Parallel Programming Laboratory
University of Illinois at Urbana-Champaign

Converse and Charm++ Libraries Manual

Version 1.0

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Contents

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

liveViz Library

2.1 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/examples/charm++/wave2d
make
./charmrun ./wave2d +p2 ++server ++server-port 1234
~/ccs_tools/bin/liveViz localhost 1234
```

Use git to obtain a copy of ccs_tools (prior to using liveViz) and build it by:

```
cd ccs_tools;
ant;
```

2.2 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.

2.3 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 colorized 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);
```

2.4 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). You must create the chare array using a CkArrayOptions 'opts' parameter. For instance,

   ```
   CkArrayOptions opts(rows, cols);
   array = CProxy_Type::ckNew(opts);
   ```

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, opts).

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);
```

2.5 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

2.6 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
.../ccs_tools/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 compatable 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.

2.7 Caveats

If you use the old version of "liveVizInit" method that only receives 3 parameters, you will find a known bug caused by how "liveVizDeposit" internally uses a reduction to build the image.

Using that version of the "liveVizInit" method, its contribute call is handled as if it were the chare calling "liveVizDeposit" that actually contributed to the liveViz reduction. If there is any other reduction going on elsewhere in this chare, some liveViz contribute calls might be issued before the corresponding non-liveViz contribute is reached. This would imply that image data would be treated as if were part of the non-liveViz reduction, leading to unexpected behavior potentially anywhere in the non-liveViz code.

Chapter 3

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 it's 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. `netlrts-linux`.
- `cd charm/netlrts-linux/tmp/libs/ck-libs/multiphaseSharedArrays/; make`
- `#include "msa/msa.h"` in your header file.
- Compile using `charmc` with the option "`-module msa`"

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

Chapter 4

3D FFT Library

4.1 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.

4.2 Compilation and Execution

To install the FFT library, you will need to have charm++ installed in you 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 charmc.

4.3 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[0
```

```
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.

4.4 Library Interfaces

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

4.4.1 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 *'doFFT()'* 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 elem
        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);
    }
}
```

4.4.2 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_fftlib - 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 5

TRAM

5.1 Overview

Topological Routing and Aggregation Module is a library for optimization of many-to-many and all-to-all collective communication patterns in Charm++ applications. The library performs topological routing and aggregation of network communication in the context of a virtual grid topology comprising the Charm++ Processing Elements (PEs) in the parallel run. The number of dimensions and their sizes within this topology are specified by the user when initializing an instance of the library.

TRAM is implemented as a Charm++ group, so an *instance* of TRAM has one object on every PE used in the run. We use the term *local instance* to denote a member of the TRAM group on a particular PE.

Most collective communication patterns involve sending linear arrays of a single data type. In order to more efficiently aggregate and process data, TRAM restricts the data sent using the library to a single data type specified by the user through a template parameter when initializing an instance of the library. We use the term *data item* to denote a single object of this datatype submitted to the library for sending. While the library is active (i.e. after initialization and before termination), an arbitrary number of data items can be submitted to the library at each PE.

On systems with an underlying grid or torus network topology, it can be beneficial to configure the virtual topology for TRAM to match the physical topology of the network. This can easily be accomplished using the Charm++ Topology Manager.

The next two sections explain the routing and aggregation techniques used in the library.

5.1.1 Routing

Let the variables j and k denote PEs within an N-dimensional virtual topology of PEs and x denote a dimension of the grid. We represent the coordinates of j and k within the grid as $(j_0, j_1, \ldots, j_{N-1})$ and $(k_0, k_1, \ldots, k_{N-1})$. Also, let

$$f(x, j, k) = \begin{cases} 0, & \text{if } j_x = k_x \\ 1, & \text{if } j_x \neq k_x \end{cases}$$

j and k are *peers* if

$$\sum_{d=0}^{N-1} f(d, j, k) = 1.$$

When using TRAM, PEs communicate directly only with their peers. Sending to a PE which is not a peer is handled inside the library by routing the data through one or more *intermediate destinations* along the route to the *final destination*.

