Pushing the Boundaries of Small Tasks: Scalable Low-Overhead Data-Flow Programming in TTG

Joseph Schuchart*, Poornima Nookala†, Thomas Herault*, Edward F. Valeev‡, George Bosilca*
* Innovative Computing Laboratory
The University of Tennessee
Knoxville, TN USA
Email: schuchart@icl.utk.edu
† Institute for Advanced Computational Science
Stony Brook University
Stony Brook, NY, USA
‡ Department of Chemistry
Virginia Polytechnic Institute and State University
Blacksburg, VA, USA

***Abstract*—Shared memory parallel programming models strive to provide low-overhead execution environments. Task-based programming models, in particular, are well-suited to cope with the ubiquitous multi- and many-core systems since they allow applications to express all available concurrency to a scheduler, which is tasked with exploiting the available hardware resources. It is general consensus that atomic operations should be preferred over locks and mutexes to avoid inter-thread serialization and the resulting loss in efficiency. However, even atomic operations may serialize threads if not used judiciously. In this work, we will discuss several optimizations applied to TTG and the underlying PaRSEC runtime system aiming at removing contentious atomic operations to reduce the overhead of task management to a few hundred clock cycles. The result is an optimized data-flow programming system that seamlessly scales from a single node to distributed execution and which is able to compete with OpenMP in shared memory.**

***Index Terms*—Dataflow graph, Template Task Graph, PaRSEC, TTG, Task-Based Programming**

I. Introduction

The number of cores per processor chip has been increasing for the past decade, reaching 64 cores on AMD's EPYC architecture [1]. Parallel runtime systems have countered this rise in shared memory concurrency by avoiding strong inter-thread synchronization in the form of mutexes and locks and instead focused on using atomic operations to safely update shared state inside the runtime. However, at current and future core counts, atomic operations, too, can become a bottleneck that may serialize the execution of concurrent threads inside the runtime system, negatively affecting the performance and scalability of applications, esp. if short-running tasks are involved. Selecting the right task-granularity is a trade-off between available concurrency and task management overhead and has been extensively studied [2], [3], [4], [5].

At the same time, many high-performance computing systems utilize specialized accelerators to achieve maximum performance at higher energy-efficiencies as is typically possible with general purpose CPUs. In such systems, the host CPU is often tasked with managing the execution streams of these accelerators, requiring low-latency responses to completions of kernels on the devices in order to initiate data transfers and new kernel launches. While on-device graph execution mechanisms [6] aim at reducing the cost of launching kernels, the irregular data movement of dynamic runtime systems often conflict with the unresponsiveness of such batching mechanisms. Hence, the threads controlling the device execution streams should be able to respond as fast as possible in order to offset the overheads of the device kernel management [7].

For these reasons, it is important for runtime system implementations to minimize synchronization between threads and provide efficient parallel execution at full node scale. Any time spent inside the runtime system is CPU time wasted for the application. Internal thread synchronization is an important culprit of wasted CPU cycles.

Figure 1 shows the thread-scaling behavior of atomic increments on both contended and uncontended (thread-private in our benchmark) atomic variables on two different architectures: AMD EPYC Rome and IBM Power9. The detailed system specifications are provided in Section V-A. What is immediately visible is the difference between contended and uncontended accesses. As expected, the latency for an uncontended atomic operation is independent of the number of threads performing atomic operations. This can be attributed to the fact that modern CPUs typically do not lock the whole memory bus but are able to lock individual cache lines [8]. Consequently, atomic variables that are rarely accessed by multiple threads do not typically pose a performance problem, except for the fact that atomic operations are typically slower than non-atomic operations (we measured approximately 5 ns on AMD and 20–38 ns per operation on Power9). Moreover, on Power9 a difference can be observed between sequentially consistent and relaxed memory ordering.

If, however, threads access a single shared atomic variable, these accesses are effectively serialized, increasing the average time per operation with 64 threads on AMD to around 530 ns and 1200 ns with 22 threads on Power9. In the quest for

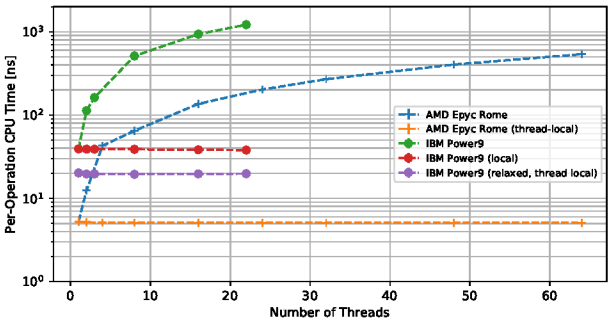


Fig. 1: Per-Operation latency of atomic increment on contended and uncontended variables.

achieving low-overhead parallel runtime systems, this is a strong incentive for avoiding atomic operations on shared variables wherever possible [9].

In this paper, we present the results of a systematic analysis of the Template Task Graph data-flow abstraction (TTG [10]) running on top of the PaRSEC runtime system [11] and discuss the bottlenecks found as well as their resolution. We will demonstrate that TTG is capable of executing tasks a short as 10k cycles with less than 2% overhead and 40k cycles with less than 1% overhead while fully utilizing all 64 cores on an AMD EPYC Rome CPU, a 10x improvement over the current implementation of *TTG* with the *PaRSEC* backend. We will also show that *TTG* is capable of competing with (and at times outperforming) OpenMP worksharing constructs in a parameterized task benchmark. While *TTG* seamlessly scales from shared memory to hundreds of nodes, we will focus on management of tasks in shared memory in this work.

The rest of the paper is structured as follows: Section II provides a short introduction to the *TTG* programming model and the underlying *PaRSEC* runtime system. An analysis of factors limiting scalability in is provided in Section III. Our solutions will be discussed in Section IV and an evaluated in Section V. Related work is discussed in Section VI and conclusions are drawn in Section VII.

II. Background

The Template Task Graph (*TTG*) data-flow based programming model has been introduced recently and aims at providing an abstraction for irregular applications [10]. In contrast to traditional task discovery models like OpenMP where the task graph is discovered up front, *TTG* applications build an abstract representation of *template tasks* (TT) connected through edges to form a *template task graph*, which may include cycles. During execution, an acyclic task graph unfolds through dynamic traversal of the template task graph. Data flows between tasks along edges and is sent dynamically during the execution of a task. This provides applications with great flexibility, e.g., to dynamically steer the unfolding of the template task graph based on input data.

In *TTG*, each task has a set of input data and is scheduled for execution once all inputs are satisfied. At any point during its execution, a task may send data into its output terminals, causing it to flow to instances of any connected successor template tasks (e.g., the *P2P* output terminal in Figure 2a is connected to the *P2P* input terminal; details will be provided in Section V-D1). Tasks are uniquely identified through *task IDs* (or *keys*), which can be any user-provided data type, e.g., an integer or a tuple uniquely describing the task.

TTG is implemented using modern C++ and can be implemented over multiple backends. Currently, *TTG* provides an implementation on top of *PaRSEC* and one on top of *MADNESS* [12]. In this work we will focus on the *PaRSEC* backend, which has become the main development backend and provides performance-relevant features, including data copy tracking and zero-copy data transfers between processes [13].

