Mapping High Level Parallel Programming Models to Asynchronous Many-Task (AMT) Runtimes

Thesis by
Sri Raj Paul
RICE UNIVERSITY

Mapping High Level Parallel Programming Models to Asynchronous Many-Task (AMT) Runtimes

by

Sri Raj Paul

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Vivek Sarkar, Chair
Research Professor, Department of Computer Science

Keith D. Cooper
Professor, Department of Computer Science

Peter J. Varman
Professor, Department of Electrical and Computer Engineering

Houston, Texas

December, 2018
ABSTRACT

Mapping High Level Parallel Programming Models to Asynchronous Many-Task (AMT) Runtimes

by

Sri Raj Paul

Asynchronous Many-Task (AMT) runtimes have recently been proposed as a promising software foundation for managing the increasing complexity of node architectures in current and future extreme-scale computing systems because of their ability to express fine-grained parallelism, to decouple computation and data from underlying machine resources, to support resilience, and to deliver scalable performance. The Open Community Runtime (OCR) is a community-led effort to explore AMT runtime principles that can support a broad range of higher-level programming models. The Habanero C/C++ library (HClib) is a library-based AMT runtime and programming interface, which focuses on lightweight task creation/termination and flexible synchronization. Unlike other AMT runtimes, both OCR and HClib include first-class support for event-driven task execution, which can help with hiding communication latencies and with reducing the number of blocking operations performed.

In this thesis, we focus on the problem of mapping high-level parallel programming models to AMT runtimes. As an exemplar of modern Partitioned Global Address Space (PGAS) parallel programming models, we show how Chapel programs can be efficiently mapped on to OCR and HClib, and also how Legion, a data-centric parallel programming model, can be mapped on to OCR. Next, we show how PGAS and event-driven execution
models can be synergistically integrated in a unique combination of server-side JavaScript and HCLib, yielding new levels of programming productivity for high performance computing. Finally, we show how the promise of supporting resilience in AMT runtimes can be realized through programming model extensions to HCLib. All these contributions are accompanied by performance evaluations of prototype implementations. Our results show that AMT runtimes can support high-level parallel programming models with comparable or improved performance relative to existing runtimes, while also providing the potential for improved resilience.
I thank my advisor, Prof. Vivek Sarkar, for helping me to come through my Ph.D. with flying colors. His expression of faith in me during difficult times helped me gain confidence and successfully finishing my Ph.D. During his busy schedule, he found time to discuss my topics and guide me through the right path.

I thank my co-mentor Dr. Akihiro Hayashi, for putting in a lot of time and effort to help me succeed in my Ph.D. He was always there to help me get familiar with the topics, design, implementation and thorough review of the paper.

I thank my committee members, Prof. Keith D. Cooper and Prof. Peter J. Varman for their time and inputs to improve the content of this thesis.

I thank my co-authors in the works presented in this thesis for the discussions and feedback. Along with their work, all of them found time to help me with my research.

I want to thank all members of the Habanero Exascale Research group for their support during my time at Rice and GeorgiaTech.

I would also like to thank Arkabandhu, Debarshi, Hamim, Nick, Prasanth, Priyanka and Rishi, Raji, Remya and Karthik, Rohan, Shams, Sourav for being good friends. I would also like to thank my other friends, all of whom I may have forgotten to mention here.

I would like to thank my family including my parents, sisters and their families and my wife who supported me throughout my journey. Without their support and motivation, I would not have overcome many challenges that came on the way.
# Contents

1 Introduction 1
   1.1 Thesis Statement .................................................. 4

2 Exploring AMT Runtimes for PGAS Languages 5
   2.1 Introduction ......................................................... 5
   2.2 Chapel ................................................................. 7
      2.2.1 Chapel Language Features ........................................ 7
      2.2.2 Chapel Tasking Layer ............................................. 10
   2.3 Implementing Tasking Runtimes ...................................... 15
      2.3.1 Qthreads support for Chapel (Past work) .................... 16
      2.3.2 Open Community Runtime Support for Chapel ............... 17
      2.3.3 Habanero C/C++ Library Support for Chapel ................ 20
   2.4 Experimental Results ................................................. 23
      2.4.1 Experimental Protocol ........................................... 23
      2.4.2 Preliminary Performance Results ............................... 25
      2.4.3 Sync Variables .................................................. 27
   2.5 Related Work .......................................................... 28
      2.5.1 PGAS + Tasking ............................................... 28
      2.5.2 Pluggable Parallel Runtimes .................................. 28
   2.6 Summary ............................................................... 29

3 Enabling a Data-Centric Programming Model on AMT Runtimes 30
3.1 Introduction .................................................. 30
  3.1.1 Legion .................................................. 31
3.2 Implementing Legion on OCR ................................. 34
3.3 Experimental Results ........................................ 38
  3.3.1 Preliminary Performance Results ....................... 40
3.4 Related Work ............................................... 42
3.5 Summary .................................................. 43

4 A Unified Runtime for PGAS and Event-Driven Programming 44
4.1 Introduction ................................................ 44
4.2 JavaScript .................................................. 47
  4.2.1 JavaScript Features .................................... 47
  4.2.2 ECMAScript Specification ............................... 49
4.3 Components of the PGAS JavaScript Runtime .............. 52
  4.3.1 Node.js ................................................ 52
  4.3.2 OpenSHMEM ........................................... 52
  4.3.3 Habanero C/C++ Library (HClib) ...................... 53
4.4 JavaScript API Design for PGAS ............................ 56
  4.4.1 Execution Model ....................................... 56
  4.4.2 User-Visible JavaScript Communication API .......... 57
4.5 Implementation ............................................ 60
  4.5.1 Launching Node.js instances across multiple PEs .... 60
  4.5.2 Selecting a C/C++ Interface Layer .................... 62
  4.5.3 Exposing Native Functions using the C/C++ interface 63
  4.5.4 Implementing Asynchronous communication .......... 65
  4.5.5 Implementing Asynchronous Locks ...................... 68
4.6 Experimental Results ....................................... 69
  4.6.1 Performance of (a)synchronous One-sided Communication 70
4.6.2 Performance of Distributed Locking ........................... 73
4.6.3 Performance of GUPS (Giga Updates per Second) ............... 74
4.7 Related Work .................................................................. 76
4.8 Summary ....................................................................... 77

5 Enabling Resiliency on AMT Runtimes 78
5.1 Introduction ................................................................... 78
5.2 Background .................................................................... 80
5.2.1 Types of Failures: Hard vs Soft ................................. 80
5.2.2 Types of Recovery ..................................................... 81
5.2.3 Asynchronous Many-Task Programming and Execution Model ... 83
5.3 Design ........................................................................... 85
5.3.1 Resilient API Specifications ....................................... 86
5.3.2 Memory Management ............................................... 96
5.4 Implementation .............................................................. 97
5.4.1 HClib ........................................................................ 97
5.4.2 Enabling Resilience in HClib ..................................... 97
5.4.3 Memory management ............................................... 101
5.5 Experimental Results .................................................... 101
5.5.1 Performance numbers without failures ....................... 104
5.5.2 Performance numbers with failures ........................... 107
5.5.3 Memory Profile ....................................................... 107
5.6 Related work ................................................................. 108
5.7 Summary ....................................................................... 109

6 Conclusions and Future Work 110

References 115
List of Figures

1.1 Comparison of improvement in bandwidth and latency over a 20 year period (Graphic source: [1]). .................................................. 3

2.1 Task parallelism constructs in Chapel. ........................................... 8
2.2 Sync variables in Chapel. ............................................................. 9
2.3 Chapel Runtime API ........................................................................ 10
2.4 Code generation and runtime implementation for begin ................... 11
2.5 Code generation and runtime implementation for sync variables. ....... 13
2.6 Overall Performance Numbers on the Cray XC30™ supercomputer. .... 25
2.7 The percentage of tasking overhead (UTS). ...................................... 26
2.8 The impact of sync variable optimization using a Ticket Lock (UTS-REC) 27

3.1 Legion Software Stack (Based on http://legion.stanford.edu/pdfs/bauer_thesis.pdf) ...................................................... 33
3.2 Legion-OCR Software Stack .......................................................... 34
3.3 Legion-OCR Machine Model .......................................................... 36
3.4 Overall Performance Numbers for the Legion benchmarks on single node. 40
3.5 Overall Performance Numbers for the Daxpy Legion benchmark on distributed environment. .................................................. 41

4.1 Standard execution environment for OpenSHMEM programs ............ 56
4.2 The internal structure of a single PE. ............................................... 57
4.3 Execution times of synchronous and various asynchronous versions of the 
get benchmark without work(8PEs). .............................................. 70
4.4 Execution times of synchronous and various asynchronous versions of the 
get benchmark with interleaving work(8PEs). .............................. 71
4.5 Comparison of execution times of synchronous and various asynchronous 
versions of the lock microbenchmark without and with interleaving 
computation (8PEs). ............................................................... 74
4.6 Comparison of execution times of synchronous and various asynchronous 
versions of the GUPS benchmark without and with interleaving 
computation (8PEs). ............................................................... 75

5.1 Comparison of execution times of different resiliency schemes on the five 
benchmarks without faults. ...................................................... 104
5.2 Comparison of execution times of the stencil benchmarks while mixing 
replay and replication with percent of replication shown. ............... 106
5.3 Comparison of percentage of change in execution times w.r.t baseline of 
different resiliency schemes on the five benchmarks with faults injected at 
1% and 10% rate. ................................................................. 107
List of Listings

4.1 An example of Callback function in JavaScript .................................. 49
4.2 An example of Callback hell in JavaScript ........................................... 51
4.3 An example of Promise in JavaScript .................................................. 51
4.4 An example of Async/Await in JavaScript .......................................... 51
4.5 An OpenSHMEM code example with shmem_int_p (PUT) ....................... 54
4.6 An HClib code example with finish/async .......................................... 55
4.7 JavaScript code using direct return to calculate sum of an array ............... 58
4.8 JavaScript code using callback to calculate sum of an array .................... 58
4.9 JavaScript code using promise to calculate sum of an array .................. 59
4.10 JavaScript code using promise and await to calculate sum of an array .... 59
4.11 JavaScript Equivalent of Listing 4.5 using direct return APIs ................... 61
4.12 The Node.js startup code that is executed by each OpenSHMEM PE (from our internal library, not exposed to the users) ....................... 62
4.13 Exposing native addon function to JavaScript (from our internal library) ... 64
4.14 Pseudo code for a general asynchronous communication implementation ... 65
4.15 Microbenchmark that performs the sum of a distributed array ............... 71
4.16 Microbenchmark that performs put with distributed locking .................. 73
5.1 A baseline non-resilient AMT program to perform some operation of two values. .......................................................... 86
5.2 Resilient AMT program based on replication to perform some operation of two values. .......................................................... 89
5.3 Resilient AMT program based on replay to perform some operation of two values. .......................................................... 90
5.4 Resilient AMT program based on ABFT to perform some operation of two values. .......................................................... 92
5.5 Resilient AMT program based on replay to perform some operation of two values that also checkpoints the results. ....................... 94
5.6 The checkpoint storage abstraction. Any class that implements the two APIs can be used by the runtime to save the checkpoint. .............. 95
5.7 Archive object that is accepted by the Archive store. ....................... 95
5.8 equals function is used as the equality operator to determine equivalence during replication. .................................................. 99
5.9 A sample integer archive object that is accepted by the Archive store. . . . 100
5.10 Reference Counting Promise creation. ...................................... 101
List of Tables

2.1 Important Chapel Tasking API (14 tasking and 8 synchronization functions based on profile.) ....... 14
2.2 Benchmarks used in our evaluation. ......................... 23

3.1 Mapping between Realm and OCR objects. ................. 38

5.1 Cache miss obtained using HPCToolkit for the stencil 3D application .... 105
Chapter 1

Introduction

Future extreme-scale computing systems require several challenges to be tackled by the software, such as (i) addressing their hierarchical organization of processors and memories, (ii) heterogeneity in parallelism and (iii) reduced mean time between failures. At the same time, it is important to maintain the productivity and expressibility of today’s high-level application programming constructs. There is a need for new runtime systems that bridge this gap between high-level programming models and complex extreme-scale hardware, while at the same time decoupling the computation (e.g., whether to perform computation on latency or throughput optimized hardware) and data (e.g., whether to place data in near or far memory) from the underlying system resource management. Task-based runtimes\(^1\), with their fine-grained parallelism, have recently been studied as a promising approach to address these challenges. Computations, decomposed into a large number of asynchronous tasks, can be distributed across multiple processing elements to achieve load balance. To support task migration, data used in computation can be represented as relocatable data objects. Asynchronous Many-Task models (AMT) [2–7] include the concepts of asynchronous tasks and relocatable data objects. Future extreme-scale architectures that are expected to consist of a large number of heterogeneous cores and extremely non-uniform hierarchical memories, aim to support efficient mappings of distributed applications, data and parallel computations. Dynamic task-based execution models hold promise in enabling this goal, as they help express fine-grained parallelism,

\(^1\) We use “runtimes” and “runtime systems” interchangeably in this document.
along with decoupling computation and data from underlying resources.

One of the significant projected obstacles for next-generation computing systems is reliability, due to an increase in soft/transient failures. For example, according to [8], the application failure probability on the Blue Waters system increases by a factor of $20 \times$ even when the number of nodes is only doubled, which emphasizes the importance of providing failure mitigation mechanisms. This implies that application-level resilience can play an important role in improving overall system reliability. Asynchronous many-task (AMT) programming models are better suited to enabling resiliency in next-generation platforms since AMT models provide explicit abstractions of data and tasks that are not tied to resources. The decoupling of computation and data from underlying resources enables the runtime to migrate task and data across faults. Enabling resiliency in the software layer opens the door to other possibilities. Computer hardware vendors have been unwilling to explore reducing reliability requirements (so as to offer performance and energy efficiency benefits) because of the lack of programming models that are resilient to errors and failures during program execution. Thus, AMT programming models that provide both high performance and resilience could motivate vendors toward this space for better application-driven system co-design.

Extreme-scale systems are characterized by high levels of on-node concurrency, leading to a significant decrease in communication bandwidth per core relative to past systems. As a consequence, the computation bandwidths of extreme scale systems far exceed their communication bandwidths. There is an old network saying: "Bandwidth problems can be cured with money. Latency problems are harder because the speed of light is fixed - you can’t bribe God" [1]. This implies that the issue of computation to communication latency gap will be more significant than the computation to communication bandwidth gap. The latency scaling problem has been well studied [1] and it has been observed that
Figure 1.1: Comparison of improvement in bandwidth and latency over a 20 year period (Graphic source: [1]).
network latency improved by only $15\times$ over a period of 20 years whereas computation bandwidth improved by around $2250\times$ over the same period as shown in Figure 1.1. This motivates the need for mechanisms to hide the computation to communication latency gap. The event-driven asynchronous execution model found in AMT runtimes provide hope for bridging this gap due to its inherent asynchronous nature that can help with hiding communication latency delays.

In summary, AMT runtimes show promise in solving many challenges in extreme-scale systems, including the mapping of high-level programming models to complex low-level hardware, and dealing with transient failures and the computation-communication performance gap, all of which are expected to show up in extreme-scale systems.

The rest of this thesis is organized as follows. Chapter 2 and 3 describe our experiences with mapping a high level PGAS language (Chapel [9]) and a data-centric programming model (Legion [6]) on to AMT runtimes. Chapter 4 describes the design of a unified runtime that uses event-based asynchronous mechanisms to hide communication latency, and demonstrates this combination in the context of the JavaScript event-driven programming model. Chapter 5 describes how to extend an AMT runtime to support various resiliency techniques. Finally, Chapter 6 presents our conclusions and summarizes directions for future research.

### 1.1 Thesis Statement

Our thesis is that Asynchronous Many-Task (AMT) runtimes can support high-level parallel programming models with comparable or improved performance relative to existing runtimes, while also providing the potential for improved resilience.
Chapter 2

Exploring AMT Runtimes for PGAS Languages

2.1 Introduction

While conventional message-passing programming models such as MPI [11] have proven successful for achieving scalable parallelism on distributed-memory platforms, they have done so at a large cost to productivity for many classes of applications (especially those with dynamic and irregular data accesses). Studies have shown that the human effort required to develop MPI based parallel programs can be significantly high [12]. As a result, there has been significant effort in past work to explore alternatives programming models that offer greater productivity than MPI. 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 [9, 13–17] are examples of highly productive programming models. Early 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. More recent versions also include support for AMT parallelism, synchronization, and mutual exclusion. It has also been shown that PGAS runtimes can enable higher productivity compared to MPI [18–20].

