Chapter 5

Messages

Thus far, we have encountered Charm++ programs demonstrating inter-object communication through *marshalled* parameters. In this scheme the Charm++ interface (`.ci`) translator uses the signature of an entry method to generate C++ code at the source and target of invocations of the entry method. At the sender, the generated code packs the input parameteres of the invocation into a buffer. This buffer is communicated to the receiver chare as a message, whereupon generated code unpacks the arguments from the received buffer and passes them into a local method invocation on the recipient.

This style of parameter marshalling gives programmers the facility of specifying communication through method invocations. The associated syntax is both familiar and convenient to C++ programmers. However, for reasons of efficiency, it is sometimes beneficial to directly specify the contents of the message buffer communicated between a sender and a recipient chare. This is demonstrated in the example below.

Consider a simplistic parallel program that computes the product C of two square input matrices A and B, of dimensions $N \times N$. Suppose that the tiles of A and B are decomposed over elements of W, such that tiles $A[i,j]$ of A and $B[i,j]$ of B are available at $W[i,j]$. Here, $i,j \in \{1..N/T\}$, where T is the dimension of each square tile. Chare array element $W[i,j]$ computes the matrix products $P_k = A[i,k] \times B[k,j]$ for all $k \in \{1..N/T\}$. Then, $C[i,j]$ can be obtained as the sum $C[i,j] = \sum_k P_k$. To compute an individual product P_r in the sum, $W[i,j]$ must await the receipt of a row broadcasts by $W[i,r]$, giving it $A[i,r]$, and column broadcasts by $W[r,j]$ giving it $B[r,j]$. Since these tiles arrive in different messages, in the parameter-marshalled version of the program, we would have to copy the tile received first until we both of them have been received. On the other hand, if we were given access to the underlying messages themselves, we could simply save a *pointer* to the message received first, and perform the tile multiplication when the second one is received. This would result in less copying overhead, yielding a more efficient parallel program.

Implicit in the discussion above was the fact that once a parameter-marshalled entry

method finishes, the runtime system discards the memory buffer underlying the marshalled parameters. Therefore, in order to access received parameters in a different entry method invocation, they must be saved via copying. In contrast, the runtime system does *not* discard the memory buffer passed into an entry method that is invoked with a message. It is up to the programmer to manage the memory associated with the input message at the receiving end. This fact can be used to improve performance in parallel programs.

In this chapter, we will see how objects can communicate with each other by directly specifying the contents of the messages that they exchange.

5.1 Fixed-Size Messages

Charm++ lets programmers use two types of messages in their programs: those whose size is fixed, and can be determined at compile-time, and those with variable sizes, which depend on the values of variables at run-time. We discuss fixed-size messages first, using two examples, namely a *ping-pong* program and Gauss-Seidel relaxation for linear system solution.

5.1.1 A Ping-Pong Program

In this subsection, we will develop a simple ping-pong program, which involves two chare-array elements. The first of these chares creates a message that encapsulates a single integer field, which represents the number of ping-pong iterations remaining. This is set by the first chare to a predetermined value, after which the message is sent to the second chare. This is the *ping* message, upon receipt of which the second chare examines the integer field sent to it and responds to the first chare with a *pong* message. When it receives the *pong* message, the first chare decrements the number of iterations remaining, thereby completing one round of computation. If there is a non-zero number of *ping-pong* iterations remaining, the next round is initiated, and so on. The Charm++ interface code for this program is presented in Figure 5.1.

```
1 mainmodule pingpong {
2   message StartMsg;
3   message PingPongMsg;
4   ...
5 };
```

Figure 5.1: Excerpt form the CI file for the ping-pong program. Message types are declared through the `message` keyword.

Note the use of the `message` keyword in code above. This declares the types `StartMsg`

and PingPongMsg to be types of message. The types StartMsg and PingPongMsg are then declared as in Figure 5.2. We do not include the declarations of the main chare or the one-dimensional chare array Worker, the members of which participate in the *ping-pong* operation.

```
#include "pingpong.decl.h"

struct StartMsg : public CMessage_StartMsg {
  int totalIterations;
  StartMsg(int i) : totalIterations(i) {}
};

struct PingPongMsg : public CMessage_PingPongMsg {
  int left;
  PingPongMsg(int i) : left(i) {}
};
```

Figure 5.2: Declaration of message types.

Note that a class (or a struct) that is to be passed as a message must inherit from a translator-generated base class. For a message type TYPE, the generated class will bear the name CMessage_TYPE. In the example, the base class for StartMsg is CMessage_StartMsg, and that for PingPongMsg is CMessage_PingPongMsg. Now, let us see how these user-defined types can be used to achieve inter-object communication through asynchronous sending of messages.

```
Main::Main(CkArgMsg *m){
  CProxy_Worker workerProxy = CProxy_Worker::ckNew(2);
  workerProxy[0].start(new StartMsg(atoi(m->argv[1])));
  delete m;
}
```

Figure 5.3: Definition of the Main::Main constructor for the *ping-pong* program. The main chare creates a chare array with two members, and sends a message to the first of these members, thereby initiating the *ping-pong* computation.

First, we examine the main chare, which is a singleton of type Main. The constructor of this class, where control first enters user code, is shown in Figure 5.3. Note that the constructor itself receives a system-defined message of type CkArgMsg. The argc and argv

fields of this message provide the number of command-line arguments passed to the program, and the array of C strings holding the arguments themselves, respectively. For instance, once the two-element object array of type `Worker` is created in line 2, `m->argv[1]` is used to obtain the number of *ping-pong* rounds to perform. The number of rounds is embedded within a *StartMsg* (*cf.* the StartMsg constructor in Figure 5.2.) This message is sent to the first worker, `worker[0]` by passing it as an input argument to the method `CProxy_Worker::start` of the corresponding proxy, `workerProxy[0]`. Note that, given the asynchronous nature of method invocations in Charm++, no guarantee is provided as to *when* `Worker::start` will be invoked on `worker[0]` after `CProxy_Worker::start` has been called on `workerProxy[0]`. In particular, it cannot be assumed that `worker[0]` will have received the initiating message when control returns from `CProxy_Worker::start`. On line 4, the received `CkArgMsg` is `delete`d, since management of memory associated with messages is the responsibility of the programmer.

```
1 void Worker::start(StartMsg *m){
2   PingPongMsg *pm = new PingPongMsg(m->totalIterations);
3   CkPrintf("[%d] ping (%d)\n", thisIndex, pm->left);
4   thisProxy[1].ping(pm);
5   delete m;
6 }
```

Figure 5.4: The `Worker::start` method, where the *ping-pong* control flow begins. This method is invoked on `worker[0]`, which creates a *ping* message and sends it to `worker[1]`.

