TaskTorrent: a Lightweight Distributed Task-Based Runtime System in C++

Leopold Cambier[1], Yizhou Qian[1], Eric Darve[1,2]
[1]ICME, [2]ME, Stanford

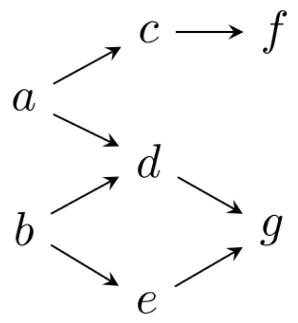
PAW-ATM 2020

Introduction

- Increasingly complex and heterogeneous computer architectures
  - Many nodes, many cores
  - A hierarchy of memory
  - Accelerators (GPUs)
- Bulk synchronous = many artificial synchronization points
- Runtime systems
  - Directed Acyclic Graph (DAG) of tasks
  - Avoid synchronization
  - Exploit all resources

```
for (auto i : local0)
  compute0(i);
if ([...])
  { MPI_Send(m, ...); }
else
  { MPI_Recv(m, ...); }
for (auto i : local1)
  compute1(i);
```

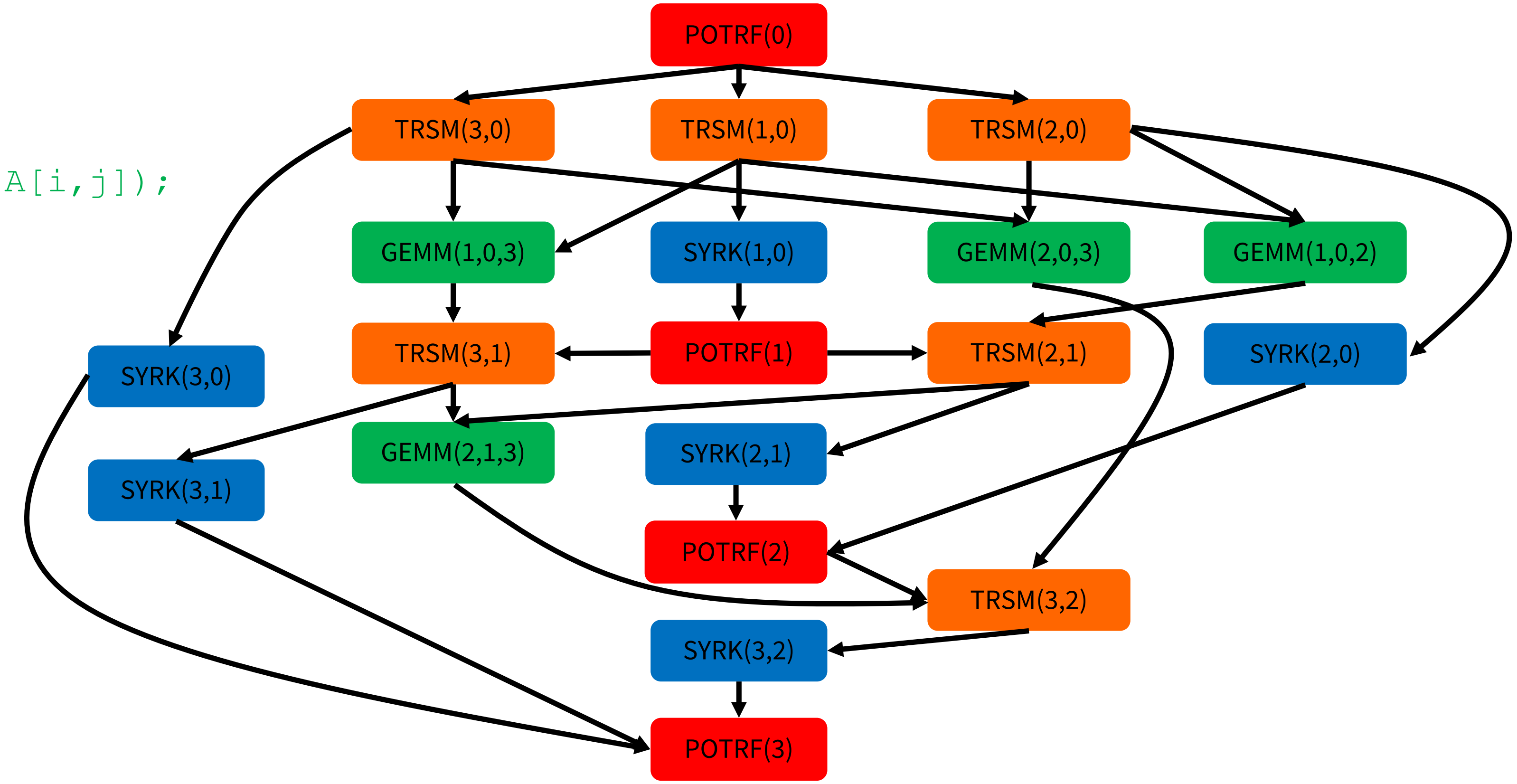
A typical MPI program



A DAG of tasks

Example: Cholesky Factorization

```
for (k=0; k<n; k++) {
    POTRF(A[k,k]);
    for(i=k+1; i<n; i++) {
        TRSM(A[k,k],A[i,k]);
    }
    for (i=k+1; i<n; i++) {
        SYRK(A[i,k], A[i,i]);
        for (j=k+1; j<i; j++) {
            GEMM(A[i,k], A[j,k], A[i,j]);
        }
    }
}
```



