Tutorial Material: https://bit.ly/TTG-tutorial

Template Task Graphs: An Introduction

ECP Tutorial Week, February 8, 2023

Thomas Herault
Joseph Schuchart

Collaborative work with G. Bosilca, E. Valeev, R.J. Harrison, P. Nookala

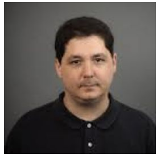
INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Tutorial Material:
https://bit.ly/TTG-tutorial

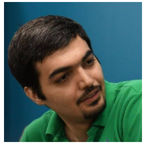
TTG: Product of EPEXA Collaboration

Objective: better composition of <u>irregular</u> algorithms on top of distributed-memory task runtimes (PaRSEC, MADNESS)

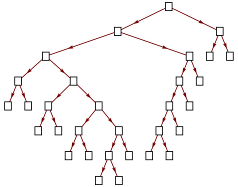




George Bosilca Thomas Herault Joseph Schuchart
ICL/UTK

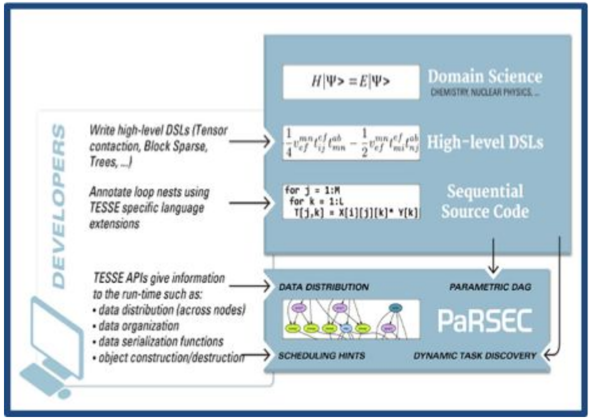




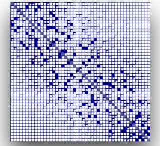
Robert Harrison Poornima Nookala M. Mahdi Javanmard
IACS/SBU IACS/SBU IACS/SBU



adaptive spectral-element calculus



Ed Valeev
VT



block-rank sparse algebra for quantum chemistry/physics

Tutorial Material:
https://bit.ly/TTG-tutorial

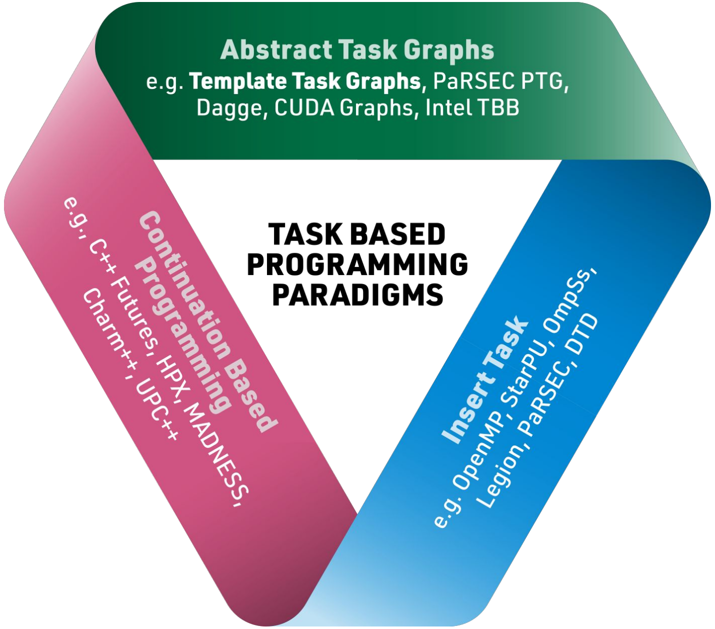
INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Task-Based Programming Interface

3 categories of task programming approaches:

- Insert Task
  - Sequential task discovery
  - Apparent data access order defines the DAG of tasks
- Continuation-Based Programming
  - Tasks become available when data is available in a future
- Abstract Task Graphs
  - Programmer defines an internal representation of the DAG of tasks



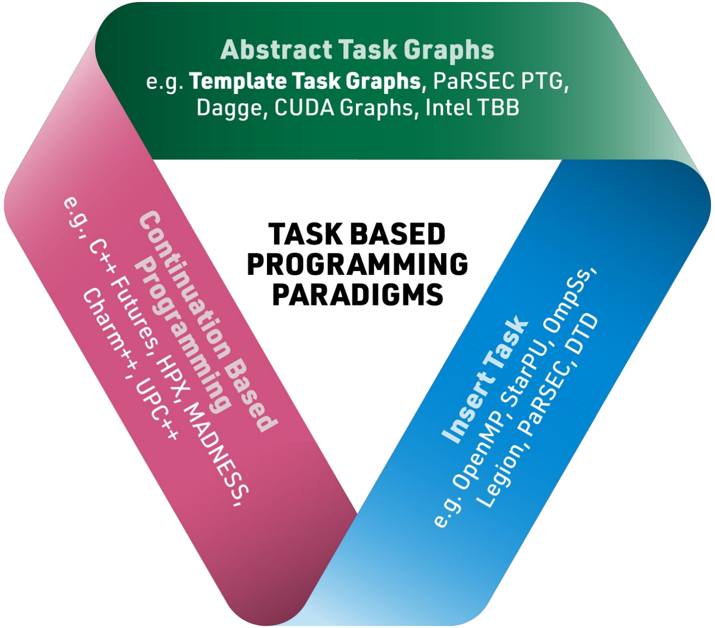
Tutorial Material:
https://bit.ly/TTG-tutorial

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Why TTG?

- Split graph setup from execution
  - Abstract Task Graph interface
  - Enables fully distributed DAG of task generation (and execution)
    - Any thread discovers new tasks
    - (Any thread can execute tasks, as in all task-based systems)
- Data-dependent graph traversal
  - Hard/Impossible in most ATG
- Hidden complexities of data movement / task result management



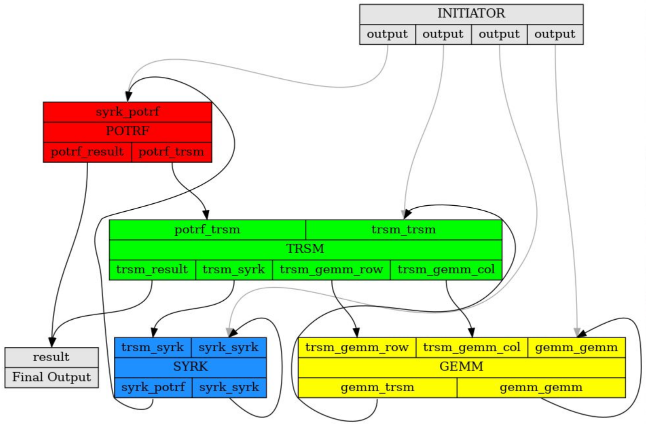
Tutorial Material:
https://bit.ly/TTG-tutorial

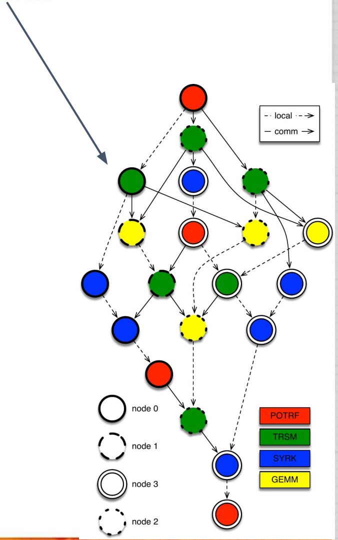
INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

TTG: Overview

- Abstract task graph unfolds into DAG during execution
  - **Template Tasks**: instantiated into tasks at execution
    - Represents large number of unique Tasks
    - A task is a template task + a task identifier + input data for that task
  - **Terminals**: input/output points of template tasks
  - **Edges**: connect input/output terminals
    - Represent **set of future values**
- Tasks **send** data to successors
- Multiple **backends**: PaRSEC, MADNESS



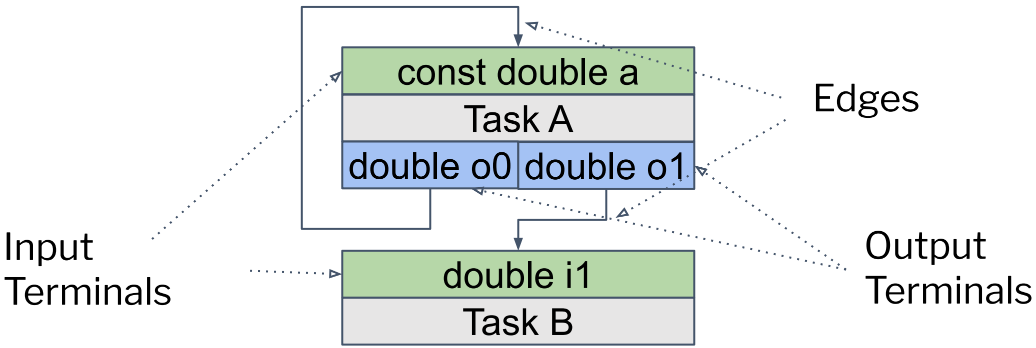


Tutorial Material:
https://bit.ly/TTG-tutorial

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Template Task Graphs: Visual Representation



