Reconverse: A new scheduling library for multithreaded task-based runtime systems

Ritvik Rao*, Jiakun Yan†, Aditya Bhosale*, Laxmikant Kale*
* University of Illinois Urbana-Champaign, Urbana, IL, USA
{rsrao2, adityapb, kale}@illinois.edu
† Advanced Micro Devices, Bellevue, WA, USA
jiakun.yan@amd.com

***Abstract*—Task-based runtime systems, including Charm++, provide benefits not available with MPI, such as dynamic load balancing and automatic computation-communication overlap. Charm++ is currently built on top of an old scheduling layer called Converse. However, Converse lacks many features that are essential for distributed multithreaded performance, and other programming models cannot directly use the Converse scheduler. To solve these problems, we present Reconverse, a new scheduling layer for task-based runtime systems. Reconverse is a separately compiled library that can be linked to Charm++ or other runtime systems. Compared to Converse, Reconverse contains experimental features for optimizing fine-grained applications such as lottery queue scheduling. Reconverse also supports new machine layers such as LCI, a new communication layer for multithreaded runtime systems. We show that Reconverse improves performance compared to Converse on multiple benchmarks and also demonstrates speedups of up to 2x on large-scale applications, showing the benefits of Reconverse for multithreaded distributed workloads.**

***Index Terms*—Task-based computing, runtime systems, task scheduling, high-performance computing, parallel processing**

I. Introduction

The most common programming model for writing large-scale parallel programs is the Message Passing Interface (MPI). However, MPI has some shortcomings, such as a fixed rank model and a requirement to explicitly specify communication. There are many alternatives to MPI that seek to address these issues. One such model is Charm++ [1], a long-standing programming model designed around objects called *chares* that can be migrated between physical processing elements (PEs). Chares allow for a program to be decomposed independent of the number of PEs, and migration allows for many powerful features such as dynamic load balancing and fault tolerance. Each chare is a C++ class with its own data and methods that can be called by any other chare. Calls to these *entry methods* are sent between PEs as messages, which are enqueued by the receiving PE and executed at some point. Charm++ programs can be described as the creation and processing of many messages, which makes Charm++ one of many *task-based runtime systems*.

The Charm++ software stack can be divided into three layers. The top layer contains Charm++ itself, which describes chares and load balancing strategies. This top layer also supports derivatives of Charm++, such as Charm4Py [11] (an implementation of Charm++ in Python) and Adaptive MPI [6] (a library for virtualizing MPI ranks). The lowest layer contains many communication libraries that Charm++ supports, including IB Verbs and UCX. The layer in between the runtime and the communication layer is called *Converse* and is responsible for scheduling tasks on each PE, while serving as the start and end point of any communication. Converse was developed over 30 years ago and has been in continuous use since then [13]. In this paper, we explore the existing state of Converse, its current issues, and the motivation behind creating a new scheduling layer. Then, we introduce and evaluate *Reconverse*, which serves as both a new task scheduling library for Charm++ as well as an open-source library that may be used by other task-based runtimes.

This paper makes the following contributions:

- The introduction of Reconverse as a common abstraction for implicitly scheduled task-based runtime systems
- New techniques for efficiently using scheduler queues for fine-grained parallel programs
- An evaluation of the performance of Reconverse on multiple relevant benchmarks, along with performance results on two large-scale message-driven applications

Reconverse is both a re-implementation and a redesign of Converse. Rebuilding the scheduler and communication paths as a standalone library is largely an engineering effort. The new capabilities that this restructuring enables, and that old Converse does not provide, are the runtime selection of a communication library, the treatment of network progress as another scheduler queue, and the registration of queues with polling frequencies that adapt during execution.

This work was performed while J. Yan was a Ph.D. student at the University of Illinois Urbana-Champaign.

This work used Delta at NCSA and Anvil at Purdue through allocation ASC-050025 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by U.S. National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296. This research used the Delta advanced computing and data resource which is supported by the National Science Foundation (award OAC 2005572) and the State of Illinois. Delta is a joint effort of the University of Illinois Urbana-Champaign and its National Center for Supercomputing Applications. This research used the Anvil machine at the Rosen Center for Advanced Computing at Purdue University [17].

```
#include "converse.h"

struct HelloMsg { CmiMessageHeader header; };

CpvDeclare(int, helloId);

// Runs when the message is delivered to a PE.
void helloHandler(void *msg) {
  CmiPrintf("Hello World from PE %d of %d\n",
            CmiMyPe(), CmiNumPes());
  CmiFree(msg);       // handler owns the message
  CsdExitScheduler(); // this PE is done
}

// Executed once per PE at startup.
CmiStartFn myMain(int argc, char **argv) {
  CpvInitialize(int, helloId);
  CpvAccess(helloId) =
      CmiRegisterHandler(helloHandler);

  if (CmiMyPe() == 0) {
    HelloMsg *m =
        (HelloMsg *)CmiAlloc(sizeof(HelloMsg));
    m->header.handlerId = CpvAccess(helloId);
    m->header.messageSize = sizeof(HelloMsg);
    CmiSyncBroadcastAllAndFree(sizeof(*m), m);
  }
  return 0;
}

int main(int argc, char **argv) {
  ConverseInit(argc, argv, (CmiStartFn)myMain);
  return 0;
}
```

Fig. 1. A "Hello World" program in Reconverse. Each PE registers the same handler at startup; PE 0 broadcasts a single message, and the runtime's scheduler invokes `helloHandler` on every PE.

II. Overview of Converse

Converse is a layer that handles all task scheduling for Charm++ programs. It is possible to write fully functional parallel programs by directly calling Converse API functions, but Converse was primarily conceived as a way to support higher level programming models, particularly Charm++. Therefore, Converse is a part of the Charm++ codebase and currently not separately compilable from Charm++. A Converse program consists of one or more processes, each of which has one or more PEs that are implemented as POSIX threads, each anchored to a core. Any Converse program with two or more PEs per process is running in symmetric multiprocessing (SMP) mode and communication for all threads on the process is handled by a special communication thread. As shown in Figure 1, a Converse program begins with a call to ConverseInit (called once on each process) that launches one thread per PE and a communication thread (if applicable). Converse then starts a scheduler and triggers a start function for each PE.

A. Tasks

In Converse, a task is defined as some unit of work that is executed asynchronously on one PE without any blocking. Tasks are triggered by messages, and tasks often create more messages to launch new tasks. Tasks come in multiple forms. The most common type of task is a handler, which is just a C++ function. Fig. 1 shows `helloHandler` as an example. In `CmiStartFn`, all handlers must be registered and assigned a handler number. If a message is created and assigned that handler number in its header, the scheduler will call the C++ function.

One limitation of regular handlers is that they must be non-blocking, and attempting to add synchronization in a handler results in undefined behavior. To solve this problem, Converse supports user-level threads (ULTs), which contain their own stack and are implemented with an outside library such as boost-context (used for Reconverse) or QuickThreads (used for old Converse). When a handler is called in a ULT, the thread can be suspended with a call to `CthYield` and woken with `CthAwaken`. This allows users to add explicit synchronization in Converse and Charm++ programs. User-level threads also support certain Charm++ features to improve programmability, such as structured dagger [12]. An alternative to ULTs is to use stackless C++ coroutines, which can be wrapped into messages and enqueued into the scheduler.

Another type of task is a callback. Callbacks are messages that are automatically enqueued when a certain condition is met. One example is a timer-based callback, which enqueues a message after a certain amount of time has passed since the callback was created. Another is enqueued when a kernel completes, such as a GPU kernel, which is how Converse supports GPUs in a task-based programming model. Callbacks are also used to contribute to collective communications.

B. Communication

After startup, each PE communicates with each other using Converse messages. A Converse message conceptually represents a task and has two parts: a header that includes information such as the destination PE and message size, followed by a field for message data. Messages are sent from one PE to another (or from a PE to itself since a message just represents a ready task), to be *enqueued* (see §II-C) at the destination; Converse provides an API to enable these sends. Since Converse is designed for asynchronous communication, all sends are non-blocking and return immediately. The most basic send is CmiSyncSend, which sends a message to one destination PE. However, other send functions are available, including sends to processes (with messages that can be executed by any PE on that process) and asynchronous collective communication functions such as broadcasts and reductions. The communication mechanism itself is handled by a lower-level communication layer, but Converse is the beginning and end point of all communication, so Converse defines communication primitives.

