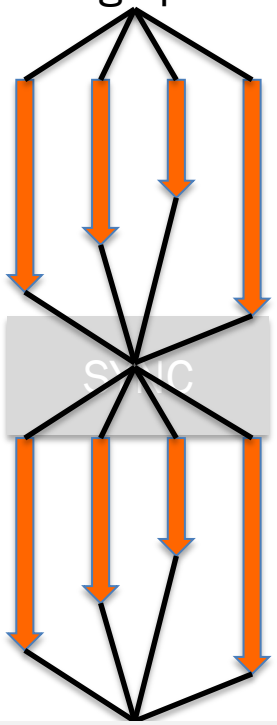
A heterogeneous distributed runtimes for fine-granularity tasks: dataflow as an alternative programming paradigm

George Bosilca & many others @ ICL, @ Riken, @ KAUST, @INRIA

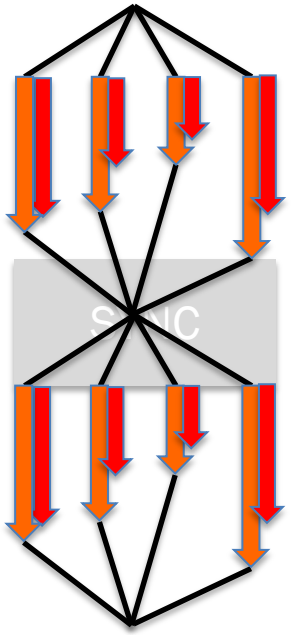
WAMTA 2023
Baton Rouge, LA

A [very short] history of computing paradigms

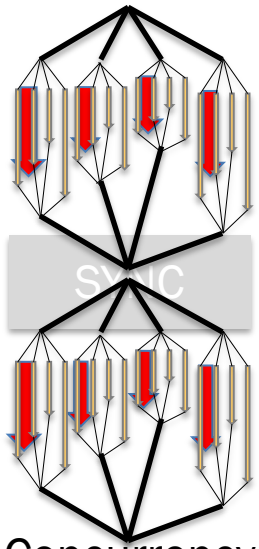
BSP & early message passing



MPI + X



MPI + X + Y + Z + …



Concurrency*
Heterogeneity
Resiliency

Over-subscription:

- User level threads (Qthreads, MassiveThreads, Nanos++, Argobots)

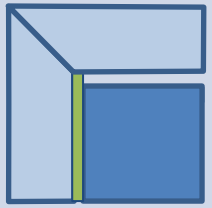
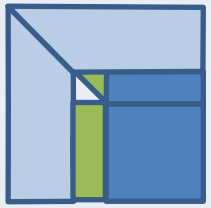
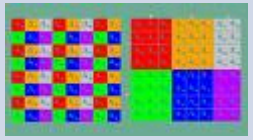
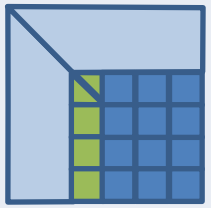
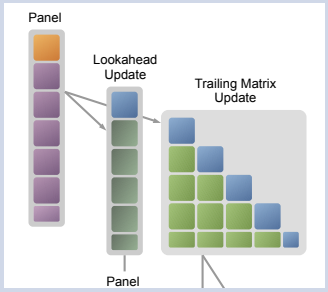
Task-ification:

- Shared memory: OpenMP, Tascel, Quark, TBB*, PPL, Kokkos…
- Distributed Memory: StarPU, StarSS*, Legion, CnC, HPX, Dagger, X10, Hihat, …

* explicit communications

- Difficult to express the potential inter-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 about developer's productivity ?

Software/Algorithms follow hardware evolution in time

LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations
ScaLAPACK (90's) (Distributed Memory)		Rely on - PBLAS Mess Passing
PLASMA New Algorithms (many-core friendly) DPLASMA (Distributed Memory)		Rely on - block data layout, a DAG/scheduler, some extra kernels, hybrid scheduler, hybrid kernels
SLATE (decrease the number of dependencies by increasing the granularity of tasks)		Rely on - a DAG scheduler, an efficient support for batched operations, a fork-join execution model

Understanding the field

- No standard in concepts of Task-based Runtime systems
- No classification based on critical features
- Difficult to compare various Runtimes supporting distributed memory systems on similar problems

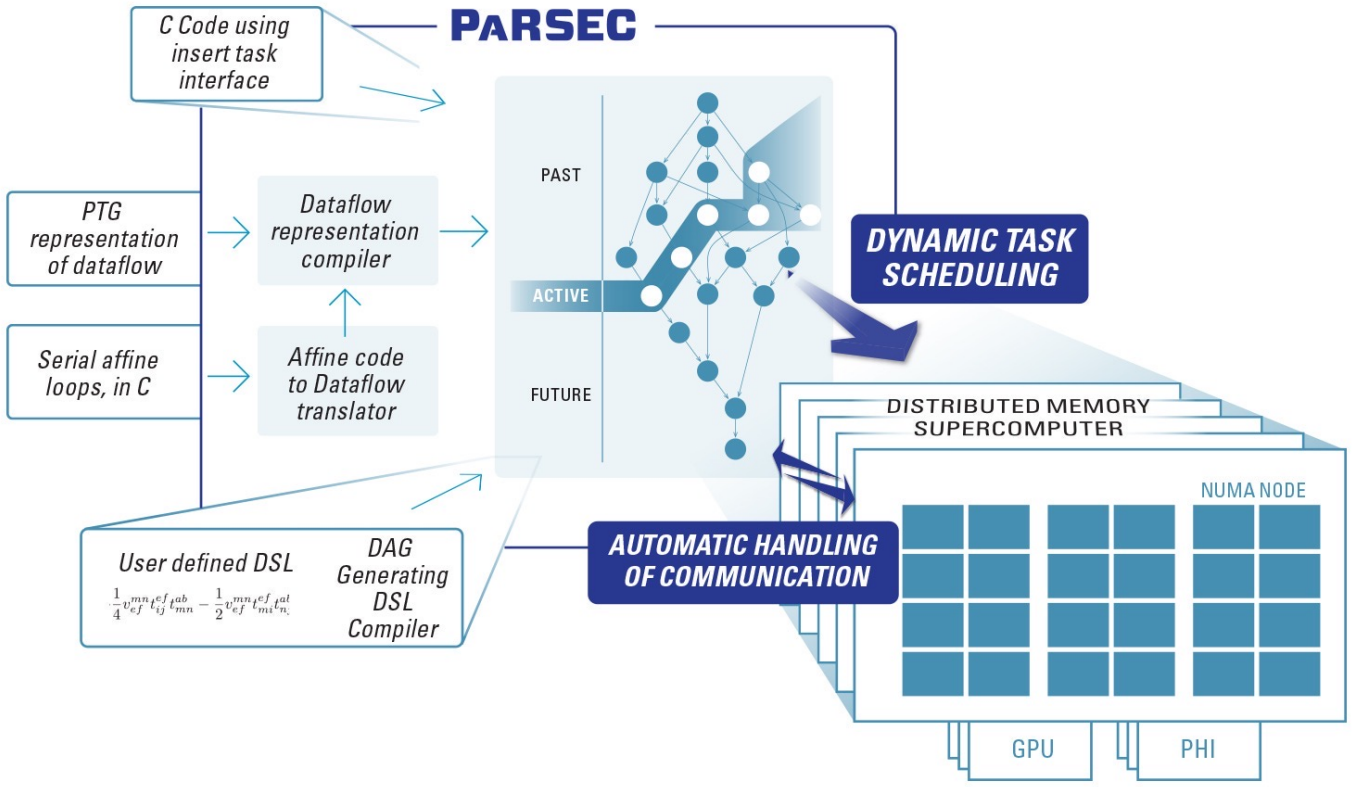
Table 1: Summary of the features and characteristics of each Runtime

Runtime	Runtime Level		Task-Graph		Load Balancer		Fault-Tolerance		Profiling Tools	Modular	Heterogeneous
	Low	High	Static	Dynamic	Intra-node	Inter-node	Soft	Hard			
parsec	✓	✓	✓	✓	✓	×	✓	×	✓	✓	✓
Legion	✓	✓	×	✓	User	User	×	×	✓	✓	✓
StarPU	✓	✓	×	✓	✓	×	×	×	✓	×	✓
HPX	✓	×	×	✓	✓	×	×	×	✓	✓	×
Charm++	✓	×	×	✓	✓	✓	✓	✓	✓	✓	✓
Uintah	×	✓	×	✓	✓	✓	×	×	✓	✓	✓
STAPL	✓	✓	✓	✓	✓	×	×	×	✓	✓	×

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 non-essential concept

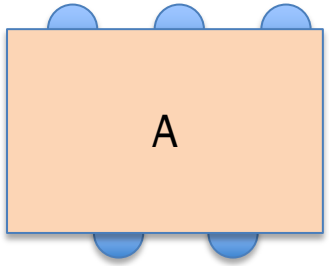


Runtime

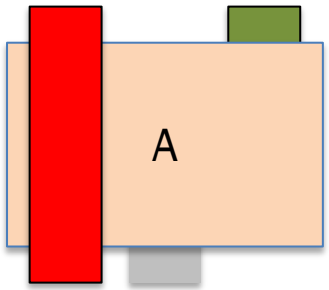
- Portability layer for heterogeneous architectures
- Scheduling policies adapt every execution to the hardware & ongoing system status
- Data movements between producers and consumers are inferred from dependencies. Communications/computations overlap naturally unfold
- Coherency protocols minimize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

PaRSEC concepts: Tasks / Collections / Domains

- A task is somewhat a familiar concept, a pure function with a well-defined number of terminals (input and outputs)
  - Except that is meant to be reused, so the basic concept is a task-class

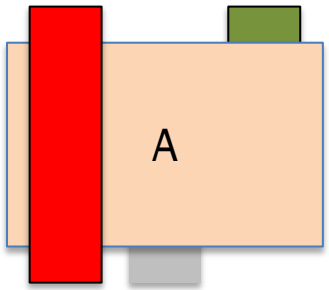


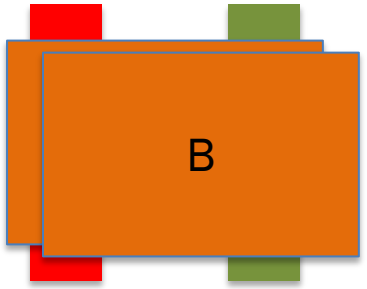
PaRSEC concepts: Tasks / Collections / Domains

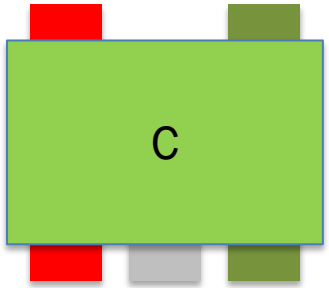


- The task-class is somewhat a familiar concept, a pure function with a well-defined number of terminals (input and outputs)
  - Terminals are named, and they cross the task boundaries
  - Terminals are tagged with properties R/RW/W/T
  - Depending on the DSL the outputs might be made available at any time

PaRSEC concepts: Tasks / Collections / Domains

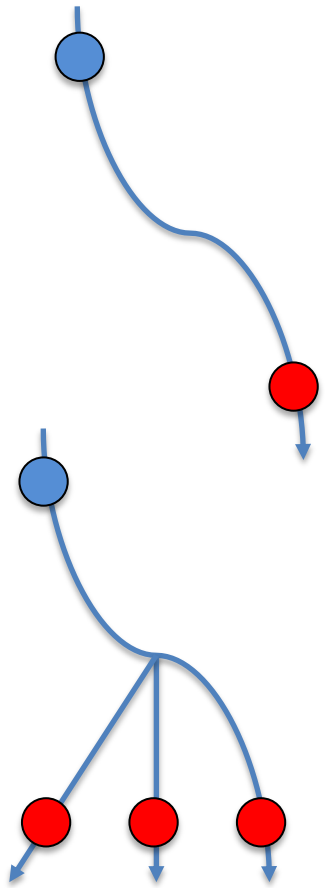






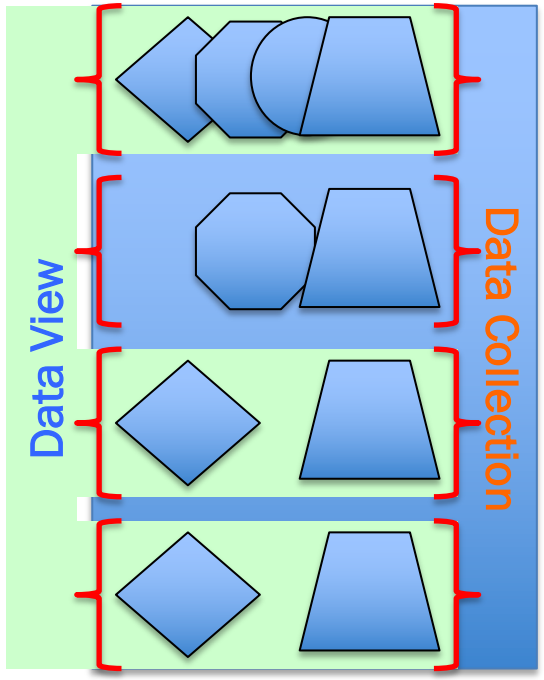
- The task-class is somewhat a familiar concept, a pure function with a well-defined number of terminals (input and outputs)
  - Terminals are named, and they cross the task boundaries
  - Terminals are tagged with properties R/RW/W/T
  - Depending on the DSL the outputs might be made available at any time
- Tasks-classes can be extended with multiple incarnations/backends (CPU, GPU, recursive, OpenCL, JIT, …)
  - The execution backend is dynamically selected at runtime
  - Specialized terminals shall exist (pull, push, reduce, control, gather, scatter)
- A **task** become a particular instantiation of a task-class (aka. named)
  - PaRSEC was designed for tasks with single-digit μsec granularity
- A collection of tasks and their dependencies is both a taskpool (DSL generate taskpools) and a form of task (it has input and output, possibly distributed)

PaRSEC concepts: Tasks / Collections / Domains

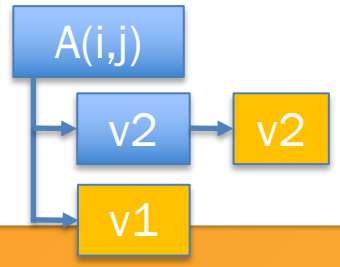


- Tasks-classes terminals are connected by Edges
  - They have start and end transformers that can reshape the data
    - Logically equivalent data with different shapes (think of it as an MPI_Datatype)
  - They are movers of the data from the generator task to its successors (feedback from the task-class and the scheduler on where the edge should deliver the data)
- Specialized edges and transformers: IO, redistribute, compress/reduce, low-rank, push/pull, validate
- A one-to-one edge is a corner-case of a one-to-many edge, PaRSEC's default edge is one-to-many
  - They can work as push or pull edges:
    - Push: make the data available before triggering their associated terminals
    - Pull: only transfer data once the associated terminal was triggered (behave as a dynamic distributed promise)

PaRSEC concepts: Tasks / Collections / Domains

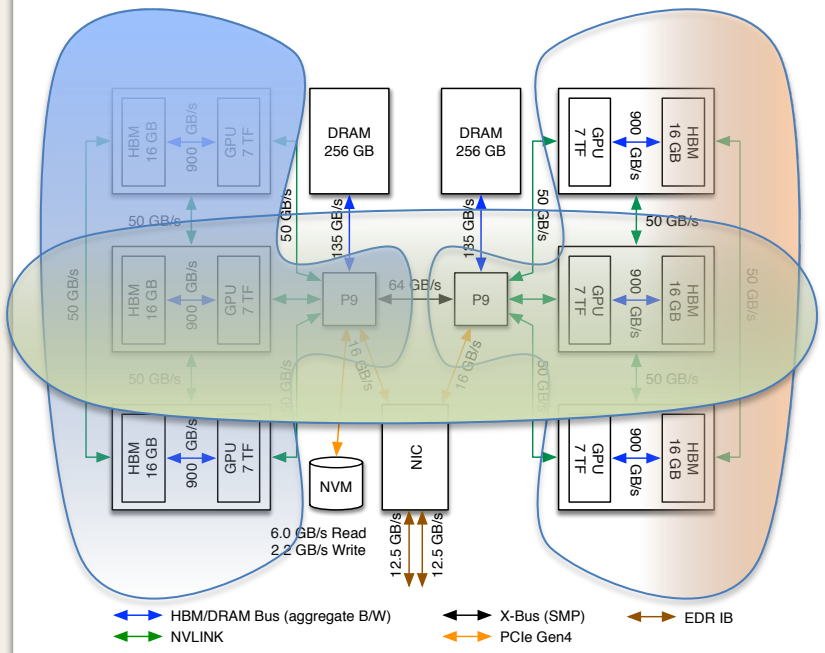


- A data is a manipulation token, the basic logical element used in the description of the dataflow
  - Locations: have multiple coherent copies (remote node, device, checkpoint)
  - Shape: can have different memory layout
  - Visibility: only accessible via the most current version of the data
  - State: 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
  - Can have associated memory or be virtual (mapper or dynamically populated)
- Data View a subset of the data collection used in a particular algorithm (aka. submatrix, row, column,...)



- Everything related to data behaves as a promise: a data (version) is a promise, a data collection is a promise, a data view is a promise, and will be fulfilled by a task or a taskpool
- The promise will be delivered where it is expected by the task that will use it (distributed, GPU task on GPU, ...)

PaRSEC concepts: Tasks / Collections / Domains



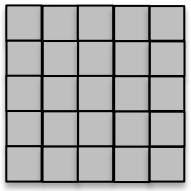
- A PaRSEC domain is a distributed executor extended with a set of resources (core(s), accelerators, networks), memory allocators, and schedulers
  - Multiple executors could exist simultaneously, but the runtime does not police their use of resources
  - Domains execute taskpools without control dependencies (termination detection provided by the runtime)

PaRSEC

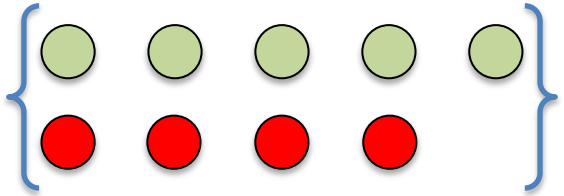
= a data centric programming environment based on asynchronous tasks executing on a heterogeneous distributed environment

- 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)

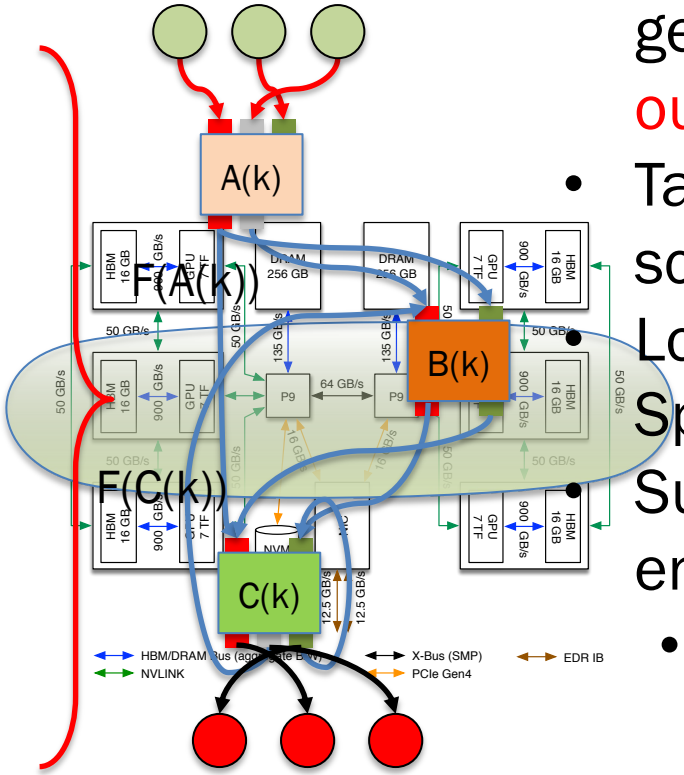
Tasks Classes



User data: dense matrix, sparse, structured or unstructured



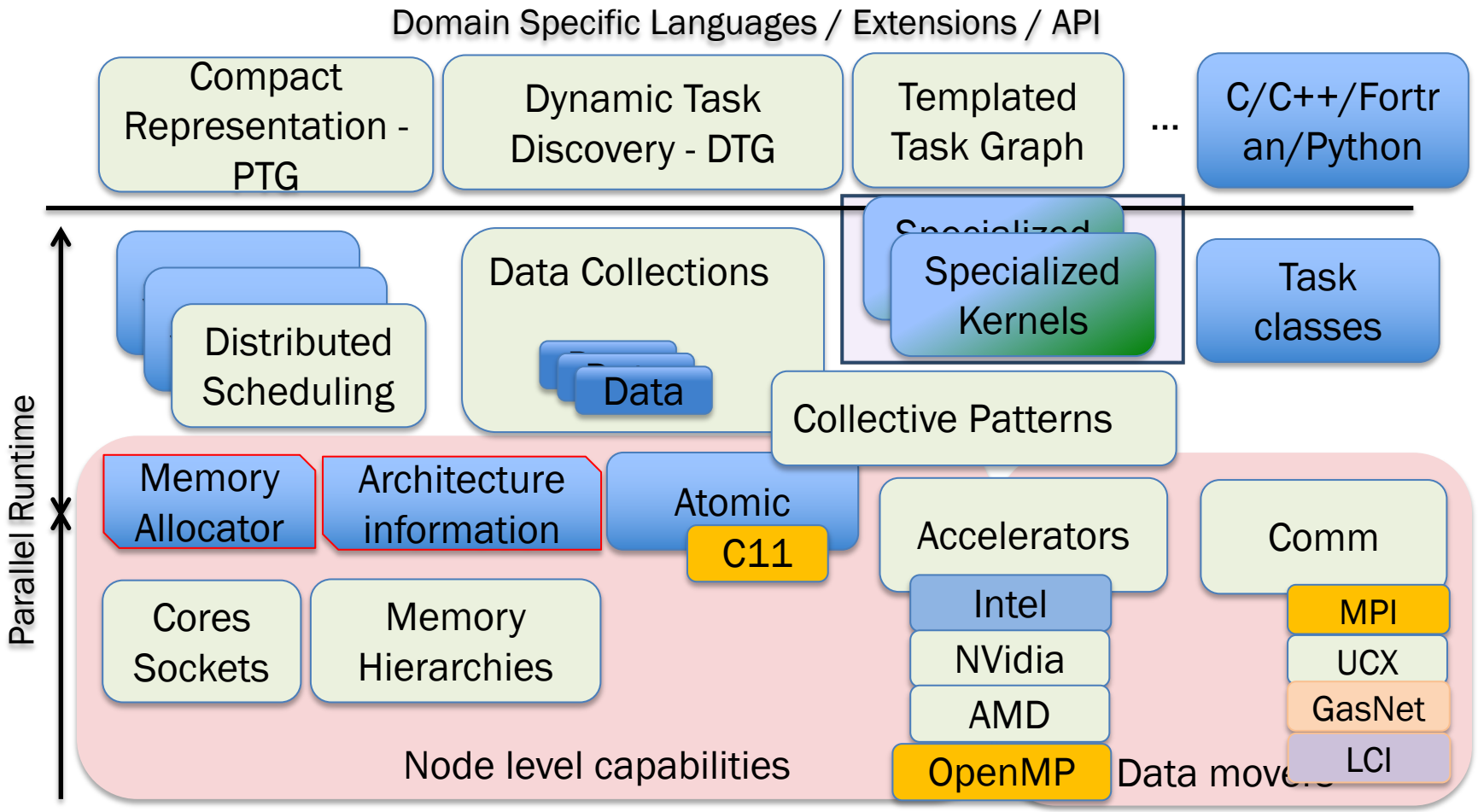
Data collections



Graph of tasks

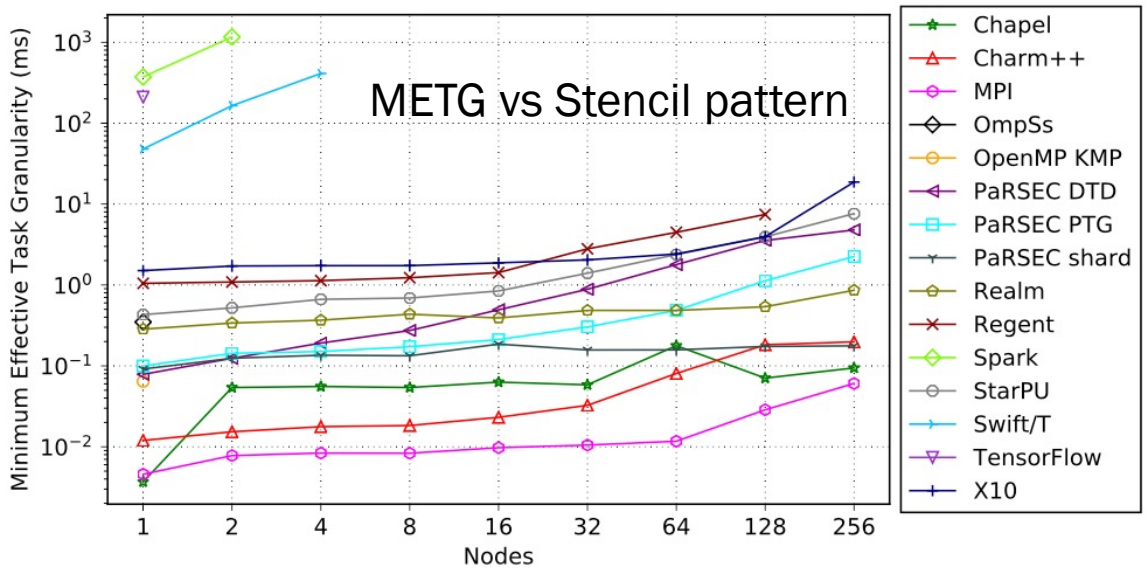
PaRSEC software stack

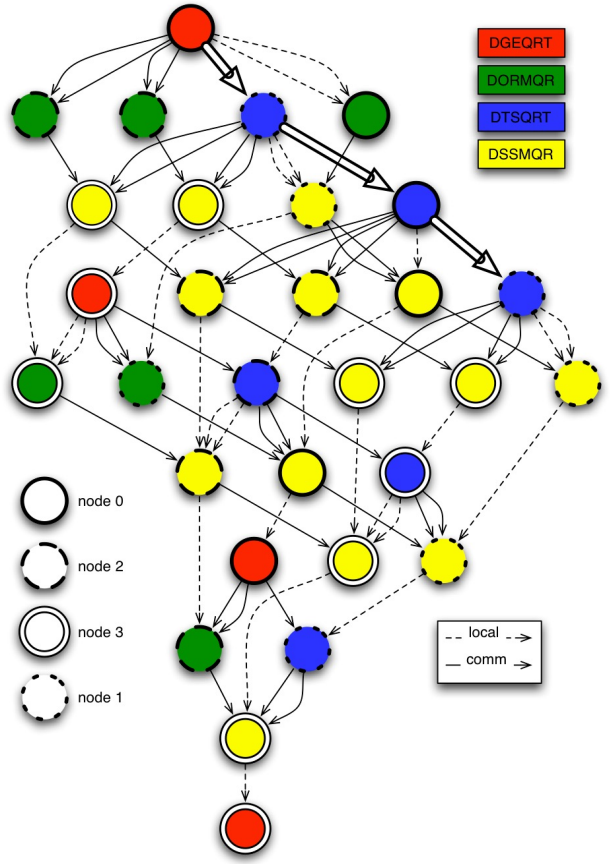
- Most of the software stack (except red parts) is platform independent
- Adding a new architecture or adding support for new hardware is mostly a work at the runtime level
  - With the exception of the specialized kernels to be executed
- Most of the low-level capabilities can be provided by portable libraries
  - MPI, OpenMP/OpenACC
  - Possible impact on performance and scalability



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, …





Minimum Effective Task Granularity (METG)

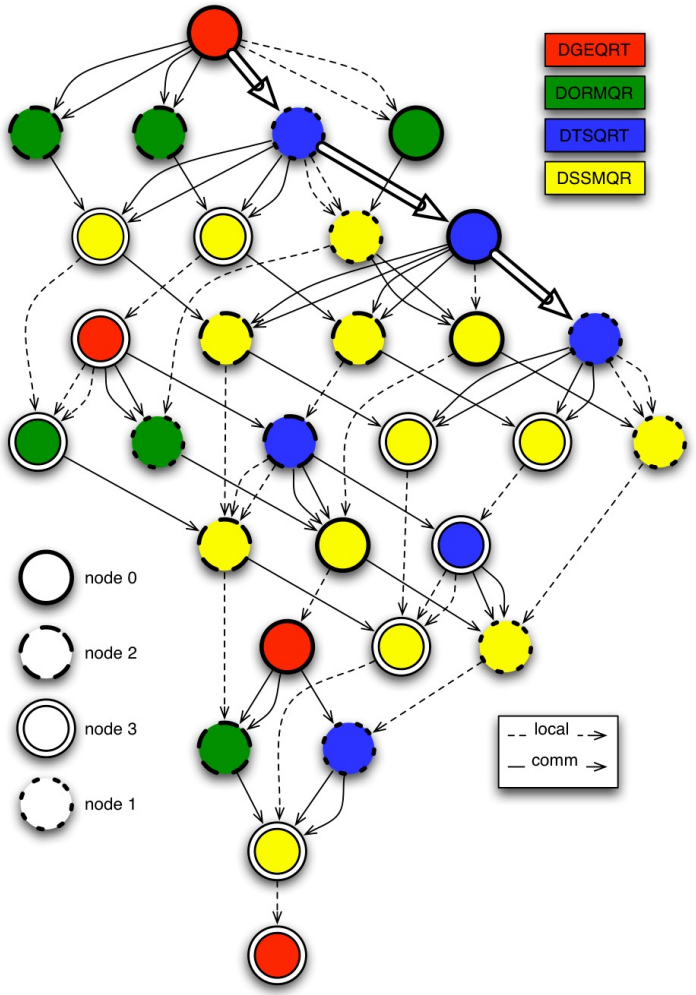
METG(50%) is the smallest task granularity that maintains at least 50% efficiency, meaning that the application achieves at least 50% of its highest performance

METG therefore has a direct relationship with the smallest problem size that can be weak scaled to a given node count with a given level of efficiency

How to describe a graph of tasks ?

From the PaRSEC runtime perspective

- The runtime is agnostic to the domain specific language (DSL)
- Different DSL interoperate through the data collections
- The DSL share the PaRSEC infrastructure
  - Distributed schedulers
  - Communication engine
  - Hardware resources
  - Data management (coherence, versioning, …)
  - Termination detection
- They don’t share
  - The task structure
  - The internal dataflow depiction
  - Some transformers are DSL specific



Dynamic Task Discovery (DTD)

aka. insert_task

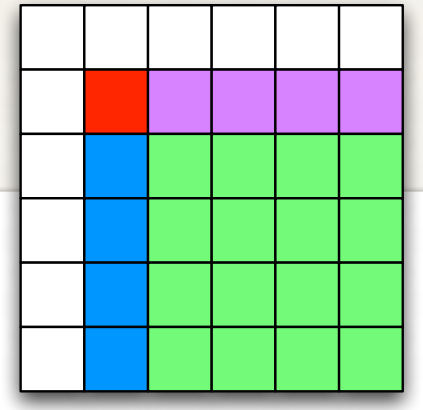
```
int task_hello_with_arg( parsec_execution_stream_t *es,
                         parsec_task_t *this_task )

{
  int *i;
  parsec_dtd_unpack_args( this_task , UNPACK_VALUE , &i);
  printf("Hello World my index is: %d\n" , *i);
  return PARSEC_HOOK_RETURN_DONE ;
}
int discover_tasks()
{
  for(int i = 0; i < 10; i++) {
    parsec_dtd_taskpool_insert_task( dtd_tp, task_hello_with_arg,
                                     0,  "hello_world_task",
                                     sizeof( int), &i, VALUE,
                                     0);  /* No more arguments */
  }
}
```

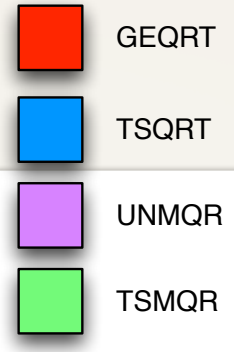
- Possible for each process to only discover local tasks, but data consistency **must** be maintained globally
- Data versioning and caching become a requirement
- Difficult to identify **collective patterns**
- Selecting the **window size** is difficult, all data movement must be known globally (and their order is critically important)

- Dynamic Task Discovery (DTD) enables simple DAG expression through sequential task discovery
- PaRSEC DTD engine builds the DAG of tasks, based on the dependencies of the data flow
- The semantics of sequential execution (the algorithm critical path) are enforced while keeping a DAG with maximal parallelism
- For distributed execution, all computing elements need to discover the same DAG, impairing the runtime scalability
- Only local tasks are kept, and a reference to last accessors / writers on given data to track remote dependencies
- The internal data structure representing the DAG is problem-size dependent, and task discovery window dependent

DTD: insert_task

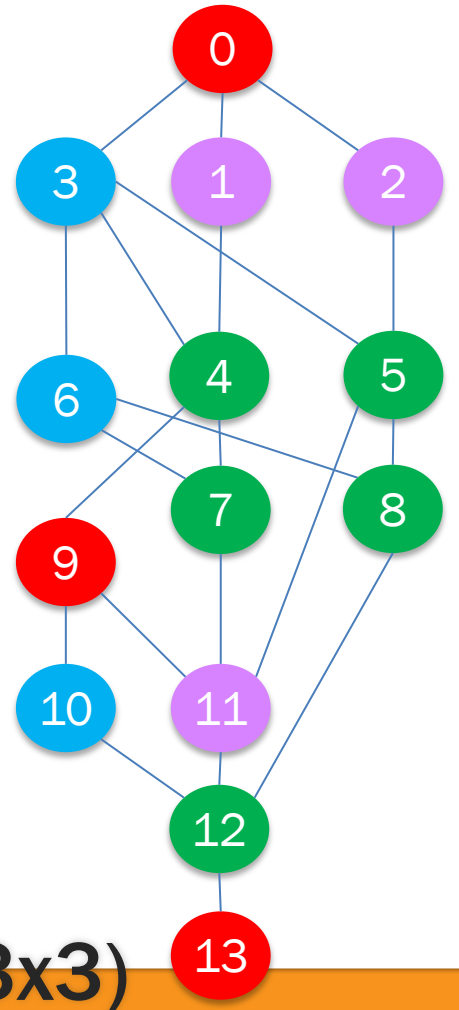
```
for( k = 0; k < SIZE; k++ ) {
    parsec_insert_task( "GEQRT",
                        DATA_OF(A, k, k), INOUT|AFFINITY,
                        DATA_OF(T, k, k), OUTPUT|TILE_RECT)

    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)

Challenge

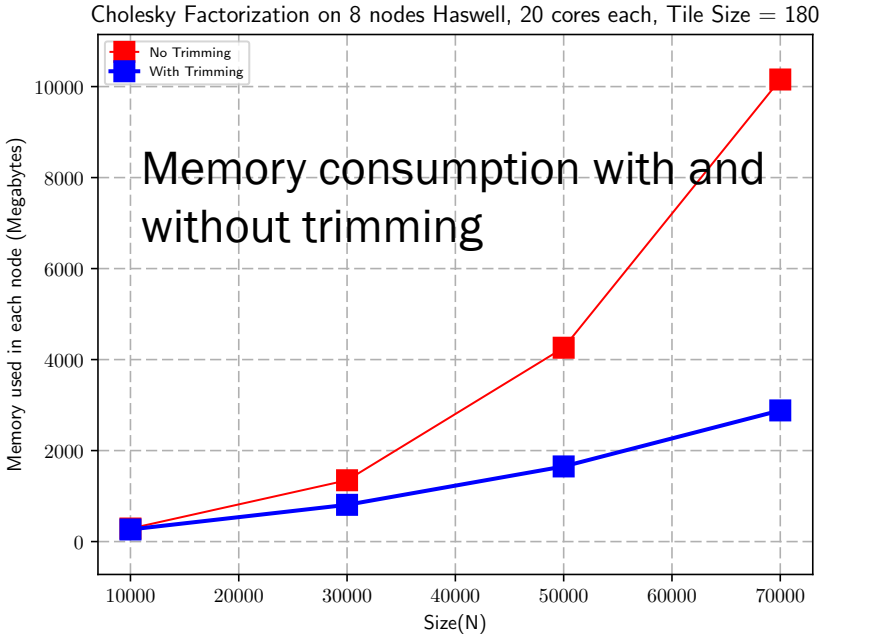
- All participating nodes in distributed setting needs to discover the full task-graph (consistent view)
- DAG of large problem might not fit in memory

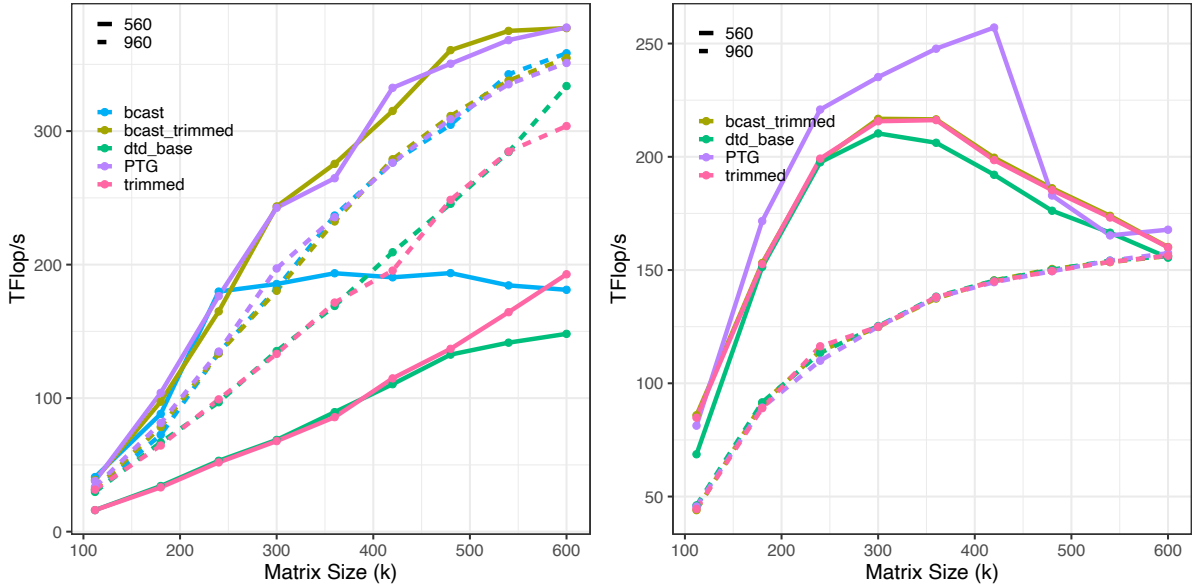
Solution: Partially Unrolling the DAG

- Create partial DAG, progress, repeat (sliding window of DAG)
  - How the DAG is described directs the execution
- Memory usage is bound to the size of sliding window
- Size of window determines how far in future we can see both locally and remotely (affects performance)

Other DTD optimizations

- Trimming (idea popularized by StarPU)
  - Removing remote tasks that do not have any impact locally
- Untying Task Insertion:
  - Users can insert task using one specific thread
  - Users can also insert task that can insert more tasks in the runtime, untying any specific thread from the responsibility of task insertion
  - Allow **recursive task insertion**
  - Allow users to generate independent tasks simultaneously
  - Eliminates performance drop in case of responsible thread being de-scheduled by OS
- Communication
  - Keep track of data version and cache them remotely to avoid sending the same version multiple times
    - What is the life expectancy of these remote copies ?
  - Recycle buffers to optimize memory usage
- PaRSEC Specific Extensions
  - Add collective communications, specialized tasks that operate on a variable number of data
  - Implement owner tracks uses – the opposite concept of tasks trimming





Shaheen II, 512 nodes
Left, Cholesky, Right, QR factorization

The Parameterized Task Graph (JDF)

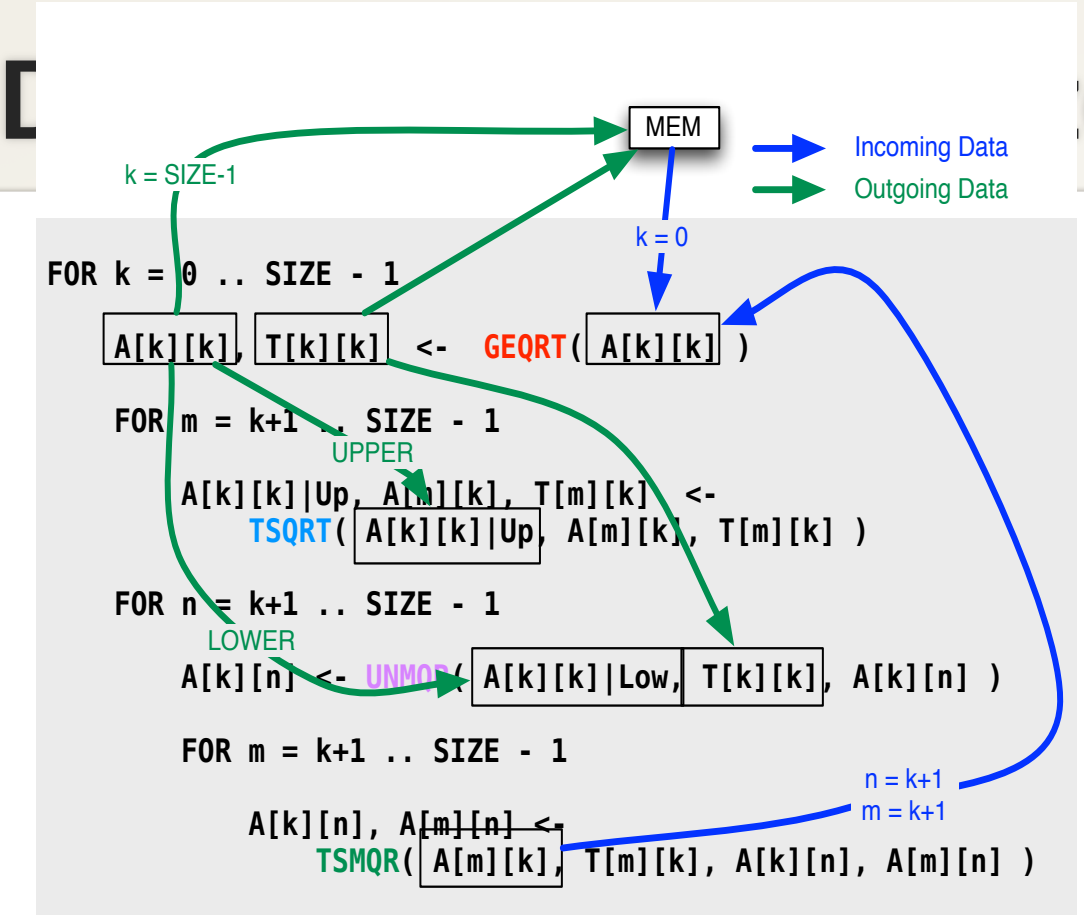
```
GEQRT(k)

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
```

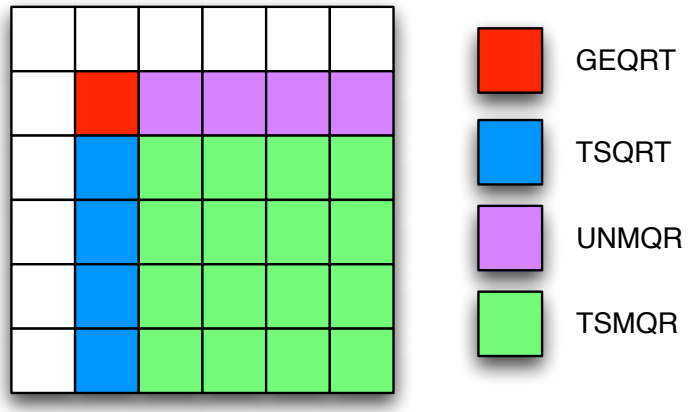
Control flow is possible but not necessary, maximum parallelism is exposed

- A concise parameterized dataflow language, with non-dense iterators and extended expressions via inlined C/C++ code to augment the language
- Only local tasks are instantiated: internal data structures size is inversely proportional to the number of nodes
- The language features multiple collective communication patterns
- Data flows can be typed, to transmit variable data elements
- Tasks can be specialized to target specific devices and refined to adapt to multiple granularities
- Termination mechanism part of the runtime (counting or distributed termination detection)

Data-dependent problems (where the DAG structure depends on the data itself) are more challenging

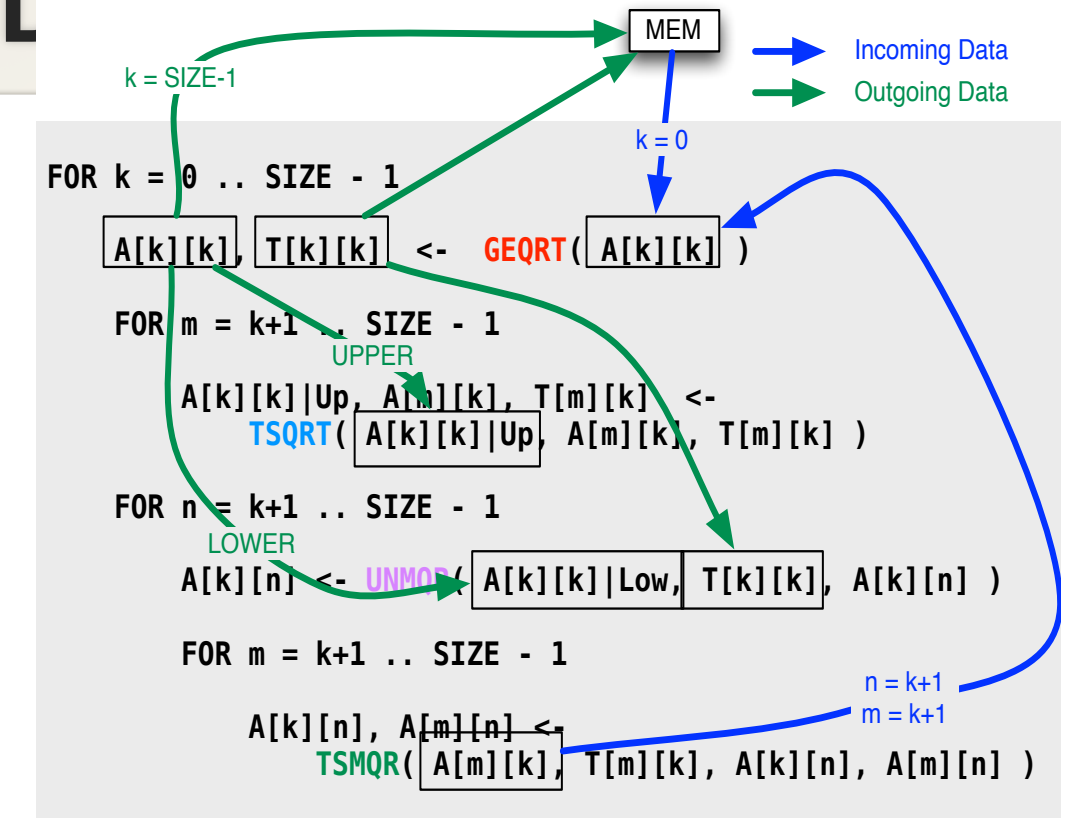
D ed Task Graph (JDF)



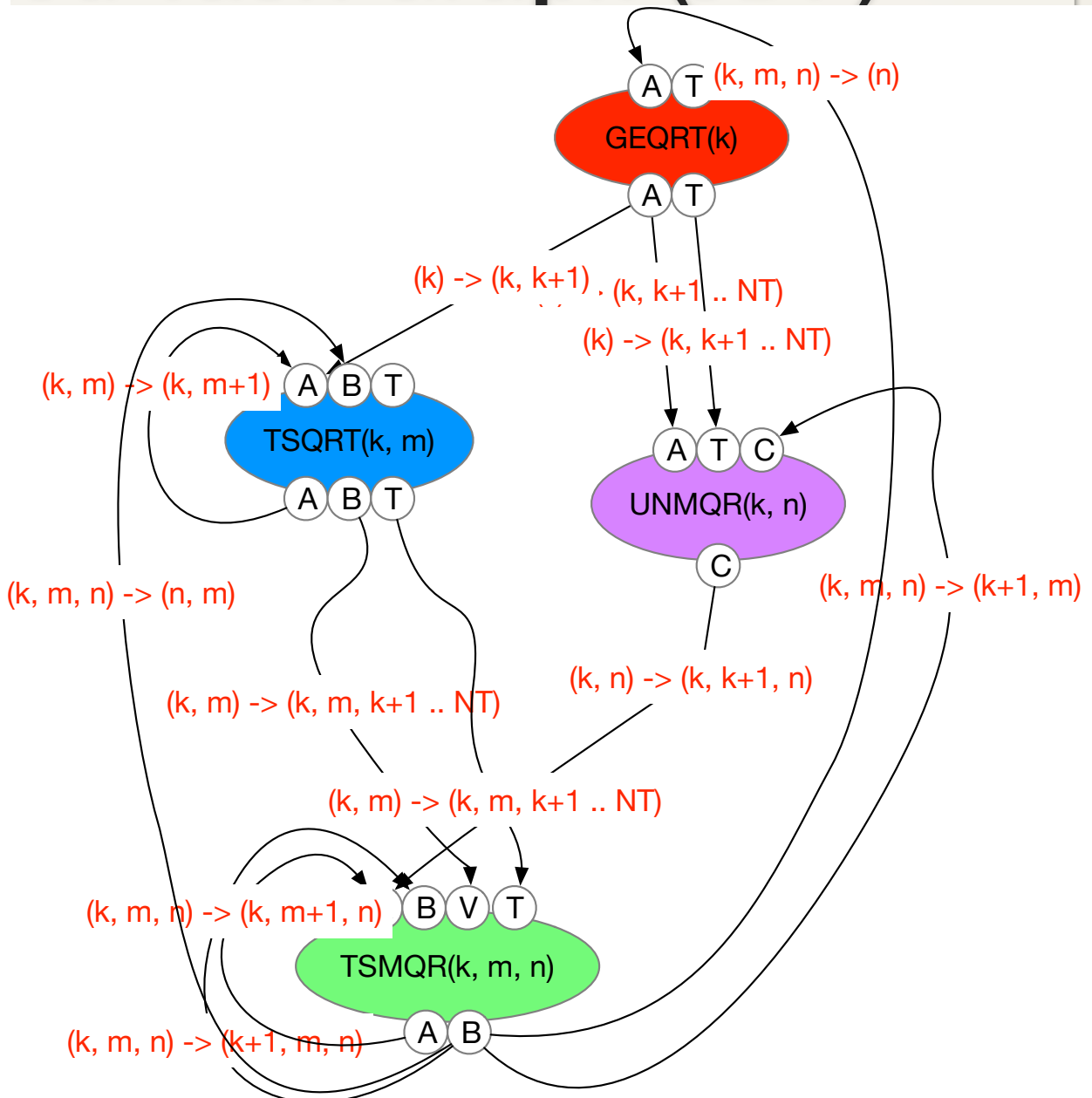


- A dataflow description based on data tracking
- A simple affine description of the algorithm can be understood and translated by a compiler into a control-flow free form (pure dataflow)
- Abide to all constraints imposed by current compiler technology

D ed Task Graph (JDF)



- A dataflow description based on data tr
- A simple affine description of the algori translated by a compiler into a control-f (pure dataflow)
- Abide to all constraints imposed by curr



Overhead of DTD

$$T_{PTG} = \frac{N \times C_T}{P \times n}$$

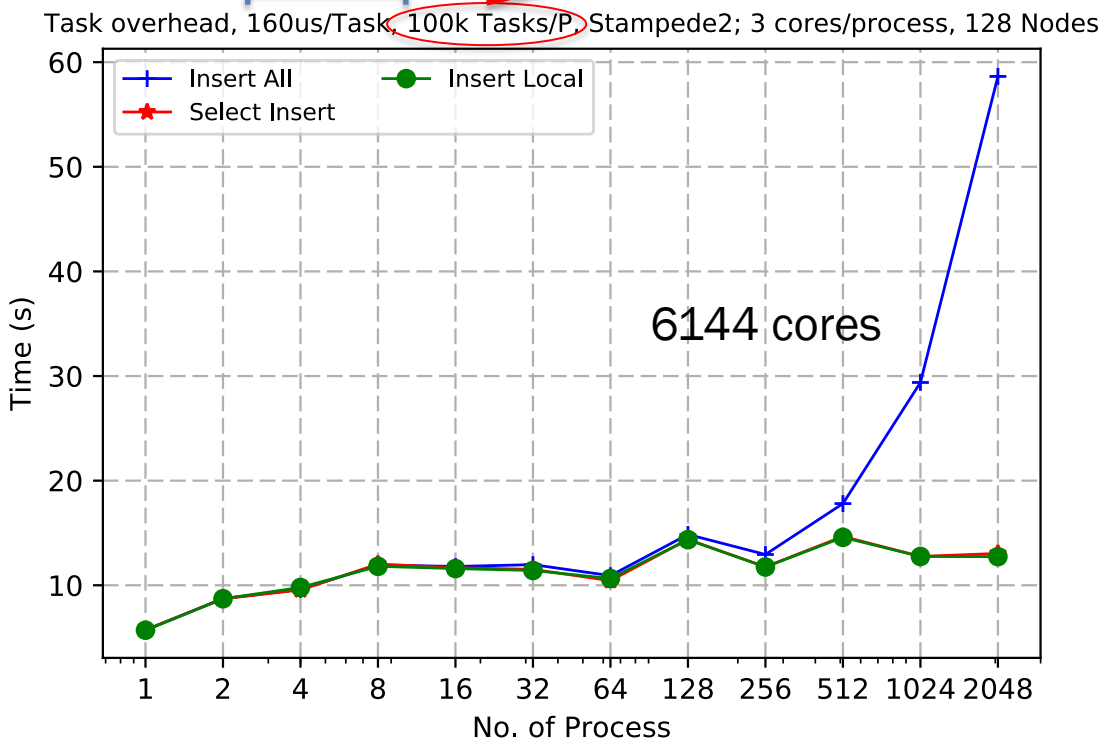
$$T_{DTD} = \frac{N \times C_T}{P \times n} + N \times C_D + \frac{N \times C_R}{P}$$

DTD overhead

$T_{DTD/PTG}$: Overall time
N: Total number of tasks
C_T: Cost/duration of each task
P: Total number of nodes/process
n: Total number of cores
C_D: Cost of discovering a task
C_R: Cost of building DAG/relationship

Fixed task duration

Weak scaling: Fixed number of tasks per process

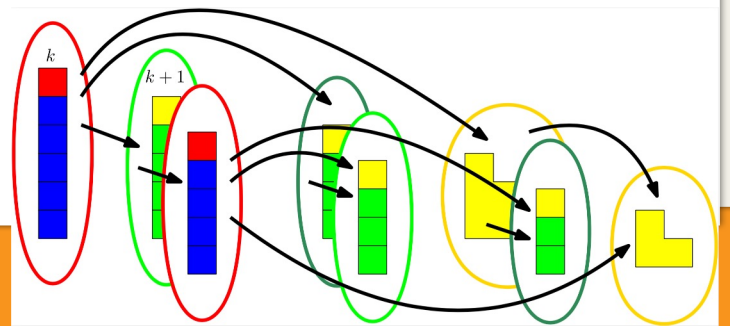


- There are three types of scenario
  - Insert All: Each rank inserts all tasks, and executes only locals
  - Select Insert: Each rank inserts only local tasks but iterates over all tasks.
  - Insert Local: Each rank only inserts local tasks.

SLATE-ish API (templated C++)

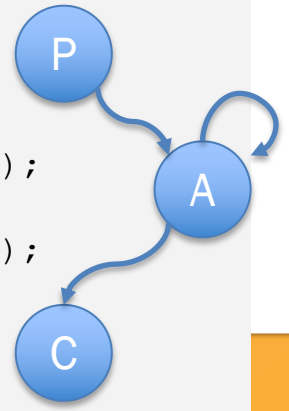
```
for (int64_t k = 0; k < A.nt(); ++k) {
  PaRSEC::potrf_panel
     <HermitianMatrix<scalar_t>, scalar_t>(A, k, A.column(k));

  if (k+1 < A.nt())
    PaRSEC::broadcast_column
      <HermitianMatrix<scalar_t>, scalar_t>
           (A, k, A.column(k), A.column(k+1), A.column(A.nt()-1));
  }
  for (int64_t n = k+1; n < k+1+lookahead && n < A.nt(); ++n) {
    // lookahead column(s)
    PaRSEC::potrf_lookahead
      <HermitianMatrix<scalar_t>, scalar_t>
        (A, k, n, A.column(k), A.column(n));
  }
  if (k+1+lookahead < A.nt()) {  // trailing submatrix
    PaRSEC::potrf_trailing_update
      <HermitianMatrix<scalar_t>, scalar_t>
        (A, k, k+1+lookahead, A.column(k), A.column(k+1+lookahead),
         A.column(A.nt()-1));
  }
  PaRSEC::data_flush (A.parsec_high_level_tp,
                      A.column_range(k, k, A.nt()-1));
}
```

- The SLATE-ish API targets regular algorithms: tile-based task discovery algorithms with explicit synchronization and communications
- Use of templating to manage multiple precision and data representations
- Task discovery based on maintaining the sequential semantic
  - Computing elements need to discover only local tasks
- Communications and synchronizations are both implicit and explicit
- The language/API expresses a control flow
- Explicit communication happens within the progress of these containers and in the background.



Templated Task Graph (TTG)

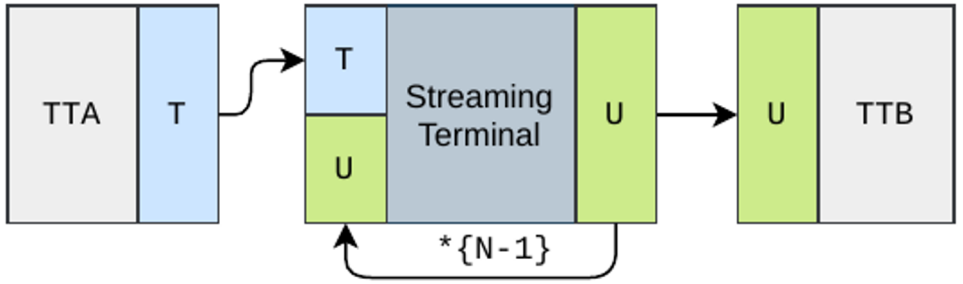
```
class A : public Op<keyT,
                    std::tuple<Out<void, int>,
                    Out<keyT, int>>,
                    A, const int> {
  using baseT = Op<keyT, std::tuple<Out<void,
int>,
                   Out<keyT, int>>, A, const int>;

 public:
  void op(const keyT &key,
          baseT::input_values_tuple_type &&t,
          baseT::output_terminals_type &out) {
    auto &value = baseT::get<0>(t);
    ::ttg::print("A got value ", value);
    if (value >= 100) {
        ::sendv<0>(value, out);
    } else {
        ::send<1>(key + 1, value + 1, out);
    }
  }
};

auto p = std::make_unique<Producer>("P");
auto a = std::make_unique<A>("A");
auto c = std::make_unique<Consumer>("C");

connect<1, 0>(a, a);
connect<0, 0>(a, c);
connect<0, 0>(p, a);
```



- TTG aims at providing the conciseness of intermediate representations, while keeping the flexibility of dynamic task discovery
- The graph expressed statically is more general than the executed DAG: it declares possible flows between classes representing tasks
- The API features explicit emission of flows on output terminals, within the task code, allowing data-dependent DAGs representation
- Because of the send semantics, additional copies of data might be necessary (but we support immutable data, std::move)
- The API features multiple collective communication routines (broadcast, reduction, …)
- Targets irregular applications (block sparse tensor contraction, 6d-tree wave functions representation, MRA)
- Supports multiple runtime (MADNESS, PaRSEC)
- Exploits C++ high-level features (templating, std::move semantics, coroutines)
- Integration of accelerators and multiple devices underway (ongoing development effort)
- Data Serialization: split approach consistent with HPC communication protocols

Streaming terminals

- Flexible task input API
  - Variable number of inputs
  - Gather or Accumulated input, ordered or unordered
- Convertor between types
- Stream size controllable at runtime
- Aggregates N inputs from TTA and pass to TTB

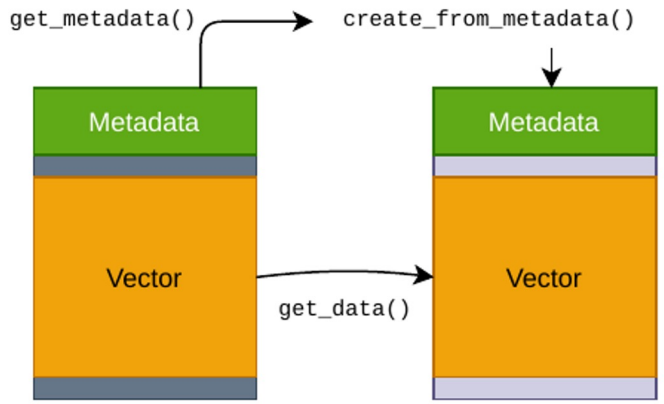


Data Copy Management

- Data traverses through unfolding task graph
- Goal: minimize number of data copies
  - Utilize C++ move and constness semantics
  - Avoid copying data if we know data is immutable

```
1  void taskfn(const TaskID& task_id, const MatrixTile& input) {
2    MatrixTile output = compute_output_tile(input);
3    send<0>(task_id, output); // new copy required
4    send<1>(task_id, move(output)); // no copy due to move
5    send<2>(task_id, input); // no copy as input is const
6  }
```

- TTG supports Boost.Serialization & MADNESS archives
  - Requires (de-)serialization of objects into/from message buffers
- Enable zero-copy transfers through overloading
  - Copy metadata
  - Instantiate object
  - Transfer data between objects directly



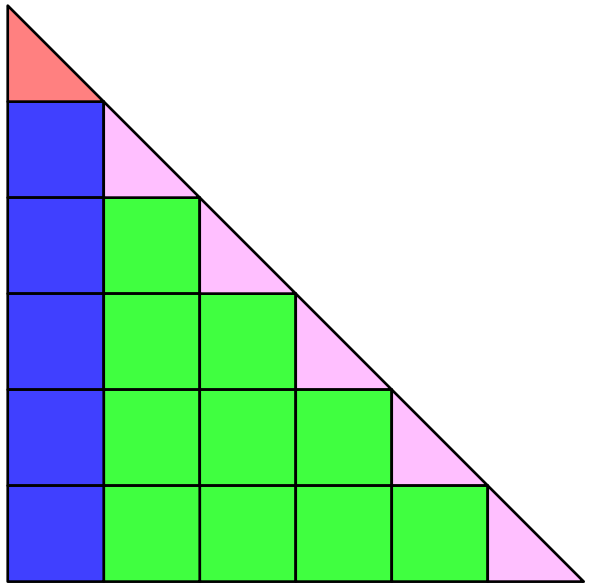

