

CONVERSE and CHARM++ Libraries Manual

The *iRecv* library was developed by Gengbin Zheng. Barrier library was developed by Terry Wilmarth. TEMPO library was originally developed by Zehra Sura, and later rewritten by Milind Bhandarkar. Multiphase shared arrays library was written by Rahul Joshi and enhanced by Jayant DeSouza. The matrix multiplication library was developed by Esteban Pauli.

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
2	<i>iRecv</i> Library	4
3	Barrier Library	6
4	TeMPO Library	9
5	DDT Library	11
6	liveViz Library	14
6.1	Introduction	14
6.2	How to use liveViz with CHARM++ program	14
6.3	Format of deposit image	15
6.4	liveViz Initialization	15
6.5	Compilation	16
6.6	Poll Mode	16
7	Multi-phase Shared Arrays Library	18
8	3D FFT Library	19
8.1	Introduction and Motivation	19
8.2	Compilation and Execution	19
8.3	Library Initialization and Data Format	19
8.4	Library Interfaces	21
8.4.1	Charm++ interface	21
8.4.2	AMPI Interface	23
9	Matrix Multiplication Library	24
9.1	Introduction and Motivation	24
9.2	Compiling and Linking	24
9.3	Usage Description	24
9.4	Library Interface Details	25
9.4.1	Library Initialization	25
9.4.2	Performing Multiplications	25
9.5	Example	26

Chapter 1

Introduction

This manual describes CHARM++ and CONVERSE libraries. This is a work in progress towards a standard library for parallel programming on top of the CONVERSE and CHARM++ system. All of these libraries are included in the source and binary distributions of CHARM++/CONVERSE.

Chapter 2

iRecv Library

The *iRecv* library provides asynchronous communication mode for use in chare arrays in the message-passing style. The control flow of a message-passing program is broken into two parts (to provide the efficient "split phase" structure): nonblocking communication operations and `await` with callback functions as continuations. It thus provides a style that MPI programmers may find intuitive. This library aids in porting existing MPI codes to CHARM++ without using expensive context-switching of threads. In this approach, a chare array element is used to represent a virtual processor.

There are three functions in *iRecv* library.

```
void send(buf, size, dest, tag, refno)
```

```
void *buf;  
int size, dest, tag, refno;
```

Sends message which is pointed by *buf* to another array element whose index is specified by *dest* with *tag* and *refno*. *buf* is a message buffer containing the data to be sent; *size* is the total size of the message in bytes. Like in `MPI_send`, the *tag* is used for matching message on destination array element. The integer *refno* is a reference number, usually the iteration number.

```
void irecv(buf, size, source, tag, refno)
```

```
void *buf;  
int size, source, tag, refno;
```

This function registers *tag* and *buf* with the library. When the desired message arrives, it copies the matching message into the location given by the *buf*.

```
void awaitAll(f, data, refno)
```

```
recvCallback f;  
void *data;  
int refno;
```

This function registers a callback function *f* with the library. This function is invoked with *data* as its argument when all the previously issued `irecvs` with *refno* as reference number complete.

To use the *iRecv* library, first one has to create a chare array, which is inherited from class `receiver`. The sender entry method of the chare array element prepares the message buffer and calls `send` function to send message to another array element; The receiver specifies the matching tags and buffer to get the message. After `irecv`, the receiver needs to call `awaitAll` function to wait for all the `irecv` function calls to complete. However, `awaitAll` is a nonblocking function. The callback function will be called after the relevant `irecv` calls complete.

Here is an example:

```
int size = 100;  
for (int i=0; i<size; i++) buf[i] = data[i];  
send(buf, size*sizeof(double), neighbor, tag, iter);  
irecv(buf, size*sizeof(double), neighbor, tag, iter);  
awaitAll(callfunc, this, iter);
```

and callback function can be declared as:

```
void callfunc(void *obj)
{
    ... do something with obj
}
```

Chapter 3

Barrier Library

We needed a way to synchronise a subset of tasks in CHARM++ programs. Quiescence detection is one way this can be done, if we want to synchronize **all** tasks, but barriers provide a more flexible option with less overhead.

To make use of barriers, the user initializes a *barrier group* in the main function. A virtual binary tree structure is imposed on the PEs, with PE 0 at the root. PEs perform computations and then execute `atBarrier` when they want to synchronize. `atBarrier` takes a single parameter of type `FP *`. The user must declare and initialize some variable *fnptr* as follows:

```
FP *fnptr = new FP;
fnptr->fp = theFn;
```

The variable *theFn* above is the void function to be executed when all PEs have synchronized.

The barrier keeps track of child PEs, and when a given PE has heard from all its children, it then notifies the parent. When the root hears from both children, all PEs are accounted for, and the respective void function is executed on each PE.

What follows is a very simple test program illustrating the usage of a barrier.

```
// File: test.ci

mainmodule Test {
  extern module Barrier;
  readonly CkChareID mainhandle;
  readonly int barrierGroup;

  mainchare main {
    entry main();
    entry void Quiescence1(void);
  };

  group busy {
    entry busy(void);
  };
};

// File: test.h

#include "Test.decl.h"
int barrierGroup;
CkChareID mainhandle;
```

```

class main : public Chare \{
public:
    main(CkArgMsg *m);
    void Quiescence1(void);
\};

class busy : public Group \{
public:
    busy(void);
\}
\end{alltt}

\begin{verbatim}
// File: test.C

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "charm++.h"

#include "test.h"
#include "Test.def.h"

#include "barrier.h"

main::main(CkArgMsg *m)
\{
    barrierGroup = barrierInit();
    CProxy_busy::ckNew();
    CkStartQD(CProxy_main::ckIdx_Quiescence1(), \&mainhandle);
\}

void main::Quiescence1()
\{
    CkPrintf("All done... Exiting.\n");
    CkExit();
\}

void theFn();

busy::busy()
\{
    int i, j=0;
    FP *fnptr = new FP;

    for (i=0; i<(CkMyPe()+1)*1000000; i++)
        j = j + i;
    CkPrintf("[%d] going to barrier with j=%d\n", CkMyPe(), j);
    fnptr->fp = theFn();
    CProxy_barrier(barrierGroup).ckLocalBranch()->atBarrier(fnptr);
\}

```

```
void theFn()  
\{  
    CkPrintf("[%d] fnptr executing!\n", CkMyPe());  
\}
```

Chapter 4

TeMPO Library

TEMPO (Threaded Message Passing Objects) provides a simple way to communicate between threaded methods of objects. Using TEMPO, chares, groups, and arrays can have threads associated with them. Any CHARM++ object can send a tagged message to a TEMPO Object, and a threaded method in a TEMPO Object can block on a specific tagged message.

In order to use TEMPO functionalities, chares, groups, and arrays need to inherit from TEMPO objects called `TempoChare`, `TempoGroup`¹, and `TempoArray` respectively. This inheritance also needs to be specified in the CHARM++ interface file. Thus, a chare C in module M that needs to receive a tagged message directed to it in entry E , needs to inherit from `TempoChare` as shown below:

```
// file.ci
module M {
  ...
  chare C : TempoChare {
    entry C(void);
    entry [threaded] void E(Msg *); // note use of threaded
  };
  ...
}

// file.h
#include "file.decl.h"
...
class C : public TempoChare { // TempoChare inherits from Chare already
public:
  C(void);
  E(Msg *);
};
...
```

The code for entry method E can contain a call to `ckTempoRecv` to block for a tagged message. Any ordinary chare can send a tagged message to a TEMPO object using `ckTempoSend` static method of `TempoChare`.

`TempoGroup` and `TempoArray` provide additional static methods to send tagged messages to individual elements, as well as broadcast. In addition, `TempoArray` provides methods to perform “reduction” over all the elements of an array. Currently supported reduction operations are max, min, sum, and product over datatypes float, int, and double. The signatures of these methods are given below:

```
class TempoChare : public Chare, public ... {
```

¹Currently, a `nodegroup` variety of TEMPO object does not exist, though it will be added in future.

```

void ckTempoRecv(int tag, void *buffer, int buflen);
void ckTempoRecv(int tag1, int tag2, void *buffer, int buflen);
static void ckTempoSend(int tag1, int tag2, void *buffer,int buflen,
                        CkChareID cid);
static void ckTempoSend(int tag, void *buffer,int buflen, CkChareID cid);
int ckTempoProbe(int tag1, int tag2);
int ckTempoProbe(int tag);
};

// TempoGroup includes all the TempoChare methods

class TempoGroup : public Group, public ... {
    static void ckTempoBcast(int tag, void *buffer, int buflen, CkGroupID grpID);
    static void ckTempoSendBranch(int tag1, int tag2, void *buffer, int buflen,
                                  CkGroupID grpID, int processor);
    static void ckTempoSendBranch(int tag, void *buffer, int buflen,
                                  CkGroupID grpID, int processor);
    void ckTempoBcast(int sender, int tag, void *buffer, int buflen);
    void ckTempoSendBranch(int tag1, int tag2, void *buffer, int buflen,
                           int processor);
    void ckTempoSendBranch(int tag, void *buffer, int buflen, int processor);
};

// TempoArray includes all the TempoChare methods

class TempoArray : public ArrayElement, public Tempo {
    static void ckTempoSendElem(int tag1, int tag2, void *buffer, int buflen,
                               CkArrayID aid, int idx);
    static void ckTempoSendElem(int tag, void *buffer, int buflen,
                               CkArrayID aid, int idx);
    void ckTempoSendElem(int tag1, int tag2, void *buffer, int buflen, int idx);
    void ckTempoSendElem(int tag, void *buffer, int buflen, int idx);
    void ckTempoBarrier(void);
    void ckTempoBcast(int sender, int tag, void *buffer, int buflen);
    void ckTempoReduce(int root, int op, void *inbuf, void *outbuf, int count,
                       int type);
    void ckTempoAllReduce(int op,void *inbuf,void *outbuf,int count,int type);
};

```

All the `ckSend*` methods have versions that send messages with one tag or with two tags. A wild card tag (`TEMPO_ANY` may be in specified in `ckTempoRecv` that matches with any tag. All the tags must range between 0 and 1024. All the other tags have other uses in the system. Reduction operations are indicated by integer constants. They are: `TEMPO_MAX`, `TEMPO_MIN`, `TEMPO_SUM`, and `TEMPO_PROD`. Reduction data types can be specified using integer constants: `TEMPO_FLOAT`, `TEMPO_INT`, and `TEMPO_DOUBLE`. The *root* parameter in `ckTempoReduce`, and the *sender* parameter in `ckTempoBcast` indicate whether the calling element is the root of the collective operation or not. In case of a reduction, the root element is returned the result of the reduction operation in *outbuf*² In case of a broadcast, *buffer* on the sender contains the message to be sent to other elements. For each of the send methods, the specified message buffer could be reused once the method returns.

All the `TEMPO` include files are automatically includes from `charm++.h`. There is no need to include any other file to use `TEMPO`.

²*outbuf* and *inbuf* can be the same.

Chapter 5

DDT Library

DDT (Derived Data Types) provides a simple way to create user defined datatypes. With standard datatypes such as int, double, char one can use a contiguous memory space. Often it is desirable to use data that is not homogeneous such as structure or that is not contiguous in memory such as some selection in array. This library allows The user to define derived datatypes, that specify more general data layouts. These derived datatypes can then be used in a similar way as basic datatypes.

