Charm++ & MPI: Combining the Best of Both Worlds

Nikhil Jain*, Abhinav Bhatele†, Jae-Seung Yeom†, Mark F. Adams§, Francesco Miniati¶, Chao Mei||, Laxmikant V. Kale*

*Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801 USA
†Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, California 94551 USA
§Lawrence Berkeley National Laboratory, Berkeley, California 94720 USA
¶Institute for Astronomy, ETH Zurich, 8093 Zurich, Switzerland
||Google Inc., Mountain View, California 94043 USA
E-mail: *nikhil@illinois.edu, †bhatele@llnl.gov

***Abstract*—Charm++ and MPI embody two distinct perspectives for writing parallel programs. While MPI provides a process-centric, user-driven model for developing parallel codes, Charm++ supports work-centric, system-driven parallel programming. One of them might be a better or more natural fit for individual modules that constitute a parallel application. In this paper, we present a framework that enables hybrid parallel programming with Charm++ and MPI, and allows programmers to develop different modules of a parallel application in these two languages while facilitating smooth interoperation. We describe the challenges in enabling interoperation between Charm++ and MPI, and present techniques for managing the control flow and resource sharing between the two. Finally, we demonstrate the benefits of interoperation between Charm++ and MPI through several case studies that use production applications and libraries, including CHARM/Chombo, EpiSimdemics, NAMD, FFTW, MPI-IO and ParMETIS.**

I. Introduction

The increasing computational power of supercomputers is expected to lead to significant breakthroughs in science and engineering. These breakthroughs will come from accurate predictions arising from faithful modeling of physical phenomena, which in turn will require multi-physics modeling and coupled simulations. Further, effective use of increased computational power will require more sophisticated techniques such as dynamic adaptive refinements.

The toolbox of the parallel programmer should also be rich to match the complexity of parallel simulations. In particular, it should include multiple programming languages. For simplicity, we use the term *language* to refer to all entities that provide a mechanism for writing a parallel program – a communication library, a runtime system, a programming model, a compiler-supported parallel language, or another form of expression. Different languages provide various *features* that are instrumental when designing and implementing parallel applications in a particular language.

Most of the existing parallel applications are implemented in a single language. Hence, they are limited in their ability to exploit features provided by multiple languages. As modern parallel codes get more complex and are implemented as a collection of several diverse modules, this limitation can severely impact the productivity of the programmer and the performance of the program. This is because features from different programming languages might be the best or most natural fit for each of these diverse modules. Moreover, the restriction of using one language prevents reuse of parallel software developed in different languages.

To overcome this restriction, the high performance computing (HPC) community is actively exploring interoperation of multiple parallel programming languages within an application. For example, OpenMP [1] is frequently used with MPI [2] to facilitate intra-node shared memory parallelism [3], [4]. In this paper, we explore the combination of Charm++ [5] and MPI [2] for writing parallel applications. We describe the challenges we faced in the process, the methodology we developed to enable interoperation, and the applicability and scalability of the proposed framework. Our primary goal is to use interoperation in multi-module parallel applications, wherein the combined use of Charm++ and MPI for implementing distinct modules can lead to better performance, increased productivity, and code reuse.

The biggest challenge in enabling interoperation between Charm++ and MPI is the management and transfer of control flow between these languages because they have different models for driving program execution. MPI is a process-centric, user-driven language where the programmer explicitly defines the control flow, while Charm++ is a work-centric, overdecomposition-based, system-driven language. In Charm++ programs, a runtime system drives the execution based on the availability of data.

Simultaneous use of Charm++ and MPI in a parallel application also requires sharing of system resources such as processing elements and the interconnection network, and information such as the application data. Depending on the application in consideration and the machine being used, the best method to enable this sharing may also vary, which poses a significant challenge.

Finally, for wide-spread acceptance, it is critical that the interoperation methodology for combining Charm++ and MPI be easy to use and provide scalable performance. Moreover, code reuse can be ensured only if minor modifications are required for interoperation.

In the rest of the paper, we present a framework for enabling interoperation between Charm++ and MPI that addresses the above mentioned challenges. Four case studies based

on production codes such as CHARM/Chombo [6], EpiSimdemics [7], and NAMD [8], and libraries including FFTW [9], MPI-IO, and ParMETIS [10], executed on thousands of cores of IBM Blue Gene/Q and Cray XE6 are presented. These examples establish the utility of hybrid programming using Charm++ and MPI in exploiting the best features of each language and eliminating performance bottlenecks in the applications with minimal effort. The case studies also demonstrate that interoperation leads to code reuse and eases programmers' burden by allowing them to use features from different programming languages that match the requirements of the individual application modules.

II. Background

MPI [2], the de facto standard for parallel programming, provides a process-centric model with a localized view of application data within a process. Data domains are typically divided among MPI processes mapped one-to-one to physical processors; two-sided and global communication is performed to exchange information among these processes. MPI is a quintessential *user-driven language* in which the program control flow is explicitly defined by the programmer. Data exchanges among MPI processes are mostly pre-determined and the execution is defined as a single flow of control. Exceptions such as `MPI_ANY_*` exist, wherein the programmer delegates the ordering to the system, but are not commonly used.

Charm++ [5] is a work-centric overdecomposition-based programming language, which also provides a localized view of application data but within a C++ object. Application domains are divided among *data* objects and computation is divided among *work* objects. These objects are chosen based on application requirements, independent of the number of processors. Charm++ is a *system-driven language* in which a runtime system (RTS) decides what computation to execute next based on the availability of data for various objects. This execution model allows for many concurrent control flows, with progress driven by the availability of data (data-driven execution) and the RTS's guidelines.

Table I lists some language features and their relative ease-of-use in MPI and Charm++. In MPI, it is easy to take advantage of important features such as expression of global control flow and global communication. However, it may not be ideal for a dynamic, data-dependent control flow due to limited support for load balancing and handling message-driven interactions. Significant changes are required to provide such support in MPI [11], [12].

In contrast to MPI, Charm++ can effectively adapt to dynamic environments due to its powerful RTS that enables automatic load balancing and message-driven interactions. However, the inability to easily express global control flow and difficulty in performing global communication limits its use in some scenarios. Given these limitations of MPI and Charm++, a desirable end-point is a framework that allows the programmer to use either MPI or Charm++ for different modules in their applications and have them interoperate.

TABLE I
RELATIVE EASE-OF-USE OF VARIOUS FEATURES IN MPI AND CHARM++.

Language Feature	MPI	Charm++
Express Global Control Flow	Easy	Hard
Message-driven Interaction	Hard	Easy
One-sided Interaction	Hard	Easy
Global Communication	Easy	Hard
Exploit Comm.-Comp. Overlap	Hard	Easy
Concurrency Management	Easy	Hard
Load Balancing	Hard	Easy
Existing Libraries	Many	Few

A. Related Work

In parallel computing, interoperation was initially explored by Harper who developed a library that allowed programs written for version 3 of the Parallel Virtual Machine (PVM [13]) to execute in the Legion environment. A runtime library, Meta-Chaos, was developed to enable data exchange between data parallel programs written using High Performance Fortran, Chaos, Multiblock Parti libraries and pC++ [14]. Kale et al. [15] proposed and demonstrated the use of a common runtime framework (Converse) for interoperation of various parallel programming languages such as MPI [2], PVM [13] and Charm++ [16].

More recently, STAPL [17] has provided the capability to use third party libraries in its programs. Hybrid use of MPI [2] and OpenMP [1] has received significant attention and has been widely adopted [18], [19]. Use of MPI in Unified Parallel C [20] programs, where MPI is available as an additional communication interface, has also been explored [21]. Efforts have also been made by MPI implementors to facilitate interoperation and support other languages. Zhao et al. [12] present an extension to MPI which supports asynchronous active messages that may overlap with other communication in MPI applications. Dinan et al. [22] have proposed adding flexible communication end-points to MPI to relax the one-to-one relation between processes and MPI ranks.

The research in this paper differs from previous work in several aspects. We demonstrate the interoperation of languages that control parallelism for the entire machine (intra-node and inter-node) and have very different control flow styles – one is user-driven and the other is system-driven. We also focus on the capability to reuse existing code written in diverse languages. These have not been attempted before. For MPI and Charm++ interoperation, unlike the previous work that is only usable with MPI reimplemented on top of Converse [15], we present a method that requires minimal changes to both MPI and Charm++, and works with any MPI implementation.

III. Control Flow Management

The simultaneous use of Charm++ and MPI for writing a parallel program raises several interesting questions about managing different aspects of the program. Among these, *management and transfer of control* between them is critical, even more so because the languages differ with respect to the driver of program execution. In this section, we attempt

to answer the following important questions: 1) How many control flows should be used to execute different language modules? 2) How should the control be transferred from one language module to another? 3) How frequently should the control be transferred?

When interoperating between two languages of the same type (user-driven or system-driven), transfer of control from one language to another is simple. For example, in hybrid programming with MPI and UPC, the user explicitly drives the program with the control being returned to them after every system-invocation [21]. In contrast, for hybrid programming with Charm++ and MPI, there is no obvious solution. If the execution begins in MPI, there exists no mechanism to progress Charm++-based modules as the *RTS is hidden from the user*. Similarly, if the execution begins in Charm++, there exists no mechanism to transfer the control to MPI because *the RTS does not support yielding control to the user*.

Solution I – Concurrent Flows: One possible solution is to avoid the need for transfer of control by using *concurrent flows*. In this method, Charm++ and MPI modules are executed in their own *home* threads (e.g. pthreads). Thus, these modules make progress when their home threads are scheduled. While this scheme is simple and easy to use, it may lead to significant performance problems.

First, the overhead of scheduling threads can impact performance negatively. Second, a default time-sharing based scheduling of the threads on processors may result in significant idle time. While a module in one language wastes cycles busy-waiting for data, other modules that could have used these cycles will have to wait for their turn. Such waste is even higher for our primary use case where modules are written in the language suitable for them, and sequential dependencies may exist among these modules. Third, although the performance degradation caused due to busy-waiting can be addressed by an idle module voluntarily yielding control, implementing this can require significant effort. Such a solution is feasible only if extensive changes are made to the existing implementations of Charm++ and MPI.

Solution II – Exposing the Scheduler: In light of the drawbacks of the approach discussed above, we propose an alternate solution that executes different modules using a single control flow. This is made feasible by *exposing the scheduler* in the Charm++ RTS and empowering the user to control it. In this approach, the program execution begins in MPI, wherein MPI semantics are followed (Figure 1, Step 1). When required, the exposed Charm++ scheduler is activated (Step 2). From this point, the execution is driven by the Charm++ RTS following its semantics (Step 3). At a later time, the scheduler is explicitly deactivated and the control is returned back to MPI (Figure 1, Step 4). Next, the MPI module may again activate the Charm++ scheduler, and hence repeat the cycle. This approach eliminates the major disadvantages of the previous one – no thread scheduling overheads, minimal busy-waiting for module-based applications, and limited changes to Charm++ and MPI.

The idea of exposing Charm++'s scheduler is not flawless. It increases programmer's burden by demanding explicit control transfer. It also prevents seamless automated exchange of control between Charm++ and MPI that concurrent flows can provide. However, these negatives are minimized by the use of a module-based design for interoperable programs. Note that seamless automated exchange between Charm++ and MPI can also be provided if both of them are reimplemented on top of a common lower-level runtime. Such an exploration is out of scope for this paper because we aim at minimal changes to Charm++, MPI, and application codes to enable interoperation.

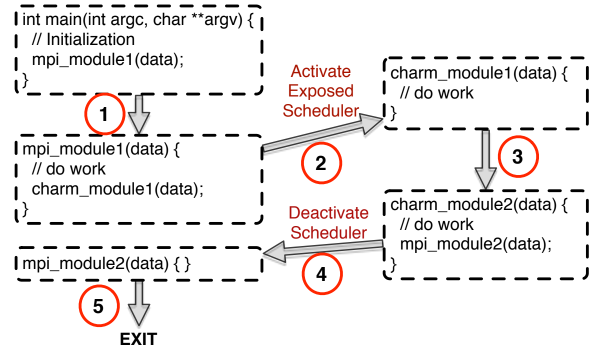

Fig. 1. Flowchart showing the steps in transfer of control between MPI and Charm++ using an exposed scheduler

Control Transfer Frequency: Explicit transfer of control by the user leads to another important question – how frequently should the control be transferred? If an application is written in two user-driven languages, say MPI and UPC, the programmer is encouraged to make a fine-grained selection between MPI and UPC calls, i.e. for every communication operation, select between MPI or UPC calls [21]. However, a frequent transfer of control between the user and the system may have a negative impact on productivity and performance. Since the availability of data drives the execution in Charm++, the user will have to consider all possible orderings to exchange control with MPI safely. This may be significantly more demanding if the control is transferred frequently and may result in deadlocks. At the same time, performance degradation may be observed if there is a mismatch in the modules executed by different processes, which may block one or the other, leading to idle time.

The disadvantages of frequent control transfer listed in the previous paragraph are eliminated by transferring the control between Charm++ and and MPI infrequently. This leads to ease of use and maintains simplicity of interoperation between the two languages. Coarse-grained control transfer also allows for reuse of independent modules/libraries as one unit without significant modifications.

IV. Writing Hybrid Code in Charm++ and MPI

Based on the discussion in Section III, we have developed a framework that enables interoperation between Charm++ and MPI. In this framework, both languages are augmented to

provide the following constructs that allow hybrid applications to be developed in them:

- **Initialize:** given a set of processes, perform setup such as identifying rank space, initializing low-level communication substrate, etc. to create a language instance.
- **Execute:** make progress in the given language instance following the semantics of the associated language.
- **Transfer:** stop execution in this instance in order to transfer control to another instance.
- **Clean up:** destroy the language instance.

All of these constructs already exist in the MPI standard. Along with `MPI_Init`, creation of a sub-communicator is sufficient to perform the initialization. *Execute* and *transfer* constructs are implicitly available since every MPI call returns the control back to the user after completion. Freeing the communicator and `MPI_Finalize` perform the necessary clean up.

For Charm++, a new API has been added to perform these tasks. `CharmLibInit` initializes a Charm++ instance for a given set of processes. In order to execute a Charm++ module, one should invoke `StartCharmScheduler` to transfer control to the Charm++ RTS. The scheduler can be stopped either on a single process using `StopCharmScheduler` or collectively on the full set of processes by calling `CkExit`. Finally, clean up is performed by invoking `CharmLibExit`.

For a programmer, developing hybrid MPI-Charm++ programs and enabling the use of stand-alone MPI and Charm++ modules in them requires *minor* additional work. Other than including the necessary headers, following is a list of *all the required* steps needed to write an interoperable program that uses both Charm++ and MPI.

Common Tasks: Initialize MPI, create sub-communicator(s), initialize Charm++ instance(s), destroy Charm++ instance(s), free sub-communicator(s), finalize MPI.

MPI Module: Provide a C/C++ interface function to transfer control to the module; to transfer control to Charm++ modules, call interface function provided by the Charm++ modules.

Charm++ Module: Provide an interface function callable from MPI – this interface function should initiate start up messages to the module and activate the Charm++ RTS; to transfer control to MPI modules, call interface function provided by the MPI modules.

The code snippet in Figure 2 shows the MPI portion of a hybrid program with all the changes required to interoperate with a Charm++ module. As usual, execution begins in `main` and `MPI_Init` is invoked first. After that, the processes are divided into two sets by creating sub-communicators. One set of processes continues with MPI work while Charm++ is initialized on the other. This second set of processes creates the Charm++ instance, invokes the Charm++ module and after returning, destroys the Charm++ instance. If needed, control can be transferred back and forth multiple times between MPI and Charm++ modules before the instance is destroyed.

When using a Charm++ module for interoperation, execution in Charm++ begins only when it is invoked explicitly

```
#include "mpi-interoperate.h"

int main(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_split(MPI_COMM_WORLD, myrank%2, myrank, &
       newComm);
  if(myrank % 2) {
    // Create Charm++ instance on subset of processes
    CharmLibInit(newComm, argc, argv);
    StartCharm(16); // Call Charm++ library
    CharmLibExit(); // Destroy Charm++ instance
  } else {
    // MPI work on rest of the processes
  }
  MPI_Finalize();
}
```

Fig. 2. Interoperable MPI module: create MPI and Charm++ instances; transfer control by calling simple interface functions.

```
#include "mpi-interoperate.h"

// invoked from MPI, marks the begining of Charm++
void StartCharm(int elems) {
  if(CkMyPe() == 0) {
    workerProxy.StartWork(elems);
  }
  StartCharmScheduler();
}

// Charm++ function that deactivates scheduler
void Worker::StartWork(int elems) {
  // Charm++ work on a subset of processes
  CkExit();
}
```

Fig. 3. Interoperable Charm++ module: provides an interface function that invokes the Charm++ scheduler.

by initiating a message to one of its objects and starting the Charm++ scheduler using `StartCharmScheduler`. In the code snippet in Figure 3, `StartCharm` is an interface function that performs these tasks. On process 0, a message is initiated to the `Worker` objects after which all processes activate the Charm++ scheduler. In this simple example, when the RTS receives the StartWork message and schedules it, the objects do some work and then call `CkExit` to collectively stop the scheduler on all processes, thus returning control to the MPI interface function.

V. Sharing Resources and Data

In the framework presented above, the presence of both Charm++ and MPI requires explicit coordination of certain aspects that are otherwise handled by the language implementations. We focus on two such important issues here – 1) How are resources shared? and 2) How is data shared? (between Charm++ and MPI).

A. Resource Sharing

Execution of modules written in Charm++ and MPI on the same physical resources is only possible through the sharing of hardware such as cores, the memory subsystem and the network. These resources can be allocated to individual

modules either explicitly by the programmer, or implicitly by the framework based on the preferences expressed in the application. Figure 4 presents three schemes provided in our framework for sharing resources – time division, space division and hybrid division.

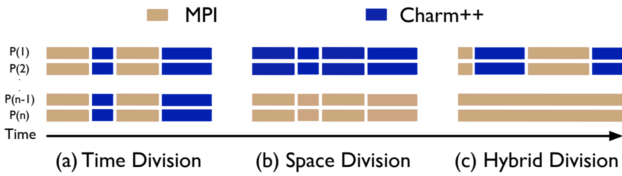

Fig. 4. Different schemes for sharing resources between Charm++ and MPI.

Time Division: In this mode, during the execution of an application, all the processes switch from one language module to another synchronously. As depicted in Figure 4(a), the execution of an application begins in one of the languages. At some point in the execution, all the processes switch to executing a module written in another language. This exchange may happen multiple times before the application exits. This method of interoperation is useful for applications that have a constraint on the ordering of the tasks to be executed in different language modules.

Space Division: Instead of time slicing the resources, this mode assigns subsets of processes to different languages for the entire duration of the program execution. Figure 4(b) shows this scenario in which modules written in Charm++ run on some of the processes, while modules written in MPI run on the rest. Space division is useful for making simultaneous progress in modules that can be executed in parallel and are loosely connected to one another.

Hybrid Division: A combination of time division and space division provides a hybrid method of resource sharing. In this scheme, subsets of processes execute modules written in Charm++ and MPI. Different subsets may execute different modules independently of other subsets. For example, in Figure 4(c), a subset of processes transfer control between modules written in Charm++ and MPI, while another subset executes modules written in MPI only. A hybrid model of interoperation can be particularly useful in applications that require different subsets to perform different tasks during application execution.

Simultaneous use of low-level resources such as network FIFOs and links by multiple high-level language clients may require a customized solution for each type of hardware. In Section VI, we describe the mechanisms used on machines such as IBM Blue Gene/Q and Cray XE6 to divide low-level resources between Charm++ and MPI.

B. Data Sharing

Modules implemented in Charm++ and MPI need to exchange data during program execution. Unlike programs written in a single language, it is not possible to invoke regular communication mechanisms, e.g. it is not possible to invoke an `MPI_Send` for sending data from an MPI module to a Charm++ module. To solve these problems, the following schemes are supported.

Pointer-based Data Sharing: This simple method is based on exchanging data by explicitly passing memory pointers. If data is to be transferred between Charm++ and MPI instances within a process, it can be directly exchanged via use of reserved memory space. However, if the data is to be transferred between Charm++ and MPI instances on different processes, a two step mechanism is needed. First, the data is transferred to the destination process using the source language (Charm++ or MPI), and then it is transferred to the destination language using shared memory (alternatively, transfer to the destination language first, and then to the destination process).

It is obvious that this mechanism puts the entire burden of data exchange on the programmer. In addition to implementing the code responsible for data transfer, the programmer is also responsible for ensuring correctness and avoiding race conditions. However, this scheme is very flexible, and is often the best option if few data exchanges are performed.

Data Transfer Repository: Alternatively, a generic data transfer repository can be used for intra-process and inter-process communication. An API is used for depositing and retrieving data to and from the local client modules in Charm++ and MPI (a pull model). Under the hood, the data transfer repository communicates with its counterparts on other processes to service the requests.

Use of a data transfer repository increases productivity as it leads to code reuse and relieves the end user from the burden. It also allows for implementing more complex schemes for data exchange that may be used by a wide range of applications. For example, in addition to data exchange via deposition and retrieval based on source and destination, data can be elevated to being named entities and be universally accessible (as is done by PGAS languages [20]).

C. Rank Mapping

Dinan et al. [21], [22] have provided various alternatives for managing the *rank space* between interoperable MPI and UPC modules. We believe that the *flat* and *nested* models proposed by them are adequate for MPI-Charm++ interoperation. We will refer to their *flat* model as a *one-to-one* mapping – for every rank in one module, a corresponding rank exists in other modules on the same process. The *nested* model can be seen as a *one-to-many* mapping – for every rank in one module, multiple ranks exist in other modules on the same process. However, in the case of space division of processes, the rank mapping is neither one-to-one nor one-to-many. If ranks for certain modules are not available on certain processes, we refer to this mapping as *one-to-none* or *many-to-none*.

VI. Implementation of the Framework

Next, we explain the sharing of low-level resources between modules in the MPI-Charm++ interoperation framework.

A. Communication Substrate

Inter-process communication in most languages is implemented using a low-level communication API exposed by the machine, e.g. PAMI on IBM Blue Gene/Q and uGNI on Cray XE6. The presence of multiple language modules requires that the communication started by one be delivered to the corresponding receiver module at the destination. For each language, we use distinct communication domains in the low-level API to ensure this property.

For Cray's uGNI API, a domain is created by `GNI_CdmCreate`, which enables interoperation on Cray machines such as Cray XE6/XK7 (Blue Waters, Titan, Hopper) and Cray XC30 (Edison). When using PAMI on IBM Blue Gene/Q (Sequoia, Mira), communication is isolated by creating a distinct communication client for each module using `PAMI_Client_create`. Alternatively on Blue Gene/Q, it is possible to register distinct *dispatch IDs* with a common communication client for different modules. This approach may be better since it avoids a static division of resources among the clients. However, we use the former approach in our framework due to the unavailability of the client created by MPI outside of its implementation.

An alternative, which has also been implemented, is to use MPI as the communication substrate for Charm++. This enables interoperation between Charm++ and MPI on any system that supports MPI. A potential disadvantage of this approach is the lower performance of Charm++ built on top of MPI in comparison to a low-level communication API.

B. Resource Sharing

The three resource sharing schemes described in Section V-A are implemented by means of MPI communicators. The user splits the given set of processes into sub-communicators that should execute various modules. The correct sub-communicator is passed to Charm++ as an argument during its initialization. If Charm++ is built on top of MPI, the sub-communicator is passed directly as an argument in the communication calls, thus dividing the set of processes and their communication in a manner that most MPI programmers are familiar with. If Charm++ is not built on top of MPI, the RTS uses this information to find the set of processes on which the given Charm++ instance should be initialized.

C. Data Repository

The repository for data exchange has been implemented as a C++ module, which uses Charm++ for communication. To keep things simple, the current interface to the data repository is not generic, but is customized based on the application needs. As a result, depending on the application, the data repository stores different data types. Work on generalizing the data repository (using templates and related concepts) is under progress.

D. Multi-threading

Unlike MPI, Charm++ can be built in a special shared memory mode. In this setup, the RTS launches only one Charm++ process for each multi-socket compute node. The RTS spawns multiple threads within that process, which share their address space and a communication thread. In MPI, similar shared memory optimizations are typically enabled via OpenMP [1]. Currently, our framework supports interoperation only if both MPI and Charm++ are being used in similar modes, i.e. if MPI has one rank per compute node, Charm++ will also have one process per compute node. In this scenario, both MPI and Charm++ spawn threads of their own to enable shared memory based optimizations.

VII. Application Studies

An important goal of this study is to explore the ease-of-use and benefits of hybrid programming with Charm++ and MPI. This section examines the interoperation of the two languages in production settings wherein for applications written in one language, a module written in another language is added (or swapped). These examples, summarized in Table II, demonstrate the productivity and performance benefits derived from the synergistic existence of Charm++ and MPI.

A. CHARM/Chombo (MPI) and HistSort (Charm++)

Our first example explores the use of a Charm++-based parallel sorting library, HistSort [23], in a production cosmological and astrophysical code called CHARM [6] (not to be confused with Charm++). CHARM is implemented on top of the Chombo framework [24] which is written in MPI. Use of HistSort in CHARM eliminates a performance bottleneck in the code that arises from a critical global sort operation, and hence enables CHARM to scale to large core counts.

CHARM, and cosmology codes in general, often have very non-uniform particle distributions. Load balance and data locality of the particles, with respect to the mesh, are of critical importance for the performance of such particle-in-cell (PIC) codes. To optimize load balance and data locality (and hence optimize particle-mesh interactions), CHARM takes the approach of periodically sorting particles with a space-filling curve index. This global sorting of particles is a critical component of this algorithm but has been a scalability bottleneck in its current implementation.

Figure 5 shows *all* the changes that were made to make Charm++'s HistSort an interoperable library callable from any MPI program, and its use in CHARM. The interface function shown in Figure 5 (right) performs the actions described in Section IV – initiate a message to the main object and activate the Charm++ RTS. CHARM uses HistSort by invoking the interface function instead of the default Multiway-merge Sort implementation as shown in Figure 5 (left).

Benefits: Charm++ is a suitable candidate for performing an operation such as sorting because of the features it provides (Table I): message-driven interaction and ease of exploiting communication-computation overlap. Moreover, a highly scalable histogram-based sorting library, HistSort, already exists in Charm++ [23].

Resource Sharing: The global sorting in CHARM needs to be performed in every iteration before the computation

```
Original CHARM code with Multi-way Merge Sort
/* CHARM code that prepares the input */
...
195 lines of Multi-way Merge sort in MPI
/* Computation code in CHARM */
...

--------------------------------------------------

/* CHARM code that prepares the input */
...
// call to HistSort
HistSorting<key_type, std::pair<partType,
     char[MAX_PART_SZ]>>(loc_s_len, dataIn,
     &loc_r_len, &dataOut);
/* Computation code in CHARM */
...
Modified CHARM code with Charm++'s HistSort
```

```
// interface function for HistSort
template <class key, class value>
void HistSorting(int input_elems_, kv_pair<key,
     value>* dataIn_, int * output_elems_, kv_pair<
     key, value>** dataOut_) {
  // store parameters to global locations
  dataIn = (void*)dataIn_;
  dataOut = (void**)dataOut_;
  in_elems = input_elems_;
  out_elems = output_elems_;
  // initiate message to main object
  if(CkMyPe() == 0) {
    static CProxy_Main<key,value> mainProxy =
        CProxy_Main<key,value>::ckNew(CkNumPes());
    mainProxy.DataReady();
  }
  StartCharmScheduler();
}
```

Fig. 5. Modifications required to transfer control from CHARM to HistSort (left); The interface function in HistSort callable from any MPI program (right).

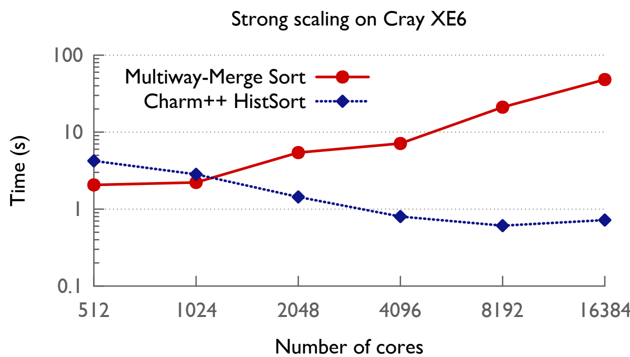

Fig. 6. CHARM using Charm++'s histogram sorting library: scaling bottleneck caused due to sorting is resolved using Charm++'s HistSort library.

of particle-mesh interactions can proceed. This dependency suggests that a *time division* of the resources between HistSort (Charm++) and CHARM (MPI) with *one-to-one* rank mapping would be ideal.

Data Sharing: The data is explicitly transferred between CHARM and HistSort using local memory pointers. These pointers are passed between the modules when the MPI code invokes HistSort through a simple C++ function call. This is possible because CHARM stores the data in a distributed manner that matches the input/output of HistSort.

Figure 6 compares the performance of HistSort with Multiway-merge Sort. The plot shows the global sorting time for a strong-scaling experiment with $131,884,914$ keys (72 bytes of data attached to each key) executed on Hopper, a Cray XE6. HistSort, written in Charm++, outperforms the MPI-based Multiway-merge Sort for large core counts ($48\times$ speed up on $16,384$ cores). While the performance of Multiway-merge Sort gets worse, HistSort's performance improves significantly with increasing core count. The improvement in performance resolves the scaling bottleneck of CHARM due to sorting. In addition, replacing the sorting code in CHARM with a call to HistSort reduces its lines of code by 195.

B. EpiSimdemics (Charm++) and MPI-IO (MPI)

This second case study shows the coupling of the MPI-IO library [2] with a contagion simulation code called EpiSimdemics [7], implemented in Charm++. Use of MPI-IO enables generation of output data at scale, enables fast writing to a single file, and helps alleviate the performance bottlenecks in EpiSimdemics caused by I/O operations.

EpiSimdemics is an agent-based simulator used to study the spread of contagious diseases over social contact networks. EpiSimdemics requires three input files: the *person* file, the *location* file, and the *visit* file. The sizes of these files for the entire US population are 2.1 GB, 1 GB, and 28 GB respectively. Among the many output files of EpiSimdemics, the *disease* and *dendogram* files are of large sizes. These two files, in addition to the one recording the summary of global simulation states, allow the scientists to understand the simulation results in detail.

Given the large input files, the use of sequential input is a performance bottleneck in EpiSimdemics. Performance tests using sequential input showed that while the actual simulation may complete in tens of minutes, the *setup* including the input takes approximately an hour! EpiSimdemics has a custom, application-specific parallel output scheme in which the output is written by all processes to distinct files. This scheme is good for performance but requires post-processing of data before it can be used for any analysis. Also, due to a limitation on the number of file descriptors per job on Blue Gene/Q, this output scheme is not feasible at scale. To solve these issues, we added MPI-IO support in EpiSimdemics.

Benefits: Included in the MPI standard, MPI-IO defines an API for parallel I/O. Most vendors provide a high-performance implementation of MPI-IO, making it a portable solution expected to deliver good performance on high-end parallel computers. Scalable performance of MPI collectives helps improve the performance of these implementations. The use of MPI collectives also helps in performing efficient global communication required for orchestrating writes to the same file in EpiSimdemics.

Resource Sharing: Input in EpiSimdemics is read once by

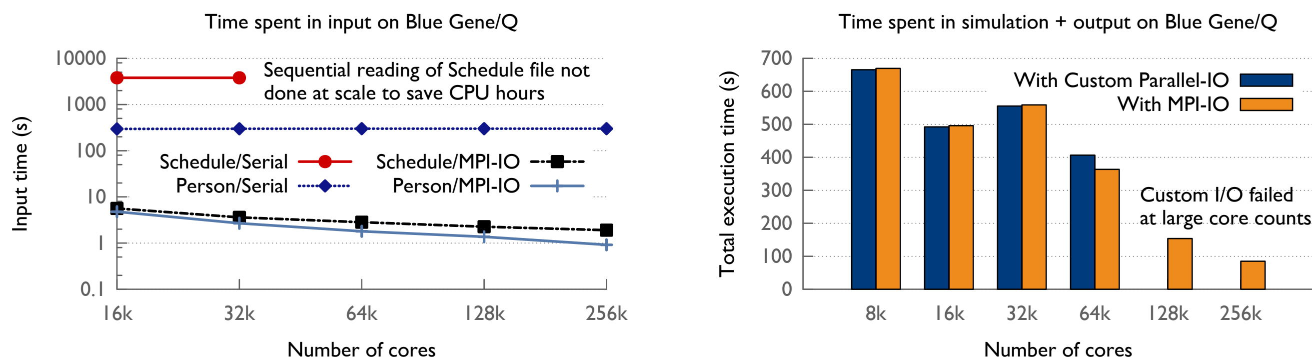

Fig. 7. EpiSimdemics using MPI-IO: Reduction in time for input (left); Use of the MPI-IO library enables faster execution with writes to a single file (right). At scale, the Custom I/O runs out of file descriptors as each node writes to individual files.

all processes at program startup. Output is produced by processes in every iteration, hence periodic flushing is required. This suggests a *hybrid division* of resources to interoperate EpiSimdemics with MPI-IO. A set of processes switch from Charm++ to MPI, once at program startup to input data and periodically for output. The rank mapping is *many-to-one* as Charm++ uses threads for optimal performance.

Data Sharing: Data is transferred between Charm++ and MPI during input and output through a data transfer repository. When the input data is read by MPI tasks, it is deposited locally for retrieval by the corresponding Charm++ tasks. For the output, data is first buffered on some processes that maintain the data repository. Every few iterations, these processes transfer control to MPI, retrieve the data and perform a collective write to a single file.

The productivity benefits of using MPI-IO are obvious – reimplementation of a parallel I/O library is avoided, and all output data is obtained as a single file which eliminates the post processing step. Figure 7 compares the performance obtained using MPI-IO and EpiSimdemics' default schemes. Figure 7 (left) shows that the total input time is reduced significantly from $4,086.56$ seconds to 17.34 seconds using MPI-IO. On 262,144 cores, the time spent in the input phase is only 4.77 seconds.

Figure 7 (right) compares the sum of the simulation time and output time when using MPI-IO with EpiSimdemics' custom scheme that outputs to multiple files. Note that this time does not include the time spent in reading input files. At small scales, the performance of the two versions is similar. At 65,536 cores, use of MPI-IO improves the performance of the application by 10% in comparison to the custom scheme. Beyond this, use of the custom parallel-IO scheme is not feasible given the restriction on the number of file descriptors per job. Use of MPI-IO enables us to execute the application at very large scales with output being obtained as desired.

C. NAMD (Charm++) and Parallel FFTW (MPI)

NAMD [8] is a parallel molecular dynamics code designed for high-performance simulations of large biomolecular systems. NAMD uses a fast Fourier transform (FFT) calculation over a charge grid to approximate long-range force calculations. Through this example, we demonstrate the replacement of a custom implementation of parallel 3D FFT in NAMD with a standard parallel library.

Benefits: Many parallel FFT libraries written in MPI exist, e.g. FFTW and ESSL. It is desirable from a productivity standpoint that NAMD should use one of these libraries, and thus benefit from reduced workload in code development and maintenance. Moreover, vendors often provide highly optimized implementations of FFT algorithms. Use of these vendor-provided versions may also improve performance.

Resource Sharing: During one iteration of NAMD, short-range forces and long-range forces can be computed in parallel. A *space division* of the resources enables the progress of both modules in parallel. Hence, the Charm++ tasks calculate the short-range forces while the MPI tasks perform a parallel FFT for long-range forces. As stated earlier, the rank mapping for space-division is *one-to-none*.

Data Sharing: Data is communicated using a data transfer repository. The Charm++ objects that produce the charge grid deposit their data with the repository. On receiving the data, the repository triggers the execution of parallel FFT in MPI.

The changes required to replace NAMD's FFT code with a parallel FFTW call are minimal and similar to the changes made in the CHARM/HistSort example (§VII-A). Replacing the parallel FFT code in NAMD reduces the source lines of code (SLOC) by 280. More importantly, use of a well-known, actively-developed, third-party library relieves the NAMD developers from the additional task of maintaining the FFT library. It also ensures that any improvements made to FFTW (or an alternative FFT library) will be available to NAMD without any extra effort.

Figure 8 presents the time step comparison between NAMD using its highly optimized FFT implementation and that using parallel FFTW. These runs on Blue Gene/Q use the ApoA1 dataset. It can be seen that the two versions of NAMD have similar performance. Thus, the use of a generic FFT library in NAMD provides similar performance, but leads to the productivity benefits listed above.

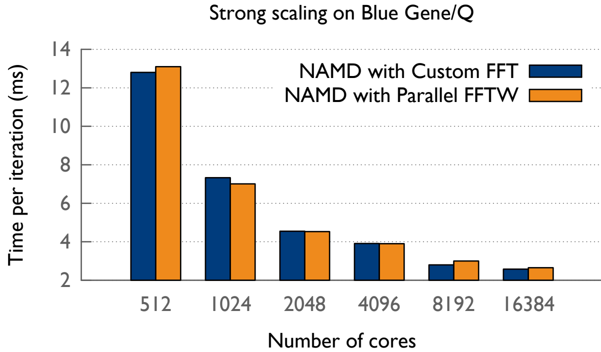

Fig. 8. NAMD's performance on Blue Gene/Q for the ApoA1 benchmark with Custom FFT and Parallel FFTW.

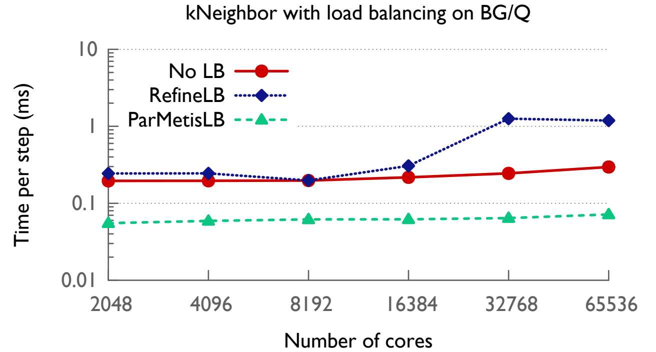

Fig. 9. Load balancing Charm++ applications using ParMETIS: kNeighbor is communication intensive, and benefits from a graph partitioning.

D. Charm++ Applications and ParMETIS (MPI)

Our final example demonstrates the use of ParMETIS [10] to enable parallel graph partitioning in the automatic load balancing framework of Charm++. Measurement-based strategies in Charm++ instrument the application (computational load and communication graph) for a brief period of time, and use the instrumented data to redistribute the objects to balance the load. In addition to the built-in strategies, end-users can easily integrate new strategies specific to their applications. This provides a great opportunity for using external MPI libraries for load balancing if interoperation is possible. ParMETIS [10], and Trilinos [11] are examples of such libraries.

Benefits: Using state-of-the-art graph partitioners, e.g. ParMETIS, enables Charm++ to perform fast communication-aware load balancing. By extension, interoperation can also enable load balancing of application defined tasks in MPI.

Resource Sharing: Most iterative Charm++ applications perform periodic load redistribution, which results in a barrier-style load balancing. This makes it an ideal candidate to use *time division* of resources between Charm++ applications and the MPI-based ParMETIS library. Rank mapping is either *one-to-one* or *many-to-one* based on Charm++'s use of threads.

Data Sharing: Data is shared between the load balancing framework in Charm++ and ParMETIS through pointers when calls are made to the ParMETIS library. This is feasible since the load balancing database is stored in a distributed manner.

We use kNeighbor, a communication-intensive Charm++ benchmark, to demonstrate the benefits of using ParMETIS for load balancing. In kNeighbor, each object exchanges 256 KB messages with 14 other objects in every iteration. These objects also have varying computational loads.

Figure 9 presents the performance improvement in the time per step of kNeighbor from using ParMetisLB, a ParMETIS-based load balancer in Charm++. The time per step is reduced to one-third or one-fourth (66%-75% improvement) of the time per step obtained when no load balancing is performed. ParMetisLB does much better than RefineLB, an existing strategy in Charm++ that aims at balancing computational load only. The time spent in load balancing is similar for both ParMetisLB and RefineLB.

VIII. Lessons Learned

Our experience with hybrid programming for various production applications in the previous section has helped us formulate some basic guidelines for selecting the right technique for sharing resources and data in various scenarios. When deciding on the best strategy to share resources, it is important to understand various phases and modules in an application. If the modules that need to be implemented in different languages are clearly demarcated by phases in time, then time division of the resources is advisable. This is often the case when there is an input-output sort of data dependency across different phases. When the application has modules that can proceed in parallel, a space division of resources can help overlap the progress in these modules. In any other situation, hybrid division of resources has to be performed.

The sharing of data depends on how interoperating modules are implemented. If the data to be exchanged between the modules is already local to each process where it is required, then pointer-based sharing is straightforward. In most other cases, the user has to develop a scheme or set up a data transfer repository for exchange of data across modules.

In this paper, we presented an easy-to-use, scalable method to enable hybrid programming between Charm++ and MPI. For the framework presented, we implemented multiple schemes for managing important attributes of programs running in an interoperation environment. We demonstrated the productivity and performance benefits of interoperation using production applications and libraries implemented in Charm++ and MPI on IBM Blue Gene/Q and Cray XE6 systems. Table II summarizes our findings on productivity and performance benefits. It is evident that enabling interoperation can bring the best of both worlds together for achieving good performance, high programmer productivity, and code reuse.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-663041). This work was partly funded by the National Science Foundation under grant 1339715 and the Lawrence Berkeley National Laboratory under contract 7089247.

TABLE II
PRODUCTIVITY AND PERFORMANCE BENEFITS FOR THE APPLICATION STUDIES PRESENTED IN THIS PAPER.

Application	Library	Productivity	Performance
CHARM	HistSort	Efficient sorting requires support for asynchronous and unexpected messages – a feature provided by Charm++; Reuse of Charm++'s HistSort.	48x speed up in sorting; Removes scaling bottleneck.
EpiSimdemics	MPI-IO	EpiSimdemics I/O is a synchronous operation that can be implemented efficiently using MPI collectives; Enabled organized output to a single file (avoids post processing); Reuse of a standard library, MPI-IO, implemented by vendors.	256x input speed up; Enables output at scale.
NAMD	FFTW	Offloads development of the critical FFT component to experts; Reuse of FFTW library.	Similar performance.
kNeighbor	ParMETIS	Enables parallel graph partitioning based load balancing in Charm++; Reuse of ParMETIS.	Better time per step for applications: 66-75% better for kNeighbor.

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. This research also used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science & Engineering*, vol. 5, no. 1, January-March 1998.

[2] "MPI: A Message Passing Interface Standard," in *MPI Forum*, http://www.mpi-forum.org/.

[3] H. Brunst and B. Mohr, "Performance Analysis of Large-scale OpenMP and Hybrid mpi/openmp Applications with VampirNG," in *Proceedings of the International Workshop on OpenMP (IWOMP)*, Eugene, OR, June 2005.

[4] J. Corbalan, A. Duran, and J. Labarta, "Dynamic load balancing of MPI+OpenMP applications," *icpp*, vol. 00, pp. 195–202, 2004.

[5] L. Kale, A. Arya, N. Jain, A. Langer, J. Lifflander, H. Menon, X. Ni, Y. Sun, E. Totoni, R. Venkataraman, and L. Wesolowski, "Migratable objects + active messages + adaptive runtime = productivity + performance a submission to 2012 HPC class II challenge," Parallel Programming Laboratory, Tech. Rep. 12-47, November 2012.

[6] F. Miniati and P. Colella, "Block structured adaptive mesh and time refinement for hybrid, hyperbolic+n-body systems," *J. Comput. Phys.*, vol. 227, no. 1, pp. 400–430, Nov. 2007.

[7] 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*, ser. IPDPS '14. IEEE Computer Society, May 2014.

[8] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale, "Overcoming scaling challenges in biomolecular simulations across multiple platforms," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.

[9] M. Frigo and S. Johnson, "FFTW: an adaptive software architecture for the FFT," *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3, pp. 1381–1384 vol.3, May 1998.

[10] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning scheme for irregular graphs," in *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, 1996, p. 35.

[11] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the Trilinos project," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, 2005.

[12] X. Zhao, D. Buntinas, J. A. Zounmevo, J. Dinan, D. Goodell, P. Balaji, R. Thakur, A. Afsahi, and W. Gropp, "Toward asynchronous and MPI-interoperable active messages," in *CCGRID*, 2013, pp. 87–94.

[13] V. S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice & Experience*, vol. 2, 4, pp. 315–339, December 1990.

[14] G. Edjlali, A. Sussman, and J. Saltz, "Interoperability of data parallel runtime libraries with Meta-Chaos," in *In Proceedings of the Eleventh International Parallel Processing Symposium. IEEE Computer*. Society Press, 1997.

[15] L. V. Kale, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon, "Converse: An Interoperable Framework for Parallel Programming," in *Proceedings of the 10th International Parallel Processing Symposium*, April 1996, pp. 212–217.

[16] L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng, "Programming Petascale Applications with Charm++ and AMPI," in *Petascale Computing: Algorithms and Applications*, D. Bader, Ed. Chapman & Hall / CRC Press, 2008, pp. 421–441.

[17] A. Buss, T. Smith, G. Tanase, N. Thomas, M. Bianco, N. Amato, and L. Rauchwerger, "Design for interoperability in stapl: pmatrices and linear algebra algorithms," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, J. Amaral, Ed. Springer Berlin Heidelberg, 2008, vol. 5335, pp. 304–315. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89740-8_21

[18] E. Lusk and A. Chan, "Early experiments with the OpenMP/MPI hybrid programming model," in *Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, ser. IWOMP'08, 2008, pp. 36–47.

[19] G. Tang, E. F. D'Azevedo, F. Zhang, J. C. Parker, D. B. Watson, and P. M. Jardine, "Application of a hybrid mpi/openmp approach for parallel groundwater model calibration using multi-core computers," *Comput. Geosci.*, vol. 36, pp. 1451–1460, November 2010. [Online]. Available: http://dx.doi.org/10.1016/j.cageo.2010.04.013

[20] T. S. Tarek El-Ghazawi, William Carlson and K. Yelick, *UPC: Distributed Shared Memory Programming*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2005.

[21] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur, "Hybrid parallel programming with MPI and Unified Parallel C," in *Proceedings of the 7th ACM International Conference on Computing Frontiers*, ser. CF '10, 2010, pp. 177–186.

[22] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur, "Enabling MPI interoperability through flexible communication endpoints," in *EuroMPI 2013*, Madrid, Spain, 2013.

[23] E. Solomonik and L. V. Kale, "Highly Scalable Parallel Sorting," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010.

[24] "Chombo – infrastructure for adaptive mesh refinement," http://seesar.lbl.gov/anag/chombo/. [Online]. Available: http://seesar.lbl.gov/anag/chombo