Chapel-on-X: Exploring Tasking Runtimes for PGAS Languages

Akihiro Hayashi
Rice University
Houston, Texas
ahayashi@rice.edu

Sri Raj Paul
Rice University
Houston, Texas
sriraj@rice.edu

Max Grossman
Rice University
Houston, Texas
jmg3@rice.edu

Jun Shirako
Rice University
Houston, Texas
shirako@rice.edu

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

ABSTRACT

With the shift to exascale computer systems, the importance of productive programming models for distributed systems is increasing. Partitioned Global Address Space (PGAS) programming models aim to reduce the complexity of writing distributed-memory parallel programs by introducing global operations on distributed arrays, distributed task parallelism, directed synchronization, and mutual exclusion. However, a key challenge in the application of PGAS programming models is the improvement of compilers and runtime systems. In particular, one open question is how runtime systems meet the requirement of exascale systems, where a large number of asynchronous tasks are executed.

While there are various tasking runtimes such as Qthreads, OCR, and HClib, there is no existing comparative study on PGAS tasking/threading runtime systems. To explore runtime systems for PGAS programming languages, we have implemented OCR-based and HClib-based Chapel runtimes and evaluated them with an initial focus on tasking and synchronization implementations. The results show that our OCR and HClib-based implementations can improve the performance of PGAS programs compared to the existing Qthreads backend of Chapel.

CCS CONCEPTS

• **Software and its engineering → Parallel programming languages**; **Distributed programming languages**; **Runtime environments**;

KEYWORDS

PGAS languages, Chapel, Task Parallelism, Runtime Systems, Qthreads, Open Community Runtime, Habanero-C

ACM Reference Format:
Akihiro Hayashi, Sri Raj Paul, Max Grossman, Jun Shirako, and Vivek Sarkar. 2017. Chapel-on-X: Exploring Tasking Runtimes for PGAS Languages. In *ESPM2'17: ESPM2'17: Third International Workshop on Extreme Scale Programming Models and Middleware, November 12–17, 2017, Denver, CO, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3152041.3152086

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESPM2'17, November 12–17, 2017, Denver, CO, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5133-1/17/11…$15.00
https://doi.org/10.1145/3152041.3152086

1 INTRODUCTION

While conventional message-passing programming models such as MPI [13] can greatly accelerate distributed-memory programs, the tuning effort required with message-passing APIs imposes additional burden on programmers. To improve software productivity and portability, a more efficient approach would be to provide a high-level programming model for distributed systems.

PGAS (Partitioned Global Address Space) programming languages such as Chapel, Co-array Fortran, Habanero-C, Unified Parallel C (UPC), UPC++, and X10 [5, 7, 8, 11, 21, 26] are examples of highly productive programming models. PGAS programming languages aim to reduce the complexity of writing distributed-memory parallel programs by introducing a set of high-level parallel language constructs that support globally accessible data, task parallelism, synchronization, and mutual exclusion.

A key challenge in the development and use of PGAS programming models is the improvement of compilers and runtime systems. Because PGAS languages can be dynamic or used for dynamic applications, it is likely that a large number of asynchronous tasks are executed. Hence, the tasking and threading mechanisms used by PGAS runtime systems are essential components that greatly affect performance. Important features of these runtime systems in existing literature include lightweight task creation/termination [25], efficient synchronization [23], and efficient task scheduling including work-stealing [3, 17].

There have been several tasking runtime systems designed for PGAS languages. For example, Qthreads [25] is a threading library for spawning and managing lightweight threads, which has been used for Chapel's tasking runtime over the years. The Open Community Runtime (OCR) [19] was designed to meet the needs of extreme-scale computing through an event-driven programming model with event-driven tasks (EDTs) and data blocks. HClib is a library-based tasking runtime and API, which is semantically derived from X10 [7] and focuses on lightweight task creation/termination and flexible synchronization.

While several tasking and threading runtimes have been designed for or co-designed with PGAS programming models, there is no comparative study on PGAS tasking/threading runtime systems using modern PGAS applications. To study and explore future runtime systems for PGAS languages, we have implemented

```
1   // begin construct
2   begin {
3     task(); // spawns a task executing task()
4   }
5   // cobegin construct
6   cobegin {
7     taskA(); // spawns a task executing taskA()
8     taskB(); // spawns a task executing taskB()
9   }
10  // coforall construct
11  coforall i in 1..N {
12    // spawns a separate task for each iteration
13    task(i);
14  }
15  // forall construct
16  forall i in 1..N {
17    // may use an arbitrary number of tasks
18    task(i);
19  }
```

Figure 1: Task parallelism constructs in Chapel.

OCR-based and HClib-based tasking/threading Chapel runtimes and conducted performance evaluations using various Chapel programs.

This paper makes the following contributions:

(1) Implementation of new tasking/threading runtime systems for Chapel using the following runtimes:
  - Open Community Runtime: An asynchronous event-driven runtime.
  - Habanero C/C++ Library (HClib): A compiler-free lightweight tasking runtime.

(2) Performance evaluations and analyses using numerical computing, graph analytics, physical simulation, and machine learning applications written in Chapel.

