PaRSEC, SLATE integration
Bridging PTG and DTD interfaces

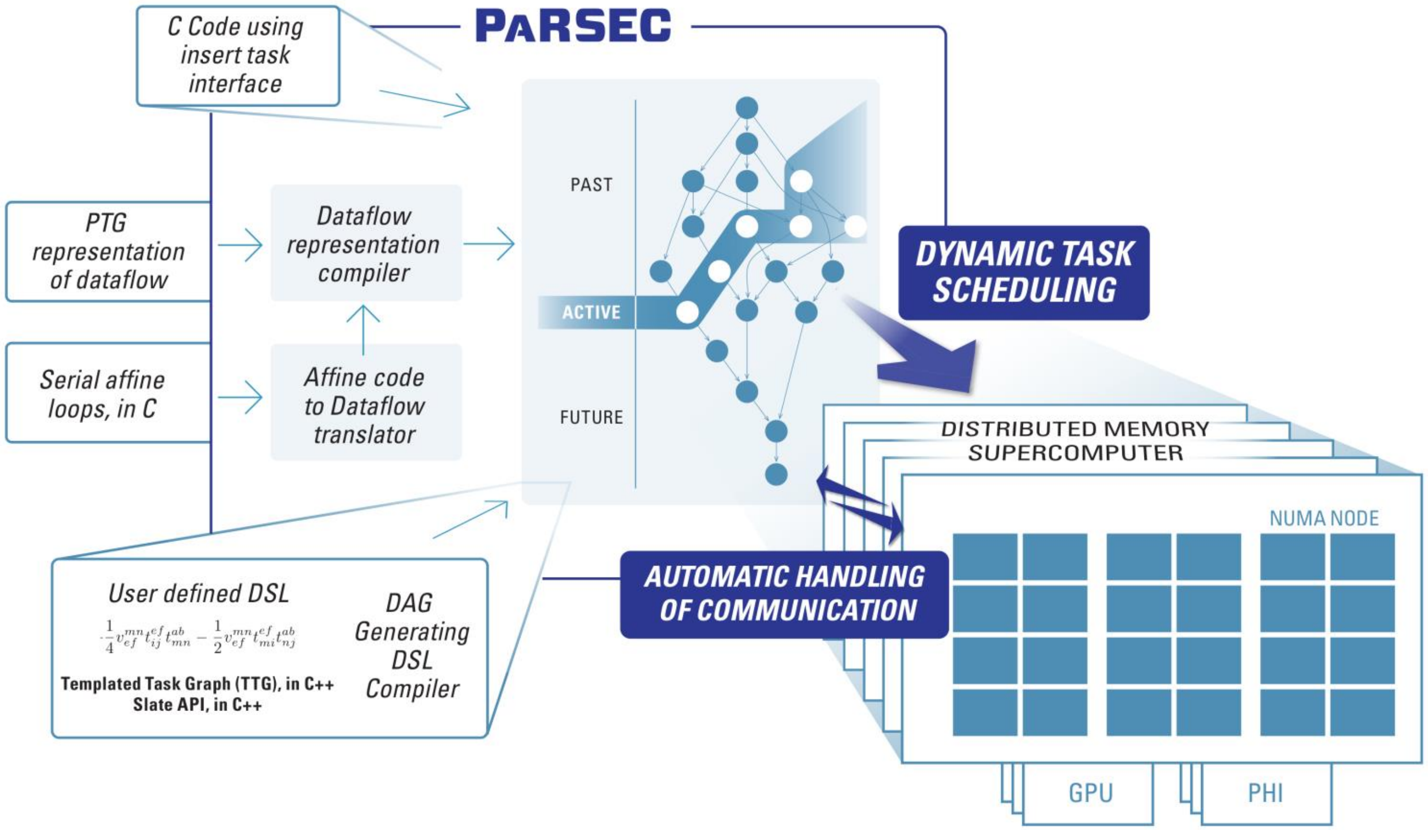
Damien Genet, Innovative Computing Laboratory,
University of Tennessee, Knoxville

UTK Conference Center, Knoxville, April 17th, 2019

PaRSEC

- Dataflow task-based runtime system
- Separation of concerns to ease the developer's life
- Multiple APIs built to interact with the runtime
- Runtime hides all data movement between memories
- Abstraction of hardware specifications through hardware model
- Internal representation of algorithms as **D**irected **A**cyclic **G**raphs of tasks connected by data dependencies

- Following slides use the Cholesky factorization (POTRF) as illustration



PARSEC

C Code using insert task interface

PTG representation of dataflow

Serial affine loops, in C

Dataflow representation compiler

Affine code to Dataflow translator

User defined DSL

$-\frac{1}{4}v_{ef}^{mn}t_{ij}^{ef}t_{mn}^{ab} - \frac{1}{2}v_{ef}^{mn}t_{mi}^{ef}t_{nj}^{ab}$

Templated Task Graph (TTG), in C++
Slate API, in C++

DAG Generating DSL Compiler

DYNAMIC TASK SCHEDULING

AUTOMATIC HANDLING OF COMMUNICATION

DISTRIBUTED MEMORY SUPERCOMPUTER

PTG language

Algorithms described as a set of operations (tasks). Dependencies between tasks are either data flowing or controls.

:descA (k,k) specifies that the owner of this data will execute the task.

POTRF is a function class, POTRF(k) is an instance of that class. k's space definition includes natural integers from 0 to descA->mt-1.

Each operation describes its input and output flows with access mode, source tasks, targets tasks and memory location.

User implementation of function is provided in the BODY sections, specialized by hardware.

This language is processed by a source-to-source compiler into C code.

```
POTRF (k)

k = 0 .. MT-1

: descA (k, k)

RW A <- (k==0) ? descA (k, k) : A SYRK (k-1, k)
        -> A TRSM (k+1 .. MT-1, k)
        -> descA (k, k)

BODY[type=CPU]
{
  // device specific code
}
BODY [type=CUDA]
{
   ...
}
BODY [type=RECURSIVE]
{
   ...
}
```

DTD interface, insert task

- Starting from a sequential algorithm of the POTRF factorization,

```
for( k = 0; k < MT; k++ ) {
   POTRF(
      A (k, k)) /*INOUT*/

   for( m = k+1; m < MT; m++ )
      TRSM(
         A (k, k), /*IN*/
         A (m, k)) /*INOUT*/

   for( n = k+1; n < NT; n++ ) {
      SYRK(
         A (n, k), /*IN*/
         A (n, n), /*INOUT*/

      for( m = k+1; m < SIZE; m++ ) {
         GEMM(
            A (m, k), /*IN*/
            A (n, k), /*INOUT*/
            A (m, n)) /*INOUT*/
      }
   }
}
```

DTD interface, insert task

- Starting from a sequential algorithm of the POTRF factorization,
- A sequential loop unrolls all the operations and insert them into the runtime.
- The sequential exploration of the algorithm ensures that data is accessed in the right order. The runtime can infer data communication.

Pros and Cons:

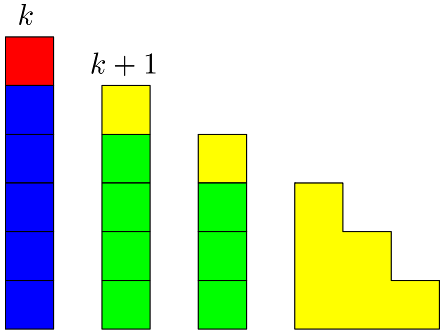
- It's easy to interact with the runtime this way.
- The naïve approach does not scale, only ntasks/nprocs concern each rank, but they "need" to see all tasks to infer communication.

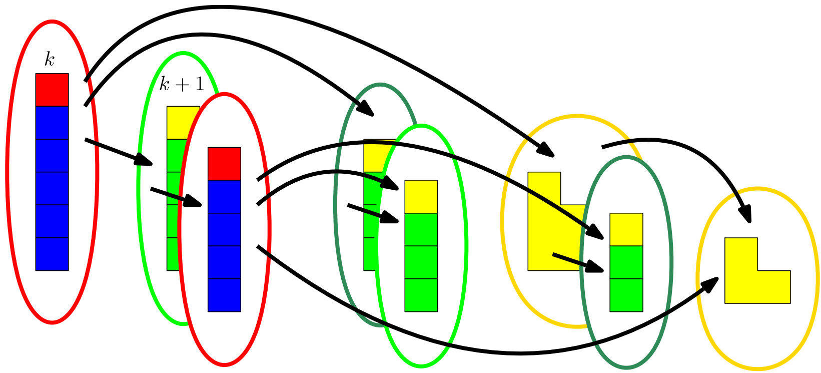
```
for( k = 0; k < MT; k++ ) {
  insert_task(potrf, "POTRF",
    TILE_OF(A, k, k), INOUT)

  for( m = k+1; m < MT; m++ )
     insert_task(trsm, "TRSM",
       TILE_OF(A, k, k), INPUT,
       TILE_OF(A, m, k), INOUT)

  for( n = k+1; n < NT; n++ ) {
    insert_task(syrk, "SYRK",
       TILE_OF(A, n, k), INPUT,
       TILE_OF(A, n, n), INOUT)

    for( m = k+1; m < SIZE; m++ ) {
       insert_task(gemm, "GEMM",
         TILE_OF(A, m, k), INPUT,
         TILE_OF(A, n, k), INOUT,
         TILE_OF(A, m, n), INOUT)
    }
  }
}
```

SLATE interface

We step back from the fully tiled algorithm and consider operations on panels.