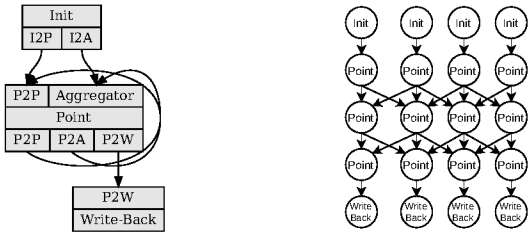
The *PaRSEC* backend in *TTG* utilizes the communication infrastructure of *PaRSEC* for distributed memory execution, including an active messaging layer and one-sided communication abstrations. Its main use, however, are *PaRSEC*'s task scheduling capabilities. Once a task becomes eligible for execution, it is dispatched to *PaRSEC* for execution, which owns the execution resources (thread pool) and provides a flexible scheduling infrastructure.

TTG also relies on *PaRSEC*'s termination detection mechanism, used to track outstanding tasks and in-flight messages and signaling completion once all work has completed (see Section III-A). This feature is essential in enabling the seamless transition from shared-memory to distributed-memory execution in *TTG*.

In the past, we have observed significant limitations in the scalability of *TTG* applications employing short tasks. One such example is the *multi-resolution analysis* (MRA) [14], which computes the multi-wavelet representation of 3D Gaussian functions. As we will show in Section V-E, the scaling of the MRA implementation under current *TTG* is limited to a speedup of 5x at 64 threads. We will demonstrate that the systematic analysis and elimination of bottlenecks in *PaRSEC* will lead to significant scalability improvements for MRA.

III. Performance Bottlenecks

Using the Linux perf tool [15], we have carefully analyzed the hot spots in the execution of small tasks in *TTG*.



(a) Template Task Gaph. (b) Unrolled sample task graph.

Fig. 2: Task-bench benchmark in TTG(Section V-D): the *Init* task feeds data into the graph, dispatching *Point* tasks that compute a point in the graph and send data to new *Point* instances for a given number of timesteps. Eventually, data is returned to the input data structure by the *Write-Back* task.

Three features of *PaRSEC* heavily used in *TTG* turned out to be critical for the performance of short-running tasks: the termination detection, the scalable hash table used to track discovered tasks, and the management of eligible tasks inside the scheduler. We carefully analyzed the usage of atomic operations throughout all the code paths used by *TTG* and removed any heavily contended atomic variables. While some features are shared between different *PaRSEC* frontends, *TTG* is the only frontend that utilizes all components discussed below.

A. Termination Detection

The termination of a *TTG* data-flow application is dependent on the execution of all tasks at all processes. Even when running on a single process, the difference N_P (pending tasks) between the number of discovered task N_D and the number of executed tasks N_E must become equal for the application to complete, i.e., $N_P = N_D - N_E = 0, N_D \geq N_E$. *TTG* over the *PaRSEC* backend makes use of the 4 counters wave algorithm implemented in *PaRSEC* [16]. In essence, this algorithm locally keeps track of the number of pending tasks and internal actions (N_A) as well as the number of messages sent and received. When termination is possible (i.e., $N_P + N_A = 0$ so any new work might only come from a new message), the process contributes to a reduction operation that accumulates at a root process the total number of messages sent and received. When these two counters are equal and remain unchanged for two consecutive reductions, global termination is announced.

The communication of local termination typically occurs infrequently, i.e., every time the process has completed all locally known tasks and pending messages. In data-flow based applications, this is not a significant source of overhead, as shown in [16]. However, the local accounting of pending tasks and pending actions has so far been implemented using atomic variables shared among all threads in a process, creating significant contention.

B. Scheduler Task Management

At the heart of every task-based runtime system is a scheduler mapping eligible tasks to a set of worker threads for execution. The scheduler is typically a passive element, i.e., no explicit scheduling thread is used to dispatch tasks. Instead, threads continuously query a data structure for eligible tasks.

The requirements for *TTG* are twofold. First, the scheduler must provide low-overhead, low-contention task distribution among threads (e.g., through stealing from thread-local queues) because the discovery of tasks is not balanced across threads. Second, the scheduler must support priorities in order to fully support the semantics of *TTG*, allowing applications to steer the execution along a critical path.

PaRSEC provides a set of scheduler implementations with different characteristics, all of which compromise on either one of the two requirements. An example of a queue that provides low-contention but is missing support for priorities is the *local-lifo* (LL) scheduler where each thread owns a LIFO into which tasks are pushed and from which other threads may steal tasks in case of starvation. The fundamental characteristics of a LIFO, however, do not allow for sorting based on priorities, since tasks are pushed to and extracted from its beginning.

The default scheduler in *PaRSEC* is *local-flat-queues* (LFQ), which defines a hierarchical structure to steal tasks between threads: each thread owns a bounded buffer of tasks and a global FIFO shared between all threads serves as overflow queue. If possible, tasks are pushed into free slots of the thread's bounded buffer. Otherwise, the tasks are pushed into the global FIFO. Tasks with the highest priority are kept to fill up the bounded buffer, and tasks with the lowest priority are enqueued into the LIFO, if necessary. When a thread selects a task for execution, it takes the task with highest priority in its local bounded buffer. If that buffer is empty, it tries to steal one task from the bounded buffer of any thread in the same domain of the cache and NUMA hierarchy. If no task can be found in any of the threads' bounded buffer, the first task in the system level FIFO queue is selected. The global FIFO may quickly become a bottleneck due to the global lock used to ensure consistency.

Priorities impact the order of selection in the bounded buffers and delay tasks with lower priority, which have a higher chance of landing in the global FIFO. However, they are not followed strictly, i.e., a task with lower priority might be executed before another task with higher priority that was discovered simultaneously.

C. Scalable Hash Table

At the heart of the task management in *TTG* is a scalable, thread-safe hash table. Each template task maintains such a hash table to store newly discovered tasks whose inputs are not fully satisfied immediately. Tasks are removed from it once the task becomes eligible for execution. When data is sent to successors, a lookup is performed for the task ID to check whether a task with this ID has been discovered already, and if so its inputs are updated.

Since inputs can be sent from within any task and thus by any thread at any time, the hash table has to ensure thread-safe insertion, removal, and lookup at a minimum cost. Moreover, the number of discovered tasks not yet eligible for execution is unbounded and in large applications may exceed several thousand entries (large scale runs easily reach millions of tasks per process). At the same time, allocating a large hash table upfront is not desirable due to the unnecessarily large memory footprint of the hash table in applications with small numbers of tasks. Even in large scale applications, the distribution of tasks between TTs is highly irregular. As a consequence, the hash table needs to dynamically adapt during the execution.

1) Scalabilty: Thus, the hash table implementation in *PaRSEC* supports efficient growth by chaining of tables. If the fill of one bucket in the main table exceeds a threshold (e.g., 16) a new main table with twice the number of buckets is allocated. All new entries will be inserted into the new table but old entries are not immediately moved into the new table, as depicted in Figure 3. Finding (and removing) entries involves

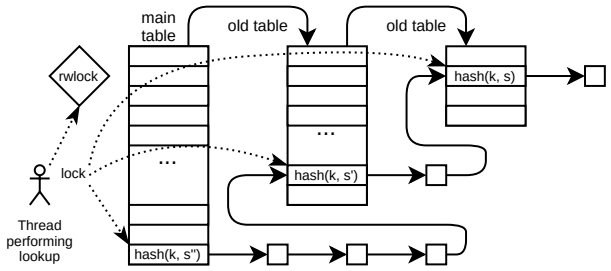


Fig. 3: Hash table in *PaRSEC*: threads performing lookups, insertions, and removal take a reader lock and lock individual buckets. For lookup and removal, threads traverse buckets in old tables until the element is found or the end of the table list is reached. Keys are remapped using the size s of a table.

traversing from the main table through the old tables until either the element is found or the end of the table chain is reached. A found element is moved into the main table to speedup the next search. As tasks only remain for a bounded time in the hash table, tasks that live in the old (smaller) tables are eventually removed, and small tables become empty. Once an old table is empty, it is removed from the list of tables, eventually leaving the main table as the only table in the list.