2 CHAPEL LANGUAGE

2.1 Chapel Overview

Chapel is an object-oriented PGAS language developed by Cray Inc. Development of the Chapel language was initiated as part of the DARPA High Productivity Computing Systems program (HPCS). The HPCS program sponsored new work in highly productive languages for next-generation supercomputers. This section briefly summarizes key features of the Chapel language, compiler, and runtime.

2.2 Chapel Language Features

Chapel is classified as an APGAS (Asynchronous + PGAS) programming language, where each node can run multiple tasks in parallel and create new local or remote tasks. Also, Chapel supports multiple parallel programming paradigms including a `global-view` model and a `local-view` model. Thus, programmers can choose their programming model depending on their situation.

```
1   var sy$: sync int; // value = 0, state = empty
2   begin {
3     // 1. blocked until the state of sy$ is full
4     // 2. read the value of sy$
5     // 3. the state of sy$ is set to empty
6     var sy = sy$; // equivalent to sy$.readFE();
7     writeln("new task spawned");
8     writeln("sy = ", sy);
9     ...
10  }
11  // 1. blocked until the state of sy$ is empty
12  // 2. write 1 to the value of sy$
13  // 3. the state of sy$ is set to FULL
14  sy$ = 1; // equivalent to sy$.writeEF(1);
```

Figure 2: Sync variables in Chapel.

Dynamic Task Creation: Chapel has several parallel constructs related to dynamic lightweight task creation. The list below summarizes those constructs. Figure 1 illustrates examples of their use.

- `begin`: spawns a task running independently from the main thread of execution. (Line 2-4 in Figure 1)
- `cobegin`: spawns a block of tasks, one for each statement. The current (main) thread is blocked until all the tasks within the `cobegin` are complete. (Line 6-9 in Figure 1)
- `coforall`: spawns a separate task for each iteration. The current (main) thread is blocked until every iteration is complete. (Line 11-13 in in Figure 1)
- `forall`: similar to `coforall`, but Chapel may choose to use an arbitrary number of tasks to execute the loop (e.g., by loop chunking). (Line 16-19 in Figure 1)

Synchronization: Chapel uses synchronization variables (sync variables) to support flexible synchronization between tasks. A sync variable has a logical state and a value. The logical state can be either *full* or *empty*. When writing/reading a sync variable, the execution can be blocked depending on the state of the sync variable. For example, the read of `sy$` (Line 6 in Figure 2) is blocked until the state is set to *full* (Line 14 in Figure 2). Then, the state of `sy$` is set to *empty* after the read. Conversely, the write of `sy$` (Line 14 in Figure 2) is blocked until the state is *empty*. In this case, note that the initial state of `sy$` is *empty*. After the write, the state is set to *full*, resulting in unblocking the read of `sy$` in Line 6. It is worth noting that the name of sync variables ends in $ by convention.

Normal reads and writes of `sy$` are equivalent to `sy$.readFE()` and `sy$.writeEF()` respectively. These functions are defined in the Chapel sync variable API, which offers more control in operating on sync variables. For example, `sy$.readEF()` is blocked until the state is *empty*, and the state is set to $full$ after the read. The sync variable APIs also support `sy$.writeFE()`, `sy$.readFF()`, `sy$.writeFF()` and so on. More details can be found in the Chapel specification [5].

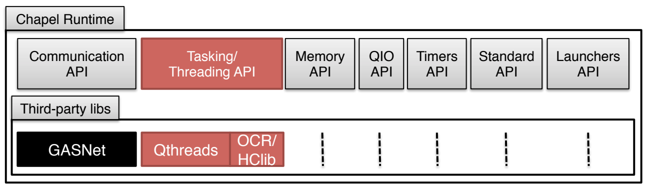

Figure 3: Chapel Runtime API

```
// Chapel code with the begin construct
begin {
  task(); // spawns a task executing task()
}

// C code generated by the Chapel compiler
void task() { ... };
fid = ...; // get a function id of task()
chpl_task_addToTaskList(fid, ...);

// Chapel runtime: Chapel Tasking API
void chpl_task_addToTaskList(fid, ...) {
  // getting a function pointer to the task
  chpl_fn_p fptr = chpl_ftable[fid];
  if (serial_state == true) {
    //directly invoke the function
    fptr(...)
  } else {
    // spawn a task
    spawn(fptr) // Discussed in Section 4
  }
}
```

Figure 4: Code generation and runtime implementation for `begin`

3 CHAPEL'S TASKING LAYER

3.1 Chapel Compiler and Runtime

The Chapel compiler is written in C++ and generates C code which can then be compiled by any C compiler. The C code contains API function calls defined in the Chapel runtime for enabling communication, dynamic tasking, memory allocation, I/O, and so on (Figure 3). In the runtime, those API functions are actually wrapper calls to third-party libraries, enabling users to choose between concrete implementations of runtime capabilities by setting environment variables depending on their configurations and platforms. This also helps runtime designers to integrate a new library into the Chapel runtime.

