Parallel Programming Laboratory
University of Illinois at Urbana-Champaign

Adaptive MPI Manual

Version 1.0

University of Illinois
Charm++/Converse Parallel Programming System Software
Non-Exclusive, Non-Commercial Use License

Upon execution of this Agreement by the party identified below ("Licensee"), The Board of Trustees of the University of Illinois ("Illinois"), on behalf of The Parallel Programming Laboratory ("PPL") in the Department of Computer Science, will provide the Charm++/Converse Parallel Programming System software ("Charm++") in Binary Code and/or Source Code form ("Software") to Licensee, subject to the following terms and conditions. For purposes of this Agreement, Binary Code is the compiled code, which is ready to run on Licensee's computer. Source code consists of a set of files which contain the actual program commands that are compiled to form the Binary Code.

1. The Software is intellectual property owned by Illinois, and all right, title and interest, including copyright, remain with Illinois. Illinois grants, and Licensee hereby accepts, a restricted, non-exclusive, non-transferable license to use the Software for academic, research and internal business purposes only, e.g. not for commercial use (see Clause 7 below), without a fee.

2. Licensee may, at its own expense, create and freely distribute complimentary works that interoperate with the Software, directing others to the PPL server (`http://charm.cs.illinois.edu`) to license and obtain the Software itself. Licensee may, at its own expense, modify the Software to make derivative works. Except as explicitly provided below, this License shall apply to any derivative work as it does to the original Software distributed by Illinois. Any derivative work should be clearly marked and renamed to notify users that it is a modified version and not the original Software distributed by Illinois. Licensee agrees to reproduce the copyright notice and other proprietary markings on any derivative work and to include in the documentation of such work the acknowledgement:

   > "This software includes code developed by the Parallel Programming Laboratory in the Department of Computer Science at the University of Illinois at Urbana-Champaign."

   Licensee may redistribute without restriction works with up to 1/2 of their non-comment source code derived from at most 1/10 of the non-comment source code developed by Illinois and contained in the Software, provided that the above directions for notice and acknowledgement are observed. Any other distribution of the Software or any derivative work requires a separate license with Illinois. Licensee may contact Illinois (`kale@illinois.edu`) to negotiate an appropriate license for such distribution.

3. Except as expressly set forth in this Agreement, THIS SOFTWARE IS PROVIDED "AS IS" AND ILLINOIS MAKES NO REPRESENTATIONS AND EXTENDS NO WARRANTIES OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO WARRANTIES OR MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY PATENT, TRADEMARK, OR OTHER RIGHTS. LICENSEE ASSUMES THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE AND/OR ASSOCIATED MATERIALS. LICENSEE AGREES THAT UNIVERSITY SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, OR INCIDENTAL DAMAGES WITH RESPECT TO ANY CLAIM BY LICENSEE OR ANY THIRD PARTY ON ACCOUNT OF OR ARISING FROM THIS AGREEMENT OR USE OF THE SOFTWARE AND/OR ASSOCIATED MATERIALS.

4. Licensee understands the Software is proprietary to Illinois. Licensee agrees to take all reasonable steps to insure that the Software is protected and secured from unauthorized disclosure, use, or release and will treat it with at least the same level of care as Licensee would use to protect and secure its own proprietary computer programs and/or information, but using no less than a reasonable standard of care. Licensee agrees to provide the Software only to any other person or entity who has registered with Illinois. If licensee is not registering as an individual but as an institution or corporation each member of the institution or corporation who has access to or uses Software must agree to and abide by the terms of this license. If Licensee becomes aware of any unauthorized licensing, copying or use of the Software, Licensee shall promptly notify Illinois in writing. Licensee expressly agrees to use the Software only in the manner and for the specific uses authorized in this Agreement.

5. By using or copying this Software, Licensee agrees to abide by the copyright law and all other applicable laws of the U.S. including, but not limited to, export control laws and the terms of this license. Illinois shall have the right to terminate this license immediately by written notice upon Licensee's breach of, or non-compliance with, any terms of the license. Licensee may be held legally responsible for any copyright infringement that is caused or encouraged by its failure to abide by the terms of this license. Upon termination, Licensee agrees to destroy all copies of the Software in its possession and to verify such destruction in writing.

6. The user agrees that any reports or published results obtained with the Software will acknowledge its use by the appropriate citation as follows:

   > "Charm++/Converse was developed by the Parallel Programming Laboratory in the Department of Computer Science at the University of Illinois at Urbana-Champaign."

   Any published work which utilizes Charm++ shall include the following reference:

   > "L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In 'Parallel Programming using C++' (Eds. Gregory V. Wilson and Paul Lu), pp 175-213, MIT Press, 1996."

   Any published work which utilizes Converse shall include the following reference:

   > "L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. Proceedings of the 10th International Parallel Processing Symposium, pp 212-217, April 1996."

   Electronic documents will include a direct link to the official Charm++ page at `http://charm.cs.illinois.edu/`

7. Commercial use of the Software, or derivative works based thereon, REQUIRES A COMMERCIAL LICENSE. Should Licensee wish to make commercial use of the Software, Licensee will contact Illinois (kale@illinois.edu) to negotiate an appropriate license for such use. Commercial use includes:

   (a) integration of all or part of the Software into a product for sale, lease or license by or on behalf of Licensee to third parties, or

   (b) distribution of the Software to third parties that need it to commercialize product sold or licensed by or on behalf of Licensee.

8. Government Rights. Because substantial governmental funds have been used in the development of Charm++/Converse, any possession, use or sublicense of the Software by or to the United States government shall be subject to such required restrictions.

9. Charm++/Converse is being distributed as a research and teaching tool and as such, PPL encourages contributions from users of the code that might, at Illinois' sole discretion, be used or incorporated to make the basic operating framework of the Software a more stable, flexible, and/or useful product. Licensees who contribute their code to become an internal portion of the Software agree that such code may be distributed by Illinois under the terms of this License and may be required to sign an "Agreement Regarding Contributory Code for Charm++/Converse Software" before Illinois can accept it (contact `kale@illinois.edu` for a copy).

UNDERSTOOD AND AGREED.
Contact Information:
The best contact path for licensing issues is by e-mail to `kale@illinois.edu` or send correspondence to:

Prof. L. V. Kale
Dept. of Computer Science
University of Illinois
201 N. Goodwin Ave
Urbana, Illinois 61801 USA
FAX: (217) 244-6500

Contents

1 Introduction

This manual describes Adaptive MPI (AMPI), which is an implementation of the MPI standard[1] on top of Charm++. AMPI acts as a regular MPI implementation (akin to MPICH, OpenMPI, MVAPICH, etc.) with several built-in extensions that allow MPI developers to take advantage of Charm++'s dynamic runtime system, which provides support for process virtualization, overlap of communication and computation, load balancing, and fault tolerance with zero to minimal changes to existing MPI codes.

In this manual, we first describe the philosophy behind Adaptive MPI, then give a brief introduction to Charm++ and rationale for AMPI. We then describe AMPI in detail. Finally we summarize the changes required for existing MPI codes to run with AMPI. Appendices contain the details of installing AMPI, and building and running AMPI programs.

1.1 Overview

Developing parallel Computational Science and Engineering (CSE) applications is a complex task. One has to implement the right physics, develop or choose and code appropriate numerical methods, decide and implement the proper input and output data formats, perform visualizations, and be concerned with correctness and efficiency of the programs. It becomes even more complex for multi-physics coupled simulations, many of which are dynamic and adaptively refined so that load imbalance becomes a major challenge. In addition to imbalance caused by dynamic program behavior, hardware factors such as latencies, variability, and failures must be tolerated by applications. Our philosophy is to lessen the burden of application developers by providing advanced programming paradigms and versatile runtime systems that can handle many common programming and performance concerns automatically and let application programmers focus on the actual application content.

