Using Coroutines to Support Accelerators in TTG

Joseph Schuchart

HiHAT Monthly Review

January 16, 2024

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Who we are

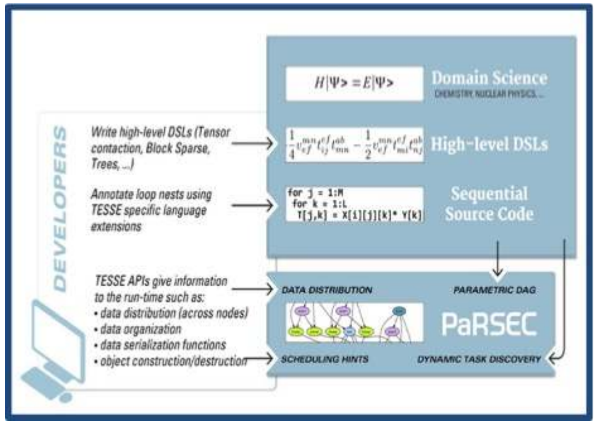

George Bosilca

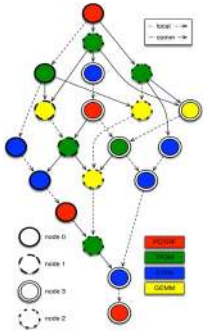

Thomas Herault

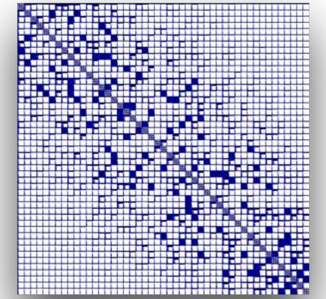

Joseph Schuchart

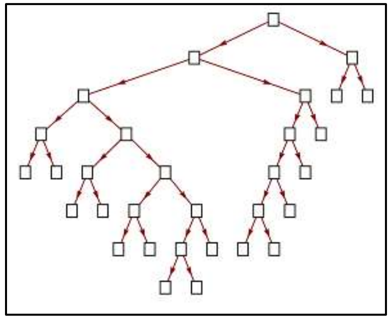

Eduard Valeev


Robert Harrison



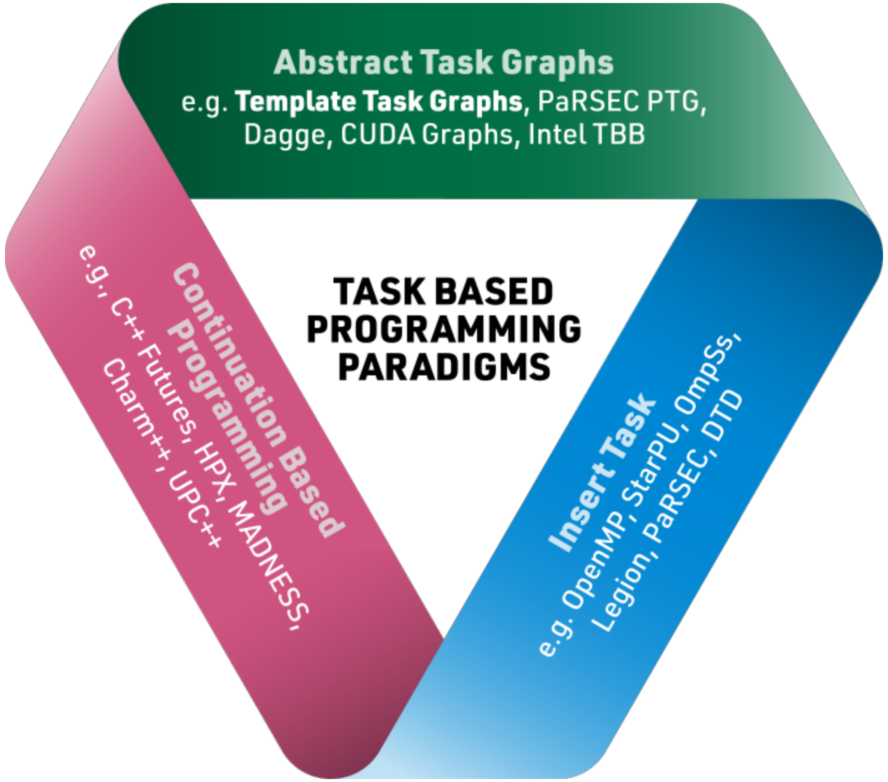

dense linear algebra


block-rank sparse algebra for quantum chemistry/physics


adaptive spectral-element calculus Multi-Resolution Analysis

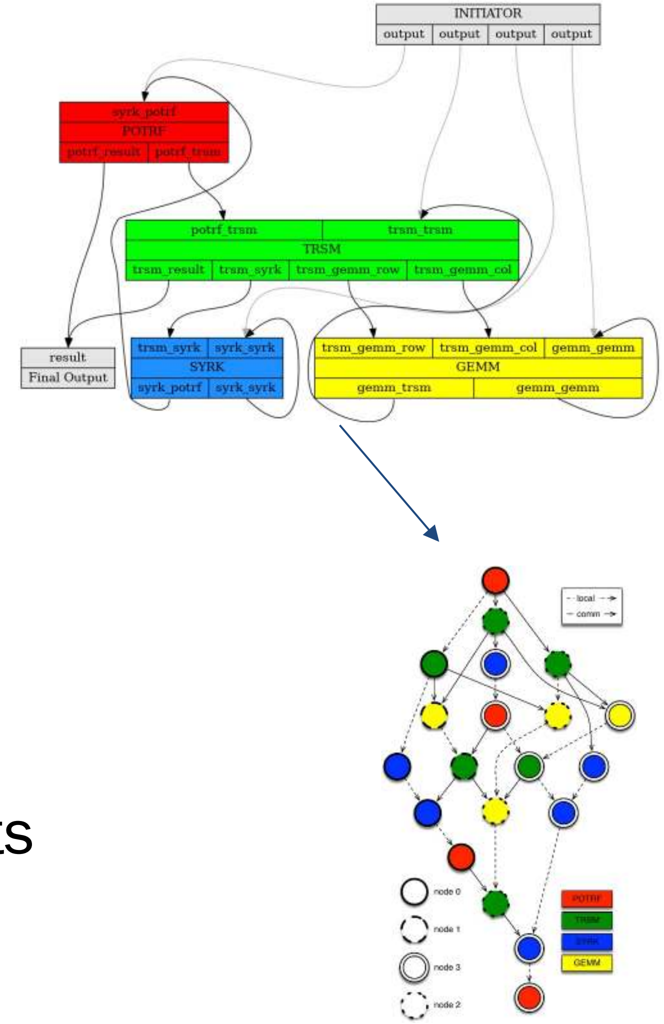
Task Systems

- 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



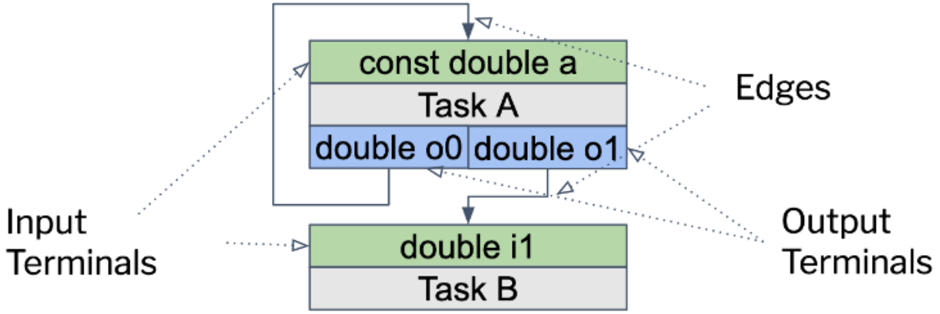
TTG: Overview

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



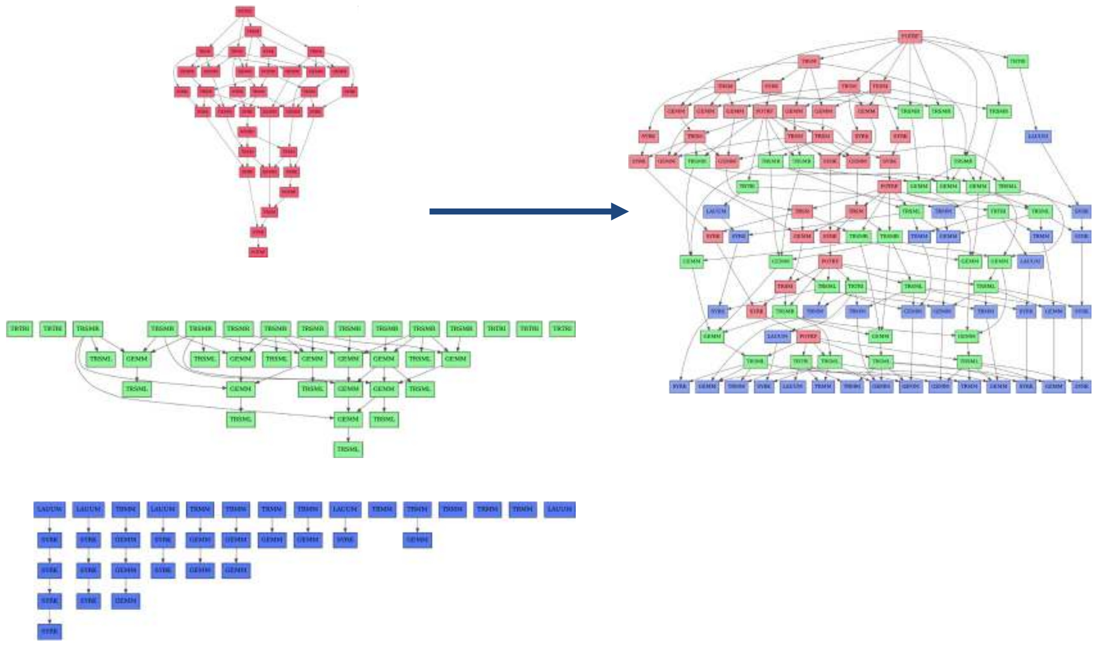
TTG: Tasks, Terminals, and Edges

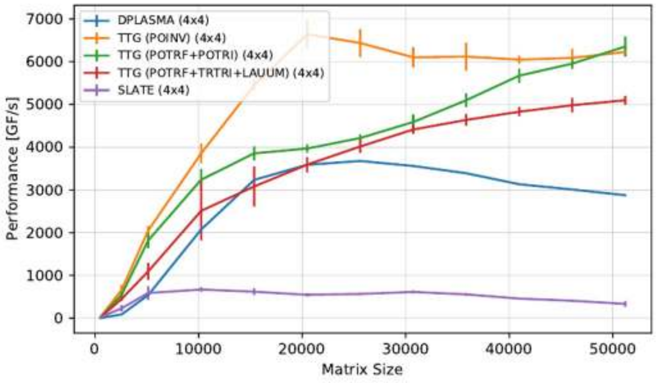
- Tasks: task with set number of inputs and outputs
  - Instantiated when first discovered