Goals of TaskTorrent

Main obstacle of runtimes is user adoption

Adoption requires

- Easy to learn API
    - No new language
    - API with predictable behaviors
    - Small set of primitives
- Good interaction with existing codebases
    - Plays well with MPI and message-passing codes
    - Incremental adoption
    - Standard tools (MPI and C++)
    - Any user data structures
    - Minimal overhead, no task refactoring

Runtime Systems: STF vs PTG

```
/** Define task **/
void task(...) {...}
/** Register data **/
data = [...]
/** Process DAG **/
for (k ...)
    task(data[k], data[k1], ...)
```

- Sequential Task Flow (STF) based program
- Data dependencies are inferred through data sharing rules

```
/** Task deps. expressed
 *  as functions of K
 **/
 in_deps  = (K k) {...}
 task     = (K k) {...}
 out_deps = (K k) {...}
/** Seed tasks **/
for(k in kinit)
   start(k)
```

- Parametrized Task Graph (PTG) based program
- Task dependencies are defined using functions over K.
- Computation is triggered by seeding the tasks

STF examples: StarPU and Regent

StarPU[1]

```
for (k=0; k<n; k++) {
  starpu_mpi_task_insert(potrf, RW, A[k,k]);
  for(i=k+1; i<n; i++) {
    starpu_mpi_task_insert(trsm,
                           R, A[k,k],
                           RW, A[i,k]);
  }
  for (i=k+1; i<n; i++) {
    starpu_mpi_task_insert(syrk,
                           R, A[i,k],
                           RW, A[i,i]);
    for (j=k+1; j<i; j++) {
        starpu_mpi_task_insert(gemm,
                               R, A[i,k],
                               R, A[j,k],
                               RW, A[i,j]);
    }
  }
}
```

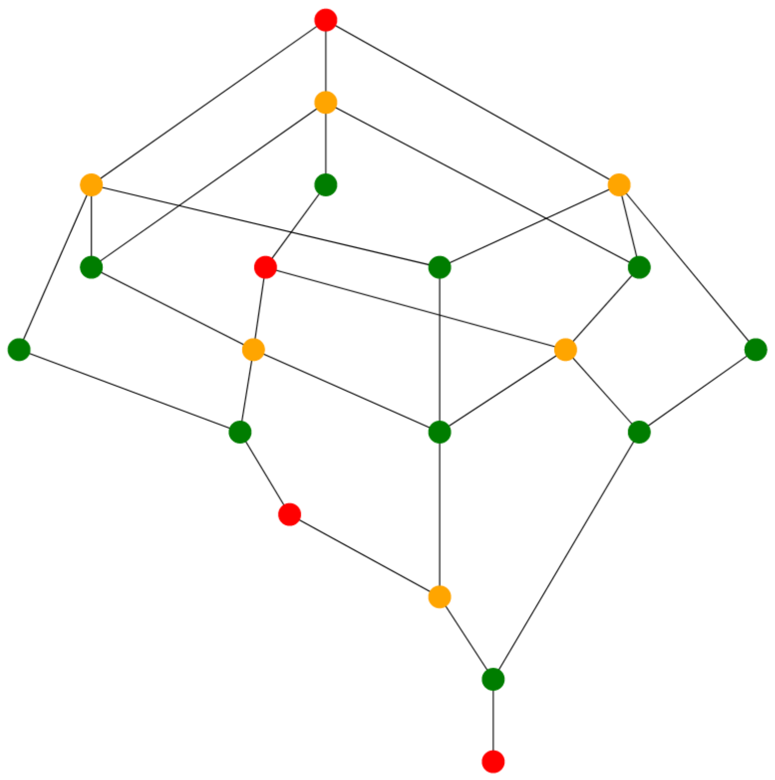
Legion/Regent[2]

```
for k = 0, n do
  dpotrf(k, n, pA[f2d{i = k, j = k}])
  for i = k+1, n do
    dtrsm(i, k, n, pA[f2d{i = i, j = k}])
  end
  for i = k+1, n do
    dsyrk(i, k, n, pA[f2d{i = i, j = i}],
                   pA[f2d{i = i, j = k}])
    for (j=k+1; j<i; j++) {
      dgemm(i, j, k, n,
            pA[f2d{i = i, j = k}]),
            pA[f2d{i = i, j = k}]),
            pA[f2d{i = i, j = k}]))
    end
  end
end
```

- Data flow and dependency are inferred implicitly
- Easy to implement and understand

[1]C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, “StarPU: a unified platform for task scheduling on heterogeneous multicore architectures,” Concurrency and Computation: Practice and Experience, vol. 23, no. 2, pp. 187–198, 2011.
[2]M. E. Bauer, “Legion: Programming distributed heterogeneous architectures with logical regions,” Ph.D. dissertation, Stanford University, 2014.

