Parallel Programming Laboratory
University of Illinois at Urbana-Champaign

Charm++
Frequently Asked Questions

Version 6.8.0

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

For answers to questions not on this list, please contact us at charm AT cs.illinois.edu

Contents

1 Big Questions

1.1 What is Charm++?

Charm++ is a runtime library to let C++ objects communicate with each other efficiently. The programming model is thus an asynchronous message driven paradigm, like Java RMI, or RPC; but it is targeted towards tightly coupled, high-performance parallel machines. Unlike MPI's "single program, multiple data" (SPMD) programming and execution model, Charm++ programs do not proceed in lockstep. The flow of control is determined by the order in which remote method invocations occur. This can be controlled by the user through Structure Control Flow using Structure Dagger, or Charisma, or compiler supported higher level languages Charj

Charm++ has demonstrated scalability up to hundreds of thousands of processors, and provides extremely advanced load balancing and object migration facilities.

1.2 Can Charm++ parallelize my serial program automatically?

No.

Charm++ is used to write *explicitly parallel* programs–we don't have our own compiler, so we don't do automatic parallelization. We've found automatic parallelization useful only for a small range of very regular numerical applications.

However, you should *not* have to throw away your serial code; normally only a small fraction of a large program needs to be changed to enable parallel execution. In particular, Charm++'s support for object-oriented programming and high-level abstractions such as Charm++ Arrays make it simpler and more expressive than many other parallel languages. So you will have to write some new code, but not as much as you might think. This is particularly true when using one of the Charm++ frameworks.

1.3 I can already write parallel applications in MPI. Why should I use Charm++?

Charm++ provides several extremely sophisticated features, such as application-independent object migration, fault tolerance, power awareness, and automatic overlap of communication with computation, that are very difficult to provide in MPI. If you have a working MPI code but have scalability problems because of dynamic behavior, load imbalance, or communication costs, Charm++ might dramatically improve your performance. You can even run your MPI code on Charm++ unchanged using AMPI.

1.4 Will Charm++ run on my machine?

Yes.

Charm++ supports both shared-memory and distributed-memory machines, SMPs and non-SMPs. In particular, we support serial machines, Windows machines, Apple machines, ARM machines, clusters connected via Ethernet, or Infiniband, IBM Power series and BlueGene/Q, Cray XC/XE/XK series, and any machine that supports MPI. We normally do our development on Linux workstations, and our testing on large parallel machines. Programs written using Charm++ will run on any supported machine.

1.5 Does anybody actually use Charm++?

Several large applications use Charm++.

The large, production-quality molecular dynamics application NAMD.
The cosmological simulator ChaNGa.

The atomistic simulation framework OpenAtom.

We have significant collaborations with groups in Materials Science, Chemistry, Astrophysics, Network Simulation, Operation Research, Contagion Effects, in Illinois, New York, California, Washington, and Virginia. See also Applications for a more complete list.

1.6 Who created Charm++?

Prof. L.V. Kale, of the Computer Science Department of the University of Illinois at Urbana-Champaign, and his research group, the Parallel Programming Lab. Nearly a hundred people have contributed something to the project over the course of approximately 20 years; a partial list of contributors appears in the people's page.

1.7 What is the future of Charm++?

Our research group of approximately twenty people are actively engaged in maintaining and extending Charm++; and in particular the Charm++ frameworks. Several other groups are dependent on Charm++, so we expect to continue improving Charm++ indefinitely.

1.8 How is Charm++ Licensed?

Charm++ is open-source and free for research, educational, and academic use. The University of Illinois retains the copyright to the software, and requires a license for any commercial redistribution of our software. The actual, legal license is included with Charm++ (in charm/LICENSE). Contact Charmworks, Inc. for commercial support and licensing of Charm++ and AMPI.

1.9 I have a suggestion/feature request/bug report. Who should I send it to?

Our mailing list is charm AT cs.illinois.edu. We're always glad to get feedback on our software.

2 Installation and Usage

2.1 How do I get Charm++?

See our download page.

2.2 Should I use the GIT version of Charm++?

The developers of Charm++ routinely use the latest GIT versions, and most of the time this is the best case. Occasionally something breaks, but the GIT version will likely contain bug fixes not found in the releases.

2.3 How do I compile Charm++?

Run the interactive build script `./build` with no extra arguments If this fails, email charm AT cs.illinois.edu with the problem. Include the build line used (this is saved automatically in `smart-build.log`)

If you have a very unusual machine configuration, you will have to run `./build --help` to list all possible build options. You will then choose the closest architecture, and then you may have to modify the associated conf-mach.sh and conv-mach.h files in src/arch to point to your desired compilers and options. If you develop a significantly different platform, send the modified files to charm AT cs.illinois.edu so we can include it in the distribution.

2.4 How do I compile AMPI?

Run the build script `./build` and choose the option for building "Charm++ and AMPI," or just replace "charm++" in your full build command with "AMPI", as in `./build AMPI netlrts-linux-x86_64`.

2.5 Can I remove part of charm tree after compilation to free disk space?

Yes. Keep src, bin, lib, lib_so, include, tmp. You will not need tests, examples, doc, contrib for normal usage once you have verified that your build is functional.

2.6 If the interactive script fails, how do I compile Charm++?

See Appendix V of the Charm manual for Installation and Usage.

2.7 How do I specify the processors I want to use?

On machines where MPI has already been wired into the job system, use the –mpiexec flag and -np arguments.

For the netlrts- versions, you need to write a nodelist file which lists all the machine hostnames available for parallel runs.

```
group main
  host foo1
  host foo2 ++cpus 4
  host foo3.bar.edu
```

For the MPI version, you need to set up an MPI configuration for available machines as for normal MPI applications.

You can specify the exact cores to use on each node using the `+pemap` option. When running in SMP or multicore mode, this applies to the worker threads only, not communication threads. To specify the placement of communication threads, use the `+commap` option. For example, to place 8 threads on 2 nodes (16 threads total) with the comm thread on core 0 and the worker threads on cores 1 - 7, you would use the following command:

```
./charmrun +p14 ./pgm ++ppn 7 +commap 0 +pemap 1-7
```

See section C.2.2 SMP and Multicore Options of the Charm++ manual for more information.

2.8 How do I use *ssh* instead of the deprecated *rsh*?

You need to set up your `.ssh/authorized_keys` file correctly. Setup no-password logins using ssh by putting the correct host key (ssh-keygen) in the file `.ssh/authorized_keys`.

