CHARARE ARRAY SECTIONS
Chare Array Review

- Arbitrarily-sized collection of chares
- Every item in the collection has a unique index and proxy
- Can be indexed like an array or by an arbitrary object
- Can be sparse or dense
- Elements may be dynamically inserted and deleted
- Elements can be migrated
Motivation

• It is often convenient to define subcollections of elements within a chare array
  – Example: rows or columns of a 2D chare array
  – One may wish to perform collective operations on the subcollection (e.g. broadcast, reduction)

• *Sections* are the standard subcollection construct in Charm++
  – A section is a subset of a Chare Array
Section Creation

• Through explicit enumeration:

```cpp
CkVec<CkArrayIndex3D> elems; // add array indices
for (int i=0; i<10; i++)
    for (int j=0; j<20; j+=2)
        for (int k=0; k<30; k+=2)
            elems.push_back(CkArrayIndex3D(i, j, k));

CProxySection_Hello proxy =
    CProxySection_Hello::ckNew(helloArrayID,
    elems.getVec(), elems.size());
```
Section Creation

• Through index range specification:
• Specify array ID of the base char array and the individual char array elements of the array participating in the section

```cpp
CProxySection_Hello proxy =
CProxySection_Hello::ckNew(helloArrayID,
    0, 9, 1, 0, 19, 2, 0, 29, 2);
```
Section Class Generation

- Section proxy classes are automatically generated for each chare and group defined in the .ci file
- Placed into decl.h and def.h files
Using Sections

CProxySection_Hello proxy;

// section broadcast
proxy.someEntry(...)

// sections are unranked, not allowed
proxy[0].someEntry(...)

• For example implementations, see
  • $(CHARM)/examples/charm++/arraysection
  • https://charmplusplus.org/miniApps/#leanmd
Spanning Trees

• CkMulticast implements tree algorithms for multicasts and reductions
  – Messages are routed over a *spanning tree* of the section elements
• Default branching factor is 2,
  – but a different number can be specified while creating a section
  – Add branching factor as a last integer parameter
CkMulticast Messages

- To use CkMulticast library, all multicast messages must inherit from CkMcastBaseMsg
  - CkMcastBaseMsg must be inherited from first
  - No parameter marshalling is allowed in entry methods used as targets of multicast

```cpp
class HiMsg : public CkMcastBaseMsg, public CMessage_HiMsg
{
    public:
        int *data;
        ..
};
```
Reductions: setReductionClient

• An array element can be a member of multiple array sections
  – It is necessary to disambiguate which array section reduction it is participating in each time it contributes to one
• The reduction callback should be set at the time of creation.
  – This callback will be invoked after each reduction is complete

```c
CkCallback *cb = new CkCallback(
    CkReductionTarget (Cell, reduceForces),
    thisProxy(thisIndex.x,thisIndex.y,thisIndex.z)));
mySecProxy.setReductionClient(cb);
```
Reductions: CkSectionInfo

- A data structure called "CkSectionInfo" is created by CkMulticastMgr for each array section that the array element belongs to
  - During a section reduction, the array element must pass the CkSectionInfo as a parameter in the contribute()
  - This CkSectionInfo “cookie” can be retrieved from a previous message that was sent through CkMulticastMgr
    - Therefore, you can contribute into a reduction **only** following a broadcast to the same section.
Reductions with CkMulticast

CkSectionInfo cookie;

void SayHi(HiMsg *msg)
{
    // this is a broadcast to SayHi using
    // the section we want to to contribute to
    // update section cookie every time
    CkGetSectionInfo(cookie, msg);
    int data = thisIndex;
    mcastGrp->contribute(sizeof(int), &data,
                           CkReduction::sum_int, cookie);
}

Callbacks

• As with array reductions, a callback needs to be specified with each contribute
  – OR a default callback should be specified using setReductionClient
Example: Matrix Multiplication

• Inputs: 2D chare arrays A, B of matrix blocks
• Output: 2D chare array C of matrix blocks
• Elements of A and B multicast their blocks to a section comprising a row or column of C
• Exercise: implement algorithm
Example: LeanMD

• Lennart-Jones Dynamics
• We have a 3D array of Cells
• And a 6D array of cell-pairs
  – (also called “compute” objects in the leanmd miniApp at https://charmplusplus.org/miniApps/#leanmd)
Object Based Parallelization for MD: Force Decomposition + Spatial Decomposition

Now, we have many objects to load balance:

- Each diamond can be assigned to any proc.
- Number of diamonds (3D):
  - $14 \cdot \text{Number of Patches}$
  - 2-away variation:
    - Half-size cubes
    - 5x5x5 interactions
  - 3-away interactions: 7x7x7
Parallelization Using Charm++

The computation is decomposed into “natural” objects of the application, which are assigned to processors by Charm++ RTS.
entry void run() {
    for(stepCount = 1; stepCount <= finalStepCount; stepCount++) {
        atomic {
            sendPositions();
        }
        for(forceCount = 0; forceCount < inbrs; forceCount++) {
            when receiveForces[stepCount](int iter, vec3
                forces[n], int n)
            atomic {
                addForces(forces);
            }
            atomic {
                updateProperties();
            }
            if ((stepCount % MIGRATE_STEP_COUNT) == 0) {
                atomic {
                    sendParticles();
                }
                when statements for receiving particles from
                neighbors
            }
        }
    }
}
} //end of for loop
atomic {
    contribute(0, CkReduction::NULL,
        CkCallback(CkReductionTarget(Main, done), mainProxy));
}
} //end of run
entry void run() {
    for(stepCount = 1; stepCount <= finalStepCount; stepCount++) {
        if (thisIndex.x1==thisIndex.x2 &&
            thisIndex.y1==thisIndex.y2 &&
            thisIndex.z1==thisIndex.z2)
            when calculateForces[stepCount](ParticleData *data)
                atomic { selfInteract(data); }
        else {
            when calculateForces[stepCount] (ParticleData *data)
                atomic { bufferedData = data; }
            when calculateForces[stepCount](ParticleData *data)
                atomic { interact(data); }
        }
        // contribute/send forces to the cells involved
    }
    // end of for loop
};//end of run
Using section in sendPositions

• Especially useful if you are using a 2-away formulation:
  – There are $5 \times 5 \times 5 = 125$ pairs to which each cell must send its coordinates
    • Same data to everyone, so it is a Multicast

• This happens repeatedly, every iteration
  – At load balancing time the locations of pairs may change, but the set is the same

• So, each cell sets up its own section of pairs
• Each pair is a member of two [or one] sections
Expressing in Charm++

• Two chare arrays:
  – Cells: a 3D array of chares
  – Pairs: one object for each “neighboring” chare

• What is the dimensionality of “pairs”?
  – Idea 1: make it a 3D array.. Does it work?
  – Idea 2: Make it a 1D array,
    • Explicitly assign indices to chares: the pair object between Cells[2,3,4] and Cells[2,3,5] is Pairs[somelIndex].
  – Idea 3: Make it a 6D array
    • Pairs[2,3,4,2,3,5]
    • But: (a) it is sparse and
    • (b) symmetry? Do we also have Pairs[2,3,5,2,3,4]
    • Use only one of them.. (say “smaller” in dictionary order)
Object Based Parallelization for MD (with sections)

• All pairs in the box constitute a section for the central proc:
  • Central chare uses CkMulticast for optimized broadcasts to this section
  • Without CkMulticast, it would have been point-to-point sends for all
  • Reductions are used across the section to aggregate results for force calculation