

Parallel Programming Laboratory
University of Illinois at Urbana-Champaign

Adaptive MPI Manual

AMPI has been developed by Milind Bhandarkar with inputs from Gengbin Zheng and Orion Lawlor. The derived data types (DDT) library, which AMPI uses for the derived data types support, has been developed by Neelam Saboo. The current version of AMPI is maintained by Chao Huang.

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.uiuc.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@cs.uiuc.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.uiuc.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@cs.uiuc.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@cs.uiuc.edu for a copy).

UNDERSTOOD AND AGREED.

Contact Information:

The best contact path for licensing issues is by e-mail to kale@cs.uiuc.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) 333-3501

Contents

1	Introduction	3
1.1	Our Philosophy	3
1.2	Terminology	3
2	Charm++	4
3	AMPI	4
3.1	AMPI Status	4
3.2	Name for Main Program	5
3.2.1	Fortran	5
3.2.2	C or C++	5
3.3	Global Variable Privatization	5
3.3.1	Automatic Globals Swapping	6
3.3.2	Manual Change	6
3.4	Extensions for Migrations	8
3.4.1	Registering Chunk data	8
3.4.2	Migration	9
3.4.3	Packing/Unpacking Thread Data	9
3.5	Extensions for Checkpointing	12
3.6	Extensions for Memory Efficiency	12
3.7	Extensions for Interoperability	12
3.8	Extensions for Sequential Re-run of a Parallel Node	14
3.9	Communication Optimizations for AMPI	14
3.10	User Defined Initial Mapping	15
3.11	Compiling AMPI Programs	15
A	Installing AMPI	16
B	Building and Running AMPI Programs	16
B.1	Building	16
B.2	Running	16

1 Introduction

This manual describes Adaptive MPI (AMPI), which is an implementation of a significant subset¹ of MPI-1.1 Standard over CHARM++. CHARM++ is a C++-based parallel programming library developed by Prof. L. V. Kalé and his students over the last 10 years at University of Illinois.

We first describe our philosophy behind this work (why we do what we do). Later we give a brief introduction to CHARM++ and rationale for AMPI (tools of the trade). We describe AMPI in detail. Finally we summarize the changes required for original MPI codes to get them working with AMPI (current state of our work). Appendices contain the gory details of installing AMPI, building and running AMPI programs.

1.1 Our Philosophy

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 such as the solid propellant rocket simulation application. Our philosophy is to lessen the burden of the application developers by providing advanced programming paradigms and versatile runtime systems that can handle many common performance concerns automatically and let the application programmers focus on the actual application content.

One such concern is that of load imbalance. 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. Centralized load balancing strategy built into an application is impractical since each individual modules are developed almost independently by various developers. Thus, the runtime system support for load balancing becomes even more critical.

Automatic load balancing is infeasible for a program about which nothing is known. Other approaches to automatic load balancing therefore require the applications to provide hints about the load to the runtime system, or restrict load balance to a certain kind of algorithms such as Adaptive Mesh Refinement or to certain architectures such as shared memory machines. 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 exemplifies 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.

1.2 Terminology

Module A module refers to either a complete program or a library with an orchestrator subroutine². An orchestrator subroutine specifies the main control flow of the module by calling various subroutines from the associated library and does not usually have much state associated with it.

Thread A thread is a lightweight process that owns a stack and machine registers including program counter, but shares code and data with other threads within the same address space. If the underlying operating system recognizes a thread, it is known as kernel thread, otherwise it is known as user-thread. A context-switch between threads refers to suspending one thread’s execution and transferring control to another thread. Kernel threads typically have higher context switching costs than user-threads because of operating system overheads. The policy implemented by the underlying system

¹Currently, 110 MPI-1.1 Standard functions have been implemented.

²Like many software engineering terms, this term is overused, and unfortunately clashes with Fortran 90 module that denotes a program unit. We specifically refer to the later as “Fortran 90 module” to avoid confusion.

for transferring control between threads is known as thread scheduling policy. Scheduling policy for kernel threads is determined by the operating system, and is often more inflexible than user-threads. Scheduling policy is said to be non-preemptive if a context-switch occurs only when the currently running thread willingly asks to be suspended, otherwise it is said to be preemptive. AMPI threads are non-preemptive user-level threads.