C. Scheduler and queues

When a PE receives a message, the message is placed in one of many queues that exist on the PE. Queues are used as a way to control the order in which messages are processed by a PE. They also help minimize scheduling overhead by preventing the unnecessary use of locks and atomics (for

example, creating queues dedicated for messages from a PE to itself). Examples of Converse queues include a queue for PE-specific messages and a shared process-level queue for messages intended for a process (which any PE on that process may execute). Each queue has its own enqueue and dequeue logic, such as FIFO, LIFO, or priority-based queuing. These queues are accessed by a scheduler, which is an infinite loop that runs on each PE during the entire duration of a Converse program. During each scheduler loop iteration, each queue is checked in a specific hard-wired order. When a queue is checked, if it is empty, the scheduler moves onto the next queue. If a message is found, it is executed: the scheduler looks up the handler associated with the handler number in the message header, then calls it with the message data as its argument. When the handler is finished executing, control returns to the scheduler. The scheduler loop runs forever, even if all the queues are empty, and the loop can only be stopped by a program abort or by a message that exits the loop. When the loop exits, the Converse program is over and cleanup code is run before termination.

D. Interface with communication layer

Converse uses lower-level communication libraries to send messages between processes. Many such libraries are supported by Converse and Charm++, including IB Verbs, OFI, UCX, and even MPI. For workstations connected by ethernet (the old "Networks of Workstations" mode), or for development on a laptop, inter-process communication can be done via UDP, and if multiple processes are not needed, there is a "multicore" layer for shared-memory communication. All of these communication layers have different APIs and capabilities, along with configurable options such as thresholds for eager vs. rendezvous communication and memory pool sizes. To maximize portability and performance, Converse interfaces with all supported communication layers with the Low-Level Runtime System (LRTS) layer. The LRTS is intended to be a software layer that supports many different communication libraries, including UDP, Cray's uGNI, and IBM's PAMI. Historically, new supercomputers have often come with new interconnects and communication libraries, and while these libraries are well optimized for particular hardware, their software interfaces have little in common with other libraries on other machines, hurting portability. The LRTS implements portability not with a common API that Converse can use, but by including direct calls to these libraries from Converse, while choosing which code to call based on the choice of library when building Charm++ and Converse. Over time, LRTS has grown to include many files with complex code paths and common vs specialized functionalities, complicating attempts to debug and maintain the software.

III. Issues with Converse

Although Converse has supported Charm++ for over 30 years, there are shortcomings in the current implementation that motivate the creation of an alternative.

A. Performance

Converse introduces various overheads compared to using MPI or a low-level communication library. These overheads include the amount of time spent in the scheduler (and therefore not executing tasks) and the time spent copying messages to and from data structures, such as queues, during communication. A small amount of overhead is acceptable because the abstraction provided by Converse empowers runtimes such as Charm++ which have sophisticated features for parallel applications. However, after decades of organic evolution, the current form of Converse is not the best fit for modern hardware and compilers. For most of the first decade of Converse, nodes on supercomputers each had a single CPU core. Large multicore NUMA nodes necessitate new scheduling and communication techniques for applications with many threads per process; Converse exhibits various inefficiencies in this area, such as requiring all threads to funnel all inter-process communication through a dedicated per-process communication thread. Removing such inefficiencies would amount to fundamentally redesigning Converse, so it makes sense to replace Converse.

Furthermore, many queues and other data structures in Converse were written against older C++ standards and would benefit from modern replacements, but doing so would require rewrites so extensive that they would take the same effort as writing a new scheduling layer.

B. Usability by other runtimes

Converse is intended to be a lightweight software layer that supports messages, message queues, and scheduling based on those queues. Ideally, a new runtime that requires this functionality could take advantage of the existing Converse code without developing another scheduling layer from scratch. However, over time, Converse has become tightly coupled with Charm++ and cannot be compiled separately. Building and using Converse requires building Charm++. This prevents Converse from being used by other runtimes, which ties the future maintainability of Converse to Charm++. Any new vendor or developer of a communication library who wishes to demonstrate or optimize their layer for Charm++ (or Converse) applications has to deal with the whole Charm++ stack, rather than Converse, which is the communication substrate. Therefore, it is better to write a replacement for Converse that exists as a separate repository that can be independently compiled into a library that can be linked into Charm++ or any other parallel runtime, and that can be tested with any lower level communication library. A separate scheduling layer also improves the development of higher-level runtimes by guaranteeing good scheduling performance, allowing runtime designers to focus on designing novel high-level abstractions.

C. Software complexity

The current implementation of Converse includes over 40,000 lines of code, the majority of which are in LRTS files. Some of this code includes support for deprecated features, including communication libraries that are no longer in use

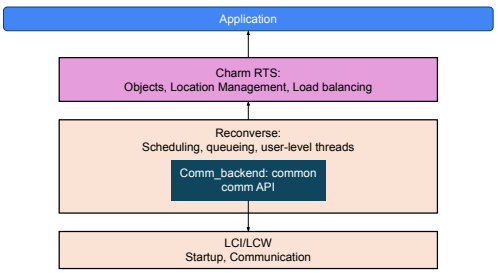

Fig. 2. The Charm++ software stack with the inclusion of Reconverse. The arrows represent parts of the stack with an interface that needs to be implemented by another layer. For example, LCI and LCW must conform to the `comm_backend` interface, and Charm++ must conform to the Reconverse API.

and experimental Converse features that are never enabled by any working Charm++ build. Exacerbating this issue is a lack of documentation that describes changes over time. This has led to a lack of a detailed understanding of the Converse code base among current developers. Very few changes to Converse have been made in recent years, due to the fear of introducing new bugs, resurrecting old failures, or inadvertently disabling certain optimizations. In short, it is difficult to maintain Converse as is, and it would be preferable to replace Converse with a new scheduling layer.

IV. Overview of Reconverse

Reconverse is a new scheduling layer for implicit tasking runtime systems. While the initial motivation to create Reconverse was to serve as a replacement for Converse in the Charm++ software stack, Reconverse is also designed for any runtime designed around execution by creating and processing messages via scheduling queues.

A. Software separation

Reconverse is implemented as a separate repository that can be compiled and run without the support of any higher-level runtime, similar to how the Realm [18] scheduling library can be run independently of Legion [5]. Any parallel runtime that uses implicit asynchronous message-driven execution can use Reconverse as its scheduling layer, allowing software designers to direct efforts on higher-level designs instead of dealing with the minutiae of scheduler design. Existing runtimes such as Charm4Py can be rewritten to interface directly with Reconverse, resulting in a simpler software implementation with potentially better performance. Furthermore, if Reconverse is adopted by other runtimes, more developers will contribute to Reconverse, which increases the likelihood that Reconverse will be actively maintained long-term.

B. Communication libraries

Reconverse is supported by improved communication layers and an improved interface with those layers. The primary communication library compatible with Reconverse is the Lightweight Communication Interface (LCI) [22]. LCI is a new communication library that contains optimizations for asynchronous multi-threaded runtimes. In particular, LCI provides data structures such as message completion queues and *devices* (sets of low-level network resources) that are accessed with atomics instead of locks. A multithreaded runtime should allow many (or all) threads in a process to communicate directly with other processes, but this was previously impractical with old Converse because of contention on locks that protected access to communication operations. Converse delegates all inter-process communication to a dedicated per-process *comm thread*, serializing communication for a process and sacrificing CPU cores just for communication. With LCI atomics, Reconverse has no comm thread, allowing threads to concurrently send and receive messages across processes while using all CPU cores for computation. LCI allows scheduling layers such as Reconverse to have fine-grained control of message flow, such as controlling which threads poll the network for message progress and how often this polling occurs. This allows for the use of wider processes (running with more threads per process and fewer processes for a given number of cores), further improving performance by replacing inter-process communication with shared memory communication.

Most modern HPC networks support remote direct memory access (RDMA) communication, in which the NIC transfers data directly between registered memory regions on two nodes, without the remote CPU participating. For large messages, an RDMA-based active message send has lower latency than regular sends because it avoids extra memory occupancy and copying costs. Because of this benefit, Reconverse supports a flexible RDMA API. RDMA requires several handshake (control) messages. However, many computational science and engineering (CSE) applications are iterative with fixed (or heuristically bounded) message sizes. In such cases, and when the application logic can guarantee the safety of the send and receive buffers, some of the control messages can be avoided. Reconverse supports a "persistent" message API for this purpose, which sets up a communication ahead of time and allows its repeated reuse.

