PaRSEC: A Distributed Tasking Environment for scalable hybrid applications

https://bitbucket.org/icldistcomp/parsec

1.3.1.11 STPM11

JLESC Meeting – Urbana-Champaign, July. 18, 2017

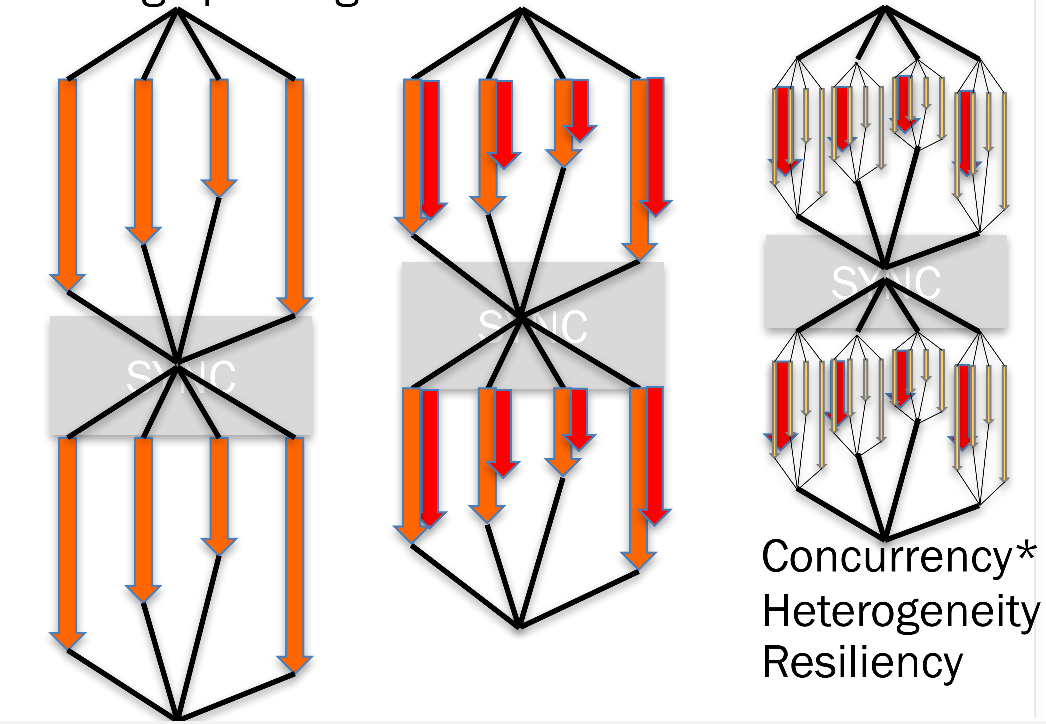
ICL UT INNOVATIVE COMPUTING LABORATORY THE UNIVERSITY of TENNESSEE
IN COLLABORATION WITH Inria
WITH SUPPORT FROM Microsoft
SPONSORED BY U.S. DEPARTMENT OF ENERGY
National Science Foundation

A [very short] history of computing paradigms

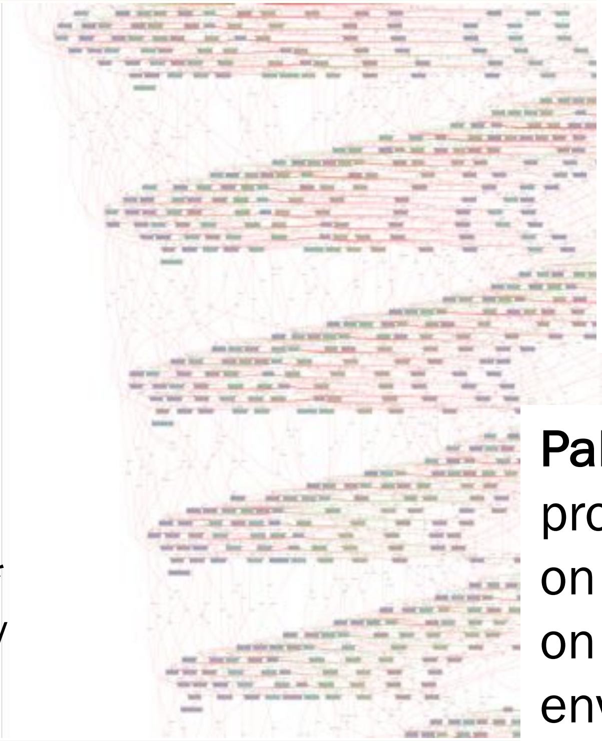
BSP & early message passing

MPI + X

MPI + X + Y + Z + ...



Concurrency*
Heterogeneity
Resiliency



Task-centric runtimes:

- Shared memory: OpenMP, Tascel, Quark, TBB*, PPL, Kokkos**...
- Distributed Memory: StarPU*, StarSS*, DARMA**, Legion, CnC, HPX, Dagger, Hihat**, ...

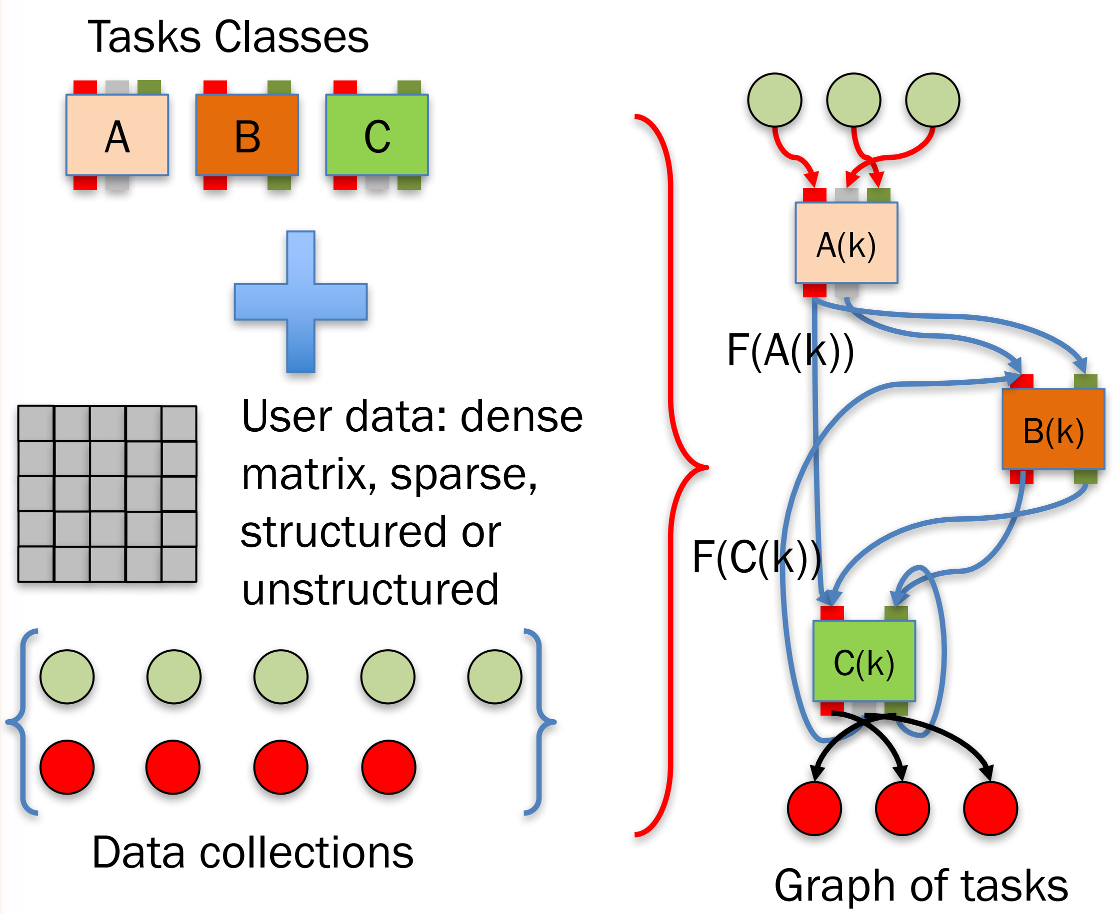
* explicit communications

** nascent effort

PaRSEC: a data centric programming environment based on asynchronous tasks executing on a heterogeneous distributed environment

- Difficult to express the potential algorithmic parallelism
  - Why are we still struggling with control flow ?
  - Software became an amalgam of algorithm, data distribution and architecture characteristics
- Increasing gaps between the capabilities of today's programming environments, the requirements of emerging applications, and the challenges of future parallel architectures
- What is productivity ?

PaRSEC



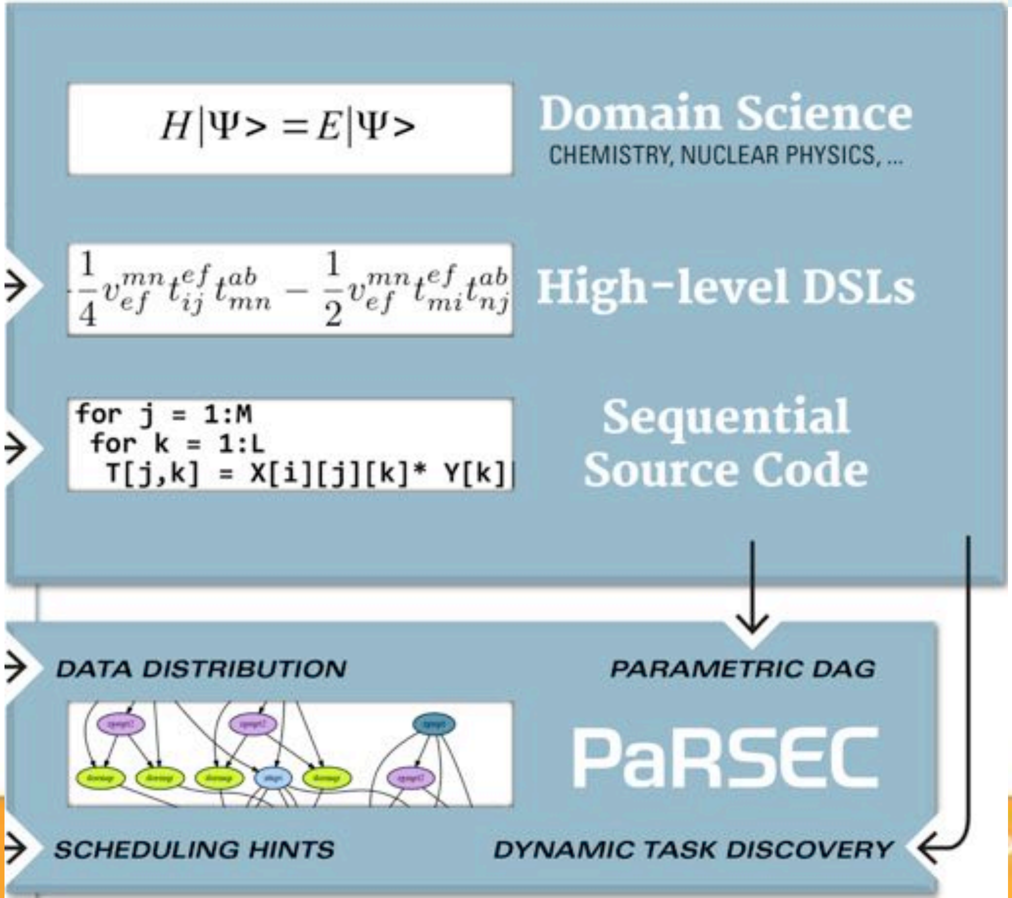
= a data centric programming environment based on asynchronous tasks executing on a heterogeneous distributed environment

- An execution unit taking a set of input data and generating, upon completion, a different set of output data.
- Tasks and data have a coherent distributed scope (managed by the runtime)
- Low-level API allowing the design of Domain Specific Languages (JDF, DTD, TTG)
- Supports distributed heterogeneous environments.
  - Communications are implicit (the runtime moves data)
  - Built-in resilience, performance instrumentation and analysis (R, python)

PaRSEC: a generic runtime system for asynchronous, architecture aware scheduling of fine-grained tasks on distributed many-core heterogeneous architectures

Concepts

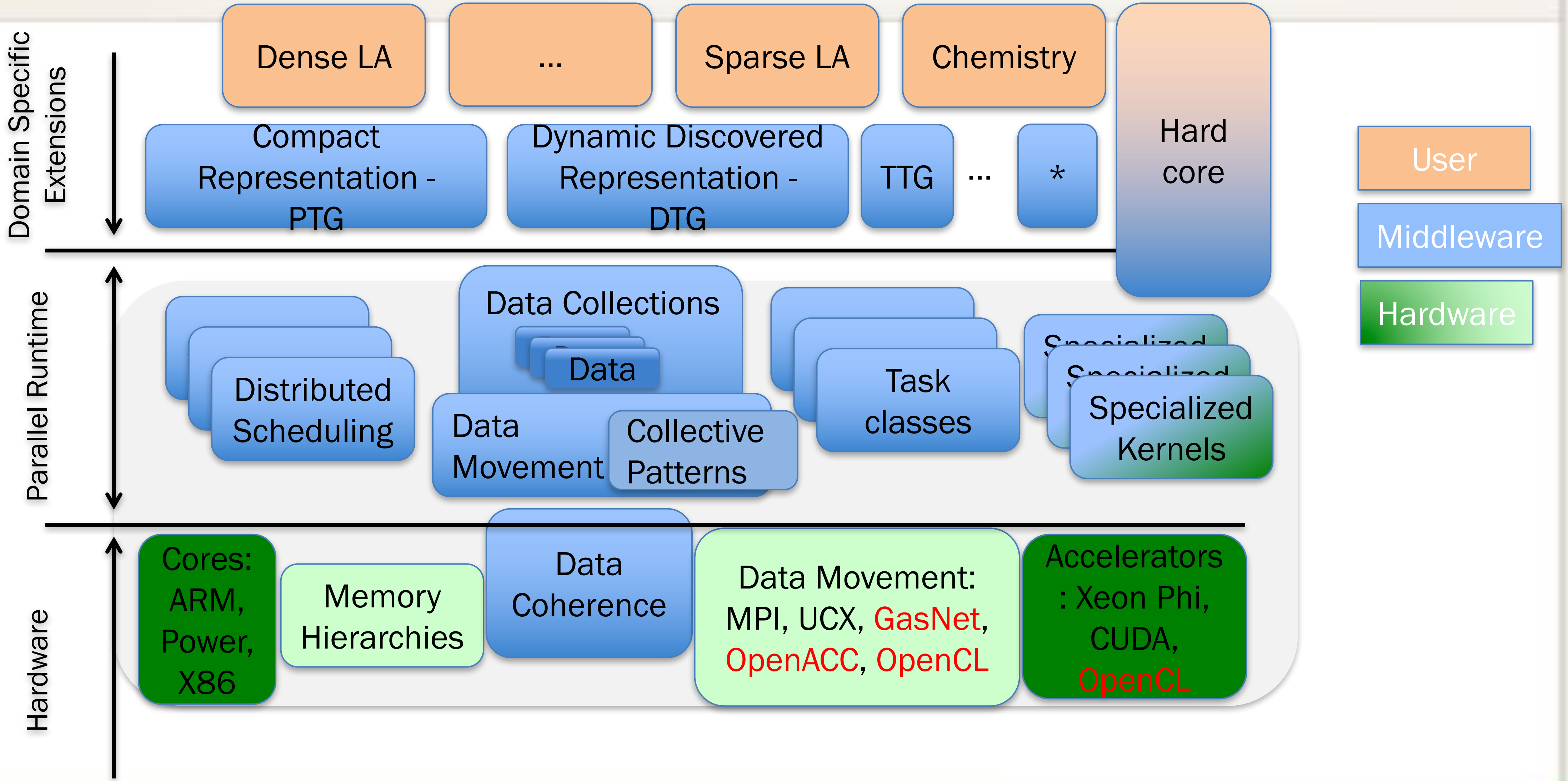
- Clear separation of concerns: compiler optimize each task class, developer describe dependencies between tasks, the runtime orchestrate the dynamic execution
- Interface with the application developers through specialized domain specific languages (PTG/JDF/TTG, Python, insert_task, fork/join, ...)
- Separate algorithms from data distribution
- Make control flow executions a relic