Figure 5.4 shows the definition of the `Worker::start` method, which is the end-point for the message sent by the main chare to initiate the computation (*cf.* Figure 5.3, above.) A message of type `PingPongMsg` is created on line 1, with a number of iterations given by the `totalIterations` field of the `StartMsg` received by the method as input. On line 3, this newly constructed message is sent to `worker[1]`, by (asynchronously) invoking the `Worker::ping` method on the associated proxy.

To round out the example, we present the code for `Worker::ping` and `Worker::pong` in Figures 5.5 and 5.6. Recall that method `Worker::ping` was invoked *on* `worker[1]` *by* `worker[0]`. Chare array object `worker[1]` responds to this *ping* by sending `worker[0]` a *pong* via the `CProxy_Worker::pong` method of the corresponding proxy (referred to as `thisProxy[0]`, on line 3.) Notice that instead of creating a new `PingPongMsg` and `delete`ing the input message in `Worker::ping`, we reuse the input by passing it to the `CProxy_Worker::pong` method. This is a perfectly valid Charm++ idiom and moreover, more convenient than parameter marshalling, the previously mentioned alternative. A word of caution, however: whereas it is legal to repeatedly invoke a *parameter-marshalled* entry method with the same buffers

```
void Worker::ping(PingPongMsg *m){
  CkPrintf("[%d] pong (%d)\n", thisIndex, m->left);
  thisProxy[0].pong(m);
}
```

Figure 5.5: The `Worker::ping` method, which initiates a round of *ping-pong*, and is invoked by object `worker[0]` on `worker[1]`.

in memory, one cannot reuse the memory allocated for a previously sent message. One can obtain a valid buffer for a message through the `new` operator, as shown in Figure 5.3.

```
void Worker::pong(PingPongMsg *m){
  if(--m->left > 0){
    CkPrintf("[%d] ping (%d)\n", thisIndex, m->left);
    thisProxy[1].ping(m);
  }
  else{
    delete m;
    CkExit();
  }
}
```

Figure 5.6: The `Worker::pong` method, which is invoked on `worker[0]` by `worker[1]`, and constitutes `worker[1]`'s response to the `ping` it received from `worker[0]`.

Object `worker[0]` receives the *pong* response in `Worker::pong` (Figure 5.6.) It decrements the number of remaining *ping-pong* rounds by decrementing the `left` field of the input `PingPongMsg`. If this number is non-zero, `worker[0]` initiates the next round of the *ping-pong* computation. Once again, we reuse the memory buffer associated with the input message in the asynchronous method invocation `thisProxy[1].ping(m)`. On the other hand, if we have completed the designated number of *ping-pong* iterations (set in `Worker::start`, Figure 5.4), we `delete` the input message and signal the end of the parallel program through `CkExit()`.

It is worth noting that one of the common sources of overheads in programs (especially, fine-grained computations) is the overhead of memeory allocation. This overhead is reduced here because we reuse the message.

```
void updateAll(){
  for(int I = 1; I < matrixDimY-1; I++)
   for(int J = 1; J < matrixDimX-1; J++)
    D[I,J] = AVG(D[I,J],D[I,J-1],D[I,J-1],D[I,J+1],D[I+1,J]);
}
```

Figure 5.7: Ordering the computations in the Gauss-Seidel method.

5.1.2 Gauss-Seidel Relaxation

Now that we have gained an understanding of the mechanics of fixed-size messages, we will use them to create a more substantive parallel program than the *ping-pong* computation in the previous subsection. The *Gauss-Seidel relaxation* procedure is used to solve differential equations using the method of differences. We are given a large matrix of points, which represents the discretization of a continuous domain. Our objective is to obtain a numerical solution of a system of difference equations representing the evolution of some physical phenomenon over the domain. The solution will give us the state of the domain under steady state, i.e. we do not consider the behavior of the system with time: our goal is only to arrive at the *final* state of the system.

The use of the method of differences yields an iterative computation pattern in which the value of each point is computed as a function of the values of its neighbors, and the value of the point itself. But *which* values of the neighbors do we use, the ones computed in the previous iteration, or those computed in the current one? The Gauss-Seidel procedure differs from the Jacobi algorithm (see § 3.5) in its decision with regard to this particular question. In the case of the Jacobi algorithm, we saw that all of the points were updated in iteration I using the values of the its neighbors from iteration $I-1$. In Gauss-Seidel relaxation, the value of a point in iteration I depends on the values from both iteration I and iteration $I-1$. Depending on the order in which the set of points is updated (i.e. left-to-right and top-to-bottom, or right-to-left and top-to-bottom, *etc.*), we get a variety of dependency structures. For example, let us update the points in a left-to-right and top-to-bottom manner. Then, the value of an interior point in iteration I depends on the values of its right and bottom neighbors from iteration $I-1$, and on the values of its left and top neighbors from the *current* iteration, I. This causes the solution to converge quicker than the Jacobi method, since updated values are available in the current iteration along a wavefront that proceeds towards the right and bottom from the top-left corner of the domain. Moreover, since (most) updated values are used within iterations, the Gauss-Seidel method requires far less memory than the Jacobi algorithm.

