# Using Kokkos in Template Task Graphs

Joseph Schuchart

Kokkos User Group Meeting

December 13, 2023, Albuquerque, NM

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

# Who we are

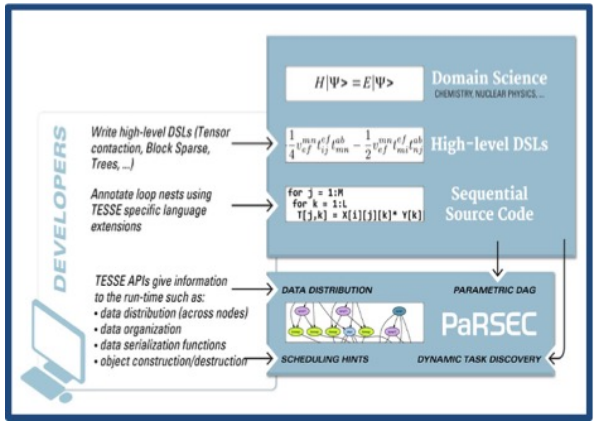
George Bosilca

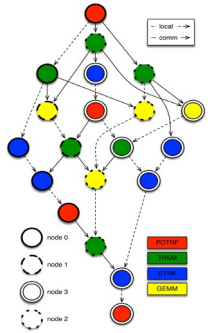
Thomas Herault

Joseph Schuchart

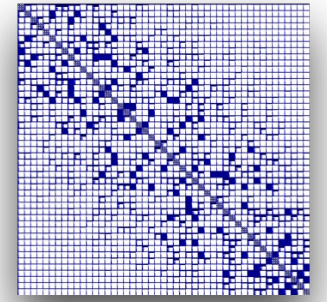
Eduard Valeev

Robert Harrison

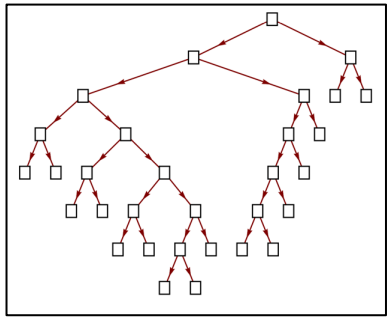



dense linear algebra



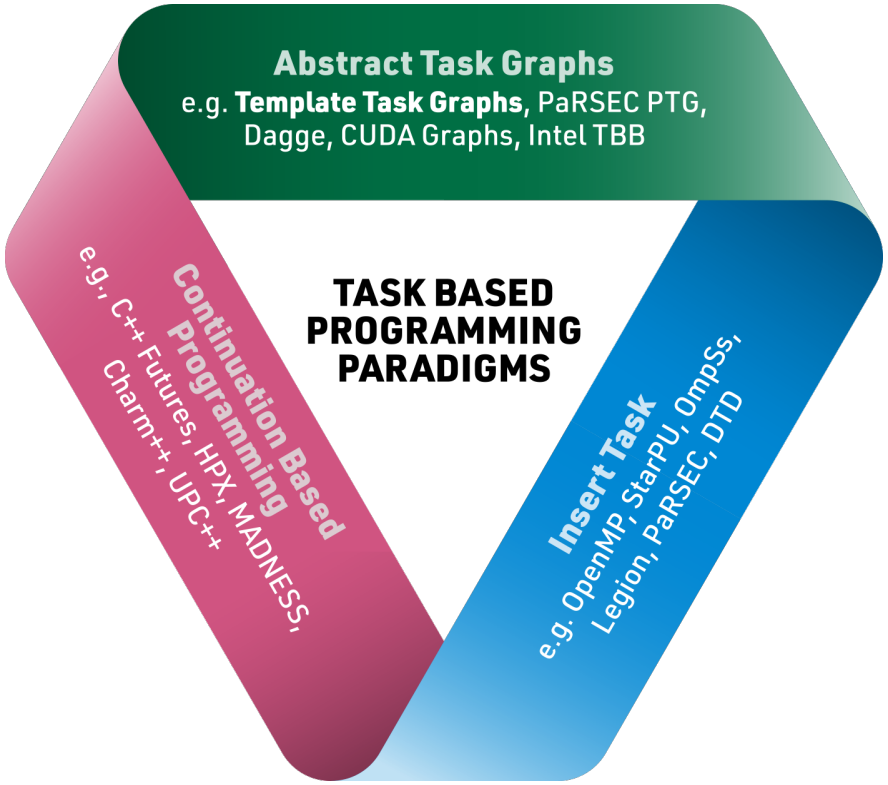
block-rank sparse algebra for quantum chemistry/physics



