Optimizing Load Balance Using Parallel Migratable Objects

Laxmikant V. Kalé, Eric Bohm

Parallel Programming Laboratory
University of Illinois Urbana-Champaign

2012/9/25

Optimization Do's and Don'ts

- Correcting load imbalance is an optimization
  - Like all optimizations, it is a cure to a performance ailment
    - Diagnose the ailment before applying treatment
  - Use performance analysis tools to understand performance
    - Ironically, we cover that material later...
    - But your process should be to use them early
  - A sampling of tools of interest:
    - Compiler reports for inlining, instruction level parallelism, etc
    - Profiling tools (gprof, xprof, manual timing)
    - Hardware counters (PAPI, PCL, etc)
    - Valgrind memory tool suite
    - Parallel analysis tools: Projections, HPCToolkit, TAU, JumpShot, etc.

How to Diagnose Load Imbalance

- Often hidden in statements such as:
  - Very high synchronization overhead
    - Most processors are waiting at a reduction
- Count total amount of computation (ops/flops) per processor
  - In each phase!
  - Because the balance may change from phase to phase

Golden Rule of Load Balancing

Fallacy: objective of load balancing is to minimize variance in load across processors

Example:

- 50,000 tasks of equal size, 500 processors:
  - A: All processors get 99, except last 5 gets $100 + 99 = 199$
  - OR, B: All processors have 101, except last 5 get 1

Identical variance, but situation A is much worse!

Golden Rule: It is ok if a few processors idle, but avoid having processors that are overloaded with work

Finish time = maxi(Time on processor i)

excepting data dependence and communication overhead issues

The speed of any group is the speed of slowest member of that group.

Automatic Dynamic Load Balancing

- Measurement based load balancers
  - Principle of persistence: In many CSE applications, computational loads and communication patterns tend to persist, even in dynamic computations
  - Therefore, recent past is a good predictor of near future
  - Charm++ provides a suite of load-balancers
  - Periodic measurement and migration of objects
- Seed balancers (for task-parallelism)
  - Useful for divide-and-conquer and state-space-search applications
  - Seeds for charm++ objects moved around until they take root

Using the Load Balancer

- link a LB module
  - `-module <strategy>`
  - RefineLB, NeighborLB, GreedyCommLB, others
  - EveryLB will include all load balancing strategies
- compile time option (specify default balancer)
  - `-balancer RefineLB`
  - runtime option
  - `+balancer RefineLB`

Code to Use Load Balancing

- Insert `if (myLBStep) AtSync() else ResumeFromSync();` call at natural barrier
- Implement `ResumeFromSync()` to resume execution
  - Typical ResumeFromSync contribute to a reduction

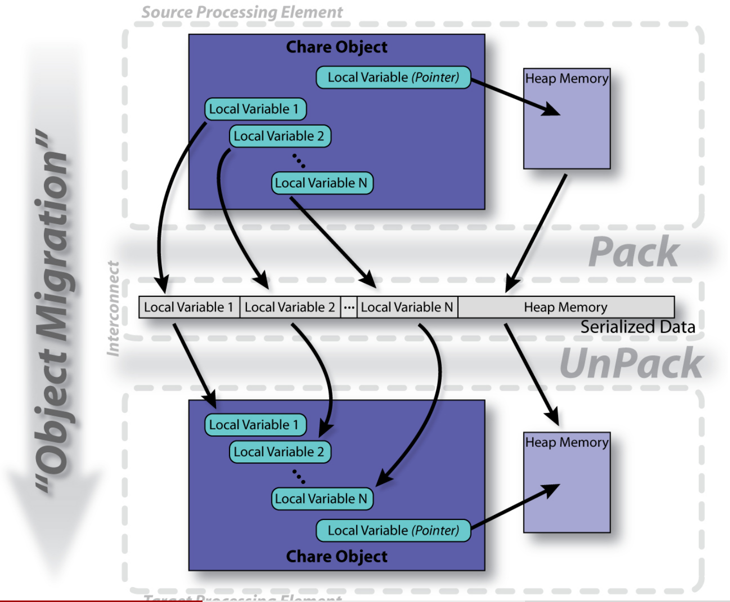
Example: Stencil

```
while (!converged) {
  atomic {
    int x = thisIndex.x, y = thisIndex.y, z = thisIndex.z;
    copyToBoundaries();
    thisProxy(wrapX(x−1),y,z).updateGhosts(i, RIGHT, dimY, dimZ, right);
    /* ...similar calls to send the 6 boundaries... */
    thisProxy(x,y,wrapZ(z+1)).updateGhosts(i, FRONT, dimX, dimY, front);
  }
  for (remoteCount = 0; remoteCount < 6; remoteCount++) {
    when updateGhosts[i](int i, int d, int w, int h, double b[w*h])
    atomic { updateBoundary(d, w, h, b); }
  }
  atomic {
    int c = computeKernel() < DELTA;
    CkCallback cb(CkReductionTarget(Jacobi, checkConverged), thisProxy);
    if (i%5 == 1) contribute(sizeof(int), \&c, CkReduction::logical_and, cb);
  }
  if (i % lbPeriod == 0) { atomic { AtSync(); } when ResumeFromSync() {} }
  if (++i % 5 == 0) {
    when checkConverged(bool result) atomic {
      if (result) { mainProxy.done(); converged = true; }
    }
  }
}
```