With this scheme, the hash table implementation in *PaRSEC* can hold anywhere between a handful and millions of elements, providing the basis for scalable task discovery.

2) Thread-safety: This dynamic table management requires a careful locking regime. Threads may decide to allocate a new main table while other threads access the hash table. The simplest solution of employing a central lock would severely limit thread-scalability. Instead, the API allows threads to lock individual buckets (identified by the key, not the index of the bucket) using a simple atomic lock (e.g., using `atomic_flag` in C11). While holding the lock, threads can safely find, insert, and remove an element for the key used to lock the bucket. With a reasonably balanced hash function and sufficient space in the hash table, this allows for thread-parallel access without synchronization. A typical pattern in *TTG* is to lock the bucket for a task ID, perform a lookup, insert an element if not found or remove an element if all inputs have been satisfied, and then unlock the bucket again.

However, a thread inserting an element may decide to create a new main table because the high water mark of a bucket was reached. The thread then has to wait for all other threads to release their bucket locks because they naturally do not own the lock on the bucket in the new table that would hold the key for which they have a lock. Otherwise, duplicated entries could occur, breaking the lookup of discovered tasks in *TTG*.

PaRSEC prevents such inconsistencies using a reader-writer lock, with threads locking a bucket taking a table-wide reader lock and threads wishing to resize the hash table taking a writer lock. While resizing is a rare event, taking and releasing the reader lock still incurs atomic operations, creating a synchronization point.

IV. Improvements

A. Atomic Memory Ordering

C11 has introduced a memory model that provides fine-grain control over the synchronization of memory accesses. By default, atomic memory operations provide *sequential consistent* ordering, preventing both the hardware and the compiler from reordering instructions around atomic memory operations. However, for atomic locks, the more relaxed *acquire-release* semantics are sufficient and provide both the compiler and CPU with leverage for some optimizations. For example, taking a lock on x86 using *acquire* memory ordering still involves the hardware infrastructure for atomic operations. However, releasing a lock with the *release* memory ordering is implemented using regular store, thanks to the *total store ordering* (TSO) of the x86 architecture [17]. The use of proper memory ordering for atomic locks thus removes one of two atomic operations from a lock-unlock cycle.

In addition, we use the *relaxed* memory ordering for all other atomic operations. In *TTG*, most atomic operations do not require acquire-release semantics and for the ones that do (e.g., atomic compare-and-swap for LIFO implementations) we use *acquire* and *release* memory barriers (using `atomic_thread_fence`).

B. Thread-Local Termination Detection

Instead of counting tasks on a per-process basis using atomic increment and decrement operations, we introduce a third layer and count executed and discovered tasks at the thread-level. Each thread maintains a counter that is incremented for each task discovered and decremented for each task executed by that thread. These updates occur non-atomically.

If a thread falls idle, it pushes its locally accumulated values to the process-wide counter using atomic updates, initiating the contribution to the global reduction if no other threads have pending local updates. Unless starvation and recovery occur regularly, the updates of process-wide counters should remain rare events. By introducing a third level of termination detection (thread-level, process-level, global) we managed to eliminate a choke point in *TTG*.

C. Local Task Queues with Priorities

In order to eliminate the serialization of the LFQ scheduler once the global FIFO is accessed, we designed a new scheduler that avoids single points of contention while preserving the ability to support priorities.

The LL scheduler fulfills the first requirement but does not support priorities. We thus implemented a variation of LL, called Local LIFO with Priorities (LLP). Similar to LL, every thread owns a LIFO into which it pushes tasks and from which other threads may steal tasks. However, we make two observations: i) only the owning thread may push tasks into its queue; and ii) a LIFO is a single-linked list whose head pointer is atomically changed during insertion and removal.

In order to support priorities, the owning thread pushes directly into the LIFO if the new task's priority is higher than the priority of the existing first element. In that case, the cost

of insertion is a single compare-and-swap (CAS) operation on the head pointer. If the priority is lower than the head element the thread detaches the head pointer, essentially marking the LIFO as empty, inserts the new task into the single-linked list, and reattaches the new list. Detaching the head pointer requires a CAS operation. Reattaching the LIFO can be done using a single store with release semantics.

In the worst case, the insertion of tasks into the single-linked list requires $\mathcal{O}(N)$ steps, with N the length of the list. We mitigate this by bundling new tasks into sorted lists that are then inserted in one pass. Moreover, new tasks will be inserted before old tasks that have the same priority, implicitly prioritizing tasks that may consume data already in the cache and potentially reducing the number of elements to traverse.

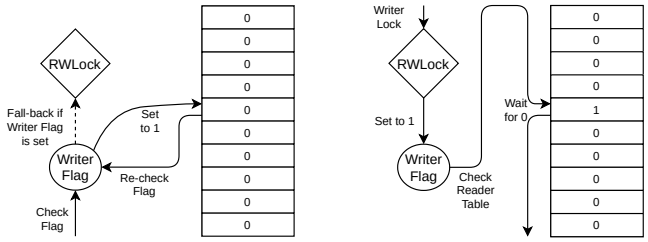
D. Reader-Biased Reader-Writer Locks

As described in Section III-C, the scalable hash table implementation provided by *PaRSEC* is an essential component for the data flow task management in *TTG* and provides a fine-grained locking mechanism for individual buckets. However, even with the acquire-release semantics introduced in Section IV-A, two atomic operations are required to lock a bucket: one for taking the bucket lock and one for the reader lock. For unlocking the bucket, on the other hand, one atomic operation is required, namely to release the reader lock. The reader-writer lock is thus a heavily contended variable.

However, for a given run of a given application, each TT holds a finite number of active tasks that need to reside simultaneously in its hash table. This means that for each hash table, there is a maximum number of resize operations during the execution. In practice, rarely more than 10 resize operations are observed, since each resize doubles the size of the hash table. Thus, the use of the reader-writer lock in the hash table is heavily biased towards readers.

This is a common occurrence in low-level system programming and a range of prior art exists on reader-biased locks [18], [19], [20]. We chose the BRAVO lock wrapper [19], which can sit on top of any custom reader-writer lock implementation and prevents the use of the underlying lock for the most common cases. The basis of the BRAVO lock wrapper is a table of flags which are set by threads taking the reader lock, and a global flag set by a thread taking the writer lock. As long as no writer lock is being taken, all that is required to take a reader lock is to set and unset the flag in the table, as depicted in Figure 4. Using proper memory ordering, a reader (Figure 4a) taking a lock checks the writer flag and—if not set—proceeds to set its flag in the table, before rechecking the writer flag to ensure no writer has arrived. If at any point while the reader lock is taken a writer is detected through the writer flag, the reader has to fall back to the underlying reader-writer lock. A writer (Figure 4b) takes the underlying lock and waits for all readers to release their flag in the table before proceeding.

Overall, no atomic operations are required for taking the reader lock in the fast path. In *TTG*, the number of threads in each process is static and known during initialization. Thus, we can allocate a table that is large enough to hold an entry for each thread and avoid sharing of cache lines, i.e., allocating at least one cache-line per thread in the table. Moreover, while the original paper proposes a single table per lock and hashing the thread and lock IDs to find a slot in the table, we implemented one table per lock to eliminate the chance for collisions and prevent any cache line sharing between threads. While taking a writer lock in this scheme is rather expensive, its rare occurrence ensures a minuscule impact on the overall application performance.



(a) Taking the reader lock. (b) Taking the writer lock.

Fig. 4: Steps for taking reader and writer locks using the BRAVO lock wrapper.

Overall, the use of the biased reader-writer lock removed two of the three atomic memory operations needed to lock a bucket. The only atomic operation left is that to take the bucket lock. Provided that there are at least 2^8 buckets and at most 16 collisions per bucket, these locks are unlikely to be contended.

E. Modeling the Use of Atomic Operations

We can model the number of atomic operations as follows. An instance of a task is represented through a task object. To manage these objects, *TTG* employs a free-list that contains a per-thread memory pool. Allocated elements are returned to the thread's memory pool from which they were allocated, to avoid imbalances between allocating and deallocating threads. Thus, the creation and destruction of a task involves two atomic operations ($N_{CD} = 2$).

For each of the N_I inputs of a task that have to be satisfied, one atomic operation has to be performed to increment the counter of available input data ($N_{ID} = 1$). If that data is reused (i.e., if no new copy is created), two additional atomic operation ($N_{IC} = 2$) are required on the reference count of the copy used to manage its lifetime, one while retaining the copy and one while releasing it (certain optimizations are applied if the current task is the final owner and the copy is either released or ownership is moved to a single successor). If, on the other hand, a new copy is created, i.e., because the data has to be assumed to be mutated by two different tasks, then memory is allocated, which we assume also involves at least one atomic operation in the underlying system allocator.

Moreover, the hash table bucket has to be locked and unlocked for each input, resulting in one atomic operation on the atomic lock of the bucket ($N_{IB} = 1$). For single-input tasks, access to the hash table can be eliminated because the a newly discovered task can be scheduled immediately.

Once a task becomes eligible for execution, it is pushed into the LLP scheduler using one atomic operation and taken out of the scheduler by a worker thread ready to execute the task using another atomic operation ($N_S = 2$).

Overall, the number of required atomic operations N_A in the lifetime of a task that works on existing data are

$$\begin{aligned} N_A &= (N_{ID} + N_{IC} + N_{IB}) \times N_I + N_{CD} + N_S \\ &= 4 \times N_I + 4 \end{aligned} \tag{1}$$

None of the atomic variables are contended, i.e., there may be a few threads accessing these variables in close proximity but at no point are all threads required to access them.

V. Evaluation

A. Systems under Test

We conducted our experiments on two platforms: *Summit*[1], an IBM system equipped with 22 core IBM Power9 dual-socket nodes installed at Oak Ridge National Laboratory (ORNL), and *Hawk*[2], a Hewlett Packard Enterprise system equipped with 64 core AMD EPYC Rome dual-socket nodes. On Summit, all codes compiled by us were compiled using GCC 11.1.0. On Hawk, we used GCC 10.2.0 and the Intel compiler in version 2022.0.0. In all cases, `-O3` was used to enable aggressive optimizations.

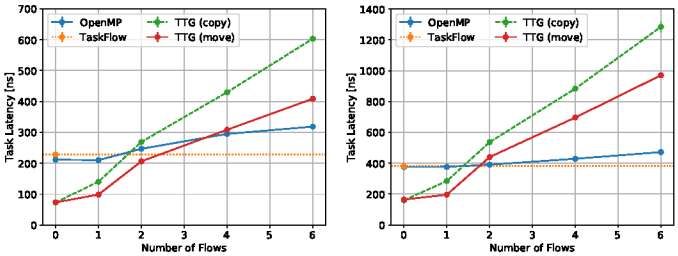
B. Minimum Task Latency

In order to determine the bare minimum task latency, we measure the execution of 10 million tasks with data flow or dependencies serializing their execution in TaskFlow [21], OpenMP [22], and *TTG*. These tasks were executed by a single thread, avoiding any cross-thread interference. The TaskFlow implementation of the benchmark only supports control-flow between tasks. In OpenMP, the benchmark uses N task dependencies between two successive tasks. In order to prevent a common optimization where OpenMP implementations inline tasks when running with a single thread, we ran 2 threads, blocking one in a sleep for a sufficiently long time to allow the other thread to complete the execution of the benchmark.

In the *TTG* implementation, each task sends data on its N output terminals to the N input terminals of its successor task. Currently, *TTG* does not merge data flows but instead inspects each data flow individually. Thus, for two or more flows the hash table described in Section III-C has to be queried to track the task between the handling of the different flows.

We measured two variants of the *TTG* benchmark: one in which the data is moved through the DAG and one in which the data is copied. In both cases, the data is a single integer value. While in the former case, no copies beyond the initial ingress copy has to be created, the variant in which data is copied between tasks requires creating a new copy between each task, causing significant allocation overheads.

Figure 5a shows the task latency in all four implementations on Hawk. For control-flow only (no actual data flowing between tasks), both TaskFlow and OpenMP show a latency of over 200 ns, while *TTG* shows a latency of 75 ns. Based on Figure 1 and Equation 1, 25% are due to atomic operations. For a single data flow, the latency of *TTG* increases to 100 ns and eventually meets the latency of OpenMP at 4 flows (300 ns). The jump between 1 and 2 flows is due to the addition of the hash table at 2 flows. The slope for OpenMP is smaller than for *TTG*, leading to approximately 400 ns for *TTG* and 310 ns for OpenMP at 6 flows/dependencies. This is likely due to the ability of OpenMP to inspect all dependencies at once. While it would be possible to extend *TTG* in that direction, the value of such an optimization would be small in real-world applications where tasks commonly send data to different successors.



(a) Hawk. (b) Summit.

Fig. 5: Task latency on Hawk for different number of flows in *TTG* and number of dependencies in OpenMP, using a chain of tasks. TaskFlow does not support multiple flows between the two same tasks at the time of this writing.

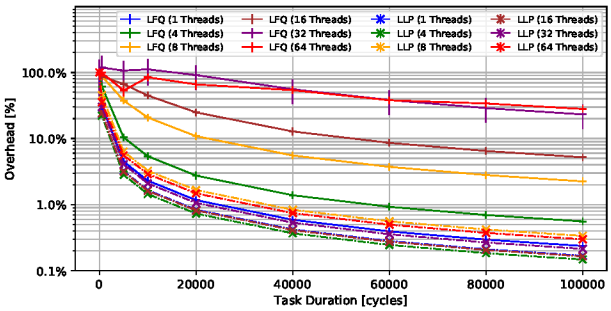
A similar picture emerges on Summit (Figure 5b), where *TTG* shows a minimum latency of 190 ns compared to 390 ns for OpenMP and TaskFlow. However, beyond two flows/dependencies, the gap between *TTG* and OpenMP grows larger, which we attribute to the necessary atomic operations in *TTG* and their relatively high cost on Power9 (cf. Figure 1). In all cases, approximately 50% of the latency in *TTG* can be attributed to atomic operations.

C. Task Scaling

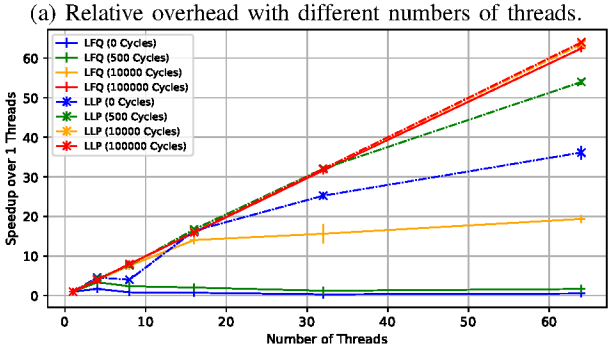
In order to compare the LLP scheduler described in Section IV-C against the LFQ scheduler, we use a benchmark that traverses a binary tree of height N, passing a single datum from the root to the leaves. Each non-leaf task discovers two successor tasks and each task has exactly one input and one output edge. Thanks to the single input, accessing the hash table can be avoided because discovered tasks can be scheduled directly for execution. Moreover, pure control flow is used with a single integer as task ID, avoiding any data life-time management. This puts significant pressure on the scheduler to handle an influx of tasks from all threads.