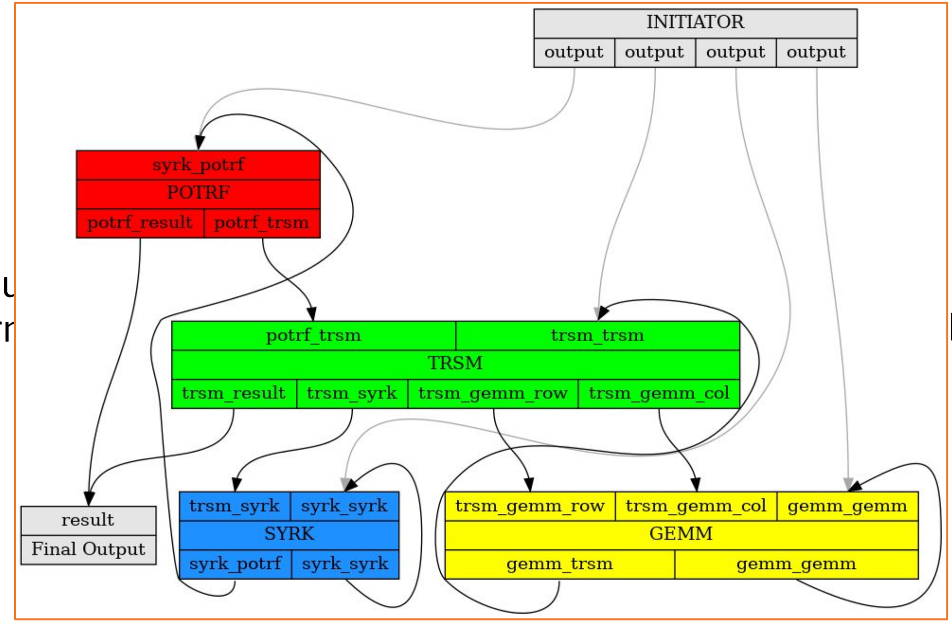
Tutorial Material:
https://bit.ly/TTG-tutorial

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Template Task Graphs: Visual Representation

Inpu
Terr



ls

Tutorial Material:
https://bit.ly/TTG-tutorial

TTG Execution Model

- SPMD model: all processes execute the same program
- `ttg::World` to query number of processes and ranks
  - Split processes between multiple 'worlds'
- Single or multiple entry points into the DAG
  - One or more processes kick off computation by feeding information into the task graph
- Worker threads execute (non-preemptive) tasks
  - And while doing so may discover new tasks (and/or fulfill input for existing tasks)
- Fence at the end of the application (ideally), or to synchronize with non-task-based computations
- Composition of Template Task Graphs using Edges (see later)

Tutorial Material:
https://bit.ly/TTG-tutorial

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

TTG Memory Model

- TTG runtime manages all data fed into the DAG
- Data moves transparently based on the placement of the task consuming the data (see later)
  - No explicit receives
  - Sent/Broadcast data is implicitly received by new tasks, as task input arguments
- Immutable data (const lvalue reference arguments – `const T &`) may be shared between tasks
- Mutable data (rvalue reference arguments – `T&&`) receive private copies, unless the source task moves the data in the send (depending on the runtime backend for TTG).

Tutorial Material:
https://bit.ly/TTG-tutorial

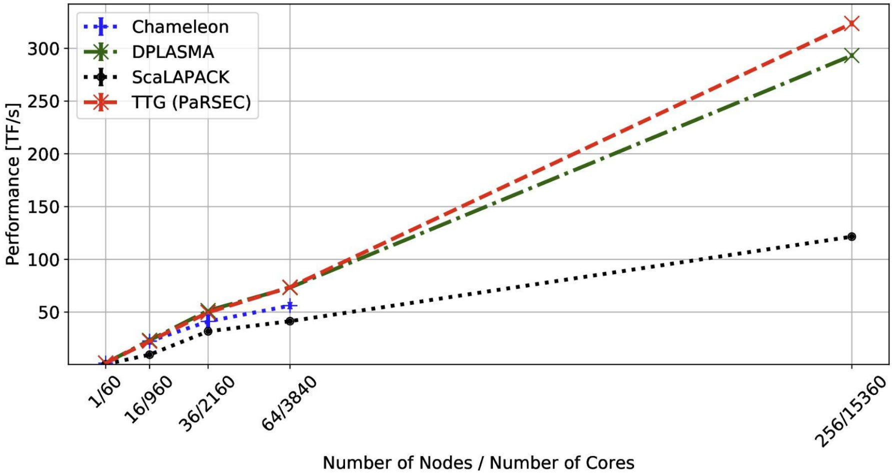
INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Cholesky Factorization: Weak Scaling

- Hawk, 1 – 256 nodes
- Matrix: 30k per node, tiles size 512

Performance of TTG matches DPLASMA



J. Schuchart, P. Nookala, M. M. Javanmard, T. Herault, E. F. Valeev, G. Bosilca, R. J. Harrison. Generalized Flow-Graph Programming Using Template Task-Graphs: Initial Implementation and Assessment. IPDPS, 2022.

Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

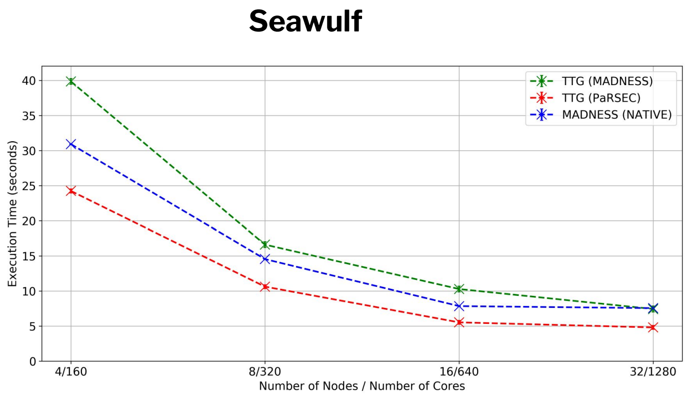
THE UNIVERSITY OF TENNESSEE KNOXVILLE

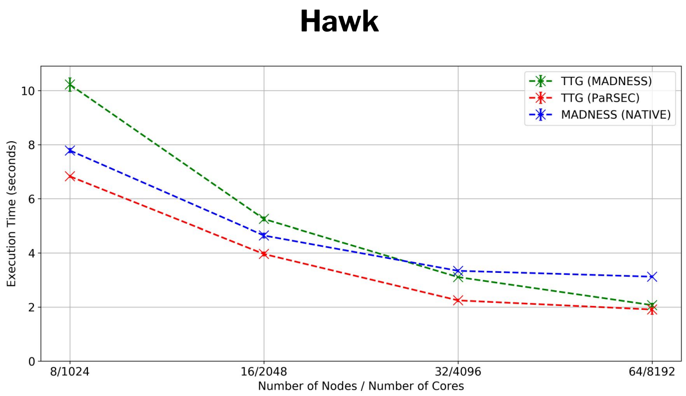
Multi-Resolution Analysis (MRA)

In Spectral-Element Calculus

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





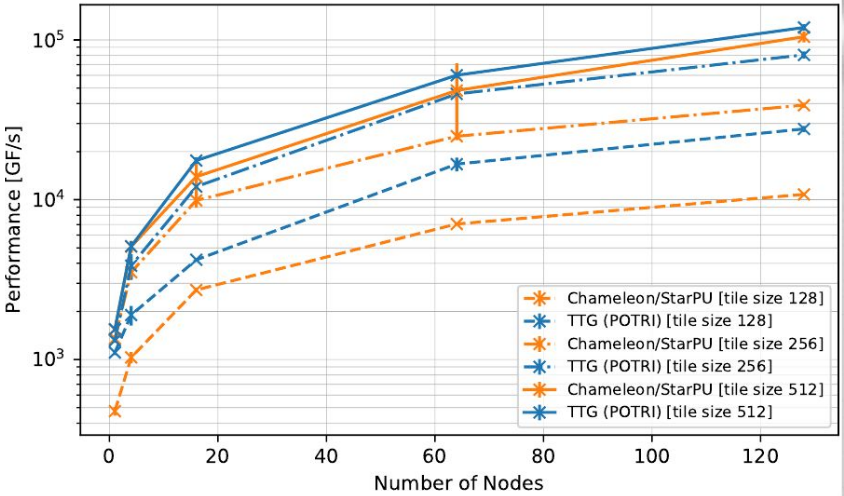
Tutorial Material:
https://bit.ly/TTG-tutorial

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Performance of POTRI (TRTRI+LAUUM)

- Composing graphs of two consecutive operations allows data to flow between the graphs without synchronization
- Results show performance benefits of graph composition and of the distributed task discovery (small tile sizes)



Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

TTG API: Basics Overview

`ttg::make_tt()`	Create a new Template Task with input/output edges

`ttg::send()`
`ttg::sendk()`
`ttg::sendv()`
	Send data along an edge (key+value, key only, value only)

`ttg::Edge<K, T>`	Edge connecting two Template Tasks with key type K and value type T (K and T may be void)
`ttg::edges()`	Collection of edges to pass to ttg::make_tt as input and output edes of TT
`ttg::make_executable()`	Check that all Template Tasks are connected and reachable; prepare for execution
`ttg::execute()`	Start execution of a Template Task Graph
`ttg::fence()`	Wait for completion of running graphs
`TT::invoke()`	Create a new task instance with given arguments
`ttg::print()`	Pretty print (avoids multithreaded garbled output, and adds some information like the rank)

Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

TTG API: Task Functions

Function object invoked once all inputs are satisfied.

