

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

1	Introduction	4
2	liveViz Library	5
2.1	Introduction	5
2.2	How to use liveViz with Charm++ program	5
2.3	Format of deposit image	6
2.4	liveViz Initialization	6
2.5	Compilation	7
2.6	Poll Mode	7
2.7	Caveats	8
3	Multi-phase Shared Arrays Library	9
4	3D FFT Library	10
4.1	Introduction and Motivation	10
4.2	Features	10
4.3	Compilation and Execution	10
4.4	Library Interface	11
5	TRAM	14
5.1	Overview	14
5.1.1	Routing	14
5.1.2	Aggregation	15
5.2	Application User Interface	15
5.2.1	Start-Up	15
5.2.2	Initialization	17
5.2.3	Sending	18
5.2.4	Receiving	19
5.2.5	Termination	19
5.2.6	Re-initialization	20
5.2.7	Charm++ Registration of Templated Classes	20
5.3	Example	21
6	GPU Manager Library	22
6.1	Overview	22
6.2	Building GPU Manager	22
6.3	Using GPU Manager	23

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 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);
```

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 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.

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 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. `netlrts-linux`.
- `cd charm/netlrts-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 4

3D FFT Library

The previous 3D FFT library has been deprecated and replaced with this new 3D FFT library. The new 3D FFT library source can be downloaded with following command: `git clone https://charm.cs.illinois.edu/gerrit/libs/fft`

4.1 Introduction and Motivation

The 3D Charm-FFT library provides an interface to do parallel 3D FFT computation in a scalable fashion. The parallelization is achieved by splitting the 3D transform into three phases, using 2D decomposition. 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.

This library allows users to create multiple instances of the library and perform concurrent FFTs using them. Each of the FFT instances run in background as other parts of user code execute, and a callback is invoked when FFT is complete.

4.2 Features

Charm-FFT library provides the following features:

- *2D-decomposition*: Users can define fine-grained 2D-decomposition that increases the amount of available parallelism and improves network utilization.
- *Cutoff-based smaller grid*: The data grid may have a cut off. Charm-FFT improves performance by avoiding communication and computation of the data beyond the cutoff.
- *User-defined mapping of library objects*: The placement of objects that constitute the library instance can be defined by the user based on the application's other concurrent communication and placement of other objects.
- *Overlap with other computational work*: Given the callback-based interface and Charm++'s asynchrony, the FFTs are performed in the background while other application work can be done in parallel.

4.3 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. Then, ensure that FFTW3 is installed. FFTW3 can be downloaded from <http://www.fftw.org>. The Charm-FFT library source can be downloaded with following command: `git clone https://charm.cs.illinois.edu/gerrit/libs/fft`

Inside of Charm-FFT directory, you will find *Makefile.default*. Copy this file to *Makefile.common*, change the copy's variable *FFT3_HOME* to point your FFTW3 installation and *CHARM_DIR* to point your Charm++ installation then run *make*. To use Charm-FFT library in an application, add the line *extern module fft_ Charm;*

to its charm interface (.ci) file and include `fft_charm.h` and `fftw3.h` in relevant C files. Finally to compile the program, pass `-lfft_charm` and `-lfftw3` as arguments to `charm.c`.

4.4 Library Interface

To use Charm-FFT interface, the user must start by calling `Charm_createFFT` with following parameters.

```
Charm_createFFT(N_x, N_y, N_z, z_x, z_y, y_x, y_z, x_zy, cutoff, hmati, fft_type, CkCallback);
```

Where:

```
int N_x : X dimension of FFT calculation
int N_y : Y dimension of FFT calculation
int N_z : Z dimension of FFT calculation
int z_x : X dimension of Z pencil chare array
int z_y : Y dimension of Z pencil chare array
int y_x : X dimension of Y pencil chare array
int y_z : Z dimension of Y pencil chare array
int x_zy: A dimension of X pencil chare array
double cutoff: Cutoff of FFT grid
double *hmati: Hamiltonian matrix representing cutoff
FFT_TYPE: Type of FFT to perform. Either CC for complex-to-complex or RC for real-complex
CkCallback: A Charm++ entry method for callback upon the completion of library initialization
```

This creates necessary proxies (Z,Y,X etc) for performing FFT of size $N_x \times N_y * N_z$ using 2D chare arrays (pencils) of size $n_y \times n_x$ (ZPencils), $n_z \times n_x$ (YPencils), and $n_x \times n_y$ (XPencils). When done, calls `myCallback` which should receive `CProxy_fft2d id` as a unique identifier for the newly created set of proxies.

An example of Charm-FFT initialization using `Charm_createFFT`:

```
// .ci

extern module fft_charm;

mainchare Main {
    entry Main(CkArgMsg *m);
}

group Driver {
    entry Driver(FFT_Type fft_type);
    entry void proxyCreated(idMsg *msg);
    entry void fftDone();
}

// .C

Main::Main(CkArgMsg *m) {

    ...
    /* Assume FFT of size N_x, N_y, N_z */
    FFT_Type fft_type = CC

    Charm_createFFT(N_x, N_y, N_z, z_x, z_y, y_x, y_z, x_zy, cutoff, hmati
, fft_type, CkCallback(CkIndex_Driver::proxyCreated(NULL), driverProxy));
}

Driver::proxyCreated(idMsg *msg) {
```

```

    CProxy_fft2d fftProxy = msg-> id;
    delete msg;
}

```

In this example, an entry method *Driver::proxyCreated* will be called when an FFT instance has been created. Using the newly received proxy, the user can identify whether a local PE has XPencils and/or ZPencils.

```

void Driver::proxyCreated(idMsg *msg) {
    CProxy_fft2d fftProxy = msg->id;

    delete msg;

    bool hasX = Charm_isOutputPE(fftProxy),
        hasZ = Charm_isInputPE(fftProxy);

    ...
}

```

Then, the grid's dimensions on a PE can be acquired by using *Charm_getOutputExtents* and *Charm_getInputExtents*.

```

if (hasX) {
    Charm_getOutputExtents(gridStart[MY_X], gridEnd[MY_X],
                          gridStart[MY_Y], gridEnd[MY_Y],
                          gridStart[MY_Z], gridEnd[MY_Z],
                          fftProxy);
}

if (hasZ) {
    Charm_getInputExtents(gridStart[MY_X], gridEnd[MY_X],
                        gridStart[MY_Y], gridEnd[MY_Y],
                        gridStart[MY_Z], gridEnd[MY_Z],
                        fftProxy);
}

for(int i = 0; i < 3; i++) {
    gridLength[i] = gridEnd[i] - gridStart[i];
}

```

With the grid's dimension, the user must allocate and set the input and output buffers. In most cases, this is simply the product of the three dimensions, but for real-to-complex FFT calculation, FFTW-style storage for the input buffers is used (as shown below).

```

dataSize = gridLength[MY_X] * gridLength[MY_Y] * gridLength[MY_Z];

if (hasX) {
    dataOut = (complex*) fftw_malloc(dataSize * sizeof(complex));

    Charm_setOutputMemory((void*) dataOut, fftProxy);
}

if (hasZ) {
    if (fftType == RC) {
        // FFTW style storage
        dataSize = gridLength[MY_X] * gridLength[MY_Y] * (gridLength[MY_Z]/2 + 1);
    }
}

```

```
    dataIn = (complex*) fftw_malloc(dataSize * sizeof(complex));

    Charm_setInputMemory((void*) dataIn, fftProxy);
}
```

Then, from *PE0*, start the forward or backward FFT, setting the entry method *fftDone* as the callback function that will be called when the FFT operation is complete.

For forward FFT

```
if (CkMyPe() == 0) {
    Charm_doForwardFFT(CkCallback(CkIndex_Driver::fftDone(), thisProxy), fftProxy);
}
```

For backward FFT

```
if (CkMyPe() == 0) {
    Charm_doBackwardFFT(CkCallback(CkIndex_Driver::fftDone(), thisProxy), fftProxy);
}
```

The sample program to run a backward FFT can be found in *Your_Charm_FFT_Path/tests/simple_tests*

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 $(j_0, j_1, \dots, j_{N-1})$ and $(k_0, k_1, \dots, k_{N-1})$. 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 $(j_0, j_1, \dots, j_{i-1}, k_i, j_{i+1}, \dots, j_{N-1})$, 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

6.1 Overview

GPU Manager is a task offload and management library for efficient use of CUDA-enabled GPUs in Charm++ applications. CUDA code can be integrated in Charm++ just like any C++ program, but the resulting performance is likely to be far from ideal. This is because overdecomposition, a core concept of Charm++, creates fine-grained objects and tasks which causes problems on the GPU.

GPUs are throughput-oriented devices with peak computational capabilities that greatly surpass equivalent-generation CPUs but with limited control logic. This currently constrains them to be used as accelerator devices controlled by code on the CPU. Traditionally, programmers have had 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 progress and good performance. The latter option, which is practically a requirement in Charm++ to preserve asynchrony, becomes significantly more difficult with numerous concurrent objects that issue kernels and data transfers to the GPU.

The Charm++ programmer is strongly recommended to use CUDA streams to mitigate this problem, by assigning separate streams to chares. This allows operations in different streams to execute concurrently. It should be noted that concurrent data transfers are limited by the number of DMA engines, and current GPUs have one per direction of the transfer (host-to-device, device-to-host). The concurrent kernels feature of CUDA allows multiple kernels to execute simultaneously on the device, as long as resources are available.

An important factor of performance with using GPUs in Charm++ is that the CUDA API calls invoked by chares to offload work should be non-blocking. The chare that just offloaded work to the GPU should yield the PE so that other chares waiting to be executed can do so. Unfortunately, many CUDA API calls used to wait for completion of GPU work, such as `cudaStreamSynchronize` and `cudaDeviceSynchronize`, are blocking. Since the PEs in Charm++ are implemented as persistent kernel-level threads mapped to each CPU core, this means other chares cannot run until the GPU work completes and the blocked chare finishes executing. To resolve this issue, GPU Manager provides Hybrid API (HAPI) to the Charm++ user, which includes new functions to implement the non-blocking features and a set of wrappers to the CUDA runtime API functions. The non-blocking API allows the user to specify a Charm++ callback upon offload which will be invoked when the operations in the CUDA stream are complete.

6.2 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.

6.3 Using GPU Manager

As explained in the Overview section, use of CUDA streams is strongly recommended. This allows kernels offloaded by chares to execute simultaneously on the GPU, which boosts performance if the kernels are small enough for the GPU to be able to allocate resources.

In a typical Charm++ application using CUDA, `.C` and `.ci` files would contain the Charm++ code, whereas a `.cu` file would include the definition of CUDA kernels and a function that serves as an entry point from the Charm++ application to use GPU capabilities. CUDA/HAPI calls for data transfers or kernel invocations would be placed inside this function, although they could also be put in a `.C` file provided that the right header files are included (`<cuda_runtime.h>` or `"hapi.h"`). The user should make sure that the CUDA kernel definitions are compiled by `nvcc`, however.

After the necessary data transfers and kernel invocations, `hapiAddCallback` would be placed where typically `cudaStreamSynchronize` or `cudaDeviceSynchronize` would go. This informs the runtime that a chare has offloaded work to the GPU, allowing the provided Charm++ callback to be invoked once it is complete. The non-blocking API has the following prototype:

```
void hapiAddCallback(cudaStream_t stream, CkCallback* callback);
```

Other HAPI calls:

```
void hapiCreateStreams();
cudaStream_t hapiGetStream();
```

```
cudaError_t hapiMalloc(void** devPtr, size_t size);
cudaError_t hapiFree(void* devPtr);
cudaError_t hapiMallocHost(void** ptr, size_t size);
cudaError_t hapiFreeHost(void* ptr);
```

```
void* hapiPoolMalloc(int size);
void hapiPoolFree(void* ptr);
```

```
cudaError_t hapiMemcpyAsync(void* dst, const void* src, size_t count,
                             cudaMemcpyKind kind, cudaStream_t stream = 0);
```

```
hapiCheck(code);
```

`hapiCreateStreams` creates as many streams as the maximum number of concurrent kernels supported by the GPU device. `hapiGetStream` hands out a stream created by the runtime in a round-robin fashion. The `hapiMalloc` and `hapiFree` functions are wrappers to the corresponding CUDA API calls, and `hapiPool` functions provides memory pool functionalities which are used to obtain/free device memory without interrupting the GPU. `hapiCheck` is used to check if the input code block executes without errors. The given code should return `cudaError_t` for it to work.

Example Charm++ applications using CUDA can be found under `examples/charm++/cuda`. Codes under `#ifndef USE_WR` use the `hapiWorkRequest` scheme, which is now deprecated.