Finally, in the `.nodelist` file, you specify the shell to use for remote execution of a program using the keyword *++shell*.

```
group main ++shell ssh
  host foo1
  host foo2
  host foo3
```

2.9 Can I use the serial library X from a Charm program?

Yes. Some of the known working serial libraries include:

- The Tcl/Tk interpreter (in NAMD)
- The Python interpreter (in Cosmo prototype)
- OpenGL graphics (in graphics demos)

- Metis mesh partitioning (included with charm)
- ATLAS, BLAS, LAPACK, ESSL, FFTW, MASSV, ACML, MKL, BOOST

In general, any serial library should work fine with Charm++.

2.10 How do I get the command-line switches available for a specific program?

Try

```
./charmrun ./pgm --help
```

to see a list of parameters at the command line. The charmrun arguments are documented in the Installation and Usage Manual the arguments for the installed libraries are listed in the library manuals.

2.11 What should I do if my program hangs while gathering CPU topology information at startup?

This is an indication that your cluster's DNS server is not responding properly. Ideally, the DNS resolver configured to serve your cluster nodes should be able to rapidly map their hostnames to their IP addresses. As an immediate workaround, you can run your program with the `+skip_cpu_topology` flag, at the possible cost of reduced performance. Another workaround is installing and running `nscd`, the "name service caching daemon", on your cluster nodes; this may add some noise on your systems and hence reduce performance. A third workaround is adding the addresses and names of all cluster nodes in each node's `/etc/hosts` file; this poses maintainability problems for ongoing system administration.

3 Basic Charm++ Programming

3.1 What's the basic programming model for Charm++?

Parallel objects using "Asynchronous Remote Method Invocation":

Asynchronous in that you *do not block* until the method returns–the caller continues immediately.

Remote in that the two objects may be separated by a network.

Method Invocation in that it's just C++ classes calling each other's methods.

3.2 What is an "entry method"?

Entry methods are all the methods of a chare where messages can be sent by other chares. They are declared in the .ci files, and they must be defined as public methods of the C++ object representing the chare.

3.3 When I invoke a remote method, do I block until that method returns?

No! This is one of the biggest differences between Charm++ and most other "remote procedure call" systems like, Java RMI, or RPC. "Invoke an asynchronous method" and "send a message" have exactly the same semantics and implementation. Since the invoking method does not wait for the remote method to terminate, it normally cannot receive any return value. (see later for a way to return values)

3.4 Why does Charm++ use asynchronous methods?

Asynchronous method invocation is more efficient because it can be implemented as a single message send. Unlike with synchronous methods, thread blocking and unblocking and a return message are not needed.

Another big advantage of asynchronous methods is that it's easy to make things run in parallel. If I execute:

```
a->foo();
b->bar();
```

Now foo and bar can run at the same time; there's no reason bar has to wait for foo.

3.5 Can I make a method synchronous? Can I then return a value?

Yes. If you want synchronous methods, so the caller will block, use the `[sync]` keyword before the method in the .ci file. This requires the sender to be a threaded entry method, as it will be suspended until the callee finishes. Sync entry methods are allowed to return values to the caller.

3.6 What is a threaded entry method? How does one make an entry method threaded?

A threaded entry method is an entry method for a chare that executes in a separate user-level thread. It is useful when the entry method wants to suspend itself (for example, to wait for more data). Note that threaded entry methods have nothing to do with kernel-level threads or pthreads; they run in user-level threads that are scheduled by Charm++ itself.

In order to make an entry method threaded, one should add the keyword *threaded* withing square brackets after the *entry* keyword in the interface file:

```
module M {
  chare X {
    entry [threaded] E1(void);
  };
};
```

3.7 If I don't want to use threads, how can an asynchronous method return a value?

The usual way to get data back to your caller is via another invocation in the opposite direction:

```
void A::start(void) {
  b->giveMeSomeData();
}
void B::giveMeSomeData(void) {
  a->hereIsTheData(data);
}
void A::hereIsTheData(myclass_t data) {
  ...use data somehow...
}
```

This is contorted, but it exactly matches what the machine has to do. The difficulty of accessing remote data encourages programmers to use local data, bundle outgoing requests, and develop higher-level abstractions, which leads to good performance and good code.

3.8 Isn't there a better way to send data back to whoever called me?

The above example is very non-modular, because *b* has to know that *a* called it, and what method to call a back on. For this kind of request/response code, you can abstract away the "where to return the data" with a *CkCallback* object:

```
void A::start(void) {
  b->giveMeSomeData(CkCallback(CkIndex_A::hereIsTheData,thisProxy));
}
void B::giveMeSomeData(CkCallback returnDataHere) {
  returnDataHere.send(data);
}
void A::hereIsTheData(myclass_t data) {
  ...use data somehow...
}
```

Now *b* can be called from several different places in *a*, or from several different modules.

3.9 Why should I prefer the callback way to return data rather than using `[sync]` entry methods?

There are a few reasons for that:

- The caller needs to be threaded, which implies some overhead in creating the thread. Moreover the threaded entry method will suspend waiting for the data, preventing any code after the remote method invocation to proceed in parallel.
- Threaded entry methods are still methods of an object. While they are suspended other entry methods for the same object (or even the same threaded entry method) can be called. This allows for potential problems if the suspending method does leave some objects in an inconsistent state.
- Finally, and probably most important, `[sync]` entry methods can only be used to return a value that can be computed by a single chare. When more flexibility is needed, such in cases where the resulting value needs to the contribution of multiple objects, the callback methodology is the only one available. The caller could for example send a broadcast to a chare array, which will use a reduction to collect back the results after they have been computed.

3.10 How does the initialization in Charm work?

Each processor executes the following operations strictly in order:

1. All methods registered as *initnode*;
2. All methods registered as *initproc*;
3. On processor zero, all *mainchares* constructor method is invoked (the ones taking a `CkArgMsg*`);
4. The read-onlies are propagated from processor zero to all other processors;
5. The nodegroups are created;
6. The groups are created. During this phase, for all the chare arrays have been created with a block allocation, the corresponding array elements are instantiated;
7. Initialization terminated and all messages are available for processing, including the messages responsible for the instantiation of array elements manually inserted.

This implies that you can assume that the previous steps has completely finished before the next one starts, and any side effect from all the previous steps are committed (and can therefore be used).