```
template<>
struct SplitMetadataDescriptor<MatrixTile> {
  auto get_metadata(const MatrixTile& t) {
    return t.metadata();
  }
  auto get_data(MatrixTile& t) {
    return std::array<iovec, 1>({t.size(),
                                 t.data()});
  }
  auto create_from_metadata(metadata_t& meta) {
    return MatrixTile(meta);
  }
};
```

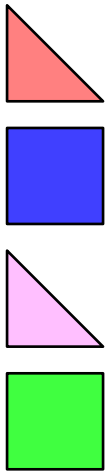
Broadcasting Data in TTG

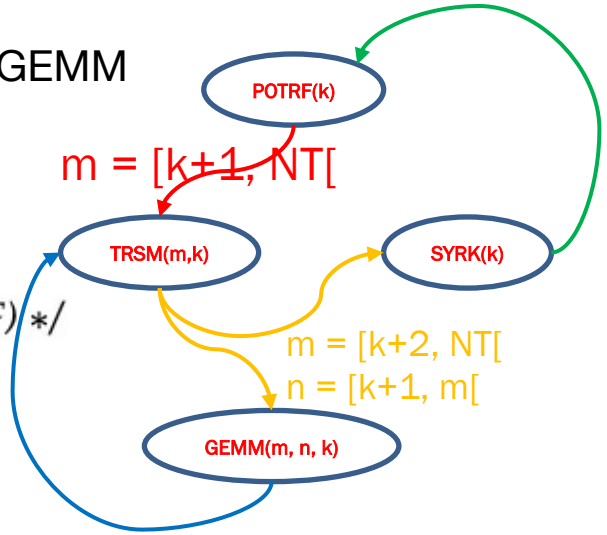
- Broadcast data to multiple keys on multiple output terminals
  - Populate successor keys
  - Broadcast keys and data

```
1  ttg::Edge<Key1, MatrixTile<double>> syrk_potrf("syrk_potrf");
2  ttg::Edge<Key2, MatrixTile<double>> potrf_trsm("potrf_trsm");
3  auto f = [=](const Key1& key, MatrixTile<T>&& tile_kk){
4      const int K = key.K;
5       /* invoke POTRF kernel */
6      lapack::potrf(lapack::Uplo::Lower, tile_kk.rows(), tile_kk.data(), tile_kk.rows());
7
8      /* send the tile TRSMs and SYRK */
9      std::vector<Key2> keylist;
10     keylist.reserve(A.rows() - K);
11     for (int m = K+1; m < A.rows(); ++m) {
12         keylist.push_back(Key2(m, K));
13     }
14     ttg::broadcast<0, 1>(std::make_tuple(Key2(K, K), keylist), std::move(tile_kk));
15   };
16 ttg::make_tt(f, ttg::edges(input), ttg::edges(output_result, output_trsm), "POTRF",
17              {"tile_kk"}, {"output_result", "output_trsm"});
```

Cholesky in TTG



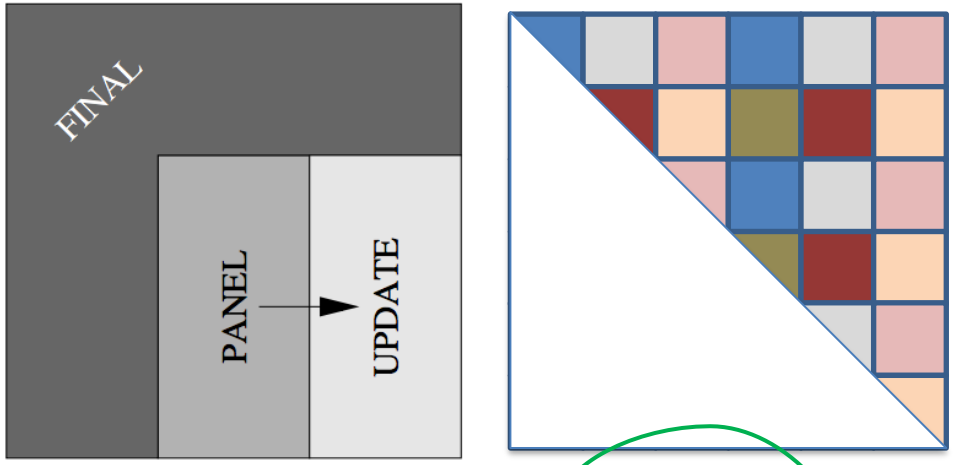


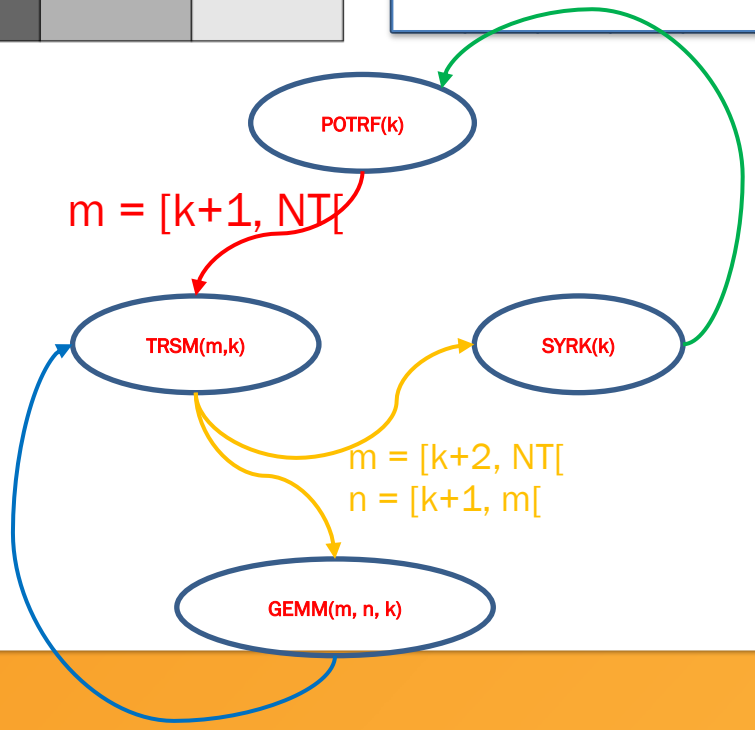


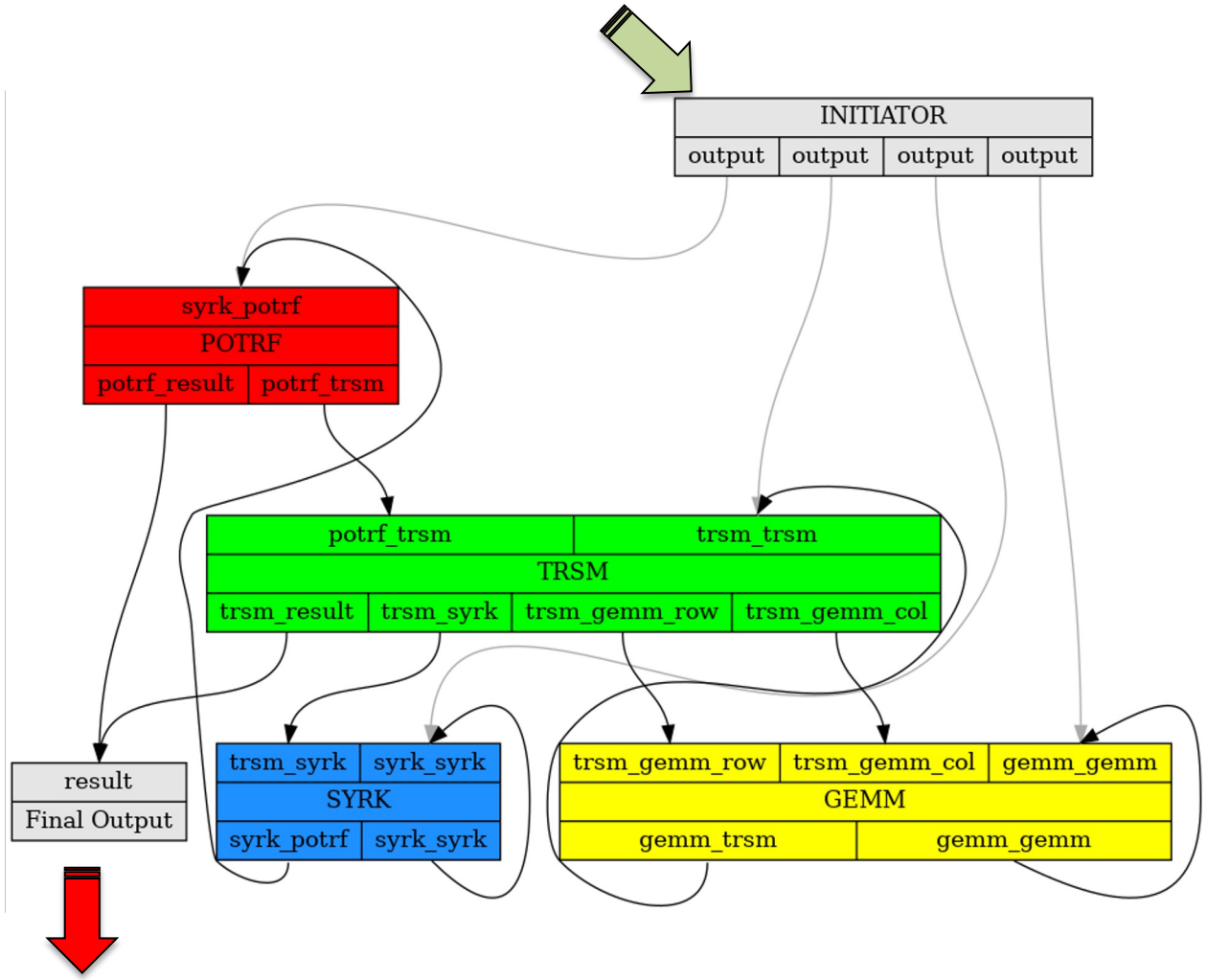
1 **for** $k = 0$ *to* $NT - 1$ */* Panel Factorization (PF) */*
2 DPOTRF (C_{kk})
3 **for** $m = k + 1$ *to* $NT - 1$
4 DTRSM (C_{kk}, C_{mk})
5 **for** $m = k + 1$ *to* $NT - 1$
6 DSYRK (C_{mk}, C_{mm})
7 **for** $m = k + 2$ *to* $NT - 1$ */* Trailing Submatrix Update */*
8 **for** $n = k + 1$ *to* $m - 1$
9 DGEMM (C_{mk}, C_{nk}, C_{mn})

```
/* Edges with 1-tuple task IDs */
ttg::Edge<Int1, Tile> init_potrf;
/* Edges with 2-tuple task IDs */
ttg::Edge<Int2, Tile> potrf_trsm, trsm_result,
                      trsm_syrk, gemm_trsm;
/* Edges with 3-tuple task IDs, encodes the iteration K */
ttg::Edge<Int3, Tile> trsm_gemm_row, trsm_gemm_col;

auto POTRFOp = ttg::make_tt(potrf_fn /* not shown here */,
                            /* input edges */
                            ttg::edges(init_potrf),
                            /* output edges */
                            ttg::edges(potrf_results,
                                       potrf_trsm));


auto trsm_fn =
    [](const Int2& id,
       const Tile<T>&  tile_kk,
       Tile<T>&& tile_mk,
       std::tuple<ttg::Out<Int2, Tile<T>>,
                  ttg::Out<Int2, Tile<T>>,
                  ttg::Out<Int3, Tile<T>>,
                  ttg::Out<Int3, Tile<T>>>& out){
  const auto [I, J] = id;
  const auto K = J;

  /* call LAPACK library's tsrm function */
  TRSM(tile_kk, tile_mk);

  std::vector<Int3> row_ids, col_ids;
  /* ids for gemms row I */
  for (int n = J+1; n < I; ++n)
    row_ids.push_back(Int3(I, n, K));

  /* ids for gemms column I */
  for (int m = I+1; m < NROWS; ++m)
    col_ids.push_back(Int3(m, I, K));

