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

Léopold Cambier
ICME, Stanford University
lcambier@stanford.edu

Yizhou Qian
ICME, Stanford University
yzqian@stanford.edu

Eric Darve
ME & ICME, Stanford University
darve@stanford.edu

***Abstract*—We present TaskTorrent, a lightweight distributed task-based runtime in C++. TaskTorrent uses a parametrized task graph to express the task DAG, and one-sided active messages to trigger remote tasks asynchronously. As a result the task DAG is completely distributed and discovered in parallel. It is a C++14 library and only depends on MPI. We explain the API and the implementation. We perform a series of benchmarks against StarPU and ScaLAPACK. Micro benchmarks show it has a minimal overhead compared to other solutions. We then apply it to two large linear algebra problems. TaskTorrent scales very well to thousands of cores, exhibiting good weak and strong scalings.**

I. Introduction

A. Parallel runtime systems

Classical parallel computing has traditionally followed a fork-join (as in OpenMP) or bulk-synchronous (MPI) approach. (Figure 1a shows the skeleton of a typical MPI program). This has many advantages, including ease of programming and predictable performance. It has however a key downside: many points of synchronization during execution are added, even when not necessary.

Runtime systems take a different approach. The key concept is to express computations as a graph of tasks with dependencies between them (Figure 1b). This graph is directed and acyclic, and we will later refer to it as the task DAG. Given the DAG, the runtime system is able to extract parallelism by identifying which tasks can run in parallel. Tasks are then assigned to processors (either individual cores, nodes, accelerators, etc). The advantage of this method is that it removes all unnecessary synchronization points.

B. Existing approaches to describe the DAG

A key design choice in runtime systems is how to express the DAG. At a high-level, two approaches have been primarily used.

1) Sequential Task Flow (STF): In this approach, the graph is discovered by the runtime using a sequential semantics, that is, typically, on each node a single thread is responsible for building the DAG. Different mechanisms to compute task dependencies can be used. Often, this takes the form of inferring dependencies based on specifiying data sharing rules (e.g., READ, WRITE, READWRITE).

L.C. was supported by a fellowship from Total S.A. Y.Q. and E.D. were supported by Grant no. 80NSSC18M0152 from NASA.

This is the approach taken by Legion/Regent [1] [1] and StarPU [2]. In both, the user first defines data regions and tasks operating on those regions (as inputs or outputs). Regent maintains a global view of the data, and data regions correspond to a partitioning of the data. The user is also able to write mappers to indicate how to map and schedule tasks to the available hardware. StarPU uses data handles referring to distributed memory buffers. The program is then written in a sequential style (with for loops, if/else statements, etc.), creating tasks on previously registered data regions. The runtime system then discovers task dependencies, builds the DAG and executes tasks in parallel.

The key in the STF approach is that the DAG has to be discovered through sequential enumeration. This restriction may have performance implications but is attractive to the programmer, since the program is easy to write and understand.

2) Parametrized Task Graph (PTG): The PTG approach is another method to express the DAG. Using some index space (`K`) to index all tasks, functions of `K` are used to express tasks and their dependencies. As an example, the DAG could be defined by specifying three functions of `K` (other choices are possible): one for the in-dependencies, one for the computational task itself and one for the out-dependencies. By running these functions as needed, the runtime discovers the DAG dynamically.

PaRSEC [3] takes that approach, using a custom language (JDF) to express the PTG. In PaRSEC, in and out-dependencies specifications contain both tasks and data.

[1]Legion is the name of the lower level C++ API, while Regent is the name of the higher-level language based on Lua.

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

(a) A typical MPI program

$$\begin{tikzcd}
 & c \arrow[r] & f \\
a \arrow[ur] \arrow[dr] & & \\
 & d \arrow[dr] & \\
b \arrow[ur] \arrow[dr] & & g \\
 & e \arrow[ur] &
\end{tikzcd}$$

(b) Example of DAG of tasks.

Fig. 1: More parallelism can be extracted using a tasks DAG: task d needs to wait for task a and b. However, task f can run as soon as task c has finished.

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

(a) STF based program. Dependencies are inferred through data sharing rules.

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

(b) PTG based program. Task dependencies are defined using functions over `K`. Computation is triggered by seeding the initial tasks.

Fig. 2: Schematic of STF and PTG programs.

The PTG format has multiple advantages. Since task in/out-dependencies can be independently queried at any time, it simplifies task management, leading to minimal overhead during execution. It also naturally scales by parallelizing both the DAG creation and DAG execution. In contrast, a STF code uses, in its purest form, a single thread to discover the DAG. It also removes the need to store in memory large portions of the DAG of tasks. Instead, the runtime can query the relevant functions only as needed and discover the DAG piece by piece.

The main drawback of the PTG approach is that the program no longer has a sequential semantics, which makes it harder to understand the program's behavior at first sight. Figure 2 illustrates at a higher level the differences between the STF and the PTG approach.

C. Contributions

In this paper, we present TaskTorrent (`TTor`). `TTor` is a lightweight, distributed task based runtime that uses a PTG approach. Our main contributions are:

- We show how to combine a PTG approach with one-sided active messages.
- A mathematical proof is provided for the correctness of our implementation.
- We benchmark `TTor` and show that it matches or exceeds performance of StarPU on sample problems.

`TTor` has a couple of notable features compared to existing solutions

- It is a C++14 library with no dependencies other than MPI.
- `TTor`'s implementation leads to a small overhead and handles well small task granularity (about 10 μs and up). This means that `TTor` can be used on any existing code, without needing to fuse or redefine tasks, or change existing algorithms.
- Default options in `TTor` are designed to provide good performance "out-of-the-box" without requiring the user to tune or optimize internal parameters or functionalities of the library.
- The user can use their own data structures without having to wrap their data in opaque data structures.
- It is perfectly scalable in the following sense. Consider a provably scalable numerical algorithm (e.g., there exists an iso-efficiency curve). Assume that (1) the parallel computer is composed of nodes with a bounded number of cores, but with an unbounded number of nodes, and (2) that each node in the DAG has a bounded number of dependencies. Then if the algorithm is executed using `TTor` it will remain scalable. Said more simply, `TTor` does not introduce any parallel bottleneck.

We emphasize that `TTor` is a general purpose runtime system. The applications in this paper are mostly in dense linear algebra, but there are no features or optimizations that are specific to linear algebra in this version of `TTor`.

D. Organization of the paper

This paper is organized as follows. Section II describes `TTor`'s API and implementation. Section III compares `TTor` to StarPU and ScaLAPACK, first validating its shared memory component and then comparing it on large linear algebra problems. We finally survey previous work in Section IV before concluding.