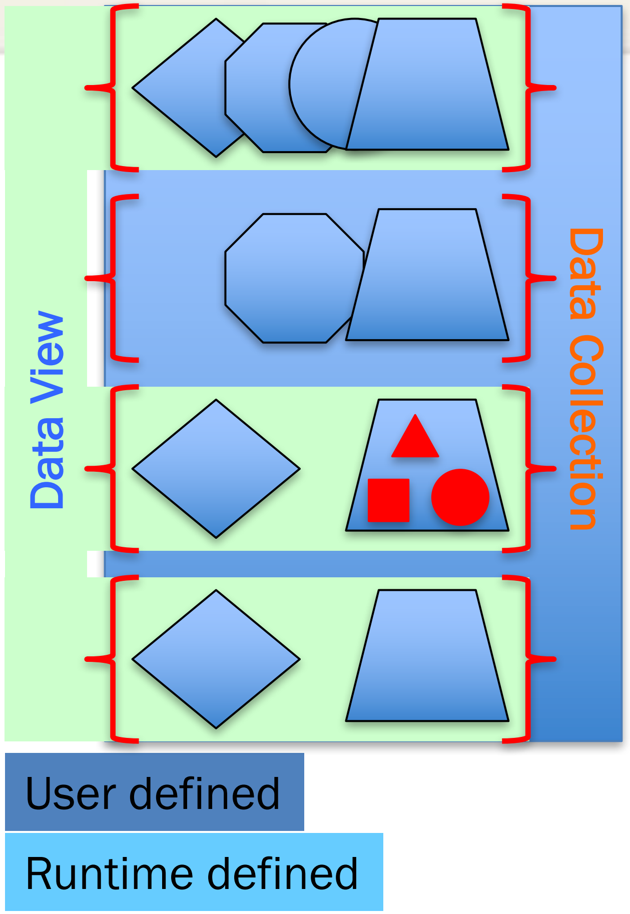
Runtime

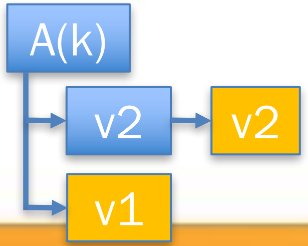
- Portability layer for heterogeneous architectures
- Scheduling policies adapt every execution to the hardware & ongoing system status
- Data movements between producers and consumers are inferred from dependencies. Communications/computations overlap naturally unfold
- Coherency protocols minimize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

The PaRSEC framework



The PaRSEC data





- A data is a manipulation token, the basic logical element (view) used in the description of the dataflow
  - Locations: have multiple coherent copies (remote node, device, checkpoint)
  - Shape: can have different memory layout
  - Visibility: only accessible via the most current version of the data
  - State: can be migrated / logged
- Data collections are ensemble of data distributed among the nodes
  - Can be regular (multi-dimensional matrices)
  - Or irregular (sparse data, graphs)
  - Can be regularly distributed (cyclic-k) or user-defined
- Data View a subset of the data collection used in a particular algorithm (aka. submatrix, row, column,...)

- A data-copy is the practical unit of data
  - Has a memory layout (think MPI datatype)
  - Has a property of locality (device, NUMA domain, node)
  - Has a version associated with
  - Multiple instances can coexist

A PaRSEC application

Start PaRSEC

```
parsec_context_t* parsec;
parsec = parsec_context_init(NULL, NULL);  /* start a PaRSEC engine */
```

Create a tasks placeholder and associate it with the PaRSEC context

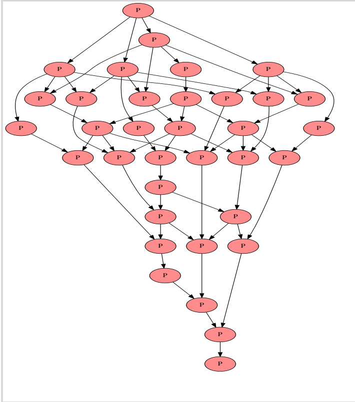
```
parsec_taskpool_t* parsec_tp = parsec_taskpool_new ();
parsec_enqueue(parsec, parsec_tp);
```

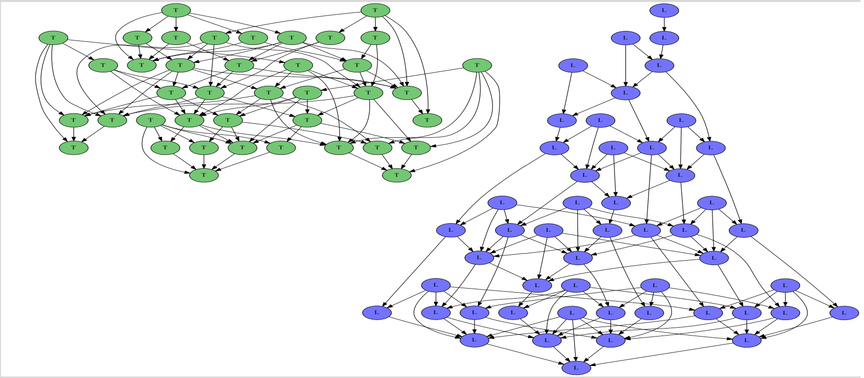
Define a distributed collection of data (vector)

```
parsec_vector_t dDATA;
parsec_vector_init( &dDATA, matrix_Integer, matrix_Tile,
                    nodes, rank,
                    1, /* tile_size*/
                    N, /* Global vector size*/
                    0, /* starting point */
                    1 );  /* block size */
```

Data initialization and PaRSEC context setup. Common to all DSL

Add tasks.



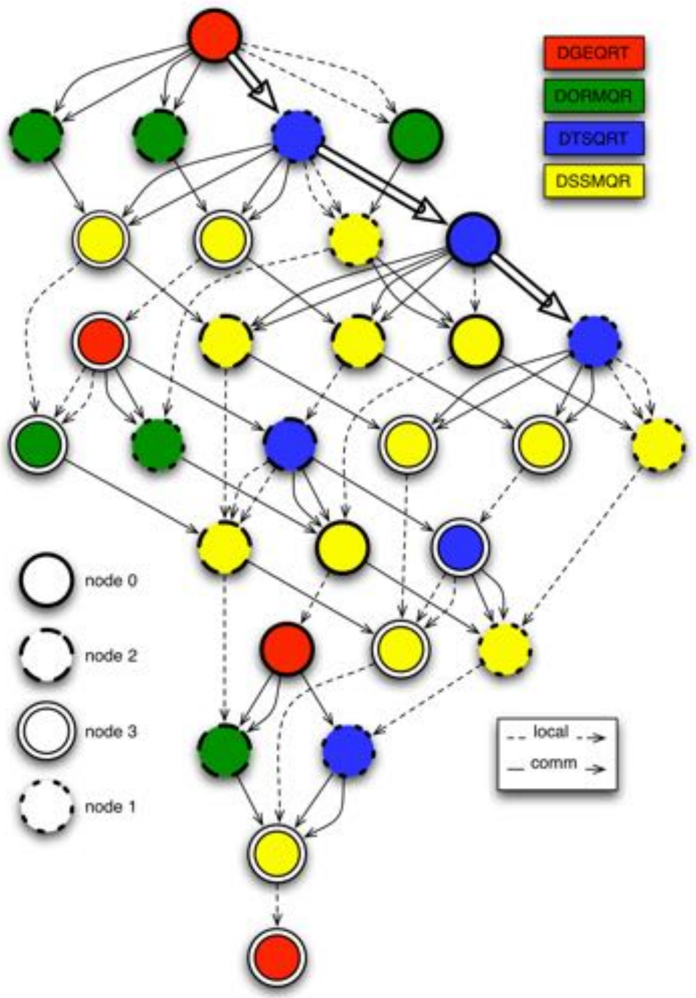


Wait 'till completion

```
parsec_taskpool_wait( parsec_tp);
```

How to describe a graph of tasks ?

- Uncountable ways
  - Generic: Dagguer (Charm++), Legion, ParalleX, Parameterized Task Graph (PaRSEC), Dynamic Task Discovery (StarPU, StarSS), Yvette (XML), Fork/Join (spawn). CnC, Uintah, DARMA, Kokkos, RAJA
  - Application specific: MADNESS
- PaRSEC runtime
  - The runtime is agnostic to the domain specific language (DSL)
  - Different DSL interoperate through the data collections
  - The DSL share
    - Distributed schedulers
    - Communication engine
    - Hardware resources
    - Data management (coherence, versioning, …)
  - They don’t share
    - The task structure
    - The internal dataflow



DSL: The insert_task interface

Data initialization and PaRSEC context setup. Common to all DSL

Start PaRSEC

```
parsec_context_t* parsec;
parsec = parsec_context_init(NULL, NULL);  /* start a PaRSEC engine */
```

Create a tasks placeholder and associate it with the PaRSEC context

```
parsec_taskpool_t* parsec_tp = parsec_taskpool_new ();
parsec_enqueue(parsec, parsec_tp);
```

Define a distributed collection of data (vector)

```
parsec_vector_t dDATA;
parsec_vector_init( &dDATA, matrix_Integer, matrix_Tile,
                    nodes, rank,
                    1, /* tile_size*/
                    N, /* Global vector size*/
                    0, /* starting point */
                    1 );  /* block size */
```

Add tasks.

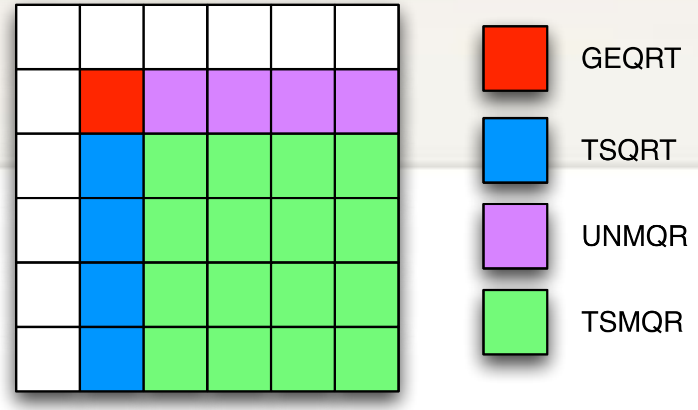
```
for( n = 0; n < N-2; n += 2 ) {
    parsec_insert_task( parsec_tp,
       ping_task,    “PING",
       PASSED_BY_REF,   DATA_AT(&dDATA, n),    INPUT | FULL,
       PASSED_BY_REF,   DATA_AT(&dDATA, n+1),   OUT | FULL | HERE,
       0 /* Last Argument */);
    parsec_insert_task( parsec_tp,
       pong_task,    “PONG",
       PASSED_BY_REF,   DATA_AT(&dDATA, n+1),    INPUT | FULL,
       PASSED_BY_REF,   DATA_AT(&dDATA, n+2),     OUT | FULL | HERE,
       0 /* Last Argument */); }
```

Wait ’till completion

```
parsec_taskpool_wait( parsec_tp);
```

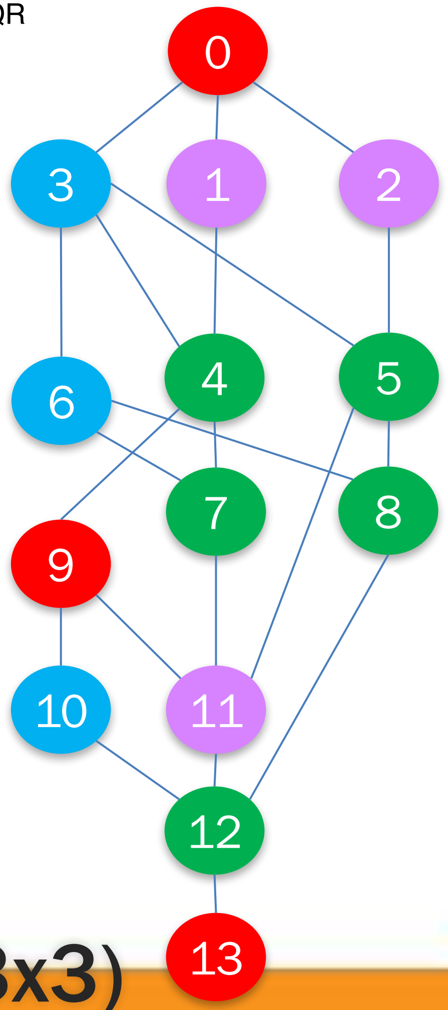
DSL: insert_task

```
for( k = 0; k < SIZE; k++ ) {
        parsec_insert_task( "GEQRT",
                            DATA_OF(A, k, k),  INOUT|AFFINITY,
                            DATA_OF(T, k, k),  OUTPUT|TILE_RECT)

        for( n = k+1; n < SIZE; n++ )
                parsec_insert_task( "UNMQR",
                                    DATA_OF(A, k, k),  INPUT|TILE_L,
                                    DATA_OF(T, k, k),  INPUT|TILE_RECT,
                                    DATA_OF(A, k, n),  INOUT|AFFINITY)

        for( m = k+1; m < SIZE; m++ ) {
                parsec_insert_task( "TSQRT",
                                    DATA_OF(A, k, k),  INOUT|TILE_U,
                                    DATA_OF(A, m, k),  INOUT|AFFINITY,
                                    DATA_OF(T, m, k),  OUTPUT|TILE_RECT)

                for( n = k+1; n < SIZE; n++ ) {
                        parsec_insert_task( "TSMQR",
                                            DATA_OF(A, k, n),  INOUT,
                                            DATA_OF(A, m, n),  INOUT|AFFINITY,
                                            DATA_OF(A, m, k),  INPUT,
                                            DATA_OF(T, m, k),  INPUT|TILE_RECT)
                }
        }
}
```



