CHARM++ EXAMPLES
A few examples

• Some with code and some top level designs
  1. How to find median of data spread out over a chare array
  2. How to send a small number of “wrong” elements to their correct homes in an otherwise sorted array
  3. How to sort elements in a chare array:
     1. Using a parallel version of quick sort (may skip)
     2. Using histogram sort
Discussion and Idea for median finding

- N chares in a chare array
- Each containing a set of numbers
- Median:
  - a number X such that about half of all the numbers are smaller than it and half larger
- How to find the median?
- Idea:
  - Main or chare0 makes a guess (how?)
  - Broadcast to everyone
  - Everyone counts smaller/larger
  - Reduce to main
  - Main updates the guess and repeats
Median Example: median.ci

mainmodule Median {
  readonly CProxy_Main mainProxy;
  mainchare Main {
    entry Main(CkArgMsg* m);
    entry [reductiontarget] void informRoot(int counts[2]);
    entry void computeMedian(){ ... }
  }
}

array [1D] Partition {
  entry Partition();
  entry void queryCounts(double median);
}

entry void computeMedian(){
  while(true){
    serial { partition_array.queryCounts(median); } 
    when informRoot(int counts[]) serial {
      int nSmaller = counts[0];
      int nLarger = counts[1];
      double error =
        (double)abs(nSmaller-nLarger)/(nSmaller + nLarger);
      if(error < 0.01){
        CkPrintf("Median = %Lf (in %d iterations)\n", median, iter);
        CkExit(); }
      if(nSmaller > nLarger)
        max_range = median;
      else
        min_range = median;
      median = (min_range+max_range)/2;
      iter++;
    }
  }
}
#include "Median.decl.h"

/*readonly*/ CProxy_Main mainProxy;

/*readonly*/ int K;

class Main: public CBase_Main { Main_SDAG_CODE

private:
    CProxy_Partition partition_array;
    double median, min_range, max_range;
    int iter;

public:
    Main(CkArgMsg* m) {
        iter = 0, min_range = 0.0, max_range = 1.0;
        K = atoi(m->argv[2]);
        median = atof(m->argv[3]); // initial guess provided on command line
        mainProxy = thisProxy;
        partition_array = CProxy_Partition::ckNew(atoi(m->argv[1]));
        mainProxy.computeMedian();
    }
};
Median Example: median.C II

class Partition: public CBase_Partition {

public:
    double *numbers;

    Partition(int guess) {
        numbers = new double[K];
        srand48(time(NULL));
        for(int i=0; i<K; i++)
            numbers[i] = drand48();
    }
    void queryCounts(double median) {
        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,
                   CkCallback(CkReductionTarget(Main, informRoot), mainProxy));
    }

#include "Median.def.h"
Relaxing an assumption

• We assumed in the above code:
  – The main chare knows the smallest and largest possible values
  – Under what conditions is that valid or efficient?

• How can we relax that assumption?
• Discuss
Improving Our Median Program further

• How can we improve its efficiency?
• What are the costs?
  – Discuss
  – Number of rounds
  – Cost of each round
Improving Our Median Program further

• How can we improve its efficiency?
• What are the costs?
• For each probe, the queryCounts method has to loop through the entire array
  – What if we pre-sort the array?
  – What if we partially sort the array (and keep improving it at every probe)
• How to improve the initial guess?
  – So as to reduce the number of broadcast-reduction iterations
• How to get more information with each reduction?
  – After all the cost of reduction doesn’t change much if we reduce a small vector instead of just 2 counts
  – Histogramming
A somewhat related problem

- Consider a situation in which a chare array is sorted
  - Values are between 0 and M, long integers
  - Without worrying too much about data balance
  - The data distribution is uniform, so, we decide that chare I will hold values between \((I \times M/P, (I+1) \times M/P - 1)\)
    - Where P is the number of chares in the array

- Now, each chare generates a few new items
  - Their value may be anywhere in the range 0..M
  - Let us assume the “few” is really small, like 5 or 10
  - And P is large (say > 10,000)
  - Also, the total data on each chare is large. But that’s immaterial

- How can we send them to their correct home places?
Send a few stragglers home

• Easy enough:

• Just send a message for each new value to its home
  – (it is easy to calculate home)
  – Optimize: don’t send message to yourself
  – Optimize?: combine messages going to the same processor?
    • Rare so we will ignore for now

• The problem?
  – How do we know when we are done
  – So, each chare can start the next phase of computation, for example
Quiescence Detection

• In Charm++, *quiescence* is defined as the state in which no processor is executing any entry method, no messages are awaiting processing, and there are no messages in-flight.

• Charm++ provides a method to detect quiescence:

  • From any chare, invoke `CkStartQD(callback);`

• The system is always doing the background bookkeeping so it can detect quiescence:
  
  – The call just starts a low-overhead distributed algorithm to detect the condition
  
  – It runs concurrently with your application
  
  – So, call `CkStartQD` as late as possible to reduce the overhead.
Quiescence Detection applied to stragglers

• For our problem,
  – we can have one of the charEs (say with index 0) call CkStartQD after it has done its sending
  – With a callback that broadcasts to every charE in the array that quiescence has been attained
  – This means all the stragglers have reached their home
Histogram sort

- Idea: extend the median finding algorithm
- If we have P chares, we need to find P-1 separator keys
  - I.e. values that act as (or define) boundaries between chares
  - Such that everyone has an (approximately) equal number of values
- We can
  - make a guess (called a probe)
  - Collect a histogram (counts)
  - Correct the guesses and prepeat
- When adequate separators are identified:
  - Everyone sends the data to where it belongs
  - Use quiescence detection to make sure all the data is received
  - Sort locally
Histogram sort: interesting optimizations

• Some optimizations to this algorithm exploit charm++’s message driven execution

• E.g. Some chares’ separators may be found early on:
  – Everyone can start sending their values in parallel with histogramming for other chares

• Histogramming and initial local sorting may be overlapped

• Histogram may be decomposed into multiple portions
  – So that it can be pipelined
  – While root is preparing the next guess for one section, the other section is doing it distributed histogramming

See paper by Vipul Harsh: https://charm.cs.illinois.edu/papers/19-02