Task-Graph Scheduling Extensions for Efficient Synchronization and Communication

Seonmyeong Bak*
Georgia Institute of Technology
Atlanta, GA, USA
sbak5@gatech.edu

Oscar Hernandez
Oak Ridge National Laboratory
Oak Ridge, TN, USA
oscar@ornl.gov

Mark Gates
University of Tennessee, Knoxville
Knoxville, TN, USA
mgates3@icl.utk.edu

Piotr Luszczek
University of Tennessee, Knoxville
Knoxville, TN, USA
luszczek@icl.utk.edu

Vivek Sarkar
Georgia Institute of Technology
Atlanta, GA, USA
vsarkar@gatech.edu

Abstract

Task graphs have been studied for decades as a foundation for scheduling irregular parallel applications and incorporated in many programming models including OpenMP. While many high-performance parallel libraries are based on task graphs, they also have additional scheduling requirements, such as synchronization within inner levels of data parallelism and internal blocking communications.

In this paper, we extend task-graph scheduling to support efficient synchronization and communication within tasks. Compared to past work, our scheduler avoids deadlock and oversubscription of worker threads, and refines victim selection to increase the overlap of sibling tasks. To the best of our knowledge, our approach is the first to combine gang-scheduling and work-stealing in a single runtime. Our approach has been evaluated on the SLATE high-performance linear algebra library. Relative to the LLVM OMP runtime, our runtime demonstrates performance improvements of up to 13.82%, 15.2%, and 36.94% for LU, QR, and Cholesky, respectively, evaluated across different configurations related to matrix size, number of nodes, and use of CPUs vs GPUs.

CCS Concepts: • **Computing methodologies** → **Parallel programming languages**; • **Software and its engineering** → **Runtime environments**.

*This author's current affiliation is NVIDIA Corporation.

ICS '21, June 14–17, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8335-6/21/06…$15.00
https://doi.org/10.1145/3447818.3461616

Keywords: Gang Scheduling, Task Graph, OpenMP, Runtime System, Work Stealing

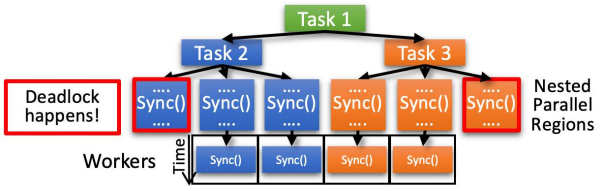
ACM Reference Format:
Seonmyeong Bak, Oscar Hernandez, Mark Gates, Piotr Luszczek, and Vivek Sarkar. 2021. Task-Graph Scheduling Extensions for Efficient Synchronization and Communication . In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3447818.3461616

1 Introduction

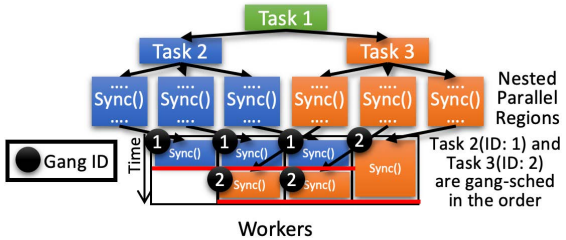
On-node parallelism in high-performance computing systems has increased significantly in recent years. This massive amount of parallelism has the potential to deliver significant speedups, but there is a concomitant burden on application developers to exploit this parallelism efficiently while facing challenges such as inherent load imbalances and communication/synchronization requirements. One popular approach to reducing the complexity of application development for modern processors is to introduce high-performance libraries. High-performance linear algebra libraries have pioneered the use of task graphs to deal with load imbalances in parallel kernels such as LU, QR, and Cholesky factorizations while also exploiting data locality across dependent tasks.

At the same time, there is now increased support for task-parallel execution models with task dependencies in modern parallel programming models, such as OpenMP. Many task graphs in real-world applications include library calls or nested instances of parallel regions that involve blocking operations such as barriers. They often include interleaved sequences of communication and computation operations for latency hiding. It's critical for performance to efficiently compose these multiple parallel instances with low-level synchronization primitives. A key motivation for our work is that current task-based programming systems do not handle this composition effectively.

The most typical way to schedule the multiple parallel instances with blocking sychronizations on task graphs is to spawn a pool of kernel-level threads for each instance, which leads to oversubscription on the underlying hardware threads. This oversubscription can delay intra/inter-node communication or synchronization operations, which often occur in periodic time steps. The delay can lead to overall degraded performance as a result. Scheduling these operations without interference from other parallel regions helps reduce the overall critical path of the application. One approach to addressing the challenge of oversubscription due to multiple parallel instances is to adopt the use of user-level contexts such as tasks and user-level threads (ULTs)[5, 23] to map multiple parallel instances onto a fixed number of worker threads. However, adopting user-level contexts can lead to deadlock because all user-level contexts from the same parallel region are not guaranteed to be scheduled simultaneously onto worker threads when a low-level blocking operation occurs. This is because the user-level contexts are usually non-preemtible and the operating system has lack of control over them, which causes a deadlock when they're blocked on low-level synchronization primitives. Figure 1(a) shows how adopting user-level contexts such as tasks or ULTs can lead to deadlock when a parallel instance contains blocking synchronization operations. This composition problem exists in most task-based programming models because they cannot take control of the low-level blocking primitives in parallel instances that arise from user-written codes or library calls.



(a) Deadlock in multiple parallel instances from tasks or ULTs



(b) Deadlock avoidance with gang-scheduling of parallel instances

Figure 1. Deadlock issues across multiple parallel instances created as user-level contexts such as tasks or user-level threads(ULTs)

In this work, we show how a standard task scheduling runtime system can be extended to support the real-world constraints discussed above by (1) combining gang-scheduling [33] and work-stealing [8] and (2) supporting hybrid victim selection. We also implement our approach in LLVM OpenMP runtime. Thus, for the rest of this paper, we use OpenMP terms to explain our approach.

Our approach provides deadlock-avoidance in the scenario where multiple user-level contexts are synchronized with blocking operations. The integration of gang-scheduling with work-stealing helps nested parallel regions run efficiently without oversubscription and deadlock. The parallel regions to be gang-scheduled are created as ULTs and scheduled onto a set of least loaded cores, as shown in Figure 1(b). Workers can schedule other tasks in work-stealing mode while they are gang-scheduling ULTs from specified parallel regions. Gang-scheduled parallel regions(or simply, gangs) are assigned a monotonic identifier to enforce an implicit ordering of gangs and their ULTs. This monotonic identifier guarantees multiple gangs and their ULTs can be scheduled without being blocked as in Figure 1(a) and able to be stolen by idle workers with deadlock avoidance, which compares the identifier of the current worker and the gang-scheduled ULTs without bookkeeping on a global data structure as in prior work on gang-scheduling implementations. This hybrid combination of gang-scheduling and work-stealing reduces interference and increases data locality for data parallel tasks that involve synchronization and communication in each time step. An application developer can use our API and environmental variable to apply this gang scheduling to a specific parallel region or globally throughout the program.

In addition to gang-scheduling, our runtime system adopts a hybrid victim selection policy in work-stealing to facilitate communication-computation overlap. We've discovered that a heuristic for victim-selection, which starts random stealing with the last successful victim for the first steal trial, can prevent the overlapping of communication and computation on task graphs as in *Locality* case of Figure 2. Figure 2 shows the performance difference from different victim selection policies.

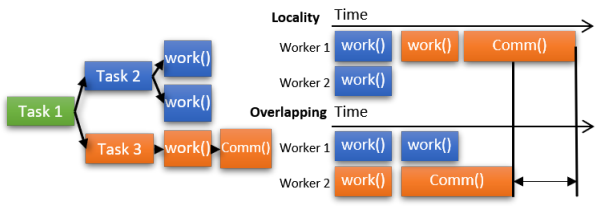


Figure 2. Difference in critical path of mixed sequences of communication and computations. Hybrid victim selection makes work-stealing alternate between these two sequences.