¹ Much of the work in this chapter was published at ESPM2’17 [10]
A key challenge in the development and use of PGAS programming models is the improvement of compilers and runtime systems to integrate PGAS and AMT models, since it is widely acknowledged that purely SPMD-style MPI or PGAS models will not suffice for extreme-scale platforms. Because PGAS languages can be dynamic or used for dynamic applications, it is likely that they can be expressed most naturally using large numbers of asynchronous tasks. 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 [21], efficient synchronization [22], and efficient task scheduling including work-stealing [23, 24].

There have been several tasking runtime systems designed for PGAS languages. For example, Qthreads [21] 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) [7] was designed to meet the needs of extreme-scale computing through an event-driven programming model with event-driven tasks (EDTs) and data blocks. Habanero C/C++ library (HClib) [5] is a library-based tasking runtime and API, which is semantically derived from X10 [17] 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 applications written in modern PGAS languages. To study and explore future runtime systems for PGAS languages, we have implemented OCR-based and HClib-based tasking/threading Chapel runtimes and conducted performance evaluations using various Chapel programs.

This thesis chapter 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 with a storage model consisting of disjoint relocatable data blocks.
   - Habanero C/C++ Library (HClib): A lightweight tasking runtime with support for synchronous/blocking operations with locality control.

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

### 2.2 Chapel

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.

#### 2.2.1 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.

**Dynamic Task Creation:** Chapel has several parallel constructs related to dynamic lightweight task creation. The list below summarizes those constructs. Figure 2.1 illustrates
// begin construct
begin {
    task(); // spawns a task executing task()
}

// cobegin construct
cobegin {
    taskA(); // spawns a task executing taskA()
    taskB(); // spawns a task executing taskB()
}

// coforall construct
coforall i in 1..N {
    // spawns a separate task for each iteration
    task(i);
}

// forall construct
forall i in 1..N {
    // may use an arbitrary number of tasks
    task(i);
}

Figure 2.1: Task parallelism constructs in Chapel.
var sy$: sync int; // value = 0, state = empty
begin {
    // 1. blocked until the state of sy$ is full
    // 2. read the value of sy$
    // 3. the state of sy$ is set to empty
    var sy = sy$; // equivalent to sy$.readFE();
    writeln("new task spawned");
    writeln("sy = ", sy);
    ...
}

// 1. blocked until the state of sy$ is empty
// 2. write 1 to the value of sy$
// 3. the state of sy$ is set to FULL
sy$ = 1; // equivalent to sy$.writeEF(1);

Figure 2.2: Sync variables in Chapel.

examples of their use.

- **begin**: spawns a task running independently from the parent task. (Line 2-4 in Figure 2.1)
- **cobegin**: spawns a block of tasks, one for each statement. The current task is blocked until all the tasks within the cobegin are complete. (Line 6-9 in Figure 2.1)
- **coforall**: spawns a separate task for each iteration. The current task is blocked until every iteration is complete. (Line 11-13 in Figure 2.1)
- **forall**: similar to coforall, but Chapel may choose to use a smaller number of tasks than iterations to execute the loop (e.g., by loop chunking). (Line 16-19 in Figure 2.1)

**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 \texttt{sy}$ (Line 6 in Figure 2.2) is blocked until the state is set to \textit{full} (Line 14 in Figure 2.2). Then, the state of \texttt{sy}$ is set to \textit{empty} after the read. Conversely, the write of \texttt{sy}$ (Line 14 in Figure 2.2) is blocked until the state is \textit{empty}. In this case, note that the initial state of \texttt{sy}$ is \textit{empty}. After the write, the state is set to \textit{full}, resulting in unblocking the read of \texttt{sy}$ in Line 6. It is worth noting that the name of sync variables ends in $ by convention.

Normal reads and writes of \texttt{sy}$ are equivalent to \texttt{sy}.readFE()} and \texttt{sy}.writeEF()} respectively. These functions are defined in the Chapel sync variable API, which offers more control in operating on sync variables. For example, \texttt{sy}.readEF()} is blocked until the state is \textit{empty}, and the state is set to \textit{full} after the read. The sync variable APIs also support \texttt{sy}.writeFE()}, \texttt{sy}.readFF()}, \texttt{sy}.writeFF()} and so on. More details can be found in the Chapel specification [9].

2.2.2 Chapel Tasking Layer

Chapel Compiler and Runtime

The Chapel compiler is written in C++ and generates C code which can then be compiled using a C compiler. The C code contains API invocations that are defined in the Chapel
// 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 2.4: Code generation and runtime implementation for begin
runtime for enabling communication, dynamic tasking, memory allocation, I/O, and so on (Figure 2.3). In the runtime, those APIs are implemented using different 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.

**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 2.1 summarizes the important Chapel Tasking/Threading API functions. This section provides a brief overview of dynamic task creation and synchronization using code examples (Figure 2.4 and Figure 2.5). More detailed discussions of the implementation of these APIs with Qthreads, OCR, and HClib can be found in Section 2.3.

**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 2.4 shows code generation and Chapel Tasking API implementations for the `begin` construct. The `begin` construct is compiled to the `chpl_task_addToTaskList()` API
// 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 2.5: Code generation and runtime implementation for sync variables.
<table>
<thead>
<tr>
<th>Kind</th>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td>chpl_task_init();</td>
<td>Call before executing the main function (Initialization).</td>
</tr>
<tr>
<td></td>
<td>chpl_task_callMain();</td>
<td>Create a task that runs main and then execute it.</td>
</tr>
<tr>
<td></td>
<td>chpl_task_exit();</td>
<td>Called when exiting.</td>
</tr>
<tr>
<td></td>
<td>chpl_task_yield();</td>
<td>Yield the execution to another thread (e.g., by calling sched_yield()).</td>
</tr>
<tr>
<td></td>
<td>chpl_task_addToTaskList();</td>
<td>Create a task and execute it (e.g., begin, cobegin, coforall, and forall).</td>
</tr>
<tr>
<td></td>
<td>chpl_task_executeTasksInList();</td>
<td>Do nothing in most of the available tasking models (e.g., Qthreads, OCR, and HCLib).</td>
</tr>
<tr>
<td></td>
<td>chpl_task_getId();</td>
<td>Returns the ID of thread.</td>
</tr>
<tr>
<td></td>
<td>chpl_task_get/setSerial();</td>
<td>Usually getSerial returns false - i.e., create tasks.</td>
</tr>
<tr>
<td></td>
<td>chpl_task_getMaxPar();</td>
<td>Returns the number of workers in the node.</td>
</tr>
<tr>
<td></td>
<td>chpl_task_getCallStackSize();</td>
<td>Returns the size of call stack size in the node.</td>
</tr>
<tr>
<td></td>
<td>chpl_task_createCommTask();</td>
<td>Create a dedicated task for communication (For multi-locale execution).</td>
</tr>
<tr>
<td></td>
<td>chpl_task_taskCallFTable();</td>
<td>Create a task that runs a function specified with function table indices and then execute it (For multi-locale execution).</td>
</tr>
<tr>
<td></td>
<td>chpl_task_startMovedTask();</td>
<td>Create a task that runs the logical continuation of some other task and then execute it on a different node (For multi-locale execution).</td>
</tr>
<tr>
<td>Sync</td>
<td>chpl_sync_lock(sync_var s);</td>
<td>Acquire a lock on the specified sync variable.</td>
</tr>
<tr>
<td></td>
<td>chpl_sync_unlock(sync_var s);</td>
<td>Release a lock on the specified sync variable.</td>
</tr>
<tr>
<td></td>
<td>chpl_sync_initAux();</td>
<td>Initialize meta-information associated with a sync var.</td>
</tr>
<tr>
<td></td>
<td>chpl_sync_destroy();</td>
<td>Destroy meta-information associated with a sync var.</td>
</tr>
<tr>
<td></td>
<td>chpl_sync_waitFullAndLock();</td>
<td>Block until the specific sync variable is FULL.</td>
</tr>
<tr>
<td></td>
<td>chpl_sync_waitEmptyAndLock();</td>
<td>Block until the specific sync variable is EMPTY.</td>
</tr>
<tr>
<td></td>
<td>chpl_sync_markAndSignalFull();</td>
<td>Set the specific sync variable to FULL.</td>
</tr>
<tr>
<td></td>
<td>chpl_sync_markAndSignalEmpty();</td>
<td>Set the specific sync variable to EMPTY.</td>
</tr>
</tbody>
</table>

Table 2.1: Important Chapel Tasking API (14 tasking and 8 synchronization functions based on profile.)

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.

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 2.5 illustrates the code generated and Chapel Tasking API implementation for sync variables. The read of the sync variable (Line 3 in Figure 2.5) 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 2.5) 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.

2.3 Implementing Tasking Runtimes

This section discusses the detailed implementation [25] of Chapel Tasking/Threading API with Qthreads, the Open Community Runtime (OCR), and the Habanero C/C++ Library (HClib). While the Qthreads implementation is not a part of our contributions, it is worth describing it as the baseline of the three runtime systems.
2.3.1 Qthreads support for Chapel (Past work)

Summary

Qthreads [21] is designed for executing and managing a large number of threads and is Chapel’s default tasking runtime (CHPL_TASKS=qthreads). Threads in Qthreads are created with small stacks (4KB-8KB) and are entirely in user-space. In the following section, we will give a brief summary of how the Qthreads API is used to implement the Chapel runtime.

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 as is done in OCR and HClib. There is an alternate task scheduler named sherwood available for Chapel that supports work stealing, but it is not used by default because it incurs larger overheads than nemesis.

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 2.2.1, and are used for implementing reads/writes of...
sync variables.

2.3.2 Open Community Runtime Support for Chapel

Summary

The Open Community Runtime (OCR) [7] is a community-led effort to develop an AMT runtime system for extreme scale computing. The OCR execution model is based on performing computation using dynamic tasks, referred to as event driven tasks (EDTs), which are synchronized using events. To help with data management, OCR includes the concept of a data-block (DB), 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 its 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 a finish EDT, which satisfies its post-slot only after all EDTs launched within its scope (i.e., all descendant EDTs) have completed execution.

Event is the synchronization mechanism in OCR and is used to express the dependence between EDTs and other Events. As outlined below, OCR has a few different event types to support different synchronization patterns:

- Once-event: is the default event type which is “triggered” when its pre-slot is satisfied. It is destroyed upon satisfaction.
- Sticky-event: is the same as once-event except that it is not destroyed automatically.
- Latch-event: has an increment and a decrement pre-slot, and gets triggered when it receives an equal number of satisfactions in both pre-slots.
• Channel-event: triggers when it has been satisfied a certain number of times (n), and the same number of dependences has been added to it. It gets reset after it is triggered and can be reused.

All OCR objects are uniquely addressable using a Global unique identifier (GUID) which is assigned to them when they are created.

Dynamic Task Creation

As mentioned in Section 2.2.2, 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 (`ocrLegacyBlockProgress`) to perform a blocking wait until `chpl_task_callMain()` is completed.

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 \texttt{chpl\_sync\_lock()} and \texttt{chpl\_sync\_unlock} API functions in the tasking layer are mapped to \texttt{pthread\_mutex\_lock()} and \texttt{pthread\_mutex\_unlock()}.

\texttt{chpl\_sync\_waitFullAndLock()} waits on a condition variable for the state to be set as full. \texttt{chpl\_sync\_markAndSignalFull()} is the corresponding signaling call which sets the state to full and signals the condition variable. \texttt{chpl\_sync\_waitEmptyAndLock()} and \texttt{chpl\_sync\_markAndSignalEmpty()} performs similar operations when the state is empty instead of full.

\textbf{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 when the program overflowed the stack size limit, 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.
2.3.3 Habanero C/C++ Library Support for Chapel

Summary

Habanero C/C++ Library (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.
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).

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 [26] 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.
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. One way to avoid this problem in the future is to use the Boost thread implementation to avoid deadlock in this case, as is done for other blocking operations in HClib.

2.4 Experimental Results

2.4.1 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 2.2 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 [27] 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 [28]. KMeans is a well-established, unsupervised machine learning algorithm that divides data samples into \( k \) clusters. CoMD [29] is DoE proxy application that performs molecular dynamics simulations. All the benchmarks except CoMD [29, 30] can be found in the Chapel repository [31].

**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 [32].
  - OCR-VSM: An alternative OCR implementation on top of Intel Threading Building Blocks [33] 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`. 
Figure 2.6: Overall Performance Numbers on the Cray XC30™ supercomputer.

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.

2.4.2 Preliminary Performance Results

Figure 2.6 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 2.3.1), 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
Figure 2.7: The percentage of tasking overhead (UTS).

(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 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 2.7 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.
2.4.3 Sync Variables

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 2.3. 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 2.8 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 implementation.
2.5 Related Work

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

2.5.1 PGAS + Tasking

X10 [17] 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 [13] is an SPMD-style PGAS programming model, which was integrated into Fortran 2008 standard. UPC++ [16] is a compiler-free PGAS library that provides a PGAS programming model with C++ templates. OpenSHMEM [34] 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++ [35] 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 [36] integrates the existing OpenSHMEM reference implementation with a thread-pool-based, intra-node, work-stealing runtime based on HClib.

2.5.2 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 [6] has a well-defined tasking API the only results to-date on any runtime other than the Realm [37] runtime released with it is the one with OCR.

2.6 Summary

In this chapter, 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 AMT runtime based implementations can improve the performance of PGAS programs compared to the existing Qthreads-based implementation.
Chapter 3

Enabling a Data-Centric Programming Model on AMT Runtimes

3.1 Introduction

Achieving good performance on extreme-scale systems requires addressing challenges arising from the complexity of extreme-scale architectures as well as the resulting application complexity. Extreme-scale architectures are expected to consist of large number of heterogeneous cores and deep non-uniform memory hierarchies. To obtain good performance on these platforms, we will have to take more of a data-centric approach to distributing computations and data structures than in the past. Asynchronous Many-task (AMT) runtimes can help in achieving this, as they have mechanisms to decouple computation and data from underlying resources, thereby enabling more flexibility in co-locating data regions and computational tasks.

As mentioned in subsection 2.3.2 of chapter 2, Open Community Runtime (OCR) is a community-led effort to explore various asynchronous task-parallel runtime principles that can support a broad range of higher-level programming constructs. Legion is a data-centric programming model in which the runtime extracts task-based parallelism from annotated sequential programs, thereby freeing the developer from having to express them explicitly. Our thesis aims to demonstrate that Legion programs can be run on top of OCR as their underlying runtime, so as to combine the parallelism extracted by Legion with the

---

1 Much of the work in this chapter was published in the 'Traleika Glacier X-Stack Extension Final Report', 2017 [38]
This part of our thesis makes the following contributions:

1. Implementation of a new tasking layer for Legion using the Open Community Runtime.
2. Performance evaluation and analyses of the new software stack using applications written in Legion.

3.1.1 Legion

Legion [6] is a high-level programming model and runtime system that targets distributed heterogeneous architectures. Data is represented in Legion using *logical regions* and computation using *tasks*. Logical regions help to organize data and are treated as first-class values in Legion. Logical regions help to automatically extract parallelism by reasoning about their independence from each other. Logical regions can be partitioned into sub-regions, which can be either disjoint or overlapping, which helps in determining the independence of computations, which in turn helps with parallelization. Legion allows computations to access logical regions with different privileges such as read-only, read-write, and reduce. Privileges provide tasks with data dependence information that can also be used to help with the extraction of parallelism. Furthermore, logical regions support coherence properties such as exclusive access, atomic access, and some others modes to express the required behavior of concurrent accesses to regions.

**Legion programming model**

Legion program execution can be viewed as a tree of tasks starting from a top-level task as the root. Every task is allowed to create as many child tasks as it chooses. Data management is performed using logical regions as mentioned earlier. Logical regions define the structure
and usage of data, such as whether the data is single-dimensional or multi-dimensional, and whether the task has exclusive access to a subset of the data. But logical regions do not specify the placement or layout of data in the memory hierarchy. Thus, logical regions separate application related data specification from machine-related issues. During execution, each task needs to be mapped to a processor, and each logical region/sub-region to a physical memory. The application programmer performs the mapping decisions by implementing a mapper interface. The Legion runtime queries the mapper dynamically to determine the state of the system, such as which processors are free, and where to place a task/datum so as to minimize data movement and thereby make good mapping decisions.