  /* broadcast the result to 4 output terminals:
   * 0: to final output task writing back the tile;
   * 1: to the SYRK kernel;
   * 2: to the gemm tasks on in row I;
   * 3: to the gemm tasks in column K; */
  ttg::broadcast<0, 1, 2, 3>(
    std::make_tuple(id, Int2(I, K), row_ids, col_ids),
    std::move(tile_mk), out);
};
auto TRSMOp = ttg::make_tt(trsm_fn,
                           /* input edges */
                           ttg::edges(potrf_trsm, gemm_trsm),
                           /* output edges */
                           ttg::edges(trsm_result, trsm_syrk,
                                      trsm_gemm_row,
                                      trsm_gemm_col));
```

Cholesky in Templated Task Graph (TTG)





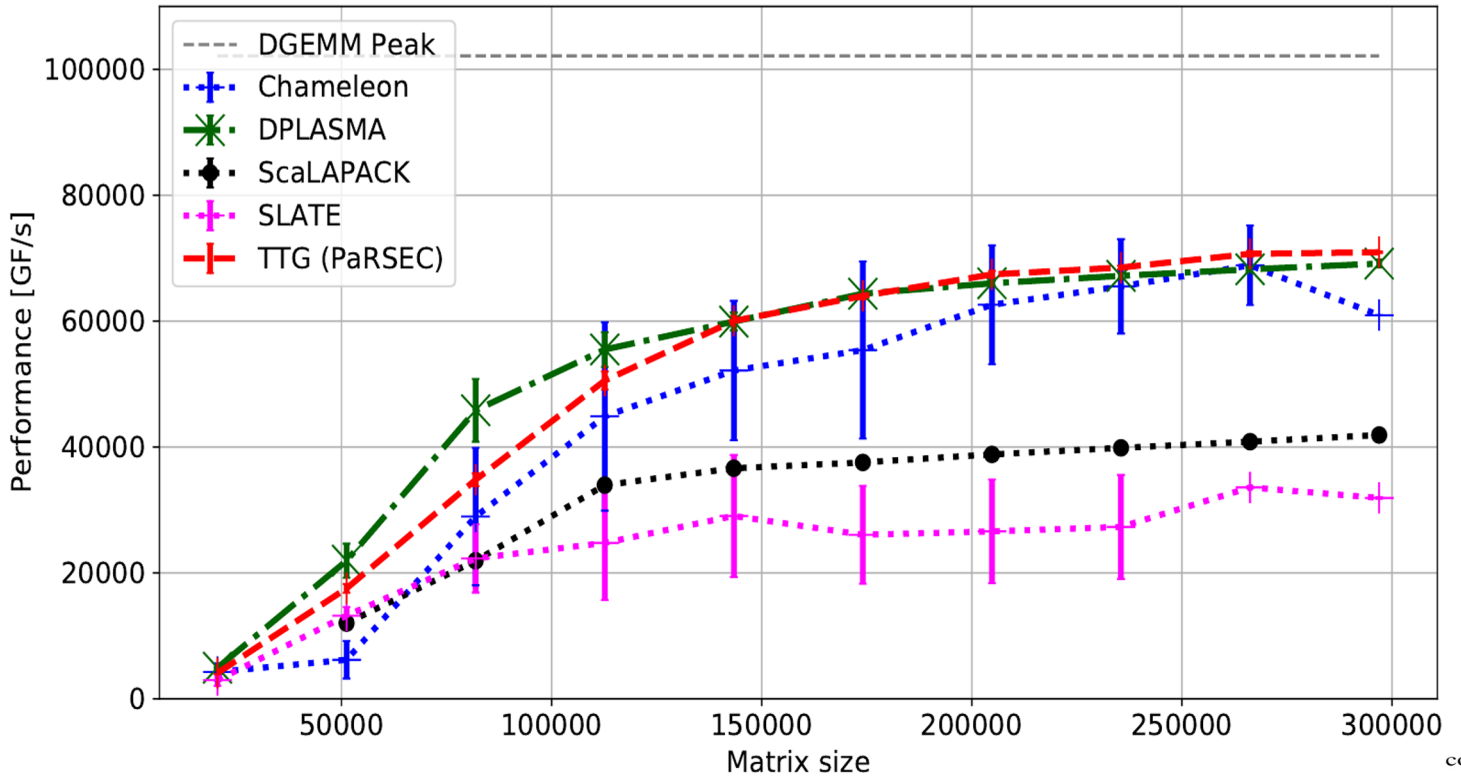


Cholesky Factorization: Matrix Scaling

- Hawk, 64nodes
- Matrix: increasing

https://www.hlrs.de/systems/hpe-apollo-hawk/

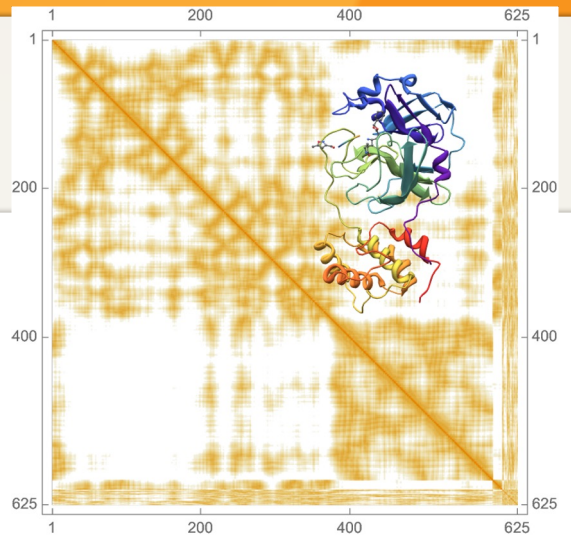
Performance of TTG matches DPLASMA asymptotically



configurations

Software	*Hawk*	*Seawulf*
MPI	Open MPI 4.1.1, UCX 1.10.0	Intel MPI 20.0.2
Compiler	GCC 10.2.0	GCC 10.2.0
HWLOC	1.11.9	1.11.12
MKL	19.1.0	20.0.2

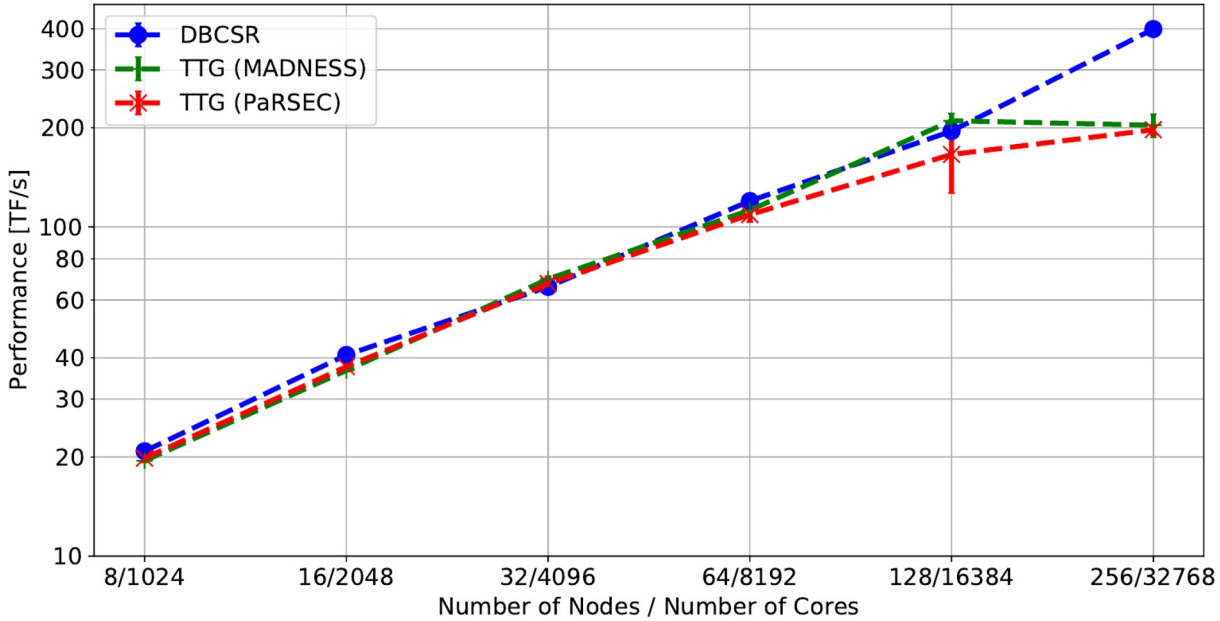
Blocked SpMM



- Yukawa integral operator for the main protease of the SARS-CoV-2 virus in complex with the N3 inhibitor (2,500 atoms)
- Matrix size 140,440, tile size <= 256

TTG matches DBCSR up to 128 nodes

DBCSR's 2.5D beats TTG at scale



Multi-Resolution Analysis (MRA)

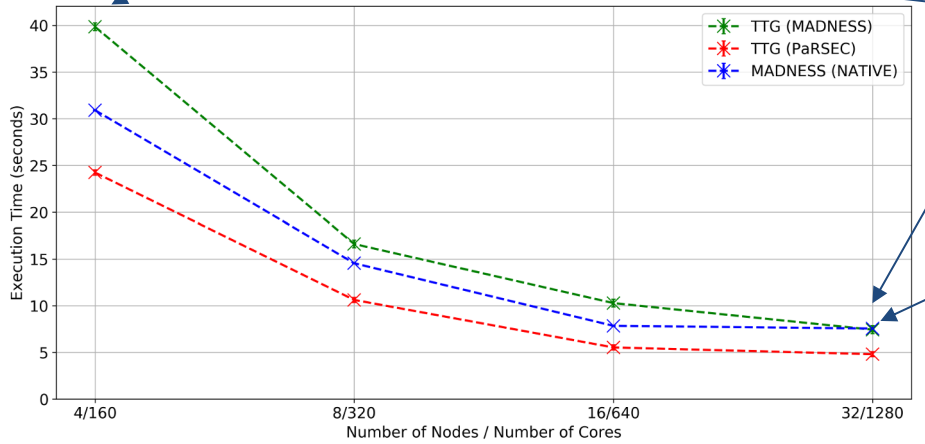
- Order-10 multiwavelet representation of 3-D Gaussian functions, originally implemented in MADNESS
  - **Seawulf**: 120 Functions, 2x20 threads per node
  - **Hawk**: 400 Functions, 8x16 threads per node

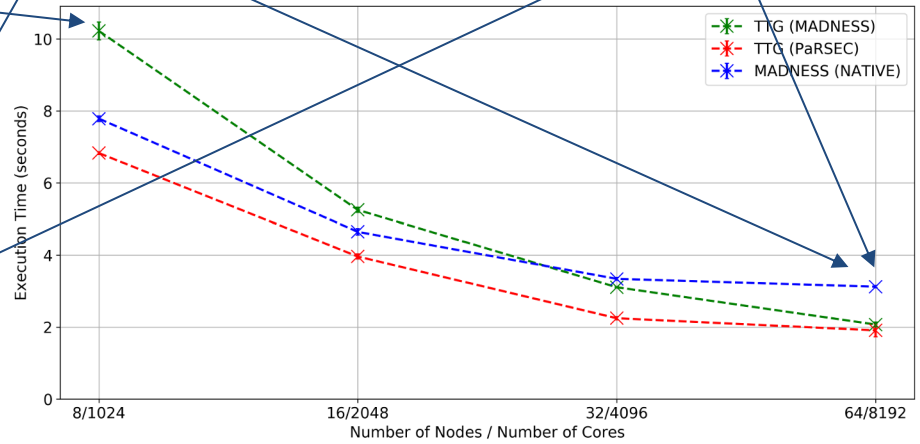
Memory size

1.5x speedup over MADNESS

Concurrency

Seawulf





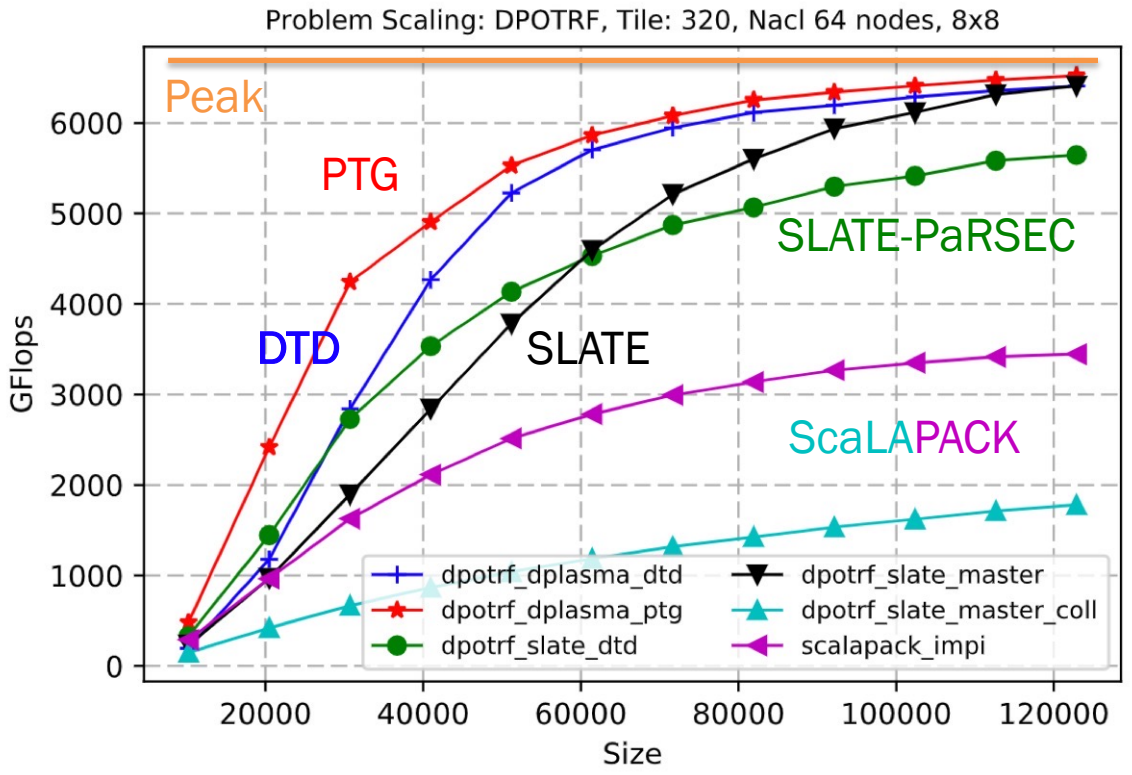
Hawk

PaRSEC Domain Specific Languages

simplicity ← Dynamic Task Discovery — SLATE – C++ — Parameterized Task Graph — Templated Task Graph → flexibility

Problem Scaling of a Cholesky Factorization (DPOTRF)

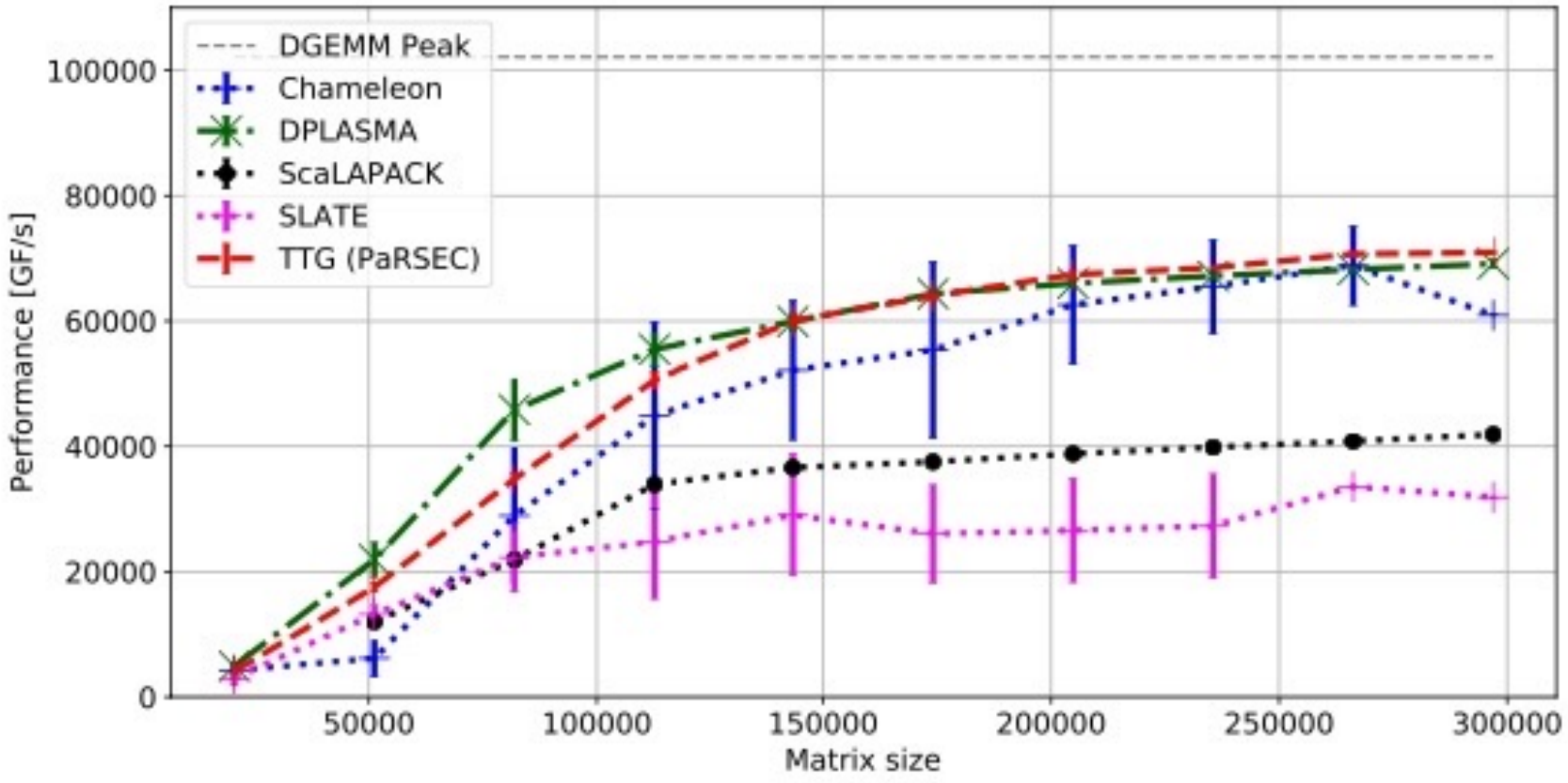
64 nodes, Intel Xeon X5660@2.8GHz, 12 core / node



PaRSEC Domain Specific Languages

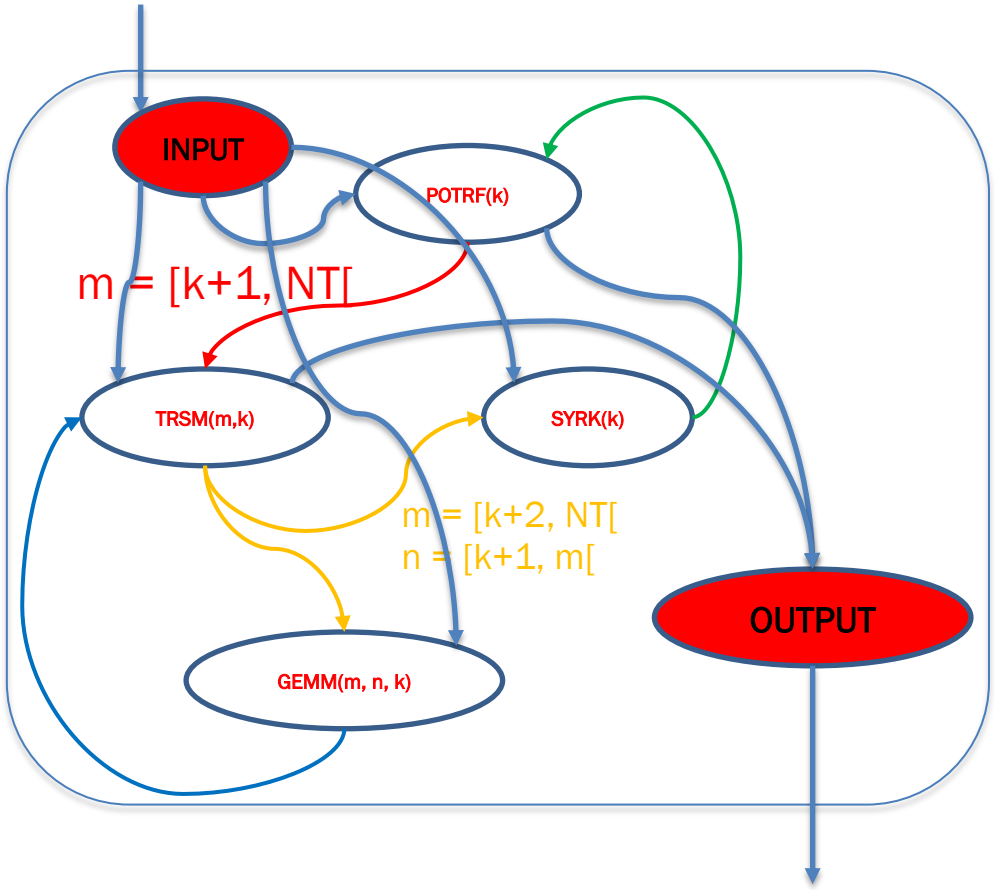
simplicity ← Dynamic Task Discovery — SLATE – C++ — Parameterized Task Graph — Templated Task Graph → flexibility

Problem scaling on 64 nodes on Hawk (tile size 512x512)



https://www.hlrs.de/systems/hpe-apollo-hawk/

Natural data-dependent DAG Composition



- So far:
  - **Edges** connect template tasks through **terminals**
    - A single output terminal (the edge input)
    - Multiple input terminals (the edge outputs)
  - Tasks decide on which output terminal(s) to send data

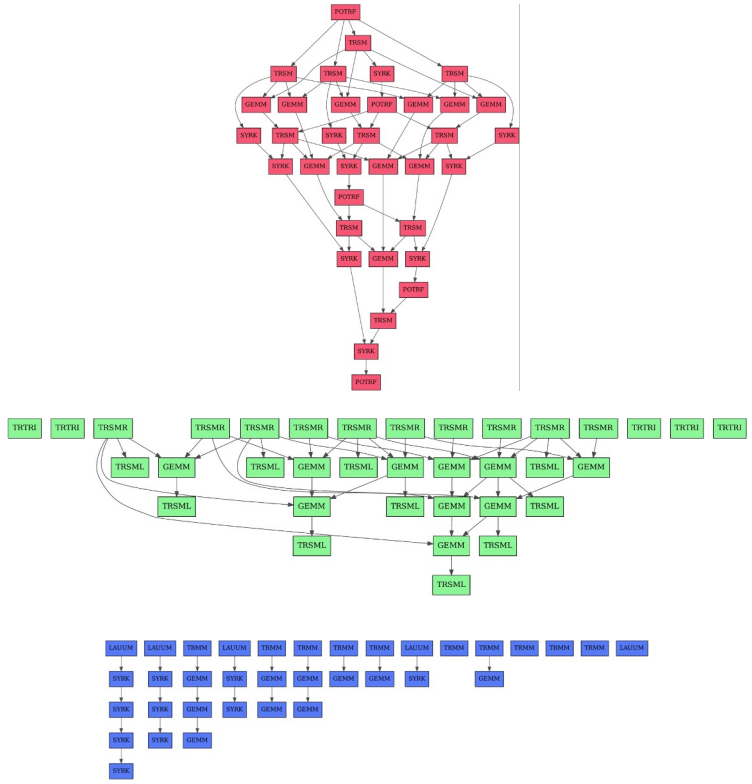
- Edges as Composition Devices
- Introduce **Dispatch Tasks** to Task Graphs
  - Dispatch incoming data to internal tasks
  - Internal edges and tasks not visible to outside
  - Single input edge to receive data from predecessors
  - Single output edge, to connect to successors

Example: Cholesky Matrix Inversion

- Cholesky Factorization (POTRF) followed by matrix inversion
  - Given A, compute A^{-1}
  - A: Hermitian positive-definite matrix

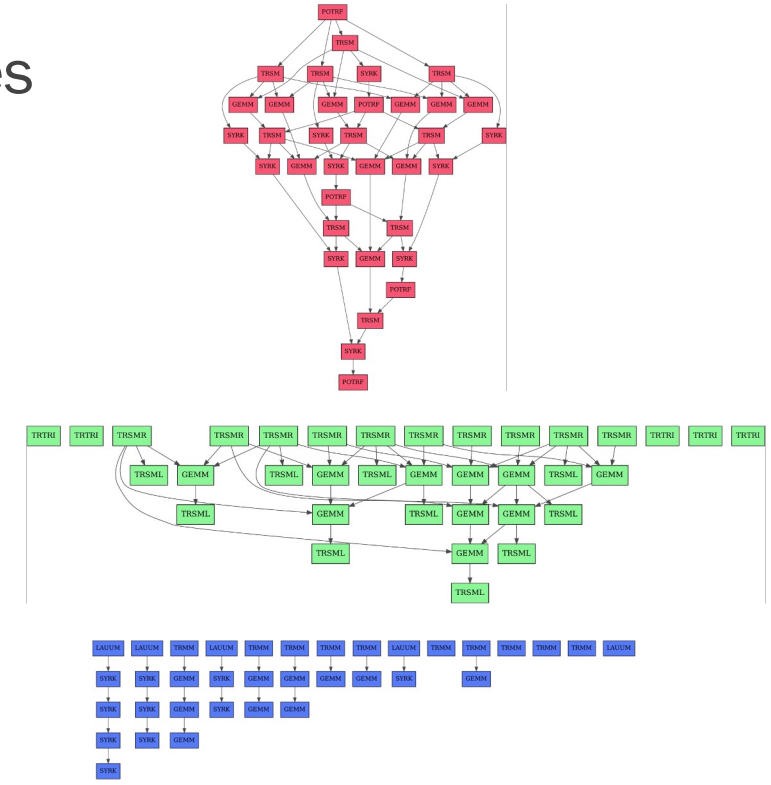
- Inversion: Given L from POTRF
  - Compute L^{-1} from L (TRTRI)
  - Compute $A^{-1} = (L^{-1})^T L^{-1}$ (LAUUM)
  - POTRI = TRTRI ⊕ LAUUM

- POINV = POTRF ⊕ POTRI
  = POTRF ⊕ TRTRI ⊕ LAUUM



Existing Approaches to Task Graph Composition

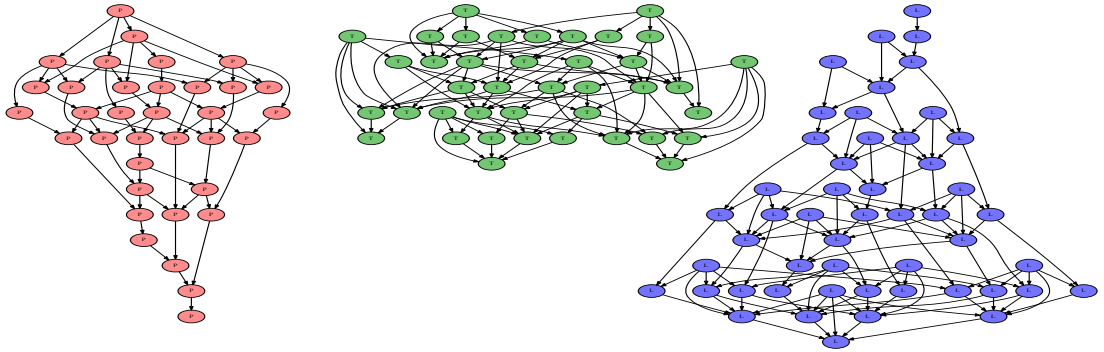
- **Dependency**-based approaches
  - OpenMP, OmpSs, StarPU, PaRSEC DTD, …
  - Sequential discovery of tasks and their dependencies
  - Limited discovery windows
  - Limited scalability
- **Handle**-based approaches (HPX)
  - Composition through future passing
  - One future per element
  - Significant efforts for LA required
- **Parameterized Task Graphs** (PaRSEC PTG)
  - Flow within a graph
  - Operates exclusively on data collections
  - Missing: external interfaces



Natural data-dependent DAG Composition

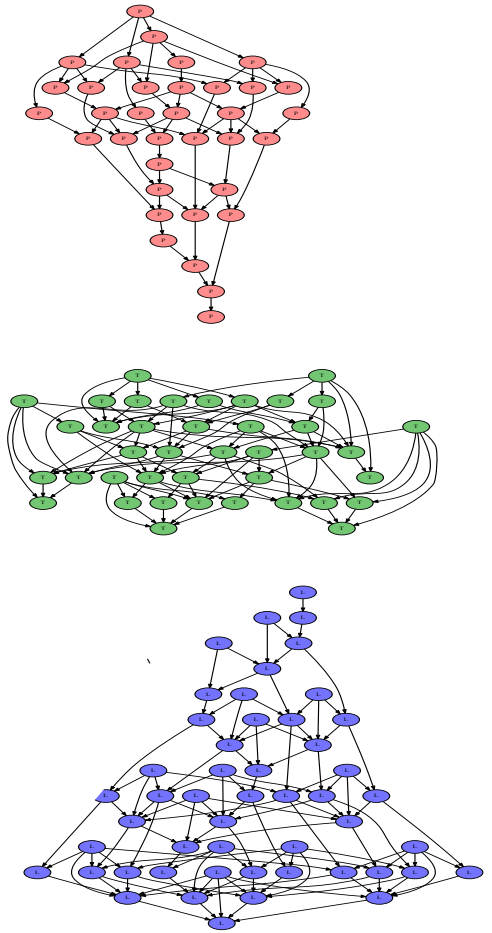
Example POTRI = POTRF + TRTRI + LAUUM

POTRF TRTRI LAUUM

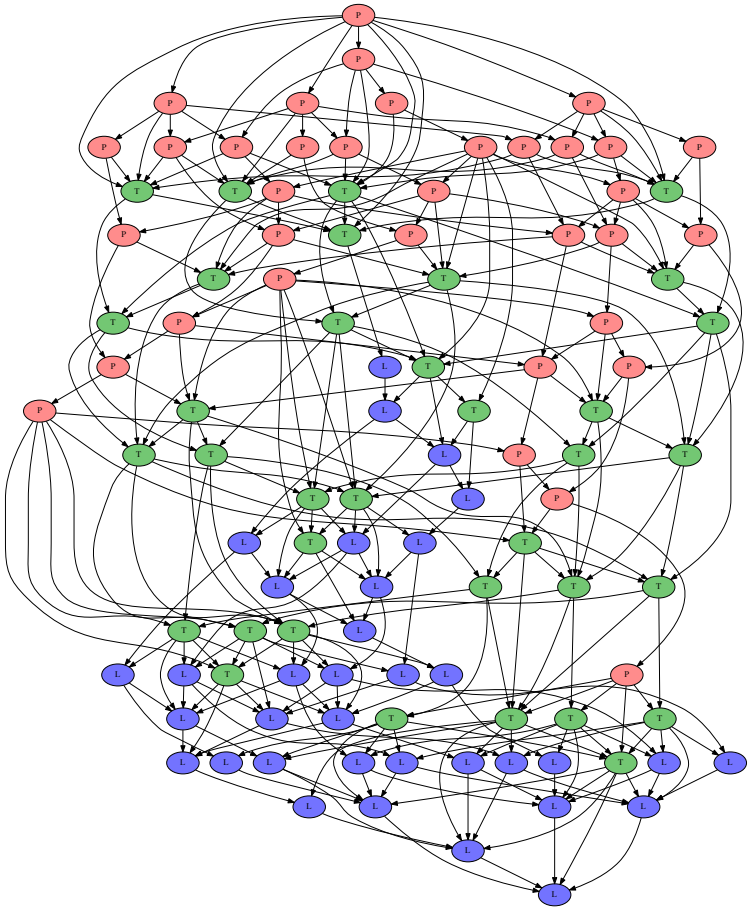


- 3 approaches:
  - **Fork/join**: complete POTRF before starting TRTRI
  - **Compiler-based**: give the three sequential algorithms to the Q2J compiler, and get a single PTG for POINV
  - **Runtime-based**: tell the runtime that after POTRF is done on a tile, TRTRI can start, and let the runtime compose

Traditional



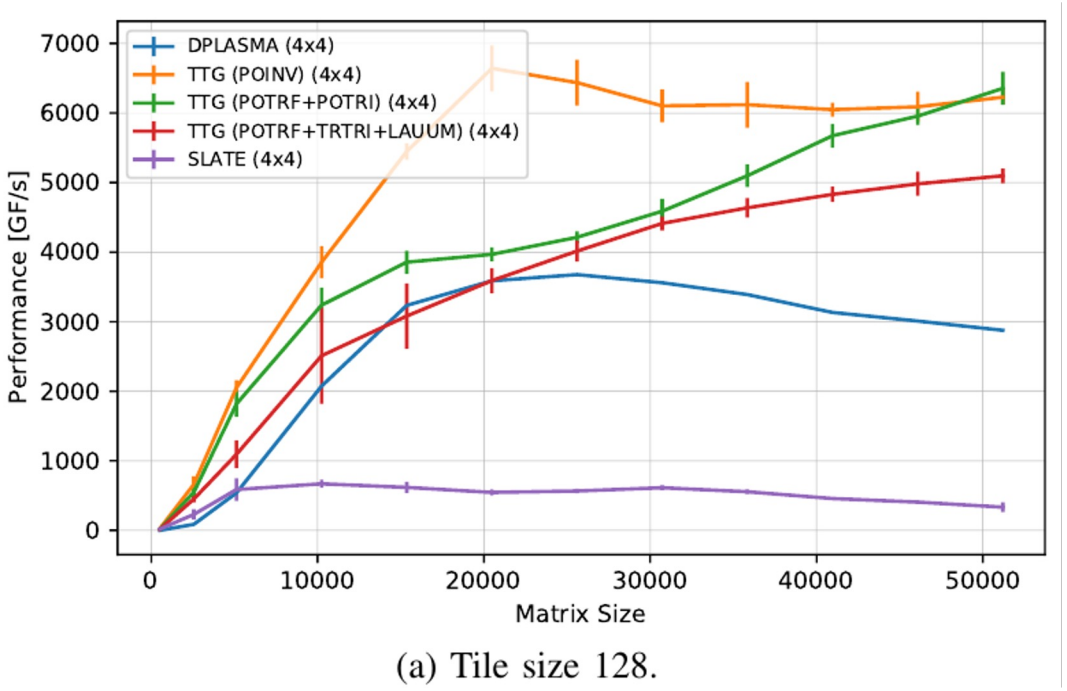
PaRSEC



POINV Composition

https://www.hlrs.de/systems/hpe-apollo-hawk/

- 16 nodes on Hawk, 64 threads each
- Full composition beneficial for small tile sizes
  - Fine-grain composition helps hide communication latency
  - Beats both DPLASMA (based on PaRSEC PTG) and SLATE

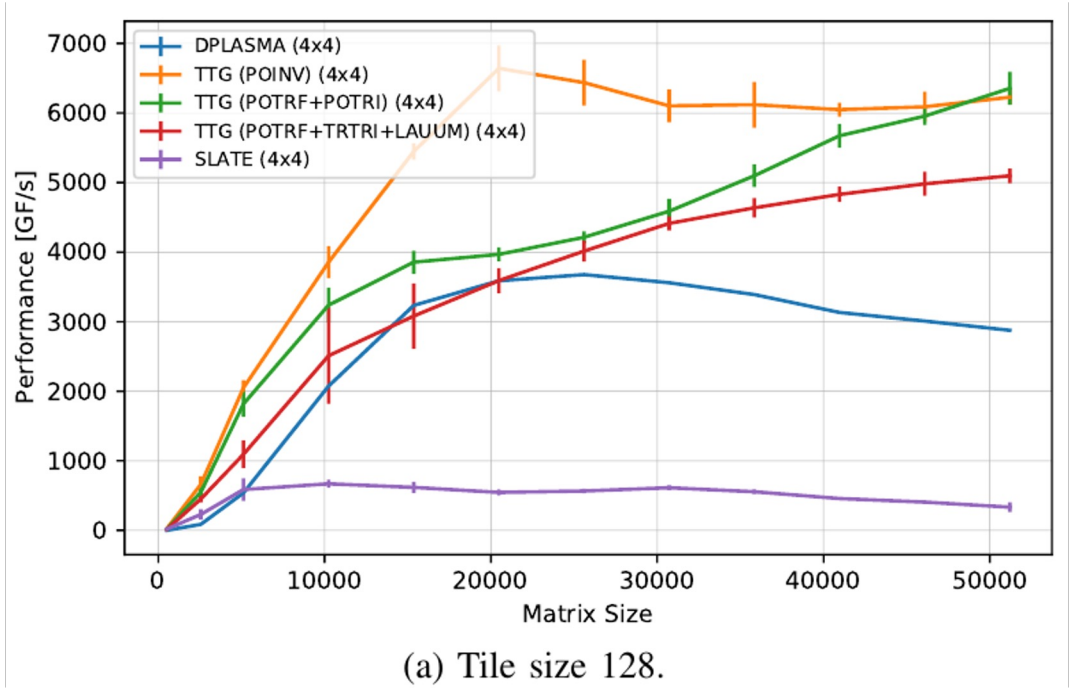


(a) Tile size 128.

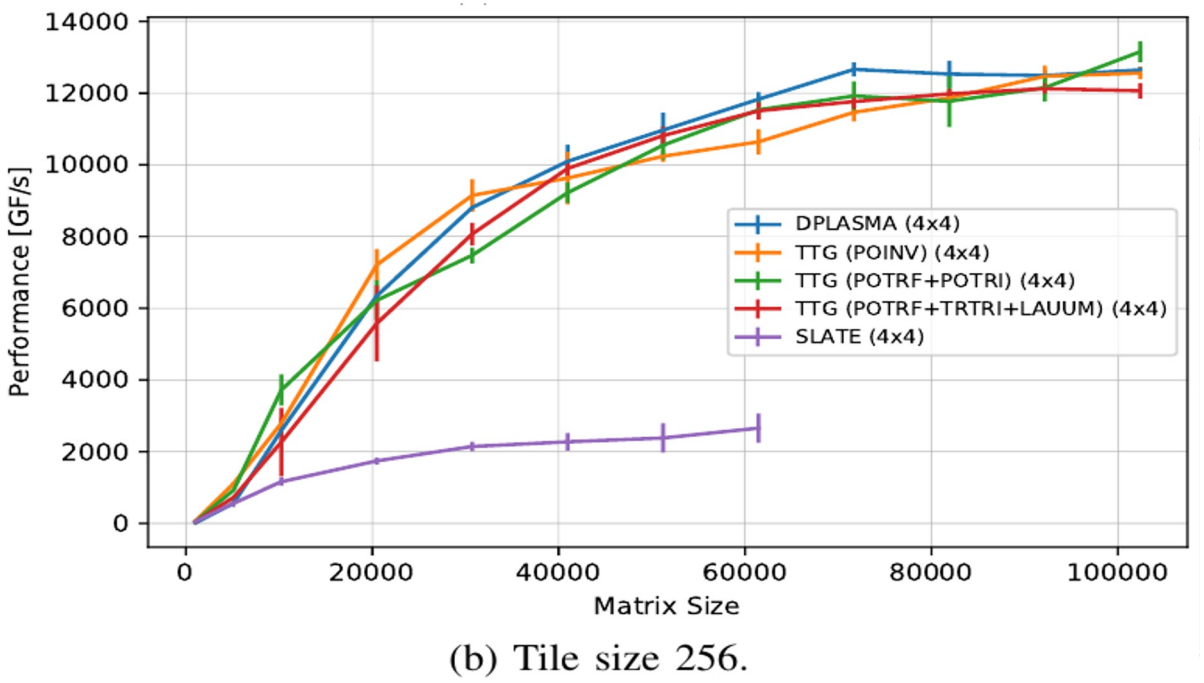
POINV Composition

https://www.hlrs.de/systems/hpe-apollo-hawk/

- 16 nodes on Hawk, 64 threads each
- Full composition beneficial for small tile sizes
  - Fine-grain composition helps hide communication latency
  - Beats both DPLASMA (based on PaRSEC PTG) and SLATE



(a) Tile size 128.

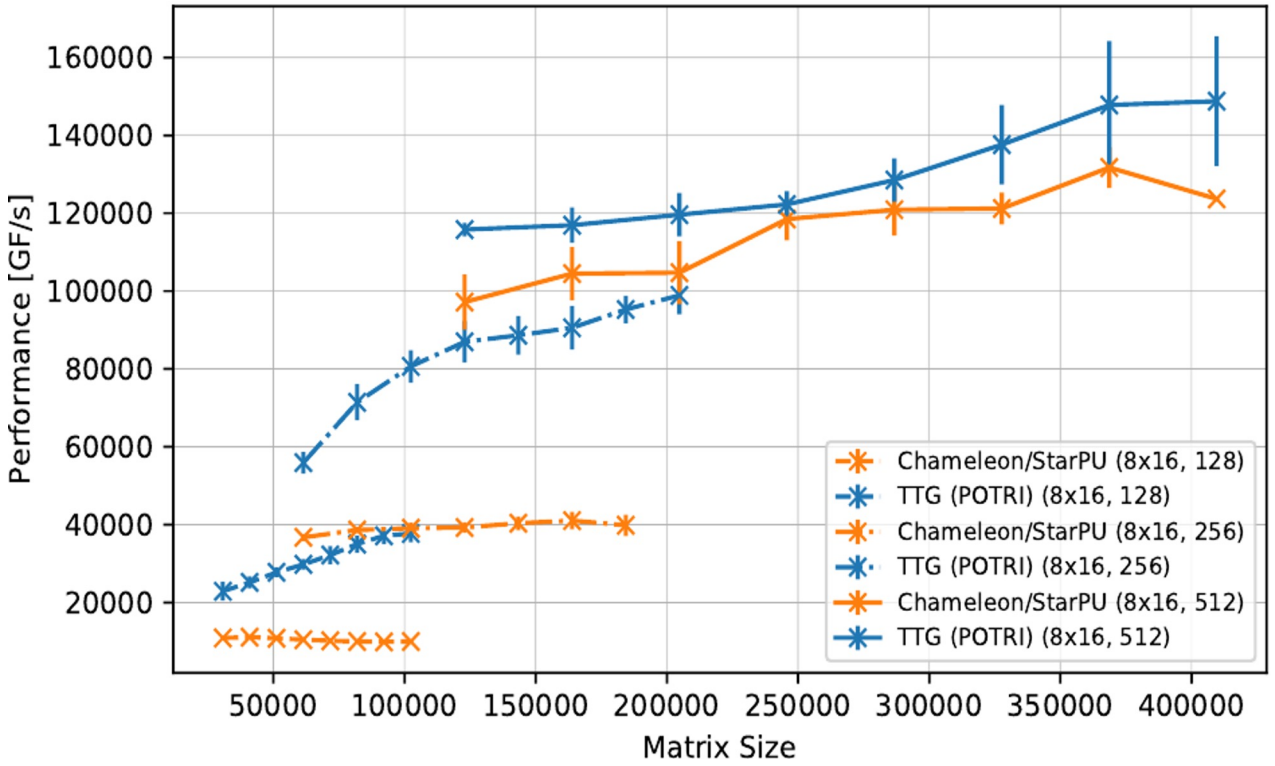


(b) Tile size 256.

POTRI: Comparison with Chameleon

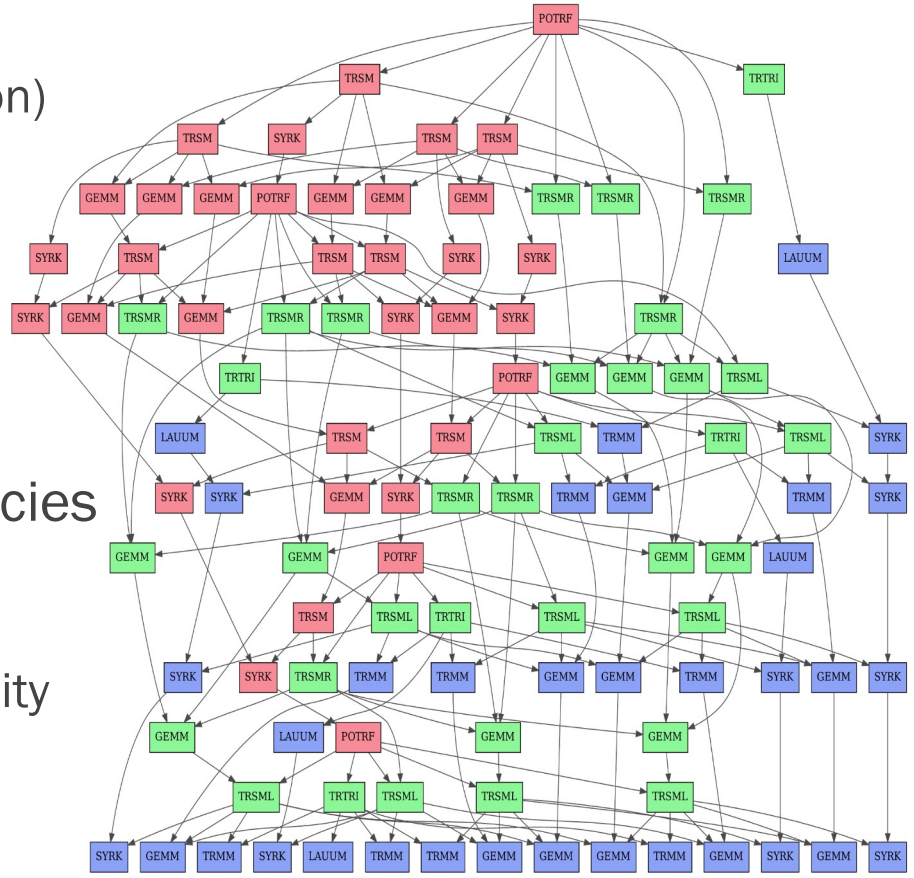
- 128 nodes on Hawk
- Chameleon (v1.1.0, using StarPU 1.3.9)
- POTRI : TRTRI ⊕ LAUUM
- TTG performance benefits
  - Depth-first execution
  - Parallel distributed task discovery

https://www.hlrs.de/systems/hpe-apollo-hawk/

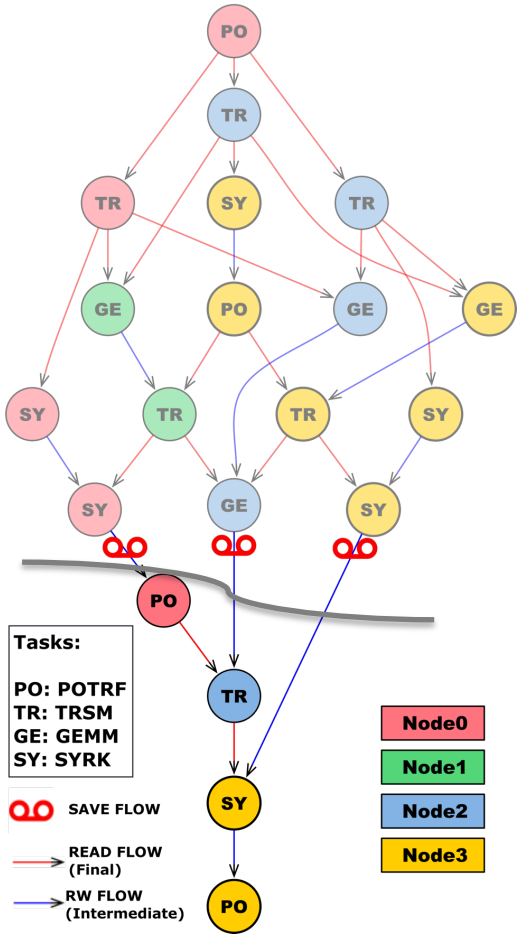


Benefits of Composition

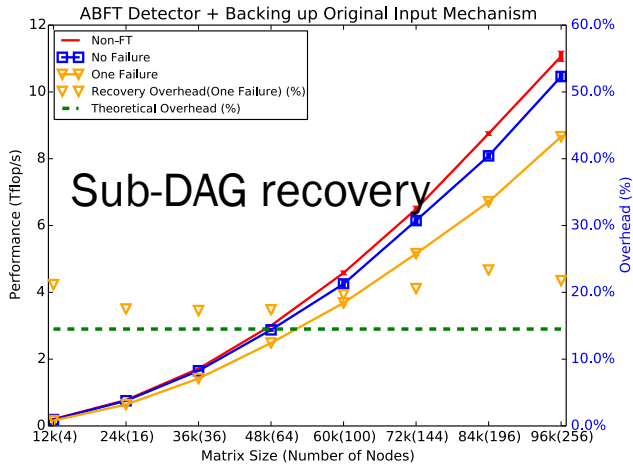
- Minimized serial application parts
- Application-level **depth-first execution** (memory reuse)
  - Cache reuse
  - Device data transfer minimization (for out-of-core computation)
- **Flexibility**: reusing intermediate results becomes trivial
  - Example: tiles of a Cholesky factorized matrix
    - As input for POTRI
    - Used together with POTRI results
  - PaRSEC DSL will manage tile copies
- Task graph composition helps hide communication latencies
- **Edges represent sets of future values**
  - Distribute values to multiple subscribers (input terminals)
  - Provide a clean interface to encapsulate black-box functionality
  - Coupling of graphs through single input, single output Edge
- Full-scale application composition without breaking abstraction barriers

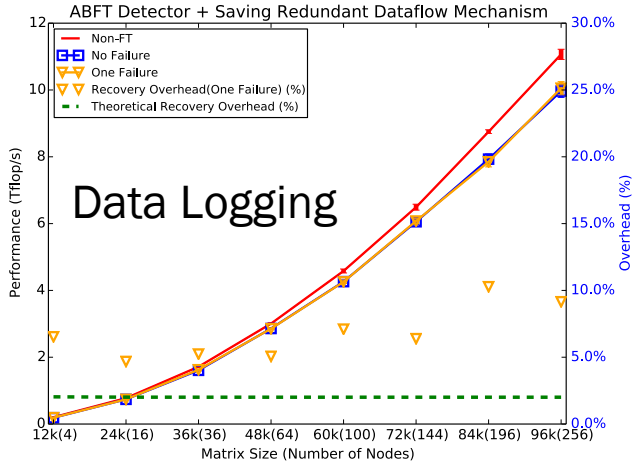


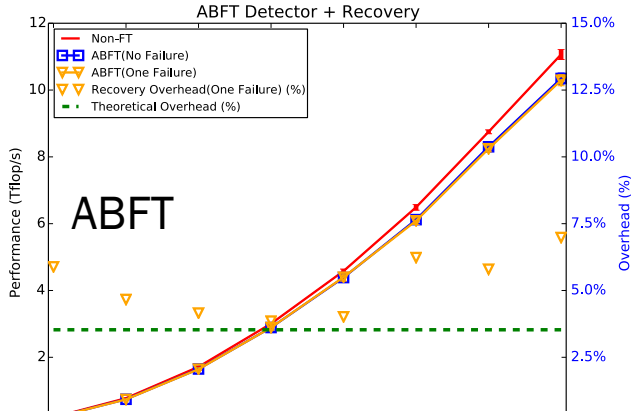
Resilience support from runtime



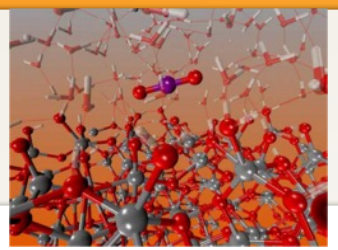
- Recovery based on leaving data safely behind (generic & low-overhead)
  - Partial DAG recovery
- Burst of errors are supported, multiple sub-DAGs will be executed in parallel with the original
- Merge resilient features into runtime:
  - Reserve minimum dataflow for protection
  - Minimize task re-execution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



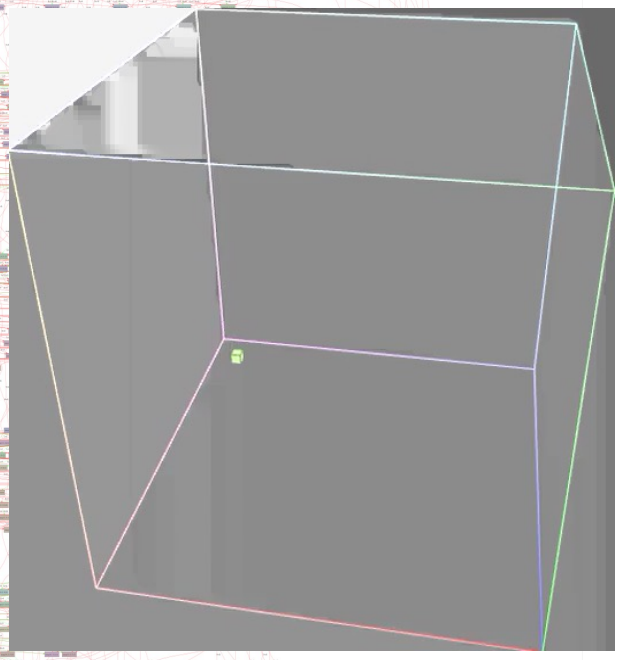


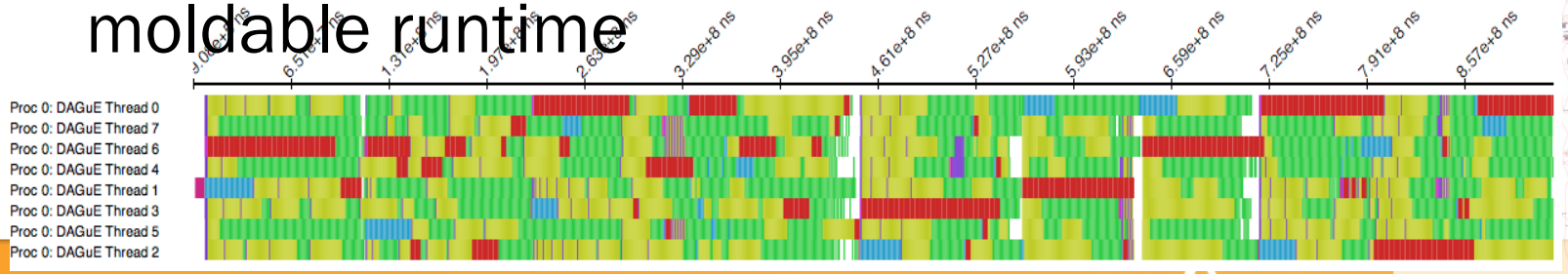


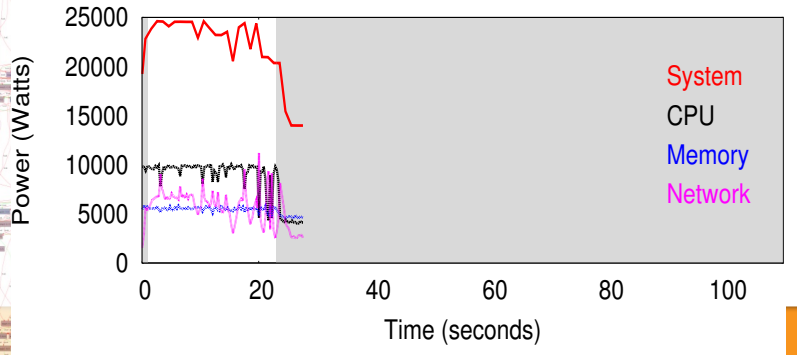
The PaRSEC ecosystem



- 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.

DPLASMA (An improved heterogeneous ScaLAPACK)

Extensions to DPLASMA library and PaRSEC runtime

- DPLASMA provides a wrapped version of the ScaLAPACK API.
- Wrapper hides PaRSEC API from the application and builds transition layer.

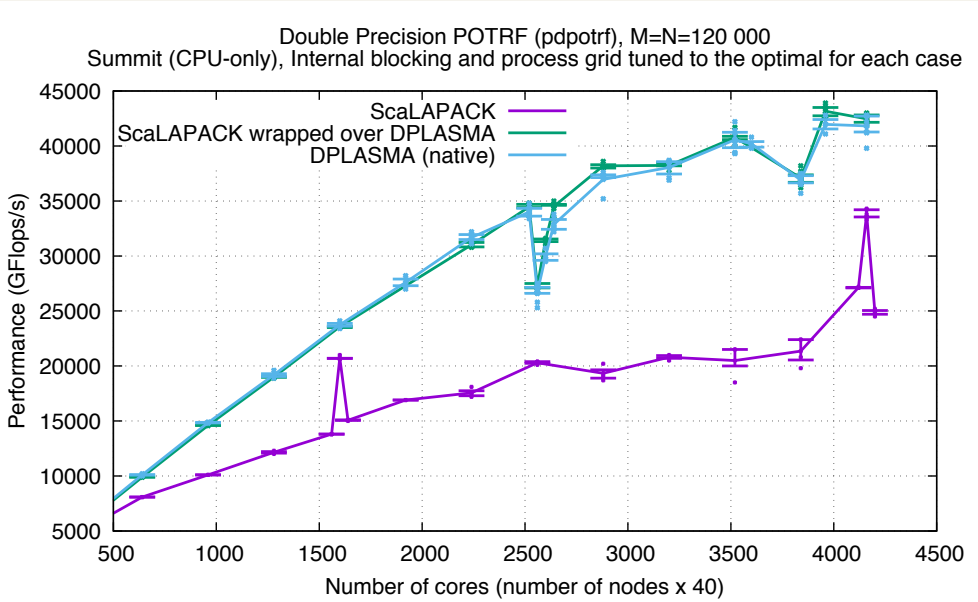
Minimal changes on the ScaLAPACK applications:

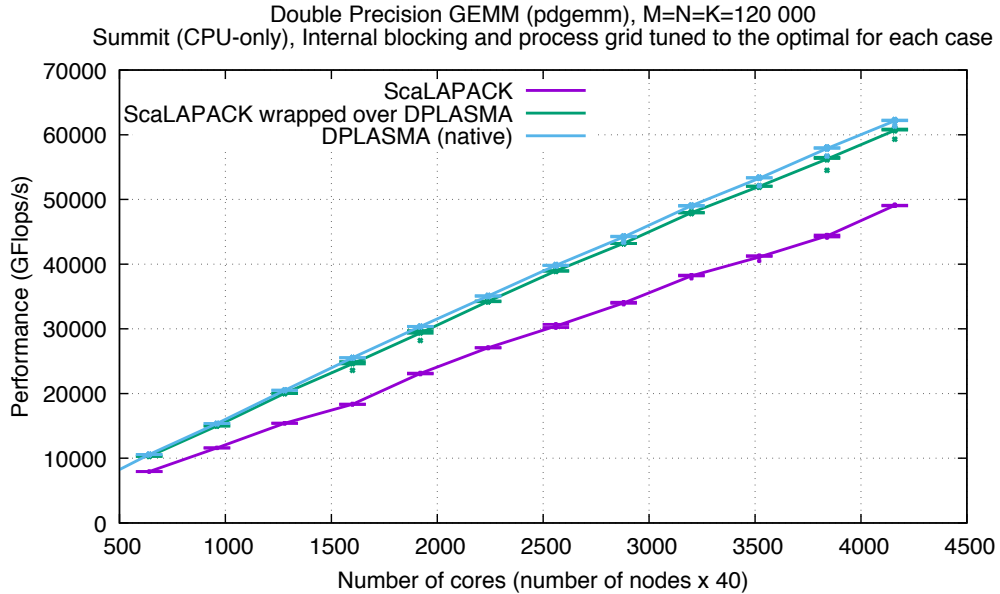
- Linking against wrappers transparently replaces the ScaLAPACK calls.
- Initialization and finalization calls at the beginning and end of the ScaLAPACK application.

Supporting the ScaLAPACK memory layout:

- No memory and performance overhead converting between ScaLAPACK and tiled (DPLASMA) memory layouts.

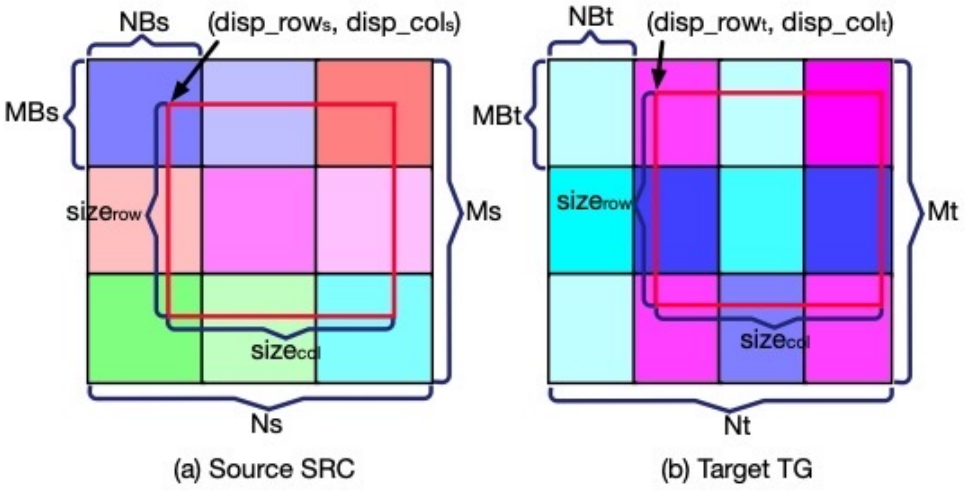
Functionality	Coverage
Linear Systems of Equations	Cholesky, LU (inc. pivoting, PP), and LDL (prototype)
Least Squares	QR and LQ
Symmetric Eigenvalue Problem	Reduction to Band (prototype)
Level 3 Tile BLAS	GEMM, TRSM, TRMM, HEMM / SYMM, HERK / SYRK, and HER2K / SYR2K
Auxiliary Subroutines	Matrix generation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

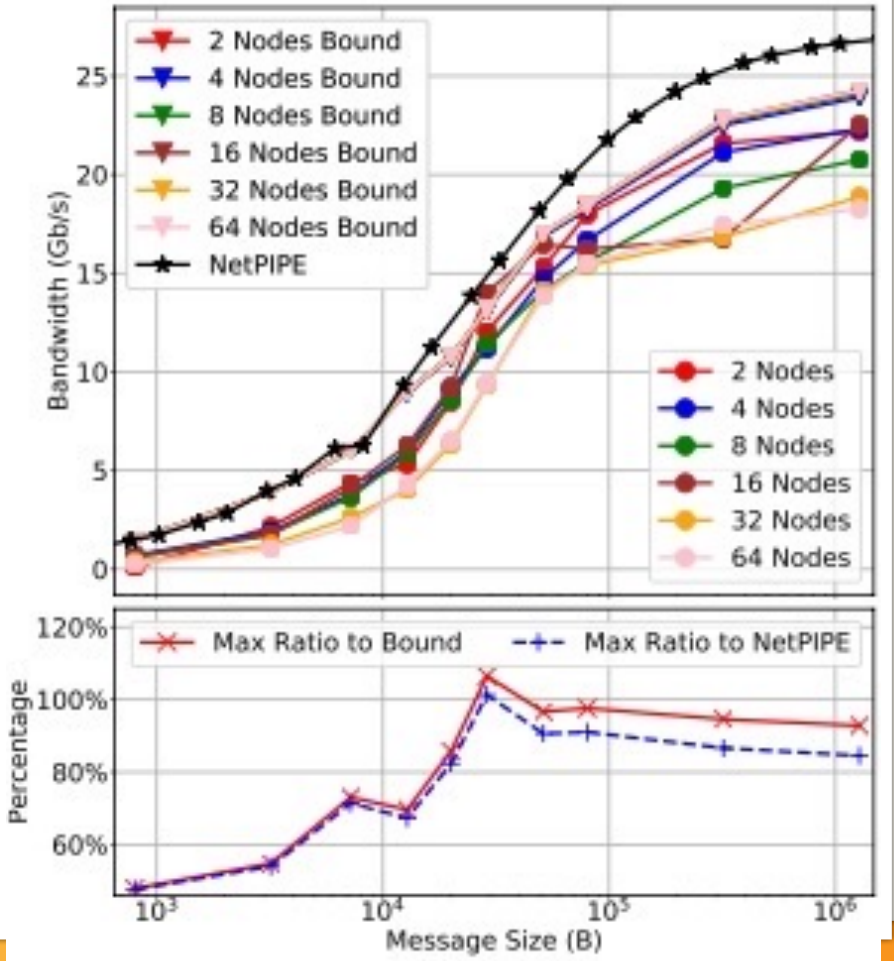




Data Redistribution

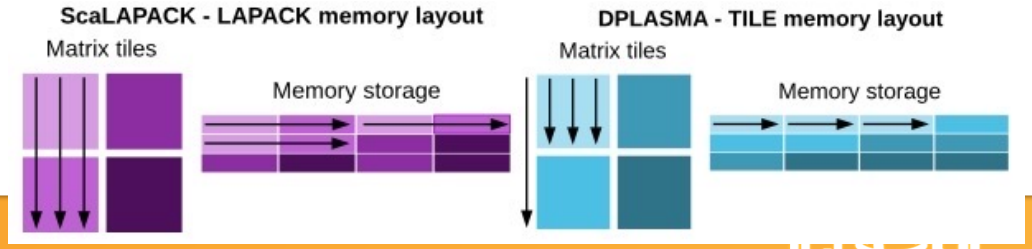
- A generic operation to change the shape and/or distribution of any multi-dimensional data distribution
  - Large number of communications require some form of ordering
  - How to inject messages into the network: multi-threaded vs funneled
    - Coordination message aggregation
    - Limit the inflight messages

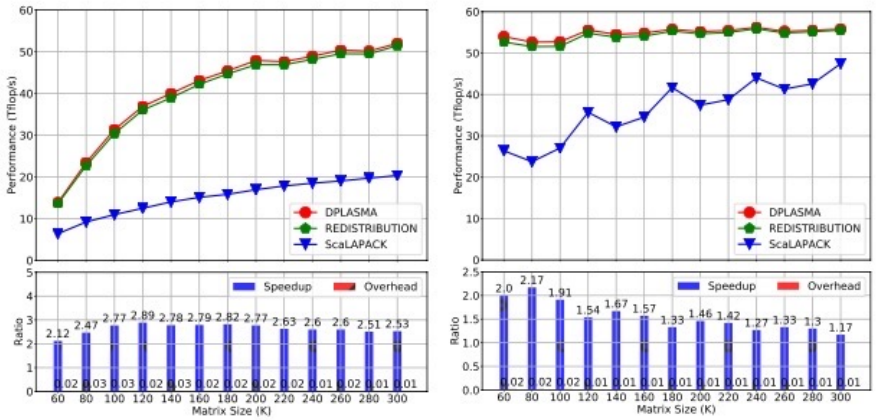




Data Redistribution

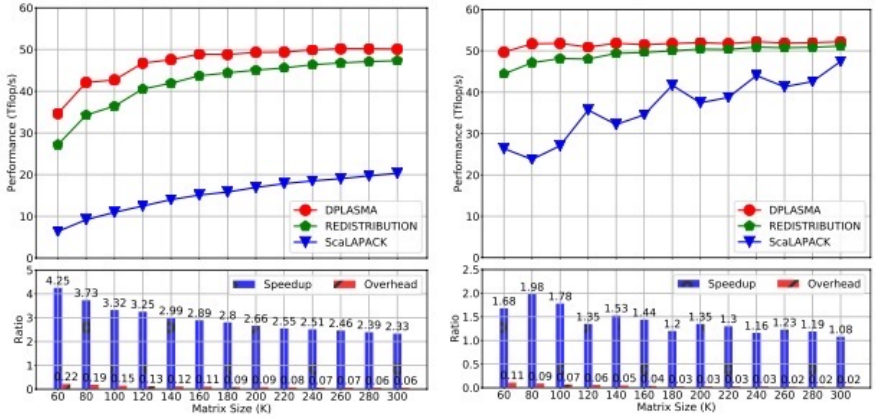
- Pull terminals: the operation is known to PaRSEC but will be delayed until needed
  - The definition of “needed” is left to the DSL/runtime
  - In these experiments we use “at least another input data is available”
    - Using DPLASMA with the ScaLAPACK API will require not only a redistribution but also a reshape to move from LAPACK format to a tile format





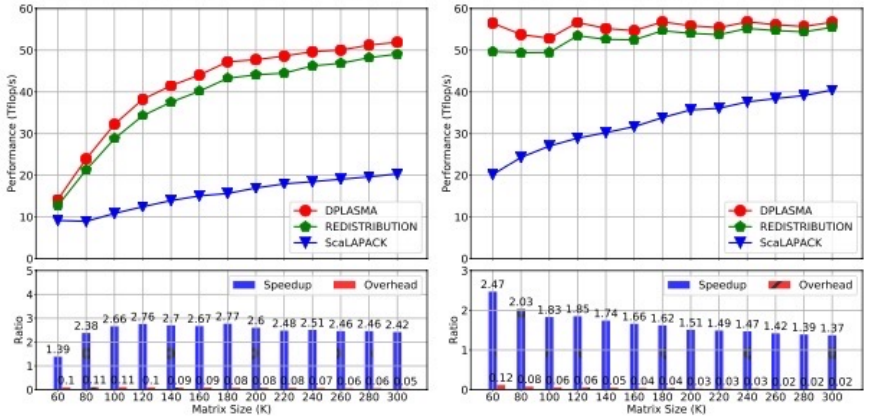
(a) Cholesky; 960 → 960.

(b) GEMM; 960 → 960.



(c) Cholesky; 960 → 400.

(d) GEMM; 960 → 400.

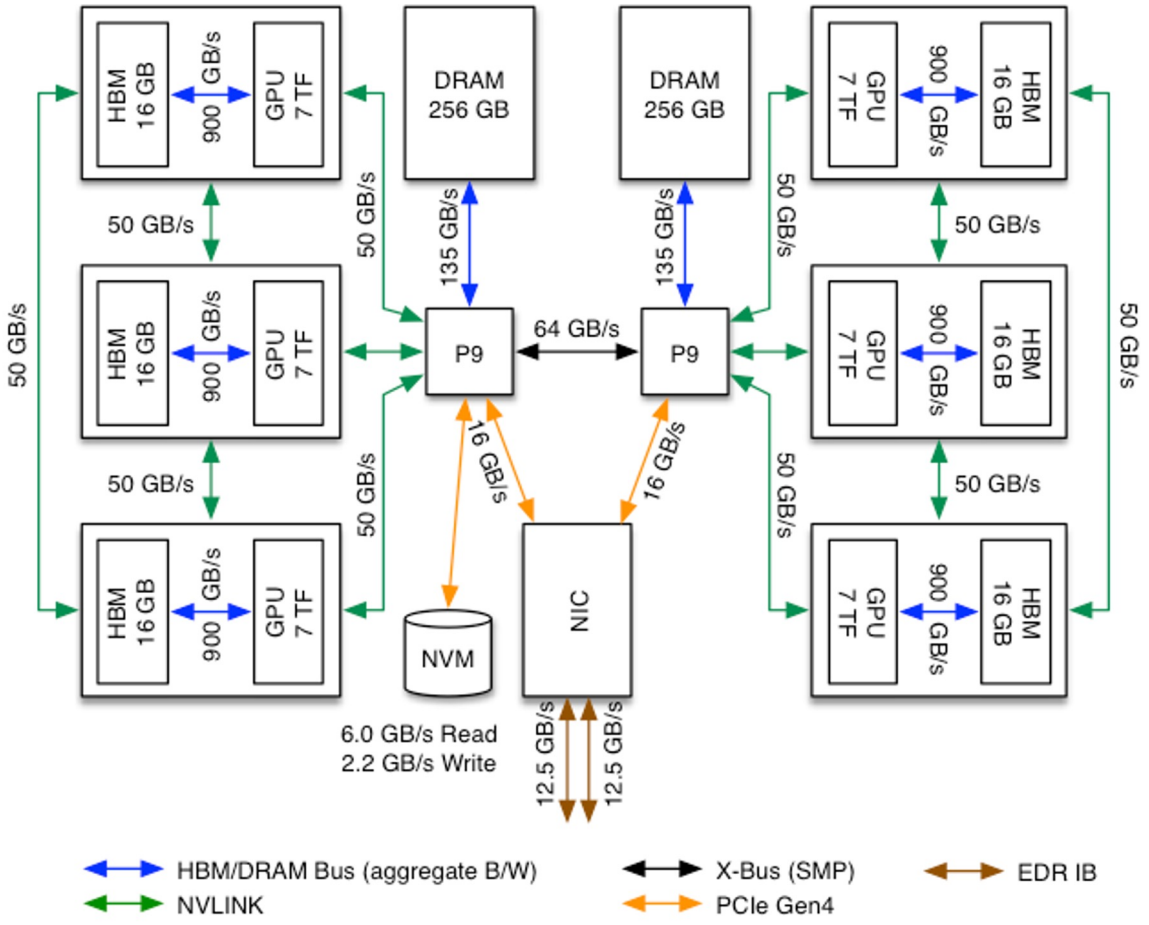


(e) Cholesky; 400 → 960.

(f) GEMM; 400 → 960.

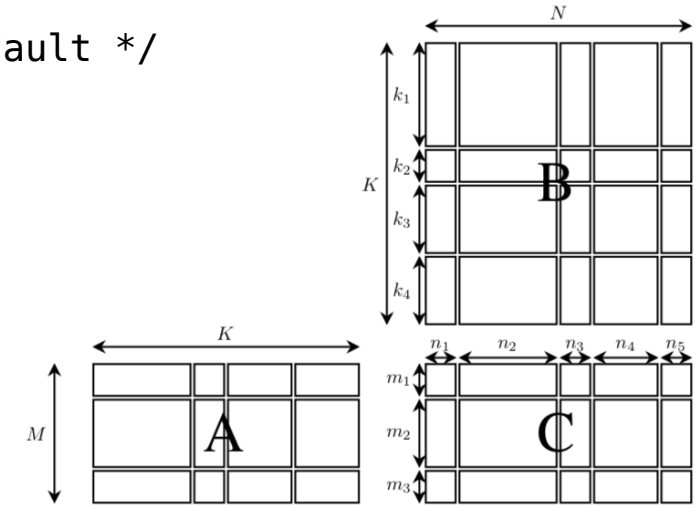
Redesigning PaRSEC multi-GPU support for Summit-like Architecture

- Runtime System Level:
  - Extend the PaRSEC API to provide memory affinity hints wrt GPUs
  - Enable exploitation of NVLINK
  - Extend the programming paradigms to support GPUs on all DSL, and allow definition of control flow on top of existing dataflow
  - Increase support of multiple threads to interact with the network in order to reach the NIC bandwidth
- Algorithmic level:
  - Increase control flows to pace data transfers on shared buses and avoid bus thrashing by controlling memory requirements on each GPU



ZGEMM: The PTG description

```
GEMM(m, n, k)
  m = 0 .. C->mt-1
  n = 0 .. C->nt-1
  k = 0 .. A->nt-1
 : C(m, n)

READ A <- A READ_A(m, k)
READ B <- A READ_B(k, n)
RW   C <- 0 == k           ? C(m, n) : C GEMM( m, n, k-1 )
       -> k == (A->nt-1)) ? C(m, n) : C GEMM( m, n, k+1 )

BODY [type = CPU]  /* default */
   zgemm( … );
END

BODY [type = CUDA
      dyld=cublasZgemm]
   body.dyld_fn( … );
END
```



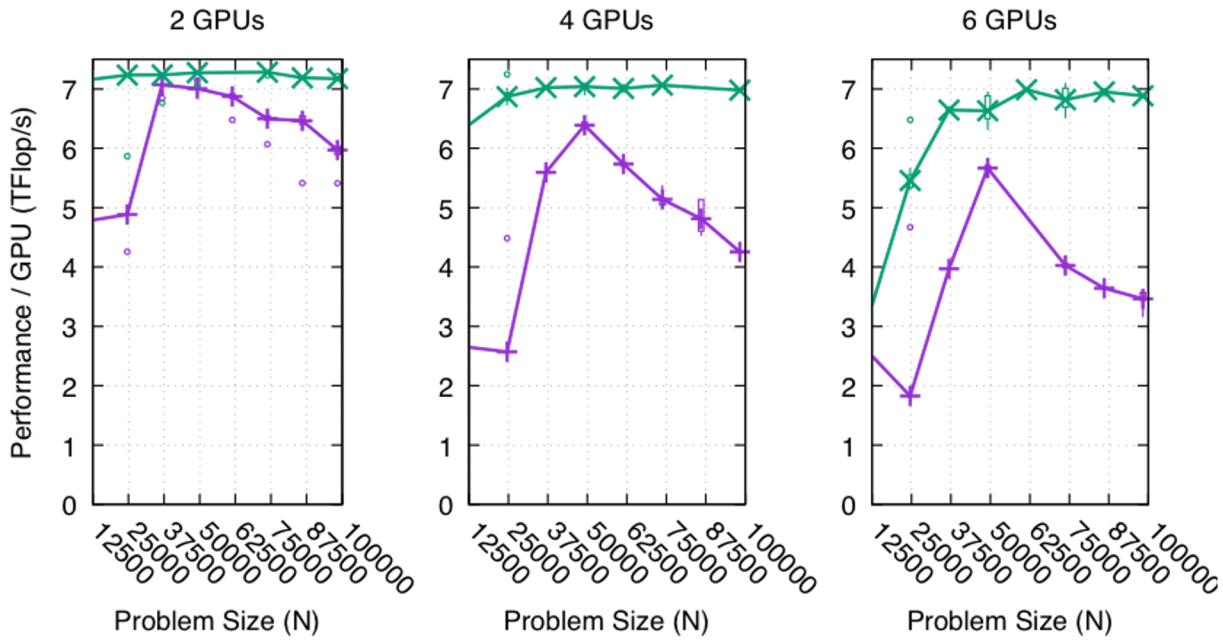
- A concise parameterized dataflow language, with non-dense iterators and extended expressions via inlined C/C++ code to augment the language
- Tasks can be specialized to target specific devices and refined to adapt to multiple granularities
- The good: there is NT*MT parallelism available, the runtime could maximize reuse
- The bad: there is NT*MT parallelism available, the runtime might maximize reuse
- The ugly: we need to direct the runtime to minimize transfers
- To this dataflow representation we superimpose a control flow description to describe a blocking approach similar to algorithm[1]

1. Anatomy of high-performance matrix multiplication, K. Goto and R. Van de Geijn, ACM TOMS, 2008

Redesigning PaRSEC multi-GPU support for Summit-like Architecture

DGEMM, Single Node, SUMMIT

- Added strict control flow to pace the movement of data over the PCI-Express bus and keep the active set within GPUs memory constraints
- Added Memory Advise capabilities to the PaRSEC GPU API to bind tiles on specific GPUs and force a fair load balance
- Added support of NVLink in multi-GPU environment in the PaRSEC data management system to reduce transfers on the PCI-Express bus

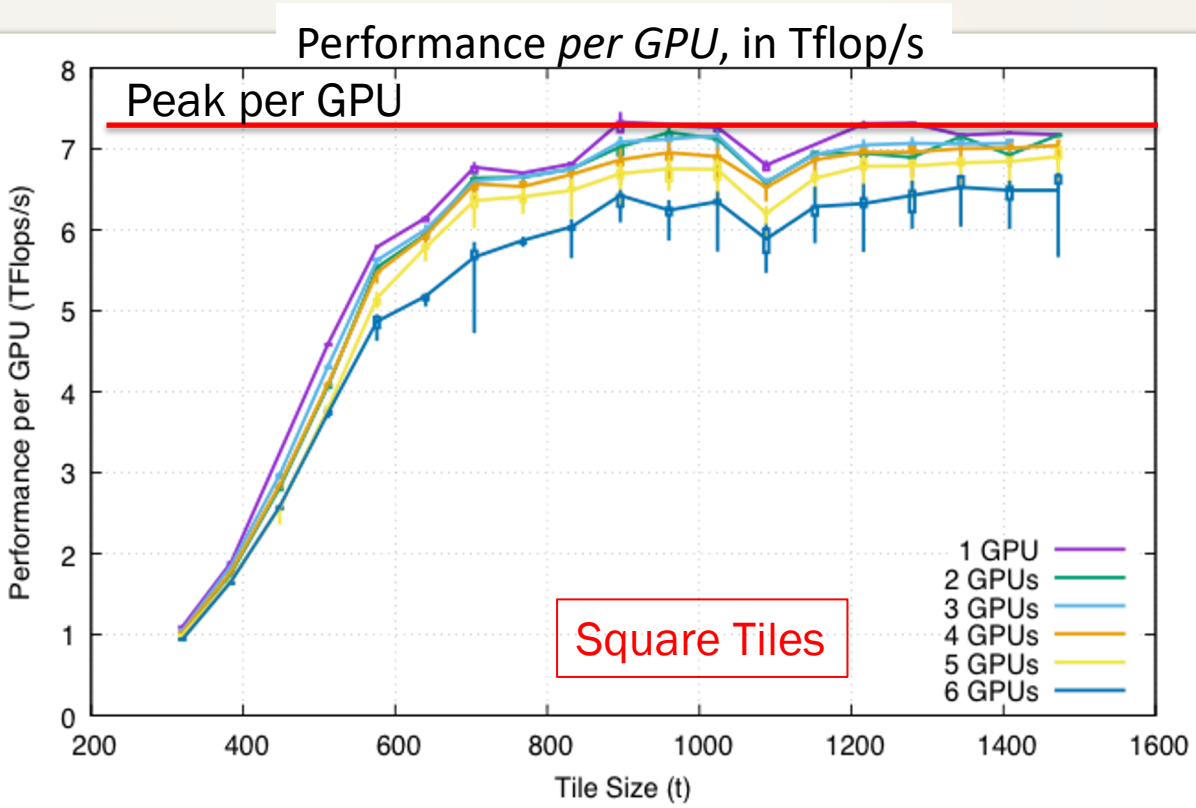


Without Transfer Control and NVLink support, when the problem size outgrows the (accumulated) GPU memory, the scheduler would start thrashing the PCI-Express bus by ejecting and re-requesting many tiles. Tight control flow and exploitation of NVLink allows to keep the performance close to peak and reach the peak for smaller problem sizes

Redesigning PaRSEC multi-GPU support for Summit-like Architecture

New PDGEMM over PaRSEC: Single SUMMIT node tuning

- Practical peak performance: 7.2TFlop/s per GPU
- Out-of-core: runs of problem of size $72k^2$. On 3 GPUs or less, not even a single matrix fits on the accumulated GPU memory: all problems are 'out-of-core'
- Tile size of 832^2 or 1024^2 (doubles) reach peak
- Strong scaling from 1-3 GPUs is near optimal

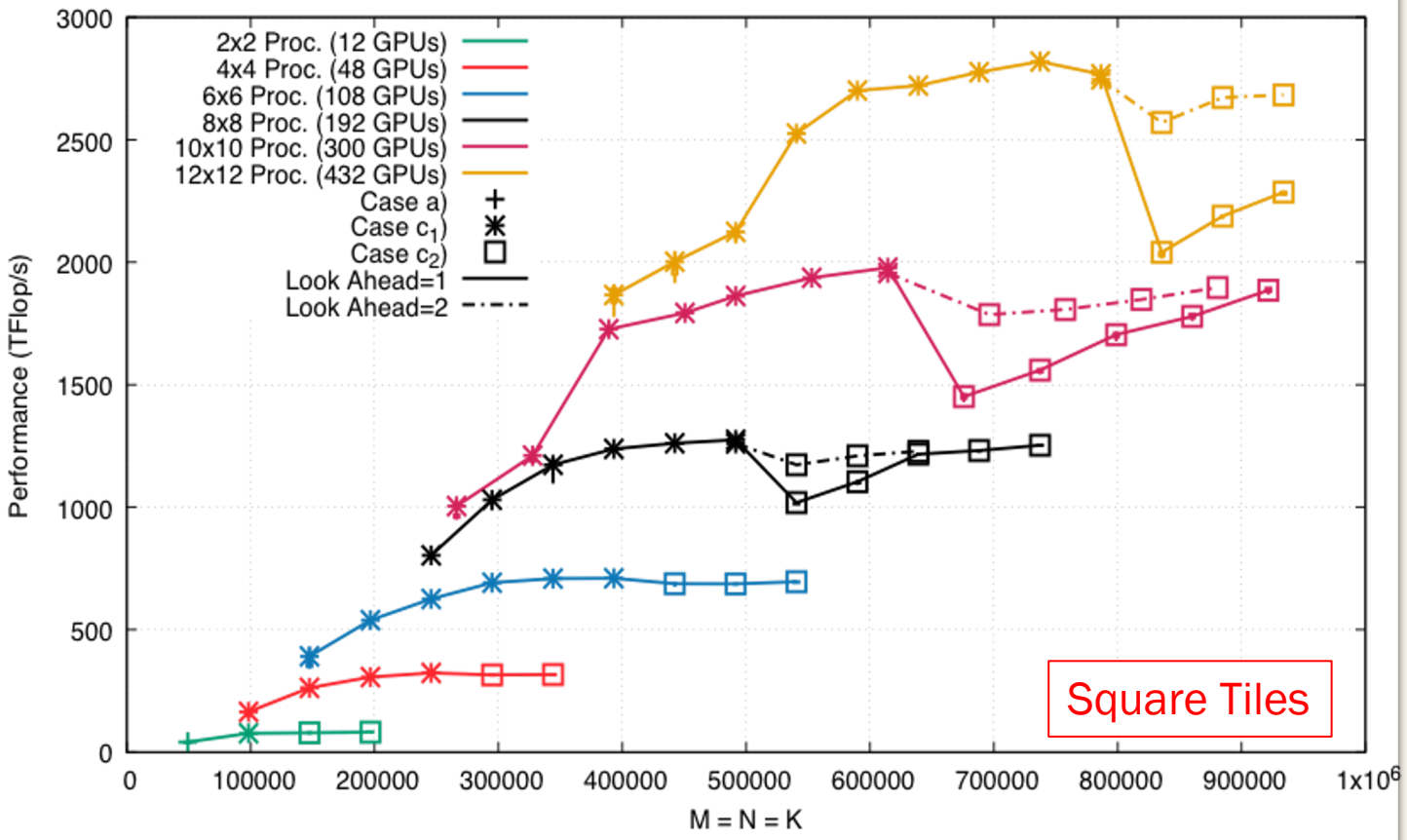


- NUMA effects / bus sharing impacts performance at 4-6 GPUs
  - Deployments with 1 process per socket / 3 GPUs per process should be preferred to address
  - Added benefit: such deployments are necessary to reach peak network bandwidth

Redesigning PaRSEC multi-GPU support for Summit-like Architecture

PDGEMM, SUMMIT

- Square GEMM on a square process grid (2 processes/node)
- Problem size:
  - **Case a)**: all three matrices fit in GPU memory
  - **Case c_1)**: C is distributed between the GPUs and local parts fit on the GPUs, A and B cycle from node to node and RAM to GPU memory
  - **Case c_2)**: no matrix fits entirely on GPU memory, A and B cycle from node to node, and A, B, and C cycle from RAM to GPU memory

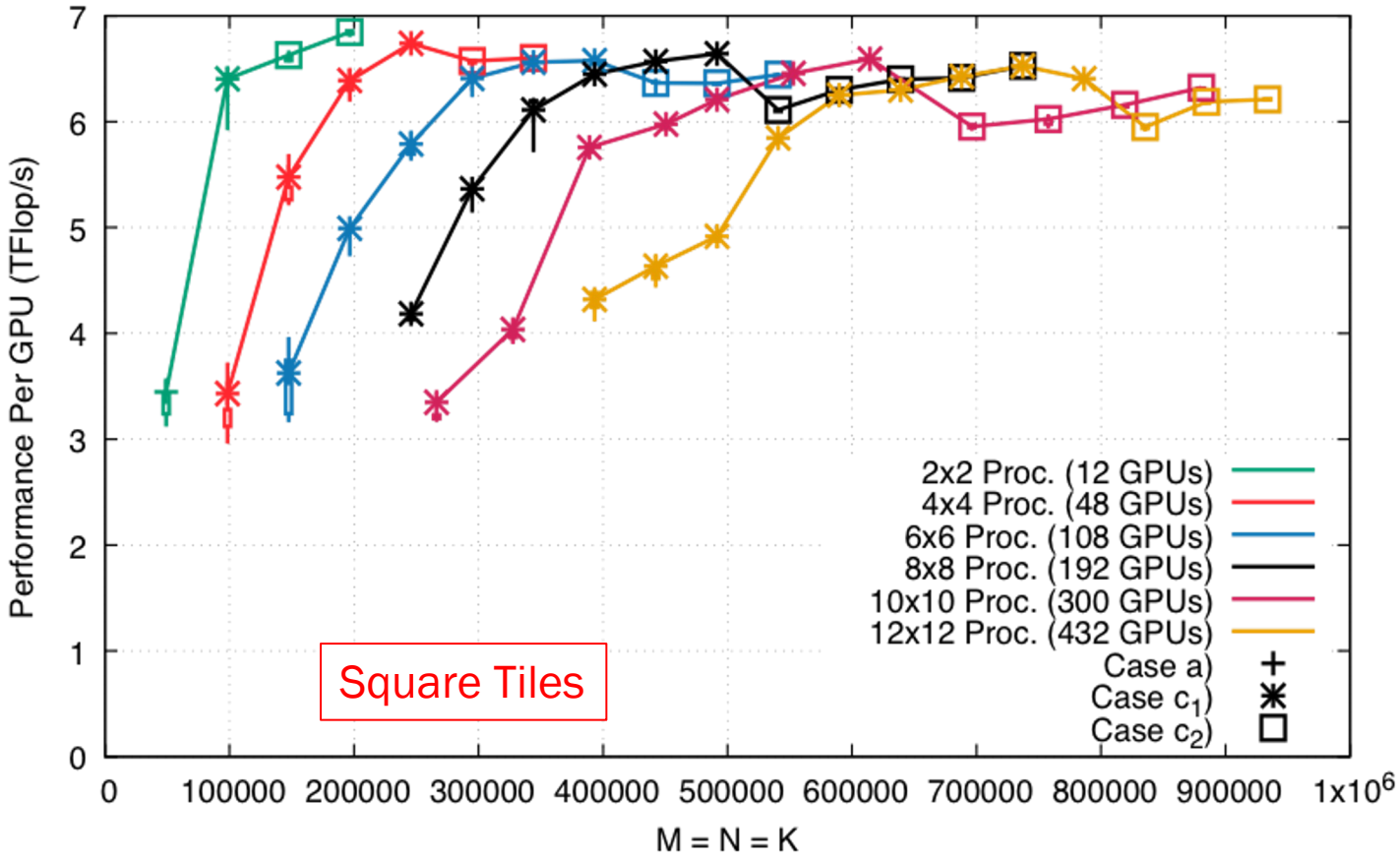


- At 64 processes or more, going from case c_1) to case c_2) introduces performance degradation without lookahead: when changing the part of C that is updated, the network receives a surge of tiles requests for A and B, that can be automatically pre-fetch by the runtime system by allowing a look-ahead in the algorithm

Redesigning PaRSEC multi-GPU support for Summit-like Architecture

PDGEMM, SUMMIT

- Square GEMM on a square process grid (2 processes/node)
- Problem size:
  - **Case a)**: all three matrices fit in GPU memory
  - **Case c_1)**: C is distributed between the GPUs and local parts fit on the GPUs, A and B cycle from node to node and RAM to GPU memory
  - **Case c_2)**: no matrix fits entirely on GPU memory, A and B cycle from node to node, and A, B, and C cycle from RAM to GPU memory

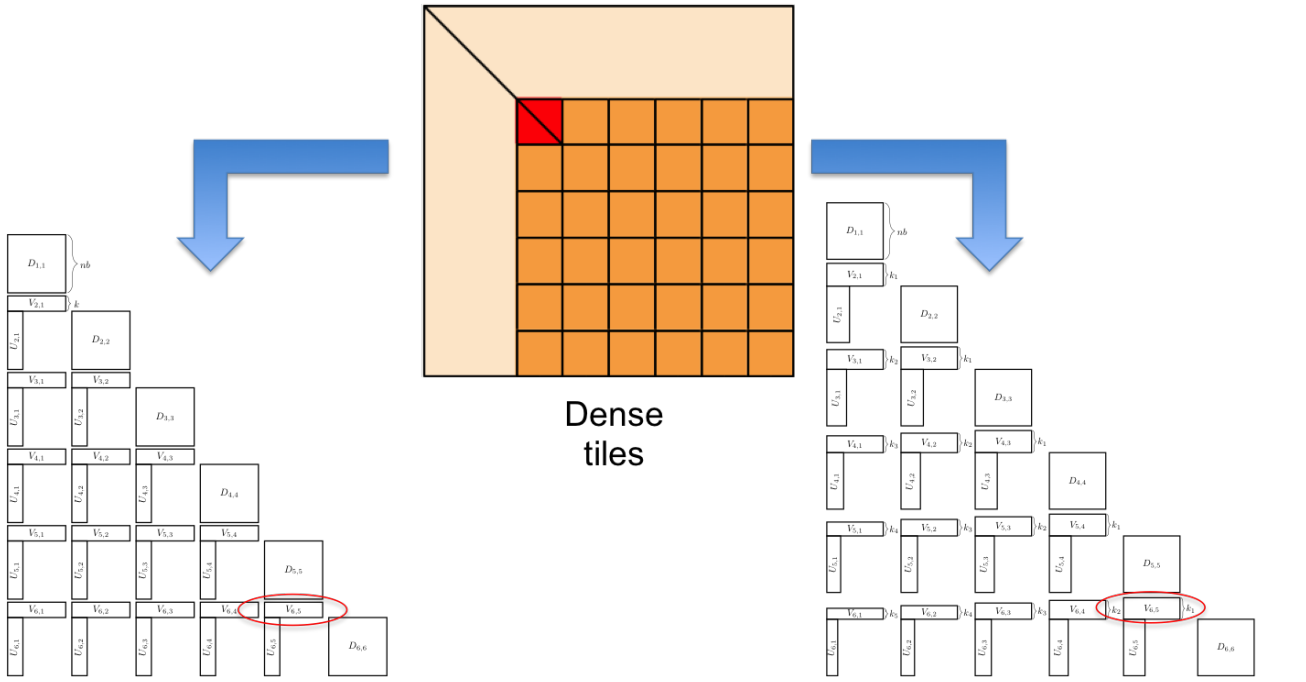


- At 64 processes or more, going from case c_1) to case c_2) introduces performance degradation without lookahead: when changing the part of C that is updated, the network receives a surge of tiles requests for A and B, that can be automatically pre-fetch by the runtime system by allowing a look-ahead in the algorithm

Drifting away from dense linear algebra

- PaRSEC separates data distribution and the dataflow but the algorithm remains the same
  - We need to change the data collections to allow for multiple hybrid data representations in the same collection, support non-regular the data distribution, have more flexible kernels to execute and deal with what travels through the dataflow edges
  - We don’t need to rewrite the Cholesky factorization PTG
- The challenges for the runtime:
  - A totally different critical path
  - A different computational balance
- The solution:
  - A versatile lookahead strategy
  - A nested Block Cyclic Data distribution to evenly distribute computationally intensive tasks among the most computationally capable resources (aka. Accelerators)
  - A more dynamic scheduling to rebalance the work

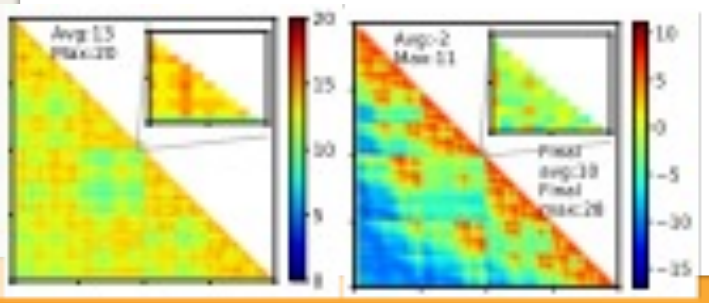
HiCMA: Low Rank mixed-precision Cholesk



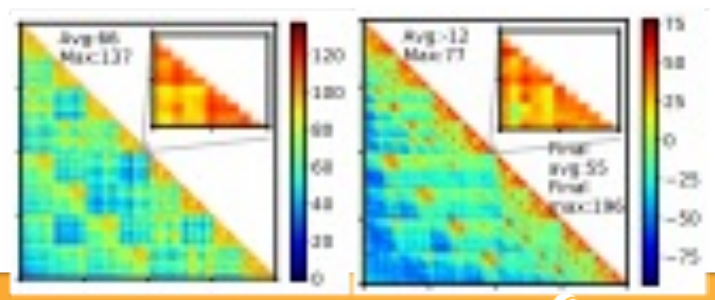
Fixed ranks
Preconditioners
Performance oriented

Fixed accuracy
Variable ranks
Dense/Sparse Direct Solvers

- Low rank change the balance of the algorithm
  - Closer to diagonal the tiles are dense
  - Tiles are larger to maintain accuracy for the low rank (so diagonal tiles are very expensive)
- Implement recursive operations for the dense tiles
- Look-ahead remains meaningful, but need to become adaptive
- Tiling size has impact not only on performance but also accuracy



2D sq-exp



3D sq-exp

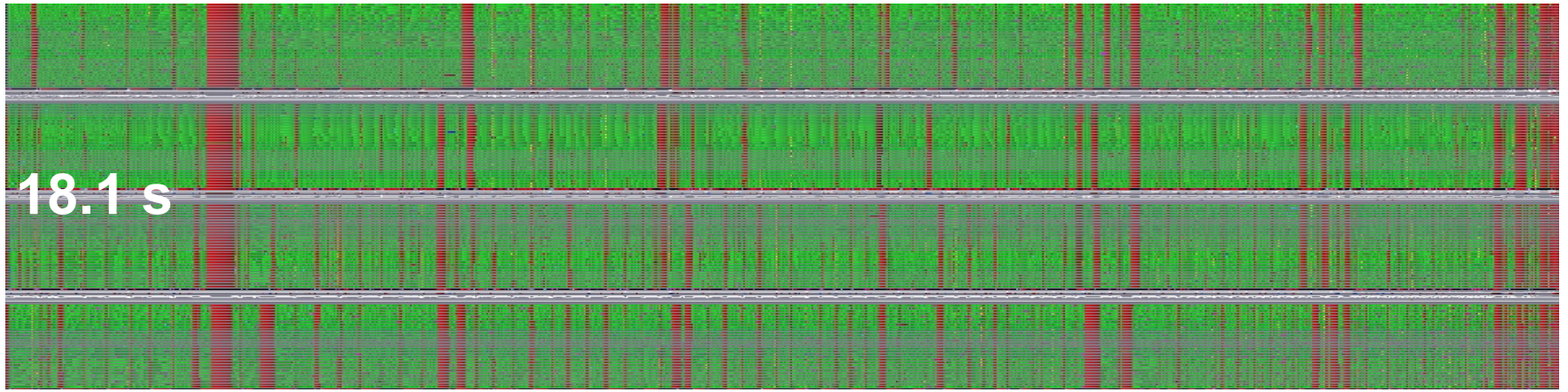
Kernel	Dense Cholesky	TLR Cholesky
POTRF	1/3 * nb^3	1/3 * nb^3
TRSM	nb^3	nb^2 * rank
SYRK/LR_SYRK	nb^3	2 * nb^2 * rank + 4 * nb * rank^2
GEMM/LR_GEMM	2 * nb^3	36 * nb * rank^2
Total	O(N^3)	O(N^2 * rank)

Our journey began in 2018 with these time traces…

… for factorization of a dense 54K covariance matrix on four 32-core nodes of Shaheen-2

Dense Tile-based Cholesky factorization (Chameleon)

cores



time →

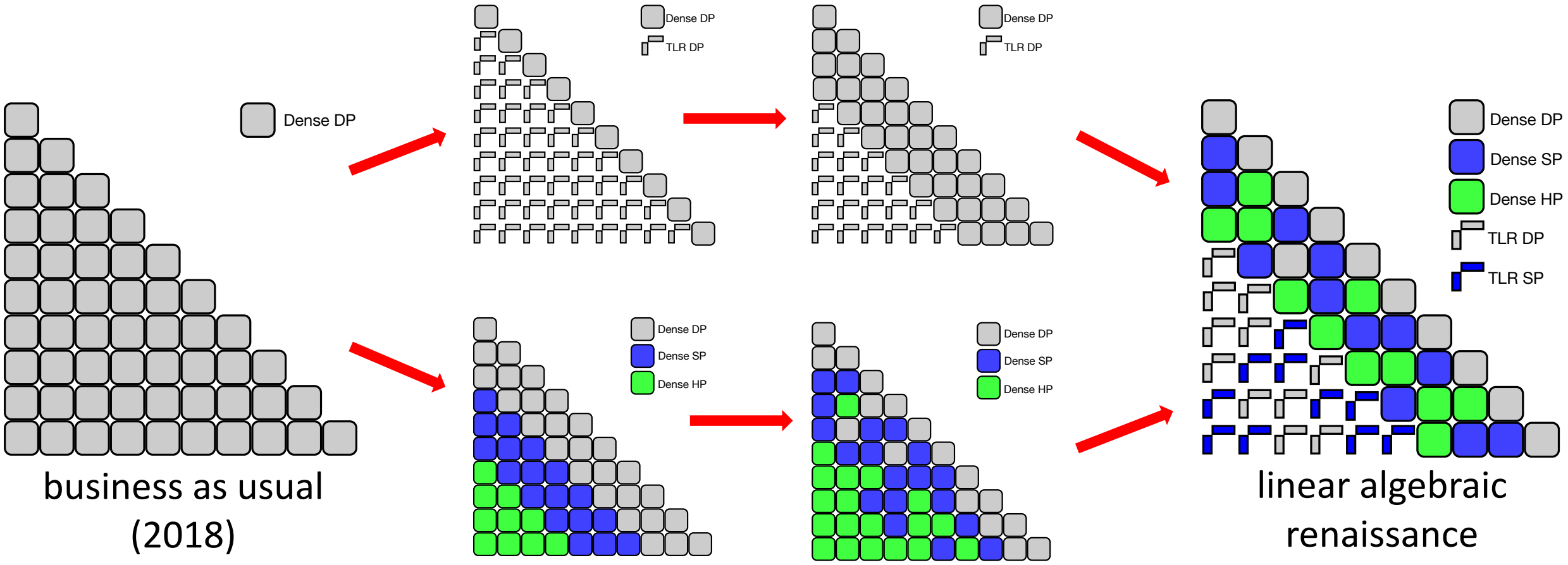
Tile low rank (TLR) Cholesky factorization (HiCMA)

cores

- TLR has a higher percentage of idle time (red) vs. computation (green), relative to flop-intensive dense
  → scales less efficiently (less able to cover data motion with computation)
- TLR scores a lower percentage of peak after squeezing out flops
- TLR is, however, **10X superior in time** for required application accuracy, at **about 65% of average power** compared to dense

Akbudak, Ltaief, Mikhalev, Charara & K., *Exploiting Data Sparsity for Large-scale Matrix Computations,* Euro-Par 2018

Journey followed two lanes, today merged for GB’22



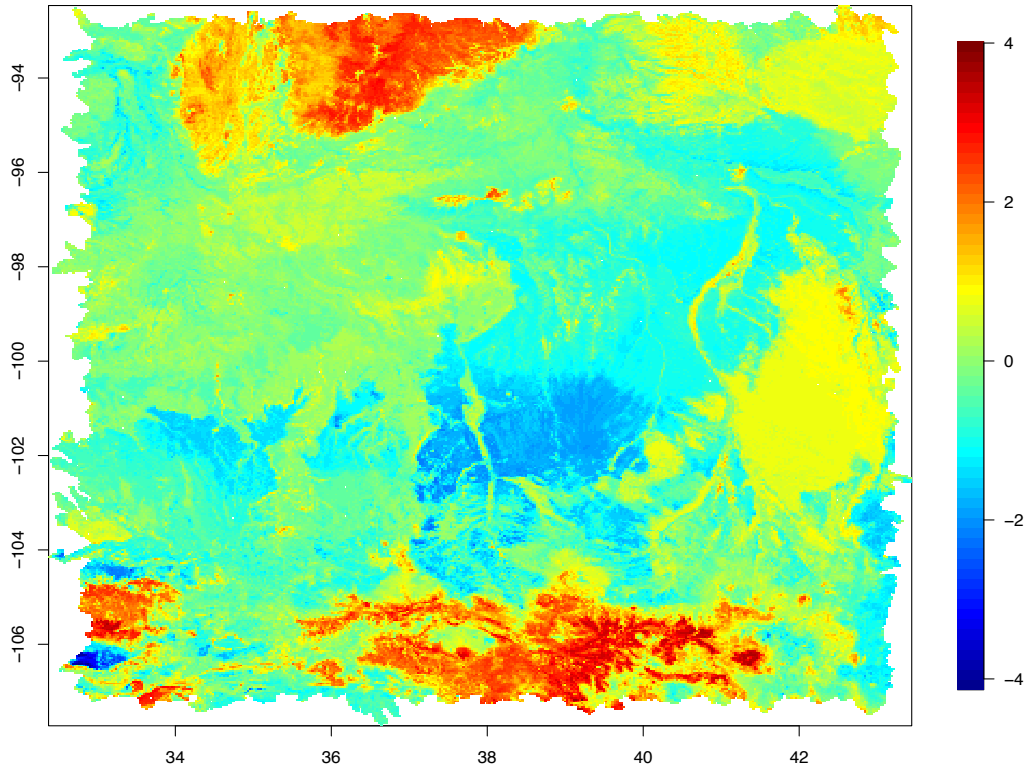
business as usual (2018)

linear algebraic renaissance (2022)

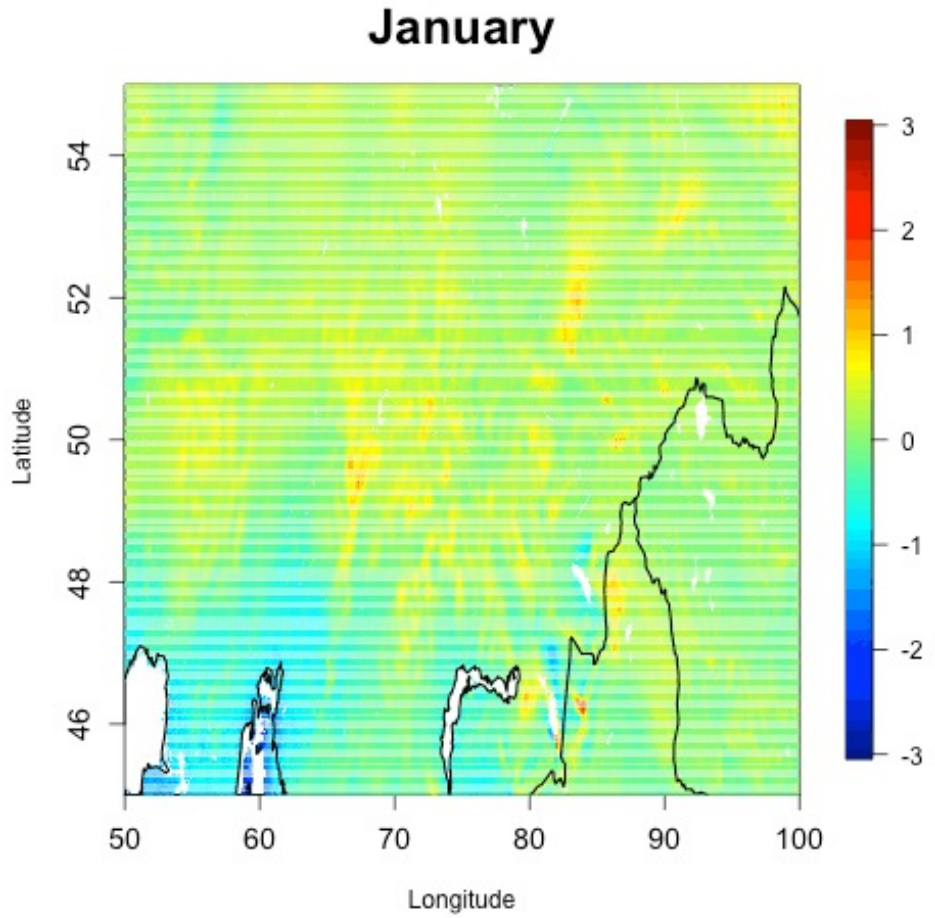
Don’t over-solve: maintain just enough accuracy for the application purpose

Don’t over-store: no extra copies of the original matrix

Application: geostatistics (spatial & spatio-temporal)

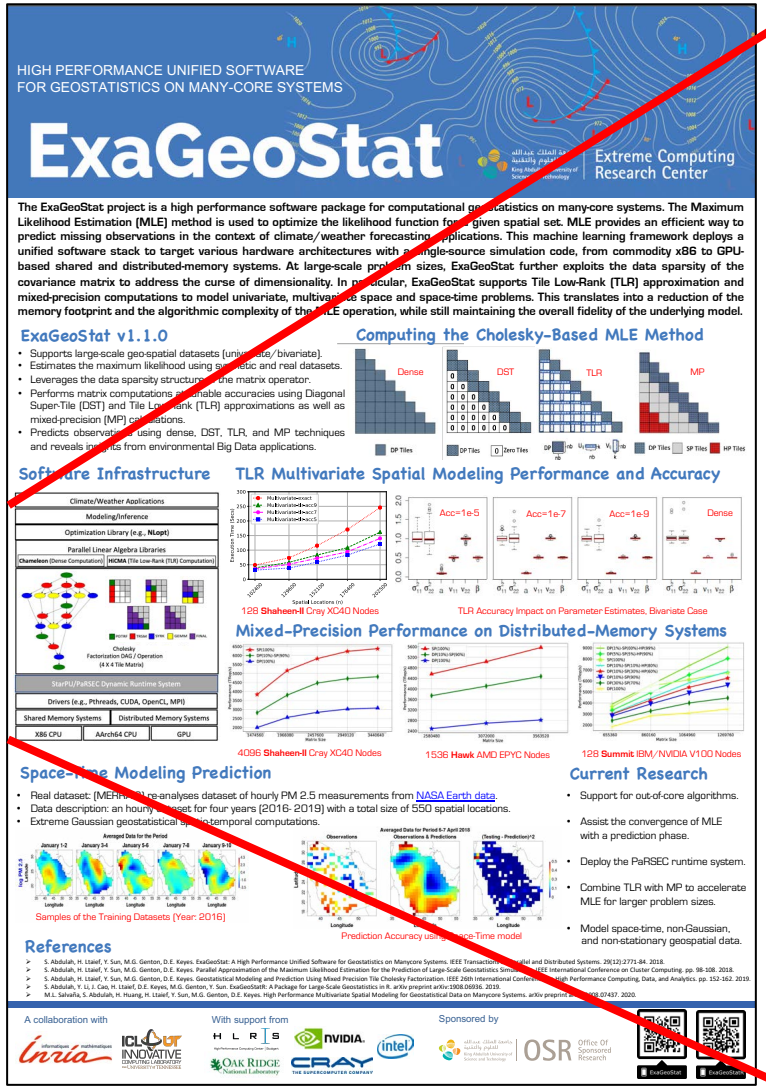


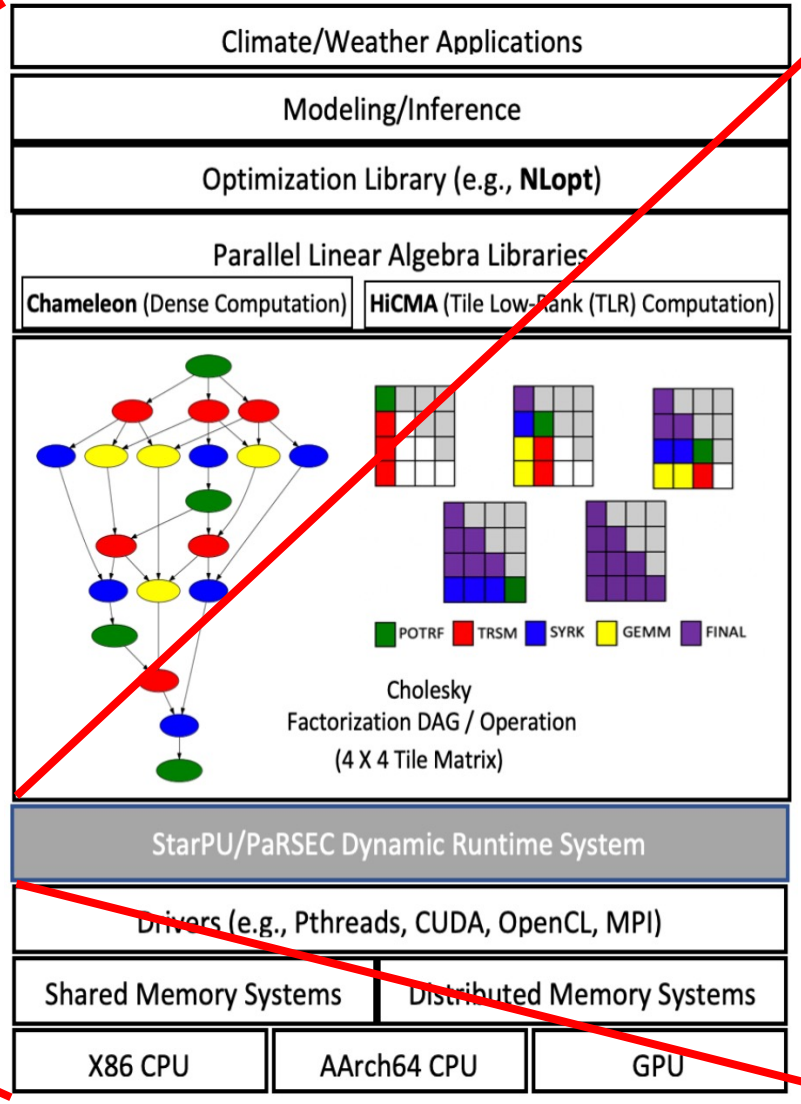
2D soil moisture
at top layer of the Mississippi River basin

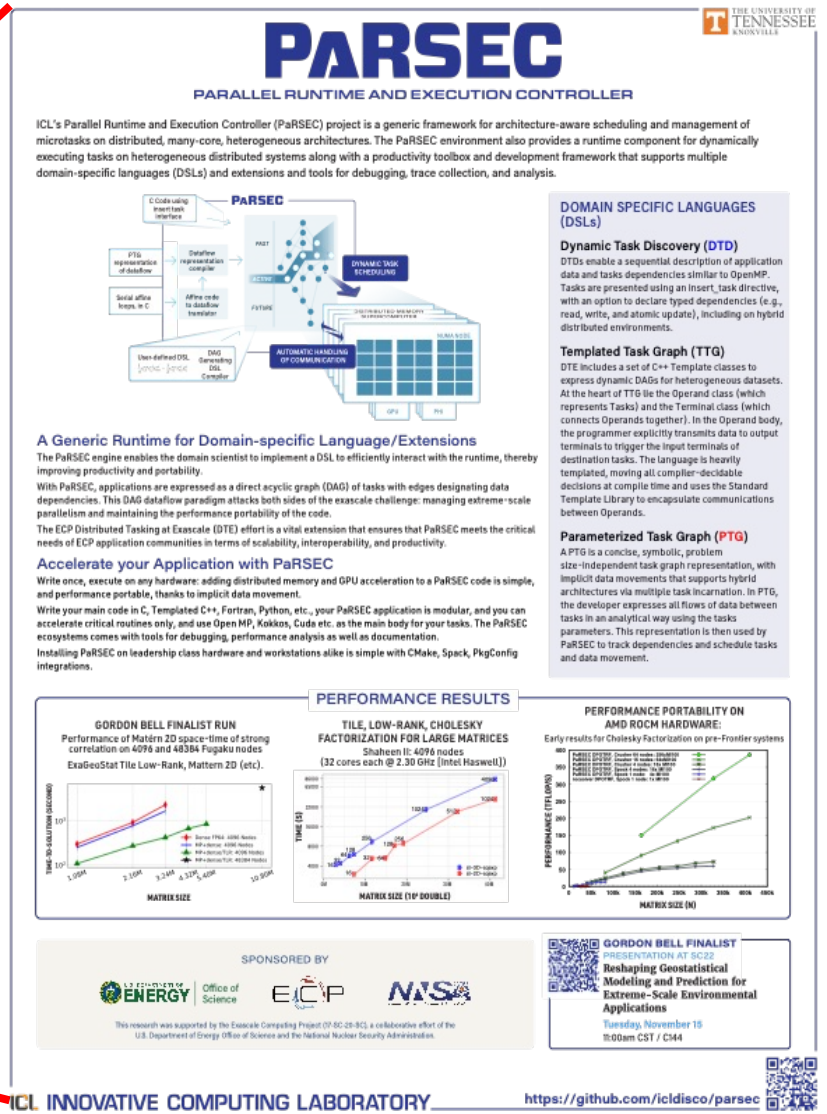


2D plus time evapotranspiration
over Central Asia monthly in 2021

Software @ github: ExaGeoStat (KAUST) and PaRSEC (UTK)







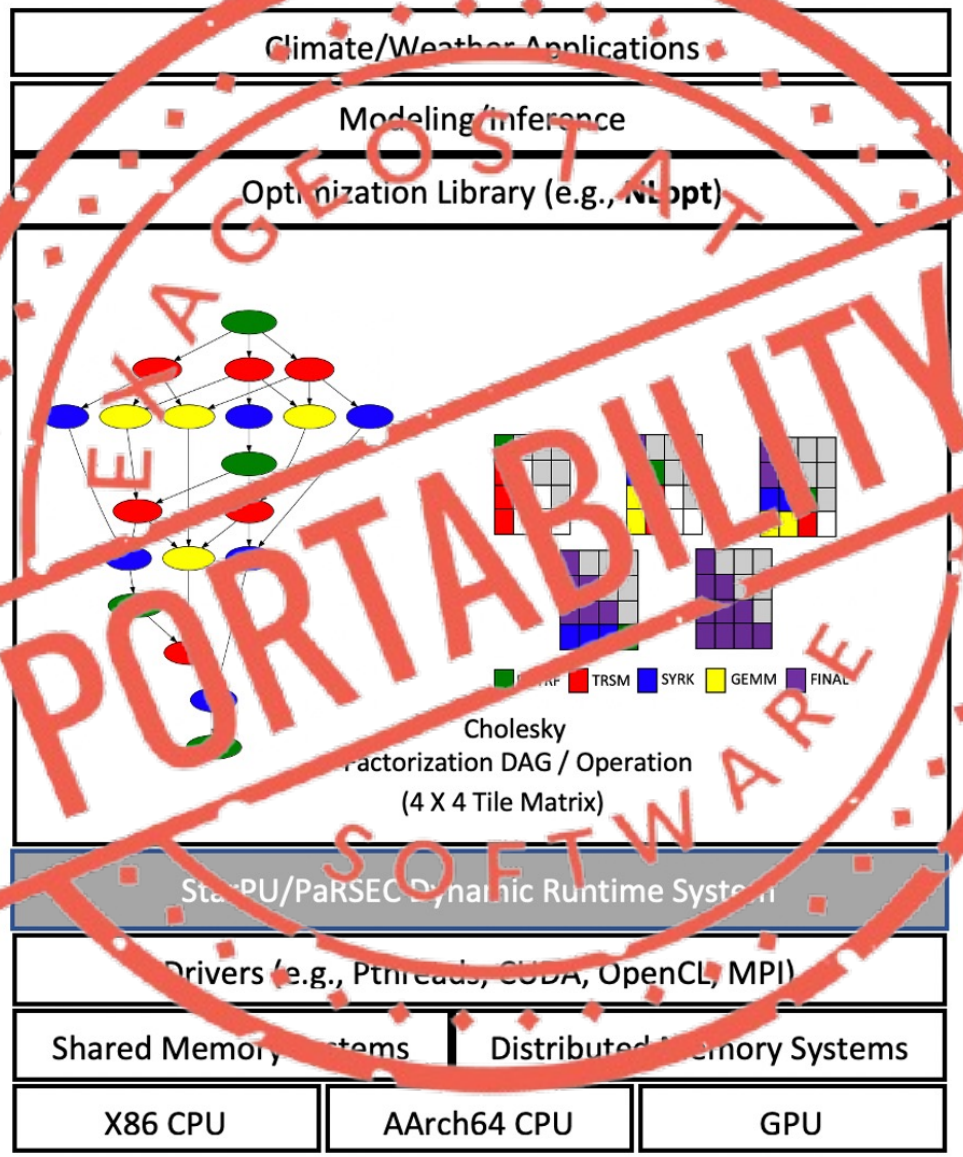
A portable full-app software stack

Fujitsu A64FX

NVIDIA V100

AMD EPYC

Intel X86



#2 Fugaku

#5 Summit

#30 HAWK

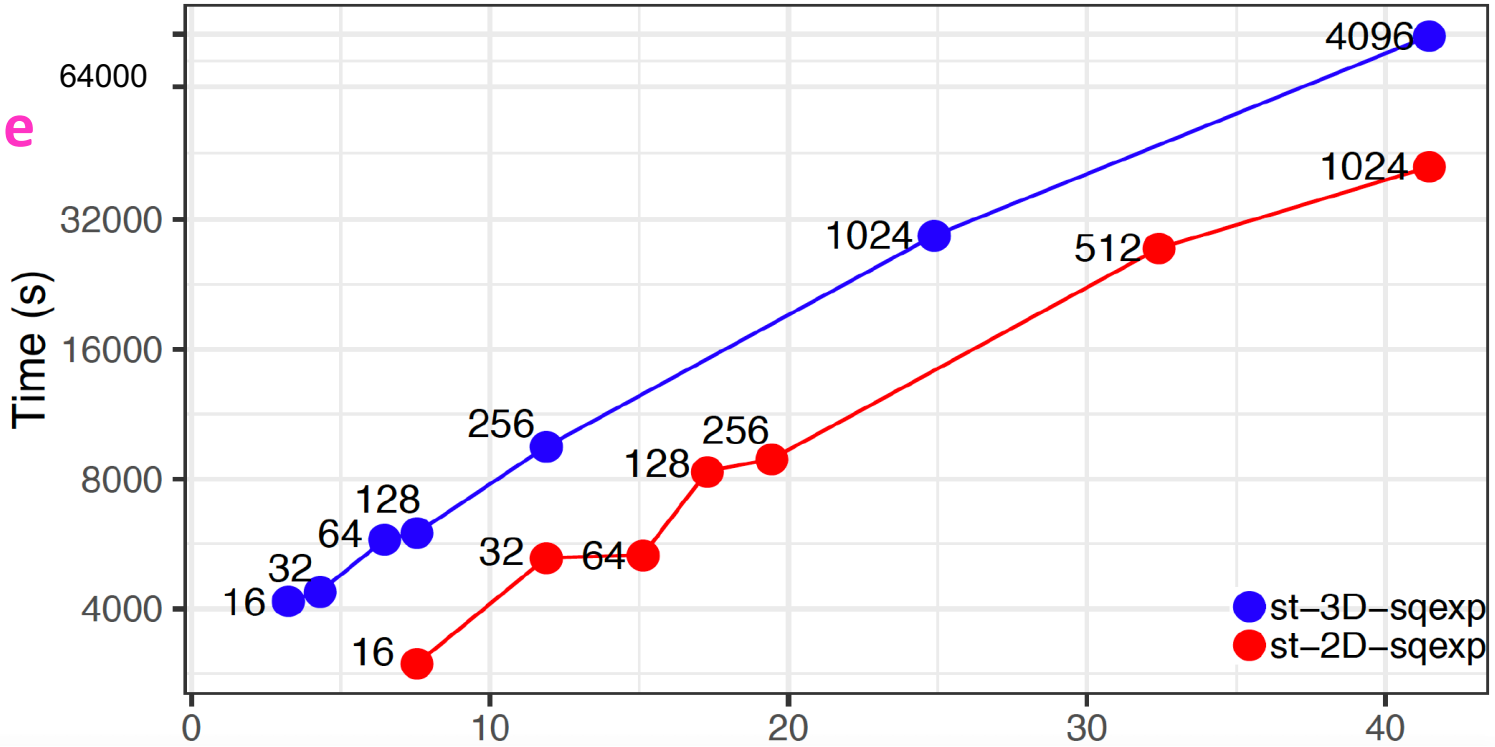
#104 Shaheen-2

Extreme example of the TLR advantage

Cholesky factorization of a TLR matrix derived from Gaussian covariance of random distributions, up to 42M DOFs, on up to 4096 nodes (131,072 cores) of a Cray XC40

- would require 7.05 PetaBytes in dense DP (using symmetry)
- would require 77 days by ScaLAPACK (at the TLR rate of 3.7 Pflop/s)

NB: log scale

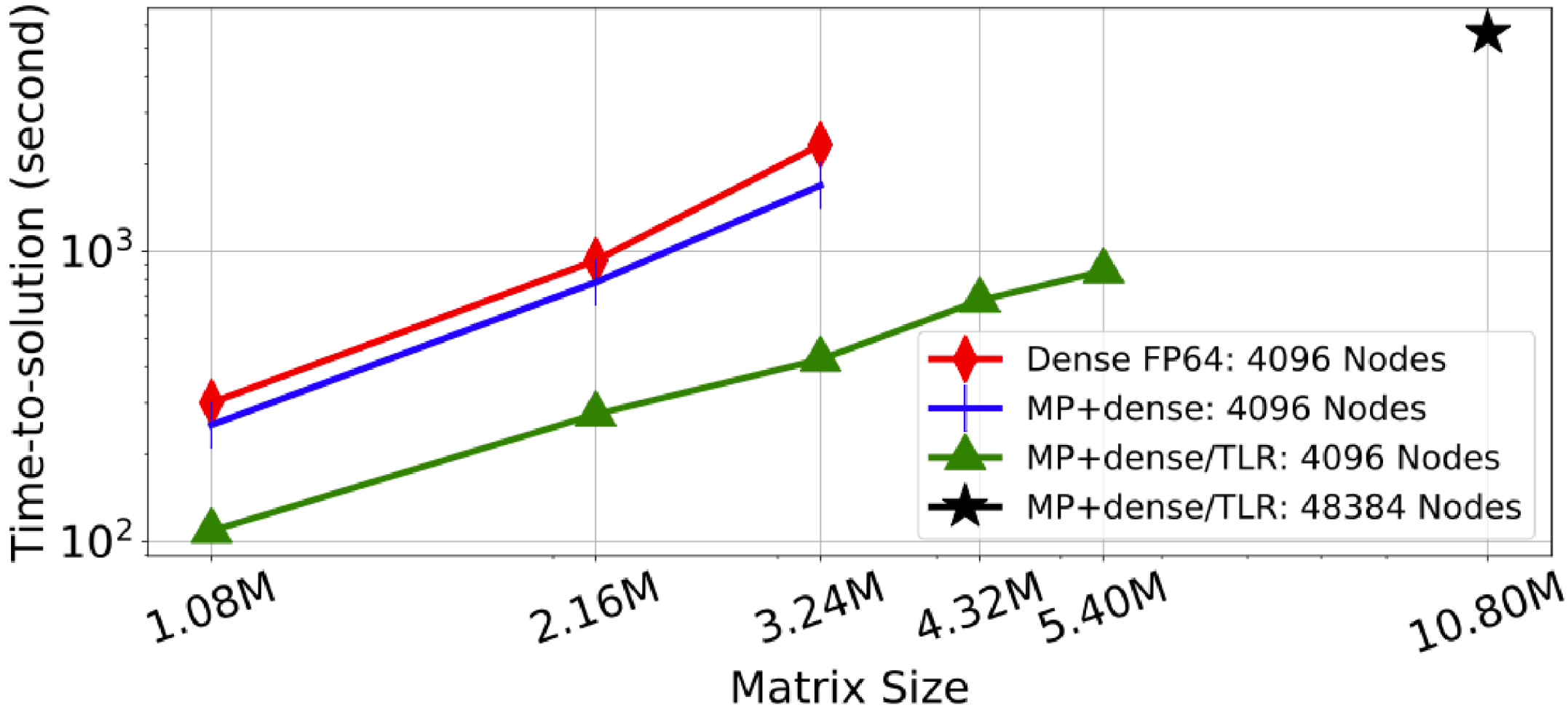


Fully dense computation would have cost about $1.03M in electricity and generated about 2500 metric tons of CO2e

Cao, Pei, Akbudak, Mikhalev, Bosilca, Ltaief, K. & Dongarra, *Extreme-Scale Task-Based Cholesky Factorization Toward Climate and Weather Prediction Applications*. PASC'20 (ACM)

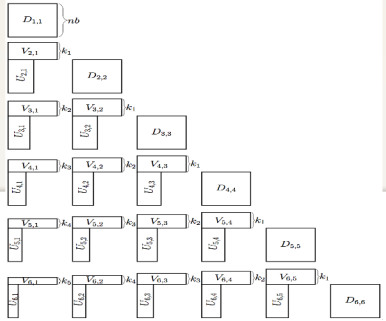
Scaling up to 48K nodes of Fugaku, 2D space+time

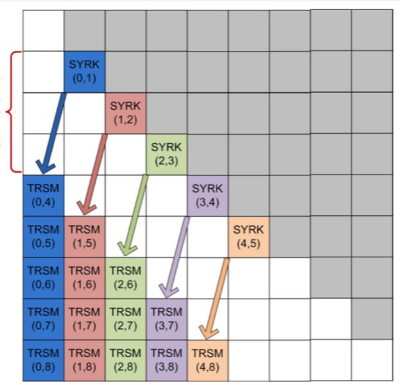
Largest Fugaku run: TLR/MP on 48,384 nodes (2,322,438 cores) for 10.8M matrix

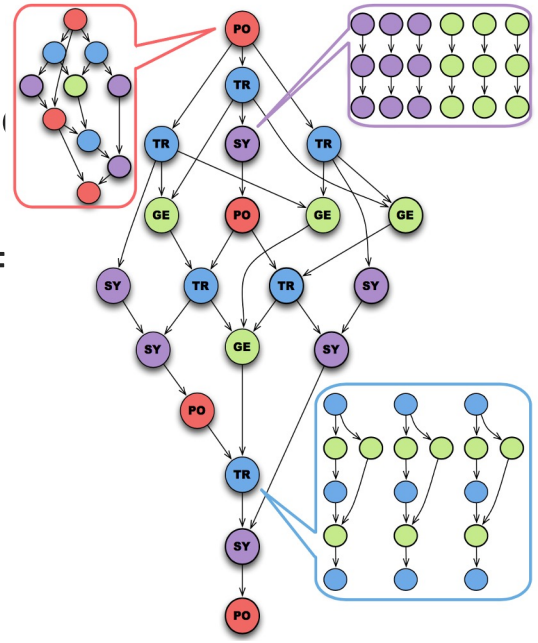


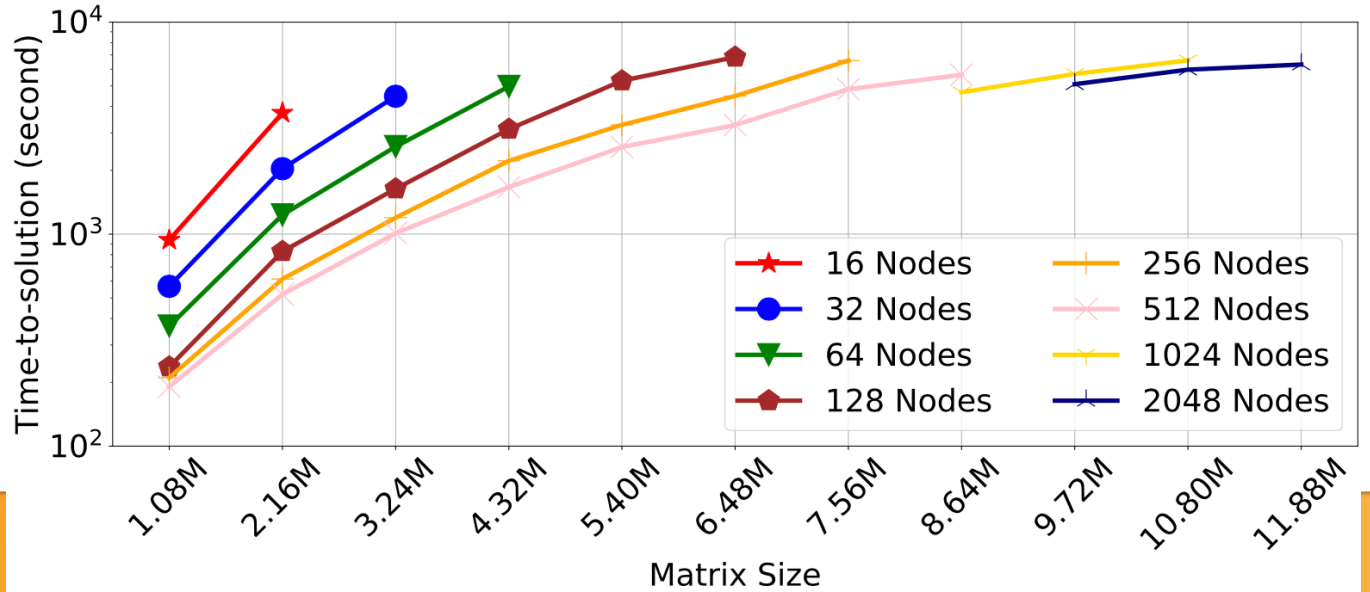
HiCMA - KAUST

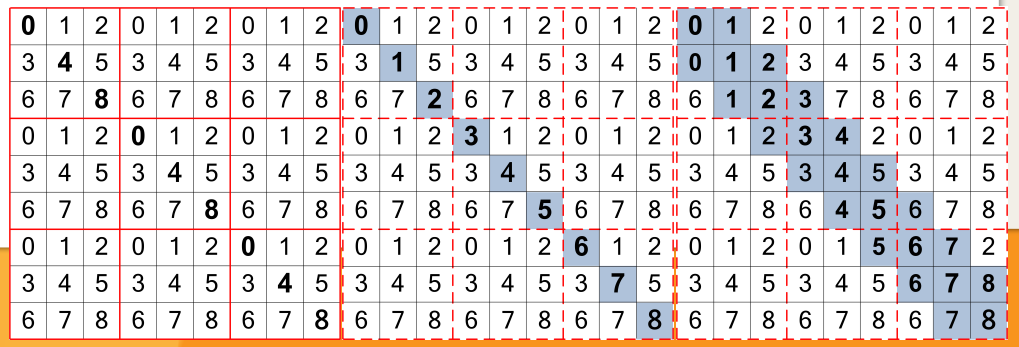
- TLR Cholesky Factorization
- Optimizations
  - Dynamic communications (rank can vary but communications remains
  - Dynamic memory reallocation leading to larger problems
  - No left or right looking, but flexible lookahead based on the rank (thus f
  - Hierarchical operations: tasks too dense can be divided on the flight



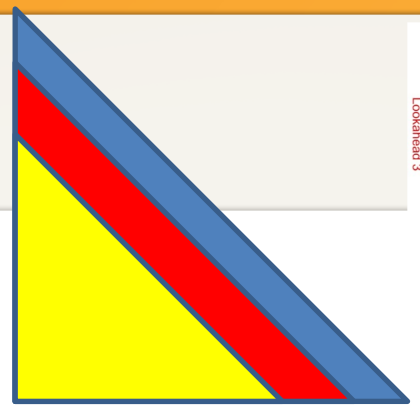


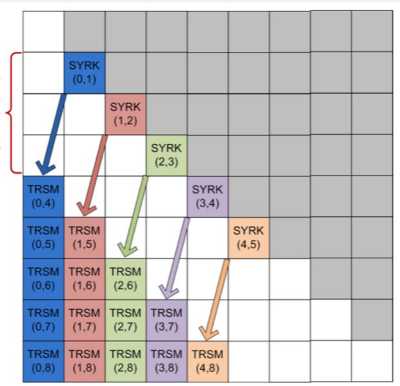




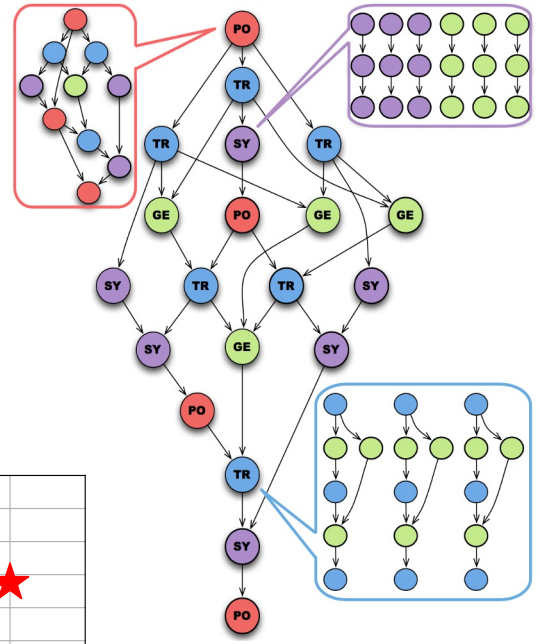


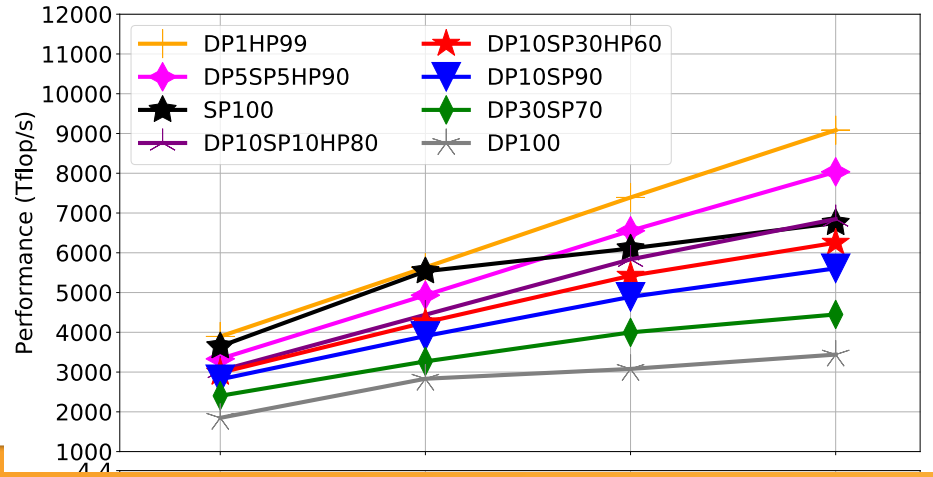
HiCMA - KAUST



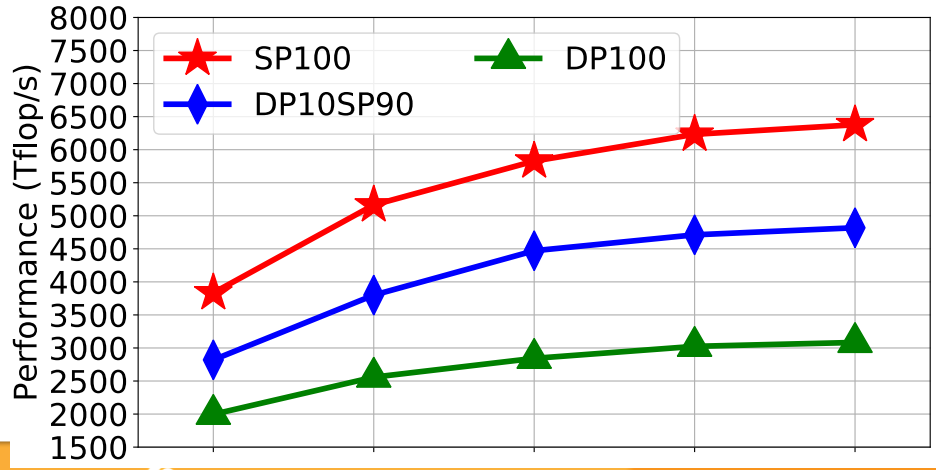


- Mixed-precision Cholesky
- Optimizations
  - Dynamic communications (tiles have different representations)
  - Dynamic memory reallocation leading to larger problems
  - No left or right looking, but flexible lookahead (compute intensity)
  - Hierarchical operations: tasks too dense can be divided on the flight



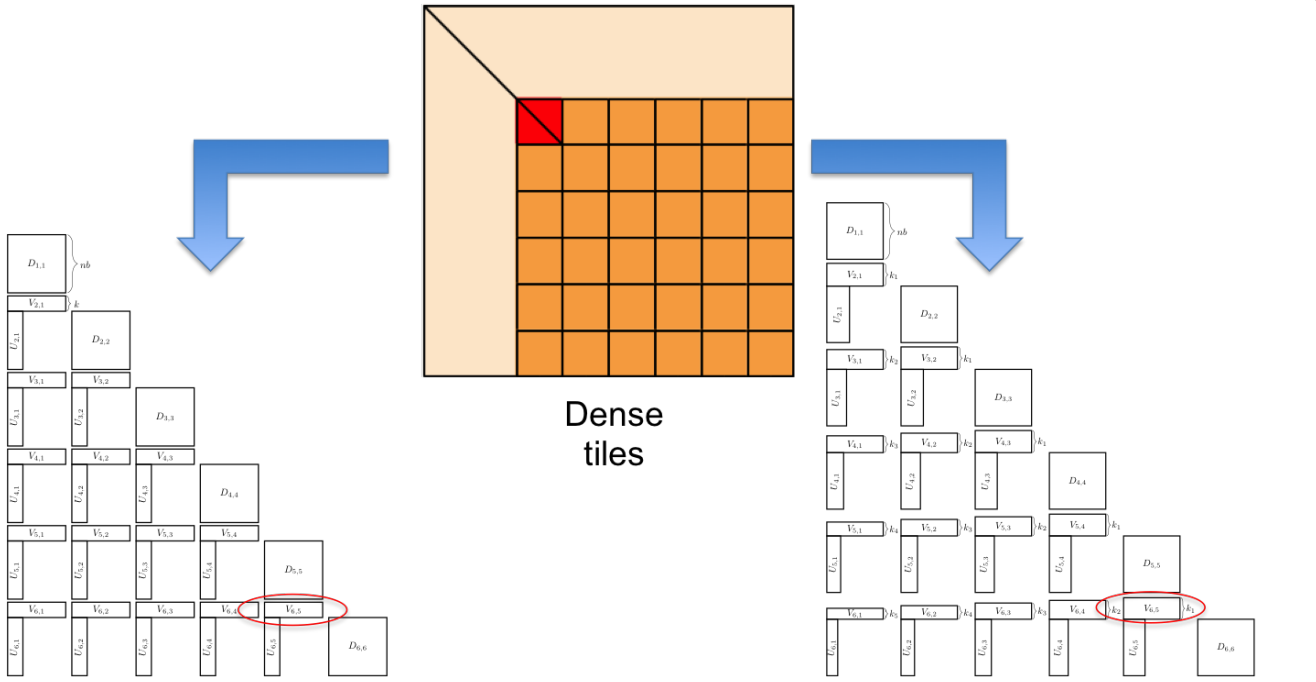


128 nodes on Summit



4096 nodes on Shaheen II

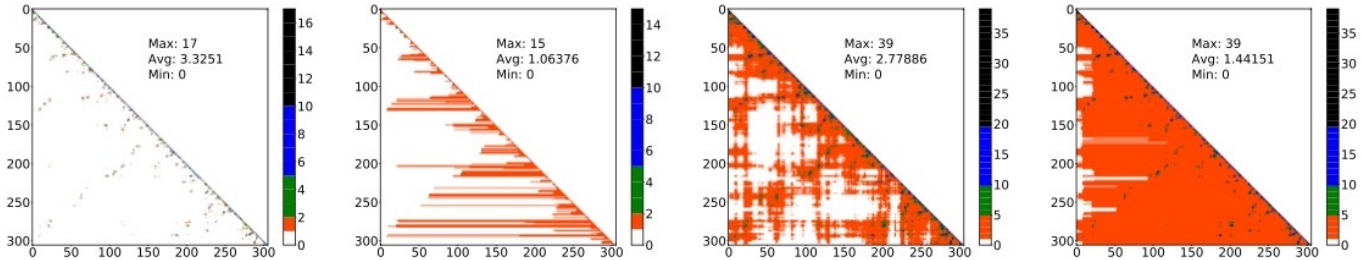
HiCMA: Sparse Low Rank mixed-precision



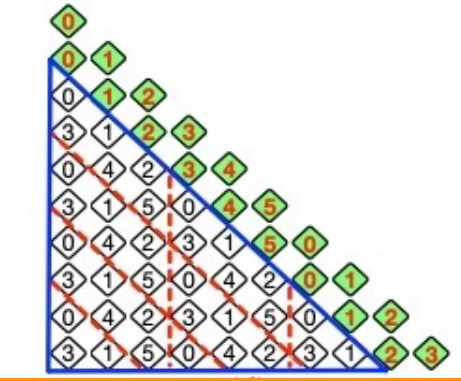
Fixed ranks
Preconditioners
Performance oriented

Fixed accuracy
Variable ranks
Dense/Sparse Direct Solvers

- Shared issues with Low rank
  - Implement recursive operations for the dense tiles
  - Look-ahead remains meaningful, but need to become adaptive
  - Tiling size has impact not only on performance but also accuracy
- Tiles that have low contributions should be nullified, but they can reappear later on
- Null tiles should not be taken in account by the algorithm
- A new, diamond-shaped data distribution

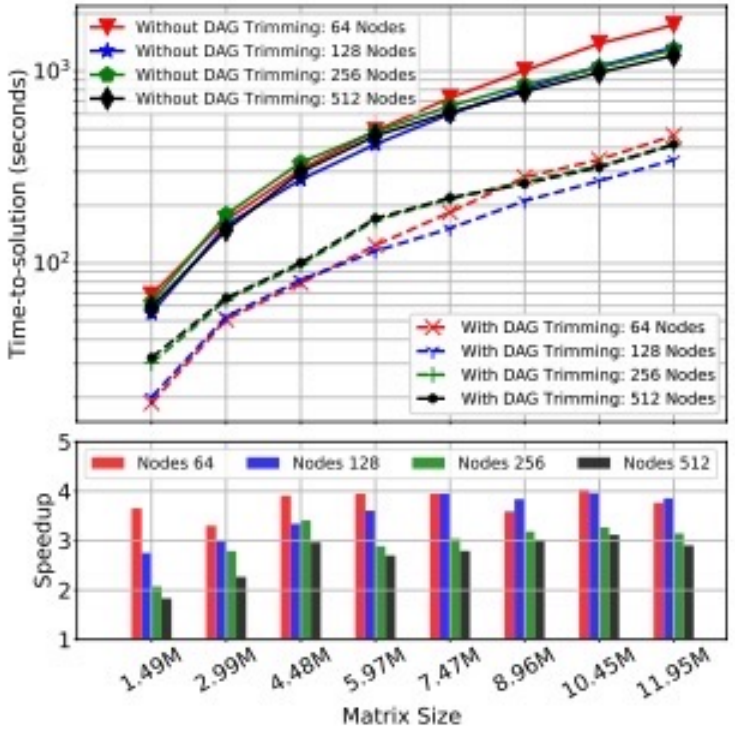


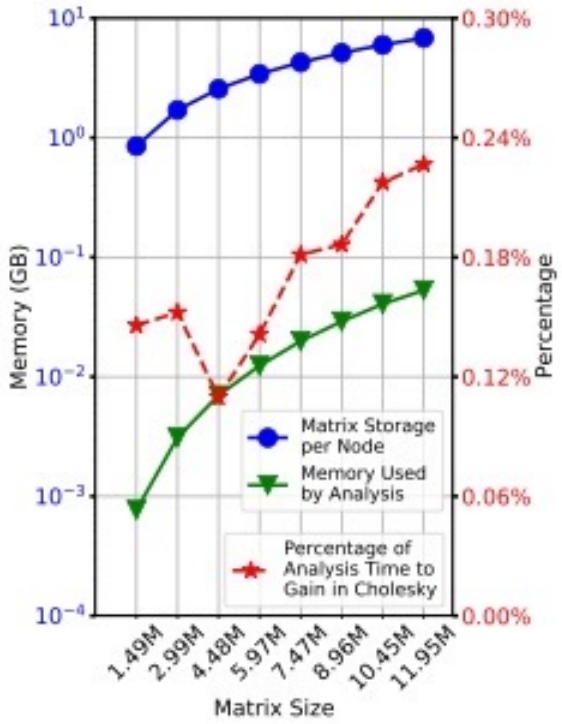
(a) Initial rank; shape parameter 3.7×10^{-4}, density 0.024606.
(b) Final rank; shape parameter 3.7×10^{-4}, density 0.206908.
(c) Initial rank; shape parameter 5.0×10^{-2}, density 0.552031.
(d) Final rank; shape parameter 5.0×10^{-2}, density 0.936969.

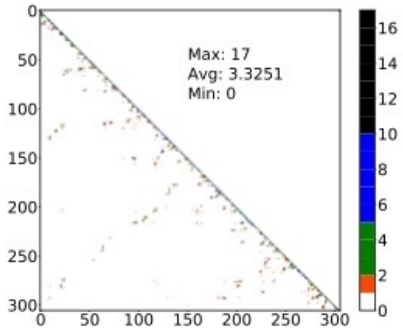


Sparse Low-Rank Cholesky

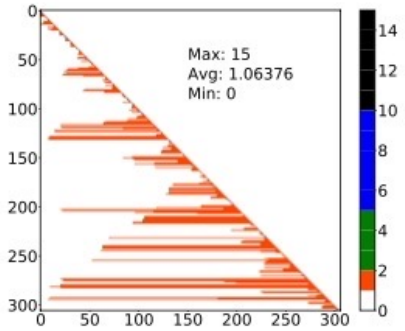
- Automatically trimming the PTG
- Adaptive data distributions
  - More work is needed to allow the algorith



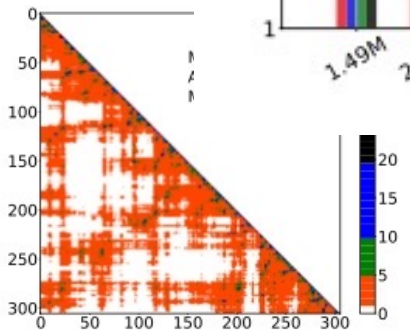




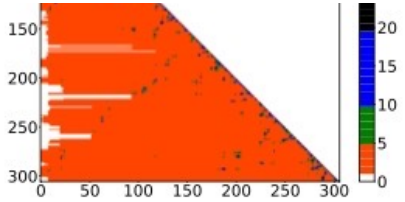
(a) Initial rank; shape parameter 3.7×10^{-4}, density 0.024606.



(b) Final rank; shape parameter 3.7×10^{-4}, density 0.206908.



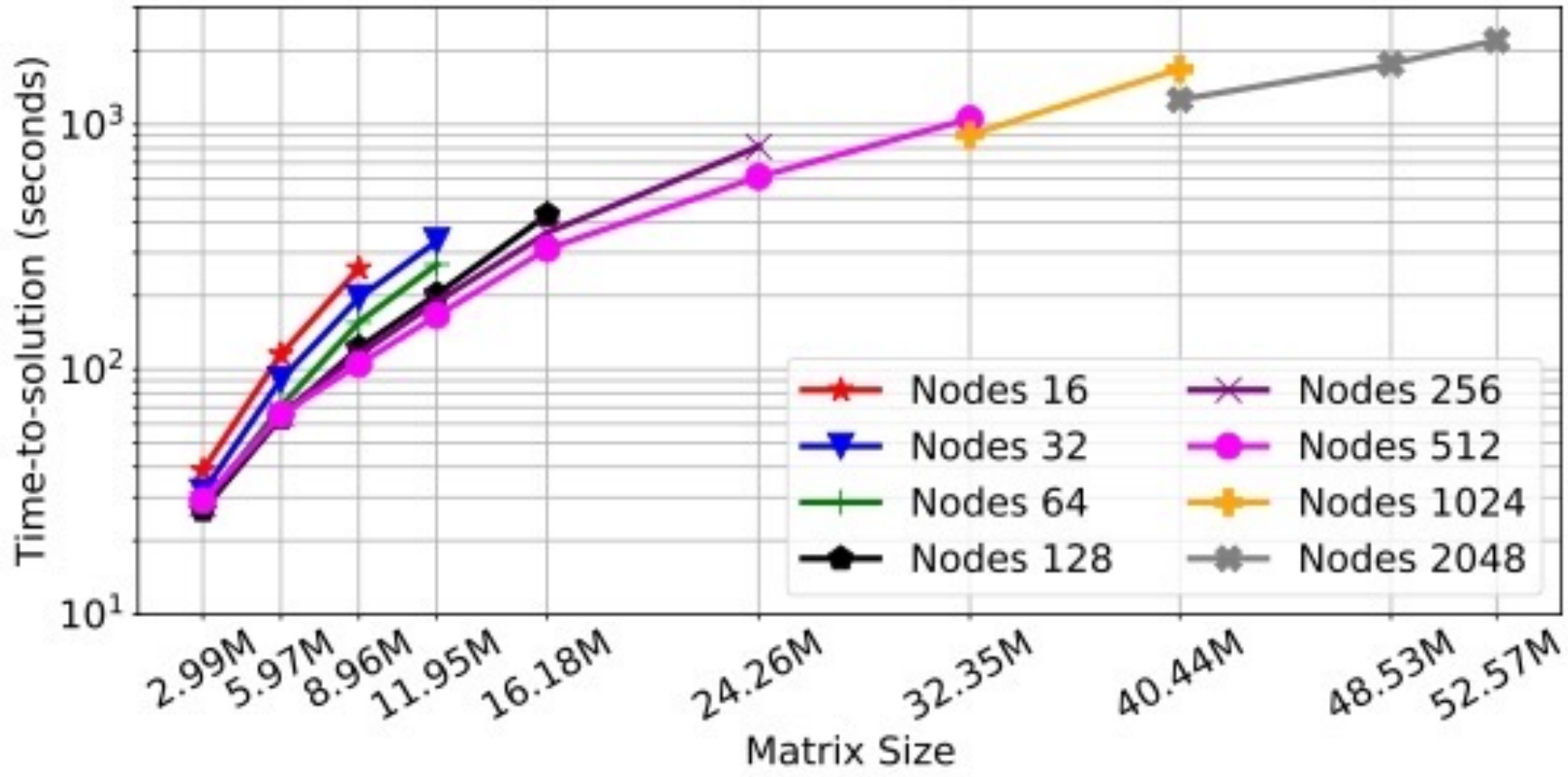
(c) Initial rank; shape parameter 5.0×10^{-2}, density 0.552031.



(d) Final rank; shape parameter 5.0×10^{-2}, density 0.936969.

Sparse Low-Rank Cholesky

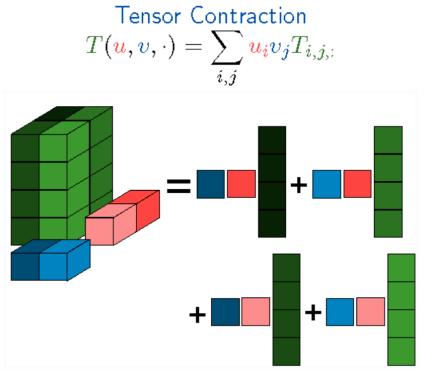
- Shaheen II @ KAUST



CCSD, Tensor Contraction

- If N is the number of atoms considered, CCSD's operation complexity is proportional to N^6 and, memory footprint is in order of N^4.
- The elephant in the room is the following tensor contraction:

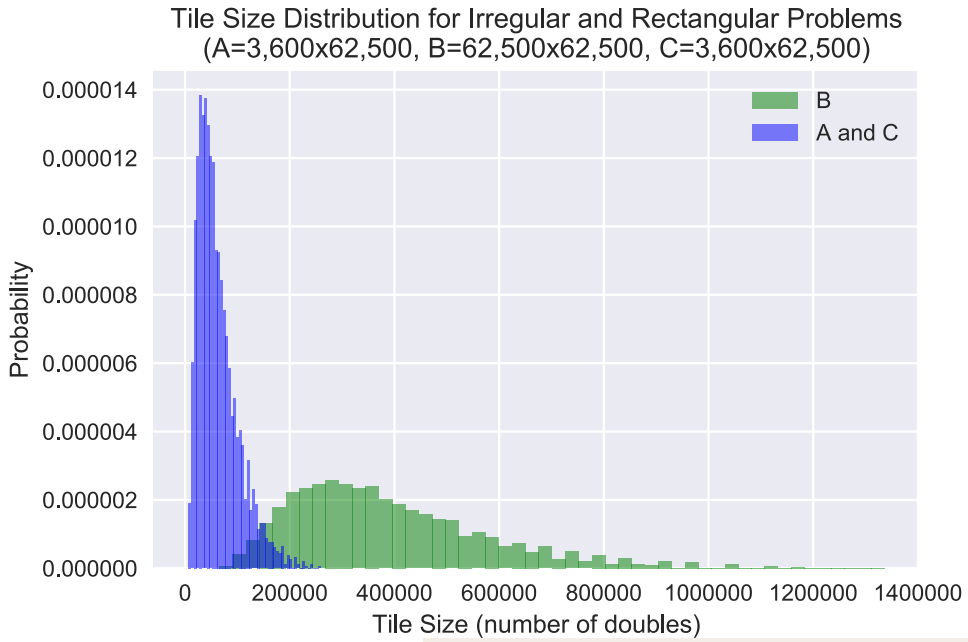
$$R_{ab}^{ij} = \sum_{cd} T_{cd}^{ij} G_{ab}^{cd} + \ldots,$$

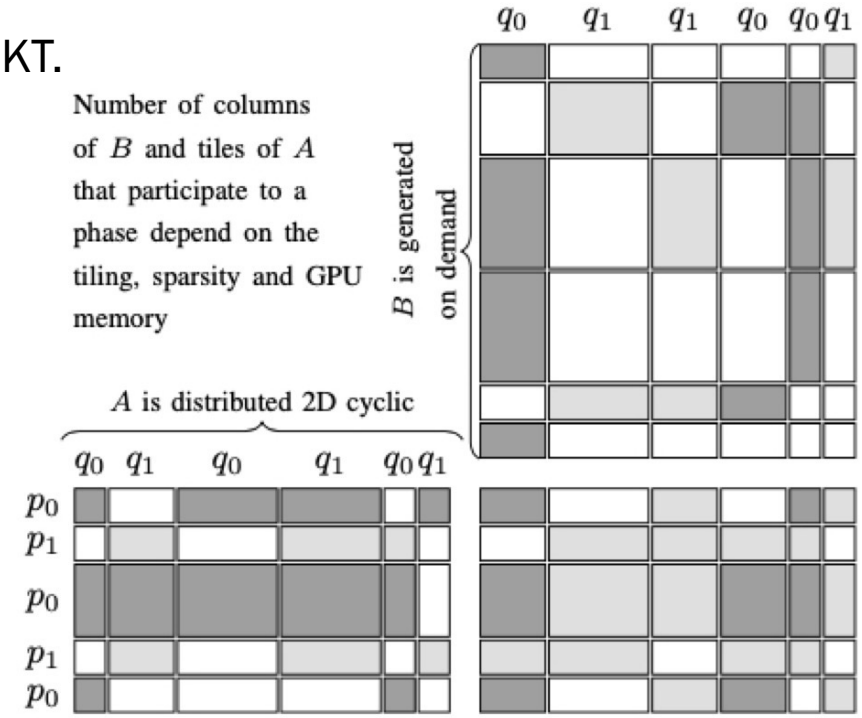


- This operation consumes 90% or more of the execution time and is the target for optimization.
- Over a few dozen iterations, tensor T is modified to nullify tensor R. Tensor G is a reference tensor and doesn't change.
- This tensor operation can be viewed as a matrix-matrix multiply where indices are fused in pairs, i and j, a and b, c and d.
- Indices a, b, c and d have ranges that are 5 to 20 times larger than i and j. It makes the tensor of Unoccupied (G) 25 to 400 times larger than the Occupied (ijcd).
- T and R are tall and skinny matrices, G is square.

NOT the traditional GEMM !

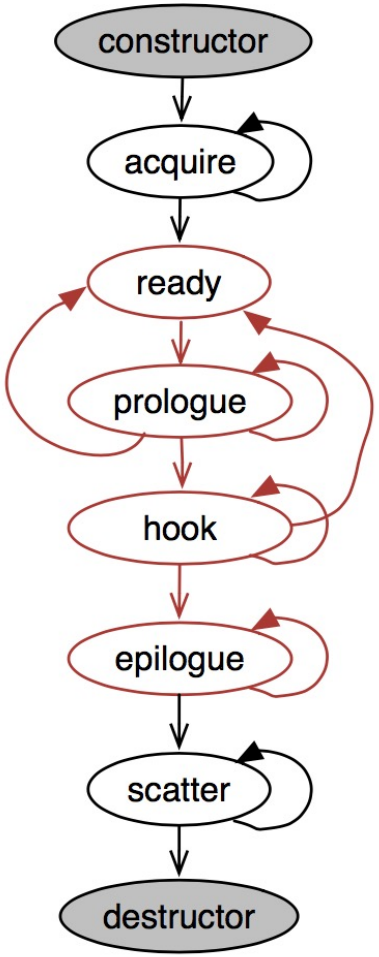
- Due to the nature of the underlying problem the distribution of tiles is not 2D block cyclic, it's not even 2D block, and certainly not cyclic. The benchmark is seeing all blocks in an array and cutting that array in *np* chunks.
- G is very different from T and R. G is block-sparse, while T and R can be low-rank or element-sparse.
- MT (number of tiles along M) is order of magnitude smaller than NT and KT.





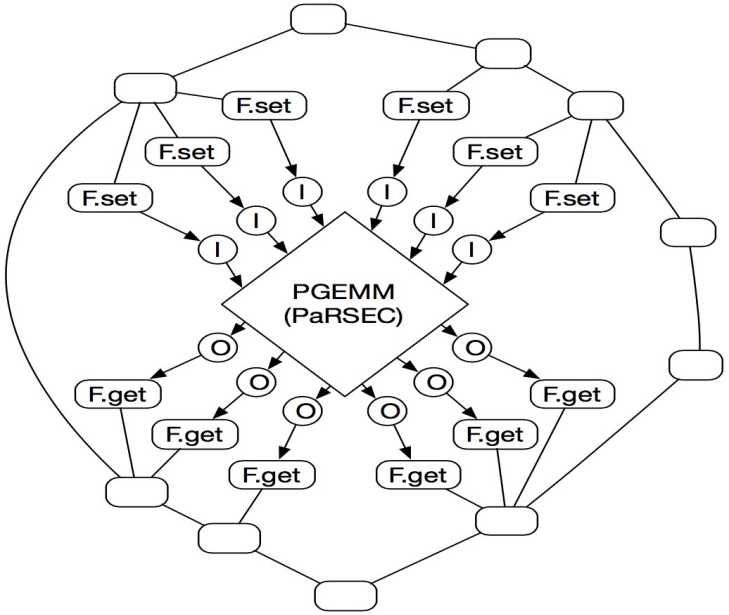
Did I mention the size of the matrices ? And that we target hybrid multi-GPU pre-exascale nodes?

Integration PaRSEC + (MADNESS + TiledArray)



- Use PaRSEC as a support for MADNESS ready-tasks
- Similar to providing another threading management support
  - With benefits of thread binding, NUMA- aware schedulers
- Data movement, dependencies are still managed entirely by MADNESS
- Enables seamless integration of new PaRSEC-enabled components:
  - Interaction between TTG (TESSE) and other MADNESS programming paradigms
  - Interaction between DPLASMA (PaRSEC, Linear Algebra library) and MADNESS operations

- Inputs and Outputs of existing PaRSEC DAGs (e.g. DPLASMA) are presented to MADNESS as Futures
- acquire/scatter steps of the input/output tasks are specialized to expose the task as a MADNESS Future
- Operations are made asynchronous:
  - If a future is not set at acquire time, the task
  - is unscheduled, and a callback to reschedule it is registered with MADNESS when the future is set
  - Get operation on output tasks is empty (does not allocate resource) until the data is ready



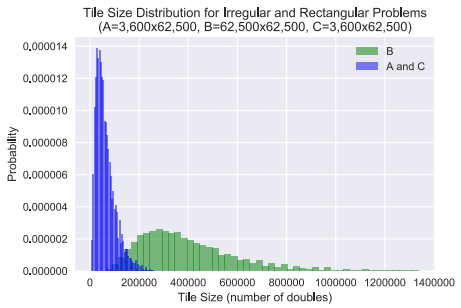
TESSE runtime, strong scaling

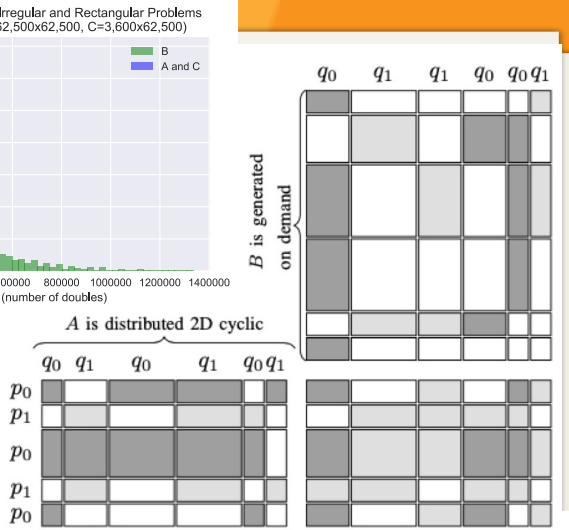
- Tensor contraction of the ABCD term in the coupled-cluster doubles equation for $(H_2O)_{12}$ in aug-cc-pVDZ basis set.
- The problem's ranges are O=60, U=432, nO=2, nU=12;
- M = 3600, N = K = 186,624, MT = 4, NT = 144 or mean(MB) = 900, mean(NB) = 1296
- Peak performance for GPUs are for blocks that are multiples (like 64). If you don't have that, you automatically lose ~10-15% performance.

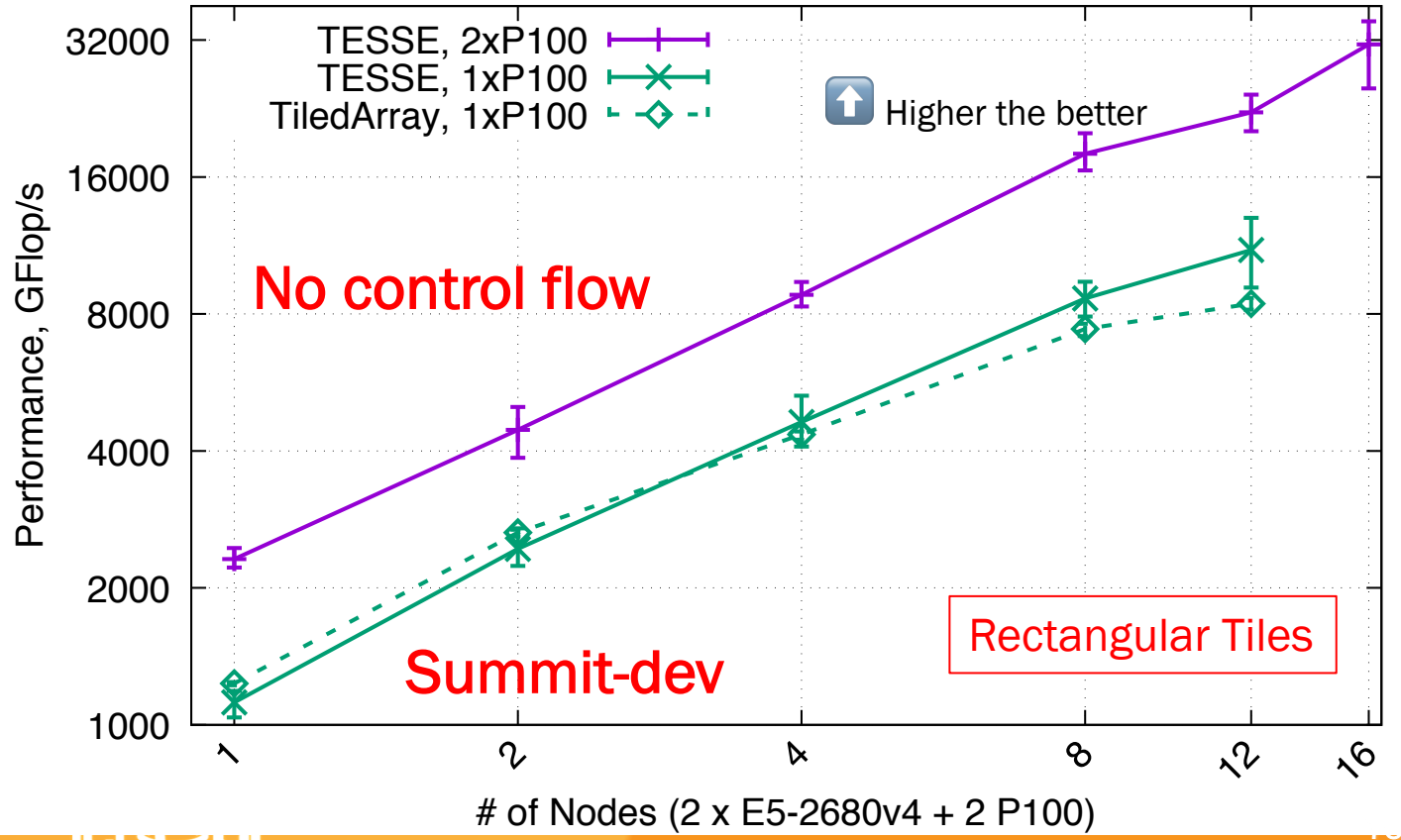
- TESSE runtime achieves x12.8 speedup from 1 to 16 nodes
- Native TiledArray achieves x6.8

- TESSE runtime supports 2 or more GPU per node, which is not the case for TiledArray.

- Overall performance of TESSE runtime is ~20% of peak. Legacy applications achieve ~2% of peak!



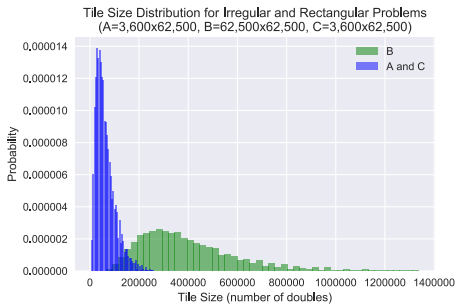


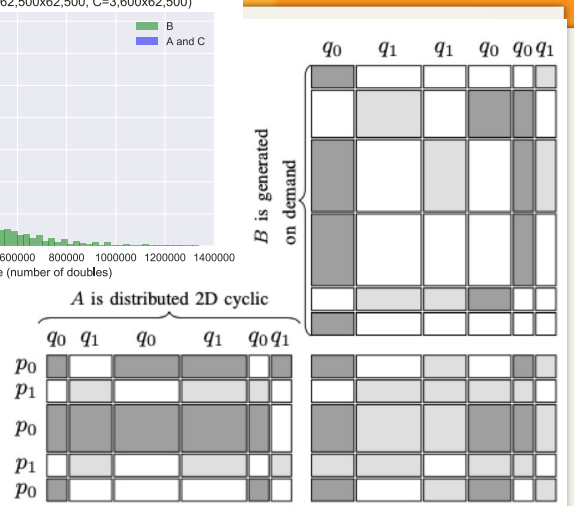


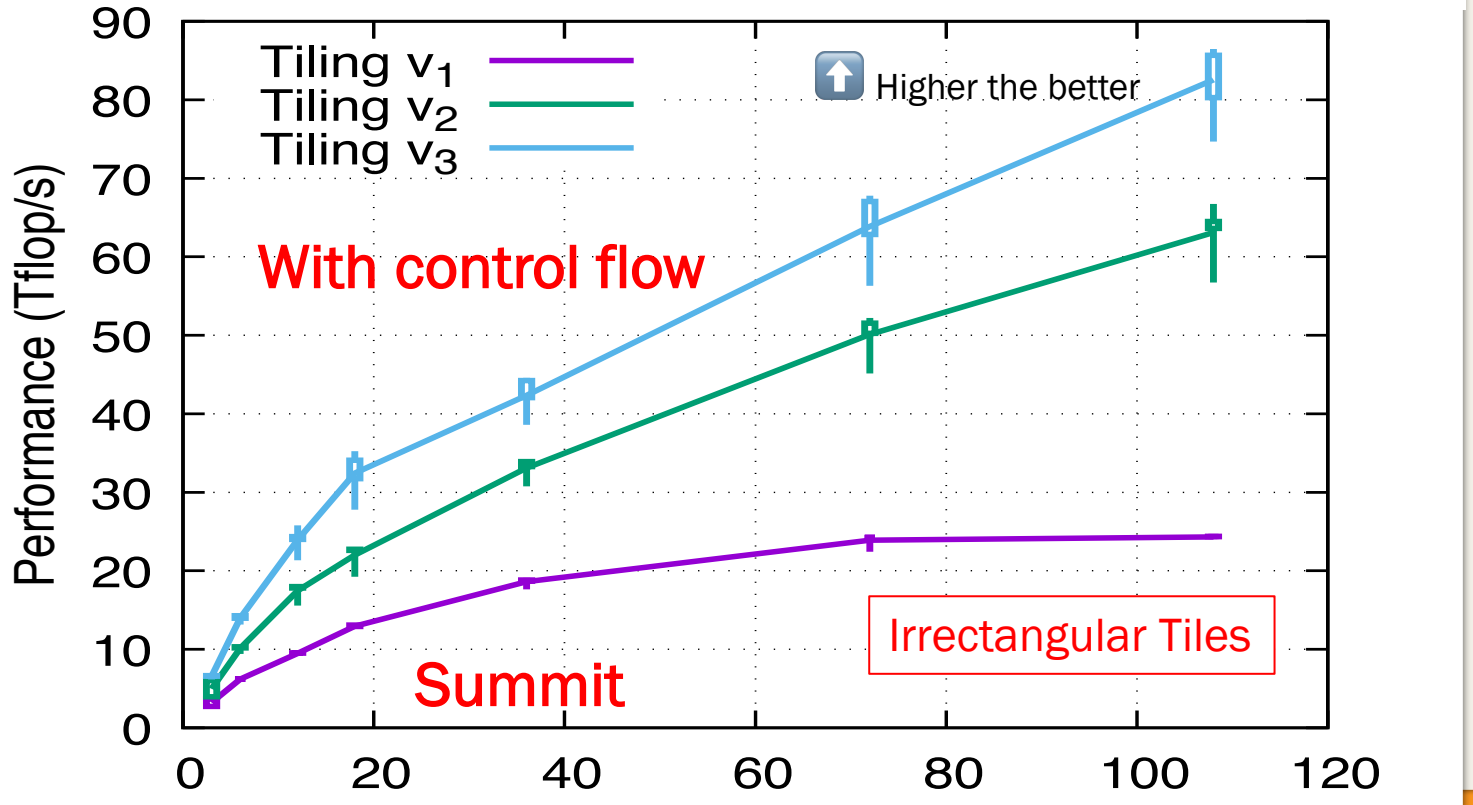
TESSE runtime, strong scaling

- Tensor contraction of the ABCD term in the coupled-cluster doubles equation for $(H_2O)_{12}$ in aug-cc-pVDZ basis set.
- The problem's ranges are O=60, U=432, nO=2, nU=12;
- M = 3600, N = K = 186,624, MT = 4, NT = 144 or mean(MB) = 900, mean(NB) = 1296
- Peak performance for GPUs are for blocks that are multiples (like 64). If you don't have that, you automatically lose ~10-15% performance.

- Overall performance of TESSE runtime is ~20% of peak. Legacy applications supporting multiple GPU were not available at the time of this experiment.







Conclusions

- Task-based runtime can provide an interesting programming solution
  - Well designed support from the runtime is critical
  - DSL to expose a reasonable amount of parallelism are critical
  - Portability across multiple architecture
- With PaRSEC we build a scientific enabler allowing different communities to focus on different problems
  - Application developers on their algorithms via DSL
  - Language specialists on Domain Specific Languages
  - System developers on system issues
  - Compilers on optimizing the task code

More info about PaRSEC

- ICL PaRSEC website and DTE website
- PaRSEC
  - Issues, pull requests, documentation, user community