3.2 Chapel Tasking/Threading API

For dynamic tasking and synchronization between tasks, the Chapel runtime defines 9 synchronization functions, 23 tasking functions, and 5 threading functions. In other words, adding a new tasking model requires implementing these functions on a new tasking/threading library. Table 1 summarizes the important Chapel Tasking/Threading API functions. This section provides a brief overview of dynamic task creation and synchronization using code examples (Figure 4 and Figure 5). More detailed discussions of the implementation of these APIs with Qthreads, OCR, and HClib can be found in Section 4.

```
// Chapel code with sync variables
var sy$: sync int;
begin { var sy = sy$; ... }
sy$ = 1;

// C code generated by the Chapel compiler
void task() {
  chpl_sync_waitFullAndLock();
  int sy = sy$;
  chpl_sync_markAndSignalEmpty();
}
fid = ...; // get a function id of task()
chpl_task_addToTaskList(fid, ...); // creating a task

chpl_sync_waitEmptyAndLock();
sy$ = 1;
chpl_sync_markAndSignalFull();

// Chapel runtime:
// Discussed in Section 4
```

Figure 5: Code generation and runtime implementation for sync variables.

Dynamic Task Creation: The Chapel runtime firstly creates a main task that runs the compiler-generated main function of a Chapel program. Then, the main task dynamically creates asynchronous tasks as it encounters Chapel tasking/threading API functions derived from `begin`, `cobegin`, `coforall`, and `forall` constructs. Based on our profiling, we have identified that the following two API functions are essential:

- `chpl_task_callMain()`: Create a task that runs the compiler-generated main function and then execute it.
- `chpl_task_addToTaskList()`: Create a task and execute it.

Figure 4 shows code generation and Chapel Tasking API implementations for the `begin` construct. The `begin` construct is compiled to the `chpl_task_addToTaskList()` API and passed the ID of the function being executed as an asynchronous task. In the runtime, `chpl_task_addToTaskList()` first gets a function pointer to the specified function. Based on whether `serial` execution is enabled, this API executes the task either synchronously by directly invoking the function-pointer or asynchronously by spawning a task. When `serial` execution is not enabled, it packs all the required parameters into a `struct` and passes it to the spawned task.

Kind	API	Description
Task	`chpl_task_init();`	Call before executing the `main` function (Initialization).
	`chpl_task_callMain();`	Create a task that runs `main` and then execute it.
	`chpl_task_exit();`	Called when exiting.
	`chpl_task_yield();`	Yield the execution to another thread (e.g., by calling `sched_yield()`).
	`chpl_task_addToTaskList();`	Create a task and execute it (e.g., `begin`, `cobegin`, `coforall`, and `forall`).
	`chpl_task_executeTasksInList();`	Do nothing in most of the available tasking models (e.g., Qthreads, OCR, and HCLib).
	`chpl_task_getId();`	Returns the ID of thread.
	`chpl_task_get/setSerial();`	Usually `getSerial` returns false - i.e., create tasks.
	`chpl_task_getMaxPar();`	Returns the number of workers in the node.
	`chpl_task_getCallStackSize();`	Returns the size of call stack size in the node.
	`chpl_task_createCommTask();`	Create a dedicated task for communication (For multi-locale execution).
	`chpl_task_taskCallFTable();`	Create a task that runs a function specified with function table indices and then execute it (For multi-locale execution).
	`chpl_task_startMovedTask();`	Create a task that runs the logical continuation of some other task and then execute it on a different node (For multi-locale execution).
Sync	`chpl_sync_lock(sync_var s);`	Acquire a lock on the specified sync variable.
	`chpl_sync_unlock(sync_var s);`	Release a lock on the specified sync variable.
	`chpl_sync_initAux();`	Initialize meta-information associated with a sync var.
	`chpl_sync_destory();`	Destroy meta-information associated with a sync var.
	`chpl_sync_waitFullAndLock();`	Block until the specific sync variable is FULL.
	`chpl_sync_waitEmptyAndLock();`	Block until the specific sync variable is EMPTY.
	`chpl_sync_markAndSignalFull();`	Set the specific sync variable to FULL.
	`chpl_sync_markAndSignalEmpty();`	Set the specific sync variable to EMPTY.

Table 1: Important Chapel Tasking API (14 tasking and 8 synchronization functions based on profile.)

Interestingly, all the `begin`, `cobegin`, `coforall`, and `forall` constructs eventually call `chpl_task_addToTaskList()` API function, meaning that the runtime does not differentiate each construct. This emphasizes the importance of lightweight task creation in the Chapel runtime.

Synchronization: Briefly, Chapel's sync variables are implemented using the following four functions:

- `chpl_sync_waitFullAndLock(s)`: Block until the specific sync variable `s` is *full*.
- `chpl_sync_waitEmptyAndLock(s)`: Block until the specific sync variable `s` is *empty*.
- `chpl_sync_markAndSignalFull(s)`: Atomically set the state of the specific sync variable `s` to *full*.
- `chpl_sync_markAndSignalEmpty(s)`: Atomically set the state of the specific sync variable `s` to *empty*

Figure 5 illustrates the code generated and Chapel Tasking API implementation for sync variables. The read of the sync variable (Line 3 in Figure 5) is compiled to 1) `chpl_sync_waitFullAndLock(s)`, 2) the read of the value of the sync variable, and 3) `chpl_sync_markAndSignalEmpty(s)`. Similarly, the write of the sync variable (Line 4 in Figure 5) is compiled to 1) `chpl_sync_waitEmptyAndLock(s)`, which is unblocked immediately since the initial value of the sync variable is *empty*, 2) the write of the value of the sync variable, and 3) `chpl_sync_markAndSignalFull(s)` that unblocks the execution of the spawned task.