Sequential Task Flow



- In general, STF requires enumerating the entire DAG → Difficult to scale

PTG example: PaRSEC

PaRSEC[1]

```
TRSM(k, m)

// Execution space
k = 0   .. NT-1
m = k+1 .. NT-1

// Partitioning
: dataA(m, k)

// Flows & their dependencies
READ  A <- A POTRF(k) [type = LOWER]
RW    C <- (k == 0) ? dataA(m, k)
        <- (k != 0) ? C GEMM(k-1, m, k)
        -> A SYRK(k, m)
        -> A GEMM(k, m, k+1..m-1)
        -> B GEMM(k, m+1..NT-1, m)
        -> dataA(m, k)

BODY
  trsm(A, C);
END
```

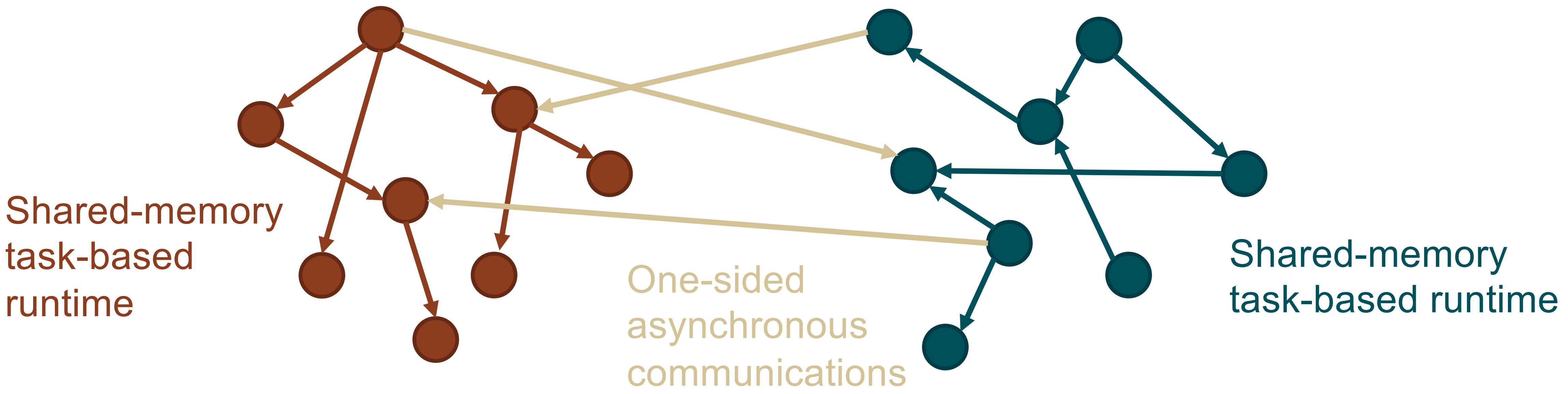
```
for (k=0; k<N; k++) {
  POTRF(RW, A[k][k]);
  for (m=k+1; m<N; m++)
    TRSM(R,  A[k][k],
         RW, A[m][k]);
  for (n=k+1; n<N; n++) {
    SYRK(R,  A[n][k],
         RW, A[n][n]);
    for (m=n+1; m<N; m++)
      GEMM(R,  A[m][k],
           R,  A[n][k],
           RW, A[m][n]);
  }
}
```

- Explicit data flow and task dependency
- Implicit control flow
- PTG language (JDF) designed for linear algebra

[1] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, “PaRSEC: Exploiting heterogeneity to enhance scalability,” Computing in Science Engineering, vol. 15, no. 6, pp. 36–45, 2013.
Code from http://icl.utk.edu/projectsfiles/parsec/pubs/parsec-tutorial-1.pdf

TaskTorrent

- A lightweight distributed task-based runtime in C++
  - Lightweight: parametrized task graph (PTG) + active messages (AMs)
  - Message passing-based: easy to interface with existing codes
  - Portable: MPI and C++ threads
  - No need to modify or wrap around user's data structures

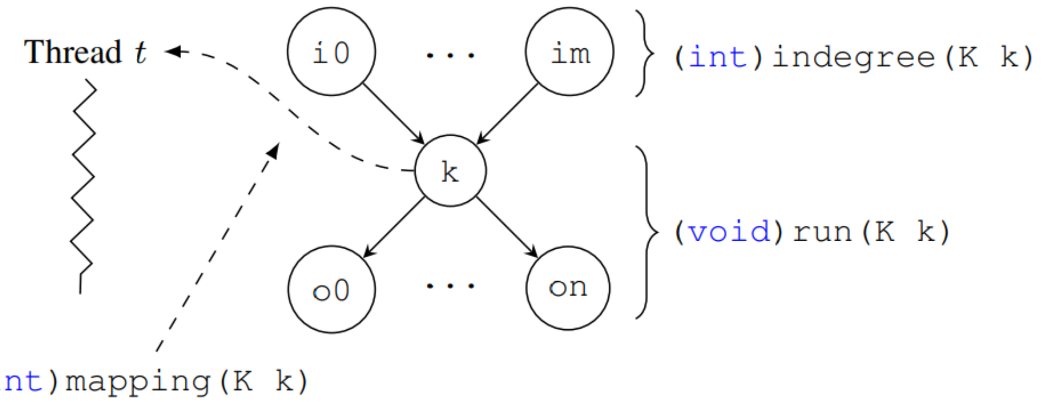


PTG in TaskTorrent

A. Tasks defined as `functions(Key k)`
   1. Mapping of task to thread
   2. Number of incoming dependencies
   3. Computational routine + fulfill dependencies