Some examples where user might want to use derived datatypes are: using only upper tringle of a matrix, using only elements in alternate rows and columns in a 2D array. It might be cumbersome to program it in user code every time it is needed. DDT library provides a simple and convinient way to create such datatypes, find out their lengths and extents.

Currently supported datatypes are

- Contiguous - Contiguous datatype constructs a typemap consisting of the replication of a datatype into contiguous locations.
- Vector - Vector datatype allows replication of a datatype into locations that consist of equally spaced blocks.
- HVector - Similar to Vector except that it allows a stride which consists of an arbitrary number of bytes as opposed to stride being multiple of block size in Vector.
- Indexed - The Indexed datatype allows one to specify a noncontiguous data layout where displacements between successive blocks need not be equal.
- HIndexed - Similar to Indexed except that displacement can be arbitrary number of bytes.
- Struct - This is the most general type constructor. It allows each block to consist of replications of different datatypes.

Derived datatypes can be constructed from basic datatypes as well as derived data types. For example, a Struct datatype can be made of contiguous, vector datatypes.

DDT library provides following functionality.

- First, a pool of datatypes should be created, by calling `new DDT()`.
- new data types can be created by calling `newContiguous()`, `newVector()` etc functions of DDT which will give an integer index for a datatype. This index can be used later to refer to this derived datatype. Creating a datatype just initializes the type, extent, size of a datatype. It does not copy buffers.
- `DDT_datatype` can be retrieved using `getType()` function of DDT. Previously created index needs to be passed to this function.
- To copy non-contiguous bytes based on a data type, `serialize` funtion needs to be called on `DDT_Datatype`.
- Also, `getSize()`, `getExtent()` functions can be used to get size and extent of a datatype.

```

//file test.C

#include "ddt.h"

myDDT = new DDT((void*)0);

DDT_Type  newType1, newType2 ;

//Create a new contiguous datatype which will copy the given buffer (in this
// case, int) count times in a new buffer

myDDT->newContiguous(count, DDT_INT, &newType1) ;

//Create a new vector datatype which will copy count elements of type DDT_DOUBLE
// with length blocklength, spaced stride apart in the original buffer.
myDDT->newVector(count, blocklength, stride, DDT_DOUBLE, newtype);

//newType1 and newType2 can be used for further referring to contiguous and
//vector type of datatype.

// Create a new buffer with characteristics specified while creating
// the new datatype.

DDT_DataType *ddt = myDDT->getType(newType1);

ddt->serialize(buf1, buf2, size, dir);

// dir = 1, specifies that data is copied from buf1 to buf2, buf1 being old
// buffer and buf2 being the new buffer which should contains only the
// selected data.

//When needed to do the reverse, keep dir = -1

Following datatype constants are supported.
Primitive Datatypes:
DDT_DOUBLE, DDT_INT, DDT_FLOAT, DDT_COMPLEX, DDT_LOGICAL, DDT_CHAR, DDT_BYTE, DDT_PACKED,
DDT_SHORT, DDT_LONG, DDT_UNSIGNED_CHAR, DDT_UNSIGNED_SHORT, DDT_UNSIGNED, DDT_UNSIGNED_LONG,
DDT_LONG_DOUBLE Derived Datatypes:
DDT_CONTIGUOUS, DDT_VECTOR, DDT_HVECTOR, DDT_INDEXED, DDT_HINDEXED, DDT_STRUCT
DDT library provides following interface - To create new datatypes.

class DDT {
public:
void newContiguous(int count, DDT_Type  oldType, DDT_Type* newType);
void newVector(int count, int blocklength, int stride, DDT_Type oldtype,
  DDT_Type* newtype) ;
void newHVector(int count, int blocklength, int stride, DDT_Type oldtype,
  DDT_Type* newtype);
void newIndex(int count, int* arrbLength, int* arrDisp , DDT_Type oldtype, DDT_Type* newtype);
void newHIndexed(int count, int* arrbLength, int* arrDisp , DDT_Type oldtype, DDT_Type* newtype);
void newStruct(int count, int* arrbLength, int* arrDisp , DDT_Type *oldtype, DDT_Type* newtype);
}

```

- To Use the derived datatype.

```

class DDT_Datatype {
public:
virtual int getSize();
virtual int getExtent();
virtual int serialize(char* userdata, char* buffer, int num, int dir);
}

```

Examples of Derived Datatype

- Contiguous -

If oldType is a derived datatype struct consisting a char and a double,

```
ddt->newContiguous(5, oldType, &newType1) ;
```

creates a contiguous datatype which has char, double, char, double, char, double, char double, char double

- Vector and HVector - To select alternate elements from each row and to select alternate rows in a 5 X 5 2-D array of doubles.

```
ddt->newVector(3,1,2, DDT_DOUBLE, &oneSlice) ;
```

This will give a row in which alternate elements are selected.

```
ddt->newHVector(3, 1, 5*sizeof(double), oneSlice, &twoSlice) ;
```

This will give three rows each of type oneSlice.

- Indexed -

To select 2, 3, 4 blocks of Integer spaced 4,5,6 blocks in a given 1-D array of integers. blength[3] = 2, 3, 4 ; stride[3] = 4, 5, 6 ;

```
ddt->Indexed(1, blength, stride, MPI_INT, &newArray);
```

- HIndexed -

To select 2, 3, 4 blocks of Integer spaced 4,5,6 bytes in a given 1-D array of integers. blength[3] = 2, 3, 4 ; stride[3] = 4, 5, 6 ;

```
ddt->HIndexed(1, blength, stride, MPI_INT, &newArray);
```

In Indexed datatype, displacements will be 4, 5 and 6 multiples of int, while in HIndexed, displacements will be 4, 5 and 6 bytes.

Chapter 6

liveViz Library

6.1 Introduction

If array elements compute a small piece of a large 2D image, then these image chunks can be combined across processors to form one large image using the liveViz library. In other words, liveViz provides a way to reduce 2D-image data, which combines small chunks of images deposited by chares into one large image.

This visualization library follows the client server model. The server, a parallel Charm++ program, does all image assembly, and opens a network (CCS) socket which clients use to request and download images. The client is a small Java program. A typical use of this is:

```
cd charm/pgms/charm++/ccs/liveViz/server
make
./charmrun ./pgm +p2 ++server ++server-port 1234 &
~/charm/bin/liveViz localhost 1234
```

6.2 How to use liveViz with Charm++ program

The liveViz routines are in the Charm++ header “liveViz.h”.

A typical program provides a chare array with one entry method with the following prototype:

```
entry void functionName(liveVizRequestMsg *m);
```

This entry method is supposed to deposit its (array element’s) chunk of the image. This entry method has following structure:

```
void myArray::functionName (liveVizRequestMsg *m)
{
    // prepare image chunk
    ...

    liveVizDeposit (m, startX, startY, width, height, imageBuff, this);

    // delete image buffer if it was dynamically allocated
}
```

Here, “width” and “height” are the size, in pixels, of this array element’s portion of the image, contributed in “imageBuff” (described below). This will show up on the client’s assembled image at 0-based pixel (startX,startY). The client’s display width and height are stored in `m->req.wid` and `m->req.ht`.

By default liveViz combines image chunks by doing a saturating sum of overlapping pixel values. If you want liveViz to combine image chunks by using max (i.e. for overlapping pixels in deposited image chunks,

final image will have the pixel with highest intensity or in other words largest value), you need to pass one more parameter (`liveVizCombine_t`) to the “`liveVizDeposit`” function:

```
liveVizDeposit (m, startX, startY, width, height, imageBuff, this,
               max_image_data);
```

You can also reduce floating-point image data using `sum_float_image_data` or `max_float_image_data`.

6.3 Format of deposit image

“`imageBuff`” is run of bytes representing a rectangular portion of the image. This buffer represents image using a row-major format, so 0-based pixel (x,y) (x increasing to the right, y increasing downward in typical graphics fashion) is stored at array offset “`x+y*width`”.

If the image is gray-scale (as determined by `liveVizConfig`, below), each pixel is represented by one byte. If the image is color, each pixel is represented by 3 consecutive bytes representing red, green, and blue intensity.

If the image is floating-point, each pixel is represented by a single ‘float’, and after assembly colored by calling the user-provided routine below. This routine converts fully assembled ‘float’ pixels to RGB 3-byte pixels, and is called only on processor 0 after each client request.

```
extern "C"
void liveVizFloatToRGB(liveVizRequest &req,
                      const float *floatSrc, unsigned char *destRgb,
                      int nPixels);
```

6.4 liveViz Initialization

`liveViz` library needs to be initialized before it can be used for visualization. For initialization follow the following steps from your main chare:

1. Create your chare array (array proxy object ‘a’) with the entry method ‘`functionName`’ (described above).
2. Create a `CkCallback` object (‘c’), specifying ‘`functionName`’ as the callback function. This callback will be invoked whenever the client requests a new image.
3. Create a `liveVizConfig` object (‘cfg’). `liveVizConfig` takes a number of parameters, as described below.
4. Call `liveVizInit` (cfg, a, c).

The `liveVizConfig` parameters are:

- The first parameter is the pixel type to be reduced:
 - “false” or `liveVizConfig::pix_greyscale` means a greyscale image (1 byte per pixel).
 - “true” or `liveVizConfig::pix_color` means a color image (3 RGB bytes per pixel).
 - `liveVizConfig::pix_float` means a floating-point color image (1 float per pixel, can only be used with `sum_float_image_data` or `max_float_image_data`).
- The second parameter is the flag “serverPush”, which is passed to the client application. If set to true, the client will repeatedly request for images. When set to false the client will only request for images when its window is resized and needs to be updated.
- The third parameter is an optional 3D bounding box (type `CkBbox3d`). If present, this puts the client into a 3D visualization mode.

A typical 2D, RGB, non-push call to `liveVizConfig` looks like this:

```
liveVizConfig cfg(true,false);
```

6.5 Compilation

A CHARM++ program that uses liveViz must be linked with '-module liveViz'.

Before compiling a liveViz program, the liveViz library may need to be compiled. To compile the liveViz library:

- go to `.../charm/tmp/libs/ck-libs/liveViz`
- `make`

6.6 Poll Mode

In some cases you may want a server to deposit images only when it is ready to do so. For this case the server will not register a callback function that triggers image generation, but rather the server will deposit an image at its convenience. For example a server may want to create a movie or series of images corresponding to some timesteps in a simulation. The server will have a timestep loop in which an array computes some data for a timestep. At the end of each iteration the server will deposit the image. The use of LiveViz's Poll Mode supports this type of server generation of images.

Poll Mode contains a few significant differences to the standard mode. First we describe the use of Poll Mode, and then we will describe the differences. `liveVizPoll` must get control during the creation of your array, so you call `liveVizPollInit` with no parameters.

```
liveVizPollInit();
CkArrayOptions opts(nChares);
arr = CProxy_lvServer::ckNew(opts);
```

To deposit an image, the server just calls `liveVizPollDeposit`. The server must take care not to generate too many images, before a client requests them. Each server generated image is buffered until the client can get the image. The buffered images will be stored in memory on processor 0.

```
liveVizPollDeposit( this,
                    startX,startY,           // Location of local piece
                    localSizeX,localSizeY,   // Dimensions of the piece I'm depositing
                    globalSizeX,globalSizeY, // Dimensions of the entire image
                    img,                     // Image byte array
                    sum_image_data,         // Desired image combiner
                    3                        // Bytes/pixel
                    );
```

The last two parameters are optional. By default they are set to `sum_image_data` and 3 bytes per pixel.

A sample `liveVizPoll` server and client are available at:

```
.../charm/examples/charm++/lvServer
.../charm/java/bin/lvClient
```

This example server uses a `PythonCCS` command to cause an image to be generated by the server. The client also then gets the image.

LiveViz provides multiple image combiner types. Any supported type can be used as a parameter to `liveVizPollDeposit`. Valid combiners include: `sum_float_image_data`, `max_float_image_data`, `sum_image_data`, and `max_image_data`.

The differences in Poll Mode may be apparent. There is no callback function which causes the server to generate and deposit an image. Furthermore, a server may generate an image before or after a client has sent a request. The deposit function, therefore is more complicated, as the server will specify information about the image that it is generating. The client will no longer specify the desired size or other configuration options, since the server may generate the image before the client request is available to the server. The `liveVizPollInit` call takes no parameters.

The server should call Deposit with the same global size and combiner type on all of the array elements which correspond to the “this” parameter.

The latest version of liveVizPoll is not backwards compatible with older versions. The old version had some fundamental problems which would occur if a server generated an image before a client requested it. Thus the new version buffers server generated images until requested by a client. Furthermore the client requests are also buffered if they arrive before the server generates the images. Problems could also occur during migration with the old version.

Chapter 7

Multi-phase Shared Arrays Library

The Multiphase Shared Arrays (MSA) library provides a specialized shared memory abstraction in CHARM++ that provides automatic memory management. Explicitly shared memory provides the convenience of shared memory programming while exposing the performance issues to programmers and the “intelligent” ARTS.

Each MSA is accessed in one specific mode during each phase of execution: **read-only** mode, in which any thread can read any element of the array; **write-once** mode, in which each element of the array is written to (possibly multiple times) by at most one worker thread, and no reads are allowed and **accumulate** mode, in which any threads can add values to any array element, and no reads or writes are permitted. A **sync** call is used to denote the end of a phase.

We permit multiple copies of a page of data on different processors and provide automatic fetching and caching of remote data. For example, initially an array might be put in **write-once** mode while it is populated with data from a file. This determines the cache behavior and the permitted operations on the array during this phase. **write-once** means every thread can write to a different element of the array. The user is responsible for ensuring that two threads do not write to the same element; the system helps by detecting violations. From the cache maintenance viewpoint, each page of the data can be over-written on its owning processor without worrying about transferring ownership or maintaining coherence. At the **sync**, the data is simply merged. Subsequently, the array may be **read-only** for a while, thereafter data might be **accumulate**d into it, followed by it returning to **read-only** mode. In the **accumulate** phase, each local copy of the page on each processor could have its accumulations tracked independently without maintaining page coherence, and the results combined at the end of the phase. The **accumulate** operations also include set-theoretic union operations, i.e. appending items to a set of objects would also be a valid **accumulate** operation. User-level or compiler-inserted explicit **prefetch** calls can be used to improve performance.

A software engineering benefit that accrues from the explicitly shared memory programming paradigm is the (relative) ease and simplicity of programming. No complex, buggy data-distribution and messaging calculations are required to access data.

To use MSA in a CHARM++ program:

- build CHARM++ for your architecture, e.g. `net-linux`.
- `cd charm/net-linux/tmp/libs/ck-libs/multiphaseSharedArrays/; make`
- `#include ‘msa/msa.h’` in your header file.
- Compile using `charmcc` with the option “`-module msa`”

The API is as follows: See the example programs in `charm/pgms/charm++/multiphaseSharedArrays`.

Chapter 8

3D FFT Library

8.1 Introduction and Motivation

The 3D FFT library provides an interface to do parallel FFT computation in a scalable fashion.

The parallelization is achieved by splitting the 3D transform into multiple phases. There are two possibilities for doing the splitting: One is dividing the data space (over which the fft is to be performed) into a set of slabs (figure 1). Each slab is essentially a collection of planes). First, 2D FFTs are done over the planes in the slab. Then a distributed 'transform' will send the data to destination so that fft in the third direction is performed. This approach takes two computation phases and one 'transform' phase. The second way for splitting is dividing the data into collections of pencils. First, 1D FFTs are computed over the pencils; then a 'transform' is performed and 1D FFTs are done over second dimension; again a 'transform' is performed and FFTs are computed over the last dimension. So this approach takes three computation phases and two 'transform' phases. In first approach, the parallelism is limited by the number of planes. While in second approach, it's limited by the number of pencils. So the second approach provides finer grained parallelism and it's possible to perform better when the number of processing units is larger than the number of planes.

8.2 Compilation and Execution

To install the FFT library, you will need to have charm++ installed in your system. You can follow the Charm++ manual to do that. Also you will need to have FFTW (version 2.1.5) installed. FFTW can be downloaded from <http://www.fftw.org>.

The FFT library source is at `your-charm-dir/src/libs/ck-libs/fftlib`. Before installation of the library, make sure that the path for FFTW library is consistent with your FFTW installation. Then `cd` to `your-charm-dir/tmp`, and do `'make fftlib'`. To compile a program using the `fftlib`, pass the `'-lfftlib -L(your-fftwlib-dir) -lfftw'` flag to `charmcc`.

8.3 Library Initialization and Data Format

To initialize the library, user will need to construct a data struct and pass it to the library.

For plane-based version, the struct is called: `NormalFFTinfo`. And the constructor of `'NormalFFTinfo'` is defined as:

```
NormalFFTinfo(CProxy_NormalSlabArray &srcProxy, CProxy_NormalSlabArray &destProxy,
              int srcDim[2], int destDim[2], int isSrcArray, complex *dataptr,
              int srcPlanesPerSlab=1, int destPlanesPerSlab=1)
```

Where:

```
CProxy_NormalSlabArray &srcProxy : proxy for source charm array
```

```

CProxy_NormalSlabArray &destProxy : proxy for destination charm array
int srcDim[2] : FFT plane data dimension at source array (*)
int destDim[2] : FFT plane data dimension at destination array ( srcDim[1] must equal to destDim[1])
int isSrcArray : whether this array is source (1) or destination (0)
complex *dataptr : pointer to FFT data
int srcPlanesPerSlab : number of planes in each slab at source array, default value is 1 (**)
int destPlanesPerSlab : number of planes in each slab at destination array, default value is 1

* Data layout : The multi-dimensional FFT data are supposed
to be stored in a contiguous one-dimensional array in
row-major order. For example, in source array, data is
srcPlanesPerSlab planes, each plane is
srcDim[0] rows of size srcDim[1] numbers. Similar
rules apply to destination side.

** Currently, srcPlanesPerSlab/destPlanesPerSlab has to be
the same across all array elements.

*** Total data size can be calculated by:
srcPlanesPerSlab*srcDim[0]*srcDim[1] at source array, and
destPlanesPerSlab*destDim[0]*destDim[1] at destination array

```

For pencil-based version, the struct is called: LineFFTinfo.

```

LineFFTinfo(CProxy_NormalLineArray &xProxy,
            CProxy_NormalLineArray &yProxy,
            CProxy_NormalLineArray &zProxy,
            int size[3], int isSrcArray, complex *dataptr,
            int srcPencilsPerSlab=1, int destPencilsPerSlab=1)

```

where:

```

CProxy_NormalSlabArray &xProxy : proxy for first charm array
CProxy_NormalSlabArray &yProxy : proxy for second charm array
CProxy_NormalSlabArray &zProxy : proxy for third charm array
int size[3] : FFT plane data dimension (*)
int isSrcArray : whether this array is source (1) or intermediate (2) or destination (0)
complex *dataptr : pointer to FFT data
int srcPencilsPerSlab : number of pencils in each slab at source array, default value is 1
int destPencilsPerSlab : number of pencils in each slab at destination array, default value is 1

```

```

*data layout : pencils in the three arrays are of size
size[0]/size[1]/size[2]. And if there is more than one
pencil per slab, the other dimension is the dimension for
pencils in the next array.

```

In both cases, data is deposited by passing in a pointer to the data field, and the pointer will be stored in 'complex *dataptr' in the struct. Memory allocation and initialization of data field needs to be done by user before pointer is passed in. The library doesn't allocate any memory for data field. Also note that FFT's done internally in the library are in-place FFTs, which means that data field will be overwritten with results.

8.4 Library Interfaces

There are two types of interfaces provided by the library: Charm++ based and AMPI based.

8.4.1 Charm++ interface

The Charm++ interface is the raw interface of the library and slightly more difficult to use but gives more flexibility. To use the charm++ based library, user has to create their own charm arrays which derive from predefined arrays in library. By overriding default methods, user can add in additional functions.

For the plane-based library, there are several relevant member functions: *'doFFT'*, *'doIFFT'*, *'doneFFT'* and *'doneIFFT'*. *'doFFT'* and *'doIFFT'* need to be called to start the computation. *'doneFFT'* and *'doneIFFT'* are callback functions, and they need to be inherited.

The sample codes below should shed more light on this. For complete sample programs, refer to file under `your-charm-dir/pgms/charm++/fftdemo/`.

In the sample code below, we will illustrate how to use the plane-based library in 4 steps: initializing the data struct; creating array element; starting the computation and finally ending the computation.

For initializing, a NormalFFTinfo struct will be used. Keep in mind that data storage needs to be allocated and initialized by the user. Since in-place FFT will occur, user should also make duplicate copies of data when needed.

```
main::main(CkArgMsg *m)
{
    ...
    /* Assume FFT of size nx*ny*nz */
    int srcDim[] = ny, nx, destDim[] = nx, nz;

    complex *plane = new complex[nx*ny];
    ... // Initialize FFT data here

    NormalFFTinfo src_fftinfo(srcArray, destArray,
                              srcDim, destDim, true, plane, 1, 1);

    ...
}
```

Next step is to create the charm array:

```
main::main(CkArgMsg *m)
{
    ...
    /* Assume FFT of size nx*ny*nz */
    int srcDim[] = ny, nx, destDim[] = nx, nz;

    /* create the source array */
    srcArray = CProxy_SrcArray::ckNew();
    for (z = 0; z < dim; z+=1) {
        complex *plane = new complex[nx*ny];
        ... // Initialize FFT data here: data needs to be in x-major order

        NormalFFTinfo src_fftinfo(srcArray, destArray,
                                    srcDim, destDim, true, plane, 1, 1);

        // insert one plane object: this contains data of x-y plane at z coordinate
        srcArray(z).insert(src_fftinfo);
    }
}
```

```

        /* destination array will be created in similar fashion */
        ...
    }

```

Following we will start the FFT computation by making a call to `'doFFT()'`. `'doFFT(int id1, int id2)'` takes two inputs: `id1` defines the ID number of the source FFT, while `id2` defines the ID number of the destination FFT. There is a similar method called `'doIFFT()'` to be used to invoke inverse FFTs. In this example, 3 FFT's are done simultaneously by invoking a `'doAllFFT()'` method. And `'doAllFFT()'` is defined as:

```

void SrcArray::doAllFFT() {
    doFFT(0, 0);
    doFFT(1, 1);
    doFFT(2, 2);
}

```

The last step is to get data at destination side. For this purpose, inheritance of method `'doneFFT()'` is defined below. `'doneFFT(int id)'` takes the FFT ID number as input. For inverse FFTs, relevant member function is `'doneIFFT()'`.

```

void destArray::doneFFT(int id) {
    count ++;
    if(count==3) {
        count = 0;
        /* A reduction is induced: this will call the predefined reduction client when all array e
        contribute(sizeof(int), &count, CkReduction::sum_int);
    }
}

```

Next we will demonstrate the usage of pencil-based library in similar steps. First is the initialization of data struct `LineFFTinfo`:

```

main::main()
{
    ...
    /* Assume FFT of size nx*ny*nz */
    int size[] = nx, ny, nz;

    complex *pencil = new complex[nx];
    ... /* Initialize FFT data here */

    LineFFTinfo fftinfo(xlinesProxy, ylinesProxy, zlinesProxy, size, true, pencil);
    ...
}

```

Second is the creation of array:

```

main::main()
{
    ...
    /* Assume FFT of size nx*ny*nz */
    int size[] = nx, ny, nz;

    xlinesProxy = CProxy_myXLines::ckNew();
    for (z = 0; z < sizeZ; z++)
        for (y = 0; y < sizeY; y+=THICKNESS)

```

```

        complex *pencil = new complex[nx];
        ... /* Initialize FFT data here */

        LineFFTinfo fftinfo(xlinesProxy, ylinesProxy, zlinesProxy,
                            size, true, pencil);
        xlinesProxy(y, z).insert(fftinfo);

        xlinesProxy.doneInserting();

        /* ylinesProxy /zlinesProxy are created in similar fashion */
        ...
    }

```

Next is the starting of the computation. A method called `doFirstFFT()` needs to be called. *doFirstFFT(int id, int direction)* takes two parameters: *id* specifies the ID number of the target FFT, *direction* tells whether FFTs is to be done in forward (*direction=1*) or backward (*direction=0*) direction.

```

void myXLines::doAllFFT() {
    doFirstFFT(0, 1);
    doFirstFFT(1, 1);
    doFirstFFT(2, 1);
}

```

Finally, it's the step to finish the FFT at receiver side. In this case, we call the array of destination *myZLines*. Similarly as in the plane-based version, *doneFFT()* is inherited. *doneFFT(int id, int direction)* takes two inputs, which are explained the same as in *doFirstFFT(int id, int direction)*.

```

void myZLines::doneFFT(int id, int direction) {
    count ++;
    if(count==3) {
        count = 0;
        contribute(sizeof(int), &count, CkReduction::sum_int);
    }
}

```

8.4.2 AMPI Interface

The MPI-like interface aims at easy migration of MPI program to use the library. not available in CVS yet. The AMPI interface has five functions:

- `init_fftplib` - initialization of library. This will create charm++ level data structures, prepare for FFT computation.
- `start_fft` - start the FFT.
- `wait_fft` - wait for the FFT to finish.
- `start_ifft` - start the inverse FFT. (similar as `start_fft`)
- `wait_ifft` - wait for the inverse FFT to finish. (similar as `wait_fft`)

(sample code here)

Chapter 9

Matrix Multiplication Library

9.1 Introduction and Motivation

The matrix multiplication library provides a simple way to add the capabilities to do matrix-matrix multiplications to any two-dimensional chare arrays in CHARM++. It is envisioned that in the future this will become part of a larger CHARM++linear algebra (CLA) library.

The matrix multiplication library was designed as a way to add the capability of doing the matrix-matrix multiplication to a user's already-existing 2D chare array. It assumes that the user will evenly distribute their data among the array elements. The library uses bound arrays to access this data locally in order to minimize communication between the user's and the library's arrays. The library provides both a "2D" and a "3D" algorithm to carry out the multiplication. The "2D" algorithm is faster but requires more memory. The "3D" algorithm requires less memory and provides greater opportunity to overlap communication and computation if the user's program has other work that can be done while the multiplication is carried out.

9.2 Compiling and Linking

To use the matrix multiplication library, do

```
charmc CLA_Matrix.ci  
charmc -c CLA_Matrix.C
```

. This will make the necessary `.decl.h` and `.def.h` files as well as the object file for the library.

In the user's main `.ci` file, the user must specify

```
extern module CLA_Matrix;
```

. If not, CHARM++ will complain about missing modules.

The declarations of all classes and functions needed for compilation are in `CLA_Matrix.h`.

For linking the application, a few things are needed. First, the user must link `CLA_Matrix.o` into their application. Second, the user must pass the `-module CkMulticast` to `charmc`. Finally, a BLAS library must be linked to the application. In reality, only the `dgemm` routine (and as a result, usually `xerbla`) is needed. Copies of `dgemm` and `xerbla` which can be used are included in the package. However, it is recommended that you link an optimized BLAS into your application to get higher performance.

9.3 Usage Description

The library performs the operation $C \leftarrow \beta C + \alpha AB$. In order to do this, we first initialize the library telling it to which user array it should bind the chare arrays used to represent A , B , and C . When each element of the user arrays bound to A and B is ready, it makes a library call telling it to start the multiplication. The array bound to C makes a similar call, specifying a callback to be called then the operation is done. As each chunk of C is ready, the library executes the callback.

9.4 Library Interface Details

9.4.1 Library Initialization

To initialize the library, the user should call `make_multiplier`. It has the following signature:

```
int make_multiplier(CLA_Matrix_interface *A, CLA_Matrix_interface *B,
  CLA_Matrix_interface *C, CProxy_ArrayElement bindA,
  CProxy_ArrayElement bindB, CProxy_ArrayElement bindC,
  int M, int K, int N, int m, int k, int n, int strideM, int strideK,
  int strideZ, CkCallback cbA, CkCallback cbB, CkCallback cbC,
  CkGroupID gid, int algorithm);
```

- The `CLA_Matrix_interface` class (described in the following section) is used to interact with the matrix multiplication library. The arguments `A`, `B`, and `C` should point to different instances of objects of this class.
- The arguments `bindA`, `bindB`, and `bindC` should be proxies to which the libraries `A`, `B`, and `C` arrays, respectively, will be bound. There is no restriction as to whether these arrays must all be different or the same.
- The integers `M`, `K`, and `N` refer to the size of the global data matrices. The `A` matrix is of size $M \times K$. The `B` matrix is of size $K \times N$. The `C` matrix is of size $M \times N$.
- The arguments `m`, `k`, and `n` determine how many elements of the global matrices each array element will hold. A “typical” chunk of a matrix which has `M` as a dimension will have `m` elements. If `m` does not divide `M`, the last element of the array along the given dimension will have $M \bmod m$ elements. The above is the same for the `K` and `k` as well as `N` and `n`.
- The striding arguments are used to specify the stride at which elements along a given dimension should be created. For example, if `bindA` is a dense matrix, `strideM` and `strideK` should both be 1. If `bindC` has elements at $(0, 0)$, $(0, 2)$, $(0, 4)$, \dots , `strideN` should be 2. Note that although each of the dimensions is shared by two (potentially) distinct arrays, there is no support for them to have different strides.
- Since the library takes advantage of bound arrays to access memory directly between the user and library objects, it is necessary to make sure the library is properly initialized before it is used. As the `A`, `B`, and `C` arrays finish initializing all their elements, they will make callbacks to `cbA`, `cbB`, and `cbC`, respectively. These can all be the same, but this is not checked, so there will be three calls made to it.
- The `gid` argument is a `CkGroupID` identifier which should be used by the library to do its collective communications.
- The `algorithm` argument specifies which algorithm the library should use to carry out the multiplication. The valid options are `MM_ALG_2D` for the “2D” algorithm, and `MM_ALG_3D` for the “3D” algorithm.

If the initialization success, zero is returned. Otherwise, a negative number is returned.

9.4.2 Performing Multiplications

The user interacts with the library through `CLA_Matrix_interface` objects. Their constructor takes no arguments. A call to `make_multiplier` (described below) will properly initialize them. The class has two methods the user needs to use. First, is the `multiply` method. It has the following signature:

```
void multiply(double alpha, double beta, double *data, void (*fptr) (void *),
  void *usr_data, int x, int y);
```

Each element of the user’s `A`, `B`, and `C` arrays must call this method to perform the multiplication.

- The parameters `alpha` and `beta` correspond to α and β in $C = \beta C + \alpha AB$. The same value must be passed for each of these two parameters at each elements of A , B , and C .
- The parameter `data` specifies where the data to be multiplied is held. This is a two-dimensional array stored in a one-dimensional array. The size of the array is determined at creation time (based on user parameters) and does not have to be specified at this point. For A and B `data` should contain the matrices to be multiplied. For C , it is the destination where the result will be stored (and if $\beta \neq 0$, the data which will be multiplied by β and added to αAB).
- The `fptr` parameter specifies the callback which the library calls (at array C when the computation is complete. When it is called, it passes `usr_data` to it. Only the elements of the C array must specify a non-null value for `fptr`.
- The values of `x` and `y` specify the coordinates of the calling element in the users char array (these should always be `thisIndex.x` and `thisIndex.y`, respectively).

The second function of the `CLA_Matrix_interface` objects is the `sync` method. This should be called at each element of the array when it is ready to migrate. It takes as arguments two integers, `x` and `y`, which should always be `thisIndex.x` and `thisIndex.y`, respectively. Note that migration is currently only supported for the “2D” algorithm.

9.5 Example

A complete example can be found in `projects/matrix_multiply/CLA_Matrix` in CVS. What follows are some of the more important files from that example.

In the users `.ci` file:

```
mainmodule matTest {
  extern module CLA_Matrix; // make sure to include, or get Charm++ errors
  ...
  readonly CProxy_tester dataProxyA;
  readonly CProxy_tester dataProxyB;
  readonly CProxy_tester dataProxyC;
  ...
  mainchare Main {
    ...
    entry void chunk_initiated(); // callback to know library is ready
  };
  ...
  array [2D] tester{
    ...
    entry void multiply(double alpha, double beta); // start multiplication
  };
};
```

In the main function:

```
...
/* create interface objects, data proxies*/
CLA_Matrix_interface matA, matB, matC;

dataProxyA = CProxy_tester::ckNew();
dataProxyB = CProxy_tester::ckNew();
dataProxyC = CProxy_tester::ckNew();
```

```

/* make multicast manager, callback */
CkGroupID gid = CProxy_CkMulticastMgr::ckNew();
CkCallback *cb = new CkCallback(CkIndex_Main::chunk_initiated(), mainProxy);

/* size and stride variable determined earlier, ready to init library */
make_multiplier(&matA, &matB, &matC, dataProxyA, dataProxyB, dataProxyC, M, K,
  N, m, k, n, M_stride, K_stride, N_stride, *cb, *cb, *cb, gid, MM_ALG_2D);

...
/* create tester objects */
for(i = ...)
  for(j = ...)
    dataProxyA(i, j).insert(...);
dataProxyA.doneInserting();
...

```

When the library is ready, tell the tester (user) objects to multiply:

```

void chunk_initiated(){
  /* make sure all three proxies are ready */
  if(++msg_received != 3)
    return;
  dataProxyA.multiply(1, 0);
  dataProxyB.multiply(1, 0);
  dataProxyC.multiply(1, 0);
}

```

The user's objects start the multiplication:

```

void tester::multiply(double alpha, double beta){
  /* "matrix" is the CLA_Matrix_interface object created in main. */
  /* pass "this" as the user_data argument so that we can dereference it
  * in done_cb below. */
  matrix.multiply(alpha, beta, data, tester::done_cb, (void*) this,
  thisIndex.x, thisIndex.y);
}

```

The C tester object will be notified when the multiplication has completed.

```

class tester {
  ...
  static void done_cb(void *obj){
    ((tester*) obj)->round_done();
  }
  void round_done(){
    CkPrintf("[%d %d] has its chunk of C ready.\n", thisIndex.x, thisIndex.y);
    // continue with user's code, doing something useful
    ...
  }
  ...
};

```