II. TaskTorrent

`TTor` uses a PTG. The DAG is expressed by providing at least three functions: (1) one returning the number of in-dependencies of every task; (2) one that runs the computational task and fulfills dependencies on other tasks; (3) one returning the thread each task should be mapped to (an option is provided to bound the task to the thread or leave it stealable). When their dependencies are satisfied, tasks are inserted into a thread pool, where a work-stealing algorithm keeps the load balanced between the threads.

Tasks then run and fulfill other tasks' dependencies, locally (on the same rank) or remotely on a different rank. In the case of remote dependencies, since all computations are asynchronous, the receiver rank cannot explicitly wait for data to arrive. Hence, one-sided active messages are used. An active message (AM) is a pair (function, data). Once the AM arrives on the receiver, the function is run with the data passed as argument. This is typically used to store the data and fulfill dependencies, eventually triggering more tasks.

This approach means `TTor` never needs to store the full DAG. Task dependencies are queried only when needed, and the DAG is discovered piece by piece. In particular, `TTor` becomes aware of the existence of a specific task **only** when a task fulfills its first dependency. This makes `TTor` scalable and lightweight. The full DAG is never stored or even explored by any specific thread or rank, and the task management overhead is minimal. Figure 3 illustrates this local DAG + AM model.

A. API Description

`TTor`'s API can be divided into two parts, a shared memory component (expressing the PTG) and a distributed component (used for AMs). The combination of those two features is what distinguishes `TTor` from other solutions and is one of the factors that makes `TTor` lightweight.

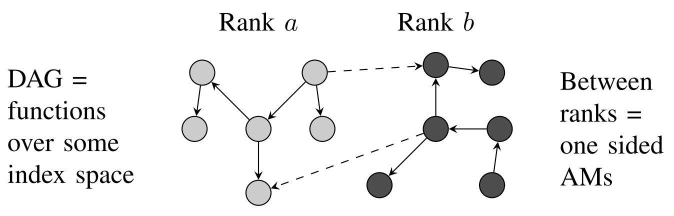

Rank a
Rank b
DAG = functions over some index space
Between ranks = one sided AMs

Fig. 3: The model of `TTor`: a distributed graph of tasks expressed using a parametrized task graph (solid arrows), with explicit active messages (dashed arrows) between ranks to asynchronously insert/trigger tasks.

1) Shared memory components:

a) Threadpool: A `Threadpool` is a fixed set of threads that receive and process tasks. A threadpool with `n_threads` threads can be created by `Threadpool tp(n_threads, &comm)`. (`comm` is a `Communicator`; see Section II-A2). Tasks can be inserted directly in the threadpool, but typically this is done using a `Taskflow`. The threadpool joins when calling `tp.join()`. This returns when all the threads are idle and all communications have completed. Section II-B3 explains in details the distributed completion mechanism.

b) Taskflow: A `Taskflow<K> tf` (for some index space `K`, typically an integer or a tuple of integers) represents a Parametrized Task Graph. It is created using `Taskflow<K> tf(&tp)` where `tp` is a `Threadpool`. It is responsible for managing task dependencies and automatically inserting tasks in `tp` when ready. At least three functions have to be provided:

- `(int)indegree(K k)` returns the number of dependencies for task `k`.
- `(void)task(K k)` indicates what task `k` should be doing when running. Typically this is some computational routine followed by the trigger of other tasks. For instance task `k1` can fulfill one dependency of task `k2` by `tf.fulfill_promise(k2)`.
- `(int)mapping(K k)` indicates what thread should task `k` be initially mapped to.

In general, tasks can be stolen between threads to avoid starvation. This is done using a work stealing algorithm. `tf.set_binding(binding)` can be used to make some tasks bound to their thread. Optional priorities can also be provided through `tf.set_priority(priority)`. Finally, `tf.fulfill_promise(k)` is used to fulfill one of the dependencies of task `k` on Taskflow `tf`. See Figure 4.

2) Distributed memory components: Active Messages (AMs) are used to allow tasks on rank a to trigger tasks on rank $b \neq a$ without rank b explicitly waiting for messages.

An AM is a pair `(function, payload)`. When an AM is sent from rank a to rank b, the payload is sent through the network, and upon arrival, the function (with the associated payload passed as argument) is run on the receiver rank. This allows for instance to store the payload at some location in local memory and then trigger tasks.

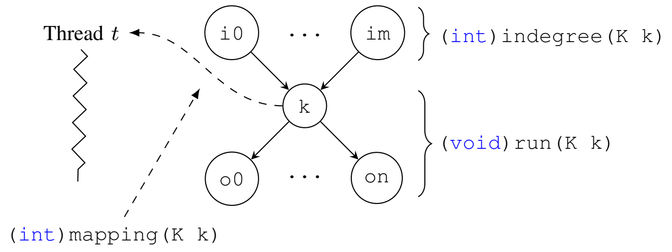


Fig. 4: The `Taskflow<K>` API. `(int)indegree(K k)` returns the number of incoming dependencies of task `k`. `(void)run(K k)` indicates what function to run. `(int)mapping(K k)` returns what thread the task should be mapped (but not bound) to.

a) Active message: An `ActiveMsg<Ps...> am` pairs a function `(void)fun(Ps... ps)` and a payload `ps`. Note that `Ps...` is a variadic template: different types can be used as arguments. A `view<T>` can be used to identify a memory buffer (i.e., a pointer and a length) and is built as `view<T> v(pointer, num_elements)`.

The AM can be sent to rank `dest` over the network using `am->send(dest, ps...)`. When sent, the payload is serialized on the sender, sent over the network, deserialized on the receiver and the function is run as `fun(ps...)`. The payloads are always serialized in a temporary buffer by the library. As such, the user-provided arguments can be immediately reused or modified as soon as `send` returns. `am->send` is thread-safe and can be called by any thread.

`TTor` also provides *large* active messages. A large AM can be used to avoid temporarily copying large buffers. A large AM payload is made of one `view<T>` and a series of arguments `Ps...`. The view will be sent and received directly without any extra copy. It is associated with three functions: (1) a function to be run on the receiver rank that returns a pointer to a user-allocated buffer, where the data will be stored; (2) a function to be run on the receiver rank to process the data upon arrival; (3) a function to be run on the sender rank when the buffer on the sender side can be reused. This is an important feature to avoid costly copies and/or when memory use is constrained.

b) Communicator: A `Communicator comm` is a C++ factory to create AMs and is responsible for sending, receiving and running AMs. `Communicator comm(mpi_comm)` creates a communicator using the `mpi_comm` MPI communicator. An AM can then be created by `am = comm.make_active_msg(f)` where `f` is a `(void)f(Ps...)` function. AMs always have to be created in the same order on all ranks because we need to create a consistent global indexing of all the AM that need to be run.