Key is optional (for task templates with single task instance)

```
[=](const keyT &k) {
  ttg::print("This is task B(" , k, ")");
}
```

control flow (= flow of "void" data)

```
[=](const keyT &k, const T& val) {
  ttg::print("This is task B(" , k, ", " val, ")");
}
```

immutable data

```
[=](const keyT &k, T&& val) {
  ttg::print("This is task B(" , k, ", " val, ")");
}
```

mutable data

```
[=](const keyT &k, auto& val) {
  ttg::print("This is task B(" , k, ", " val, ")");
}
```

immutable generic data

```
[=](const keyT &k, auto&& val) {
  ttg::print("This is task B(" , k, ", " val, ")");
}
```

mutable generic data

Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Alternative 1: build your own programming environment

- Get TTG
  - `git clone https://github.com/TESSEorg/ttg.git`
  - `cd ttg`
  - `git checkout 73b59714461ae5b18307fec01cfac1ada30a108a`
- Pre-requisites:
  - C++-17 compiler (gcc>10, clang>10,icc…)
  - MPI, HWLOC, flex, bison, (Ninja), CMake (>3.20), git
  - For profiling capabilities: python3, pandas
  - To get all the tests: Eigen3, Boost, MKL
- Compile in a build directory (no compilation in source)
  - `mkdir build`
  - `cd build`
  - `cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=/usr [-DPARSEC_PROF_TRACE=ON -DPARSEC_PROF_GRAPHER=ON] [-DBUILD_TESTING=FALSE] [-G Ninja] ../`
  - `make (or ninja)`
  - `make install`

Tutorial Material:
https://bit.ly/TTG-tutorial

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Alternative 2: Use Docker

Get the docker from docker hub

```
> docker pull therault/ttg-tutorial-23-02-08:latest
```

Run it in interactive mode, with volume sharing

```
> docker run --name ttg-tutorial \
             -it --rm \
             -v $PWD/:/home/tesse/tutorial \
             therault/ttg-tutorial-23-02-08
tesse@dcede646b2c1:~$ ls
Dockerfile  ttg  tutorial  ubuntu-preseed.txt
tesse@dcede646b2c1:~$ ls tutorial
ECP-Feb-23
```

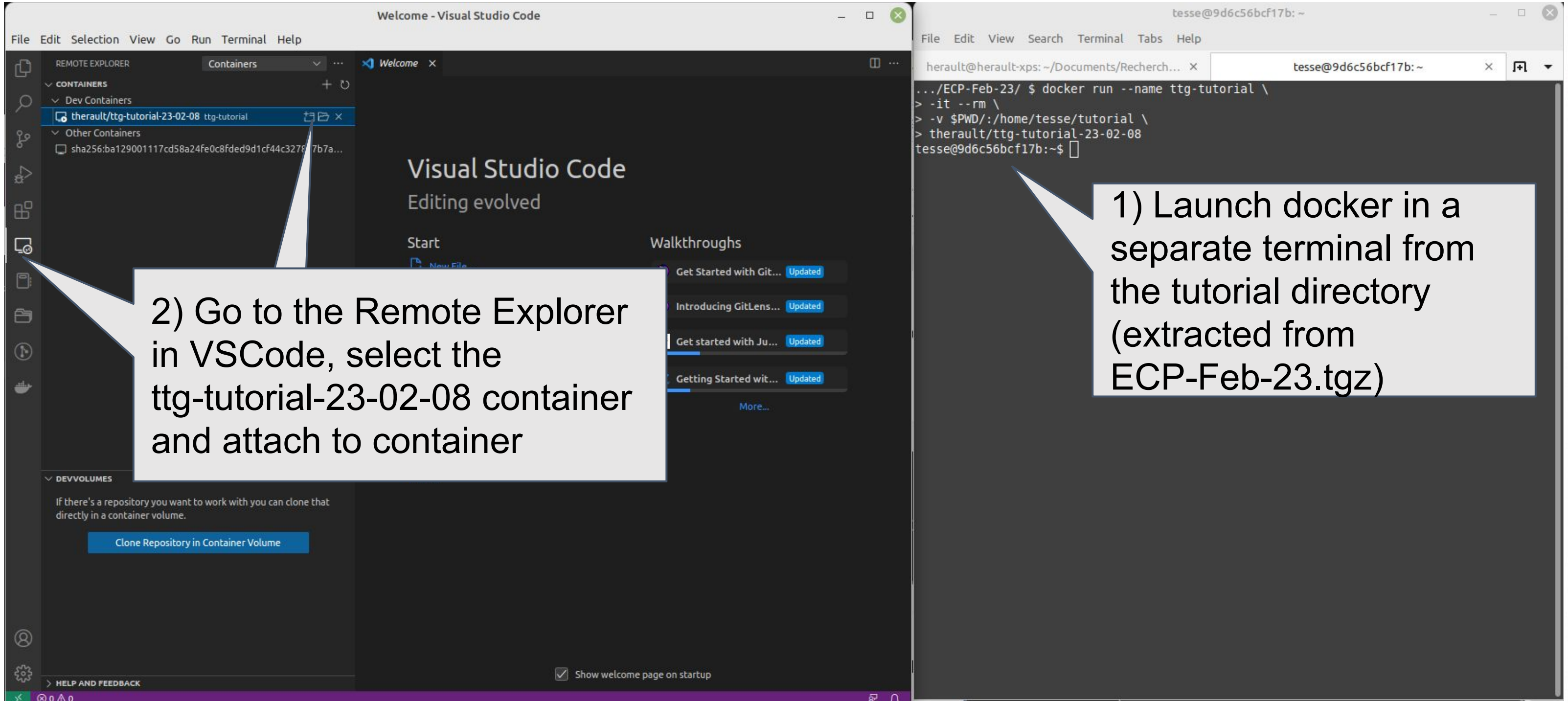
Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