Suppose a data item destined for PE k is submitted to the library at PE j. If k is a peer of j, the data item will be sent directly to k, possibly along with other data items for which k is the final or intermediate destination. If k is not a peer of j, the data item will be sent to an intermediate destination m along the route to k whose index is $(j_0, j_1, \ldots, j_{i-1}, k_i, j_{i+1}, \ldots, j_{N-1})$, where i is the greatest value of x for which $f(x, j, k) = 1$.

Note that in obtaining the coordinates of m from j, exactly one of the coordinates of j which differs from the coordinates of k is made to agree with k. It follows that m is a peer of j, and that using this routing process at m

and every subsequent intermediate destination along the route eventually leads to the data item being received at k. Consequently, the number of messages $F(j,k)$ that will carry the data item to the destination is

$$F(j,k) = \sum_{d=0}^{N-1} f(d,j,k).$$

5.1.2 Aggregation

Communicating over the network of a parallel machine involves per message bandwidth and processing overhead. TRAM amortizes this overhead by aggregating data items at the source and every intermediate destination along the route to the final destination.

Every local instance of the TRAM group buffers the data items that have been submitted locally or received from another PE for forwarding. Because only peers communicate directly in the virtual grid, it suffices to have a single buffer per PE for every peer. Given a dimension d within the virtual topology, let s_d denote its *size*, or the number of distinct values a coordinate for dimension d can take. Consequently, each local instance allocates up to $s_d - 1$ buffers per dimension, for a total of $\sum_{d=0}^{N-1}(s_d - 1)$ buffers. Note that this is normally significantly less than the total number of PEs specified by the virtual topology, which is equal to $\prod_{d=0}^{N-1} s_d$.

Sending with TRAM is done by submitting a data item and a destination identifier, either PE or array index, using a function call to the local instance. If the index belongs to a peer, the library places the data item in the buffer for the peer's PE. Otherwise, the library calculates the index of the intermediate destination using the previously described algorithm, and places the data item in the buffer for the resulting PE, which by design is always a peer of the local PE. Buffers are sent out immediately when they become full. When a message is received at an intermediate destination, the data items comprising it are distributed into the appropriate buffers for subsequent sending. In the process, if a data item is determined to have reached its final destination, it is immediately delivered.

The total buffering capacity specified by the user may be reached even when no single buffer is completely filled up. In that case the buffer with the greatest number of buffered data items is sent.

5.2 Application User Interface

A typical usage scenario for TRAM involves a start-up phase followed by one or more *communication steps*. We next describe the application user interface and details relevant to usage of the library, which normally follows these steps:

1. **Start-up** Creation of a TRAM group and set up of client arrays and groups
2. **Initialization** Calling an initialization function, which returns through a callback
3. **Sending** An arbitrary number of sends using the insertData function call on the local instance of the library
4. **Receiving** Processing received data items through the process function which serves as the delivery interface for the library and must be defined by the user
5. **Termination** Termination of a communication step
6. **Re-initialization** After termination of a communication step, the library instance is not active. However, re-initialization using step 2 leads to a new communication step.

5.2.1 Start-Up

Start-up is typically performed once in a program, often inside the main function of the mainchare, and involves creating an aggregator instance. An instance of TRAM is restricted to sending data items of a single user-specified type, which we denote by dtype, to a single user-specified chare array or group.