B. Communication (send data + fulfill remote tasks)
   - One-sided asynchronous & nonblocking active messages

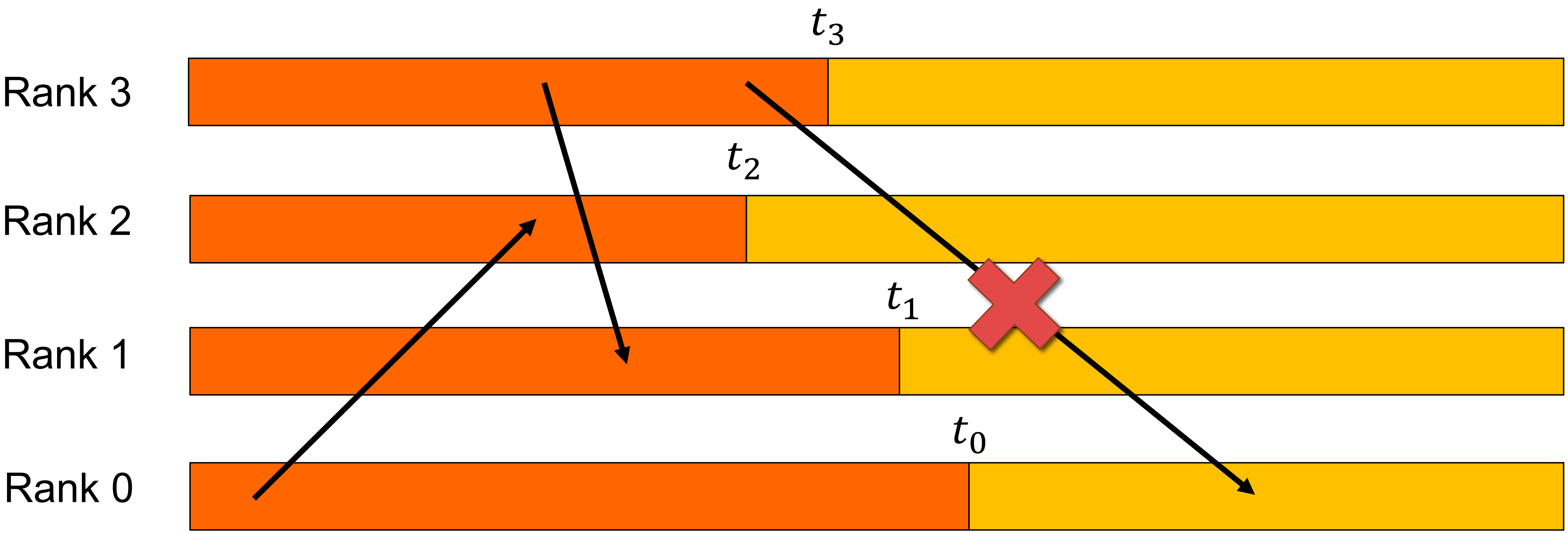
C. Seed initial tasks & join



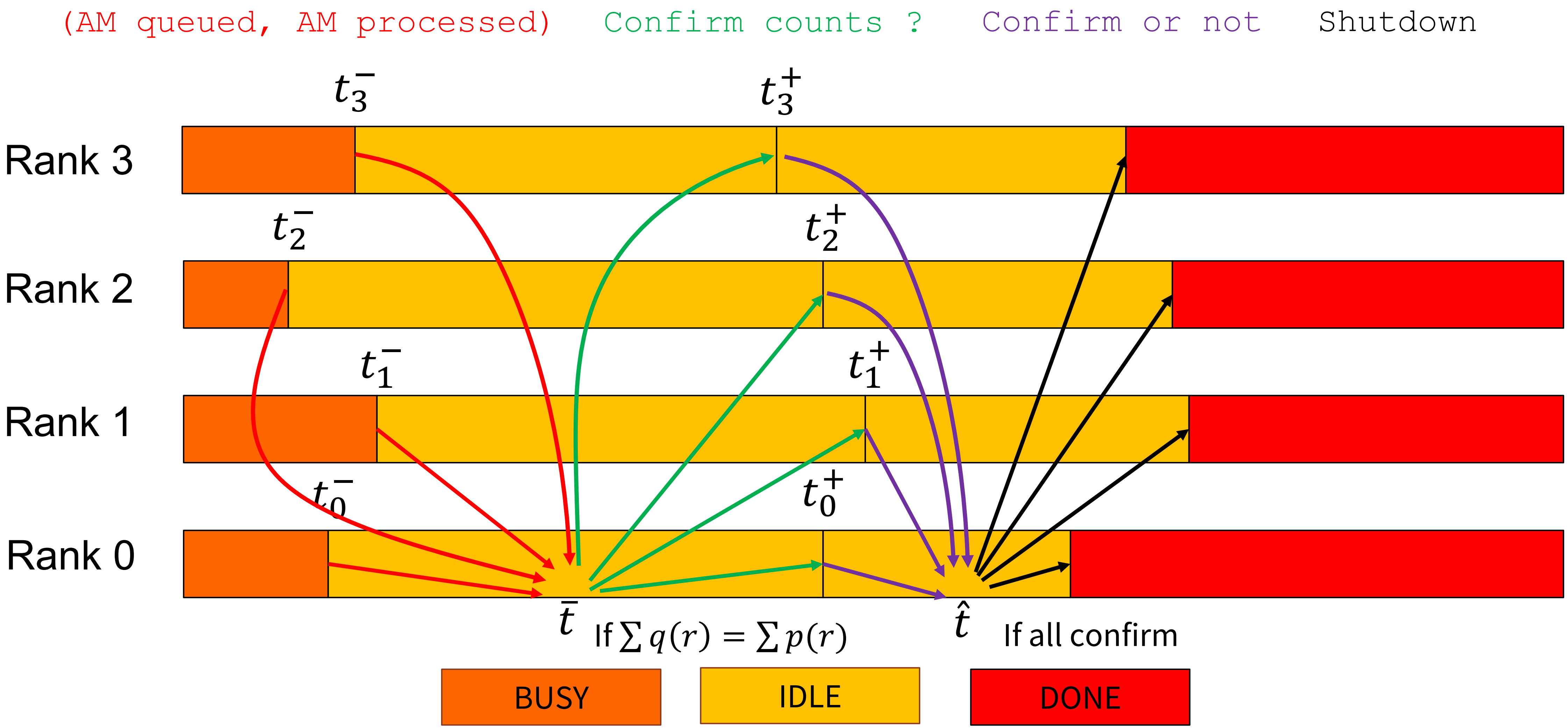

```
/** Initialize structures **/
Communicator comm(MPI_COMM_WORLD);
Threadpool tp(n_threads, &comm);
Taskflow<int> tf(&tp);
/** Create active message **/
am = comm.make_active_msg(
  [&](int d, int k, payload pk) {
    data[k] = pk;
    tf.fulfill_promise(d);
  });
/** Define Taskflow **/
tf.set_mapping(mapping);
tf.set_indegree(n_deps);
tf.set_run([&](int k) {
  compute(k);
  for (auto d : deps(k)) {
    int dest = task_2_rank(d);
    if (dest == my_rank) {
      tf.fulfill_promise(d);
    } else {
      am->send(dest, d, k, data[k]);
    }
  }
});
/** Start initial tasks **/
for (auto k : initial_tasks)
  tf.fulfill_promise(k);
/** Wait for completion **/
tp.join();
```

Completion

$\{t_r\}_r$ is completion sequence if...



Detecting completion in two stages



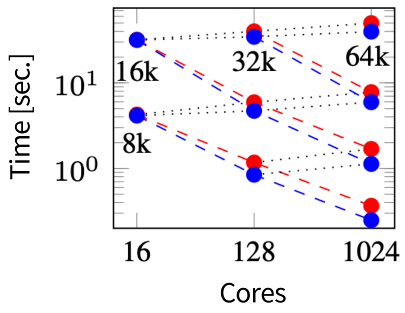
Completion

Algorithm

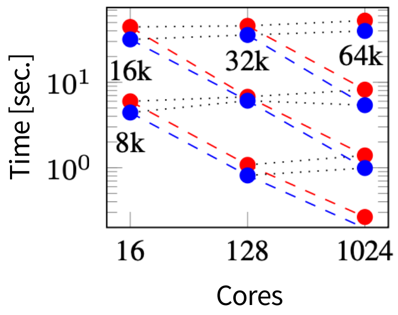
- Is provably correct (it implies completion)
- Has a finite number of messages (if user has finite number of messages)

GEMM

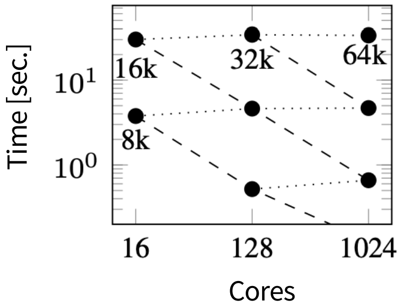
```
gemm_Cikj.set_task([&](int3 ikj){
    int i = ikj[0];
    int k = ikj[1];
    int j = ikj[2];
    C_ij[i + j * num_blocks].noalias() +=
        A_ij[i + k * num_blocks] *
        B_ij[k + j * num_blocks];
    if(k < num_blocks-1) {
        gemm_Cikj.fulfill_promise({i,k+1,j});
    }
}).set_indegree([&](int3 ikj) {
    return (ikj[1] == 0 ? 2 : 3);
}).set_mapping([&](int3 ikj) {
    return (ikj[0] / nprows + ikj[2] / npcols
        * (num_blocks / nprows)) % n_threads;
});
```



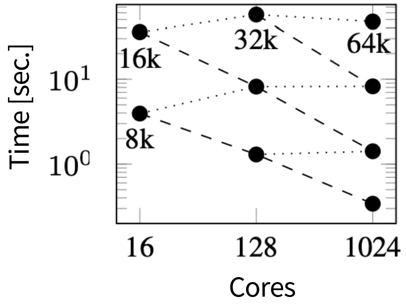
Ttor 2D GEMM



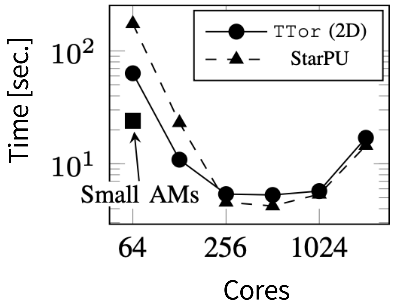
Ttor 3D GEMM



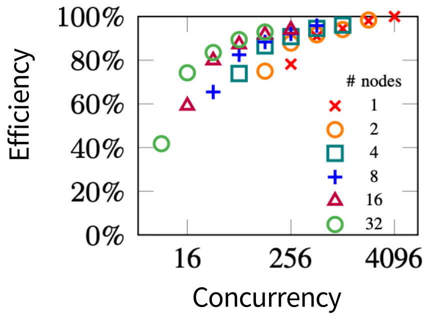
StarPU 2D GEMM



ScaLAPACK 2D GEMM



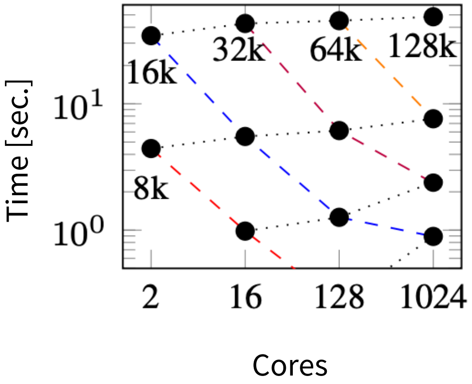
Block size impact
(N=32k, 1024 CPUs)