VSCode Integration

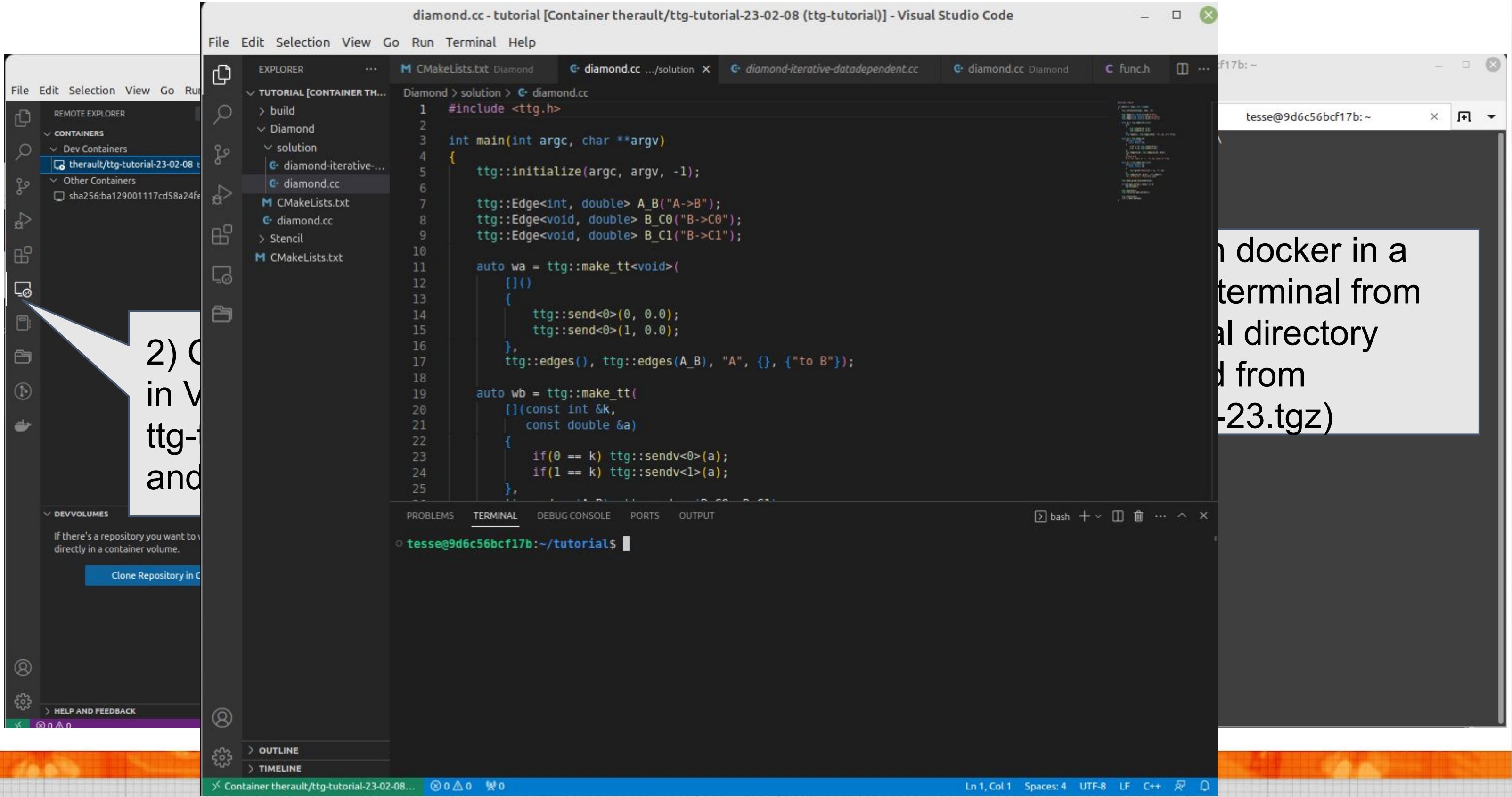
VSCode Addon: DevContainers



Tutorial Material:
https://bit.ly/TTG-tutorial

VSCode Integration

VSCode Addon: DevContainers



Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

First TTG program: single template task

```
auto tb = ttg::make_tt<int>([](const int &k) {
    ttg::print("This is task tB(" , k, ")");
  },
  ttg::edges(),
  ttg::edges(),
  "tB", {}, {});
ttg::make_graph_executable(tb);
```

Template Task Graph

tB(k)

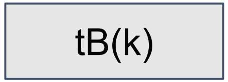
Tutorial Material:
https://bit.ly/TTG-tutorial

First TTG program: single template task

```
auto tb = ttg::make_tt<int>([](const int &k) {
    ttg::print("This is task tB(" , k, ")");
  },
  ttg::edges(), ttg::edges(),
  "tB", {}, {});
```

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

Template Task Graph

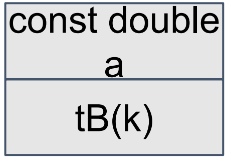


DAG of Tasks

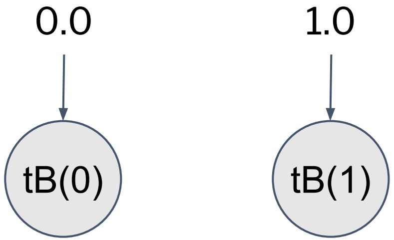
First TTG program: single template task

```
ttg::Edge<int, double> to_B("to_B");
auto tb = ttg::make_tt(
  [](const int &k, const double &a) {
     // This is task tB(k) it received value a for input a
  },
  ttg::edges(to_B), ttg::edges(),
  "tB", {"a"}, {});
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(tb->get_world());
```

Template Task Graph



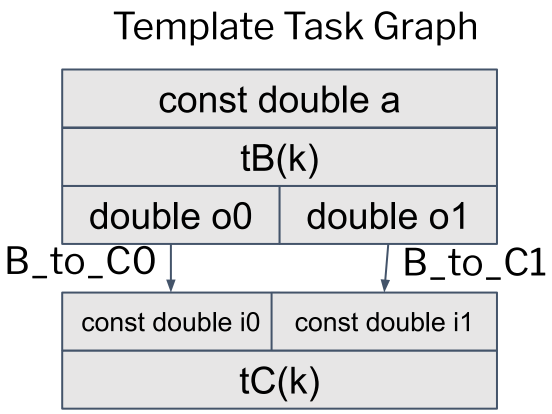
DAG of Tasks

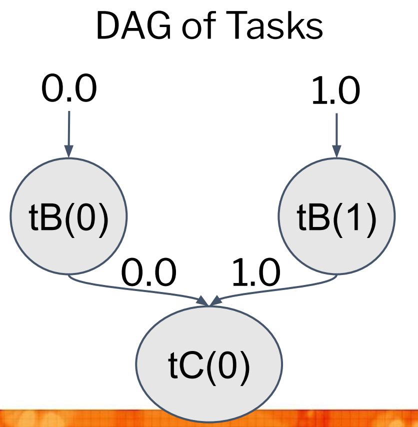


Tutorial Material:
https://bit.ly/TTG-tutorial

First TTG program: task to task

```
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) {
    // This is task tB(k) it received value a for input a
    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),
  "tB", {"a"}, {"o0", "o1"});
auto tc = ttg::make_tt([](const int &k, const double &i0, const double &i1)
  {
    // This is task tC(k) it received two inputs: i0 and i1
  },
  ttg::edges(B_to_C0, B_to_C1), ttg::edges(),
  "tC", {"i0", "i1"}, {});
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(tb->get_world());
```





Tutorial Material:
https://bit.ly/TTG-tutorial

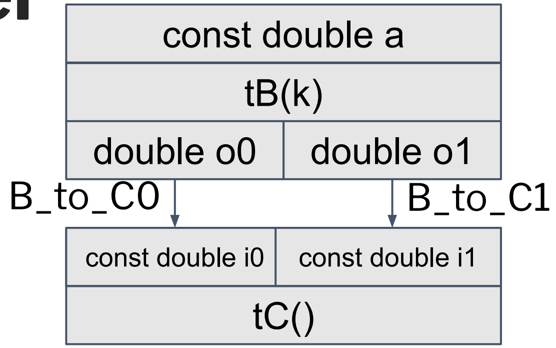
INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

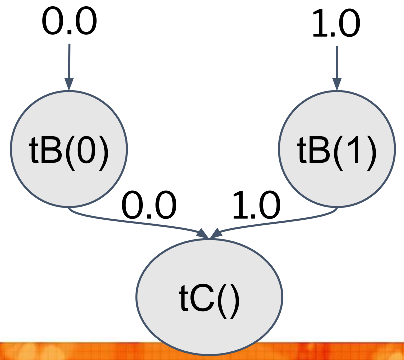
First TTG program:
C doesn't need a parameter

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

auto tb = ttg::make_tt( [](const int &k, const double &a) {
    // This is task tB(k) it received value a for input a
    if(0 == k) ttg::sendv<0>(a);
    if(1 == k) ttg::sendv<1>(a);
  }, ttg::edges(to_B), ttg::edges(B_to_C0, B_to_C1),
  "tB", {"a"}, {"o0", "o1"});
auto tc = ttg::make_tt<void>([](const double &i0, const double &i1) {
    // This is task tC() it received two inputs: i0 and i1
  }, ttg::edges(B_to_C0, B_to_C1), ttg::edges(),
  "tC", {"i0", "i1"}, {});
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(tb->get_world());
```

Template Task Graph



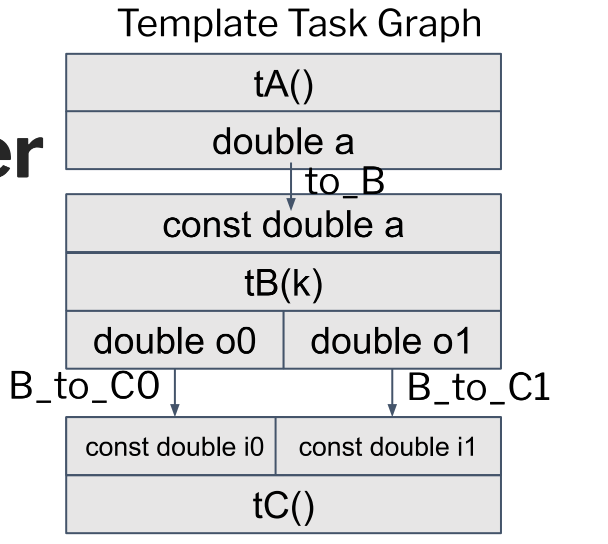
DAG of Tasks

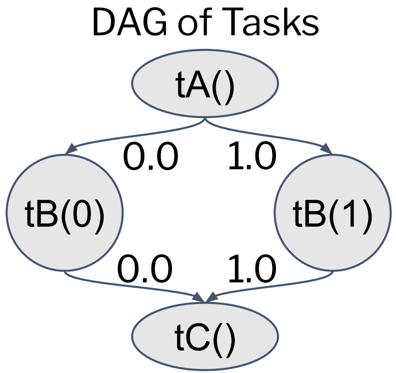


Tutorial Material:
https://bit.ly/TTG-tutorial

First TTG program:
C doesn't need a parameter

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

auto ta = ttg::make_tt<void>([]() {
    ttg::send<0>(0, 0.0);
    ttg::send<0>(1, 1.0);
  }, ttg::edges(), ttg::edges(to_B),   "tA", {}, {"a"});
auto tb = ttg::make_tt([=](const int &k, const double &a) { <...>
},
  ttg::edges(to_B), ttg::edges(B_to_C0, B_to_C1),
  "tB", {"a"}, {"o0", "o1"});
auto tc = ttg::make_tt<void>(
  [](const double &i0, const double &i1) { <...>  },
  ttg::edges(B_to_C0, B_to_C1), ttg::edges(),
  "tC", {"i0", "i1"}, {});
ttg::make_graph_executable(ta);
if(ta->get_world().rank() == 0) {
  ta->invoke();
}
ttg::execute();
ttg::fence(tb->get_world());
```





Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

First TTG program: full code

ttg-tutorial/Diamond/diamond.cc

```
#include <ttg.h>

int main(int argc, char **argv)
{
   ttg::initialize(argc, argv, -1);
   ttg::Edge<int, double> to_B("to_B");
   ttg::Edge<void, double> B_to_C0("B_to_C0");
   ttg::Edge<void, double> B_to_C1("B_to_C1");

   auto ta = ttg::make_tt<void>([]() {
         ttg::print("This is task tA(), it's sending data"
                    " to tB(0) and tB(1)");
         ttg::send<0>(0, 0.0);
         ttg::send<0>(1, 1.0);
      }, ttg::edges(), ttg::edges(to_B),  "tA", {}, {"a"});

   auto tb = ttg::make_tt([](const int &k,
                             const double &a) {
         ttg::print("This is task tB(" , k, "),"
                    " it received input ", a);
         if(0 == k) ttg::sendv<0>(a);
         if(1 == k) ttg::sendv<1>(a);
      },
      ttg::edges(to_B),
      ttg::edges(B_to_C0, B_to_C1),
      "tB", {"a"}, {"o0", "o1"});
```

ttg-tutorial/Diamond/diamond.cc (cont.)

```
   auto tc = ttg::make_tt<void>([](const double &i0,
                                   const double &i1) {
           ttg::print("This is task tC() it received"
                      " two inputs: ", i0, " and ", i1);
       },
       ttg::edges(B_to_C0, B_to_C1), ttg::edges(),
       "tC", {"i0", "i1"}, {});

   ttg::make_graph_executable(ta);

   if(ta->get_world().rank() == 0) {
       ta->invoke();
   }
   ttg::execute();
   ttg::fence(tb->get_world());

   ttg::finalize();
   return EXIT_SUCCESS;
}
```

Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

First TTG program: how to compile

ttg-tutorial/intro/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.19)

project(TTG-Example CXX)


find_package(ttg REQUIRED)


add_executable(diamond-parsec diamond.cc)

target_compile_definitions(diamond-parsec PRIVATE
                           TTG_USE_PARSEC=1)

target_link_libraries(diamond-parsec PRIVATE ttg-parsec)


add_executable(diamond-mad diamond.cc)

target_compile_definitions(diamond-mad PRIVATE
TTG_USE_MADNESS=1)

target_link_libraries(diamond-mad PRIVATE ttg-mad)
```

```
> cd ttg-tutorial/intro
> mkdir build
> cd build
> cmake -G Ninja ../
> ninja
> ./diamond-parsec
> ./diamond-mad
```

Tutorial Material:
https://bit.ly/TTG-tutorial

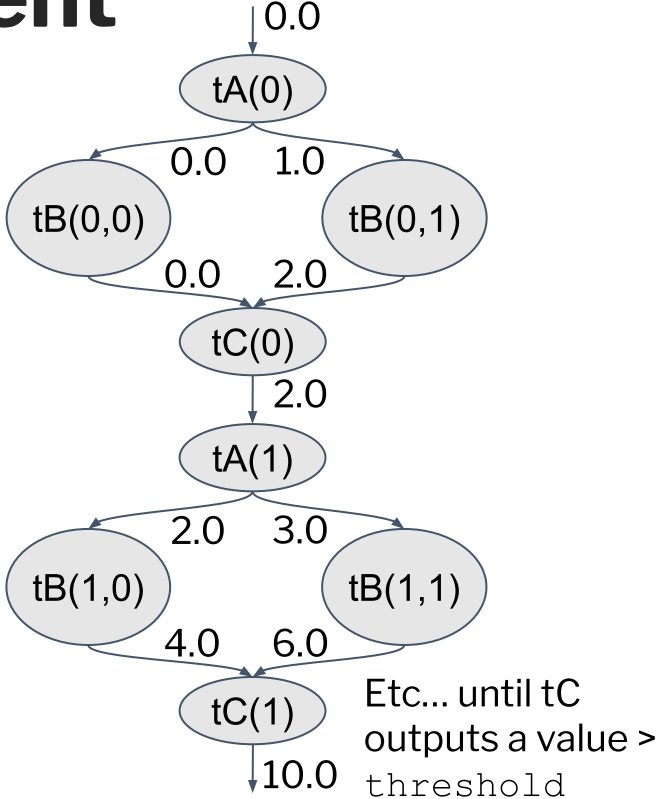
ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Hands-on: Extend first example to become data-dependent

Transform the first example so it becomes iterative and iterates until a threshold is reached.

- Add a new parameter to each tasks, which denotes the iteration number
  - Tasks of type tB now take a std::pair<int, int> as keys
  - Tasks of type tA and tC take a single int
- Tasks of type tA receive an input of type double
  - They send that input to tB(iteration, 0), and input+1.0 to tB(iteration, 1)
- Tasks of type tB double their input and send them to tC
- Tasks of type tC sum their inputs, and if it is lower than threshold (a variable defined in the main program), continue the iteration



Tutorial Material:
https://bit.ly/TTG-tutorial

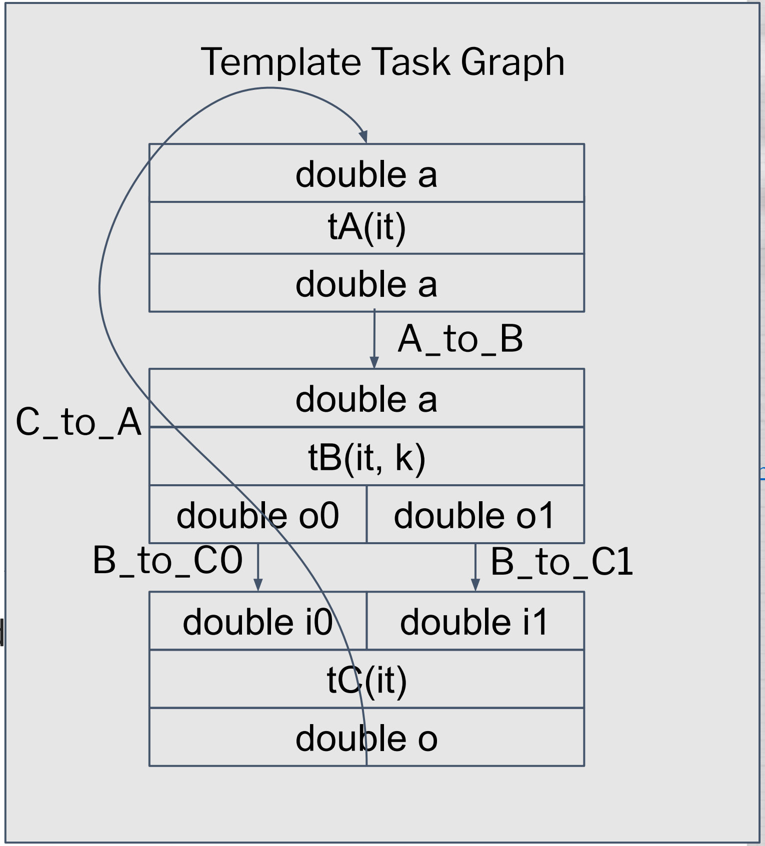
INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Hands-on: Extend first example to become data-depen

Transform the first example so it becomes iterative and iterates until a threshold is reached.

- Add a new parameter to each tasks, which denotes the iteration number
  - Tasks of type tB now take a std::pair<int, int> as keys
  - Tasks of type tA and tC take a single int
- Tasks of type tA receive an input of type double
  - They send that input to tB(iteration, 0) and input+1.0 to tB(iteration, 1)
- Tasks of type tB double their input and send them to tC
- Tasks of type tC sum their inputs, and if it is lower than threshold (a variable defined in the main program), continue the iteration



Tutorial Material:
https://bit.ly/TTG-tutorial

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Data-Dependent Diamond

ttg-tutorial/Diamond/diamond.cc

```
#include <ttg.h>
using Key2 = std::pair<int, int>;
namespace std {
 std::ostream &operator<<(std::ostream &os,
                          const Key2 &key) {
   os << "{" << std::get<0>(key) << ", "
      << std::get<1>(key) << "}";
   return os;
 }
} // namespace std

int main(int argc, char **argv) {
   ttg::initialize(argc, argv, -1);
   ttg::Edge<Key2, double> A_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_A("C_to_A");

   double threshold = 100.0;

   auto ta = ttg::make_tt([](const int &it,
                             const double &a) {
     // This is task tA(it), sending a to tB(it, 0)
     // and  a+1.0 to tB(it, 1)
     ttg::send<0>(Key2{it, 0}, a);
     ttg::send<0>(Key2{it, 1}, a+1.0);
   }, ttg::edges(C_to_A), ttg::edges(A_to_B),
      "tA", {"from C"}, {"to B"});
```

ttg-tutorial/Diamond/diamond.cc (cont.)