A

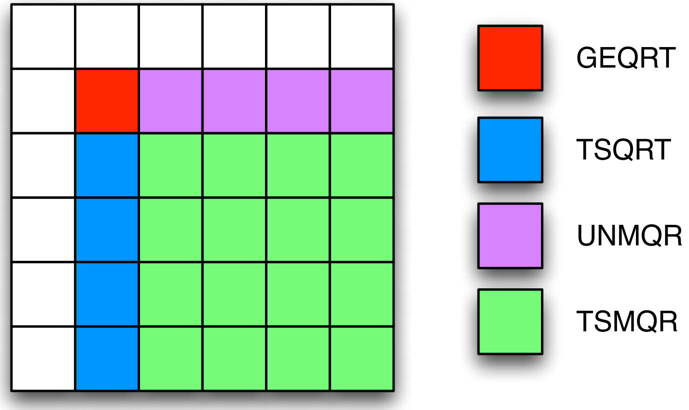
0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2



QR Factorization (3x3)

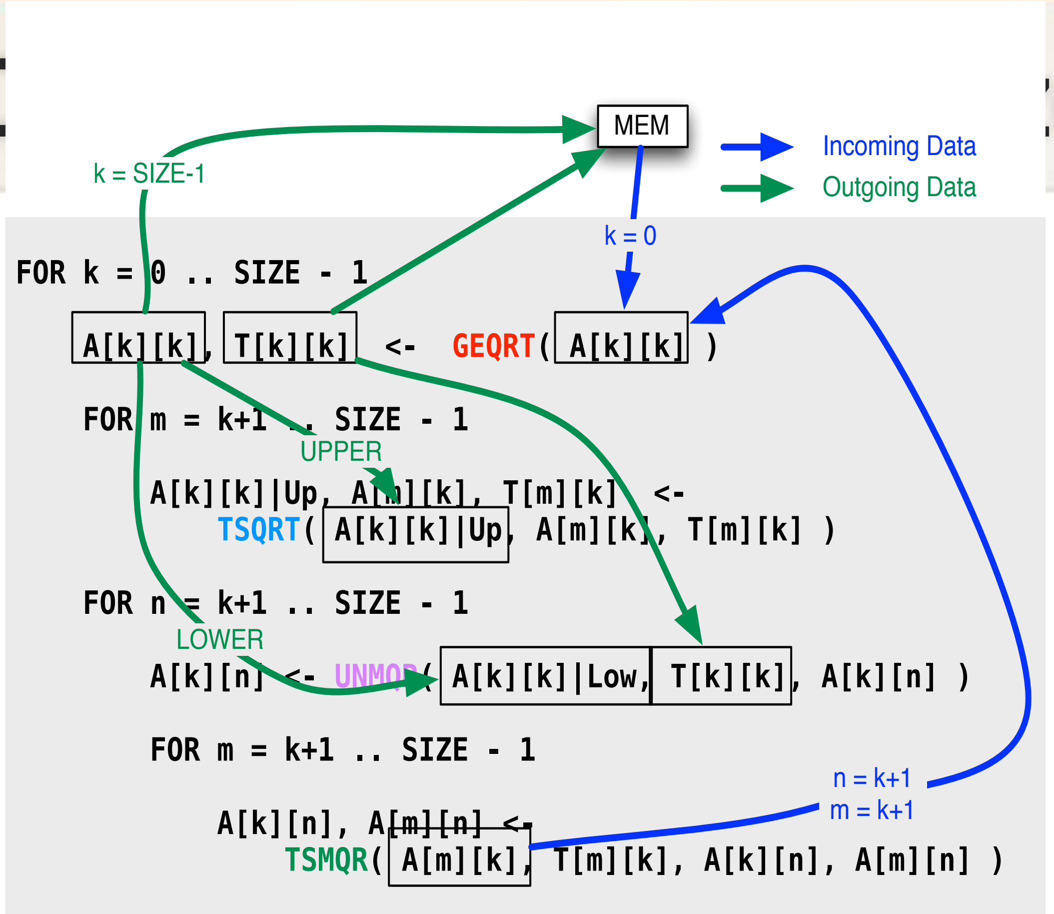
DSL: The Parameterized Task Graph (JDF)

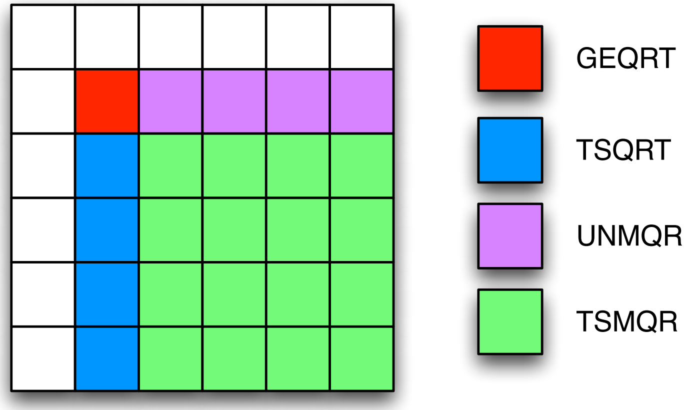
```
FOR k = 0 .. SIZE - 1

    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] )
```



- A dataflow description based on data tracking
- A simple affine description of the algorithm can be understood and translated by a compiler into a more complex, control-flow free, form
- Abide to all constraints imposed by current compiler technology

D… ed Task Graph (JDF)





- A dataflow description based on data tracking
- A simple affine description of the algorithm can be understood and translated by a compiler into a more complex, control-flow free, form
- Abide to all constraints imposed by current compiler technology

The Parameterized Task Graph (JDF)

```
GEQRT(k)

 k = 0..( MT < NT ) ? MT-1 : NT-1 )

 : 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)

BODY [type = CPU]  /* default */
   zgeqrt( A, T );
END

BODY [type = CUDA]
   cuda_zgeqrt( A, T );
END
```

Control flow is eliminated, therefore maximum parallelism is possible

- A dataflow parameterized and concise language
- Accept non-dense iterators
- Allow inlined C/C++ code to augment the language [any expression]

- Termination mechanism part of the runtime (i.e. needs to know the number of tasks per node)
- The dependencies had to be globally (and statically) defined prior to the execution
  - Dynamic DAGs non-natural
  - No data dependent DAGs

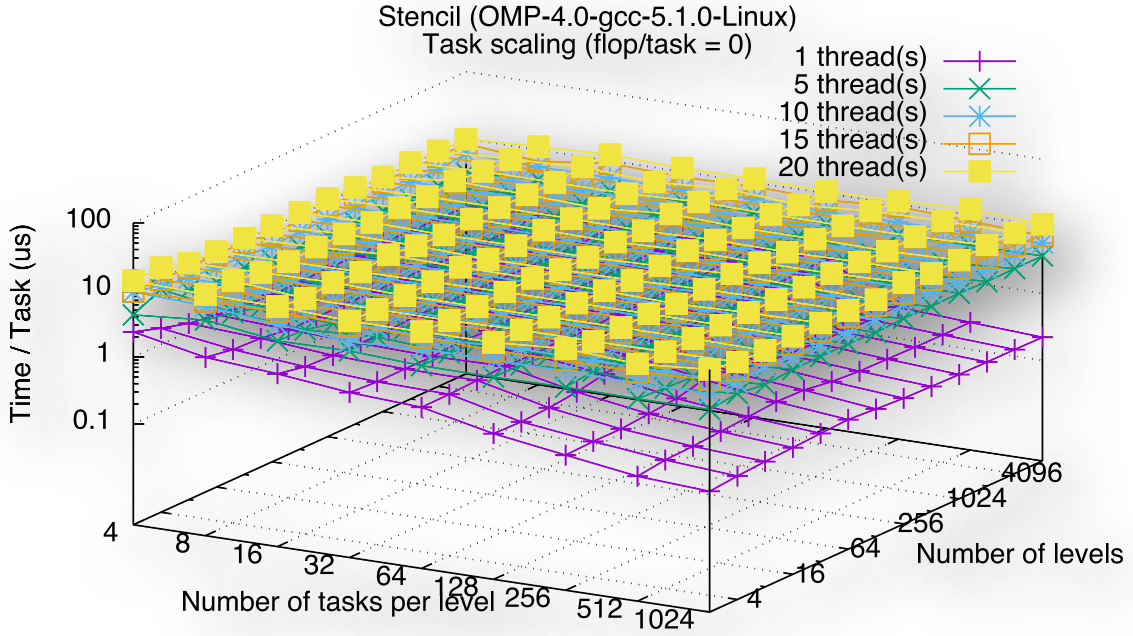
Relaxing constraints: Unhindered parallelism

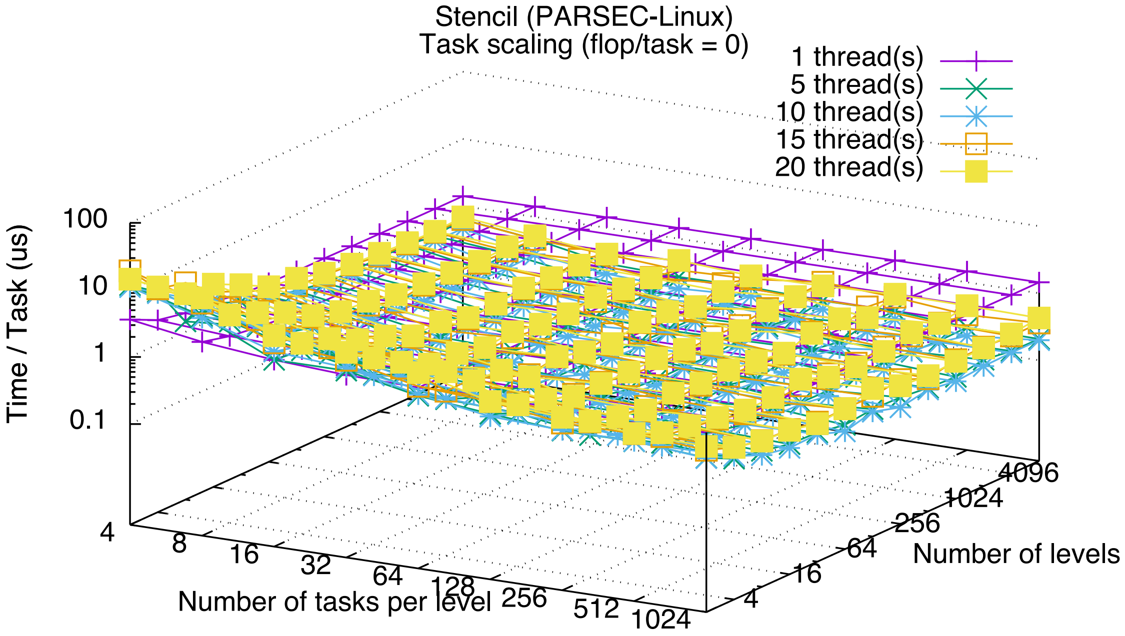
- The only requirement is that upon a task completion the descendants are locally known
  - Information packed and propagated to participants where the descendent tasks are supposed to execute
- Uncountable DAGs
  - ” %option nb_local_tasks_fn = …”
  - Provide support for user defined global termination
- Add support for dynamic DAGs
  - Properties of the algorithm / tasks
    - ”hash_fn = …”
    - ”find_deps_fn = …”

Evaluating the scheduling overhead

Benchmarking the scheduling overhead on 1D-stencil problem.

- Tasks are no-op, 0 flops per task;
- OpenMP in gcc 5.1 vs PaRSEC-rc1;

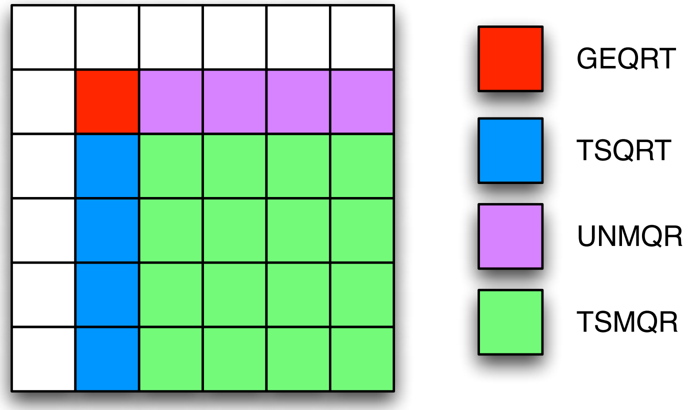


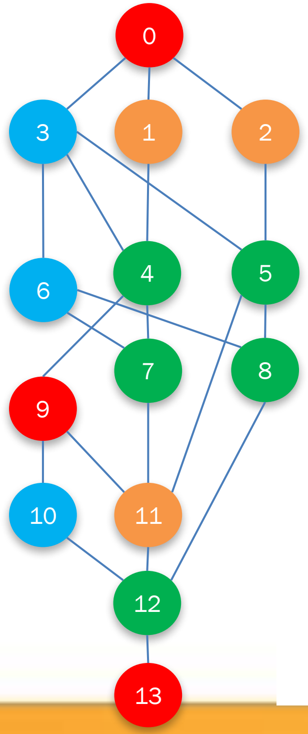


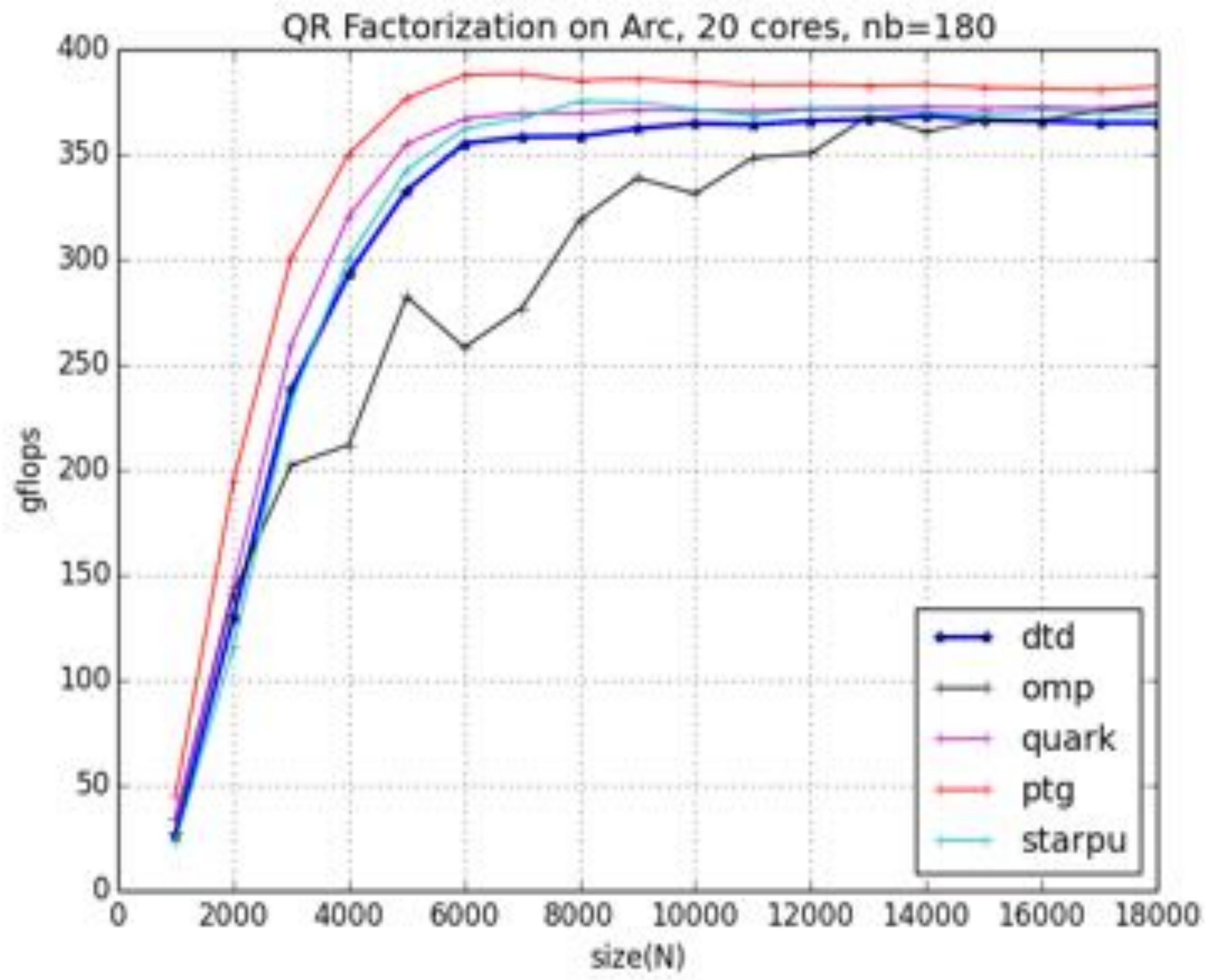
QR factorization: shared memory

Experiments on Arc machines,

- E5-2650 v3 @ 2.30GHz
- 20 cores
- gcc 6.3
- MKL 2016
- PaRSEC-2.0-rc1
- StarPU 1.2.1
- PLASMA 1.8