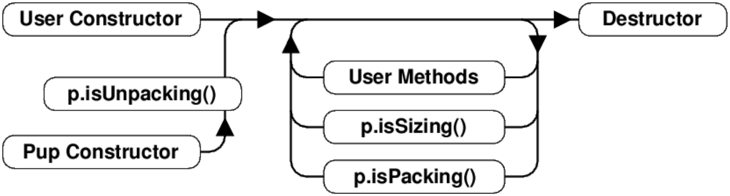
Chare Migration: motivations

- Chares are initially placed according to a placement map
  - The user can specify this map
- While running, some processors might be overloaded
  - Need to rebalance the load
- Automatic checkpoint
  - Migration to disk
- Chares are made serializable for transport using the Pack UnPack (PUP) framework

The PUP Process



PUP Usage Sequence



- Migration out:
  - ckAboutToMigrate
  - Sizing
  - Packing
  - Destructor
- Migration in:
  - Migration constructor
  - UnPacking
  - ckJustMigrated

Writing a PUP routine

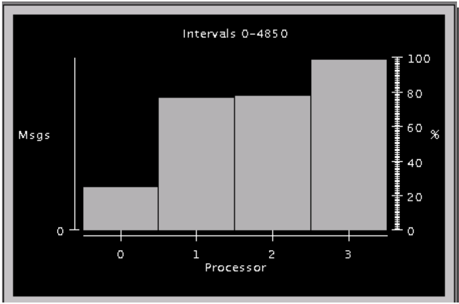
```
class MyChare : public
      CBase_MyChare {
   int a; float b; char c;
   float localArray[LOCAL_SIZE];
   int heapArraySize;
   float* heapArray;
   MyClass *pointer;

   public:
    MyChare();
    MyChare(CkMigrateMessage *
          msg) {};
    ~MyChare() {
      if (heapArray != NULL) {
        delete [] heapArray;
        heapArray = NULL;
      }
};
```

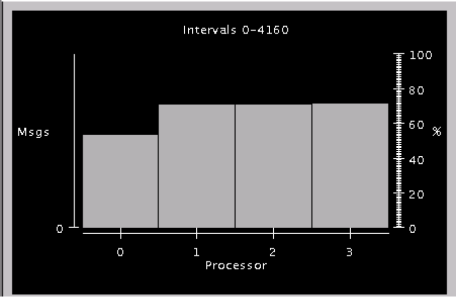
```
void pup(PUP::er &p) {
    CBase_MyChare::pup(p);
    p | a; p | b; p | c;
    p(localArray, LOCAL_SIZE);
    p | heapArraySize;
    if (p.isUnpacking()) {
      heapArray = new float[
            heapArraySize];
    }
    p(heapArray, heapArraySize);
    int isNull = (pointer==NULL)
          ? 1 : 0;
    p | isNull;
    if (!isNull) {
      if (p.isUnpacking()) pointer =
            new MyClass();
      p | *pointer;
    }
}
```