Chunk A chunk is a combination of a user-level thread and the data it manipulates. When a program is converted from MPI to AMPI, we convert an MPI process into a chunk. This conversion is referred to as chunkification.

Object An object is just a blob of memory on which certain computations can be performed. The memory is referred to as an object's state, and the set of computations that can be performed on the object is called the interface of the object.

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 perform 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. It is this feature of CHARM++ that we utilize for AMPI.

Since many CSE applications are originally written using MPI, one would have to do a complete rewrite if they were to be converted to CHARM++ to take advantage of dynamic load balancing. This is indeed impractical. However, CONVERSE – the runtime system of CHARM++ – came to our rescue here, since it supports interoperability between different parallel programming paradigms such as parallel objects and threads. Using this feature, we developed AMPI, an implementation of a significant subset of MPI-1.1 standard over CHARM++. AMPI is described in the next section.

3 AMPI

AMPI utilizes the dynamic load balancing 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 computation loads of each thread as well as 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 Status

Currently all the MPI-1.1 Standard functions are supported in AMPI, with a collection of our extensions explained in detail in this manual. One-sided communication calls in MPI-2 are implemented, but they

are not taking advantage of RMA features yet. Also ROMIO³ has been integrated to support parallel I/O features. Link with `-lmpiromio` to take advantage of this library.

Following MPI-1.1 basic datatypes are supported in AMPI. (Some are not available in Fortran binding. Refer to MPI-1.1 Standard for details.)

MPI_DATATYPE_NULL	MPI_BYTE	MPI_UNSIGNED_LONG	MPI_LONG_DOUBLE_INT
MPI_DOUBLE	MPI_PACKED	MPI_LONG_DOUBLE	MPI_2FLOAT
MPI_INT	MPI_SHORT	MPI_FLOAT_INT	MPI_2DOUBLE
MPI_FLOAT	MPI_LONG	MPI_DOUBLE_INT	MPI_LB
MPI_COMPLEX	MPI_UNSIGNED_CHAR	MPI_LONG_INT	MPI_UB
MPI_LOGICAL	MPI_UNSIGNED_SHORT	MPI_2INT	
MPI_CHAR	MPI_UNSIGNED	MPI_SHORT_INT	

Following MPI-1.1 reduction operations are supported in AMPI.

MPI_MAX	MPI_MIN	MPI_SUM	MPI_PROD	MPI_MAXLOC	MPI_MINLOC
MPI_LAND	MPI_LOR	MPI_LXOR	MPI_BAND	MPI_BOR	MPI_BXOR

Following are AMPI extension calls, which will be explained in detail in this manual.

MPI_Migrate	MPI_Checkpoint	MPI_Restart	MPI_Register	MPI_Get_userdata
MPI_Ialltoall	MPI_Iallgather	MPI_Iallreduce	MPI_Ireduce	MPI_IGet

3.2 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:

3.2.1 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.

3.2.2 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.

3.3 Global Variable Privatization

For dynamic load balancing to be effective, one needs to map multiple user-level threads onto a 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. That’s where the need arises to make some transformations to the original MPI program in order to run correctly with AMPI.

The basic transformation needed to port the MPI program to AMPI is privatization of global variables.⁴ With the MPI process model, each MPI node can keep a copy of its own “permanent variables” – variables

³<http://www-unix.mcs.anl.gov/romio/>

⁴Typical 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.
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 possibility of a context switch between the definition and use of the variable. These variables need to be privatized.

that are accessible from more than one subroutines without passing them as arguments. Module variables, “saved” subroutine local variables, and common blocks in Fortran 90 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 are two choices: automatic global swapping and manual code modification.

3.3.1 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. The only thing that the user needs to do is to set flag `-swapglobals` at compile and link time. Currently this feature only works on x86 and x86_64 (i.e. amd64) platforms that fully support ELF. Thus it will not work on PPC or Itanium, or on some microkernels such as Catamount. When this feature does not work for you, you are advised to make the modification manually, which is detailed in the following section.