Parallelization. So how do we go about parallelizing this method? We can see that the

```
1  void updateAllTiled(){
2    int numTiles = matrixDimX/TILE_SIZE;
3
4    for(int I = 0; I < numTiles; I += TILE_SIZE)
5     for(int J = 0; J < matrixDimX; J += TILE_SIZE)
6      for(int II = I; II < I+TILE_SIZE; II++)
7       for(int JJ = J; JJ < J+TILE_SIZE; JJ++)
8        D[II,JJ] = AVG(D[II,JJ],D[II,JJ-1],D[II,JJ-1],D[II,JJ+1],D[II+1,JJ]);
9  }
```

Figure 5.8: A tiled ordering of the computations in the Gauss-Seidel method.

computation of a point's value depends on the results of computations that have occurred immediately to its left, and immediately above it. More concretely, let D be the matrix of points whose values we would like to calculate. The individual point in the i-th row and j-th column of this matrix can be identified as $D[i, j]$. Considering only interior points for the moment, $D[i, j]$ *depends* on the values of $D[i, j-1]$ and $D[i-1, j]$ in the *current* iteration. Therefore, the computation at $D[i, j]$ can occur only after the computations at $D[i, j-1]$ and $D[i-1, j]$. Figure 5.7 shows one way of achieveing the correct, left-to-right and top-to-bottom ordering of computations. Notice that the loop nest in Figure 5.7 *overspecifies* the dependencies in the computation. In particular, we needn't update *all* the points in a row before moving to the next row. We could adopt a *tiled* update scheme without violating the dependencies. This is shown in Figure 5.8.

Note that just as the pseudocode in Figure 5.7, Figure 5.8 too overspecifies the order of computations. Here, we find that even though tile t_1 may not have a data dependency on tile t_2, t_1 might still occur after t_2 due to the sequential nature of the iterations of the `for` construct. For instance, the tile corresponding to $(I, J) = (0, 1)$ is not data-dependent on the tile corresponding to $(I, J) = (2, 0)$, but still occurs after it in the sequence of tile updates. In order to construct a good parallel algorithm for this tile-based relaxation technique, we must first sketch the *true* dependencies between tiles, and therefore find the tiles that can be updated concurrently.

The true data dependencies in the code from Figure 5.8 are illustrated in Figure 5.9. There are four panels in this figure, each representing a step in one iteration of the relaxation procedure. Squares of various colors represent tiles. Green squares represent tiles that have not yet been updated. Blue squares (occurring from the second panel onwards) are tiles that were updated in a previous step. Notice that in each step, each of the three rows of the matrix has at most one tile that is neither green nor blue. These are the tiles currently being updated in that step. For instance, the red tile at $(0, 0)$ is the only one being updated

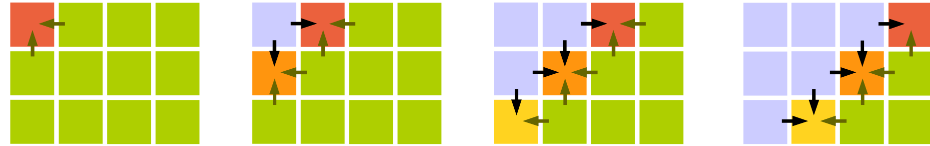

Figure 5.9:

in the first step. Dependencies between tiles are indicated by arrows. Arrows are of two types: those colored dark green and pointing either to the left or up, those colored black and pointing rightward or downward. Arrows of the former type indicate dependencies on tiles that *haven't* been updated in the current iteration (i.e. are green), whereas the black, right- and downward pointing arrows indicate dependencies on blue tiles, which were updated in the previous step of the current iteration.

Let us examine the sequence of updates depicted in Figure 5.9. The first panel shows step zero of some iteration. In it, the tile $(0, 0)$ in the top-left corner of the matrix is updated using the left and top boundaries of its right and bottom neighbors, respectively. It is worth restating that these boundaries hold data that hasn't yet been updated in this iteration. In step one, tiles $(0, 1)$ and $(1, 0)$ can be updated concurrently. In order to update tile $(0, 1)$, we require the left and top boundaries of $(0, 2)$ and $(1, 1)$, respectively, whereas $(1, 0)$ requires the left and top boundaries of $(1, 1)$ and $(2, 0)$, respectively. Once again, given the directions of the corresponding arrows, we can tell that these are *old* boundaries, in that they haven't been updated in the current iteration. Both $(0, 1)$ and $(1, 0)$ also require *new* boundaries from $(0, 0)$: the update of $(0, 1)$ cannot proceed without its right boundary, whereas the update of $(1, 0)$ requires the bottom boundary of $(0, 0)$. Note that these boundaries come from a tile in blue, which was updated in the previous step of the current iteration. We leave it to the reader to reason similarly about the dependencies in steps two and three.

This analysis leads to the following parallelization strategy: use a row decomposition to assign a number of contiguous rows of the input matrix D to each element of a one-dimensional chare array. As discussed previously, the relaxation procedure is iterative, and in each iteration a chare performs as many steps as there are tiles in a row. In the data-driven mold of Charm++, a chare takes the following actions in each step:

1. `sendOldBoundary()`. Send (old) top boundary of current tile to top neighbor, if there exists such a neighbor.

2. `recv()`. Receive as many boundaries as are expected, given the position of the current tile in the matrix. Note that even though a chare may require as many as four boundaries (two old and two new) in order to update a tile, it may receive only two of these from other chares: one from the neighboring chare above, and the other from the

chare below it. Moreover, if the chare receives a bottom boundary from its neighboring chare above, then this boundary must be a new one, updated in the current iteration. Once the requisite number of boundaries has been received, they are used to update the current tile.