adaptive spectral-element calculus Multi-Resolution Analysis

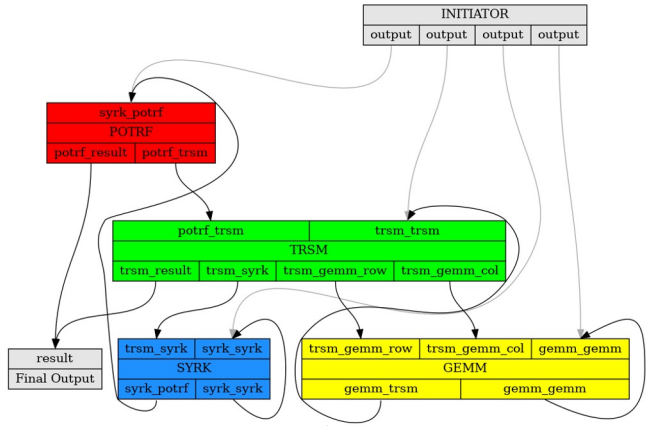
# Task Systems

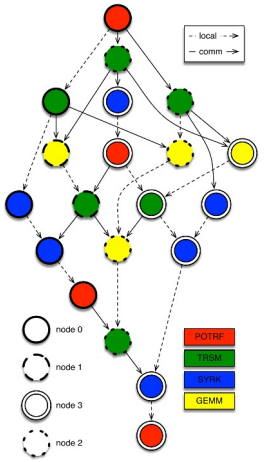
- Insert Task:
  - OpenMP, StarPU, PaRSEC DTD
  - Orchestration through dependencies on memory locations
- Continuations:
  - Futures representing results of tasks
  - Callbacks as reaction to the completion of tasks
- Abstract task graphs:
  - CUDA Graphs, C++ sender/receiver, PaRSEC, TTG
  - A priori description of task-graph instantiated during execution



# TTG: Overview

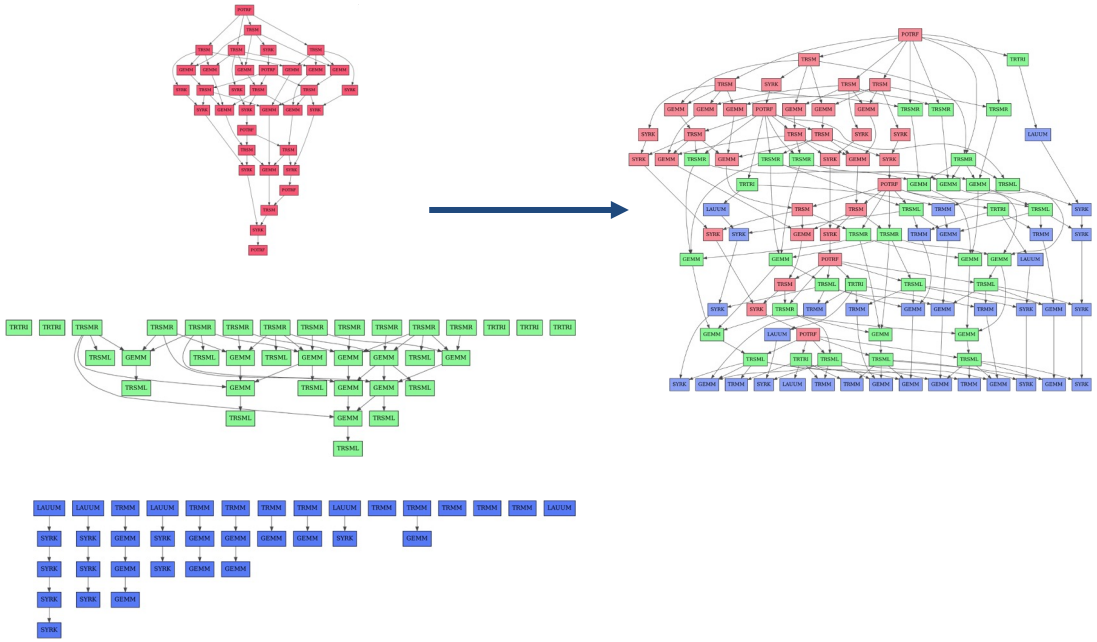
- Distributed Data Flow as Abstract Task Graph
  - May contain cycles
  - Nodes: template tasks
  - Edges: possible data flow between tasks
    - Represent sets of data
- Template Task Graph unrolled during execution
  - Tasks identified through (hashable) IDs (keys)
  - Data flows along edges as Pair {TaskID, Data}
- Data-dependent successor discovery
  - Tasks may send on different edges depending on results