4 IMPLEMENTING TASKING RUNTIMES

This section discusses the detailed implementation of Chapel Tasking/Threading API with Qthreads, the Open Community Runtime (OCR), and the Habanero C/C++ Library (HClib) [12]. While the Qthreads implementation is not a part of our contributions, it is worth describing it as the baseline of the three runtime systems.

4.1 Qthreads

4.1.1 Summary. Qthreads [25] is designed for executing and managing a large number of threads and is Chapel's default tasking runtime (`CHPL_TASKS=qthreads`). Threads of Qthreads are created with small stacks (4k-8k) and are entirely in user-space. In the following section, we will give a brief summary of how the Qthreads API is used in implementing the Chapel runtime.

4.1.2 Dynamic Task Creation. The current Qthreads implementation uses the `qthreads_fork_copyargs()` API to spawn a new thread when a `serial` task is not requested. As of this writing, `nemesis` is the default thread scheduler used for Chapel, which was originally developed for a communication subsystem for MPICH2. It employs lock-free FIFO queues using atomic swap and compare-and-swap. It is worth mentioning that the `nemesis` scheduler does not perform any work-stealing.

4.1.3 Synchronization. For synchronization between threads, Qthreads provides full/empty bits (FEBs), where a thread can wait on the state of a specific word of memory. Interestingly, Qthreads' synchronization API is analogous to Chapel's sync variable API and the implementation of sync variable with Qthreads is straightforward. For example, `qthread_readFE()` and `qthread_writeEF()` have the same semantics as the Chapel APIs `sy$.readFE()` and `sy$.writeEF()` discussed in Section 2, and are used for implementing reads/writes of sync variables.

4.2 Open Community Runtime

4.2.1 Summary. The Open Community Runtime (OCR) [19] is a community-led effort to develop a runtime system for extreme scale computing. The OCR execution model is based on performing computation using dynamic tasks named event driven tasks(EDT) which are synchronized using events. To help with data management, OCR includes the concept of a data-block, which is a relocatable chunk of memory.

Each EDT contains one or more pre-slots and one post slot, each of which can have an event attached to it. An EDT is scheduled for execution when all the events attached to pre-slots have been satisfied. Once the EDT finishes execution, it satisfies the event attached to its post-slot. There is also a special type of EDT called *finish* EDT which satisfies its post-slot only after all EDTs launched within its scope (i.e., all successor EDTs) have completed execution.

4.2.2 Dynamic Task Creation. As mentioned in Section 3, to enable dynamic task creation, we need to support the `chpl_task_addToTaskList()` API in the tasking layer. When a `serial` task is not requested, the OCR tasking layer implementation creates an EDT using the `ocrEdtCreate()` API and passes all its parameters by packing them into a `struct`.

To enable the `chpl_task_callMain()` API, which eventually invokes the compiler generated `main()` function, we use a *finish* EDT, which returns an event on which the runtime can wait for all successor tasks to complete. After the event becomes satisfied, the Chapel runtime invokes a finalization routine. Because the OCR specification does not directly allow blocking within a task, an ideal way to combine `chpl_task_callMain()` with the runtime finalization would be to create an EDT which waits on the output event of `chpl_task_callMain()` and then performs the finalization. However, this would involve making changes outside the Chapel tasking layer. To keep our changes to within the tasking layer, we used an OCR extension API to perform a blocking wait until `chpl_task_callMain()` is completed.

4.2.3 Synchronization. Because OCR does not directly support synchronization within an EDT, we used `pthread_mutex` and `pthread_condition_variables` in similar ways it is used in other pthread-based tasking layer implementations in Chapel. The `chpl_sync_lock()` and `chpl_sync_unlock` API functions in the tasking layer are mapped to `pthread_mutex_lock()` and `pthread_mutex_unlock()`.

`chpl_sync_waitFullAndLock()` waits on a condition variable for the state to be set as full. `chpl_sync_markAndSignalFull()` is the corresponding signaling call which sets the state to full and signals the condition variable. `chpl_sync_waitEmptyAndLock()` and `chpl_sync_markAndSignalEmplty()` performs similar operations when the state is empty instead of full.

4.2.4 Validation. To validate the correctness of our tasking layer implementation, we ran the parallel section from the test-suite provided in the Chapel repository. It includes 251 tests out of which we successfully passed 229 tests. In the 22 failed test cases, one case was due to the fact that we do not set the call-stack size given as a command line parameter. The test was setting the call-stack size to a very small value and expected the test to fail, whereas in our case it passed since we ignored that parameter. The remaining 21 failures were due to deadlock introduced by the pthread condition variable used to implement sync variables. When the number of tasks trying to access a sync variable exceeds the number of OCR workers, all the workers just remain to wait for the signal. However, since OCR is not aware of the wait performed by the condition variable, it cannot move the task out of execution and schedule another one. Therefore, all workers remain deadlocked.