3) Example: The following shows how the different components can be used together. This assumes `compute(k)` does the computation related to task `k`. In addition, `mapping(k)` returns a thread for task `k` (which is typically `k % n_threads`), `n_deps(k)` gives its number of in-dependencies, `deps(k)` iterates through its out-dependencies and `task_2_rank(k)` returns the rank it is mapped to. `n_threads` is the desired number of threads to use. We assume that task outputs are stored in `data`. The execution of the DAG starts when the initial tasks are seeded and finishes when `tp.join()` returns.

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

B. Implementation Details

1) Taskflow and threadpool: The threadpool is implemented with two `std::priority_queue<Task*>` per thread, storing the ready-to-run tasks. Since some tasks can be stolen and others not, each thread has two queues. The priority queues are protected using `std::mutex` so that tasks can be inserted into a thread queue by any other thread.

One of the main goals of the `Taskflow<K>` implementation is to support arbitrary task flows with keys belonging to any domain. Hence, we store dependencies in a `std::unordered_map<K,int>`. Furthermore, to avoid having one central map storing all dependencies (whose access needs to be serialized), the map is distributed across threads. Task's dependencies are split among the threads using the mapping function: the dependency count of task `k` is stored in the map associated to thread `mapping(k)`. Each distributed map is always accessed by the same thread, preventing data races.

2) Active messages and communication thread: Active messages (AM) are implemented by registering functions on every rank in the same order. Each AM then has a unique ID shared across ranks. This ID is later used to retrieve the function on the receiver side.

Communication is performed using MPI non-blocking sends and receives. The `Communicator` maintains three queues:

1) a queue of serialized and ready-to-send messages;
2) a queue of send messages, to be later freed when the associated send completes;
3) a queue of receive messages, to be later run and freed when the associated receive completes.

On the sender side, when sending (thread-safe) an active message `am->send(dest, ps...)`, the various arguments `ps...` are first serialized into a buffer, along with the AM ID. The buffer is placed in a queue in the communicator. When calling `progress()`, that buffer will eventually be sent using `MPI_Isend` and later freed when the send has completed.

On the receiver side, calling `progress()` performs the following:

1) As long as it succeeds, it calls `MPI_Iprobe` to probe for incoming messages and (1) retrieves the message size using `MPI_Getcount`, (2) allocates a buffer and (3) receives the message using `MPI_Irecv`.
2) It goes through all received messages and tests for completion with `MPI_Test`. If it succeeds (1) it retrieves the AM using the ID from the buffer and (2) deserializes the buffer, passes the arguments to the user function and runs the user function.

MPI tags are used to distinguish (1) messages of size smaller or larger than 2^{31} bytes, and (2) regular and large AMs.

3) Distributed completion algorithm: We now discuss the distributed algorithm to determine completion. We present the algorithm along with a proof of correctness. The difficulty in detecting completion lies in the fact that even if all taskflows are idle, the program may not be finished since active messages (AM) may still be in-flight. An example of a flawed strategy is to request that all ranks send an `IDLE` signal to one rank when they have no tasks running. This strategy will lead to early termination of the program in many cases. Hence, detecting completion is non-trivial in a distributed setting.

a) Completion: In the following, we will consider a series of events such as queuing and processing messages, checking certain conditions, etc. Within a thread we assume a total ordering between events which lets us associate each of them with a unique real number which we informally call "time". We consider a program with two threads per rank: a main (MPI) thread responsible for MPI communication (asynchronous sends and receives) and AMs, and a worker thread responsible for executing all the user-defined tasks (in practice, the worker thread may be in fact a thread pool, but this is not relevant).

We say that an AM is **queued** on a sending rank when it is issued either by the worker or the main thread. When issued by a worker, we assume that queueing always finishes before the completion of the enclosing task. An AM is **processed** on the receiving rank by the main thread. We assume that if an AM results in a task being inserted in the task queue of the worker thread, this insertion must complete before the end of the enclosing AM.

To define our ordering between ranks, we assume that if a message is queued at time t and processed at time t' then

$t' > t$. We assume that messages that are queued are eventually processed if the network and all ranks are idle except for handling these messages (progress guarantee), and that all communications are non blocking (no deadlocks are possible). `TTor` satisfies those assumptions by construction.

Definition 1 (Completion): We say that $\{t_a\}_a$ is completion time sequence if:

- Rank a is idle at time t_a for all a;
- For any pair of ranks (a, b) and all AMs from a to b, all AMs queued before t_a have been processed on b before t_b.

One can prove (omitted here) that this definition implies the intuitive definition of completion, which is that, after t_a, if we keep the program running, rank a remains idle forever.

b) Completion algorithm: The algorithm is based on making sure, after all ranks are idle, that the number of messages sent is equal to the number of messages received. For this verification to work, we need to proceed in two steps, leading to the following definition:

Definition 2 (Synchronization time): Assume that for all ranks a, we have defined a pair of times (t_a^-, t_a^+) with $t_a^- < t_a^+$. We say that $\bar{t}$ is a synchronization time for (t_a^-, t_a^+) if

$$t_a^- < \bar{t} < t_a^+, \quad \text{for all } a$$

Before giving the exact algorithm, we prove a sufficient condition to establish completion.

Lemma 1: Let $p_a(t)$ (resp. $q_a(t)$) be the number of processed (resp. queued) AMs on rank a at time t. Assume that there exists a synchronization time $\bar{t}$ for (t_a^-, t_a^+) and that for all a

- the worker thread on rank a is idle at t_a^-;
- $\bar{p}_a = p_a(t_a^-) = p_a(t_a^+)$ (no new processed AM between t_a^- and t_a^+);
- $\bar{q}_a = q_a(t_a^-) = q_a(t_a^+)$ (no new queued AM between t_a^- and t_a^+);
- $\sum_a \bar{q}_a = \sum_a \bar{p}_a$.

Then the sequence $\{t_a^-\}_a$ is a completion time sequence for the execution.

Proof: Let us first prove that rank a is idle during the entire period $[t_a^-, t_a^+]$. Rank a is idle at t_a^-. Since $p_a(t_a^-) = p_a(t_a^+)$, no AM was processed at any time $t \in [t_a^-, t_a^+]$. So no tasks may have been inserted in the worker task queue by the main thread. Hence, rank a is idle during $[t_a^-, t_a^+]$.

Second, because $p_a(t_a^-) = p_a(t_a^+)$ and $q_a(t_a^-) = q_a(t_a^+)$, we necessarily have that

$$p_a(\bar{t}) = p_a(t_a^-) = p_a(t_a^+), \quad q_a(\bar{t}) = q_a(t_a^-) = q_a(t_a^+).$$

This is because p_a and q_a are increasing functions of time and $t_a^- < \bar{t} < t_a^+$. Therefore: $\sum_a \bar{q}_a(\bar{t}) = \sum_a \bar{p}_a(\bar{t})$. The key is that this is true at the synchronization time $\bar{t}$.

Consider now a message m that is contributing to $\sum_a \bar{q}_a(\bar{t})$ and $\sum_a \bar{p}_a(\bar{t})$. It is not possible that m contributes $+1$ to $\sum_a \bar{p}_a(\bar{t})$ (e.g., it has been counted as processed) while contributing 0 to $\sum_a \bar{q}_a(\bar{t})$ (e.g., it has not been counted as queued). This is because the process time is always strictly greater than the queuing time and we are evaluating the terms at the synchronization time $\bar{t}$.

From this, assume now that m contributes $+1$ to $\sum_a \bar{q}_a(\bar{t})$ (queued) and 0 to $\sum_a \bar{p}_a(\bar{t})$ (but not processed yet). Then we must have:

$$\sum_a \bar{q}_a(\bar{t}) > \sum_a \bar{p}_a(\bar{t})$$

This is because not other message m' can "restore" the equality. This inequality is a contradiction.

Therefore all messages queued have been processed. With the results above, $\{t_a^-\}$ is a completion time sequence. ■

We now describe the algorithm. Rank 0 will be responsible to detect completion by synchronizing ($\bar{t}$) with other ranks $r > 0$. **When a rank is idle**, the main thread on all ranks does the following.

1) All ranks r continuously monitor $q_r(t)$ and $p_r(t)$ (which only contain the user's AM count and not the messages used in the completion algorithm). If at a time t_r^- those values differ from the latest observed ones, rank r sends a message `COUNT` $= (r, q_r(t_r^-), p_r(t_r^-))$ to rank 0 with those updated counts.
2) Rank 0 continuously observes the latest received counts. Since $q_r(\cdot)$ and $p_r(\cdot)$ are non-decreasing it is enough to consider the greatest received counts and discard the others. If at time $\tilde{t}$ (implemented as an always increasing integer counter), $\sum_r q_r(t_r^-) = \sum_r p_r(t_r^-)$ and that sum is different from the latest observed sum, rank 0 sends a `REQUEST` $= (q_r(t_r^-), p_r(t_r^-), \tilde{t})$ message back to all ranks $r > 0$.
3) All ranks r continuously monitor the `REQUEST` messages from rank 0. They process the one with the largest $\tilde{t}$, and discard the others. At time t_r^+, if $q_r(t_r^-) = q_r(t_r^+)$ and $p_r(t_r^-) = p_r(t_r^+)$, they send a `CONFIRMATION` = $(\tilde{t})$ back to rank 0.
4) Rank 0 continuously observes the received `CONFIRMATION`. If all ranks replied with the latest $\tilde{t}$, the program has completed. Rank 0 then sends a `SHUTDOWN` message to all ranks.
5) All ranks r continuously listen to the `SHUTDOWN` message. When received, the program has completed and rank r terminates.

Note that although we write the algorithm as a sequence from 1 to 5, the word "continuously" indicates that this is implemented as a loop which keeps attempting to perform each step until `SHUTDOWN` is received.

We proved the following two theorems (proof is omitted but is based on the results described above, with the assumptions provided at the beginning).

Theorem 1 (Correctness): The `SHUTDOWN` message is sent if and only if completion has been reached.

The second property guarantees that `CONFIRMATION` is sent in finite time. For example, if the number of message is potentially unbounded, messages from some ranks could always be prioritized, preventing any progress from other ranks, and the algorithm may never terminate.

Theorem 2 (Finiteness): The completion protocol is guaranteed to send `CONFIRMATION` in finite time.

III. BENCHMARKS

In this section, we present benchmarks comparing `TTor` to OpenMP, StarPU, and ScaLAPACK.

We start with micro-benchmarks to validate the low overhead of the shared memory component. This is only used to verify that the task-based management overhead is comparable, and sometimes better, to other runtime systems.

We then apply `TTor` (with its distributed component) to two classical linear algebra problems. In those sections, the goal is to compare a sequential enumeration of the DAG (STF) as implemented in StarPU versus the PTG approach as implemented in `TTor`. We note in particular that it is possible to modify the StarPU code such that the DAG is parallelized in a manner close to `TTor`. Similarly several optimizations in `TTor` are possible but were not explored for this paper (memory management, task insertion, communication). Therefore, these benchmarks cannot be interpreted as measuring the peak performance of either runtime.

In all cases, experiments are run on a cluster equipped with dual-sockets and 16 cores Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz with 32GB of RAM per node. Intel Compiler (`icpc (ICC) 19.1.0.166 20191121`) and Intel MPI are used with Intel MKL (version `2020.0.166`) for BLAS, LAPACK and ScaLAPACK. We use StarPU version `1.3.2`. We assign one MPI rank per node. `TTor`'s code, including benchmarks, is available at `github.com/leopoldcambier/tasktorrent`. StarPU and ScaLAPACK's benchmarks are available at `github.com/leopoldcambier/tasktorrent_paper_benchmarks`.

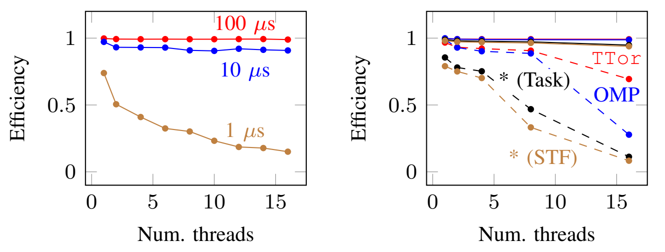
A. Micro-benchmarks

We first perform a series of micro benchmarks to validate the low overhead of the shared memory component of the runtime. In the following, we average timings across 25 runs. In every case, the standard deviation was recorded as well, to estimate the variability of the measurement. In most cases, it was negligible and we don't report it. In all cases, we pick a number of tasks so that the total runtime is about 1 second.

1) No-dependencies overhead: We begin with an estimation of the "serial" overhead of `TTor`'s shared memory runtime. We start `ntasks` tasks, without any dependencies, and assign them in a round-robin fashion to the `nthreads` threads. Each task is only spinning for `spin_time` seconds. As such, the total ideal time is `spin_time` × `ntasks` / `nthreads`. Figure 5 shows the efficiency as a function of `nthreads` and `spin_time`. Given a total wall clock time of `run_time`, efficiency is defined as `run_time` × `nthreads` / (`spin_time` × `ntasks`). `ntasks` is chosen so that `run_time` is around 2 seconds.