Inside a single step there is no order guarantee. This implies that, for example, two groups allocated from mainchare can be instantiated in any order. The only exception to this is processor zero, where chare objects are instantiated immediately when allocated in the mainchare, i.e if two groups are allocated, their order is fixed by the allocation order in the mainchare constructing them. Again, this is only valid for processor zero, and in no other processor this assumption should be made.

To notice that if array elements are allocated in block (by specifying the number of elements at the end of the `ckNew` function), they are all instantiated before normal execution is resumed; if manual insertion is used, each element can be constructed at any time on its home processor, and not necessarily before other regular communication messages have been delivered to other chares (including other array elements part of the same array).

3.11 Does Charm++ support C and Fortran?

C and Fortran routines can be called from Charm++ using the usual API conventions for accessing them from C++. AMPI supports Fortran directly, but direct use of Charm++ semantics from Fortran is at an immature stage, contact us charm AT cs.illinois.edu if you are interested in pursuing this further.

3.12 What is a proxy?

A proxy is a local C++ class that represents a remote C++ class. When you invoke a method on a proxy, it sends the request across the network to the real object it represents. In Charm++, all communication is done using proxies.

A proxy class for each of your classes is generated based on the methods you list in the .ci file.

3.13 What are the different ways one can create proxies?

Proxies can be:

- Created using ckNew. This is the only method that actually creates a new parallel object. "CProxy_A::ckNew(...)" returns a proxy, as described in the manual.
- Copied from an existing proxy. This happens when you assign two proxies or send a proxy in a message.
- Created from a "handle". This happens when you say "CProxy_A p=thishandle;"
- Created uninitialized. This is the default when you say "CProxy_A p;". You'll get a runtime error "proxy has not been initialized" if you try to use an uninitialized proxy.

3.14 What is wrong if I do `A *ap = new CProxy_A(handle)`?

This will not compile, because a *CProxy_A* is not an *A*. What you want is *CProxy_A *ap = new CProxy_A(handle)*.

3.15 Why is the *def.h* usually included at the end? Is it necessary or can I just include it at the beginning?

You can include the *def.h* file once you've actually declared everything it will reference– all your chares and readonly variables. If your chares and readonlies are in your own header files, it is legal to include the *def.h* right away.

However, if the class declaration for a chare isn't visible when you include the *def.h* file, you'll get a confusing compiler error. This is why we recommend including the *def.h* file at the end.

3.16 How can I use a global variable across different processors?

Make the global variable "readonly" by declaring it in the .ci file. Remember also that read-onlies can be safely set only in the mainchare constructor. Any change after the mainchare constructor has finished will be local to the processor that made the change. To change a global variable later in the program, every processor must modify it accordingly (e.g by using a chare group. Note that chare arrays are not guaranteed to cover all processors)

3.17 Can I have a class static read-only variable?

One can have class-static variables as read-onlies. Inside a chare, group or array declaration in the *.ci* file, one can have a readonly variable declaration. Thus:

```
chare someChare {
  ...
  readonly CkGroupID someGroup;
  ...
};
```

is fine. In the *.h* declaration for *class someChare*, you will have to put *someGroup* as a public static variable, and you are done.

You then refer to the variable in your program as *someChare::someGroup*.

3.18 How do I measure the time taken by a program or operation?

You can use `CkWallTimer()` to determine the time on some particular processor. To time some parallel computation, you need to call CkWallTimer on some processor, do the parallel computation, then call CkWallTimer again on the same processor and subtract.

3.19 What do `CmiAssert` and `CkAssert` do?

These are just like the standard C++ *assert* calls in <*assert.h*>– they call abort if the condition passed to them is false.

We use our own version rather than the standard version because we have to call *CkAbort*, and because we can turn our asserts off when *–with-production* is used on the build line. These assertions are specifically controlled by *–enable-error-checking* or *–disable-error-checking*. The *–with-production* flag implies *–disable-error-checking*, but it can still be explicitly enabled with *–enable-error-checking*.

3.20 Can I know how many messages are being sent to a chare?

No.

There is no nice library to solve this problem, as some messages might be queued on the receiving processor, some on the sender, and some on the network. You can still:

- Send a return receipt message to the sender, and wait until all the receipts for the messages sent have arrived, then go to a barrier;
- Do all the sends, then wait for quiescence.

3.21 What is "quiescence"? How does it work?

Quiescence is When nothing is happening anywhere on the parallel machine.

A low-level background task counts sent and received messages. When, across the machine, all the messages that have been sent have been received, and nothing is being processed, quiescence is triggered.

3.22 Should I use quiescence detection?

Probably not.

See the Completion Detection section of the manual for instructions on a more local inactivity detection scheme.

In some ways, quiescence is a very strong property (it guarantees *nothing* is happening *anywhere*) so if some other library is doing something, you won't reach quiescence. In other ways, quiescence is a very weak property, since it doesn't guarantee anything about the state of your application like a reduction does, only that nothing is happening. Because quiescence detection is on the one hand so strong it breaks modularity, and on the other hand is too weak to guarantee anything useful, it's often better to use something else.

Often global properties can be replaced by much easier-to-compute local properties. For example, my object could wait until all *its* neighbors have sent it messages (a local property my object can easily detect by counting message arrivals), rather than waiting until *all* neighbor messages across the whole machine have been sent (a global property that's difficult to determine). Sometimes a simple reduction is needed instead of quiescence, which has the benefits of being activated explicitly (each element of a chare array or chare group has to call contribute) and allows some data to be collected at the same time. A reduction is also a few times faster than quiescence detection. Finally, there are a few situations, such as some tree-search problems, where quiescence detection is actually the most sensible, efficient solution.

4 Charm++ Arrays

4.1 How do I know which processor a chare array element is running on?

At any given instant, you can call `CkMyPe()` to find out where you are. There is no reliable way to tell where another array element is; even if you could find out at some instant, the element might immediately migrate somewhere else!

4.2 Should I use Charm++ Arrays in my program?

Yes! Most of your computation should happen inside array elements. Arrays are the main way to automatically balance the load using one of the load balancers available.

4.3 Is there a property similar to `thisIndex` containing the chare array's dimensions or total number of elements?

No. In more sophisticated Charm++ algorithms and programs, array dimensions are a dynamic property, and since Charm++ operates in a distributed system context, any such value would potentially be stale between access and use.

If the array in question has a fixed size, then that size can be passed to its elements as an argument to their constructor or some later entry method call. Otherwise, the object(s) creating the chare array elements should perform a reduction to count them.