Reconverse also contains a new communication backend, simply referred to as `comm_backend`, to replace the LRTS system in Converse. Compared to the LRTS, the `comm_backend` removes most of the headers and configuration files, and simply requires a small set of functions to be implemented for each additional communication layer. These functions include methods to initialize and exit each library, query methods to get process numbers, functions to send one-sided messages (supported by RDMA where available), and memory registration. Reconverse only makes calls to `comm_backend` functions, not functions specific to a communication layer. This makes it possible to compile Reconverse with multiple communication layers and select which one to use at runtime with a command-line option, without having to rebuild Reconverse. This is a new capability that does not exist in Converse, which helps for easier

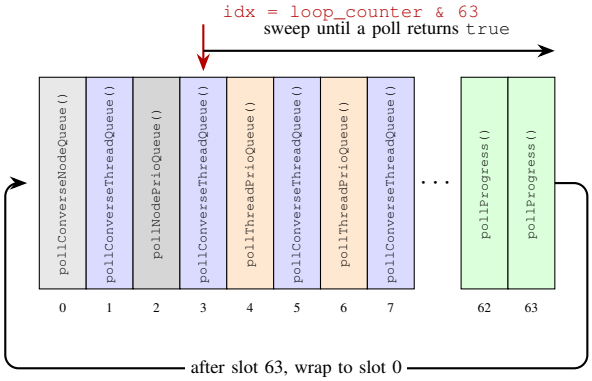

Fig. 3. The lottery-style scheduler table in Reconverse. Each poll handler is registered with a relative frequency and is awarded a proportional share of the table's 64 slots. With the default frequencies $(16, 16, 4, 1, 1)$ the thread queue and thread priority queue receive 27 slots each, network progress 6, and the node queue and node priority queue 2 each. On every iteration of `CsdScheduler`, the loop counter picks the starting slot and the scheduler sweeps forward one slot at a time, invoking that slot's poll function, until one of them reports that work was done.

performance testing and debugging of the `comm_backend` and communication layers. Along with LCI, Reconverse also supports LCW, a lightweight communication wrapper that exposes commonly used communication substrates, such as MPI, through an active-message interface. LCW helps Reconverse run on machines where LCI is not yet supported, and also helps with identifying LCI bugs (by running programs in MPI via LCW and comparing those runs to LCI).

C. Managing multiple queues

The core function of Reconverse is to schedule the execution of messages on each PE. Both Reconverse and Converse include a set of queues that handle various types of messages. The list of currently available queues is:

- A process-level message queue. This queue is a FIFO queue that handles messages intended for a process, but not a particular PE within that process. Any PE may pull a message from this queue and execute it. This is typically used for remote messages that need a quick response.
- A PE-level message queue. For Reconverse programs, this is the most commonly used queue. It handles messages intended for a single PE.
- A process-level priority queue. This is not used by Reconverse programs, but is intended for programs written in higher-level runtimes. For example, a Charm++ program to do state-space search may benefit from this queue.
- A PE-level priority queue. This works the same way as the process-level priority queue, but for messages intended for a single PE.
- An abstracted network progress queue. This abstraction is provided by the `comm_backend` layer, where any PE can call the progress() function on the network queue on a process. In the Reconverse model, we can consider the queue of network events to be just another scheduler queue, except messages are forwarded to another queue instead of being executed immediately.
- A Cilk-style stealable task queue. One such queue exists on each PE. If all other queues are empty, a PE can steal a message from the queue of another PE on its process, providing a primitive method of load balancing.

The current Converse approach is to poll all these queues in a predetermined order hardwired in the scheduler code, select a message from the first queue with at least one pending message, and execute that message. However, in practice, the usage of each queue differs by application. Certain applications may rely more on direct communication between PEs, using the PE-specific queues more. Other applications may rely more on process-level messages. The priority queues mentioned above have no use in standalone Converse and Reconverse programs, and some runtimes may choose not to use them. There is a cost to checking empty queues for messages, especially those queues that are protected by atomics or locks. Therefore, it makes little sense to have a static scheduler with the same polling logic for all applications. Modern task-based systems emphasize the utility of fine-grained tasks [16]. For such applications where each task is executed in no more than a few microseconds, scheduler overhead is a more important part of execution time. The scheduler should select and process messages as fast as possible and time inside the scheduler loop should be minimized.

These issues motivate the creation of *queue registration* in Reconverse. All Charm++ and Converse programs must use the static Converse scheduler, which checks all queues. However, Reconverse is designed to arbitrarily register and de-register any of the available queues. When a Reconverse program starts up, the runtime iterates through a list of queues and only initializes the queues in that list. Any queue can be trivially added and removed from that list without changing any other code. Currently, there are limitations to this: The list of queues must be set directly in the Reconverse code, requiring a rebuild of Reconverse to change queues. Also, certain queues may be required for basic functionality. The PE-level FIFO queue is required for basic control communication such as abort messages and clock synchronizations. Nevertheless, the other queues may be removed as needed, particularly the process-level queues protected by serialization.

Along with selecting queues, Reconverse allows polling such queues at varying frequencies. The scheduler maintains a counter for each queue, incrementing that counter when a message is pulled from that queue. Over time, these counters reflect the utility of each queue, and Reconverse uses the counters to adjust how frequently each queue is polled. This adaptive approach to queue management is enabled by a *lottery scheduler*. The traditional Converse scheduler consists of a series of if statements that resolve to true if a particular queue has a message. In each scheduler loop, each of these statements is checked until a queue has a message or there are no messages (proving that the PE is idle). Reconverse replaces these with a single table with multiple slots, each of which contains a function call to poll a particular queue.

To set up queues correctly, each queue is registered at startup with a given *relative frequency*, which is simply a positive integer. For example, if one queue has a relative frequency of 1 and a second queue has a frequency of 16, the second queue will be polled 16 times more frequently than the first queue. Along with relative frequency, each queue must include a polling function that contains logic for how to poll the queue, and this function must return a boolean that is true if a message was found in the queue. When all queues are registered, the polling functions are inserted into a table (Fig. 3) with a fixed number of slots, currently 64 slots. The relative frequencies provided are added together and normalized based on the number of slots. Each slot is then filled with a polling function for a queue, and queues that are set to be polled more frequently will be assigned to more slots, spreading them in the table so that the calls to the same queue are not bunched together. When the scheduler is started, each loop increments a loop counter, and the least significant bits of that counter provide an index into the queue table.

While this design enables adaptive scheduling, there are a few concerns. One issue is when to change polling frequencies. The table's slots must all be reassigned to properly adjust frequencies for each queue, so the scheduler must decide when to do this. We chose to hardcode this at 10 full loops around the table. Another choice is the number of slots in the table. More slots allows for more fine-grained frequency changes, while fewer slots saves memory. We have also hardcoded this value at 64 slots, and experimenting with this is a future area of research. Finally, the use of a lottery scheduler changes the logic of when to declare that a PE is idle. Determining that a PE has no messages in its queues is necessary for a variety of reasons, including quiescence detection and idle-based callbacks. It is easy to determine idleness with the traditional Converse scheduler, but more challenging with a lottery scheduler since not all queues are checked in one scheduler loop. The current solution is to declare a PE idle when every slot in the table is consecutively accessed without yielding any messages. Although reasonable in the context of the table-driven scheduler, this design is somewhat sub-optimal because individual queues can occur in multiple slots, which may be optimized in future.

D. Interface with higher-level runtimes

The previous subsections describe the required interface between Reconverse and communication libraries beneath it. The interface that a runtime must implement to use Reconverse is similarly small, and Fig. 1 shows most of it. A runtime registers each of its entry points with `CmiRegisterHandler`, which returns a handler number. To create work, the runtime allocates a message with `CmiAlloc`, writes a handler number and the message size into the `CmiMessageHeader` at the front of the message, and then either sends the message with a function such as `CmiSyncSend` (to a PE) or `CmiSyncNodeSend` (to a process), or enqueues it locally with `CsdEnqueue`. Reconverse decides when the message runs and calls its handler. Reconverse provides calls for creating, suspending and awakening (i.e., marking them as ready) user-level threads (ULTs) as well as synchronization constructs based on ULTs such as *Futures*. It supports per-PE and per-process variables in which a runtime keeps its own state, condition-based callbacks, and query functions such as `CmiMyPe` and `CmiNumPes`.

The assumption behind this interface is that the runtime above creates ready tasks and assigns tasks to a PE/process, and Reconverse chooses the order in which tasks run on each PE. This suits runtimes which manage task dependencies themselves, exposing tasks to Reconverse only when they are ready. Reconverse mostly preserves the Converse API, simplifying porting Charm++ onto Reconverse, confining changes mostly to the Charm++ build system.

V. Results

A. Experimental setup

The results presented are divided into three categories: microbenchmarks, which measure the communication performance of Reconverse compared to old Converse; scheduling benchmarks, which measure the performance of the Reconverse scheduler and quantify the effects of lottery scheduling; and applications, which are larger-scale scientific workloads written in Charm++ that demonstrate the real-world performance of Reconverse.

The microbenchmarks and scheduling benchmarks were run on NCSA Delta, while the applications were run on Purdue Anvil. CPU nodes on both machines have 2 sockets with 64 cores each, for a total of 128 cores per node, along with 256 GB of memory, using AMD EPYC 7003 processors on Delta and EPYC 7763 processors on Anvil. Delta nodes are connected with Cray's Slingshot 11 interconnect, which has a 200 Gbps bandwidth, while Anvil nodes are connected via a 100 Gbps InfiniBand interconnect.

