

Parallel Programming Laboratory
University of Illinois at Urbana-Champaign

Charisma Manual

Charisma is an orchestration language authored by Laxmikant Kalé, Mark Hills, Chao Huang and Pritish Jetley. It was developed by Chao Huang and is currently maintained by Pritish Jetley.

Version 1.0

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

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

1. The Software is intellectual property owned by Illinois, and all right, title and interest, including copyright, remain with Illinois. Illinois grants, and Licensee hereby accepts, a restricted, non-exclusive, non-transferable license to use the Software for academic, research and internal business purposes only, e.g. not for commercial use (see Clause 7 below), without a fee.
2. Licensee may, at its own expense, create and freely distribute complimentary works that interoperate with the Software, directing others to the PPL server (<http://charm.cs.uiuc.edu>) to license and obtain the Software itself. Licensee may, at its own expense, modify the Software to make derivative works. Except as explicitly provided below, this License shall apply to any derivative work as it does to the original Software distributed by Illinois. Any derivative work should be clearly marked and renamed to notify users that it is a modified version and not the original Software distributed by Illinois. Licensee agrees to reproduce the copyright notice and other proprietary markings on any derivative work and to include in the documentation of such work the acknowledgement:

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

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

3. Except as expressly set forth in this Agreement, THIS SOFTWARE IS PROVIDED "AS IS" AND ILLINOIS MAKES NO REPRESENTATIONS AND EXTENDS NO WARRANTIES OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO WARRANTIES OR MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY PATENT, TRADEMARK, OR OTHER RIGHTS. LICENSEE ASSUMES THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE AND/OR ASSOCIATED MATERIALS. LICENSEE AGREES THAT UNIVERSITY SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, OR INCIDENTAL DAMAGES WITH RESPECT TO ANY CLAIM BY LICENSEE OR ANY THIRD PARTY ON ACCOUNT OF OR ARISING FROM THIS AGREEMENT OR USE OF THE SOFTWARE AND/OR ASSOCIATED MATERIALS.
4. Licensee understands the Software is proprietary to Illinois. Licensee agrees to take all reasonable steps to insure that the Software is protected and secured from unauthorized disclosure, use, or release and will treat it with at least the same level of care as Licensee would use to protect and secure its own proprietary computer programs and/or information, but using no less than a reasonable standard of care. Licensee agrees to provide the Software only to any other person or entity who has registered with Illinois. If licensee is not registering as an individual but as an institution or corporation each member of the institution or corporation who has access to or uses Software must agree to and abide by the terms of this license. If Licensee becomes aware of any unauthorized licensing, copying or use of the Software, Licensee shall promptly notify Illinois in writing. Licensee expressly agrees to use the Software only in the manner and for the specific uses authorized in this Agreement.
5. By using or copying this Software, Licensee agrees to abide by the copyright law and all other applicable laws of the U.S. including, but not limited to, export control laws and the terms of this license. Illinois shall have the right to terminate this license immediately by written notice upon Licensee's breach of, or non-compliance with, any terms of the license. Licensee may be held legally responsible for any copyright infringement that is caused or encouraged by its failure to abide by the terms of this license. Upon termination, Licensee agrees to destroy all copies of the Software in its possession and to verify such destruction in writing.
6. The user agrees that any reports or published results obtained with the Software will acknowledge its use by the appropriate citation as follows:

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

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

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

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

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

Electronic documents will include a direct link to the official CHARM++ page at <http://charm.cs.uiuc.edu/>