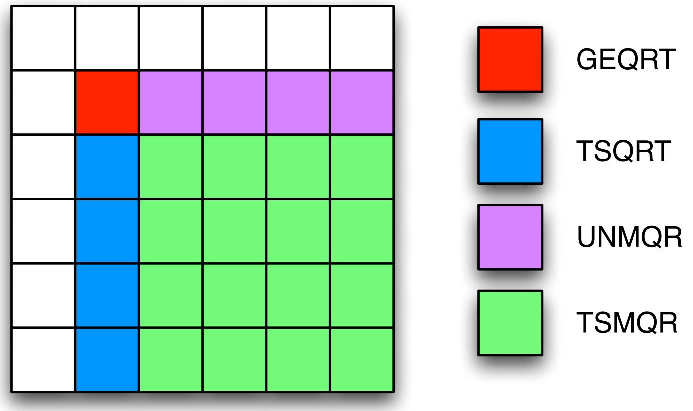


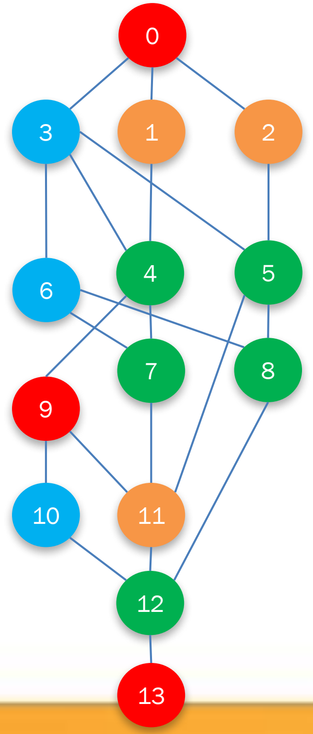


QR factorization: heterogeneous

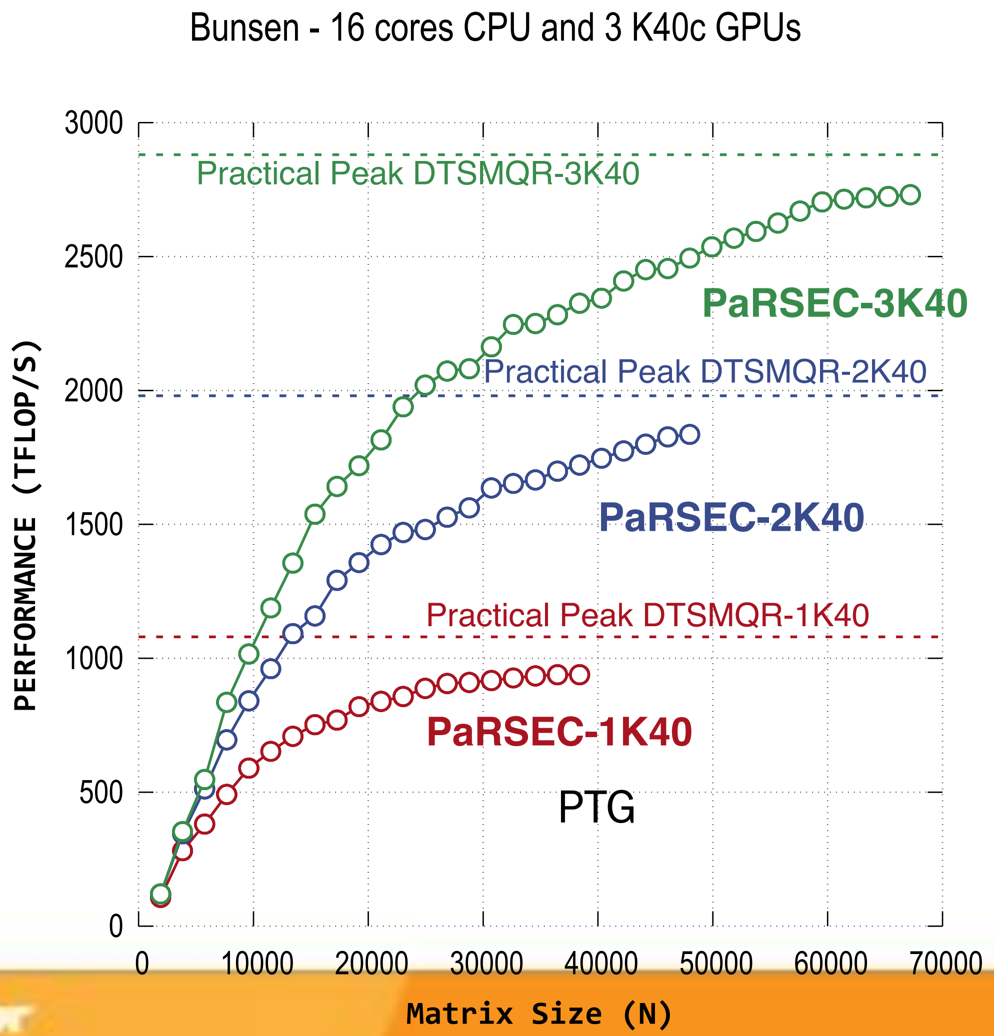
Experiments on Arc machines,

- E5-2650 v3 @ 2.30GHz
- 20 cores
- gcc 6.3
- MKL 2016
- PaRSEC-2.0-rc1
- StarPU 1.2.1
- CUDA 7.0





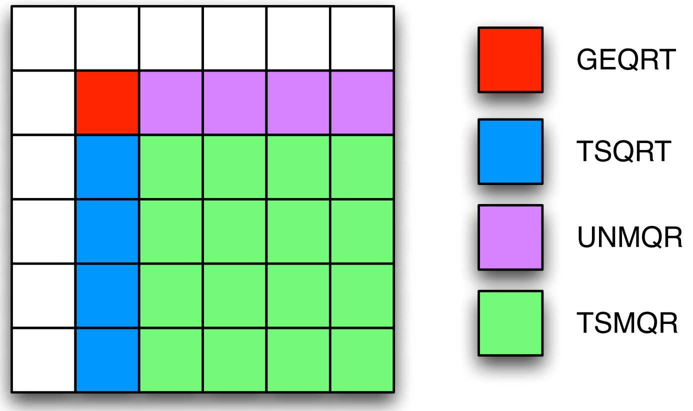
DGEQRF performance problem scaling

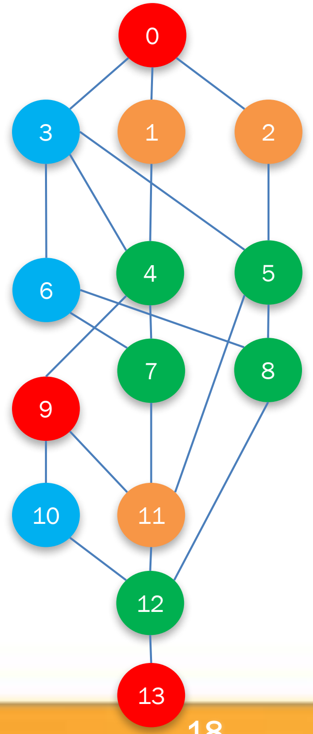


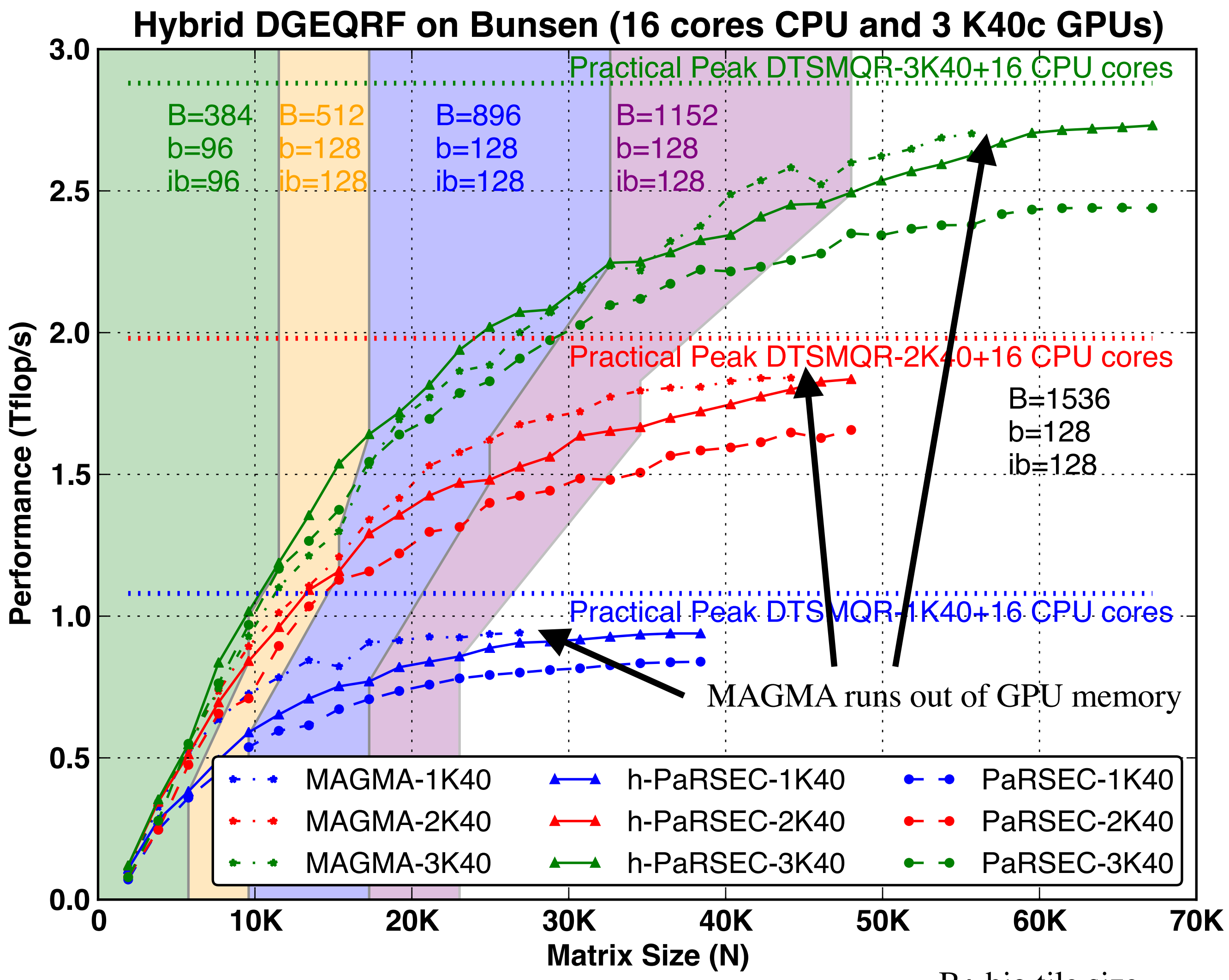
QR factorization: heterogeneous

Experiments on Arc machines,

- E5-2650 v3 @ 2.30GHz
- 20 cores
- gcc 6.3
- MKL 2016
- PaRSEC-2.0-rc1
- StarPU 1.2.1
- CUDA 7.0