On Delta, the old Converse implementation uses cray-mpich version 8.1.32, while Reconverse uses LCI built on libfabric with its CXI provider. On Anvil, old Converse uses OpenMPI version 4.0.6, while Reconverse uses LCI built on libibverbs.

Each reported measurement is a median over repeated runs: 5 repetitions for the ping-pong and ping-ack microbenchmarks, 7 for each Task Bench point, 3 for the benchmarks that compare polling strategies, and 5 for each application at every node count on each layer.

B. Microbenchmarks

The first microbenchmark in this section is ping-pong, which measures the end-to-end latency of active messages between PEs. In ping-pong, messages are repeatedly sent back and forth between PEs on different processes for 1000 iterations. Figure 4 shows the performance of ping-pong on 1 physical node between 2 processes, each with 1 thread. Message bandwidth was measured between the existing Converse implementation and three Reconverse implementations: one with regular Reconverse sends using LCI active messages, one with RDMA-enabled sends, and one using the Persistent API in Reconverse combined with RDMA. Higher bandwidth

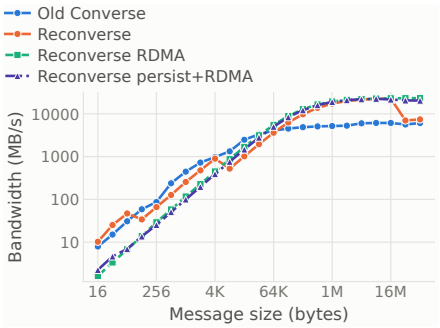

Fig. 4. Ping-pong benchmark on 1 physical node, 2 processes, 1 PE/process

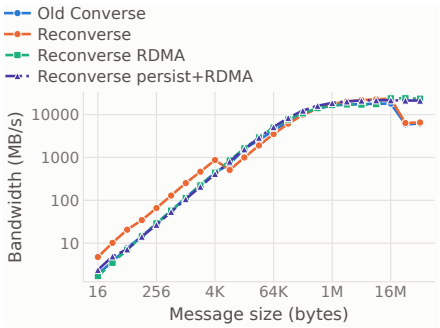

Fig. 5. Ping-pong benchmark on 2 physical nodes, 2 processes, 1 PE/process

is better, so for messages below 8 KB, Reconverse without RDMA outperforms both RDMA implementations. This is because at these message sizes, LCI active messages use the eager protocol, which sends messages immediately. For larger messages, RDMA was a faster path compared to the LCI rendezvous protocol, which requires an acknowledgment from the receiving PE before initiating a send. The old Converse results use cray-mpich on Delta, which comes with POSIX shared memory communication for all message sizes. This is why for message sizes from 64 to 8K bytes, old Converse is best. For very small messages, LCI also supports POSIX memory, so Reconverse is the best, and for larger messages, old Converse requires more expensive message copies compared to the Reconverse implementations. Implementing POSIX memory

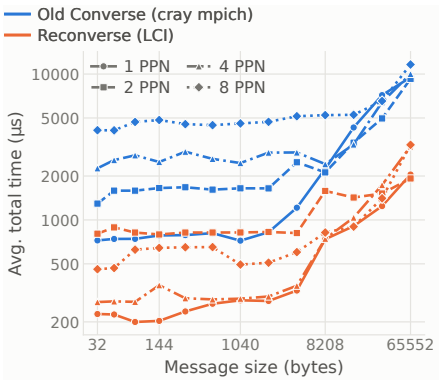

Fig. 6. Ping-ack benchmark on 2 physical nodes, 2 processes, 1-8 PEs/process

support for Reconverse to improve intra-node inter-process communication is a work in progress. Fig. 5 shows the same ping-pong experiment across 2 physical nodes. Some form of Reconverse is always equal to or better than old Converse. Reconverse without RDMA is the best for smaller messages due to the eager protocol, while RDMA beats the rendezvous protocol for larger messages.

The second microbenchmark is ping-ack, which simulates a communication pattern common in very fine-grained applications with small message sizes. In ping-ack, there are two processes, one on each physical node, with one or more threads/PEs per process. In each iteration, each PE on the sending process sends 100 messages to a paired PE on the receiving process, which sends an acknowledgment back to PE 0 once all 100 have arrived. The test ends when PE 0 has received every acknowledgment, and the elapsed time is recorded. The ping-ack results in Fig. 6 above compare Reconverse and old Converse across varying message sizes and PE counts. For each configuration, we take the average of 100 ping-ack iterations. In every configuration, Reconverse performs significantly better than old Converse, with the difference in performance being greater for smaller messages. The Reconverse speedup is up to 10x that of old Converse for smaller messages of 1 KB or less when running with 8 PEs, which shows that Reconverse is much more suitable for multithreaded fine-grained applications than old Converse. This improvement is enabled in part by reduced contention for communication resources, since Reconverse does not need to use a communication thread for an entire process. Another factor is that LCI supports the effective concurrent use of multiple sets of network resources, eliminating contention for network queues and other data structures.

C. Scheduling benchmarks

The first scheduling benchmark is Task Bench [16], which evaluates parallel runtime systems on a variety of metrics. One of the most important characteristics of a runtime is the minimum effective task granularity (METG). In fine-grained applications, tasks can execute so quickly that the amount of time needed to launch a task is greater than the length of the tasks themselves. Once the utilization of compute resources (in FLOPs) falls below 50%, there is no real reduction in the length of tasks even if they get smaller. The METG is the task granularity at the 50% mark, and the lower it is, the better. Fig. 7 shows the efficiency curves for a 1-D stencil computation on Converse, Reconverse, Charm++ (on both layers), and MPI as a baseline. Note that grainsizes become finer towards the right. On one PE, MPI consistently shows higher efficiencies for small grain sizes, with a METG that is much lower than any Charm++, Converse, or Reconverse configuration. This one PE efficiency is achieved because the MPI version of Task Bench does not rely on a scheduler. Instead, a task graph is explicitly constructed on a single MPI rank and is looped over. On larger PE counts, all Charm++, Converse, and Reconverse configurations are more competitive with MPI, even better in some cases. Additionally, Reconverse

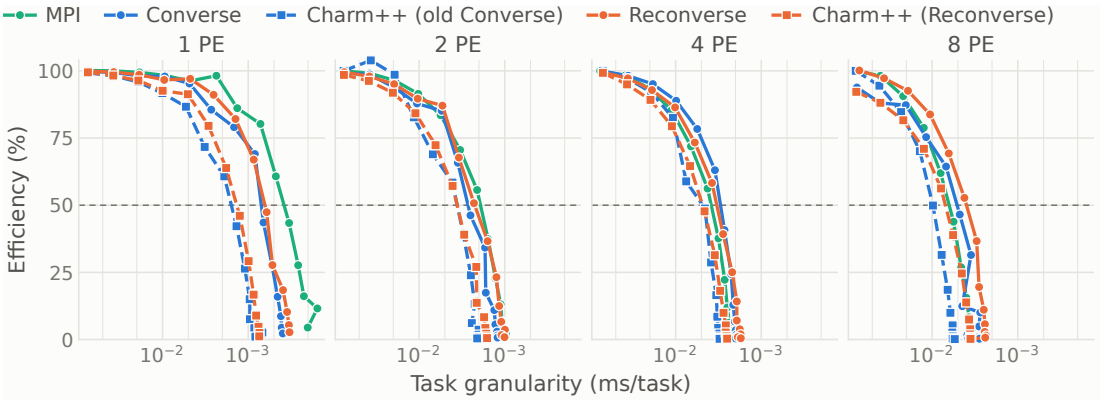

Fig. 7. Task bench efficiency of Charm++ (old Converse and Reconverse), old Converse, Reconverse, and MPI on a 1-D Stencil computation. METG is the granularity value when each curve crosses the 50% efficiency line.