7. Commercial use of the Software, or derivative works based thereon, REQUIRES A COMMERCIAL LICENSE. Should Licensee wish to make commercial use of the Software, Licensee will contact Illinois (kale@cs.uiuc.edu) to negotiate an appropriate license for such use. Commercial use includes:
 - (a) integration of all or part of the Software into a product for sale, lease or license by or on behalf of Licensee to third parties, or
 - (b) distribution of the Software to third parties that need it to commercialize product sold or licensed by or on behalf of Licensee.
8. Government Rights. Because substantial governmental funds have been used in the development of CHARM++/CONVERSE, any possession, use or sublicense of the Software by or to the United States government shall be subject to such required restrictions.
9. CHARM++/CONVERSE is being distributed as a research and teaching tool and as such, PPL encourages contributions from users of the code that might, at Illinois' sole discretion, be used or incorporated to make the basic operating framework of the Software a more stable, flexible, and/or useful product. Licensees who contribute their code to become an internal portion of the Software agree that such code may be distributed by Illinois under the terms of this License and may be required to sign an "Agreement Regarding Contributory Code for CHARM++/CONVERSE Software" before Illinois can accept it (contact kale@cs.uiuc.edu for a copy).

UNDERSTOOD AND AGREED.

Contact Information:

The best contact path for licensing issues is by e-mail to kale@cs.uiuc.edu or send correspondence to:

Prof. L. V. Kale
Dept. of Computer Science
University of Illinois
201 N. Goodwin Ave
Urbana, Illinois 61801 USA
FAX: (217) 333-3501

Contents

1	Introduction	3
2	Charisma Syntax	3
2.1	Orchestration Code	3
2.1.1	Header Section	3
2.1.2	Declaration Section	3
2.1.3	Orchestration Section	4
2.2	Sequential Code	6
2.2.1	Sequential Files	6
2.2.2	Producing and Consuming Functions	6
2.2.3	Miscellaneous Issues	7
3	Building and Running a Charisma Program	7
4	Support for Library Module	8
5	Writing Module Library	8
6	Using Module Library	8
7	Using Load Balancing Module	9
7.1	Coding	9
7.2	Compiling and Running	9
8	Handling Sparse Object Arrays	9
A	Example: Jacobi 1D	10

1 Introduction

This manual describes Charisma, an orchestration language for migratable parallel objects. Charisma can be downloaded from the CVS repository hosted on `charm.cs.uiuc.edu`:

```
cvs co orchestration
```

2 Charisma Syntax

A Charisma program is composed of two parts: the orchestration code in a `.or` file, and sequential user code in C/C++ form.

2.1 Orchestration Code

The orchestration code in the `.or` file can be divided into two part. The header part contains information about the program, included external files, defines, and declaration of parallel constructs used in the code. The orchestration section is made up of statements that forms a global control flow of the parallel program. In the orchestration code, Charisma employs a macro dataflow approach; the statements produce and consume values, from which the control flows can be organized, and messages and method invocations generated.

2.1.1 Header Section

The very first line should give the name of the Charisma program with the `program` keyword.

```
program jacobi
```

The `program` keyword can be replaced with `module`, which means that the output program is going to be a library module instead of a stand-alone program. Please refer to Section 4 for more details.

Next, the programmer can include external code files in the generated code with keyword `include` with the filename without extension. For example, the following statement tells the Charisma compiler to look for header file “particles.h” to be included in the generated header file “jacobi.h” and to look for C/C++ code file “particles.[C—cc—cpp—cxx—c]” to be included in the generated C++ code file “jacobi.C”.

```
include particles;
```

It is useful when there are source code that must precede the generated parallel code, such as basic data structure declaration.

After the `include` section is the `define` section, where environmental variables can be defined for Charisma. For example, to tell Charisma to generate additional code to enable the load balancing module, the programmer needs to define “ldb” in the orchestration code. Please refer to Section 7 for details.

2.1.2 Declaration Section

Next comes the declaration section, where classes, objects and parameters are declared. A Charisma program is composed of multiple sets of parallel objects which are organized by the orchestration code. Different sets of objects can be instantiated from different class types. Therefore, we have to specify the class types and object instantiation. Also we need to specify the parameters (See Section 2.1.3) to use in the orchestration statements.