Figure 5a shows results for `TTor`'s only, where we do **not** measure task insertion, i.e., we evaluate

```
for(int k = 0; k < n_tasks; k++) {
  tf.fulfill_promise(k);
}
tp.start(); // Start measuring time
tp.join();  // Stop measuring time
```



(a) TTOR's overhead (no dependencies) measurement. Task insertion time is not included. Numbers indicate `spin_time`.

(b) Overhead (no dependencies) comparisons. Task insertion time is included. Solid is `spin_time` = 100 μs; dashed is 10 μs; * is StarPU with direct task insertion (Task) and STF semantics (STF).

Fig. 5: Shared memory serial overhead, as a function of the number of threads `nthreads` (x-axis) and the task time `spin_time` (various lines). The plots show the mean across 25 runs.

We see that the runtime has negligible impact for tasks $\approx 100\mu$s, and it becomes significant around 1 μs where overhead dominates.

We then compare it to OpenMP and StarPU in Figure 5b where, to make the comparison fair, insertion time **is** measured (which reduces the maximum possible efficiency, as the insertion is sequential).

```
tp.start(); // Start measuring time
for(int k = 0; k < n_tasks; k++) {
  tf.fulfill_promise(k);
}
tp.join();  // Stop measuring time
```

We note that this is a spurious consequence of creating tasks with no dependencies. In practice the insertion is done by other tasks, themselves executing in parallel. We evaluate StarPU both using "direct" task insertion ("Task"), as well as using the STF approach ("STF"). In the STF approach, each independent task is associated with an artificial independent read-write piece of data. We see that for very small tasks $< 10\mu s$, overhead is significant but comparable for all runtimes.

2) Many dependencies overhead: We then estimate the overhead when dependencies are involved. Consider a 2D array of `nrows` × `ncols` tasks, with `ndeps` dependencies between task (i, j) and $((i + k)\%\texttt{nrows}, j + 1)$ for $0 \leq k <$ `ndeps`. Again, tasks are spinning for `spin_time` seconds and, in `TTor`, task (i, j) is assigned to thread $i\%$ `nthreads`.

Since this is not easily implementable in OpenMP, we only compare `TTor` with StarPU. In the "Task" version, tasks are directly inserted, and their dependencies are explicitly expressed. In the STF approach, we register data for every (i, j) task and that data is used to create dependencies with the tasks in the next column. We note that StarPU STF has the constraint that the number of input data buffers for a given

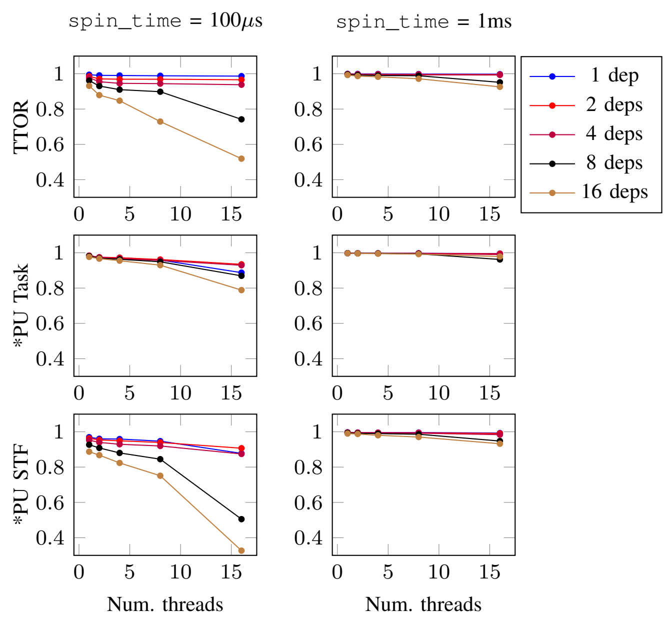


Fig. 6: Efficiency vs. number of threads. Shared memory runtime dependency management overhead. The plots show the mean across 25 runs.

task should normally be known at compile time, which makes it not well-suited for this benchmark.

Figure 6 shows the results with `nrows` set to 32. We see that `TTor` is between StarPU "Task" and StarPU "STF", with similar overhead. This validates the implementation.

The conclusion of this section is that the overhead of `TTor` is comparable (and sometimes better) to OpenMP and StarPU.

B. *Distributed Matrix-matrix Product*

We now consider a distributed matrix-matrix multiplication problem (GEMM), i.e., given $A, B \in \mathbb{R}^{N\times N}$ compute $C = AB$. We compare:

- `TTor` with an algorithm using a 2D block cyclic mapping of blocks of size 256 to ranks, using the default ("small") and large AMs;
- `TTor` with an algorithm using a 3D mapping of blocks to ranks, tiled (every GEMM is single threaded, with a block size of 256) or not (every GEMM is a single large multithreaded BLAS). We use the DNS algorithm (see for instance [4]) to map blocks to ranks.
- StarPU (with STF semantics, i.e., all ranks explore the full DAG) using a 2D block cyclic mapping of blocks of size 256 to ranks. Various scheduling strategies have been tried, without significant variation in runtime; the default local work stealing `lws` is then used.
- ScaLAPACK using a 2D block cyclic mapping (with a block size of 256) with multithreaded BLAS. We note that ScaLAPACK is not a runtime and is not actively managing a task graph.

The following code snippet shows the GEMM portion when using the 2D block cyclic data distribution. In this case, contributions $A_{ik}B_{kj}$ are ordered as function of k, i.e., $A_{ik}B_{kj}$ happens before $A_{i(k+1)}B_{(k+1)j}$. Furthermore, because of the 2D data distribution, the products $A_{ik}B_{kj}$ are mapped to a rank function of (i, j) only and, as such, always happen on a same node. The mapping of tasks to thread may be any deterministic function of `ikj`. In practice something as simple as `ikj[0] % n_threads` can be used without any visible performance degradation. It is merely used to distribute task dependency management evenly across threads. `noalias()` is from the linear algebra library Eigen.

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

Figure 7 presents strong and weak scalings results. Scalings are done multiplying the number of rows and columns by 2 and/or the number of nodes by 8, and the largest test case are matrices of size 32 768. We make multiple observations:

- `TTor` benefits from the large messages (Figure 7c) over small ones, decreasing the total time by up to 30%.
- `TTor` with large messages and StarPU using the 2D mapping have similar performance (Figure 7c vs Figure 7e). `TTor` performs better than StarPU with small blocks (Figure 7g).
- `TTor` with the 3D mapping and the tiled algorithm has better performance than without (see Figure 7d as well as Figure 7a vs Figure 7b for results on 8 nodes). This shows the importance of having a small task granularity, to increase overlap between communication and computation. It has however similar performance to the 2D mapping.
- Runtime-based implementations outperform ScaLAPACK (Figure 7f), showing the benefits of a task-based runtime system.