```
   auto tb = ttg::make_tt([](const Key2 &k,
                             const double &a) {
     // This is task tB(k), it received input a
     if(0 == std::get<1>(k)) ttg::send<0>(std::get<0>(k), 2.0*a);
     if(1 == std::get<1>(k)) ttg::send<1>(std::get<0>(k), 2.0*a);
   }, ttg::edges(A_to_B), ttg::edges(B_to_C0, B_to_C1),
      "tB", {"a"}, {"o0", "o1"});

   auto tc = ttg::make_tt([=](const int &it,
                          const double &i0, const double &i1) {
     // This is task tC(it) it received i0 and i1
     if(i0+i1 < threshold) ttg::send<0>(it+1, i0+i1);
   }, ttg::edges(B_to_C0, B_to_C1), ttg::edges(C_to_A),
      "tC", {"i0", "i1"}, {"C to A"});

   ttg::make_graph_executable(ta);

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

   ttg::finalize();
   return EXIT_SUCCESS;
}
```

Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Template Task Graph Extended Features

Tutorial Material:
https://bit.ly/TTG-tutorial

Custom Key Types

Arbitrary types can be used to identify tasks. They need to be **equality-comparable** and **hashable**.

By default, key hashes are used to determine the process to execute on (more later).

Custom types should provide a custom hash function.

The PaRSEC backend will warn about too many collisions.

```
struct Key {
  int32_t x;
  int16_t y;
  bool operator==(const Key& other)
const{
    return other.x == x && other.y == y;
  }
  size_t hash() const {
    return ((int64_t)x << 32) ^ y;
  }
};
```

Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

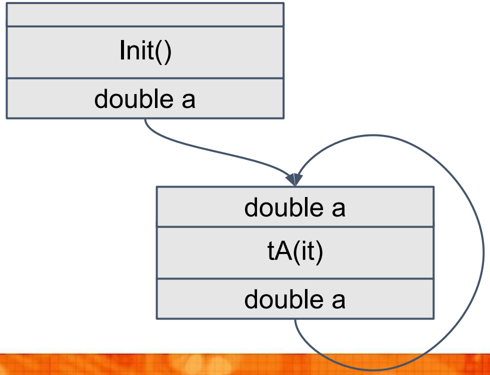
THE UNIVERSITY OF TENNESSEE KNOXVILLE

Fusing Edges

Inputs to a terminal can come from multiple edges, but only one at a time.

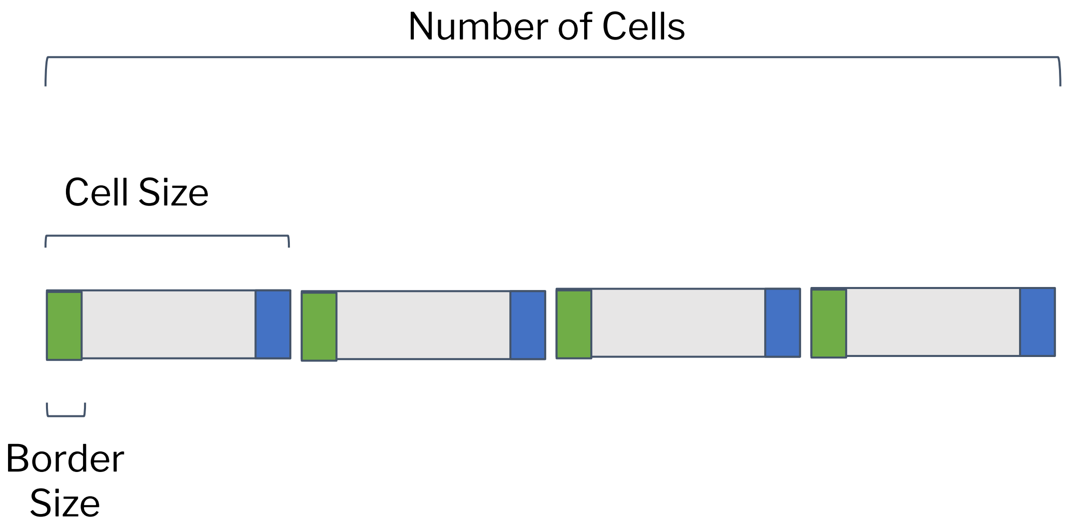
Edges can be fused into a virtual edge, each satisfying the input. More than one input for a task is erroneous.

```
ttg::Edge<int, double> A_A("A->A");
ttg::Edge<int, double> I_A("Init->A");

ttg::Edge<int, double>  a_in = ttg::fuse(A_A, I_A);

auto init = ttg::make_tt([](const int it,
                            const double &a) {
    ...
  }, ttg::edges(), ttg::edges(I_A), "Init", ...);

auto wa = ttg::make_tt([](const int it,
                          const double &a) {
    ...
  }, ttg::edges( a_in), ttg::edges(A_A), "A", ...);
```



Tutorial Material:
https://bit.ly/TTG-tutorial

Stencil Cells



Tutorial Material:
https://bit.ly/TTG-tutorial

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Hands On: 1D Stencil

A simple 1D stencil with 3 template tasks: Init, Update, Result. Cell size and count is configurable (see -h).
One task per cell.

First task: Add the `ttg::send` calls to the Init and Update tasks. `myLeft` border should be sent on the `I_L` and `V_L` edges, `myRight` on the `I_R` and `V_R` edges.

Reminder:

```
ttg::send<TerminalId>(key, std::move(data));
```

Where the keys are

`left_k, next_k, right_k`

for the left, center, and right successors.

Build and run:

```
$ cmake ../
$ make -C ../Stencil/Task1
$ mpirun -n 2 Stencil/Task1/stencil1-parsec
```

Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Process Placement & Priorities

By default, tasks are mapped round-robin to processes based on the hash of the key.

Process placement can be controlled through the **keymap** of a TT

Similar mechanism to set priorities of tasks using **priomap**

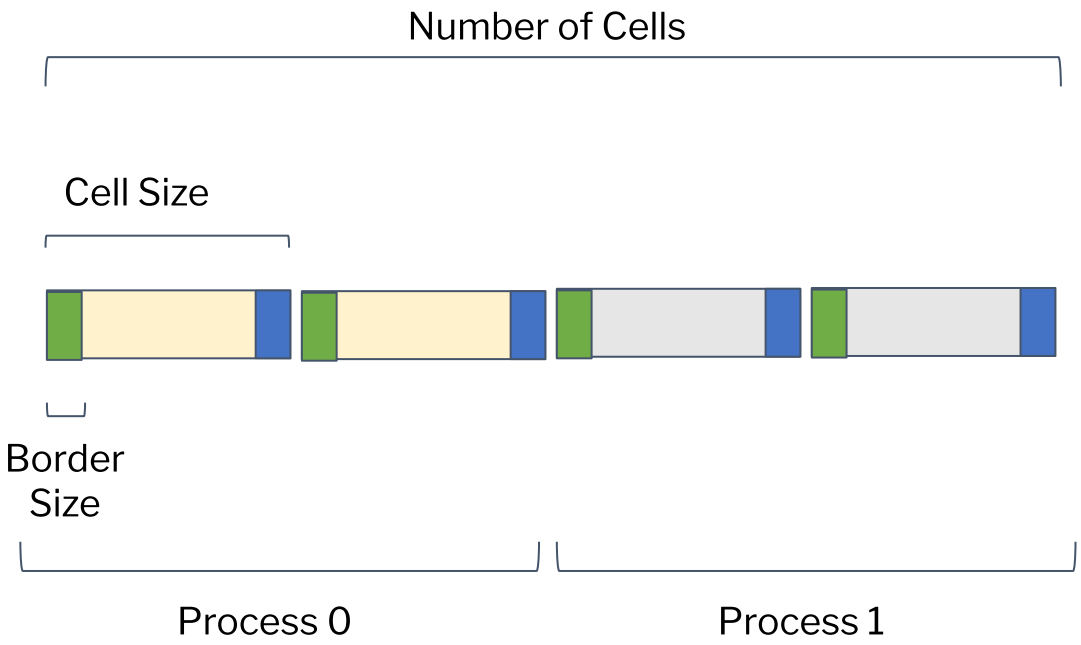
```
struct Key {
  int x;
  int y;
};

auto wa = ttg::make_tt(
    [](const Key& key){
        ...
    }, ...);

// return the process that own x
wa->set_keymap([=](const Key& key){
    return process_of(key.x);
});

// y determines the priority of the task
wa->set_priomap([](const Key2& key){ return key.y; });
```

Tutorial Material:
https://bit.ly/TTG-tutorial

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Stencil Cells



Hands On: Process Placement

Second Task:
Add process placement control for the Init, Update, and PrintResult tasks to minimize communication.

