Optimizing Fine-Grained Parallelism Through Dynamic Load Balancing on Multi-Socket Many-Core Systems

1st Wenyi Wang
Department of Computer Science
The University of Chicago
Chicago, USA
wenyiw@uchicago.edu

2nd Maxime Gonthier
Department of Computer Science
The University of Chicago
Chicago, USA
mgonthier@uchicago.edu

3rd Poornima Nookala
Intel
Chicago, USA
nookala.poornima@gmail.com

4th Haochen Pan
Department of Computer Science
The University of Chicago
Chicago, USA
haochenpan@uchicago.edu

5th Ian Foster
Department of Computer Science
The University of Chicago
Chicago, USA
foster@uchicago.edu

6th Ioan Raicu
Department of Computer Science
Illinois Institute of Technology
Chicago, USA
iraicu@cs.iit.edu

7th Kyle Chard
Department of Computer Science
The University of Chicago
Chicago, USA
chard@uchicago.edu

***Abstract*—Achieving efficient task parallelism on many-core architectures is an important challenge. The widely used GNU OpenMP implementation of the popular OpenMP parallel programming model incurs high overhead for fine-grained, short-running tasks due to time spent on runtime synchronization. In this work, we introduce and analyze three key advances that collectively achieve significant performance gains. First, we introduce XQueue, a lock-less concurrent queue implementation to replace GNU's priority task queue and remove the global task lock. Second, we develop a scalable, efficient, and hybrid lock-free/lock-less distributed tree barrier to address the high hardware synchronization overhead from GNU's centralized barrier. Third, we develop two lock-less and NUMA-aware load balancing strategies. We evaluate our implementation using Barcelona OpenMP Task Suite (BOTS) benchmarks. Results from the first and second advances demonstrate up to 1522.8× performance improvement compared to the original GNU OpenMP. Further improvements from lock-less load balancing show up to 4× improvement compared to GNU OpenMP using XQueue. Through a rich set of profiling and instrumentation tools, we are able to investigate the runtime behavior of GNU OpenMP and improve its performance on fine-grained tasks by many orders of magnitude.**

***Index Terms*—OpenMP, Fine-grained tasking, Load Balancing, Lock-less Programming, NUMA-Aware Scheduling**

I. Introduction

The emergence of many-core computing systems with concurrency levels from hundreds on CPUs to thousands on GPUs has motivated the adoption of Non-Uniform Memory Access (NUMA) architectures, which offer asymmetric access to cache and memory banks. Programming these systems using a shared memory programming model requires efficiently mapping threads to cores, which becomes increasingly challenging as the number of cores grows and task runtimes decrease. We focus here on addressing two significant challenges: first, overheads associated with use of hardware primitives to synchronize shared memory accesses across many cores and threads of execution; and second, that shared memory programming models are not NUMA-aware.

In the task parallel programming model, computation is broken down into inter-dependent tasks that can be executed concurrently on various cores while respecting data dependencies. Various parallel languages and libraries support this model, such as OpenMP [1], Cilk [2], and Unified Parallel C (UPC) [3]. We focus on OpenMP, a task-centric model in which higher-level parallel constructs such as loops are translated into fine-granularity tasks with dependencies, which the runtime must dynamically schedule to available resources. When a task is enabled by some thread, it is conceptually queued for execution by a future available thread. Unfortunately, OpenMP implementations and their tasking data structures often scale poorly because of excessive use of expensive synchronization operations such as locks to resolve dependencies [4]–[7].

Two approaches to avoiding the performance degradation associated with locks are *lock-free programming*, which typically rely on atomic hardware operations such as compare-and-swap to synchronize without locks, and *lock-less programming*, which relies on methods for safely manipulating

shared data without using locks. We focus here on lock-less programming in which no atomic primitive is used. Prior work has proposed the use of a lock-less, concurrent, multi-producer multi-consumer (MPMC) task queue, called XQueue, in the LLVM-based OpenMP (LOMP) [4]. That work showed significant performance improvements, often delivering 4–6× speedup, compared to native LLVM OpenMP by reducing the time spent on synchronization operations.

In this paper, we focus on GNU OpenMP (GOMP)—the most common OpenMP implementation that is a part of the mainstream compiler infrastructure GNU Compiler Collection (GCC). We specifically address the unscalable, heavily entangled implementation of GNU OpenMP for fine-grain tasks, overcoming restrictive synchronization such as the excessive use of locks. We then propose, implement, and evaluate lock-less NUMA-aware, dynamic load balancing (DLB) algorithms. The main contributions of this work are as follows:

1) We integrate XQueue, a lock-less, relaxed order MPMC task queue, with GOMP, replacing GNU's unscalable priority task queue and its global task lock. Our benchmark results show that XQueue-enhanced GNU OpenMP (XGOMP) achieves up to 96.5× performance improvement (for an NQueens application) compared to the original GOMP.
2) We propose a new hybrid lock-free (gathering)/lock-less (releasing) distributed tree barrier that yields a theoretical lower bound of half the atomic memory access operations than GNU's existing barrier which relies on a globally shared atomic task counter. Our approach differs from prior work in LOMP/XLOMP that use LLVM's default lock-free (not lock-less) barrier. We show that this optimization elevates the performance by up to 15.8× (NQueens) compared to XGOMP, and up to 1522.8× (NQueens) compared to GOMP.
3) To the best of our knowledge, we propose the first lock-less, NUMA-aware dynamic load balancing algorithms for OpenMP tasking. Our approach significantly mitigates load imbalance for both fine-grained and coarse-grained parallelism and achieves significant performance improvements over all previous work. Our approach differs from prior work in XLOMP that relies on a static load balancing (SLB) approach. We also present a comprehensive study of configuration parameters that shows that all benchmarks considered achieve better performance compared to XGOMPTB.
4) We design and implement new software profiling tools for GNU to study and optimize how hardware, application, and OpenMP characteristics influence tasking performance that improve on mainstream profiling tools such as Intel VTune and Scalasca. We use these tools to collect statistics such as time spent on task creation, task queue operations, task execution, and per-thread task locality, providing a deeper understanding of performance behavior. We learn that data and task locality plays a critical role in affecting application performance.
5) We present guidelines for practitioners based on studies of our lock-less DLB strategies. We explore the relationship between task size, steal size, and performance, and analyze memory and cache behavior.

The rest of this paper is organized as follows. Section II introduces the motivation and background of this work. Section III describes our integration of XQueue in GNU OpenMP and the design of our distributed tree barrier. Section IV describes our lock-less dynamic load balancing strategies. Section V outlines how we instrument our code to measure performance. Section VI presents a performance evaluation of our work. Section IX reviews related work. Section X summarize our contributions.

II. Motivation and Background

The open source GCC is the official compiler for GNU and Linux systems. For OpenMP programs, GCC compiles and links programs with *libgomp* (GOMP), which implements OpenMP standards but with GNU-specific customizations. LLVM also has an implementation of OpenMP and uses its own compiler, *Clang*, primarily designed for performance. The performance gap between the GNU and LLVM OpenMP implementations can be significant: see Fig. 1. Here we show execution times for various OpenMP applications from the Barcelona OpenMP Task Suite (BOTS) benchmark [8] when using GNU OpenMP (GOMP), LLVM OpemMP (LOMP), and an implementation of XQueue in LLVM OpenMP (XLOMP). We run these experiments on an Intel Skylake-192 machine, with 192 cores (384 hardware threads). In some benchmarks, GOMP can be >1000× slower than LOMP and >4400× slower than XLOMP. We attribute the GOMP performance issue to its excessive use of synchronization primitives, including locks and atomic operations.

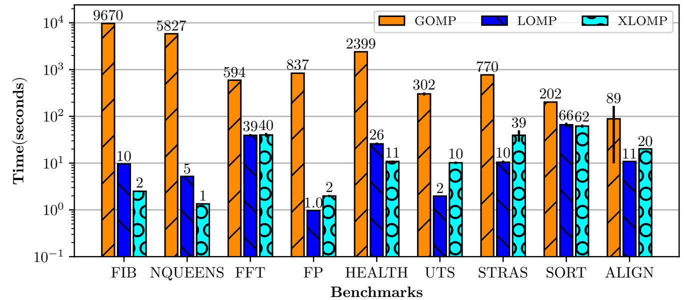

Fig. 1. Execution times for nine BOTS benchmarks with three OpenMP implementations (GOMP, LOMP, and XLOMP) each using 192 threads.

A. GNU OpenMP (GOMP)

The GOMP runtime library handles OpenMP's parallel region as a *team* of worker threads (i.e., workers). An OpenMP *team* creates and manages a group of OpenMP threads. The team itself is often created under the OpenMP directive *parallel*. OpenMP wraps platform-specific or user-defined threads, in our experiments we use the *pthread* [9] library. GNU OpenMP manages tasks, for example, enqueuing and

dequeuing tasks, using a single globally shared priority task queue and a child task queue for each task. The global priority task queue maintains the system-wide information, while the child queue maintains child tasks' dependencies and priorities. GNU OpenMP uses a single global task lock to protect critical regions for task management, scheduling, and other runtime bookkeeping operations. When a worker reaches a scheduling point, it will first acquire the lock before it enters the critical region. When there are no tasks to be executed or scheduled, and current tasks do not have any dependent tasks, the worker exits or waits for other workers to complete.