PUP: Issues

- If variables are added to an object, update the PUP routine
- If the object allocates data on the heap, copy it recursively, not just the pointer
- Remember to allocate memory while unpacking
- Sizing, Packing, and Unpacking must scan the same variables in the same order
- Test PUP routines with +balancer RotateLB

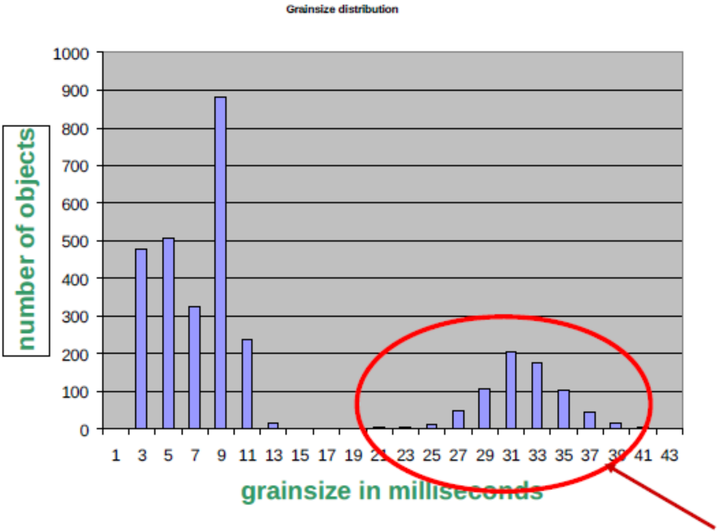
Performance





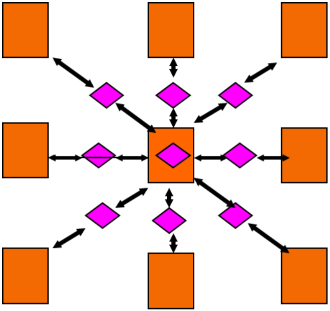
Grainsize and Load Balancing

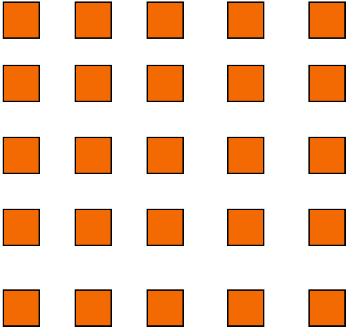
How Much Balance Is Possible?



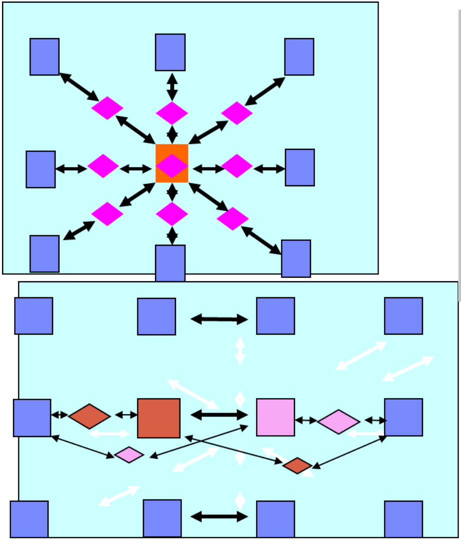
Grainsize For Extreme Scaling

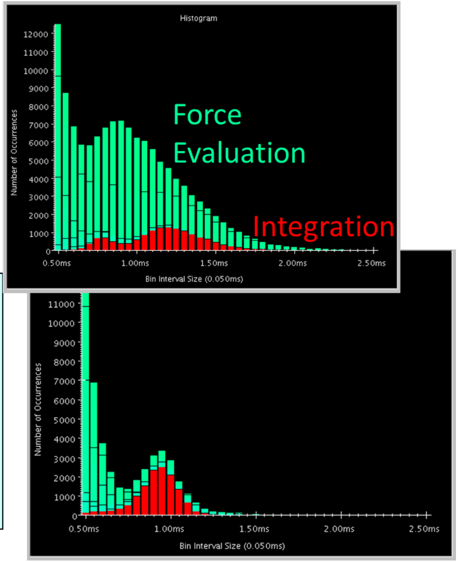
- Strong Scaling is limited by expressed parallelism
  - Minimum iteration time limited lengthiest computation
    - Largest grains set lower bound
- 1-away generalized to k-away provides fine granularity control





NAMD: 2-AwayX Example





Load Balancing Strategies

- Classified by when it is done:
  - Initially
  - Dynamic: Periodically
  - Dynamic: Continuously
- Classified by whether decisions are taken with global information
  - Fully centralized
    - Quite good a choice when load balancing period is high
  - Fully distributed
    - Each processor knows only about a constant number of neighbors
    - Extreme case: totally local decision (send work to a random destination processor, with some probability).
  - Use *aggregated* global information, and *detailed* neighborhood info.

Dynamic Load Balancing Scenarios

- Examples representing typical classes of situations
  - Particles distributed over simulation space
    - Dynamic: because Particles move.

      Highly non-uniform distribution (cosmology)

      Relatively Uniform distribution
- Structured grids, with dynamic refinements/coarsening
- Unstructured grids with dynamic refinements/coarsening

Example Case: Particles

Orthogonal Recursive Bisection (ORB)

- At each stage: divide Particles equally
- Processor dont need to be a power of 2:
  - Divide in proportion
    - 2:3 with 5 processors
- How to choose the dimension along which to cut?
  - Choose the longest one
- How to draw the line?
  - All data on one processor? Sort along each dimension
  - Otherwise: run a distributed histogramming algorithm to find the line, recursively
- Find the entire tree, and then do all data movement at once
  - Or do it in two-three steps.
  - But no reason to redistribute particles after drawing each line.

Dynamic Load Balancing using Objects

Object based decomposition (I.e. virtualized decomposition) helps

- Allows RTS to remap them to balance load
- But how does the RTS decide where to map objects?
- Just move objects away from overloaded processors to underloaded processors
- How is load determined?

Measurement Based Load Balancing

- *Principle of Persistence*
  - Object communication patterns and computational loads tend to persist over time
  - In spite of dynamic behavior
    - Abrupt but infrequent changes
    - Slow and small changes
- Runtime instrumentation
  - Measures communication volume and computation time
- Measurement based load balancers
  - Use the instrumented data-base periodically to make new decisions
  - Many alternative strategies can use the database

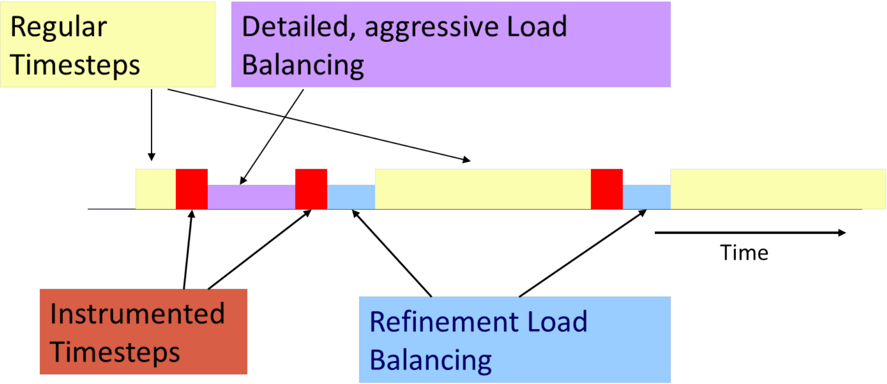
Periodic Load Balancing

Stop the computation?

Centralized strategies:

- Charm RTS collects data (on one processor) about:
  - Computational Load and Communication for each pair
- If you are not using AMPI/Charm, you can do the same instrumentation and data collection
- Partition the graph of objects across processors
  - Take communication into account
    - Pt-to-pt, as well as multicast over a subset
    - As you map an object, add to the load on both sending and receiving processor
  - Multicasts to multiple co-located objects are effectively the cost of a single send

Typical Load Balancing Steps



Object Partitioning Strategies

- You can use graph partitioners like METIS, K-R
  - BUT: graphs are smaller, and optimization criteria are different
- Greedy strategies:
  - If communication costs are low: use a simple greedy strategy
    - Sort objects by decreasing load
    - Maintain processors in a heap (by assigned load)
    - In each step:

      assign the heaviest remaining object to the least loaded processor
  - With small-to-moderate communication cost:
    - Same strategy, but add communication costs as you add an object to a processor
  - Always add a refinement step at the end:
    - Swap work from heaviest loaded processor to "some other processor"
    - Repeat a few times or until no improvement

Object Partitioning Strategies 2

When communication cost is significant:

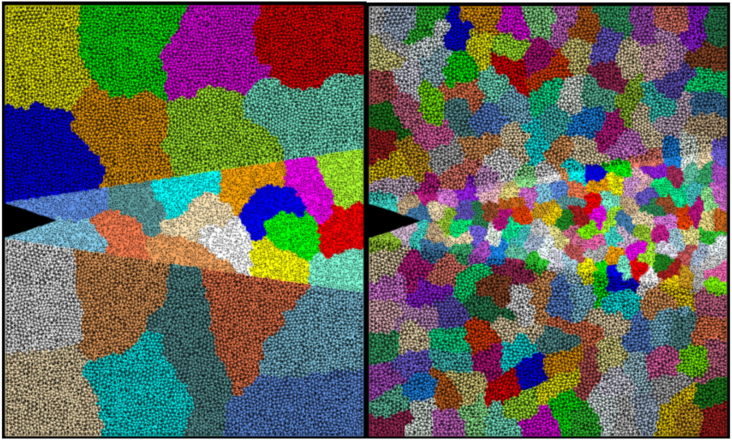
- Still use greedy strategy, but:
  - At each assignment step, choose between assigning O to least loaded processor and the processor that already has objects that communicate most with O.
    - Based on the degree of difference in the two metrics
    - Two-stage assignments:

      In early stages, consider communication costs as long as the processors are in the same (broad) load class,
      In later stages, decide based on load

Branch-and-bound

- Searches for optimal, but can be stopped after a fixed time

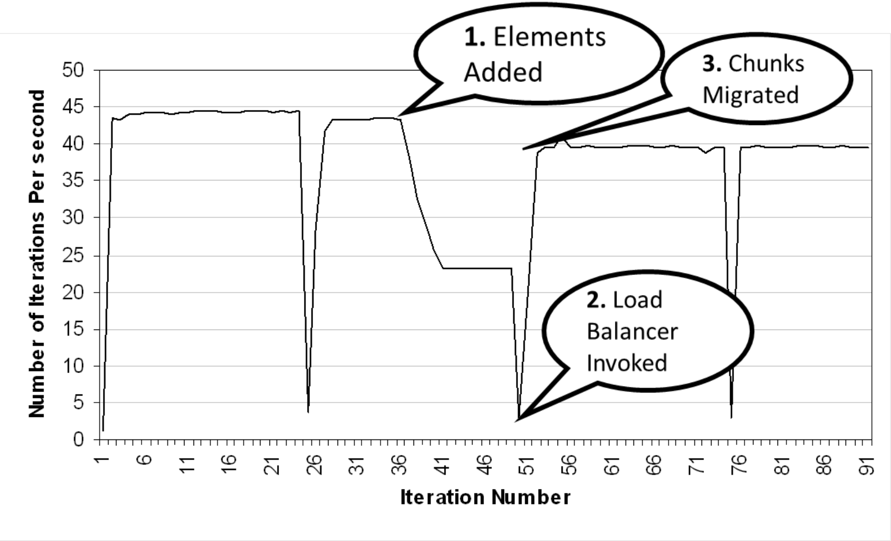
Crack Propagation



Decomposition into 16 chunks (left) and 128 chunks, 8 for each PE (right). The middle area contains cohesive elements. Both decompositions obtained using Metis. Pictures: S. Breitenfeld, and P. Geubelle

As computation progresses, crack propagates, and new elements are added, leading to more complex computations in some chunks

Load Balancing Crack Propagation



Distributed Load balancing

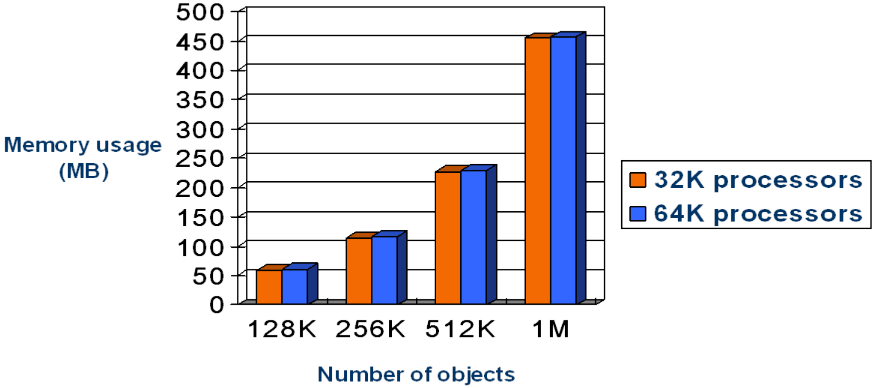
- Centralized strategies
  - Still ok for 3000 processors for NAMD
- Distributed balancing is needed when:
  - Number of processors is large and/or
  - load variation is rapid
- Large machines:
  - Need to handle locality of communication
    - Topology sensitive placement
  - Need to work with scant global information
    - Approximate or aggregated global information (average/max load)
    - Incomplete global info (only neighborhood)
    - Work diffusion strategies (1980s work by Kale and others!)
  - Achieving global effects by local action

Load Balancing on Large Machines

- Existing load balancing strategies dont scale on extremely large machines
- Limitations of centralized strategies:
  - Central node: memory/communication bottleneck
  - Decision-making algorithms tend to be very slow
- Limitations of distributed strategies:
  - Difficult to achieve well-informed load balancing decisions

Simulation Study - Memory Overhead

lb_test experiments performed with the performance simulator BigSim

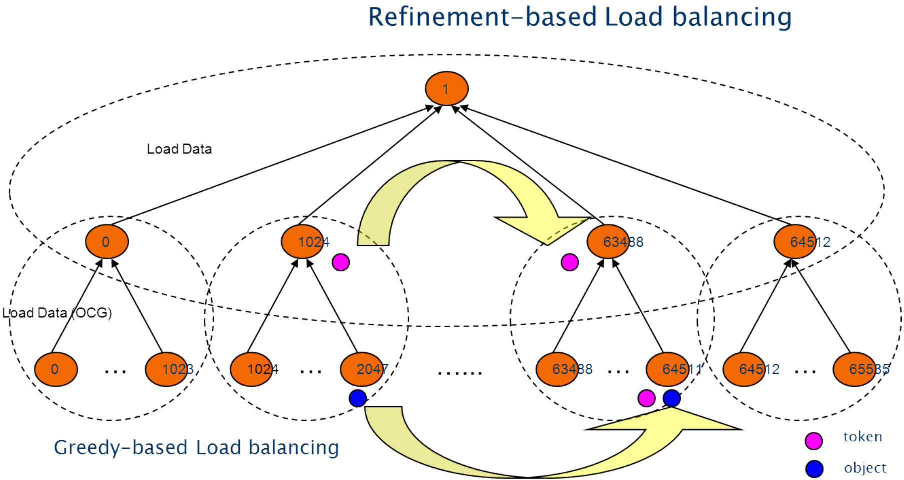


- lb_test benchmark is a parameterized program that creates a specified

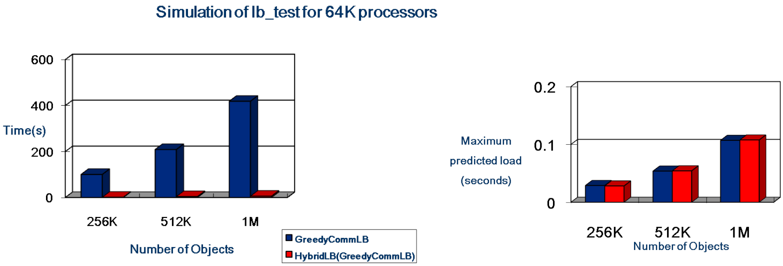
Hierarchical Load Balancers

- Partition processor allocation into processor groups
- Apply different strategies at each level
- Scalable to a large number of processors

Our Hybrid Scheme



Hybrid Load Balancing Performance



Summary

- Use Profiling and Performance Analysis Tools Early
  - *Measure twice, cut once!*
  - Look for overloaded processors, not underloaded processors
- Use PUP for object serialization
  - Enables Migration for Load Balancing or Fault Tolerance
- Don't forget to consider granularity