```
auto init = ttg::make_tt(
    [](const int& key){
        ...
    }, ...);

// key determines the process to execute on
init->set_keymap([=](const int& key){
    // each process receives one chunk of cells
    // to reduce communication
    return key / (numCells / numProcs);
});


auto update = ttg::make_tt(
    [](const Key2& key){
        ...
    }, ...);

// x determines the process to execute on
update->set_keymap([=](const Key2& key){
    // each process receives one chunk of cells
    // to reduce communication
    return key.x / (numCells / numProcs);
});
```

Tutorial Material:
https://bit.ly/TTG-tutorial

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Copy Handling, constness, and moving of data

Each object moving through the DAG is tracked by TTG

We rely on C++ constness and move semantics to facilitate copy handling

This helps us reduce copies of large objects

Local data leads to a new copy

```
// 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
}
```

Tutorial Material:
https://bit.ly/TTG-tutorial

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Hands On: Reusing Data Copies

Observation:

Only one task consumes the left and right boundary so we can reuse the boundary buffers, instead of allocating a new one. This saves 2 allocations for each task.

Third Task:
Reuse the boundary objects and move them back into `ttg::send`.

Reminder:

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

Question:
What happens if there are multiple tasks taking L and R buffers as inputs?

Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Serializing Custom Types

TTG uses memcpy for trivially copyable types

Custom types are serialized using serialization APIs from MADNESS or Boost

```
namespace madness {
  namespace archive {
    template <class Archive, typename T>
    struct ArchiveStoreImpl<Archive, MatrixTile<T>> {
      static inline void store(const Archive& ar,
                               const MatrixTile<T>& t)
{
        ar << tile.rows() << t.cols() << t.lda();
        ar << wrap(t.data(), t.rows() * t.cols());
      }
    };
```

Store metadata and payload in the archive

```
    template <class Archive, typename T>
    struct ArchiveLoadImpl<Archive, MatrixTile<T>> {
      static inline void load(const Archive& ar,
                              MatrixTile<T>& t) {
        int rows, cols, lda;
        ar >> rows >> cols >> lda;
        tile = MatrixTile<T>(rows, cols, lda);
        ar >> wrap(t.data(), t.rows() * t.cols());
      }
    };
  }  // namespace archive
}  // namespace madness
```

Read metadata and payload from the archive

Tutorial Material:
https://bit.ly/TTG-tutorial

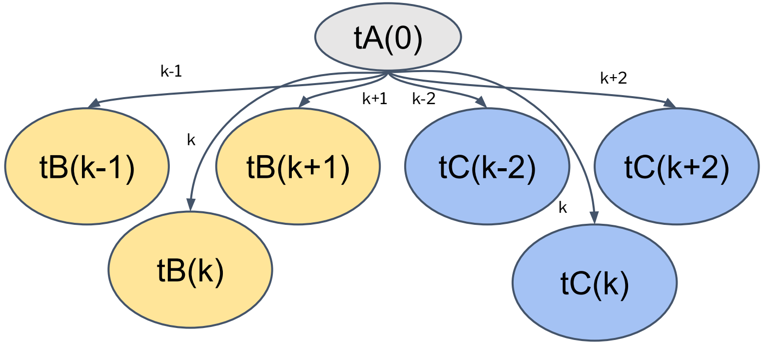
Broadcasting Data

Sending the same data to multiple successor **at once** is more efficient if done using ttg::broadcast

One set of keys for each output terminal

Avoids additional data copies

```
[](const int &k,
   T&& a) {
    mutate(a); // updates a
    std::array<int, 3> successorsA = {k-1, k, k+1};
    std::array<int, 3> successorsB = {k-2, k, k+2};
    ttg::broadcast<0, 1>(succesorsA,  // out terminal 0
                         successorsB, // out terminal 1
                         std::move(a));
}
```



Efficient Communication: split-metadata interface

Zero-copy data transfers are available through a type trait:
`ttg::SplitMetadataDescriptor<T>`

Overloads provide access to metadata, access to data, and a way to construct an empty object from metadata.

TTG will copy directly between objects.

```
struct MatrixTile {
  using metadata_t = typename std::tuple<int, int,
int>;
  ...
}

namespace ttg {

  template <typename T>
  struct SplitMetadataDescriptor<MatrixTile<T>> {
    auto get_metadata(const MatrixTile<T>& t) {
      return t.get_metadata();
    }

    auto get_data(MatrixTile<T>& t) {
      std::array<ttg::iovec, 1> v =
                    {t.size() * sizeof(T),
                     t.data()});
      return v;
    }

    auto create_from_metadata(
      const typename MatrixTile<T>::metadata_t& meta) {
        return MatrixTile<T>(meta);
    }
  };

}  // namespace ttg
```

iovecs of payload ranges

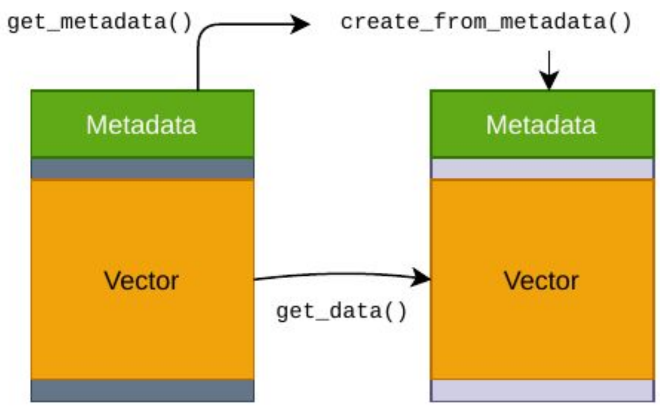
Tutorial Material:
https://bit.ly/TTG-tutorial

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Efficient Communication: split-metadata interface

Zero-copy data transfers are available through a type trait:
`ttg::SplitMetadataDescriptor<T>`




```
struct MatrixTile {
  using metadata_t = typename std::tuple<int, int,
int>;
  ...
}

namespace ttg {

  template <typename T>
  struct SplitMetadataDescriptor<MatrixTile<T>> {
    auto get_metadata(const MatrixTile<T>& t) {
      return t.get_metadata();
    }

    auto get_data(MatrixTile<T>& t) {
      std::array<ttg::iovec, 1> v =
                    {t.size() * sizeof(T),
                     t.data()});
      return v;
    }

    auto create_from_metadata(
      const typename MatrixTile<T>::metadata_t& meta) {
        return MatrixTile<T>(meta);
    }
  };

}  // namespace ttg
```

iovecs of payload ranges

Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Generic Task Arguments

Since C++14, Lambdas can have generic arguments using `auto`

TTG supports generic arguments, using `auto&&` for mutable arguments and `auto&` for const arguments. Types are inferred from the Edges.

Generic and concrete argument types should not be mixed.

```
auto tt = ttg::make_tt(
            [](auto& k,   /* the key */
               auto&& a, /* mutable */
               auto& b   /* immutable/const */ ) {
                 ...
            }, ...);
```

Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Making a TTG

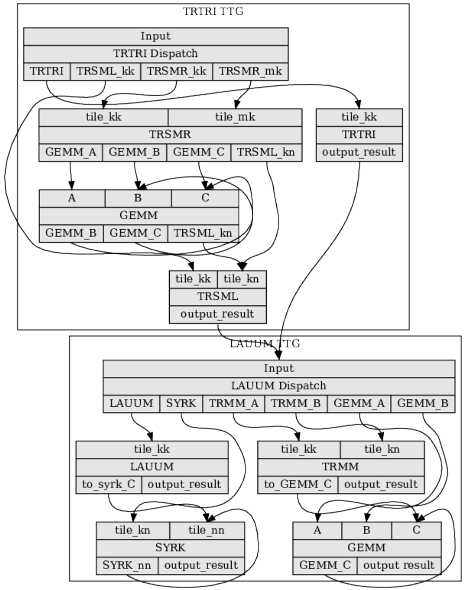
- Template Task Graph = collection of template tasks + collection of (exposed) input terminals + collection of (exposed) output terminals + name
- Until now: implicit TTG
    - empty input terminals, empty output terminals, implicit name
    - All TTs declared since last fence belong to the implicit TTG
- Use of explicit TTG: library-like fine-grain composition of operations