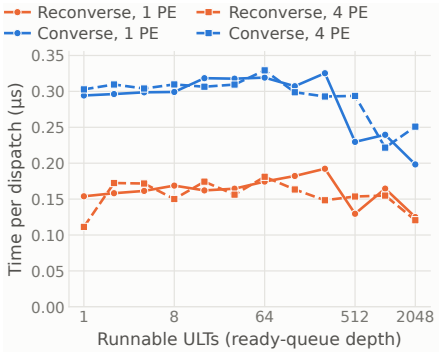

Fig. 8. Dispatch rate of user-level threads in Reconverse vs. old Converse

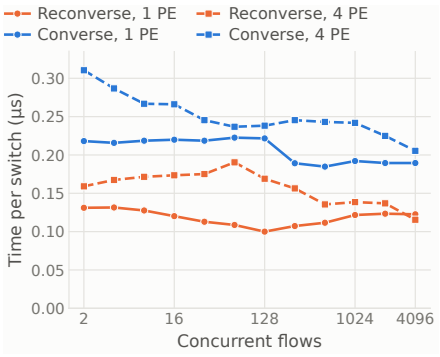

Fig. 9. Time needed to context-switch between different user-level threads

consistently has lower METG than Converse, and Charm++ has lower METG using Reconverse than when using Converse. This shows that Reconverse incurs lower scheduling overhead than old Converse for both the standalone benchmark and when supporting Charm++, and is also better than MPI for higher PE counts on one node. One important context for these results is that previous papers by the authors of Task Bench have shown Charm++ to outperform other task-based runtime systems and alternatives to MPI on METG [20].

The next set of benchmarks concerns control flow with user-level threads (ULTs). Fig. 8 shows the results of a benchmark

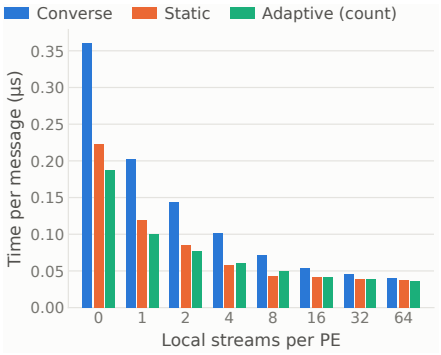

Fig. 10. Time per message for the parallel ring benchmark on Converse, Reconverse (with static scheduler) and Reconverse (with adaptive frequencies), on one process with 4 PEs. The process has a shared queue accessible by all PEs, and each PE has its own local queue that can only be processed by that PE

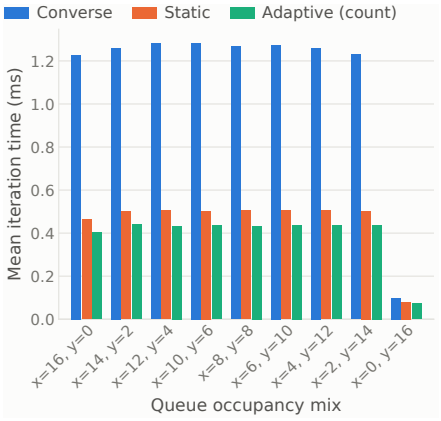

Fig. 11. Time per message for the shared queue vs. local message benchmark on Converse, Reconverse (with static scheduler) and Reconverse (with adaptive frequencies) on one process with 4 PEs. The process has a shared queue accessible by all PEs, and each PE has its own local queue that can only be processed by that PE. x is the number of shared queue messages, and y is the number of local queue messages. The benchmark generates messages on each PE in sets of 16, with x+y always adding up to 16.

that measures the dispatch rate of ULTs, where dispatch refers to the process of awakening a suspended ULT. Lower times per dispatch are better, and Reconverse consistently dispatches faster than Converse regardless of how many runnable ULTs are created. There is also no meaningful difference in dispatch rate on 1 or 4 PEs, which is expected as each ULT is re-awoken locally. The second ULT benchmark is the cost of context switches between ULTs, which happens when one ULT yields to the scheduler and another ULT is selected to run. Fig. 9 shows that Reconverse has lower context switch overheads than Converse. This shows that Reconverse better supports programs that require multiple control flows that may be migrated between PEs during execution.

The final set of benchmarks measures the benefits of adapting queue polling frequency using Reconverse's lottery scheduler. The first of these benchmarks is a parallel ring benchmark. Every PE injects one ring message into the shared queue (so the shared queue is essentially never empty) and also maintains multiple streams of messages repeatedly enqueued into the local PE queue (so each stream message is a PE communicating with itself). The number of local streams is varied, so the more local streams, the more the local queues will be used. Fig. 10 shows how much time is spent scheduling and executing each message on average, where lower is better. Both the static and adaptive Reconverse schedulers are better than old Converse, but the adaptive scheduler shows a small improvement at most stream counts. In cases where the adaptive scheduler is worse than the static scheduler, this appears to reflect a shortcoming with adapting polling frequencies based solely on message counts: not all messages are equal. The ring that is passed around via the shared queue is in the critical path, but the volume of the queue is small at higher stream counts compared to the local queues. Accounting for the critical path in the scheduler is an area for future work. The second benchmark to measure adaptive queue polling involves each PE generating either local queue messages or shared queue messages at varying ratios. At first, all messages are placed in the shared queue at a 16:0 ratio. Over the course of the application, more local queue messages are inserted instead, until the ratio is 0:16. Fig. 11 shows a full run of the benchmark, and that during each phase, the adaptive scheduler is faster than the static scheduler. All times drop during the last phase (all local messages) as each scheduler no longer spends any time processing shared queue messages. The shared queue is protected with an atomic which is not released until a message from the shared queue has been fully processed, so not using it at all will reduce runtime for all schedulers.

D. Applications

For the best performance for Converse, scaling runs used 16 processes per node, 6 PEs per process, 1 communication thread per process, and 1 core left alone for OS processes, for a total of 96 PEs used for computation per node. For the best Reconverse performance, scaling runs used 8 processes, 15 PEs per process, and 1 core left alone for OS processes, for a total of 120 PEs used for computation per node. This means that at any given node count, the level of parallelism will not be exactly comparable, but we believe the comparison is still fair as one of the benefits of Reconverse is the removal of the comm thread and wider processes. All Reconverse application runs use the lottery scheduler with adaptive polling frequencies. We did not collect application results with the static Reconverse scheduler, so the speedups reported below do not show the amount of speedup from adaptive polling.

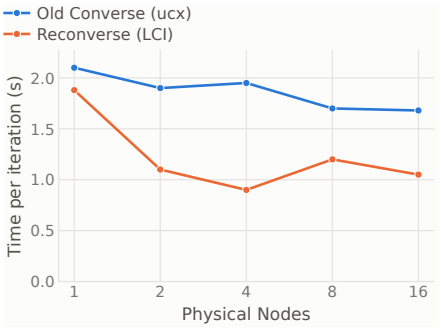

Fig. 12. Barnes-Hut strong scaling on input of 8M particles

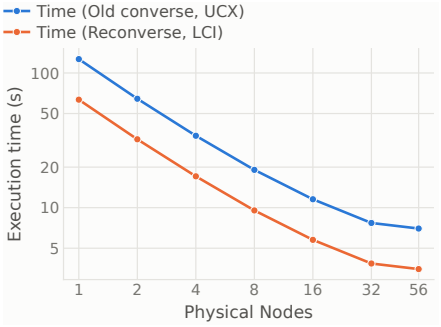

Fig. 13. ChaNGa strong scaling on input of 80M particles

The first application is a Barnes-Hut simulator written in Charm++. The Barnes-Hut algorithm [4] divides N bodies or particles into an oct-tree decomposition, which reduces the computational complexity of N-body simulations from $O(N^2)$ to $O(N \log N)$. Fig. 12 shows strong scaling results for Barnes-Hut from 1 to 16 nodes on Anvil for a dataset with 8 million particles. Running Barnes-Hut with Charm++ built with Reconverse shows up to a 2x speedup compared to Charm++ built with old Converse, with larger speedups for 2 and 4 nodes. Generally, speedups for Reconverse should be larger with higher node counts due to the higher performance of LCI and wider process widths, but running at 8 and 16 nodes is likely too large of a scale for an 8 million particle input. At those node counts, both old Converse and Reconverse do not scale to higher speedups.

The second application is the Charm N-Body Gravity (ChaNGa) simulator [14], which runs an N-body gravity simulation (along with other computations such as smooth particle hydrodynamics). ChaNGa is designed primarily for highly imbalanced inputs, and is built around a distributed tree

decomposition of particles. Fig. 13 shows the strong scaling performance of ChaNGa up to 56 nodes (5376 cores for old Converse, 6720 cores for Reconverse), with an input data set of 80 million particles. For both old Converse and Reconverse, scaling is almost perfectly linear from 1 to 16 nodes because ChaNGa uses various techniques enabled by Charm++, primarily dynamic load balancing [14]. Beyond that, the problem size is too small to benefit from the extra parallelism. Overall, Charm++ on Reconverse shows a consistent 2x speedup over Charm++ on Converse at all levels of parallelism. The primary cause of this speedup is the wider process width, which reduces communication cost, as it did with Barnes-Hut. However, the benefit is even larger for ChaNGa because ChaNGa uses a software cache (implemented as a Charm++ library called ckCache) to store commonly requested remote data such as pieces of the decomposed particle tree and related information needed to make gravity calculations. One software cache exists per process, so if more PEs share a process, the reduction in remote requests enabled by the software cache is greater than with smaller process widths. This communication reduction is why Reconverse has a higher speedup in ChaNGa than it does with microbenchmarks such as ping-pong.

VI. Related Work

Reconverse is just one practical use of the LCI communication layer. Another asynchronous many-task runtime that uses LCI is HPX [21]. HPX is a runtime system that interfaces with communication libraries through its Parcelport API. When replacing the original MPI implementation of Parcelport with LCI, HPX applications such as OctoTree (a simulation of binary star systems) show strong speedups of up to 2x, similar to the results with ChaNGa and Reconverse. HPX uses C++ worker threads to execute tasks, while Reconverse schedules user-level threads efficiently, so an HPX port onto Reconverse instead of Parcelport would be worth exploring.

Other programming systems also have factored out libraries related to task scheduling. For example, the Legion programming model [5] uses Realm [18] to implement dynamic task-graph construction, a functionality that is analogous to how Converse and Reconverse implement scheduler queues for Charm++. Like Converse, Realm has not yet seen use as a layer directly underneath a runtime other than the one for which it was originally designed. Reconverse and Realm both seek to become a standard layer to support parallel runtimes, and both can coexist: Reconverse is best suited for runtimes with implicit task dependencies, while Realm is best for runtimes with explicit dependencies represented with a task graph [20].

Two existing libraries for scheduling multiple threads on one node are Argobots [15] and Qthreads [19]. Like Reconverse, both libraries support scheduling user-level threads over physical cores or OS-level threads. Both libraries also support future objects. Reconverse instead provides the *suspend* and *wake* primitives from which such constructs are built, and Charm++ implements its own futures on top of them. Furthermore, Argobots pools are analogous to Reconverse queues, and can also be dynamically registered during startup. Reconverse differs from Argobots and Qthreads in that it is co-designed with a distributed communication backend, instead of relying on an independently developed library (such as Chapel's [9] use of Qthreads with GASNet [7]). This allows network communication (such as polling progress) to be registered and scheduled as if it were another scheduler queue, and makes the scheduler more communication-aware. Reconverse also adjusts queue polling frequencies at run time based on how often each queue yields a message, while Argobots requires the runtime above it to provide a custom scheduler to vary how its pools are accessed.

VII. Future Work

One area of additional work is to adapt other runtime systems to work on top of Reconverse. In particular, Adaptive MPI [6] and Charm4Py are currently implemented on top of Charm++, in part because Converse is tightly coupled to Charm++. Re-implementing both directly on top of Reconverse would likely improve their performance. We are also interested in exploring the feasibility of porting other runtime systems onto Reconverse, such as PaRSEC [8], Legion, StarPU [3], nOS-V [2], and DDDF [10].

Another potential area of research is improving the design of the lottery scheduler. As demonstrated in the scheduling benchmarks, the assumption that it is best to poll the queues with the most messages is not always accurate, and more factors should be considered when tuning the polling frequency of queues, such as critical-path messages. We also hope to develop ideas for new queues that do not currently exist in Reconverse that may benefit certain applications, like how priority queues can help with state-space searches, or how a queue devoted to messages from a PE to itself can be used to eliminate the use of atomics for self-messages (which would improve the performance of Reconverse on Task Bench and other fine-grained applications).

One drawback of Reconverse is the amount of manual tuning required to maximize performance, including setting packet sizes for LCI messages, determining the number of LCI devices to use, and deciding how many threads in a process poll network queues. Old Converse does not require as much tuning, and while that is in part due to the lack of features in old Converse compared to Reconverse, reducing these requirements improves usability. Algorithms should be explored to automatically tune Reconverse for maximum performance.

VIII. Conclusion

We present Reconverse, a new scheduling and communication library for task-based runtime systems. It employs LCI, a communication layer designed for multithreaded systems with active-message-style communication. Reconverse supports sophisticated high-level abstractions such as Charm++ while lowering scheduling and communication overheads relative to Converse across the benchmarks and applications presented here, showing that alternatives to MPI are practical for multithreaded distributed workloads.

References

[1] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. Parallel Programming with Migratable Objects: Charm++ in Practice. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, pages 647–658, 2014.

[2] David Álvarez, Kevin Sala, and Vicenç Beltran. nos-v: Co-executing hpc applications using system-wide task scheduling. *arXiv preprint arXiv:2204.10768*, 2022.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and computation: practice and experience*, 23(2):187–198, 2011.

[4] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446–449, 1986.

[5] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[6] Milind Bhandarkar, L. V. Kale, Eric de Sturler, and Jay Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, pages 108–117, May 2001.

[7] Dan Bonachea and Paul H Hargrove. Gasnet-ex: A high-performance, portable communication library for exascale. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 138–158. Springer, 2018.

[8] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[9] Bradford L. Chamberlain. Chapel. In Pavan Balaji, editor, *A Brief Overview of Parallel Programming Models*, chapter 6, pages 129–159. MIT Press, 2015.

[10] Sanjay Chatterjee, Sagnak Tasırlar, Zoran Budimlic, Vincent Cavé, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. Integrating asynchronous task parallelism with mpi. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 712–725. IEEE, 2013.

[11] Juan J Galvez, Karthik Senthil, and Laxmikant Kale. Charmpy: A python parallel programming model. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 423–433. IEEE, 2018.

[12] L. V. Kale and Milind Bhandarkar. Structured Dagger: A Coordination Language for Message-Driven Programming. In *Proceedings of Second International Euro-Par Conference*, volume 1123-1124 of *Lecture Notes in Computer Science*, pages 646–653, September 1996.

[13] L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[14] Harshitha Menon, Lukasz Wesolowski, Gengbin Zheng, Pritish Jetley, Laxmikant Kale, Thomas Quinn, and Fabio Governato. Adaptive techniques for clustered n-body cosmological simulations. *Computational Astrophysics and Cosmology*, 2(1):1–16, 2015.

[15] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, et al. Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):512–526, 2017.

[16] Elliott Slaughter, Wei Wu, Yuankun Fu, Legend Brandenburg, Nicolai Garcia, Wilhem Kautz, Emily Marx, Kaleb S Morris, Qinglei Cao, George Bosilca, et al. Task bench: A parameterized benchmark for evaluating parallel runtime performance. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[17] X Carol Song, Preston Smith, Rajesh Kalyanam, Xiao Zhu, Eric Adams, Kevin Colby, Patrick Finnegan, Erik Gough, Elizabett Hillery, Rick Irvine, et al. Anvil-system architecture and experiences from deployment and early user operations. *PEARC'22: Practice and Experience in Advanced Research Computing*, 2022.

[18] Sean Treichler, Michael Bauer, and Alex Aiken. Realm: An event-based low-level runtime for distributed memory architectures. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 263–276, 2014.

[19] Kyle B Wheeler, Richard C Murphy, and Douglas Thain. Qthreads: An api for programming with millions of lightweight threads. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE, 2008.

[20] Rohan Yadav, Joseph Guman, Sean Treichler, Michael Garland, Alex Aiken, Fredrik Kjolstad, and Michael Bauer. On the duality of task and actor programming models. *arXiv preprint arXiv:2508.16522*, 2025.

[21] Jiakun Yan, Hartmut Kaiser, and Marc Snir. Understanding the communication needs of asynchronous many-task systems–a case study of hpx+ lci. *arXiv preprint arXiv:2503.12774*, 2025.

[22] Jiakun Yan and Marc Snir. Lci: a lightweight communication interface for efficient asynchronous multithreaded communication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '25, page 1043–1059. ACM, November 2025.

Artifact Description (AD)

IX. Overview of Contributions and Artifacts

A. Paper's Main Contributions

- C_1 The introduction of Reconverse as a common abstraction for implicitly scheduled task-based runtime systems
- C_2 New techniques for efficiently using scheduler queues for fine-grained parallel programs
- C_3 An evaluation of the performance of Reconverse on multiple relevant benchmarks, along with performance results on two large-scale message-driven applications

B. Computational Artifacts

All five artifacts are archived together, as snapshots at the exact commits used for this paper and with a manifest of those commits, at https://doi.org/10.5281/zenodo.22698187. The development repositories below are given for reference.

