Messages, Entry methods that return values, and Threaded entry methods

Laxmikant (Sanjay) Kale
http://charm.cs.illinois.edu
Parallel Programming Laboratory
Department of Computer Science
University of Illinois at Urbana Champaign

Relaxing a restriction

- Earlier we said that:
  - Entry methods, once started, do not pause. They return control to the charm++ scheduler only after they've finished execution
- Today, we will describe constructs that relax this restriction
  - Also, we will define a special class of entry methods that have return values
    - i.e. regular entry methods, rather than "asynchronous" ones
- The baseline model, with the original restrictions:
  - Is a conceptually simpler model, and
  - Is adequate: powerful enough for most situations
    - Especially when extended with structured dagger
  - You should continue to use that whenever possible

sync methods

- Synchronous as opposed to asynchronous
- They return a value - always a “message” type
  - Always a pointer to a message, MsgType *
- Other than that, just like any other entry method:

```
In interface file:
 entry [sync] MsgData * f(double A[2*m], int m );
```

```
In C++ file:
MsgData * f(double X[], int size) {
    …..
    m = new MsgData(..);
    …..
    return m;
}
```

How to invoke a sync method

- We might invoke a sync method just like any other method:
  - MsgData * m = A[i].foo(.. parameters..);
- Do you see any problem with this?

Threaded methods

- Any method that calls a sync method must be able to suspend
  - Need to be declared as a “threaded” method.
  - A threaded method of a chare C
    - Can suspend, without blocking the processor
    - Other chares can then be executed
    - Even other methods of chare C can be executed

A complete example

- A chare array A of N elements, and each element holds a single number
- Check if the numbers are already in a sorted order?
- Each chare checks with its right neighbor, in parallel, and combine there results via a reduction

Interface File - .ci

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

  mainchare Main {
    entry Main(CkArgMsg *msg);
    entry [reductiontarget] void allDone(CkReductionMsg *m);
  };

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

Class Main - .C

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

      CProxy_SimpleArray elems =
                CProxy_SimpleArray::ckNew(numElements);

      CkCallback *cb = new CkCallback(
            CkIndex_Main::allDone(NULL), thisProxy);

      elems.ckSetReductionClient(cb);
      elems.run();
    }
```

Class Main - .C

```
class Main : public CBase_Main….contd

    public:
        void allDone(CkReductionMsg *m) {
            int *sorted = (int *) m->getData();
            if( *sorted == numElements) {
                printf(" Elements were sorted \n");
            } else {
                printf(" Elements were not sorted \n");
            }
            CkExit();
        }
};
```

.C contd.

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

```
class SimpleArray : public CBase_SimpleArray {
    private:
        double myValue;
    public:
        SimpleArray() {
            myValue = drand48();
        }
```

.C contd.

```
        void run() { // threaded method
           int contrib = 1;
           if(thisIndex < numElements - 1) {
              MsgData *m = thisProxy(thisIndex+1).blockingGetValue();
              if(myValue > m->value) contrib = 0;
           }
           contribute(sizeof(int), &contrib, CkReduction::sum_int);
        }

        MsgData* blockingGetValue() { // blocking method
            MsgData * m = new MsgData();
            m->value = myValue;
            return m;
        }
};
```

Discussion

- How can you write the same code without threaded methods:
  - Without sdag? (structured dagger)
  - With sdag?
- Which way is better?
- Which way is more efficient?
- What can you say about other situations beyond this simple example?
- Can you write doubly recursive Fibonacci code with sync methods?

Once you have threaded methods…

- You can make them suspend in multiple ways ways:
  - Futures (CkFuture)
  - Suspend and Awaken

Fibonacci with Futures - .ci

```
mainmodule fib {
  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);
  };
};
```

Fibonacci with Futures - main

```
class Main : public CBase_Main {
public:
        Main(CkMigrateMessage *m) {};
    Main(CkArgMsg* m) {
        thisProxy.run(atoi(m->argv[1]));
    }
        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();
    }
};
```

Fibonacci with Futures - fib

```
class Fib : public CBase_Fib {
public:
  int result;
  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);
  }
};
```

Fibonacci with Futures - continued

```
int seqFib(int n) {
  if (n<2) return n;
  else return (seqFib(n-1) + seqFib(n-2));
}
class ValueMsg : public CMessage_ValueMsg {
public: int value;
};
```

Fibonacci with explicit thread calls

- All synchronization constructs, such as futures, are implemented using these basic thread library calls
  - CthThread tid = CthSelf();
  - CthSuspend();
  - CthAwaken(tid);

```
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);
  };
};
```

```
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::ckNew(1, n, NULL);
    }
};
```

```
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);
 }
```

Finding Approximate Median

- You are given K*num_chares numbers spread across num_chares chares of a chare array
- Find the approxmiate median of those numbers
  - i.e. approximately half numbers are smaller and half are larger
  - Approximate, so that a small percentage difference is tolerated
    - E.g. a number with 49.5% smaller and 51.5% larger than it is acceptable as approximate median
    - This tolerance is allowed so as to make it converge faster

the interface file median.ci

```
mainmodule Median {

  mainchare Main {
    entry Main(CkArgMsg* m);
    entry [threaded] void computeMedian();
  };

  array [1D] Partition {
    entry Partition(void);
    entry void queryCounts(double median, CkCallback &cb);
  };
};
```

class Main in the C++ file median.C

```
#include "Median.decl.h"

/*readonly*/ int K;

class Main: public CBase_Main {

private:
  CProxy_Partition partition_array;
  double median;

public:
  Main(CkArgMsg* m) {
    if(m->argc < 4){
      CkAbort("\nUsage: ./median [num_chares] [numbers_per_chare] [suggested median]\n");
    }
    int num_chares = atoi(m->argv[1]);
    K = atoi(m->argv[2]);
    median = atoi(m->argv[3]);
    delete m;

    partition_array = CProxy_Partition::ckNew(num_chares);
    thisProxy.computeMedian();
  }

  void computeMedian() {
    int iteration = 0;
    double min_range = 0.0;
    double max_range = 1.0;

```

class Main in the C++ file median.C

```
    do{
      CkReductionMsg* msg;
      partition_array.queryCounts(median, CkCallbackResumeThread((void*&)msg));

      int *counts=(int *) msg->getData();
      int numSmaller = counts[0];
      int numLarger = counts[1];
      double error = (double)abs(numSmaller-numLarger)/(numSmaller + numLarger);
      if(error < 0.01) break;

      if(numSmaller > numLarger)
           max_range = median;
      else min_range = median;

      median = (min_range+max_range)/2;
      iteration++;
    }  while(true);

    CkPrintf("\nResult calculated median = %lf (in %d iterations)\n", median, iteration);
    CkExit();
  }
};

class Partition: public CBase_Partition {

  public:
```

class Partition in the C++ file median.C

```

    Partition() {
      numbers = new double[K];
      srand48(time(NULL));
      for(int i=0;i<K;i++){
        numbers[i] = drand48();
      }
    }

    Partition(CkMigrateMessage* m):CBase_Partition(m){
    }

    void queryCounts(double median, CkCallback &cb){
      int counts[2];
      counts[0] = counts[1] = 0;
      for(int i=0;i<K;i++){
        if(numbers[i]<median)
          counts[0]++; // # smaller than median
        else
          counts[1]++; // # larger than median
      }

      contribute(2*sizeof(int), counts, CkReduction::sum_int, cb);
    }
};

#include "Median.def.h"
```