A total of $\sum_{n=0}^{N} 2^n$ tasks is discovered during the execution. We used $N = 22$ for our benchmarks, resulting in the discovery of approximately 4M tasks. In contrast to OpenMP [23], [24], *TTG* currently does not throttle the discovery of tasks to avoid the risk of starvation on other processes.

[1] https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/
[2] https://www.hlrs.de/systems/hpe-apollo-hawk/



(a) Relative overhead with different numbers of threads.



(b) Thread-scaling efficiency.

Fig. 6: Comparison of the LFQ and LLP scheduler under pressure on Hawk.

We varied the amount of work each task performs by blocking the execution of the task until a given number of cycles has passed (using the `rdtsc` counter). Figure 6a shows the relative overhead when using either the LFQ or LLP scheduler with varying numbers of threads on Hawk. We defined the overhead as $100 \times \frac{t_c}{t_0}$, where t_c is the time with c cycles spent in each task and t_0 the time with an empty task. The results indicate that the LLP scheduler by far outperforms the LFQ scheduler, even for a single thread. While with the LLP scheduler, even with 64 threads the overhead drops below 1% at about $c = 40k$, only configurations with 1 or 4 threads under LFQ reach below 1% overhead. The number of tasks exceeds the size of the thread-local bounded task buffer with LFQ, and the vast majority of tasks end up in the overflow FIFO that is shared between threads. As a consequence, almost all schedule operations cause contention on the lock protecting the global FIFO, severely inhibiting scalability of LFQ.

Figure 6b shows the speedup under different task granularities. The LLP scheduler achieves nearly perfect scaling with increasing numbers of threads for tasks of size 10k and 100k cycles and 90% efficiency for tasks of 1k cycles ($0.5\,\mu s$). For empty taks, the LLP scheduler still yields an efficiency of over 50%. We attribute that drop in efficiency to contention in the event of stealing due to imbalanced execution. The LFQ scheduler, on the other hand, exhibits relatively poor scalability for all but the largest task size.

While these results demonstrate that the the LLP scheduler is capable of handling high pressure, we note that the numbers provided here stem from an artificial benchmark in which a) half of the tasks (the leafs) do not discover successor tasks and b) the copy management is trivial. However, the LLP scheduler shows promising results when scaling both the number of tasks and threads.

D. Parameterized Task-Bench

In order to compare against other shared-memory and distributed task libraries, we implemented the parameterized Task-Bench benchmark [25] with *TTG*. Task-Bench provides some core functionality that was used by various programming model experts to implement a parameterized task benchmark in their respective model. Implementations must support a variable number of dependencies, which can be queries both forward and backward and are provided by the framework. The TTG used in our implementation and an unrolled graph of the 1D stencil are provided in Figure 2.

The design of the benchmark and its internal API are geared towards dependency-based models where a task's inputs and outputs are expressed in terms of points in a grid at task instantiation. In contrast, tasks in *TTG* have no knowledge of where input data originated from but must be aware of their successor tasks. The variable number of inputs are supported by backward-looking memory-based models such as OpenMP by satisfying task input dependencies from any previously discovered task with a matching output dependency, if any. In a forward-looking model such as *TTG*, however, a task ordinarily will have to know which successor task to send data to and how many inputs that task expects in order to satisfy the correct input. Some implementations in Task-Bench have faced a similar challenge, e.g., *PaRSEC* PTG.

We thus provide a quick overview over the implementation of Task-Bench in *TTG* and how we solved this challenge.

1) Implementation in TTG: While in real-world applications most tasks in *TTG* would expect a fixed number of inputs, the problem of flexible inputs came up in real-world applications before. The existing solution has been to use streaming terminals that accumulate the required number of elements into a custom data structure [13]. However, in this approach the *TTG* implementation loses track of data copies, requiring the copying of data between tasks even if the inputs are immutable and thus tasks could share the same copy.

We thus introduced a new abstraction called *aggregator terminals*. These terminals are similar to streaming terminals in that they allow for the accumulation of a certain number of input data. However, the data remains under the management of *TTG*, reducing the number of copies needed.

An example of how aggregators can be used to create the *Point* template task in Task-Bench is provided in Listing 1. The call to `ttg::make_aggregator` wraps an input edge such that an aggregate of inputs will be passed to the task. The number of inputs to be aggregated can be either a fixed value or determined for each task separately, e.g., through `compute_num_inputs` in Line 17.

In Task-Bench, the equivalent of `compute_num_inputs` queries the number of inputs of each task. Since there is no guaranteed order of the inputs in the aggregator, the task body

```
1  /**
2   * Create a template task that aggregates
3   * a number of elements from in_edge
4   * before executing.
5   */
6  void make_point_tt(ttg::Edge<Key, data_t>& in_edge) {
7    auto taskfn = [](const Key& key
8                     const ttg::Aggregator<data_t>& values){
9      std::vector<data_t> sorted_inputs;
10     for (const data_t& value : values) {
11       sorted_insert(value, sorted_inputs);
12     }
13     execute_point(key, sorted_inputs);
14     broadcast_successors(key, values);
15   };
16   auto aggregator_count = [](const Key& key){
17       return compute_num_inputs(key);
18   };
19   // the aggregator edge calls the provided callback
20   // to determine the number of inputs for each task
21   auto aggregator_edge = ttg::make_aggregator(
22                              in_edge,
23                              aggregator_count);
24   return ttg::make_tt(fn, ttg::edges(aggregator_edge));
25 }
```

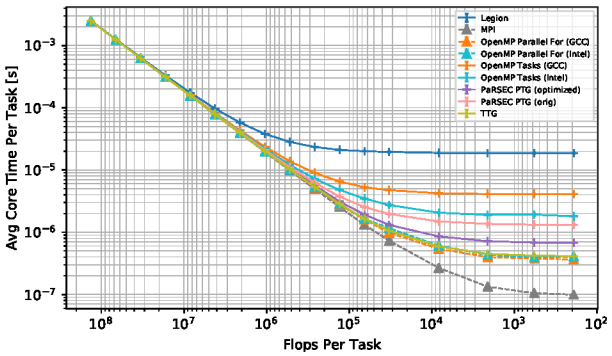
Listing 1: Skeleton implementation of Task-Bench using aggregator terminals in *TTG*.

orders these inputs according to their origin (Line 11), based on information about its origin stored in the data element. Finally, the task has to query its successor tasks and broadcast its output data (Line 14). In summary, each task has to query its predecessors twice and its successors once. While this benchmark induced overhead cannot be eliminated here, we expect real-world applications to be more constrained and thus require less effort when using aggregators.