4.4 How many array elements should I have per processor?

To do load balancing, you need more than one array element per processor. To keep the time and space overheads reasonable, you probably don't want more than a few thousand array elements per processor. The optimal value depends on the program, but is usually between 10 and 100. If you come from an MPI background, this may seem like a lot.

4.5 What does the term reduction refer to?

You can *reduce* a set of data to a single value. For example, finding the sum of values, where each array element contributes a value to the final sum. Reductions are supported directly by Charm++ arrays, and

some operations most commonly used are predefined. Other more complicated reductions can implement if needed.

4.6 Can I do multiple reductions on an array?

You *can* have several reductions happen one after another; but you *cannot* mix up the execution of two reductions over the same array. That is, if you want to reduce A, then B, every array element has to contribute to A, then contribute to B; you cannot have some elements contribute to B, then contribute to A.

In addition, *Tuple* reductions provide a way of performing multiple different reductions using the same reduction message. See the Built-in Reduction Types section of the manual for more information on Tuple reductions.

4.7 Does Charm++ do automatic load balancing without the user asking for it?

No. You only get load balancing if you explicitly ask for it by linking in one or more load balancers with *-balancer* link-time option. If you link in more than one load balancer, you can select from the available load balancers at runtime with the *+balancer* option. In addition, you can use Metabalancer with the *+MetaLB* option to automatically decide when to invoke the load balancer, as described in Load Balancing Strategies section.

4.8 What is the migration constructor and why do I need it?

The migration constructor (a constructor that takes `CkMigrateMessage *` as parameter) is invoked when an array element migrates to a new processor, or when chares or group instances are restored from a checkpoint. If there is anything you want to do when you migrate, you could put it here.

A migration constructor need not be defined for any given chare type. If you try to migrate instances of a chare type that lacks a migration constructor, the runtime system will abort the program with an error message.

The migration constructor should not be declared in the *.ci* file. Of course the array element will require also at least one regular constructor so that it can be created, and these must be declared in the *.ci* file.

4.9 What happens to the old copy of an array element after it migrates?

After sizing and packing a migrating array element, the array manager `delete`s the old copy. As long as all the array element destructors in the non-leaf nodes of your inheritance hierarchy are *virtual destructors*, with declaration syntax:

```
class foo : ... {
  ...
  virtual ~foo(); // <- virtual destructor
};
```

then everything will get deleted properly.
Note that deleting things in a packing pup happens to work for the current array manager, but *WILL NOT* work for checkpointing, debugging, or any of the (many) other uses for packing puppers we might dream up - so DON'T DO IT!

4.10 Is it possible to turn migratability on and off for an individual array element?

Yes, call *setMigratable(false);* in the constructor.

4.11 Is it possible to insist that a particular array element gets migrated at the next *AtSync()*?

No, but a manual migration can be triggered using *migrateMe*.

4.12 When not using `AtSync` for LB, when does the LB start up? Where is the code that periodically checks if load balancing can be done?

If not using `usesAtSync`, the load balancer can start up at anytime. There is a dummy `AtSync` for each array element which by default tells the load balancer that it is always ready. The LDBD manager has a syncer (`LBDB::batsyncer`) which periodically calls `AtSync` roughly every 1ms to trigger the load balancing (this timeout can be changed with the *+LBPeriod* option). In this load balancing mode, users have to make sure all migratable objects are always ready to migrate (e.g. not depending on a global variable which cannot be migrated).

4.13 Should I use AtSync explicitly, or leave it to the system?

You almost certainly want to use AtSync directly. In most cases there are points in the execution where the memory in use by a chare is bigger due to transitory data, which does not need to be transferred if the migration happens at predefined points.

5 Charm++ Groups and Nodegroups

5.1 What are groups and nodegroups used for?

They are used for optimizations at the processor and node level respectively.

5.2 Should I use groups?

Probably not. People with an MPI background often overuse groups, which results in MPI-like Charm++ programs. Arrays should generally be used instead, because arrays can be migrated to achieve load balance.

Groups tend to be most useful in constructing communication optimization libraries. For example, all the array elements on a processor can contribute something to their local group, which can then send a combined message to another processor. This can be much more efficient than having each array element send a separate message.

5.3 Is it safe to use a local pointer to a group, such as from ckLocalBranch?

Yes. Groups never migrate, so a local pointer is safe. The only caveat is to make sure *you* don't migrate without updating the pointer.

A local pointer can be used for very efficient access to data held by a group.

5.4 What are migratable groups?

Migratable groups are declared so by adding the "[migratable]" attribute in the .ci file. They *cannot* migrate from one processor to another during normal execution, but only to disk for checkpointing purposes.

Migratable groups must declare a migration constructor (taking `CkMigrateMessage *` as a parameter) and a pup routine. The migration constructor *must* call the superclass migration constructor as in this example:

```
class MyGroup : public CBase_MyGroup {
  ...
  MyGroup (CkMigrateMessage *msg) : CBase_MyGroup(msg) { }
  ...
}
```

5.5 Should I use nodegroups?

Almost certainly not. You should use arrays for most computation, and even quite low-level communication optimizations are often best handled by groups. Nodegroups are very difficult to get right.

5.6 What's the difference between groups and nodegroups?

There's one group element per processor (CkNumPes() elements); and one nodegroup element per node (CkNumNodes() elements). Because they execute on a node, nodegroups have very different semantics from the rest of Charm++.

Note that on a non-SMP machine, groups and nodegroups are identical.

5.7 Do nodegroup entry methods execute on one fixed processor of the node, or on the next available processor?

Entries in node groups execute on the next available processor. Thus, if two messages were sent to a branch of a nodegroup, two processors could execute one each simultaneously.

5.8 Are nodegroups single-threaded?

No. They *can* be accessed by multiple threads at once.

5.9 Do we have to worry about two entry methods in an object executing simultaneously?

Yes, which makes nodegroups different from everything else in Charm++.

If a nodegroup method accesses a data structure in a non-threadsafe way (such as writing to it), you need to lock it, for example using a CmiNodeLock.

6 Charm++ Messages

6.1 What are messages?

A bundle of data sent, via a proxy, to another chare. A message is a special kind of heap-allocated C++ object.

6.2 Should I use messages?

It depends on the application. We've found parameter marshalling to be less confusing and error-prone than messages for small parameters. Nevertheless, messages can be more efficient, especially if you need to buffer incoming data, or send complicated data structures (like a portion of a tree).

6.3 What is the best way to pass pointers in a message?

You can't pass pointers across processors. This is a basic fact of life on distributed-memory machines.

You can, of course, pass a copy of an object referenced via a pointer across processors–either dereference the pointer before sending, or use a varsize message.

6.4 Can I allocate a message on the stack?

No. You must allocate messages with *new*.

6.5 Do I need to delete messages that are sent to me?

Yes, or you will leak memory! If you receive a message, you are responsible for deleting it. This is exactly opposite of parameter marshalling, and much common practice. The only exception are entry methods declared as [nokeep]; for these the system will free the message automatically at the end of the method.

6.6 Do I need to delete messages that I allocate and send?

No, this will certainly corrupt both the message and the heap! Once you've sent a message, it's not yours any more. This is again exactly the opposite of parameter marshalling.

6.7 What can a variable-length message contain?

Variable-length messages can contain arrays of any type, both primitive type or any user-defined type. The only restriction is that they have to be 1D arrays.

6.8 Do I need to delete the arrays in variable-length messages?

No, this will certainly corrupt the heap! These arrays are allocated in a single contiguous buffer together with the message itself, and is deleted when the message is deleted.

6.9 What are priorities?

Priorities are special values that can be associated with messages, so that the Charm++ scheduler will generally prefer higher priority messages when choosing a buffered message from the queue to invoke as an entry method. Priorities are often respected by Charm++ scheduler, but for correctness, a program must never rely upon any particular ordering of message deliveries. Messages with priorities are typically used to encourage high performance behavior of an application.

For integer priorities, the smaller the priority value, the higher the priority of the message. Negative value are therefore higher priority than positive ones. To enable and set a message's priority there is a special *new* syntax and *CkPriorityPtr* function; see the manual for details. If no priority is set, messages have a default priority of zero.

6.10 Can messages have multiple inheritance in Charm++?

Yes, but you probably shouldn't. Perhaps you want to consider using generic or meta programming techniques with templated chares, methods, and/or messages instead.

7 PUP Framework

7.1 How does one write a pup for a dynamically allocated 2-dimensional array?

The usual way: pup the size(s), allocate the array if unpacking, and then pup all the elements.

For example, if you have a 2D grid like this:

```
class foo {
 private:
  int wid,ht;
  double **grid;
  ...other data members

  //Utility allocation/deallocation routines
  void allocateGrid(void) {
    grid=new double*[ht];
    for (int y=0;y<ht;y++)
```

```
      grid[y]=new double[wid];
  }
  void freeGrid(void) {
    for (int y=0;y<ht;y++)
      delete[] grid[y];
    delete[] grid;
    grid=NULL;
  }

 public:
  //Regular constructor
  foo() {
    ...set wid, ht...
    allocateGrid();
  }
  //Migration constructor
  foo(CkMigrateMessage *) {}
  //Destructor
  ~foo() {
    freeGrid();
  }

  //pup method
  virtual void pup(PUP::er &p) {
    p(wid); p(ht);
    if (p.isUnpacking()) {
      //Now that we know wid and ht, allocate grid
      allocateGrid(wid,ht);
    }
    //Pup grid values element-by-element
    for (int y=0;y<ht;y++)
      PUParray(p, grid[y], wid);
    ...pup other data members...
  }
};
```

7.2 When using automatic allocation via PUP::able, what do these calls mean? `PUPable_def(parent); PUPable_def(child);`

For the automatic allocation described in *Automatic allocation via* `PUP::able` of the manual, each class needs four things:

- A migration constructor
- `PUPable_decl(className)` in the class declaration in the *.h* file
- `PUPable_def(className)` at file scope in the *.C* file
- `PUPable_reg(className)` called exactly once on every node. You typically use the *initproc* mechanism to call these.

See `charm/tests/charm++/megatest/marshall.[hC]` for an executable example.

7.3 What is the difference between `p|data;` and `p(data);`? Which one should I use?

For most system- and user-defined structure *someHandle*, you want `p|someHandle;` instead of `p(someHandle);`

The reason for the two incompatible syntax varieties is that the bar operator can be overloaded *outside* `pup.h` (just like the `std::ostream`'s `operator<<`); while the parenthesis operator can take multiple arguments (which is needed for efficiently PUPing arrays).

The bar syntax will be able to copy *any* structure, whether it has a pup method or not. If there is no pup method, the C++ operator overloading rules decay the bar operator into packing the *bytes* of the structure, which will work fine for simple types on homogeneous machines. For dynamically allocated structures or heterogeneous migration, you'll need to define a pup method for all packed classes/structures. As an added benefit, the same pup methods will get called during parameter marshalling.

8 Other PPL Tools, Libraries and Applications

8.1 What is Structured Dagger?

Structured Dagger is a structured notation for specifying intra-process control dependencies in message-driven programs. It combines the efficiency of message-driven execution with the explicitness of control specification. Structured Dagger allows easy expression of dependencies among messages and computations and also among computations within the same object using `when-blocks` and various structured constructs. See the Charm++ manual for the details.

8.2 What is Adaptive MPI?

Adaptive MPI (AMPI) is an implementation of the MPI standard on top of Charm++. This allows MPI users to recompile their existing MPI applications with AMPI's compiler wrappers to take advantage of Charm++'s high level features, such as overdecomposition, overlap of communication and computation, dynamic load balancing, and fault tolerance. See the AMPI manual for more details on how AMPI works and how to use it.

8.3 What is Charisma?

Charisma++ is a prototype language for describing global view of control in a parallel program. It is designed to solve the problem of obscured control flow in the object-based model with Charm++.

8.4 Does Projections use wall time or CPU time?

Wall time.

9 Debugging

9.1 How can I debug Charm++ programs?

There are many ways to debug programs written in Charm++:

print By using `CkPrintf`, values from critical point in the program can be printed.

gdb This can be used both on a single processor, and in parallel simulations. In the latter, each processor has a terminal window with a gdb connected.

charmdebug This is the most sophisticated method to debug parallel programs in Charm++. It is tailored to Charm++ and it can display and inspect chare objects as well as messages in the system. Single *gdb*s can be attached to specific processors on demand.

9.2 How do I use charmdebug?

Currently charmdebug is tested to work only under netlrts- non-SMP versions. With other versions, testing is pending. To get the Charm Debug tool, check out the source code from the repository. This will create a directory named ccs_tools. Move to this directory and build Charm Debug.

```
git clone git://charm.cs.uiuc.edu/ccs_tools.git
cd ccs_tools
ant
```

This will create the executable `bin/charmdebug`. To start, simply substitute "charmdebug" to "charmrun":

```
shell> <path>/charmdebug ./myprogram
```

You can find more detailed information in the debugger manual in here.

9.3 Can I use distributed debuggers like Allinea DDT and RogueWave TotalView?

Yes, on mpi- versions of Charm++. In this case, the program is a regular MPI application, and as such any tool available for MPI programs can be used. Notice that some of the internal data structures (like messages in queue) might be difficult to find.

Depending on your debugging needs, see the other notes about alternatives such as CharmDebug and directly-attached gdb.

9.4 How do I use *gdb* with Charm++ programs?

It depends on the machine. On the netlrts- versions of Charm++, like netlrts-linux-x86_64, you can just run the serial debugger:

```
shell> gdb myprogram
```

If the problem only shows up in parallel, and you're running on an X terminal, you can use the *++debug* or *++debug-no-pause* options of charmrun to get a separate window for each process:

```
shell> export DISPLAY="myterminal:0"
shell> ./charmrun ./myprogram +p2 ++debug
```

9.5 When I try to use the *++debug* option I get: `remote host not responding... connection closed`

First, make sure the program at least starts to run properly without *++debug* (i.e. charmrun is working and there are no problems with the program startup phase). You need to make sure that gdb or dbx, and xterm are installed on all the machines you are using (not the one that is running `charmrun`). If you are working from a Windows machine, you need an X-win application such as exceed. You need to set this up to give the right permissions for X windows. You need to make sure the DISPLAY environment variable on the remote machine is set correctly to your local machine. I recommend ssh and putty, because it will take care of the DISPLAY environment automatically, and you can set up ssh to use tunnels so that it even works from a private subnet(e.g. 192.168.0.8). Since the xterm is displayed from the node machines, you have to make sure they have the correct DISPLAY set. Again, setting up ssh in the nodelist file to spawn node programs should take care of that.

9.6 My debugging printouts seem to be out of order. How can I prevent this?

Printouts from different processors do not normally stay ordered. Consider the code:

```
...somewhere... {
  CkPrintf("cause\n");
  proxy.effect();
}
void effect(void) {
  CkPrintf("effect\n");
}
```

Though you might expect this code to always print "cause, effect", you may get "effect, cause". This can only happen when the cause and effect execute on different processors, so cause's output is delayed.

If you pass the extra command-line parameter *+syncprint*, then CkPrintf actually blocks until the output is queued, so your printouts should at least happen in causal order. Note that this does dramatically slow down output.

9.7 Is there a way to flush the print buffers in Charm++ (like `fflush()`)?

Charm++ automatically flushes the print buffers every newline and at program exit. There is no way to manually flush the buffers at another point.

9.8 My Charm++ program is causing a seg fault, and the debugger shows that it's crashing inside *malloc* or *printf* or *fopen*!

This isn't a bug in the C library, it's a bug in your program – you're corrupting the heap. Link your program again with *-memory paranoid* and run it again in the debugger. *-memory paranoid* will check the heap and detect buffer over- and under-run errors, double-deletes, delete-garbage, and other common mistakes that trash the heap.

9.9 Everything works fine on one processor, but when I run on multiple processors it crashes!

It's very convenient to do your testing on one processor (i.e., with *+p1*); but there are several things that only happen on multiple processors.

A single processor has just one set of global variables, but multiple processors have different global variables. This means on one processor, you can set a global variable and it stays set "everywhere" (i.e., right here!), while on two processors the global variable never gets initialized on the other processor. If you must use globals, either set them on every processor or make them into *readonly* globals.

A single processor has just one address space, so you actually *can* pass pointers around between chares. When running on multiple processors, the pointers dangle. This can cause incredibly weird behavior – reading from uninitialized data, corrupting the heap, etc. The solution is to never, ever send pointers in messages – you need to send the data the pointer points to, not the pointer.

9.10 I get the error: "`Group ID is zero-- invalid!`". What does this mean?

The *group* it is referring to is the chare group. This error is often due to using an uninitialized proxy or handle; but it's possible this indicates severe corruption. Run with *++debug* and check it you just sent a message via an uninitialized proxy.

9.11 I get the error: `Null-Method Called. Program may have Unregistered Module!!` What does this mean?

You are trying to use code from a module that has not been properly initialized.

So, in the *.ci* file for your *mainmodule*, you should add an "extern module" declaration:

```
mainmodule whatever {
  extern module someModule;
  ...
}
```

9.12 When I run my program, it gives this error:

```
Charmrun: error on request socket--
Socket closed before recv.
```

This means that the node program died without informing `charmrun` about it, which typically means a segmentation fault while in the interrupt handler or other critical communications code. This indicates severe corruption in Charm++'s data structures, which is likely the result of a heap corruption bug in your program. Re-linking with *-memory paranoid* may clarify the true problem.

9.13 When I run my program, sometimes I get a `Hangup`, and sometimes `Bus Error`. What do these messages indicate?

`Bus Error` and `Hangup` both are indications that your program is terminating abnormally, i.e. with an uncaught signal (SEGV or SIGBUS). You should definitely run the program with gdb, or use *++debug*. Bus Errors often mean there is an alignment problem, check if your compiler or environment offers support for detection of these.

10 Versions and Ports

10.1 Has Charm++ been ported to use MPI underneath? What about OpenMP?

Charm++ supports MPI and can use it as the underlying communication library. We have tested on MPICH, OpenMPI, and also most vendor MPI variants. Charm++ also has explicit support for SMP nodes in MPI version. Charm++ hasn't been ported to use OpenMP, but OpenMP can be used from Charm++.

10.2 How complicated is porting Charm++/Converse?

Depends. Hopefully, the porting only involves fixing compiler compatibility issues. The LRTS abstraction layer was designed to simplify this process and has been used for the MPI, Verbs, uGNI, PAMI and OFI layers. User level threads and Isomalloc support may require special platform specific support. Otherwise Charm++ is generally platform independent.

10.3 If the source is available how feasible would it be for us to do ports ourselves?

The source is always available, and you're welcome to make it run anywhere. Any kind of UNIX, Windows, and MacOS machine should be straightforward: just a few modifications to `charm/src/arch/.../conv-mach.h` (for compiler issues) and possibly a new *machine.c* (if there's a new communication system involved). However, porting to embedded hardware with a proprietary OS may be fairly difficult.

10.4 To what platform has Charm++/Converse been ported to?

Charm++/Converse has been ported to most UNIX and Linux OS, Windows, and MacOS.

10.5 Is it hard to port Charm++ programs to different machines?

Charm++ itself it fully portable, and should provide exactly the same interfaces everywhere (even if the implementations are sometimes different). Still, it's often harder than we'd like to port user code to new machines.

Many parallel machines have old or weird compilers, and sometimes a strange operating system or unique set of libraries. Hence porting code to a parallel machine can be suprisingly difficult.

Unless you're absolutely sure you will only run your code on a single, known machine, we recommend you be very conservative in your use of the language and libraries. "But it works with my gcc!" is often true, but not very useful.

Things that seem to work well everywhere include:

- Small, straightforward Makefiles. gmake-specific (e.g., "ifeq", filter variables) or convoluted makefiles can lead to porting problems and confusion. Calling charmc instead of the platform-specific compiler will save you many headaches, as charmc abstracts away the platform specific flags.
- Basically all of ANSI C and fortran 77 work everywhere. These seem to be old enough to now have the bugs largely worked out.
- C++ classes, inheritance, virtual methods, and namespaces work without problems everywhere. Not so uniformly supported are C++ templates, the STL, new-style C++ system headers, and the other features listed in the C++ question below.

10.6 How should I approach portability of C language code?

Our suggestions for Charm++ developers are:

- Avoid the nonstandard type "long long", even though many compilers happen to support it. Use CMK_INT8 or CMK_UINT8, from conv-config.h, which are macros for the right thing. "long long" is not supported on many 64-bit machines (where "long" is 64 bits) or on Windows machines (where it's "__int64").
- The "long double" type isn't present on all compilers. You can protect long double code with *#ifdef CMK_LONG_DOUBLE_DEFINED* if it's really needed.
- Never use C++ "//" comments in C code, or headers included by C. This will not compile under many compilers.
- "bzero" and "bcopy" are BSD-specific calls. Use memset and memcpy for portable programs.

If you're writing code that is expected to compile and run on Microsoft Windows using the Visual C++ compiler (e.g. modification to NAMD that you intend to submit for integration), that compiler has limited support for the C99 standard, and Microsoft recommends using C++ instead.

Many widely-used C compilers on HPC systems have limited support for the C11 standard. If you want to use features of C11 in your code, particularly `_Atomic`, we recommend writing the code in C++ instead, since C++11 standard support is much more ubiquitous.

10.7 How should I approach portability and performance of C++ language code?

The Charm++ system developers are conservative about which C++ standard version is relied upon in runtime system code and what features get used to ensure maximum portability across the broad range of HPC systems and the compilers used on them. Through version 6.8.x, the system code requires only limited support for C++11 features, specifically variadic templates and R-value references. From version 6.9 onwards, the system will require a compiler and standard library with at least full C++11 support.

A good reference for which compiler versions provide what level of standard support can be found at http://en.cppreference.com/w/cpp/compiler_support

Developers of several Charm++ applications have reported good results using features in more recent C++ standards, with the caveat of requiring that those applications be built with a sufficiently up-to-date C++ compiler.

The containers specified in the C++ standard library are generally designed to provide a very broad API that can be used correctly over highly-varied use cases. This often entails tradeoffs against the performance attainable for narrower use cases that some applications may have. The most visible of these concerns are the tension between strict iterator invalidation semantics and cache-friendly memory layout. We recommend that developers whose code includes container access in performance-critical elements explore alternative implementations, such as those published by EA, Google, and Facebook, or potentially write custom implementations tailored to their application's needs.

In benchmarks across a range of compilers, we have found that avoiding use of exceptions (i.e. `throw/catch`) and disabling support for them with compiler flags can produce higher-performance code, especially with aggressive optimization settings enabled. The runtime system does not use exceptions internally. If your goal as an application developer is to most efficiently use large-scale computational resources, we recommend alternative error-handling strategies.

10.8 Why do I get a link error when mixing Fortran and C/C++?

Fortran compilers "mangle" their routine names in a variety of ways. g77 and most compilers make names all lowercase, and append an underscore, like "foo_". The IBM xlf compiler makes names all lowercase without an underscore, like "foo". Absoft f90 makes names all uppercase, like "FOO".

If the Fortran compiler expects a routine to be named "foo_", but you only define a C routine named "foo", you'll get a link error ("undefined symbol foo_"). Sometimes the UNIX command-line tool *nm* (list symbols in a .o or .a file) can help you see exactly what the Fortran compiler is asking for, compared to what you're providing.

Charm++ automatically detects the fortran name mangling scheme at configure time, and provides a C/C++ macro "FTN_NAME", in "charm-api.h", that expands to a properly mangled fortran routine name. You pass the FTN_NAME macro two copies of the routine name: once in all uppercase, and again in all lowercase. The FTN_NAME macro then picks the appropriate name and applies any needed underscores. "charm-api.h" also includes a macro "FDECL" that makes the symbol linkable from fortran (in C++, this expands to extern "C"), so a complete Fortran subroutine looks like in C or C++:

```
FDECL void FTN_NAME(FOO,foo)(void);
```

This same syntax can be used for C/C++ routines called from fortran, or for calling fortran routines from C/C++. We strongly recommend using FTN_NAME instead of hardcoding your favorite compiler's name mangling into the C routines.

If designing an API with the same routine names in C and Fortran, be sure to include both upper and lowercase letters in your routine names. This way, the C name (with mixed case) will be different from all possible Fortran manglings (which all have uniform case). For example, a routine named "foo" will have the same name in C and Fortran when using the IBM xlf compilers, which is bad because the C and Fortran versions should take different parameters. A routine named "Foo" does not suffer from this problem, because the C version is "Foo, while the Fortran version is "foo_", "foo", or "FOO".

10.9 How does parameter passing work between Fortran and C?

Fortran and C have rather different parameter-passing conventions, but it is possible to pass simple objects back and forth between Fortran and C:

- Fortran and C/C++ data types are generally completely interchangeable:

C/C++ Type	Fortran Type
int	INTEGER, LOGICAL
double	DOUBLE PRECISION, REAL*8
float	REAL, REAL*4
char	CHARACTER

- Fortran internally passes everything, including constants, integers, and doubles, by passing a pointer to the object. Hence a fortran "INTEGER" argument becomes an "int *" in C/C++:

```
/* Fortran */
SUBROUTINE BAR(i)
    INTEGER :: i
    x=i
END SUBROUTINE

/* C/C++ */
FDECL void FTN_NAME(BAR,bar)(int *i) {
    x=*i;
}
```

- 1D arrays are passed exactly the same in Fortran and C/C++: both languages pass the array by passing the address of the first element of the array. Hence a fortran "INTEGER, DIMENSION(:)" array is an "int *" in C or C++. However, Fortran programmers normally think of their array indices as starting from index 1, while in C/C++ arrays always start from index 0. This does NOT change how arrays are passed in, so x is actually the same in both these subroutines:

```
/* Fortran */
SUBROUTINE BAR(arr)
    INTEGER :: arr(3)
    x=arr(1)
END SUBROUTINE

/* C/C++ */
FDECL void FTN_NAME(BAR,bar)(int *arr) {
    x=arr[0];
}
```

- There is a subtle but important difference between the way f77 and f90 pass array arguments. f90 will pass an array object (which is not intelligible from C/C++) instead of a simple pointer if all of the following are true:
  - A f90 "INTERFACE" statement is available on the call side.
  - The subroutine is declared as taking an unspecified-length array (e.g., "myArr(:)") or POINTER variable.

  Because these f90 array objects can't be used from C/C++, we recommend C/C++ routines either provide no f90 INTERFACE or else all the arrays in the INTERFACE are given explicit lengths.

- Multidimensional allocatable arrays are stored with the smallest index first in Fortran. C/C++ do not support allocatable multidimensional arrays, so they must fake them using arrays of pointers or index arithmetic.

```
/* Fortran */
SUBROUTINE BAR2(arr,len1,len2)
    INTEGER :: arr(len1,len2)
    INTEGER :: i,j
    DO j=1,len2
      DO i=1,len1
        arr(i,j)=i;
      END DO
```

```
    END DO
END SUBROUTINE

/* C/C++ */
FDECL void FTN_NAME(BAR2,bar2)(int *arr,int *len1p,int *len2p) {
    int i,j; int len1=*len1p, len2=*len2p;
    for (j=0;j<len2;j++)
    for (i=0;i<len1;i++)
        arr[i+j*len1]=i;
}
```

- Fortran strings are passed in a very strange fashion. A string argument is passed as a character pointer and a length, but the length field, unlike all other Fortran arguments, is passed by value, and goes after all other arguments. Hence

```
/* Fortran */
SUBROUTINE CALL_BARS(arg)
    INTEGER :: arg
    CALL BARS('some string',arg);
END SUBROUTINE

/* C/C++ */
FDECL void FTN_NAME(BARS,bars)(char *str,int *arg,int strlen) {
    char *s=(char *)malloc(strlen+1);
    memcpy(s,str,strlen);
    s[strlen]=0; /* nul-terminate string */
    printf("Received Fortran string '%s' (%d characters)\n",s,strlen);
    free(s);
}
```

- A f90 named TYPE can sometimes successfully be passed into a C/C++ struct, but this can fail if the compilers insert different amounts of padding. There does not seem to be a portable way to pass f90 POINTER variables into C/C++, since different compilers represent POINTER variables differently.

10.10 How do I use Charm++ on Xeon Phi?

In general, no changes are required to use Charm++ on Xeon Phis. To compile code for Knights Landing, no special flags are required. To compile code for Knights Corner, one should build Charm++ with the `mic` option. In terms of network layers, we currently recommend building the MPI layer (`mpi-linux-x86_64`) except for machines with custom network layers, such as Cray systems, on which we recommend building for the custom layer (`gni-crayxc` for Cray XC machines, for example). To enable AVX-512 vector instructions, Charm++ can be built with `-xMIC-AVX512` on Intel compilers or `-mavx512f -mavx512er -mavx512cd -mavx512pf` for GNU compilers.

10.11 How do I use Charm++ on GPUs?

Charm++ users have two options when utilizing GPUs in Charm++.

The first is to write CUDA (or OpenCL, etc) code directly in their Charm++ applications. This does not take advantage of any of the special GPU-friendly features the Charm++ runtime provides and is similar to how programmers utilize GPUs in other parallel environments, e.g. MPI.

The second option is to leverage Charm++'s GPU library, GPU Manager. This library provides several useful features including:

- Automated data movement

- Ability to invoke callbacks at various points
- Host side pinned memory pooling
- Asynchronous kernel invocation
- Integrated tracing in Projections

To do this, Charm++ must be built with the `cuda` option. Users must describe their kernels using a work request struct, which includes the buffers to be copied, callbacks to be invoked, and kernel to be executed. Additionally, users can take advantage of a pre-allocated host side pinned memory pool allocated by the runtime via invoking `hapi_poolMalloc`. Finally, the user must compile this code using the appropriate `nvcc` compiler as per usual.

More details on using GPUs in Charm++ can be found in the GPU Manager Library entry in the larger Libraries Manual.

11 Converse Programming

11.1 What is Converse? Should I use it?

Converse is the low-level portable messaging layer that Charm++ is built on, but you don't have to know anything about Converse to use Charm++. You might want to learn about Converse if you want a capable, portable foundation to implement a new parallel language on.

11.2 How much does getting a random number generator "right" matter?

drand48 is nonportable and woefully inadequate for any real simulation task. Even if each processor seeds drand48 differently, there is no guarantee that the streams of pseduo-random numbers won't quickly overlap. A better generator would be required to "do it right" (See Park & Miller, CACM Oct. 88).

11.3 What should I use to get a proper random number generator?

Converse provides a 64-bit pseudorandom number generator based on the SPRNG package originally written by Ashok Shrinivasan at NCSA. For detailed documentation, please take a look at the Converse Extensions Manual on the Charm++ website. In short, you can use *CrnDrand()* function instead of the unportable *drand48()* in Charm++.

12 Charm++ and Converse Internals

12.1 How is the Charm++ source code organized and built?

All the Charm++ core source code is soft-linked into the `charm/<archname>/tmp` directory when you run the build script. The libraries and frameworks are under `charm/<archname>/tmp/libs`, in either `ck-libs` or `conv-libs`.

12.2 I just changed the Charm++ core. How do I recompile Charm++?

cd into the `charm/<archname>/tmp` directory and make. If you want to compile only a subset of the entire set of libraries, you can specify it to make. For example, to compile only the Charm++ RTS, type *make charm++*.

12.3 Do we have a *#define charm_version* somewhere? If not, which version number should I use for the current version?

Yes, there is a Charm++ version number defined in the macro `CHARM_VERSION`.