- By expressing dependencies between columns, we reduce the number of dependencies compared to the fully-tiled formulation. (Was it an issue?)
- By using a lookahead, we let iterations overlap, and thus parallelism increases. (Is it always a good idea?)
- By seeing the update a single operation, we can anticipate larger, optimized, operations (larger SYRK on GPU, batching of tiled operations) (That would be a brilliant idea, if it was really a thing)

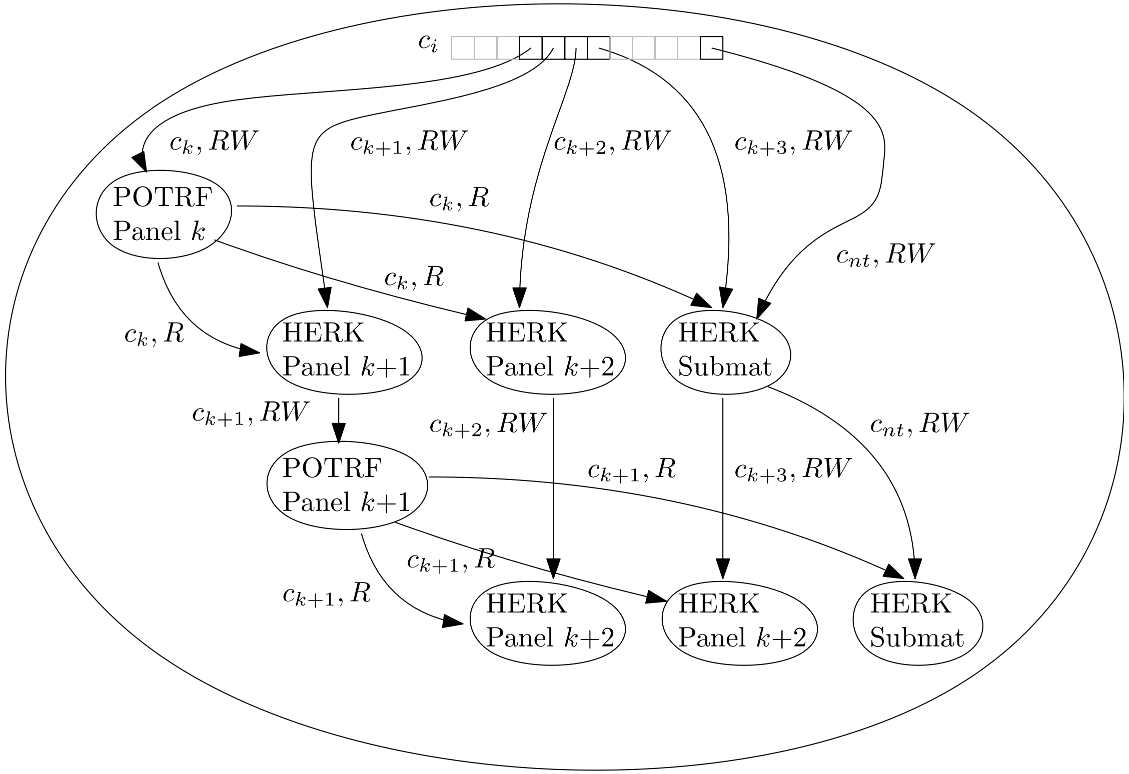




Multi-level implementation

Going into details, at the high-level:

- The “main thread” of each process inserts high-level "panel"-tasks in the high-level taskpool.
- Dependencies between high-level tasks are expressed using a 1-dimensional array.
- These "panel"-tasks use dedicated taskpools to submit work to the runtime (1 potrf taskpool, 1 taskpool per lookahead, 1 taskpool for the syrk update)
- Communication are inserted into dedicated taskpools. It is possible to synchronize with these communication taskpools.