A Charisma program or module has one “MainChare” class, and it does not require explicit instantiation since it is a singleton. The statement to declare MainChare looks like this:

```
class JacobiMain : MainChare;
```

For object arrays, we first need to declare the class types inherited from 1D object array, 2D object array, etc, and then instantiate from the class types. The dimensionality information of the object array is given in a pair of brackets with each dimension size separated by a comma.

```
class JacobiWorker : ChareArray1D;
obj workers : JacobiWorker[N];

class Cell : ChareArray3D;
obj cells : Cell[M,M,M];
```

Note that key word “class” is for class type derivation, and “obj” is for parallel object or object array instantiation. The above code segment declares a new class type `JacobiWorker` which is a 1D object array, (and the programmer is supposed to supply sequential code for it in files “`JacobiWorker.h`” and “`JacobiWorker.C`” (See Section 2.2 for more details on sequential code). Object array “workers” is instantiated from “`JacobiWorker`” and has 16 elements.

The last part is orchestration parameter declaration. These parameters are used only in the orchestration code to connect input and output of orchestration statements, and their data type and size is declared here. More explanation of these parameters can be found in Section 2.1.3.

```
param lb : double[N];
param rb : double[N];
```

With this, “lb” and “rb” are declared as parameters of that can be “connected” with local variables of double array with size of 512.

2.1.3 Orchestration Section

In the main body of orchestration code, the programmer describes the behavior and interaction of the elements of the object arrays using orchestration statements.

- **Foreach Statement**

The most common kind of parallelism is the invocation of a method across all elements in an object array. Charisma provides a *foreach* statement for specifying such parallelism. The keywords `foreach` and `end-foreach` forms an enclosure within which the parallel invocation is performed. The following code segment invokes the entry method `compute` on all the elements of array `myWorkers`.

```
foreach i in workers
  workers[i].compute();
end-foreach
```

- **Publish Statement and Produced/Consumed Parameters**

In the orchestration code, an object method invocation can have input and output (consumed and produced) parameters. Here is an orchestration statement that exemplifies the input and output of this object methods `workers.produceBorders` and `workers.compute`.

```
foreach i in workers
  (lb[i], rb[i]) <- workers[i].produceBorders();
  workers[i].compute(lb[i+1], rb[i-1]);

  (+error) <- workers[i].reduceData();
end-foreach
```

Here, the entry method `workers[i].produceBorders` produces (called *published* in Charisma) values of `lb[i]`, `rb[i]`, enclosed in a pair of parentheses before the publishing sign “<-”. In the second statement, function `workers[i].compute` consumes values of `lb[i+1]`, `rb[i-1]`, just like normal function parameters. If a reduction operation is needed, the reduced parameter is marked with a “+” before it, like the `error` in the third statement.

A entry method can have arbitrary number of published (produced and reduced) values and consumed values. In addition to basic data types, each of these values can also be an object of arbitrary type. The values published by `A[i]` must have the index `i`, whereas values consumed can have the index `e(i)`, which is an index expression in the form of `i±c` where `c` is a constant. Although we have used different symbols (`p` and `q`) for the input and the output variables, they are allowed to overlap.

The parameters are produced and consumed in the program order. Namely, a parameter produced in an early statement will be consumed by the next consuming statement, but will no longer be visible to any consuming statement after a subsequent statement producing the same parameter in program order. Special rules involving loops are discussed later with loop statement.

- **Overlap Statement**

Complicated parallel programs usually have concurrent flows of control. To explicitly express this, Charisma provides a `overlap` keyword, whereby the programmer can fire multiple overlapping control flows. These flows may contain different number of steps or statements, and their execution should be independent of one another so that their progress can interleave with arbitrary order and always return correct results.

```

overlap
{
  foreach i in workers1
    (lb[i], rb[i]) <- workers1[i].produceBorders();
  end-foreach
  foreach i in workers1
    workers1[i].compute(lb[i+1], rb[i-1]);
  end-foreach
}
{
  foreach i in workers2
    (lb[i], rb[i]) <- workers2[i].compute(lb[i+1], rb[i-1]);
  end-foreach
}
end-overlap

```

This example shows an `overlap` statement where two blocks in curly brackets are executed in parallel. Their execution join back to one at the end mark of `end-overlap`.

- **Loop Statement**

Loops are supported with `for` statement and `while` statement. Here are two examples.

```

for iter = 0 to MAX_ITER
  workers.doWork();
end-for

```

```

while (err > epsilon)
  (+err) <- workers.doWork();
  MainChare.updateError(err);
end-while

```

The loop condition in `for` statement is independent from the main program; It simply tells the program to repeat the block for so many times. The loop condition in `while` statement is actually updated in the `MainChare`. In the above example, `err` and `epsilon` are both member variables of class `MainChare`, and can be updated as the example shows. The programmer can active the “autoScalar” feature by including a “`define autoScalar;`” statement in the orchestration code. When `autoScalar` is enabled, Charisma will find all the scalars in the `.or` file, and create a local copy in the `MainChare`. Then every time the scalar is published by a statement, an update statement will automatically be inserted after that statement. The only thing that the programmer needs to do is to initialize the local scalar with a proper value.

Rules of connecting produced and consumed parameters concerning loops are natural. The first consuming statement will look for values produced by the last producing statement before the loop, for the first iteration. The last producing statement within the loop body, for the following iterations. At the last iteration, the last produced values will be disseminated to the code segment following the loop body. Within the loop body, program order holds.

```

for iter = 1 to MAX_ITER
  foreach i in workers
    (lb[i], rb[i]) <- workers[i].compute(lb[i+1], rb[i-1]);
  end-foreach
end-for

```

One special case is when one statement’s produced parameter and consumed parameter overlaps. It must be noted that there is no dependency within the same `foreach` statement. In the above code segment, the values consumed `lb[i]`, `rb[i]` by `worker[i]` will not come from its neighbors in this iteration. The rule is that the consumed values always originate from previous `foreach` statements or `foreach` statements from a previous loop iteration, and the published values are visible only to following `foreach` statements or `foreach` statements in following loop iterations.

- **Scatter and Gather Operation**

A collection of values produced by one object may be split and consumed by multiple object array elements for a scatter operation. Conversely, a collection of values from different objects can be gathered to be consumed by one object.

```

foreach i in A
  (points[i,*]) <- A[i].f(...);
end-foreach
foreach k,j in B
  (...) <- B[k,j].g(points[k,j]);
end-foreach

```

A wildcard dimension “*” in `A[i].f()`’s output `points` specifies that it will publish multiple data items. At the consuming side, each `B[k,j]` consumes only one point in the data, and therefore a scatter communication will be generated from A to B. For instance, `A[1]` will publish data `points[1,0..N-1]` to be consumed by multiple array objects `B[1,0..N-1]`.

```

foreach i,j in A
  (points[i,j]) <- A[i,j].f(...);
end-foreach
foreach k in B
  (...) <- B[k].g(points[*,k]);
end-foreach

```

Similar to the scatter example, if a wildcard dimension “*” is in the consumed parameter and the corresponding published parameter does not have a wildcard dimension, there is a gather operation generated from the publishing statement to the consuming statement. In the following code segment, each `A[i,j]` publishes a data point, then data points from `A[0..N-1,j]` are combined together to for the data to be consumed by `B[j]`.

Many communication patterns can be expressed with combination of orchestration statements. For more details, please refer to PPL technical report 06-18, “Charisma: Orchestrating Migratable Parallel Objects”.

Last but not least, all the orchestration statements in the `.or` file together form the dependency graph. According to this dependency graph, the messages are created and the parallel program progresses. Therefore, the user is advised to put only parallel constructs that are driven by the data dependency into the orchestration code. Other elements such as local dependency should be coded in the sequential code.