4.3 HClib

4.3.1 Summary. HClib is a lightweight, work-stealing, task-based programming model and runtime that focuses on offering simple tasking APIs with low overhead task creation. Similar to Qthreads, HClib is entirely library-based (i.e. does not require a custom compiler, as is the case for Chapel) and supports both a C and C++ API. HClib's runtime consists of a static thread pool, across which tasks are load balanced using lock-free concurrent deques. Like Qthreads, HClib also uses runtime-managed, user-level call stacks to allow suspension of tasks without blocking CPU cores. Locality is a first-class citizen in the HClib runtime, which uses hierarchical place trees (HPTs) to encourage load balancing with nearby threads.

At the user-facing API level, HClib exposes several useful programming constructs. A brief summary of the relevant APIs is below:

1. `hclib_async`: Dynamic, asynchronous task creation.
2. `hclib_forasync`: Dynamic, bulk, asynchronous task creation (i.e. parallel loops).
3. `hclib_finish`: Bulk, nested task synchronization. Waits on all tasks spawned within a given scope.
4. `hclib_future` and `promise`: Standard single-assignment future and promise objects. Waiting on a future causes a task to suspend, but does not block the underlying runtime thread.
5. `hclib_launch`: Initialize the HClib runtime, including spawning runtime threads.

4.3.2 Dynamic Task Creation. Supporting dynamic Chapel task creation on the HClib tasking backend via the `chpl_task_addToTaskList` API is relatively straightforward. If a `serial` task is requested, we naturally short-circuit to a direct function call. Otherwise, the closure for the Chapel task is copied to a newly allocated buffer on the heap and passed to the `hclib_async` task creation API, which then immediately schedules the task on the HClib runtime.

The main entrypoint to the Chapel program must also be wrapped in a call to `hclib_launch` so as to initialize the HClib runtime before any tasks are spawned. `hclib_launch` implicitly waits for all tasks spawned in the runtime, so no additional synchronization is necessary. This requires a very small change to the Chapel runtime (~5 LOC).

4.3.3 Synchronization. The primary constructs used for point-to-point synchronization in HClib are futures and promises, so it was natural to focus on them when mapping the Chapel full-empty synchronization APIs on to HClib. HClib futures also have the desirable property of not blocking OS threads during blocking synchronization through their use of runtime-managed call stacks. In this section we present our initial implementation of the Chapel synchronization APIs on promises and futures, and then describe additional optimizations done on top of that initial implementation.
Promises and Futures: A wait or signal on a Chapel sync variable eventually maps to a call to `chpl_sync_lock` or `chpl_sync_unlock` in the tasking layer, both of which accept a `chpl_sync_aux_t` data structure representing the sync variable being synchronized on.

In this initial implementation of the synchronization APIs on HClib promises and futures, we add a queue of `hclib_promise_t` objects to the `chpl_sync_aux_t` data structure. When a Chapel task waits on a synchronization variable, it allocates a promise, adds that promise to the end of the queue for that sync variable, and immediately waits on the future object associated with that promise.

When signalling on a Chapel sync variable, the calling task simply removes the head of the promise queue associated with that sync variable and puts into it, waking up the next waiting HClib task.

We use pthread mutexes to protect these promise lists from concurrent access by multiple Chapel tasks. While this design is attractive in its simplicity, the use of a single pthread mutex per sync variable is naturally a source of concern in the scalability of this implementation.

Additional Optimizations using Ticket Locks: To address possible scalability issues with the initial implementation of Chapel synchronization APIs on HClib, we explored the extension of ticket locks [20] for managing concurrent accesses to sync variables. To be more specific, the ticket locks were used for implementing the `chpl_sync_lock` or `chpl_sync_unlock` API functions.

Ticket locks maintain the FIFO guarantees of our initial implementation. We use two stages of inter-task coordination in our ticket lock-based sync variable implementation. In the first stage, we use a spin wait with a timeout to gain access to the sync variable. This offers lighter weight synchronization and waiting than mutexes, particularly in the face of little contention for sync variables. However, a spin wait has the downside of consuming CPU cycles. Therefore, if the spin wait timeout is reached, we switch to an approach that is similar to our initial promise-based implementation which allows us to give up the current OS thread.

4.3.4 Validation. To validate the correctness of our implementation, we ran the parallel section from the test-suite provided in the Chapel repository. It includes 251 tests out of which we successfully passed 230 tests. In the 21 failed test cases, one case was due to the fact that we do not set the call-stack size given as a command line parameter as in OCR. The remaining 20 failures were due to deadlock introduced when the number of tasks trying to acquire a sync variable is more than the number of workers. During this case, all the workers just spins trying to acquire the sync variable, thereby starving the task which was supposed to release the sync variable.

5 PRELIMINARY EVALUATION

5.1 Experimental Protocol

Purpose: The goal of this performance evaluation is to validate our Chapel tasking implementation on OCR and HClib and to conduct a comparative performance evaluation. For that purpose, we benchmark the performance of PGAS programs on different Chapel's tasking/threading runtimes.