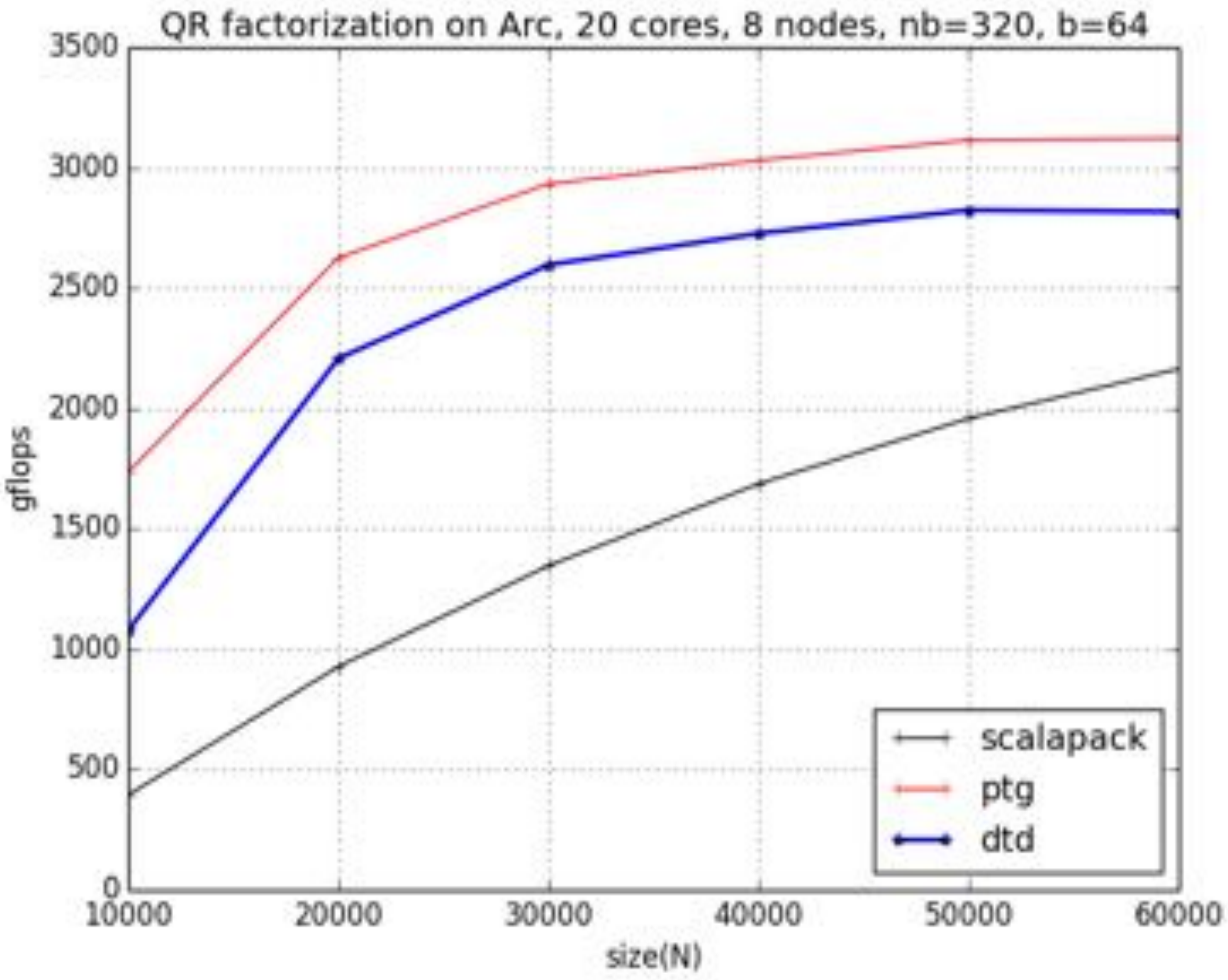


B: big tile size
b: small tile size
ib: inner block size

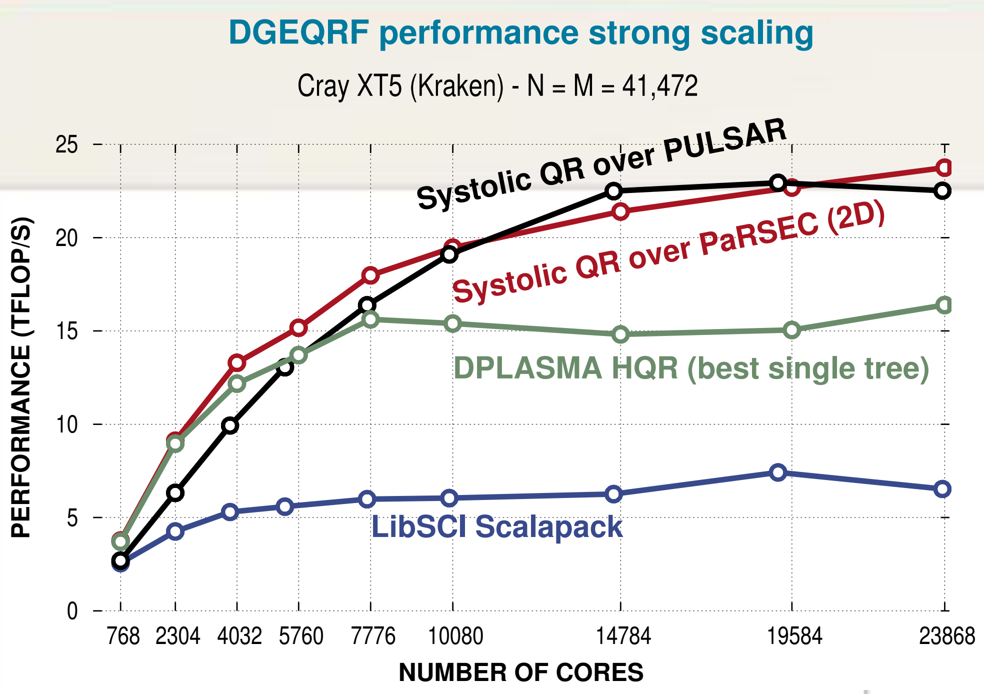
QR factorization: distributed memory

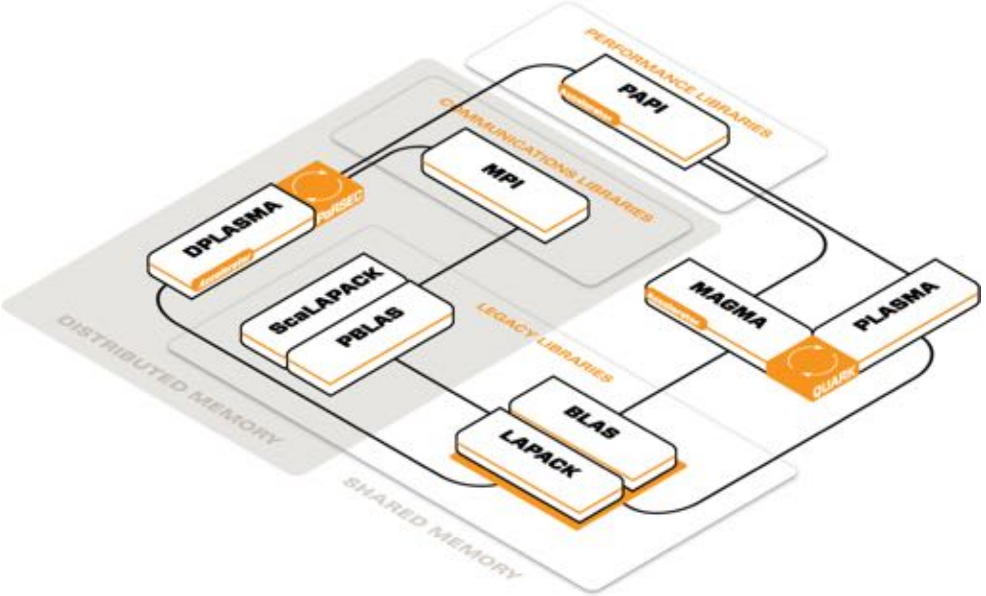
- Optimizations for distributed memory:
  - Controlling Task Insertion Rate
  - DAG Trimming
  - No redundant data transfer
  - Flushing not needed data

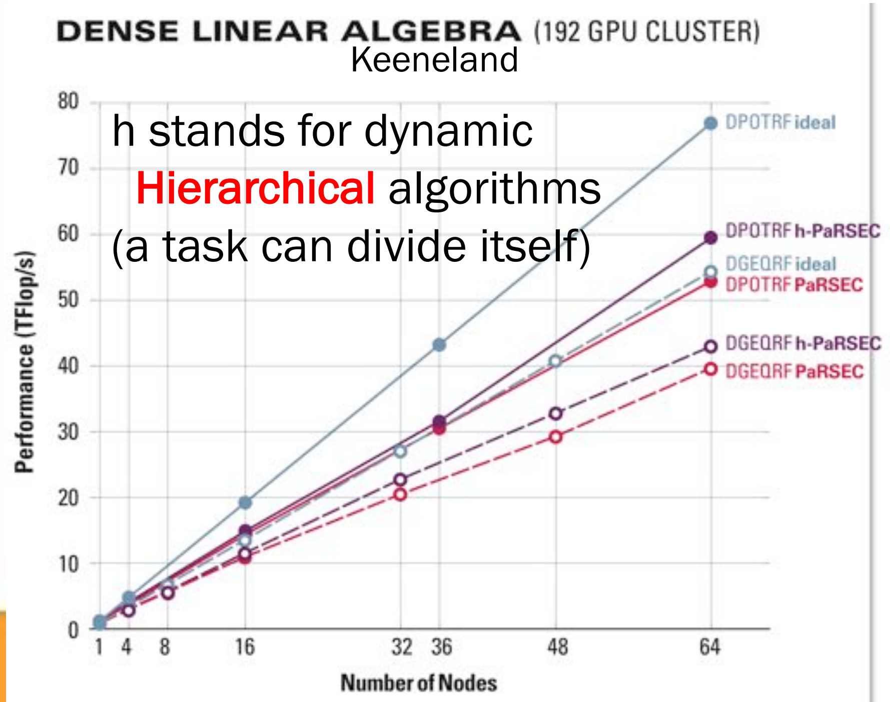
- Experiments
  - Intel Xeon CPU E5-2650 v3 @ 2.30GHz
  - 8 nodes with 20 cores each
  - 64GB RAM, Infiniband FDR 56G
  - Open MPI 2.0.1

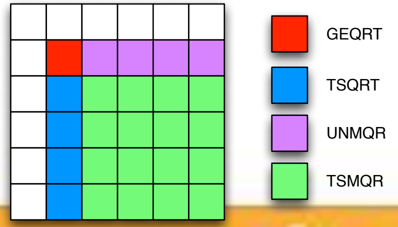


Dense Linear Algebra
SLATE = ScaLAPACK + runtime (PaRSEC)





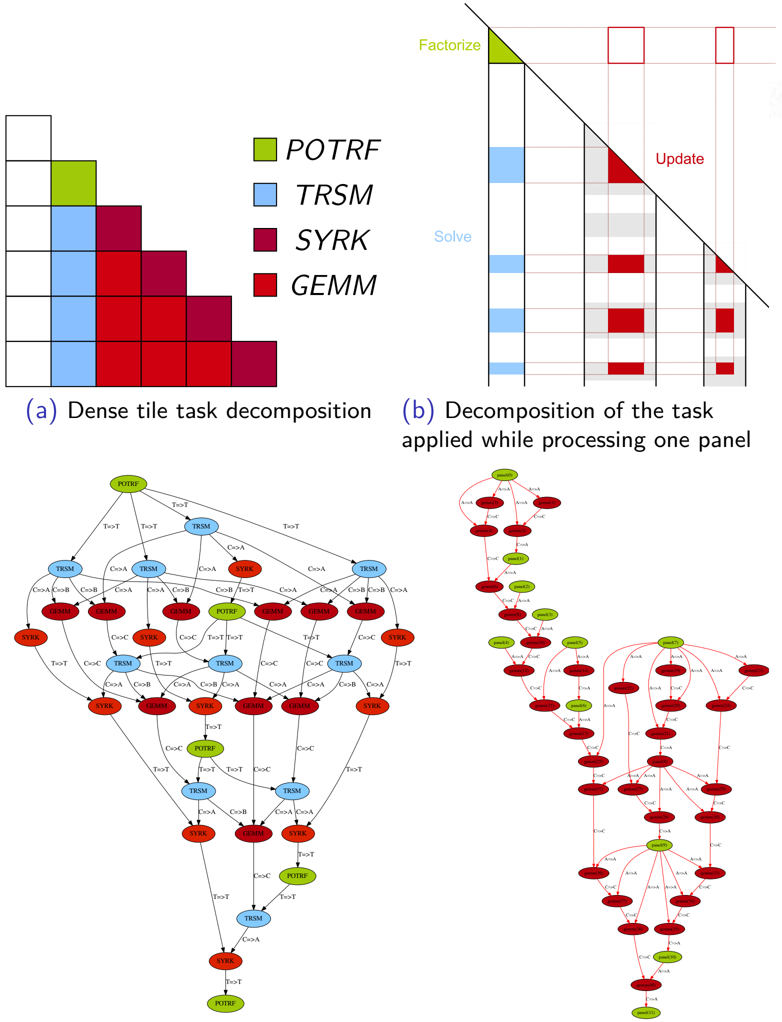




FUNCTIONALITY	COVERAGE
Linear Systems of Equations	Cholesky, LU (inc. pivoting, PP), LDL (prototype)
Least Squares	QR & LQ
Symmetric Eigenvalue Problem	Reduction to Band (prototype)
Level 3 Tile BLAS	GEMM, TRSM, TRMM, HEMM/SYMM, HERK/SYRK, HER2K/SYR2K
Auxiliary Subroutines	Matrix generation (PLRNT, PLGHE/PLGSY, PLTMG), Norm computation (LANGE, LANHE/LANSY, LANTR), Extra functions (LASET, LACPY, LASCAL, GEAD, TRADD, PRINT), Generic Map functions

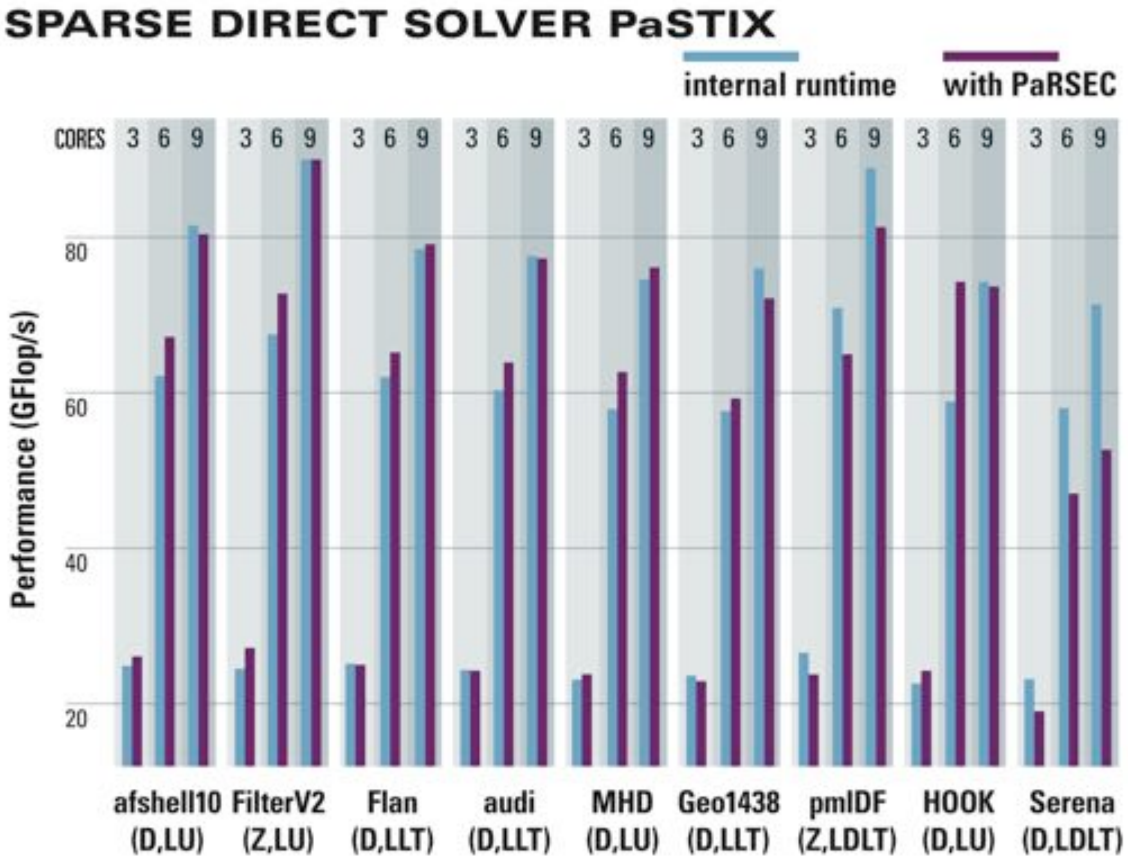
Total, Inria Bordeaux, Inria Pau, LaBRI, ICL

