Towards PDES in a Message-Driven Paradigm: A Preliminary Case Study Using Charm++

Eric Mikida, Nikhil Jain,
Laxmikant Kale
University of Illinois at
Urbana-Champaign
{mikida2,nikhil,kale}@illinois.edu

Elsa Gonsiorowski,
Christopher D. Carothers
Rensselaer Polytechnic
Institute
{gonsie,chrisc}@rpi.edu

Peter D. Barnes, Jr.,
David Jefferson
Lawrence Livermore
National Laboratory
{barnes26,jefferson6}@llnl.gov

ABSTRACT

Discrete event simulations (DES) are central to exploration of "what-if" scenarios in many domains including networks, storage devices, and chip design. Accurate simulation of dynamically varying behavior of large components in these domains requires the DES engines to be scalable and adaptive in order to complete simulations in a reasonable time. This paper takes a step towards development of such a simulation engine by redesigning ROSS, a parallel DES engine in MPI, in CHARM++, a parallel programming framework based on the concept of message-driven migratable objects managed by an adaptive runtime system. In this paper, we first show that the programming model of CHARM++ is highly suitable for implementing a PDES engine such as ROSS. Next, the design and implementation of the CHARM++ version of ROSS is described and its benefits are discussed. Finally, we demonstrate the performance benefits of the CHARM++ version of ROSS over its MPI counterpart on IBM's Blue Gene/Q supercomputers. We obtain up to 40% higher event rate for the PHOLD benchmark on two million processes, and improve the strong-scaling of the dragonfly network model to $524,288$ processes with up to 5× speed up at lower process counts.

1. INTRODUCTION

Discrete event simulations (DES) are a key component of predictive analysis tools. For example, network designers often deploy DES to study strengths and weaknesses of different network topologies. Similarly, the chip design process makes heavy use of DES to find the optimal layout of the circuit on a new chip. As the complexity of such analyses increases over time, either due to the larger number of components being studied or due to higher accuracy requirement, the capability of DES engines also needs to increase proportionately. Parallel discrete event simulation (PDES) holds promise for fulfilling these expectations by taking advantage of parallel computing. Many scientific domains, e.g. cosmology [22], biophysics [27], computational chemistry [14], etc., have already exploited parallel computing to great effect. Thus, it is natural that PDES should be explored to match the growing requirements posed by domains of interest.

Development of scalable PDES engines and models is a difficult task with many important differences from common High Performance Computing (HPC) applications, which are predominantly from the science and engineering domain. First, while the interaction pattern of common HPC applications can be determined apriori, the communication pattern in most PDES models is difficult to predict. The communication in PDES models is also more likely to be one-sided, i.e. the source entity may create work or data for a destination without the destination expecting it. Second, typical PDES models are asynchronous and do not have a predetermined time step; simulated entities are allowed to proceed through the simulation at a pace dictated by the availability of work. In contrast, many common HPC applications perform iterative operations with similar work repeated across iterations. Third, unlike common HPC applications, the amount of computation per communication byte is typically low for PDES models.

Due to the unique features of PDES described above, MPI [2], the de-facto standard for parallel programming, may not be the best fit for developing PDES engines and models. This is because MPI is suitable for bulk-synchronous models of parallel programming based on two-sided and collective communication. Support for one-sided indeterministic communication, as required by PDES, is limited in MPI [15]. Moreover, long running complex models may require load balancing and checkpointing infrastructure, both of which are programmer's responsibility in MPI. Despite these limitations, Rensselaer's Optimistic Simulation System (ROSS) [6], a PDES engine implemented using MPI, has been shown to be highly scalable for homogeneous models with low communication volume [5].

For models with heavy communication and high complexity, the capabilities and performance of the MPI version of ROSS exhibit limitations. To overcome these limitations, a new version of ROSS is being designed and implemented on top of CHARM++ [4], a parallel programming paradigm which is a better fit for performing PDES. Powered by an intelligent adaptive runtime system, CHARM++ is an alternative method for developing parallel programs based on object-oriented programming in contrast to MPI's processor based programming. Several large scale scientific applications, including NAMD [27], ChaNGa [22], EpiSimdemics [31] and OpenAtom [16], have been developed in

SIGSIM-PADS '16, May 15-18, 2016, Banff, AB, Canada
© 2016 ACM. ISBN 978-1-4503-3742-7/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2901378.2901393

Charm++. Additionally, the Parallel Object-oriented Simulation Environment (POSE) [30] is a PDES engine implemented as a DSL on top of Charm++ which explores various scheduling and load balancing techniques but is ultimately limited by poor sequential performance. In this paper, we describe our experience and results related to the Charm++ version of ROSS. The main contributions of this work are:

- **Suitability of Charm++** for implementing a PDES engine in general, and ROSS in particular, is demonstrated.
- **A Charm++ version of ROSS** is presented along with a detailed description of its design and implementation.
- **Features and benefits** of the Charm++ version of ROSS are presented.
- **Superior performance** of the Charm++ version of ROSS over its MPI counterpart is shown on two models running on IBM's Blue Gene/Q supercomputers:
    1. 40% higher event rate is obtained for the PHOLD benchmark on two million processes.
    2. Strong-scaling of the dragonfly network model for simulating different communication patterns is improved to 524, 288 processes with up to 5× speed up at lower process count.

2. BACKGROUND

Charm++ [4] and ROSS [5] are central to the work presented in this paper. This section first provides an overview of PDES, followed by details of its concrete implementation in ROSS. Finally we introduce Charm++ and discuss why it is a good fit for PDES.

2.1 PDES

A DES system consists of *Logical Processes* (LPs) and events. LPs represent the entities within the simulation and encapsulate the majority of simulation state. Changes within DES are driven by execution of events that are communicated from one LP to another at specific points in virtual time. In order to obtain a correct simulation, the events must be executed in timestamp order, i.e. causality violations should not happen wherein an event with higher timestamp is executed before an event with a lower timestamp.

For a sequential DES, the simulation progresses in a very straight forward manner: the pending event with the lowest timestamp is processed by its receiving LP. As each successive event is processed, simulation time moves forward. After an event is successfully processed, its memory can be reclaimed by the simulation system.

In PDES, special care must be taken to ensure the correct ordering of events. The LP objects are distributed across concurrent processes with events traversing across the network. Therefore, parallel simulators must handle inter-node communication as well as maintain global synchronization to avoid causality violations. There are two main schools of thought on how to avoiding causality violations in PDES: conservative and optimistic [13, 17, 26]. In a conservative synchronization algorithm, events can only be executed if there is a guarantee that it will not result in a causality violation. In optimistic algorithms, events are executed speculatively and if a causality violation does eventually occur, the simulation engine must recover from it.

In this paper our focus is on the optimistic Time Warp algorithm proposed by Jefferson et al. [17]. In Time Warp, when a causality violation is detected, states of the relevant LPs are reverted to a point in time before the causality violation occurred. This process is referred to as rollback. Once an LP has rolled back to a previous point in virtual time, it is allowed to resume forward execution. In order to correctly rollback an LP, the simulation engine needs to store history information about that LP, which causes an increase in the memory footprint of the program. To mitigate this, the simulation engine periodically synchronizes and computes the Global Virtual Time (GVT). The GVT is a point in virtual time which every LP has safely progressed to, and therefore memory used to store the history before the GVT can safely be reclaimed by the simulator.

2.2 ROSS

ROSS is a framework for performing parallel discrete event simulations. It has demonstrated highly scalable, massively parallel event processing capability for both conservative and optimistic synchronization approaches [5, 6, 8, 20, 25]. For optimistic execution, ROSS mitigates Time Warp state-saving overheads via *reverse computation* [9]. In this approach, rollback is realized by performing the inverse of the individual operations that were executed in the event's forward execution. This reduces the need to explicitly store prior LP state, leading to efficient memory utilization.