3. `sendNewBoundary()`. Having computed the updated values of the points in the current tile, the chare must now send the updated, new bottom boundary of this tile to its neighbor chare below it. Once the neighbor has received this new boundary, as well as the old top boundary from *its* bottom neighbor, it will update its current tile and send its updated boundaries to its neighbors, thereby advancing the wavefront of computation along the sub- and super-antidiagonals of the matrix.

```
1  Main(CkArgMsg *m){
2    Nx = atoi(m->argv[1]); Ny = atoi(m->argv[2]);
3    n = atoi(m->argv[3]); errorTolerance = strtod(m->argv[4], NULL);
4    previousError = std::numeric_limits<double>::max();
5
6    workerProxy = CProxy_GSWorker::ckNew(n);
7    workerProxy.sendOldBoundary();
8  }
```

Figure 5.10:

Let's discuss this parallelization in terms of concrete Charm++ code. Figure 5.10 shows the initialization code for our parallel Gauss-Seidel relaxation program. Chare array *workers* is a one-dimensional array of type *GSWorker* and size n. Each $workers[I]$ holds elements $D[0 : N_x - 1, I(N_y/n) : (I+1)(N_y/n) - 1]$ of the matrix, where N_x and N_y are the dimensions of D along the x and y dimensions, respectively. The dimensions of each tile are $T \times N_y/n$. We initiate the computation by acquiring values for the various parameters, creating the *workers* array, and directing each element in the chare array to send its old top boundary to its top neighbor if it exists (line 7, `sendOldBoundary()`).

Before we look at the code for `sendOldBoundary()`, we consider the structure of messages exchanged between chares. Figure 5.11 shows the fixed size structure `BoundaryMsg`, which is used to communicate tile boundaries between chares. First, note that even though an *array* of values, `data`, is embedded within this message type, its size can still be determined via the `sizeof` operator at compile-time. Therefore, this constitutes a fixed-size message.

The code for `GSWorker::sendOldBoundary()` is given in Figure 5.12. Each chare on which this method is invoked, uses the utility function `rowToMsg` to copy its top row into a

```
struct BoundaryMsg : public CMessage_BoundaryMsg {
  int tag;
  double data[TILE_SIZE];

  BoundaryMsg(int t) : tag(t) {}
};
```

Figure 5.11:

```
void GSWorker::sendOldBoundary(){
  if(thisIndex > 0){
    BoundaryMsg *msg = rowToMsg(TOP_ROW,BOTTOM_BDRY);
    thisProxy[thisIndex-1].recv(msg);
  }
}

BoundaryMsg *GSWorker::rowToMsg(int row, int tag){
  BoundaryMsg *msg = new BoundaryMsg(tag);
  memcpy(msg->data,&myDomain[AT(row,myTile,0)],sizeof(double)*TILE_SIZE);
  return msg;
}
```

Figure 5.12:

`BoundaryMsg` (line 3). This message is sent to the `GSWorker::recv` method of the chare above the sender (line 4). Note that the sent message is tagged as carrying the bottom boundary (`BOTTOM_BDRY`) for the benefit of the *recipient* chare: it allows the recipient to distinguish the two boundaries that it might receive, one from the chare above it, and from the chare below it. A brief discussion of the creation of messages in `rowToMsg` is warranted. On line 9, a `BoundaryMsg` is allocated. Next, the appropriate row, whose starting position is a function (*AT(...)*) of the row and current tile, is copied into the message's *data* buffer.

Chares receive boundaries through the `GSWorker::recv(BoundaryMsg*)` entry method, shown in Figure 5.13. On line 2, we see how the tag of a boundary message is used by the recipient chare to store different boundaries. Next, it checks whether it has received all the boundaries needed for its current tile (line 5; recall that the number of expected boundaries depends on the position of the tile current within the matrix). If so, the current tile is updated, followed by the communication of the updated bottom boundary to the chare

below it in the row decomposition. Thereafter, the chare checks whether it has updated all the chunks assigned to it (line 11), and if so, contributes to a reduction that computes the residual of the relaxation (line 13). Control is transferred to the main chare at this point. If the chare still has some outstanding tiles to update, it moves to the next tile and sends out the old, top boundary of the next tile to the neighbor above it (line 16). Since the received boundaries have now been consumed, the corresponding messages are deleted in line 19.

```
1  void GSWorker::recv(BoundaryMsg *m){
2    myBoundaries[m->tag] = m;
3    nExpected--;
4
5    if(nExpected == 0){
6      compute();
7      resetExpected();
8      sendNewBoundary();
9      myTile++;
10
11     if(myTile == Nx/TILE_SIZE){
12       contribute(sizeof(double),&myError,CkReduction::max_double,CkCallback(CkReductionTa
13       reset();
14     }
15     else sendOldBoundary();
16
17     for(int i = 0; i < NUM_BOUNDARIES; i++){
18       if(myBoundaries[i] != NULL) delete myBoundaries[i];
19       myBoundaries[i] = NULL;
20     }
21   }
22 }
```

Figure 5.13:

It remains for us to examine the actual use of received messages in `GSWorker::compute`. Recall that a chare that receives boundary messages saves pointers to them in the `myBoundaries` array (*cf.* Figure 5.13, line 2.) When the expected number of messages has been received, a chare calls `compute`, wherein a number of functions are invoked (not yet presented) to calculate the updated values of the points in the current tile. We investigate two such methods, namely `updateTopRow` and `updateBottomRow`, in Figure 5.14.

The first of these functions, `updateTopRow`, uses the (new) bottom boundary of its upper

neighbor to update its topmost row. In this function, for each element in the top row of the current tile, i.e. `myDomain[AT(TOP_ROW,myTile,j)]`, we compute an average value based on the four neighbors of the element. Note that the upper neighbor of this element in the topmost row comes from the boundary message (line 6). Similarly, `updateBottomRow` uses the (old) top boundary of its lower neighbor to update elements in the bottom row of its chare's domain. In this function, the bottom neighbor for element `myDomain[AT(BOTTOM_ROW,myTile,j)]` is the element `myBoundaries[BOTTOM_BDRY]->data[j]`, received from its lower neighbor. The `AVG` function also computes the iterative maximum residual (lines 10 and 23). This residual is contributed to the global reduction that ends each iteration (*cf.* Figure 5.13.)