Figure 7g shows the impact of the block size on the runtime. We see that `TTor` is about 2.5x faster than StarPU at small sizes. This highlights the advantages of a distributed DAG exploration. We note that in this case small blocks are not optimal. However, GEMM is in some sense an "easy" benchmark since it offers a large amount of concurrency. Therefore, to stress the runtimes and observe measurable differences we need to deviate from the optimal GEMM settings. Although we could not investigate other algorithms for this paper, more complex applications would probably reveal additional differences between `TTor` and StarPU.

Finally, Figure 7h shows the efficiency of `TTor` (2D GEMM) as a function of the concurrency. Since the GEMMs are sequential as a function of k, `num_blocks^2/n_cores` indicates how much parallelism is available per core. This represents the number of blocks that are processed on each core between communication steps. We see that efficiency decreases sharply at around 16 blocks per core.

C. Distributed dense cholesky factorization

We now consider an implementation of the Cholesky algorithm, i.e., given a symmetric positive definite matrix $A \in \mathbb{R}^{N\times N}$, compute L such that $A = LL^\top$. In its sequential and blocked form, the algorithm is described in Algorithm 1.

Algorithm 1

```
1: procedure CHOLESKY($A$, $n$)                ▷ $A \succ 0$, $n \times n$ blocks
2:     for $1 \le k \le n$ do
3:         $L_{kk}L_{kk}^\top = A_{kk}$                   ▷ potrf($k$)
4:         for $k+1 \le i \le n$ do
5:             $L_{ik} = A_{ik}L_{kk}^{-\top}$              ▷ trsm($i, k$)
6:             for $k+1 \le j \le i$ do
7:                 $A_{ij} \leftarrow A_{ij} - L_{ik}L_{jk}^\top$     ▷ gemm($k, i, j$)
```

The algorithm is made of three main computational routines: potrf(k), trsm(i, k) and gemm(k, i, j) (in practice syrk when $i = j$). We show a PTG formulation of Algorithm 1 in Figure 8. Large active messages are used.

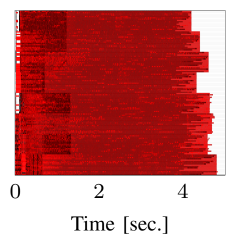
We compare TTor, StarPU (with STF semantics) and ScaLAPACK. A 2D block cyclic data distribution is used with a block size of 256. Task priorities in TTor are computed using [5]. As before, in ScaLAPACK the block size is related to the data distribution but there are no tasks per se.

Weak and strong scalings are performed by multiplying the number of rows and columns by 2 or the number of cores by 8. The larger test case is a matrix of size $N = 131\,072$. Figure 9 shows the results.

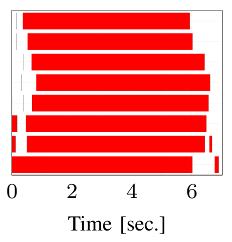
We see that on large problems, both TTor and StarPU reach very similar performances, both outperforming ScaLAPACK by far: for $N = 131\,072$ on 1024 cores, ScaLAPACK takes more than 125 secs (not shown). On the $N = 131\,072$ test case, TTor and StarPU differ by less than 10%. StarPU shows better strong scaling for small problems on many nodes. We conjecture that this may be due to a better task scheduler, memory management (thread-memory affinity), and mapping of the computation across nodes.

Figure 9d shows the runtime as a function of the block size for a test case of size $65\,536 \times 65\,536$ on 64 nodes (1024 CPUs). We see that 256 gives the best results for both TTor and StarPU. Furthermore, we observe that for small task size, TTor degrades less quickly than StarPU. The small block size leads to many tasks and unrolling the DAG on one node becomes prohibitive, even for reasonably large tasks (block size of 128). For a block size of 64, TTor is about 10x faster. Thanks to its lightweight runtime and distributed DAG exploration, TTor suffers less from the small task size. For large task sizes, both degrade similarly. The poor performance at large size is caused by a lack of concurrency.

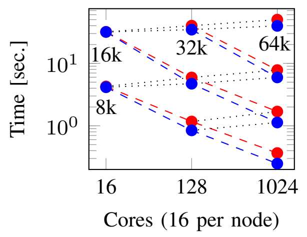
Figure 9e shows a load balancing test using random block sizes with a fixed number of blocks. ρ is the ratio of the largest



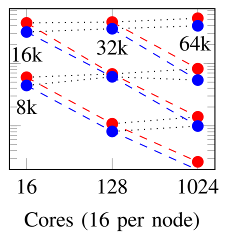
(a) Tiled 3D DNS GEMM trace for 8 nodes. Every red line is 1 TTor thread using single-threaded BLAS.



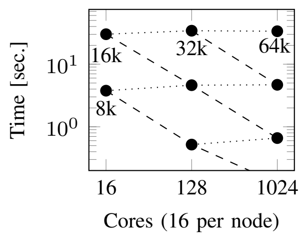
(b) Non-tiled 3D DNS GEMM trace for 8 nodes. Every red line is 1 TTor thread using multithreaded BLAS.



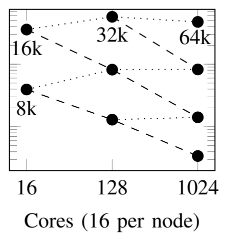
(c) TTor 2D GEMM. Red = small AMs, blue = large AMs.



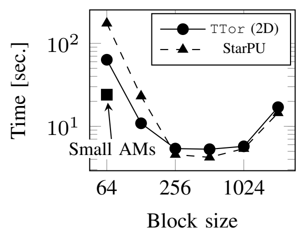
(d) TTor 3D GEMM. Red = non-tiled, blue = tiled.



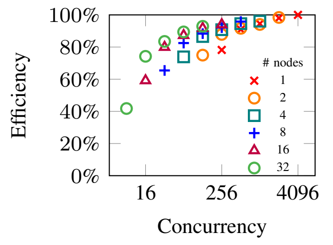
(e) StarPU 2D GEMM.



(f) ScaLAPACK GEMM.

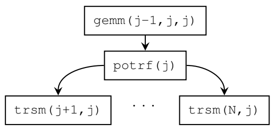


(g) Block size impact, $N = 32\,768$ with 1024 CPUs.

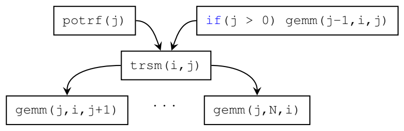


(h) TTor 2D GEMM. Concurrency = num_blocks^2/n_cores, $N = 16\,384$.

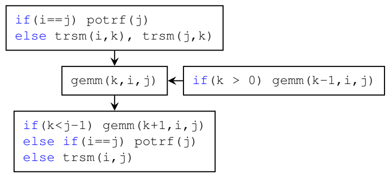
Fig. 7: GEMM scalings. (a-b): impact of task granularity on 3D GEMM. Smaller tasks give higher overlap of computation and communication. (c-f): weak (dotted) and strong (dashed) scalings. Numbers indicate the matrix size N. Largest test case is $N = 65\,536$. (g): optimal block size (i.e., task granularity) for the $N = 32\,768$ test case. The extra data point shows the improvement when using small AMs instead of large AMs on small block sizes. The decrease in the number of messages sent improves the runtime by 3x. (h): efficiency as a function of concurrency for $N = 16\,384$. Reference timing is with 1 core.



(a) `potrf(j)`



(b) `trsm(i,j)`



(c) `gemm(k,i,j)`

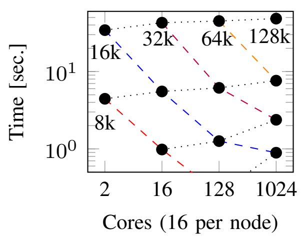
Fig. 8: PTG description of Algorithm 1. In `TTor`, when out-dependencies are remote, an AM is sent to the remote rank, carrying the associated block and triggering remote tasks.

over the average block size. For $\rho = 1.5$, the ratio of flops from smallest to largest task is $(1.5/0.5)^3 = 27$. We see that `TTor` handles tasks of various granularity very well, with less than 25% degradation from $\rho = 1$ to $\rho = 2$ for an average block size of 256.

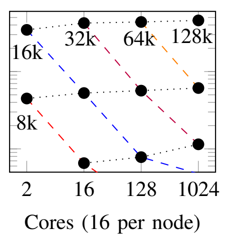
IV. Previous work

a) Runtime systems: As mentioned in Section I-B, other task-based runtime systems exist. We highlight some of their characteristics. PaRSEC [3] is a runtime system centered around dense linear algebra. It takes the PTG approach but uses a custom programming language, the JDF. This can make adoption harder for new users. Legion [1] is a general purpose STF runtime. It has many features and can be used from C++ but requires the user to express everything using Legion's data structures. It is also intended to be used primarily with GASNet [6] and not MPI. Regent [7] proposes a higher level language on top of Legion, making programming more productive. Unfortunately, obtaining high performance requires the user to program directly the mapper which is time-consuming and requires a detailed understanding of the inner workings of Legion. Finally, StarPU [2] uses C++ and is STF-based. The data is initially distributed by the user like a classical MPI code, and various scheduling strategies can be used to further improve performance. However, user data still has to be wrapped using StarPU's data structures.

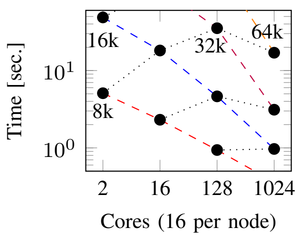
In designing `TTor` we chose to focus on the following features. The message passing paradigm requires the programmer to distribute data but simplifies the design of the library with the goal of minimizing global synchronization and communication. MPI and C++ makes integration into other codes easier. Active messages are necessary because of the asynchronous nature of computations. Finally the PTG approach leads to a minimal runtime overhead. Note however that the choice of PTG has drawbacks: it can be difficult for the programmer to reason about tasks dependencies. This can be easier in some applications (like linear algebra) than others.



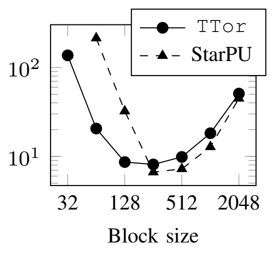
(a) TaskTorrent.



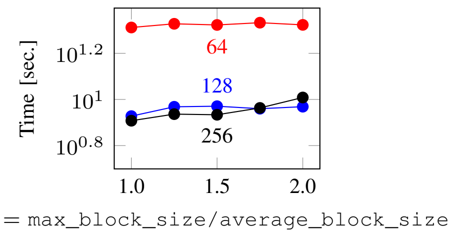
(b) StarPU



(c) ScaLAPACK



(d) Block size impact, $N = 65\,536$ with 1024 CPUs.



(e) Load balancing test with random block sizes. $N = 65\,536$ with 1024 CPUs. Block sizes are random uniform on $((2-\rho)b, \rho b)$ with b the maximum block size. Numbers indicate the average block size.

Fig. 9: Cholesky scalings. (a-c): weak (dotted) and strong (dashed) scalings. Numbers indicate the matrix size N. Largest (top right) test case is $N = 131\,072$. (d): optimal block size (i.e., task granularity) for the $N = 65\,536$ test case. (e): load balancing test using random block sizes for the $N = 65\,536$ test case.

`TTor` also does not consider concepts like memory affinity or accelerators at the moment. This is reserved for future work.

b) Task-based parallelism: Task-based parallelism is now a common feature of many parallel programming systems.

Cilk [8], [9] introduced a multi-threading component to C in 1996, and Cilk-5 introduced `spawn` and asynchronous computations. Many other efforts followed, including OpenMP [10] (with tasking introduced in version 3.0), Intel TBB [11] (where task DAGs can be expressed), Cilk Plus [12], XKaapi [13], OmpSs [14], Superglue [15], and the SMPSs programming model [16], [17]. The Plasma [18], [19] (for CPU) and Magma [20] (for CPU and GPU) libraries are replacements for multithreaded LAPACK, where parallelism is obtained through tiled algorithms using a dynamic runtime, Quark [21].

Notice that all the previously mentioned work is typically only usable in a shared-memory context. In particular, there is no support to let one rank trigger (or fulfill the dependency of) a task on another rank.

c) Distributed programming: An explicit goal of `TTor` is to provide support for distributed computing.

