Chapter 9

Expressing Program Flow: Threaded Entry Methods

Imagine a situation in which you are writing the code for an entry method of a chare, and realize that you need to obtain a value from another chare. What you can do, using techniques we learned in earlier chapters, is to send a request to that chare, and have it send a response to one of your entry methods, where you continue your execution. This technique, sometmes called "continuation passing" style, works, but makes programs look awkward, especially in the simple scenarios, such as the request-response scenario described above.

In this chapter, we will first learn threaded entry methods, and their counterpart the blocking or "sync" entry methods. Then we see examples of how blocking invocations of sync methods are carried out from threaded entry methods, and how to utilize this mechanism for expressing control flow within an object effectively. We will then see a few advanced ways of using threaded entry methods.

9.1 Threaded and Sync Methods

One can declare an entry method to be "threaded" simply by adding the attribute "threaded" to the entry method declaration in the interface file:

```
entry [threaded] void f(...);
```

This tells the charm runtime system to create a thread every time this method is invoked, and run it within its own thread. An object may have multiple threaded entry methods, and at any particular time there may be many instances of threaded entry methods in operation; in practice, the common usage is to have a single thread associated with each object.

A method can be declared to be a `sync` method using the attribute "sync" in its declaration in the interface file.

```
entry [sync] MsgType * g(..);
```

A sync method must have a return type which is a pointer to a message type (see Section 5). This is in contrast to normal entry methods that are required to have a void as a return type.

9.2 A simple illustrative example of blocking invocation

Let us consider the following problem: we have a chare-array A of N elements, and each element holds a single value. We want to check if the values are already in a sorted order: i.e. value held by i^{th} element of the chare array is less than or equal to value held by j^{th} element iff $i < j$. One way is to have each chare with an index i (except the last one), compare it's own value with that held by the chare next to it (i.e. with an index $i + 1$). We could make a chain of these checks, so A[0] sends its value to A[1], which checks it and if it is in order sends its own value to A[2] and so on. If the last chare (index $N - 1$) finds everything in order, it sends a reply to the main chare reporting success. If any chare finds that its value is out of order with the previous chare's value, it reports a failure to the main chare, and does not send a message to its next chare. (Exercise: write this program).

But this is not a very efficient way of checking. It takes a *sequence* of N messages to get the result back to main chare, and only one chare is doing the check at a time, i.e., there is no parallelism in this way of doing things.

```
mainmodule checkOrder {
  message MsgData;
  readonly int numElements;

  mainchare Main {
    entry Main(CkArgMsg *msg);
    entry [reductiontarget] void isSorted(int result);
  };

  array [1D] SimpleArray {
    entry SimpleArray();
    entry [threaded] void run();
    entry [sync] MsgData * getValue();
  };
}
```

Figure 9.1: A Blocking invocation from a threaded entry method : the interface file `blocking.ci`

Instead, we can have each chare check with its right neighbor, in parallel, and have them combine there results via a reduction (which takes only log N time). We will do this using a threaded entry method, just to illustrate how to use them. The program is shown in Figures 9.1, 9.2 and 9.3. A threaded entry method is declared by adding the keyword `threaded` in its declaration in the `.ci` file, as shown in line 12 of Figure 9.1. A threaded method is allowed to suspend or make blocking calls. The simplest blocking call we will learn is the invocation of a blocking (or "sync") method. Here, method `getValue` is declared as a sync method, in the `.ci` file. `sync` methods return a pointer to a message, unlike the regular entry methods, which have a void return value.

```
#include "checkOrder.decl.h"
#include <stdlib.h>

/* readonly */ int numElements;

class MsgData: public CMessage_MsgData
{ public:   double value; };

class Main : public CBase_Main {
 public:
  Main(CkMigrateMessage *msg) { }
  Main(CkArgMsg* msg) {
    numElements = 10;
    if (msg->argc > 1) numElements = atoi(msg->argv[1]);
    delete msg;

    // Create the chare array with numElements and set callBack
    CProxy_SimpleArray A = CProxy_SimpleArray::ckNew(numElements);
    CkCallback *cb = new CkCallback(CkReductionTarget(Main, isSorted), thisProxy);
    A.ckSetReductionClient(cb);
    A.run(); // start the run thread of each object.
  }

  void isSorted(int result) {
    if(result == 1) {
      CkPrintf("The array was sorted \n");
    } else {
      CkPrintf("The array was not sorted \n");
    }
    CkExit();
   }
};

```

Figure 9.2: A Blocking invocation from a threaded entry method: class Main in the C++ file `blocking.C`

Class Main is shown in Figure 9.2. This class performs simple tasks - creates chare-array A, sets reduction callback to one of its method and invokes run on A. When invoked, the constructor of chare-array A initializes the value every chare contains using a pseudo-random value (Figure 9.3). Thereafter, when run method is invoked, it calls getValue, a sync method, on the chare to its right in A, and obtains the value stored in it. Having obtained the value, it compares that value with its own value, and stores the result. Finally, it contributes the result to the reduction, whose result is delivered to the main chare.

```

class SimpleArray : public CBase_SimpleArray {
 private:
  double myValue;

 public:
  SimpleArray(CkMigrateMessage *msg) { }
  SimpleArray() { myValue = drand48();  }

  void run() {
    int result = 1;
    printf("[%d](%d): myValue = %f\n", thisIndex, CkMyPe(), myValue);
    if(thisIndex != numElements-1) {
      MsgData *m = thisProxy(thisIndex+1).getValue();
      if(myValue > m->value) result = 0;
    }
    contribute(sizeof(int), &result, CkReduction::logical_and);
  }

  MsgData* getValue() {
    MsgData * m = new MsgData();
    m->value = myValue;
    return m;
  }
};

#include "checkOrder.def.h"
```

Figure 9.3: A Blocking Invocation from a threaded entry method: class SimpleArray in the C++ file blocking.C

9.3 Odd-Even SillySort

Typically, threads are suspended waiting for something to happen, e.g., arrival of data via another entry method. Such an entry method can then awaken the thread if it is waiting.

To demonstrate the way threads work, we will consider a simple idea for sorting numbers in a situation where each chare in an array of chares holds exactly one number. This somewhat silly situation (silly because if there is only one number held by each chare, it is too small a problem to require parallel processing) will illustrate several attributes of the threaded entry methods in the context of a simple example.

Our method is motivated by a human analogy: imagine N people standing in a row, each one holding a number. They can only interact with their immediate neighbors. The interaction here simply consists of looking the number held by the neighbor, and possibly exchanging your number with them. Our objective is to make sure that each person's number is smaller than that held by the person on their right (so, in the end to get the smallest number to the leftmost person in the row).

Going with our human analogy, we can just have each person peek at the number of the person to their right. If it is smaller than the one held by them, they propose an exchange - send their number to the right. With people, this might work fine. But there is a problem: when do you know that you (and the whole group) are done? Just because you have not seen a smaller number to your right for a while does not mean it would not show up in a few more moments, as it make its way from the rightmost person. We can say that at most N steps should be enough to get the smallest number moved all the way to the left. But what is a step? You will have to synchronize all the people to a central clock for that. Finally, with people, its easy to notice if the person to your right is engaged in an exchange with the one to *its* right, and wait until that finishes. With chares and computers, you are not going to notice an empty container for the number.

```
mainmodule BuggyArraySort {
  message ValueMsg;
  readonly int numElements;
  readonly CProxy_Main mainProxy;

  mainchare Main {
    entry Main(CkArgMsg* msg);
    entry [reductiontarget] void finished();
  };

  array [1D] OddEvenSorter {
    entry OddEvenSorter();
    entry [threaded] void run();
    entry [sync] ValueMsg * sendSmallerBack(double x);
  };
}
```

Figure 9.4: A sorting program with a race condition: the interface file `buggyArraySort.ci`

The code shown in Figures 9.4, 9.5 and 9.6 attempts to deal with these issues. As in the previous example, we have a threaded entry method called `run`, and a sync method `sendSmallerBack` as shown in the interface file in Figure 9.4. Class Main for this example program (and for many others to follow) is shown in Figure 9.5. This class is responsible for creating the chare-array of the class OddEvenSorter, and invoking the threaded `run` method on it.

```
// THIS PROGRAM DOES NOT WORK BECAUSE OF A RACE CONDITION
// It is an intentional bug, for the sake of a tutorial example

#include "buggyArraySort.decl.h"
#include <stdlib.h>
#include <time.h>

#define INFINITE (9999.9999)
/* readonly */ int numElements;
/* readonly */ CProxy_Main mainProxy;

class ValueMsg: public CMessage_ValueMsg
{ public:   double value; };

class Main : public CBase_Main {
 public:
  Main(CkMigrateMessage *msg) { }
  Main(CkArgMsg* msg) {
    mainProxy = thisProxy;
    numElements = 10;
    if (msg->argc > 1) numElements = atoi(msg->argv[1]);
    delete msg;

    CkPrintf("  numElements = %d, #PEs() = %d\n",numElements, CkNumPes());
    // Create the chare array with numElements and set callBack
    CProxy_OddEvenSorter sorter = CProxy_OddEvenSorter::ckNew(numElements);
    sorter.run(); // start the run thread of each object.
  }

  void finished() { CkExit(); }
};
```

Figure 9.5: A sorting program with a race condition: class Main in the C++ file `buggyArraySort.C`

Class OddEvenSorter, which performs the sorting, is shown in Figure 9.6. Since a chare cannot actually peek at the data held by another chare, we get a chare to send its number to the chare to its right, requesting it to send the smaller of the two (one held by it and the one

```
33 class OddEvenSorter : public CBase_OddEvenSorter {
34  private:
35   double myValue;
36
37  public:
38   OddEvenSorter(CkMigrateMessage *msg) { }
39   OddEvenSorter() { myValue = drand48(); }
40
41   void run() {
42     CkPrintf("[%d on %d]:  before sorting. myValue = %f\n", thisIndex, CkMyPe(), myValue);
43     if (thisIndex != numElements-1) { // the last element is passive.
44       for (int i=0; i<numElements; i++) {
45         double tmp = myValue;
46         myValue = INFINITE;
47         ValueMsg *m = thisProxy(thisIndex+1).sendSmallerBack(tmp);
48         myValue = m->value; delete m;
49       }
50     }
51     printf("[%d]:  after sorting. myValue = %f\n", thisIndex, myValue);
52     contribute(0,NULL,CkReduction::nop,CkCallback(CkReductionTarget(Main,finished),mainProxy));
53   }
54
55   ValueMsg* sendSmallerBack(double value) {
56     ValueMsg * m = new ValueMsg();
57     if (value < myValue)
58       m->value = value;
59     else { // exchange
60       m->value = myValue;
61       myValue = value;
62     }
63     return m;
64   }
65 };
66 #include "buggyArraySort.def.h"
```

Figure 9.6: A sorting program with a race condition: class OddEvenSorter in the C++ file `buggyArraySort.C`

coming from its left) back (`sendSmallerBack`). In `run`, we decide to execute this N times in accordance with the observations above. To make sure that values are not duplicated or lost, we use the a trick: just as you send your data to your right neighbor (by calling sendSmallerBack in line 51), you set your own value to the largest possible value (INFINITY). This ensures that even if your left neighbor were to invoke *your* `sendSmallerBack` method, while you are in a limbo (i.e. your value is gone to your right neighbor, and they are going to send it or their number back to you), you will end up returning the left neighbor's value, because it is smaller than INFINITY.

Did you spot the error in this code? If not, think about it for a few minutes.

The problem is a race condition: your left neighbor may send you lots of messages (may be even N) repeatedly, while you happen to be waiting for your right neighbor to respond (i.e. return a value for sendSmallerBack()). After all, the call goes via a message in the network, and may sit in the receiving processor's scheduler's queue for a while. Thus, it is possible that the smallest value doesn't reach the left end. Essentially, it is the notion of *step* that is the problem: some processors think they have done with N steps, but some of those were "wasted" steps because their right neighbor was not ready, and sent the same value back.

We could fix it by sending a special additional flag back in the return message. But instead, we will solve the problems in two different ways that demonstrate ways in which threaded methods are used.

9.3.1 Using a Simple Barrier in Odd-Even Sort

One way to introduce the notion of step among chares is to introduce a user-defined barrier after every exchange. Figure 9.7 presents the interface file for the example code that implements this scheme. The only addition to the interface file is the new reduction target `barrierEP` to class OddEvenSorter.

```
mainmodule arraySortUsingBarrier {
  message ValueMsg;
  readonly int numElements;
  readonly CProxy_Main mainProxy;

  mainchare Main {
    entry Main(CkArgMsg* msg);
    entry [reductiontarget] void finished();
  };

  array [1D] OddEvenSorter {
    entry OddEvenSorter();
    entry [reductiontarget] void barrierEP();
    entry [threaded] void run();
    entry [sync] ValueMsg * sendSmallerBack(double x);
  };
}
```

Figure 9.7: A sorting program using user-implemented barrier: the interface file `arraySortUsingBarrier.ci`

```
class OddEvenSorter : public CBase_OddEvenSorter {
 private:
  double myValue;
  CthThread t;
 public:
  OddEvenSorter(CkMigrateMessage *msg) { }
  OddEvenSorter() { myValue = drand48(); }

  void myBarrier() {
    contribute(0,NULL,CkReduction::nop,CkCallback(CkReductionTarget(OddEvenSorter,barrierEP),thisProxy));
    t = CthSelf();
    CthSuspend(); // contined after barrierEP executes
  }

  void barrierEP() { CthAwaken(t);}

  void run() {
    ValueMsg * m;
    for (int i=0; i<numElements; i++) {
      if ((thisIndex%2 == 0) &&(thisIndex != (numElements-1))) {
        m = thisProxy(thisIndex+1).sendSmallerBack(myValue);
        myValue = m->value; delete m; }
      myBarrier();
      if ((thisIndex%2 == 1) &&(thisIndex != (numElements-1))) {
        m = thisProxy(thisIndex+1).sendSmallerBack(myValue);
        myValue = m->value; delete m;}
      myBarrier();
    }
    contribute(0,NULL,CkReduction::nop,CkCallback(CkReductionTarget(Main,finished),mainProxy));
  }

  ValueMsg* sendSmallerBack(double value) {
    ValueMsg * m = new ValueMsg();
    if (value < myValue)
      m->value = value;
    else { // exchange
      m->value = myValue;
      myValue = value;
    }
    return m;
  }
};

#include "arraySortUsingBarrier.def.h"
```

Figure 9.8: A sorting program using user-implemented barrier: class OddEvenSorter in the C++ file `arraySortUsingBarrier.C`

The implementation of class OddEvenSorter using user-defined barrier is shown in Figure 9.8. The important difference to note, in comparison to the previous implementation, is the use of barrier in `run`. In each iteration, we have divided the chares into two sets (determined by parity of the index of the chares). In the first phase, chares with even parity exchange value with their right neighbors, followed by a barrier. This barrier helps avoid any race condition with the second phase of exchange in which chares with odd parity performs the swap. The barrier has been implemented using a simple reduction among the chares, along with constructs involving threads you have not seen before. I suggest, for now, not to look inside the implementation of the barrier, but just take my word for what it does: it suspends the calling thread until all the chares in the chare array have called the barrier, and then resumes all the suspended threads. You can return to understanding the implementation after reading the next section.

A barrier is not a very satisfactory way of solving this problem. For one thing, it is relatively expensive to synchronize all chares via a barrier. But it illustrated the mechanisms we wanted to illustrate. We leave it to the reader to find a solution that avoids using a barrier.

9.4 Advanced uses of Threaded Entry Methods

We will now illustrate some additional programming techniques and constructs that use threaded entry methods. These overcome a particular limitation of the blocking invocation: a threaded entry method can (naturally) only have one outstanding blocking method. But there are situations in which we may want to wait for multiple events, and in some cases, it may not be a "return" of your call. As simple example, we will use recursive definition of n^{th} Fibonacci number:

```
Fib(N) = if N<2 then N else Fib(N-1) + Fib(n-2)
```

(Again, not the best way of computing n^{th} Fibonacci number, but this formulation is a conventional stand-in for all divide-and-conquer algorithms.)

We already saw how this was computed using ordinary entry methods in an earlier chapter (**REF**). Here, let us see if expressing it using threaded methods can simplify its expression.

The problem we have in using threaded entry method is that we need to fire two instances of the fib chare. If we were to use sync method for that, we would be stuck waiting for the result of the first cal, before we can make the second call. That would serialize the computation. Instead, in this example, we will use "bare" threads:

The lowest level primitives you are allowed to call on a thread are CthSuspend() to suspend the currently running thread, and return control to the scheduler and CthAwaken(threadID) to put the thread with a given threadID (presumably suspended at the time of the call) in the scheduler's queue of ready threads. In addition, a function CthSelf() returns the threadID of the running thread.

So, our idea is this: we will call a threaded entry method called "run" from the constructor of the Fibonacci chare a threaded entry method * [1]. It will fire two children chares, which will return result to our "response" entry method . However, after firing both children, the calling entry method simply suspends itself, after storing its threadID in an object variable. The "response" method adds up the results being returned, counts up the number of response received, and when it has received two responses, "awakens" the suspended thread. Note that CthAwaken only puts the thread in the ready queue, and does not immediately return control to it. It must wait for its turn, since there may be other chares and messages waiting to be executed on this processor. When scheduled, the `run` thread simply continues (i.e. it is as if it "returned" from the CthSuspend call), and sends a response to its parent.

We need to take into account who the parent is: ff your parent is another fib chare (as will be the case for most chares), you send the result to it's response method. But if you are the root of the tree of chares, created by the main chare, then you just print the result and terminate the program by calling CkExit().

Figures 9.9 and 9.10 show the code corresponding to this idea. Make sure you understand how it expresses each of the ideas mentioned above.

```
mainmodule fib_threads {

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

  chare fib {
    entry fib(int amIroot, int n, CProxy_fib parent);
    entry [threaded] void run(int n);
    entry void response(int);
  };
};
```

Figure 9.9: Finding fibonnaci using threads: the interface file `fib_threads.ci`

How does this code compare with the original code which did not use threaded entry methods? Not that much simpler or shorter, is it? But at least now you know the most elementary method for controlling threads. And yes, the main thread kind of looks nicer, since it captures the logic in one place; except that we have to understand how the response method captures the data and resumes the thread.

[1]* it is conventional to name a threaded method "run", especially if it is the only threaded method in the chare, and includes the main life cycle of the chare. But it can be named any other name you wish to use

```
class Main : public CBase_Main
{
  public:
    Main(CkMigrateMessage *m) {}
    Main(CkArgMsg * m) {
      if(m->argc < 2) CmiAbort("./fib_threads N");
      int n = atoi(m->argv[1]);
      CProxy_fib pfib;
      CProxy_fib::ckNew(1, n, pfib);
    }
};

class fib : public CBase_fib
{
  private:
    int result, count, IamRoot;
    CthThread  tid; CProxy_fib parent;
  public:
    fib(CkMigrateMessage *m) {}
    fib(int amIRoot, int n, CProxy_fib _parent) {
      IamRoot = amIRoot;
      parent = _parent;
      thisProxy.run(n);
    }

    void run(int n) {
      tid = CthSelf();
      if (n< THRESHOLD) {
        result = seqFib(n);
      } else {
        CProxy_fib::ckNew(0,n-1, thisProxy);
        CProxy_fib::ckNew(0,n-2, thisProxy);
        result = 0;
        count = 2;
        CthSuspend(); }
        if (IamRoot) {
          CkPrintf("The requested Fibonacci number is : %d\n", result);
          CkExit();
        } else parent.response(result);
    }

    void response(int fibValue) {
      result += fibValue;
      count--;
      if(!count)        CthAwaken(tid);
    }
};
#include "fib_threads.def.h"
```

Figure 9.10: Finding fibonnaci using threads: the C++ file fib_threads.C

9.4.1 Fibonacci Numbers with Futures

The second method gets rid of the "response" entry method. It uses a construct we have not seen before: a *Future*. A *Future* is a construct that you can create from a threaded entry method; it will contain a value at some future time. If you "touch" a Future before it has a value, the calling thread blocks (i.e. suspends) "touching" here simply means accessing the value, and you have to use a special function call to access the value.

With that construct at our disposal, the idea for the program is: we will make the constructor call "run", a threaded entry method, as above. But when we create a child chare, we provide it with a reference to a new future (one future for each call). Then we just wait for each of them in succession by trying to access their values. Once both futures have "realized" (i.e. have their values set), we add their values, and set the future that our parent sent us.

The function calls provided by Charm++ for implementing its version of *futures* are as follows. `CkFuture` is a system defined type that holds a "future" value. `CkCreateFuture()` creates and returns a future structure. This is an opaque structure (i.e. you don't get to see its data members or what else is inside of it, except via the function calls described here) that can be sent in messages to other chares. `CkWaitFuture` function, given a future structure as a parameter, checks if the future value has been set; if so it returns with the value (which must be a pointer to a `message` type). If not, it suspends the thread and resumes it when the value is available. A remote chare which holds a future structure can set its value by calling: CkSentToFuture(f, m) where f is a future and m is a `message *` (i.e. pointer to a message). You can also "probe" to check if a future's value is set, without making the thread block, using `CkProbeFuture()`...but we do not use that call here.

```
mainmodule fib_futures {

  message ValueMsg;

  mainchare Main {
    entry Main(CkArgMsg *m);
    entry [threaded] void run(int n);
  };

  chare fib {
    entry fib(int n, CkFuture f);
    entry  [threaded] void run(int n, CkFuture f);
  };
};
```

Figure 9.11: Finding fibonnaci using Future: the interface file `fib_futures.ci`

Figures 9.11 and 9.12 show the code corresponding to this idea. Note that now a chare does not have to worry about whether it is at the root (and so must return the result to the main chare) or not. Both types of chares are getting their results back via futures. This further simplifies the code (of both the main chare and the fib chare); e.g., notice that we do not now have to keep track of whether a fib chare "isRoot", unlike our previous implementation.

9.5 Why not to use threads

Threaded methods can be convenient, and threads are pretty light weight. For example, creation and context switching between threads typically cost less than a microsecond on today's machines. But they do add to the cost that tiny bit; in contrast, the structured dagger has a much lower overhead.

More significantly, a thread requires a stack of its own. By default, the system creates a pretty small stack (typically 4 KB). You can specify a larger stack for a threaded entry method using the following command line argument: `+stacksize <size in bytes>`.

Yet, the problem is the need to specify adequate stack size makes the method somewhat error prone, and possibly wasteful, because a stack once allocated occupies memory until the threaded method finishes, even when the stack is not "filled".

These are some reasons why you should avoid threads and use structured dagger as much as possible. But there are situations where a threaded method is indispensable. For example, if an entry method `M` calls a function `f` which calls a function/method `g`, and inside of `g`, you need some value or data form another chare. It is much more convenient to make `M` a threaded method, and use a sync method call inside `g` to get the remote value. Without `M` being threaded, you will have to split the code of `M`, `f` and `g`, and put the parts that execute after the remote value is returned into separate methods or functions. Such situations, along with situations when elegance is more important than small performance impact (and the stack size can be easily guessed), are times when using threaded methods make sense.

```
class Main : public CBase_Main
{
  public:
    Main(CkMigrateMessage *m) {};
    Main(CkArgMsg * m) {
      if(m->argc < 2) CmiAbort("./fib N.");
      int n = atoi(m->argv[1]);
      thisProxy.run(n);
    }

    void run(int n) {
      CkFuture  f = CkCreateFuture();
      CProxy_fib::ckNew(n, f);
      ValueMsg * m = (ValueMsg *) CkWaitFuture(f);
      CkPrintf("The requested Fibonacci number is : %d\n", m->value);
      CkExit();  delete m;
    }
};

class fib : public CBase_fib
{
  private:
    int result, count, IamRoot;
    CProxy_fib parent;
  public:
    fib(CkMigrateMessage *m) {};
    fib(int n, CkFuture f){ thisProxy.run(n, f); }

    void run(int n, CkFuture f) {
      if (n< THRESHOLD)
        result =seqFib(n);
      else {
        CkFuture f1 = CkCreateFuture();
        CkFuture f2 = CkCreateFuture();
        CProxy_fib::ckNew(n-1,  f1);
        CProxy_fib::ckNew(n-2,  f2);

        ValueMsg * m1 = (ValueMsg *) CkWaitFuture(f1);
        ValueMsg * m2 = (ValueMsg *) CkWaitFuture(f2);

        result = m1->value + m2->value;
        delete m1; delete m2; }
        ValueMsg *m = new ValueMsg();
        m->value = result;
        CkSendToFuture(f, m);
    }
};
#include "fib_futures.def.h"
```

Figure 9.12: Finding fibonnaci using Future: the C++ file fib_futures.C