The existing OpenMP runtime systems such as LLVM OpenMP schedule tasks as in the *Locality* case using the last successful victim for the first steal trial, while our approach pursues both the *Locality* and the *Overlapping* cases. Ours is the first work to propose and implement a hybrid scheduling of gang-scheduling and work-stealing. Our implementation is demonstrated on real-world examples.

The contributions of this paper are as follows:

- Extension of task-based runtime systems to integrate gang-scheduling with work-stealing in an efficient manner.
- Introduction of hybrid victim selection to increase the overlap of tasks in task graphs while still preserving data locality.
- Evaluation of our approach on real-world linear algebra kernels in the SLATE library: LU, QR, and Cholesky factorizations. Relative to the LLVM OMP runtime, our runtime demonstrates performance improvements of up to 13.82%, 15.2%, and 36.94% for LU, QR, and Cholesky, respectively, evaluated across different configurations.

2 Background

2.1 Task graphs in Task-Level Programming Models

Many task-level parallel programming models have introduced task graphs in different ways to extract parallelism from irregular parallel applications. The first type of interface for task graphs is *explicit task dependency* through objects such as promises and futures in C++ 11 [24], Habanero [6] and Go [17]. Tasks wait on objects until the predecessors of the objects put data on the objects, which resolve the dependencies of the successors. The other type is *implicit task dependency*, which automates the management of objects with the help of compiler and runtime that form dependencies through directives as *depend* in OpenMP 4.0 [31] or data flow of variables in Legion [7]. After the preceeding tasks are completed, dependency of the successors are automatically resolved and they become *ready tasks*.

2.2 User-level threads for Task-Level Programming Models

In parallel programming models, user-level threads (ULTs) have been used to benefit from their lightweight context switching overhead. Storing necessary data for context switching in user space rather than in kernel space reduces the context switching overhead of user-level threads relative to that of kernel-level threads. There have been several implementations of user-level threads that benefit from its lightweight context switching overhead in different contexts [26, 38, 39]. In spite of the benefits of ULTs, they have deadlock issues because of a lack of coordination with kernels as described in Figure (a). The OS kernel cannot identify the status of each ULT, which can lead to deadlock when ULTs perform blocking operations such as barriers and locks. There have also been efforts where runtime systems share ULT information with the OS kernel, such as scheduler activations [4]. However, all previous works required significant changes in both the ULT runtime and OS kernel, which has inhibited the adoption of their APIs in operating systems.

2.3 Gang-scheduling and Work-stealing

Gang-scheduling [19, 33] was initially proposed to reduce the interference of a group of tightly-coupled threads by other threads or processes. Gang-scheduling, as first introduced, uses a matrix to pack thread requests from processes in which each row is scheduled one at a time. Thus, context switching occurs when it moves from one row to the next row, which reduces the delay in communication across threads incurred by unnecessary context switching. However, a waste of resources results when the threads in each gang have a load imbalance or insufficient cores are available to meet their requests. Different packing policies have been proposed to address these inefficiencies [18, 19, 40], but they did not solve the issue completely. Also, gang-scheduling introduces significant overhead through its use of global data structures.

In contrast, work-stealing enables each worker to execute tasks whenever there are available tasks to steal. Each worker creates tasks and pushes them into work-stealing queues which can be global or local. Then, other workers steal tasks from the work-stealing queues by running a work-stealing algorithm. Depending on the work-stealing algorithm used, work-stealing can maximize either load balancing or locality. Optimizing work-stealing for load balancing can reduce locality through context switching and communication delays. Extended work-stealing algorithms have been introduced to alleviate the cost of work-stealing by considering the locality of participating processing elements [1, 13, 21, 30]. Some of the previous work also extended work-stealing to distributed systems [16, 27, 29, 36].

3 Design

This section describes the algorithm and interface we designed to address the limitations of current task-parallel runtimes mentioned in Section 1. We propose the use of *gang-scheduling* to schedule ULTs of a parallel region without oversubscription and deadlock. Our design supports the use of gang-scheduling for specific parallel regions or globally, while other parallel regions and tasks are scheduled with work-stealing. In addition to

gang-scheduling, we also discuss how the victim selection policy, which impacts how a task graph is traversed, affects the overlapping of communication and computation tasks, and we propose a *hybrid victim selection policy* to improve the overlapping supported by the task scheduler.

3.1 Gang-Scheduling of Data-Parallel Tasks

3.1.1 Integrating gang-scheduling with workstealing. Gang-scheduling and work-stealing have not been used together in task scheduling. Each has its advantages and disadvantages as compared to the other. Integrating them so that each can be used in cases when it is beneficial can improve the overall performance of task-parallel applications. We propose extending the *omp parallel* construct to schedule threads of selected parallel regions in gang-scheduling mode. Users can apply gang-scheduling to upcoming or all parallel regions through our proposed API in Listing 1. By default, all top-level parallel regions are scheduled in gang-scheduling mode. Other parallel regions that are not set by the proposed API are scheduled in work-stealing mode by putting all their ULTs into the calling worker's local work-stealing queue. For the rest of this paper, we refer to ULTs to be scheduled in gang-scheduling mode as *gang* ULTs, while other ULTs and tasks are referred to as *normal* ULTs and tasks.

```
export OMP_GANG_SCHED=1; //Apply gang-scheduling to
    all parallel regions
void ompx_set_gang_sched(); // All following parallel
    regions are gang-scheduled after this call
void ompx_reset_gang_sched(); // Parallel regions
    after this call are scheduled in default
    scheduling policy
```

Listing 1. API to apply gang-scheduling to parallel regions

3.1.2 Gang-scheduling of user-level threads. When multiple gang-scheduled parallel regions are running simultaneously, it is important they be scheduled without the possibility of deadlock. To prevent deadlock as described in Figure 1a, we assign a monotonically increasing ***gang id*** to each parallel region, which is incremented atomically across all workers. We use this *gang id* to restrict the scheduling order of gangs so as to guarantee that deadlock does not occur while gangs run in parallel without serialization. Algorithm 1 describes how the *gang* ULTs from a parallel region are assigned the *gang_id* and *nest_level* of the current worker; the runtime system then gang-schedules *gang* ULTs of each parallel region. **GANG_SCHED()** is synchronized by a shared lock in the *fork* stage of a region in the OpenMP runtime. The *fork* phase involves access to global data structures which are synchronized by a global lock for the *fork* and *join* phases in the runtime system. Thus, parallel regions have an inevitable serialization in the *fork* phase, and *gang_sched* contributes a marginal additional waiting time to the *fork* phase of each region.

Algorithm 1 Gang-schedule a Parallel Region with Load Balancing and Monotonic Gang ID

```
1: Init: worker->gang_id = worker->nest_level = -1
       for all workers
2: function FORK_PARALLEL_REGION(n_request, threads)
3:     Obtain_Fork_Lock()
4:     if gang_sched_enabled then
       ▷ set by the interface in Listing 1
5:         GANG_SCHED(n_request, threads)
6:     else
7:         Push(Local_TaskQueue, threads)
8:     Release_Fork_Lock()
9: function GANG_SCHED(n_request, threads)
10:            ▷ Gang-schedule threads to n_request workers
11:    gang_id = GET_MONOTONIC_GANG_ID()
12:    workers = GET_WORKERS(n_request)
13:    n_gang_threads += n_request
14:    for i = 0 to n_request-1 do
15:        threads[i]->gang_id = gang_id
16:        threads[i]->nest_level = cur_worker->nest_level
17:        Push(worker[i]->gang_deq, thread[i])
18: function GET_WORKERS(n_request)
19:        ▷ Retrieve a list of least loaded n_request workers
20:    avg_load = n_gang_threads / n_workers
21:    reserved_workers
         = least_loaded_workers(n_request, avg_load)
       ▷ obtain n_request workers of which gang deq has
    ULTs less than or equal to avg_load
22:    return reserved_workers
```

When each gang is assigned a set of workers ("reserved" workers), the workers that are closer to the current worker and less loaded with gang-scheduled ULTs have higher priority in **GET_WORKERS()**. We assume that all the worker threads are pinned to avoid any migration cost and uncertainty that may be caused by the OS thread scheduler.

Gang ULTs become stealable after they are pushed to the reserved workers' gang ULT deq. Other workers can steal the *gang* ULTs from the reserved workers, which enables an earlier start of *gang* ULTs if the reserved workers for the gang are busy executing other *normal* ULTs and tasks. This is because we only consider the number of *gang* ULTs on each worker in **GANG_SCHED()**. This additional work-stealing resolves unidentified load imbalance without tracking all *normal* ULTs and tasks. The work-stealing of *gang* ULTs happens at every scheduling point, such as barriers, along with *normal* tasks and ULTs. **SCHED_ULT_AND_TASK()** in Algorithm 2 is the scheduler function, which schedules a ULT or task on each worker in the synchronization points. *Gang*

Algorithm 2 Scheduling Gang ULTs with Deadlock Avoidance through the Eligibility Function

```
1: function SCHED_ULT_AND_TASK() ▷ Schedule a ULT or task
2:     if ULT = STEAL_GANG_ULT(Local_Gang_deq) then
3:         return ULT
4:     else if task = POP_TASK(Local_TaskQueue) then
5:         return task
6:     victim = SELECT_VICTIM()
7:     if ULT = STEAL_GANG_ULT(Remote_Gang_deq
8: (victim)) then
9:         return ULT
10:    else
11:        return Steal_Task(Remote_TaskQueue(victim))
12: function STEAL_GANG_ULT(deq)
13:    head = deq->head
14:    ULT = deq[head]
15:    if IS_ELIGIBLE_TO_SCHED(ULT) then
16:        if CAS(&deq->head, head, (head+1)) then
17:            worker->gang_id = ULT->gang_id
18:            worker->nest_level = ULT->nest_level
19:            return ULT
20:    return false
21: function IS_ELIGIBLE_TO_SCHED(ULT)
22:                          ▷ Check if worker can steal ULT
23:    if ULT->nest_level > worker->nest_level then
24:        return true
25:    else if ULT->nest_level == worker->nest_level
           ∧ ULT->gang_id < worker->gang_id then
26:        return true
27:    return false
```

ULTs have the highest priority in work-stealing and go through an additional function, **IS_ELIGIBLE_TO_SCHED()**, to check if each *gang* ULT from a victim worker can be scheduled on the caller.

3.1.3 Deadlock avoidance with the eligibility function. The eligibility function, **IS_ELIGIBLE_TO_SCHED()** compares the *nest-level* and *gang_id* of the current worker with the corresponding variables in the victim *gang* ULT which are assigned in **GANG_SCHED()**. This function guarantees parallel regions are scheduled in a certain partial order where gangs, which are started earlier or in lower nested levels, have precedence over those that started later or are in upper levels. This implicit ordering through the identifier also allows multiple gangs to run simultaneously without serialization of gangs based on certain global data structures. Further, any idle worker can steal and schedule the gang ULTs in the partial order with the eligibility function, which removes unnecessary waiting time which existed in the previous gang-scheduling implementations.

With Algorithm 1 and 2, gangs are pushed in the order of *gang_id*. When some of workers are trying to steal in synchronization points, they're allowed to steal from ULTs with smaller *gang_id* and bigger *nest_level*, through the eligiblity function. However, the function prevents stealing ULTs with a bigger *gang_id* and smaller *nest_level* than the caller, which makes gangs in the task graph to be scheduled in one direction without cycles.

In this way, our gang-scheduling approach prevents deadlock of multiple parallel regions contending on the same pool of workers as described in Figure 1(b).

3.1.4 Comparison with previous work. With the algorithms and heuristics described in this section, only selected parallel regions are guaranteed to be scheduled in gang-scheduling mode. The gang-scheduling we proposed is relatively relaxed compared with previous work because our algorithm guarantees a parallel region to run simultaneously at some point in runtime. Some of the threads in the region can run earlier than others, which results in less waiting time and more efficient use of workers. Our scheme doesn't require a global table to keep track of threads and reduces waiting time by allowing each region to start immediately and to make ULTs stealable after being gang-scheduled.

3.2 Hybrid Victim Selection for Overlapping and Data Locality

Task graphs involving communication and computation tasks are commonly used to exploit parallelism by overlapping tasks in different iterations of iterative applications. In linear algebra kernels, block-based algorithms have similar task graphs to overlap the waiting time of current tasks by doing some computation for the next tasks. As mentioned in Section 1, many task-level runtime systems use heuristics to schedule tasks in task graphs to maximize data locality and minimize failed steal trials. One of the common heuristics is to do random stealing with the first victim from previous victims

Algorithm 3 Work-Stealing with hybrid of history and random victim selection

```
1: Init: prev_victim_id[] for all workers
2: function DO_WORKSTEALING()
3:     victim = SELECT_VICTIM()
4:     if task = STEAL_TASK(victim) then
5:         Push(prev_victim_id, victim)
6:         Push(prev_victim_id, -1)
7:     else
8:         Pop(prev_victim_id)
9:     return task
10: function SELECT_VICTIM
11:    if Top(prev_victim_id) >= 0 then
12:        return Top(prev_victim_id)
13:    else
14:        return Random_VICTIM()
```

where the steal was successful. This heuristic is intuitively helpful when there are a few tasks in the graph while other workers are idle. Tasks graphs for irregular workload often have a few sibling tasks at a time. Each of the tasks on the graph is paralellized in fine-grained manner through a group of child tasks or nested-parallel regions. This heuristic makes all workers to work on the few victim workers where the tasks in the top-level graph are scheduled and child tasks generated from them to be stolen without unnecessary steal failures.

However, this heuristic may prevent the overlapping of communication and computation across sibling tasks. For example, there's a task graph starting with a root task. With the heuristic, the initial thread, which runs the root task and creates the sibling tasks in the middle of the root tasks, is the only thread with tasks, which makes all other workers mark this thread as their successful victim. So, this initial thread keep creating tasks as a producer while others continue stealing from them. Even though the stolen tasks have also some child tasks, other workers keep stealing from the initial thread until it becomes empty. This heuristic doesn't hamper the overall progress of the task graph if all tasks are computation but may reduce the overlapping of computation and communication in the task graphs we're targeting.

To resolve these unintended anomalies while we keep the benefit of this heuristic, we came up with a simple heuristic to make random stealing to start with and without previous successful victims alternatively. This simple heuristic can make threads to steal from the few loaded busy threads in the scenarios where the heuristic is beneficial with at most one failed steal trial, while the overlapping of communication and computation tasks happens as intended.

Algorithm 3 is a combined algorithm that chooses victim workers for stealing. Each worker calls *do_workstealing* when their local-task queue is empty and waiting for other threads on any synchronization point. First, each worker tries to steal the victim thread where the previous successful steal happens. If this steal turns out to be successful, then it pushes the victim thread id and a value, which cannot be a thread id(-1 in Algorithm 3). This value makes the worker try random-stealing after a successful steal. If the current steal fails, regardless of whether it uses the previous victim or a randomly chosen victim, it moves back to its previous slot in the *prev_victim_id*. If the entry has a valid victim thread id, this worker will try to steal from the victim where the latest successful steal occurred. If not, it keeps stealing from randomly chosen victims. This combined selection of victim from history and random method prevents workers from repeatedly stealing from the same victim, which would result in a serialized sequence of communication and computation without overlapping.

4 Implementation

In this section, we introduce our integrated runtime system of Habanero-C library and LLVM OpenMP runtime to implement the proposed gang-scheduling algorithm and victim selection policy.

4.1 Overview of Our Implementation

We integrated LLVM OpenMP runtime and Habanero-C library (HClib) to use HClib's user-level threading routines. This integrated runtime creates OpenMP threads as user-level threads that run on HClib workers. This runtime can run pure C++ codes using HClib APIs, OpenMP codes, and HClib with OpenMP codes. In this work, we use pure OpenMP codes to focus on the task dependency graph issues in production-level applications. The user needs to load this library to their application binary using OpenMP through *LD_PRELOAD*. The LLVM OpenMP runtime supports gcc, icc, and clang, so any OpenMP binary built with the compilers can run on our integrated runtime without any change to their codes, which runs on HClib workers without our gang-scheduling. User needs to call our APIs to schedule their parallel regions in gang-scheduling as described in Section 3. Hybrid-victim selection is by default enabled.

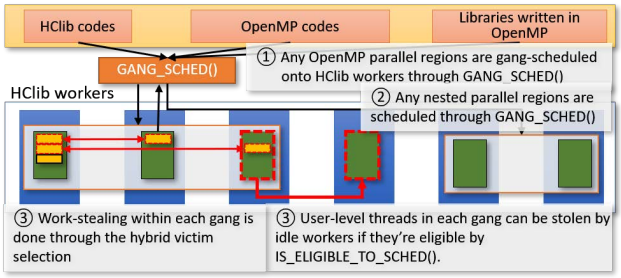


Figure 3. Implementation of Integrated HClib and OpenMP runtime

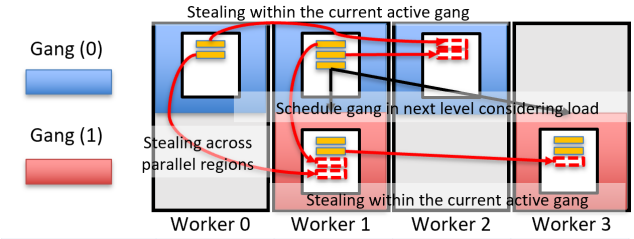
Figure 3 shows how OpenMP instances are scheduled onto HClib workers when gang-scheduling is enabled through the interface in Algorithm 1. User-level threads in each gang can be stolen by idle workers. When idle workers try to steal a ULT from any gang, they check with *IS_ELIGIBLE_SCHED* function if it is fine to schedule the ULT by comparing their active *gang_id* and *nest_level* with the ULT. Within each gang, OpenMP threads steal tasks through the hybrid victim selection. In the following sections, we will describe how we implement gang-scheduling and work-stealing for nested-parallel regions in this integrated runtime system.

4.2 Scheduling of Parallel Regions on the shared pool of workers

Multiple OpenMP instances can run on this integrated runtime system by gang-scheduling and work-stealing, so

workers may have different nest-levels. User-level threads from each OpenMP instance running on the workers should be able to get access to each other. So, we implemented that each worker has arrays for its active *gang_id*, *nest_level* and *thread_array*. These arrays are indexed by *internal_nest_level* of each worker to point to active entries in the arrays for the current running parallel region. These variables are used to schedule multiple parallel regions simultaneously in gang-scheduling and support work-stealing of ULTs and tasks within/across parallel regions.

Figure 4 shows how our implementation schedules multiple parallel regions onto the shared workers. When any ULT on each worker tries to schedule a new OpenMP instance onto workers, it creates a new *thread_array* which is assigned an atomically incremented *gang_id*. Each ULT also contains a copy of *gang_id*, *nest_level* and pointer to *thread_array*. In Figure 4, there are two parallel regions, *gang(0)* with 3 ULTs and *gang(1)* with 2 ULTs. *Gang(0)* is pushed to the *worker [0,1,2]* and *gang(1)* is placed on *worker 1* and *3*. The active entries on each worker are accessed through *internal_nest_level* which starts from 0 and is incremented when a new parallel region is scheduled. For example, *worker 1* first schedules *thread 1* from *gang(0)* and then it schedules the nested parallel region, *gang(1)*, which increments *internal_nest_level* from **0** to **1**. When ULTs are scheduled by worker threads, *gang_id*, *nest_level* and *thread_array* on each worker are updated. ULTs also update its corresponding entry in the shared *thread_array*. *Gang_0* and *gang_1* in Figure 4 show the updated information of worker id and internel_nest_level where ULTs are



Internal_nest_level (current active level)	0	0 -> 1	0	0
gang_id [internal_nest_level]	gang_id[0] = 0	gang_id[0] = 0 gang_id[1] = 1	gang_id[0] = 0	gang_id[0] = 1
nest_level [internal_nest_level]	nest_level[0]=0	nest_level[0]=0 nest_level[1]=1	nest_level[0]=0	nest_level[0]=1
thread_array [internal_nest_level]	thread_array[0] = gang_0	thread_array[0] = gang_0 thread_array[1] = gang_1	thread_array[0] = gang_0	thread_array[0] = gang_1
queues [internal_nest_level]	queues[0] = WSQ[0,0]	queues[0] = WSQ[0,1] queues[1] = WSQ[1,0]	queues[0] =WSQ[0,2]	queues[0] =WSQ[1,1]

gang_0 = {{0,0}, {1,0}, {2,0}}, gang_1 = {{1,1}, {3,0}}
gang_x[i] = {***worker id***, ***internal_nest_level*** when the ULT[i] at gang(x) is scheduled}
WSQ[x,y] = Work-stealing queue for thread y at gang x)

Figure 4. Gang-scheduling for nest-parallel regions and Work-stealing within and across gangs

scheduled. When a ULT tries to steal a task from other ULTs within each gang, this information is used to get access to a work-stealing queue of a victim ULT.

When a idle worker tries to steal a ULT from other busy workers, first it checks whether the ULT is eligible through the *IS_ELIGIBLE_SCHED*, and then steal the ULT. After the ULT is stolen, the worker copies the information of the ULT to its local entries indexed by incremented *internal_nest_level* for *gang_id*, *nest_level* and *thread_array*. Each worker keeps a separate array of queues for *normal* ULTs and tasks indexed by *internal_nest_level*, which are reused without being reallocated for each new instance. For *gang* ULTs, each worker has a local *gang_deq* where a master thread initiating a parallel region pushes a *gang* ULT through *gang_sched* function in Algorithm 1, which has highest priority over other queues. Each worker gets a ULT by atomically popping from this *gang_deq*. On any scheduling point, each worker checks this queue first before they schedule tasks in *queues[internal_nest_level]*.

5 Application Study

We use three linear algebra kernels from the SLATE library [20] to showcase the benefits of our work: LU, QR, and Cholesky. SLATE is a state-of-the-art library developed by the University of Tennessee that is designed to make efficient use of the latest multicore CPUs and GPUs in large-scale computing with common parallel computing techniques such as wavefront parallelism for latency hiding and heterogeneous use of CPU and GPU in distributed environments. SLATE outperforms existing vendor-provided libraries and its predecessor, ScaLAPACK [14]. For our evaluation, we used the NERSC Cori GPU cluster and built SLATE from its main repository(https://bitbucket.org/icl/slate) with the configuration in Table 1. For the baseline OpenMP runtime system, we used the LLVM OpenMP runtime, which was forked from the LLVM github repository on 06/29/2020.

Hardware Configuration (per node)		Software Configuration	
Cluster	NERSC Cori GPU	**SLATE**	06/22/2020 Commit
CPU	2 x Intel Skylake 6148 (20C, 40SMT)	**Compiler**	GCC 8.3
GPU	8 x Nvidia V100	**MKL**	2020.0.166
NIC	4 x dual-port Mellanox EDR	**CUDA/MPI**	10.2.89, OpenMPI 4.0.3

Table 1. Hardware/Software Configuration for Experiments

We tested different configurations of ranks-per-node and cores-per-rank using the LLVM OMP baseline, and selected the best configurations for all our experiments as follows. For LU and QR, we ran each kernel with 4 MPI ranks on each node with 10 OpenMP threads per rank , while for Cholesky, we used 2 MPI ranks per node with 20 OpenMP threads per rank. For GPU runs, we used 4 GPUs per node which showed the best baseline performance. The OpenMP threads and HClib workers

are pinned in the same fashion, using the best affinity setting among those tested.

We ran SLATE's performance test suite to measure the performance of each kernel in GFlops with different configurations. Each performance measure is a mean of 6 runs after the first run as warm-up. We ran the kernels with small and large matrices to cover common sizes of input matrices on single and multi-node runs. For GPU runs, we used only large matrices where the GPU version starts to outperform the CPU-only runs. For Cholesky, we ran the CPU-only version because the GPU version of Cholesky offloads the trailing matrix update to the GPU, without offering an opportunity to overlap the trailing task and panel task (since no prior runtime was able to exploit this overlap using the victim selection approach in our runtime).

For comparison, we ran the test suite with the ScaLAPACK reference implementation using sequential MKL (denoted by *ScaLAPACK (MKL)*), the SLATE default implementation using *omp task depend* on LLVM OpenMP runtime (denoted by *LLVM OMP*), and the same SLATE implementation on our integrated runtime (denoted by *HClib OMP*).

5.1 Overview of Task Graphs for LU, QR, and Cholesky in SLATE

Figure 5 shows the general form of task graphs for factorization kernels in SLATE. SLATE uses lookahead tasks and panel factorization for overlapping of computation and communication as well as data locality. Factorization kernels factor panels (each panel is a block column) and then send tiles in the factored panel to other ranks so that they can update their next block column and trailing submatrix. Lookahead tasks update the next block column for the next panel factorization, and the trailing submatrix task updates the rest of the trailing submatrix. Panel and lookahead tasks are assigned a higher priority than trailing submatrix computation with a *priority* clause to accelerate the critical path of the task graph, which is supported by only a few OpenMP runtime systems such as GNU OpenMP. Regardless of the support of *priority*, it doesn't guarantee that the scheduling of higher priority tasks will precede lower priority tasks even when it is supported because a *priority* clause simply gives precedence to only *ready tasks* specified with higher priority. The *trailing submatrix[i-1]* task and its child tasks become ready earlier than the *panel task[i]* and its child tasks. For this sequence of tasks, the common history-based work-stealing can prevent the expected overlapping of computation in *trailing submatrix* and communication in *panel task*. Cholesky factorization has significant degradation from this anomaly as shown in Figure 7(b).

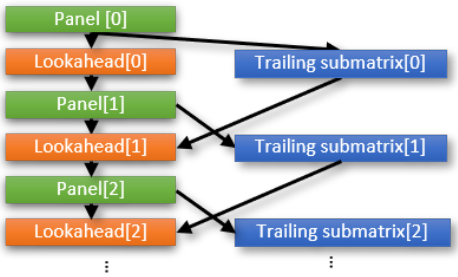


Figure 5. Simplified task graph of factorization kernels in SLATE

Each factorization kernel has a different series of computations and communication routines in the panel, lookahead, and trailing submatrix tasks depending on its algorithm. Each of the tasks consists of a block of columns. In the following sections, we'll discuss in detail how our suggested approaches improve the performance of these kernels.

5.2 LU, QR Factorization: Gang-Scheduling of Parallel Panel Factorization

LU factorization is a basic factorization kernel for solving linear systems of equations in which the coefficient matrices are non-symmetric. Several optimizations for LU factorization have been suggested. SLATE adopts a multi-threaded panel algorithm to achieve a best-performing LU implementation [28]. Figure 6 shows what each task in the task graph in Figure 5 does in the LU and QR factorization of SLATE. First, the LU factorization in SLATE does a panel factorization on a block of columns in panel tasks. The panel factorization is parallelized in a nested-parallel region.

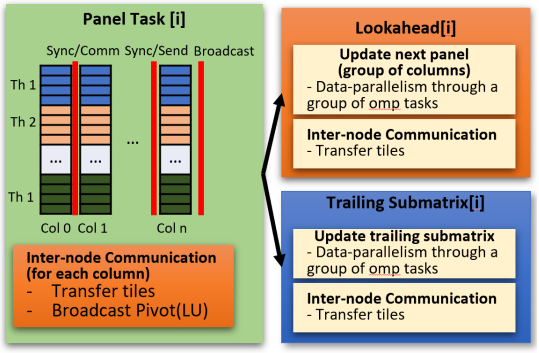


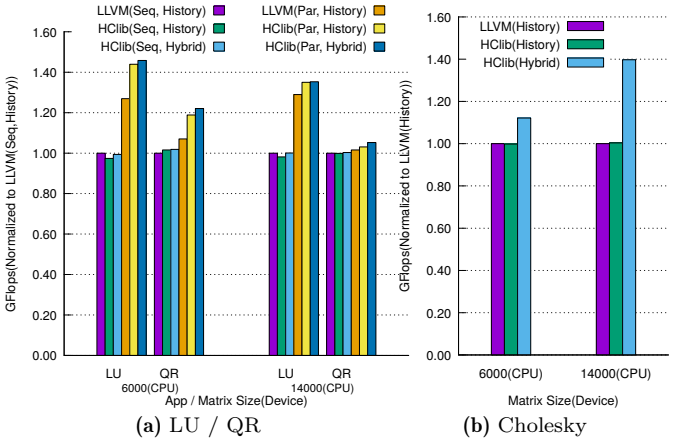
Figure 6. Panel, lookahead, and submatrix tasks of LU and QR in SLATE with 2 threads for the nested parallel region in Panel task

Each panel is internally decomposed into tiles. Each thread is persistently assigned tiles in a round-robin manner, which helps cache reuse and load balancing. Each thread factors a column, and an updated trailing matrix

in the assigned blocks is synchronized at the end of each step (using a custom barrier operation in the library), until a master thread does partial pivoting across threads and other ranks. Because of these synchronizations, a user-level threaded runtime without coordination can lead to deadlock. After the panel factorization, all ranks exchange the rows to be swapped for partial pivoting; the first rank broadcasts the top row down the matrix. The default implementation in SLATE uses a nested parallel region for the parallel panel factorization. However, this nested parallel region interrupts the communication and synchronization by oversubscription of threads on the same cores. Our gang-scheduling makes sure the nested parallel region runs on reserved workers without interference from OpenMP threads in the upper level while other workers can schedule trailing submatrix tasks for overlapping. As Figure 5 implies, *trailing submatrix task[i-1]* can run concurrently with *panel task[i]*. The workers, which are scheduled for gang-scheduling, help to execute the trailing submatrix tasks by work-stealing when they reach the join barrier of the nested parallel region.

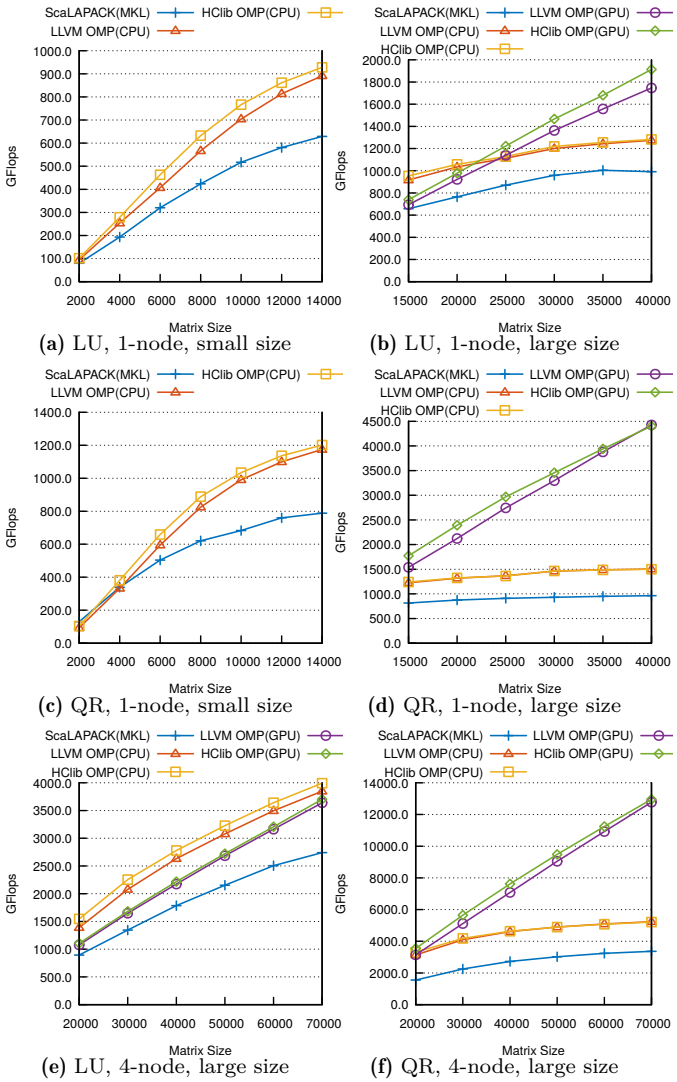
Prior to the detailed evaluation of LU and QR, let's see how much each of our approaches affects the performance of LU and QR. Figure 7a shows the performance difference between LLVM and HClib OMP with gang-scheduling and hybrid victim selection. As shown in Figure 7(a), gang-scheduling gives significant speed-up to both LU and QR while hybrid-victim selection gives incremental benefit to only QR. Hybrid victim selection makes the intended overlapping of sibling tasks to happen. It helps LU and QR to schedule panel and submatrix tasks concurrently but panel tasks in LU and QR takes much longer than trailing submatrix. So, the improved overlapping doesn't lead to significant reduction in the overall execution time of LU and QR. Cholesky is improved significantly by the hybrid victim selection, because its panel tasks take much shorter time than trailing submatrix tasks. This will be explained in Section 5.4. Figure 7 shows that both LLVM and our runtime show similar performance under the same condition(history stealing, sequential panel in LU and QR) , which means both runtimes have similar runtime overhead by default.

Figures 8(a), 8(b), 8(e) show the performance of LU factorization on single- and multi-node runs on Cori GPU in double precision. The LU implementation of



(a) LU / QR (b) Cholesky

Figure 7. Performance Difference of LU,QR and Cholesky with Gang-scheduling(only LU and QR) and and hybrid victim Selection on LLVM and HClib OMP (Seq: Sequential Panel Factorization, Par: Parallel Panel Factorization, Gang-scheduling is applied to HClib(Par))



(a) LU, 1-node, small size (b) LU, 1-node, large size
(c) QR, 1-node, small size (d) QR, 1-node, large size
(e) LU, 4-node, large size (f) QR, 4-node, large size

Figure 8. Performance of LU / QR factorization on single / 4-node of Cori-GPU (Skylake + V100) with double precision (CPU: CPU-Only, GPU: CPU+GPU)

SLATE includes the sequential global pivoting phase after the OpenMP region, so the overall improvement is relatively small compared with other kernels, which is up to 13.82% on CPU-only runs. Our gang-scheduling has diminishing improvement in CPU-only runs with bigger matrices. However, with bigger matrices, the GPU version of LU outperforms CPU-only runs and the reduction in synchronization and communication leads to noticeable improvement in GPU runs. We'll explain this performance trend in CPU-only and GPU runs in the following section.

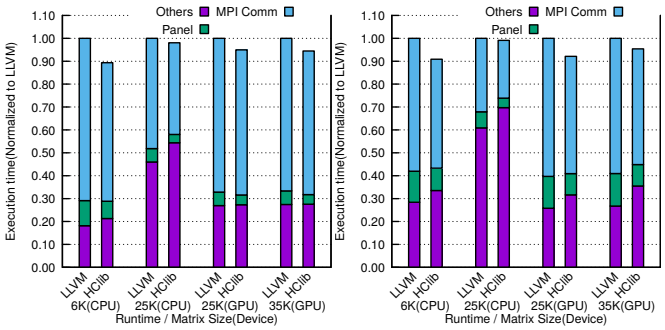
Similarly, QR factorization does parallel panel factorization. Unlike LU, QR doesn't include partial pivoting, so panel tasks in QR do not involve global communication for pivoting and QR doesn't have sequential global pivoting after the parallel region. Thus, QR factorization shows relatively more significant speed-up with our runtime over the baseline LLVM OpenMP runtime with oversubscription compared with LU factorization. SLATE uses a communication-avoiding QR algorithm for QR factorization. It doesn't include any communication in the panel factorization, while each panel task transfers the tiles factored after the panel factorization to other ranks before it proceeds with lookahead and trailing submatrix tasks. The panel factorization is also the most critical task to the task graph of QR factorization in SLATE. Thus, gang-scheduling helps minimize the interference of the nested parallel regions as it does for LU.

Figures 8(c), 8(d), 8(f) show the performance of QR factorization on single- and multi-node runs. Our work improves the QR factorization up to 14.7% at CPU-only runs and 15.2% at GPU runs on a single node over CPU-only and GPU runs with LLVM OpenMP runtime. Gang-scheduling shows considerable improvement in 4-node runs up to 12.8%. QR factorization also has diminishing returns of improvement with bigger matrices, as explained in the following section.

5.3 Detailed Analysis of Improvement in LU and QR

Figure 9 represents how much MPI routines, panel task and other routines consist of the overall execution time in terms of critical path. The tasks transfer tiles between ranks in the beginning and end of panel, lookahead, and submatrix tasks. So, MPI communication and panel factorization determines the length of the critical path of LU and QR task graphs. Child tasks from lookahead and trailing submatrix tasks run in parallel with these routines to overlap the critical routines, which consists of most portion of *Others*. Each bar is normalized to the total execution time of LLVM with the corresponding input matrix.

The benefits of gang-scheduling in our integrated runtime for single- and multi-node runs diminish for both



(a) LU **(b)** QR

Figure 9. Detailed Critical Path of LU and QR factorization on a single node with LLVM and HClib OMP (CPU: CPU-Only, GPU:CPU+GPU)

LU and QR factorization. Gang-scheduling helps remove the delayed synchronization by oversubscription with deadlock avoidance, which leads to reduction in *Panel*. The reduction makes the tile transfer happen earlier, at the end of the panel task, which shortens the waiting time in other MPI ranks that need the tiles to proceed. This is shown on the reduction of *MPI Comm* in Figure 9. This improvement is diluted with the combined effect of oversubscription. The degree of degradation incurred by oversubscription depends on the inter-barrier time of an application [22]. The bigger input matrix has longer inter-barrier time, which leads to less significant degradation from context switching by oversubscription. Rather, oversubscription hides waiting time from OS and hardware events monitored at the kernel-level, which makes our runtime shows increase in *Others* consisting of single-threaded BLAS kernels. It is because the latency hiding of oversubscription is removed. The decreasing degradation of oversubscription on bigger matrices leads to diminishing returns of gang-scheduling over oversubscription.

However, the benefit of gang-scheduling becomes more significant on the GPU offloaded version because a significant portion of computation in *others* is offloaded to GPUs where oversubscription helps on the large matrices. A larger portion of the single-threaded BLAS kernels is offloaded in LU than in QR. So, QR has diminishing returns on the GPU version as the size of the input matrix becomes bigger. If more computation in QR is offloaded, our gang-scheduling can bring more improvement in QR.

5.4 Cholesky Factorization: Maximized Overlap of Communication and Computation

Cholesky factorization is a decomposition of a Hermitian positive definite matrix into a lower triangular matrix and its conjugate transpose. Cholesky is used for standard scientific computations such as linear least squares and Monte Carlo simulations. It has proven to be twice

as efficient as LU when it is applicable. The panel factorization is much lighter, so lookahead and trailing submatrix tasks are critical to improving the performance of Cholesky. As we mentioned above, *trailing submatrix tasks [i-1]* and *panel task[i]* can run concurrently. LU and QR factorization have heavy *panel tasks* which are parallelized in a nested-parallel region, so any workers that finish lookahead tasks will push dependent panel tasks into ready queues. Most often, they're pushed to the worker's work-stealing queue, so panel tasks are likely to be scheduled just after lookahead tasks. Also, the panel tasks are heavy and take a large portion of execution time, so the degree of overlapping of the panel tasks and trailing submatrix tasks have limited impact on the performance. In Figure 7(b) , Cholesky is highly influenced by the victim policies which affect the overlapping of the two tasks while LU and QR doesn't have much difference by the victim policies as shown in Figure 7(a).

Figure 10 shows that the panel factorization of Cholesky factorization is done in a bunch of independent tasks and takes less time than trailing submatrix tasks, so when the panel task becomes available after its preceding lookahead task is done, child tasks from the preceding trailing submatrix task are already being scheduled. The timing for the child tasks from the panel tasks is determined by how each worker chooses a victim for work-stealing. If they use the typical history-based victim selection, every worker will keep stealing from the worker in which the trailing submatrix is running and create its child tasks. This work-stealing from the same victim leads to a delay in the scheduling of the panel task and less overlapping of inter-node communication on the panel task with the child tasks from the trailing submatrix task.

Figures 11(a), 11(b), 11(c) show the performance of Cholesky factorization. As we expected, the improved overlapping of computation in trailing submatrix tasks and communication in panel tasks enhances the performance of Cholesky factorization significantly. The improvement is more significant with bigger matrices because it takes more time to transfer tiles to other ranks and update the trailing submatrix, which gives more opportunity for overlapping. On a single node, the improvement is up to 36.94% with double-precision. On 4-node runs, the kernel is improved up to 28.83%.

We analyze Cholesky in detail to clarify where the improvement comes from. We profile each OpenMP worker

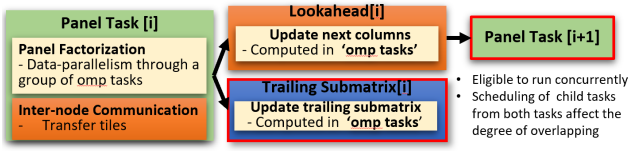
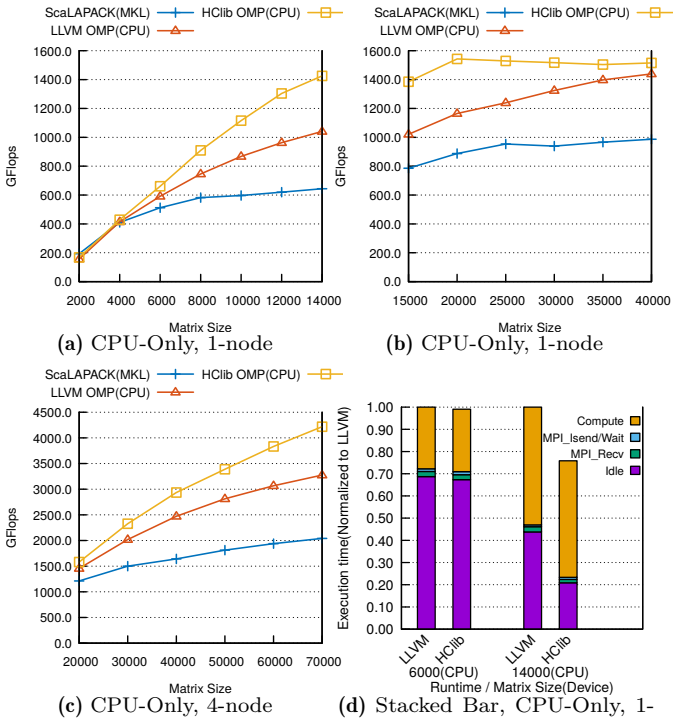


Figure 10. Panel, Lookahead, and Submatrix Tasks of Cholesky in SLATE



(a) CPU-Only, 1-node (b) CPU-Only, 1-node

(c) CPU-Only, 4-node (d) Stacked Bar, CPU-Only, 1-node

Figure 11. Performance of Cholesky factorization on single/4-node of Cori-GPU (Skylake + V100) with double precision (CPU: CPU-Only)

in different MPI ranks and compute the average of each event such as *Idle*, *MPI_Recv*, *MPI_Isend/Wait*, and *Compute* which includes all computations from panel, lookahead, and trailing submatrix tasks. The largest portion of *Idle* consists of waiting time until the updated tiles are received through *MPI_Recv* from other MPI ranks. Figure 11(d) shows the detailed analysis of Cholesky factorization on a single node with two matrix sizes on LLVM and HClib OMP. In the small matrix, the amount of computation is relatively small, which doesn't affect the degree of overlapping significantly regardless of when MPI routines are called. However, on the large matrix, the computation from the trailing submatrix takes longer time, which can overlap MPI routines. So, our victim selection successfully hides the latency of MPI routines, which leads to significant reduction in the overall idle time.

6 Related Work

6.1 Priority and Criticality of Tasks on Task Graphs

There have been a couple of previous works regarding criticality and priority of tasks on a task graph. Criticality aware task scheduler and its application [10, 15] suggested runtime extensions to assign higher priority(critical) to tasks on the longest path of a task graph.

J. Richard et al. [37] studied the overlapping of OpenMP tasks with asynchronous MPI routines in which the application uses the *priority* clause and task loops. These works accelerate the execution of a task graph by scheduling certain tasks first, which is at odds with the task graphs where multiple sibling tasks are created for latency hiding. It can rather prevent intended overlapping of multiple sibling tasks and its child tasks.

6.2 Runtime Systems Based on User-level Threads

User-level threads have been adopted to benefit from their lightweight context switching cost. One of the most common uses of ULTs is to remove the oversubscription by multiple parallel regions. Lithe [34] resolved the composability of different OpenMP instances by providing a dedicated partition of cores to each instance through user-level contexts. However, this partitioning can lead to less resource utilization because of imbalanced loads across instances. Several runtime systems [5, 23, 32, 35] share the underlying kernel-level threads through work-stealing or their own scheduling algorithm with ULTs. They tried to make use of the lightweight context switching cost of ULTs in different contexts but couldn't resolve the deadlock issue completely. Shenango [32] tried to provide a bypass for blocking kernel calls, but other blocking operations used in library calls or written by users can lead to a deadlock. Our work benefits from the advantages of ULTs without deadlock or inefficient resource utilization due to coarse-grained partitioning.

6.3 Communication and Computation Overlap

Asynchronous parallel programming models [2, 11, 12, 25] have been suggested for overlapping by making all of the function calls asynchronous, which directs the runtime system to interleave communication and computation inherently. However, the asynchronous parallel programming models require significant effort on the part of users to write their applications explicitly without deadlock, and tracking control flow of functions calls is not intuitive. To reduce this burden in explicit parallel programming, there have been introduced implicit parallel programming models such as Legion, PaRSEC, and StarPU [3, 7, 9]. These implicit parallel programming models extract parallelism from user codes and handles the communication and synchronization implicitly in their runtime internals.

7 Conclusion

In this work, we proposed gang-scheduling and hybrid victim selection in our runtime system to improve the performance of task graphs involving inter/intra-node communication and computation. Our approach schedules nested parallel regions involving blocking synchronizations and global communications with minimal interference as well as with desirable data locality. It is implemented efficiently using a monotonic identifier and an eligibility function to enforce an ordering of gangs so as to ensure the absence of deadlock. Also, it interoperates with work-stealing to minimize unused resources within and across gangs. Our suggested victim selection resolved the problem of the common heuristic based on a history of previously successful steals by applying random-stealing and history-based alternatives within a fixed window size to overlap communication and computation.

We evaluated our work on three commonly used linear algebra kernels, LU, QR, and Cholesky factorizations, from the state of the art SLATE library. Our approach showed an improvement for LU of 13.82% on a single node in double precision and of 11.36% on multiple nodes. The improvements for QR went up to 15.21% on a single node and 12.78% on four nodes with double precision. Cholesky factorization was improved up to 36.94% on a single node and 28.83% on multiple nodes by our hybrid victim selection. Finally, our approach is applicable to any application written using task graphs that also needs to perform additional synchronization and communication operations as in the SLATE library.

Acknowledgment

This work was supported 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. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2000. The Data Locality of Work Stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures* (Bar Harbor, Maine, USA) *(SPAA '00)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/341800.341801

[2] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. 2014. Parallel Programming with Migratable Objects: Charm++ in Practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. 647–658.

[3] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. P. Thibault. 2017. Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. *IEEE Transactions on Parallel and Distributed Systems* (2017), 1–1. https://doi.org/10.1109/TPDS.2017.2766064

[4] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. 1992. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 53–79. https://doi.org/10.1145/146941.146944

[5] Seonmyeong Bak, Harshitha Menon, Sam White, Matthias Diener, and Laxmikant V. Kalé. 2018. Multi-Level Load Balancing with an Integrated Runtime Approach. In *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018*. 31–40. https://doi.org/10.1109/CCGRID.2018.00018

[6] Rajkishore Barik, Zoran Budimlic, Vincent Cave, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, et al. 2009. The Habanero Multicore Software Research Project. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM, 735–736.

[7] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*. IEEE Computer Society Press, 66.

[8] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748. https://doi.org/10.1145/324133.324234

[9] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. 2013. PaRSEC: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering* 15, 6 (2013), 36–45.

[10] E. Castillo, M. Moreto, M. Casas, L. Alvarez, E. Vallejo, K. Chronaki, R. Badia, J. L. Bosque, R. Beivide, E. Ayguade, J. Labarta, and M. Valero. 2016. CATA: Criticality Aware Task Acceleration for Multicore Processors. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 413–422. https://doi.org/10.1109/IPDPS.2016.49

[11] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.* 21, 3 (Aug. 2007), 291–312.

[12] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. ACM, New York, NY, USA, 519–538.

[13] Quan Chen, Minyi Guo, and Haibing Guan. 2014. LAWS: Locality-Aware Work-Stealing for Multi-Socket Multi-Core Architectures. In *Proceedings of the 28th ACM International Conference on Supercomputing* (Munich, Germany) *(ICS '14)*. Association for Computing Machinery, New York, NY, USA, 3–12. https://doi.org/10.1145/2597652.2597665

[14] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. 1996. ScaLAPACK: A portable linear algebra library for distributed memory computers — Design issues and performance. In *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, Jack Dongarra, Kaj Madsen, and Jerzy Waśniewski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 95–106.

[15] Kallia Chronaki, Alejandro Rico, Rosa M. Badia, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 329–338. https://doi.org/10.1145/2751205.2751235

[16] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. 2009. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–11.

[17] Alan A.A. Donovan and Brian W. Kernighan. 2015. *The Go Programming Language* (1st ed.). Addison-Wesley Professional.

[18] Dror G. Feitelson. 1996. Packing schemes for gang scheduling. In *Job Scheduling Strategies for Parallel Processing*, Dror G. Feitelson and Larry Rudolph (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 89–110.

[19] Dror G. Feitelson and Larry Rudolph. 1992. Gang scheduling performance benefits for fine-grain synchronization. *J. Parallel and Distrib. Comput.* 16, 4 (1992), 306 – 318. https://doi.org/10.1016/0743-7315(92)90014-E

[20] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. 2019. SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 26, 18 pages. https://doi.org/10.1145/3295500.3356223

[21] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. 2010. SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *ACM Sigplan Notices*, Vol. 45. ACM, 341–342.

[22] C. Iancu, S. Hofmeyr, F. Blagojević, and Y. Zheng. 2010. Oversubscription on multicore processors. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–11.

[23] S. Iwasaki, A. Amer, K. Taura, S. Seo, and P. Balaji. 2019. BOLT: Optimizing OpenMP Parallel Regions with User-Level Threads. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 29–42. https://doi.org/10.1109/PACT.2019.00011

[24] Nicolai M. Josuttis. 2012. *The C++ Standard Library: A Tutorial and Reference* (2nd ed.). Addison-Wesley Professional.

[25] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 6.

[26] L. V. Kale, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. 1996. Converse: an interoperable framework for parallel programming. In *Proceedings of International Conference on Parallel Processing*. 212–217. https://doi.org/10.1109/IPPS.1996.508060

[27] Vivek Kumar, Karthik Murthy, Vivek Sarkar, and Yili Zheng. 2016. Optimized Distributed Work-Stealing. In *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*. 74–77. https://doi.org/10.1109/IA3.2016.019

[28] Jakub Kurzak, Mark Gates, Ali Charara, Asim YarKhan, Ichitaro Yamazaki, and Jack Dongarra. 2019. Linear Systems Solvers for Distributed-Memory Machines with GPU Accelerators. In *Euro-Par 2019: Parallel Processing*, Ramin Yahyapour (Ed.). Springer International Publishing, Cham, 495–506.

[29] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. 2012. Work Stealing and Persistence-Based Load Balancers for Iterative Overdecomposed Applications. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing* (Delft, The Netherlands) *(HPDC '12)*. Association for Computing Machinery, New York, NY, USA, 137–148. https://doi.org/10.1145/2287076.2287103

[30] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. 2014. Optimizing Data Locality for Fork/Join Programs Using Constrained Work Stealing. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 857–868.

[31] OpenMP ARB. 2013. OpenMP Application Program Interface Version 4.0. In *The OpenMP Forum, Tech. Rep.* https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

[32] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[33] John K. Ousterhout. 1982. Scheduling Techniques for Concurrent Systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems, Miami/Ft. Lauderdale, Florida, USA, October 18-22, 1982*. IEEE Computer Society, 22–30.

[34] Heidi Pan, Benjamin Hindman, and Krste Asanoviundefined. 2010. Composing Parallel Software Efficiently with Lithe. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 376–387. https://doi.org/10.1145/1806596.1806639

[35] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 145–160. https://www.usenix.org/conference/osdi18/presentation/qin

[36] Jean-Noël Quintin and Frédéric Wagner. 2010. Hierarchical Work-Stealing. In *Euro-Par 2010 - Parallel Processing*, Pasqua D'Ambra, Mario Guarracino, and Domenico Talia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 217–229.

[37] Jérôme Richard, Guillaume Latu, Julien Bigot, and Thierry Gautier. 2019. Fine-Grained MPI+OpenMP Plasma Simulations: Communication Overlap with Dependent Tasks. In *Euro-Par 2019: Parallel Processing*, Ramin Yahyapour (Ed.). Springer International Publishing, Cham, 419–433.

[38] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman. 2018. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (March 2018), 512–526. https://doi.org/10.1109/TPDS.2017.2766062

[39] K. B. Wheeler, R. C. Murphy, and D. Thain. 2008. Qthreads: An API for programming with millions of lightweight threads. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–8. https://doi.org/10.1109/IPDPS.2008.4536359

[40] Y. Wiseman and D. G. Feitelson. 2003. Paired gang scheduling. *IEEE Transactions on Parallel and Distributed Systems* 14, 6 (June 2003), 581–592. https://doi.org/10.1109/TPDS.2003.1206505