  - Executed once all inputs are available
- Terminals: inputs and outputs of a task, hidden from user code
- Edges: connects output terminals to input terminals
  - Data flows along edges
  - All possible paths between template tasks expressed through edges
  - Represent sets of data



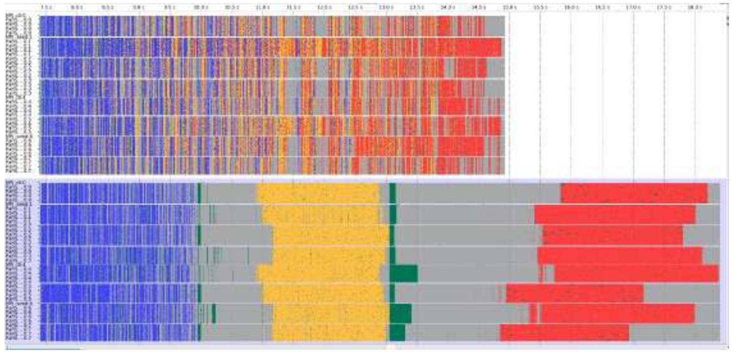
Task Graph Composition: POINV

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



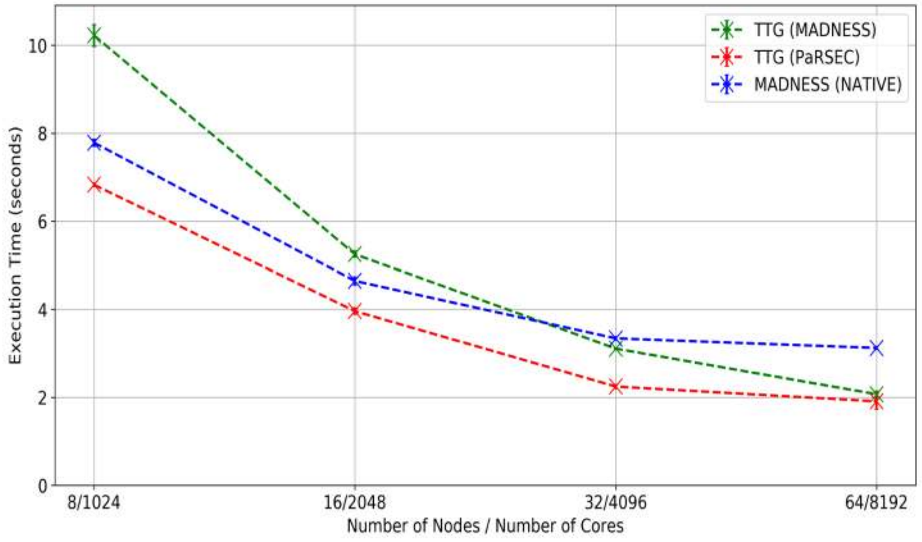


(a) Tile size 128.



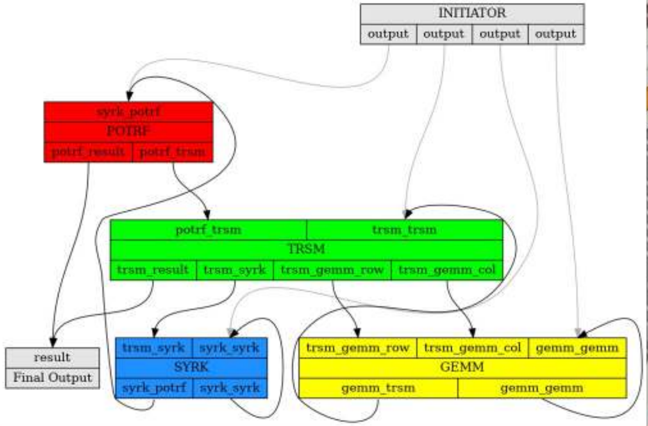
Target Applications: Multi-Resolution Analysis

- 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



ICL THE UNIVERSITY OF TENNESSEE KNOXVILLE

TTG: Small Example

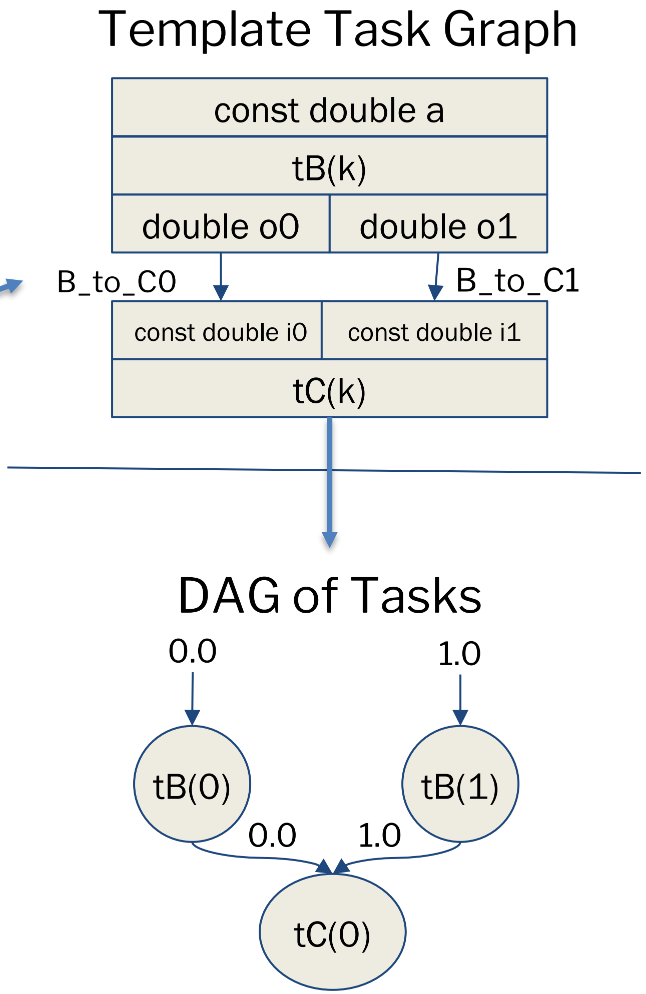
```
ttg::Edge<int, double> to_B("to_B");
ttg::Edge<int, double> B_to_C0("B_to_C0");
ttg::Edge<int, double> B_to_C1("B_to_C1");

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>(0, 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

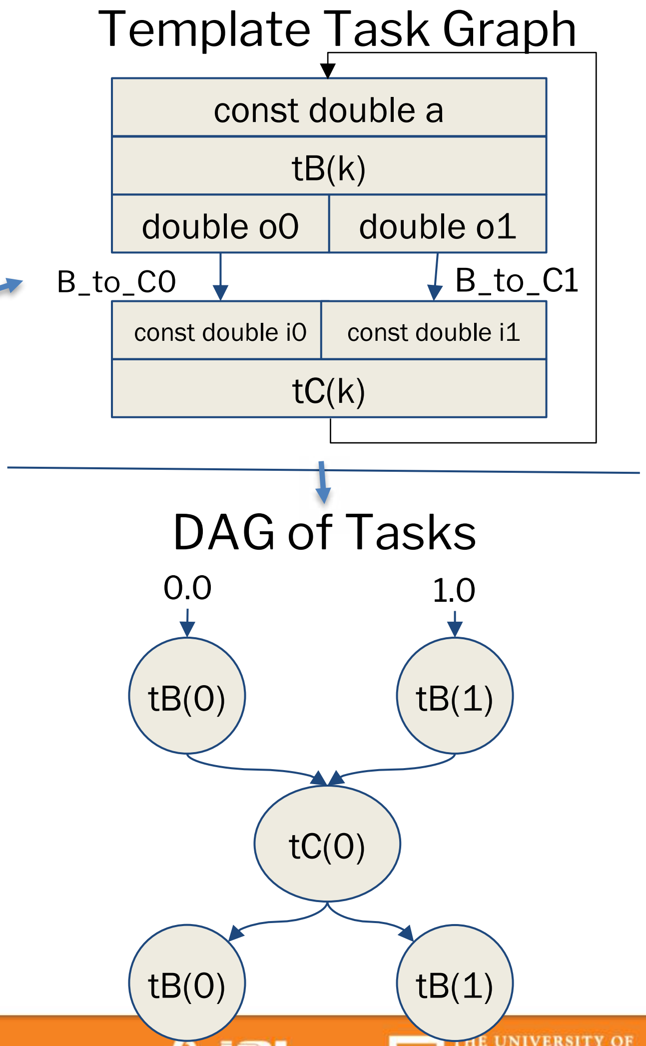


Simplifications work in progress

TTG: Small Example / Cycle

