On Interoperation among User-driven and System-driven Parallel Languages

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
‡Department of Computer Science, Virginia Tech, Blacksburg, Virginia 24061 USA
§Lawrence Berkeley National Laboratory, Berkeley, California 94720 USA
¶Institute for Astronomy, ETH Zurich, Zurich, Switzerland
‖Google Inc., Mountain View, California 94043 USA

***Abstract*—Modern parallel codes are often written as a collection of several diverse modules. Different programming languages might be the best or natural fit for each of these modules or for different libraries that are used together in an application. For such applications, the restriction of implementing the entire application in a single parallel language may impact the application's performance and programmer's productivity negatively.**

This paper studies interoperation among parallel languages that differ with respect to the driver of program execution. We describe the challenges in enabling interoperation among user-driven and system-driven languages, and present techniques for managing important attributes of a program, such as the control flow, resource sharing, and data sharing, in an interoperable environment. We also present a generalized framework that enables interoperation between two production languages, MPI and Charm++. Finally, we study the application of the presented techniques and demonstrate the benefits of interoperation through several case studies using production codes including CHARM, EpiSimdemics, NAMD, FFTW, MPI-IO and ParMETIS, executed on IBM Blue Gene/Q and Cray XE6.

I. Introduction

The increasing computational power of supercomputers, as attested by the existence of tens of machines with Petaflop/s performance, is expected to lead to 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 manage the complexity of parallel simulations. In particular, it should include multiple programming languages. For simplicity, we use the term *language* to refer to all the entities that provide a mechanism for writing a parallel program — a communication library, a runtime system, a programming model or a compiler supported parallel language. Different languages provide various *features* that are instrumental in designing and implementing parallel applications in them. Most of the existing parallel applications are implemented in a single language. Hence, they are limited in their ability to exploit features provided by other languages. As modern parallel codes get more complex and are implemented as a collection of several diverse modules, features from different programming languages might be the best or natural fit for each of these modules. As a result, the restriction of using only one language may impact the application's performance and programmer's productivity negatively. Moreover, it is crucial to be able to reuse parallel software developed in different languages since it is complex and expensive to develop.

Effectively developing a rich, interoperable toolbox that allows for seamless mixing of different parallel languages is fraught with challenges. In this paper, we describe and address the challenges in enabling interoperation among languages that differ with respect to the driver of program execution — user-driven languages, e.g. MPI, where the programmer explicitly defines the control flow, and system-driven languages, e.g. Charm++, where a runtime system drives the execution based on availability of data (described in §II). The most critical component of enabling interoperation among these languages is the transfer and management of control flow between the languages. We present an easy-to-use method that can be deployed for any pair of user-driven and system-driven languages (§III).

In an interoperable environment, presence of multiple languages requires sharing of resources, such as cores, network, etc. and information, such as application data. Depending on the application at hand and the physical system being used, the best method to enable such sharing may also vary. In §V & §VI, we describe various techniques for resource and data sharing, and later summarize our experience on their suitability to various scenarios (§VIII).

For wide-spread acceptability, it is critical that the methods for enabling interoperation are both easy-to-use and scalable. In order to demonstrate these capabilities of the proposed ideas, we have developed a generalized framework that enables interoperation between MPI and Charm++ (§IV). Using this framework, we study the application of the proposed methods and demonstrate the benefits of interoperation using production parallel codes — CHARM [1], EpiSimdemics [2], and NAMD [3], and libraries including FFTW [4], MPI-IO, and ParMETIS [5] — executed on thousands of cores of IBM Blue Gene/Q and Cray XE6 (§VII, summarized in Table II). These examples establish the utility of interoperation

in eliminating performance bottlenecks in the applications with minimal effort. At the same time, they demonstrate how interoperation leads to code reuse and eases programmers' burden by allowing them to use features that match the requirements of the individual application modules.

II. Background

We divide languages into two sets for the purposes of this work: *user-driven* and *system-driven*. A language is considered user-driven if the program control flow is explicitly defined by the programmer. Data exchanges among processes are predetermined and the execution is defined as a single flow of control. MPI [6], UPC [7], and High Performance Fortran [8] are examples of user-driven languages; exceptions such as `MPI_ANY_*` exist, wherein the programmer delegates the ordering to the system, but are discouraged.

In a system-driven language, a runtime system (RTS) decides what computation to execute next, typically selecting one from many potential computations. This execution model allows for many concurrent control flows with progress driven by the availability of data (data-driven execution) and system's preferences. Charm++ [9], X10 [10], and HPX [11] are examples of system-driven languages.

In this paper, we focus on MPI and Charm++ as concrete instances of user-driven and system-driven languages respectively. Table I lists some of the features provided by them and their ease-of-use. MPI provides support for 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 [12], [13].

TABLE I. Different features and their relative ease-of-use in the context of MPI and Charm++ as representatives of user-driven and system-driven paradigms respectively.

Language Feature	MPI [6] User-driven	Charm++ [9] System-driven
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

In contrast to MPI, a system-driven language such as 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 express global control flow and difficulty in performing global communication limits its use in various scenarios.

Similar conclusions about the benefits and limitations of other languages can be made. One can imagine a scenario where a perfect language is designed which is both user-driven and system-driven, and supports all the features. However, a more likely end-point is a rich ecosystem of diverse languages whose interoperation is required to create a powerful toolbox for the parallel programmer.

A. Related Work

In parallel computing, interoperation was initially explored by Harper who developed a library that allowed programs written for version 3 of Parallel Virtual Machine (PVM [14]) 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++ [15]. Kale et al. [16] proposed and demonstrated the use of a common runtime framework (Converse) for interoperation of various parallel programming languages such as MPI [6], PVM [14] and Charm++ [9].

More recently, STAPL [17] has provided a methodology that enables third party libraries to be used with it. In terms of languages, hybrid use of MPI [6] and OpenMP [18] has received significant attention and has been widely adopted [19], [20]. Use of MPI in Unified Parallel C [7] 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. [13] 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 flows – 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. In addition, we describe and implement multiple schemes for resource sharing in a generalized framework for MPI and Charm++.

III. Interoperation: The Challenge and Solution

The simultaneous use of multiple languages for writing a parallel program raises several interesting questions about managing different aspects of the program. Among these, *management and transfer of control* between different language modules is critical, even more so when the languages differ with respect to the driver of program execution. In this section, we try to find answers for 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?, and 3) How frequently should the control be transferred?

When interoperating between two languages of the same kind (user-driven or system-driven), transfer of control from one language to another is simple. If the two languages are user-driven, e.g. MPI and UPC, the user explicitly drives the program with the control being returned to him after every system-invocation [21]. For the case of two interoperating system-driven languages, the availability of data drives the modules written in those languages (either the two languages will share the same RTS or the two RTSs will coordinate).

In contrast, if one language is user-driven and the other is system-driven, there is no obvious solution. If the execution begins in the user-driven language module, there exists no

mechanism to progress the system-driven language module because the RTS is typically hidden from the user. Similarly, if the execution begins in the system-driven module, there exists no mechanism to transfer the control to the user-driven module because typically the RTS does not support yielding control to the user.

One possible solution is to avoid the need for transfer of control by using *concurrent flows*. In this method, the user-driven and system-driven language modules are executed in their own *home* threads. 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, the modules that could have used these cycles will have to wait for their turn. 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 production languages such as MPI and Charm++. Finally, the presence of concurrent flows is likely to cause cache pollution and branch mispredictions leading to degraded performance.

In light of the drawbacks of the approach presented above, we propose an alternate solution that executes the different modules in a single control flow. This is made feasible by *exposing the scheduler* of the system-driven language and empowering the user to control it. In this approach, the execution of a program begins in a user-driven language module wherein the semantics of the user-driven language are followed. When required, the exposed scheduler of the system-driven language is activated. From this point, the execution is driven by the RTS following the semantics of the system-driven language. At a later time, the scheduler is explicitly deactivated and the control is returned back to the user-driven module as shown in Figure 1 for MPI and Charm++. Following it, the user-driven module may again activate the scheduler of the system-driven language, and hence repeat the cycle. This approach eliminates the major disadvantages of the former approach — no thread scheduling overheads and minimal busy-waiting if implemented correctly. Although, this approach empowers the users significantly, it also increases their burden by demanding explicit control transfer.

Explicit transfer of control by the user leads to another important question — how frequently should the control be transferred? If an application is written using 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 in a program may have negative impact on productivity and performance. Since the availability of data drives the execution in a system-driven language, the user will have to consider all possible orderings for it to exchange the control with a user-driven language 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 since

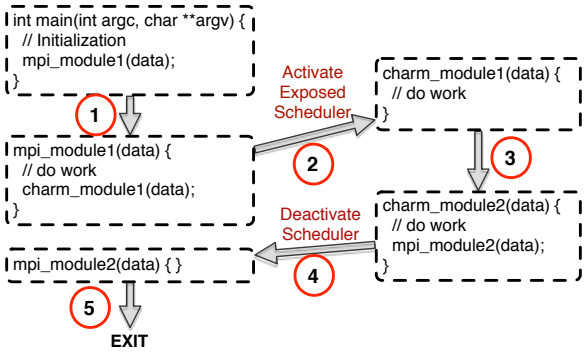

Fig. 1. Transfer of control between MPI and Charm++ using an exposed scheduler

every process is now cycling through the modules in different languages frequently. A mismatch in the modules executed by different processes may block one or the other leading to idle time. Finally, given the differences in semantics of these languages, it may also be difficult to verify the correctness of a program in such scenarios.

The disadvantages of frequent control transfer listed in the previous paragraph suggest that the control is best transferred between a user-driven and a system-driven language module infrequently to maintain ease of use and simplicity. More importantly, coarse grained control transfer allows for reuse of independent modules/libraries as one unit without significant modifications. The major drawback is the inability to make progress simultaneously in different modules.

IV. The Charm++/MPI Interoperation Framework

Based on the discussion in Section III, we have developed a generalized framework that enables interoperation between a user-driven language, MPI, and a system-driven language, Charm++. Enabling interoperation using this framework does not require any changes to the MPI implementation being used. A new API has been added to Charm++ that exposes its scheduler to the user.

A. Enabling Interoperation

In general, in order to prepare a language for interoperation using our framework, a language should implement the following constructs to:

- **Initialize:** given a set of processes, perform setup such as identify rank space, initialize low-level communication substrate, etc. to create a language instance.
- **Execute:** make progress in the given 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.

For MPI, all of these constructs already exist in its 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 it is complete. 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` that transfers the control to the Charm++ RTS. The scheduler can be stopped either on a single processor using `StopCharmScheduler` or collectively on all processes by calling `ckExit`. Finally, the clean up is performed by invoking `CharmLibExit`.

B. Writing Interoperable MPI-Charm++ Programs

For a programmer, interoperation between independent MPI and Charm++ modules requires *minor* modifications to both the modules. Other than including the necessary headers, following is a list of *all the required* additional tasks a module must perform:

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 an interface function callable from Charm++ (a C/C++ function); 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 Charm++ RTS; to transfer control to MPI modules, call interface function provided by the MPI modules.

The code snippet below shows an MPI 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 invokes the Charm++ module and on return, the Charm++ instance is destroyed. If needed, control can be transferred back and forth multiple times between MPI and Charm++ modules before the instance is destroyed.

```
#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);
    HiStart(16); // Call Charm++ library
    CharmLibExit(); // Destroy Charm++ instance
  } else {
    // MPI work on rest of the processes
  }
  MPI_Finalize();
}
```

A standalone Charm++ program begins execution in the constructor of a special C++ object called *mainchare* and exits the program by calling `ckExit`. To enable interoperation, we have modified this aspect of Charm++. When using a Charm++ module for interoperation, execution in Charm++ begins only when it is invoked explicitly by initiating a message to one of its objects and starting the Charm++ RTS using `StartCharmScheduler`. In the code snippet below, `HiStart` is an interface function that performs these tasks. On processor 0, a message is initiated to the `mainHi` object after which all processes activate the Charm++ RTS. In this simple example, when the RTS receives this message and schedules it, calling `ckExit` collectively stops the scheduler on all processes, thus returning the control to the interface function.

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

// function invoked from MPI
// marks the begining of Charm++
void HiStart(int elems) {
  if(CkMyPe() == 0) {
    mainHi.StartHi(elems);
  }
  StartCharmScheduler();
}

// Charm++ function that deactivates scheduler
void MainHi::StartHi(int elems) {
  ckExit();
}
```

V. Sharing Resources and Data

In an interoperable environment, the presence of different modules requires explicit coordination of certain aspects that are otherwise handled by the language implementations. We focus on two such important issues – 1) How are resources shared?, and 2) How is data shared? between modules written in different languages.

A. Resource Sharing

Execution of multiple modules written in different languages 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 a framework based on the preferences expressed in the application. Figure 2 presents three schemes provided in our framework for sharing resources — time division, space division and hybrid division.

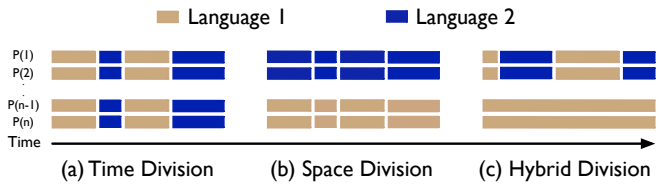

Fig. 2. Schemes for sharing resources between languages in an interoperable environment.

Time Division: During the execution of an application on a system, if all the processes switch from one language module to another synchronously, we refer to this way of interoperation as time division. As depicted in Figure 2(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. Such an exchange may happen multiple times before the application exits. This method of interoperation is useful for applications that have an ordering among the tasks to be executed in different language modules.

Space Division: Instead of time slicing the resources, if subsets of processes are assigned to different languages for the entire duration of program execution, it is referred to as a space-based division of resources. Figure 2(b) shows this scenario in which modules written in one language run on some of the processes, while modules written in other languages run on the rest. Space division is useful for making simultaneous progress in modules that are loosely connected to one another.

Hybrid Division: Combination of time division and space division provides a hybrid method of resource sharing. In this scheme, a subset of processes execute modules written in different languages during an execution. Different subsets may execute different modules independently of other subsets. For example, in Figure 2(c), a subset of processes transfer control among modules written in two languages, while another subset executes modules written in only one language. 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 among the languages.

B. Data Sharing

Modules implemented in different programming languages may 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 methods are supported for exchanging data among different modules in the presented framework.

Pointer-based Data Sharing: This method is based on exchanging data by explicitly passing memory pointers. Let us consider two processes, *P1* and *P2*, executing two modules, *A* and *B*, written in different languages (Figure 3). If data is to be transferred between modules within a process, say from *P1-A* to *P1-B*, it can be exchanged via use of reserved memory space. *P1-A* copies the data to a predefined memory space, and thereafter *P1-B* accesses it.

This approach can be extended to communicate data to a different module on a different process in two steps, say from *P1-A* to *P2-B*. Using *Path 1*, *P1-A* on *P1* first communicates data to *P2-A* on *P2*, and then to destination *P2-B* — transfer across processes first and then to the destination module. Alternatively, using *Path 2*, data can be sent to module *P1-B* on the source process first, and then transferred to the destination process, *P2*.

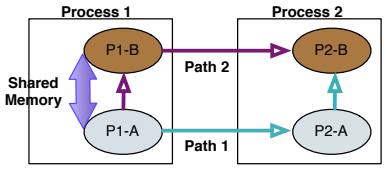

Fig. 3. Pointer-based Data Sharing.

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 various languages (a pull model). Under the hood, the data transfer repository communicates with its counterparts on other processes to service the requests.

Development 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 [7]).

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 interoperation between user-driven and system-driven languages. 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 presence 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 Details

In this section, we provide details of how low-level resources are used and shared between different modules in the Charm++/MPI 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 and uGNI on IBM Blue Gene/Q and Cray XE6 respectively. The presence of multiple modules requires that the communication started by any one of them be delivered

to the intended receiver module at the destination. We use distinct communication domains for each module in the low-level API to ensure this property.

For uGNI, a domain is created by `GNI_CdmCreate`, which enables interoperation on Cray machines such as Cray XE6 (Blue Waters, Titan) 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 it.

An interesting alternative, which has also been implemented, is to use MPI as the communication substrate for Charm++ modules. It enables interoperation between MPI and Charm++ 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 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 data repository for exchange of data has been implemented as a C++ module, which uses Charm++ in the background 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 being used, the data repository stores data of one type or the other. Work on fully generalizing the data repository (using templates and related concepts) is under progress.

D. Multi-threading

Unlike MPI, Charm++ can also be built in a 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 the memory and a communication thread. In MPI, similar shared memory optimizations are typically enabled via use of OpenMP [18]. 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 benefits of multi-language interoperation in the context of production parallel codes. This section examines the interoperation of MPI, Charm++, and OpenMP using many production codes, and demonstrates the productivity and performance benefits derived from their synergistic existence.

A. CHARM and HistSort

Our first example demonstrates the use of a parallel sorting library, HistSort [23], written in Charm++, in a production cosmological and astrophysical code called CHARM [1] (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, 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. Hence, this global sort of particles is a critical component of this algorithm but has been a scalability bottleneck in its current implementation.

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]. The HistSort library in Charm++ overlaps the search for global splitters with the sorting of input data local to each process. The splitters are based on a global histogram and are used to determine the final destination of the input data. As the splitters are determined, each process sends the sorted local data to their respective destinations asynchronously, thereby exploiting Charm++'s ability to handle unexpected messages.

Resource Sharing: The global sorting in CHARM needs to be performed in every iteration before the computation 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 4 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 4 (right) performs the actions described in Section IV-B — initiate a message to the main object and activate Charm++ RTS. CHARM uses HistSort by invoking

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

CHARM code flow with Multi-way Merge Sort
-----------------------------------------------
/* 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*/
...

CHARM code flow 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. 4. (left) Modifications required to transfer control from CHARM to Charm++'s HistSort; (right) The interface function for HistSort library that can be called from any MPI program.

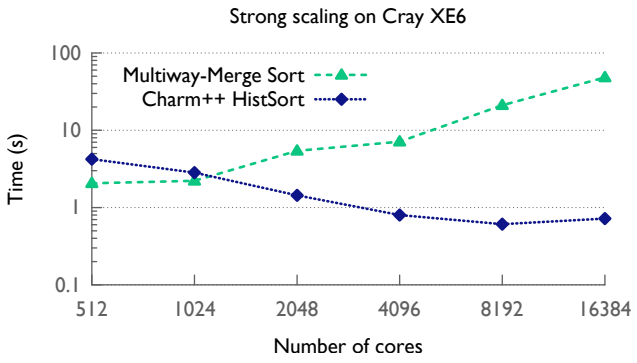

Fig. 5. CHARM using Charm++'s histogram sorting: scaling bottleneck caused due to sorting can be resolved using Charm++'s sorting library.

the interface function instead of the default Multiway-merge Sort implementation as shown in Figure 4 (left).

Figure 5 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 the source lines of code (SLOC) by 195.

B. EpiSimdemics and MPI-IO

This second case study shows the coupling of the MPI-IO library [6] with a contagion simulation code called EpiSimdemics [2], 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 bottleneck 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 types of input files. The *person* file and the *location* file contain the attributes of each person and each location respectively. The *schedule* file contains a list of edges, where an edge represents the *visit* of a person to a location. 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. The *disease* file records the time of every health state transition for the people. The *dendogram* file records the information related to every disease transmission event. For our simulation setup, the former is 7.7 GB and the latter is 5.5 GB in binary format. 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. It impacts the performance in two ways - time taken to read the data on one process and time spent in a scatter operation from that process to distribute data among all processes. 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 improves the performance but requires post-processing of data before it is 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. A possible solution is to implement an ad-hoc scheme to collect data on a limited number of processes, and perform writing from these designated processes. Instead, we propose the use of MPI-IO enabled by the interoperation of Charm++ with MPI.

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

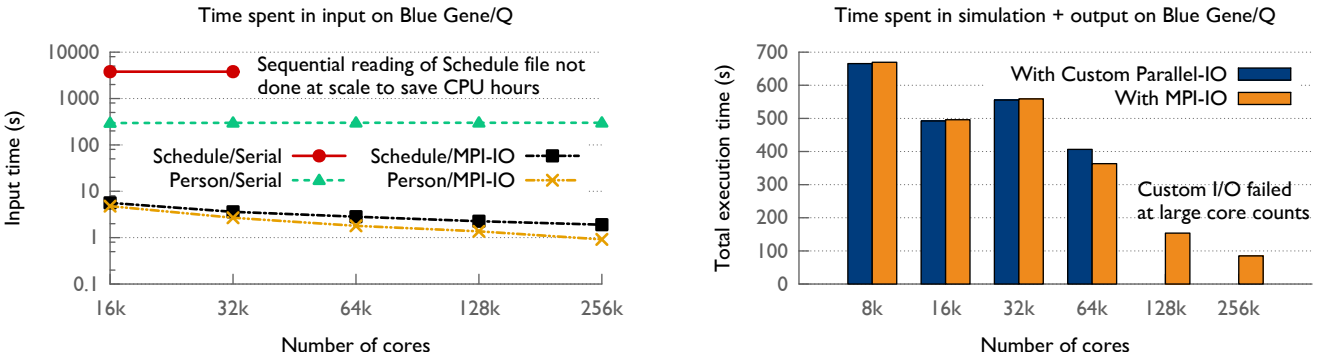

Fig. 6. EpiSimdemics using MPI-IO: use of MPI-IO library enables faster execution with writes to a single file. At scale, the Custom I/O runs out of file descriptors as each node writes to individual files.

improve 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 all processes at program startup. The output is produced incrementally 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 perform input and periodically for output. The rank mapping is *many-to-one* as Charm++ uses threads for optimal performance.

Data Sharing: Data is transferred between MPI and Charm++ 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, the control is transferred on these processes to MPI, which 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. In Figure 6, we compare the performance obtained using MPI-IO and EpiSimdemics' default schemes. Figure 6 (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. The location file follows similar trends as the person file.

Figure 6 (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. However, the use of MPI-IO enables us to execute the application at very large scales with output being obtained as desired.

C. NAMD and Parallel FFTW

NAMD [3] is a parallel molecular dynamics code, written in Charm++, and 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. This three-dimensional transform is broken down into one-dimensional FFTs (that use serial FFTW [4]) with transposes in between. Through this example, we demonstrate the replacement of a custom implementation of a parallel 3D FFT in NAMD with a standard parallel library.

Features: Many parallel FFT libraries exist, e.g. FFTW and ESSL; most of them are written using MPI. It is desirable from a productivity standpoint that NAMD utilizes 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 versions, provided by the vendors, 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 can enable 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 dedicated data transfer repository. The Charm++ tasks that produce the charge grid deposit their data with this repository and on receiving the data, the repository triggers the execution of a parallel FFT in MPI.

The changes required to replace NAMD's FFT code with 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 any other FFT library that can be used instead of FFTW) will be available to NAMD without any extra effort.

Figure 7 presents the time step comparison between NAMD

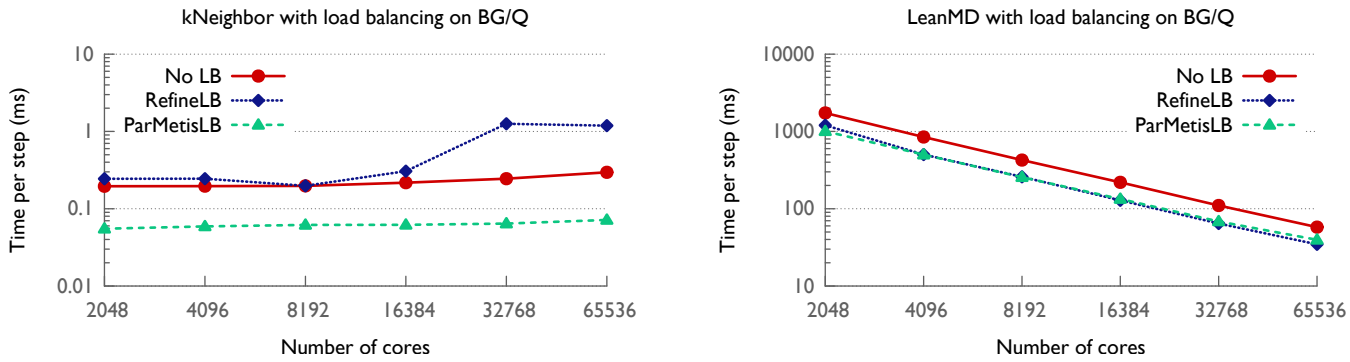

Fig. 8. Load balancing Charm++ applications using ParMETIS: kNeighbor is communication intensive, and benefits significantly from a global graph partitioning enabled by use of ParMETIS. LeanMD is compute intensive and uses OpenMP to further parallelize its computation.

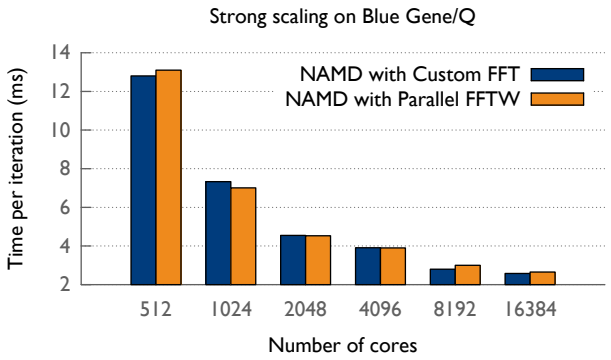

Fig. 7. NAMD with Parallel FFTW

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.

D. Charm++ codes and ParMETIS and OpenMP

Our final example demonstrates the use of ParMETIS [5] 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 [5], and Trilinos [12] are examples of such libraries.

Resource Sharing: Most iterative Charm++ applications use periodic load balancing. In this mode, all objects in the application invoke the load balancer periodically (every n iterations) where by the control is transferred to the load balancing framework. Once load balancing is completed, the objects resume execution. This barrier-style load balancing makes it an ideal candidate to use the *time division* of resources between Charm++ applications and the MPI-based ParMETIS library. Rank mapping is either *one-to-one* or *many-to-one* depending 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 Charm++ stores the load balancing database in a distributed manner.

We use two Charm++ programs, LeanMD and kNeighbor to demonstrate the performance benefits of using ParMETIS. LeanMD is a proxy application for NAMD that calculates the Lennard-Jones potential for a molecular system. kNeighbor is a communication-intensive benchmark where each Charm++ object exchanges 256 KB messages with 14 other objects in each iteration. Imbalanced computational load is also associated with these programs.

Figure 8 (left) 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.

In contrast to kNeighbor, LeanMD is a computation-intensive benchmark. It uses OpenMP to parallelize parts of code that perform force calculations among particles. Hence, minimizing the edge cut in a partitioned communication graph is not optimal for this benchmark. In this case, we do not provide edge weights to ParMetisLB, and hence the graph partitioner tries to balance the vertex weights among partitions (processes). Figure 8 (right) shows the performance benefits of using ParMetisLB. Use of ParMETIS for balancing load reduces the time per step by 30% to 40%. Charm++'s computation aware strategy, RefineLB, also obtains similar performance.

VIII. Lessons Learned

Our experience with interoperation for various production applications in the previous section has helped us formulate some basic guidelines for selecting the right technique for

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 LeanMD	ParMETIS	Enables parallel graph partitioning based load balancing in Charm++; Reuse of ParMETIS.	Better time per step for applications: 30-40% better for LeanMD; 66-75% better for kNeighbor.

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 programming 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, either time or space division can be used in different phases of the application, referred to as a hybrid division of resources.

The sharing of data depends on how various modules that have to interoperate have been 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.

A. Conclusion

The use of multiple programming languages is necessary to match the complexity of modern applications that consist of diverse software modules. Each module could be written in a programming language that is most suited to the feature requirements of that module, independent of the language choices for other modules. Moreover, code reuse should not be limited by the choice of the implementation language.

In this paper, we have presented an easy to use scalable method to enable interoperation among user-driven and system-driven languages. For a generalized framework, we have proposed and implemented multiple schemes for managing important attributes of programs in an interoperation environment. Productivity and performance benefits of interoperation using production applications and libraries implemented in MPI and Charm++ have been demonstrated 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 different worlds together for achieving good performance and high programmer productivity.

REFERENCES

[1] 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.

[2] 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 (to appear)*, ser. IPDPS '14. IEEE Computer Society, May 2014.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel, *The High Performance Fortran Handbook*. MIT Press, 1994.

[9] 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.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA*. New York, NY, USA: ACM, 2005, pp. 519–538.

[11] H. Kaiser, M. Brodowicz, and T. Sterling, "Parallex an advanced parallel execution model for scaling-impaired applications," in *ICPPW '09: Proceedings of the 2009 International Conference on Parallel Processing Workshops*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 394–401.

[12] 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.

[13] 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.

[14] V. S. Sunderam, "PVM: A framework for parallel distributed com-

puting," *Concurrency: Practice & Experience*, vol. 2, 4, pp. 315–339, December 1990.

[15] 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.

[16] 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.

[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] 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.

[19] 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.

[20] 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

[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