Machine: We present the performance results on a Cray XC30™ supercomputer. The platform has multiple Intel E5 nodes connected over the Cray Aries interconnect with Dragonfly topology with 23.7 TB/s global bandwidth. Each node has two 12-core Intel Xeon E5-2695 v2 CPUs at 2.40GHz and 64GB of RAM. Also, only a single node of the platform was used to evaluate this work.

Benchmarks: Table 2 lists five Chapel benchmarks that were used in these experiments. We chose these benchmarks as they use standard parallel constructs including `begin`, `forall`, `forall` with an intent (`reduce`), and `coforall`. `UTS` [22] is an unbalanced tree search benchmark that simulates different types of load imbalance. `Stream` is a simple vector kernel. `Label Propagation` is an algorithm that identifies communities of users [1]. `KMeans` is a well-established, unsupervised machine learning algorihm that divides data samples into k clusters. `CoMD` [15] is DoE proxy application that performs molecular dynamics simulations. All the benchmarks except `CoMD` [9, 15] can be found in the Chapel repository [4].

Experimental variants: Each benchmark was evaluated by comparing the following runtimes:

- **Qthreads:** Chapel's default tasking runtime based on the Qthreads library, configured by setting `CHPL_TASKS=qthreads`.
- **OCR (Open Community Runtime):** We used two OCR-based runtimes that were configured by setting `CHPL_TASKS=ocr`.
  - OCR-REF: A reference OCR implementation by Intel [16].
  - OCR-VSM: An alternative OCR implementation on top of Intel Threading Building Blocks [10] by the University of Vienna. OCR-VSM, which is for shared-memory systems only, was used for the evaluation.
- **HClib:** Our HClib-based runtime that was configured by setting `CHPL_TASKS=hclib`.

For all the variants, we used the Chapel compiler 1.14.0 with the `--fast` option. The Chapel runtimes were built using the Intel Compiler 17.0.2 unless otherwise indicated. For the Qthreads variant, the Chapel runtime uses Qthreads 1.11. For OCR variants, the OCR-REF variant is based on Intel's OCR 1.1.0, and the OCR-VSM variant is based on Intel TBB 2017 Update 7. For fair and clear performance comparisons, all the variants were executed within a single socket of the platform, meaning that 12-cores were used for the evaluation to avoid inter-socket communication. Performance was measured in terms of elapsed milliseconds from the start of parallel computation(s) to their completion. We ran each variant five times and reported the median value.

5.2 Preliminary Performance Results

Figure 6 shows absolute performance numbers for each variant. In general, the results show that 1) the HClib variants are the fastest due to the efficient task creation and scheduler, 2) the OCR-VSM variants are faster than the OCR-REF variants because the TBB-based implementation is better than the reference implementation, and 3) the Qthreads variants are in some cases faster, in some cases slower than the other variants.

A key difference between the Qthreads variants and the others is work-stealing (see Section 4.1), which can affect the performance of irregular applications such as `UTS` and `KMeans`. For `UTS`, the Qthreads variant is the slowest due to the lack of work-stealing. Based on our profiling with the Linux profiler `perf`, the Qthreads's scheduler (`qt_scheduler_get_thread`) is a major performance bottleneck whereas the other variants focus on the main computation (`sha1_compile`). Conversely, for `KMeans`, the Qthreads variant is the fastest. Additional experiments with Qthreads confirmed that

Benchmark	Applicagion Field	Description	Data Size	Constructs Used
UTS [22]	Tree Search	Unstructured Tree Seach (Deque)	T1 Tree (4M nodes)	`begin`
Stream	Numerical Computing	A Simple Vector Kernel	$n = 2^{28}$	`forall`
Labelprop [1]	Graph Analytics	An Analysis of Tweets on Twitter	nUsers=10^4, nTweets=10^5	`forall`
KMeans	Machine Learning	K-Means Clustering	$n = 10^7, k = 30, dim = 3$	`reduce`
CoMD [15]	Simulation	Molecular Dynamics Simulation	Cu, Lennard-Jones, $Grid = 20 \times 20 \times 20$	`coforall`

Table 2: Benchmarks used in our evaluation.

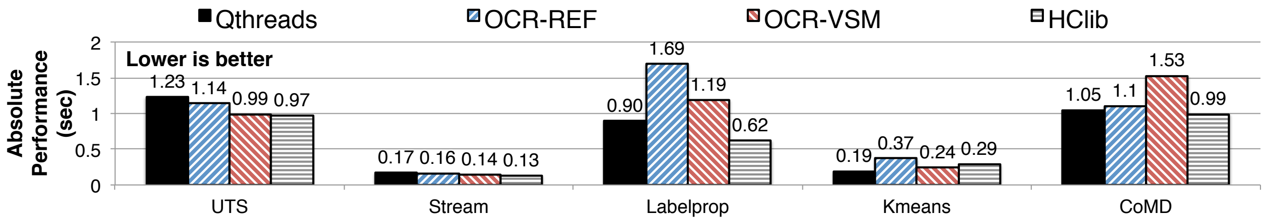

Figure 6: Overall Performance Numbers on the Cray XC30™ supercomputer.

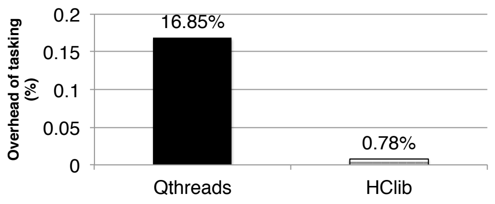

Figure 7: The percentage of tasking overhead (UTS).

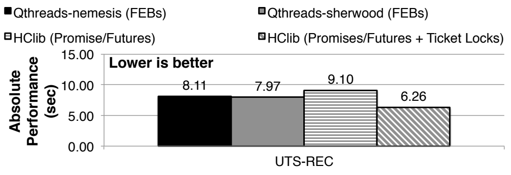

Figure 8: The impact of sync variable optimization using a Ticket Lock (UTS-REC).

the performance of UTS improves and that of Kmeans degrades when the `sherwood` scheduler with work-stealing is enabled.

For CoMD, we demonstrated that OCR-VSM can be slower than the other variants in a case where many tasks are spawned at the same time with the `coforall` construct. Based on our analysis with a synthetic `coforall` program, task creation overheads of OCR-VSM can be larger if many tasks are spawned in a short period.

Additionally, Figure 7 shows the percentage of tasking overhead out of the overall execution time and the results show that HClib is more light weight than Qthreads. These numbers are obtained by calculating $(T_1 - T_{seq})/T_{seq}$, where T_{seq} is the execution time of sequential UTS and T_1 is the single-thread execution time of parallel UTS. Note that the OCR overhead is not reported because T_1 can not be easily measured for OCR.

5.3 The Impact of Sync Variable Implementation

While Chapel's sync variables provide flexible synchronization between tasks, their implementation can significantly affect performance. To explore different sync variable implementations, we benchmark their performance on top of Full/Empty Bits (FEBs) in Qthreads, Promises/Futures in HClib, and the optimized Ticket Lock-based version on HClib discussed in Section 4. To that end, we used another version of UTS, UTS-REC, which is a recursive version that makes extensive use of sync variables. In this experiment, the Chapel runtimes were built by the GNU compiler collection (GCC) 6.3.0 due to some errors in supporting atomic intrinsics in the Intel Compiler.

Figure 8 shows absolute performance numbers for each variant. For fair comparison, we provide the performance of the `sherwood` scheduler as well as the `nemesis` scheduler to show the impact of work-stealing. The Qthreads variants are faster than the HClib variant with Promises/Futures. However, the optimized version of HClib outperforms the Qthreads variants. The results emphasizes the importance of an optimized sync variable implemenation.

6 RELATED WORK

There is an extensive body of literature on PGAS programming models and task-based runtime systems.

6.1 PGAS + Tasking

X10 [7] provides a `async-finish` parallel programming model. Like Chapel, X10 relies on compiler transformation to provide dynamic tasking capabilities and uses a work-stealing scheduler for load balancing of the dynamically spawned asynchronous tasks.

Co-Array Fortran [21] is an SPMD-style PGAS programming model, which was integrated into Fortran 2008 standard. UPC++[26] is a compiler-free PGAS library that provides a PGAS programming model with C++ templates. OpenSHMEM [6] is a low-latency communication library for PGAS programming that focuses on small- to medium-sized packets. In terms of task parallelism, these programming models normally rely on well-established threading models such as OpenMP and pthreads, but do not intrinsically support dynamic task parallelism.

Habanero-UPC++ [18] extends UPC++ to support a tight integration of intra-node and inter-node dynamic task parallelism by providing C+11 lambda-based user interfaces. Similarly, AsyncSHMEM [14] integrates the existing OpenSHMEM reference implementation with a thread-pool-based, intra-node, work-stealing runtime based on HClib.

6.2 Pluggable Parallel Runtimes

There is a smaller body of work exploring the ability to plug different parallel or tasking runtimes into the backend of higher level programming system, largely as a result of higher level programming systems either 1) lacking a well-defined, compartmentalized tasking layer, or 2) making subtle assumptions about the tasking runtime they sit on top of.

For example, while Legion [2] has a well-defined tasking API there are no published results to-date on any runtime other than the Realm [24] runtime released with it. However, work to support Legion on top of OCR is in-progress.

7 CONCLUSIONS

In this paper, we implemented OCR and HClib-based Chapel runtime systems to explore tasking runtime systems for PGAS programs. To do so, we first identified an important subset of the Chapel tasking API and implemented those API functions on top of the OCR and HClib libraries. We conducted performance evaluations using numerical computing, graph analytics, physical simulation, and machine learning applications written in Chapel. The results show that our OCR and HClib-based implementations can improve the performance of PGAS programs compared to the exisiting Qthreads-based implementation. In particular, we identified that 1) optimizing dynamic task creation, 2) optimizing sync variable implementation, and 3) optimizing work-stealing schedulers are essential for further performance improvements of PGAS programs.

For future work, we plan to conduct a comparable performance evaluation for distributed Chapel programs.

Another direction of this work is to add more flexibility in supporting high level constructs to Chapel's tasking layer because the current implementation (including Qthreads implementation) does not differentiate each construct. One example would be introducing a divide and conquer strategy (e.g., `cilk_for`) for executing `forall` and `coforall` constructs.