Legion exposes sequential semantics to the application programmer, and parallelism is extracted automatically by the runtime, similar to parallel extraction for OpenMP task dependencies and in the OmpSS project [39]. A Legion task is required to receive all the logical regions that it operates upon as input parameters. This enables the dynamic dependence analyzer in the Legion runtime to associate tasks with logical regions and extract parallelism. Legion tasks use a deferred execution model in which the application asynchronously creates tasks and their mapping, and execution are done independently later without blocking the application.

**Legion Software Stack**

The Legion runtime contains several layers of abstraction as shown in Figure 3.1. Towards the top left, we have the Legion application which is a machine-independent specification of the application. Towards the top right, we have the mapper interface which deals with the machine-dependent specification. The interaction between the Legion application and the mapper is enabled by the high-level runtime which also performs the dynamic analysis to extract parallelism. The high-level runtime converts the Legion application to
operations exposed by the low-level runtime called Realm. Realm provides primitives to perform computation, synchronization and data management. There is also a Machine model interface that enables the mapper to introspect the underlying hardware and get information such as the number of processors, memories, bandwidth and so on.

**Realm**

Realm is the low-level runtime that provides services to the high-level runtime to enable task scheduling and distributed execution. The main components of the Realm layer are,

1. Tasks: used to perform computation on a specific processor (local or remote).
2. Physical Regions: used to allocate data on memories.
3. Reservations: used to enable synchronization.

Realm provides the deferred execution model to the Legion runtime. All operations are asynchronous and return immediately with an event. The event is triggered when the
associated operation completes in future. Also, every operation can take an event as a precondition, i.e., the operation is scheduled for execution only after the pre-condition event is satisfied/triggered. Logical regions are mapped on to physical regions through which the tasks access the data. Physical regions include the storage space for data along with some metadata to efficiently support operations such as copying, allocation/deallocation and so on.

Another aspect of Realm is its support for portability. Future architectures can be supported either by extending or replacing Realm. In both cases, there is no need to reimplement the entire software stack. Since most of the code resides in the high-level runtime layer, separating Realm and the high-level runtime enables portability with reduced code modifications that are confined to Realm.

### 3.2 Implementing Legion on OCR

![Figure 3.2: Legion-OCR Software Stack](image)

The ideal way to run Legion on top of OCR is to replace Realm by OCR. This change
would require us to map logical regions directly on to OCR data blocks. Also doing so would need the high-level runtime to make the call to OCR instead of Realm, which would, in turn, require significant modifications to the high-level runtime source code. Therefore, we took a different approach in which we kept the Realm interfaces intact, and implemented some of those interfaces using OCR. We kept intact some parts of the Realm implementation that are unrelated to task scheduling and have no corresponding objects in OCR. This new software stack after the inclusion of OCR is shown in Figure 3.2, and is referred to as “Legion-OCR”.

**Machine Model**

Legion uses a machine model in which the whole system consists of multiple nodes with each node consisting comprising various processing elements (PEs). All PEs within a node share the same memory address space. This view is generally consistent with the hardware configuration of the system. On the other hand, OCR does not expose distinct PEs or separate memory address spaces. It exposes a system with a shared global namespace (shared data blocks accessible via guids). But the OCR runtime has knowledge of the hardware configuration such as the number of nodes, PEs per node and memory per node. This helps the runtime to schedule tasks on to PEs based on some scheduling policy. Similarly, data blocks are mapped on to memory in some node and copying of data block between nodes is managed internally by the runtime.

Legion-OCR uses a machine model which is in between that of Legion and OCR. Similar to the one exposed by the Legion, it consists of multiple nodes, such that each node has a memory object that is not directly accessible by other nodes. Each node exposes only one PE even though the underlying hardware might contain multiple cores. However, since OCR supports parallelism within a node, the single PE that is exposed in the machine
model gets mapped to multiple OCR workers which in turn gets mapped to the physical cores. Physical regions get allocated in the memory object (data block) that is exposed, as indicated in Figure 3.3.

**Mapping Realm to OCR**

Realm interfaces mainly include Tasks for computation and Events for dependency between them, Regions for data and Reservations for synchronization. Each of these objects is mapped to OCR as follows.

Realm task is implemented using an OCR EDT. The pre-condition event of the task is attached to the pre-slot of the EDT. The post-slot of the EDT is attached to the return event of the task. All the required parameters of the task are passed using the paramv parameter of the EDT. OCR affinity extension is used to place a task on a particular node.

Realm event is implemented using a combination of sticky event and latch event in OCR. Events in Realm are persistent, i.e., they can be used even after they are triggered. Therefore we map them to sticky events in OCR which is the persistent type of event in
OCR. Realm allows merging of events, i.e., a set of input events can be merged into a single output event such that the output event is triggered only when all the input events are triggered. The merging of events is implemented using a latch event in OCR. A latch event is created with a trigger count equal to the number of input events, and the input events are attached to its decrement slot. The latch event is attached as the dependency to the output sticky event. Realm also allows synchronous waiting on events within a task. Since OCR does not support synchronous waiting, we use the legacy extension of OCR (using the ocrLegacyBlockProgress API which performs spin-wait) to enable this functionality.

Realm memory object on each node is mapped to a large OCR data block. Region instances are then allocated within these data-blocks.

Realm reservations are used to support mutual exclusion in Legion. A reservation exposes acquire and release APIs. When multiple tasks invoke an acquire, only one task is granted access. And when it invokes release later, access is granted to another task that was waiting on an acquire operation. Reservation is implemented using an OCR channel event. An OCR channel event is triggered when it is satisfied a certain number of times, and the same number of dependencies are added to it. To implement reservation, we create a channel event with a satisfy/dependency count of one. Acquire is mapped to the addition of dependency to the channel event and release to the satisfying of the event.

The mapping between Realm and OCR Objects are given in Table 3.1.
3.3 Experimental Results

**Purpose:** The goal of this performance evaluation is to validate our OCR based tasking implementation for Legion runtime and to conduct a comparative performance evaluation. For that purpose, we benchmark the performance of the various Legion application program on the Realm-based default Legion runtime and our OCR-based new Legion runtime.

**Machine:** We present the performance results on a Davinci Cluster at Rice University. The platform has multiple Intel Westmere nodes connected over QDR InfiniBand interconnect with 40 Gb/s bandwidth. Each node has two sockets with each socket containing a 6-core Intel Xeon X566 CPU running at 2.83GHz and 48GB of RAM. The benchmark and runtime are compiled using GCC 6.4.0. We used up to eight nodes of the platform to evaluate this work. For multinode experiments, we used GASNet-1.26.0 build on top of OpenMPI 2.1.3 to do the internode communication.

**Benchmarks:** Three Legion benchmarks are used in these experiments.

Fibonacci benchmark finds the first n fibonacci numbers. To find the $n^{th}$ fibonacci number, it creates two tasks to find the n-1 and n-2 fibonacci numbers whose result is returned as a future. These futures are passed to another task which just adds the value inside the futures and returns the sum in another future. When a `get` operation is performed on the future, it waits until the result is available.
Daxpy kernel performs the computation $A^*x[i]+y[i]$ where $A$ is a constant and $x$ and $y$ are double arrays. We performed daxpy computation on an array of 24 million elements. The array is partitioned using logical regions into 160 partitions and the logical regions are passed on to different tasks. The placement of task is decided by the high-level runtime with inputs from the mapper. For example, when there are 16 partitions and 8 nodes, each node might get two partitions to work on.

Stencil computation kernel performs a 5 point stencil computation of the form,

$$x[i] = -x[i-2] + c1^*x[i-1] - c1^*x[i+1] + x[i+2]) /c2$$

As in the case of Daxpy, we performed stencil computation on an array of 24 million elements partitioned into 160 blocks. Compared to Daxpy, stencil involves communication of the edge elements (halo regions) between partitions.

**Experimental variants:** Each benchmarks are evaluated by comparing the following runtimes:

- **Legion-OCR:** Our OCR based Legion Runtime where tasking is offloaded to OCR. It has two variants.
  - x86: The shared memory OCR implementation that performs wait without blocking. When wait is invoked execution continues by creating a continuation and performing a context switch.
  - x86-gasnet: The distributed memory OCR implementation that performs wait and remote communications with blocking.

- **Legion-Realm:** The Legion reference implementation based on Realm tasking runtime.
3.3.1 Preliminary Performance Results

Figure 3.4 shows absolute performance numbers for each benchmark on a single node with different number of workers. Here we can see that for Daxpy and Stencil benchmarks the performance of Legion-Realm is slightly better than Legion-OCR. This overhead in Legion-OCR is mainly contributed by two factors:

1. The overhead of the translation through Realm interfaces rather than directly using OCR objects.
2. The overhead of OCR runtime itself compared to Realm because Realm was tuned for performance whereas OCR was initially targeted towards supporting multiple programming models rather than performance.

But an interesting observation is that in the case of the Fibonacci benchmark, Legion-OCR performs better than Legion-Realm. This performance improvement is due to the...
optimization introduced by Vrvilo [40] where during a wait operation, instead of blocking execution is continued by creating a continuation and performing a lightweight context switch. The Fibonacci benchmark contains a large number of wait operations, since each task which calculates Fibonacci of n needs to wait for the result from Fibonacci of n-1 and n-2 recursively.

Figure 3.5 shows absolute performance numbers for Daxpy benchmark on a distributed environment with up to 8 nodes. Here we can see that for Daxpy benchmark the performance of Legion-Realm is slightly better than Legion-OCR. As mentioned before this due to the overhead of translation through Realm interfaces rather than directly using OCR objects and overheads of the OCR runtime itself. The Fibonacci and Stencil benchmarks were causing deadlock\(^2\) because they were using a significant number of wait operations.

\(^2\)It is possible to run distributed experiments with very small inputs that will create a lower number of tasks and thus includes less number of wait operations and internode communication.
and internode communication both of which gets blocked inside the distributed version of OCR runtime³.

It is encouraging to see comparable (although slightly slower) performance of Legion-OCR in single node and distributed environments since OCR offers more flexibility than Realm in terms of supporting more high-level programming models.

### 3.4 Related Work

Chapel [9] is a high-level language that enables distributed execution with support for managing locality. Domains in Chapel are similar to logical regions, but domains are a higher level concept than regions. X10 [41] is another high-level programming language that works on distributed memory. Unlike Legion, X10 requires the application programmer to place the synchronization operations to join asynchronous computations.

OCR uses a dynamic explicit representation of graphs. Few systems that use static explicit graph include CnC [42], Sequoia [43] and Deterministic Parallel Java [44]. Static graphs can be scheduled at compile time resulting in low runtime overheads but might not be able to capture all patterns supported by dynamic data graphs.

Integration of tasking runtimes with other programming models has been attempted several times. Habanero-UPC++ [45] attempts integration of Habanero C/C++ library [5] with UPC++ [46]. Similarly AsyncSHMEM [47] integrates OpenSHMEM [34] PGAS communication library with the tasking capabilities of Habanero C/C++ library. Integration of CnC [42] with OCR was explored Vrvilo’s thesis [40].

---

³ Distributed OCR includes a non-blocking version that uses function stacking. Although devoid of blocking this version increases the problem deadlock as described in Chapter 3 of Vrvilo’s thesis [40]
3.5 Summary

In this chapter, we implemented an OCR based Legion runtime system to further explore the usage of OCR as a base AMT runtime for the development of a wide range of high-level programming models. To do so, we mapped Realm interfaces, which Legion used as its tasking runtime to various OCR objects. We conducted performance evaluation with three benchmarks to verify the correctness and scalability of our implementation. We conducted single node and multi node experiments using these benchmarks. Results show that our OCR based Legion runtime works with slower but comparable performance as the original implementation. We also found that the continuation based non-blocking implementation of wait operations can improve performance by a large margin.
Chapter 4

A Unified Runtime for PGAS and Event-Driven Programming

4.1 Introduction

Future extreme scale systems are characterized by high levels of on-node concurrency, leading to significant decreases in memory per core and communication bandwidth per core relative to past systems. As a consequence, the computation bandwidths of extreme scale systems far exceed their communication bandwidths. While the Message-Passing Interface (MPI) [49] has been the de facto standard for horizontal scaling of distributed applications in the past, it is clear that MPI by itself will not suffice for exposing all the parallelism needed by extreme scale systems. As a result, significant attention is being paid to the development of hybrid MPI+X models to expose hierarchical parallelism at the inter-node and intra-node levels. Also studies have shown that the human effort required to develop MPI based parallel programs can be significantly higher [12] than that required to develop the serial and even OpenMP [50] counterparts. These serve as motivations to explore alternatives to MPI, as well as to advance MPI in the future to address these limitations. In light of this disruption, many researchers are also exploring a broader hybrid D+X space, in which D represents a distributed runtime that could be an alternative to MPI and X represents a runtime for on-node parallelism that integrates well with D.

PGAS (Partitioned Global Address Space) [51] parallel programming models provide

---

1 Much of the work in this chapter was published at ESPM2’18 [48]
higher-level languages constructs for distributed memory systems compared to MPI. They aim to provide the illusion of a distributed memory space that functionally supports uniform addressability for programming convenience, while exhibiting explicit locality/non-uniformity to enable programmer-directed performance tuning and optimizations. Some notable examples of PGAS runtimes include Co-array Fortran [13], Chapel [9], Dash [52], Fortress [53], Global Arrays [54], Habanero-UPC++ [45], OpenSHMEM [55], X10 [41], Unified Parallel C (UPC) [15] and UPC++ [46]. Various studies have shown how PGAS runtimes can contribute to higher productivity compared to MPI [18–20]. PGAS runtimes have also proven to be effective in enabling efficient use of communication bandwidth for many applications, due to their efficient support for short nonblocking one-sided messages. However, they were not designed for exploiting the massive levels of intra-node parallelism found in extreme scale systems. Thus, a key question in the use of PGAS runtimes on extreme scale platforms is: what kind of node-level runtime system should be integrated with PGAS runtimes?

In this thesis chapter, we explore the premise that event-driven intra-node runtimes could be promising candidates for integration with PGAS runtimes, due to their ability to overlap useful computation with background long-latency operations. For this exploration, we use OpenSHMEM as an exemplar of PGAS runtimes, and Node.js as an exemplar of event-driven runtimes. While Node.js may seem an unusual choice for high performance computing, its prominent role as an event-driven runtime for server-side Javascript [56] is a good match for what we need for optimizing the use of communication bandwidth. Further, based on reports from Github [57] and Stack Overflow [58], JavaScript is currently the most popular programming language. This popularity is mainly because JavaScript is a very high-level and dynamic language, and thus it is not necessary for application developers to have “ninja” programming skills to use JavaScript. For instance, like other
popular languages such as Python, JavaScript starts its execution using an interpreter and provides quick feedback enabling faster debugging since we do not have to go through the whole compilation cycle.

While JavaScript has become the de facto standard for creating dynamic web applications, we observe that it has some potential synergies for high-performance computing as well. First, there is the availability of Node.js as a server-side JavaScript runtime environment that enables JavaScript code to run natively outside of a web browser. Second, the asynchronous event-driven nature of the JavaScript environment can enable it to tolerate long-latency operations such as remote memory accesses in a PGAS system. Finally, though there is the possibility of nondeterminism in event ordering, the JavaScript environment is single-threaded, thereby ensuring that no data races will occur on JavaScript objects. Thus, despite its potential performance overheads in compute operations, JavaScript has the potential to offer both productivity benefits and throughput benefits for network operations thereby making it an interesting direction to explore for programming PGAS systems. At the same time, the availability of excess computation bandwidth in extreme scale systems can help mitigate the local computation overheads of Javascript relative to native languages such as C++. Our integration of OpenSHMEM and Node.js makes it possible to explore nonblocking PGAS operations as JavaScript constructs that can seamlessly be used with the JavaScript asynchronous mechanisms that are based on callbacks and promises.