```
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");
ttg::Edge<int, double> C_to_B(”C_to_B");

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>(0, a);
  },
  ttg::edges(ttg::fuse(to_B, C_to_B),
  ttg::edges(B_to_C0, B_to_C1));

auto tc = ttg::make_tt([](const int &k, const double &i0, const double &i1)
  {
    if (need_recursion(i0, i1)) {
      ttg::send<0>(0, i0); // send i0 back to task B
      ttg::send<0>(1, i1); // send i1 back to task B
    }
  },
  ttg::edges(B_to_C0, B_to_C1),
  ttg::edges(C_to_B));

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();
```

Fused Edges



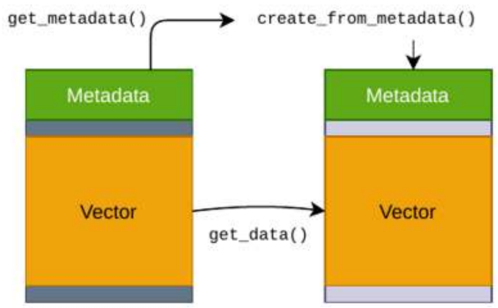
Task IDs and Process Mapping

- Tasks identified by hashable objects
  - Typically pairs, tuples, small structs
- Default mapping: round-robin
- Application may provide custom mapping of task IDs to processes
- Similar mechanism for task priorities

```
struct Key {
  int i, j;
  std::size_t hash() {
    return (i<<32) + j;
  }
};

Matrix A = ...; // matrix instance
auto tt = ttg::make_tt(...);
tt->set_procmap(
  [&](const Key& key){
    // map {i, j} coords to owner of tile
    return A.process_of({key.i, key.j});
  });

tt->set_priomap(
  [&](const Key& key){
    // generate priority for task {i, j}
    return priority_of(key.i, key.j);
  });
```

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



Const and Move Semantics

- Runtime tracks and reuses object copies wherever possible
- Const input parameters allow objects shared between tasks
- Move semantics signal that mutable objects are not mutated anymore
- Non-const inputs may require additional copies (except for single-use inputs)

```
// Sending mutable data
[](const int &k, T&& a) {
    mutate(a);              // updates a
    ttg::send<0>(k+1, a); // creates a new copy
    reset(a);               // a is mutated again
    ttg::send<1>(k+1, a); // create yet another copy
}

// Moving input data
[](const int &k, T&& a) {
    mutate(a); // update a
    ttg::send<0>(k+1, std::move(a)); // no new copy
}

// Forwarding const input data
[](const int &k, const T& a) {
    ttg::send<0>(k-1, a); // no new copy
    ttg::send<0>(k,   a); // no new copy
    ttg::send<0>(k+1, a); // no new copy
}

// Sending stack-based data
[](const int &k) {
    T a = new_obj(k);
    ttg::send<0>(k-1, std::move(a)); // potential copy
}
```

Send and Broadcast

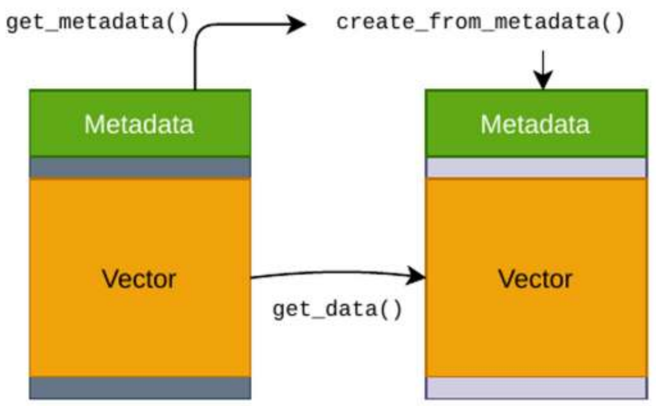
- Broadcasts provide single-statement data transfers to multiple successor tasks
- May send on one or more output terminals (i.e., along multipe edges)

```
// Sending mutable data to one successor
[](const int &k, T&& a) {
    mutate(a);            // updates a
    ttg::send<0>(k+1, std::move(a));
}

// Sending mutable data to multiple successors
[](const int &k, T&& a) {
    mutate(a);
    std::vector<int> broadcast_keys;
    for (int i = 0; i < num_successor; ++i) {
      broadcast_keys.push_back(k+i);
    }
    ttg::broadcast<0>(broadcast_keys, std::move(a));
}

// Sending mutable data to multiple successors on
different output terminals
[](const int &k, T&& a) {
    mutate(a);
    std::vector<int> broadcast0_keys;
    std::array<int, 3> broadcast1_keys = {k+1, k+2, k+3};
    for (int i = 0; i < num_successor; ++i) {
      broadcast0_keys.push_back(k+i);
    }
    ttg::broadcast<0,1>(broadcast0_keys,
                        broadcast1_keys,
                        std::move(a));
}
```

Zero-copy Data Movement

- Used for transfers between processes to avoid serialization

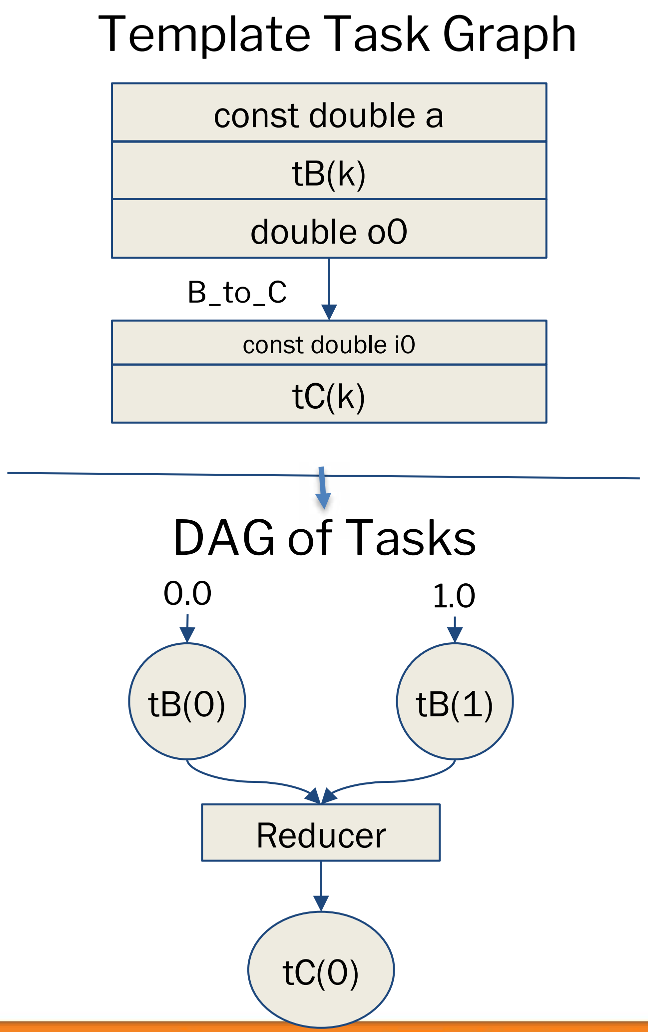
```
struct Tile {
  std::size_t m, n, lda;
  std::vector<double> data;
  struct metadata {
    std::size_t m, n, lda;
  };
  ... // ctors, dtors, accessors
};

template<typename T>
struct ttg::SplitMetadataDescriptor<Tile<T>> {
 // provide metadata
  auto get_metadata(const Tile<T>& t) {
    return Tile<T>::metadata{t.m(), tile.n(), tile.lda()};
  }
 // provide payload to be transferred
  auto get_data(Tile<T>& t) {
    return std::array<ttg::iovec, 1>{{t.size(), t.data()}};
  }
  // create an empty tile from metadata
  auto create_from_metadata(const typename Tile<T>::metadata& md) {
    return Tile<T>{md.m, md.n, md.lda};
  }
};
```



Reduction Terminals

- So far: one parameter per input
- Tasks may have a large number of inputs that can be reduced to a single value

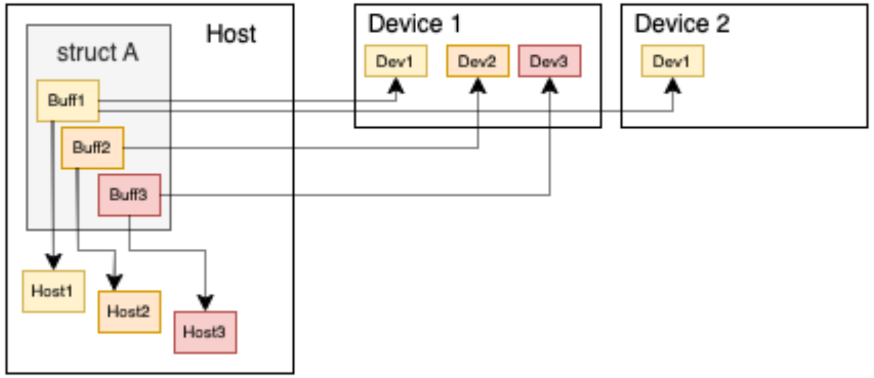
```
ttg::Edge<int, double> to_B("to_B");
ttg::Edge<int, double> B_to_C("B_to_C");