To round out the example, Figure 5.15 shows the target of the global reduction that computes the maximum residual for each iteration of the relaxation. This method receives a system-defined message type of `CkReductionMsg`, from which is extracted the value of the global maximum residual (line 2). If the residual is too large (line 4), we continue with the next iteration by invoking `sendOldBoundary` on each chare (line 5). Otherwise, we deem the relaxation to have converged, and terminate the computation (line 8).

5.2 Variable-Size Messages

(As a simple example, we can even make the gauss seidel using varsize messages, providing motivation for modifying the above program; Say, the array size is meant to be provided on the command line and/or we want to experiment with different $TILE_SIZEs$ for performance).

So far, we have only discussed the use of fixed-size messages. What if objects need to send variable amounts of data to each other, such that the size of sent messages can only be determined dynamically? For instance, in the MD code from § 4.1, how would a chare send its particles to its neighbor in the ring using a message, given that we do not know at compile time the number of particles held by each chare? Charm++ provides *variable-size* messages for situations such as these. We can pack arbitrary amounts of typed data into a message through the following three steps.

Register message type with Charm++. Suppose that we would like to communicate an array of elements of data type `MyFirstStruct`, and another of elements of type `MySecondStruct` through a variable-size message of type `MyVarSizeMsg`. To indicate to the Charm++ translator that the true size of a message of type `MyVarSizeMsg` can only be determined at run-time, we write the `.ci` code shown in Figure 5.16. Note the square bracket notation used to mark `firstArray` and `secondArray` as arrays of variable size. This causes the Charm++ translator to generate dynamic allocation and packing code for members `firstArray` and `secondArray` of `MyVarSizeMsg`. We will discuss how the sizes of individual arrays are given momentarily.

Complete message structure. Only those members of `MyVarSizeMsg` are of interest to the Charm++ translator for which dynamic allocation and packing code must be gen-

erated. Other, fixed-size, member fields are specified in the C++ declaration of the message type, and not in the .ci code. For instance, if MyVarSizeMsg had another data member, MyThirdStruct anotherField, we would specify it only in the C++ declaration of the MyVarSizeMsg class, as shown in Figure 5.17.

Specify sizes of varsize arrays in message. Next, we must specify the size of each individual typed array in the varsize message. This is done when the message type is instantiated using the new operator, For varsize messages, we must use an overloaded version of the new operator, one which allows us to specify the size, in number of elements, of each of the variable-size arrays in the message. For instance, if we wanted to create a MyVarSizeMsg message large enough to hold a firstArray with nElementsFirst elements and a secondArray with nElementsSecond, we would create a new message as shown in Figure 5.18. Note the order of arguments given to the new operator: this order corresponds to the order in which the varsize arrays appear in the .ci file.

Note that in the above, we assumed that data types MyFirstStruct, MySecondStruct and MyThirdStruct were of *fixed size*, even though arrays firstArray and secondArray could be of arbitrary size. In particular, we assumed that data elements of type MySecondStruct and MyThirdStruct didn't maintain pointers to other data structures that were meant to be shipped along with them. We will discuss the packing of such complex data types in § 5.3.

5.2.1 Example: Simple Molecular Dynamics

Let's round out this section with an example that gives a concrete illustration of the declaration, definition, allocation and deallocation of variable-size messages in a Charm++ program. For this purpose, we turn back to the simple molecular dynamics simulation from § 4.1. This time, we will use explicit messaging to communicate particles between chares instead of using parameter marshalling.

Recall that we are given a (large) set of identical atoms enclosed within a simulation volume. The atoms are not bonded together, and are assumed to have no electrostatic interactions: we intend only to calculate the van der Waals forces between them, and thus compute the trajectories of individual atoms. We won't discuss the actual decomposition of particles onto chares, but simply note that each chare is assigned a subset of particles that is disjoint from every other chare's subset. Furthermore, the union of these subsets is the input set of particles.

First, we define a ParticleMsg that will hold a subset of particles to be communicated between chares. The structure of this message is shown in Figure 5.19. It contains two (variable-length) arrays: positions, which holds the positions of some subset of particles, and forces, which holds the accumulated forces on the particles in that subset. Both arrays are of type Vector3D<rtype>, where rtype denotes the type of each component of the vectors (likely float or double). The message also has an integer field, nParticles, which holds

the number of particles in the subset. The `.ci` code that designates `positions` and `forces` as variable-length arrays is shown in Figure 5.20.

Next, we look at how to instantiate a `ParticleMsg` with a requisite amount of space for particle position and force coordinates. A chare would instantiate a `ParticleMsg` when sending its particles to its neighbor in the communication ring. This happens in method `Worker::start`, which is basically unchanged from Figure 4.8 of § 4.1, except for line 76, where the particles owned by a chare are sent to its neighbor in the communication ring. In its place, we have the code shown in Figure 5.21. Line 3 of Figure 5.21 shows the allocation of a `ParticleMsg` with `numParticles` position vectors and `numParticles` force vectors. Thereafter, the positions of the particles owned by the chare are copied into the message, and the force values therein are initialized. Note that `struct ParticleMsg` has data members `source` and `iteration` to replace the marshalled parameters that are received by `Worker::compute` as input. Finally, the chare sends this message to the `compute` method to its neighbor in the communication ring (line 8). Since `Worker::compute` now expects to receive a `ParticleMsg`, its signature must change to: `void Worker::compute(ParticleMsg *msg);`

Of course, particles are communicated in other contexts as well: when a chare receives a set of particles from one of its ring neighbors, it must "consume" them to calculate forces on its local particles due to these received particles, and having done so, must pass them on to its other neighbor (line 102 of Figure 4.8, in the body of `Worker::compute`). Moreover, once a subset of particles has been "consumed" in this manner by half the chares in the ring, it must be sent back to its owner, which will have accumulated forces on its local particles due to the *other* half of the ring. This happens on line 97 of Figure 4.8, in `Worker::compute`. We replace line 102 (see above) with the code on line 4 of Figure 5.22 and line 97 with line 9 of the same figure. Notice how we are able to *reuse* the message passed into the method `Worker::compute`, unlike in the parameter-marshalled version of the code in Figure 4.8. In the latter, the marshalling of the user-provided `array` incurs copying overhead, making the message-based version more efficient. Finally, note that in the definitions of `Worker::compute` and `Worker::forces`, where we would have used marshalled parameters in *rexpressions* in the parameter-marshalled version of the code, we use the corresponding data members of `ParticleMsg` in the varsize-message based version.

