Sequential Task Flow Runtime Model Improvements and Limitations

Yu Pei*, George Bosilca, Jack Dongarra

University of Tennessee, Innovative Computing Laboratory, USA

November 13th, 2022

12th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)

*now working at Microsoft

Outline

Introduction

- High Performance Computing officially in the Exascale era
- Recent hardware trend indicates that many cores supported by multiple memory hierarchies and accelerators will continue to dominate the top systems
  - It is a challenge at the software level, also an opportunity
- Task-based runtime systems as alternative programming model: performant and portable. Balance with usability
- A need to evaluate the current tasking models, possible extensions and limitations

June 2022 TOP500
No.1 machine
Frontier at ORNL with HPL score of 1.1 Exaflop/s[1]

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

1. https://www.top500.org/lists/top500/2022/06/

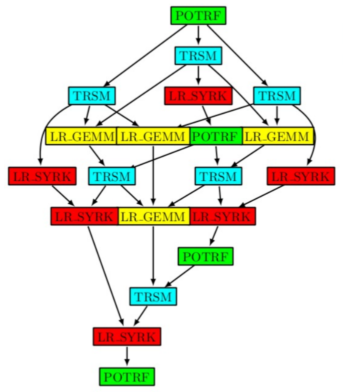
Programming Model

- Is a view of data and execution; sits between applications and architectures
- **Distributed memory models:**
  - Message Passing Interface (MPI) standard is the dominant model for inter-process communication
    - point to point, collective, one-sided operations
    - single program, multiple data (SPMD) where user manages communication explicitly
  - Partitioned Global Address Space (PGAS)
    - Chapel, UPC++, OpenSHMEM (either as programming language, or library)
- **Shared memory models:**
  - OpenMP, Kokkos, CUDA/HIP, Pthreads
- Effective interoperanility between MPI and shared memory models itself is a challenging issue

INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Task-based Runtime System



Example DAG from Cholesky factorization

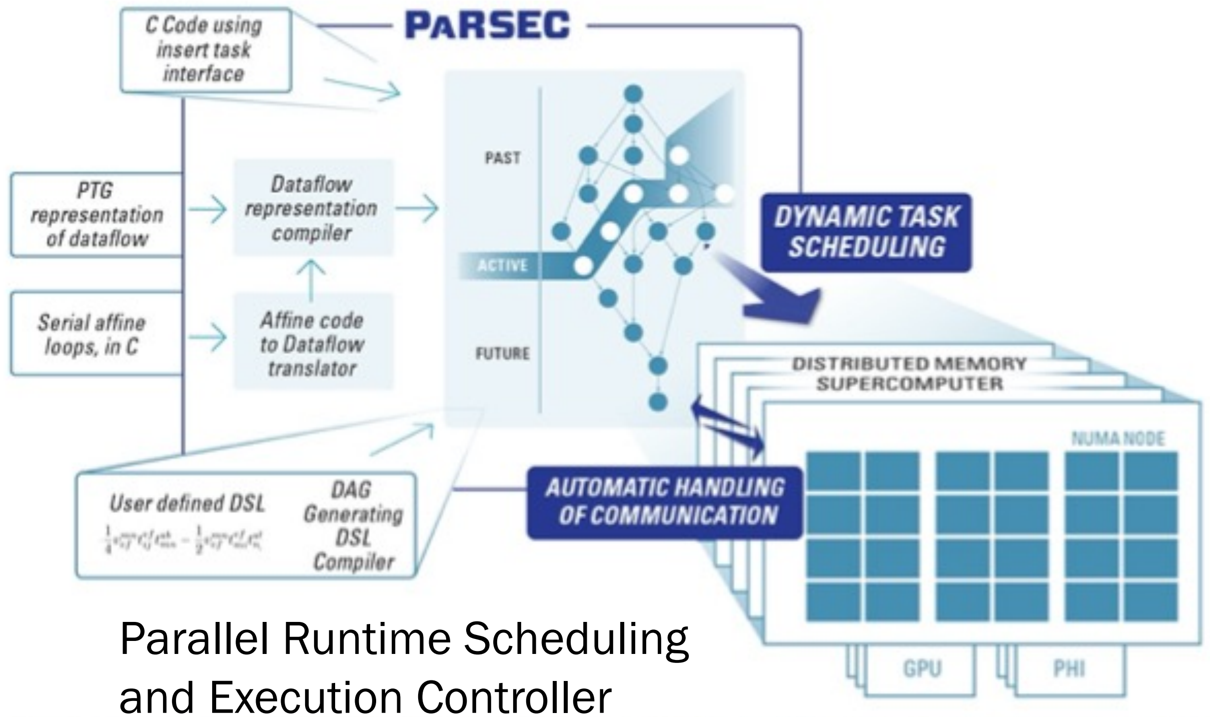
- Follows the dataflow programming model, computation on data as task, the data input/output creates the dependencies among tasks and form a directed acyclic graph (DAG)
- Finer grain synchronization, manage threading, accelerators and across networks (algorithm and performance separation of concerns)
- Runtimes with some traction in the community:
  - PaRSEC: Provides multiple interfaces to express the task graph, supports a wide variety of applications
  - Legion: Logical regions to represent partitioned data to infer task dependencies
  - StarPU: Sequential task insertion, like PaRSEC DTD, application in dense and sparse linear algebra
  - Taskflow: Motivated by computer-aided design (CAD), support complex task graph with control flow
  - OmpSs/OpenMP: directive based, support accelerators, can work in conjunction with high-level programming language
  - Many others exists highlighting a rich and active research field

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

PaRSEC Runtime System

- Three interfaces: Parameterized task graph (PTG), Dynamic task discovery (DTD) and Template task graph (TTG)
- Modular design, developers can work at application level, design interface or optimize runtime internals:
  - matrix operations; new interface or new APIs; task scheduling, communication engine, thread binding etc


C Code using insert task interface
PARSEC
PTG representation of dataflow
Dataflow representation compiler
Serial affine loops, in C
Affine code to Dataflow translator
PAST
ACTIVE
FUTURE
DYNAMIC TASK SCHEDULING
DISTRIBUTED MEMORY SUPERCOMPUTER
NUMA NODE
User defined DSL
DAG Generating DSL Compiler
AUTOMATIC HANDLING OF COMMUNICATION
GPU
PHI

Parallel Runtime Scheduling and Execution Controller

Motivations

- PaRSEC Parameterized Task Graph (PTG) Domain Specific Language (DSL) expresses the task graph with a pure dataflow programming paradigm. The approach might be challenging, with a steep learning curve but is extremely portable and effective for non dynamic and non-data dependent programming

- Sequential Task Flow (STF) model (PaRSEC lingo Dynamic Task Discovery) has several advantages:
  - dependencies don't need to be specified a-priori
  - Provides an API based interface to generate the task graph dynamically
  - User can write the algorithm as a sequential code
- Still provide the benefit of finer grain synchronization, and easier for user adoption
- However, each node needs to iterate over the entire graph for dependency analysis resulting in an analysis cost that increases with the number of nodes

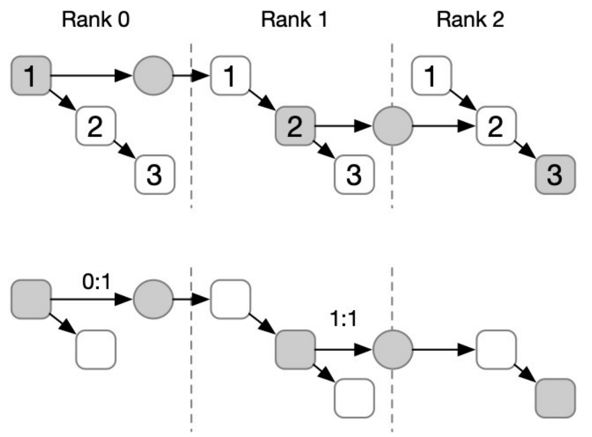
- In this research we optimize DTD by providing the ability to trim task graph to reduce overhead, and integrate collective communications in the programming paradigm
- Evaluated the changes to the programming model, limitations of the approach with Cholesky and QR factorizations

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Handling dependencies for DTD and overheads

- Current approach with analysis overhead increase as more nodes are used
  - Legion/Regent proposed DAG trace replay technique to reduce overhead[1]
  - StarPU propose a form of DAG trimming
- We propose changes to the programming model itself instead, where user can supply knowledge to trim the graph, and call explicit collective as addition to the DTD interface



Send ID: {0:0, 1:1, 2:0} {0:0, 1:0, 2:1} {0:0, 1:0, 2:0}
Recv ID: {0:0, 1:0, 2:0} {0:1, 1:0, 2:0} {0:0, 1:1, 2:0}

Current scheme and new data-level key to enable transparent user graph trimming

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

$$T_{DTD} = \boxed{\frac{N \times C_T}{P \times n}} + \underbrace{\boxed{N \times C_D + \frac{N \times C_R}{P}}}_{\text{DTD overhead}}$$

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

Model from Dynamic Task Discovery in a Data-Flow, Task-Based Runtime System, Reazul Hoque, PhD dissertation

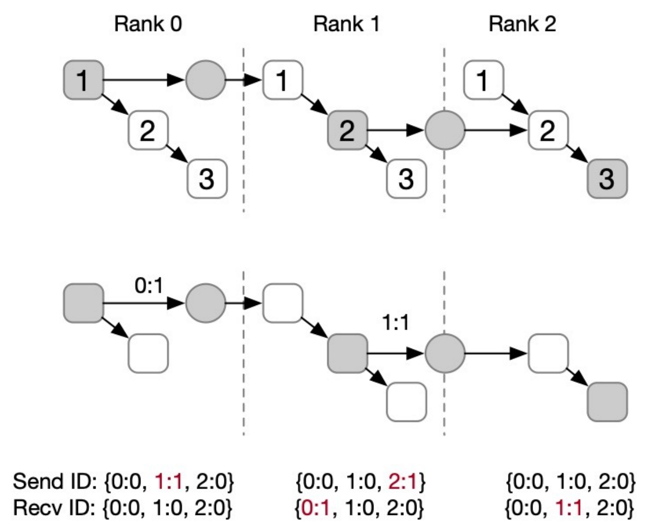
INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

1. Dynamic Tracing: Memorization of Task Graphs for Dynamic Task-Based Runtimes

Trimming to reduce overheads: label the data flow

- Instead of naming tasks uniquely, we only need to make sure the data send/recv on both side can be uniquely matched
- Transparent to the original code, since the remote tasks with no data dependency with local tasks will not require any matching
- Trimming the graph can significantly reduce the overhead of task insertion
- In the new scheme, data flow ID is a combination of sender rank and sequence number to uniquely
  label each data transfer
- Both the sender and the receiver has the dependent tasks inserted; the data ID will be assigned correctly for the two sides to match the data transferred



Grey square represents local task, white square represents remote task. Circle represents the structure passed to communication engine.

User level dynamic collective operation

- MPI collective calls have a communicator parameter, predefined with participant processes
- Contains not only broadcast, but gather, allgather, allreduce(contain operations) etc
- Our goal is to making distributed copy of a data
- StarPU implements broadcast without explicit API, assume all remote tasks will be known when data is ready to be sent (might have missing participants)[1]
- PaRSEC PTG has compact representation of all the tasks, have unique key for each task
  - With this information, broadcast via MPI P2P (Chain, Binomial tree)

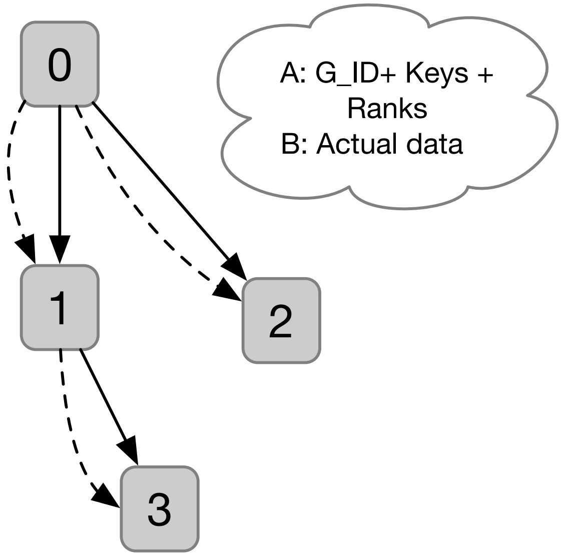
```
dgetrf(k)
k = 0 .. NT

: descA(k, k) //locality

RW A  <- (FIRST) ? descA(k,k)
      <- (!FIRST) ? C dgemm(k_prev, DIAG, DIAG)

      -> (END>=START) ? A dtrsm_l(k, START..END)
      -> (END>=START) ? A dtrsm_u(k, START..END)

RW IP <- IP ipiv_in(k)  [type = PIVOT count = NB]
      -> IP ipiv_out(k) [type = PIVOT count = NB]

/* Priority */
;1

BODY
{
  // Factorizing diagonal block (k, k)
  int mb = descA.nbi[k];
  double *dA = &(((double*)A)[1]);
  iinfo = LAPACKE_dgetrf(LAPACK_COL_MAJOR,
                         mb, mb, dA, mb, ipiv);
}
```

Example PaRSEC PTG kernel where broadcast pattern is used

User level dynamic collective operation

- Instead of implicit broadcast, we decided to provide an explicit API call
- Maintain the default P2P behavior, can selectively decide where to convert to collective operation (with different topologies)
- Algorithm writer needs to clearly articulate the grouping information, and this metadata will guide the data propagation (ability to adjust priority)
- Based on the previous feature, we generate collective operation with data level P2P keys

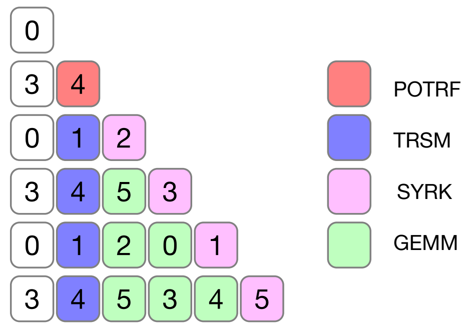
1. propagate with bcast keys and ranks for the descendants
2. Propagate actual data with G_ID, independent from step 1
3. On completion of receiving metadata via P2P local key, populate message with G_ID and metadata to match with propagation of actual data from step 2
4. Continue the steps for downstream ranks

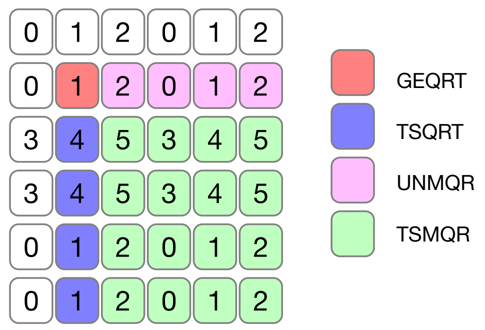


two stages approach, where we prepare: the global key, the data flow keys and participating ranks as the first message, and the actual broadcast will use the global ID to progress

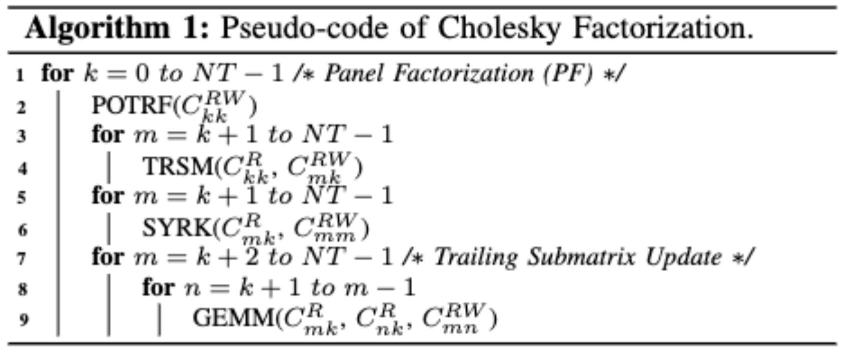
Test Problems

- Tile-based Cholesky and QR, with nested for loops and regular data dependencies between tasks
- Respectively have $\frac{1}{3}n^3$ and $\frac{4}{3}n^3$ floating point operations, and QR has a tighter dependency



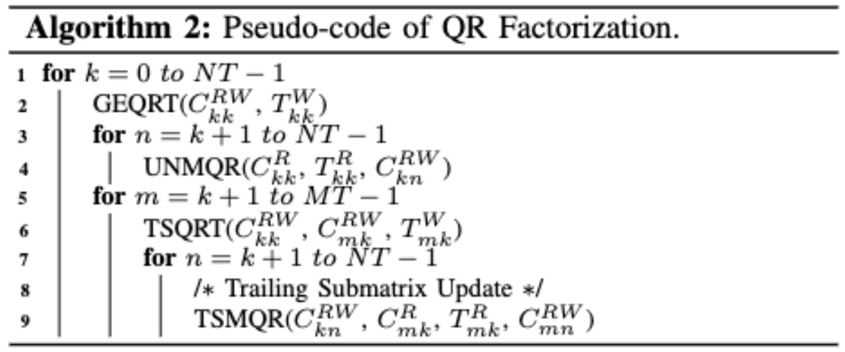


The four different kernels from Cholesky and QR respectively. Both runs on a 2X3 compute grid with 2-D block cyclic distribution. For QR, a super-tiling of 2 is used on the grid row to reduce cross node P2P communication.



Algorithm 1: Pseudo-code of Cholesky Factorization.

```
for $k = 0$ to $NT - 1$ /* Panel Factorization (PF) */
    POTRF($C_{kk}^{RW}$)
    for $m = k + 1$ to $NT - 1$
        TRSM($C_{kk}^{R}$, $C_{mk}^{RW}$)
    for $m = k + 1$ to $NT - 1$
        SYRK($C_{mk}^{R}$, $C_{mm}^{RW}$)
    for $m = k + 2$ to $NT - 1$ /* Trailing Submatrix Update */
        for $n = k + 1$ to $m - 1$
            GEMM($C_{mk}^{R}$, $C_{nk}^{R}$, $C_{mn}^{RW}$)
```

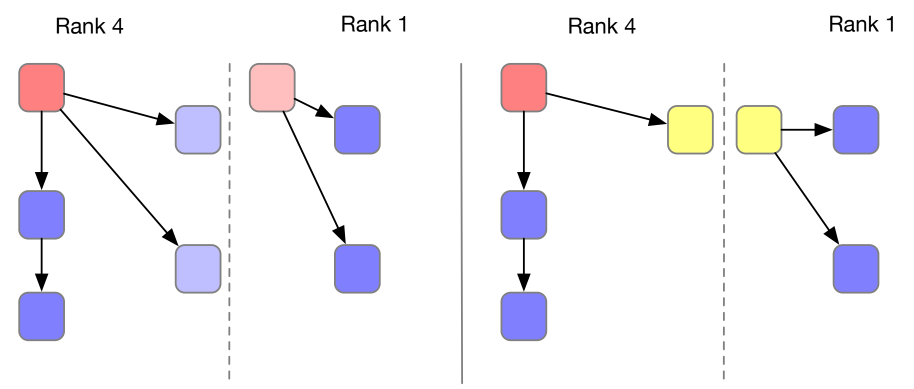


Algorithm 2: Pseudo-code of QR Factorization.

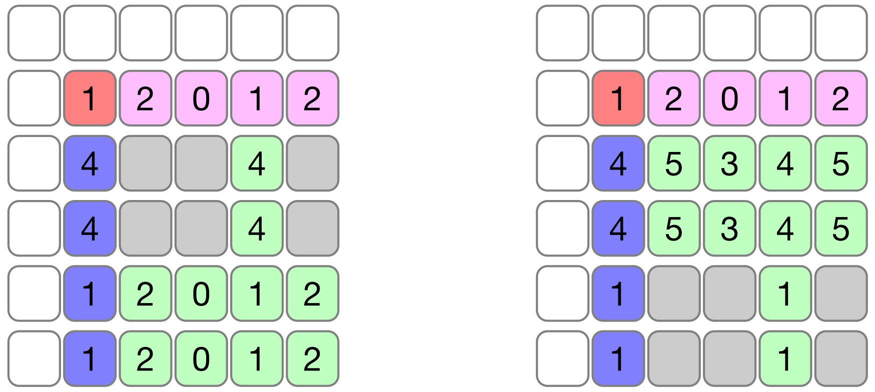
```
for $k = 0$ to $NT - 1$
    GEQRT($C_{kk}^{RW}$, $T_{kk}^{W}$)
    for $n = k + 1$ to $NT - 1$
        UNMQR($C_{kk}^{R}$, $T_{kk}^{R}$, $C_{kn}^{RW}$)
    for $m = k + 1$ to $MT - 1$
        TSQRT($C_{kk}^{RW}$, $C_{mk}^{RW}$, $T_{mk}^{W}$)
        for $n = k + 1$ to $NT - 1$
            /* Trailing Submatrix Update */
            TSMQR($C_{kn}^{RW}$, $C_{mk}^{R}$, $T_{mk}^{R}$, $C_{mn}^{RW}$)
```

Modification to User Code

- User level code will have conditionals to decide whether to insert a given task
- Broadcast call will affect whether remote tasks are inserted, or serve as central point of connection
- Writer task needs to insert remote readers, reader tasks need to insert at least the correct data writer task, simpler for nested for-loops but error prone for complicated graph (similar to SPMD)



Left, trimmed task graph without broadcast call; Right, explicit broadcast call to propagate POTRF data. Color scheme and data distribution follows that from previous page. Lighter red and purple represent remote tasks, yellow represents broadcast task. Data dependency between TRSM and GEMM omitted



Since only the TSMQR tasks are of order O(N 3), we can insert all the other tasks in all the nodes while inserting TSMQR only on ranks that are in the same row or column of the current panel tasks. Left is for tasks inserted on rank 1, while right figure is for tasks on rank 4

Modification to User Code

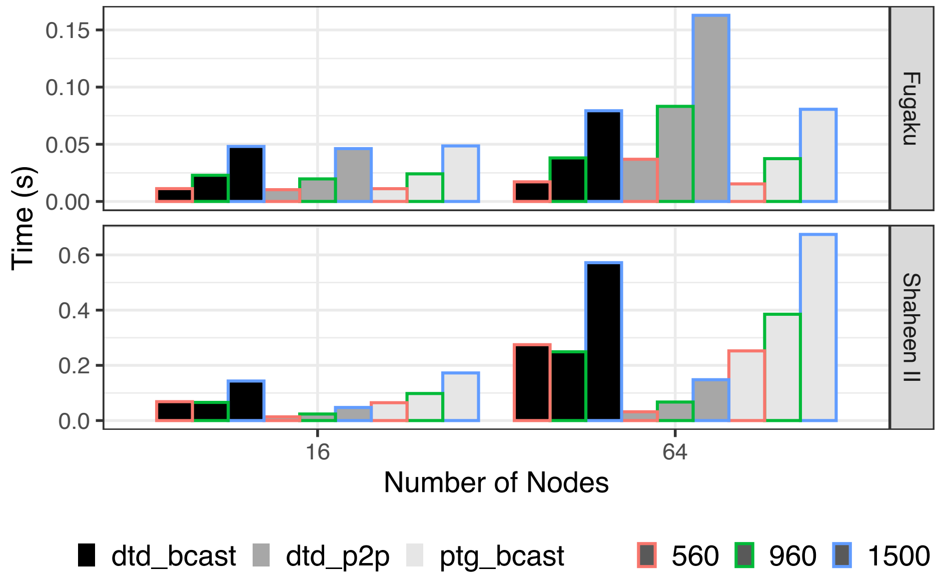
```
for(k = 0; k < NT; k++){
  // diagonal DGETRF
  insert_task(taskpool, parsec_dgetrf,
    1, "getrf",
    sizeof(int)  , &k                  ,VALUE,
    PASSED_BY_REF, TILE_OF(A, k, k) ,INOUT | AFFINITY,
    PASSED_BY_REF, TILE_OF(IP, k, 0),OUTPUT,
    PARSEC_DTD_ARG_END);
  if(k < NT-1){
    for(int i = k+1; i < NT; i++){
      insert_task(taskpool, parsec_dtrsm_l,
                  ...);
      insert_task(taskpool, parsec_dtrsm_u,
                  ...);
    }
    data_flush(dtd_tp, TILE_OF(A, k, k));
    data_flush(dtd_tp, TILE_OF(IP, k, 0));

    for(int i = k+1; i < NT; i++){
      for(int j = k+1; j < NT; j++){
        insert_task(taskpool, parsec_dgemm,
                    ...);
      }
    }
  }
}
```

DTD interface of the LU algorithm
Update data size to send in the body

```
With Trimming and Broadcast
if(rank_of_data(A, k, k) == rank) {
  insert_task( dtd_tp, parsec_core_potrf,
...);
}
// Broadcast diagonal tile to current
panel
for(int m = k+1; m < total; m++) {
  int tile_rank = rank_of_data(A, k, m);
  // collect ranks into destination rank
array
  dest_ranks[dest_rank_idx] = tile_rank;
}
if( participate_in_bcast ) {
  dtd_tile_root = PARSEC_DTD_TILE_OF(A, k,
k);
  parsec_dtd_broadcast(
    dtd_tp, root,
    dtd_tile_root, TILE_FULL,
    dest_ranks, dest_rank_count);
}
```

Experiments

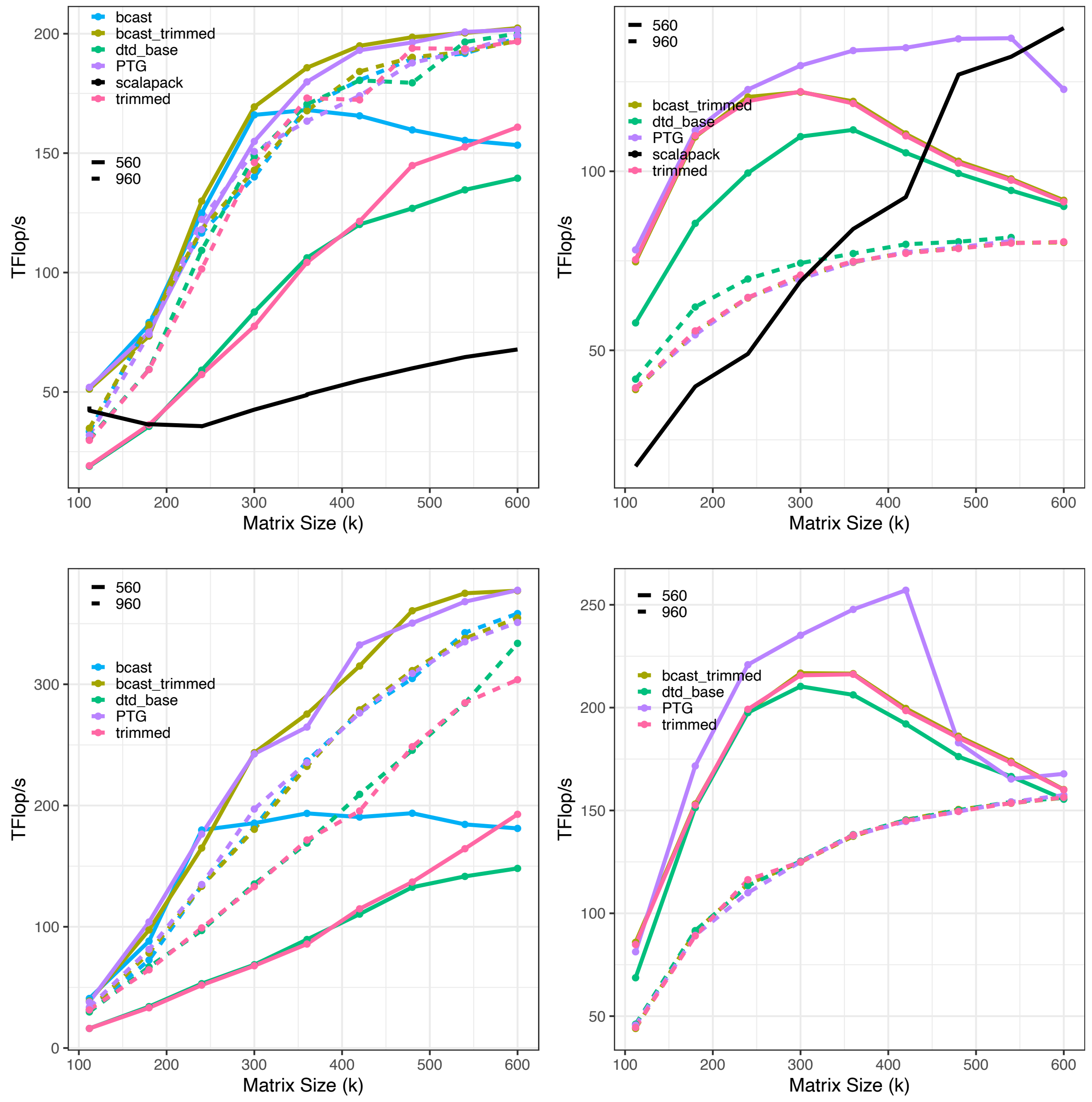
- Run on two HPC systems:
  - Shaheen II, a Cray XC40 with 6174 nodes of Intel Haswell processor with 128GB memory. Cray Aries network interconnect. Compiled with Cray MPI and Intel toolchain (MKL)
  - Fugaku, Fujitsu ARM (SVE) system with A64FX processor, 32GB memory, TofuD interconnect. Compiled with Fujitsu MPI and SSL2 library
- Benchmark the performance of one broadcast, compare with PTG broadcast, DTD p2p
- Broadcast is faster than p2p on Fugaku, but the opposite on Shaheen II. Network topology impact; situation will change in real application



Benchmark of a broadcast operation for sending a square tile of double floating points

Experiments

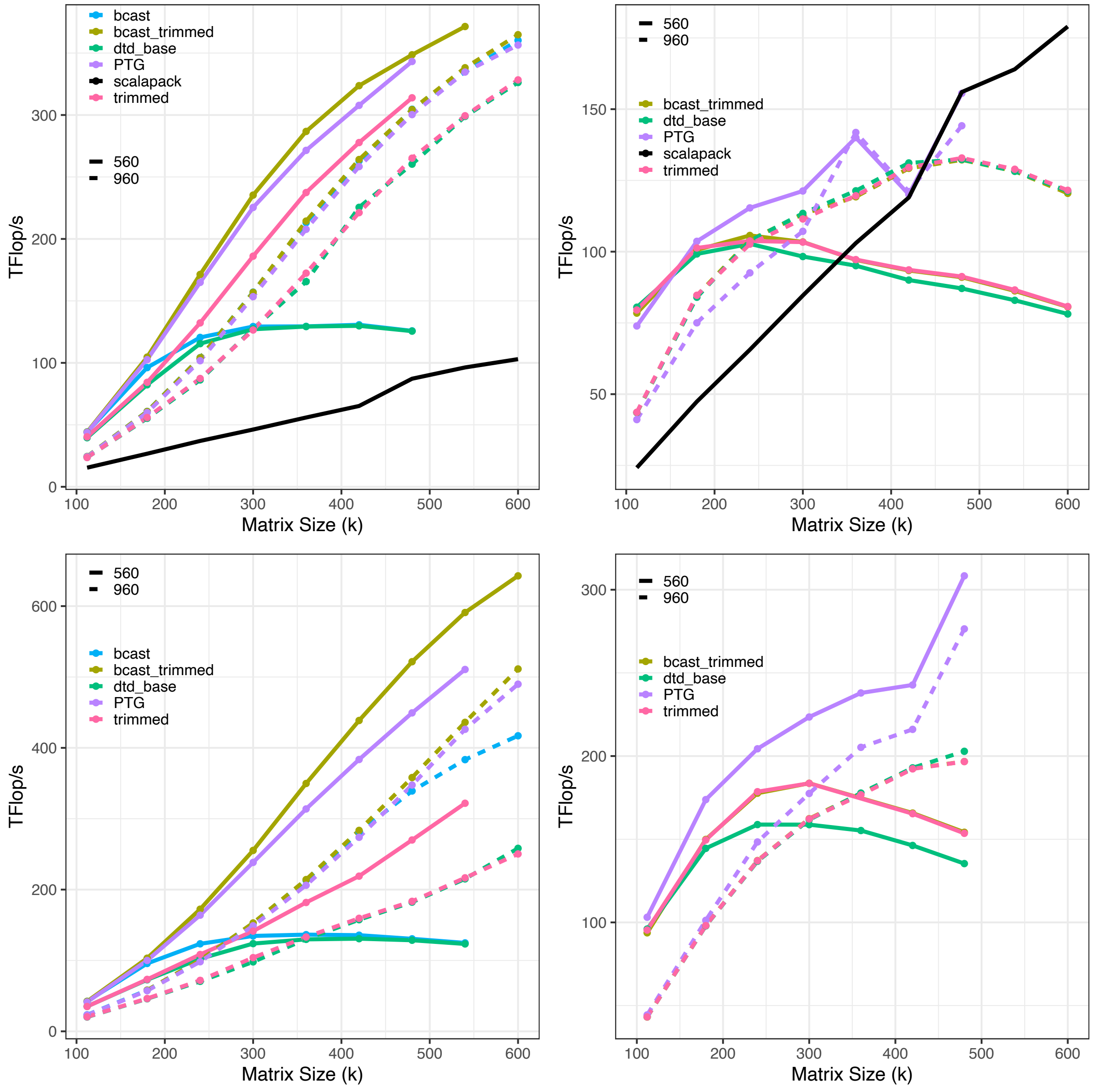
- ScaLAPACK result as reference
- Different versions with trimming, broadcast operation or both
- Base DTD and PTG version too
- Runtime implementation performs better than ScaLAPACK
- Single node DPOTRF can achieve 860 GFLOP/s
- Tile size needs to be tuned. DTD QR requires bigger tile size
- Trimming and broadcast both have benefits, but reducing number of tasks inserted is key
- Broadcast version is better despite slower than P2P on benchmark
- Slowdown in larger matrix for QR, need further investigation



Shaheen II, top 256 nodes, bottom 512 nodes
Left, Cholesky, Right, QR factorization

Experiments

- Similar trend as on Shaheen II
- DPOTRF on one node of Fugaku can achieve 1700 GFLOP/s
- Base DTD performs poorly, and broadcast alone doesn't help
- Trimming the graph, thus reducing insertion overhead provide relief
- For QR, due to the tighter dependencies, only small improvement
- QR performance with tile size 960 takes over that with size 560 sooner



Fugaku, top 256 nodes, bottom 512 nodes
Left, Cholesky, Right, QR factorization

Conclusions and Future Works

- Performance is portable across the two systems, with different FLOPs to bandwidth ratio
- Even for applications like dense factorizations, further tuning is needed
- To scale to entire systems, minimization of the runtime overhead (trimming or PTG) is a requirement
- Communication speed is critical for current systems, collectives provide a lot of benefits for runtime system

- Further investigate the collective and scaling issues
- Reduce metadata transfer, adopt different broadcast topologies

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

Questions?