This thesis chapter makes the following two primary contributions:

1. The design and implementation of a unified runtime for PGAS and event-driven programming based on an integration of OpenSHMEM and Node.js. With this runtime, JavaScript programmers can write distributed parallel programs by manipulating shared OpenSHMEM objects using native JavaScript asynchronous mechanisms such as Callback and Promise.
2. A preliminary performance evaluation of our unified runtime using a few micro-benchmarks and HPC benchmarks.

We believe that the exploration and preliminary results in this thesis offer a new direction for future research on building high productivity runtimes for extreme scale systems.

4.2 JavaScript

JavaScript [56] is a weakly-typed, interpreted programming language created by Brendan Eich at Netscape and released in 1995. Its primary purpose was to complement Java in the Netscape Navigator. JavaScript has since been a fundamental scripting language for web development.

Contrary to most programming languages, JavaScript was designed to run in a client host environment; thus, it does not include any system functionalities such as I/O. Instead, these functionalities are provided by the host environment that embeds. In turn, JavaScript uses an event-driven execution model to interoperate with long-latency operations in the host environment.

4.2.1 JavaScript Features

JavaScript exposes a single-threaded execution model to the user. As a result, I/O operations such as HTTP requests to remote resources are handled asynchronously by the runtime. Otherwise, browsers will not be able to respond to user interactions if time-consuming I/O operation is executing. Thus, unlike programming languages like C and Java, JavaScript has the concept of an Event Loop [59]. The JavaScript runtime is primarily composed of three main components, call stack, heap memory and a message queue. The call stack indicates the current point of execution in which the state of every function call is represented by a frame. Heap memory is an unstructured memory space where programmers can freely
construct and manipulate JavaScript objects. As with the message queue, it is core to the JavaScript Event Loop concept [59] where JavaScript events [60] are registered and ready to be processed by the call stack. For client code, events [60] typically correspond to changes in the Document Object Model (DOM) [61].

Since browsers serve as host environments for JavaScript code, they are required to provide a web API for I/O operations to enable JavaScript to make I/O related function calls. In all web browsers, I/O functions are asynchronous which means that the return value is not available immediately. The lifecycle of an asynchronous operation can be broken down into the following steps:

1. Current function is pushed to the main stack
2. Corresponding Web API is invoked
3. Current function is popped from the main stack
4. An event is generated and pushed to the message queue
5. The event loop picks up the event for further processing

Given the asynchronous execution scenario summarized above, JavaScript introduces the concept of Callback functions [62] where the function execution is scheduled after the completion of the associated Web API. For programmers, the callback function is usually passed as a parameter to the asynchronous function call. When an asynchronous request finishes, the generated event along with the given callback function are pushed to the message queue. Once the event loop picks up this event, the callback function will then be executed.

Listing 4.1 illustrates a JavaScript example that executes a function after waiting for three seconds using the setTimeout API (Line 8). The setTimeout function invokes the callback function given in the first parameter after waiting for the given milliseconds
Listing 4.1: An example of Callback function in JavaScript

```javascript
// Define the callback function to
// invoke after waiting
function callbackFunction() {
  console.log("Printing after 3 seconds");
}

// Execute the callbackFunction after 3 sec
setTimeout(callbackFunction, 3000);

// Outputs a message to the console
console.log("End of JavaScript file");
```

specified in the second parameter. For the purpose of demonstration, the callback function only outputs a message to the console (Line 3-5). By running this program, we will first see the message from line 11 and then the message from line 4 after a short duration. The reason is tied to the lifecycle of asynchronous functions; after the corresponding API is invoked, `setTimeout` is removed from the call stack to make space for next instruction. Once the timeout concludes, an event is generated and pushed onto the message queue along with the callback function. Thus, line 11 is executed before line 8 even though the reverse is expected to happen from the perspective of top-down sequential execution.

### 4.2.2 ECMAScript Specification

ECMAScript [63] is a vendor-neutral scripting language standard published by ECMA International in 1997. It was published after Netscape submitted JavaScript to ECMA International in order to establish a standard for browser vendors. With newer versions of ECMAScript being published, more features are added to JavaScript. The notion of Promise [64] was introduced in ECMAScript 2015 (ES6); they represent the future values of asynchronous operations. In terms of functionality, they can be understood as the
syntactic sugar for callback functions in terms of readability. Imagine a scenario where three asynchronous operations must occur in sequence instead of parallel; intuitively, they should be arranged in nested callback functions as shown in Listing 4.2. In Listing 4.2, every callback function is an anonymous function and takes a value from the invoking asynchronous function. As the number of sequential asynchronous operations grows, the code will become unreadable. This situation is generally referred to as the Callback hell. Leveraging JavaScript Promise can turn nested callback functions into a trackable top-down code block as shown in Listing 4.3. In Listing 4.3, the return values from asynchronous function calls are forwarded via the .then handlers. It is important to note that the asynchrony of the program does not change with Promise, we would still see End of program before End of async actions as explained previously. ECMAScript 2016 (ES7) introduces async and await that provides local scope top-down execution for asynchronous function calls; essentially, the new feature provides an easier way to work with JavaScript Promises. Listing 4.4 is a rewrite of the program in Listing 4.3 using async and await. The async keyword modifies the function such that it always returns a promise (wrap the return value in a Promise Object if necessary). On the other hand, the await keyword makes JavaScript waits on the promise to finish and returns the corresponding value. From line 3 to line 5, await is used to wait for each asynchronous function to finish before moving to the next asynchronous function. As a result, in the scope of function functionWithAsyncAndAwait, everything is executed sequentially as if every line is blocking. However, the message End of program is expected to show up first since only the code block in the async function are executed sequentially. As of right now, global scope await is not yet available.
Listing 4.2: An example of Callback hell in JavaScript

```
firstAsync(function (firstVal) {
    secondAsync(firstVal, function (secondVal) {
        thirdAsync(secondVal, function (thirdVal) {
            // Done
        })
    })
})
```

Listing 4.3: An example of Promise in JavaScript

```
firstAsync().then(function (firstVal) {
  return secondAsync(firstVal);
}).then(function (secondVal) {
  return thirdAsync(secondVal);
}).then(function (thirdVal) {
  // Done
  console.log("End of async actions");
})

// Outputs a message
console.log("End of Program");
```

Listing 4.4: An example of Async/Await in JavaScript

```
// Define async function
async function functionWithAsyncAndAwait() {
    let firstVal = await firstAsync();
    let secondVal = await secondAsync(firstVal);
    let thirdVal = await thirdAsync(secondVal);

    // Done
    console.log("End of async actions");
    console.log("End of async function");
}

// Invoke the async function
functionWithAsyncAndAwait();

// Outputs a message
console.log("End of Program");
```
4.3 Components of the PGAS JavaScript Runtime

At a high-level, we provide a JavaScript runtime with an SPMD-style Partitioned Global Address Space (PGAS) programming model for distributed systems. Our JavaScript runtime essentially consists of Node.js [65], OpenSHMEM [55], and HClib [66, 67]. In the following sections, we gives an overview of each building block (Section 4.3.1, 4.3.2, and 4.3.3) followed by the high-level runtime design discussion in section 4.4.

4.3.1 Node.js

Node.js [65, 68] is a JavaScript runtime environment on top of the V8 [69] Engine. In simple terms, Node.js can run JavaScript outside of a browser (i.e., Server-side scripting). Node.js was created by Ryan Dahl and released in 2009. Node.js greatly extended the capability of web developer by allowing them to write server-side logic without picking up another programming language. Similar to JavaScript on a browser, Node.js employs an event-driven architecture with asynchronous I/O. Node.js also provides an interface called N-API [70] for developers to interact with C++ modules. N-API provides a stable Application Binary Interface (ABI) such that a module created using N-API can work on different versions of Node.js from different vendors.

4.3.2 OpenSHMEM

OpenSHMEM [55] is a communication library for Partitioned Global Address Space (PGAS) programming. Similar to MPI, OpenSHMEM offers an SPMD-style distributed programming model where multiple processing elements (PEs) execute the same program simultaneously.

The OpenSHMEM APIs essentially includes 1) memory management operations (e.g., shmem_malloc), 2) distributed atomic operations (e.g., shmem_add), 3) communication
operations (e.g., `shmem_put/get`), 4) collective operations (e.g., `shmem_barrier_all`), 5) point-to-point synchronization operations (e.g., `shmem_wait`), 6) distributed lock operations (e.g., `shmem_set/clear_lock`).

Listing 4.5 shows an OpenSHMEM code example in which 1) the source PE (PE0) puts an integer value of 42 to the destination PE (PE1) by using the `shmem_int_p` routine (Line 17), and then 2) PE1 reads the value (Line 23) after the bulk synchronization (Line 20). It is worth mentioning that all PEs must participate in the allocation of an object (Line 11) so each PE can easily obtain a pointer to the object on remote PEs, which is called Symmetric Heap allocation.

4.3.3 Habanero C/C++ Library (HClib)

HClib [66, 67] is a lightweight, work-stealing, task-based programming model and runtime that focuses on offering simple tasking APIs with low overhead task creation. HClib is entirely library-based (i.e. does not require a custom compiler) and supports both a C and C++ API. HClib’s runtime consists of a persistent thread pool, across which tasks are load balanced using lock-free concurrent deques. At the user-visible API level, HClib exposes several useful programming constructs. A brief summary of the relevant APIs is below:

1. `async([] { body; })`: Dynamic, asynchronous task creation.
2. `finish([] { body; })`: Bulk, nested task synchronization. Waits on all tasks spawned within a given scope.
3. `launch`: Initialize the HClib runtime, including spawning runtime threads.

The `async([] { body; })` API creates a task executing `body` given through a C++ lambda expression. The `finish([] { body; })` API waits for all tasks created in `body`, including transitively spawned tasks, before returning. Listing 4.6 illustrates an example
Listing 4.5: An OpenSHMEM code example with shmem_int_p (PUT).

```c
int main (int argc, char ** argv) {
    int mype, dstpe, npes;
    shmem_init();
    // The number of PEs
    npes = shmem_n_pes();
    // The ID of the current PE
    mype = shmem_my_pe();
    if (npes > 1) {
        // Allocate x on the symmetric heap
        long *x = (long *)shmem_malloc(8);
        // compute the ID of the destination PE
        dstpe = (mype + 1) % npes;
        if (mype == 0) {
            // PE0: writing 42 to PE1's x
            shmem_long_p(x, 42, dstpe);
        } else {
            // bulk synchronization (all-to-all barrier)
            shmem_barrier_all();
            if (mype == 1) {
                // PE1: reading the value written by PE0
                assert(*x == 42);
            }
        }
        shmem_barrier_all();
    }
    shmem_finalize();
}
```
with finish/async. \texttt{hclib::finish} (Line 4) creates a finish scope in which a new task is spawned (\texttt{hclib::async} in Line 6) and the completion of the spawned task can be ensured after this scope (Line 10).

Listing 4.6: An HClib code example with finish/async.

```cpp
int main (int argc, char ** argv) {
    hclib::launch([]() {
        int ran = 0;
        hclib::finish([]() {
            printf("Hello\n");
            hclib::async([](){
                ran = 1;
            });
        });
        assert(ran == 1)
    });
}
```

Also, HClib accepts a target platform model which is an abstraction of the homogeneous/heterogeneous hardware resources across which the workload of an application will be distributed. With this feature enabled, users can use the \texttt{async\_at}([[] { body; }], place) API to create a task at a specific place, which will encourage load balancing with nearby threads to improve data locality. More detailed discussions can be found in [66]. In this particular work, we use this feature to offload asynchronous communication tasks onto a dedicated communication worker.
4.4 JavaScript API Design for PGAS

Overall, the basic strategy for enabling JavaScript interfaces for PGAS programming is to introduce JavaScript API functions built on top of OpenSHMEM and HClib. This section particularly focuses on our communication API design.

Since JavaScript exposes a single-threaded interface to the user, executing long-running communication operations can cause a significant pause. Thus, in addition to synchronous communication API, we also introduce asynchronous versions which allow JavaScript applications to move forward without waiting for the completion of communication operations and asynchronously make progress on other computation tasks. In the followings, we first define the execution model of the runtime (subsection 4.4.1) that enables such functionalities and then discuss the design of the user-visible JavaScript communication APIs (subsection 4.4.2).

4.4.1 Execution Model

Figure 4.1 shows a standard execution environment for OpenSHMEM programs, with each processing element (PE) containing local memory as well as a slice of the distributed memory.

![Figure 4.1: Standard execution environment for OpenSHMEM programs](image)

Further, the current implementation launches one Node.js instance per OpenSHMEM PE (Process[i]). Figure 4.2 further illustrates the internal structure of a single PE from
Figure 4.1. In general, the single PE can be mapped to $m + 1$ cores, one for the Node.js instance, and $m$ for the HClib instance associated with the PE. One out of the $m$ cores is dedicated for OpenSHMEM communication. The remaining $m - 1$ cores are free to perform other operations. In the current implementation, only one HClib communication worker per PE serves the Node.js instance (i.e., $m = 1$), and the exploration of the best ratio of computation to communication worker is future work.

![Figure 4.2: The internal structure of a single PE.](image)

To enable asynchronous communication, the Node.js instance offloads the remote accesses on to the communication worker by spawning an HClib task. Then, the communication worker picks up the task and performs the actual remote communication. Once the communication is complete, the communication worker notifies the arrival of data to the Node.js instance.

4.4.2 User-Visible JavaScript Communication API

At the user-visible API level, the JavaScript runtime exposes several JavaScript communication API functions that are consistent with the execution model discussed in subsection 4.4.1. In other words, the APIs are capable of notifying the completion of remote operations in one synchronous (direct return) and two asynchronous ways (callback and promise).
Synchronous Communication API

In the direct return variant, the communication operation is invoked synchronously from JavaScript and waits for the output to arrive. An example using direct return to compute array sum of a remote array is shown in Listing 4.7. The `long_g_sync()` API is used to get a `long` element from a remote PE:

```
for (let i = 0; i < arrSize; i++) {
    let val = comm_addon.long_g_sync(arr + 8*j, remote_PE);
    sum += val;
}
```

Asynchronous Communication API (Callbacks)

The first option to perform asynchronous communication is callback, where the user provides a callback function as an additional function parameter. The `long_g_async_callback` API returns immediately and the JavaScript code proceeds ahead. Later when the remote data arrives the callback is invoked in the main event with the data passed to it. An asynchronous version of the remote get operation can be written using callback as follows:

```
for (let i = 0; i < arrSize; i++) {
    comm_addon.long_g_async_callback(arr + 8*j, remote_PE, function(val) {
        sum += val;
    });
}
```
A high-level sketch of the implementation that involves callback will look as follows. First, we extract the parameters passed including the callback. Then, we create an HClib task which invokes the actual communication operation. Once the communication operation finishes, we need to send the result back to JavaScript. However, it is not possible to directly invoke the callback from another thread onto JavaScript. We describe how we solved this problem in the subsection 4.5.4.

**Asynchronous Communication API (Promises)**

In the second option, the function parameters remain the same as the synchronous counterpart. However, instead of returning the data value, it returns a promise which is resolved when the remote communication is finished. An asynchronous version of the remote get operation can be written using promise as follows:

Listing 4.9: JavaScript code using promise to calculate sum of an array

```javascript
for (let i = 0; i < arrSize; i++) {
    let val_p = comm_addon.long_g_async_promise(arr + 8*j, remote_PE);
    val_p.then(function(val) {
        sum += val;
    });
}
```

To escape from the inversion of the flow of program and to make it appear synchronous we can use `await` on the promise version of communication operation as shown in Listing 4.10:

Listing 4.10: JavaScript code using promise and `await` to calculate sum of an array

```javascript
for (let i = 0; i < arrSize; i++) {
    let val_p = comm_addon.long_g_async_promise(arr + 8*j, remote_PE);
    await val_p.then(function(val) {
        sum += val;
    });
}
```
let val = await comm_addon.long_g_async_promise(arr + 8*j, remote_PE);
sum += val;
}

A high-level sketch of the implementation of the promise version of the API looks as follows. First, we extract the parameters passed and create a new promise within the native addon. Then we create an HClib task which invokes the actual communication operation. Once the communication operation finishes, we need to resolve the promise so that the continuation can be scheduled for execution in JavaScript. Similar to the callback, it is not possible to directly resolve the promise from the communication thread. We describe how we solved this problem in subsection 4.5.4.