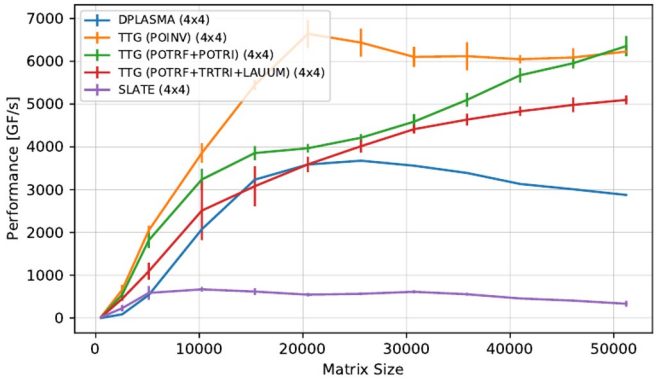


# Task Graph Composition: POINV

- Edges enable composition of black-box task-graphs
- POINV = POTRF ⊕ TRTRI ⊕ LAUUM
- Benefits esp for small tiles






(a) Tile size 128.

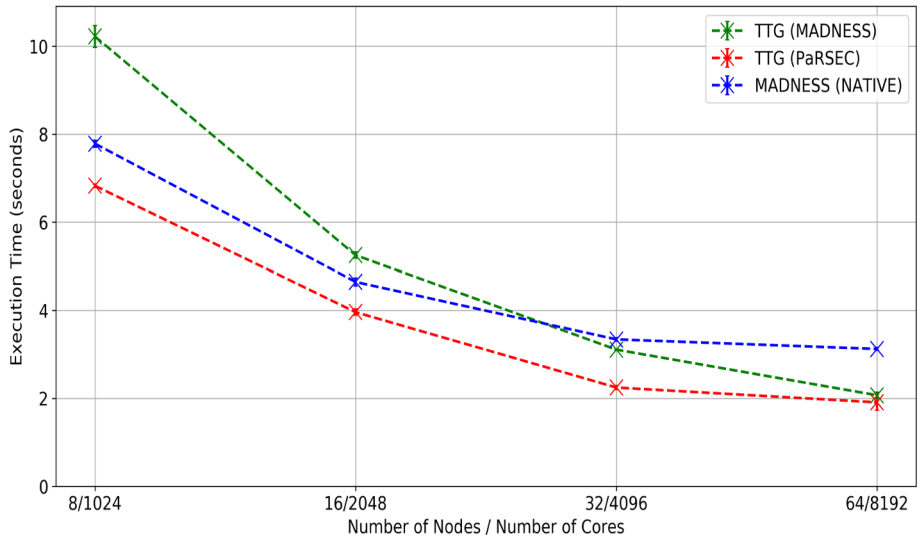


ICL

THE UNIVERSITY OF TENNESSEE KNOXVILLE

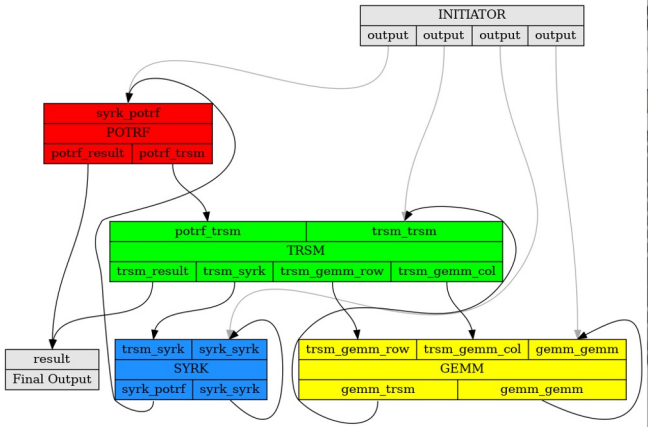
# Target Applications: Multi-Resolution Analysis

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

# TTG Execution Model (General)

- SPMD: all processes execute the same program in main thread
- ttg::World: query number of processes and local rank
  - Split processes between multiple worlds (i.e., communicators)
- Single or multiple entry points into the DAG
  - Process(es) kick off computation by feeding data into the task graph
  - Executing process controlled through mapper function
- Worker threads non-preemptively execute tasks
- Fence to wait for execution to complete
- Multiple task-graphs can be active concurrently



# TTG: Small Example

Simplifications in the works

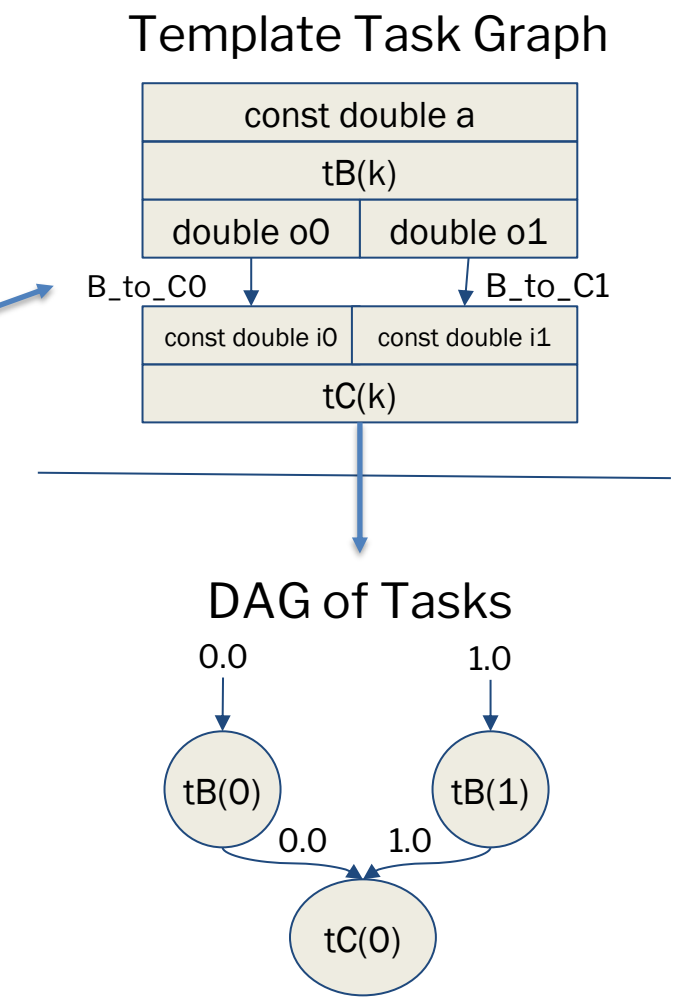
```
ttg::Edge<int, double> to_B("to_B");
ttg::Edge<int, double> B_to_C0("B_to_C0");
ttg::Edge<int, double> B_to_C1("B_to_C1");

auto tb = ttg::make_tt([](const int &k, const double &a) {
    // Task tB(k) received value a for input 0
    if(0 == k) ttg::send<0>(0, a);
    if(1 == k) ttg::send<1>(1, a);
  },
  ttg::edges(to_B),
  ttg::edges(B_to_C0, B_to_C1));

auto tc = ttg::make_tt([](const int &k, const double &i0, const double &i1)
  {
    // Task tC(k) received two inputs: i0 and i1
  },
  ttg::edges(B_to_C0, B_to_C1),
  ttg::edges());

ttg::make_graph_executable(tb);
if(tb->get_world().rank() == 0) {
  tb->invoke(0, 0.0);
  tb->invoke(1, 1.0);
}
ttg::execute();
ttg::fence();
```

Input edges

Output edges

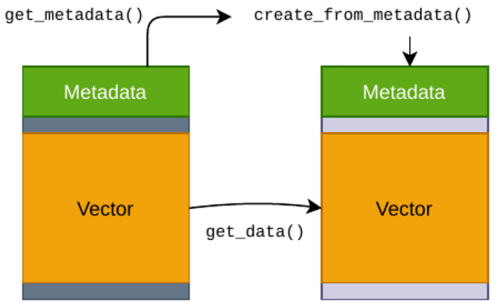
Input edges

Kick off tasks



# TTG Memory Model (General)

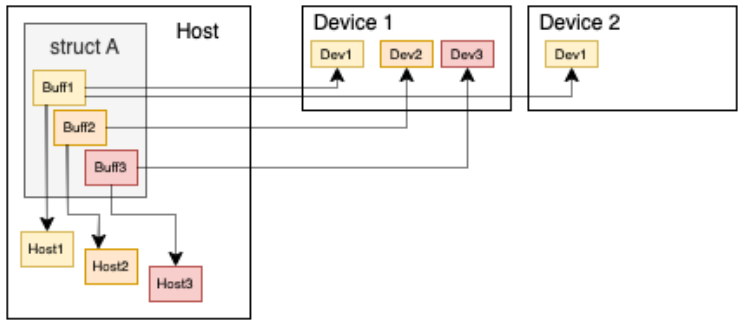
- TTG manages transfers between processes
  - No explicit receives
  - Only send/broadcast to successor tasks, addressed by keys
- All data flowing along edges must be
  - Serializable (MADNESS/Boost/trivially_copyable); or
  - Zero-copyable (Split Metadata API)
- Immutable objects shared between tasks
  - C++ const and move semantics
- Mutable data copied unless moved



# TTG Memory Model (Device)

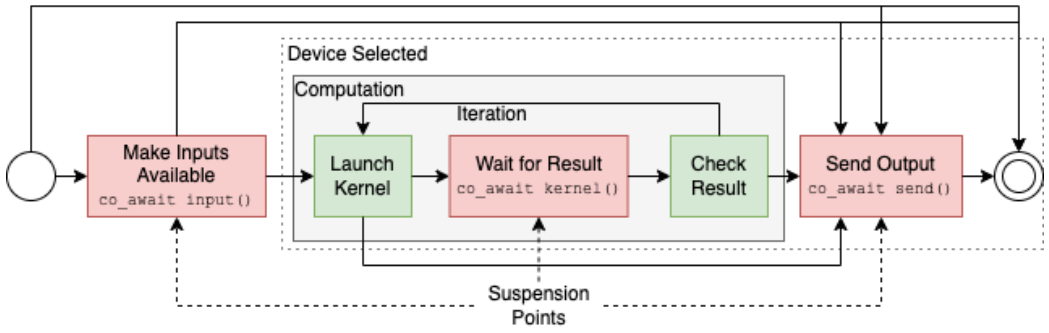
- TTG manages device memory
- Host memory serves as backup for eviction
  - Memory oversubscription supported by default
- Transparent data movement between devices and host
  - Automatic migration between device and host tasks
- ttg::buffer: owning/non-owning host memory mirror
  - Provides host and device pointer for current device
  - Like Kokkos::DualView without the sync/fence ☺

TTG Device integration still somewhat experimental



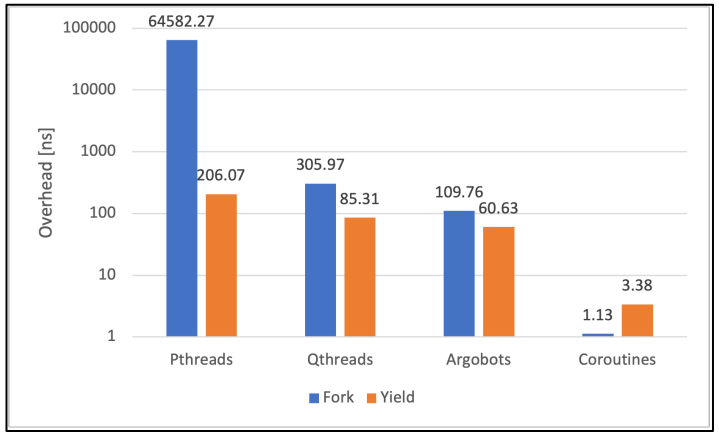
# TTG Execution Model (Device)

- Tasks declare input data (ttg::buffer, scratch data)
- TTG runtime assigns a device and execution stream based on inputs
  - One management thread per device (PaRSEC)
- Tasks submit kernels and H2D transfers into stream and suspend
- Runtime returns once execution completed
- Task may:
  - Submit more kernels; or
  - Send out results to successors
- Task suspension via coroutines

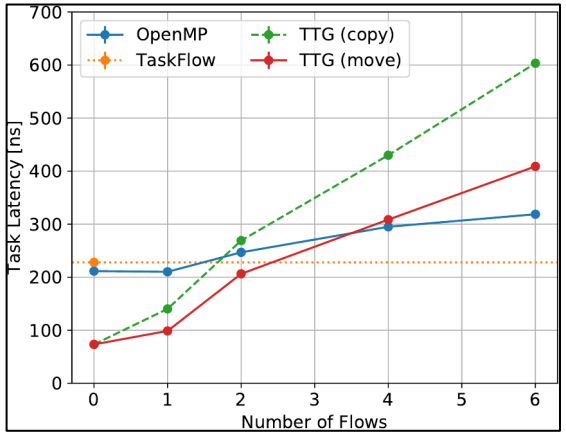


# Why Coroutines?

- TTG provides low-overhead task execution
- Device tasks:
  - Kernel submission
  - Successor discovery
- Fibers/ULTs provide flexibility at a cost
- Compiler-assisted suspension ~ cost of a function call
- Limited code fragmentation