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

auto tc = ttg::make_tt([](const int &k, const double &i)
  {
    // Task tC(k) received sum of inputs: i
  },
  ttg::edges(B_to_C), ttg::edges());

// set reducer on input terminal 0: lhs = lhs ⊕ rhs
tc->template set_input_reducer<0>(
  [](double& lhs, const double& rhs){
    lhs += rhs;
  }, 2);
```



TTG Memory Model (Device)

- TTG manages device memory
- Host memory serves as backup for transparent eviction
  - Memory oversubscription supported by default
- Transparent data movement between devices and host
  - Automatic migration from device to host tasks
- ttg::Buffer: owning/non-owning host memory mirror

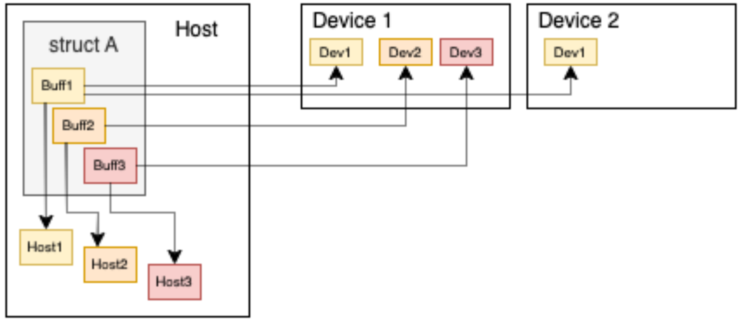
TTG Device integration still experimental



Buffers: Device memory containers

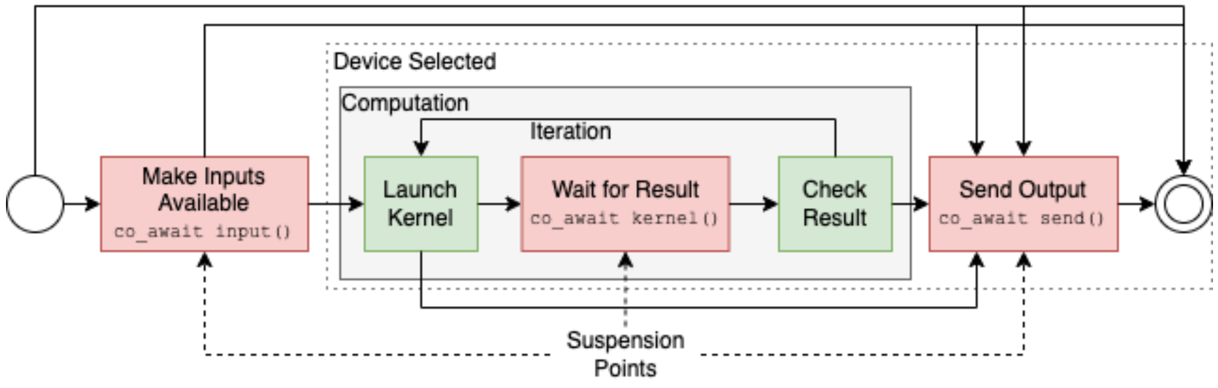
- Owns host memory, unless user-provided
- Tracks last device task location
- Enables transparent migration of data between devices and host
- Allows partial mapping of complex data structures to devices
  - Some tasks may not require all object data on the device

```
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
};
```



TTG Execution Model (Device)

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



TTG Device Tasks

- "co_await input()" selects device
- "co_await kernel()" accepts buffers/scratch to return to host
- Task may submit and wait for multiple kernels
- Sending outputs is last step of task

```
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
};
```

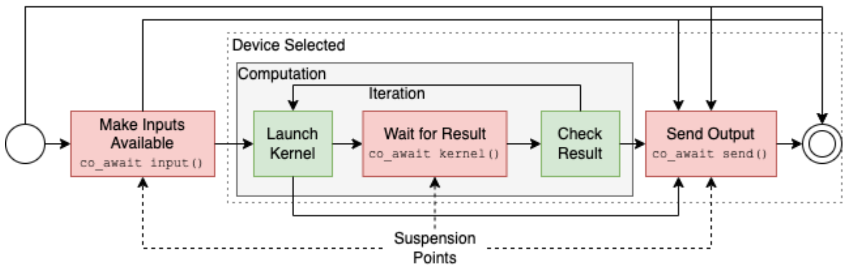
```
using Key = std::pair<int, int>; // tile position in matrix

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

Make inputs available

Submit work

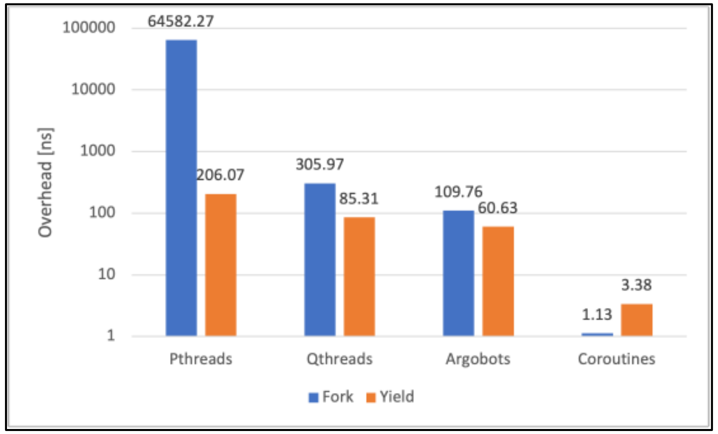
Wait for kernel to execute (optional)

Send Output

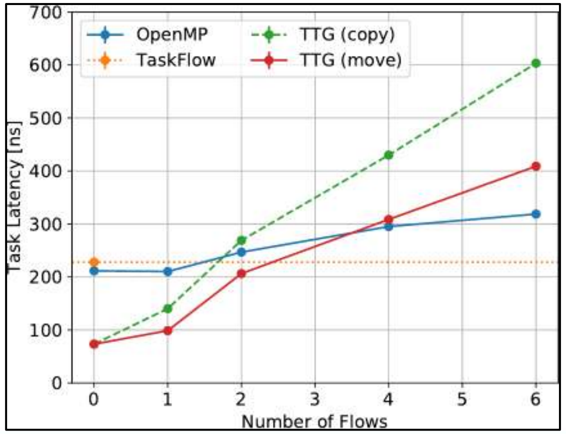


Why Coroutines?

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



Task Overhead



Coroutines vs Continuations

- Continuation-passing causes to code fragmentation
- Worse yet: careful handling of task-local state

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

auto tb = ttg::make_tt([](const Key& k, Tile&& a) {
    ttg::device::input(A.buf)
      .then(
        [](double *ptr){ return submit_kernel(ptr); }
      ).then(
        [&](){ ttg::send<0>(k, std::move(a)); }
      )
    );
  }, ...);
```

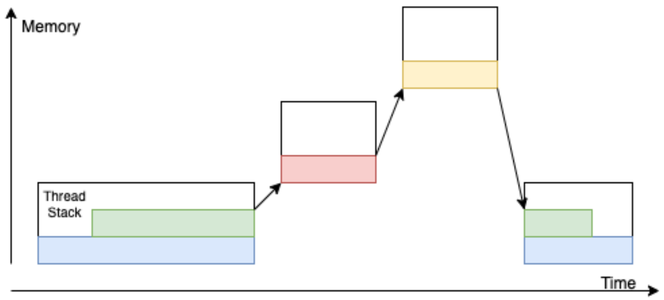
Reduced code fragmentation

```
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::input(A.buf); // make A available
    submit_kernel(a);
    co_await ttg::device::kernel(); // optional if no result required
    co_return ttg::device::send<0>(k, std::move(a));
  }, ...);
```

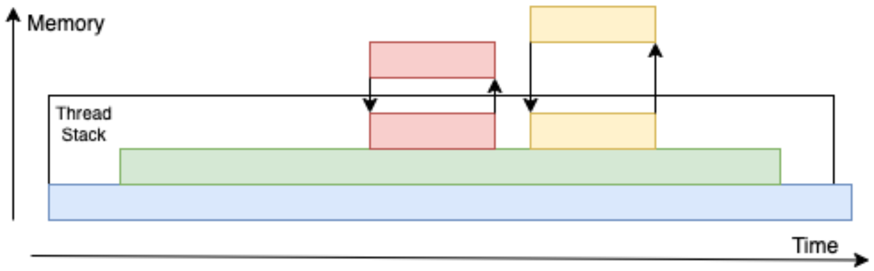
Coroutines vs Fibers/ULTs

- Fibers switch the stack of the executing thread
- Require allocation of stacks, likely unused
- Coroutines only allocate state saved between invocations and act like function calls

Fibers



Coroutines



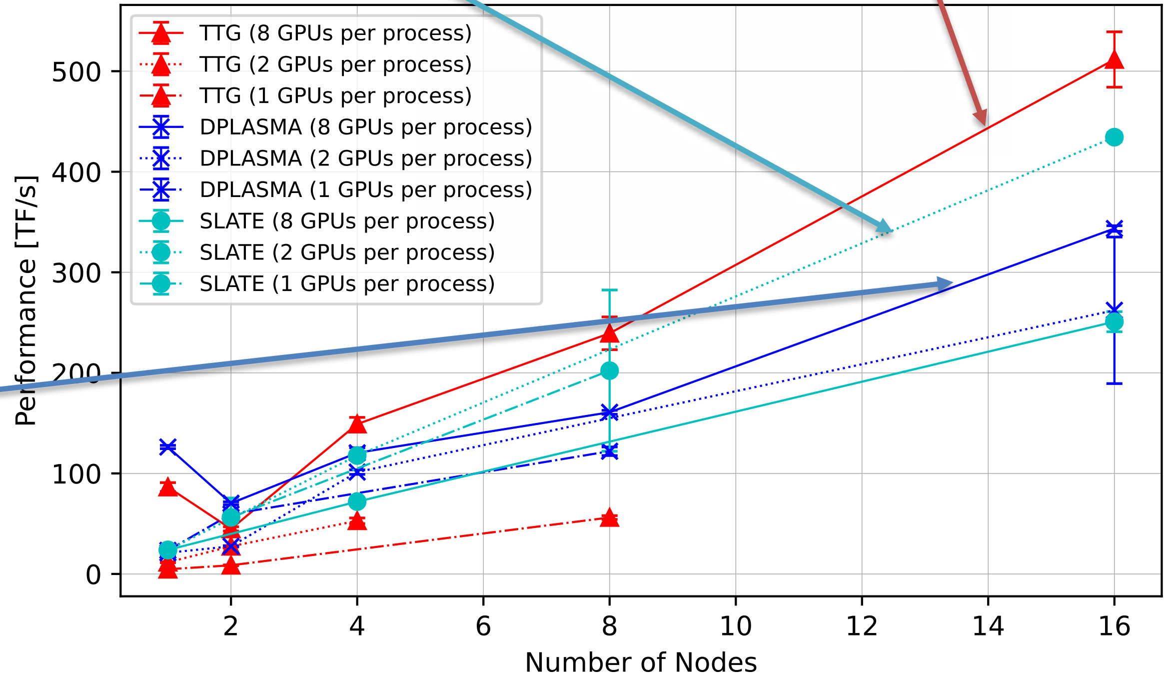
Preliminary Result: Dense GEMM

- Frontier: 8x MI250X per node
- Weak scaling: 32x32 1k tiles per GPU

SLATE: 2 GPUs per process

TTG: 8 GPUs per process

DPLASMA: 8 GPUs per process

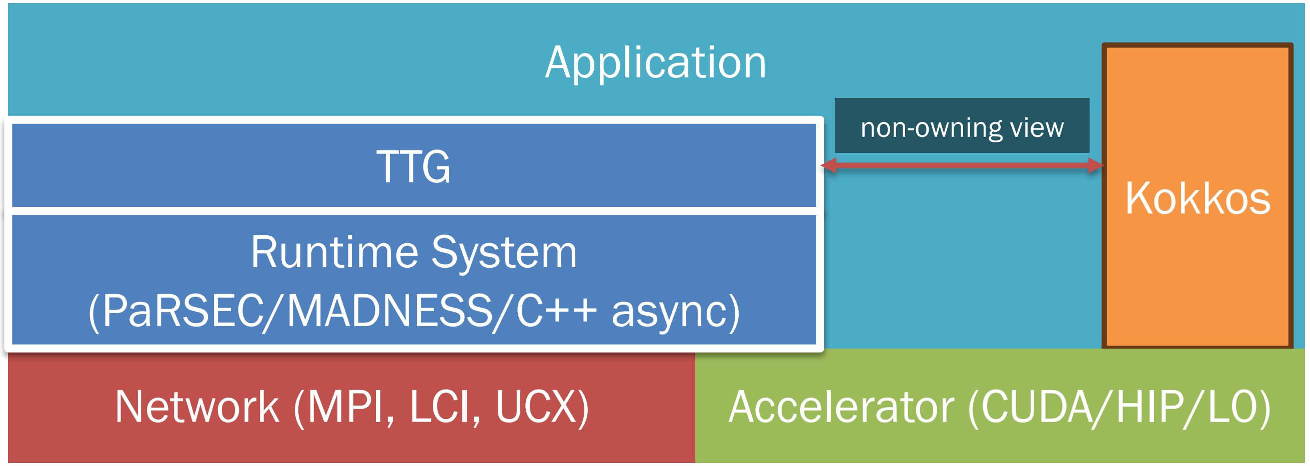


TTG shows good scaling but results are incomplete…

Kokkos and TTG

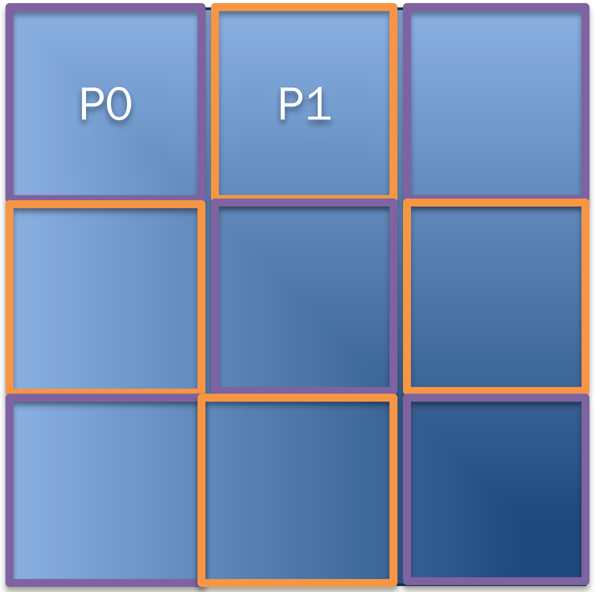
- TTG: distributed data-flow programming
  - No interest in providing task-level concurrency
- Integration point: non-owning Kokkos::View & execution environments
  - TTG manages device memory (ttg::buffer) and distributed execution
  - Kokkos provides accelerator programming infrastructure

Open challenge: Multi-GPU support in Kokkos



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

```
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));
  }, ...);
```

- Stride Layout
- CUDA w/ explicit stream
- MDRange
- Make tile memory available
- Submit Kokkos kernel
- Send to successor

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

Summary

- TTG provides scalable task discovery through abstract task graphs
- Coroutines simplify data and kernel management on the device
- Early results on GPUs promising

- Future Work:
  - Aggregation and pull terminals
  - Expanded use of coroutines (e.g., task ID generator in broadcasts)
  - Device-first memory allocation (on-demand host allocation)
  - Batched kernel tasks
  - Applications (SPMM, MRA, MADNESS/TA integration, ?)

Part 2: (Results of) Using Co-routines to Support Accelerators in TTG

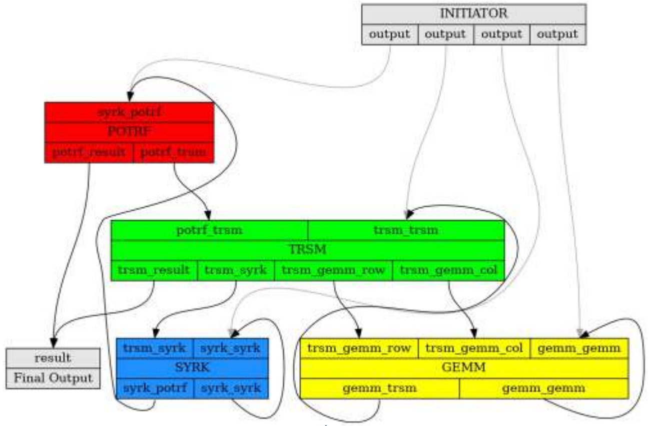
Joseph Schuchart
Stony Brook University (IACS)

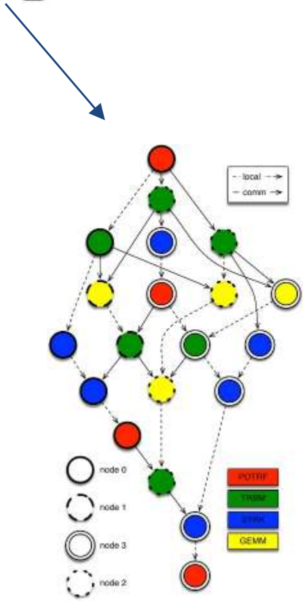
HiHAT Monthly Review
January 16, 2024

Recap

TTG: Overview

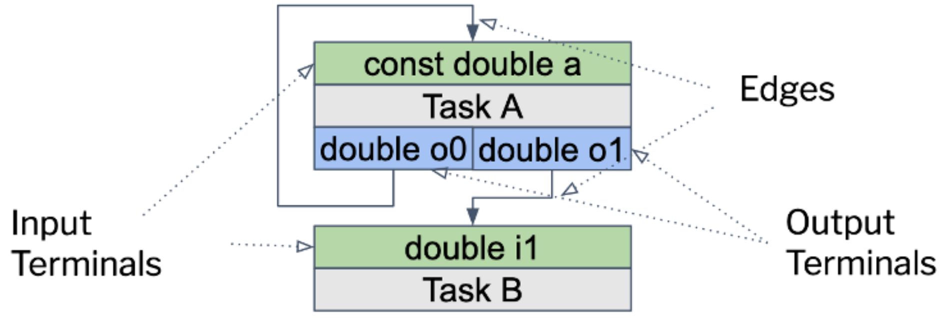
- Distributed Data Flow as Abstract Task Graph
  - May contain cycles
  - Nodes: template tasks
  - Edges: possible data flow between tasks
- Template Task Graph unrolled during execution
  - Tasks identified through (hashable) IDs (keys)
  - Data flows along edges as Pair {TaskID, Data}
- Data-dependent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





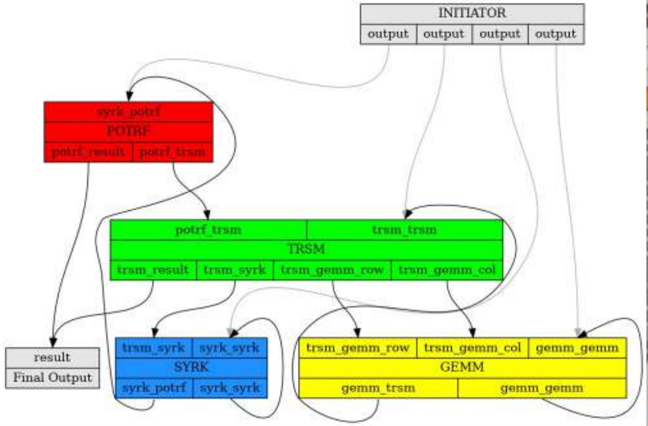
TTG: Tasks, Terminals, and Edges

- Tasks: task with set number of inputs and outputs
  - Instantiated when first discovered
  - Executed once all inputs are available
- Terminals: inputs and outputs of a task, hidden from user code
- Edges: connects output terminals to input terminals
  - Data flows along edges
  - All possible paths between template tasks expressed through edges
  - Represent sets of data



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



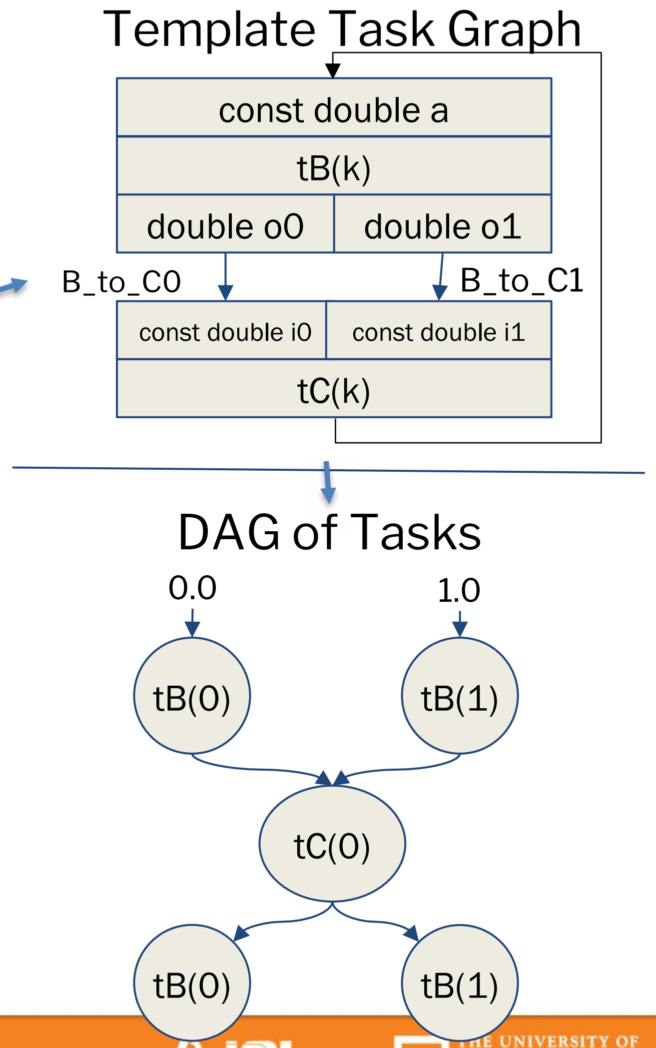
ICL
THE UNIVERSITY OF TENNESSEE KNOXVILLE

Simplifications work in progress

TG: Small Example / Cycle

```
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");
ttg::Edge<int, double> C_to_B(”C_to_B");

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>(0, a);
  },
  ttg::edges(ttg::fuse(to_B, C_to_B),
  ttg::edges(B_to_C0, B_to_C1));

auto tc = ttg::make_tt([](const int &k, const double &i0, const double &i1)
  {
    if (need_recursion(i0, i1)) {
      ttg::send<0>(0, i0); // send i0 back to task B
      ttg::send<0>(1, i1); // send i1 back to task B
    }
  },
  ttg::edges(B_to_C0, B_to_C1),
  ttg::edges(C_to_B));

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();
```

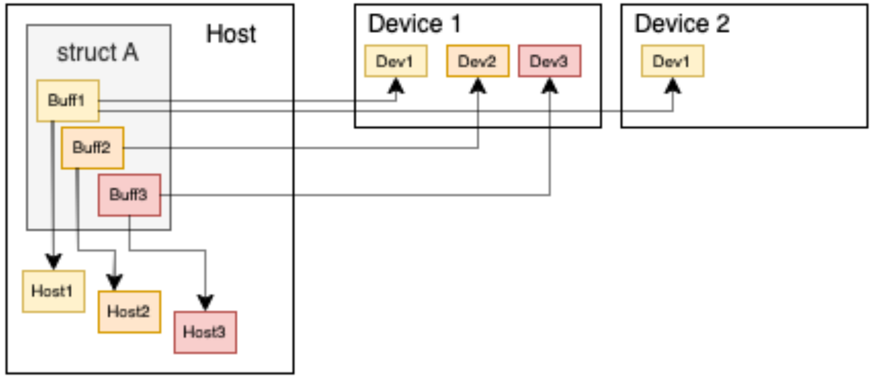
Fused Edges



TTG Memory Model (Device)

- TTG manages device memory
- Host memory serves as backup for transparent eviction
  - Memory oversubscription supported by default
- Transparent data movement between devices and host
  - Automatic migration from device to host tasks
- ttg::Buffer: owning/non-owning host memory mirror

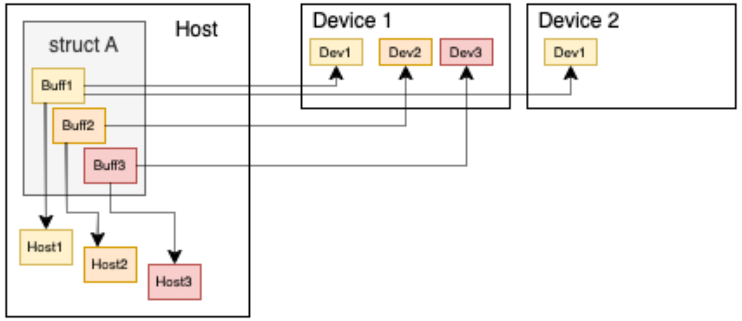
TTG Device integration still experimental



Buffers: Device memory containers

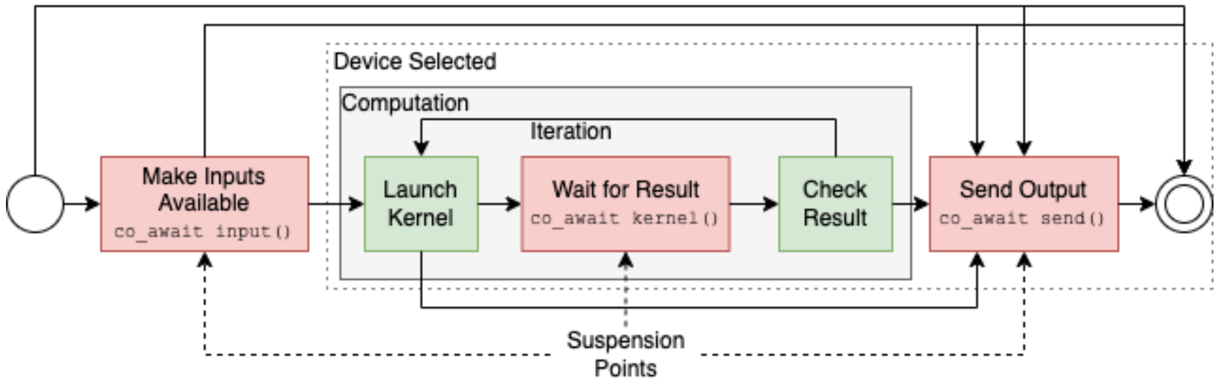
- Owns host memory, unless user-provided
- Tracks last device task location
- Enables transparent migration of data between devices and host
- Allows partial mapping of complex data structures to devices
  - Some tasks may not require all object data on the device

```
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
};
```



TTG Execution Model (Device)

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



TTG Device Tasks

- “co_await select()” selects device
- “co_await kernel()” waits for kernel and potential transfers back to host
- Task may submit and wait for multiple kernels
- Sending outputs is last step of task

```
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
};
```

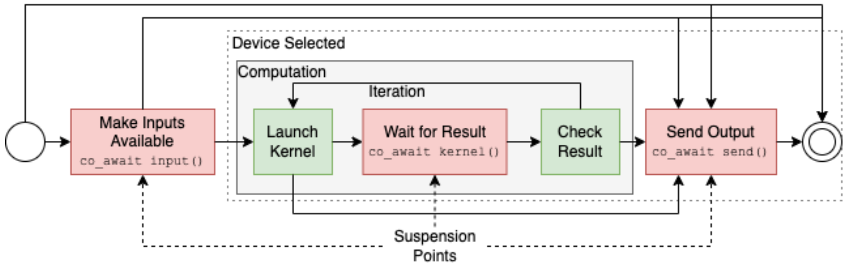
```
using Key = std::pair<int, int>; // tile position in matrix

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

Select device based on inputs

Submit work

Wait for kernel to execute (optional)

Send Output



ICL
THE UNIVERSITY OF TENNESSEE KNOXVILLE

Since the last meeting…

- Reworked part of PaRSEC's GPU integration
- Improved transparent handling of device copies
- Alas, issues running on Frontier beyond handful of nodes…

- Tool integration: generation of Perfetto traces

- Block sparse matrix multiplication

GPU Binding: POTRF

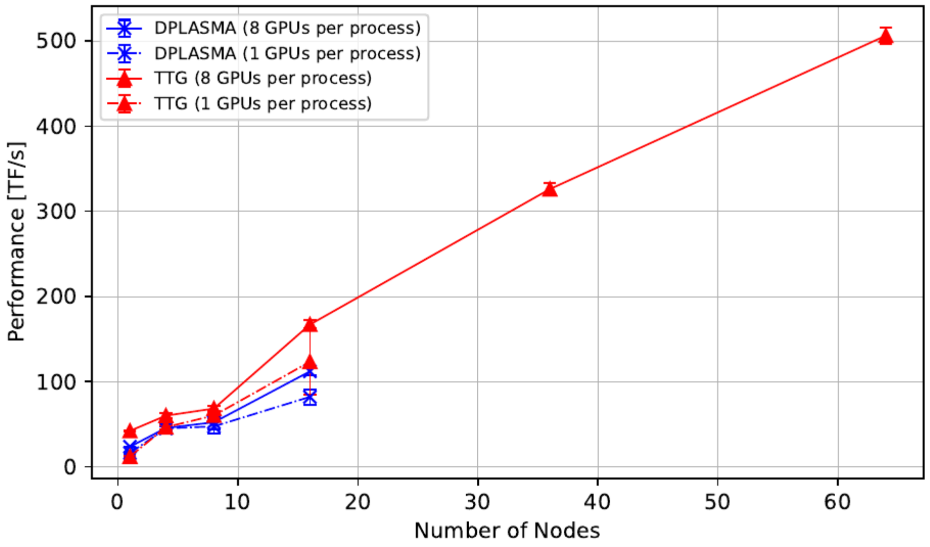
- Transparent multi-device scheduling
- Automatic load-balancing is great but not perfect
- Provide a hint to runtime which device to use
- Schedule tiles on row to same device

```
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::input(A.buf); // make A available
    submit_kernel(a);
    co_await ttg::device::kernel(); // optional if no result required
    co_return ttg::device::send<0>(k, std::move(a));
  }, ...);
tb->set_devicehint([](const Key& k){
    return (key[0] / A.P()) % ttg::device::num_devices();
  });
```

POTRF on Frontier

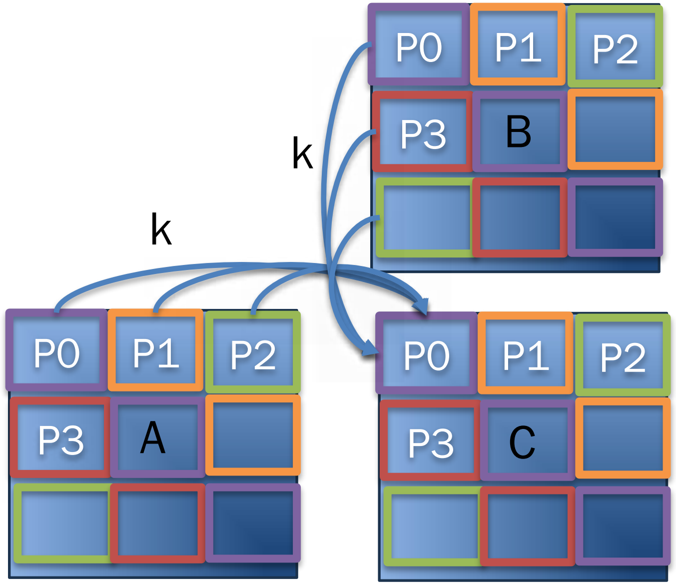
- Significant performance improvement over DPLASMA
- Issues running PaRSEC at scale on Frontier

64k / node, 2k tiles, DP



Towards BSpMM in TTG

- Input tiles flow A/B owners to C owner
- Challenge: throttling tile exchange in dataflow system
- Manual control flow (void Edges) error-prone and complex



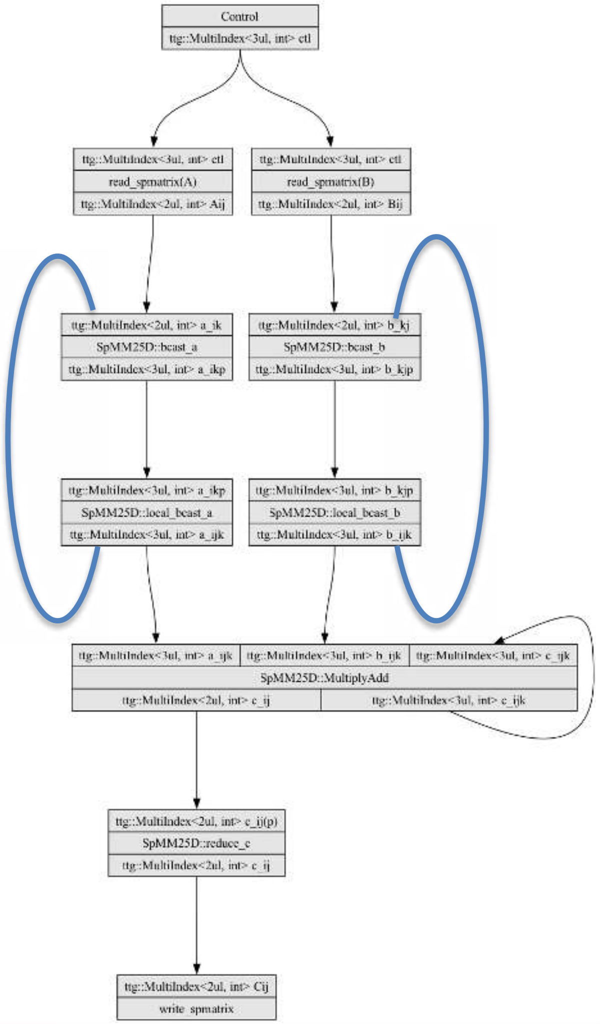
Throttling Data Exchange Through Constraints

- Constrain tile broadcasts based on previous k iterations
- Attach a `SequencedKeysConstraint` to the broadcast and release from GEMM
- Blocks [k+1…K] broadcasts until prior [0…k] broadcasts completed
- Mapper: key → sequence ID (here: k)
- **Auto release**: automatically release on prior k
- **Manual release:**
  GEMM releases k+1 bcasts once all GEMM of k are done

```
struct Key{ int m, n, k; }; // tile position in matrix
auto constraint = ttg::make_shared_constraint<ttg::SequencedKeysConstraint<Key>>();

auto bcast_a = make_A_broadcast(...);
auto bcast_b = make_B_broadcast(...);
bcast_a->add_constraint(constraint, [](const Key<2>& key){ return key.k; });
bcast_b->add_constraint(constraint, [](const Key<2>& key){ return key.k; });
```

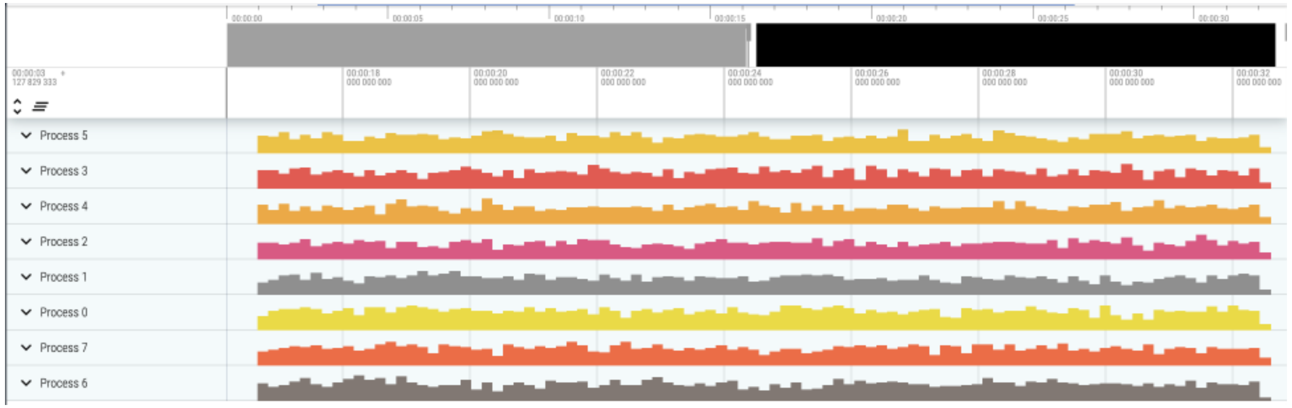
Automatic Constraint Release

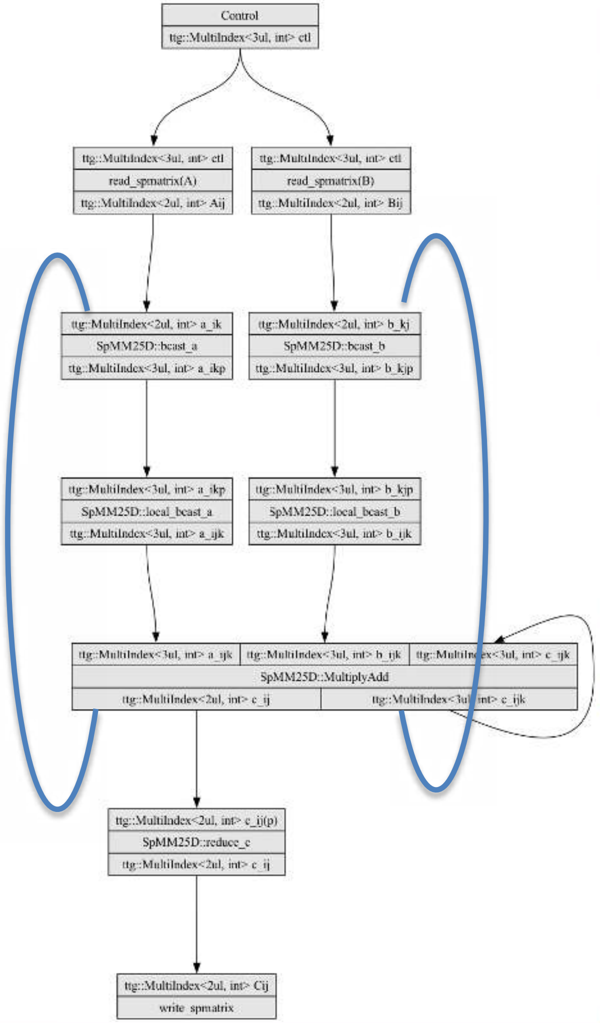
- Inserts local control to replace explicit control flow edges
- Global broadcasts implicitly depend on local broadcasts
- Still risk of flooding due to slower consumer



Manual Constraint Release

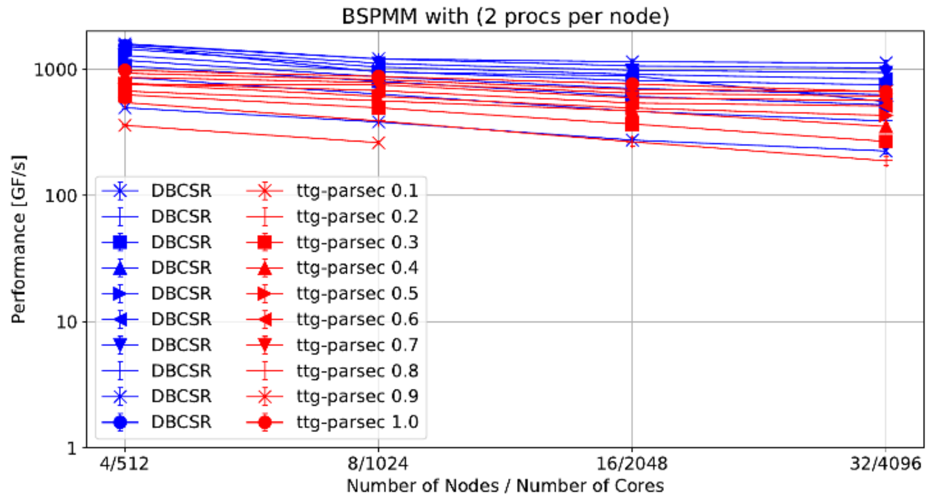
- Inserts local control to replace explicit control flow edges
- Global broadcasts released by previous k GEMM
  - Window of k to execute at a time
  - Overlap of broadcasts and GEMM
  - Counting broadcasts/GEMM for each k





BSpMM Results

- Performance on Hawk (2x64C AMD EPYC)
  - 64k matrix, 128 elements per tile
- Higher density yields better performance
- Performance not yet competitive with DBCSR



Also in the works…

- Coroutine-based key generators (for LA implementations)
- Improved memory allocation
  - Avoid host-side backup allocations
  - Integrate with application allocators
- Device to device communication
  - Source device enabled today (if MPI allows)
  - Target device will need restructuring of GPU/Comm backends

MADNESS Integration

- Implementation of Multi-Resolution Analysis with GPU support

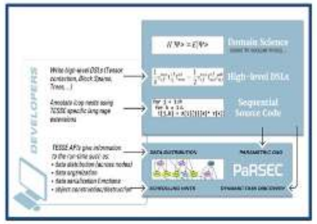

George Bosilca

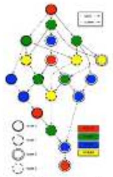

Thomas Herault

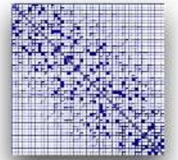

Joseph Schuchart

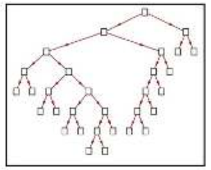

Eduard Valeev

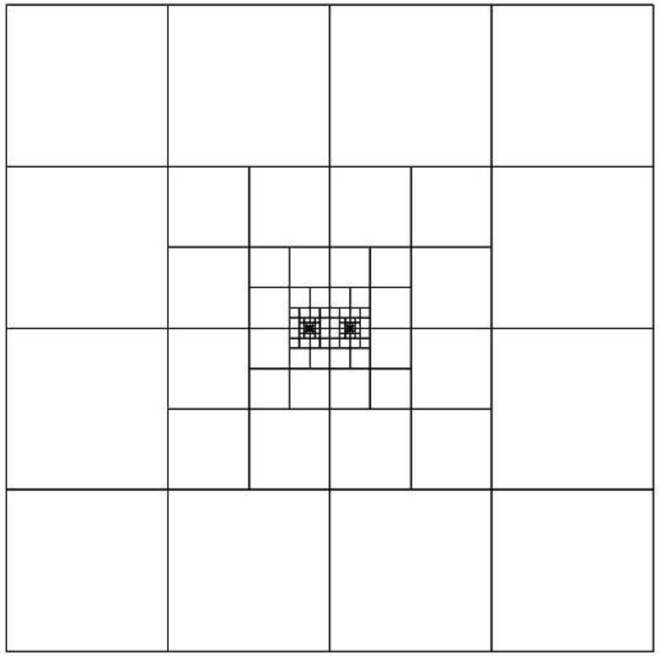

Robert Harrison




dense linear algebra


block-rank sparse algebra for quantum chemistry/physics


adaptive spectral-element calculus Multi-Resolution Analysis



- Slice thru grid used to represent the nuclear potential for H_2 using k=7 to a precision of 10^{-5}.
- Automatically adapts – it does not know a priori where the nuclei are.
- Nuclei at dyadic points on level 5 – refinement stops at level 8
- If were at non-dyadic points refinement continues (to level ??) but the precision is still guaranteed.
- In future will unevenly subdivide boxes to force nuclei to dyadic points.

Who we are

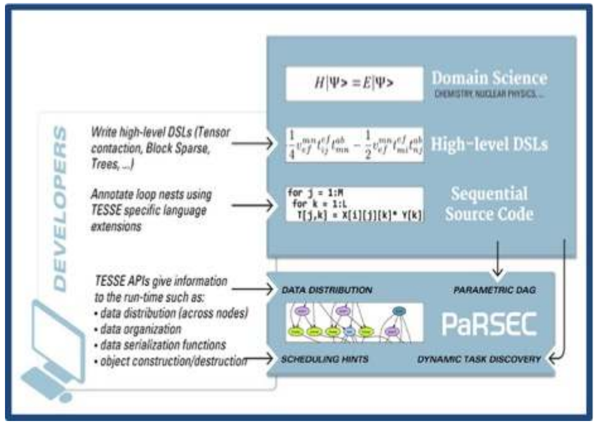
George Bosilca

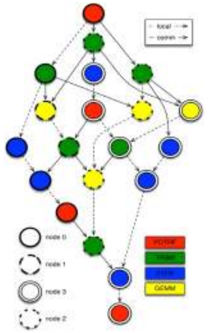
Thomas Herault

Joseph Schuchart

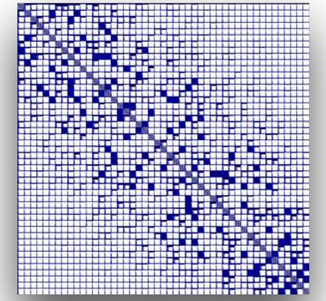
Eduard Valeev

Robert Harrison

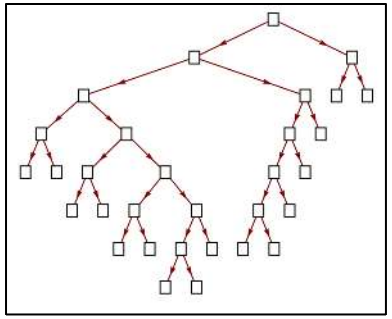




dense linear algebra



block-rank sparse algebra for quantum chemistry/physics



adaptive spectral-element calculus Multi-Resolution Analysis

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.

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.