Sending to a Group

To use TRAM for sending to a group, a GroupMeshStreamer group should be created. Either of the following two GroupMeshStreamer constructors can be used for that purpose:

```
template<class dtype, class ClientType, class RouterType>
GroupMeshStreamer<dtype, ClientType, RouterType>::
GroupMeshStreamer(int maxNumDataItemsBuffered,
                  int numDimensions,
                  int *dimensionSizes,
                  CkGroupID clientGID,
                  bool yieldFlag = 0,
                  double progressPeriodInMs = -1.0);

template<class dtype, class ClientType, class RouterType>
GroupMeshStreamer<dtype, ClientType, RouterType>::
GroupMeshStreamer(int numDimensions,
                  int *dimensionSizes,
                  CkGroupID clientGID,
                  int bufferSize,
                  bool yieldFlag = 0,
                  double progressPeriodInMs = -1.0);
```

Sending to a Chare Array

For sending to a chare array, an ArrayMeshStreamer group should be created, which has a similar constructor interface to GroupMeshStreamer:

```
template <class dtype, class itype, class ClientType,
          class RouterType>
ArrayMeshStreamer<dtype, itype, ClientType, RouterType>::
ArrayMeshStreamer(int maxNumDataItemsBuffered,
                  int numDimensions,
                  int *dimensionSizes,
                  CkArrayID clientAID,
                  bool yieldFlag = 0,
                  double progressPeriodInMs = -1.0);

template <class dtype, class itype, class ClientType,
          class RouterType>
ArrayMeshStreamer<dtype, itype, ClientType, RouterType>::
ArrayMeshStreamer(int numDimensions,
                  int *dimensionSizes,
                  CkArrayID clientAID,
                  int bufferSize,
                  bool yieldFlag = 0,
                  double progressPeriodInMs = -1.0);
```

Description of parameters:

- maxNumDataItemsBuffered: maximum number of items that the library is allowed to buffer per PE
- numDimensions: number of dimensions in grid of PEs
- dimensionSizes: array of size numDimensions containing the size of each dimension in the grid
- clientGID: the group ID for the client group

- clientAID: the array ID for the client array
- bufferSize: size of the buffer for each peer, in terms of number of data items
- yieldFlag: when true, calls CthYield() after every 1024 item insertions; setting it true requires all data items to be submitted from threaded entry methods. Ensures that pending messages are sent out by the runtime system when a large number of data items are submitted from a single entry method.
- progressPeriodInMs: number of milliseconds between periodic progress checks; relevant only when periodic flushing is enabled (see Section 5.2.5)

Template parameters:

- dtype: data item type
- itype: index type of client chare array (use int for one-dimensional chare arrays and CkArrayIndex for all other index types)
- ClientType: type of client group or array
- RouterType: the routing protocol to be used. The choices are:
  (1) SimpleMeshRouter - original grid aggregation scheme;
  (2) NodeAwareMeshRouter - base node-aware aggregation scheme;
  (3) AggressiveNodeAwareMeshRouter - advanced node-aware aggregation scheme;

5.2.2 Initialization

A TRAM instance needs to be initialized before every communication step. There are currently three main modes of operation, depending on the type of termination used: *staged completion*, *completion detection*, or *quiescence detection*. The modes of termination are described later. Here, we present the interface for initializing a communication step for each of the three modes.

When using completion detection, each local instance of TRAM must be initialized using the following variant of the overloaded init function:

```
template <class dtype, class RouterType>
void MeshStreamer<dtype, RouterType>::
init(int numContributors,
     CkCallback startCb,
     CkCallback endCb,
     CProxy_CompletionDetector detector,
     int prio,
     bool usePeriodicFlushing);
```

Description of parameters:

- numContributors: number of done calls expected globally before termination of this communication step
- startCb: callback to be invoked by the library after initialization is complete
- endCb: callback to be invoked by the library after termination of this communication step
- detector: an inactive CompletionDetector object to be used by TRAM
- prio: Charm++ priority to be used for messages sent using TRAM in this communication step
- usePeriodicFlushing: specifies whether periodic flushing should be used for this communication step

When using staged completion, a completion detector object is not required as input, as the library performs its own specialized form of termination. In this case, each local instance of TRAM must be initialized using a different interface for the overloaded init function:

```
template <class dtype, class RouterType>
void MeshStreamer<dtype, RouterType>::
init(int numLocalContributors,
     CkCallback startCb,
     CkCallback endCb,
     int prio,
     bool usePeriodicFlushing);
```

Note that numLocalContributors denotes the local number of done calls expected, rather than the global as in the first interface of init.