JavaScript-OpenSHMEM API Demonstration

A direct translation of Listing 4.5 to JavaScript using the synchronous direct return APIs is shown in Listing 4.11

4.5 Implementation

In this section, we discuss the implementation of our unified PGAS-JavaScript runtime in detail.

4.5.1 Launching Node.js instances across multiple PEs

As discussed in subsection 4.4.1, the current implementation launches one Node.js instance per OpenSHMEM PE. To do so, the current approach assumes that each node launches a single Node.js instance from a normal C++ program which includes the main() function (e.g., by using the oshrun/srun). The JavaScript source file which needs to be executed is
Listing 4.11: JavaScript Equivalent of Listing 4.5 using direct return APIs.

```javascript
const comm_addon = require(<path>)

comm_addon.init_hc();
// The number of PEs
let npes = comm_addon.my_rank();
// The ID of the current PE
let mype = comm_addon.num_ranks();

if (npes > 1) {
  // Allocate x on the symmetric heap
  let x = comm_addon.malloc_sync(8);
  // compute the ID of the destination PE
  let dstpe = (mype + 1) % npes;

  if (mype == 0) {
    // PE0: writing 42 to PE1’s x
    comm_addon.long_p_sync(x, 42, dstpe);
  }
  // bulk synchronization (all-to-all barrier)
  comm_addon.barrier_all_sync();
  if (mype == 1) {
    // PE1: reading the value written by PE0
    assert(comm_addon.get_long_value(x) == 42);
  }
}
comm_addon.barrier_all_sync();
comm_addon.finalize_hc();
```
given as a command line argument to this program. For this purpose, we compile Node.js into a shared library (as opposed to the Node.js executable which can directly run the JavaScript code) and link it to this C++ program. The C++ program first launches the HClib runtime by using `hclib::launch`. Then, from the HClib runtime thread, a Node.js instance is started to execute the JavaScript program as shown in Listing 4.12 (Line 2).

Listing 4.12: The Node.js startup code that is executed by each OpenSHMEM PE (from our internal library, not exposed to the users)

```cpp
hclib::launch(deps, 1, [=]() {
    node::Start(argc, argv);
});
```

4.5.2 Selecting a C/C++ Interface Layer

Since OpenSHMEM and HClib are C/C++ libraries, it is important to discuss how Node.js and C/C++ programs interoperate in the context of 1) utilizing OpenSHMEM’s PGAS communication and HClib’s asynchronous tasking capabilities from users’ JavaScript program, and 2) accessing internal JavaScript objects from the OpenSHMEM and HClib side.

Node.js is built on top of the V8 JavaScript engine, which is implemented in C++, and utilizes many C/C++ libraries to implement many functionalities such as event loop (using libuv [71]), OpenSSL secure communication, HTTP parsing and so on. Also, Node.js allows the creation of custom C++ addons called as native addons that can be invoked from Node.js thereby enabling JavaScript to interface with custom C/C++ code.

While using Node.js, the JavaScript objects used in the application program internally gets translated to V8 as C++ objects. Thus to access those JavaScript objects from the native addon we will need to interface with V8 objects and functions. However, the V8 code base
undergoes dramatic changes from one version to another, and therefore native addons that are written directly to target V8 APIs or data types will need to be reviewed/updated and rebuilt with each new version of V8.

To avoid this problem, we use an abstraction-layer such as Native Abstractions for Node.js (NAN) [72] or Node.js APIs for native modules (N-API) [70] instead of interfacing directly with V8 internals. We decided to go with N-API in this project to create the communication addon due to the following reasons:

- While NAN provides a good abstraction layer to create native addons, it is specific to V8. Thus, the use of NAN will lead to portability issues if we decide to use another JavaScript engine such as Chakra or SpiderMonkey instead of V8 in the future.
- N-API, on the other hand, provides abstractions directly related to JavaScript and therefore can be used with any JavaScript engines. Also, N-API is already a part of Node.js source tree and its node-addons-api [73] module provides a C++ wrapper over N-API and allows to write more readable object model based programs.

To create N-API based C++ addons, we use the node-gyp build tool, which is based on the GYP build system from the Chromium project. We provide a binding.gyp build file on which we specify the C++ source files and its dependencies. The node-gyp tool compiles and builds the C++ program into a JavaScript native addon, which is essentially a dynamically-linked shared object. The addon can now be loaded into the JavaScript program similar to a JavaScript module using the require() API.

4.5.3 Exposing Native Functions using the C/C++ interface

To expose N-API based C++ addons that are backed by HClib and OpenSHMEM API, it is required to register these native functions to Node.js. This can be accomplished
by registering C/C++ functions to the exports object inside the Init function as shown in Listing 4.13 (Line 1-7). Here the C++ function named shmem_my_rank_fn() is exposed as the JavaScript function named my_rank(). Assuming the addon module is created with the name comm_addon and is imported using the require() API in Node.js, the function can be accessed as comm_addon.my_rank().

Also, Line 8-13 in Listing 4.13 shows the actual implementation of shmem_my_rank_fn() in C++. It is worth noting that every native addon function receives a Napi::CallbackInfo object from the JavaScript runtime. This object contains the JavaScript context from which the function call is made. The parameters passed to the function from the JavaScript code can be extracted from this object as shown in Line 11. Also, note that the data returned to JavaScript are not the plain C++ data types (Line 12). The returned valuable has to be converted to a Napi::Number before sending to JavaScript.

Listing 4.13: Exposing native addon function to JavaScript (from our internal library)

```javascript
// Registering shmem_my_rank_fn
Napi::Object Init(Napi::Env env, Napi::Object exports) {
    exports.Set(Napi::String::New(env, "my_rank"),
    Napi::Function::New(env, shmem_my_rank_fn));
    return exports;
}

// Actual implementation of shmem_my_rank_fn
Napi::Number shmem_my_rank_fn(const Napi::CallbackInfo& info) {
    Napi::Env env = info.Env();
    return Napi::Number::New(env, shmem_my_pe());
}
```
The implementation of JavaScript API for synchronous communication can be done in a similar way.

### 4.5.4 Implementing Asynchronous communication

As discussed in subsection 4.4.1, to enable asynchronous communication, the communication worker is responsible for 1) creating an HClib task that performs OpenSHMEM API and 2) notifying the completion of it. This can be done by using the `async_at([] { body; }, place)` API to pin an asynchronous communication task to the communication worker (Listing 4.14).

Listing 4.14: Pseudo code for a general asynchronous communication implementation

```c
// the Node.js worker creates a new HClib task
// and delegate it to the communication worker
hclib::async_at([=] () {
  val = SOME_SHMEM_API(location);
  // the communication is complete at this point
  // However, the communication worker is not allowed to 1) resolve the promise
  // and 2) invoke the callback
  ...
}, comm_worker);
```

However, the main difficulty in enabling the notification part is that the communication worker is not allowed to perform callbacks invocations and promise resolutions (in other words, only the Node.js worker is allowed to do so from the main event loop). To address this problem, we discuss the possible three design choices:
1. *Timer*

2. *UV Loop*


**Timer**

The first option we tried was to use the JavaScript timers to send the results back to JavaScript. In this approach we push the result of communication on to a queue along with the associated promise/callback. We also create a timer function which gets invoked periodically and pops elements from the queue and performs either a callback invocation or promise resolution. The timer functions are invoked from the main event loop and therefore it can interact with the JavaScript objects seamlessly. Since the push and pop happens from different threads we currently ensure atomicity using locks. This can be improved using a lock free single-producer single-consumer concurrent queue.

**UV Loop**

The problem with timer is that it can create high latency for communication operations that are dependent one other. Say for example, if the user performs a communication operation and based on its result invokes the next communication. In this case, if we use the timer approach, the second communication can only start based on the frequency of the timer rather than the actual latency of communication. Therefore we needed a more proactive approach where we can eagerly send the results back to JavaScript as soon as the communication is finished.

The main problem to send communication results to JavaScript is that the JavaScript runtime and communication worker are running in different threads. The data returned to JavaScript runtime are not the plain C++ data types that is returned by OpenSHMEM.
For example, a long data object has to be converted to a Napi::Number before sending to JavaScript. The conversion from long to Napi::Number can only be performed from the main event loop i.e inside a call made from the JavaScript runtime. So it is not possible to return the value to JavaScript from the communication worker once it is finished. Therefore we decided to move ahead and look into the internals of the main event loop mechanism in Node.js and how can it be access from a different thread.

Node.js uses libuv [71] to implement the event loop. libuv exposes an API called uv_async_send (handle) through which it is possible to enqueue a callback on the main event loop from a different a different thread. The uv_async_send accepts a handle through which we can pass data on to the main event loop. This implies that once the communication is finished we can enqueue a callback on to main event loop using uv_async_send and pass the value of communication result inside its handle. When the callback is selected for execution in the main event loop, it can fetch the value from the handle and pass it on to JavaScript. But libuv coalesces all calls made to uv_async_send, which means not every call to it will result in an execution of the callback. For example, if we invoke uv_async_send n times consecutively before the callback is invoked in the main event loop, the callback will only be called once. If uv_async_send is called again the n+1 th time after the callback happens, it will be called again. Therefore we stick to the previous design of using a queue to save all the communication results. When the callback happens in the main event loop, it just iterates through the queue and performs the promise resolution or JavaScript callback invocation with the result values.

**Thread-safe N-API functions**

N-API introduced a mechanism to invoke callback from a different thread in a recent release. This thread-safe asynchronous function call extensions was introduced in Node.js
version 10.6.0 (July 2018). This API extensions allows to create a thread-safe callback object from a normal callback object. This thread-safe callback object can be invoked from a different thread which will enqueue the associated callback to get invoked from the main event loop.

While creating a thread-safe callback object we need to provide another helper callback. When we a call is made to the thread-safe callback object with some data, this helper function is invoked in the main event loop with the data and the associated callback as its parameters. The helper function also receives a JavaScript environment object which will help to create the JavaScript objects of the C++ data and pass it on to the callback.

4.5.5 Implementing Asynchronous Locks

As discussed in subsection 4.3.2, the OpenSHMEM APIs include distributed lock operations to acquire/release a distributed lock (i.e., shmem_set/clear_lock) and our JavaScript APIs for distributed locking are available as with communication APIs.

When making the synchronous locking APIs asynchronous, unlike the other APIs, it is possible that the user acquires a lock for a specific object and proceed to acquire another lock for the same object without waiting for the first lock to be completed and then get unlocked. This semantics emphasizes the importance of serializing all lock/unlock calls to the same data.

To remove the users' burden in maintaining the order of lock operations manually, we decided to perform the serialization within the addon. For each data location used for locking, we maintain a counter storing the number of outstanding locks. If the number is more than zero, this means the location is currently locked and therefore the locking request is suspended. From the implementation viewpoint, the runtime pushes the associated promise/callback to a queue. Later when the unlock happens on the location and if the
queue is not empty, the runtime pops data from the queue and tries to acquire a lock again.

4.6 Experimental Results

Purpose: The goal of this performance evaluation is 1) to validate that JavaScript programs on our runtime perform PGAS communication, and 2) to compare the relative performance of synchronous vs asynchronous versions. For this purpose, we evaluated the correctness and performance using a few microbenchmarks and an OpenSHMEM benchmark.

Machine: We present the performance results on the Cori supercomputer located at NERSC. Each node has two sockets, each of which has a 16-core Intel Xeon E5-2698 v3 CPUs at 2.30GHz. Also, each physical core has a dedicated L1 cache of 32KB and L2 cache of 256KB. Each socket has an L3 cache of 40MB shared between the sixteen physical cores. Each node contains 128 GB DDR4 2,133 MHz memory. Cori uses Cray Aries interconnect with Dragonfly topology having a global peak bisection bandwidth is 45.0 TB/s.

Environment: For all the experiments, we used GCC 7.3.0 compiler to build the project. For communication, we used Cray-SHMEM 7.7.0, a high-performance implementation of the OpenSHMEM specification on Cray Systems. We used Node.js 10.8.0 as the JavaScript framework for our experiments.

Benchmarks: We evaluated three JavaScript programs to measure the performance of 1) one-sided communication, 2) distributed locking, and 3) random memory access.

Experimental variants: Each microbenchmark is evaluated by comparing the following communication variants:

- Direct return: In this option (section 4.4.2), the communication calls are executed synchronously and the call returns with the result.
- Callback: In this option (section 4.4.2), we use an asynchronous version of the communication call which returns immediately after enqueuing the communica-
Figure 4.3: Execution times of synchronous and various asynchronous versions of the get benchmark without work (8PEs).

- **Promise**: In this option (section 4.4.2), we use an asynchronous version of the communication call which returns a `Promise` immediately after enqueuing the communication operation on the communication worker. This return promise gets resolved with the result of the communication operation once it is finished.

Note that each of the two asynchronous variants (callback or promise) further has two variants (UV loop or Threadsafe N-API) as discussed in subsection 4.5.4. Also, we ran each variant five times and reported the average value and the standard deviation (as an error bar).

### 4.6.1 Performance of (a)synchronous One-sided Communication

The first microbenchmark computes the sum of a distributed array. The PE with rank 0 fetches data from each location one by one from each remote node and performs a local
Figure 4.4: Execution times of synchronous and various asynchronous versions of the `get` benchmark with interleaving work (8PEs).

sum as shown in Listing 4.15.

Listing 4.15: Microbenchmark that performs the sum of a distributed array

```java
if (rank == 0) {
    for (each remote_rank) {
        for (each arr_location) {
            // GET
            val = addon.long_g_sync(arr_location, remote_PE)
            // val = perform_computation(val)
            sum += val
        }
    }
    return sum
}
```

Figure 4.3 shows the absolute performance numbers for each variant for the mi-
crobenchmark. For each variant, we ran the benchmark using eight PEs. We placed each PEs on a different node to ensure the communication between them takes place over the network. We used an array size of 50,000 on each PE and rank 0 fetches the array data from every other rank. This means, in the execution with eight PEs, rank 0 fetches 350,000 remote elements in total. We report two execution times for each experimental variant; the first one (invocation) is the time taken to invoke all the communication calls from the JavaScript code and the second (Total) is the total time that includes the communication API invocation, the actual communication, and receiving the results. Note that the numbers for Direct return are the same because this variant is synchronous.

The first observation is that introducing asynchronous mechanisms introduce some additional overhead. This overhead is because of the extra work such as creating communication tasks, creating promises or persistent callback, and communicating values between the JavaScript event loop and communication worker. We can see from the figure that invoking the asynchronous calls alone takes a smaller amount of time (invocation, shorter blue bar) compared to the total time (Total, longer red bar). This means that the asynchronous versions can perform some computation on the received data in the JavaScript event loop while the communication worker proceeds and fetches the next remote data. To verify this scenario, we added some synthetic computation on the received data. This is represented in Line 7 in Listing 4.15. Figure 4.4 shows the absolute performance numbers for each variant for the microbenchmark after adding the computation work. Here we see that the UV loop based promise resolution variant is able to hide some of the overheads of the additional computation work and perform slightly better than all other versions including the original synchronous variant. Other asynchronous variants are not able

---

2 The synthetic computation performs square and square root of an input multiple times on a loop and returns the result which is the same as the input.
to absorb the overheads very well. We believe this scenario will change as the relative performance of processors increases with respect to networks and then all asynchronous versions will perform better than the synchronous counterpart.

4.6.2 Performance of Distributed Locking

The second microbenchmark puts a value to remote PEs’ memory location with distributed locking. Similar to subsection 4.6.1, we used eight PEs, each of which has an array of 50,000 elements. In this benchmark, Rank 0 acquires a lock on a remote location, writes a value to the location, and then release the lock as shown in Listing 4.16.

Listing 4.16: Microbenchmark that performs put with distributed locking

```c
if(rank == 0) {
    for(each remote_rank) {
        for(each arr_location) {
            // LOCK
            addon.set_lock_sync(lock_remote_rank)
            // PUT
            addon.long_p_sync(arr_location, some_value, remote_PE);
            // UNLOCK
            addon.clear_lock_sync(lock_remote_rank)
        }
    }
}
```

As with subsection 4.6.1, we experimented with the synchronous and asynchronous versions along with the introduction of some work to fill the idle JavaScript event loop
while waiting for communication results. Figure 4.5 shows the absolute performance numbers for each variant for the lock microbenchmark without and without interleaving work. In this case, also we are able to see a similar pattern, in which the UV loop based promise resolution variant is able to absorb some of the overheads of the additional work and perform slightly better than all other versions.

4.6.3 Performance of GUPS (Giga Updates per Second)

The third benchmark is GUPS (Giga Updates per Second) [74], which is a benchmark used to measure the random memory access performance. It is a part of the HPC Challenge Benchmark suite. The benchmark performs a read-modify-write operation on a distributed table of 64-bit words. The read-modify-write on a remote location is be done first directly without acquiring a lock. Later the read-modify-write is performed again after acquiring a distributed lock for the remote node. The benchmark works as follows: a distributed

Figure 4.5: Comparison of execution times of synchronous and various asynchronous versions of the lock microbenchmark without and with interleaving computation (8PEs).
address is generated, then the value at that address is read and modified using an integer operation (XOR) with some other value, and finally, the new value is written back to the same location. We created a JavaScript version of the GUPS benchmark that can be launched by multiple Node.js instances (one Node.js instance per OpenSHMEM rank). We created an asynchronous version of the benchmark based on promises and also added some synthetic computation work to fill the JavaScript event loop while waiting for the completion of remote operations. The benchmark uses a table of 1,024 words per node and performs a total of 32,768 read-modify-writes. Since we have eight nodes, this amounts to 4,096 updates done by each node.

Figure 4.6 shows the absolute performance numbers for each synchronous and asynchronous versions before and after adding the synthetic computation work. Towards the left of the graph is the timing for doing the read-modify-write without a lock and towards the right are the ones with a lock. We included the C version as the baseline which runs the fastest. The green bars (Without Work) represent the runs without interleaving synthetic

---

**Figure 4.6**: Comparison of execution times of synchronous and various asynchronous versions of the GUPS benchmark without and with interleaving computation (8PEs).
work, and the brown ones include the interleaving computation. When there is no interleaving computation, the synchronous versions outperform the asynchronous versions as expected due to the task creation and management overhead in the asynchronous methods. Once we interleave computation work, the asynchronous versions are able to absorb the additional work and run with similar performance as the synchronous counterpart. Since the rate of increase of computing power is more than that of network bandwidth and latency, we expect the asynchronous versions to perform better in the future.

4.7 Related Work

While there is an extensive body of literature on PGAS programming models, to our knowledge, this is the first work that enables PGAS programming using the JavaScript language. A summary of the literature on PGAS programming models is as follows:

X10 [41] and Chapel [9] provide a set of high-level parallel language constructs that support globally accessible data, task parallelism, synchronization, and mutual exclusion. Co-Array Fortran [13] is an SPMD-style PGAS programming model, which was integrated into the Fortran 2008 standard. Unified Parallel C (UPC) [15] also provides an SPMD programming model built on top of the standard C language.

Unlike these language-based approaches, there are several library-based approaches that facilitate adding new features. UPC++ [46] is a compiler-free PGAS library that provides a PGAS programming model with C++ templates. Habanero-UPC++ [45] extends UPC++ to support a tight integration of intra-node and inter-node dynamic task parallelism by providing C++11 lambda-based user interfaces.

Also, some approaches support asynchronous communication to hide data transfer latencies. For example, UPC++/Habanero UPC++ supports inter-node asynchronous copy and asynchronous function shipping. Similarly, AsyncSHMEM [47] enables the use of
asynchronous computation.

In summary, this work integrates the JavaScript language with an SPMD-style PGAS programming model and (a)synchronous communication based on OpenSHMEM [55], and HClib [66, 67].

4.8 Summary

In this chapter, we created a unified runtime that integrates PGAS communication with the event-driven JavaScript runtime. We used Node.js for executing JavaScript on the server and OpenSHMEM library to perform PGAS style communication. We exposed the communication APIs in three ways, such as:

1. Synchronous communication using direct return
2. Asynchronous communication using Callbacks
3. Asynchronous communication using Promises

To implement this, we used HClib AMT library to create a computation worker to execute the Node.js JavaScript runtime and a communication worker to perform the OpenSHMEM communication. We used N-API to expose the communication interfaces to JavaScript as a native C++ addon. To evaluate the correctness and performance of our unified runtime, we used few microbenchmarks and HPC benchmarks. The results show that introducing asynchrony adds some overhead, but also helps to hide the communication latency thereby improving the performance.
Chapter 5

Enabling Resiliency on AMT Runtimes

5.1 Introduction

High performance computing has been the norm for science and engineering involving simulations and analyses of very complex and large systems. Due to the insatiable demand for computing capability, various countries have committed to the development of exascale supercomputers that will be deployed in the 2020 to 2024 time frame. One of the major obstacles of exascale computing is reliability due to projected increases in soft/transient failures. For example, according to [8], the application failure probability on the Blue Waters system increases by a factor of 20× even when the number of nodes is only doubled, which emphasizes the importance of providing failure mitigation mechanisms. One insightful conclusion from [8] is that application-level resilience plays an essential role in improving application resiliency.

The most popular resilience technique for application users today is coordinated checkpoint and restart (C/R) typically with bulk-synchronous parallel programming models [76–78], which involves global coordination of processing elements (PEs) for accommodating a consistent global application state. However, this global recovery model suffers from inherent performance issues such as 1) the scalability and performance of the persistent (secondary) storage for checkpoint data, 2) a disproportionate use of resources by triggering global recovery even for local failures, and 3) a large overhead associated

\[1\] Much of the work in this chapter was published in the 'ASC CSSE Level 2 Milestone 6362: Resilient Asynchronous Many Task Programming Model.' Technical report, 2018 [75]
with garbage collection between global tear-down and restart. Furthermore, these efforts did not address resilience support for the emerging HPC programming models such as asynchronous many-task (AMT) programming models [6, 79–85] intended for managing the increasing complexity of node architecture and heterogeneity.

Building an unreliable computing system is not a big challenge, but it requires a solution to manage the reliability issues. Due to a lack of programming models that embrace errors and failures during program execution, computer system and chip vendors have been reluctant to deliver unreliable systems. Thus, a programming model that serves high performance and resilience together could motivate vendors toward better application-driven system co-design.

We claim that AMT models are better suited to enabling resiliency in next-generation platforms since AMT models provide explicit abstractions of data and tasks - i.e., 1) a task represents a small piece of program execution, 2) failures are manifested as failed or lost tasks, and 3) failures can typically be remediated using lighter-weight mechanisms such as task replay and task replication than checkpoint/restart. Recent efforts have demonstrated asynchronous application recovery [86] and explore ideas such as task replay and replication [87–89], suggesting a possible extension of a popular node-parallel programming model, such as OpenMP. However, the progress of the OpenMP standard specification is relatively slow, and the current focus of OpenMP 5.0 is performance portability and heterogeneity for the recent divergence in compute node architectures. It is also unclear how these efforts contribute to the ongoing effort on parallel programming models, such as DARMA [90] and Kokkos [91].

In this thesis chapter, we introduce a comprehensive approach to enabling resiliency in AMT programming models. While some of the prior approaches discuss different resilience techniques including task replay, task replication, algorithm-based fault tolerance
(ABFT [92]), and checkpoint/restart for different AMT programming models such as OmpSs [87, 93, 94] and PaRSEC [95], they are usually limited to a specific technique or prone to be application-specific. To the best of our knowledge, this is the first work discussing the design, implementation, and evaluation of a unified programming model that supports various resilience techniques.

This thesis chapter makes the following contributions:

1. Programming model extensions to enable resilience techniques from past work (task replay, task replication, ABFT, checkpoint/restart) to be applied to AMT applications.
2. Support for arbitrary composition of the extensions in 1).
4. Efficient memory management with reference counting.
5. Implementation of our approach as extensions to the Habanero-C/C++ library for many-task parallelism.
6. Comprehensive performance evaluation and analyses of our implementation

5.2 Background

In this section, we discuss the background of resilient programming models and runtimes for HPC.

5.2.1 Types of Failures: Hard vs Soft

On HPC systems, failures in application programs are manifested as job failures or incorrect application outputs, resulting in a waste of computing cycles or a use of wrong results. From the resilience perspective, these failures are classified into two major categories: hard and soft failures.
Hard failures are program crashes that cause a loss of computation and data. Typically, hard failures are easy to detect by the operating system, middleware, or transport layer of HPC systems. In parallel programs written with MPI, a crash of single rank (process) is detected by MPI message passing calls to the lost rank or the underlying process manager interface.

Soft failures are unexpected alterations in computation or data that can go undetected by the hardware and operating system. A soft failure itself can be either harmful or benign because its severity depends on the subsequent use of the altered data. This can mainly cause three behaviors.

1. Soft failures that alter the values of pointers (i.e. addresses) can lead to segmentation violations which may result in hard failures.
2. Soft failures that happen in the lowest bits of a mantissa or in application data that is never referenced after the error. In this case, the final output of the program is likely to be unchanged.
3. Soft failures that happen in the more significant bits can impact the final output of the application in a way that is either obvious or subtle.

5.2.2 Types of Recovery

In this section we discuss the resilience enhancement techniques targeted at today’s dominant SPMD model. In particular, resilience and fault-tolerance for MPI programming models and emerging algorithm-based fault tolerance are discussed.

Global Recovery

Today, Coordinated Checkpoint/Restart (C/R) is widely practiced and a few production-ready software packages [76–78] are available in the public domain. In a parallel program
execution, Coordinated C/R synchronizes all running processes to create a consistent global snapshot of the program, called a checkpoint, which is then stored in persistent storage such as the global file system. When failure is detected in the program execution, rollback is initiated.

This model fits a large class of HPC application programs written with MPI or its equivalents because of their bulk synchronous approach for global checkpointing and relatively simple garbage collection for the rollback. However, scalability has been hindered by two major issues:

1. The cumulative I/O bandwidth of the target system.
2. The overhead of the process restart.

The first I/O bandwidth issue can be addressed by a reduction of the size of the checkpoint (via compression or hierarchical checkpointing) at the software level, together with the availability of novel technologies at the hardware level, such as NVRAM. For large scale parallel programs, a lack of scalable process termination and restart in the existing process management middleware incurs a huge performance penalty and is accountable for most of the overhead of current recovery techniques. This overhead includes not only the required interaction with the system resource manager, but also the teardown of the previous, faulty, parallel job, the allocation of the new resources, the launching of the new application, the reconnection of all MPI processes, and finally the recovery of the last application state (from a previous checkpoint). Moreover, the exclusive use of global recovery leads to disproportionate use of computational resources to handle the most common failures occurring on a single thread, process, or node, resulting in a huge waste of computational resources.
**Local Recovery**

Along with the emergence of fault-tolerance proposals in the Message Passing Interface (MPI) standard, there has been an emerging idea of local recovery of parallel programs to overcome the shortcomings of C/R. This idea is based on the observation and anecdotes that the majority of application failures are attributed to local node/process failure as reported by [76], and that the recovery can be applied only to the corrupted processes and data without global coordination. Despite the simplicity of the idea, there are several challenges in implementation across the layers of computing systems. For example, uncoordinated Checkpoint Restart (UC/R) [96] exploits message contents (message logging) exchanged between MPI ranks to enable localized recovery. Several papers address the reduction of message logging overhead [97–101], while others describe a hybrid with global C/R [102]. Another example of local recovery uses Containment Domains (CDs) [103] that provide an abstraction of failure detection and correction intended for efficient and transparent recovery of HPC applications.

**5.2.3 Asynchronous Many-Task Programming and Execution Model**

The asynchronous many-task (AMT) model [6, 79–85] is a categorization of programming and execution models proposed as an alternative to the dominant SPMD programming models. AMT programming models and runtime softwares have several common functionalities across different implementations and packages. Typically, these frameworks decompose an application program into small, transferable units of work (many tasks) with associated inputs (dependencies or data blocks) rather than simply decomposing at the process level (MPI ranks). To enable more sophisticated decomposition of a program, the architecture of a typical AMT runtime involves several software components as listed below.
- Tasks
- Data blocks
- Runtime scheduler (task queues, dependency graph and task/data tables)
- Workers (thread/processes)

Despite minor differences between individual AMT implementations, an AMT runtime provides APIs to instantiate these components. The most important features are task and data objects encapsulated with their meta-data representations so that the runtime scheduler can orchestrate these objects. The runtime scheduler consists of task queues and a special construct to represent the task dependencies and monitor the status of task and data objects. Task dependencies can be expressed either explicitly or implicitly. For example, ParSEC [81] employs a static parametric task-graph to express all task dependencies, and the open community runtime (OCR) [85] employs event objects to notify state changes of individual tasks and data objects. OpenMP has supported task parallel computing since version 3.0, and extends the capability in the later versions. The latest version of Kokkos [104] supports task parallel computing to extend its performance-portable, data-parallel computing interface. Like Kokkos, HClib [66], described in section 5.4, exploits modern C++ features to instantiate tasks and data objects.

The term many-task encompasses the idea that the application is decomposed into many transferable or migratable units of data/work, to enable the overlap of communication and computation as well as asynchronous load balancing strategies. The transferable units and load balancing can be used as a mechanism to support easy incorporation of fault tolerance.

In this thesis chapter we discuss the design of a unified interface for AMT programming models that enables local recovery from soft errors.
5.3 Design

The first question is to find a program location around which we can perform error checking and recovery. For AMT programming models, the task boundary provides an ideal location around which resiliency can be implemented. The task constructs that are of our interest do not involve internal synchronization, i.e., once a task is started, it runs to completion without blocking or waiting for other tasks or data. This implies that a task can start only after it acquires all its inputs, and we can publish the results once it is finished; therefore, the task boundary provides a natural fit as the location around which resiliency can be implemented without much consideration of the internal task state or the global application state of the execution. This is in contrast to SPMD programming models where no such boundaries can be easily identified and, therefore, the global state is mostly required to make it resilient.

Once the program location around which resiliency can be implemented is identified, the next step is to identify the data that needs to be checked to ensure correctness. A trivial choice is to ensure the integrity of the whole data used in the program, but this could be very expensive to implement and also unnecessary. The next obvious choice is to look at data that is live, i.e., data which is going to be used later and therefore live at the end of task boundary (similar to the idea of live variables at the end of each data block in compilers). AMT runtimes and a recent version of C++11 provide promise and future constructs to enable transfer of data between tasks along with synchronization to avoid data races. Thus, if the application programmer uses only promise-future pairs to perform communication between tasks, then the live data at the task boundary is the data added to the promise. This implies the data that we need to check at the task boundary to ensure correctness is the data added to a promise within the same task. Thus, we have identified both the program location and data that needs to be checked to enable resiliency.
of applications based on AMT runtimes.

We assume the tasks are non-chaotic, i.e., for the same input dependencies, the task generates promises with data that is within some known range. Tasks do not need to be entirely deterministic - random numbers, etc. can be used within tasks so long as errors within the margin of the randomization’s effect are permissible. Also, we consider only side-effect-free tasks for this exercise.

5.3.1 Resilient API Specifications

To reiterate, key components to enable resiliency in AMT runtimes are tasks and promise/futures. This section discusses how to extend these AMT components to enable various resiliency techniques. In the following, suppose async is a generic AMT construct that creates an asynchronous task with a user-provided lambda expression, and async_await is a variant of async that can wait on a future.

First, as a baseline implementation without resilience, Listing 5.1 shows a code example where the function operation_val() in Line 17 creates an asynchronous task waiting on the completion of two tasks, namely read_first_val() and read_second_val(). Here get_val_from_src is a representative function that fetches and returns an input value. Like in Line 8 and 14, a future is satisfied by performing a put operation on the corresponding promise. Once the promises are satisfied, the operation_val() task which depends on the two promises gets scheduled for execution. After the completion of the task, the result (res) gets printed in the print_result() task.

Listing 5.1: A baseline non-resilient AMT program to perform some operation of two values.

```c++
auto val1_dep = new promise();
auto val2_dep = new promise();
```
auto res_dep = new promise();

void read_first_val() {
    async(=[] {
        val1 = new value(get_val_from_src());
        val1_dep->put(val1);
    });
}

void read_second_val() {
    async(=[] {
        val2 = new value(get_val_from_src());
        val2_dep->put(val2);
    }); // async
}

void operation_val() {
    async-await(=[] {
        val1 = get_value(val1_dep);
        val2 = get_value(val2_dep);
        res = new value(op(val1, val2));
        res_dep->put(res);
    }, val1_dep->get_future(), val2_dep->get_future()); // END async-await
}

void print_result() {
Task Creation
Satisfying a promise
Waiting on a promise

Replication

Task replication is aimed at proactive reliability enhancement by executing the same task multiple times, assuming that at least one replica can survive, or that the majority of the replicas produce the same output for determining correctness. The obvious drawback is the increase in computational cost, but it is still effective in situations where a few tasks in a critical path of the task graph leave the computing system underutilized. The replication overhead can be reduced by selective replication to control the trade-offs between the reliability and performance penalties.