Many of these concerns can be addressed using the processor virtualization and over-decomposition philosophy of Charm++. Thus, the developer only sees virtual processors and lets the runtime system deal with underlying physical processors. This is implemented in AMPI by mapping MPI ranks to Charm++ user-level threads as illustrated in Figure1. As an immediate and simple benefit, the programmer can use as many virtual processors ("MPI ranks") as the problem can be easily decomposed to. For example, suppose the problem domain has $n * 2^n$ parts that can be easily distributed but programming for general number of MPI processes is burdensome, then the developer can have $n * 2^n$ virtual processors on any number of physical ones using AMPI.

AMPI's execution model consists of multiple user-level threads per Processing Element (PE). The Charm++ scheduler coordinates execution of these user-level threads (also called Virtual Processors or VPs) and controls execution. These VPs can also migrate between PEs for the purpose of load balancing or other reasons. The number of VPs per PE specifies the virtualization ratio (degree of over-decomposition). For example, in Figure 1 the virtualization ratio is 3.5 (there are four VPs on PE 0 and three VPs on PE 1). Figure 2 show how the problem domain can be over-decomposed in AMPI's VPs as opposed to other MPI implementations.

Another benefit of virtualization is communication and computation overlap, which is automatically realized in AMPI without programming effort. Techniques such as software pipelining require significant programming effort to achieve this goal and improve performance. However, one can use AMPI to have more virtual processors than physical processors to overlap communication and computation. Each time a VP is blocked for communication, the Charm++ scheduler picks the next VP among those that are ready to execute. In this manner, while some of the VPs of a physical processor are waiting for a message to arrive, others can continue their execution. Thus, performance improves without any changes to the application source code.

Another potential benefit is that of better cache utilization. With over-decomposition, a smaller subdomain is accessed by a VP repeatedly in different function calls before getting blocked by communication and switching to another VP. That smaller subdomain may fit into cache if over-decomposition is enough. This concept is illustrated in Figure 1 where each AMPI rank's subdomain is smaller than the corresponding MPI

[1] Currently, AMPI supports the MPI-2.2 standard, and the MPI-3.1 standard is under active development, though we already support non-blocking and neighborhood collectives among other MPI-3.1 features.

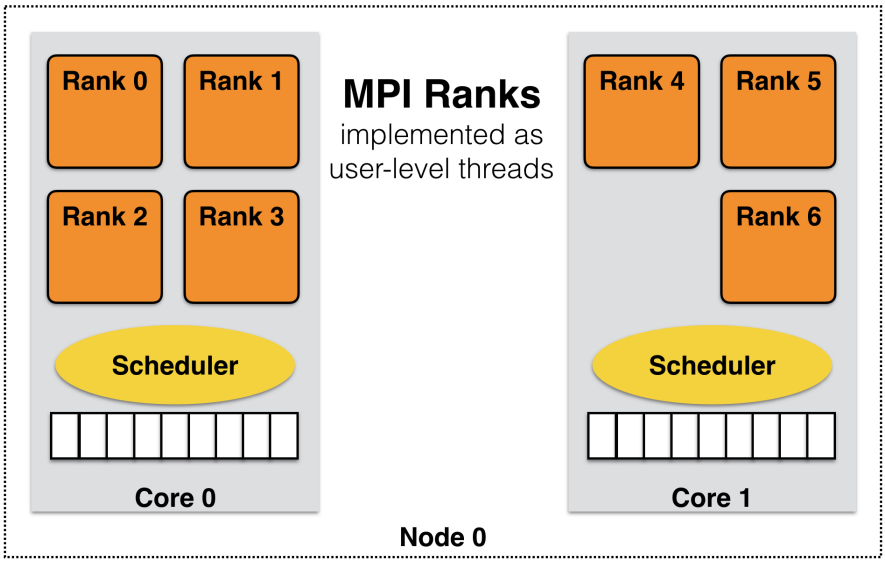

Figure 1: MPI ranks are implemented as user-level threads in AMPI rather than Operating System processes.

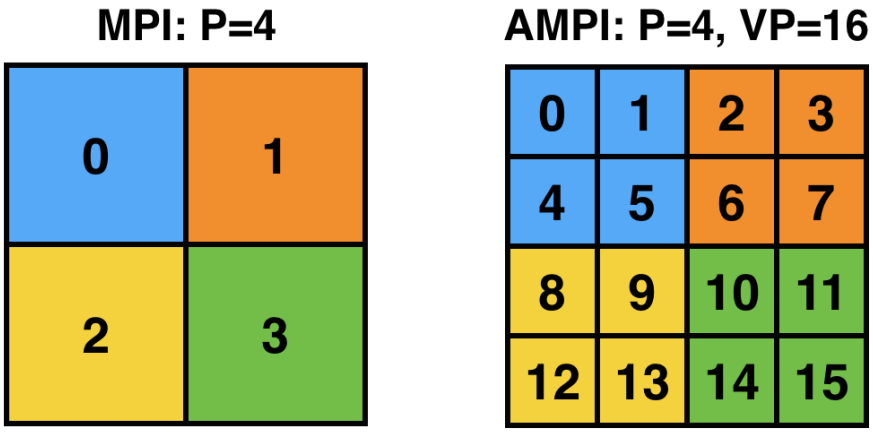

Figure 2: The problem domain is over-decomposed to more VPs than PEs.

subdomain and so may fit into cache memory. Thus, there is a potential performance improvement without changing the source code.

One important concern is that of load imbalance. New generation parallel applications are dynamically varying, meaning that processors' load is shifting during execution. In a dynamic simulation application such as rocket simulation, burning solid fuel, sub-scaling for a certain part of the mesh, crack propagation, particle flows all contribute to load imbalance. A centralized load balancing strategy built into an application is impractical since each individual module is developed mostly independently by various developers. In addition, embedding a load balancing strategy in the code complicates it greatly, and programming effort increases significantly. The runtime system is uniquely positioned to deal with load imbalance. Figure 3 shows the runtime system migrating a VP after detecting load imbalance. This domain may correspond to a weather forecast model where there is a storm cell in the top-left quadrant, which requires more computation to simulate. AMPI will then migrate VP 1 to balance the division of work across processors and improve performance. Note that incorporating this sort of load balancing inside the application code may take a lot of effort and complicate the code.

There are many different load balancing strategies built into Charm++ that can be selected by an

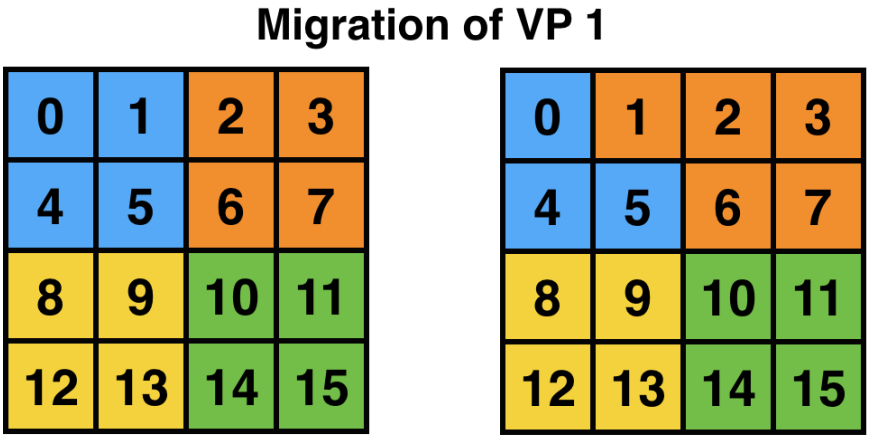

Figure 3: AMPI can migrate VPs across processes for load balancing.