A common case is to have a single chare array perform all the sends in a communication step, with each element of the array as a contributor. For this case there is a special version of init that takes as input the CkArrayID object for the chare array that will perform the sends, precluding the need to manually determine the number of client chares per PE:

```
template <class dtype, class RouterType>
void MeshStreamer<dtype, RouterType>::
init(CkArrayID senderArrayID,
     CkCallback startCb,
     CkCallback endCb,
     int prio,
     bool usePeriodicFlushing);
```

The init interface for using quiescence detection is:

```
template <class dtype, class RouterType>
void MeshStreamer<dtype, RouterType>::init(CkCallback startCb,
                                           int prio);
```

After initialization is finished, the system invokes startCb, signaling to the user that the library is ready to accept data items for sending.

5.2.3 Sending

Sending with TRAM is done through calls to insertData and broadcast.

```
template <class dtype, class RouterType>
void MeshStreamer<dtype, RouterType>::
insertData(const dtype& dataItem,
           int destinationPe);

template <class dtype, class itype, class ClientType,
          class RouterType>
void ArrayMeshStreamer<dtype, itype, ClientType, RouterType>::
insertData(const dtype& dataItem,
           itype arrayIndex);

template <class dtype, class RouterType>
void MeshStreamer<dtype, RouterType>::
broadcast(const dtype& dataItem);
```

- dataItem: reference to a data item to be sent
- destinationPe: index of destination PE
- arrayIndex: index of destination array element

Broadcasting has the effect of delivering the data item:

- once on every PE involved in the computation for GroupMeshStreamer
- once for every array element involved in the computation for ArrayMeshStreamer

5.2.4 Receiving

To receive data items sent using TRAM, the user must define the process function for each client group and array:

```
void process(const dtype  &ran);
```

Each item is delivered by the library using a separate call to process on the destination PE. The call is made locally, so process should not be an entry method.

5.2.5 Termination

Flushing and termination mechanisms are used in TRAM to prevent deadlock due to indefinite buffering of items. Flushing works by sending out all buffers in a local instance if no items have been submitted or received since the last progress check. Meanwhile, termination detection is used to send out partially filled buffers at the end of a communication step after it has been determined that no additional items will be submitted.

Currently, three means of termination are supported: staged completion, completion detection, and quiescence detection. Periodic flushing is a secondary mechanism which can be enabled or disabled when initiating one of the primary mechanisms.

Termination typically requires the user to issue a number of calls to the done function:

```
template <class dtype, class RouterType>
void MeshStreamer<dtype, RouterType>::
done(int numContributorsFinished = 1);
```

When using completion detection, the number of done calls that are expected globally by the TRAM instance is specified using the numContributors parameter to init. Safe termination requires that no calls to insertData or broadcast are made after the last call to done is performed globally. Because order of execution is uncertain in parallel applications, some care is required to ensure the above condition is met. A simple way to terminate safely is to set numContributors equal to the number of senders, and call done once for each sender that is done submitting items.

In contrast to using completion detection, using staged completion involves setting the local number of expected calls to done using the numLocalContributors parameter in the init function. To ensure safe termination, no insertData or broadcast calls should be made on any PE where done has been called the expected number of times.

Another version of init for staged completion, which takes a CkArrayID object as an argument, provides a simplified interface in the common case when a single chare array performs all the sends within a communication step, with each of its elements as a contributor. For this version of init, TRAM determines the appropriate number of local contributors automatically. It also correctly handles the case of PEs without any contributors by immediately marking those PEs as having finished the communication step. As such, this version of init should be preferred by the user when applicable.

Staged completion is not supported when array location data is not guaranteed to be correct, as this can potentially violate the termination conditions used to guarantee successful termination. In order to guarantee correct location data in applications that use load balancing, Charm++ must be compiled with -DCMK_GLOBAL_LOCATION_UPDATE, which has the effect of performing a global broadcast of location data for chare array elements that migrate during load balancing. Unfortunately, this operation is expensive when migrating large numbers of elements. As an alternative, completion detection and quiescence detection modes will work properly without the global location update mechanism, and even in the case of anytime migration.

When using quiescence detection, no end callback is used, and no done calls are required. Instead, termination of a communication step is achieved using the quiescence detection framework in Charm++, which supports passing a callback as parameter. TRAM is set up such that quiescence will not be detected until all items sent in the current communication step have been delivered to their final destinations.

The choice of which termination mechanism to use is left to the user. Using completion detection mode is more convenient when the global number of contributors is known, while staged completion is easier to use if

the local number of contributors can be determined with ease, or if sending is done from the elements of a chare array. If either mode can be used with ease, staged completion should be preferred. Unlike the other mechanisms, staged completion does not involve persistent background communication to determine when the global number of expected done calls is reached. Staged completion is also generally faster at reaching termination due to not being dependent on periodic progress checks. Unlike completion detection, staged completion does incur a small bandwidth overhead (4 bytes) for every TRAM message, but in practice this is more than offset by the persistent traffic incurred by completion detection.

Periodic flushing is an auxiliary mechanism which checks at a regular interval whether any sends have taken place since the last time the check was performed. If not, the mechanism sends out all the data items buffered per local instance of the library. The period is specified by the user in the TRAM constructor. A typical use case for periodic flushing is when the submission of a data item B to TRAM happens as a result of the delivery of another data item A sent using the same TRAM instance. If A is buffered inside the library and insufficient data items are submitted to cause the buffer holding A to be sent out, a deadlock could arise. With the periodic flushing mechanism, the buffer holding A is guaranteed to be sent out eventually, and deadlock is prevented. Periodic flushing is required when using the completion detection or quiescence detection termination modes.

5.2.6 Re-initialization

A TRAM instance that has terminated cannot be used for sending more data items until it has been re-initialized. Re-initialization is achieved by calling init, which prepares the instance of the library for a new communication step. Re-initialization is useful for iterative applications, where it is often convenient to have a single communication step per iteration of the application.

5.2.7 Charm++ Registration of Templated Classes

Due to the use of templates in TRAM, the library template instances must be explicitly registered with the Charm++ runtime by the user of the library. This must be done in the .ci file for the application, and typically involves three steps.

For GroupMeshStreamer template instances, registration is done as follows:

- Registration of the message type:

```
message MeshStreamerMessage<dtype>;
```

- Registration of the base aggregator class

```
group MeshStreamer<dtype, RouterType>;
```

- Registration of the derived aggregator class

```
group GroupMeshStreamer<dtype, ClientType, RouterType>;
```

For ArrayMeshStreamer template instances, registration is done as follows:

- Registration of the message type:

```
message MeshStreamerMessage<ArrayDataItem<dtype, itype> >;
```

- Registration of the base aggregator class

```
group MeshStreamer<ArrayDataItem<dtype, itype>,
                   RouterType>;
```

- Registration of the derived aggregator class

```
group ArrayMeshStreamer<dtype, itype, ClientType,
                        RouterType>;
```

5.3 Example

For example code showing how to use TRAM, see examples/charm++/TRAM and tests/charm++/streamingAllToAll in the Charm++ repository.

Chapter 6

GPU Manager Library

GPU Manager is a task offload and management library for efficient use of CUDA-enabled GPUs in Charm++ applications. Compared to direct use of CUDA (through issuing kernel invocation and GPU data transfer calls in user code) GPU Manager provides the following advantages:

1. Automatic management and synchronization of tasks
2. Automatic overlap of data transfer and kernel invocation for concurrent tasks
3. A simplified work flow mechanism using CkCallback to return to user code after completion of each work request
4. Reduced synchronization overhead through centralized management of all GPU tasks

6.1 Building GPU Manager

GPU Manager is not included by default when building Charm++. In order to use GPU Manager, the user must build Charm++ using the `CUDA` option, e.g.

```
./build charm++ netlrts-linux-x86_64 cuda -j8
```

Building GPU Manager requires an installation of the CUDA toolkit on the system.

6.2 Overview and Work Flow

GPUs are throughput-oriented devices with peak computational capabilities that greatly surpass equivalent-generation CPUs but with limited control logic that constraints them to use as accelerator devices controlled by code executing on the CPU.

The GPU's dependence on the CPU for dispatch and synchronization of coarse-grained data transfer and kernel execution has traditionally required programmers to either (a) halt the execution of work on the CPU whenever issuing GPU work to simplify synchronization or (b) issue GPU work asynchronously and carefully manage and synchronize concurrent GPU work in order to ensure satisfactory progress and good performance. Further, the latter option becomes significantly more difficult in the context of a parallel program with numerous concurrent objects that all issue kernel and data transfer calls to the same GPU.

The Charm++ GPU Manager is a library designed to address this issue by automating the management of GPUs. Users of GPU Manager define *work requests* that specify the GPU kernel and any data transfer operations required before and after completion of the kernel. The system controls the execution of the work requests submitted by all the chares on a particular processor. This allows it to effectively manage execution of work requests and overlap CPU-GPU data transfer with kernel execution. In steady-state operation, GPU Manager overlaps kernel execution of one work request with data transfer out of GPU memory for a preceding work request and the data transfer into GPU memory for a subsequent work request. This approach avoids blocking the CUDA DMA engine by only submitting data transfers when they are ready to execute. When using GPU Manager, the user does not need to poll for completion of GPU operations. The system manages execution of a work request

throughout its life cycle and returns control to the user upon completion of a work request through a CkCallback object specified by the user per work request. Another advantage of using GPU Manager is that the system polls only for a handful of currently executing operations, which avoids the problem of multiple chares all polling the GPU when using CUDA streams directly. GPU Manager has options for recording profiling data for kernel execution and data transfer which can be visualized using the Charm++ Projections profiler.

6.2.1 Execution Model and Progress Engine

Like any Charm++ application, programs using GPU Manager typically consist of a large number of concurrently executing objects. Each object executes code in response to active messages received from some object within the parallel run, during which it can send its own active messages or issue one or more work requests to the GPU Manager for asynchronous execution. Work requests are always submitted to the local GPU Manager instance at the processing element where the call is issued. Incoming GPU work requests are simply copied into the GPU Manager's scheduling queue, at which point the library returns and the caller can continue with other work.

Charm++ employs a message driven programming model. This includes a runtime system scheduler that is triggered every time a method finishes execution. Under typical CPU-only execution the scheduler examines the queue of incoming messages and selects one based on priority and location in the queue. In a `CUDA` build of Charm++, the scheduler is also programmed to periodically invoke the GPU Manager progress engine.

GPU Manager contains a queue of all pending work requests. When its progress function is called, GPU Manager determines whether pending GPU work has completed since the last time the progress function was called, and whether additional work requests can begin executing. A `workRequest` undergoes the following stages during its execution:

1. Device memory allocation and data transfer from host to device
2. Kernel execution
3. Data transfer back to host from device
4. Invocation of a callback function (specified in the `workRequest`)

Based on the instructions contained in each work request, the GPU Manager will allocate the required buffers in GPU global memory and issue asynchronous CUDA data transfer operations directly. In order to execute kernels, the GPU Manager calls the `runKernel` function that must be defined by the user. This function specifies the CUDA kernel call for your work request.

Under steady state execution with multiple concurrent work requests, as one `workRequest` progresses to the execution stage, GPU Manager will initiate the data transfer for the second `workRequest` in the queue, and so on.

In a typical application, the work request definition, kernel run functions, CUDA kernel definitions, and code for submission of work requests would all go in a `.cu` file that is compiled with `nvcc` separately from the other files (e.g. `.C`, `.ci`) in the Charm++ application. We make a function call to `createWorkRequest` from a .C file to create and enqueue the workRequest. The various resulting object files of the application are then to be linked together into the final executable.

6.3 API

Using GPU Manager involves:

1. Defining CUDA kernels as in a regular CUDA application
2. Defining work requests and their callback functions
3. Defining the `void runMyKernel(workRequest *wr, cudaStream_t kernelStream, void **deviceBuffers)` functions, used by the GPU Manager to issue a kernel call based on the kernel identifier defined in the work request
4. Submitting work requests to the GPU Manager

6.3.1 Work Request

`workRequest` is a simple structure which contains the necessary parameters for CUDA kernel execution along with some additional members for automating data transfer between the host and the device. A work request consists of the following data members:

dim3 dimGrid - a triple which defines the grid structure for the kernel; in the example below `dimGrid.x` specifies the number of blocks. `dimGrid.y` and `dimGrid.z` are unused.

dim3 dimBlock - a triple defining each block's structure; specifies the number of threads in up to three dimensions.

int smemSize - the number of bytes in shared memory to be dynamically allocated per block for kernel execution.

int nBuffers - number of buffers used by the work request.

dataInfo *bufferInfo - array of `dataInfo` structs containing buffer information for the execution of the work request. This array must be of size `nBuffers`, e.g.

```
codewr->bufferInfo = (dataInfo *) malloc(wr->nBuffers * sizeof(dataInfo))
```

We will explain the contents of `dataInfo` struct later.

void *callbackFn - a pointer to a `CkCallback` object specified by the user; executed after the kernel and memory transfers have finished.

const char *traceName - A short identifier used for tracing and logging.

function runKernel - A user defined host function to run the kernel. We will pass this function three parameters:

workRequest - The workrequest being run.

kernelStream - The cuda stream to run the kernel in.

deviceBuffers - An array of device pointers, indexed by bufferID.

int state - the stage of a `workRequest`'s execution, set and used internally by the GPU Manager

void *userData - may be used to pass scalar values to kernel calls, such as the size of an array.

6.3.1.1 dataInfo

int bufferID - the ID of a buffer in the runtime system's buffer table. May be specified by the user if direct control over the buffer space is desired. Otherwise, if it is set to a negative value, the GPU Manager will assign a valid buffer ID.

int transferToDevice, transferFromDevice - flags to indicate if the buffer should be transferred to the device prior to the execution of a kernel, and/or transferred out after the kernel

int freeBuffer - a flag to indicate if the device buffer memory should be freed after execution of `workRequest`.

void *hostBuffer - pointer to host data buffer. In order to allow asynchronous memory transfer and data computation on device this buffer must be allocated from page-locked memory.

```
void *hostBuffer = hapi_poolMalloc(size);
```

This returns the buffer of required size from the GPU Manager's pool of pinned memory on the host. Direct allocation of pinned memory (e.g. using `cudaMallocHost`) is discouraged, as it will block the CPU until pending GPU work has finished executing. The user must add the `-DGPU_MEMPOOL` flag while compiling CUDA files. This is required to enable fetching of page-locked memory from GPU Manager. You may add it with your `NVCC_FLAGS`.

size_t size - size of buffer in bytes.

6.3.1.2 Work Request Example

Here is an example method for creating a `workRequest` of the addition of two vectors A and B.

```
#include "wr.h"
#define BLOCK_SIZE 256
void createWorkRequest(int vectorSize, float *h_A, float *h_B, float **h_C, int myIndex, CkCallback *cb)
{
    dataInfo *info;
    workRequest *vecAdd = new workRequest;
    int size = vectorSize * sizeof(float);

    vecAdd->dimGrid.x = (vectorSize - 1) / BLOCK_SIZE + 1;
    vecAdd->dimBlock.x = BLOCK_SIZE;
    vecAdd->smemSize = 0;
    vecAdd->nBuffers = 3;
    vecAdd->bufferInfo = new dataInfo[vecAdd->nBuffers];

    /* Buffer A */
    info = &(vecAdd->bufferInfo[0]);

    /* The Buffer ID will be given by the API,
       or it can be given by the user. */
    info->bufferID = -1;

    info->transferToDevice = YES;
    info->transferFromDevice = NO;
    info->freeBuffer = YES;

    /* This fetches the pinned host memory already allocated by API,
       required for asynchronous data transfers. */
    info->hostBuffer = hapi_poolMalloc(size);

    /* Copy the data to the workRequest's buffer. */
    memcpy(info->hostBuffer, h_A, size);

    info->size = size;

    /* Buffer B will be same as A.*/

    /* Buffer C */
    info = &(vecAdd->bufferInfo[2]);
    info->transferFromDevice = YES;
    info->hostBuffer = hapi_poolMalloc(size)

    / * We change the address to the address returned by the API
        to read the copied result */
    *h_C = (float *)info->hostBuffer;

    /* a CkCallback pointer */
    vecAdd->callbackFn = cb;

    vecAdd->traceName = "add";

    /* kernel run function */
    vecAdd->runKernel = run_add;
```

```
    vecAdd->userData = new int;
    memcpy(vecAdd->userData, &vectorSize, sizeof(int));

    /* enqueue the workRequest in the workRequestQueue.
    wrQueue is declared by our API during the init phase for every processor. */
    enqueue(wrQueue, vecAdd);
}
```

6.3.2 Writing Kernels

Writing a kernel is unchanged from normal CUDA programs. Kernels are written in one (or more) .cu files. Here is an example of `vectorAdd.cu`. The full example can be found in `examples/charm++/cuda/vectorAdd/`.

```
__global__ void vecAdd(float *a, float *b, float *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x * blockDim.x + threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}
```

6.3.3 Launching Kernels

Kernel launches are identical to regular kernel launches in normal CUDA programs, run in a small dedicated function.

```
void run_add(workRequest *wr, cudaStream_t kernelStream, void **deviceBuffers)
{
    printf("Add KERNEL \n");
/*
 *  devBuffers is declared by our API during the init phase on every processor.
 *  It jumps to the correct array index with the help of bufferID,
 *  which is supplied by the API or user.
 */
    vecAdd<<< wr->dimGrid, wr->dimBlock, wr->smemSize, kernelStream>>>
        ((float *) deviceBuffers[wr->bufferInfo[0].bufferID],
         (float *) deviceBuffers[wr->bufferInfo[1].bufferID],
         (float *) deviceBuffers[wr->bufferInfo[2].bufferID],
         *((int *) wr->userData));
}
```

6.4 Compiling

As mentioned earlier, there are no changes to the `.ci` and `.C` files. Therefore there are no changes in compiling them. CUDA code, however, must be compiled using `nvcc`. You can use the following example makefile to compile a .cu file. More example codes can be found in the `examples/charm++/cuda` directory.

```
CUDA_LEVEL=35

NVCC = /usr/local/cuda/bin/nvcc

NVCC_FLAGS = -O3 -c -use_fast_math -DGPU_MEMPOOL

NVCC_FLAGS += -arch=compute_$(CUDA_LEVEL) -code=sm_$(CUDA_LEVEL)
```

```
NVCC_INC = -I/usr/local/cuda/include

CHARMINC = -I${CHARMDIR}/include

LD_LIBS= -lcublas

all: vectorAdd
    $(NVCC) $(NVCC_FLAGS) $(NVCC_INC) $(CHARMINC) -o vectorAddCU.o vectorAdd.cu
```

GPU Manager also supports `CuBLAS` or other GPU libraries in exactly the same way. Call `CuBLAS` or the other GPU library directly from a kernel run function; creating the `workRequest` works the same as any other kernel.

6.5 Debugging

A few useful things for debugging:

1. Enabling the `GPU_MEMPOOL_DEBUG` flag (using `-DGPU_MEMPOOL_DEBUG`) during execution prints debug statements, including when buffers are allocated and freed.

2. When using `++debug`, add the debugging flags `-g` and `-G` during compilation.