5.3 Packing Complicated Datatypes into Messages

We now address the caveat that we had made towards the end of § 5.2. That is, how do we communicate a complicated, non-linear data structure such as a tree or a graph through a message? In the following, we will learn about *custom packed* messages. Custom packed messages can embed non-linear data structures, such that embedded data structures are serialized only when the message crosses an address space boundary. This feature of packed messages makes them especially efficient on multicore nodes and clusters of shared memory

nodes.

Briefly, the custom-packing approach consists of three steps: (1) Define a packed message type; (2) overload the `pack` and `unpack` methods of the defined message type; and (3) allocate and use packed messages just as you would fixed-size messages. Now, let's consider these steps in detail, in the context of an example.

Suppose that we would like to do a Barnes-Hut style parallel simulation of N particles interacting through gravitational forces. This simulation involves a binary tree data structure that is distributed across chares, so that each chare holds only a portion of the distributed tree, that is a *subtree*. Since gravity is a long range force, a chare requests parts of other chares' subtrees. It is worth noting that a request will likely not be for the *entire* subtree held by a chare, but for a particular node by the chare, plus the subtree underneath it, up to some cutoff depth. When a chare receives a request for some part of its subtree, it must serialize that subtree into a message, and send it to the requesting chare as its response. The requesting chare can then use the received subtree to do computations, possibly requesting other subtrees in the process. Here, we will focus not on the particulars of the algorithm, but on writing code to communicate subtrees via custom-packed messages.

Let `MyNode` be the type of each node in the binary tree. We declare custom-packed message types to the Charm++ translator just as we did fixed-size messages. In the present example, we would write `message SubtreeMsg;` in the appropriate `.ci` file. Next, we define the message type, as shown in Figure 5.23. The data structure `MyNode` is also presented. In addition to data fields `key`, `centerOfMass` and `mass`, the structure has fields `isLeaf` and `children` which describe the local structure of the tree. For simplicity, we assume that if a node is not a leaf (i.e. its `isLeaf` field is `false`), then both `children[0]` and `children[1]` point to valid child nodes.

Notice the `root` data member, of type `MyNode`, that each `SubtreeMsg` contains. This represents the root of the subtree that is held by the message. Field `numNodes` holds the number of nodes in this subtree, which is at most `maxDepth` deep, where `maxDepth` is a constant, readonly variable. If we were to construct a `SubtreeMsg` with a valid tree node (call it n) for its `root`, the `root` would be a *shallow* copy of n. As a result, it would contain `children` pointers that would be invalid if the `SubtreeMsg` were sent as-is to a chare in a different address space. Therefore, we must tell the Charm++ system how to perform a *deep* copy of the `maxDepth`-limited subtree under `root` that interests us. That is, we must provide customized packing code for the data structure that the `root` field of the `SubtreeMsg` represents. We do this by overriding the static method `SubtreeMsg::pack`. Notice the signature of this method: it takes as input a `SubtreeMsg` and returns a generic `void*` buffer that represents the linearized contents of the non-linear data structure present in `root`. Similarly, at the receiving end, we must translate (unpack) a generic `void*` buffer into a non-linear subtree that correctly recreates the binary tree structure at the sender's side. The code for this is provided in the second method that we must overload, namely `SubtreeMsg::unpack`. This static method takes as input a `void*` buffer, and deserializes it

to obtain a `SubtreeMsg` whose `root` data member points to the `maxDepth`-limited subtree that was requested by the receiving chare.

The code for `SubtreeMsg::pack` is shown in Figure 5.24. As mentioned before, the objective of this function is to perform a deep copy of the `maxDepth`-limited subtree underneath `root`. To do this, we first obtain the total size, in bytes, of the tree underneath `root` up to a maximum depth of `maxDepth` below it. This is done by the `MyNode::size` method (line 3), which recursively processes nodes until it either encounters a leaf, or the depth of an internal node relative to `root` exceeds `maxDepth`. We leave the definition of this method to the reader as an exercise. Having obtained the size of the subtree, we ask the Charm++ system for a buffer large enough to hold all the nodes in the subtree (line 5). Note that we use the function `CkAllocBuffer` for this purpose, instead of the `new` operator: this is because in addition to the buffer of size `totalSize`, an outgoing message must also encapsulate an *envelope* that is used by the Charm++ runtime. However, `CkAllocBuffer` returns a pointer to the "user" portion of the message, into which we serialize the `maxDepth`-limited subtree under `root` (line 7; we leave the definition of `MyNode::serialize` to the reader). Finally, having performed a deep copy via `serialize`, we discard the input message, `in`, and return the serialized buffer to be transmitted to the target chare, which would be located in a different address space.

Next, we must specify what actions are to be taken by the runtime at the receiving end of a custom-packed message. We do this by overriding the static `SubtreeMsg::unpack` method (Figure 5.25). As you would expect, this method takes a linear (serialized) buffer as input, and unpacks its contents into a non-linear data structure that is usable in user code. We begin by allocating a `SubtreeMsg`. At this point, the `root` member of the allocated `msg` represents an invalid tree, since we have not unpacked the contents of the serial buffer `in` into it yet. This unpacking is done by the `MyNode::deserialize` method (once again, we leave its definition to the reader as an exercise). Note that the `deserialize` method should allocate nodes as required and copy the contents of the serialized buffer into these nodes. After freeing the memory occupied by the serialized buffer that this method received as input (`CkFreeMsg`, line 9), we return the `SubtreeMsg` that will finally be passed back into user code.