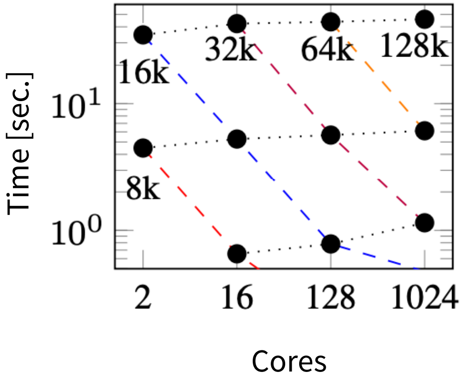
Ttor 2D GEMM concurrency

- Similar performance than StarPU
- PTG is faster than STF when tasks are small

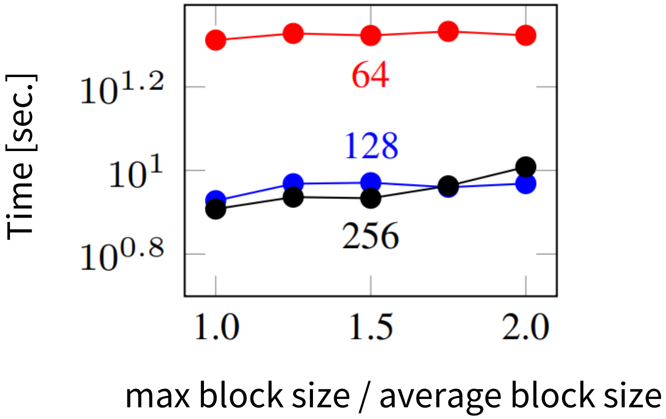
Dense Cholesky



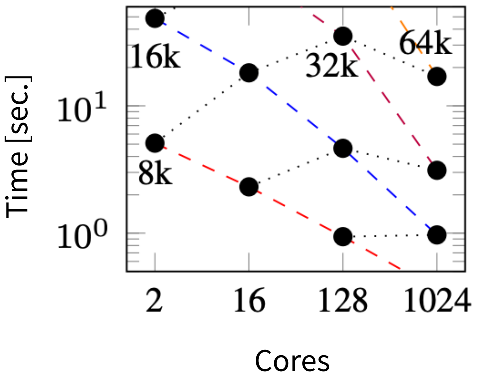
TaskTorrent



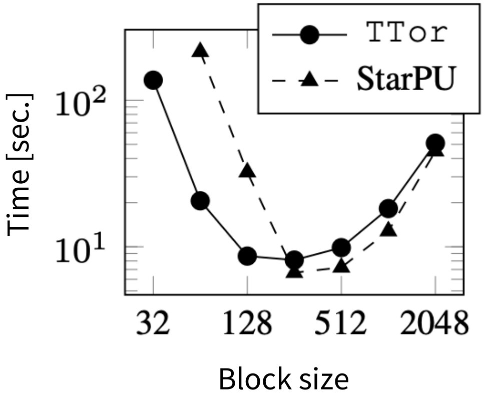
StarPU



Load balancing test
N=65k, 1024 CPUs

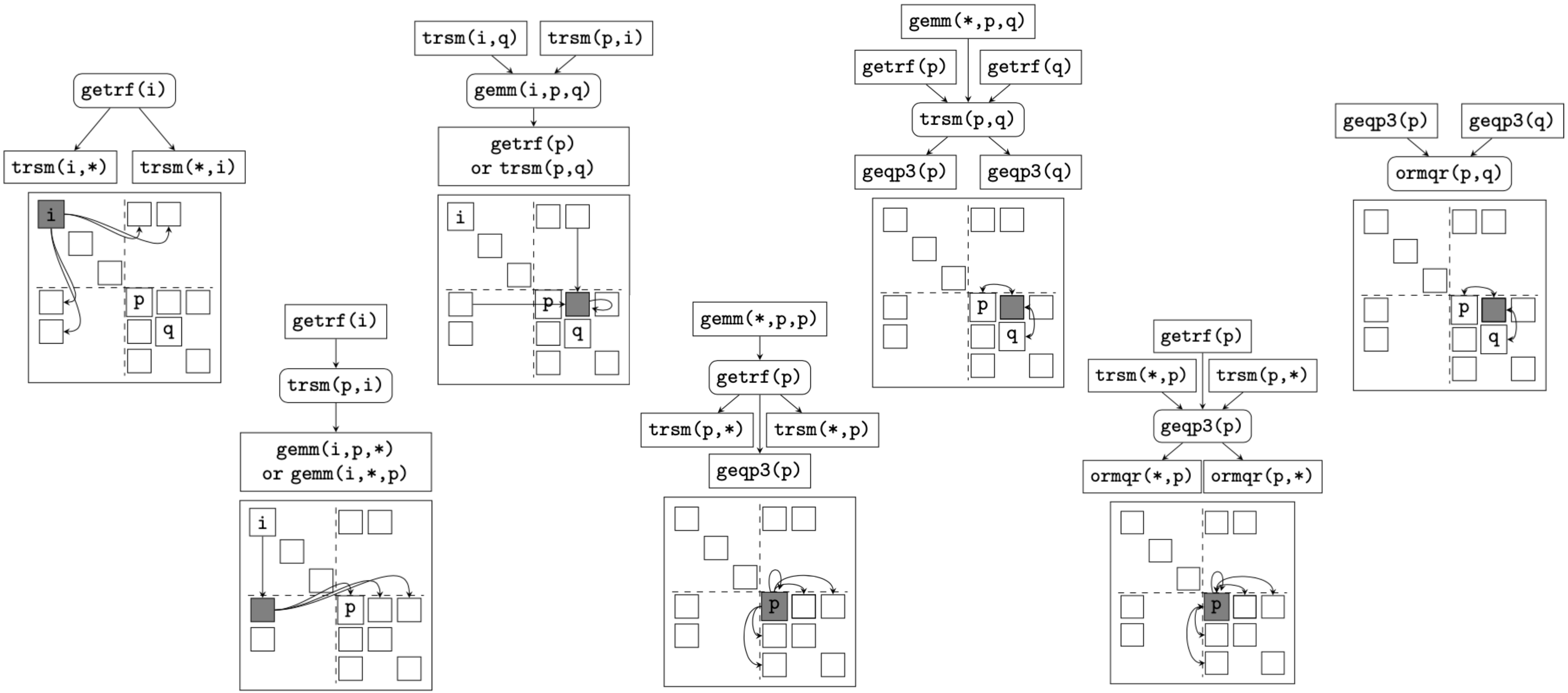


ScaLAPACK



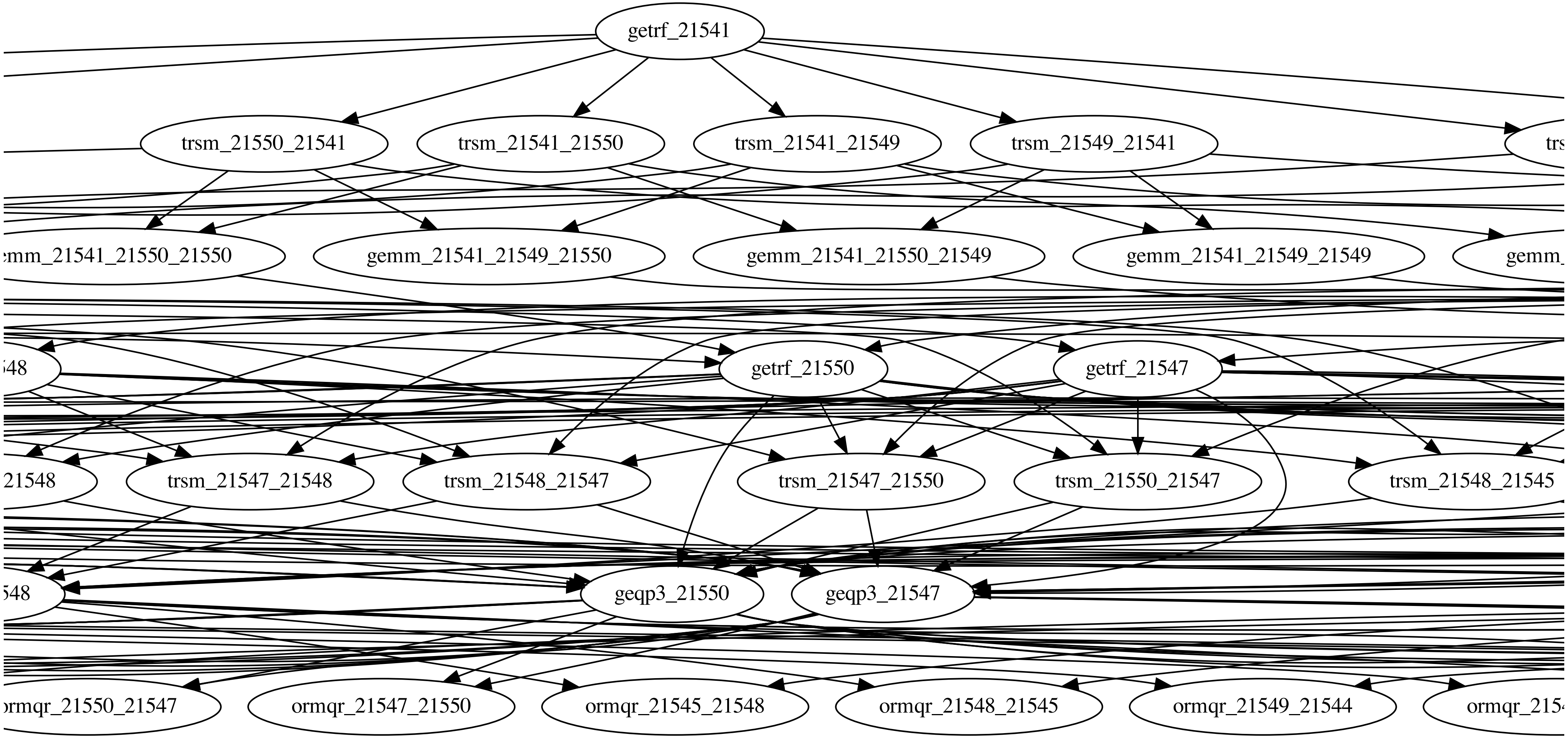
Block size impact
(N=65k, 1024 CPUs)

Fast Sparse Linear Solver



Cambier, Léopold, et al. "An algebraic sparsified nested dissection algorithm using low-rank approximations." *SIAM Journal on Matrix Analysis and Applications* 41.2 (2020): 715-746.

Fast Sparse Linear Solver



Factorization algorithm has log(N) levels. DAG is level-wise.

Fast Sparse Linear Solver

		spaND					AMG (Hypre)	
cores	N	t_{fact}	t_{app}	t_{solve}	$t_{fact}+t_{solve}$	n_{CG}	$t_{fact}+t_{solve}$	n_{CG}
36	1M	6.2	0.14	0.9	7.1	6	15	427
144	4M	7.3	0.15	1.2	8.5	6	16	456
576	18M	8.9	0.15	1.6	10.5	7	22	527
2304	74M	9.8	0.17	1.9	11.7	8	29	627
9216	296M	13.2	0.21	3.7	16.9	12	39	623

On Quartz (LLNL)

TaskTorrent: Links

- Repo with tutorial
  https://github.com/leopoldcambier/tasktorrent
- Benchmarks
  https://github.com/leopoldcambier/tasktorrent/tree/master/miniapp
- To try it out (requires MPI)

```
$ git clone https://github.com/leopoldcambier/tasktorrent.git
$ cd tasktorrent/tutorial
$ make
$ mpirun -n 2 ./tuto
> Rank 1 hello from ...
> Task 0 is now running on rank 0
> ...
```

ttor

Conclusion & Acknowledgments

TaskTorrent is

- Lightweight, distributed and scalable (with PTG + AMs)
- Easy to learn (no new language, just C++)
- Easy to integrate into existing code, portable (message-passing style with MPI and C++)

Acknowledgements

- Léopold Cambier was supported by a fellowship from Total S.A.
- Yizhou Qian and Eric Darve were supported by a grant from NASA, number 80NSSC18M0152

https://github.com/leopoldcambier/tasktorrent