B. XQueue

XQueue [4] is a lock-less MPMC, out-of-order queuing mechanism that can scale to 100s of threads with little contention for hardware resources. XQueue uses B-queue, a concurrent Single-Producer-Single-Consumer (SPSC) lock-free queue designed for efficient core-to-core communication. The latency of B-queue operations can be as low as 20 cycles.

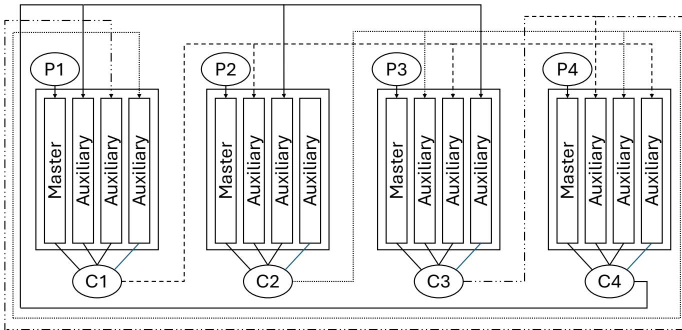

Fig. 2. XQueue on a 4-core system. C_i is consumer i, P_i is producer i.

Fig. 2 shows XQueue on a 4-core system. Each core has one worker thread with one SPSC queue for each other worker: one *Master* queue and the others *Auxiliary* queues. Each item in a queue is a task pointer. Upon task creation, the worker enqueues the task either to its own master queue or to an auxiliary queue of another worker. It applies a round-robin approach across these queues starting with the master queue. If the chosen queue is full, the worker instead executes that task immediately. When dequeuing, the worker first dequeues from the master queue, if no task is found, it then dequeues a task from its auxiliary queues. The dequeued task is also executed immediately.

C. Load Balancing and Work Stealing

Integrating XQueue with OpenMP is likely to lead to load imbalance as the SLB approach will assign tasks to threads without considering other tasks assigned to those threads. Thus it is important to consider more advanced load balancing techniques to improve performance.

One popular load balancing technique is **work stealing**. A worker may be either a *victim* or a *thief*. A thief is an under-loaded worker who tries to steal work from other overloaded workers. A worker being stolen from is a victim. Typically, *pull-based work stealing* is used, in which a thief initiates and pulls a task from a victim. However, the queue from which tasks are stolen requires mutual exclusive access as it is accessed by both thief and victim. As a result, traditional work stealing methods do not fit into our lock-less scenario due to their use of synchronization mechanisms like mutexes, spinlocks, and atomic operations to ensure thread safety and correctness [10]. Use of such methods would lead to the runtime wasting more cycles on cache invalidation, cache misses, and excessive main memory access.

A second issue with existing work stealing techniques is that they are not NUMA-aware. We can distinguish between NUMA-local and NUMA-remote data locality on a shared-memory NUMA architecture with respect to a core. Accessing NUMA-local data incurs less latency, and thus cores spend less time waiting for memory operations and higher throughput. Work stealing algorithms must therefore consider whether a thief is to steal from a NUMA-local worker or a NUMA-remote worker, depending on application characteristics.

III. Fine-Grained Parallelism in GOMP

Our goal is to transform GOMP into a high-performance runtime library that can scale to hundreds of threads and support extremely fine-grained tasks. To this end, we leverage the lock-less, concurrent *XQueue* data structure and replace GOMP's centralized barrier with a distributed tree barrier.

A. XGOMP: XQueue Integration with GOMP

GOMP starts a *parallel* region by first calling `gomp_team_start` and then `gomp_thread_start` to spawn a team of worker threads. The master thread and the worker threads allocate XQueue's task queues inside `gomp_team_start` and `gomp_thread_start`, respectively, to ensure each thread has its task queue ready before it enters the thread dock (the thread dock is a simple barrier that ensures that all worker threads have been properly initialized before the OpenMP tasking routine starts).

We modified `GOMP_task` and `GOMP_taskwait` to decouple GOMP tasking from the global task lock and priority queue. `GOMP_task` is called when an explicit *task* directive is encountered. It allocates a task and ensures its correct state before entering the critical region using the global task lock to enqueue this task to the priority queues. With XQueue, we atomically update the parent task's dependency, push the task directly to the target queue, and atomically increment the global task count, which is used by the centralized barrier.

Many of GOMP's runtime routines rely on variables that are protected by the global task lock. We removed the runtime dependencies of all those variables except the variable that tracks the global task count. The GOMP native tasking barrier uses this count as a termination signal. The count is updated when global task status changes, for example, it decrements when a task finishes. We convert this variable to an atomic variable with an acquire-release memory order strategy.

This phase of implementation removes GNU's significant lock contention for massively parallel, fine-grained tasks.

However, it is possible that excessive use of atomic operations could still be a performance bottleneck. The atomic task count is deeply entangled with GNU's barrier routine, and therefore, we next replace it with a new barrier design.

B. XGOMPTB: Adding an Efficient Distributed Tree Barrier

There are several different barrier implementations in GOMP, all of which are implemented in a centralized manner. Here we specifically focus on the team barrier, which is usually implicitly placed at the end of a *parallel* region during compilation and manages access to the global task count. Updating the states of the team barrier requires acquisition of the global task lock. The centralized team barrier releases when the worker acquires the lock and meets the following conditions: i) it is the last worker entering the same barrier, and ii) the global task count is 0.

In our XGOMP implementation, we modified the centralized team barrier such that it releases when each worker meets the following conditions: i) the global task count is 0; ii) current task of the worker has no dependencies; and iii) all other workers have reached the same barrier.

We then replaced GNU's centralized team barrier with an efficient distributed tree barrier. We adopt a similar gather-release pattern as used in LLVM; however, unlike LLVM's lock-free barrier, we design a new hybrid barrier that performs lock-free gathering and lock-less releasing. Our approach yields a theoretical lower bound of half the atomic memory access operations. Our barrier connects workers in a binary tree topology. A worker is gathered when: i) all workers have entered the barrier; ii) the worker is idle, finding no tasks to execute; iii) the worker's current task has no unfinished dependencies; and iv) all of its children workers' barriers are gathered. A gathered worker atomically updates the *complete* flag of its parent. As this complete flag is only accessible by the child and parent, its hardware contention is low. When the root worker is gathered, it then signals a release with a tree broadcast, lock-lessly updating the *release* flag of each worker so that the worker can safely exit the barrier.

We have now removed all excessive atomic operations in GNU OpenMP. The tree barrier combines lock-free and lock-less techniques; it is also easy to understand and implement. To the best of our knowledge, our approach is the first implementation of a hybrid distributed tree barrier for OpenMP tasking with carefully designed lock-less logic. A distributed tree barrier was briefly introduced to LLVM in 2021; however, it was later reverted due to unresolved errors. At this time, there is no distributed barrier in LLVM. LLVM's barrier employs locks, atomic operations, and does not implement lock-less optimizations.

IV. Dynamic Load Balancing

XQueue uses a static round-robin load balancer to distribute tasks among workers. This approach has two limitations: first, load imbalance can occur, and second, data locality in NUMA machines is not considered which may result in slower data access and poor performance.

To address these limitations, we design a lock-less messaging protocol and propose two lock-less DLB strategies: NUMA-aware Redirect Push **(NA-RP)** and NUMA-aware Work Stealing **(NA-WS)**. These strategies are the first lock-less DLB techniques for OpenMP tasking, with NUMA-awareness. We first analyze load imbalance in our XGOMPTB implementation and then present our lock-less messaging protocol and the two DLB strategies.

A. Measuring load imbalance

To understand the load imbalance, we collect and study statistics generated by our profiling tools (see: Section V). We specifically look at the time spent on a set of runtime operations, for example, task creation and team barrier, and the number of tasks generated and executed by each thread.

Fig. 3 shows the time spent by each thread in various states for Fibonacci (**Fib**) and Multisort (**Sort**) applications. In the left-hand figures, each stacked horizontal bar summarizes one of the 192 threads, with the colors representing, from left to right, threading (purple), tasking (green), task creation (blue), *taskwait* directive (yellow), barrier (red), and stall (grey).

We focus on tasking (green), task creation (blue), and stall (grey) as threading and barrier are almost indistinguishable in the figure. We consider utilized time as the time spent on tasking (green) and task creation (blue). In the right chart, the horizontal bars represent tasks created (blue) and tasks executed per thread (green). **Fib**'s Timeline Summary shows that threads with lower thread ID have much shorter utilized time than other threads; in the Task Count Summary, they generated and executed fewer tasks, indicating a clear imbalance in both utilization and task count per thread. We conclude that **Fib** is poorly balanced in terms of both utilization and task count per thread; For **Sort**, although the task count appears balanced, mid-range threads have higher utilization, leaving other threads idle and causing performance degradation.

B. Lock-less Messaging Protocol

We first need a lock-less messaging protocol for communicating between threads. We extend prior work [10] with a conditionally random victim selection strategy [11], as follows.

We extend the OpenMP-defined thread data structure with two 64-bit memory cells: a *round* cell and a *request* cell. The *round* number is maintained by the victim to track steal requests that have been "handled" enabling potential thieves to check if the victim is able to be stolen from. It is a monotonically increasing number that is initialized to 1 and incremented by the victim each time a steal request is successfully handled. We consider successful handling of a steal request to be when the victim sees the request and processes whether the request is valid. The *request* cell contains a 40-bit round number and a 24-bit worker ID. The thief sends a request to the victim by writing to the *request* cell the victim's round number combined with the thief's worker ID.