Now all that remains is the actual instantiation of the custom-packed `SubtreeMsg`. Well, we instantiate custom-packed messages just as we did *fixed-size* messages! So, a simple `SubtreeMsg *msg = new SubtreeMsg(subtreeRoot);` would do. This might surprise the reader at first, but consider that the `pack/unpack` methods are only called when a message is to cross address space boundaries. The construction of a `SubtreeMsg` as above creates a message with a *shallow* copy of the node `subtreeRoot`. If the message is destined for a chare in the same address space, all the pointers in this shallow copy would be valid at the destination too, so the `pack/unpack` methods are not called at all. On the other hand, if the Charm++ determines that the recipient of the message resides in a different address space, it performs a deep copy of the tree underneath `subtreeRoot`, serializing it in the process (with the code that we supplied), and sends it to the appropriate address space. There, the serialized information is unpacked into a usable format (deserialized), and passed back into

user code. A corollary of this observation is that for safety, the contents of a custom-packed message should be accessed by chares in a *read-only* mode. Otherwise, chares in the same address space may corrupt each other's memory.

5.3.1 Serializing Data into Messages with PUP

There exists an alternative approach to communicating non-linear data structures via messages, although we do not discuss it in detail here. In § 11, we learnt about the PUP framework, and how it is used to inform the runtime system of the dynamic structure of complicated data structures. Basically, the PUP framework lets the programmer relate the blueprint of an object to the runtime system, so that the object may be serialized and deserialized, whether it be for communication over the network, checkpointing to the hard disk or fault tolerance.

We can also use PUP to *explicitly* serialize non-linear data structures of variable size into messages, so that they can be communicated between chares. We would do this by creating a variable-size message that embeds a generic, `char*` array. Then, we would use a `PUP::sizer` to find the size of the data structure to communicate. This size would be used to create a variable-size message that is large enough to hold the linearized data structure. Next, we would serialize the data structure into the variable-size `char*` array of the allocated message, using an object of type `PUP::toMem`. The message would then be ready for transmission to the target chare. At the receiving end, we would unpack the contents of the message through an object of type `PUP::fromMem`.

5.4 Conclusion

Let's close this chapter by restating the situations in which it might be appropriate to choose messages over parameter marshalling, as well as the various types of messages, their declaration and instantiation.

When should you choose messages over marshalled parameters? The performance-conscious programmer will choose messages over marshalled parameters in two situations: (i) Recall that marshalled parameters are available to your code only in the scope of the entry method in which they are received. Therefore, if a chare needs to access marshalled parameters after the entry method has finished, they must be copied into heap-allocated buffers. This often happens in programs in which a *chare receiving data may not be ready to process the data immediately upon receipt*. In this case, data received via a message can be saved for later use by simply recording the pointer to the input message received by the entry method. This is often more efficient than copying received marshalled parameters onto the heap, especially if the entry method receives arrays and large, composite data structures. Of course, you must remember to `delete` received messages yourself: Charm++ will not do

this for you, except in special cases. (ii) You should also use a message instead of marshalled parameters if your entry method exhibits the following pattern: it receives data of a certain size, possibly altering it in the body of the entry method, and sends it to another chare, such that the size of sent data is the same as that of the received data. In such a case, the contents of an incoming message could be altered, and the message *reused* in the invocation of an entry method on some other chare. For instance, the ring communication example from § 3.2 would benefit from using messages: instead of accepting a number of marshalled `int` type parameters, the `Ring::doSomething` method could accept a message, alter its contents to reflect the number of chare array elements yet to receive the message, and send *the same* message to the next chare in the ring. Note that when user code passes a message to the Charm++ runtime system (for instance, through an entry method invocation) the memory associated with the is no longer available to the user. In particular, attempts to dereference a pointer to a message object that has already been used in an entry method invocation, will yield incorrect results and possibly cause your program to crash. Therefore, a received message can only be reused *once* at the receiving end, i.e. it can only be passed as input to a *single* entry method invocation by the chare that received it, although that invocation may be a point-to-point communication, or a broadcast/multicast. However, a received message *cannot* be used in multiple entry method invocations at the receiving end. For instance, if you were writing a Charm++ program in which the contents of a message were being passed down the nodes of a spanning tree constructed from an array of chares, the code that forwards a message received by an internal node could send the received message to only *one* of the chare's spanning tree children; for all other children, new messages would have to be allocated.

What are the various types of message in Charm++? There are three types of message, namely *fixed-size*, *variable-size* and *packed*. Fixed-size messages (§ 5.1) have constant size, and usually encapsulate several data members, each of which is of fixed size. Therefore, the size of the message can be obtained by applying the C++ `sizeof` operator to the message type. Note that fixed-size messages can include arrays as members too, but these arrays cannot be allocated dynamically. To allow chares to communicate variable amounts of data, there are *variable-size* messages (§ 5.2), which may encapsulate dynamically allocated memory of arbitrary size and type. Finally, packed messages are used to communicate (possibly non-linear) data structures between chares in an efficient manner. Packed messages (§ 5.3) incur serialization overhead only when the encapsulated data is to be communicated across address spaces.

How are messages of various types declared? Before defining a message type, you must inform the Charm++ translator that a certain user-defined data type is to be considered as a message that can be communicated between chares. For a fixed-size message of type `MyMessageType`, this is done by including a statement of the form `message MyMessageType;` in the appropriate `.ci` file of the program. See example `.ci` code in Figure 5.1 for a simple

ping-pong program. In declaring a variable-size message of type MyMessageType we must tell the Charm++ translator about all the variable-length arrays that are embedded within the message. Code showing the form of such declarations is given in Figures 5.16 and 5.20. For custom-packed messages, we require the same .ci code as we did for fixed-size messages (see § 5.3).