Sparse Linear Algebra



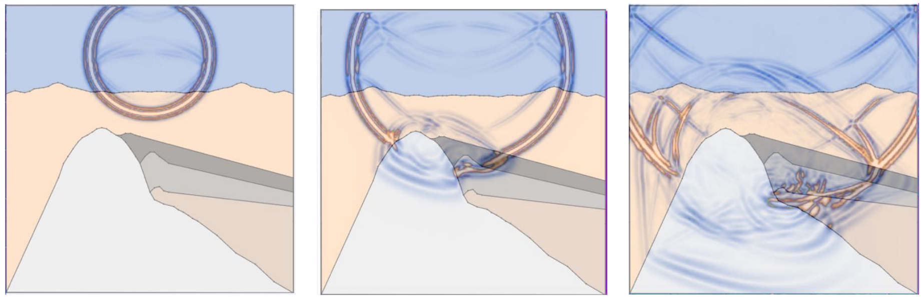
(a) Dense tile task decomposition (b) Decomposition of the task applied while processing one panel

(c) Dense DAG (d) Sparse DAG representation of a sparse LDL^T factorization



DIP: Elastodynamic Wave Propagation

Total, Inria Bordeaux, Inria Pau, ICL

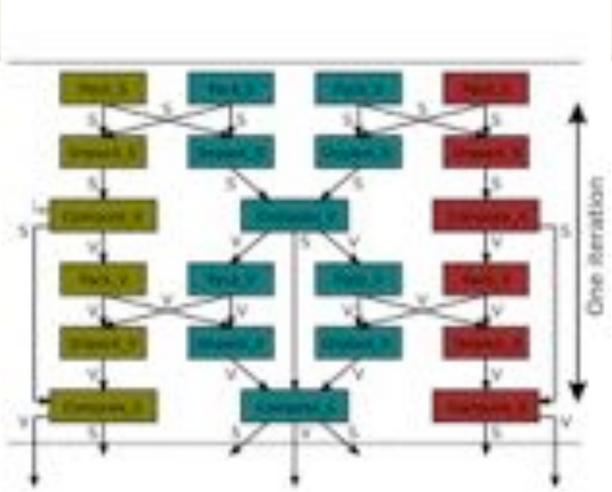


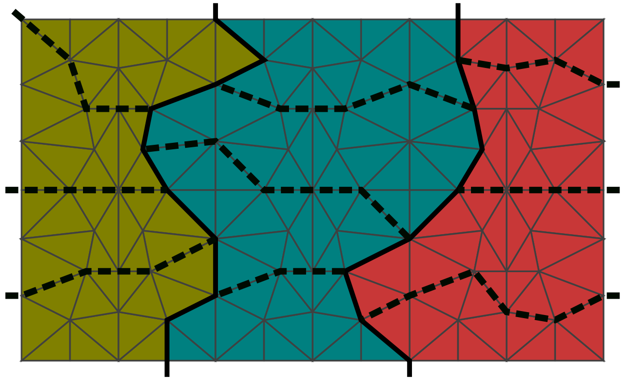
$$\begin{cases} v_h^{n+1} = v_h^n + M_v^{-1}[\Delta t R_{\underline{\sigma}} \underline{\sigma}_h^{n+1/2}] & \textit{UpdateVelocity} \\ \underline{\sigma}_h^{n+3/2} = \underline{\sigma}_h^{n+1/2} + M_{\underline{\sigma}}^{-1}[\Delta t R_v v_h^{n+1}] & \textit{UpdateStress} \end{cases}$$

For $n = 1 : n\_timesteps\_T$
- $\text{Communication}(\sigma_h^{n+1/2})$
- $v_h^{n+1} \leftarrow computeVelocity(v_h^n, \sigma_h^{n+1/2}, \Delta_t)$
- $\text{Communication}(v_h^{n+1})$
- $\sigma_h^{n+3/2} \leftarrow computeStress(\sigma_h^{n+1/2}, v_h^{n+1}, \Delta_t)$

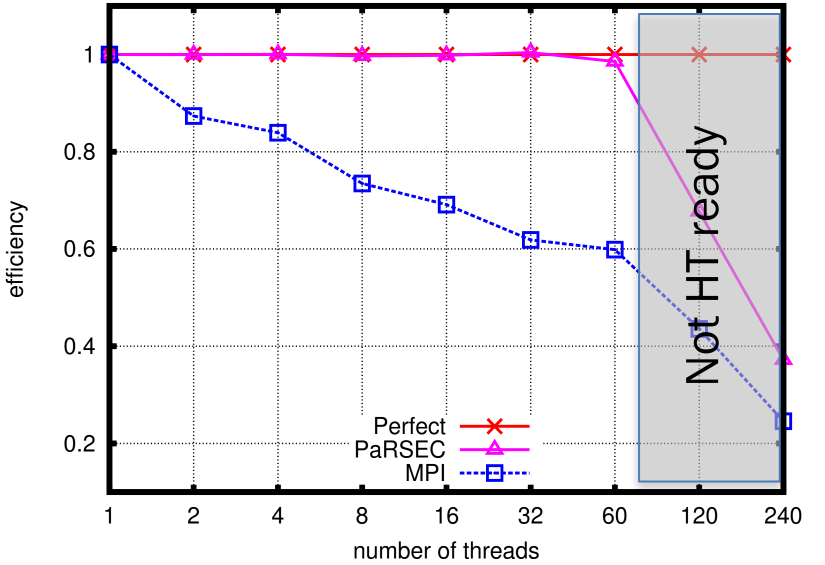
End For t

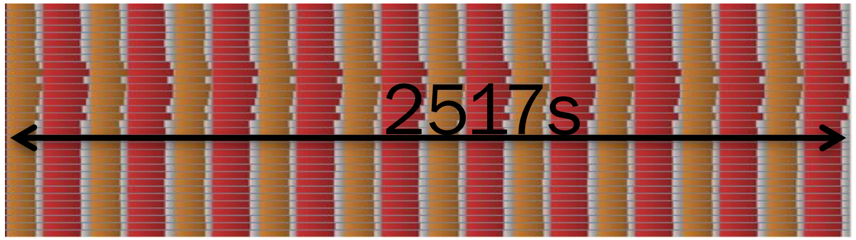
Finer grain partitioning compared with MPI
Increased communications but also increased potential for parallelism
Need for load-balancing

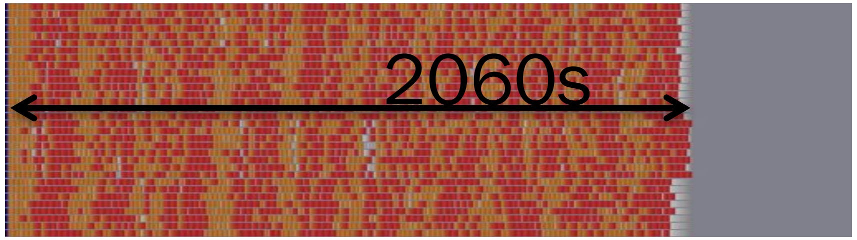




Dynamically redistribute the data
- use PAPI counters to estimate the imbalance
- reshuffle the frontiers to balance the workload

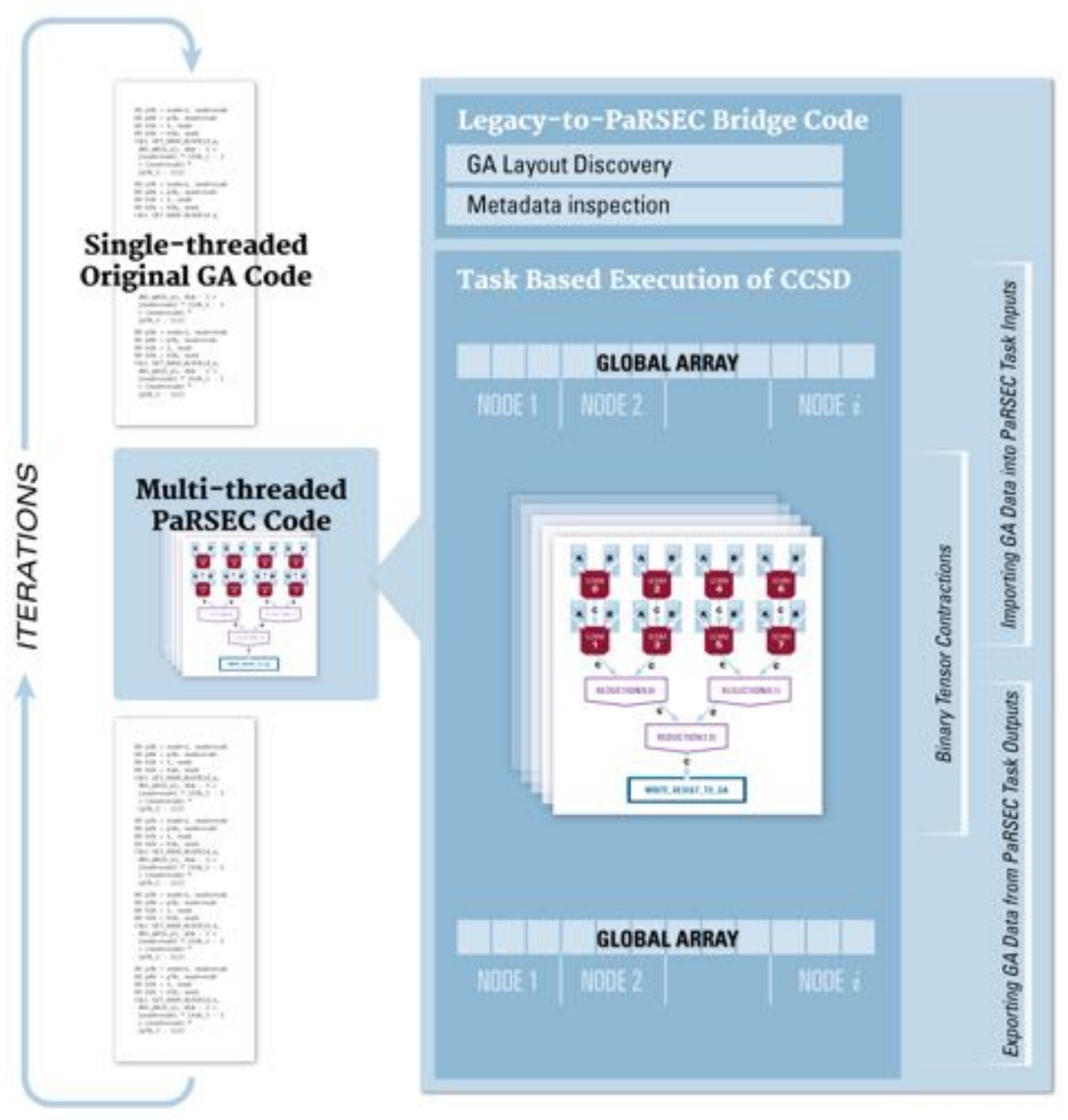






Quantum Chemistry: PaRSEC NWChem Integration

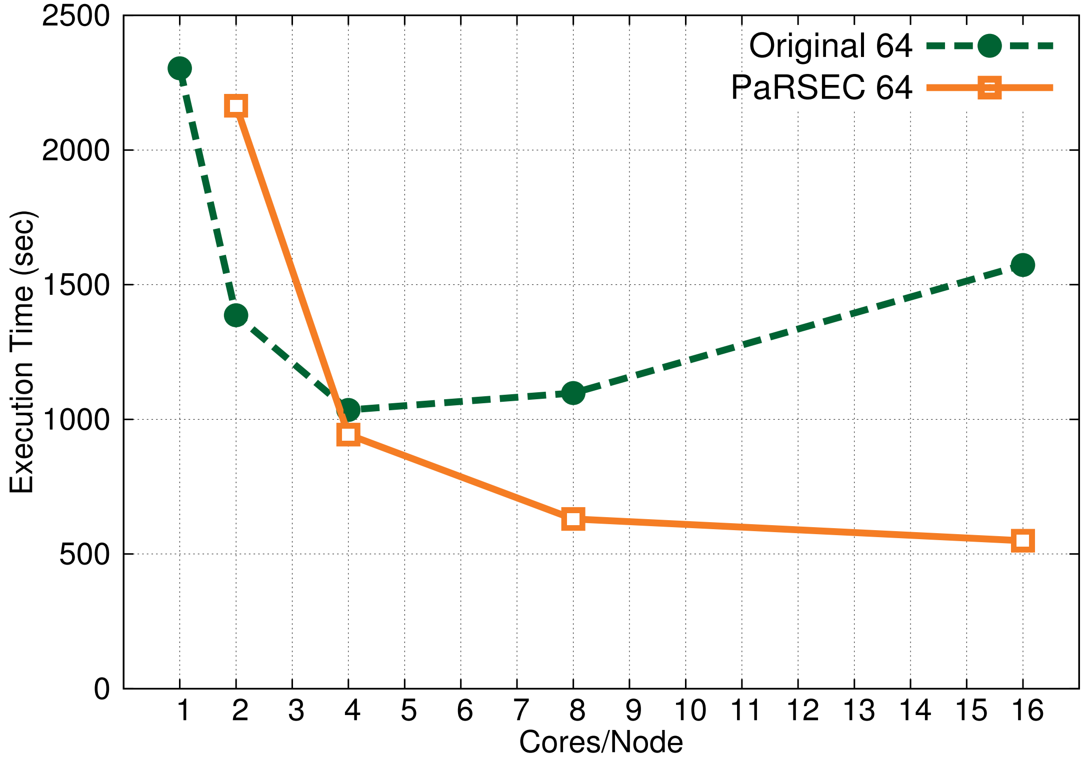
- ”Seamless” integration: NWChem holds kernels above Global Array, we replaced ¾ of them as PaRSEC operations
- Interoperability: In PaRSEC operations, the data is pulled from Global Array locally, then dispatched, computed, and pushed back into the Global Array

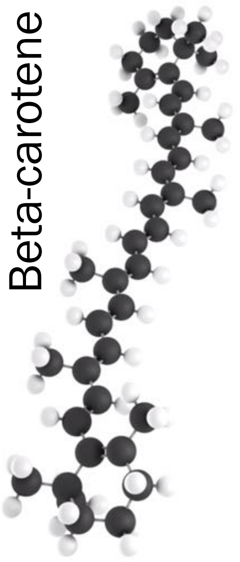


Quantum Chemistry:
PaRSEC NWChem Integration

- ”Seamless” integration: NWChem holds kernels above Global Array, we replaced ¾ of them as PaRSEC operations
- Interoperability: In PaRSEC operations, the data is pulled from Global Array locally, then dispatched, computed, and pushed back into the Global Array

- Better scaling is due to increased parallelism in the PaRSEC representation:
  - Reduction trees instead of chains of operations
  - Parallel independent sort operations
  - Optimized data gather / dispatch
  - Global Array read / write made local, then data transfers are asynchronous and overlapped with computations



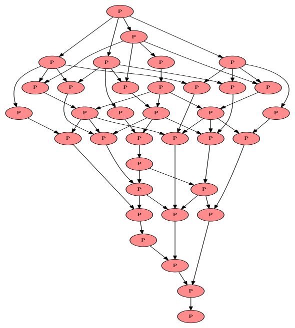


$C_{40}H_{56}$

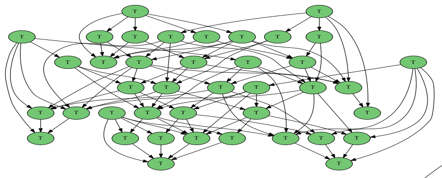
Natural data-dependent DAG Composition

Example POTRI = POTRF + TRTRI + LAUUM

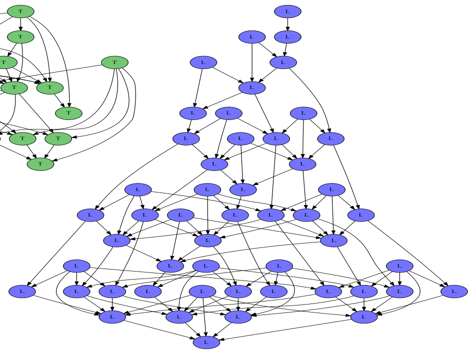
POTRF



TRTRI

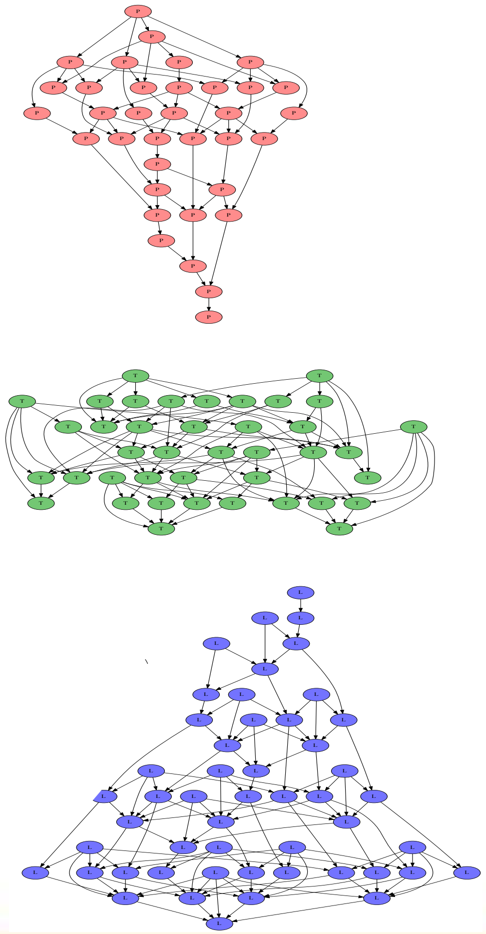


LAUUM

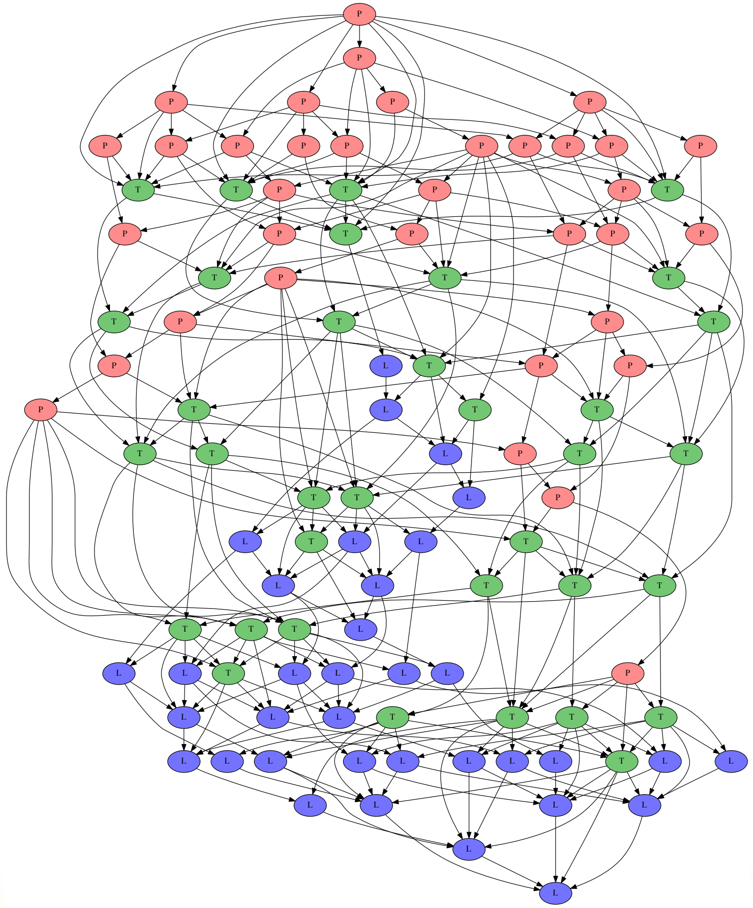


- 3 approaches:
  - **Fork/join:** complete POTRF before starting TRTRI
  - **Compiler-based:** give the three sequential algorithms to the Q2J compiler, and get a single PTG for POINV
  - **Runtime-based:** tell the runtime that after POTRF is done on a tile, TRTRI can start, and let the runtime compose

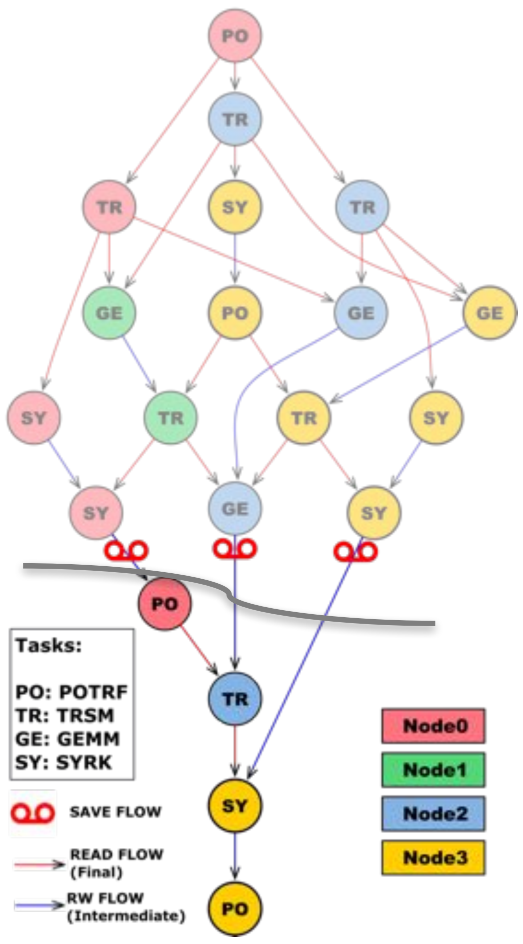
Traditional



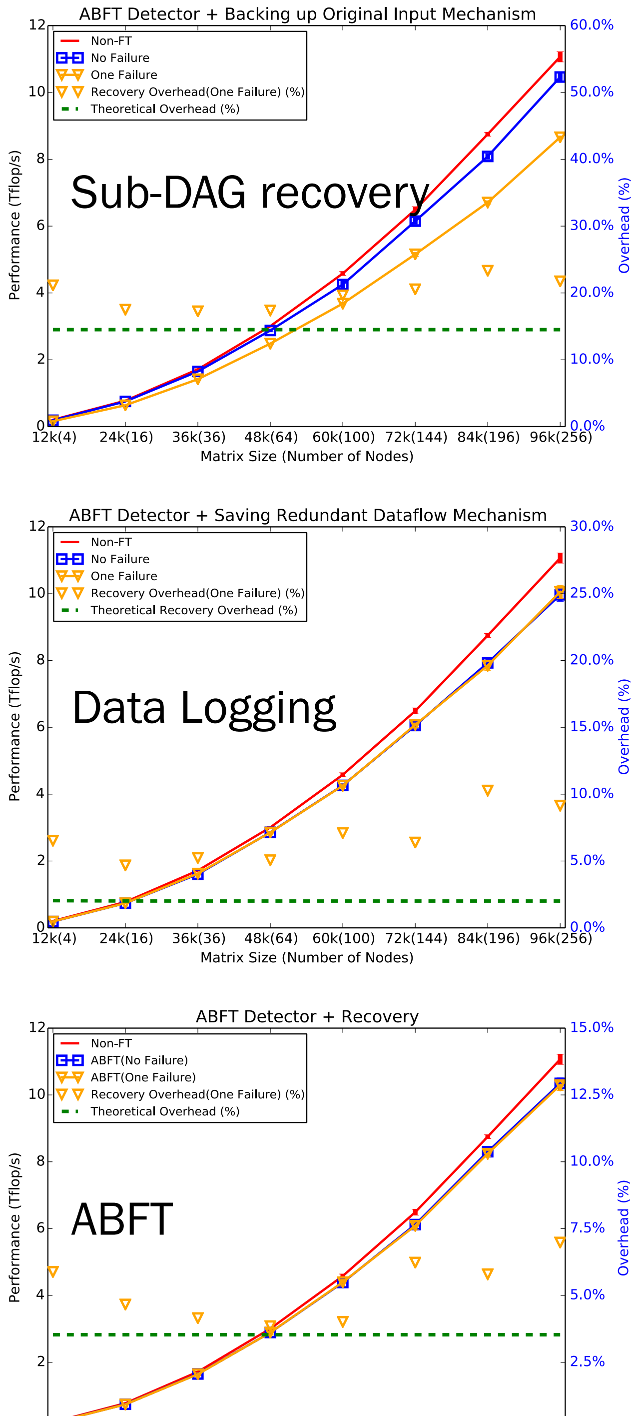
PaRSEC



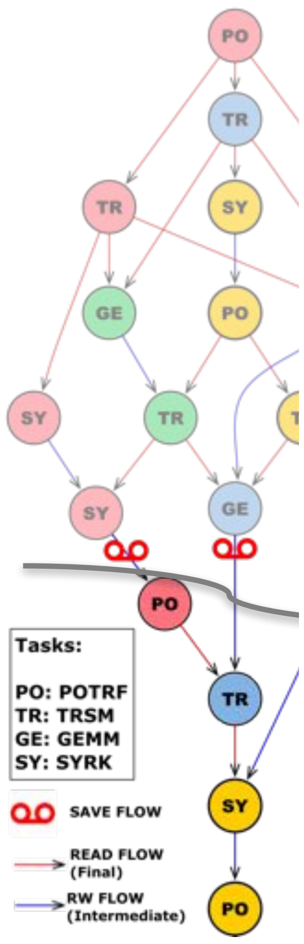
Resilience support from runtime

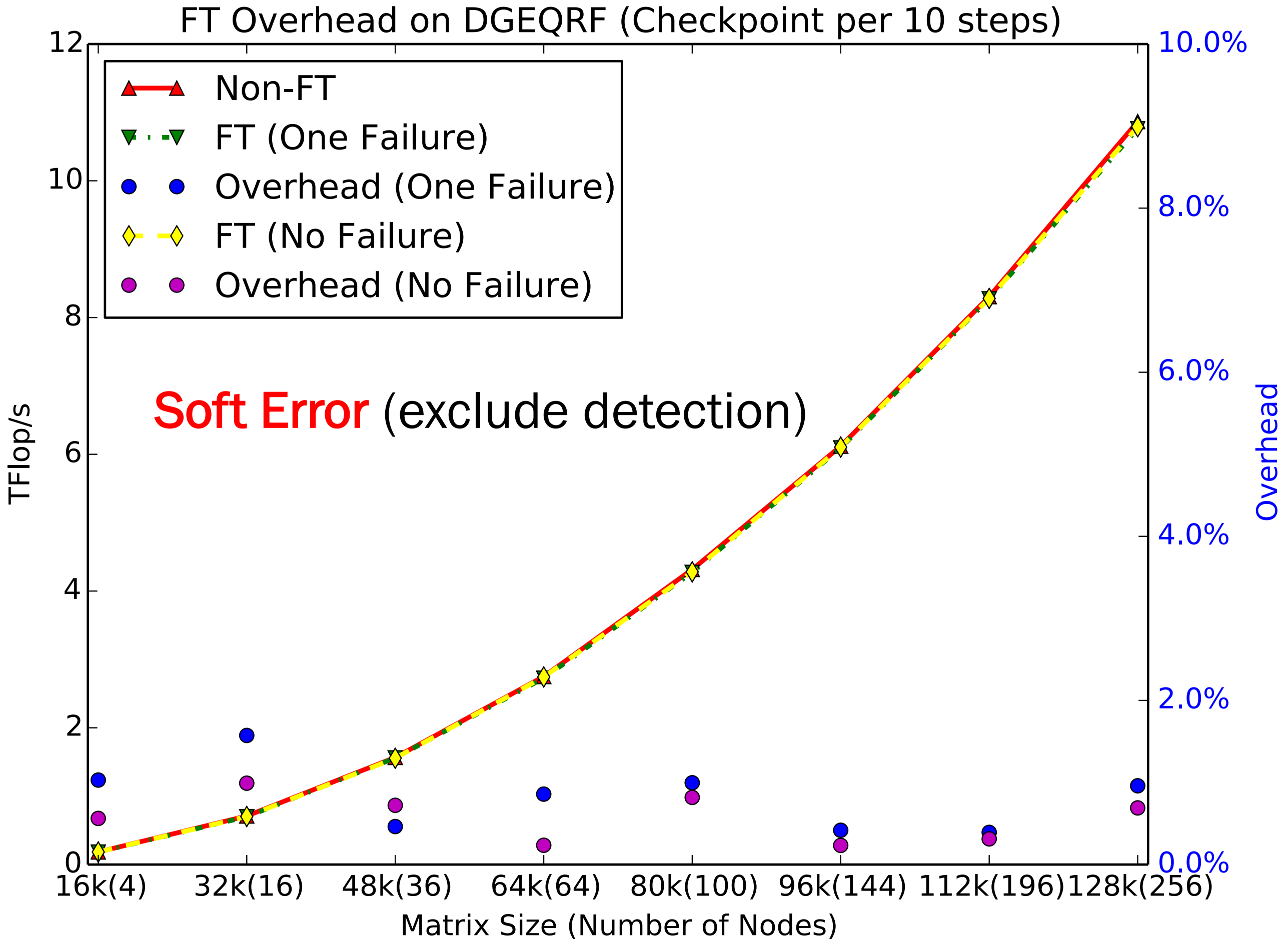


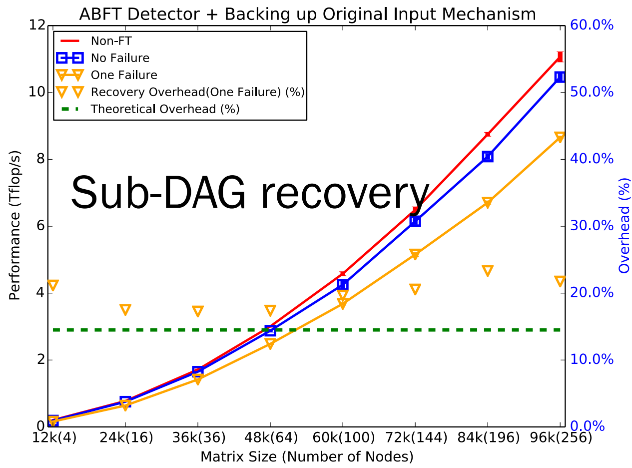
- Recovery based on leaving data safely behind (generic & low-overhead)
  - Partial DAG recovery
- Burst of errors are supported, multiple sub-DAGs will be executed in parallel with the original
- Merge resilient features into runtime:
  - Reserve minimum dataflow for protection
  - Minimize task re-execution
  - Minimize extra memory
- Export interface for user/tool – configurable data logging scheme
- Automatic resilience for non-FT applications over PaRSEC

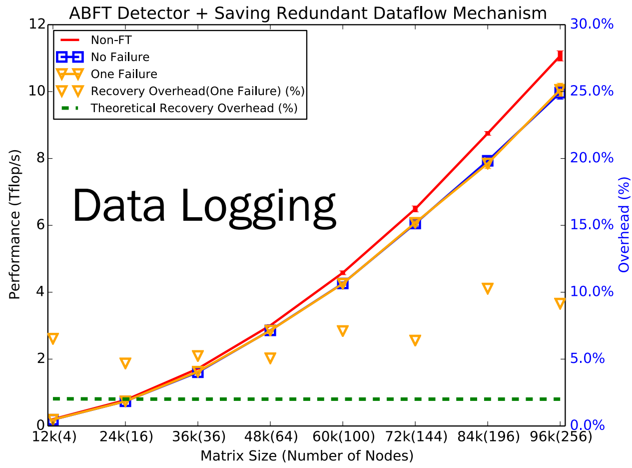


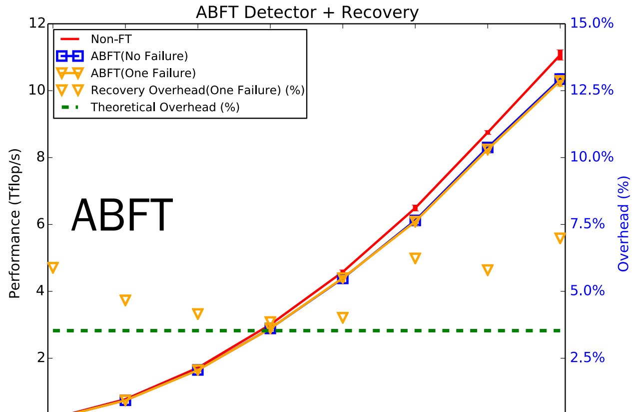
Resilience support from runtime



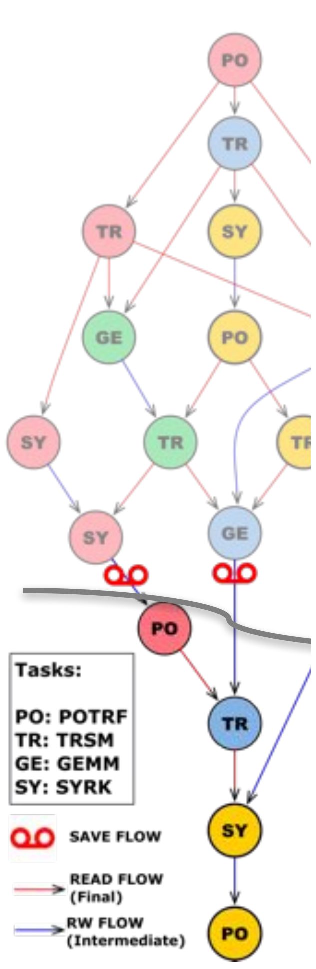


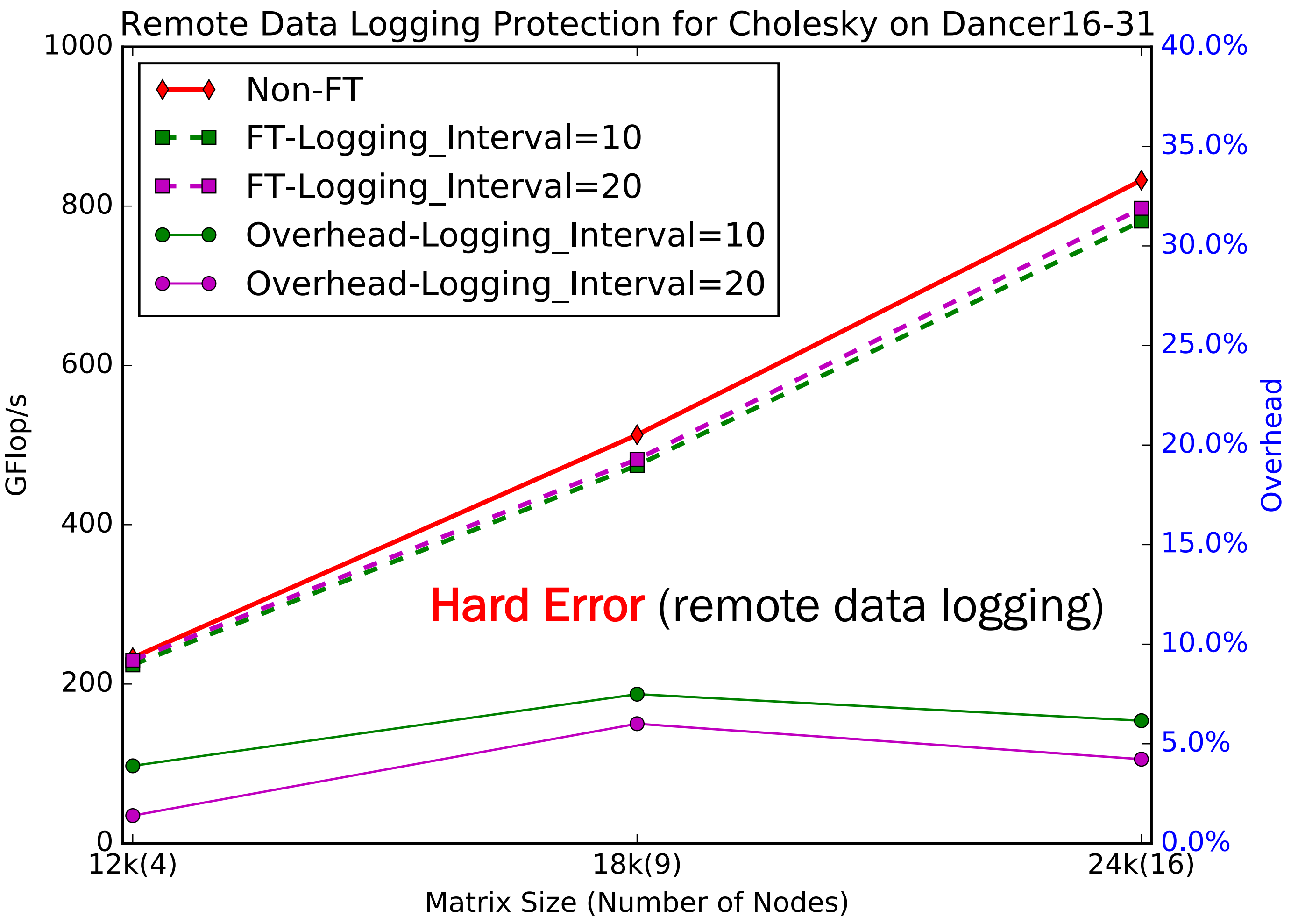


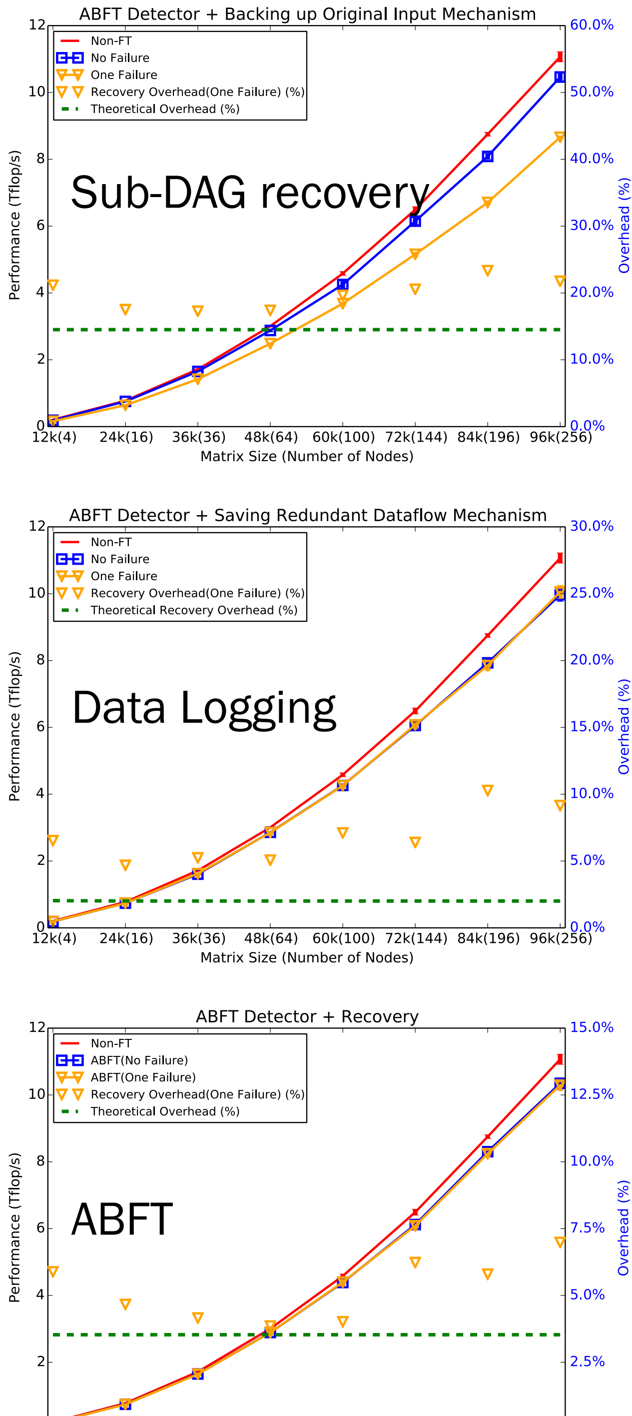




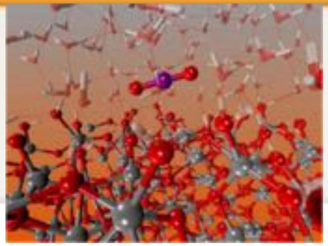
Resilience support from runtime

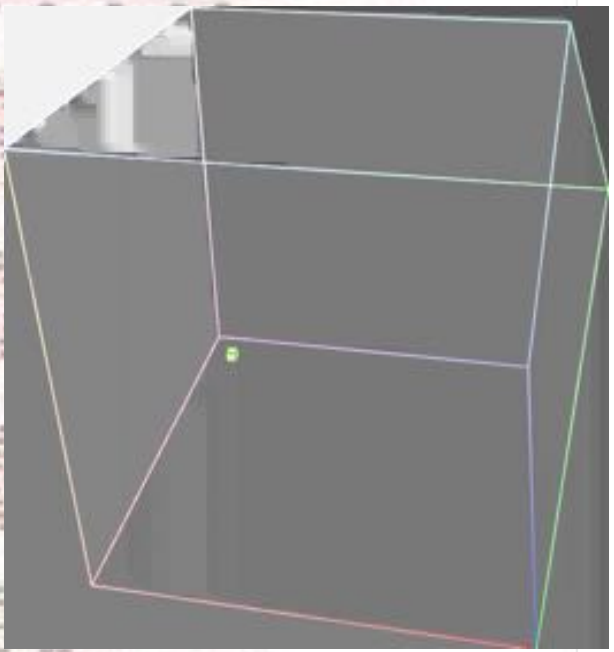




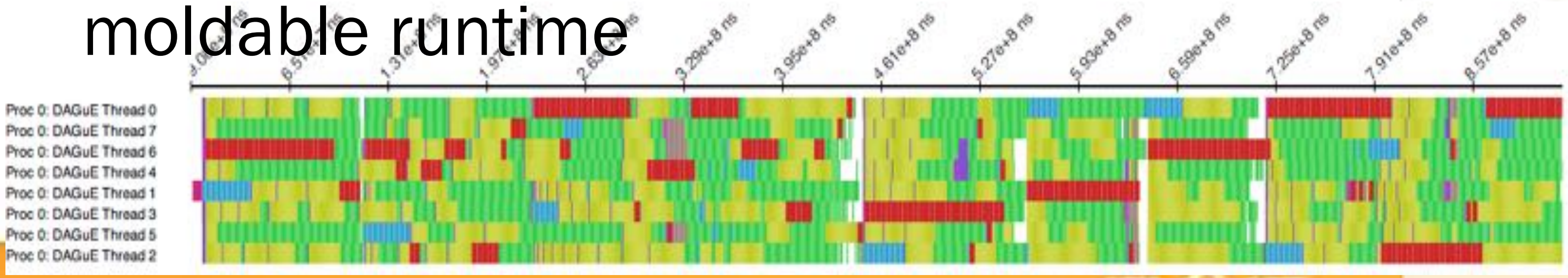


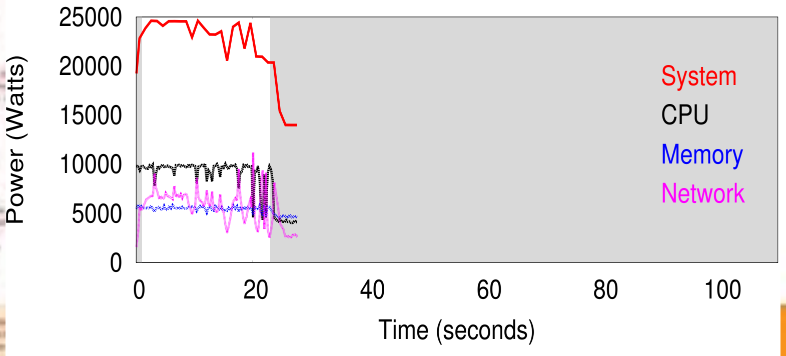
The PaRSEC ecosystem





- Support for many different types of applications
  - Dense Linear Algebra: DPLASMA, MORSE/Chameleon
  - Sparse Linear Algebra: PaSTIX
  - Geophysics: Total - Elastodynamic Wave Propagation
  - Chemistry: NWChem Coupled Cluster, MADNESS, TiledArray
  - *: ScaLAPACK, MORSE/Chameleon, SLATE
- A set of tools to understand performance, profile and debug
- A resilient distributed heterogeneous moldable runtime





(b) DPLASMA.

Conclusions

- Programming can be made easy(ier)
  - Portability: inherently take advantage of all hardware capabilities
  - Efficiency: deliver the best performance on several families of algorithms
  - Domain Specific Languages to facilitate development
  - Interoperability: data is the centric piece
- Build a scientific enabler allowing different communities to focus on different problems
  - Application developers on their algorithms
  - Language specialists on Domain Specific Languages
  - System developers on system issues
  - Compilers on optimizing the task code
- Interact with hardware designers to improve support for runtime needs
  - HiHAT: A New Way Forward
    for Hierarchical Heterogeneous Asynchronous Tasking

Distributed Database: TileDB & PaRSEC

- TileDB: Distributed database for LAQL (Linear Algebra Query Language)

```
SELECT QR(A.values) FROM A WHERE d(A.coord,
0.0) < 10.0;
```

- Existing Implementation: ScaLAPACK interface
  - External program runs ScaLAPACK
  - Data is redistributed and moved to the program using phase-out; compute; phase-in approach
- Integration with PaRSEC: driver in a separate process pulls data from the database
  - Locally
  - Asynchronously
  - Building a pipeline of data in and out