The most common distributed programming paradigm is using explicit message passing like in MPI. In MPI, ranks are completely independent and only communicate with each other through explicit message passing. Charm++ [22] takes an object-oriented approach. It exposes *chares* which are concurrent objects communicating through messages. We also mention `DARMA/vt` [23], a tasking and active message library in C++, with other features such as load balancing and asynchronous collectives. Finally, in the PGAS (partitioned global address space) model (like GASNet [6]), each rank can access a global address space through read (get) and write (put) operations. Chapel [24], Fortran Co-arrays [25], UPC [26] and UPC++ [27] are examples of PGAS-based parallel programming languages.

d) Active messages: One-sided active messages is another important feature of TaskTorrent. Von Eicken et al. [28] argued in 1992 that active messages are a powerful mechanism to hide latency and improve performance. Active messages are also a central part of UPC++ where they resemble the ones in `TTor`. In UPC++, however, remote data is referred to using global data structures, while `TTor` tends to use the C++ variable capture mechanism in lambda functions.

V. Conclusion

We presented TaskTorrent (`TTor`), a lightweight distributed task-based runtime system in C++. It has a friendly API, and rely on readily available tools (C++14 and MPI). It enables shared-memory task-based parallelism coupled with one-sided active messages. Those two concepts naturally work together to create a distributed task-based parallel computing framework. We showed that `TTor` is competitive with both StarPU (a state of the art runtime) and ScaLAPACK on large problems. Its lightweight nature allows it to be more forgiving when task granularity is not optimal, which is key to integrating this approach in legacy codes.

References

[1] M. E. Bauer, "Legion: Programming distributed heterogeneous architectures with logical regions," Ph.D. dissertation, Stanford University, 2014.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[3] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *Computing in Science Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[4] A. Grama, V. Kumar, A. Gupta, and G. Karypis, *Introduction to parallel computing*. Pearson Education, 2003.

[5] O. Beaumont, J. Langou, W. Quach, and A. Shilova, "A makespan lower bound for the scheduling of the tiled cholesky factorization based on alap schedule," 2020.

[6] D. Bonachea and P. Hargrove, "GASNet specification, v1. 8.1," Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), Tech. Rep., 2017.

[7] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: a high-productivity programming language for HPC with logical regions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[8] C. F. Joerg, "The Cilk system for parallel multithreaded computing," Ph.D. dissertation, Massachusetts Institute of Technology, 1996.

[9] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, 1998, pp. 212–223.

[10] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[11] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.

[12] A. D. Robison, "Composable parallel patterns with Intel Cilk Plus," *Computing in Science & Engineering*, vol. 15, no. 2, pp. 66–71, 2013.

[13] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 1299–1308.

[14] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel processing letters*, vol. 21, no. 02, pp. 173–193, 2011.

[15] M. Tillenius, "Superglue: A shared memory framework using data versioning for dependency-aware task-based parallelization," *SIAM Journal on Scientific Computing*, vol. 37, no. 6, pp. C617–C642, 2015.

[16] J. M. Perez, R. M. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *2008 IEEE International Conference on Cluster Computing*. IEEE, 2008, pp. 142–151.

[17] ——, "Handling task dependencies under strided and aliased references," in *Proceedings of the 24th ACM International Conference on Supercomputing*, 2010, pp. 263–274.

[18] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The plasma and magma projects," in *Journal of Physics: Conference Series*, vol. 180. IOP Publishing, 2009, p. 012037.

[19] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan, "Plasma users guide," Technical report, ICL, UTK, Tech. Rep., 2009.

[20] S. Tomov, J. Dongarra, V. Volkov, and J. Demmel, "Magma library," *Univ. of Tennessee and Univ. of California, Knoxville, TN, and Berkeley, CA*, 2009.

[21] A. YarKhan, J. Kurzak, and J. Dongarra, "Quark users' guide: Queueing and runtime for kernels," *University of Tennessee Innovative Computing Laboratory Technical Report ICL-UT-11-02*, 2011.

[22] L. V. Kale and S. Krishnan, "Charm++: a portable concurrent object oriented system based on c++," in *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, 1993, pp. 91–108.

[23] J. J. Lifflander and P. P. Pebay, "DARMA/vt FY20 mid-year status report." Sandia National Lab. (SNL-CA), Livermore, CA (United States), Tech. Rep., 2020.

[24] D. Callahan, B. L. Chamberlain, and H. P. Zima, "The cascade high productivity language," in *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.* IEEE, 2004, pp. 52–60.

[25] R. W. Numrich and J. Reid, "Co-Array Fortran for parallel programming," in *ACM Sigplan Fortran Forum*, vol. 17. ACM New York, NY, USA, 1998, pp. 1–31.

[26] T. El-Ghazawi and L. Smith, "UPC: unified parallel C," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006, pp. 27–es.

[27] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "UPC++: a PGAS extension for C++," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium.* IEEE, 2014, pp. 1105–1114.

[28] T. Von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," *ACM SIGARCH Computer Architecture News*, vol. 20, no. 2, pp. 256–266, 1992.

Appendix

A. Paper Artifact Description / Article Evaluation (AD/AE) Appendix

Are there computational artifacts such as datasets, software, or hardware associated with this paper? Yes.

B. AD/AE Details

Experiments were run on a Stanford University HPC cluster equipped with dual-sockets and 16 cores `Intel(R)Xeon(R)CPU E5-2670 0 @ 2.60GHz` with 32GB of RAM per node. Intel Compiler (icpc (ICC) `19.1.0.166 20191121`) and Intel MPI are used with Intel MKL (version `2020.0.166`) for BLAS, LAPACK and ScaLAPACK. We use StarPU version `1.3.2`.

1) Artifacts Available (AA):

- Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.
- Hardware Artifact Availability: There are no author-created hardware artifacts.
- Data Artifact Availability: There are no author-created data artifacts.
- Proprietary Artifacts: There are associated proprietary artifacts that are not created by the authors. Some author-created artifacts are proprietary.

2) Author artifacts:

- Artifact 1: TaskTorrent repository, `github.com/leopoldcambier/tasktorrent`
- Artifact 2: TaskTorrent paper Scalapack and StarPU benchmarks repository, `github.com/leopoldcambier/tasktorrent_paper_benchmarks`

3) Experimental setup:

- Relevant hardware: Dual socket `Intel(R)Xeon(R)CPU E5-2670 0 @ 2.60GHz` with 16 cores and 32GB of RAM per node.
- Operating systems and versions: Linux kernel `3.10.0-693.el7.x86_64`
- Compilers and versions: Intel Compiler (icpc (ICC) `19.1.0.166 20191121`)
- Applications and versions: N/A
- Libraries and versions: Intel MPI version `2018.2.199`, Intel MKL version `19.1.0.166`, StarPU version `1.3.2`
- Key algorithms: N/A
- Input datasets and versions: N/A
- Optional link (URL) to output from commands that gather execution environment information: `stanford.edu/~lcambier/tasktorrent_paper/AD_AE.txt`

C. Artifact Evaluation

Are you completing an Artifact Evaluation (AE) Appendix? No.