3.3.2 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 a 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 Fortran 90 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

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
END SUBROUTINE

```

```

    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;
}

```

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 (can be specified from the command line) and stacks do not grow dynamically.

3.4 Extensions for Migrations

For MPI chunks to migrate, we have added a few calls to AMPI. These include ability to register thread-specific data with the run-time system, to pack all the thread's data, and to express willingness to migrate.

3.4.1 Registering Chunk 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 chunks so as to balance the load. Then AMPI runtime has to pack up the chunk's state and move it to its new home processor. AMPI packs up any internal data in use by the chunk, including the thread's stack in use. This means that the local variables declared in subroutines in a chunk, 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 chunk. Thus upon starting execution, a chunk 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 `MPI_Register`. This function takes two arguments: A data item to be transported alongwith the chunk, and the pack subroutine, and returns an integer denoting the registration identifier. In C/C++ programs, it may be necessary to use this return value after migration completes and control returns to the chunk, using function `MPI_Get_userdata`. Therefore, the return value should be stored in a local variable.

3.4.2 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 chunk'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 chunk 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 `MPI_Migrate` for this purpose. Each chunk periodically calls `MPI_Migrate`. Typical CSE applications are iterative and perform multiple time-steps. One should call `MPI_Migrate` in each chunk at the end of some fixed number of timesteps. The frequency of `MPI_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 chunks. 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 `MPI_Migrate` does NOT tell the system to migrate the chunk, but merely tells the system to check the load balance after all the chunks call `MPI_Migrate`. To migrate the chunk or not is decided only by the system's load balancing strategy.

3.4.3 Packing/Unpacking Thread Data

Once the AMPI runtime system decides which chunks to send to which processors, it calls the specified pack subroutine for that chunk, with the chunk-specific data that was registered with the system using `MPI_Register`. This section explains how a subroutine should be written for performing pack/unpack.

There are three steps to transporting the chunk's data to other processor. First, the system calls a subroutine to get the size of the buffer required to pack the chunk'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 chunk'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 chunk is created (alongwith the thread) and a subroutine is called to unpack the chunk'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 chunk's pack-unpack-sizing subroutine, and this subroutine, when called in different phases performs different tasks. An example will make this clear:

Suppose the chunk data is defined as a user-defined type in Fortran 90:

```
!FORTRAN EXAMPLE
MODULE chunkmod
  TYPE, PUBLIC :: chunk
    INTEGER , parameter :: nx=4, ny=4, tchunks=16
    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 Fortran 90. 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 the chunk is packed, these allocated data structures should be deallocated after copying them to buffers, and when the 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. 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(asize))
    ALLOCATE(c%yarr(asize))
  ENDIF

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

  IF (fpup_isdeleting(p)) THEN      !! if invocation is for packing
    DEALLOCATE(c%xarr(asize))
    DEALLOCATE(c%yarr(asize))
  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.ysize);
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.

3.5 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.

A subroutine for checkpoint purpose has been added to AMPI: `void MPI.Checkpoint(char *dirname);` This subroutine takes a directory name as its argument. It is a collective function, meaning every virtual processor in the program needs to call this subroutine and specify the same directory name. (Typically, in an iterative AMPI 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 by specifying `"+restart dirname"` on the command-line. This capability is powered by the CHARM++ runtime system. For more information about CHARM++ checkpoint/restart mechanism please refer to CHARM++ manual.

3.6 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 `MPI_IGet`. `MPI_IGet` preserves the similar semantics as `MPI_Get` except that no user buffer is provided to hold incoming data. `MPI_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. `MPI_IGet_Free` lets the runtime system free the resources being used for this get request including the data buffer. And `MPI_IGet_Data` is the utility program that returns the actual data.

```

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

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

char* MPI_IGet_Data(MPI_Status status);

```

3.7 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 chunks 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. This involves the usual “chunkification” transformation so that multiple chunks from the application can run on the same processor without overwriting each other’s data. This also involves making the main program into a subroutine and naming it `MPI_Main`.

Thus now, we have two `MPI_Mains`, 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_Mains` as `Fluids_Main` and `Solids_Main`⁵ writing a subroutine called `MPI_Setup`.

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

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

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

The number of chunks to create for each chunk type 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 chunks (`+vp` option). In the above case, suppose one wants to create 128 chunks of Solids and 64 chunks 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 chunk types representing different complete applications can co-exist within the same executable. They can also continue to communicate among their own chunk-types 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_WORLDS`”; 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_WORLDS` 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)`.

