Interoperating MPI and Charm++ for Productivity and Performance

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

I. Overview

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. Effectively developing a rich, interoperable toolbox that allows for seamless mixing of different parallel languages is fraught with challenges. In this poster, we describe and address the challenges in enabling interoperation among languages that differ with respect to the driver of program execution — MPI [1], where the programmer explicitly defines the control flow, and Charm++ [2], where a runtime system drives the execution based on availability of data.

Control Flow: The first challenge is the ability to transfer control flow between MPI and Charm++. Currently, if the execution begins in MPI, there exists no mechanism to progress Charm++ because its runtime system (RTS) is typically hidden from the user. Similarly, if the execution begins in Charm++, there exists no mechanism to transfer the control to MPI because typically the RTS does not support yielding control to the user. We tackle this challenge by *exposing the scheduler* of the Charm++ RTS and empowering the user to control it. In this approach, the execution of a program begins in MPI wherein the semantics of MPI are followed. When required, the exposed scheduler of Charm++ is activated. From this point, the execution is driven by the RTS following the semantics of Charm++. At a later time, the scheduler is explicitly deactivated and the control is returned back to the MPI as shown in Figure 1 in the poster.

Resource Sharing: The next challenge is that of sharing resources among the two languages. Figure 2 in the poster presents three schemes provided in our framework for sharing resources — time division, space division and hybrid division.

In **Time Division**, during the execution of an application on a system, all the processes switch from one language to another synchronously. This method of interoperation is useful for applications that have an ordering among the tasks to be executed in different language modules.

In **Space Division**, instead of time slicing the resources, subsets of processes are assigned to different languages for the entire duration of program execution. Space division is useful for making simultaneous progress in modules that are loosely connected to one another.

Contact author: nikhil.jain@acm.org

In **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. A hybrid model of interoperation can be particularly useful in applications that require different subsets to perform different tasks during application execution.

Data Sharing: The following simple methods are supported for exchanging data among different languages in the presented framework.

Pointer-based Data Sharing: This method is based on exchanging data by explicitly passing memory pointers. 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. For inter-process communication, the data is first transferred within the source language to the destination, and then exchanged via use of reserved memory (Figure 3).

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.

II. 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

TABLE I. PRODUCTIVITY AND PERFORMANCE BENEFITS FOR THE APPLICATION STUDIES PRESENTED IN THIS POSTER.

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.

to MPI modules, call interface function provided by the MPI modules.

The code snippet in the poster 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.

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 in the poster, `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.

III. APPLICATION STUDIES

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++. Using this framework, we study the application of the proposed methods and demonstrate the benefits of interoperation using production parallel codes — CHARM [3], EpiSimdemics [4], and NAMD [5], and libraries including FFTW [6], MPI-IO, and ParMETIS [7] — executed on thousands of cores of IBM Blue Gene/Q and Cray XE6 (summarized in Table I). 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.

ACKNOWLEDGEMENT

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-ABS-662816).

REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.