2.2 Sequential Code

2.2.1 Sequential Files

The programmer supplies the sequential code for each class as necessary. The files should be named in the form of class name with appropriate file extension. The header file is not really an ANSI C header file. Instead, it is the sequential portion of the class’s declaration. Charisma will generate the class declaration from the orchestration code, and incorporate the sequential portion in the final header file. For example, if a molecular dynamics simulation has the following classes (as declared in the orchestration code):

```

class MDMain : MainChare;
class Cell : ChareArray3D;
class CellPair : ChareArray6D;

```

The user is supposed to prepare the following sequential files for the classes: `MDMain.h`, `MDMain.C`, `Cell.h`, `Cell.C`, `CellPair.h` and `CellPair.C`, unless a class does not need sequential declaration and/or definition code. Please refer to the example in the Appendix.

For each class, a member function “`void initialize(void)`” can be defined and the generated constructor will automatically call it. This saves the trouble of explicitly call initialization code for each array object.

2.2.2 Producing and Consuming Functions

The C/C++ source code is nothing different than ordinary sequential source code, except for the producing/consuming part. For consumed parameters, a function treat them just like normal parameters passed in. To handle produced parameters, the sequential code needs to do two special things. First, the function should have extra parameter for output parameters. The parameter type is keyword `outport`, and the parameter name is the same as appeared in the orchestration code. Second, in the body of the function, the keyword `produce` is used to connect the orchestration parameter and the local variables whose value will be sent out, in a format of a function call, as follows.

```
produce(produced_parameter, local_variable[, size_of_array]);
```

When the parameter represents a data array, we need the additional `size_of_array` to specify the size of the data array.

The dimensionality of an orchestration parameter is divided into two parts: its dimension in the orchestration code, which is implied by the dimensionality of the object arrays the parameter is associated, and the local dimensionality, which is declared in the declaration section. The orchestration dimension is not explicitly declared anywhere, but it is derived from the object arrays. For instance, in the 1D Jacobi worker example, “lb” and “rb” has the same orchestration dimensionality of workers, namely 1D of size [16]. The local dimensionality is used when the parameter is associated with local variables in sequential code. Since “lb” and “rb” are declared to have the local type and dimension of “double [512]”, the producing statement should connect it with a local variable of “double [512]”.

```
void JacobiWorker::produceBorders(output lb, output rb){  
    . . .  
    produce(lb, localLB, 512);  
    produce(rb, localRB, 512);  
}
```

Special cases of the produced/consumed parameters involve scatter/gather operations. In scatter operation, since an additional dimension is implied in the produced parameter, we the `local_variable` should have additional dimension equal to the dimension over which the scatter is performed. Similarly, the input parameter in gather operation will have an additional dimension the same size of the dimension of the gather operation.

For reduction, one additional parameter of type `char[]` is added to specify the reduction operation. Built-in reduction operations are “+” (sum), “*” (product), “<” (minimum), “>” (maximum) for basic data types. For instance the following statements takes the sum of all local value of `result` and for output in `sum`.

```
reduce(sum, result, '+');
```

If the data type is a user-defined class, then you might use the function or operator defined to do the reduction. For example, assume we have a class called “Force”, and we have an “add” function (or a “+” operator) defined.

```
Force& Force::add(const Force& f);
```

In the reduction to sum all the local forces, we can use

```
reduce(sumForces, localForce, "add");
```

2.2.3 Miscellaneous Issues

In sequential code, the user can access the object’s index by a keyword “thisIndex”. The index of 1-D to 6-D object arrays are:

```
1D: thisIndex  
2D: thisIndex.{x,y}  
3D: thisIndex.{x,y,z}  
4D: thisIndex.{w,x,y,z}  
5D: thisIndex.{v,w,x,y,z}  
6D: thisIndex.{x1,y1,z1,x2,y2,z2}
```

3 Building and Running a Charisma Program

There are two steps to build a Charisma program: generating Charm++ program from orchestration code, and building the Charm++ program.

1) Charisma compiler, currently named `orchc`, is used to compile the orchestration code (.or file) and integrate sequential code to generate a Charm++ program. The resultant Charm++ program usually consists of the following code files: Charm++ Interface file ([modulename].ci), header file ([modulename].h) and C++ source code file ([modulename].C). The command for this step is as follows.

```
> orchc [modulename].or
```

2) Charm++ compiler, `charmcc`, is used to parse the Charm++ Interface (.ci) file, compile C/C++ code, and link and build the executable. The typical commands are:

```
> charmcc [modulename].ci
> charmcc [modulename].C -c
> charmcc [modulename].o -o pgm -language charm++
```

Running the Charisma program is the same as running a Charm++ program, using Charm++'s job launcher `charmrun`. (On some platforms like CSE's Turing Cluster, use the customized job launcher `rjq` or `rj`.)

```
> charmrun pgm +p4
```

Please refer to Charm++'s manual and tutorial for more details of building and running a Charm++ program.

4 Support for Library Module

Charisma is capable of producing library code for reuse with another Charisma program. We explain this feature in the following section.

5 Writing Module Library

The programmer uses the keyword `module` instead of `program` in the header section of the orchestration code to tell the compiler that it is a library module. Following keyword `module` is the module name, then followed by a set of configuration variables in a pair parentheses. The configuration variables are used in creating instances of the library, for such info as problem size.

Following the first line, the library's input and output parameters are posted with keywords `inparam` and `outparam`.

```
module FFT3D(CHUNK, M, N);
inparam indata;
outparam outdata1,outdata2;
```

The body of the library is not very different from that of a normal program. It takes input parameters and produces out parameters, as posted in the header section.

6 Using Module Library

To use a Charisma module library, the programmer first needs to create an instance of the library. There are two steps: including the module and creating an instance.

```
use FFT3D;
library f1 : FFT3D(CHUNK=10,M=10,N=100);
library f2 : FFT3D(CHUNK=8,M=8,N=64);
```

The keyword `use` and the module name includes the module in the program, and the keyword `library` creates an instance with the instance name, followed by the module name with value assignment of configuration variables. These statements must appear in the declaration section before the library instance can be used in the main program's orchestration code.

Invoking the library is like calling a publish statement; the input and output parameters are the same, and the object name and function name are replaced with the library instance name and the keyword `call` connected with a colon.

```
(f1_outdata[*]) <- f1:call(f1_indata[*]);
```

Multiple instances can be created out of the same module. Their execution can interleave without interfering with one another.

7 Using Load Balancing Module

7.1 Coding

To activate load balancing module and prepare objects for migration, there are 3 things that needs to be added in Charisma code.

First, the programmer needs to inform Charisma about the load balancing with a "define ldb;" statement in the header section of the orchestration code. This will make Charisma generates extra Charm++ code to do load balancing such as PUP methods.

Second, the user has to provide a PUP function for each class with sequential data that needs to be moved when the object migrates. When choosing which data items to pup, the user has the flexibility to leave the dead data behind to save on communication overhead in migration. The syntax for the sequential PUP is similar to that in a Charm++ program. Please refer to the load balancing section in Charm++ manual for more information on PUP functions. A typical example would look like this in user's sequential .C file:

```
void JacobiWorker::sequentialPup(PUP::er& p){
    p|myLeft; p|myRight; p|myUpper; p|myLower;
    p|myIter;
    PUPArray(p, (double *)localData, 1000);
}
```

Thirdly, the user will make the call to invoke load balancing session in the orchestration code. The call is `AtSync()`; and it is invoked on all elements in an object array. The following example shows how to invoke load balancing session every 4th iteration in a for-loop.

```
for iter = 1 to 100
    // work work
    if(iter % 4 == 0) then
        foreach i in workers
            workers[i].AtSync();
        end-foreach
    end-if
end-for
```

If a while-loop is used instead of for-loop, then the test-condition in the `if` statement is a local variable in the program's `MainChare`. In the sequential code, the user can maintain a local variable called `iter` in `MainChare` and increment it every iteration.

7.2 Compiling and Running

Unless linked with load balancer modules, a Charisma program will not perform actual load balancing. The way to link in a load balancer module is adding `-module EveryLB` as a link-time option.

At run-time, the load balancer is specified in command line after the `+balancer` option. If the balancer name is incorrect, the job launcher will automatically print out all available load balancers. For instance, the following command uses `RotateLB`.

```
> ./charmrun ./pgm +p16 +balancer RotateLB
```

8 Handling Sparse Object Arrays

In Charisma, when we declare an object array, by default a dense array is created with all the elements populated. For instance, when we have the following declaration in the orchestration code, an array of $N \times N \times N$ is created.

```
class Cell : ChareArray3D;
obj cells : Cell[N,N,N];
```

There are certain occasions when the programmer may need sparse object arrays, in which not all elements are created. An example is neighborhood force calculation in molecular dynamics application. We have a 3D array of `Cell` objects to hold the atom coordinates, and a 6D array of `CellPair` objects to perform pairwise force calculation between neighboring cells. In this case, not all elements in the 6D array of `CellPair` are

necessary in the program. Only those which represent two immediately neighboring cells are needed for the force calculation. In this case, Charisma provides flexibility of declaring a sparse object array, with a `sparse` keyword following the object array declaration, as follows.

```
class CellPair : ChareArray6D;
obj cellpairs : CellPair[N,N,N,N,N,N],sparse;
```

Then the programmer is expected to supply a sequential function with the name `getIndex_ARRAYNAME` to generate a list of selected indices of the elements to create. As an example, the following function essentially tells the system to generate all the $N \times N \times N \times N \times N \times N$ elements for the 6D array.

```
void getIndex_cellpairs(CkVec<CkArrayIndex6D>& vec){
  int i,j,k,l,m,n;
  for(i=0;i<N;i++)
    for(j=0;j<N;j++)
      for(k=0;k<N;k++)
        for(l=0;l<N;l++)
          for(m=0;m<N;m++)
            for(n=0;n<N;n++)
              vec.push_back(CkArrayIndex6D(i,j,k,l,m,n));
}
```

A Example: Jacobi 1D

Following is the content of the orchestration file `jacobi.or`.

```
program jacobi

class JacobiMain : MainChare;
class JacobiWorker : ChareArray1D;
obj workers : JacobiWorker[M];
param lb : double[N];
param rb : double[N];

begin
  for iter = 1 to MAX_ITER
    foreach i in workers
      (lb[i], rb[i]) <- workers[i].produceBorders();
      workers[i].compute(lb[i+1], rb[i-1]);
    end-foreach
  end-for
end
```

The class `JacobiMain` does not need any sequential code, so the only sequential code are in `JacobiWorker.h` and `JacobiWorker.C`. Note that `JacobiWorker.h` contains only the sequential portion of `JacobiWorker`'s declaration.

```
#define N 512
#define M 16

int currentArray;
double localData[2][M][N];
double localLB[N];
double localRB[N];
int myLeft,myRight,myUpper,myLower;

void initialize();
void compute(double lghost[], double rghost[]);
void produceBorders(outport lb,outport rb);
double abs(double d);
```

Similarly, the sequential C code will be integrated into the generated C file. Below is part of the sequential C code taken from `JacobiWorker.C` to show how consumed parameters (`rghost` and `lghost` in `JacobiWorker::compute`) and produced parameters (`lb` and `rb` in `JacobiWorker::produceBorders`) are handled.

```
void JacobiWorker::compute(double rghost[], double lghost[]){
    /* local computation for updating elements*/
}

void JacobiWorker::produceBorders(output lb, output rb){
    produce(lb,localData[currentArray][myLeft],myLower-myUpper+1);
    produce(rb,localData[currentArray][myRight],myLower-myUpper+1);
}
```

The user compile these input files with the following command:

```
> orchc jacobi.or
```

The compiler generates the parallel code for sending out messages, organizing flow of control, and then it looks for sequential code files for the classes declared, namely `JacobiMain` and `JacobiWorker`, and integrates them into the final output: `jacobi.h`, `jacobi.C` and `jacobi.ci`, which is a Charm++ program and can be built the way a Charm++ program is built.