Task Overhead

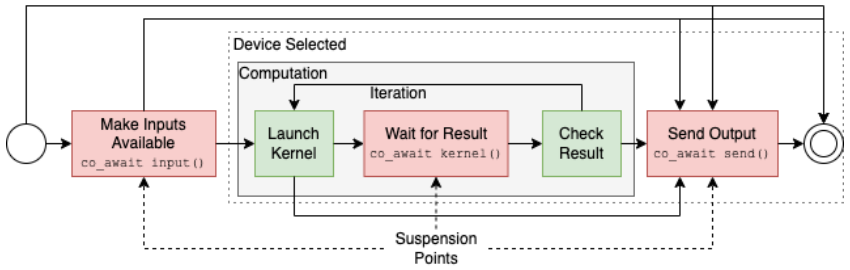





ICL THE UNIVERSITY OF TENNESSEE KNOXVILLE

# TTG Device Tasks

- “co_await to_device” selects device
- “co_await wait_kernel” accepts buffers/scratch to return to host
- Task may submit and wait for multiple kernels
- Sending outputs is last step of task




Make inputs available

Submit work

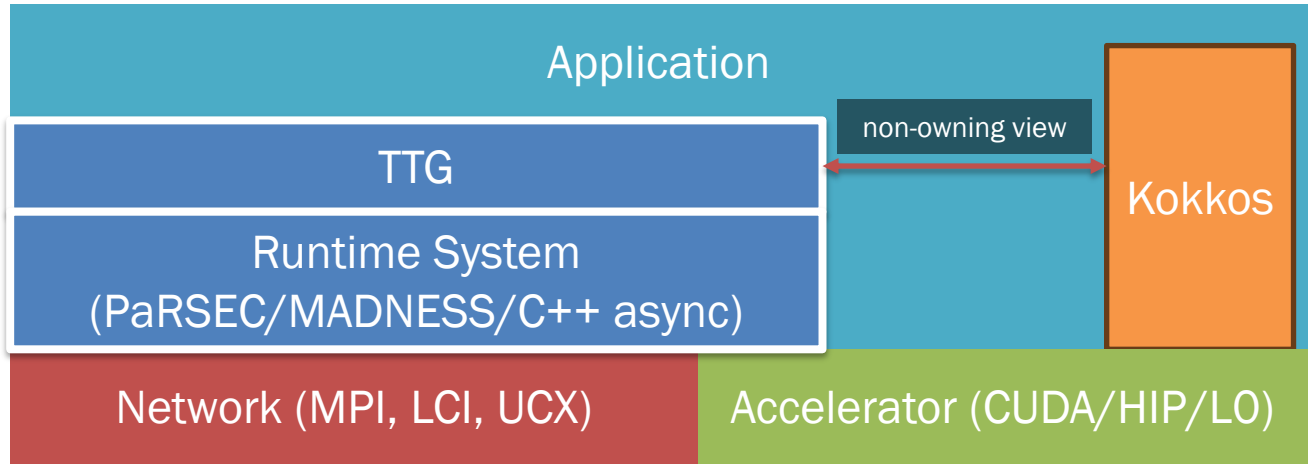
Wait for kernel to execute (optional)

Send Output

```
template<typename T>
struct Tile {
  ttg::buffer<T> buf;
  size_t m, n, lda;
  Tile(size_t m, size_t n, :size_t lda)
  : buf(m*n) // buffer owns host memory
  { }
  Tile(T *ptr, size_t m, size_t n, size_t lda)
  : buf(ptr, m*n) // buffer does not own host memory
  { }
  // other constructors and accessors
};

using Key = std::pair<int, int>; // tile position in matrix

auto tb = ttg::make_tt([](const Key& k, Tile&& a)
          -> ttg::device::task {
    co_await ttg::device::to_device(A.buf); // make A available
    submit_kernel(A);
    co_await ttg::device::wait_kernel(); // optional if no result required
    co_return ttg::device::send<0>(k, std::move(A));
  }, ...);
```

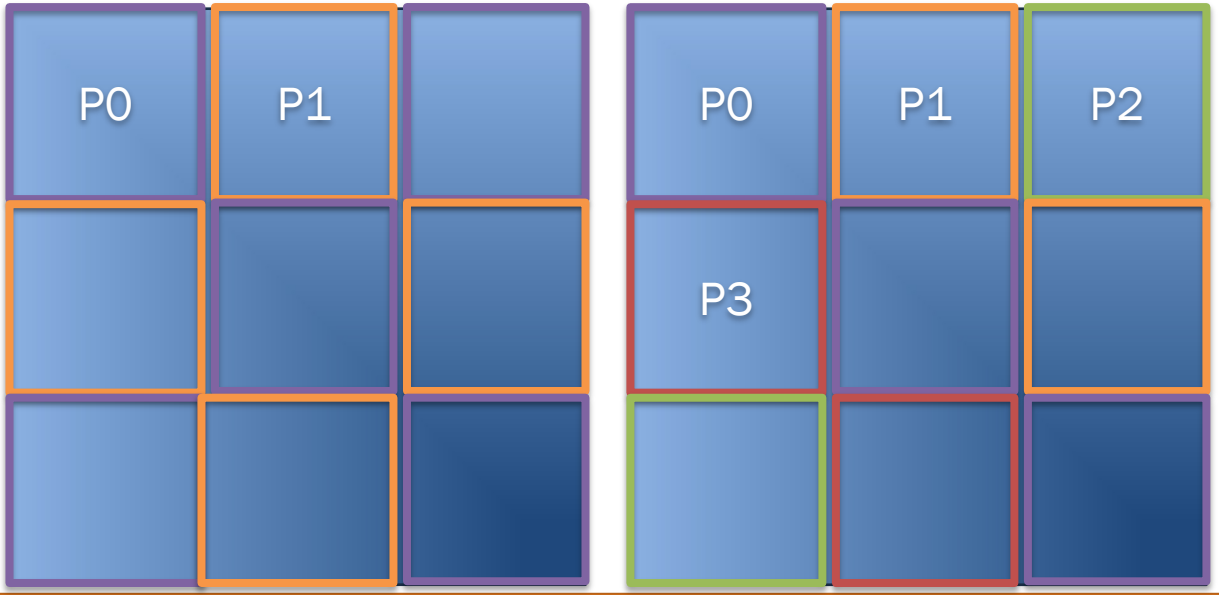
# Kokkos and TTG

- TTG: distributed data-flow programming
  - No interest in providing task-level concurrency
- Kokkos: accelerator programming
- Integration point: non-owning Kokkos::View & execution environments
  - TTG manages device memory (ttg::buffer) and distributed execution
  - Kokkos provides accelerator programming infrastructure



# TTG and Kokkos: PLGSY

- PLGSY: generation of symmetric diagonally dominant matrix
  - Independent of process grid and tile size
- Position of tile: encoded in task ID
- Not an official BLAS function
  - Part of (D)PLASMA
- Initialization of tiles fed into task graph



# Kokkos PLGSY Kernel

Stride Layout

CUDA w/ explicit stream

MDRange

Make tile memory available

Submit Kokkos kernel

Send to successor

```
TiledMatrix<T> matrix; // provides

using Key = std::pair<int, int>; // tile position in matrix

void CORE_plgsy(const Key& key, T* tile, int m, int n, int lda, T bump) {
  using layout_type = Kokkos::LayoutStride;
  auto layout = layout_type(m, lda, n, 1);
  auto view = Kokkos::View<T**, layout_type>(tile, layout);
  auto es = Kokkos::Cuda(ttg::device::current_stream());
  if ( m0 == n0 ) { // diagonal
    Kokkos::parallel_for("diagonal",
      Kokkos::MDRangePolicy<Kokkos::Cuda, Kokkos::Rank<2>>(es, {0, 0}, {n, m}),
      KOKKOS_LAMBDA(int row, int col) {
        view(row, col) = gen(m, n, lda, key); // generate value for element
        if (row == col) { // bump diagonal element
          view(row, col) += bump;
        }
      });
   } else if (...) {
    ...
}

auto tb = ttg::make_tt([](const Key& k, Tile&& tile)
          -> ttg::device::task {


    co_await ttg::device::to_device(tile.buf); // make tile available
    auto ptr = tile.buf.current_device_ptr(); // memory on assigned device
    CORE_plgsy(key, ptr, tile.m(), tile.n(), tile.n(), ...);



    co_return ttg::device::send<0>(k, std::move(A));
  }, ...);
```

# Kokkos PLGSY Kernel w/ Norm

- Useful for debugging

```
TiledMatrix<T> matrix; // provides

using Key = std::pair<int, int>; // tile position in matrix

void CORE_plgsy(const Key& key, T* tile, int m, int n, int lda, T bump) {
  using layout_type = Kokkos::LayoutStride;
  auto layout = layout_type(m, lda, n, 1);
  auto view = Kokkos::View<T**, layout_type>(tile, layout);
  auto es = Kokkos::Cuda(ttg::device::current_stream());
  if ( m0 == n0 ) { // diagonal
    Kokkos::parallel_for("diagonal",
      Kokkos::MDRangePolicy<Kokkos::Cuda, Kokkos::Rank<2>>(es, {0, 0}, {n, m}),
      KOKKOS_LAMBDA(int row, int col) {
        view(row, col) = gen(m, n, lda, key); // generate value for element
        if (row == col) { // bump diagonal element
          view(row, col) += bump;
        }
      });
   } else if (m0 > n0) {
    ...
}

auto tb = ttg::make_tt([](const Key& k, Tile&& tile)
          -> ttg::device::task { // make lambda a coroutine
    T norm;
    auto scratch = ttg::make_scratch(&norm);
    co_await ttg::device::to_device(tile.buf, scratch); // make tile available
    auto ptr = tile.buf.current_device_ptr(); // memory on assigned device
    CORE_plgsy(key, ptr, tile.m(), tile.n(), tile.n(), ...);
    compute_norm(ptr, scratch.device_ptr());
    co_await ttg::device::wait_kernel(scratch); // optional if no result required
    tile.set_norm(norm);
    co_return ttg::device::send<0>(k, std::move(A));
  }, ...);
```

Scratch valid for task lifetime

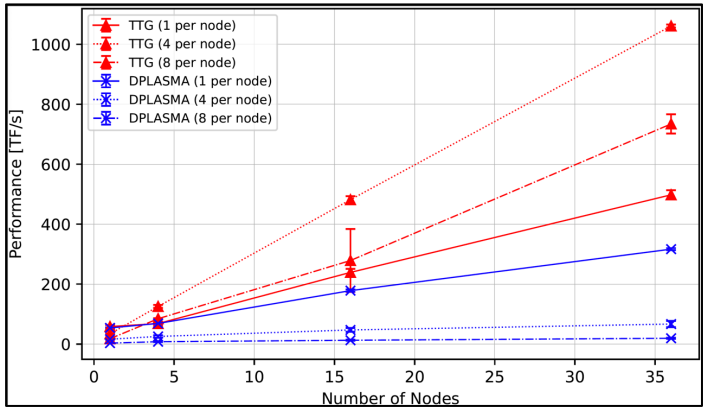
Calls *blas*nrm2

Waits for norm transfer

# Integration Challenges

- TTG relies on C++20 but Kokkos requires C++14/17
  - CMake integration is non-trivial (docs?) :/
  - Vendor-compiler error reporting sub-par

GEMM, Frontier




- Multi-GPU support in Kokkos?
  - TTG assigns device and stream to a task.
    Kokkos::Cuda() takes a stream but calls cudaSetDevice?

- Multi-threading support in Kokkos?
  - Kernel submission by multiple threads (1 thread per device)
  - Cost of creating a Kokkos::Cuda instance?

# Summary

- Mostly seamless TTG <-> Kokkos interaction
  - If we ignore the build-system headaches
- TTG can provide scalable distributed execution utilizing Kokkos' node-local performance

- Future Work:
  - Experiment with Kokkos kernels
  - TTG MRA implementation using Kokkos
  - Device-first memory allocation (on-demand host allocation)
  - Batched kernel tasks
  - Applications (SPMM, MRA, MADNESS/TA integration, ?)

# Acknowledgements

This research was supported partly by NSF awards #1931347 and #1931384, and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. We gratefully acknowledge the provision of computational resources by the Oak Ridge National Laboratory (ORNL) and the High-Performance Computing Center (HLRS) at the University of Stuttgart, Germany.


EXASCALE COMPUTING PROJECT



HLRS



ICL THE UNIVERSITY OF TENNESSEE KNOXVILLE

# Resources

- ECP Tutorial: https://www.exascaleproject.org/event/ttg-2022/
- Paper:
  - J. Schuchart *et al.*, "Generalized Flow-Graph Programming Using Template Task-Graphs: Initial Implementation and Assessment," *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS).*
  - J. Schuchart, P. Nookala, T. Herault, E. F. Valeev and G. Bosilca, "Pushing the Boundaries of Small Tasks: Scalable Low-Overhead Data-Flow Programming in TTG," *2022 IEEE International Conference on Cluster Computing (CLUSTER)*.
  - T. Herault, J. Schuchart, E. F. Valeev and G. Bosilca, "Composition of Algorithmic Building Blocks in Template Task Graphs," *2022 IEEE/ACM Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM)*.
- Github: https://github.com/TESSEorg/ttg/


This research was supported partly by NSF awards #1931347 and #1931384, and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. We gratefully acknowledge the provision of computational resources by the High-Performance Computing Center (HLRS) at the University of Stuttgart, Germany.