The thief messaging logic is in Alg. 1. In all algorithms, we define p_{local} as the probability that the thief steals within their local NUMA zone. tid_{i} is the thread ID of worker i,

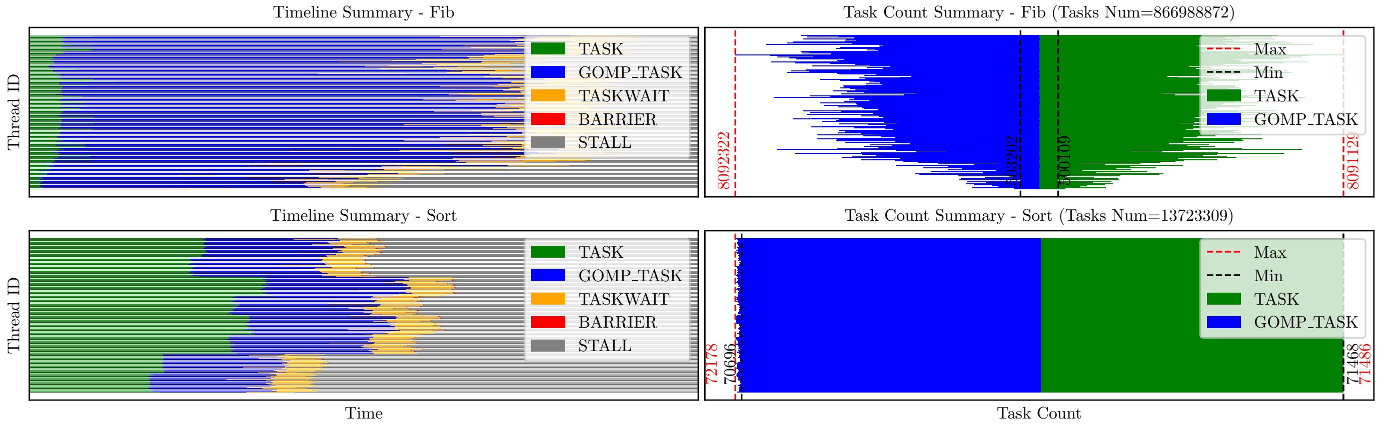

Fig. 3. Load imbalance of **Fib** (above) and **Sort** (below) with XGOMP. Y-axes are Thread ID. Left (Timeline Summary), X-axis is Time, showing the amount of time each thread spends in each state. Right (Task Count), blue bar shows the number of tasks generated, green bar shows the number of tasks executed.

while $ctid_{\text{thief}}$ is that of the current thief. req and $round$ are the current request cell and round number, respectively. The number of tasks to steal per request is N_{steal}. The thief picks a victim randomly, with probability p_{local} of NUMA-local and $1-p_{\text{local}}$ of NUMA-remote. The thief then compares the victim's *round* cell and the round number extracted from the victim's *request* cell. If the round number from *round* cell is greater than the round number from *request* cell, meaning there has not been a new request for that victim, it then writes a request combining the victim's current round number from *round* cell and the thief's worker ID to the the victim's *request* cell, otherwise the request is not sent because there has been a request already.

The thief maintains a timeout counter to deal with long idle times. This counter is incremented each time the thief reaches the same scheduling point while idle and is reset if i) the timeout counter reaches T_{interval}, triggering a retry, or ii) the worker is no longer idle. Timeout can arise if a request is not sent or is overwritten by other thieves, or a victim is idle. When a worker finds a task to execute, it becomes a victim and tries to handle a request if one exists.

1: $tid_{\text{victim}} \leftarrow$ `pickVictimTID`$(p_{local}, tid_{\text{i}})$
2: $req \leftarrow$ `getVictimRequest`(tid_{victim})
3: $round \leftarrow$ `getVictimRound`(tid_{victim})
4: $curr \leftarrow req \;\&\; ((1 \ll 40) - 1)$ ▷`extract round`
5: **if** $curr < round$ **then**
6: $\quad newReq \leftarrow (tid_{\text{i}} << 40) \;\&\; round$
7: $\quad$ `setVictimRequest`$(tid_{\text{victim}}, newReq)$

Algorithm 1: Thief worker send request logic

We show the victim logic in Alg. 2. A victim first checks its own *request*. If the round number of its *request* is equal to its current round number from its *round* cell, this is considered as a valid request. The victim then processes the request using a DLB strategy (NA-RP or NA-WS) and increments the round number so that it is willing to accept new steal requests.

The lock-less inter-core communication overhead is primarily due to thieves accessing remote memory cells. Unlike traditional inter-core communication, which often relies on atomic operations with typical lower-bound per-access latencies of around 100 ns [12], our approach can leverage multi-level, shared caches with lower-bound per-access latencies of just a few nanoseconds. This characteristic rewards spatially adjacent communications and task distribution.

1: $tid_{\text{thief}} \leftarrow req \gg 40$
2: $r_{\text{thief}} \leftarrow req \;\&\; ((1 \ll 40) - 1)$
3: **if** $r_{\text{thief}} == round$ **then**
4: $\quad$ `doLoadBalancing()`
5: $\quad round \leftarrow round + 1$

Algorithm 2: Victim worker handle request logic

C. NUMA-aware Redirect Push (NA-RP)

In our first DLB strategy, NUMA-aware Redirect Push (**NA-RP**), the runtime redistribute tasks dynamically during work-sharing: see Alg. 3. The thief first repeats the logic in Alg. 1 N_{victim} times—in effect, asking N_{victim} victims to redirect tasks to it. Each victim, upon successfully handling a request using Alg. 2, calls `doLoadBalancing()`: see Alg. 3. `doLoadBalancing()` then changes victim state to be ready to redirect N_{steal} new tasks to the thief: see `doRedirectPush()`. If the thief's task queue is full or the victim handles all redirect push requests, the victim increments the round number and is ready to accept new request.

The NA-RP strategy extends XQueue's lock-less enqueuing mechanisms by only altering the target task queue to which new tasks are pushed. It prioritizes helping under-loaded workers to mitigate work imbalance. The additional overhead lies in the exchange of messages across cores.

D. NUMA-aware Work Stealing (NA-WS)

In our second DLB strategy, NUMA-aware Work Stealing (**NA-WS**), the thief steals tasks from specific victims based on their relative NUMA location.

Function `doLoadBalancing`():

1: **if** $tid_{\text{thief}} == -1$ **then**
2: $\quad ctid_{\text{thief}} \leftarrow req \gg 40$
3: $\quad pushed_tasks \leftarrow 0$

Function `doRedirectPush`($newTask$):

4: **if** $pushed_tasks \geq N_{\text{steal}}$ **or** `isTargetQFull`($ctid_{\text{thief}}, tid_{\text{i}}$) **then**
5: $\quad ctid_{\text{thief}} \leftarrow -1$ ▷No thief
6: **else**
7: $\quad$ `pushXQueueTask`($ctid_{\text{thief}}, newTask$)
8: $\quad pushed_tasks \leftarrow pushed_tasks + 1$

Algorithm 3: Redirect push logic

Function doLoadBalancing():

1: $tid_{\text{thief}} \leftarrow req \gg 40$
2: $pushed_tasks \leftarrow 0$
3: **while** (!`isTargetQFull`($tid_{\text{thief}}, tid_{\text{i}}$))**and**(!`isMyQEmpty`()) **do**
4: $\quad task \leftarrow$ `removeXQueueTask`(tid_{i})
5: $\quad$ `pushXQueueTask`($tid_{\text{thief}}, task$)
6: $\quad pushed_tasks \leftarrow pushed_tasks + 1$
7: $\quad$ **if** $pushed_tasks \geq N_{\text{steal}}$ **then**
8: $\quad\quad$ break

Algorithm 4: Work stealing design

We first explore a queue-based strategy. Recall that each XQueue thread possesses $N_{\text{worker}} \times S_{\text{queue}}$ task queues where N_{worker} is the number of workers and S_{queue} is the size of each individual queue. We allocate N_{worker} *request* cells and *round* cells that map to each queue. The routine starts with the thief picking victims with p_{local} of picking NUMA-local workers. It then randomly picks a queue in each of these victims, sending requests using our lock-less communication protocol. Upon receiving requests, a potential victim scans a subset at a time to find one to further process. This strategy guarantees that there is only one producer and consumer for each request cell, thus avoiding requests being overwritten, which could result in retry delays for the thieves. Through experiments, we observed that this method does not mitigate work imbalance and it introduces additional overhead. With millions of steal requests sent, only few tasks are stolen. We observed that 62% of requests are handled by the victims. However, less than 1% of all requests found by victims are valid, and around 0.01% of these valid requests result in successful steals. The runtime is rarely able to steal because there is a mismatch between the number of requests sent and the number of requests handled.

We further improve this strategy by letting a thief steal a batch of tasks from a victim, rather than requesting from a victim's individual task queues: thus reducing communication from queue-to-queue to worker-to-worker. The thief uses the logic in Alg. 1. A victim, upon successfully handling a request, dequeues up to N_{steal} tasks from its queue and en-queues them to the thief's target task queue. This round of stealing completes when 1) no task is found from the victim; 2) the thief's target task queue is full; or 3) N_{steal} tasks have been migrated. We present the pseudo-code in Alg. 4.