Most recently, ROSS optimistic event processing has demonstrated super-linear performance for the PHOLD benchmark using nearly 2 million Blue Gene/Q cores on the 120 rack Sequoia supercomputer system located at LLNL [5]. Barnes et al. obtained 97x speedup at 120 racks from a base configuration of 2 racks for a PHOLD model configured with over 250 million LPs. The peak event rate was in excess of 500 billion events-per-second. PHOLD configurations similar to the one used in Barnes et al. [5] have been used in the results section of this paper.

Using ROSS's massively parallel simulation capability, many HPC system models have been developed as part of a DOE co-design project for future exascale systems. These include models for the torus and dragonfly networks and the DOE CODES storage simulation framework. The torus model has been shown to simulate 1 billion torus nodes at 12.3 billion events per second on an IBM Blue Gene/P system [19]. The dragonfly model has been scaled to simulate 50 million nodes, with a peak event rate of 1.33 billion events/second using 64K processes on a Blue Gene/Q system [25]. We use a similar dragonfly model in the results section of this paper as was used by Mubarak et al. in [25].

2.3 CHARM++

Charm++ is a parallel programming framework which consists of an adaptive runtime system that manages migratable objects communicating asynchronously with each other. Like MPI, it is available on all major HPC platforms and takes advantage of native messaging routines when possible for better performance. Applications developed for Charm++ are written primarily in C++ with a small amount of boiler-plate code to inform the runtime system of the main application entities (C++ objects).

Unlike MPI, Charm++ is more than just a messaging layer for applications. It contains a powerful runtime system that aims to remove some of the burden of parallel program-

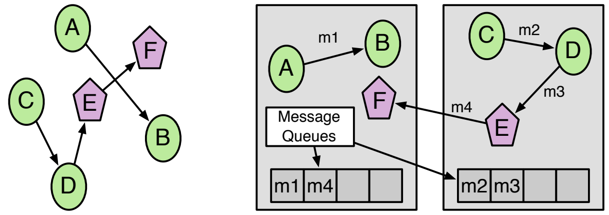


Figure 1: A depiction of communicating chares from the programmers view (left) and the runtime system's view after the chares have been mapped to the processes (right). Arrows represent messages sent between chares.

ming from the application developer by taking care of tasks such as location management, adaptive overlap of computation and communication, and object migration. Broadly speaking, the primary attributes of Charm++ that we hope to leverage in PDES are: object-level decomposition, asynchrony, message-driven execution, and migratability.

In Charm++, the application domain is decomposed into work and data units, implemented as C++ objects called chares, that perform the computation required by the application. Thus, the programmers are free to write their applications in work units natural to the problem being solved, instead of being forced to design the application in terms of cores or processes. An application may also have many different kinds of chares for different types of tasks. Note that the user is free to choose the number of chares and is not bound by the number of cores or processes. Figure 1 (left) shows a collection of 6 chares of two different types labeled A, B, C, D, E, and F. Arrows represent communication between chares. This freedom of decomposition is highly suitable for implementing PDES engines and models since they typically have simulated entities of different types whose work and count is determined by the model being simulated.

The runtime system is responsible for mapping the chares defined by an application to cores and processes on which they are executed. The chares interact with each other asynchronously in a one-sided manner: the senders send messages to the destinations and continue with their work without waiting for a synchronous response. This mode of communication is a direct match to the type of communication exhibited by a typical PDES. Figure 1 (right) shows a potential mapping of the 6 chares to specific hardware resources. The runtime system keeps track of the location of each chare and handles sending messages across the network when communicating chares are mapped to different hardware resources.

In Charm++, chares are message-driven entities, i.e., a chare is scheduled and executed only when it has a message to process from itself or from other chares. Being message-driven makes chares an ideal candidate for representing the entities modeled in a PDES. This is because LPs in a PDES are also typically driven by events that are sent to them. In Figure 1 (right), we see that each process maintains a queue of all messages directed to it's local chares. The runtime system removes messages from these queues, and executes appropriate method on each message's destination chare.

Finally, the chares in Charm++ are migratable, i.e., the runtime system can move the chares from one process to another. Moreover, since chares are the only entities visible to the application and the notion of processes is hidden from the application, the migration of chares can be done automatically by the runtime system. This allows the runtime system to enable features such as dynamic load balancing, checkpoint/restart, and fault tolerance with very little effort by the application developer. Complex PDES models can take advantage of these features, thus providing a solution to problems posed by long running dynamic models.

3. DESIGN OF THE CHARM++ VERSION OF ROSS

In order to demonstrate the benefits an asynchronous message-driven paradigm can have for PDES, we have created a version of ROSS built on top of Charm++. Our focus is on improving the parallel performance and taking advantage of features provided by Charm++ (Section 2.3), while keeping the PDES logic and ROSS API as intact as possible. The design and implementation of the Charm++ version of ROSS can be divided into five segments, each of which are described in this section: parallel decomposition, scheduler, communication infrastructure, global virtual time (GVT) calculation, and user API.

3.1 Parallel Decomposition

As described in Section 2.1, a typical PDES consists of LPs that represent and simulate entities defined by the model. Whenever an event is available for an LP, the LP is executed. This execution may result in generation of more events for the given LP or other LPs. In the MPI version of ROSS, three main data structures are used to implement this process as shown in Figure 2 (left): Processing Element (PE), Kernel Processes (KP), and Logical Process (LP).

There is a single PE per MPI rank with three main responsibilities. First, the PE manages event memory of its MPI rank with a queue of preallocated events. Second, the PE maintains a heap of pending events sent to LPs on its MPI rank. Third, the PE contains data used when coordinating the GVT computation across all MPI ranks.

KPs are used primarily to aggregate history storage of multiple LPs into a single object to optimize fossil collection. This history consists of events that have been executed by an LP. Fossil collection refers to freeing up the memory occupied by the events that occurred before the current GVT.

LPs are the key component in actually defining the behavior of a specific model. They contain model-specific state and event handlers as defined by the model writer, in addition to storing some meta-data used by ROSS such as their globally unique ID. The pending events received for LPs are stored with the PEs while the events that have been executed by the LPs are stored with the KPs.

The organization of ROSS in MPI, as described above, is mainly driven by the process-based model typical of MPI programs. On each MPI process, there is a single flow of control managing the interaction between a single PE and a collection of KPs and LPs as shown in Figure 2 (left). None of these application entities are recognized by MPI, and hence are managed as passive sequential objects that rely on the MPI process for scheduling and communication.

Decomposition with Charm++: As shown in Figure 2 (right), the Charm++ version of ROSS consists of three key entities: LPs, LP Chares, and PE Managers. The role

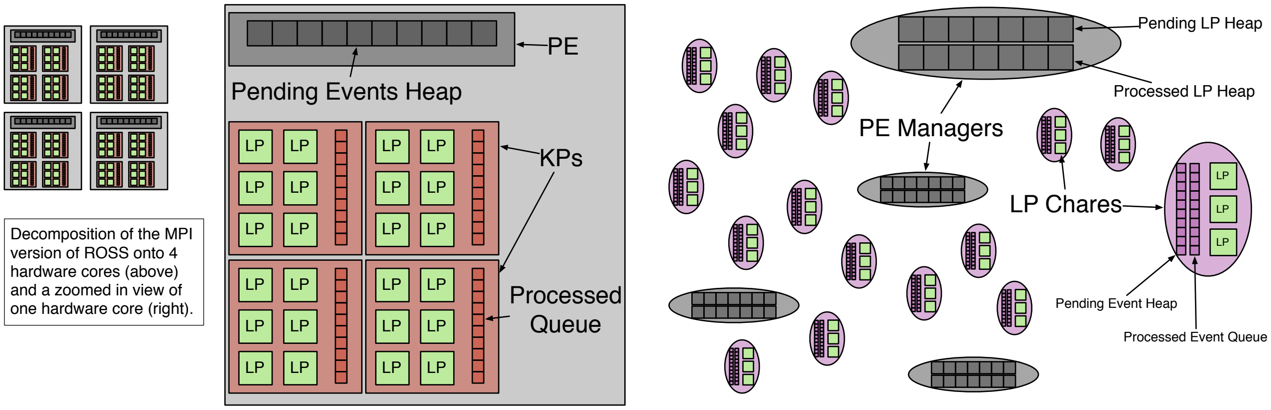


Figure 2: User view of ROSS decomposition in MPI (left) and Charm++ (right). Boxes are regular C structs, and ovals are chares (known to the Charm++ runtime system). In the MPI version, the users view requires explicit knowledge of the mapping of entities to hardware, where in the Charm++ version the runtime system manages this mapping.

and implementation of LPs is same as the MPI version of ROSS in order to preserve the user-level API of ROSS.

LP Chares now encapsulate everything an LP needs to execute. They contain the LPs themselves, which as before hold model-specific state and event handlers, in addition to storing both the pending and past events of its LPs. The charm runtime system is aware of LP Chares and allows communication of events directly between LP Chares. Upon receiving an event, an LP Chare handles any causality violations or event cancellations by performing the necessary rollbacks, and enqueues the new event into its pending queue. This removes the need for event storage at the PE and KP level. The mapping of LPs to LP Chares is left to the user so that a model-appropriate scheme can be used, but the mapping of LP Chares to hardware resources is handled by the runtime system. This also allows the runtime system to migrate LPs as needed which enables features such as checkpointing and dynamic load balancing.

PE Managers are implemented as special chares in Charm++ which guarantee a single PE Manager chare per process. Similar to PEs in the MPI version, PE Managers still manage event memory and coordinate the computation of GVT; however, they no longer store pending events. Instead, PE Managers maintain a pending LP heap and a processed LP heap, both of which store pointers to LP Chares. The pending heap is sorted based on the timestamp of the next pending event of each LP Chare, and the processed heap is sorted based on the timestamp of the oldest processed event of each LP Chare. The ROSS scheduler is now integrated into the PE Managers and utilizes these LP Chare queues to coordinate event execution and fossil collection among chares co-located with a given PE Manager. By integrating the ROSS scheduler into the PE Manager, the Charm++ runtime system is now aware of the ROSS scheduler. The changes to the ROSS scheduler are described in more detail in the next section.

Expected cost of basic operations: In the MPI version, the expected cost to add, remove, or execute an event is $\log N$ when N is the number of events in the PE's pending heap. In the Charm++ version, all event heaps are stored in the LP Chares and only store events for its own LPs, so the cost of any operation on these heaps is approximately $\log \frac{N}{C}$ where C is the number of chares on the processor/-core. In some cases, the addition or removal of an event requires updating the PE Manager-level heaps which incurs an additional $\log C$ operation. Thus, the expected cost to add, remove, or execute an event in the Charm++ version is at most $\log \frac{N}{C} + \log C = \log N$.

3.2 Scheduling

In the MPI version of ROSS, the scheduler is simply a function called from `main` after the initialization is completed. There are three types of schedulers supported in the MPI version: sequential, conservative, and optimistic. Figure 3a provides a code snippet that shows the simplified implementation of the optimistic scheduler, which is almost always the preferred option for ROSS. The scheduler is primarily an infinite loop that polls the network for driving the communication, performs rollbacks if they are needed, executes events on LPs in batches of size *batch_size*, and performs GVT computation if needed. In the conservative mode, the rollback step is skipped, while in the sequential mode only event execution happens.

As discussed in the previous section, the schedulers in the Charm++ version of ROSS have been promoted from regular C functions to methods on PE Manager chares. The runtime system schedules all chares, include PE Managers, based on availability of messages for them to process, so the schedulers are implemented as messages to PE Managers. Again, we will focus our detailed discussion on the optimistic scheduler. As shown in Figure 3b, the Charm++ version of the scheduler has two main tasks: event execution and GVT computation. Unlike the MPI version, the scheduler no longer does network polling (which is now handled by the runtime system), and no longer deals with event queuing or rollbacks (which are now handled by the LP Chares).

When a simulation begins, all PE Managers receive a message that triggers execution of execute_events method. During normal execution, this involves delegating control to LP Chares for event execution by using the pending LP heap, and then sending a message, say M, that should trigger execution of execute_events on the PE Manager. The message M is added to the runtime system's queue, and when the execution of execute_events is complete, control returns to

```
//main control loop in PE
void scheduler() {
 while(1) {
  poll_network(); //drive communication
  //rollback if cancellation event
  //rollback if event(s) with old timestamp
  for (i=0; i<batch_size; i++) {
    //execute events directly on LPs
  }
  if (ready_for_gvt) {
    do_gvt();
    //perform fossil collection on KPs
  }
 }
}
```

(a) MPI Version : PE drives the execution.

```
void PEManager::execute_events() {
 for (i=0; i<batch_size; i++) {
  /* delegate control to LP Chares
     for event execution */
 }
 if (ready_for_gvt)
  start_gvt();
 else
  // self-send an execute_events() message
}
void PEManager::gvt_done() {
 /* delegate control to LP Chares
    for fossil collection */
 // self-send an execute_events() message
}
```

(b) CHARM++ Version: PE Manager is one of the chares scheduled when work is available for it.

Figure 3: Simplified versions of the role of PE in optimistic mode.

the runtime system. At this point, the runtime system may perform other tasks such as polling the network or execute methods on other chares based on its queue. Eventually when the message M reaches the top of the runtime system's queue, execute_events is executed again on the PE.

During the execution of execute_events, if it is time to compute the GVT, start_gvt() is called instead of sending the message that triggers execution of execute_events. This starts the asynchronous computation of the GVT, which is described in more detail in Section 3.4. When the GVT computation is complete, a message to trigger execution of gvt_done is sent to PE Managers, which results in delegation of control to the LP Chares to do fossil collection. Finally, a message to trigger execution of execute_events is sent to resume forward progress.

3.3 Communication Infrastructure

ROSS requires communication across processes to deliver events generated by LPs on one process for LPs on different processes. In the MPI version of ROSS, the delivery of events is performed using point-to-point communication routines in MPI. Since MPI's point-to-point communication is two-sided, this requires both the sender and the receiver to make MPI calls. The complexity and inefficiency in this process stems from the fact that the events are generated based on the model instance being simulated, and thus the communication pattern is not know apriori. Whenever an LP, say *srcLP*, generates an event for another LP, say *destLP*, the process which hosts *destLP* is computed using simple static algebraic calculations and the sender process issue a non-blocking MPI_Isend to it. Since the communication pattern is not known ahead of time, all MPI processes post non-blocking receives that can match to any incoming message (using `MPI_ANY_TAG` and `MPI_ANY_SOURCE`). Periodically, the scheduler performs network polling to test if new messages have arrive for LPs on the current process. Both these actions, posting of receives that match to any receive and periodic polling, lead to high overheads.

In the CHARM++ version of ROSS, both LPs and the events they generate are encapsulated in LP Chares which the runtime system knows about. Hence, when an LP Chare issues an event for the another LP Chare, the runtime takes the ownership of the event and delivers it to the destination LP Chare without the involvement of PE Manager. This also means that each LP Chare can detect its own causality violations as soon as it receives events, and take the appropriate course of action immediately. The runtime also performs the lookup of the process that actually owns each LP Chare automatically and does not require any additional support from ROSS. This automation is extremely useful for cases where LPs are load balanced across processes, and their owner processes cannot be computed using simple algebraic calculation. Moreover, since CHARM++'s programming model is one-sided and message driven, there is no need to post receives in advance.

3.4 GVT Computation

Computation of the GVT requires finding the minimum active timestamp in the simulation. Hence, while computing the GVT, all of the events and anti-events (events that cancel an older event) which have not been executed yet have to be taken into consideration. This implies that any event that has been sent by a process must be received at its destination before the GVT is computed. Without such a guarantee, we are at the risk of not accounting for events that are in transit. Alternatively, a sender process can account for the events it sends out in the GVT computation. However, this scheme leads to additional overhead of explicit acknowledgment for every event received by the destination processes to the sender processes, and hence is inferior in terms of performance to the former scheme.

In the MPI version of ROSS, to guarantee that all events are accounted for the ROSS engine maintains two counters: events sent, and event received. When the GVT is to be computed, all execution of events stops and ROSS makes a series of MPI_Allreduce calls interspersed with polling of the network to guarantee communication progress. Once the global sum of the two counters match, all events are accounted for, and the global minimum timestamp of these events becomes the new GVT. Then ROSS resets the counters, performs fossil collection, and resumes event execution.

In the CHARM++ version of ROSS, we have simplified the computation of the GVT by replacing the above mentioned mechanism with a call to CHARM++'s efficient asynchronous quiescence detection (QD) library [1,28]. The QD library automatically keeps track of all communication in a Charm++ application, and invokes an application specified *callback* function when the global sum of initiated sends matches the global sum of the receives. The efficiency of the QD library is due to overlap of the library's execution with application progress obtained by performing asynchronous non-blocking communication in the background. Once it is time to compute the GVT, event execution is stopped while ROSS waits for QD to complete. During QD the runtime continues polling the network and delivering messages to chares as normal. Once quiescence is reached, the GVT is computed via a global reduction, and the PE Manager does fossil collection and restarts event execution. Because of the flexibility in QD and the asynchrony of the runtime system there are further opportunities to reduce the synchronization cost of GVT computation discussed in Section 4.4.1.

3.5 User API and Porting Models

One of the major goals when writing the Charm++ version of ROSS was preserving the existing user API. By making minimal changes to the user API, we hope to make the porting process for models written for the MPI version of ROSS simple and minimalistic. To port a model from the MPI version to the Charm++ version of ROSS, the majority of changes are performed in the startup code where the model is initialized. Once the simulation of the model begins, the logical behavior of the Charm++ version is exactly same as the MPI version, and hence there is often no change needed in the LP event handlers or reverse event handlers.

Among the startup changes, the primary change is related to LP mapping. In the MPI version, there are two mapping functions. One of them is used during startup to create LPs on the calling PE, and decide their local storage offset, KP it belongs to, and type of LP it is. The second mapping function is used during execution to determine the PE on which an LP with a given ID exists.

In the Charm++ version, since LPs are no longer bound to PEs, four mapping functions have to be defined to provide equivalent functionality w.r.t. LP Chares. The first two are used during startup. One is a function that takes an LP Chare index and an offset, and returns the global LP ID that resides on that chare at that offset. The second is a function that takes a global LP ID and returns the type of that LP. The remaining two functions are used for LP lookup during execution. One takes an LP ID and returns the LP Chare on which that LP resides and another takes an LP ID and returns the local storage offset of that LP. These two functions are the inverse of the first function that initially determined placement of LPs.

In addition to the above mapping changes, the models usually require minor changes to their main functions. These changes usually entail setting which maps to use, and setting some other Charm++ specific variables such as the number of LP Chares a simulation is going to use. Concrete examples of the changes required to port the models are further presented in the results section, where the PHOLD and Dragonfly models are discussed in detail (Section 5).

4. BENEFITS OF THE CHARM++ VERSION OF ROSS

In this section, we describe the benefits of the Charm++ version of ROSS. These include inherent benefits that the use of Charm++'s programming model provides in the current version, as well as new features enabled by the runtime system that are being implemented and will be explored in detail in a future publication.

4.1 Better Expression of Simulation Components

One of the most important advantages of using Charm++ for implementing ROSS is that the programming model of Charm++ is a natural fit for implementing PDES. In the Charm++'s programming model, as described in Section 2.3, an application implementation is composed of many chares executing concurrently, driven by asynchronous message communication. When a message is received for a chare, the runtime executes the appropriate method on the chare to handle the message. This model is analogous to PDES where several LPs are created to represent simulation entities and event handlers are executed on the LPs when events are available for them. Thus LPs and events map naturally to chares and messages. Moreover, similar to PDES models, which can have different types of LPs and events to simulate different types of entities and work, there can be many types of messages and chares in Charm++ to trigger and perform different types of computation. As a result, use of Charm++ provides a natural design and easy implementation of ROSS, which is in part evidenced by the smaller code base discussed in the next section.

4.2 Code Size and Complexity

Since the Charm++ runtime system relieves ROSS from managing inter-process communication, network polling, and scheduling computation and communication, the size and complexity of the ROSS code base has been reduced significantly. The communication module of the ROSS code base, which contains a large amount of MPI code for conducting data exchange efficiently, has been removed. This has resulted in a much smaller code base. In fact, the SLOC count (significant lines of code count) has been reduced to nearly half its original value, from 7,277 in the MPI version to 3,991 in the Charm++ version. This is despite the fact that the Charm++ version of ROSS also includes additional code required for enabling migration of LPs, a feature that is not present in the MPI version. Moreover, since the Charm++ runtime system manages scheduling of different types of work units (LPs, PEs, communication, etc.), the code base has been simplified since explicit scheduling of these work units need not be done by ROSS.

4.3 Ease and Freedom of Mapping

Charm++ programmers are encouraged to design their applications in terms of chares and the interactions between them, instead of programming in terms of processes and cores. This relieves the programmers from the burden of mapping their computation tasks to processes and cores, and thus allows them to divide work into units natural to the computation. In ROSS, and in PDES in general, this means that a model writer can focus solely on the behavior of the LPs and map them to chares based on their interactions. As a result, the model writer does not have to worry about clustering the LPs to match the number of processes and cores they may be running on now or in the future. Instead, they can choose the number of chares that is most suitable to their model, and let the runtime assign these chares to processes based on their computation load and communication pattern. A concrete example of this is shown in Section 5.2.4 for the dragonfly network model.

4.4 Features Enabled by CHARM++

In addition to the benefits described above, use of CHARM++ enables many new features that are difficult or infeasible to add to the MPI version of ROSS. Many of these features are currently under development, so their performance implications will be described in a future work.

4.4.1 Asynchronous GVT

As mentioned in Section 3.4, the computation of GVT using ROSS's current scheme has been simplified by the CHARM++ QD library. Moving forward, the flexibility provided by the adaptive and asynchronous nature of the runtime system presents opportunities to further restructure the GVT computation. Since computation and communication in CHARM++ are asynchronous and non-blocking, GVT computation can be adaptively overlapped with event execution. An obvious optimization is to overlap the global reduction performed to compute the GVT post quiescence detection with event execution. A more complex scheme, similar to the one described by Mattern [21], will enable continuous computation of the GVT in the background, without the need for globally synchronizing all processes and intermittently blocking event execution for detecting quiescence.

4.4.2 Migratability

One of the most important benefits of the object oriented programming model of CHARM++ is the ability to migrate chares. The use of LP Chares to host data belonging to LPs enables migration of LPs during execution of a simulation. In terms of implementation, this has been achieved in the CHARM++ version of ROSS by defining a simple serializing-deserializing function for LP Chares using CHARM++'s PUP framework [4]. Migratability of LPs leads to two new features in ROSS: automatic checkpoint-restart and load balancing. When directed by ROSS to checkpoint, the CHARM++ runtime system migrates LPs to disk and the simulation can be restarted as part of a new job.

Load Balancing: In complex, long-running models, load imbalance and excess communication can hinder performance and scalability. Dynamic load balancing algorithms can help address this problem [10, 12] but may add complexity to both the PDES engine, or to specific models. The CHARM++ runtime system eases this burden by making migratability a central tenet of its design. Migratability of LPs enables the CHARM++ runtime system to periodically redistribute LPs in order to balance load among processes and reduce communication overhead. In the current implementation, the CHARM++ version of ROSS is able to utilize basic load balancing strategies provided by the runtime system. Currently, we are working on developing PDES-specific load balancing strategies so that better performance can be obtained for complex models by utilizing extra information provided by the PDES engine and model.

4.4.3 TRAM

One common characteristic of many PDES simulations is communication of a high volume of fine-grained messages in the form of events. These numerous fine-grained messages can easily saturate the networks, and thus increase the simulation time. To optimize for such scenarios, CHARM++ provides the Topological Routing and Aggregation Module that automatically aggregates smaller messages into larger ones [29]. In the past, Acun et al. [4] have already demonstrated the benefits of using TRAM for PDES simulations using a simple PDES mini-application. Based on those results, we plan to utilize TRAM to further improve the performance of ROSS.

5. PERFORMANCE EVALUATION

Scalable performance is a major strength of ROSS that has lead to its widespread use in conducting large scale simulations [23,24]. One of the primary reasons for the unprecedented event rate obtained by ROSS is its MPI-based communication engine that has been fine-tuned over a decade by the developers of ROSS. Hence, despite the benefits of the CHARM++ version of ROSS described in the previous section, it is critical that it also provides performance comparable to the MPI version of ROSS. In this section, we study the performance of the two versions of ROSS using its most commonly evaluated models - PHOLD and Dragonfly [7, 23]. For these comparisons, we have used the latest version of ROSS as of December, 2015 [3].

5.1 PHOLD

PHOLD is one of the most commonly used models to evaluate the scalability of a PDES engine under varying communication load. It consists of a large number of LPs all of which perform similar work. At the beginning of a PHOLD simulation, a fixed number of events are scheduled on every LP. When an LP executes an event at time T_s, it creates a new event to be executed at time $T_s + T_o$. The offset, T_o, equals the sum of a fixed lookahead, T_l and a random delay chosen using an exponential distribution. The new event is sent either to a randomly selected LP with probability p, or to the current LP with probability $1 - p$. T_l, p, and the mean of the exponential distribution are all model input parameters. The only work done by an LP when processing an event is the generation of a few numbers from a random distribution, which results in PHOLD being extremely fine-grained and communication intensive if p is large.

5.1.1 Porting Process

Minimal changes are required to execute the version of PHOLD distributed with ROSS on top of the CHARM++ version of ROSS. First, a few global variables used in PHOLD, e.g. the number of LPs, should be removed since they are provided by the CHARM++ version of ROSS. As such, storing the model specific versions of these variables is redundant. Second, a new simple mapping function that returns the type of LP based on it global ID is required. Since PHOLD only has one type of LP, this mapper is a trivial two-line function that always returns the same LP type.

5.1.2 Experimental Set up

All the experiments have been performed on Vesta and Mira, IBM Blue Gene/Q systems at Argonne National Laboratory. The node count is varied from 512 to 32, 768, where 64 processes are launched on each node to make use of all the hardware threads. On BG/Q, due to disjoint partitioning of jobs and minimal system noise, event rate does not change significantly across multiple trials.

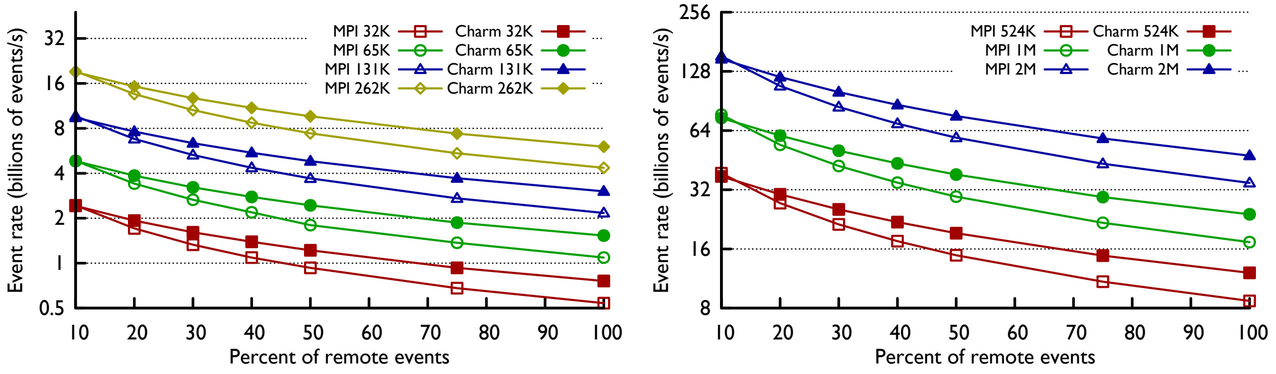


Figure 4: Simulating PHOLD on up to two million processes: for all process counts, Charm++ version of ROSS provides up to 40% higher event rate in comparison to the MPI version of ROSS.

In these weak-scaling experiments, the number of LPs is fixed at 40 LPs per process (or MPI rank), each of which receives 16 events at startup. The lookahead (T_l) and the simulation end time are set to 0.1 and $8,192$, respectively. The percentage of remote events, $(100 * p)$, is varied from 10% to 100% to compare the two versions of ROSS under different communication loads.

5.1.3 *Performance Results*

Figure 4 compares the performance of PHOLD executed using the MPI and Charm++ versions of ROSS on a wide range of process counts. For both the versions, we observe that the event rate drops significantly as the percentage of remote events increase. This is expected because at a low remote percentage, most of the events are self-events, i.e. they are targeted at the LP which generates them. Hence, the number of messages communicated across the network is low. As the percentage of remote events increases, a higher volume of events are communicated to LPs located on other processes. Thus, the message send requires network communication and the amount of time spent in communication increases, which limits the observed event rate.

At low remote percentages $(10 - 20\%)$, both versions of PHOLD achieve a similar event rate irrespective of the process count being used. However, as the percentage of remote events increase, the Charm++ version of ROSS consistently achieves a higher event rate in comparison to the MPI version of ROSS. At 100% remote events, the Charm++ version outperforms the MPI version by 40%. This shows that when communication is dominant, the runtime controlled message-driven programming paradigm of Charm++ is able to better utilize the given resources. At 50% remote events, which is a more realistic scenario based on the dragonfly results from the next section, the Charm++ version improves the event rate by approximately 28%. Figure 4 shows that these improvements in the event rate are observed on all process counts, ranging from $32K$ processes (512 nodes) to two million processes ($32K$ nodes).

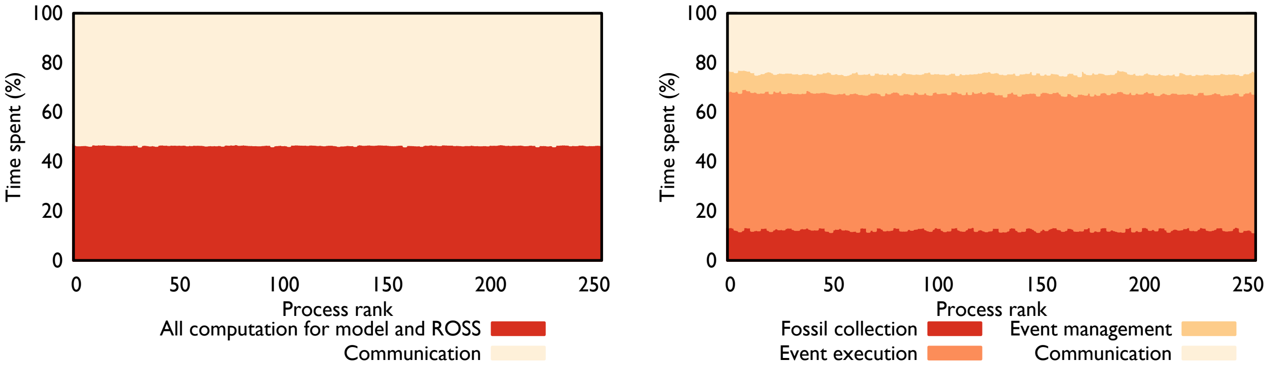
5.1.4 *Performance Analysis*

To identify the reasons for the performance differences presented in the previous section, we used several performance analysis tools to trace the utilization of processes. For the Charm++ version of ROSS, we use Projections, a tracing tool built for Charm++ applications [18]. MPI-Trace library [11] is used to monitor the execution of the MPI version of ROSS. As a representative of other scenarios, we present the analysis for execution of PHOLD on $32,768$ processes with 50% remote events.

Figure 5a shows that for the MPI version of ROSS, as much as $45 - 50\%$ of the time is spent on communication, while the rest is spent on computation. In this profile, communication is time spent in MPI calls, which include *MPI_Isend*, *MPI_Allreduce*, and *MPI_Iprobe*. A more detailed look at the tracing data collected by MPI-Trace shows that half of the communication time is spent in the *MPI_Allreduce* calls used when computing the GVT, while the remaining half is spent in sending, polling for, and receiving the point-to-point messages. A major fraction of the latter half is spent in polling the network for unexpected messages from unknown sources.

In contrast to the MPI version of ROSS, Figure 5b shows that the Charm++ version of ROSS spends approximately 75% of its time performing computation, while only the remaining 25% is spent in communication. In this profile, communication encompasses sending and receiving of events, global synchronization for the GVT, and any other overheads incurred by the runtime system while performing communication tasks. Since Projections traces are integrated with the Charm++ runtime system, Figure 5b provides a more detailed breakdown of time spent doing computation. Approximately 12% of the time is spent doing fossil collection, while 55% of time is spent in the main scheduling loop executing events. The remaining 8% is spent managing events, which entails checking for causality violations and performing necessary rollbacks, as well as pushing received events onto the pending events heap of the receiving LP.

The Charm++ version of ROSS is able to reduce the time spent in communication due to three reasons. First, Charm++ is a message-driven paradigm, so ROSS does not need to actively poll the network for incoming events. This saves a significant fraction of time since Charm++ performs such polling in an efficient manner using lower level constructs. Second, the GVT is computed using a highly optimized quiescence detection mechanism in Charm++, which is based on use of asynchronous operations. Third, since the runtime system schedules execution in Charm++, it is able to overlap communication and computation automatically. Figure 6 shows the number of messages received over time during event execution between two consecutive GVT com-



(a) Usage profile of MPI ROSS

(b) Usage profile of Charm++ ROSS

Figure 5: PHOLD model at 50% remote events: Charm++ version of ROSS spends 20 – 25% time in communication, but MPI version of ROSS communicates for 45 – 50% of execution time.

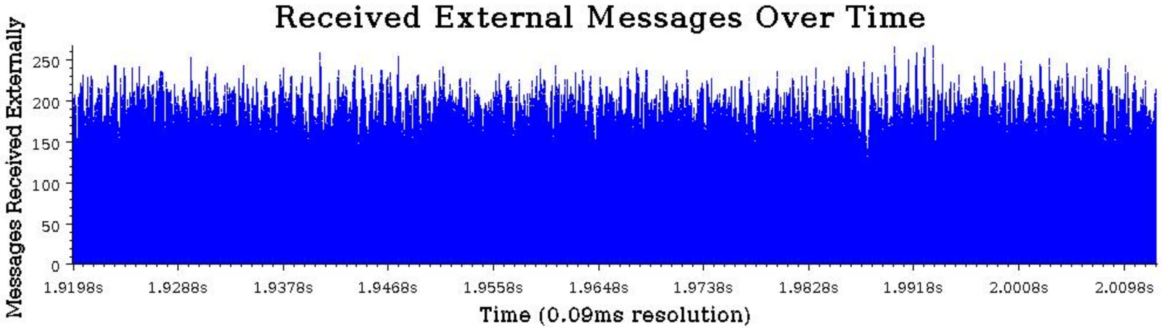


Figure 6: Charm++ version of ROSS: communication is spread uniformly over the period of event execution between GVT computation.

putations in the Charm++ version of ROSS. It shows that throughout execution, messages are actively being handled by the Charm++ runtime system. The runtime system frequently schedules the communication engine, which ensures fast overlapping communication progress.

5.2 Dragonfly

The dragonfly model used in this section, which is similar to the one described in [23], allows us to study the performance of the two ROSS versions on a real application model. The LPs in the model are of three types: routers, terminals, and MPI processes. Three different built in communication patterns are used to drive the simulation from the MPI processes: 1) Uniform random - MPI ranks send messages to randomly selected partners, 2) Transpose - every MPI rank communicates with the MPI rank situated diagonally opposite to it if the ranks are arranged in a 2D grid, and 3) Nearest Nbr - all MPI ranks connected to a given router send messages to other MPI ranks connected to the same router only. As is apparent from their descriptions, these patterns results in completely different types of communication for the simulation engine.

5.2.1 Porting Process

Although, the dragonfly model is a much more complex model than PHOLD, the porting process is still very similar to the one we described for PHOLD and involves minimal coding changes. The main difference is the use of a model specified mapping of LPs to LP Chares, instead of the default block mapping used by PHOLD. This necessitated a few key changes in the code. First, a few global variables used for mapping in the MPI version of ROSS are tied to the MPI processes, e.g. the variable that stores the number of LPs per MPI process. These variables have been either removed, or changed to be contained in the LP Chares. Second, the mapper that maps LPs to MPI processes is changed to map LPs to LP Chares instead. This change is mainly an API change, as the mapping logic itself remains the same. Third, a new mapper that returns the LP types based on their ID has been added.

5.2.2 Experimental Set up

Experiments for the dragonfly model have been done on Mira and Vulcan, a IBM Blue Gene/Q system at Lawrence Livermore National Laboratory. Allocations of sizes 512 nodes to 8,192 nodes have been used, with 64 processes being executed on every node. As mentioned above, due to the nature of BG/Q allocations, performance statistics had minimal variability between trials. Two different configurations of the dragonfly network are simulated on these supercomputers for the three traffic patterns described above.

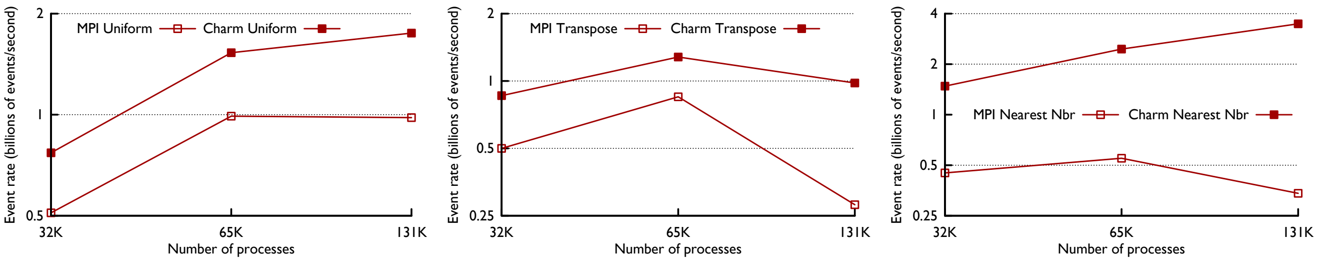


Figure 7: Performance comparison for a dragonfly with $256K$ routers and $10M$ terminals: when strong scaling is done, the Charm++ version of ROSS outperforms the MPI version.

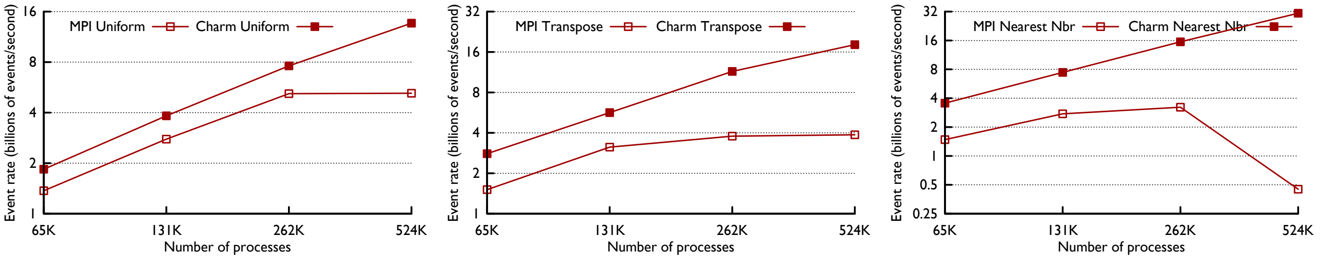


Figure 8: Strong scaling of a dragonfly with $2M$ routers and $160M$ terminals: for all traffic patterns, the Charm++ version of ROSS scales to $524K$ processes and provides up to 4× speed up over the MPI version of ROSS.

The first configuration of the dragonfly uses 80 routers per group, with 40 terminals and 40 global connection per router. This results in a system with $256,080$ routers and $10,243,200$ terminals being simulated, with one MPI process per terminal. The second larger configuration consists of 160 routers per group, with 80 terminals and 80 global connections per router. This system contains $2,048,160$ routers and $163,852,800$ terminals in total.

5.2.3 Performance Results

The $256K$ router configuration of dragonfly is simulated on $32K$ to $131K$ processes. Figure 7 shows the observed event rate for these strong scaling experiments using all three traffic patterns. It can seen that for each traffic pattern, the Charm++ version of ROSS outperforms its MPI counterpart. For two of the traffic patterns, Uniform random and Nearest Nbr, the Charm++ version provides performance gains up to $131K$ processes. In contrast, the MPI version sees a dip in performance after $65K$ processes. It is worth noting that at $131K$ processes, we are simulating the dragonfly at its extreme limits since there are only one to two routers per process. Though the number of terminals is much higher, a large fraction of communication is directed towards the routers in a dragonfly simulation, which makes it the primary simulation bottleneck.

For the larger $2M$ router configuration, Figure 8 shows distinct patterns in the performance of both versions. At every data point, the Charm++ version achieves significantly higher event rate than the MPI version. At $524K$ processes, an event rate advantage of 4× is observed for the Transpose pattern. For the Uniform random pattern, 2× improvement in the event rate is observed. The most extreme case is the Nearest Nbr pattern where we see the Charm++ version achieving more than 2× the event rate of the MPI version at $65K$ processes. This advantage increases to 5× at $262K$ processes, and skyrockets to 60× at $524K$ processes, mainly because of the drop in the performance of the MPI version.

	MPI	Charm
Uniform	51.40%	33.99%
Transpose	34.80%	5.97%
Nearest Nbr	43.11%	0.26%

Table 1: Dragonfly remote event percentage.

5.2.4 Performance Analysis

To analyze the performance for the dragonfly model, we look at two key factors: remote event percentage and efficiency. In these experiments, both these factors are impacted by how LP mapping is performed in the dragonfly model. The mapping in both versions of the dragonfly model is a modified linear mapping that maps each LP type separately. Each execution unit (MPI process in the MPI version, or LP Chare in the Charm++ version) is assigned approximately the same number of router LPs, terminal LPs, and MPI LPs. To do so, LPs are assigned IDs to preserve locality. If there are x terminals per router, then the x terminals connected to the first router are given the first x IDs, followed by the terminals connected to the second router, and so on. A similar method is taken for assigning the IDs to MPI LPs.

In an ideal scenario, when the number of routers LPs (and hence the number of terminal and MPI LPs) is a multiple of the number of processes, the above mentioned ID assignment guarantees that terminals and MPI ranks that connect to a router are mapped to the same process/execution unit. However, when the number of routers is not a multiple of the number of processes, an even distribution of terminal and MPI LPs to all processes leads to them being on processes different from their router. This results in bad performance for the MPI version of ROSS.

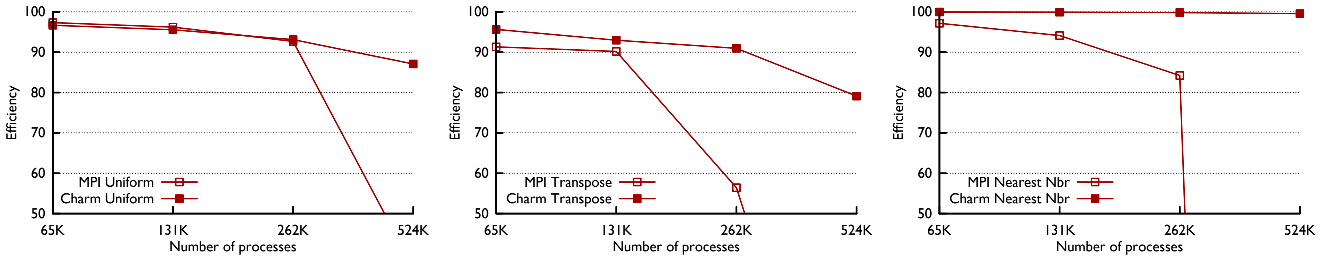


Figure 9: Efficiency comparison: as the process count increases, the efficiency of the CHARM++ version of ROSS decreases at a much slower rate in comparison to the MPI version.

In the CHARM++ version of ROSS this issue is easy to handle. Since the number of LP Chares can be chosen at runtime, we can always ensure that terminal LPs and MPI LPs are co-located with their routers on a single LP chare. This advantage of the CHARM++ version of ROSS follows from the fact that CHARM++ frees the programmer from having to worry about the specifics of the hardware the application is executed on. Instead, the work units of a CHARM++ application are the chares, and we have the flexibility to set the number of chares to yield the best performance. In this particular experiment, the best performance was achieved when there was exactly one router (and it's associated terminals and MPI processes) per chare.

Table 1 provides empirical evidence for the issue described above by presenting the average remote communication for each of the traffic patterns simulated in the dragonfly model. Here, remote communication is the percentage of committed events that had to be sent remotely. It does not include anti-events, or remote events that were rolled back. It is easy to see that the CHARM++ version requires less remote communication for every pattern, since many of the events are sent among LPs within the same chare. This is especially evident in the Nearest Nbr pattern where the CHARM++ version has less than 1% remote events in contrast to 43% remote events for the MPI version.

The mapping and its impact on remote events also directly affects the efficiency of each model. Figure 9 shows the efficiency for the different traffic patterns simulated in the dragonfly model. Here, efficiency is calculated as $1 - \frac{R}{N}$ where R is the number of events rolled back, and N is the total number of events executed. We see that the effect is particularly pronounced for the Nearest Nbr traffic pattern. In the CHARM++ version, nearly all events are sent locally, so there is very little chance for causality errors to occur. Because of this the CHARM++ version maintains 99% efficiency at all process counts, whereas the MPI version drops significantly in efficiency as the process count increases, eventually reaching an efficiency of -705% at $524K$ processes.

6. CONCLUSION

In this paper we have shown the suitability of CHARM++, a parallel adaptive runtime system, for designing and implementing PDES engines and models. The re-targeting of ROSS from MPI has simplified and reduced the ROSS code base, while simultaneously enabling new features such as asynchronous GVT computation and dynamic load balancing. Furthermore, the CHARM++ version of ROSS provides significantly better performance in comparison to its MPI counterpart. In the paper, we showed that as the communication volume increases in the PHOLD benchmark, the gap in the performance of CHARM++ and MPI version also increases. For the dragonfly model, irrespective of the communication pattern being simulated, the CHARM++ version not only provides higher event rate, it also scales to higher core counts. Moreover, due to the features discussed in this paper (checkpointing and load balancing), the CHARM++ version of ROSS is also better suited for simulating dynamic and complex models with long run times.

7. ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory (LDRD project 14-ERD-062 under Contract DE-AC52-07NA27344). This work used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357 (project allocations: PEACEndStation, PARTS, CharmRTS). This work also used resources from the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois.

8. REFERENCES

[1] The charm++ parallel programming system manual. http://charm.cs.illinois.edu/manuals/html/charm++/manual.html, visited 2016-3-20.

[2] MPI: A Message Passing Interface Standard. In *MPI Forum*. http://www.mpi-forum.org/, visited 2016-03-20.

[3] Ross source code on github. https://github.com/carothersc/ROSS, visited 2016-03-20.

[4] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale. Parallel Programming with Migratable Objects: Charm++ in Practice. SC, 2014.

[5] P. D. Barnes, Jr., C. D. Carothers, D. R. Jefferson, and J. M. LaPre. Warp speed: executing time warp on 1,966,080 cores. In *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation*, SIGSIM-PADS '13, pages 327–336, New York, NY, USA, 2013. ACM.

[6] D. W. Bauer Jr., C. D. Carothers, and A. Holder. Scalable time warp on blue gene supercomputers. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd*

Workshop on Principles of Advanced and Distributed Simulation, pages 35–44, Washington, DC, USA, 2009. IEEE Computer Society.

[7] C. D. Carothers, D. Bauer, and S. Pearce. ROSS: A high-performance, low-memory, modular Time Warp system. *Journal of Parallel and Distributed Computing*, 62(11):1648–1669, 2002.

[8] C. D. Carothers and K. S. Perumalla. On deciding between conservative and optimistic approaches on massively parallel platforms. In *Winter Simulation Conference'10*, pages 678–687, 2010.

[9] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Trans. Model. Comput. Simul.*, 9(3):224–253, July 1999.

[10] M. Choe and C. Tropper. On learning algorithms and balancing loads in time warp. In *Workshop on Parallel and Distributed Simulation*, pages 101–108, 1999.

[11] I.-H. Chung, R. E. Walkup, H.-F. Wen, and H. Yu. MPI tools and performance studies—MPI performance analysis tools on Blue Gene/L. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 123, New York, NY, USA, 2006. ACM Press.

[12] E. Deelman and B. K. Szymanski. Dynamic load balancing in parallel discrete event simulation for spatially explicit problems. In *Workshop on Parallel and Distributed Simulation*, pages 46–53, 1998.

[13] R. Fujimoto. Parallel Discrete Event Simulation. *Comm. of the ACM*, 33(10):30–53, 1990.

[14] F. Gygi, E. W. Draeger, M. Schulz, B. R. de Supinski, J. A. Gunnels, V. Austel, J. C. Sexton, F. Franchetti, S. Kral, C. W. Ueberhuber, and J. Lorenz. Large-scale electronic structure calculations of high-z metals on the bluegene/l platform. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[15] N. Jain, A. Bhatele, J.-S. Yeom, M. F. Adams, F. Miniati, C. Mei, and L. V. Kale. Charm++ & MPI: Combining the best of both worlds. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (to appear)*, IPDPS '15. IEEE Computer Society, May 2015. LLNL-CONF-663041.

[16] N. Jain, E. Bohm, E. Mikida, S. Mandal, M. Kim, P. Jindal, Q. Li, S. Ismail-Beigi, G. Martyna, and L. Kale. Openatom: Scalable ab-initio molecular dynamics with diverse capabilities. In *International Supercomputing Conference*, ISC HPC '16 (to appear), 2016.

[17] D. Jefferson and H. Sowizral. Fast Concurrent Simulation Using the Time Warp Mechanism. In *Proceedings of the Conference on Distributed Simulation*, pages 63–69, July 1985.

[18] L. V. Kale, G. Zheng, C. W. Lee, and S. Kumar. Scaling applications to massively parallel machines using projections performance analysis tool. In *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, volume 22, pages 347–358, February 2006.

[19] N. Liu, C. Carothers, J. Cope, P. Carns, R. Ross, A. Crume, and C. Maltzahn. Modeling a leadership-scale storage system. In *Proceedings of the 9th international conference on Parallel Processing and Applied Mathematics - Volume Part I*, PPAM'11, pages 10–19, Berlin, Heidelberg, 2012. Springer-Verlag.

[20] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *Proceedings of the 2012 IEEE Conference on Massive Data Storage*, Pacific Grove, CA, Apr. 2012.

[21] F. Mattern. Efficient algorithms for distributed snapshopts and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18:423–434, 1993.

[22] H. Menon, L. Wesolowski, G. Zheng, P. Jetley, L. Kale, T. Quinn, and F. Governato. Adaptive techniques for clustered n-body cosmological simulations. *Computational Astrophysics and Cosmology*, 2(1):1–16, 2015.

[23] R. B. R. Misbah Mubarak, Christopher D. Carothers and P. Carns. Modeling a million-node dragonfly network using massively parallel discrete-event simulation. *SCC, SC Companion*, 2012.

[24] R. B. R. Misbah Mubarak, Christopher D. Carothers and P. Carns. A case study in using massively parallel simulation for extreme-scale torus network codesign. In *Proceedings of the 2nd ACM SIGSIM PADS*, pages 27–38. ACM, 2014.

[25] M. Mubarak, C. D. Carothers, R. Ross, and P. Carns. Modeling a million-node dragonfly network using massively parallel discrete-event simulation. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion,*, pages 366–376. IEEE, 2012.

[26] D. M. Nicol. The cost of conservative synchronization in parallel discrete event simulations. *J. ACM.*

[27] J. Phillips, G. Zheng, and L. V. Kalé. Namd: Biomolecular simulation on thousands of processors. In *Workshop: Scaling to New Heights*, Pittsburgh, PA, May 2002.

[28] A. B. Sinha, L. V. Kale, and B. Ramkumar. A dynamic and adaptive quiescence detection algorithm. Technical Report 93-11, Parallel Programming Laboratory, Department of Computer Science , University of Illinois, Urbana-Champaign, 1993.

[29] L. Wesolowski, R. Venkataraman, A. Gupta, J.-S. Yeom, K. Bisset, Y. Sun, P. Jetley, T. R. Quinn, and L. V. Kale. TRAM: Optimizing Fine-grained Communication with Topological Routing and Aggregation of Messages. In *Proceedings of the International Conference on Parallel Processing*, ICPP '14, Minneapolis, MN, September 2014.

[30] T. L. Wilmarth. *POSE: Scalable General-purpose Parallel Discrete Event Simulation.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[31] J.-S. Yeom, A. Bhatele, K. R. Bisset, E. Bohm, A. Gupta, L. V. Kale, M. Marathe, D. S. Nikolopoulos, M. Schulz, and L. Wesolowski. Overcoming the scalability challenges of epidemic simulations on blue waters. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, IPDPS '14. IEEE Computer Society, May 2014.