REFERENCES

[1] Ben Albrecht and Michael Ferguson. 2016. Social Network Analysis on Twitter with Chapel. In *Proceedings of the Chapel Implementers and Users Workshop (CHIUW '16)*.

[2] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 66, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389086

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: an efficient multithreaded runtime system. (1995), 207–216. https://doi.org/10.1145/209936.209958

[4] Chapel. 2017. a Productive Parallel Programming Language. https://github.com/chapel-lang/chapel (Accessed 13 October 2017). (2017).

[5] Chapel. 2017. The Chapel Language Specification Version 0.983. http://chapel.cray.com/docs/latest/_downloads/chapelLanguageSpec.pdf. (April 2017).

[6] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model (PGAS '10)*. ACM, New York, NY, USA, Article 2, 3 pages. https://doi.org/10.1145/2020373.2020375

[7] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing *(OOPSLA'05)*. ACM, New York, NY, USA, 519–538.

[8] Sanjay Chatterjee, Sagnak Tasirlar, Zoran Budimlić, Vincent Cavé, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. 2013. Integrating Asynchronous Task Parallelism with MPI *(IPDPS '13)*. IEEE Computer Society, Washington, DC, USA, 712–725. https://doi.org/10.1109/IPDPS.2013.78

[9] COMD. 2017. CoMD implementation in Chapel. https://github.com/LLNL/CoMD-Chapel (Accessed 13 October 2017). (2017).

[10] Jiri Dokulil, Martin Sandrieser, and Siegfried Benkner. 2015. OCR-Vx - An Alternative Implementation of the Open Community Runtime. In *International Workshop on Runtime Systems for Extreme Scale Programming Models and Architecture (RESPA '15)*.

[11] Tarek El-Ghazawi, William W. Carlson, and Jesse M. Draper. 2003. UPC Language Specification V1.1.1. (October 2003).

[12] Sri Raj Paul et al. 2017. Chapel Tasking Runtimes with OCR and HClib. https://github.com/srirajpaul/chapel/tree/hclib_ocr (Accessed 13 October 2017). (2017).

[13] William Gropp, Ewing Lusk, and Anthony Skjellum. 1994. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, MA.

[14] Max Grossman, Vivek Kumar, Zoran Budimlić, and Vivek Sarkar. 2016. Integrating Asynchronous Task Parallelism with OpenSHMEM. In *Workshop on OpenSHMEM and Related Technologies*. Springer, 3–17.

[15] Riyaz Haque and David Richards. 2016. Optimizing PGAS Overhead in a Multilocale Chapel Implementation of CoMD. In *Proceedings of the First Workshop on PGAS Applications (PAW '16)*. IEEE Press, Piscataway, NJ, USA, 25–32. https://doi.org/10.1109/PAW.2016.9

[16] Intel. 2017. Open Community Runtime. [online] https://01.org/open-community-runtime (Accessed 13 October 2017). (2017).

[17] Vivek Kumar, Karthik Murthy, Vivek Sarkar, and Yili Zheng. 2016. Optimized Distributed Work-stealing. In *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms (IA^3 '16)*. IEEE Press, Piscataway, NJ, USA, 74–77. https://doi.org/10.1109/IA3.2016.19

[18] Vivek Kumar, Yili Zheng, Vincent Cavé, Zoran Budimlić, and Vivek Sarkar. 2014. HabaneroUPC++: A Compiler-free PGAS Library. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS '14)*. ACM, New York, NY, USA, Article 5, 10 pages. https://doi.org/10.1145/2676870.2676879

[19] T. G. Mattson, R. Cledat, V. Cave, V. Sarkar, Z. Budimlic, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, Min Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo. 2016. The Open Community Runtime: A runtime system for extreme scale computing. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2016.7761580

[20] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (Feb. 1991), 21–65. https://doi.org/10.1145/103727.103729

[21] Robert W. Numrich and John Reid. 1998. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum* 17, 2 (Aug. 1998), 1–31. https://doi.org/10.1145/289918.289920

[22] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. 2007. *UTS: An Unbalanced Tree Search Benchmark.* Springer Berlin Heidelberg, Berlin, Heidelberg, 235–250. https://doi.org/10.1007/978-3-540-72521-3_18

[23] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. 2008. Phasers: A Unified Deadlock-free Construct for Collective and Point-to-point Synchronization. In *Proceedings of the 22Nd Annual International Conference on Supercomputing (ICS '08)*. ACM, New York, NY, USA, 277–288. https://doi.org/10.1145/1375527.1375568

[24] Sean Treichler, Michael Bauer, and Alex Aiken. 2014. Realm: An Event-based Low-level Runtime for Distributed Memory Architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 263–276. https://doi.org/10.1145/2628071.2628084

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

[26] Yili Zheng, Amir Kamil, Michael B. Driscoll, Hongzhang Shan, and Katherine Yelick. 2014. UPC++: A PGAS Extension for C++ *(IPDPS '14)*. IEEE Computer Society, Washington, DC, USA, 1105–1114. https://doi.org/10.1109/IPDPS.2014.115