AMPI application developer. Among those, some may fit better for a particular application depending on its characteristics. Moreover, one can write a new load balancer, best suited for an application, by the simple API provided inside Charm++ infrastructure. Our approach is based on actual measurement of load information at runtime, and on migrating computations from heavily loaded to lightly loaded processors.

For this approach to be effective, we need the computation to be split into pieces many more in number than available processors. This allows us to flexibly map and re-map these computational pieces to available processors. This approach is usually called "multi-domain decomposition".

Charm++, which we use as a runtime system layer for the work described here, simplifies our approach. It embeds an elaborate performance tracing mechanism, a suite of plug-in load balancing strategies, infrastructure for defining and migrating computational load, and is interoperable with other programming paradigms.

2 Charm++

Charm++ is an object-oriented parallel programming library for C++. It differs from traditional message passing programming libraries (such as MPI) in that Charm++ is "message-driven". Message-driven parallel programs do not block the processor waiting for a message to be received. Instead, each message carries with itself a computation that the processor performs on arrival of that message. The underlying runtime system of Charm++ is called Converse, which implements a "scheduler" that chooses which message to schedule next (message-scheduling in Charm++ involves locating the object for which the message is intended, and executing the computation specified in the incoming message on that object). A parallel object in Charm++ is a C++ object on which a certain computations can be asked to be performed from remote processors.

Charm++ programs exhibit latency tolerance since the scheduler always picks up the next available message rather than waiting for a particular message to arrive. They also tend to be modular, because of their object-based nature. Most importantly, Charm++ programs can be *dynamically load balanced*, because the messages are directed at objects and not at processors; thus allowing the runtime system to migrate the objects from heavily loaded processors to lightly loaded processors.

Since many CSE applications are originally written using MPI, one would have to rewrite existing code if they were to be converted to Charm++ to take advantage of dynamic load balancing and other Charm++ features. This is indeed impractical. However, Converse – the runtime system of Charm++ – supports interoperability between different parallel programming paradigms such as parallel objects and threads. Using this feature, we developed AMPI, which is described in more detail in the next section.

3 AMPI

AMPI utilizes the dynamic load balancing and other capabilities of Charm++ by associating a "user-level" thread with each Charm++ migratable object. User's code runs inside this thread, so that it can issue blocking receive calls similar to MPI, and still present the underlying scheduler an opportunity to schedule other computations on the same processor. The runtime system keeps track of the computational loads of each thread as well as the communication graph between AMPI threads, and can migrate these threads in order to balance the overall load while simultaneously minimizing communication overhead.

3.1 AMPI Compliance to MPI Standards

Currently AMPI supports the MPI-2.2 standard, with preliminary support for most MPI-3.1 features and a collection of extensions explained in detail in this manual. One-sided communication calls in MPI-2 and MPI-3 are implemented, but they do not yet take advantage of RMA features. Non-blocking collectives have been defined in AMPI since before MPI-3.0's adoption of them. Also ROMIO[2] has been integrated into AMPI to support parallel I/O features.

3.2 AMPI Extensions to MPI Standards

The following are AMPI extensions to the MPI standard, which will be explained in detail in this manual. All AMPI extensions to the MPI standard are prefixed with `AMPI_` rather than `MPI_`. All extensions are available in C, C++, and Fortran, with the exception of `AMPI_Command_argument_count` and `AMPI_Get_command_argument` which are only available in Fortran.

```
AMPI_Migrate           AMPI_Register_pup            AMPI_Get_pup_data
AMPI_Migrate_to_pe     AMPI_Set_migratable          AMPI_Evacuate
AMPI_Load_set_value    AMPI_Load_start_measure      AMPI_Load_stop_measure
AMPI_Iget              AMPI_Iget_wait               AMPI_Iget_data
AMPI_Iget_free         AMPI_Type_is_contiguous      AMPI_Register_main
AMPI_Yield             AMPI_Suspend                 AMPI_Resume
AMPI_Alltoall_medium   AMPI_Alltoall_long
AMPI_Register_just_migrated          AMPI_Register_about_to_migrate
AMPI_Command_argument_count          AMPI_Get_command_argument
```

AMPI provides a set of built-in attributes on all communicators and windows to find the number of the worker thread, process, or host that a rank is currently running on, as well as the total number of worker threads, processes, and hosts in the job. We define a worker thread to be a thread on which one of more AMPI ranks are scheduled. We define a process here as an operating system process, which may contain one or more worker threads. The built-in attributes are `AMPI_MY_WTH`, `AMPI_MY_PROCESS`, `AMPI_NUM_WTHS`, and `AMPI_NUM_PROCESSES`. These attributes are accessible from any rank by calling `MPI_Comm_get_attr`, such as:

```
! Fortran:
integer :: my_wth, flag, ierr
call MPI_Comm_get_attr(MPI_COMM_WORLD, AMPI_MY_WTH, my_wth, flag, ierr)

// C/C++:
int my_wth, flag;
MPI_Comm_get_attr(MPI_COMM_WORLD, AMPI_MY_WTH, &my_wth, &flag);
```

AMPI also provides extra communicator types that users can pass to `MPI_Comm_split_type`: `AMPI_COMM_TYPE_HOST` for splitting a communicator into disjoint sets of ranks that share the same physical host, `AMPI_COMM_TYPE_PROCESS` for splitting a communicator into disjoint sets of ranks that share the same operating system process, and `AMPI_COMM_TYPE_WTH`, for splitting a communicator into disjoint sets of ranks that share the same worker thread.

[2] http://www-unix.mcs.anl.gov/romio/

For parsing Fortran command line arguments, AMPI Fortran programs should use our extension APIs, which are similar to Fortran 2003's standard APIs. For example:

```
integer :: i, argc, ierr
integer, parameter :: arg_len = 128
character(len=arg_len), dimension(:), allocatable :: raw_arguments

call AMPI_Command_argument_count(argc)
allocate(raw_arguments(argc))
do i = 1, size(raw_arguments)
    call AMPI_Get_command_argument(i, raw_arguments(i), arg_len, ierr)
end do
```

3.3 Name for Main Program

To convert an existing program to use AMPI, the main function or program may need to be renamed. The changes should be made as follows:

3.3.1 Fortran

You must declare the main program as a subroutine called "MPI_MAIN". Do not declare the main subroutine as a *program* because it will never be called by the AMPI runtime.

```
program pgm -> subroutine MPI_Main
...             ...
end program -> end subroutine
```

3.3.2 C or C++

The main function can be left as is, if `mpi.h` is included before the main function. This header file has a preprocessor macro that renames main, and the renamed version is called by the AMPI runtime by each thread.

3.4 Global Variable Privatization

For the before-mentioned benefits to be effective, one needs to map multiple user-level threads onto each processor. Traditional MPI programs assume that the entire processor is allocated to themselves, and that only one thread of control exists within the process's address space. So, they may use global and static variables in the program. However, global and static variables are problematic for multi-threaded environments such as AMPI or OpenMP. This is because there is a single instance of those variables so they will be shared among different threads in the single address space and a wrong result may be produced by the program. Figure 4 shows an example of a multi-threaded application with two threads in a single process. *var* is a global or static variable in this example. Thread 1 assigns a value to it, then it gets blocked for communication and another thread can continue. Thereby, thread 2 is scheduled next and accesses *var* which is wrong. Semantics of this program needs separate instances of *var* for each of the threads. That is where the need arises to make some transformations to the original MPI program in order to run correctly with AMPI. Note, this is the only change necessary to run an MPI program with AMPI, that the program be thread-safe and have no global variables whose values differ across different MPI ranks.

The basic transformation needed to port the MPI program to AMPI is privatization of global variables.[3] With the MPI process model, each MPI node can keep a copy of its own "permanent variables" – variables

[3]Typical Fortran MPI programs contain three types of global variables.