Since task replication is based on equality checking of the outputs of the replica tasks, the runtime can internally take care of performing the replication and equality checking. There is no need for the user to provide any additional information other than the equality checking operator for each data type used. Also, the task APIs should include a mechanism to communicate the result of equality checking. This can be done using a promise that will have a value of 1 for success and 0 for failure. The replication version of the \texttt{operation\_val()} task from Listing 5.1 is shown Listing 5.2. We can see that the
only modification required in user code is to change the name of the task creation API and add an additional parameter, the err_dep promise which tells whether the task executed without faults.

Listing 5.2: Resilient AMT program based on replication to perform some operation of two values.

```cpp
auto err_dep = new promise();

void operation_val() {
    replication::async-await_check([=] {
        val1 = get_value(val1_dep);
        val2 = get_value(val2_dep);
        res = new value(op(val1, val2));
        res_dep->put(res), err_dep,
        val1_dep->get_future(),
        val2_dep->get_future()); // END async-await_check
    }, res_dep->get_future(),
}

void print_result() {
    async-await([=] {
        recoverable = get_value(err_dep);
        if (recoverable == 0) exit(1);
        res = get_value(res_dep);
        print(res);
    }, res_dep->get_future(),
}
The task replication construct
A promise with a failure status

The only data that gets propagated to dependent tasks are those that are put to a promise. With non-resilient tasks, dependent tasks get scheduled for execution once the necessary put operations have been performed. In order to prevent errors discovered in replication from propagating to dependent tasks, we do not publish any put operations from a replicated task until the equality checking of the replicas succeeds.

Replay

Task replay is a natural extension of Checkpoint/Restart for the conventional execution models. Instead of applying a rollback of the entire program, as few as one tasks are replayed when failure is detected. Task replay is more sophisticated than replication but has much less overhead. In this form of resiliency, the task is replayed (up to N times) on the original input if its execution resulted in some errors. Compared to replication, the application programmer needs to provide an error checking function so that the runtime can use it to check for errors. User visible abstraction for a replay task is to extend the task creation API to include an error checking function and data on which that function operates. The application programmer needs to fill the data (chk_data) that needs to be checked for errors using the error checking function (err_chk_func). The replay version of the operation_val() task from Listing 5.2 is shown Listing 5.3. Similar to Replication tasks, Replay tasks also do not publish the output until error checking succeeds.
Listing 5.3: Resilient AMT program based on replay to perform some operation of two values.

```c
bool err_chk_func(void *data) {
    if (data is good) return true;
    else return false;
}

auto err_dep = new promise();
void *chk_data = nullptr;

void operation_val() {
    replay::async_await_check(=[=]{
        val1 = get_value(val1_dep);
        val2 = get_value(val2_dep);
        res = new value(op(val1, val2));
        res_dep->put(res);
        chk_data = res;
    }, err_dep, err_chk_func, chk_data,
    val1_dep->get_future(),
    val2_dep->get_future()); // END async_await_check

The task replay construct
User-defined error checking function
Arguments to the error checking function
```
Algorithm-Based Fault Tolerance (ABFT)

Algorithm-based fault tolerance (ABFT) mitigates soft errors and failures using algorithm or application specific knowledge to correct data corruptions and computation errors. One of the seminal papers [92] introduced checksums that are embedded into the matrix and vector operators in parallel dense matrix computations to enable runtime error detection and correction. Other than dense matrix computations, the ABFT community has explored the numerical properties of Krylov subspace iterative linear system solvers [105] and multigrid solvers [106] for detecting failures and tuning the reliability of the critical numerical operators. By using the numerical properties of the algorithm, ABFT uses checksums or provides alternative formulations to recover from an error thus ensuring forward progress without redoing the whole computation. Thus the API designed for an ABFT task should provide a facility to check for errors and if there is an error, a way to recover from it. Therefore, the user level abstraction to include ABFT is to extend the replay task API with a recovery facility.

The ABFT version of the operation_val() task from Listing 5.3 is shown in Listing 5.4. The main extension compared to replay tasks is the lambda (Line 9) to perform the error correction. Similar to Replication and Replay tasks, ABFT tasks also do not publish the output until error checking succeeds.

Listing 5.4: Resilient AMT program based on ABFT to perform some operation of two values.

```plaintext
void operation_val() {
    abft::async.await_check{[=] {
```
val1 = get_value(val1_dep);
val2 = get_value(val2_dep);
res = new value(op(val1, val2));
res_dep->put(res);
data = res;

// perform error correction

val1_dep->get_future(),
val2_dep->get_future(); // END async_await_check

The ABFT construct
A lambda expression with error correction

Checkpoint/Restart (C/R)

Checkpointing involves the saving of intermediate program state/outputs on to secure storage so that in case of failure, the application can be restarted from the point when the checkpoint was taken rather than from the beginning of the program’s execution. From the context of task-based runtimes, once the error/equality checking succeeds at the end of a task, the output data can be checkpointed. Later in some following task, if all other resiliency techniques fail, it can re-fetch the input data from the checkpoint and execute again. This single-level checkpointing can be extended to multiple levels; i.e., if error checking still fails after re-executing the task with input from the checkpoint, we can go a level higher and re-execute the parent tasks using their input taken from the checkpoint and so on. Checkpointing can be added to any of the resilient task listed above.
A proposed user level abstraction for a checkpoint task created by extending the replay task is the same as that of the replay task itself as shown in Listing 5.5. The only addition is to specify once where to store the checkpoint data using the set_archive_store API as shown in Line 14.

Listing 5.5: Resilient AMT program based on replay to perform some operation of two values that also checkpoints the results.

```c
void operation_val() {
    checkpoint::async_await_check()([] {
        val1 = get_value(val1_dep);
        val2 = get_value(val2_dep);
        res = new value(op(val1, val2));
        res_dep->put(res);
        chk_data = res;
    }, err_dep, err_chk_func, chk_data,
    val1_dep->get_future(),
    val2_dep->get_future()); // END async_await_check
}
// specify where to store the checkpoint data
// (just once, before async_await_check)
set_archive_store(storage object);
```

**Checkpoint Store:** To enable the runtime to work with multiple checkpoint storage systems, we propose a design which abstracts the functionalities required for checkpoint store to a class `archive_store` with two APIs to save and retrieve data as shown in Listing 5.6. To add an object to a checkpoint store, we need to serialize the data contained
in the object into `archive_obj` which is nothing but the size and blob of data. The user has to provide the serialization/de-serialization APIs for the data that needs to be checkpointed.

Listing 5.6: The checkpoint storage abstraction. Any class that implements the two APIs can be used by the runtime to save the checkpoint.

```cpp
class archive_store{
    void save(void* key, archive_obj* data){}
    archive_obj* retrieve(void* key){}
};
```

To add an object to a checkpoint store, we need to serialize the data contained in the object. The user needs to provide the serialization/de-serialization APIs for the data that needs to be checkpointed. As shown in Listing 5.6, the checkpoint store accepts a serialized data format, namely `archive_obj`. The definition of `archive_obj` is shown in Listing 5.7.

Listing 5.7: Archive object that is accepted by the Archive store.

```cpp
struct archive_obj {
    int size = -1; //size of object blob
    void *data = nullptr; //blob of object to be archived

    archive_obj() {}

    archive_obj(archive_obj *ptr) {
        size = ptr->size;
        data = malloc(size);
        memcpy(data, ptr->data, size);
    }
};
```
5.3.2 Memory Management

C++ requires the user to perform memory management; i.e., the application programmer needs to explicitly free any data that is allocated in the heap memory. This could be reasonable to manage in normal AMT programs, but when we introduce resiliency manual deallocation poses certain challenges.

Many resiliency techniques involve multiple executions of the task to get the correct results. This would mean that the user needs to keep track of the good vs bad executions of the task. For the good runs, the data generated by a task would be used later in some consumer tasks; therefore, they need to be deallocated only after the consumption of the data. For bad runs, there is no need for the data created in the task and, therefore, they need to be deallocated at the end of the producer task itself. Keeping track of good vs bad runs and selectively deallocating memory would create unnecessary complexity in the application code.

Therefore, to reduce the user’s burden of manual memory management, we decided to add the reference counting capability that deallocates the data automatically once its use is over. Since data is being transferred between tasks using promise and future, reference counting is added by extending the promise to include the reference count. Ideally, the reference count specifies the number of tasks dependent on the future associated with the promise. The reference count is passed on to a promise when it is created. In other words,
a reference count \( N \) specifies that only \( N \) tasks consume data from the given promise, and therefore the promise and the associated data can be freed once \( N \) tasks have used it.

## 5.4 Implementation

In this section, we discuss the implementation of our resilient-AMT prototype, extended from the Habanero C++ library (HClib). An overview of HClib and its runtime capability are discussed in subsection 5.4.1 followed by efforts for the extension of HClib.

### 5.4.1 HClib

HClib [66,67] is a lightweight, work-stealing, task-based programming model and runtime that focuses on offering simple tasking APIs with low overhead task creation. HClib is entirely library-based (i.e. does not require a custom compiler) and supports both a C and C++ API. HClib’s runtime consists of a persistent thread pool, across which tasks are load balanced using lock-free concurrent deques. At the user-visible API level, HClib exposes several useful programming constructs. A brief summary of the relevant APIs is as follows.

The `hclib::launch()` API initializes the HClib runtime, including spawning runtime threads. The `async([] { body; })` API creates a dynamic task executing `body` provided as a C++ lambda expression; this API optionally allows the inclusion of parameters that specify precondition events thereby supporting event-driven execution for tasks when so desired (i.e., the `async_await()`). The `finish([] { body; })` API waits for all tasks created in `body`, including transitively spawned tasks, before returning.

### 5.4.2 Enabling Resilience in HClib

On top of HClib, we have implemented the resilience constructs (subsection 5.3.1), and the reference counting capability (subsection 5.3.2).
**Resilient Futures/Promises**

All data communication between tasks happens through promises. That means the only data that is propagated to dependent tasks are those that are put to a promise. With non-resilient tasks, dependent tasks get scheduled for execution once the necessary put operations have been performed. In order to prevent errors discovered in a resilient task from propagating to dependent tasks, we do not publish any put operations from a resilient task until the error checking of the data succeeds.

To hold the put operations until equality checking succeeds, we need additional space within the promise. The normal promise can hold only one value that had been added to it using the put operation. For replication, however, all replicas perform the put operation and, therefore, we need N locations within the promise rather than one. To accommodate this, we extended the reference counting promise with an array to store N values. During a put operation inside a replication task, the $i^{th}$ replica stores the value in the $i^{th}$ location of the array. Similarly for replay or ABFT tasks, to hold the output inside the promise until it is published, we extended the reference counting promise to include a temporary storage. Unlike replication, which requires an array of temporary storage, a replay and ABFT promise needs only one temporary storage space since the replay happens sequentially.

We need to collect all the put operations within the resilient tasks so that they can be checked for equality after all replicas finish. For this purpose, we extended HClib with task-local storage. Each put operation in the replica with index zero (we assume all replicas perform the same put operations) adds the associated promise to the task-local storage. Finally, while merging the results from the replicas, we fetch the promises from the task-local storage and check for equality on the data attached to those promises.

**Equals Operator:** The equivalence operator used in replication is exposed as an equals function of the object under consideration. The user needs to extend the resilient object
class and provide an equals function for the data that is added to a promise within the replicated tasks. An example of such a resilient float object is shown in Listing 5.8. Because correctness is judged by only comparing the outputs from multiple replicas, any error that impacts multiple replicas (e.g., corruption of cache or input data) could lead to an incorrect consensus.

Listing 5.8: equals function is used as the equality operator to determine equivalence during replication.

```cpp
class float_obj: public resilience::obj{
    public:
    float n;

    bool equals(obj* obj2) {
        float n2 = ((float_obj*)obj2)->n
        return (n-n2)*(n-n2) < 10^-6;
    }
};
```

Checkpoin/Restart (C/R)

In our current implementation, adding the data to the checkpoint occurs once the equality/error is checking succeeds. Now we do the storing of data in a synchronous manner which increases the total time in the critical path. The checkpointing needs to be moved out of the critical path to a new task. We also include a reference implementation of the checkpoint store that uses diskless checkpointing [107] which essentially saves the checkpoint to memory rather than disks.

As mentioned in section 5.3.1, user needs to provide the serialization of the object so that it can be stored to the checkpoint. A sample object definition that includes the
serialization/de-serialization of `archive_obj` is shown in Listing 5.9.

Listing 5.9: A sample integer archive object that is accepted by the Archive store.

```cpp
class int_obj : public checkpoint::obj {

  int n;

  public:

    //De-serialize

    int_obj(checkpoint::archive_obj* ar_ptr) {
      n = *(int*)(ar_ptr->data);
    }

    //Serialize

    checkpoint::archive_obj* serialize() {
      auto ar_ptr = new checkpoint::archive_obj();
      ar_ptr->size = sizeof(int);
      ar_ptr->data = malloc(ar_ptr->size);
      memcpy(ar_ptr->data, &n, ar_ptr->size);
      return ar_ptr;
    }
};
```

Task Graph Transformation

Our implementation of all the resilience mechanisms listed above, namely Replication, Replay, Algorithm-based fault tolerance and Checkpointing, do not change the overall structure of the graph. All transformations to the nodes of the graph happen locally, in
the sense that each node gets replaced by a small sub-graph. If we consider each of these small sub-graphs as a single node, then the original task graph and the transformed graph after adding resiliency are isomorphic.

5.4.3 Memory management

We implemented reference counting by extending the basic promise to include the reference count. Ideally, the reference count specifies the number of tasks dependent on the future associated with the promise. The reference count is passed on to a promise when it is created as shown in Listing 5.10. In other words, the reference count specifies that only count number of tasks consume data from that promise, and therefore the promise and its data can be freed by the runtime once count number of tasks have used it.

Listing 5.10: Reference Counting Promise creation.

```cpp
auto prom = new ref_count::promise_t<T*>(count);
```

Similar to the shared pointer in C++, we need to save the deleter function inside the promise object. This is because the type of data inside the promise will get erased when it is passed on to async_await APIs. Later, when we want to deallocate the data, we need the type of the pointer to the data because the delete API requires a typed pointer (in contrast to free which uses a void pointer).

5.5 Experimental Results

This section presents the results of an empirical evaluation of our runtime system, mostly on a single-node platform with a few experiments on a multi-node platform to show its viability in a distributed environment.

**Purpose:** Our goal is:
1. to demonstrate that our programming model supports various resilience techniques and
2. to study the performance impact of the resilient runtime system.

For that purpose, we evaluated several benchmarks with different resilience policies.

**Machines:** We present the results on a single-node Intel platform. The platform is a single node of the Cori supercomputer located at NERSC, which has two sockets, each of which has 16-core Intel Xeon E5-2698 v3 CPUs at 2.30GHz. Each physical core is capable of running 2 SMT threads, resulting in 64 CPU threads per platform. Also, each physical core has a dedicated L1 cache of 32KB and L2 cache of 256KB. Each socket has an L3 cache of 40MB shared between the sixteen physical cores. Cori uses Cray Aries interconnect with Dragonfly topology having a global peak bisection bandwidth is 45.0 TB/s. We used GCC 7.3.0 compiler for building the library and most benchmarks and Intel Compiler 18.0.1 for benchmarks that require MKL support.

**Runtime Settings:** We used the HClib-based resilient runtime system described in section 5.3 for this evaluation. Each worker of HClib’s persistent thread pool was pinned to a specific physical core by the `hwloc-bind` command to avoid any uncertainties in the runtime.

**Benchmarks:** Five C++ benchmarks were used in the experiments: Cholesky Decomposition, Stencil1D, Stencil3D, Conjugate Gradient and Smith-Waterman. For these benchmarks, to make use of AMT features such as load balancing, we over-decompose the data into tiles or cubes and have a separate task operate on each tile or cube.

Our first benchmark is the stencil 1D benchmark that solves linear advection (a hyperbolic PDE). We implemented this using the Lax-Wendroff 3-point stencil. In this benchmark, we use 128 tiles of size 16000 doubles, 128 time steps per iteration (each task advances its assigned tile 128 time steps), and 8192 iterations. For our next benchmark, we solve heat
diffusion (a parabolic PDE) on a 3D domain with periodic boundary conditions using a 7-point stencil. Here we use $16^3$ cubes, each representing a subdomain of size $32^3$, and run for 1024 iterations. Our next benchmark is a tiled version of Conjugate Gradient (CG), which is an iterative method for solving sparse systems of linear equations. A square matrix from the “SuiteSparse” collection (52804 rows/columns, 5333507 non-zeros) was set up with the CG method with 128 tiles and 500 iterations. Our fourth benchmark is the Smith-Waterman algorithm that performs local sequence alignment, which is widely used for determining similar regions between two strings of nucleic acid sequences. We use two input strings of sizes 185600 and 19200, divided among 4096 tiles arranged as 64X64. Our last benchmark is the Cholesky decomposition algorithm, which is used primarily to find the numerical solution of linear equations. Here we decompose a matrix of size 6000X6000 into tiles of size 200x200.

We enabled replication by adding the equals method to the data objects. The equals method is trivial in most cases as it just compares the data for equality. In the case of replay, to enable resiliency, we need to provide the error checking function. One of the most popular ways of designing error checking mechanisms is to use checksums. For the stencil benchmarks, we can detect corruption anywhere on a subdomain using physics-based checksums because conservation requires that the sum of values over the subdomain only changes by the flux through the subdomain boundary. For the Conjugate gradient and Smith-waterman benchmarks, there are not any sophisticated error detection mechanisms, so we simply return true, implying no error occurred. In the case when we want to inject faults, we pick a few instances of the error checking function to return false. The design of checksums for Cholesky decomposition is based on the work by Cao [109] for enabling ABFT. In this work, they added two checksum rows to each tile. The first checksum row is created by taking the sum of elements of each column, and the second is created by taking
the weighted sum of elements of each column, where the weight is based on the row index. We used the first checksum row to implement the error checking mechanism needed for replay. We also created an ABFT version of Cholesky by adding both checksum rows to each tile as in Cao’s work [109]. In this case, the first checksum row is used to detect an error, and both checksum rows are used together to recover from the failure.

5.5.1 Performance numbers without failures

Single Resiliency Technique

To show the overhead of the resilient runtime, the execution time of the five benchmarks using various resiliency techniques without failures is shown in Figure 5.1. For all the benchmarks, we used replay and replication to enable resiliency. For the Cholesky benchmark, in addition to replay and replication, we included ABFT. From the figure, we can see that for the stencil benchmarks, some additional time is required for the replay variant
Table 5.1: Cache miss obtained using HPCToolkit for the stencil 3D application

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>L1_DCM</th>
<th>L2_TCM</th>
<th>L3_TCM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replay</td>
<td>40.58 sec</td>
<td>7.97E+10</td>
<td>3.32E+10</td>
<td>6.21E+08</td>
</tr>
<tr>
<td>Replication</td>
<td>73.18 sec</td>
<td>1.61E+11</td>
<td>6.36E+10</td>
<td>9.19E+08</td>
</tr>
</tbody>
</table>

For the stencil 1D benchmark, this accounts for 7% overhead whereas in stencil 3D the overhead is around 20%. A close examination reveals the overhead includes both the computation of the checksum and additional overhead from the replay runtime. After removing the overhead caused by the checksum calculation, the replay runtime incurs an overhead of around 5% for stencil 1D and 3% for stencil 3D. For the Conjugate gradient benchmark, the replay runtime incurs an overhead of around 5%. For the Smith-Waterman and Cholesky benchmarks, we did not notice any significant overhead while using replay.

For the Cholesky benchmark, we also enabled ABFT. The time required for ABFT is slightly greater than replay because it involves additional cost to maintain the ABFT_lambda compared to replay tasks.

When replication is used, we can see that the execution time increases for all the benchmarks. We expected the time to double because, in the absence of faults, duplication of the tasks occurs. However, for a few benchmarks, the execution time was significantly less than double. We performed a detailed analysis of the stencil 3D benchmark, using HPCToolkit [110], to collect a detailed profile of the execution in order to pinpoint the reason for the reduced execution time. Primarily, we collected the following events: L1 data misses (PAPI_L1_DCM), L2 cache misses (PAPI_L2_TCM), and L3 cache misses PAPI_L3_TCM.

Table 5.1 includes the performance numbers and performance counter numbers for 1) the replay runtime without any failures, resulting in executing each task just once, and 2) the replication runtime in which each task is executed twice. Table 5.1 indicates that, while
Figure 5.2: Comparison of execution times of the stencil benchmarks while mixing replay and replication with percent of replication shown.

the replication runtime doubles the number of L1/L2 cache misses, it does not double the number of L3 cache misses compared to the replay runtime because most of the cubes reside on the L3 cache. Thus, due to the cache reuse, the replication variant is not 2x slower than the variants that execute each task only once.

**Mixing Resiliency Techniques**

To illustrate that the various resiliency techniques can be seamlessly combined, we also tried to mix replay and replication in the stencil benchmarks. On one end, the application only uses replay, and on the other end, it uses just replication. In between, the amount of replication is increased in increments of 20%. Figure 5.2 shows that the execution time increases linearly while mixing replay and replication. This implies that the increase directly corresponds to the additional cost for running replication and thus no additional overhead is involved.
5.5.2 Performance numbers with failures

To check the effectiveness of our resilience mechanisms in the presence of soft errors, we ran all the benchmarks while introducing errors. We injected errors at a rate of 1% and 10%. Here, 10% implies that an error is injected into 10% of the total tasks. Figure 5.3 shows the execution time for various benchmarks and resiliency techniques in the presence of faults. Here, also, we can see that the increase in execution time closely follows the amount of failure occurred. For the 10% failure rate, in most cases, the increase in execution time is also around 10%. Failures do not cause much time increase in case of ABFT because the ABFT error correction is very lightweight compared to replaying the task.

5.5.3 Memory Profile

We looked into the memory usage of Stencil 1D to understand the memory overheads. Stencil 1D used approximately 500MB of RAM for user data and the replay and replication
runtimes used around 350MB more. The first source of the overhead of a resilient task is that we save the lambda inside the resilient runtime. The second overhead is that the resilient promises contain additional space to hold on to the output until the error check is finished. Another source of overhead is the use of a promise to return the result of error checking. Finally, the replay task involves some additional variables such as an error checking function and its parameter. These variables get saved as a part of the lambda, thereby making the lambda bigger.

5.6 Related work

As discussed in section 5.2, there is an extensive body of literature on software-based resilience scheme for SPMD programs [111,112] including coordinated checkpoint and restart (C/R). However, enabling resiliency in the AMT (Asynchronous Many Task) programming model has not yet been well studied despite the fact that the abstraction of tasks and data in AMT models facilitates modeling recovery patterns to enable asynchronous and localized application recovery with simplicity. To this end, this section focuses on discussing the recent work in AMT resilience:

Task Replication: Subasi et al [87, 93] study a combination of task replication and checkpoint/restart for a task-parallel runtime, OmpSs [83]. Their checkpoint API is integrated with the input data parameters of OmpSs directives to protect the input of individual tasks. They also suggested deferring launch of the third replica until duplicated tasks report a failure. However, the mixed use with other resilience techniques and in-depth analysis of the performance penalties are yet to be studied.

Task Replay: Subasi et al [94] also study a combination of task replay and checkpoint/restart for OmpSs. As with task replication, checkpoint/restart is utilized for preserving the input of tasks. During the execution of a task, errors notified by the operating
system trigger a replay of the task using the input data stored in the checkpoint. Cao et al [95] has a similar replay model. However, the drawback of these approaches is a lack of mitigation for failure propagation, as they assume reliable failure detection support, e.g., by the operating system, which is not always available. Our approach provides a general interface that allows user-level failure detection.

**ABFT:** Cao et al [95] also discuss an algorithm-based fault tolerance for Cholesky factorization in the PaRSEC runtime. However, they do not discuss their user-visible APIs in terms of general applicability, while our approach provides a general support for ABFT.

### 5.7 Summary

In this chapter, we extended HClib, an AMT runtime library to include various resiliency methods to mitigate transient faults. In particular, we added support for task replication, task replay, algorithm-based fault tolerance and checkpointing. The APIs are designed such that it requires few code changes from the application programmer and most of the details of the fault-tolerance algorithm are implemented within the runtime. We also allow composition of the different resiliency extensions and also added a reference counting module based on counting task dependencies. To evaluate the usability and performance of our implementation, we conducted performance evaluations using various benchmarks. The results show that The task replay can enable very efficient error checking and ABFT allows very efficient recovery. In summary, our approach 1) provides programming model extensions for various resilience techniques (e.g., task replay, task replication, ABFT, checkpoint/restart) in the context of AMT applications, 2) supports arbitrary composition of the extensions in 1), 3) enables unified execution of resilient and non-resilient tasks in a single framework, 4) provides efficient memory management with reference counting.
Chapter 6

Conclusions and Future Work

In this thesis, we addressed various challenges facing software for extreme-scale systems, as described in chapter 1. This research focused on the role of asynchronous many-task (AMT) runtime systems to address the challenges in three key areas. The first area we addressed was how to bridge this gap between high-level programming models and the complex low-level hardware. The second research area explored the usage of event-driven asynchronous mechanisms to hide network communication latency. The third area discussed how to extend an AMT runtime to support various resiliency techniques to address transient errors.

In the first part of our thesis, we implemented OCR and HClib-based Chapel runtime systems to explore tasking runtime systems for PGAS programs, and conducted performance evaluations using a wide range of applications written in Chapel. The results show that AMT-based implementations can improve the performance of PGAS programs compared to Chapel’s existing Qthreads-based implementation. In particular, we identified that: optimizing dynamic task creation, optimizing sync variable implementation, and optimizing work-stealing schedulers are essential for further performance improvements of PGAS programs.

An interesting direction for future research is to conduct a comparable performance evaluation for distributed Chapel programs. Another direction for 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. Similarly, another example would be adding additional API functions for supporting Futures.

To further demonstrate that AMT runtimes can be used to support a wide range of high-level programming models, we enabled a data centric-programming model on OCR. We implemented an OCR based Legion runtime system which is capable of running Legion programs with minimal changes. To do this, we focused on identifying the module within the Legion software stack: Realm, that enabled tasking. Then we mapped Realm interfaces to various OCR constructs thus allowing Legion to run on top of OCR. We conducted performance experiments with a few kernels to validate the correctness and scalability of our implementation. The results show that our OCR based Legion runtime works with comparable performance as the original implementation. These results demonstrate that is possible for the AMT-based runtime to support a wide range of high-level programming models thereby shielding them from the complexities of the low-level hardware. An interesting direction for future research is to extend AMT runtimes so as to support waiting within a task, as in Realm, while also retaining other benefits of AMT runtimes that Realm lacks.

In future, we want to directly target OCR from Legion runtime rather than going through the Realm-OCR shim. This is because, although Realm interfaces map well to OCR objects, going through the Realm interfaces introduces some overhead. Realm supports waiting within a task, whereas OCR generally prohibits the use of application waits within a task, requiring the use of a continuation-passing-style of code which is convoluted for humans to use. We need to extend OCR to support waiting efficiently within a task without blocking.

In the second area, we implemented the first known PGAS implementation version
of JavaScript based on the Node.js JavaScript runtime environment and an OpenSHMEM PGAS library. We provided asynchronous JavaScript versions of the PGAS communication operations to facilitate hiding of communication latency. To do so, we extended two well known non-blocking mechanism used in JavaScript, Callback and Promise, so that they can be used in the PGAS context. To implement this approach, we used the HClib AMT library to add a communication worker to perform the communication operations while the Node.js runtime makes progress with other tasks. We also used N-API to interface between the JavaScript runtime and the C/C++ communication module. To evaluate the performance of asynchronous versions of the communication APIs, we used various microbenchmarks and HPC benchmarks. The results show that introducing asynchrony adds some overhead, but also helps to hide the communication latency. This overhead points towards the need to batch communication operations together in the future, rather than doing very fine grain task creation for each communication operation.

Another direction for future work is to explore the sharing of data between JavaScript and C++. In the current implementation, we allocate all distributed data in C++ managed memory space and fetch each element and pass it to JavaScript when needed. Another direction is to automatically select between synchronous and asynchronous methods based on the overhead vs communication delay. For example, in case two PEs are on the same node, then performing a shmem_get() synchronously and returning the value will be faster than creating a promise/callback. But if the PEs are communicating over a network, then asynchronous version might be better. Yet another direction is to explore the Computation vs Communication balance for an application. This will help to select the number of node.js instances vs number of communication tasks within a node to deliver maximum performance. The balance could change according to the communication usage of the application under consideration. The current implementation uses a one-to-one mapping,
with one communication worker for each Node.js instance.

In the third area, we explored how to extend an AMT runtime to include various resiliency features. The traditional checkpoint/restart (C/R) approach for resilience was designed to support the bulk-synchronous MPI programming model under the assumption that failure is a rare event. It targets hard failures, where detection is relatively straightforward through the MPI runtime or HPC middleware, and recovery involves restart on alternate hardware components. However, C/R is not well suited for supporting higher-frequency soft errors or unexpected performance anomalies. The resilient-AMT idea for applications mitigates the shortcoming of traditional C/R, so as to support scalable failure mitigation in a "local-fail local-recover" model. Task decomposition allows localization and isolation of failures in the resilient-AMT framework, and thus keeps the recovery inexpensive. Our work realizes the four resilience programming concepts suggested by Heroux [113]. Task helps to perform **Local Failure Local Recovery** for scalable application recovery. The task replication and replay APIs allow **selective reliability**, as demonstrated in Figure 5.3; the use of replication and replay on individual tasks can be at the user’s discretion. The task replay and ABFT APIs enable **skeptical programming**, which can incorporate inexpensive user-defined error detection. The response to an error is either task replay (rollback) or recovery (application-specific correction). The AMT execution model **relaxes the assumption of bulk-synchrony** of conventional parallel programs.

An interesting direction for future research is to extend our resilient-AMT approach to support both intra-node and inter-node resilience for MPI+AMT (or equivalent). One approach is to encapsulate message passing calls in a task so that their resilience is managed by task replay or replication. Another direction is to combine both replication and replay mechanisms in an “eager replay” approach. During eager replay, if extra resources are
available, the replay task can run multiple copies instead of waiting for the task to finish and perform error checking. Once the tasks finish, we can select the correct output from the replicas using a selection function. Our current approach also depends on the use of a user-provided equals function to check for equivalence of data. However, a compiler should be able to automatically generate these equivalence operators for many data objects, which is also a promising direction for future work. Finally, our current approach supports one level of checkpointing, by only providing access to checkpoints of parent tasks. If that execution fails again, we may need to support recovery from checkpoints of further ancestors, as part of future research. This would involve the re-creation and execution of an already completed section of the task graph. Also, we can look at the structure of the task graph to decide on tasks that are more important for resiliency purposes.

In summary, we have shown that AMT runtimes can be used to bridge the gap between high-level programming models and complex low-level hardware. For this, we used a high-level PGAS programming language: Chapel and a high-level data-centric programming model: Legion. We abstracted the complexities of the hardware using two AMT runtimes: OCR and HClib. Further, we demonstrated the use of event-based asynchronous mechanisms to hide the widening gap between computation and communication-latency. We used JavaScript to enable high-level event-driven asynchronous execution and Open-SHMEM PGAS communication library to perform communication across nodes. Finally, we show how to extend an AMT-based runtime, HClib to include various resiliency method to mitigate transient faults. By addressing multiple extreme-scale hardware challenges using AMT-based software layers, we demonstrate the importance of AMT runtimes as we progress towards the exascale era and beyond.
References


[31] Chapel, “a productive parallel programming language.”


[34] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, “Introducing openshmem: Shmem for the pgas community,” in *Proceedings of the...*


[73] node-addon api, “node-addon-api module.”

[74] H. Challenge, “Randomaccess gups (giga updates per second).”


Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.