Our lock-less work stealing approach inherits both lock-less enqueuing and dequeuing mechanisms from XQueue by migrating already-queued tasks from one worker's queue to another instead of co-locating the newly spawned tasks. Theoretically, this strategy incurs slightly more overhead compared to NA-RP because of the additional dequeuing operations. However, it could result in far fewer tasks stolen than tasks redirected in the NA-RP approach because stealing N_{steal} tasks is the upper bound for each round of stealing, while redirecting N_{steal} is the upper and the lower bound for each round of redirection. Unlike NA-RP that tends to push the tasks away from where they are created to remote workers, our NA-WS approach tends to bring the tasks back to where they are created, thus exploiting benefits of data proximity.

E. DLB Configuration

We implement several configurable parameters to control how the DLB strategies operate. These parameters are the number of victims (N_{victim}) to whom a worker sends requests each time it becomes a thief; the max number of tasks to be stolen/redirected (N_{steal}) for each request; the timeout interval (T_{interval}) between two requests (addressing the situation that an under-loaded worker could remain idle for an extended period of time if requests sent while idle are not answered); and NUMA-local probability (P_{local}) controlling the probability that a thief steals from NUMA-local nodes.

V. Software Profiling Tools

To better analyze the performance characteristics of OpenMP tasking we developed new per-thread performance profiling tools. These tools enable us to track the time spent on tasking runtime operations. We categorize the operations that are most important for performance analysis into several event types. We identify the start and end of each event and insert marker functions `perf_record` with corresponding settings to record the event to profiler memory. We use *rdtscp* (ReaD Time-Stamp Counter and Processor) id value as the timestamp for each event. *rdtscp* is a light-weight processor intrinsic instruction that reads the current value of the processor's timestamp counter. The processor increments the timestamp counter monotonically every clock cycle and resets it to 0 when the processor is reset. Note that while *rdtscp* is not a serializing instruction, it ensures that all prior instructions have executed and all previous loads are globally visible [13]. At the conclusion of the program, the `xomp_perflog_dump` API can be used to save the logs to the file system in a path defined by environment variables.

Although enabling performance logging introduces overheads for fine-grained tasks, e.g., due to increased memory accesses when storing event information and the unavoidable hardware overhead of *rdtscp*, it still captures overall runtime

behaviors, as each logging operation overhead is almost constant. Below, we describe our event classes. Each event class includes a pair of start and end timestamps.

The cycles spent by a thread for different purposes are tracked as follows: By thread on the system: `THREAD`; by a task: `TASK`; creating tasks (crucial because fine-grained tasks can spend a large portion of their lifecycle on task creation): `GOMP_TASK`; waiting: `TASKWAIT`. `BARRIER` denotes the number of cycles spent on a barrier. If a thread is unoccupied, i.e., no task is scheduled in the current thread and it is busy checking its task queue, we record the cycles with `STALL`.

We also instrument per-thread statistical counters. These counters incur little overhead as they are thread-local and can use lower-level caches. We implement a set of general performance counters as well as specialized counters for our DLB strategies. To track the locality of a task, we assign the thread ID to the task upon creation.

To manage data locality, XGOMPTB tracks the number of tasks executed: i) by the thread that created it (`NTASKS_SELF`), ii) by the NUMA node that created it (`NTASKS_LOCAL`) iii) by another node (`NTASKS_REMOTE`). XGOMPTB also tracks the number of tasks that are i) statically pushed (`NTASKS_STATIC_PUSH` and ii) are not pushed due to the target queue being full and are executed immediately (`NTASKS_IMM_EXEC`).

In addition, for the two DLB strategies we track the number of steal requests (`NREQ_SENT`), handled requests (`NREQ_HANDLED`), and stolen tasks (`NTASKS_STOLEN`). Handled requests are further categorized into requests that i) result in a task being stolen (NREQ_HAS_STEAL), or ii) fail due to an attempt to steal tasks from an empty queue (NREQ_SRC_EMPTY) or to a full queue (NREQ_TARGET_FULL). Stolen tasks are further categorized into tasks stolen by i) NUMA-local thieves or ii) NUMA-remote thieves.

VI. Evaluation

We evaluate our XGOMP and XGOMPTB implementations and NA-RP and NA-WS strategies using nine applications from the Barcelona OpenMP Task Suite (BOTS) [8]: Fibonacci (**Fib**), **NQueens**, Fast Fourier Transform (**FFT**), Floor Plan (**FP**), **Health**, Unbalance Tree Search (**UTS**), Strassen Matrix Multiplication (**STRAS**), and Protein Alignment (**Align**). We first compare XGOMP with the original GNU OpenMP (GOMP). We also include two LLVM variants—LLVM OpenMP (LOMP) and LLVM OpenMP using XQueue (XLOMP)—in our results for comparison. We conduct all experiments on an Intel Skylake-192 machine, with 192 cores (384 hardware threads) and eight NUMA zones. We compile the benchmarks with GNU GCC version 12.2.1 and LLVM version 14.0.0, both with *-fopenmp* flag and O3 optimization level. We use up to 192 threads and bind OpenMP threads to cores with *close* thread affinity. We use the following input arguments for our applications. **Fib**: 42, **NQueens**: 15, **FFT**: 536M, **FP**: 20, **Health**: large, **UTS**: small, **STRAS**: 8192 (Y=16), **Sort**: 1B, and **Align**: 1000. For our DLB experiments (Section VI-B), in which we run large parameter sweeps over many configuration parameters, we scale down **Fib**: 40, **FFT**: 268M and **Sort**: 268M, **FP**: 15, **UTS**: tiny, and **STRAS**: 2048 (Y=16) to reduce experiment times. We have validated experimentally that the larger and smaller experiments yield similar results.

A. GNU OpenMP with XQueue and Distributed Tree Barrier

Fig. 4 shows the average application execution time of the nine benchmarks with different OpenMP implementations. Each bar shows the average execution time of five 192-thread runs. We use our profiling tools to measure task size (in *rdtscp* cycles) and order applications based on their task size (from small to large). We see across all benchmarks, our XQueue-based implementations (XGOMP, XGOMPTB, and XLOMP) as well as LOMP are several orders of magnitude faster than GOMP.

To understand these results, we measure the time spent on various runtime profiling events (see Section V). We observe that, applications for which LOMP and XLOMP outperform GOMP and XGOMP—**Fib**, **NQueens**, **FP**, **Health**, and **UTS**—spend much of thread time on task creation (mostly task allocation). While applications that XGOMP and XGOMPTB outperform XLOMP—**FFT**, **STRAS**, **Sort**, and **Align**—spend most of their time on task execution. These results are due to the way that memory is allocated. All OpenMP implementations use *malloc* to allocate memory for tasks. *malloc* is thread safe and thus uses synchronization. The default allocation approach used by Linux prioritizes allocating memory in the NUMA node closest to the CPU executing the task. GOMP invokes *malloc* each time it creates a new task, whereas LOMP instead allocates memory using a fast, multi-level allocator, which pre-allocates chunks of buffer space and manages this space itself. When a thread creates a new task, LOMP/XLOMP will ii) use a local buffer if available; ii) synchronously but locality-agnostically acquire a buffer from another thread if available; or iii) call *malloc*. For fine-grained tasks, large portions of time are spent on task allocation/de-allocation. For GOMP, threads contend for *malloc* operations, which leads to sequential *malloc*s. LOMP can parallelize memory allocation, because method i) is frequently used. For large tasks, LOMP/XLOMP performs a similar number of *malloc* requests to GOMP as the multi-level allocator is less beneficial. That is, methods i) and ii) are less likely to be successful (as longer tasks consume the buffer for longer periods). Further, LOMP "steals" buffer space using method ii), which results in more tasks executed remotely. LOMP has slightly higher task execution overhead because it handles a richer set of cases.

FFT task sizes range between 10^2–10^6 cycles, with the highest proportion around 10^3–10^4 cycles; **STRAS** task sizes range between 10^3–10^7 cycles with most around 10^4 cycles. **Sort** is similar to **STRAS**, with most sizes around 10^5 cycles. **Align** task sizes are between 10^5–10^7 cycles, with the highest proportion around 10^6 cycles. These task sizes allow XGOMP and XGOMPTB to outperform LOMP and XLOMP as the benefit of the multi-level allocator is reduced. **Health** has

average task size greater than **FFT**; however, it has many more tasks concentrated at lower sizes(10^3–10^4 cycles) which benefit from the multi-level allocator.

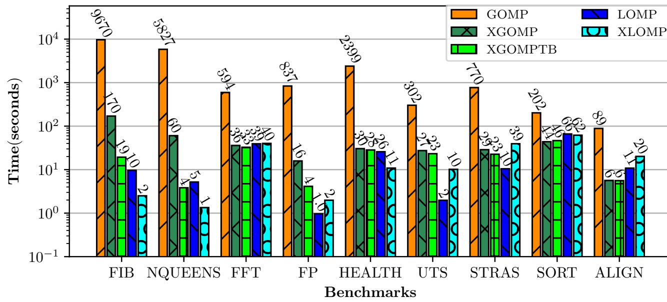

Fig. 4. Absolute execution time of BOTS benchmarks (lower is better).

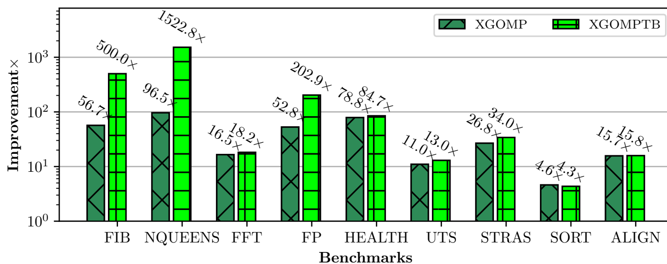

Fig. 5. XGOMP/XGOMPTB performance improvement over GOMP, 192 threads (higher is better).

In Fig. 5 we show the performance improvement of XGOMP and XGOMPTB over GOMP. We see that our methods enable performance improvements of up to 96.5× for XGOMP and 1522.8× for XGOMPTB when compared to GOMP. All benchmark applications benefit from XQueue. Applications with smaller task sizes benefit more from the distributed tree barrier (e.g. Fib, NQueens, FP); while those with larger tasks benefit least from the barrier.

Fig. 6 shows the performance of GOMP, XGOMP and XGOMPTB with increasing number of threads for each application. XGOMP and XGOMPTB perform significantly better than GOMP for most applications and thread counts. For **Align**, which has the largest average task size, we see comparable performance at lower thread counts due to extremely low lock contention. For all applications, performance improvement grows as we increase from 24 threads (one socket) to 192 threads (eight sockets). However, application performance does not scale linearly with threads due to work inflation [14], that is, due to increased costs for operations performed in a parallel vs. a single-processor run. Work inflation can be due to many factors, such as work migration, shared use of an LLC, and the need to access data on remote sockets; it can be reduced by placing computations and data on the same socket, removing the need for remote memory access [15].

B. XGOMPTB with Dynamic Load Balancing

We study our two lock-less DLBs with the same BOTS benchmarks using the Skylake-192 machine. We conduct experiments with 192 threads on eight NUMA zones, controlling runtime behavior with the configuration parameters described in Section IV-E. For each application, we conduct a parameter sweep to identify the optimal settings for each value of N_{victim}, N_{steal}, T_{interval} and P_{local}. We run each DLB configuration ten times, and compare its average execution time with XGOMPTB.

TABLE I
OPTIMAL DLB SETTINGS FOR REDIRECT PUSH AND WORK STEALING

Benchmark	**Fib**		**NQueens**		**FFT**		**FP**		**Health**		**UTS**		**STRAS**		**Sort**		**Align**	
DLB strat.	RP	WS	RP	WS	RP	WS	RP	WS	RP	WS	RP	WS	RP	WS	RP	WS	RP	WS
N_{victims}	1	1	24	16	24	24	24	16	24	8	8	8	24	8	24	24	8	16
N_{steals}	16	1	16	1	1	32	32	32	32	32	1	32	32	32	32	32	8	8
T_{interval}	10^5	10^4	10^5	10^4	10^3	10^4	10^5	10^5	10^3	10^3	10^4	10^5	10^4	10^3	10^3	10^3	10^4	10^4
P_{local}	1	1	1	1	1	1	1	1	1	0.5	1	1	1	0.03	1	0.03	0.03	1

Fig. 7 compares the best average performance of each DLB strategy (NA-RP and NA-WS) with XGOMPTB which uses static load balancing (SLB). The settings used to obtain the best performance for NA-RP and NA-WS are presented in Table I. All benchmarks except **Fib** show performance improvement. We re-run the experiments, collecting statistics for DLB in Table II and SLB in Table III. To understand the effect of locality, we present the number of tasks executed along with their origin: tasks created on the same core (self), tasks created by a worker within the same NUMA zone (local), and tasks created by a worker in a different NUMA zone (remote). Tasks that run on the same core they were created on will generally have more first-level cache hits than those that do not. Tasks that run in the same NUMA zone where they were created will have fewer first-level cache hits, but will still have more shared cache hits and faster memory access than those that run in a remote NUMA zone. Tasks that are executed immediately, will do so on the same core (self) and thus benefit from the low-level cache. We show the number of tasks pushed with SLB and tasks executed immediately due to a failed push. The number of requests sent, handled, and that result in tasks stolen demonstrate the efficacy of our lock-less messaging protocol.

Finally, we look at the locality of stolen tasks—more locally stolen tasks means the system can benefit from faster memory access compared to remote memory access. Due to profiling overhead and randomness, the results presented are slightly different from those in Fig. 7.

1) NUMA-aware Redirect Push (NA-RP): We see in Fig. 7 that **Fib** performance degrades with NA-RP. This is because NA-RP pushes more tasks away, incurring a cost of more than 100 ns for each task that could otherwise be self-executed within nanoseconds. It has the largest T_{interval}, the smallest N_{victim}, and a moderate N_{steal}. We see in Table II that NA-RP results in 8.2M stolen tasks and the fewest self-executed tasks across all strategies. **Fib**'s tasks are small, 10–80 cycles, and its DAG has a long critical path that can barely benefit from parallelism. Many tasks are redirected, removing the potential locality benefit provided by the first-level cache. Most steal requests are successfully handled, which demonstrates the

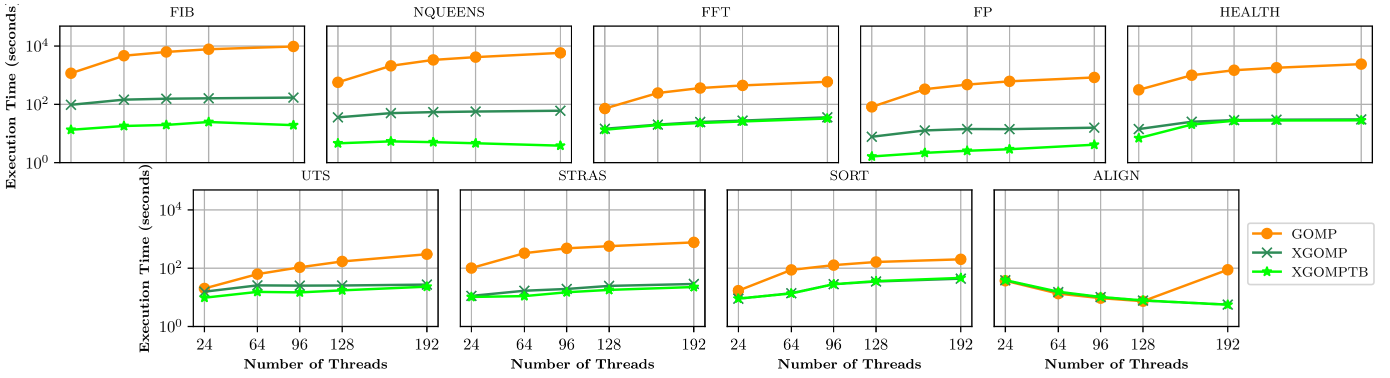

Fig. 6. Scaling performance of different methods comparing execution time as the number of threads is increased for each BOTS application (lower is better)

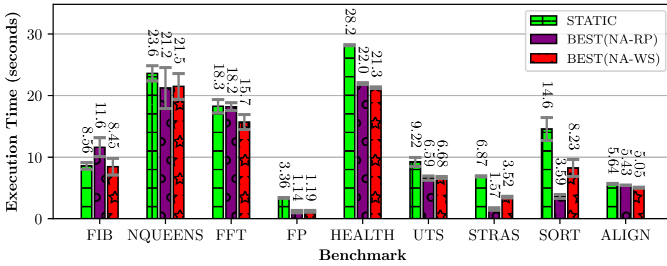

Fig. 7. XGOMP DLB performance comparisons on BOTS with best performance settings, 192 threads.

efficacy of our lock-less messaging protocol for fine-grained tasks.

For **NQueens**, NA-RP is the best performing strategy. However, it has the largest error bar in Fig. 7 indicating perhaps sensitivity to when steals occur. We see in Table II that NA-RP yields the worst average performance. We suspect that the profiling overhead may affect the dispatching pattern with SLB, which potentially results in more tasks being pushed to a remote node. **NQueens** works best when it is completely NUMA-local and the T_{interval} is also the largest, while N_{victim} and N_{steal} are large, indicating that it benefits from large batches of steal requests after a long period of time. **NQueens** with NA-RP yields the worst performance with profiling on because those runs have the least self-executed tasks, thus losing some of the benefit provided by the first-level cache. NA-RP with profiling also distributes more tasks to NUMA-remote nodes, which also increases the latency.

FFT performs slightly better with NA-RP than with SLB. It works best with the largest N_{victim}, and smallest N_{steal} and T_{interval}. It also uses fully NUMA-local. Statistics show that NA-RP has the most local tasks, 4.7× more same-core tasks than SLB, and the fewest remote tasks.

FP performs 2.6× better with NA-RP than with SLB. It works best with the largest N_{victim}, N_{steal}, T_{interval}, and P_{local}. **FP** task sizes are varied, with the highest proportion around 10^2–10^3 cycles and many ranging between 10^3–10^6 cycles. Scheduling tasks with large and varied task sizes could result in load imbalance. NA-RP mitigates imbalance by redirecting 909K tasks. 1.3M more tasks are brought to the same core and NUMA-local than with SLB. Most of its steal requests are successful.

Health performs 28.2% better with NA-RP than with SLB. It works best with the largest N_{victim}, N_{steal}, and P_{local}, and the smallest T_{interval}. NA-RP brings 33.1M more tasks to the same core and 33.3M more tasks to be immediately executed which benefits from the low-level cache. Most of its steal requests are successful.

UTS performs 40.6% better with NA-RP than with SLB. It works best with small N_{victim}, smallest N_{steal}, large T_{interval}, and largest P_{local}. NA-RP brings more tasks to the same core and NUMA zone. Most steal requests are successful.

Both **STRAS** and **Sort** perform around 4× better with NA-RP than that with SLB. They perform best with the largest N_{victim} and N_{steal}, fully NUMA-local and small T_{interval}. With SLB, most tasks are executed in a remote NUMA zone. NA-RP successfully co-locates around 711K tasks for **STRAS**, and around 2.7M tasks for **Sort** to NUMA-local nodes: 85% and 90% of the total, respectively. **STRAS** allocates large arrays in each task, bringing more tasks to the same core and NUMA zone can utilize faster memory access. Both **Sort** and **STRAS** allocate the memory space in an interleaved NUMA policy for the array before the OpenMP parallel region. Their large task sizes result in early runtime under-utilization for most workers, so NA-RP takes place before SLB can do much work. Therefore, many workers have requests to redirect tasks to NUMA-local nodes, leveraging multi-level cache and data proximity to reduce the latency and significantly increase performance.

Align does not have any steals with NA-RP and has negligible difference in performance with different parameters. **Align** uses the *single* OpenMP construct where only one thread is responsible for creating all tasks in a loop. These tasks are distributed by the default SLB and only the thread that runs the *single* construct is able to redirect tasks with NA-RP. **Align**'s task sizes follow a normal distribution, with the majority falling within the range of 10^6–10^7 cycles. Most of **Align**'s cycles are spent aligning the same set of limited-size protein sequences, which fit comfortably in the cache and therefore do not generate a significant communication between

TABLE II
BOTS RUNTIME STATISTICS WITH NA-RP AND NA-WS DLB STRATEGIES. REPORTING AVERAGE FROM 10 EXPERIMENTS.

Benchmark	**Fib**		**NQueens**		**FFT**		**FP**		**Health**		**UTS**		**STRAS**		**Sort**		**Align**	
DLB strat.	RP	WS	RP	WS	RP	WS	RP	WS	RP	WS	RP	WS	RP	WS	RP	WS	RP	WS
Time (secs)	9.9	8.0	26.5	23.2	17.0	15.5	1.3	1.3	21.2	20.9	6.4	6.6	1.6	3.7	3.8	9.5	5.6	5.2
Self tasks	310.8 M	318.2 M	2.4 B	2.5 B	946.7 K	1.7 M	17.1 M	16.5 M	33.8 M	33.5 M	6.2 M	347.1 K	277.2 K	416.6 K	146.3 K	63.5 K	2.9 K	2.6 K
Local tasks	12.2 M	1.6 M	30.1 M	6.5 M	11.6 M	2.0 M	1.2 M	302.3 K	41.2 M	10.8 M	20.4 M	3.6 M	553.8 K	65.7 K	2.7 M	376.0 K	59.8 K	59.8 K
Remote tasks	8.2 M	11.4 M	74.1 M	47.4 M	5.7 M	14.6 M	837.6 K	2.3 M	51.3 M	82.1 M	3.9 M	26.5 M	129.7 K	478.5 K	208.9 K	2.6 M	436.8 K	437.0 K
Static push	12.3 M	13.0 M	90.9 M	54.2 M	15.1 M	16.7 M	1.1 M	2.6 M	64.1 M	93.5 M	20.6 M	30.3 M	233.0 K	548.8 K	484.5 K	3.0 M	496.6 K	496.9 K
Imm. exec	310.6 M	318.1 M	2.4 B	2.5 B	468.4 K	1.6 M	17.0 M	16.5 M	33.3 M	32.9 M	5.4 M	123.2 K	272.7 K	412.0 K	134.0 K	41.9 K	2.9 K	2.6 K
Req. sent	590.1 K	1.6 M	1.3 M	2.4 M	4.2 M	16.3 M	31.3 K	884.1 K	976.8 K	13.2 M	5.5 M	19.1 M	18.9 K	233.5 K	86.8 K	5.2 M	1.8 K	266.4 K
Req. handled	514.9 K	1.3 M	872.9 K	2.3 M	2.7 M	10.9 M	28.5 K	798.0 K	905.9 K	12.0 M	4.4 M	12.5 M	14.3 K	171.6 K	76.9 K	2.0 M	192	265.8 K
Req. w/ steal	514.7 K	576.4 K	872.7 K	1.3 M	2.7 M	2.4 M	28.3 K	395.9 K	904.7 K	7.9 M	4.4 M	4.8 M	14.1 K	114.7 K	76.7 K	683.2 K	0	156.4 K
Total steal	8.2 M	576.4 K	14.0 M	1.3 M	2.7 M	19.5 M	909.1 K	7.0 M	29.0 M	176.9 M	4.4 M	48.9 M	455.1 K	2.8 M	2.5 M	9.0 M	0	861.0 K
Local steal	8.2 M	576.4 K	14.0 M	1.3 M	2.7 M	19.5 M	909.1 K	7.0 M	29.0 M	98.9 M	4.4 M	48.9 M	455.1 K	403.8 K	2.5 M	1.7 M	0	861.0 K

TABLE III
BOTS RUNTIME STATISTICS USING SLB.

Benchmark	**Fib**	**NQueens**	**FFT**	**FP**	**Health**	**UTS**	**STRAS**	**Sort**	**Align**
Time (secs)	8.1	23.8	19.1	3.4	28.2	9.0	6.7	11.1	5.7
Self tasks	318.1 M	2.5 B	203.0 K	16.7 M	658.2 K	953.4 K	5.1 K	16.1 K	3.0 K
Local tasks	1.6 M	6.1 M	2.2 M	287.9 K	15.1 M	3.5 M	115.1 K	368.3 K	59.8 K
Remote tasks	11.5 M	44.2 M	15.9 M	2.1 M	110.6 M	25.9 M	840.6 K	2.7 M	436.7 K
Static push	13.2 M	50.6 M	18.2 M	2.4 M	126.4 M	29.6 M	960.8 K	3.1 M	496.5 K
Imm. exec	318.0 M	2.5 B	108.2 K	16.7 M	0	799.2 K	0	0	3.0 K

cores and main memory.

2) NUMA-aware Work Stealing (NA-WS): All applications exhibit performance improvement with NA-WS, albeit with different configurations, as shown in Table I.

Fib achieves negligible (max 1.3%) performance improvement. NA-WS performs best with the smallest N_{victim} and N_{steal}, moderate T_{interval}, and uses only NUMA-local nodes. **Fib**'s serial task graph means that there are few opportunities for parallelism and thus task stealing. More self-executed tasks means more opportunities to utilize first-level cache. NA-WS can take advantage of such opportunities because it can bring pushed tasks back to where they are created, thus increasing self-executed tasks. However, over 95% of tasks are already immediately executed on the same core when using SLB and therefore there is little room for improvement.

NQueens uses similar parameter settings as **Fib**, with larger N_{victim} leading to the best performance improvement (9.8%). The statistics indicate that NA-WS successfully steals 1.3M tasks. As a result, 3.7M tasks are subsequently created and executed on the same core compared to SLB.

FFT performs best (16.5%) when T_{interval} and P_{local} are the same as **NQueens**, but with the largest N_{victim} and largest N_{steal}. From the statistics, NA-WS brings many more tasks back to its creators, with 1.7M self-executed tasks compared to 203K with SLB. It also effectively moves 19.5M tasks to mitigate load imbalance.

FP achieves 2.8× performance improvement with moderate N_{victim}, the largest N_{steal} and T_{interval}, and fully NUMA-local. **FP** is a heavily imbalanced program with SLB and contains many tasks between 10^3–10^6 cycles. The performance gain is primarily due to NA-WS mitigating load imbalances while keeping many tasks local.

Health achieves the best performance improvement (32.5%) with small N_{victim}, the largest N_{steal}, smallest T_{interval}, and a P_{local} of 0.5. From the statistics, NA-WS moves many tasks to its creators, with 33.5M self-executed tasks compared to only 658.2K with SLB.

UTS performs best (36.4%) with small N_{victim}, largest N_{steal} and T_{interval}, and fully NUMA-local. Though NA-WS does not improve task locality, it actively moves 48.9M tasks to mitigate load imbalance.

STRAS achieves the best performance improvement (95%) with small N_{victim}, largest N_{steal}, and smallest T_{interval} and P_{local}. Its performance gain is primarily due to moving more tasks to the same core and NUMA zone.

Sort improves the most (76.9%) with largest N_{victim} and N_{steal} and smallest T_{interval} and P_{local}. It significantly increases the number of NUMA-local and self-executed tasks to exploit data locality.

Align performs best (9.6%) with moderate N_{victim}, small N_{steals}, large T_{interval} and fully NUMA-local. Although it works well with SLB due to the application characteristics mentioned in Section VI-B2, NA-WS mitigates imbalance by moving 861K tasks.

In general, the lock-less NA-WS strategy can effectively mitigate load imbalance. With the proper settings, it can promote task locality by increasing the number of local and self-executed tasks. It tends to steal tasks back to where they are created, which can benefit performance for both extremely fine-grained and coarser-grained tasks. Although the performance for applications with larger task sizes is less than the NA-RP strategy, NA-WS achieves at least minimal performance improvement across all types of applications we tested. Therefore, we conclude that NA-WS is a well-rounded and less sensitive work stealing approach that is most likely to achieve better performance.

VII. APPLICATION IN BLOCKCHAIN CONSENSUS: PROOF-OF-SPACE (POSP)

We investigate how our methods can improve the performance of a blockchain application using the Proof-of-Space (PoSp) consensus algorithm [16], [17]. PoSp is an alternative to Proof-of-Work (PoW) that is less computationally expensive and reduces energy consumption. PoSp transforms a compute-intensive PoW problem into a data-intensive/storage problem where cryptographic puzzles are recorded in a persistent storage medium, later organized in order to be efficiently retrieved.

PoSp uses the BLAKE3 cryptographic hashing algorithm [18] (over SHA-256) due to its excellent performance on a wide range of hardware. We extend a PoSp implementation that is written in C and using OpenMP [19] as the primary mechanism to extract parallelism from the generation and storage of the cryptographic puzzles.

The PoSp implementation uses task-based parallelism in OpenMP to dynamically fill buckets with cryptographic puzzles until all buckets are full. Task-based parallelism provides a flexible way to express irregular or dynamic parallel workloads that traditional loop-based parallelism may not handle efficiently. This is particularly useful when the workload is unbalanced or when the order in which tasks become ready to run is not strictly sequential, which is typically the case for PoSp. PoSp can be configured with number of threads, memory size, and batch size. The batch size determines the number of cryptographic puzzles to be generated in a single task. Small batch sizes can be inefficient if the runtime does not efficiently support small tasks, while large batches can be sub-optimal due to load imbalance.

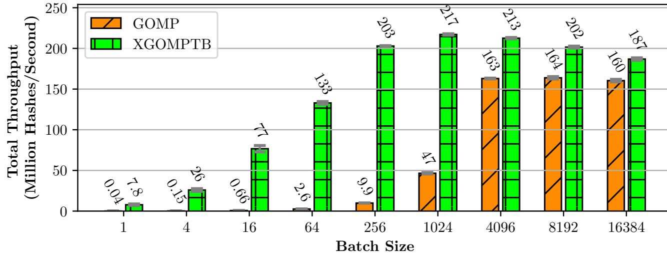

Fig. 8. PoSp throughput comparing GOMP and XGOMPTB as the batch size is increased on 192-cores (higher is better).

The state-of-the-art production PoSp blockchains (Chia [20]) use a minimum of 2^K (where $K = 32$) cryptographic puzzles to be stored in a single file [17], [21]. The PoSp evaluated is made up of 2^K cryptographic puzzles, where each puzzle is composed of a 28 Bytes BLAKE3 hash and a 4 Bytes nonce value. We conducted a parameter sweep (see Fig. 8) comparing XGOMPTB and GOMP on our 192-core system with increasing batch size. Using a batch size of 1, XGOMPTB achieves 7.8 megahashes per second (MH/s), compared to 0.04 MH/s for native GOMP, a 195× improvement. A batch size of 1 yields relatively small tasks, stressing the runtime's ability to process a high throughput of tasks per second. XGOMPTB achieves 7.8 M tasks per second while GOMP only achieves 40 K tasks per second. The significant throughput reduction for GOMP comes from shared locks in the OpenMP runtime. The best performance is achieved for a batch size of 1024 for XGOMPTB with 217 MH/s, compared to 164 MH/s for GOMP with an 8192 batch size, a 32% speedup. Batch sizes that are too large can yield lower performance stemming from load imbalance that leads to underutilized resources.

VIII. Performance Tuning

Our experiments reported in Section VI-B show that different applications react differently to the choice of DLB and are sensitive to parameter choices. For example, all applications are relatively sensitive to N_{victim}, N_{steal}, and P_{local}.

To further study relationships between the performance and parameters, we categorize the applications into five sizes based on per-task *rdtscp* cycles S_{task}. We then define steal size S_{steal} in Equation 1, where N_{victim} is the number of victims per request, N_{steal} is the number of steals per victim, and T_{interval} is a timeout counter for the worker before it retries next round of requests. Since applications lack sensitivity to T_{interval}, we reduce its influence by applying a log function.

$$S_{\text{steal}} = \frac{N_{\text{steal}} \times N_{\text{victim}}}{\log_{10} T_{\text{interval}}} \tag{1}$$

We show in Figs. 9 and 10 the performance improvement of both DLBs with 3D triangular surface plots. In each figure, the Z-axis is the performance improvement over XGOMPTB.

Fig. 9 shows that when using NA-RP, applications with task size $<10^2$ cycles suffer performance degradation. For 10^2–10^4 cycles, we see little performance improvement. For $>10^4$ cycles, performance generally benefits from larger steal sizes. For the largest tasks, NA-RP works exceptionally well, with $4\times$ performance improvement.

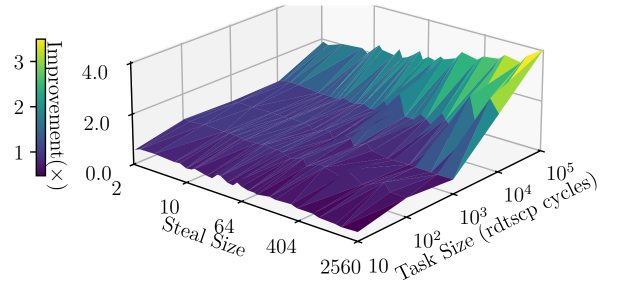

Fig. 9. NA-RP (redirect-push) approach performance improvement (×) over XGOMPTB as a function of task size and steal size (both log scale).

Fig. 10 shows that NA-WS is less sensitive to configuration and leads to performance degradation only for applications with small task size ($<10^3$) and large steal size. As task size increases, application performance generally increases, with the best improvement achieved for the largest task and steal sizes. Generally, applications with larger task sizes benefit more from larger steal sizes; applications with small task size should use small steal sizes.

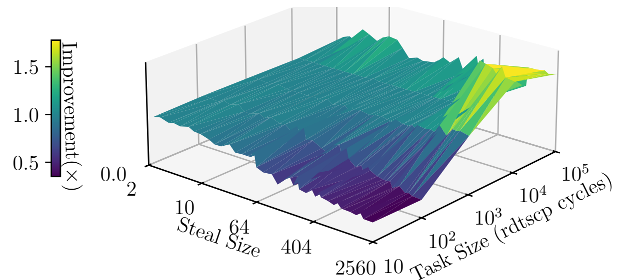

Fig. 10. A-WS (work-stealing) approach performance improvement (×) over XGOMPTB as a function of task size and steal size (both log scale).

TABLE IV
OPTIMAL DLB SETTINGS FOR DIFFERENT TASK SIZES.

S_{task} (rdtscp)	10^1–10^2	10^2	10^3	10^3–10^4	$>10^4$
Best DLB	WS	WS	WS	WS	RP
Best P_{local}	100%	100%	100%	3–50%	3–12%
Best S_{steal}	10^0–10^1	10^1–10^2	10^2–$10^{2.5}$	$10^{2.5}$–10^3	$>10^3$

Locality choice is also an important contributor to performance. For NA-RP, fully local redirection yields the best performance across all benchmarks (see Table I) because it optimizes memory access latency by pushing tasks to adjacent workers. Because NA-RP pushes more tasks away, it is the best choice to send tasks to the local node which can also utilize multi-level cache. For NA-WS, extremely fine-grained tasks perform best with fully NUMA-local steal, to optimize the overall memory access latency. For larger tasks, smaller P_{local} performs better because work stealing can return pushed tasks to where they are created. Therefore, tasks may utilize high-level cache which optimizes overall latency.

In summary, different combinations of strategies and parameters fit different application characteristics. Applications with extremely fine-grained tasks benefit from smaller steal sizes on the NA-WS approach; full NUMA-local probability should be used. Applications with larger tasks benefit from larger steal sizes. NA-RP with maximum steal size and fully NUMA-local is particularly suitable for tasks of $>10^4$ cycles. We summarize in Table IV guidelines for selecting parameters for the best performance and show in Fig. 11 results obtained for the five applications when using these guidelines. We use as input arguments, **Fib**: 42, **NQueens**: 16, **FFT**: 536M, **FP**: 20, **Health**: xlarge, **UTS**: small, **STRAS**: 4096 (Y=16), **Sort**: 1B, and **Align**: 2000.

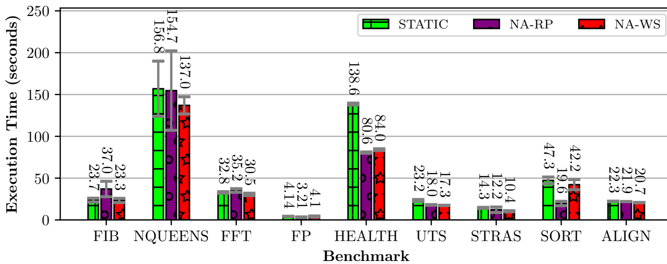

Fig. 11. XGOMP, NA-RP, and NA-WS performance comparison on BOTS when following the guidelines of Table IV.

IX. RELATED WORK

Parallel runtime systems such as OpenMP [1], Charm++ [22], and Swift/T [23] use concurrent queues to share data between threads or processes. Charm++ goes further and supports lock-free queues to demonstrate performance improvements [24]. Regarding locks, researchers have investigated contention management in thread-safe MPI libraries [25] and the use of abort locking [26] to improve performance. We aim to achieve better performance with finer granularity and task decomposition than these solutions.

Some classical with-locks task-based runtimes (e.g., OmpSs [27], PaRSEC [28], StarPU [29]) have made efforts to improve data locality. For example, OmpSs and XKaapi [30] rely on work-stealing for load balancing. XKaapi also provides a lower bound on the number of data accesses required by the scheduler [31]. Legion [32] allows users to specify locality explicitly using data regions, and provides a data mapping strategy to ensure that data are only moved when needed. Some of StarPU's scheduling strategies focus on data reuse and task stealing to increase the performance of linear algebra applications [33], [34]. These solutions are orthogonal to our research: We focus on removing the synchronization cost of barriers and thus cannot use similar techniques that require regular synchronization and updating of current system state.

Dynamic Load balancing: Several papers have proposed load balancing mechanisms [35], [36]. Quintin et al. proposed hierarchical work stealing to exploit data locality to achieve speedup over classical work stealing algorithms [37]. Shiina et al. addressed the issue of data locality by making scheduling deterministic [38]. In order to balance load efficiently, these all relied on synchronisation mechanisms. In our work, we explore lock-less techniques to achieve comparable dynamic load balancing mechanisms.

Lock-less runtime systems: XQueue [4] demonstrated the benefits of a lock-less, task-oriented runtime in OpenMP. Recent work on XQueue [10] introduced work-stealing, but focused on simply redirecting a newly spawned task to another thread. Results indicated a need to develop more sophisticated strategies and also consider NUMA-awareness, as we explore here. Among other things, in this paper, we integrate XQueue into GNU-OpenMP, introduce a tree barrier, and propose sophisticated NUMA-aware work-stealing strategies, which dynamically migrate tasks from a victim to a thief thread.

X. DISCUSSION AND FUTURE WORK

GNU OpenMP, although part of the mainstream compiler infrastructure GCC, is unable to harness modern high-performance CPUs with ever-increasing number of cores, due to its conservative design of using excessive locks and atomic operations. Our work transforms GNU OpenMP into a high-performance parallel computing library by integrating the novel lock-less concurrent data structure XQueue and an efficient distributed tree barrier to achieve up to 1522.8× performance improvement compared to the original GNU OpenMP. We further improve performance via use of dynamic load balancing strategies, demonstrating that with optimal settings, lock-less work stealing can achieve 4× more performance improvement compared to use of a static load balancer.

Our experiments show that good parameter choices are dependent on application characteristics. We provide guidance to help practitioners select parameter values based on their applications. In future work, we will decompose application characteristics to automate the selection of good settings.

REFERENCES

[1] "Welcome to the documentation of OpenMP in LLVM! — LLVM/OpenMP 20.0.0git documentation." [Online]. Available: https://openmp.llvm.org/

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," in *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 207–216. [Online]. Available: https://doi.org/10.1145/209936.209958

[3] "Berkeley UPC - Unified Parallel C." [Online]. Available: https://upc.lbl.gov/

[4] P. Nookala, P. Dinda, K. C. Hale, K. Chard, and I. Raicu, "Enabling extremely fine-grained parallelism via scalable concurrent queues on modern many-core architectures," in *29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. Houston, TX, USA: IEEE, Nov. 2021, p. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/9614292/

[5] A. Navarro-Torres, J. Alastruey-Benedé, P. Ibáñez-Marín, and M. Carpen-Amarie, "Synchronization strategies on many-core SMT systems," in *IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct. 2021, pp. 54–63, iSSN: 2643-3001. [Online]. Available: https://ieeexplore.ieee.org/document/9651585/?arnumber=9651585

[6] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *24th ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, Nov. 2013, p. 33–48. [Online]. Available: https://dl.acm.org/doi/10.1145/2517349.2522714

[7] A. Morrison, "Scaling synchronization in multicore programs: Advanced synchronization methods can boost the performance of multicore software." *Queue*, vol. 14, no. 4, pp. 56–79, Aug. 2016. [Online]. Available: https://dl.acm.org/doi/10.1145/2984629.2991130

[8] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP," in *International Conference on Parallel Processing*, ser. ICPP '09. USA: IEEE Computer Society, Sep. 2009, p. 124–131. [Online]. Available: https://doi.org/10.1109/ICPP.2009.64

[9] "ISO/IEC/IEEE 9945," https://www.iso.org/standard/50516.html.

[10] P. Nookala, K. Chard, and I. Raicu, "X-OpenMP — eXtreme fine-grained tasking using lock-less work stealing," *Future Generation Computer Systems*, vol. 159, pp. 444–458, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X24002541

[11] C. J. Williams and J. Elliott, "Libfork: Portable continuation-stealing with stackless coroutines," *arXiv:2402.18480*, no. arXiv:2402.18480, Feb. 2024. [Online]. Available: http://arxiv.org/abs/2402.18480

[12] J. Dean, "Designs, lessons and advice from building large distributed systems," 2009, https://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf.

[13] "RDTSCP — Read Time-Stamp Counter and Processor ID." [Online]. Available: https://www.felixcloutier.com/x86/rdtscp

[14] S. L. Olivier, B. R. De Supinski, M. Schulz, and J. F. Prins, "Characterizing and mitigating work time inflation in task parallel programs," *Scientific Programming*, vol. 21, no. 3-4, pp. 123–136, 2013.

[15] J. Deters, J. Wu, Y. Xu, and I.-T. A. Lee, "A NUMA-aware provably-efficient task-parallel platform based on the work-first principle," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2018, p. 59–70, arXiv:1806.11128 [cs]. [Online]. Available: http://arxiv.org/abs/1806.11128

[16] S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak, "Proofs of space," in *Advances in Cryptology - CRYPTO 2015*. Springer, 2015, pp. 585–605.

[17] B. Cohen and K. Pietrzak, "The chia network blockchain," *White Paper, Chia. net*, vol. 9, 2019.

[18] J. O'Connor *et al.*, "Blake3: One function to rule them all," https://github.com/BLAKE3-team/BLAKE3-specs, 2019, accessed: 2025-01-21.

[19] O. A. R. Board, "OpenMP application programming interface version 5.0," https://www.openmp.org/specifications/, November 2018, accessed: 2025-01-21.

[20] C. Network, "Chia green paper," https://docs.chia.net/files/ChiaGreenPaper_20241008.pdf, October 2024, accessed: 2025-01-21.

[21] "K-sizes in chia documentation," https://docs.chia.net/k-sizes/, accessed: 2025-01-21.

[22] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on C++," in *8th Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, 1993, pp. 91–108.

[23] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/T: Large-scale application composition via distributed-memory dataflow processing," in *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 2013, pp. 95–102.

[24] C. Mei, G. Zheng, F. Gioachin, and L. V. Kalé, "Optimizing a parallel runtime system for multicore clusters: A case study," in *TeraGrid Conference*, ser. TG '10. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1838574.1838586

[25] A. Amer, H. Lu, P. Balaji, M. Chabbi, Y. Wei, J. Hammond, and S. Matsuoka, "Lock contention management in multithreaded MPI," *ACM Transactions on Parallelel Computing*, vol. 5, no. 3, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3275443

[26] M. Chabbi, A. Amer, S. Wen, and X. Liu, "An efficient abortable-locking protocol for multi-level NUMA systems," *SIGPLAN Notices*, vol. 52, no. 8, p. 61–74, Jan. 2017. [Online]. Available: https://doi.org/10.1145/3155284.3018768

[27] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta, "Productive programming of GPU clusters with OmpSs," in *International Parallel and Distributed Processing Symposium*, 2012.

[28] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. Dongarra, "PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability," *Computing in Science and Engineering*, vol. 15, no. 6, 2013.

[29] 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, Special Issue: Euro-Par 2009*, vol. 23, 2011.

[30] J. V. Ferreira Lima, T. Gautier, V. Danjean, B. Raffin, and N. Maillard, "Design and analysis of scheduling strategies for multi-CPU and multi-GPU architectures," *Parallel Computing*, vol. 44, pp. 37–52, 2015.

[31] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," *Theory of Computing Systems*, vol. 35, no. 3, pp. 321–347, 2002.

[32] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Washington, DC, USA: IEEE Computer Society Press, Nov. 2012, p. 1–11.

[33] M. Gonthier, L. Marchal, and S. Thibault, "Memory-aware scheduling of tasks sharing data on multiple GPUs with dynamic runtime systems," in *IEEE International Parallel and Distributed Processing Symposium*, 2022.

[34] ——, "Locality-Aware Scheduling of Independent Tasks for Runtime Systems," in *5th Workshop on Data Locality - 27th International European Conference on Parallel and Distributed Computing*. Lisbon, Portugal: Springer, Aug. 2021, pp. 1–12. [Online]. Available: https://hal.science/hal-03290998

[35] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: Association for Computing Machinery, 2009.

[36] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for terminally strict parallel programs," in *23rd IEEE International Parallel and Distributed Processing Symposium*, vol. 10, 2009.

[37] J.-N. Quintin and F. Wagner, "Hierarchical work-stealing," in *Euro-Par 2010 - Parallel Processing*, P. D'Ambra, M. Guarracino, and D. Talia, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 217–229.

[38] S. Shiina and K. Taura, "Almost deterministic work stealing," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019.