1. Global variables that are "read-only". These are either *parameters* that are set at compile-time. Or other variables that are read as input or set at the beginning of the program and do not change during execution. It is not necessary to privatize such variables.

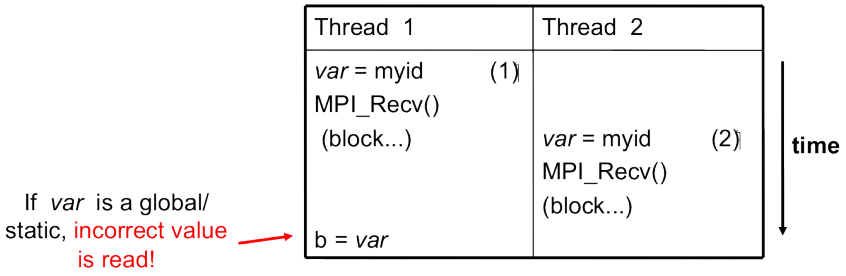

Figure 4: Global or static variables are an issue for AMPI

that are accessible from more than one subroutines without passing them as arguments. Module variables, "saved" subroutine local variables, and common blocks in Fortran90 belong to this category. If such a program is executed without privatization on AMPI, all the AMPI threads that reside on one processor will access the same copy of such variables, which is clearly not the desired semantics. To ensure correct execution of the original source program, it is necessary to make such variables "private" to individual threads. We provide two choices: automatic global swapping and manual code modification.

3.4.1 Automatic Globals Swapping

Thanks to the ELF Object Format, we have successfully automated the procedure of switching the set of user global variables when switching thread contexts. Executable and Linkable Format (ELF) is a common standard file format for Object Files in Unix-like operating systems. ELF maintains a Global Offset Table (GOT) for globals so it is possible to switch GOT contents at thread context-switch by the runtime system.

The only thing that the user needs to do is to set flag `-swapglobals` at compile and link time (e.g. "ampicc -o prog prog.c -swapglobals"). It does not need any change to the source code and works with any language (C, C++, Fortran, etc). However, it does not handle static variables and has a context switching overhead that grows with the number of global variables. Currently, this feature only works on x86 and x86_64 platforms that fully support ELF. Thus, it may not work on PPC or on some microkernels such as Catamount. When this feature does not work for you, you can try other ways of handling global or static variables, which are detailed in the following sections.

3.4.2 Manual Change

We have employed a strategy of argument passing to do this privatization transformation. That is, the global variables are bunched together in a single user-defined type, which is allocated by each thread dynamically. Then a pointer to this type is passed from subroutine to subroutine as an argument. Since the subroutine arguments are passed on the stack, which is not shared across all threads, each subroutine when executing within a thread operates on a private copy of the global variables.

This scheme is demonstrated in the following examples. The original Fortran90 code contains a module `shareddata`. This module is used in the main program and a subroutine `subA`.

```
!FORTRAN EXAMPLE
MODULE shareddata
  INTEGER :: myrank
  DOUBLE PRECISION :: xyz(100)
END MODULE
```

2. Global variables that are used as temporary buffers. These are variables that are used temporarily to store values to be accessible across subroutines. These variables have a characteristic that there is no blocking call such as `MPI_recv` between the time the variable is set and the time it is ever used. It is not necessary to privatize such variables either.

3. True global variables. These are used across subroutines that contain blocking receives and therefore the possibility of a context switch between the definition and use of the variable. These variables need to be privatized.

```
SUBROUTINE MPI_MAIN
  USE shareddata
  include 'mpif.h'
  INTEGER :: i, ierr
  CALL MPI_Init(ierr)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierr)
  DO i = 1, 100
    xyz(i) =  i + myrank
  END DO
  CALL subA
  CALL MPI_Finalize(ierr)
END PROGRAM

SUBROUTINE subA
  USE shareddata
  INTEGER :: i
  DO i = 1, 100
    xyz(i) = xyz(i) + 1.0
  END DO
END SUBROUTINE

//C Example
#include <mpi.h>

int myrank;
double xyz[100];

void subA();
int main(int argc, char** argv){
  int i;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  for(i=0;i<100;i++)
    xyz[i] = i + myrank;
  subA();
  MPI_Finalize();
}

void subA(){
  int i;
  for(i=0;i<100;i++)
    xyz[i] = xyz[i] + 1.0;
}
```

AMPI executes the main subroutine inside a user-level thread as a subroutine.

Now we transform this program using the argument passing strategy. We first group the shared data into a user-defined type.

```
!FORTRAN EXAMPLE
MODULE shareddata
  TYPE chunk
    INTEGER :: myrank
    DOUBLE PRECISION :: xyz(100)
```

```
  END TYPE
END MODULE

//C Example
struct shareddata{
  int myrank;
  double xyz[100];
};
```

Now we modify the main subroutine to dynamically allocate this data and change the references to them. Subroutine `subA` is then modified to take this data as argument.

```
!FORTRAN EXAMPLE
SUBROUTINE MPI_Main
  USE shareddata
  USE AMPI
  INTEGER :: i, ierr
  TYPE(chunk), pointer :: c
  CALL MPI_Init(ierr)
  ALLOCATE(c)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, c%myrank, ierr)
  DO i = 1, 100
    c%xyz(i) =  i + c%myrank
  END DO
  CALL subA(c)
  CALL MPI_Finalize(ierr)
END SUBROUTINE

SUBROUTINE subA(c)
  USE shareddata
  TYPE(chunk) :: c
  INTEGER :: i
  DO i = 1, 100
    c%xyz(i) = c%xyz(i) + 1.0
  END DO
END SUBROUTINE

//C Example
void MPI_Main{
  int i,ierr;
  struct shareddata *c;
  ierr = MPI_Init();
  c = (struct shareddata*)malloc(sizeof(struct shareddata));
  ierr = MPI_Comm_rank(MPI_COMM_WORLD, c.myrank);
  for(i=0;i<100;i++)
    c.xyz[i] = i + c.myrank;
  subA(c);
  ierr = MPI_Finalize();
}

void subA(struct shareddata *c){
  int i;
  for(i=0;i<100;i++)
    c.xyz[i] = c.xyz[i] + 1.0;
```

Privatization Scheme	x86	x86_64	Mac OS	BG/Q	Windows	PPC	ARM7
Transformation	Yes	Yes	Yes	Yes	Yes	Yes	Yes
GOT-Globals	Yes	Yes	No	No	No	Yes	Yes
TLS-Globals	Yes	Yes	No	No	Maybe	Maybe	Maybe

Table 1: Portability of current implementations of three privatization schemes. "Yes" means we have implemented this technique. "Maybe" indicates there are no theoretical problems, but no implementation exists. "No" indicates the technique is impossible on this platform.
}

With these changes, the above program can be made thread-safe. Note that it is not really necessary to dynamically allocate `chunk`. One could have declared it as a local variable in subroutine `MPI_Main`. (Or for a small example such as this, one could have just removed the `shareddata` module, and instead declared both variables `xyz` and `myrank` as local variables). This is indeed a good idea if shared data are small in size. For large shared data, it would be better to do heap allocation because in AMPI, the stack sizes are fixed at the beginning (and can be specified from the command line) and stacks do not grow dynamically.

3.4.3 Source-to-source Transformation

Another approach is to do the changes described in the previous scheme automatically. It means that we can use a tool to transform the source code to move global or static variables in an object and pass them around. This approach is portable across systems and compilers and may also improve locality and hence cache utilization. It also does not have the context-switch overhead of swapping globals. We have multiple tools for automating these transformations for different languages. Currently, there is a tool called *Photran*[4] for refactoring Fortran codes that can do this transformation. It is Eclipse-based and works by constructing Abstract Syntax Trees (ASTs) of the program. We also have a tool built on top of the *ROSE compiler*[5] that works for C/C++ and Fortran programs that is available upon request.

3.4.4 TLS-Globals

Thread Local Store (TLS) was originally employed in kernel threads to localize variables and provide thread safety. It can be used by annotating global/static variables in C/C++ with *__thread* in the source code. Thus, those variables will have one instance per extant thread. This keyword is not an official extension of the C language, though compiler writers are encouraged to implement this feature. Currently, the ELF file format supports Thread Local Storage.

It handles both global and static variables and has no context-switching overhead. Context-switching is just changing the TLS segment register to point to the thread's local copy. However, although it is popular, it is not supported by all compilers. Currently, Charm++ supports it for x86/x86_64 platforms. A modified *gfortran* is also available to use this feature upon request. To use TLS-Globals, one has to add *__thread* before all global variables. For the example above, the following changes to the code handles the global variables:

```
__thread int myrank;
__thread double xyz[100];
```

The runtime system also should know that TLS-Globals is used at compile time:

```
ampiCC -o example example.C -tlsglobals
```

Table 1 shows portability of different schemes.

[4]http://www.eclipse.org/photran
[5]http://rosecompiler.org/

3.5 Extensions for Migrations

AMPI provides fully automated support for migrating MPI ranks between nodes of a system without any application-specific code at all. We do so using a memory allocator, Isomalloc, that allocates memory per user-level thread to globally unique virtual memory addresses. This means that every worker thread in the system reserves slices of virtual memory for all user-level threads, allowing transparent migration of stacks and pointers into memory (Isomalloc requires 64-bit virtual memory addresses and support from the operating system for mapping memory to arbitrary virtual addresses). Applications only need to link with Isomalloc to enable automatic migratability, using *-memory isomalloc*.

For systems that do not support Isomalloc and for users that wish to have more fine-grain control over which application data structures will be copied at migration time, we have added a few calls to AMPI. These include the ability to register thread-specific data with the run-time system, to pack and unpack all of the thread's data, and to express willingness to migrate.

3.5.1 Registering User Data

When the AMPI runtime system decides that load imbalance exists within the application, it will invoke one of its internal load balancing strategies, which determines the new mapping of AMPI ranks so as to balance the load. Then the AMPI runtime packs up the rank's state and moves it to its new home processor. AMPI packs up any internal data in use by the rank, including the thread's stack in use. This means that the local variables declared in subroutines in a rank, which are created on stack, are automatically packed up by the AMPI runtime system. However, it has no way of knowing what other data are in use by the rank. Thus upon starting execution, a rank needs to notify the system about the data that it is going to use (apart from local variables). Even with the data registration, AMPI cannot determine what size the data is, or whether the registered data contains pointers to other places in memory. For this purpose, a packing subroutine also needs to be provided to the AMPI runtime system along with registered data. (See next section for writing packing subroutines.) The call provided by AMPI for doing this is `AMPI_Register_pup`. This function takes three arguments: a data item to be transported along with the rank, the pack subroutine, and a pointer to an integer which denotes the registration identifier. In C/C++ programs, it may be necessary to use this integer value after migration completes and control returns to the rank with the function `AMPI_Get_pup_data`.

3.5.2 Migration

The AMPI runtime system could detect load imbalance by itself and invoke the load balancing strategy. However, since the application code is going to pack/unpack the rank's data, writing the pack subroutine will be complicated if migrations occur at a stage unknown to the application. For example, if the system decides to migrate a rank while it is in initialization stage (say, reading input files), application code will have to keep track of how much data it has read, what files are open etc. Typically, since initialization occurs only once in the beginning, load imbalance at that stage would not matter much. Therefore, we want the demand to perform load balance check to be initiated by the application.

AMPI provides a subroutine `AMPI_Migrate(MPI_Info hints);` for this purpose. Each rank periodically calls `AMPI_Migrate`. Typical CSE applications are iterative and perform multiple time-steps. One should call `AMPI_Migrate` in each rank at the end of some fixed number of timesteps. The frequency of `AMPI_Migrate` should be determined by a tradeoff between conflicting factors such as the load balancing overhead, and performance degradation caused by load imbalance. In some other applications, where application suspects that load imbalance may have occurred, as in the case of adaptive mesh refinement; it would be more effective if it performs a couple of timesteps before telling the system to re-map ranks. This will give the AMPI runtime system some time to collect the new load and communication statistics upon which it bases its migration decisions. Note that `AMPI_Migrate` does NOT tell the system to migrate the rank, but merely tells the system to check the load balance after all the ranks call `AMPI_Migrate`. To migrate the rank or not is decided only by the system's load balancing strategy.

Essentially, a call to `AMPI_Migrate` signifies to the runtime system that the application has reached a point at which it is safe to serialize the local state. Knowing this, the runtime system can act in several ways.

The MPI_Info object taken as a parameter by `AMPI_Migrate` gives users a way to influence the runtime system's decision-making and behavior. AMPI provides two built-in MPI_Info objects for this, called `AMPI_INFO_LB_SYNC` and `AMPI_INFO_LB_ASYNC`. Synchronous load balancing assumes that the application is already at a synchronization point. Asynchronous load balancing does not assume this.

Calling `AMPI_Migrate` on a rank with pending send requests (i.e. from MPI_Isend) is currently not supported, therefore users should always wait on any outstanding send requests before calling `AMPI_Migrate`.

```
// Main time-stepping loop
for (int iter=0; iter < max_iters; iter++) {

  // Time step work ...

  if (iter % lb_freq == 0)
    AMPI_Migrate(AMPI_INFO_LB_SYNC);
}
```

Note that migrating ranks around the cores and nodes of a system can change which ranks share physical resources, such as memory. A consequence of this is that communicators created via `MPI_Comm_split_type` are invalidated by calls to `AMPI_Migrate` that result in migration which breaks the semantics of that communicator type. The only valid routine to call on such communicators is `MPI_Comm_free`.

We also provide callbacks that user code can register with the runtime system to be invoked just before and right after migration: `AMPI_Register_about_to_migrate` and `AMPI_Register_just_migrated` respectively. Note that the callbacks are only invoked on those ranks that are about to actually migrate or have just actually migrated.

AMPI provide routines for starting and stopping load measurements, and for users to explicitly set the load value of a rank using the following: `AMPI_Load_start_measure`, `AMPI_Load_stop_measure`, `AMPI_Load_reset_measure`, and `AMPI_Load_set_value`. And since AMPI builds on top of Charm++, users can experiment with the suite of load balancing strategies included with Charm++, as well as write their own strategies based on user-level information and heuristics.

3.5.3 Packing/Unpacking Thread Data

Once the AMPI runtime system decides which ranks to send to which processors, it calls the specified pack subroutine for that rank, with the rank-specific data that was registered with the system using `AMPI_Register_pup`. If an AMPI application uses Isomalloc, then the system will define the Pack/Unpack routines for the user. This section explains how a subroutine should be written for performing explicit pack/unpack.

There are three steps for transporting the rank's data to another processor. First, the system calls a subroutine to get the size of the buffer required to pack the rank's data. This is called the "sizing" step. In the next step, which is called immediately afterward on the source processor, the system allocates the required buffer and calls the subroutine to pack the rank's data into that buffer. This is called the "packing" step. This packed data is then sent as a message to the destination processor, where first a rank is created (along with the thread) and a subroutine is called to unpack the rank's data from the buffer. This is called the "unpacking" step.

Though the above description mentions three subroutines called by the AMPI runtime system, it is possible to actually write a single subroutine that will perform all the three tasks. This is achieved using something we call a "pupper". A pupper is an external subroutine that is passed to the rank's pack-unpack-sizing subroutine, and this subroutine, when called in different phases performs different tasks. An example will make this clear:

Suppose the user data, chunk, is defined as a derived type in Fortran90:

```
!FORTRAN EXAMPLE
MODULE chunkmod
  INTEGER, parameter :: nx=4, ny=4, tchunks=16
  TYPE, PUBLIC :: chunk
```

```
      REAL(KIND=8) t(22,22)
      INTEGER xidx, yidx
      REAL(KIND=8), dimension(400):: bxm, bxp, bym, byp
  END TYPE chunk
END MODULE

//C Example
struct chunk{
  double t;
  int xidx, yidx;
  double bxm,bxp,bym,byp;
};
```

Then the pack-unpack subroutine `chunkpup` for this chunk module is written as:

```
!FORTRAN EXAMPLE
SUBROUTINE chunkpup(p, c)
  USE pupmod
  USE chunkmod
  IMPLICIT NONE
  INTEGER :: p
  TYPE(chunk) :: c

  call pup(p, c%t)
  call pup(p, c%xidx)
  call pup(p, c%yidx)
  call pup(p, c%bxm)
  call pup(p, c%bxp)
  call pup(p, c%bym)
  call pup(p, c%byp)
end subroutine

//C Example
void chunkpup(pup_er p, struct chunk c){
  pup_double(p,c.t);
  pup_int(p,c.xidx);
  pup_int(p,c.yidx);
  pup_double(p,c.bxm);
  pup_double(p,c.bxp);
  pup_double(p,c.bym);
  pup_double(p,c.byp);
}
```

There are several things to note in this example. First, the same subroutine `pup` (declared in module `pupmod`) is called to size/pack/unpack any type of data. This is possible because of procedure overloading possible in Fortran90. Second is the integer argument `p`. It is this argument that specifies whether this invocation of subroutine `chunkpup` is sizing, packing or unpacking. Third, the integer parameters declared in the type `chunk` need not be packed or unpacked since they are guaranteed to be constants and thus available on any processor.

A few other functions are provided in module `pupmod`. These functions provide more control over the packing/unpacking process. Suppose one modifies the `chunk` type to include allocatable data or pointers that are allocated dynamically at runtime. In this case, when chunk is packed, these allocated data structures should be deallocated after copying them to buffers, and when chunk is unpacked, these data structures should be allocated before copying them from the buffers. For this purpose, one needs to know whether

the invocation of `chunkpup` is a packing one or unpacking one. For this purpose, the `pupmod` module provides functions `fpup_isdeleting`(`fpup_isunpacking`). These functions return logical value `.TRUE.` if the invocation is for packing (unpacking), and `.FALSE.` otherwise. The following example demonstrates this:

Suppose the type `dchunk` is declared as:

```
!FORTRAN EXAMPLE
MODULE dchunkmod
  TYPE, PUBLIC :: dchunk
      INTEGER :: asize
      REAL(KIND=8), pointer :: xarr(:), yarr(:)
  END TYPE dchunk
END MODULE

//C Example
struct dchunk{
  int asize;
  double* xarr, *yarr;
};
```

Then the pack-unpack subroutine is written as:

```
!FORTRAN EXAMPLE
SUBROUTINE dchunkpup(p, c)
  USE pupmod
  USE dchunkmod
  IMPLICIT NONE
  INTEGER :: p
  TYPE(dchunk) :: c

  pup(p, c%asize)

  IF (fpup_isunpacking(p)) THEN       !! if invocation is for unpacking
    allocate(c%xarr(c%asize))
    ALLOCATE(c%yarr(c%asize))
  ENDIF

  pup(p, c%xarr)
  pup(p, c%yarr)

  IF (fpup_isdeleting(p)) THEN        !! if invocation is for packing
    DEALLOCATE(c%xarr)
    DEALLOCATE(c%yarr)
  ENDIF


END SUBROUTINE

//C Example
void dchunkpup(pup_er p, struct dchunk c){
  pup_int(p,c.asize);
  if(pup_isUnpacking(p)){
    c.xarr = (double *)malloc(sizeof(double)*c.asize);
    c.yarr = (double *)malloc(sizeof(double)*c.asize);
  }
```

```
  pup_doubles(p,c.xarr,c.asize);
  pup_doubles(p,c.yarr,c.asize);
  if(pup_isPacking(p)){
    free(c.xarr);
    free(c.yarr);
  }
}
```

One more function `fpup_issizing` is also available in module `pupmod` that returns `.TRUE.` when the invocation is a sizing one. In practice one almost never needs to use it.

Charm++ also provides higher-level PUP routines for C++ STL data structures and Fortran90 data types. The STL PUP routines will deduce the size of the structure automatically, so that the size of the data does not have to be passed in to the PUP routine. This facilitates writing PUP routines for large pre-existing codebases. To use it, simply include pup_stl.h in the user code. For modern Fortran with pointers and allocatable data types, AMPI provides a similarly automated PUP interface called apup. User code can include pupmod and then call apup() on any array (pointer or allocatable, multi-dimensional) of built-in types (character, short, int, long, real, double, complex, double complex, logical) and the runtime will deduce the size and shape of the array, including unassociated and NULL pointers. Here is the dchunk example from earlier, written to use the apup interface:

```
!FORTRAN EXAMPLE
SUBROUTINE dchunkpup(p, c)
  USE pupmod
  USE dchunkmod
  IMPLICIT NONE
  INTEGER :: p
  TYPE(dchunk) :: c

  !! no need for asize
  !! no isunpacking allocation necessary


  apup(p, c%xarr)
  apup(p, c%yarr)


  !! no isdeleting deallocation necessary

END SUBROUTINE
```

Calling MPI_ routines or accessing global variables that have been privatized by use of tlsglobals or swapglobals from inside a user PUP routine is currently not allowed in AMPI. Users can store MPI-related information like communicator rank and size in data structures to be be packed and unpacked before they are needed inside a PUP routine.

3.6 Extensions for Checkpointing

The pack-unpack subroutines written for migrations make sure that the current state of the program is correctly packed (serialized) so that it can be restarted on a different processor. Using the *same* subroutines, it is also possible to save the state of the program to disk, so that if the program were to crash abruptly, or if the allocated time for the program expires before completing execution, the program can be restarted from the previously checkpointed state. Thus, the pack-unpack subroutines act as the key facility for checkpointing in addition to their usual role for migration. Just as in load balancing, no application specific code is required when using Isomalloc: the AMPI runtime takes care of all the details involved in migrating data.

To perform a checkpoint in an AMPI program, all you have to do is make a call to `int AMPI_Migrate(MPI_Info hints)` with an `MPI_Info` object that specifies how you would like to checkpoint. Checkpointing can be thought of as migrating AMPI ranks to storage. Users set the checkpointing policy on an `MPI_Info` object's `"ampi_checkpoint"` key to one of the following values: `"to_file=directory_name"` or `"false"`. To perform checkpointing in memory a built-in MPI_Info object called `AMPI_INFO_CHKPT_IN_MEMORY` is provided.

Checkpointing to file tells the runtime system to save checkpoints in a given directory. (Typically, in an iterative program, the iteration number, converted to a character string, can serve as a checkpoint directory name.) This directory is created, and the entire state of the program is checkpointed to this directory. One can restart the program from the checkpointed state (using the same, more, or fewer physical processors than were checkpointed with) by specifying `"+restart directory_name"` on the command-line.

Checkpointing in memory allows applications to transparently tolerate failures online. The checkpointing scheme used here is a double in-memory checkpoint, in which virtual processors exchange checkpoints pairwise across nodes in each other's memory such that if one node fails, that failed node's AMPI ranks can be restarted by its buddy once the failure is detected by the runtime system. As long as no two buddy nodes fail in the same checkpointing interval, the system can restart online without intervention from the user (provided the job scheduler does not revoke its allocation). Any load imbalance resulting from the restart can then be managed by the runtime system. Use of this scheme is illustrated in the code snippet below.

```
// Main time-stepping loop
for (int iter=0; iter < max_iters; iter++) {

  // Time step work ...

  if (iter % chkpt_freq == 0)
    AMPI_Migrate(AMPI_INFO_CHKPT_IN_MEMORY);
}
```

A value of `"false"` results in no checkpoint being done that step. Note that `AMPI_Migrate` is a collective function, meaning every virtual processor in the program needs to call this subroutine with the same MPI_Info object. The checkpointing capabilities of AMPI are powered by the Charm++ runtime system. For more information about checkpoint/restart mechanisms please refer to the Charm++ manual **??**.

3.7 Extensions for Memory Efficiency

MPI functions usually require the user to preallocate the data buffers needed before the functions being called. For unblocking communication primitives, sometimes the user would like to do lazy memory allocation until the data actually arrives, which gives the opportunities to write more memory efficient programs. We provide a set of AMPI functions as an extension to the standard MPI-2 one-sided calls, where we provide a split phase `MPI_Get` called `AMPI_Iget`. `AMPI_Iget` preserves the similar semantics as `MPI_Get` except that no user buffer is provided to hold incoming data. `AMPI_Iget_wait` will block until the requested data arrives and runtime system takes care to allocate space, do appropriate unpacking based on data type, and return. `AMPI_Iget_free` lets the runtime system free the resources being used for this get request including the data buffer. Finally, `AMPI_Iget_data` is the routine used to access the data.

```
int AMPI_Iget(MPI_Aint orgdisp, int orgcnt, MPI_Datatype orgtype, int rank,
              MPI_Aint targdisp, int targcnt, MPI_Datatype targtype, MPI_Win win,
              MPI_Request *request);

int AMPI_Iget_wait(MPI_Request *request, MPI_Status *status, MPI_Win win);

int AMPI_Iget_free(MPI_Request *request, MPI_Status *status, MPI_Win win);
```

```
int AMPI_Iget_data(void *data, MPI_Status status);
```

3.8 Extensions for Interoperability

Interoperability between different modules is essential for coding coupled simulations. In this extension to AMPI, each MPI application module runs within its own group of user-level threads distributed over the physical parallel machine. In order to let AMPI know which ranks are to be created, and in what order, a top level registration routine needs to be written. A real-world example will make this clear. We have an MPI code for fluids and another MPI code for solids, both with their main programs, then we first transform each individual code to run correctly under AMPI as standalone codes. Aside from the global and static variable privatization transformations needed, this also involves making the main program into a subroutine and naming it `MPI_Main`.

Thus now, we have two `MPI_Main`s, one for the fluids code and one for the solids code. We now make these codes co-exist within the same executable, by first renaming these `MPI_Main`s as `Fluids_Main` and `Solids_Main`[6] writing a subroutine called `MPI_Setup`.

```
!FORTRAN EXAMPLE
SUBROUTINE MPI_Setup
  USE ampi
  CALL AMPI_Register_main(Solids_Main)
  CALL AMPI_Register_main(Fluids_Main)
END SUBROUTINE

//C Example
void MPI_Setup(){
  AMPI_Register_main(Solids_Main);
  AMPI_Register_main(Fluids_Main);
}
```

This subroutine is called from the internal initialization routines of AMPI and tells AMPI how many numbers of distinct modules exist, and which orchestrator subroutines they execute.

The number of ranks to create for each module is specified on the command line when an AMPI program is run. Appendix B explains how AMPI programs are run, and how to specify the number of ranks (`+vp` option). In the above case, suppose one wants to create 128 ranks of Solids and 64 ranks of Fluids on 32 physical processors, one would specify those with multiple `+vp` options on the command line as:

```
> charmrun gen1.x +p 32 +vp 128 +vp 64
```

This will ensure that multiple modules representing different complete applications can co-exist within the same executable. They can also continue to communicate among their own ranks using the same AMPI function calls to send and receive with communicator argument as `MPI_COMM_WORLD`. But this would be completely useless if these individual applications cannot communicate with each other, which is essential for building efficient coupled codes. For this purpose, we have extended the AMPI functionality to allow multiple "`COMM_WORLD`s"; one for each application. These *world communicators* form a "communicator universe": an array of communicators aptly called *MPI_COMM_UNIVERSE*. This array of communicators is indexed [1 . . . `MPI_MAX_COMM`]. In the current implementation, `MPI_MAX_COMM` is 8, that is, maximum of 8 applications can co-exist within the same executable.

The order of these `COMM_WORLD`s within `MPI_COMM_UNIVERSE` is determined by the order in which individual applications are registered in `MPI_Setup`.

Thus, in the above example, the communicator for the Solids module would be `MPI_COMM_UNIVERSE(1)` and communicator for Fluids module would be `MPI_COMM_UNIVERSE(2)`.

[6] Currently, we assume that the interface code, which does mapping and interpolation among the boundary values of Fluids and Solids domain, is integrated with one of Fluids and Solids.

Now any rank within one application can communicate with any rank in the other application using the familiar send or receive AMPI calls by specifying the appropriate communicator and the rank number within that communicator in the call. For example if a Solids rank number 36 wants to send data to rank number 47 within the Fluids module, it calls:

```
!FORTRAN EXAMPLE
INTEGER , PARAMETER :: Fluids_Comm = 2
CALL MPI_Send(InitialTime, 1, MPI_Double_Precision, tag,
              47, MPI_Comm_Universe(Fluids_Comm) , ierr)

//C Example
int Fluids_Comm = 2;
ierr = MPI_Send(InitialTime, 1, MPI_DOUBLE, tag,
                47, MPI_Comm_Universe(Fluids_Comm));
```

The Fluids rank has to issue a corresponding receive call to receive this data:

```
!FORTRAN EXAMPLE
INTEGER , PARAMETER :: Solids_Comm = 1
CALL MPI_Recv(InitialTime, 1, MPI_Double_Precision, tag,
              36, MPI_Comm_Universe(Solids_Comm) , stat, ierr)

//C Example
int Solids_Comm = 1;
ierr = MPI_Recv(InitialTime, 1, MPI_DOUBLE, tag,
                36, MPI_Comm_Universe(Solids_Comm) , &stat);
```

3.9 Extensions for Sequential Re-run of a Parallel Node

In some scenarios, a sequential re-run of a parallel node is desired. One example is instruction-level accurate architecture simulations, in which case the user may wish to repeat the execution of a node in a parallel run in the sequential simulator. AMPI provides support for such needs by logging the change in the MPI environment on a certain processors. To activate the feature, build AMPI module with variable "AMPIMSGLOG" defined, like the following command in charm directory. (Linking with zlib "-lz" might be required with this, for generating compressed log file.)

```
> ./build AMPI netlrts-linux-x86_64 -DAMPIMSGLOG
```

The feature is used in two phases: writing (logging) the environment and repeating the run. The first logging phase is invoked by a parallel run of the AMPI program with some additional command line options.

```
> ./charmrun ./pgm +p4 +vp4 +msgLogWrite +msgLogRank 2 +msgLogFilename "msg2.log"
```

In the above example, a parallel run with 4 worker threads and 4 AMPI ranks will be executed, and the changes in the MPI environment of worker thread 2 (also rank 2, starting from 0) will get logged into diskfile "msg2.log".

Unlike the first run, the re-run is a sequential program, so it is not invoked by charmrun (and omitting charmrun options like +p4 and +vp4), and additional command line options are required as well.

```
> ./pgm +msgLogRead +msgLogRank 2 +msgLogFilename "msg2.log"
```

3.10 User Defined Initial Mapping

You can define the initial mapping of virtual processors (vp) to physical processors (p) as a runtime option. You can choose from predefined initial mappings or define your own mappings. The following predefined mappings are available:

Round Robin This mapping scheme maps virtual processor to physical processor in round-robin fashion, i.e. if there are 8 virtual processors and 2 physical processors then virtual processors indexed 0,2,4,6 will be mapped to physical processor 0 and virtual processors indexed 1,3,5,7 will be mapped to physical processor 1.

```
> ./charmrun ./hello +p2 +vp8 +mapping RR_MAP
```

Block Mapping This mapping scheme maps virtual processors to physical processor in ranks, i.e. if there are 8 virtual processors and 2 physical processors then virtual processors indexed 0,1,2,3 will be mapped to physical processor 0 and virtual processors indexed 4,5,6,7 will be mapped to physical processor 1.

```
> ./charmrun ./hello +p2 +vp8 +mapping BLOCK_MAP
```

Proportional Mapping This scheme takes the processing capability of physical processors into account for mapping virtual processors to physical processors, i.e. if there are 2 processors running at different frequencies, then the number of virtual processors mapped to processors will be in proportion to their processing power. To make the load balancing framework aware of the heterogeneity of the system, the flag *+LBTestPESpeed* should also be used.

```
> ./charmrun ./hello +p2 +vp8 +mapping PROP_MAP
> ./charmrun ./hello +p2 +vp8 +mapping PROP_MAP +balancer GreedyLB +LBTestPESpeed
```

If you want to define your own mapping scheme, please contact us for assistance.

3.11 Performance Visualization

AMPI users can take advantage of Charm++'s tracing framework and associated performance visualization tool, Projections. Projections provides a number of different views of performance data that help users diagnose performance issues. Along with the traditional Timeline view, Projections also offers visualizations of load imbalance and communication-related data.

In order to generate tracing logs from an application to view in Projections, link with `ampicc -tracemode projections`.

AMPI defines the following extensions for tracing support:

```
AMPI_Trace_begin                    AMPI_Trace_end
AMPI_Trace_register_function_name   AMPI_Trace_register_function_id
AMPI_Trace_start_function_name      AMPI_Trace_start_function_id
AMPI_Trace_end_function_name        AMPI_Trace_end_function_id
```

When using the *Timeline* view in Projections, AMPI users can visualize what each VP on each processor is doing (what MPI method it is running or blocked in) by clicking the *View* tab and then selecting *Show Nested Bracketed User Events* from the drop down menu. See the Projections manual for information on performance analysis and visualization.

AMPI users can also use any tracing libraries or tools that rely on MPI's PMPI profiling interface, though such tools may not be aware of AMPI process virtualization.

3.12 Compiling AMPI Programs

AMPI provides a cross-platform compile-and-link script called *ampicc* to compile C, C++, and Fortran AMPI programs. This script resides in the `bin` subdirectory in the Charm++ installation directory. The main purpose of this script is to deal with the differences of various compiler names and command-line options across various machines on which AMPI runs. It is recommended that the AMPI compiler scripts be used to compile and link AMPI programs. One major advantage of using these is that one does not have to specify which libraries are to be linked for ensuring that C++ and Fortran90 codes are linked together correctly. Appropriate libraries required for linking such modules together are known to *ampicc* for various machines.

In spite of the platform-neutral syntax of *ampicc*, one may have to specify some platform-specific options for compiling and building AMPI codes. Fortunately, if *ampicc* does not recognize any particular options on its command line, it promptly passes it to all the individual compilers and linkers it invokes to compile the program. See the appendix for more details on building and running AMPI programs.

A Installing AMPI

AMPI is included in the source distribution of Charm++. To get the latest sources from PPL, visit: http://charm.cs.illinois.edu/software

and follow the download links. Then build Charm++ and AMPI from source.

The build script for Charm++ is called `build`. The syntax for this script is:

```
> build <target> <version> <opts>
```

For building AMPI (which also includes building Charm++ and other libraries needed by AMPI), specify `<target>` to be `AMPI`. And `<opts>` are command line options passed to the `charmc` compile script. Common compile time options such as `-g, -O, -Ipath, -Lpath, -llib` are accepted.

To build a debugging version of AMPI, use the option: `-g`. To build a production version of AMPI, use the option: `--with-production`.

`<version>` depends on the machine, operating system, and the underlying communication library one wants to use for running AMPI programs. See the charm/README file for details on picking the proper version. Here is an example of how to build a debug version of AMPI in a linux and ethernet environment:

```
> build AMPI netlrts-linux-x86_64 -g
```

And the following is an example of how to build a production version of AMPI on a Cray XC system, with MPI-level error checking in AMPI turned off:

```
> build AMPI gni-crayxc --with-production --disable-ampi-error-checking
```

AMPI can also be built with support for shared memory on any communication layer by adding "smp" as an option after the build target. For example, on an Infiniband Linux cluster:

```
> build AMPI verbs-linux-x86_64 smp --with-production
```

AMPI ranks are implemented as user-level threads with a stack size default of 1MB. If the default is not correct for your program, you can specify a different default stack size (in bytes) at build time. The following build command illustrates this for an Intel Omni-Path system:

```
> build AMPI ofi-linux-x86_64 --with-production -DTCHARM_STACKSIZE_DEFAULT=16777216
```

The same can be done for AMPI's RDMA messaging threshold using `AMPI_RDMA_THRESHOLD_DEFAULT` and, for messages sent within the same address space (ranks on the same worker thread or ranks on different worker threads in the same process in SMP builds), using `AMPI_SMP_RDMA_THRESHOLD_DEFAULT`. Contiguous messages with sizes larger than the threshold are sent via RDMA on communication layers that support this capability. You can also set the environment variables `AMPI_RDMA_THRESHOLD` and `AMPI_SMP_RDMA_THRESHOLD` before running a job to override the default specified at build time.

B Building and Running AMPI Programs

B.1 Building

AMPI provides a compiler called *ampicc* in your charm/bin/ directory. You can use this compiler to build your AMPI program the same way as other compilers like cc. All the command line flags that you would use for other compilers can be used with the AMPI compilers the same way. For example:

```
> ampicc -c pgm.c -O3
> ampif90 -c pgm.f90 -O0 -g
> ampicc -o pgm pgm.o -lm -O3
```

To use Isomalloc for transparently migrating user heap data, link with *-memory isomalloc*. To use a Charm++ load balancer, link a strategy or a suite of strategies in with *-module <LB>*. For example:

```
> ampicc pgm.c -o pgm -O3 -memory isomalloc -module CommonLBs
```

B.2 Running

The Charm++ distribution contains a script called `charmrun` that makes the job of running AMPI programs portable and easier across all parallel machines supported by Charm++. `charmrun` is copied to a directory where an AMPI program is built using *ampicc*. It takes a command line parameter specifying number of processors, and the name of the program followed by AMPI options (such as number of ranks to create, and the stack size of every user-level thread) and the program arguments. A typical invocation of an AMPI program `pgm` with `charmrun` is:

```
> charmrun +p16 ./pgm +vp64
```

Here, the AMPI program `pgm` is run on 16 physical processors with 64 total virtual ranks (which will be mapped 4 per processor initially).

To run with load balancing, specify a load balancing strategy. If Address Space Layout Randomization is enabled on your target system, you may need to add the flag *+isomalloc_sync* when running with migration. You can also specify the size of user-level thread's stack using the *+tcharm_stacksize* option, which can be used to decrease the size of the stack that must be migrated, as in the following example:

```
> charmrun +p16 ./pgm +vp128 +tcharm_stacksize 32K +balancer RefineLB
```

Note that for AMPI programs compiled with gfortran, users may need to set the following environment variable to see program output to stdout:

```
> export GFORTRAN_UNBUFFERED_ALL=1
```