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

```
!FORTRAN EXAMPLE
```

⁵Currently, 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.

```

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 chunk 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);

```

3.8 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 “AMPIMS-GLOG” defined, like the following command in charm directory. (Linking with zlib “-lz” might be required with this, for generating compressed log file.)

```
> ./build AMPI net-linux -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 processors and 4 VPs will be executed, and the changes in the MPI environment of processor 2 (also VP 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 comamnd line options are required as well.

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

3.9 Communication Optimizations for AMPI

AMPI is powered by the CHARM++ communication optimization support now! Currently the user needs to specify the communication pattern by command line option. In the future this can be done automatically by the system.

Currently there are four strategies available: USE_DIRECT, USE_MESH, USE_HYPERCUBE and USE_GRID. USE_DIRECT sends the message directly. USE_MESH imposes a 2d Mesh virtual topology on the processors so each processor sends messages to its neighbors in its row and column of the mesh which forward the messages to their correct destinations. USE_HYPERCUBE and USE_GRID impose a hypercube and a 3d Grid topologies on the processors. USE_HYPERCUBE will do best for very small messages and small number of processors, 3d has better performance for slightly higher message sizes and then Mesh starts performing

best. The programmer is encouraged to try out all the strategies. (Stolen from the CommLib manual by Sameer :)

For more details please refer to the CommLib paper ⁶.

Specifying the strategy is as simple as a command line option `+strategy`. For example:

```
> ./charmrun +p64 alltoall +vp64 1000 100 +strategy USE_MESH
```

tells the system to use MESH strategy for CommLib. By default USE_DIRECT is used.

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. 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 chunks, 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 with different processing power, then number of virtual processors mapped to processors will be in proportion to their processing power.

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

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

3.11 Compiling AMPI Programs

CHARM++ provides a cross-platform compile-and-link script called `charmcc` to compile C, C++, Fortran, CHARM++ and 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 CHARM++ runs. While, `charmcc` handles C and C++ compiler differences most of the time, the support for Fortran 90 is new, and may have bugs. But CHARM++ developers are aware of this problem and are working to fix them. Even in its alpha stage of Fortran 90 support, `charmcc` still handles many of the compiler differences across many machines, and it is recommended that `charmcc` be used to compile and linking AMPI programs. One major advantage of using `charmcc` is that one does not have to specify which libraries are to be linked for ensuring that C++ and Fortran 90 codes are linked correctly together. Appropriate libraries required for linking such modules together are known to `charmcc` for various machines.

In spite of the platform-neutral syntax of `charmcc`, one may have to specify some platform-specific options for compiling and building AMPI codes. Fortunately, if `charmcc` 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.

⁶L. V. Kale and Sameer Kumar and Krishnan Vardarajan, 2002. <http://finesse.cs.uiuc.edu/papers/CommLib.pdf>

A Installing AMPI

AMPI is included in the source distribution of CHARM++. To get the latest sources from PPL, visit: <http://charm.cs.uiuc.edu/>

and follow the download link. Now one has to 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 `charmcc` 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 options: `“-O -DCMK_OPTIMIZE=1”`.

`<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. Following is an example of how to build AMPI under linux and ethernet environment, with debugging info produced:

```
> build AMPI net-linux -g
```

B Building and Running AMPI Programs

B.1 Building

CHARM++ provides a compiler called `charmcc` in your `charm/bin/` directory. You can use this compiler to build your AMPI program the same way as other compilers like `cc`. Especially, to build an AMPI program, a command line option `-language mpi` should be applied. All the command line flags that you would use for other compilers can be used with `charmcc` the same way. For example:

```
> charmcc -language mpi -c pgm.c -O3
> charmcc -language mpi -o pgm pgm.o -lm -O3
```

Shortcuts to the AMPI compiler are provided. If you have added `charm/bin` into your `$PATH` environment variable, simply type `mpicc`, `mpiCC`, `mpif77`, and `mpif90` as provided by other MPI implementations.

```
> mpicc -c pgm.c -g
```

B.2 Running

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 `charmcc`. It takes a command line parameter specifying number of processors, and the name of the program followed by AMPI options (such as number of chunks to create, and the stack size of every chunk) and the program arguments. A typical invocation of AMPI program `pgm` with `charmrun` is:

```
> charmrun pgm +p16 +vp32 +tcharm_stacksize 3276800
```

Here, the AMPI program `pgm` is run on 16 physical processors with 32 chunks (which will be mapped 2 per processor initially), where each user-level thread associated with a chunk has the stack size of 3,276,800 bytes.