- A_1 Reconverse and Charm++-on-Reconverse
  https://github.com/charmplusplus/reconverse
  https://github.com/charmplusplus/charm
- A_2 Communication microbenchmarks (ping-pong, ping-ack) and their driver scripts and raw results
  https://github.com/ritvikrao/reconverse_2026_pawatm_artifacts
- A_3 Scheduling benchmarks: Task Bench and the user-level-thread benchmarks
  https://github.com/UIUC-PPL/task-bench
- A_4 Adaptive queue-polling benchmarks
  https://github.com/ritvikrao/adaptive_bench
- A_5 Applications: Barnes-Hut and ChaNGa
  https://github.com/UIUC-PPL/barnes
  https://github.com/N-BodyShop/changa

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2	Sections III–IV Figures 2, 3
A_2	C_1, C_3	Figures 4–6
A_3	C_2, C_3	Figures 7–9
A_4	C_2, C_3	Figures 10, 11
A_5	C_3	Figures 12, 13

X. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

A_1 is the software contribution itself: the Reconverse repository holds the scheduler, the queue registration and lottery-scheduling mechanism, the user-level threads, and the `comm_backend` abstraction with its LCI and LCW backends (C_1, C_2); the Charm++ repository holds the build integration that lets Charm++ compile against either layer with no change to application code, which is what makes every layer-to-layer comparison possible. A_1 is a prerequisite of A_2–A_5.

Expected Results

Reconverse builds as a standalone library, and its test suite passes on one node and across nodes. Charm++ builds against both layers, and the same unmodified Charm++ programs run on either. Reconverse binaries accept `+backend lci|lcw` at run time, so one build runs over either communication library. This substantiates C_1: the scheduling layer is usable independently of the higher-level runtime, and a higher-level runtime can be re-targeted onto it.

Expected Reproduction Time (in Minutes)

Setup: about 45 minutes (LCI 5, Reconverse 10, and 15 for each of the two Charm++ builds). Execution: about 15 minutes for `ctest` plus a Charm++ "hello world" on both layers. Analysis: none beyond checking that the tests pass.

Artifact Setup (incl. Inputs)

Hardware: Any x86_64 or ARM64 Linux or macOS machine will build and test Reconverse; no accelerator is needed. Exercising LCI as the paper does requires Slingshot 11 through libfabric's CXI provider (NCSA Delta) or InfiniBand through libibverbs (Purdue Anvil). On a workstation, LCI runs over libfabric's shared-memory providers and LCW over any MPI.

Software:

- Reconverse (branch `main`; the adaptive queue polling of Section IV-C is on branch `adaptive-queues`).
- Charm++ `main` for the old-Converse baseline, branch `reconverse-specific-build` for the Reconverse triplets.
- LCI (branch `master`), fetched by `-DRECONVERSE_AUTOFETCH_LCI2=ON`; and optionally LCW, the MPI-based fallback.
- CMake $\geq$ 3.20, a C++17 compiler (GCC 11 or newer), hwloc, libfabric 1.15.2 with CXI on Delta or libibverbs on Anvil, and cray-mpich 8.1.32 (Delta) or OpenMPI 4.0.6 (Anvil) for the old-Converse Charm++.

Datasets / Inputs: None

Installation and Deployment:

```
cd reconverse && mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Release \
      -DRECONVERSE_AUTOFETCH_LCI2=ON \
      -DLCI_NETWORK_BACKENDS=ofi ..
make
```

`ofi` selects libfabric (used on Delta); `ibv` selects libibverbs (used on Anvil). Charm++ is built from the top of its tree with

```
./build charm++ reconverse-linux-x86_64 \
    --with-production -j16
./build charm++ mpi-linux-x86_64-smp \
    --with-production -j16
```

`--with-fetch-reconverse-dir=` points the build at a local checkout. `--with-production` is required.

Artifact Execution

T_1: build Reconverse (see README for how to autofetch LCI). T_2: build Charm++ twice, once per triplet. T_3: run `ctest`, and run `srun -n 2 tests/ping_ack/reconverse_ping_ack +pe 4` across two nodes. $T_1 \rightarrow T_2 \rightarrow T_3$.

Artifact Analysis (incl. Outputs)

The two libraries and the pass/fail status of the test suite

B. Computational Artifact A_2

Relation To Contributions

A_2 produces the microbenchmark results of Section V-B (Figures 4–6), supporting C_3 and the C_1 claim that the new communication abstraction is at least as good as the one it replaces. Ping-pong measures end-to-end bandwidth for one PE pair across the three Reconverse send paths (active message, RDMA, persistent) and old Converse; ping-ack measures the many-small-message regime of fine-grained multithreaded applications, where removing the per-process communication thread should matter most.

Expected Results

Across two nodes (Figure 5) some Reconverse path should equal or beat old Converse at every message size. Within one node (Figure 4) old Converse over cray-mpich should win from 64 B to 8 KB. Ping-ack should show Reconverse ahead in every configuration, by up to 10× at 8 PEs per process at 1 KB and below.

Expected Reproduction Time (in Minutes)

Setup: about 45 minutes, exactly the A_1 setup, since the benchmarks are test programs of those two trees. Execution: about 40 minutes of two-node allocation for ping-pong (5 repetitions × 5 versions × 2 placements × 13 message sizes) and about 20 minutes for ping-ack, excluding queue wait. Analysis: under 5 minutes.

Artifact Setup (incl. Inputs)

Hardware: Same as A_1, except that it requires at least 2 nodes of Delta or Anvil

Software: Everything under A_1, plus the driver and analysis scripts in `reconverse_2026_pawatm_benchscripts/`, Python 3.9 or newer. The benchmark sources are `reconverse/tests/orig-converse/pingpong` (active message), `rdma_pingpong`, `persistent`, and `ping_ack`, with the baselines in `charm/benchmarks/converse/pingpong` and `.../ping_ack`.

Datasets / Inputs: None

Installation and Deployment: The benchmarks are built as part of A_1. The scripts set `LCI_ATTR_PACKET_SIZE=8192`, placing the eager/rendezvous threshold at 8 KB, and `PMI_MAX_KVS_ENTRIES=512`.

Artifact Execution

T_1: allocate two exclusive nodes. T_2: run `run_pingpong.sbatch`, sweeping the four versions at two placements (both processes on one node, and one per node) across all message sizes. T_3: run the ping-ack sweep over message sizes at 1, 2, 4, and 8 PEs per process. T_4: run `analyze_pingpong.py`. $T_1 \rightarrow T_2 \rightarrow T_4$ and $T_1 \rightarrow T_3 \rightarrow T_4$, with T_2 and T_3 independent.

Two choices matter for reproduction. Every version runs at every point before any version repeats, with the order rotated each repetition: when run as contiguous per-version blocks, position in the sweep alone was worth about 25% as the sustained clock drifts down. Medians are reported because the node intermittently sits in a lower clock state.

Artifact Analysis (incl. Outputs)

`analyze_pingpong.py` reduces the raw logs (kept under `.../pingpong/raw/`) to `pingpong_results.txt`: bandwidth per (version, placement, message size) for Figures 4 and 5, and total time per (version, PE count, message size) for Figure 6. Reproduction should be judged on the ordering of the curves and the size of the gaps, not on absolute bandwidths.

C. Computational Artifact A_3

Relation To Contributions

A_3 produces the first two sets of scheduling results in Section V-C, supporting C_2 and C_3. Task Bench measures the minimum effective task granularity (METG) of multiple configurations, yielding Figure 7. The user-level thread benchmarks measure dispatch rate and context-switch cost on both layers, yielding Figures 8 and 9. The third set, on adaptive queue polling, is A_4.

Expected Results

Reconverse should show a lower METG than old Converse at every PE count, and Charm++ on Reconverse a lower METG than Charm++ on old Converse, by 8–38% on the per-task overhead floor. MPI should be well ahead at 1 PE, since its Task Bench implementation loops over an explicit task graph and uses no scheduler, and should lose that advantage at higher PE counts. On the ULT benchmarks Reconverse should dispatch and context-switch faster than Converse at every flow count, with no meaningful difference between 1 and 4 PEs.

Expected Reproduction Time (in Minutes)

Setup: about 70 minutes (the A_1 setup, plus 10 for the Task Bench `core` library, the six drivers, and the ULT binaries). Execution: about 2 hours of single-node allocation for the METG sweep (6 systems × 4 PE counts × 14 granularities ×

7 repetitions), 30 minutes for the ULT benchmarks, and a few minutes for the peak run that sets the efficiency denominator. Analysis: under 5 minutes.

Artifact Setup (incl. Inputs)

Hardware: Same as A_2

Software: Everything under A_1, plus:

- Task Bench (branch `master`), a fork of StanfordLegion/task-bench adding the `reconverse/` and `converse/` implementations, which build from one `main.cc` since both layers expose the same API.
- The ULT benchmarks, `reconverse/tests/ult_bench` (`sched_rate.cpp`, `ctxswitch.cpp`), and their Converse ports.
- `run_peak.sh`, `run_taskbench.sh`, `run_ult.sh`, `common.sh`, `analyze.py`, and `analyze_ult.py` from the artifact repository.

Datasets / Inputs: None

Installation and Deployment: `make -C core` first (AVX2/FMA for Zen3), then each driver: MPI through the Cray `CC` wrapper, the two Charm++ drivers and the Converse driver with `charmc` (`-language converse++` for the latter), and the Reconverse driver with CMake against `RECONVERSE_ROOT`. Three build details are load-bearing: `charmc -optimize` is `-O2`, so the charmc-built drivers must be forced to `-O3` to match MPI and Reconverse; Charm++ must be built `--with-production`; and on Delta `charmc` invokes a bare `mpicxx` that resolves to a non-functional wrapper, so `mpicxx`/`mpicc` shims that `exec CC`/`exec cc` must be on `PATH` while compiling.

Artifact Execution

T_1: allocate one exclusive node. T_2: `run_peak.sh`, the efficiency denominator. T_3: `run_taskbench.sh`, the METG sweep. T_4: `run_ult.sh`. T_5: `analyze.py` and `analyze_ult.py`, producing `metg_results.txt` and `ult_results.txt`. $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_5$, with T_4 depending only on T_1.

Three controls are necessary; without them the comparison does not survive re-running. *Interleaving:* the loop nesting is repetition, PE count, granularity, system, so all systems' launches for a point are adjacent in time, and the leading system rotates each repetition; when run as contiguous per-system blocks, whichever went first got a cool node and 55 ms on the 1-PE coarsest point against 69–78 ms later, on identical work. *Repetitions:* all points use 7, since the coarse points set the efficiency scale and vary 20–33% run to run here. *Placement:* every system is confined to cores $0..P$, one CCX of one socket for $P \leq 8$; leaving placement to Slurm moved fine-granularity overhead by $4\times$ with the coarse point unchanged.

Artifact Analysis (incl. Outputs)

Both scripts write plain-text tables next to the raw stdout under `raw/`. The reported results are in `run_ab/`, whose `FINDINGS.md` documents the analysis; other run directories are earlier passes, marked superseded in the repository README.

Two analysis choices matter. Points are the *median* of surviving repetitions, after whole repetitions that ran in the node's slow clock state are discarded on the evidence of their pure-compute coarse point (about 17% of runs); each row carries a `spread` column of (worst−best)/best. The METG curve is normalized against a *shared* per-PE-count reference (the 25th percentile of all systems' coarse points), not each system's own coarsest run, which is the measurement most exposed to clock state. An absolute `vs peak` column and two normalization-free cross-checks are reported alongside.

D. Computational Artifact A_4

Relation To Contributions

A_4 isolates the adaptive queue-polling mechanism of Section IV-C and is the most direct evidence for C_2. Both benchmarks control the split of traffic between the process-level shared queue and the per-PE local queues, so the effect of re-apportioning the lottery scheduler's 64 slots is measured directly rather than inferred from an application. Benchmark 1 (parallel ring, Figure 10) and benchmark 2 (XY, Figure 11) are described in Section V-C; both are written against the Converse API and compile unchanged against either layer.

Expected Results

Reconverse should beat old Converse by 12–72% everywhere, with or without adaptation. Adaptation itself is smaller and less uniform: as explained in Section V-C, there are some cases where automatically adjusting polling is worse than static scheduling.

Expected Reproduction Time (in Minutes)

Setup: about 50 minutes. Execution: about 45 minutes. Analysis: under 5 minutes.

Artifact Setup (incl. Inputs)

Hardware: Same as A_2

Software:

- The benchmarks (`adapt_ring.cpp`, `adapt_xy.cpp`, and `bench_compat.h`, the shim that lets one source build against both layers).
- Reconverse from A_1 on branch `adaptive-queues`, which carries the queue registration and slot re-apportionment; `main` does not accept the flags below.
- A Charm++ build from A_1, for `charmc` and the old-Converse baseline; LCI, hwloc, libfabric and a PMI library, linked directly by the `Makefile`; and `run_adaptive.sh` and `analyze_adaptive.py` from the artifact repository.

Three run-time flags select the configuration: `+no_adaptive_polling` pins the apportionment (static scheduler), `+poll_adapt_mode count|hitrate` selects the weighting rule (`count` is the rule of Section IV-C and the default), and `+no_progress_polling` leaves network progress unregistered.

Datasets / Inputs: None

Installation and Deployment:

```
make RECONVERSE=<...>/reconverse \
     CHARM_DIR=<...>/mpi-linux-x86_64-smp
```

The `reconverse_` targets compile both sources against `libreconverse.a` and the LCI libraries in the Reconverse build tree; the `converse_` targets compile them with `charmc -language converse++ -DUSE_OLD_CONVERSE`. The library paths in the `Makefile` (`/opt/cray/libfabric/1.22.0`, `/usr/lib64/libpmi*`) are Delta's.

Artifact Execution

T_1: allocate one exclusive node. T_2: build the four binaries. T_3: `run_adaptive.sh <outdir>`, which runs benchmark 1 then benchmark 2, each over four configurations (`adaptive`, `hitrate`, `static`, `converse`) at 4 and 8 PEs for 3 repetitions. T_4: `analyze_adaptive.py`, producing `adaptive_results.txt`. $T_1 \to T_2 \to T_3 \to T_4$.

The control that matters here is pairing. A whole repetition can run 30–40% faster than the next for every configuration alike, far larger than the effect being measured, so the four configurations are launched back to back inside each repetition and the tables report the median of the within-repetition ratios rather than comparing absolute times across repetitions.

Artifact Analysis (incl. Outputs)

`analyze_adaptive.py` writes `adaptive_results.txt`, with raw stdout under `raw/`. The reported results are in `.../adaptive_20260730_final/`. Each row reports a ratio against a baseline with its min..max range, as explained in the "Reading these tables" section of the results file. `plot_adaptive.py` draws both figures from it; it and the other A_4 scripts live in A_2's `benchscripts/`, not in A_4.

E. Computational Artifact A_5

Relation To Contributions

A_5 produces the application results of Section V-D, supporting C_3: that the microbenchmark and scheduling gains translate into end-to-end speedups on real message-driven Charm++ codes. Neither application was modified; both are ordinary Charm++ programs recompiled against a Charm++ built on Reconverse, which is itself evidence for C_1.

Expected Results

Barnes-Hut on Reconverse should be up to 2× faster per iteration than on Converse from 1 to 16 nodes, with the largest gaps at 2 and 4 nodes; beyond that the 8-million-particle input is too small to keep scaling. ChaNGa should show a consistent roughly 2× speedup from 1 to 56 nodes, scaling near-linearly to 16 nodes and flattening beyond. The ChaNGa speedup should exceed both the Barnes-Hut and the ping-pong speedups.

Expected Reproduction Time (in Minutes)

Setup: about 90 minutes (the A_1 setup for both Charm++ builds, plus 20 for Barnes-Hut and 30 for ChaNGa, once per layer). Execution: about 1 hour of node time for the Barnes-Hut sweep and about 3 hours for ChaNGa, with queue wait dominating wall-clock time at the larger node counts. Analysis: under 10 minutes.

Artifact Setup (incl. Inputs)

Hardware: Same as A_2

Software: Both Charm++ builds from A_1, plus Barnes-Hut, ChaNGa, and the `utility` structures library that both applications require.

Datasets / Inputs: Barnes-Hut takes a binary particle file from the `plummer` generator in its repository, `./plummer 8000000 particles.bin`, and runs with 100 TreePieces per PE and a fixed iteration count set by `-killat`. ChaNGa takes the 80-million-particle tipsy input `lambb.00500` with `lambb.param` (`nSteps` 10, `dTheta` 0.6, periodic and comoving), archived separately at https://doi.org/10.5281/zenodo.22698278.

Installation and Deployment: Barnes-Hut builds by pointing its `Makefile` at a Charm++ build (`CHARM_PATH`) and the structures library (`STRUCTURES_PATH`), then `make`; ChaNGa by `./configure` against the same `utility` tree and Charm++ build, then `make`. Both are built once per layer. The two layers run at different PPN and PE counts per node, given in Section V-D.

Artifact Execution

T_1: generate the Plummer input and stage the ChaNGa dataset. T_2: build both applications against both Charm++ builds, giving four binaries. T_3: run Barnes-Hut at 1, 2, 4, 8, and 16 nodes on both layers. T_4: run ChaNGa from 1 to 56 nodes on both layers. T_5: collect times and plot. $T_1 \to T_2 \to \{T_3, T_4\} \to T_5$. Each (application, layer, node count) point is the median of 5 runs, with the two layers interleaved within each repetition.

Artifact Analysis (incl. Outputs)

Barnes-Hut reports average time per iteration, plotted against node count on a $\log_2$ x-axis with a linear y-axis; ChaNGa reports total execution time, plotted log-log. The tabulated times are in `reconverse_times.xlsx`, replotted by `plot_barnes_hut.py` and `plot_changa.py`; all three are in the deposit. Reproduction should be judged on the ratio between layers and the shape of the scaling curves, not on absolute times.