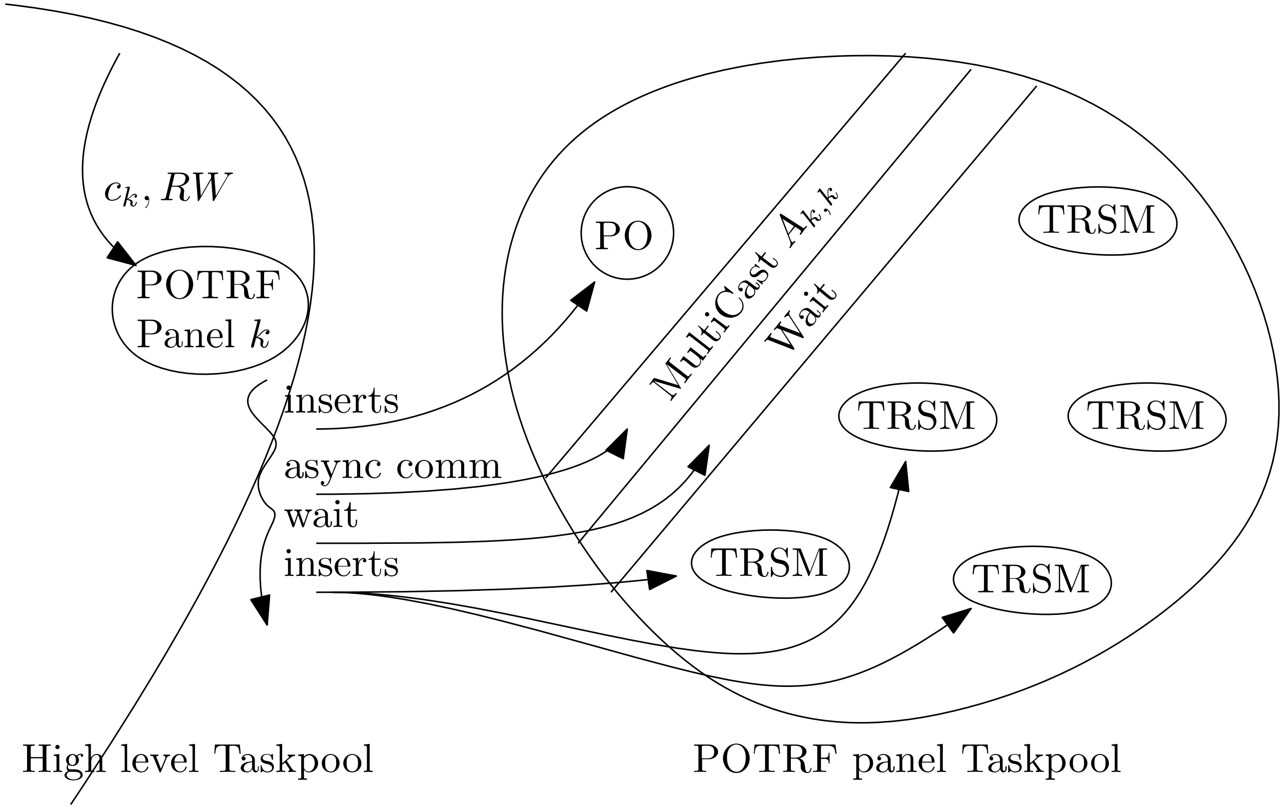
All these tasks exists on each MPI rank.

There is no data dependency. This is not dataflow anymore.

Multi-level implementation

Follow-up on the lower-level, the high-level task for the POTRF panel will:

- insert the POTRF task on the diagonal block if it owns the block;
- insert a synchronized multicast operation; Akk is sent to all the owners of Amk tiles. The non-participants will acknowledge this communication;
- wait on the completion of the communication;
- insert all the TRSM for the local blocks; TRSM are embarrassingly parallel, data is ready (Amk is local, Akk has been sent before);



Runtime improvements

"Seeing all the tasks being inserted does not scale."

- The algorithm will only insert local tasks.
- Communication will be explicit. Participating ranks will be the only ones to consider it.
- Communication are inserted as tasks into taskpool, and thus, synchronization is possible

To enable all of that, a taskpool can:

- trigger an event when all the inserted tasks (callback);
- be persistent, tasks are inserted, all the tasks are inserted, new tasks arrive, etc…

A task can:

- create taskpools to offload work;
- insert tasks in a taskpool;
- wait on a taskpool for completion of all the submitted tasks;
- decide that insertion is done, and that the taskpool is not the task responsibility anymore; its completion will be done asynchronously later by registering a callback;

Overall, we are as far away from a dataflow model as we can be.

Perspectives & Collaboration Opportunities

DTD, insert_task interface provides:

- Ease of programming, and a nice first step into the task-based runtime ecosystem;
- At the cost of a poor scalability of the naïve approach.

PTG, the parametric task graph language,

- Reaches the peak performance by providing just enough information to the runtime.
- But is requires to rethink algorithms as dataflows.

The SLATE interface tries to build on the insert_task (DTD) interface,

- Keeping the ease of programming while,
- Mitigating the scaling issues.
- Performance are not (yet) on the PTG level.

The questions are:

- Should we keep the dataflow simple and pure? Just describe an algorithm, and let the runtime optimize things?
- Or is it okay to introduce control flows, give hints to the runtime to focus resources along the "critical path"?