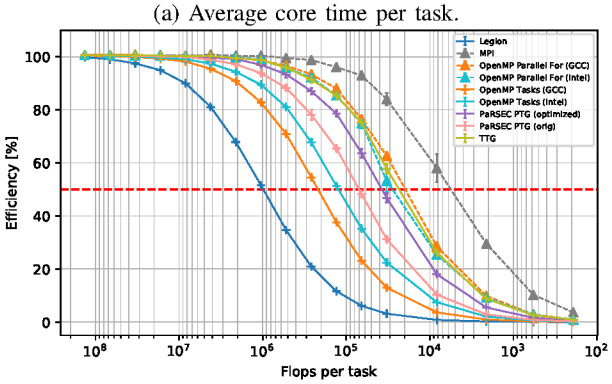
2) Single Core Results: We compare the *TTG* implementation against Legion [26] (v22.03.0), pure MPI, MPI+OpenMP work-sharing loops (only OpenMP worksharing loops if running in shared memory), OpenMP tasks (shared memory only) as well as *PaRSEC* PTG [27]. All but the *TTG* implementation were taken from the Task-Bench repository.[3] OpenMP tasks, *PaRSEC* PTG, Legion, and *TTG* all use a task-based abstraction, while pure MPI and MPI+OpenMP work-sharing loops do not explicitly define the concept of tasks and are closer to fork-join parallelism or Bulk-Synchronous Parallelism programming models. Although Legion, *PaRSEC*, *TTG*, and MPI are capable of distributed memory runs, we will focus solely on shared memory execution in this work. For all measurements, we used the 1D stencil dependency pattern (2+1 dependencies; Figure 2b) and the compute-bound kernel.

PaRSEC PTG has been found to be the most efficient truly task-based implementation [25]. In the interest of clarity we did not include all other Task-Bench contenders but refer the interested reader to the original publication. We chose to include MPI and OpenMP because they represent the *lingua franca* of parallel programming. We set `OMP_PLACES=cores` and `OMP_PROC_BIND=spread` for all OpenMP runs. Our choice of Legion was based on our understanding that it is developed by the same authors as Task-Bench.

[3] https://github.com/StanfordLegion/task-bench



(a) Average core time per task.



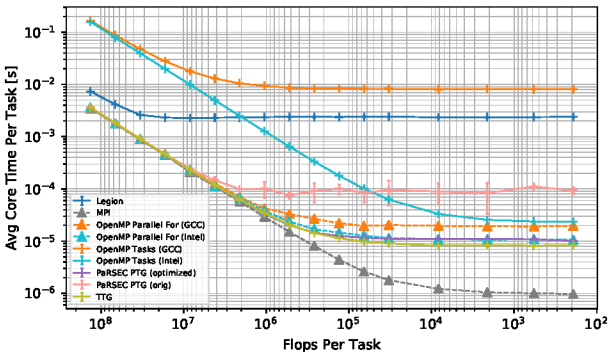
(b) Efficiency under decreasing flops-per-task on Hawk.

Fig. 7: Task-Bench results on 1 core on Hawk.

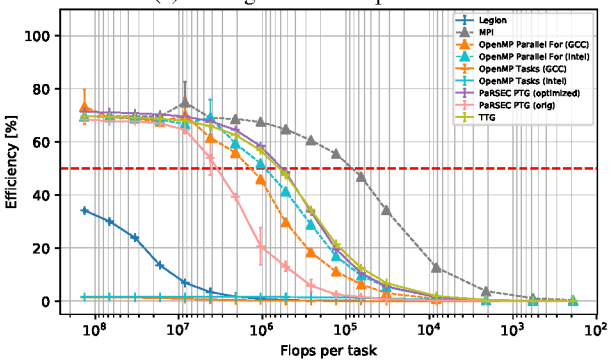
Similarly to the original paper, we ran for 1000 timesteps with one task per core per timestep, maximizing the competition of threads for tasks of varying sizes. For *TTG*, we enabled thread-local termination detection and the LLP scheduler.

Figure 7 shows the results of the Task-Bench benchmark on a single thread on Hawk. Figure 7a shows the average core time (including all overheads) per task. It becomes clear that the MPI-only variant achieves the lowest per-task execution time, approximately 4x lower than OpenMP for loops and *TTG*. While the numbers for *TTG* are significantly higher than the 75 ns observed in Section V-B it should be noted that the skeleton around the task kernels required in the Task-Bench implementation is significantly more complex and expensive, as described earlier. *TTG* is followed by *PaRSEC*, with and without the optimizations described in this paper, and by the OpenMP task-based implementations. The optimizations presented in this work have shown to benefit not only *TTG* but also *PaRSEC* PTG.

Figure 7b shows the resulting efficiency of the different implementations under decreasing task sizes, where the baseline (100%) is the highest performance observed on a single core. Using that information, we can determine the *minimum effective task granularity* (METG) [25] for a given efficiency number. For example, METG(50%) for pure MPI is at approximately 6k flops per task while for *TTG* and OpenMP work-sharing loops the same efficiency is achieved for approximately 20–25k flops per task. We attribute the superior performance of the pure MPI implementation to



(a) Average core time per task.



(b) Efficiency under decreasing task size.

Fig. 8: Task-Bench results on 64 cores on Hawk.

the fact that there is no task handling overhead and the lack of communication when running on a single core. This matches the observations in the original publication. All other implementations dispatch the execution of tasks to a runtime library. It is interesting to note that both implementations of OpenMP tasks yield a significantly higher METG(50%) above 100k flops per task.

3) Thread-Scaling Results: Figure 8 provides the same information for runs with 64 threads on Hawk. From Figure 8a it can be seen that both *TTG* and the optimized *PaRSEC* PTG are on-par with the variant using OpenMP worksharing constructs under the Intel runtime, outperforming the same variant compiled with GCC. The OpenMP variants using tasks show significantly worse scalability. This is mirrored in Figure 8b, where the efficiency is relative to the best single-core performance multiplied by the number of threads. The METG(50%) of *TTG* is at approximately 60k flops per task while the value for the best OpenMP worksharing variant is found at 1M flops.

Figure 9 provides a breakdown of the most important changes in addition to LLP (discussed in Section V-C and less relevant for small numbers of tasks), namely the thread-local termination detection and the biased reader-writer lock. The results underscore that it is important to thoroughly analyze all parts of a runtime system in order to gain the best performance and that any bottleneck will inevitably limit scalability.

We ran a similar set of benchmarks on Summit, albeit with a reduced set of task granularities and variants. However, a

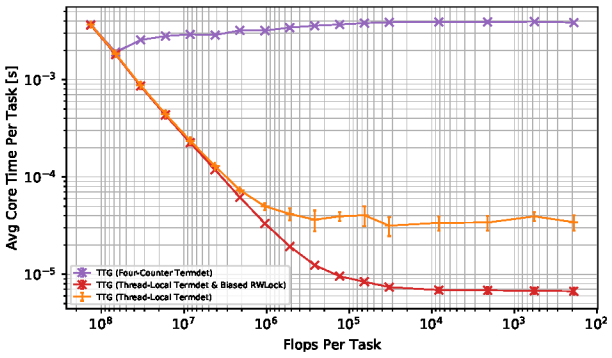
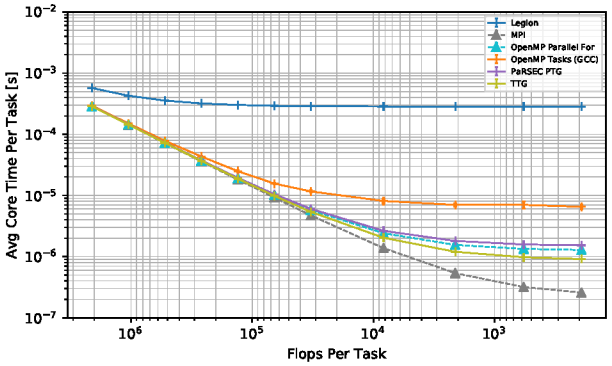
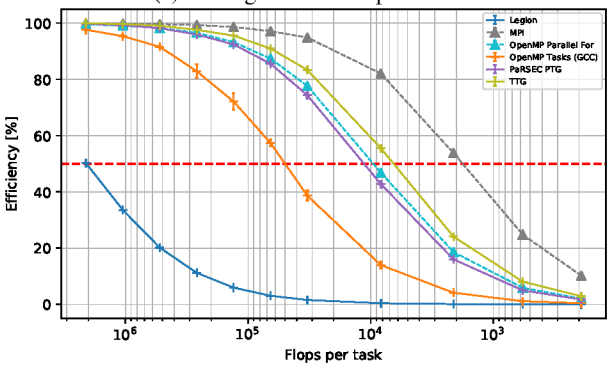


Fig. 9: Breakdown of contributions of thread-local termination detection and biased reader-writer lock on 64 threads on Hawk.



(a) Average core time per task.



(b) Efficiency under decreasing task size.

Fig. 10: Task-Bench results on a single core on Summit.

similar picture emerges from Figure 10 for a single core: MPI shows the lowest latency, followed by *TTG*, OpenMP worksharing loops, the optimized *PaRSEC*, and the OpenMP task-based variant. At 22 cores (Figure 11) the gap between the three groups—MPI, *TTG*/*PaRSEC*/OpenMP worksharing, and OpenMP tasks—widens further, suggesting significant scalability challenges. The three implementations in the second group still yield latencies around 20 μs.

E. Multi-Resolution Analysis: Thread-Scaling

The MRA benchmark computes the order-10 multi-wavelet [14] representation of 3D Gaussian functions (exponent 30 000) to precision of 10^{-8} with Gaussian centers

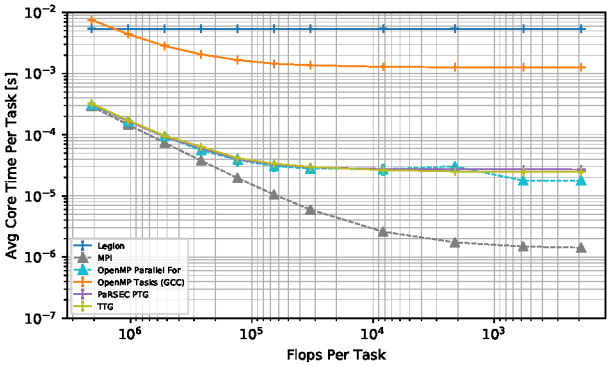


Fig. 11: Task-Bench results on 22 cores on Summit.

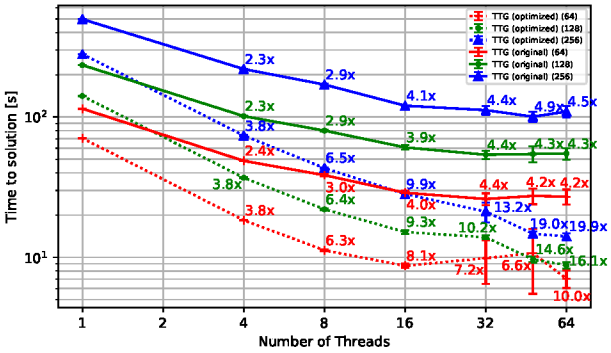


Fig. 12: Time to solution of MRA with the original (solid lines) and optimized (dotted lines) *TTG* over *PaRSEC* on Hawk with different numbers of functions computed concurrently (64, 128, 256). Numbers on each point show the speedup compared to the run with 1 thread.

distributed randomly in a $[-6, 6]^3$ volume. The computation comprises three steps: *projection* results in a 3D spatial data structure; *compression* flows data up the tree; and *reconstruction* flows data down the tree. Of those three steps, the projection step is the most costly part, each computing a GEMM on 20^2 double precision matrices.

Figure 12 shows the time to solution of MRA on Hawk under both the original *TTG* and *TTG* with the optimization presented in this work. Numbers above each point show the speedup compared to the corresponding run with 1 Thread. While the original *TTG* achieved close to a 5x speedup up to 32 threads, the optimized *TTG* achieves close to 20x speedup at 48 threads for 256 functions. Although the results are far from ideal when scaling beyond 8 threads, they do provide a significant improvement over the original *TTG* implementation. We believe that additional improvements such as inlined tasks to reduce the number of very short tasks will help yield further improvements. However, these extensions of the TTG API are beyond the scope of this paper, which focuses on runtime system improvements.

VI. Related Work

There exists a large body of literature on the topic of concurrent hash tables, and comparing with all existing approaches is outside the scope of this work. We refer the reader to [28] for a recent survey of concurrent hash table techniques. Spinlocks are used to ensure the atomicity of buckets, and reader-writer spinlocks using [29] and the BRAVO optimization [19] ensure the atomicity of resizing. A popular alternative to using reader-writer locks are lock-free hash tables [30], [31]. The focus of lock-free data structures is the avoidance of starvation of threads in the event that a thread holding a lock disappears or is blocked for significant time. Modification of entries typically involves a sequence of atomic operations and increased complexity. This typically does not apply to systems like *TTG* and *PaRSEC* where any fault of a thread will lead to termination of the application.

Other benchmarks specific to measuring the overhead of OpenMP [32] and OpenMP tasks [33] have been proposed in the past. Task-Bench [25] is an attempt to evaluate task-based runtime systems independently of their programming API or model. In it, the authors use Task-Bench to compare a variety of task-based runtime systems: OmpSs [34], OpenMP 4.0[22], two DSLs of *PaRSEC* [27], [35], Legion via both its Realm abstraction [36], and Regent [37], and StarPU [38]. They also compare implementations of Task-Bench over MPI, non-task-based Open MP, and the PGAS languages Chapel [39], Charm++ [40] and X10 [41]. In this work, we added *TTG* to this large set, and we focus the comparison with the most performing or the most popular runtime systems already evaluated in [25]. Other approaches focus on small task granularities in specific applications [42] and the trade-off of picking the right task granularities has been extensively studied [2], [3], [4], [5].

VII. Conclusions

We have presented a detailed analysis of bottlenecks in *TTG* and described how they have been eliminated. The results demonstrate that *TTG* is capable of handling control-flow tasks in less than 100 ns and in a 1D stencil benchmark is able to outperform traditional OpenMP worksharing loops in shared memory at high numbers of threads. We have also demonstrated that the improvements made in this work can substantially improve the performance and scalability of a mini-app implemented in *TTG*. Overall, the results suggest that careful optimization can lead to low task overheads that in turn enable the use of short tasks without sacrificing performance. The optimizations made here will also pave the way for future GPU support in *TTG* to provide low-latency coordination and efficient data flow between kernels executing on accelerators.

Acknowledgments

This research was supported partly by NSF awards #1450300, #1450344 and #1450262, and 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. We gratefully acknowledge the provision of computational resources by the High Performance Computing Center (HLRS) at the University of Stuttgart, Germany.

References

[1] S. Naffziger, N. Beck, T. Burd, K. Lepak, G. H. Loh, M. Subramony, and S. White, "Pioneering Chiplet Technology and Design for the AMD EPYC™ and Ryzen™ Processor Families : Industrial Product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[2] S. Bora, B. Walker, and M. Fidler, "The tiny-tasks granularity trade-off: Balancing overhead vs. performance in parallel systems," 2022. [Online]. Available: https://arxiv.org/abs/2202.11464

[3] A. Gerasoulis and T. Yang, "On the granularity and clustering of directed acyclic task graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 6, pp. 686–701, 1993.

[4] S. Shudler, A. Calotoiu, T. Hoefler, and F. Wolf, "Isoefficiency in practice: Configuring and understanding the performance of task-based applications," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 131–143. [Online]. Available: https://doi.org/10.1145/3018743.3018770

[5] A. Navarro, S. Mateo, J. M. Perez, V. Beltran, and E. Ayguadé, "Adaptive and architecture-independent task granularity for recursive applications," in *Scaling OpenMP for Exascale Performance and Portability*, B. R. de Supinski, S. L. Olivier, C. Terboven, B. M. Chapman, and M. S. Müller, Eds. Cham: Springer International Publishing, 2017, pp. 169–182.

[6] Nvidia Inc., "CUDA C++ Programming Guide," May 2022. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[7] L. Zhang, M. Wahib, and S. Matsuoka, "Understanding the overheads of launching CUDA kernels," *ICPP19*, 2019.

[8] H. Schweizer, M. Besta, and T. Hoefler, "Evaluating the cost of atomic operations on modern architectures," 2020. [Online]. Available: https://arxiv.org/abs/2010.09852

[9] 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 *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2021, pp. 1–8.

[10] G. Bosilca, R. J. Harrison, T. Herault, M. M. Javanmard, P. Nookala, and E. F. Valeev, "The Template Task Graph (TTG) - an emerging practical dataflow programming paradigm for scientific simulation at extreme scale," in *IEEE/ACM 5th Intl. Wksp. on Extreme Scale Programming Models and Middleware (ESPM2)*, Nov. 2020, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/document/9307054

[11] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. Dongarra, "PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability," *Comp in Sc. and Eng.*, vol. 99, p. 1, 2013.

[12] R. J. Harrison, G. Beylkin, F. A. Bischoff, J. A. Calvin, G. I. Fann, J. Fosso-Tande, D. Galindo, J. R. Hammond, R. Hartman-Baker, J. C. Hill, J. Jia, J. S. Kottmann, M. Y. Ou, L. E. Ratcliff, M. G. Reuter, A. C. Richie-Halford, N. A. Romero, H. Sekino, W. A. Shelton, B. E. Sundahl, W. S. Thornton, E. F. Valeev, Á. Vázquez-Mayagoitia, N. Vence, and Y. Yokoi, "MADNESS: A multiresolution, adaptive numerical environment for scientific simulation," *SIAM J. Sci. Comput.*, vol. 38, no. 5, 2016.

[13] J. Schuchart, P. Nookala, M. M. Javanmard, T. Herault, E. F. Valeev, G. Bosilca, and R. J. Harrison, "Generalized Flow-Graph Programming Using Template Task-Graphs: Initial Implementation and Assessment," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022.

[14] B. Alpert, G. Beylkin, D. Gines, and L. Vozovoi, "Adaptive solution of partial differential equations in multiwavelet bases," *Journal of Computational Physics*, vol. 182, no. 1, 2002.

[15] "perf: Linux profiling with performance counters," accessed May 19, 2022. [Online]. Available: perf.wiki.kernel.org

[16] G. Bosilca, A. Bouteiller, T. Herault, V. Le Fevre, Y. Robert, and J. Dongarra, "Comparing distributed termination detection algorithms for task-based runtime systems on HPC platforms," *International Journal of Networking and Computing*, vol. 12, no. 1, 2022.

[17] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "X86-TSO: A Rigorous and Usable Programmer's Model for X86 Multiprocessors," *Commun. ACM*, vol. 53, no. 7, jul 2010.

[18] R. Liu, H. Zhang, and H. Chen, "Scalable read-mostly synchronization using passive reader-writer locks," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. USA: USENIX Association, 2014.

[19] D. Dice and A. Kogan, "BRAVO—Biased locking for Reader-Writer locks," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019.

[20] C. E. Nemitz, S. Caspin, J. H. Anderson, and B. C. Ward, "Light Reading: Optimizing Reader/Writer Locking for Read-Dominant Real-Time Workloads," in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), B. B. Brandenburg, Ed., vol. 196. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

[21] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," *IEEE TPDS*, 2021.

[22] "OpenMP OpenMP Application Programming Interface, Version 5.2," Sep 2021. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[23] T. Gautier, C. Perez, and J. Richard, "On the Impact of OpenMP Task Granularity," in *Evolving OpenMP for Evolving Architectures*, B. R. de Supinski, P. Valero-Lara, X. Martorell, S. Mateo Bellido, and J. Labarta, Eds. Springer International Publishing, 2018.

[24] A. Duran, J. Corbalan, and E. Ayguade, "An adaptive cut-off for task parallelism," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[25] E. Slaughter, W. Wu, Y. Fu, L. Brandenburg, N. Garcia, W. Kautz, E. Marx, K. S. Morris, Q. Cao, G. Bosilca, S. Mirchandaney, W. Leek, S. Treichlerk, P. McCormick, and A. Aiken, "Task Bench: A Parameterized Benchmark for Evaluating Parallel Runtime Performance," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[26] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Supercomputing*, 2012.

[27] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra, "PTG: An abstraction for unhindered parallelism," *Proceedings of WOLFHPC'14*, 2014.

[28] T. Maier, P. Sanders, and R. Dementiev, "Concurrent hash tables: Fast and general (?)!" *ACM Transactions on Parallel Computing (TOPC)*, vol. 5, no. 4, 2019.

[29] B. B. Brandenburg and J. H. Anderson, "Spin-Based Reader-Writer Synchronization for Multiprocessor Real-Time Systems," *Real-Time Syst.*, vol. 46, no. 1, 2010.

[30] J. P. Nielsen and S. Karlsson, "A scalable lock-free hash table with open addressing," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. Association for Computing Machinery, 2016.

[31] D. R. Martin and R. C. Davis, "A Scalable Non-Blocking Concurrent Hash Table Implementation with Incremental Rehashing. Unpublished manuscript," 1997.

[32] J. M. Bull, "Measuring synchronisation and scheduling overheads in openmp," in *Proceedings of First European Workshop on OpenMP*, vol. 8, 1999.

[33] J. Schuchart, M. Nachtmann, and J. Gracia, "Patterns for OpenMP Task Data Dependency Overhead Measurements," in *Scaling OpenMP for Exascale Performance and Portability*, B. R. de Supinski, S. L. Olivier, C. Terboven, B. M. Chapman, and M. S. Müller, Eds. Springer International Publishing, 2017.

[34] A. Duran, R. Ferrer, E. Ayguade, R. M. Badia, and J. Labarta, "A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks," *Intl. Journal of Parallel Programming*, vol. 37, no. 3, 2009.

[35] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, "Dynamic Task Discovery in PaRSEC: A Data-flow Task-based Runtime," in *Proceedings of ScalA'17*, 2017.

[36] S. Treichler, M. Bauer, and A. Aiken, "Realm: An event-based low-level runtime for distributed memory architectures," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. ACM, 2014.

[37] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: a high-productivity programming language for HPC with logical regions," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015.

[38] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Conc. Comp. Pract. Exper.*, vol. 23, 2011.

[39] B. L. Chamberlain, "Chapel," in *Programming Models for Parallel Computing*, ser. MIT Press, P. Balaji, Ed., 2015.

[40] L. V. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, 1993.

[41] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *Acm Sigplan Notices*, vol. 40, no. 10, 2005.

[42] D. Haensel, L. Morgenstern, A. Beckmann, I. Kabadshow, and H. Dachsel, "Eventify: Event-Based Task Parallelism for Strong Scaling," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, ser. PASC '20. Association for Computing Machinery, 2020.