How are message types defined? Message types must inherit from their corresponding (translator-generated) CBase_ classes. For example, a message of type MyMsgType must inherit from CBase_MyMsgType. The variable-length arrays encapsulated by a variable-size message are represented by pointers of the corresponding types. For instance, see Figures 5.17 and 5.19. For a custom-packed message, which is meant to embed user-defined, non-linear data structures, we place shallow copies of the data structures in the message. In addition, we must override the static pack and unpack methods of the message type, as shown in § 5.3 (Figures 5.24 and 5.25.)

How are messages instantiated? This too depends on the type of the message: fixed-size and custom-packed messages are instantiated in similar ways, through the regular new operator of the corresponding message type (See Figure 5.12 and § 5.3 for code showing the allocation of fixed-size and custom-packed messages, respectively.) To instantiate variable-size messages, we must use overloaded versions of the new operator. The number of arguments to the new operator of a variable-size message depends on the number of variable-length typed arrays that it encapsulates. Each argument gives the size of the corresponding array, as shown in Figures 5.18 and 5.21.

```
void GSWorker::updateTopRow(){
  double err;
  for(int j = 0; j < TILE_SIZE; j++){
    err = AVG(myDomain[AT(TOP_ROW,myTile,j)],
          myDomain[LEFT(TOP_ROW,myTile,j)],
          myBoundaries[TOP_BDRY]->data[j],
          myDomain[RIGHT(TOP_ROW,myTile,j)],
          myDomain[BELOW(TOP_ROW,myTile,j)]
          );
    if(myError < err) myError = err;
  }
}

void GSWorker::updateBottomRow(){
  double err;
  for(int j = 0; j < TILE_SIZE; j++){
    err = AVG(myDomain[AT(BOTTOM_ROW,myTile,j)],
          myDomain[LEFT(BOTTOM_ROW,myTile,j)],
          myDomain[ABOVE(BOTTOM_ROW,myTile,j)],
          myDomain[RIGHT(BOTTOM_ROW,myTile,j)],
          myBoundaries[BOTTOM_BDRY]->data[j]
    );
    if(myError < err) myError = err;
  }
}
```

Figure 5.14:

```
void Main::recvError(double err){
  double globalError = err;
  delete m;
  double delta = fabs(previousError-globalError);
  if(delta/previousError > errorTolerance){
    workerProxy.sendOldBoundary();
    previousError = globalError;
  }
  else CkExit();
}
```

Figure 5.15:

```
message MyVarSizeMsg {
  MyFirstStruct firstArray[];
  MySecondStruct secondArray[];
};
```

Figure 5.16:

```
class MyVarSizeMsg : public CMessage_MyVarSizeMsg {
  private:
  MyFirstStruct *firstArray;
  MySecondStruct *secondArray;
  MyThirdStruct anotherField;

  // other member fields and methods ...
};
```

Figure 5.17:

```
MyVarSizeMsg *msg = new (nElementsFirst,nElementsSecond) MyVarSizeMsg;
```

Figure 5.18:

```
struct ParticleMsg : public CMessage_ParticleMsg {
  Vector3D<rtype> *positions;
  Vector3D<rtype> *forces;
  int nParticles;

  int source;
  int iteration;

  ParticleMsg(int src, int iter, int np) :
    source(src), iteration(iter), nParticles(np) {}
};
```

Figure 5.19:

```
message ParticleMsg {
  Vector3D<rtype> positions[];
  Vector3D<rtype> forces[];
};
```

Figure 5.20:

```
// in Worker::start(), send own particles to neighbor
ParticleMsg *msg;
msg = new (numParticles,numParticles) ParticleMsg(thisIndex,iteration);
for(int i = 0; i < numParticles; i++){
  msg->positions[i] = locals[i].position;
  msg->forces[i] = Vector3D<rtype>(0.0);
}
thisProxy[next].compute(msg);

```

Figure 5.21:

```
// in Worker::compute(),
// when done calculating forces on local particles due
// to received particles, forward them to ring neighbor...
thisProxy[next].compute(msg);

// ...or, if accumulated forces are to be returned to the
// source chare, i.e. the owner of the particles, send
// them back:
thisProxy[msg->source].forces(msg);
```

Figure 5.22:

```
struct MyNode {
  int key;
  Vector3D<float> centerOfMass;
  float mass;
  bool isLeaf;
  MyNode *children[2];

  size_t size(int depthCutoff);
  void serialize(MyNode *into, int depthCutoff);
  void deserialize(MyNode *from);

};

struct SubtreeMsg : public CMessage_SubtreeMsg {
  MyNode root;

  SubtreeMsg() {}
  SubtreeMsg(MyNode *root_) : root(*root_) {}

  static void *pack(SubtreeMsg *in);
  static SubtreeMsg *unpack(void *in);
  static size_t getStaticSize();
};
```

Figure 5.23:

```
void *SubtreeMsg::pack(SubTreeMsg *in){
  // get total serialized size of subtree under ‘root’
  size_t totalSize = in->root.size(maxDepth);
  // allocate a buffer large enough to hold serialized tree
  char *buf = (char *)CkAllocBuffer(in,totalSize);
  // serialize tree into allocated buffer
  in->root.serialize((MyNode*)buf,maxDepth);
  // we’ve done a deep copy of the data pointed to
  // by the input msg ‘in’; don’t need ‘in’ anymore
  delete in;
  return (void*) buf;
}
```

Figure 5.24:

```
SubtreeMsg *SubtreeMsg::unpack(void *in_){
  char *in = (char *)in_;
  // allocate a msg of type SubtreeMsg
  SubtreeMsg *msg = (SubtreeMsg *)CkAllocBuffer(in,sizeof(SubtreeMsg));
  // construct non-linear data structure from serialized buffer ‘in’
  msg->root.deserialize((MyNode*)in);
  // we’ve recreated the tree in the serial buffer
  // ‘in_’; don’t need ‘in_’ anymore
  CkFreeMsg(in_);
  return msg;
}
```

Figure 5.25: