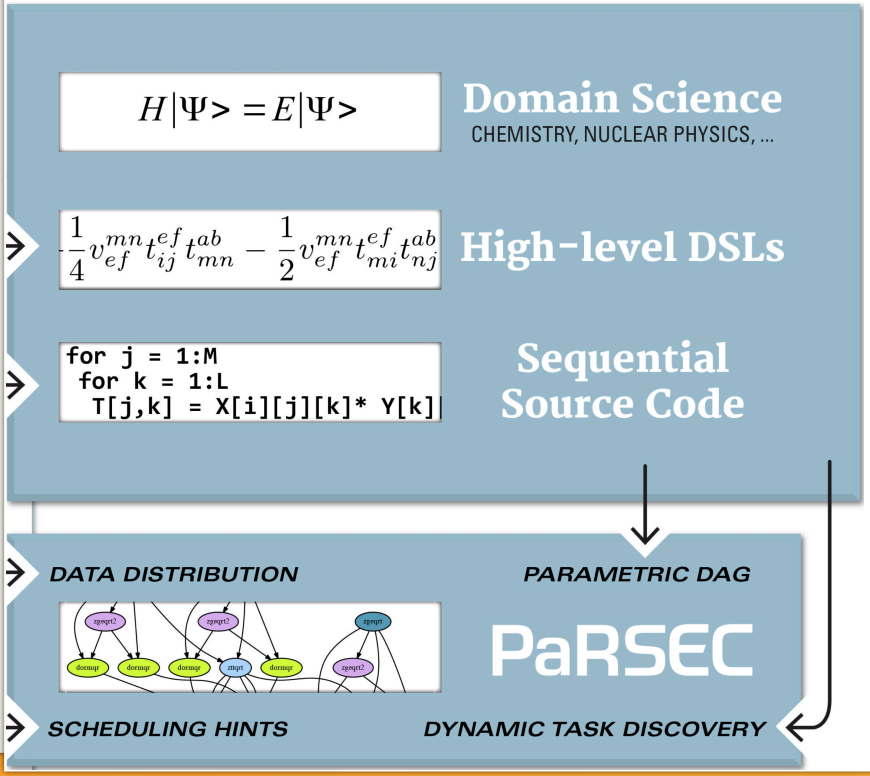
PTG: Binary Tree Reduction

```
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
```

PaRSEC: focus on the algorithm

Concepts

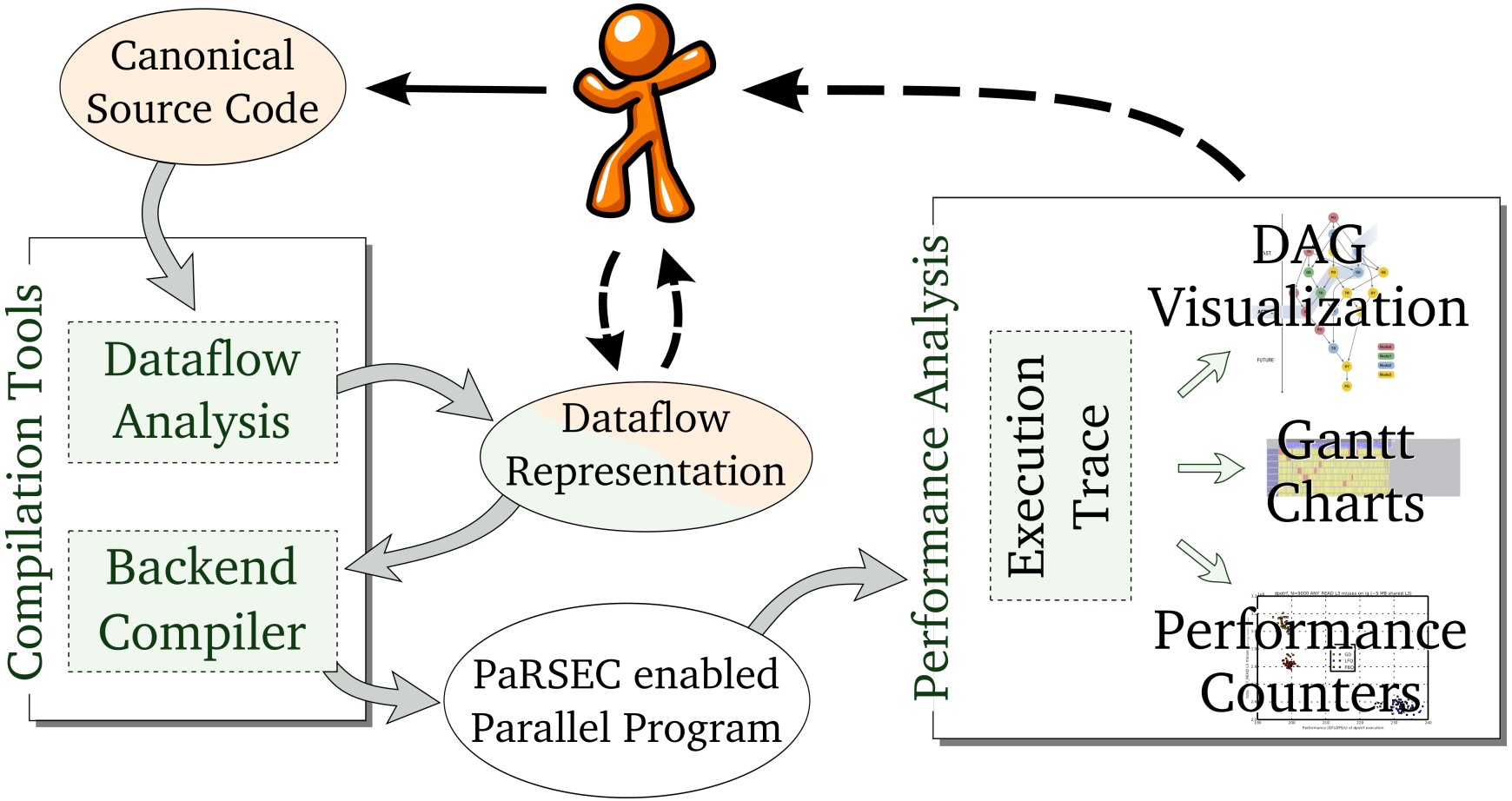
- Separation of roles: compiler optimizes each task, developer describes dependencies between tasks, runtime orchestrates dynamic execution
- Separate algorithms from data distribution
- Avoid limitations of control flow execution



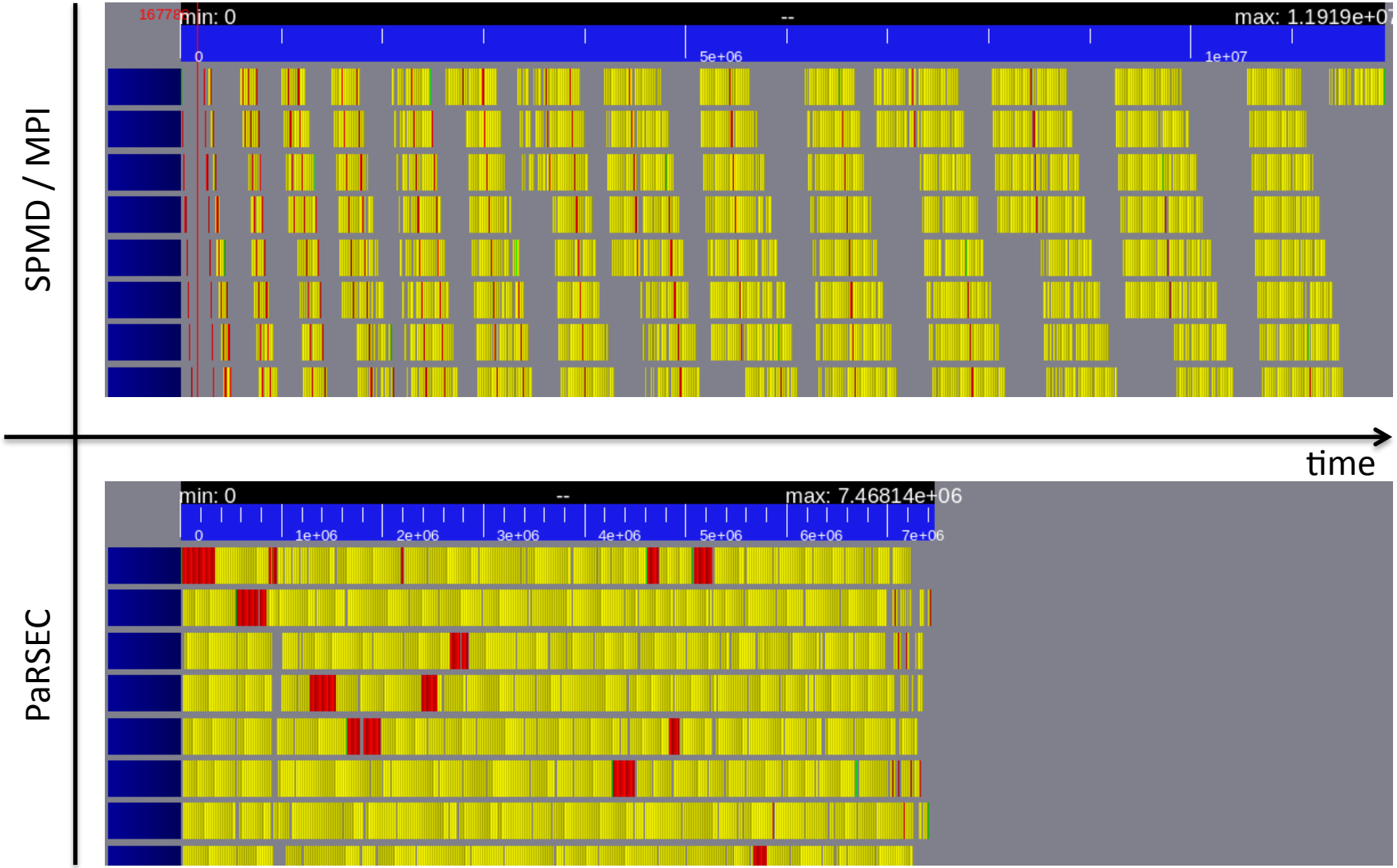
Runtime

- 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

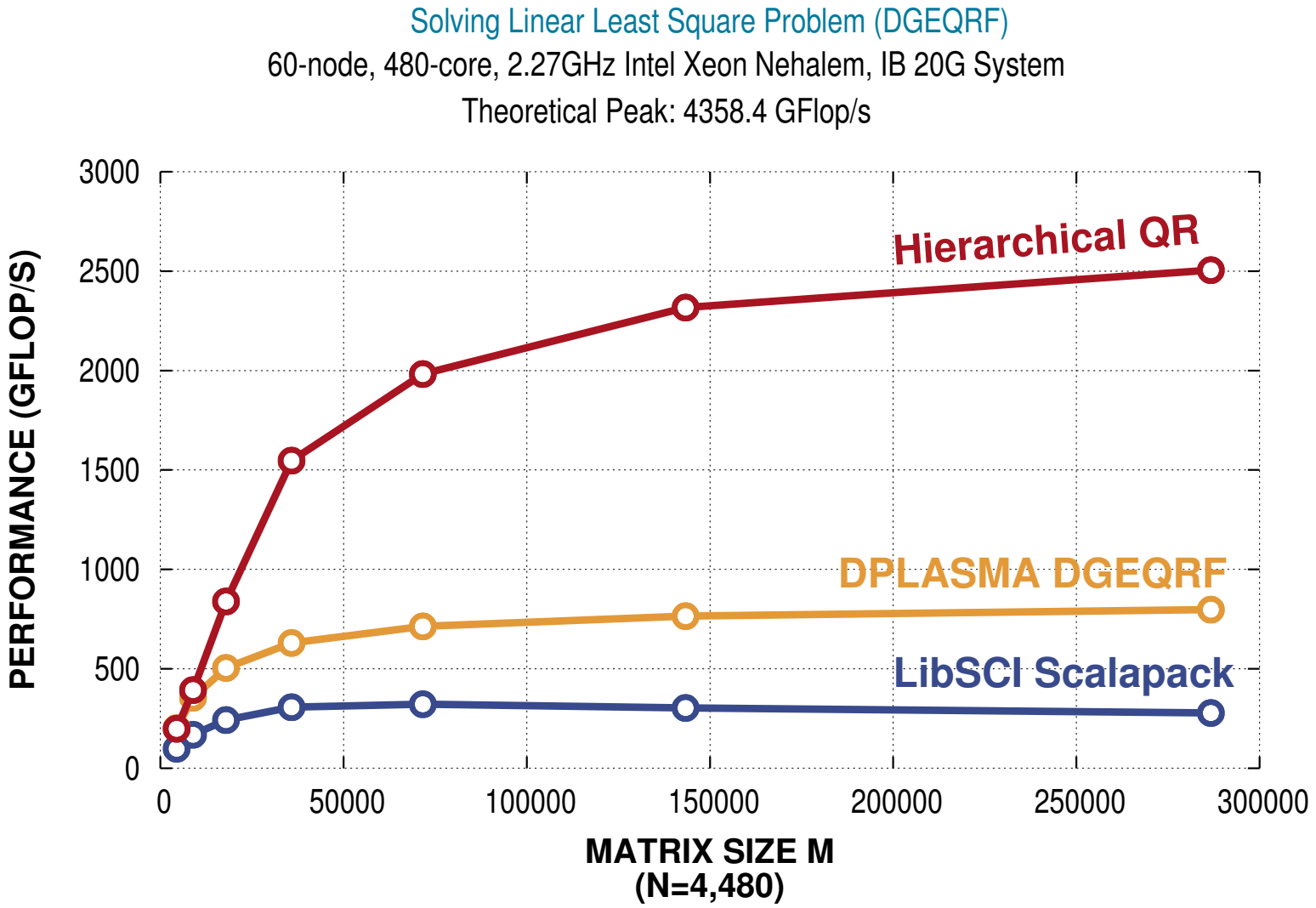
Developer's view of PaRSEC



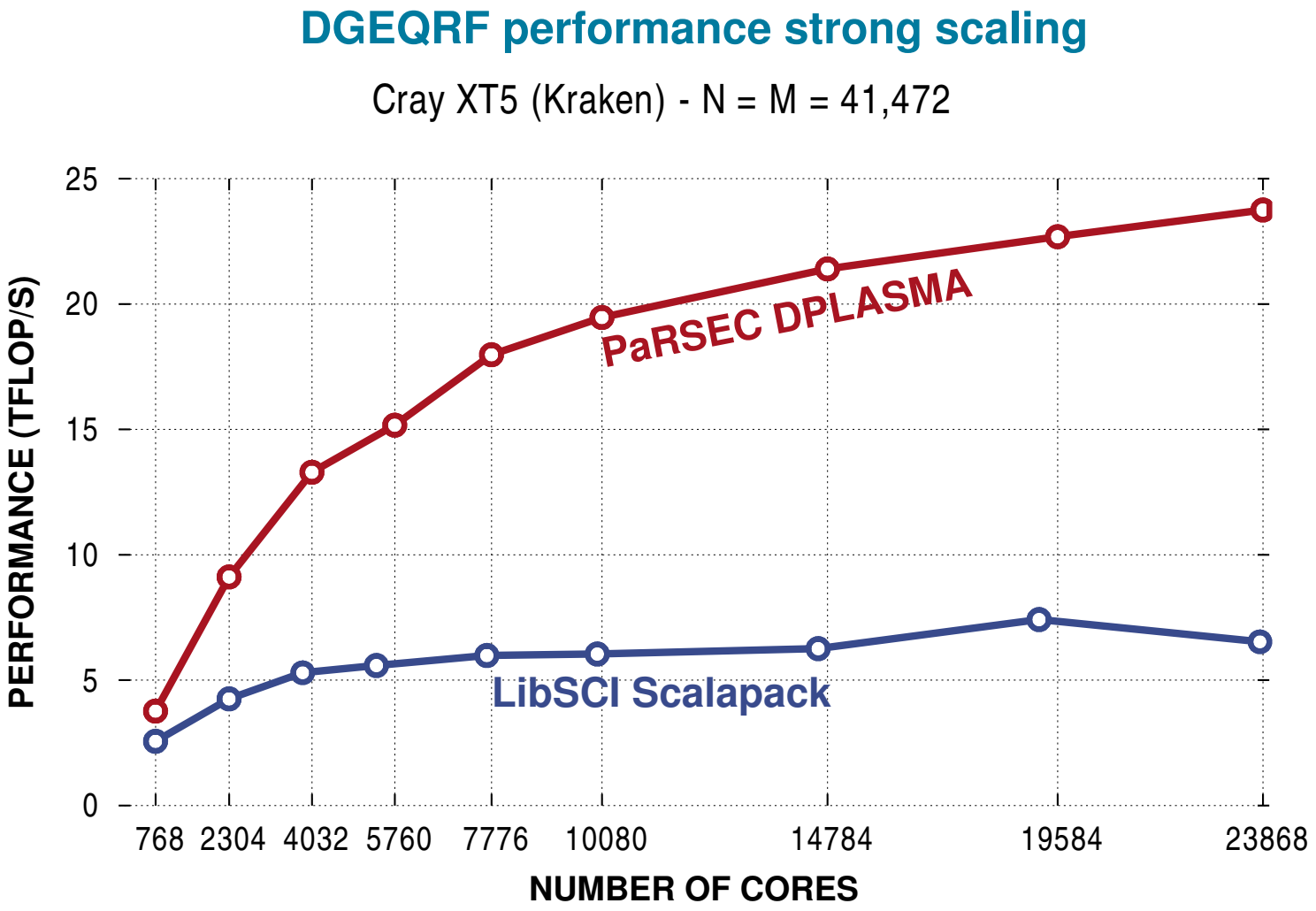
Load Balance, Idle time & Jitter



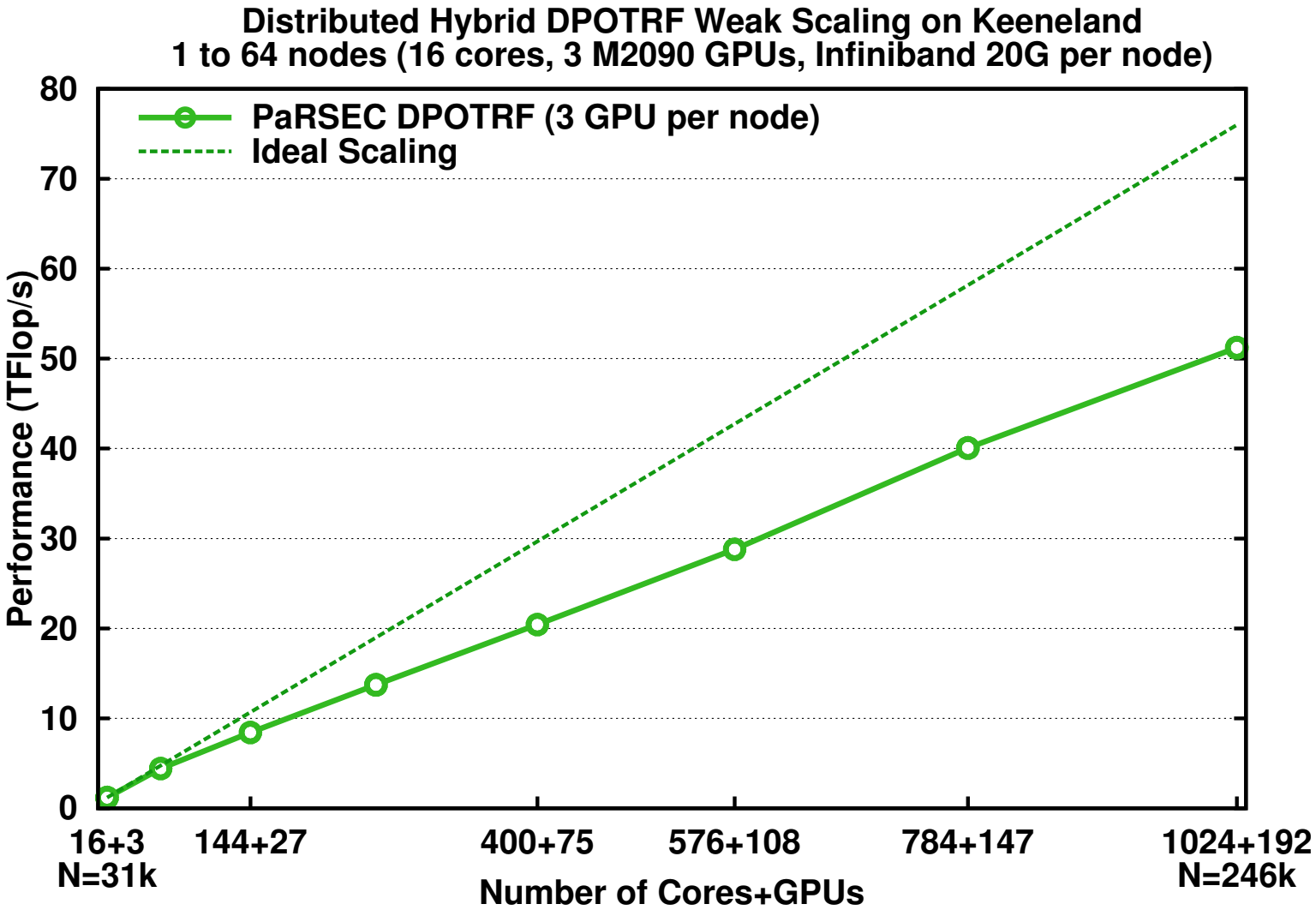
Performance: (H)QR



Performance: Systolic QR



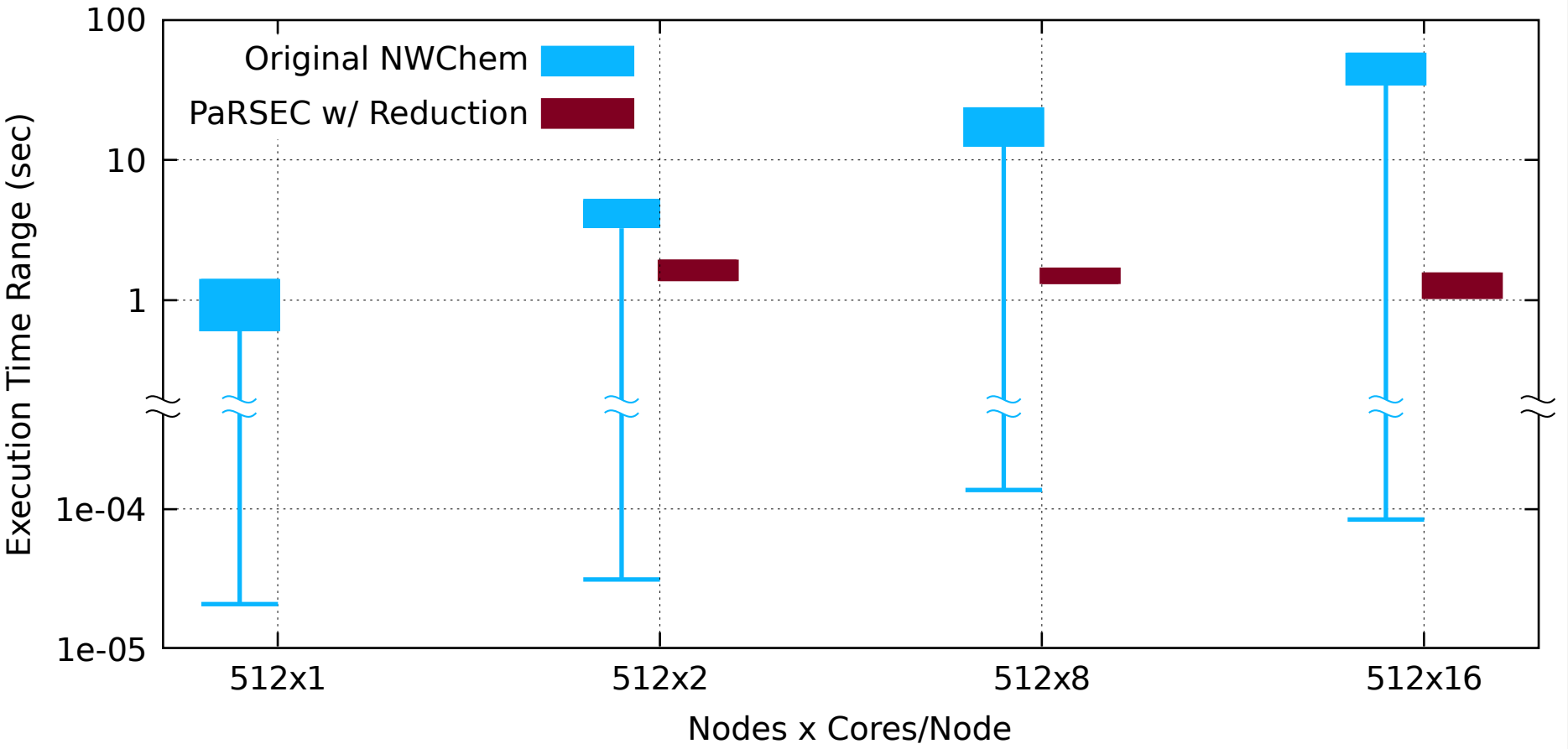
Distributed CPUs + GPUs



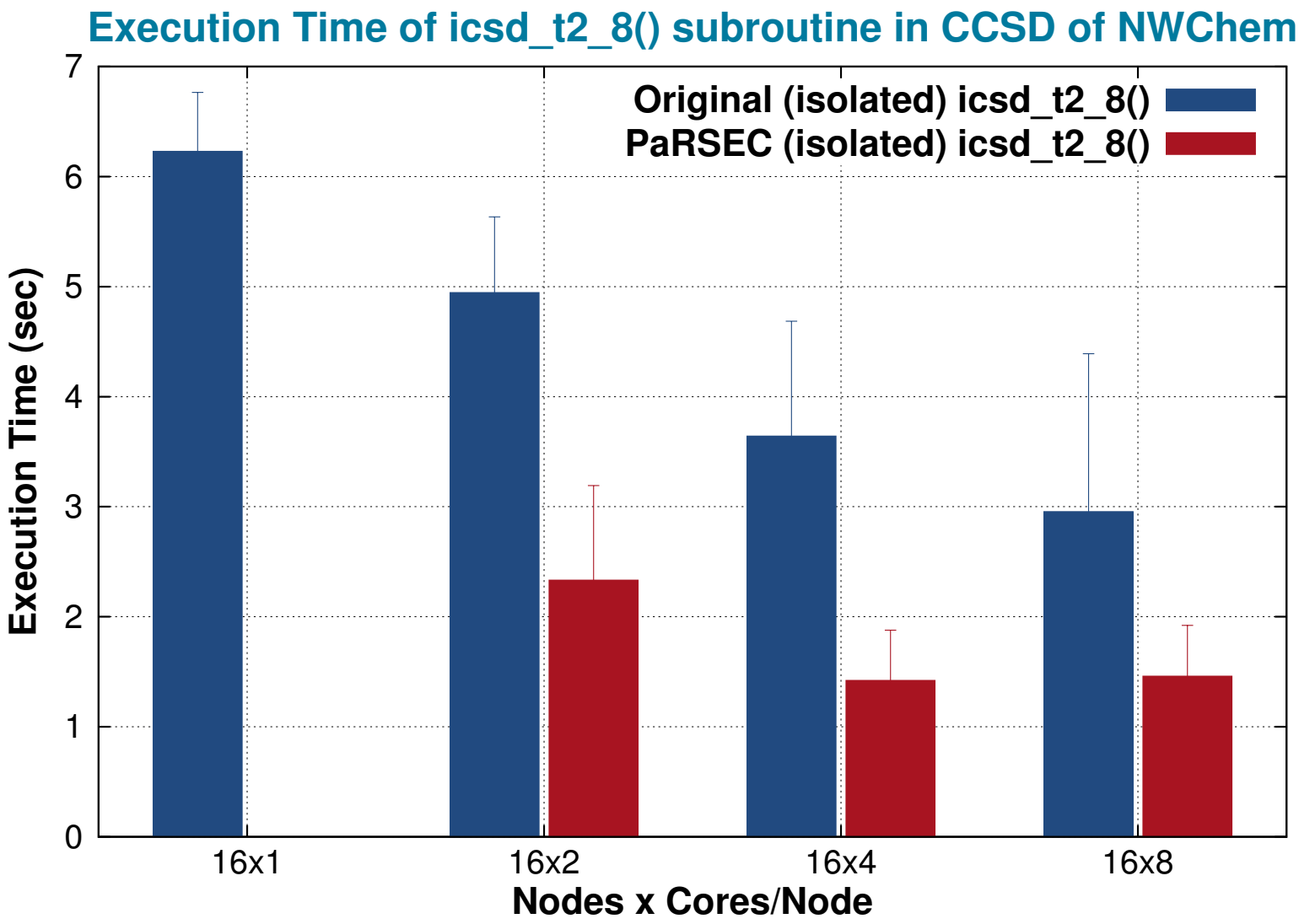
NWChem Coupled Cluster (CC)

- 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


Original NWChem
PaRSEC w/ Reduction
Execution Time Range (sec)
100
10
1
1e-04
1e-05
512x1
512x2
512x8
512x16
Nodes x Cores/Node

CC t2_8


Execution Time of icsd_t2_8() subroutine in CCSD of NWChem
Original (isolated) icsd_t2_8()
PaRSEC (isolated) icsd_t2_8()
Execution Time (sec)
0
1
2
3
4
5
6
7
16x1
16x2
16x4
16x8
Nodes x Cores/Node

Summary

- ✧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 (?)

Backup slides

Why not discover the whole DAG?

Quantify Reduced Parallelism (if using window)

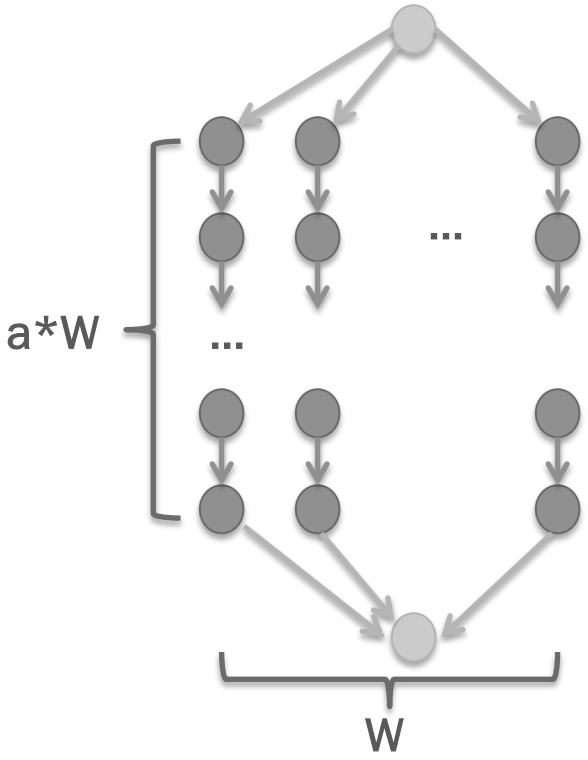
Question:

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

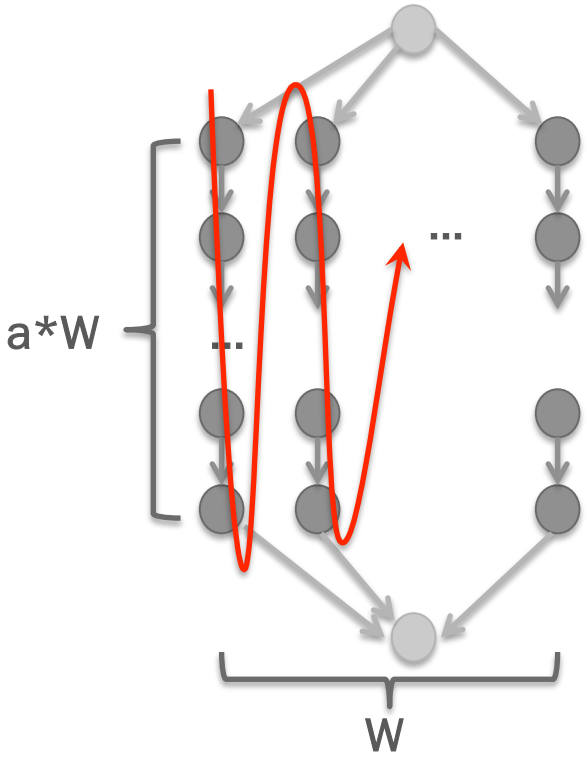
PTG vs DTG

```
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]);
    }
}
```



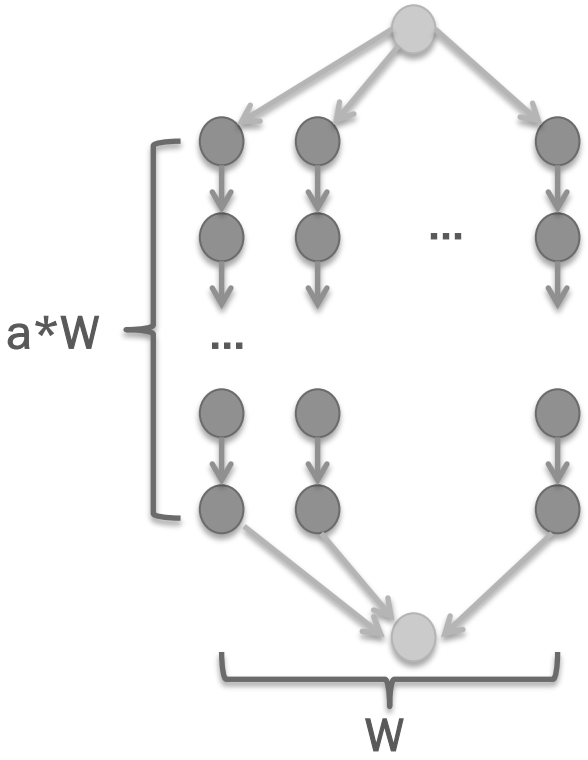
PTG vs DTG

```
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

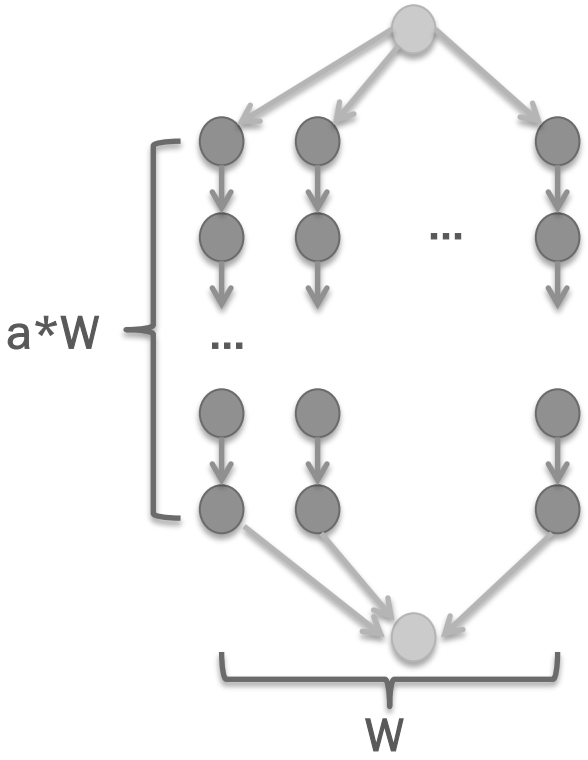
Dynamic Task Graph (DTG):
a*W+(W-1)*(a-1)*W = O(a*W^2)



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