```
auto make_diamond_ttg(ttg::Edge<void, double> &input,
  ttg::Edge<void, double> &output, double threshold) {
  ttg::Edge<int, double> dispatch_A("dispatch->A");
  ttg::Edge<Key2, double> A_B("A->B");
  ttg::Edge<int, double> B_C0("B->C0");
  ttg::Edge<int, double> B_C1("B->C1");
  ttg::Edge<int, double> C_A("C->A");

  auto dispatch = ttg::make_tt([](const double &i) {
    ttg::send<0>(0, std::move(i));
  }, ttg::edges(input), ttg::edges(dispatch_A), <...>);
  auto wa = ttg::make_tt(<...>,
        ttg::edges( ttg::fuse(dispatch_A, C_A)),
        ttg::edges(A_B), <...>);
  auto wb = ttg::make_tt(<...>,
        ttg::edges(A_B), ttg::edges(B_C0, B_C1),
<...>);
  auto wc = ttg::make_tt([=](const int &it,
                           const double &a,
                           const double &b) {
    if(a+b < threshold) ttg::send<0>(it+1, a+b);
    else ttg::sendv<1>(a + b);
  }, ttg::edges(B_C0, B_C1),
     ttg::edges(A_C,  output), <...>);

  auto ins = std::make_tuple(
                       dispatch->template in<0>());
  auto outs = std::make_tuple(wc->template out<1>());
  std::vector<std::unique_ptr<ttg::TTBase>> ops(4);
  ops[0] = std::move(dispatch);
  ops[1] = std::move(wa);
  ops[2] = std::move(wb);
  ops[3] = std::move(wc);

  return make_ttg(std::move(ops), ins, outs, "Diamond");
}
```

Tutorial Material:
https://bit.ly/TTG-tutorial

Composing TTGs

- Goal:
  - Fine grain composition of DAGs of tasks
    - Between TTGs
    - Between Task Systems
  - Re-enable Library-like software composition
  - But maintain fine-grain asynchronicity / efficiency
- Approach:
  - Expose inputs and outputs as single input and output terminals
  - Define dispatcher tasks that distribute the input data to the appropriate tasks
  - Output final updates as they are generated
  - Use fused edges to extract intermediate results when necessary




```
namespace potri {
  auto ttg(MatrixT<double> &A,
           ttg::Edge<Key2, Tile<double>>&input,
           ttg::Edge<Key2, Tile<double>>&output) {
    ttg::Edge<Key2, Tile<double>> trtri_lauum("t2l");
    auto ttg_trtri = trtri::ttg(A, input, trtri_lauum);
    auto ttg_lauum = lauum::ttg(A, trtri_lauum, output);
    <... Define ops, ins, outs ...>
    return make_ttg(std::move(ops), ins, outs, "POTRI TTG");
  }
}
```

Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Pull and Streaming Terminals

Pull Terminals

- Edges act as bi-directional links between tasks.
- Logically preceding task is instantiated to send the data.
  - Eg. reading from memory, generating data on the fly, recursive construction.
- Greedy (at task creation) or lazy (when all other inputs are available) pulling of data to control resource utilization.

Streaming Terminals

- Computation on streaming data.
  - Eg. operations on trees in multidimensional numerical calculus algorithms.
- General reduction or accumulation operations.
- Size of the stream can be set statically or dynamically.

Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Worlds and ranks

Worlds are a context for tasks and represent the set of processes executing them.

Each process is assigned a rank in a world.

Eventually, TTG will provide ways to manage process sets and derive new worlds from them.

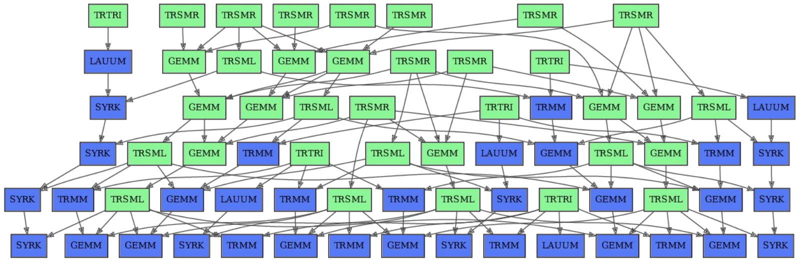
```
ttg::World world = ttg::default_execution_context();

std::cout << "Proc " <<  world.rank()
          << " of "  <<  world.size() << std::endl;


ttg::Edge<Key2, double> A_B("A->B");
ttg::Edge<int, double> C_A("C->A");

double threshold = 100.0;

auto wa = ttg::make_tt([](const int it,
                          const double &a) {
    ttg::send<0>(Key2{it, 0}, a);
    ttg::send<0>(Key2{it, 1}, a+1.0);
  }, ttg::edges(C_A), ttg::edges(A_B), "A",
  {"from C"}, {"to B"});

if (world.rank() == 0) wa->invoke(0, 0);
```

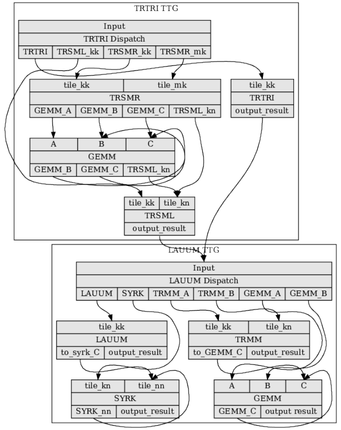
Tutorial Material:
https://bit.ly/TTG-tutorial

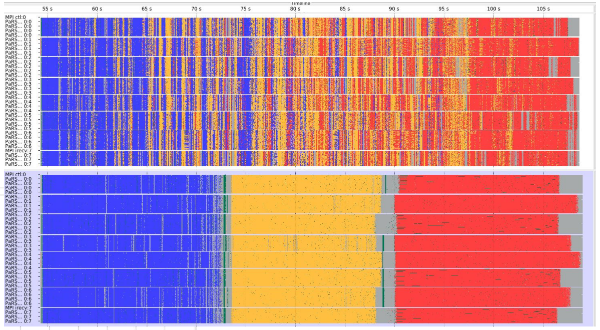
ICL INNOVATIVE COMPUTING LABORATORY

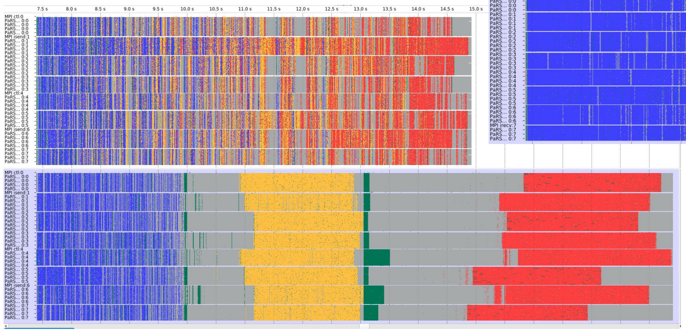
THE UNIVERSITY OF TENNESSEE KNOXVILLE

Performance Analysis: Profiling tools









Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Conclusions & Future Work

TTG provides scalable task graph discovery

Compact representation of abstract task graphs removes complexity at execution time

Task graph composition enables synchronization-free execution

Finalize support for GPU device offloading

Finalize pull terminal functionality

First official release

Tutorial Material:
https://bit.ly/TTG-tutorial

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Offloading Support Preview

Goals in TTG:

- Transparent device memory management (incl. oversubscription)
- Transparent data movement to/from device
- Avoid continuation-passing style programming
- Retain copy-tracking ability

```
auto dev_tt = ttg::make_tt<int>(
  [](const int &key,
     double    &value) -> ttg::device_task {
    auto view = ttg::make_view(value);
    // wait for the transfer to complete
    co_await val_view;
    double *dev_val = view.get_device_ptr<0>();
    <launch kernel>
    // wait for the kernel to complete
    co_await ttg::wait_kernel{};
    // send to successor
    ttg::send<0>(key, value);
  },
  ttg::edges(in_edge),
  ttg::edges(out_edge));
```

Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Tips for debugging

- `ttg::make_graph_executable(wa->get())` returns a boolean. That boolean is `true` iff the graph is connected (it should be in all cases studied here)
- You can also call `ttg::verify()(wa->get())` to verify that the graph is connected and issue additional warnings on elements that can be statically checked.
- ttg::Dot()(wa->get()) returns a String that is a GraphViz DOT representation of the TTG. Print it and use the dot tool to visualize it
- `ttg::trace_on();` will enable detailed tracing of the execution

- In the code, you can do:

```
using ttg::Debugger;
auto dbg = std::make_shared<Debugger>();
Debugger::set_default_debugger(dbg);
dbg->set_exec(argv[0]);
dbg->set_prefix(wa->get_context().rank());
// dbg->set_cmd("lldb_xterm");
dbg->set_cmd("gdb_xterm");
```

- At runtime, with PaRSEC, you can generate a DOT file per rank by setting the MCA parameter `parsec_dot` to a base filename. This will generate at runtime the DAG of tasks. Use command `parsec-dotmerger` to merge multiple DOT files into one if you do a distributed run.
  - DOT files might need some hand-fixing if you interrupt the execution because of a deadlock
- At runtime, you can also generate a profile file and analyze it (see end of talk)

Tutorial Material:
https://bit.ly/TTG-tutorial

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE