Programming with Parallel Objects

Session 2: Migratability and Load Balancing

Esteban Meneses

Advanced Computing Laboratory
Costa Rica National High Technology Center

School of Computing
Costa Rica Institute of Technology

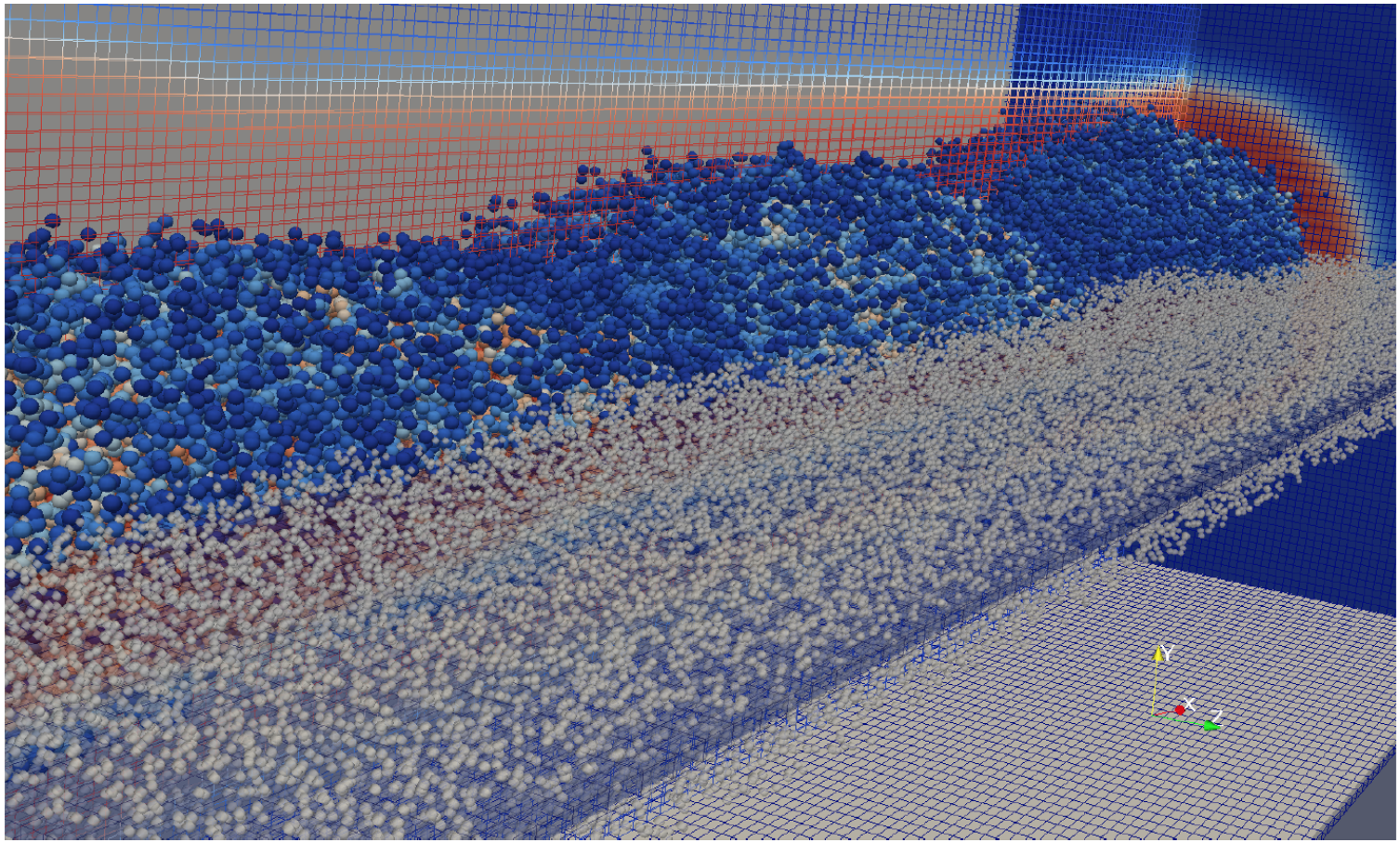
2017

ACKNOWLEDGEMENTS

Most of the content of this tutorial was provided courtesy of Prof. Laxmikant V. Kalé from the Parallel Programming Laboratory (PPL) of the University of Illinois at Urbana-Champaign. The PPL originally developed the ideas behind parallel object programming and has maintained the Charm++ code base for more than 20 years.

ADMINISTRATIVIA

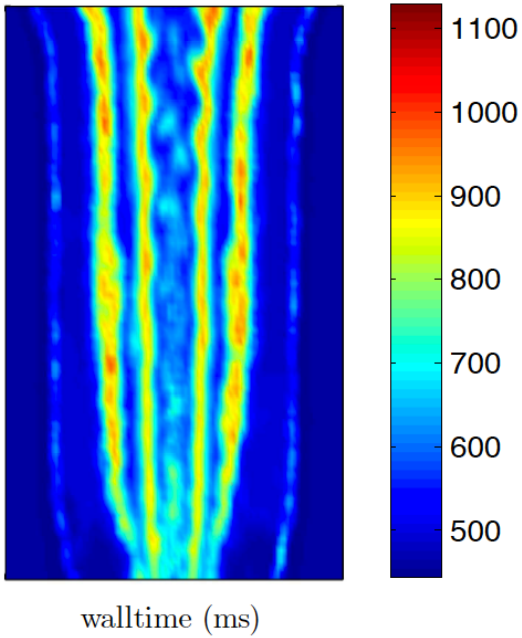
- Official Charm++ website:
  http://www.charmplusplus.org/
- Slides and code for this tutorial:
  https://github.com/emenesesrojas/parallel_objects.git
- Questions on parallel objects programming:
  https://ecar2017.slack.com
- Advanced questions on Charm++:
  ppl@cs.illinois.edu
- Official instructor's email address:
  emeneses@cenat.ac.cr

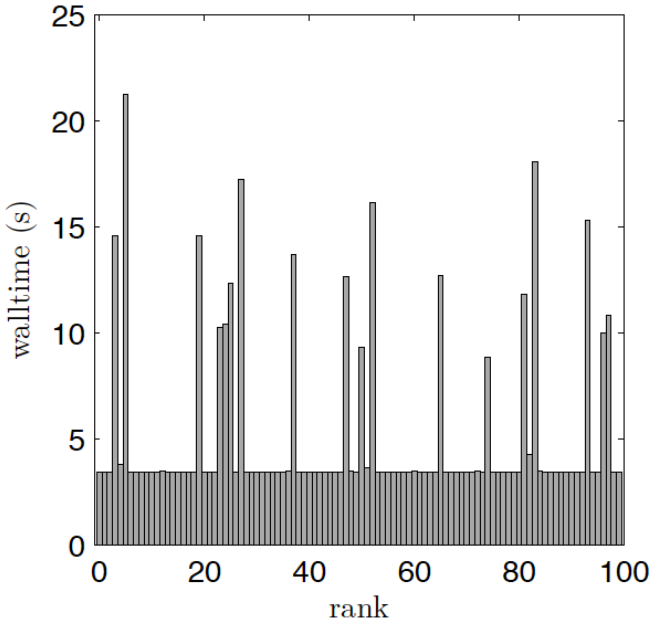


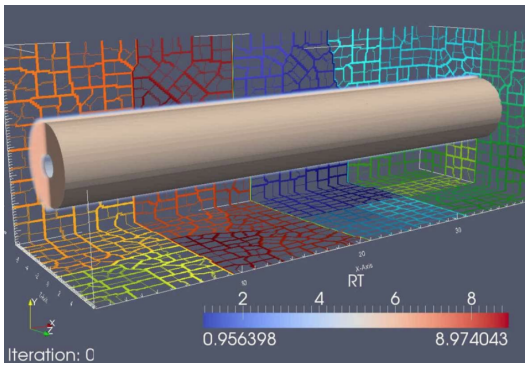
# Grids	# Particles	# Species	Required Memory GBs	GFLOP per iteration	# Iterations	Serial Run-time (1 GFLOP/s)
10^6	6×10^6	9	1.69	29.5	60,000	20.5 days
10^6	6×10^6	19	2.48	90.7	60,000	63 days
5×10^6	50×10^6	19	24.0	544.7	220,000	3.8 years

Simulating a Premixed Flame

Massive load imbalance







CeNAT

Outline

Task Parallelism

Overdecomposition

Migratability

Using Dynamic Load Balancing

CeNAT

Exercise 2.1

Building Charm++ with associated libraries

Download Charm++ code base and build it for your target machine (laptop, desktop, server, supercomputer)

Examples:

- Linux cluster:

```
./build LIBS netlrts-linux-x86_64 gcc -j8 --with-production
```

- Linux cluster on MPI:

```
./build LIBS mpi-linux-x86_64 -j8 --with-production
```

- Linux laptop:

```
./build LIBS multicore-linux-x86_64 -j8 --with-production
```

- Mac laptop:

```
./build LIBS multicore-darwin-x86_64 -j8 --with-production
```

- Windows laptop with VC++:

```
./build LIBS multicore-win-x86_64 --with-production
```

Exercise 2.2

Testing Charm++ installation

Get into Charm++'s source code and look for the test directory:

`charm-6.8.0/tests/charm++`

Run `make test`

Task Parallelism with Objects

Divide-and-conquer

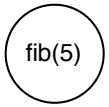
- Each object recursively creates n objects that divide the problem into subproblems
- Each object t then waits for all n objects to finish and then may 'combine' the responses
- At some point the recursion stops (at the bottom of the tree), and some sequential kernel is executed
- Then the result is propagated upward in the tree recursively
- Examples: Fibonacci, Quicksort, ...

Example

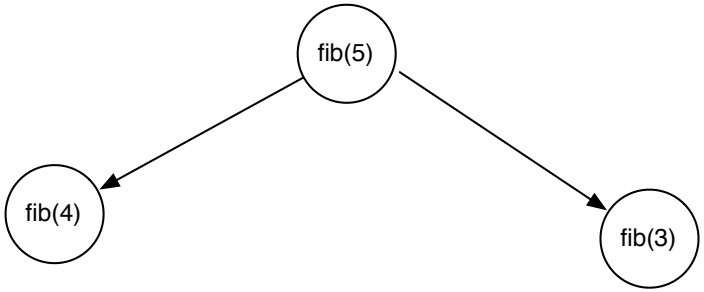
Fibonacci

- Each `Fib` object is a task that performs one of two actions:
  - Creates two new `Fib` objects to compute $fib(n-1)$ and $fib(n-2)$ and then waits for the response, adding up the two responses when they arrive
    - After both arrive, sends a response message with the result to the parent object
    - Or prints the value and exits if it is the root
  - If $n = 1$ or $n = 0$ (passed down from the parent) it sends a response message with n back to the parent object

Fibonacci Execution

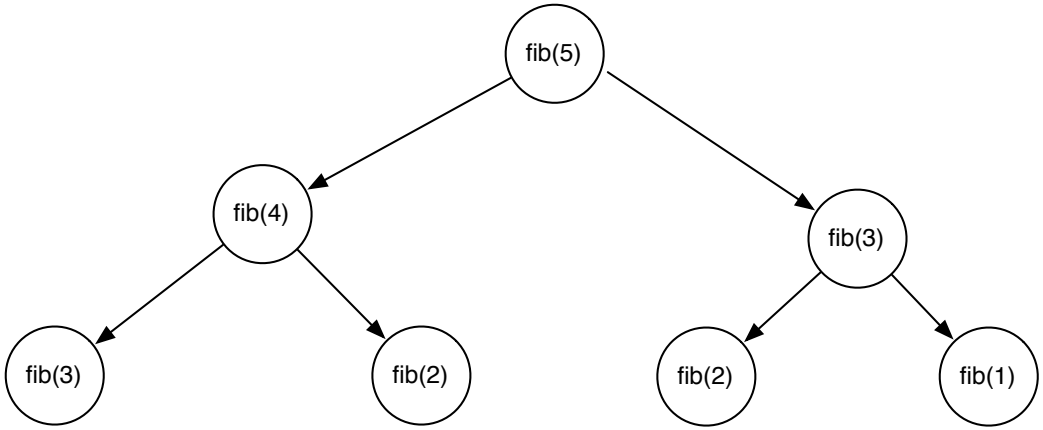


Fibonacci Execution



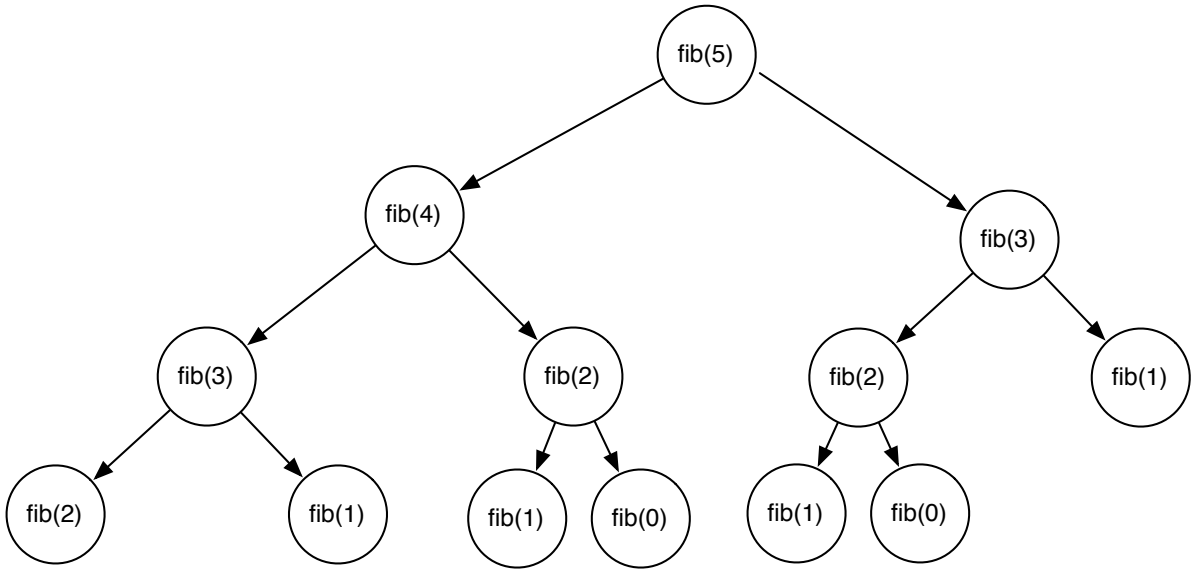
CeNAT

Fibonacci Execution



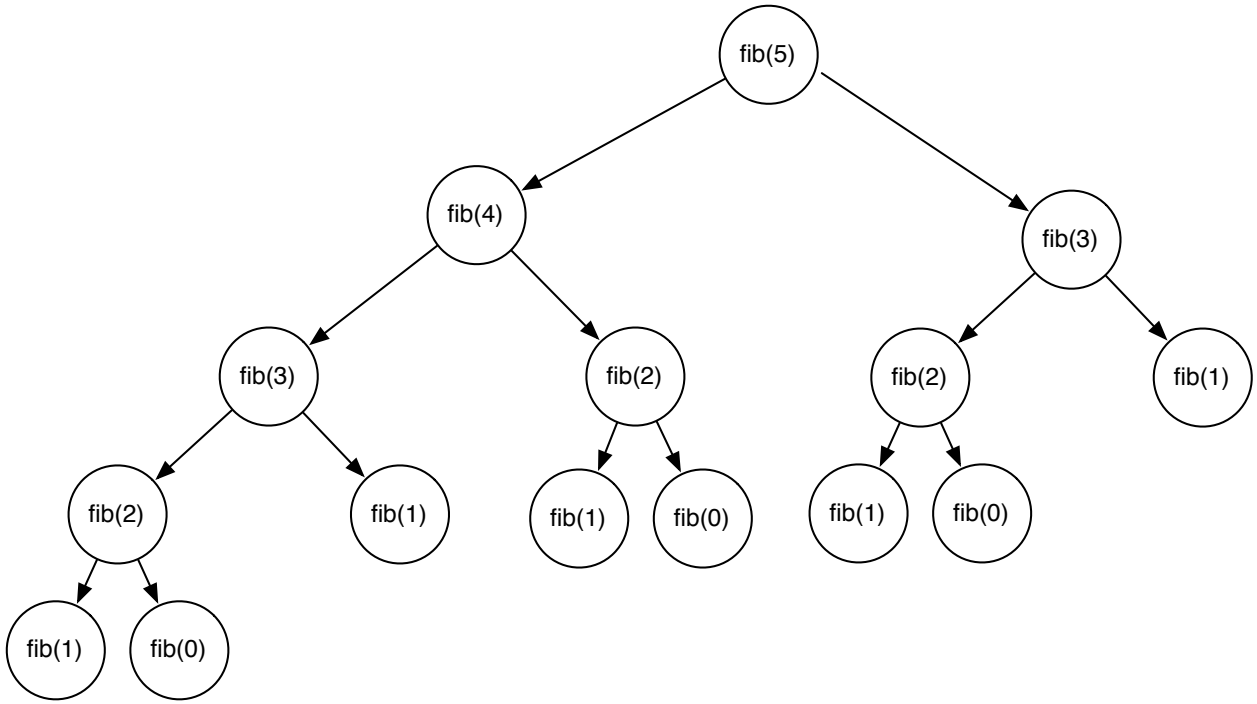
CeNAT

Fibonacci Execution



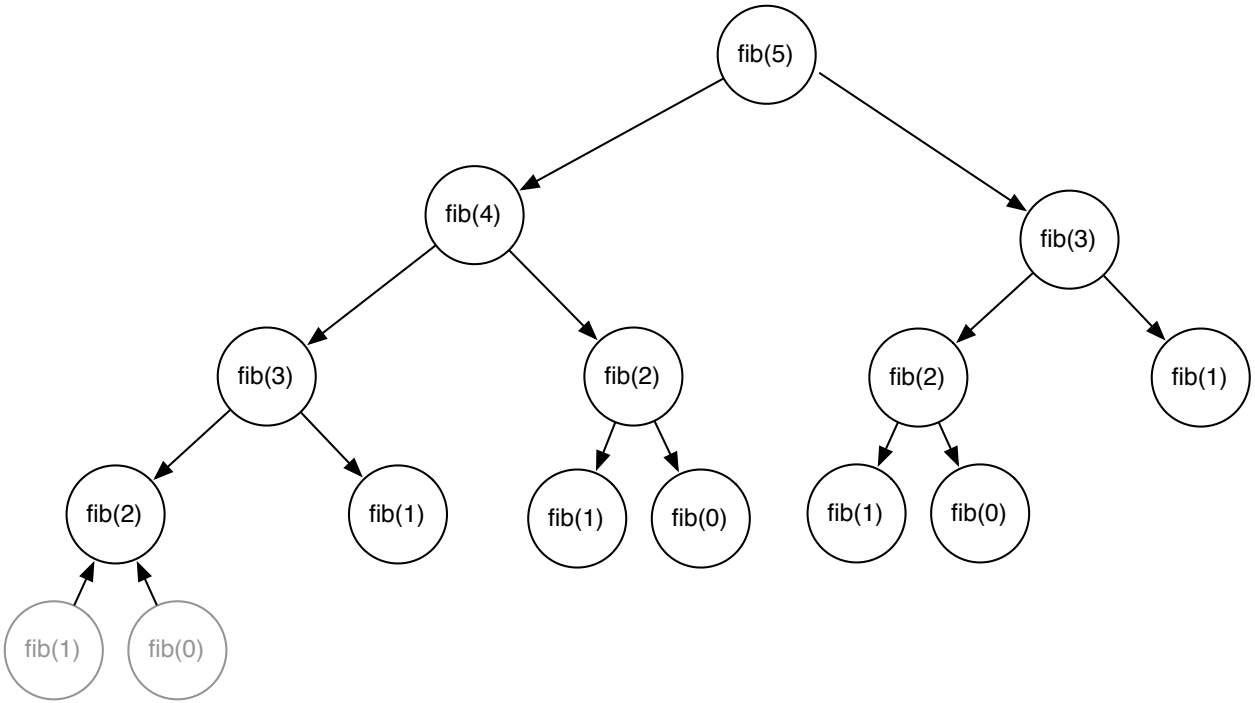
CeNAT

Fibonacci Execution



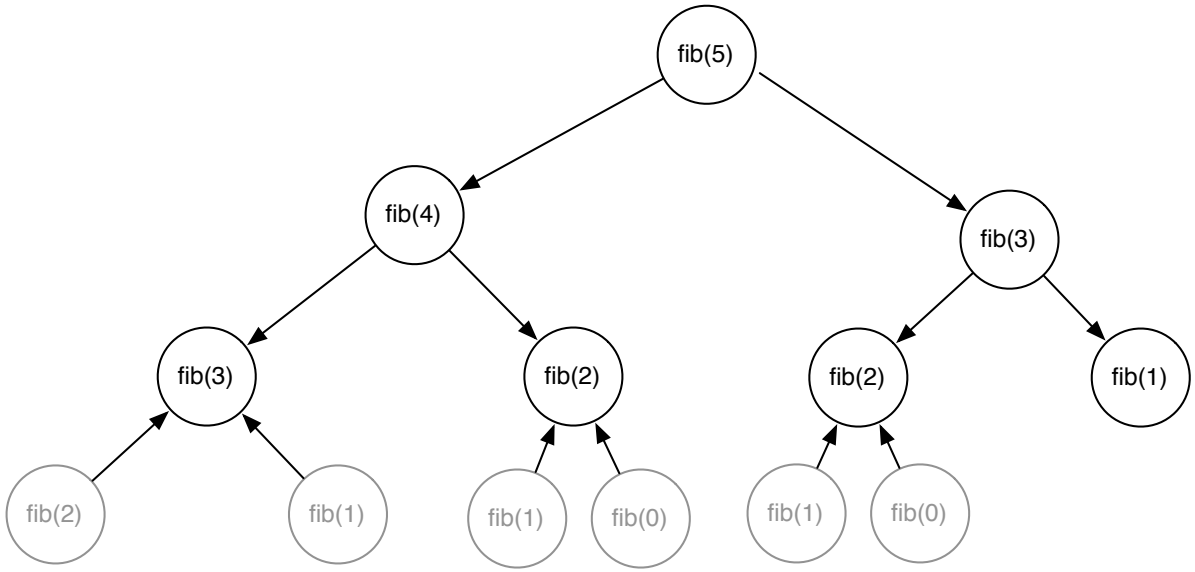
CeNAT

Fibonacci Execution



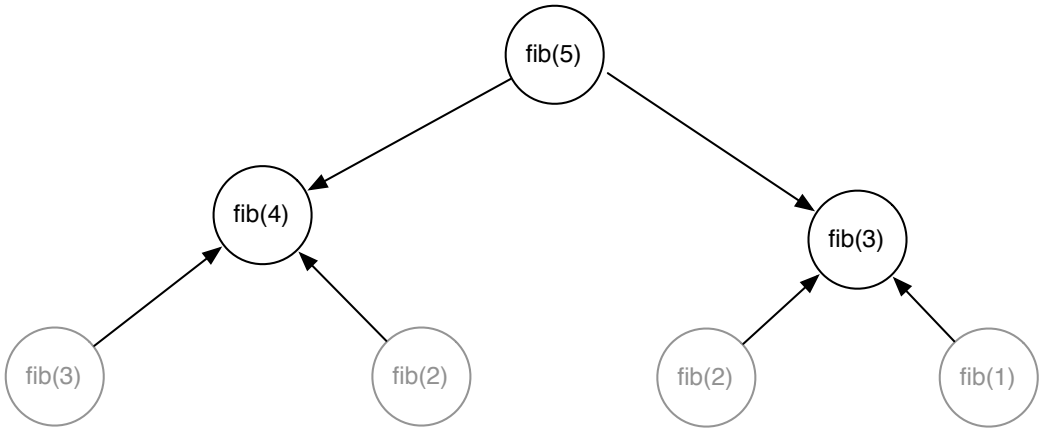
CeNAT

Fibonacci Execution



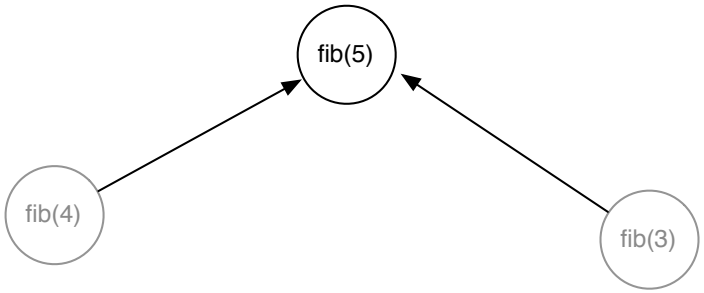
CeNAT

Fibonacci Execution



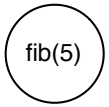
CeNAT

Fibonacci Execution



CeNAT

Fibonacci Execution



Object-based Over-decomposition

More objects than processing entities

- Let the programmer decompose computation into objects: work units, data-units, composites
- Let an intelligent runtime system assign objects to processors
  - RTS can change this assignment (mapping) during execution
  - Locality of data references is a critical attribute for performance
  - A parallel object can access only its own data
  - Asynchronous method invocation for accessing other objects data
  - RTS can schedule work whose dependencies have been satisfied

Amdahl's Law and Grainsize

Overcoming limits to scalability with overdecomposition

- Original "law":
  - If a program has $K\%$ sequential section, then speedup is limited to $\frac{100}{K}$.
    - If the rest of the program is parallelized completely
- Grainsize corollary:
  - If any individual piece of work is $> K$ time units, and the sequential program takes T_{seq},
    - Speedup is limited to $\frac{T_{seq}}{K}$
- So:
  - Examine performance data via histograms to find the sizes of remappable work units
  - If some are too big, change the decomposition method to make smaller units

Overdecomposition and Grainsize

Debunking misconceptions

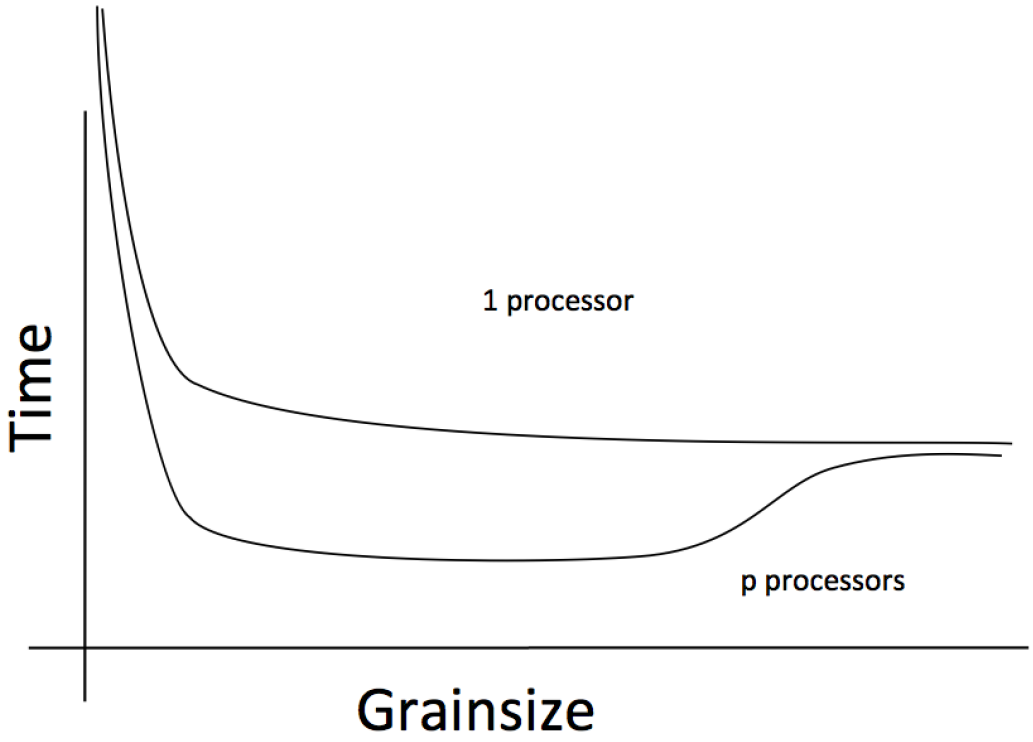
- Common belief:

  overdecomposition is always expensive

- Often times the amount of computation per parallel event (task creation, enqueue/dequeue, messaging, locking, etc.) is small enough to hide overhead
- The benefits of overdecomposition (computation-communication overlap, load balancing, fault tolerance) may offset the associated overhead

Grainsize and Scalability

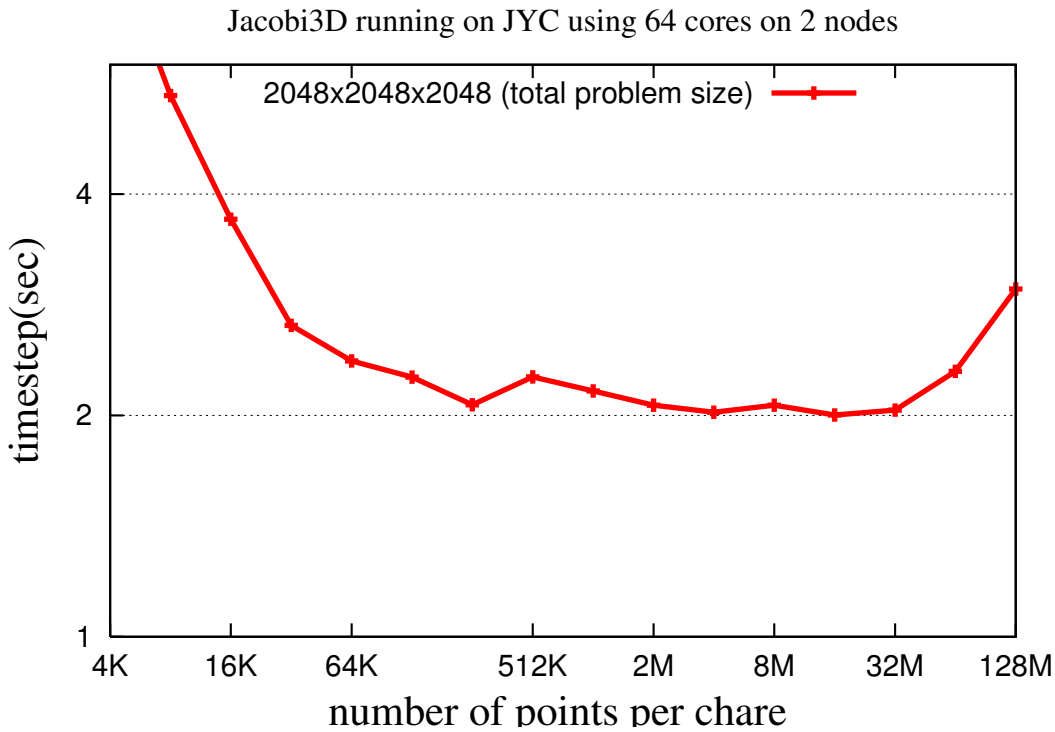
Finding a sweetspot



CeNAT

Example

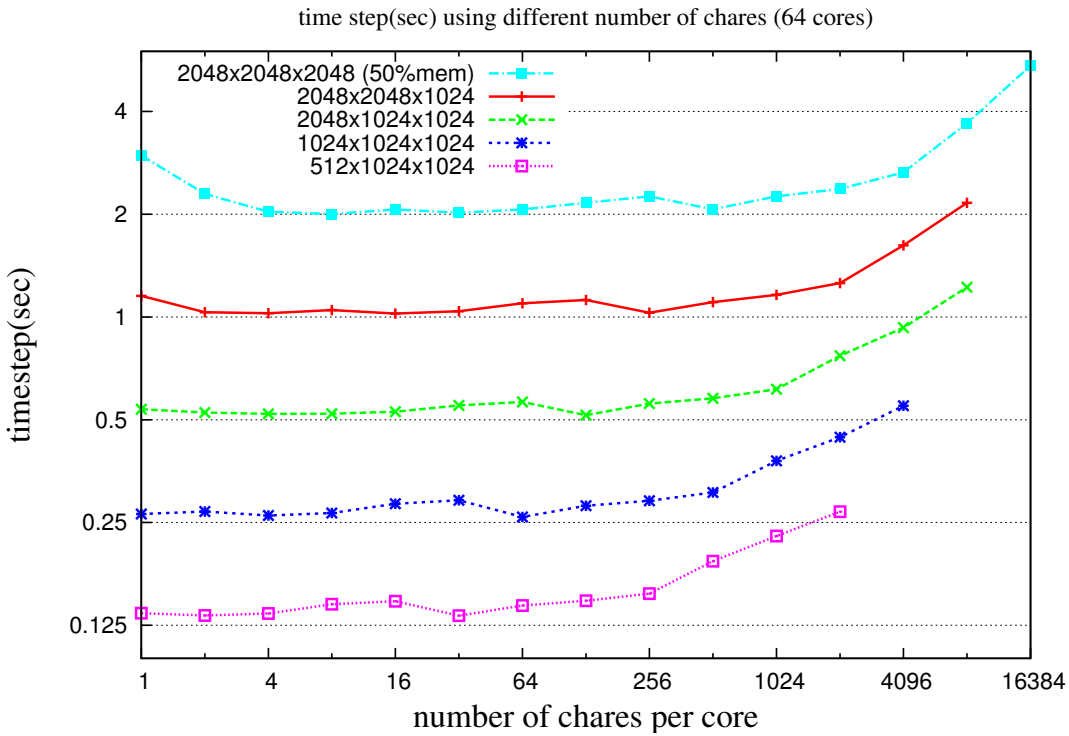
Grainsize study for Jacobi3D



CeNAT

Example

Grainsize study for stencil computation

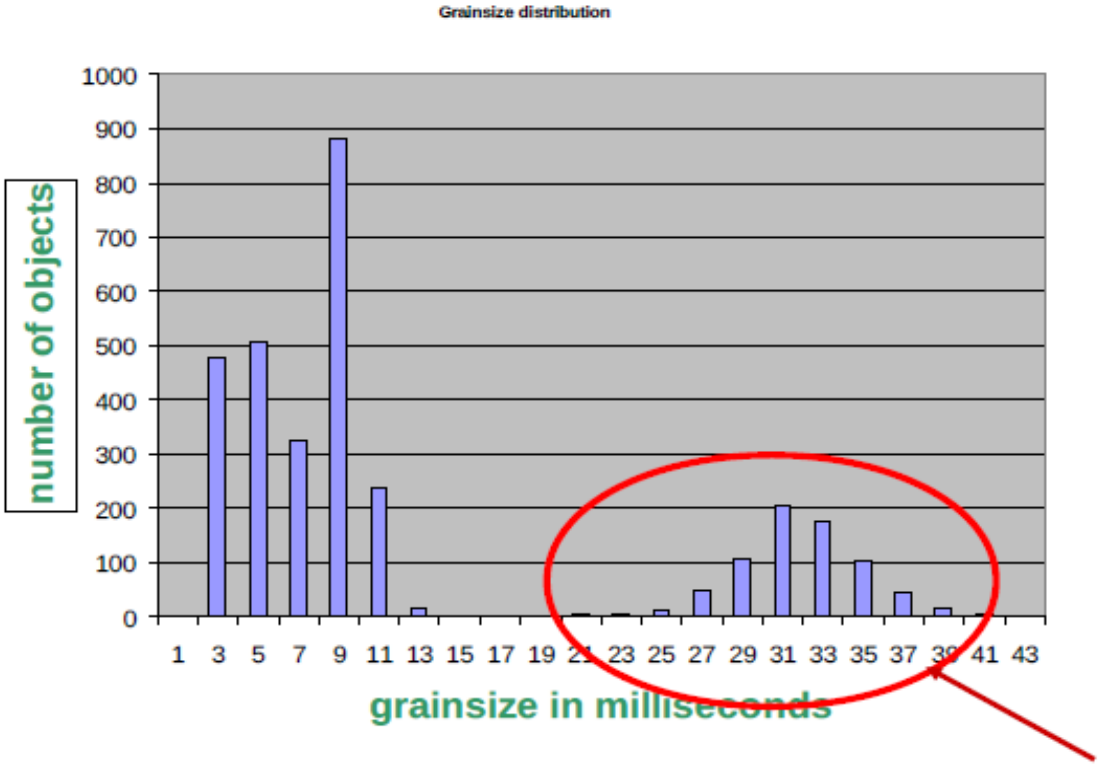


- Typically, having tens of chares per code is adequate (although reasoning should be based on computation per message)

CeNAT

Grainsize and Load Balancing

How much balance is possible?



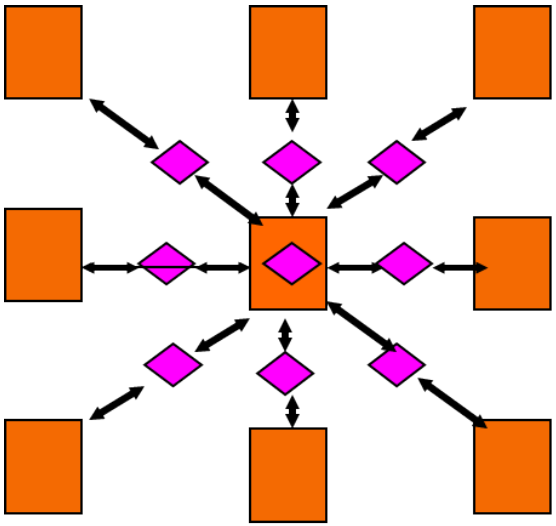
- **Solution**: split compute objects that may have too much work, using a heuristic based on number of interacting entities

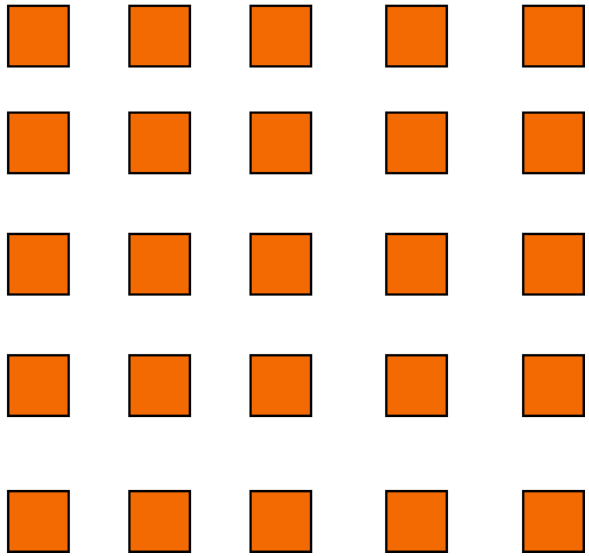
CeNAT

Grainsize for Extreme Scaling

Finer granularity opens up more performance opportunities

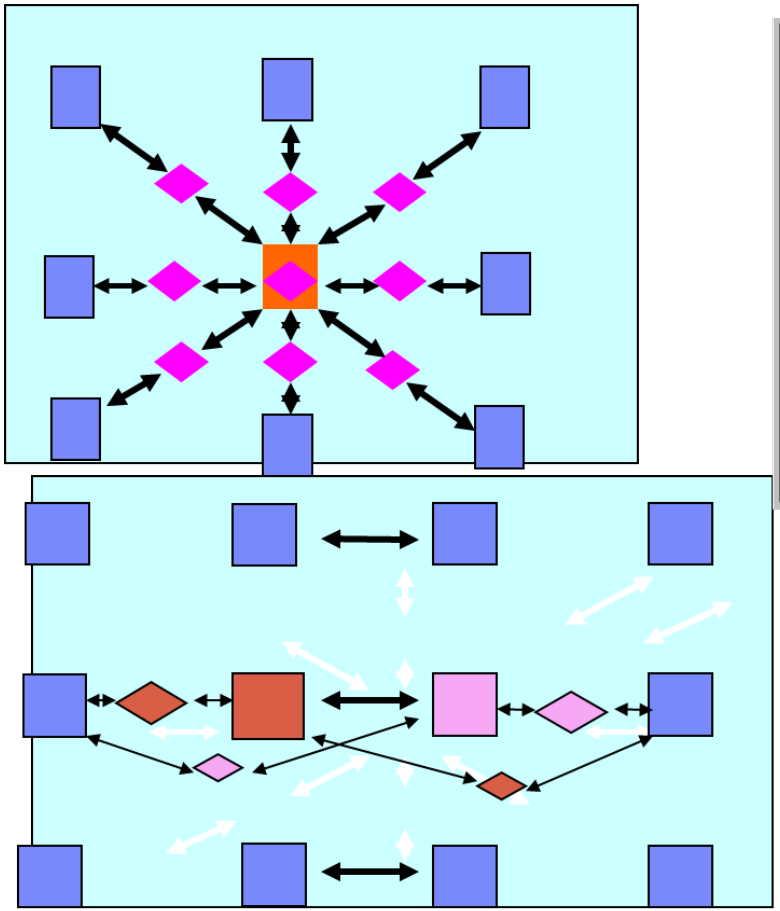
- Strong Scaling is limited by expressed parallelism
  - Minimum iteration time limited by lengthiest computation (largest grains set lower bound)
- Example in particle-interaction code: 1-away generalized to k-away provides fine granularity control

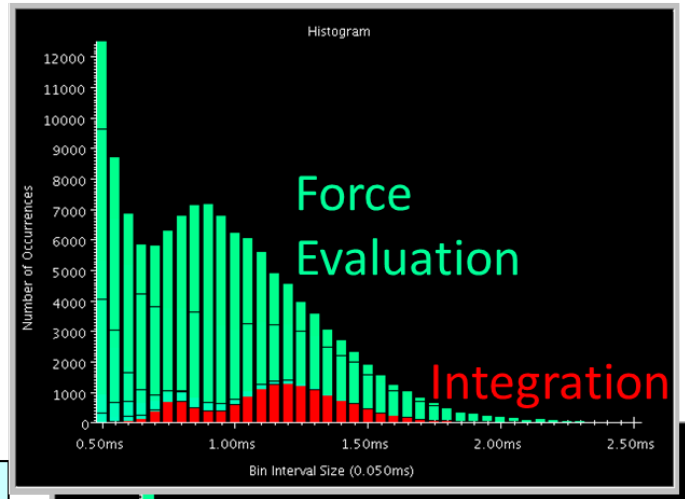


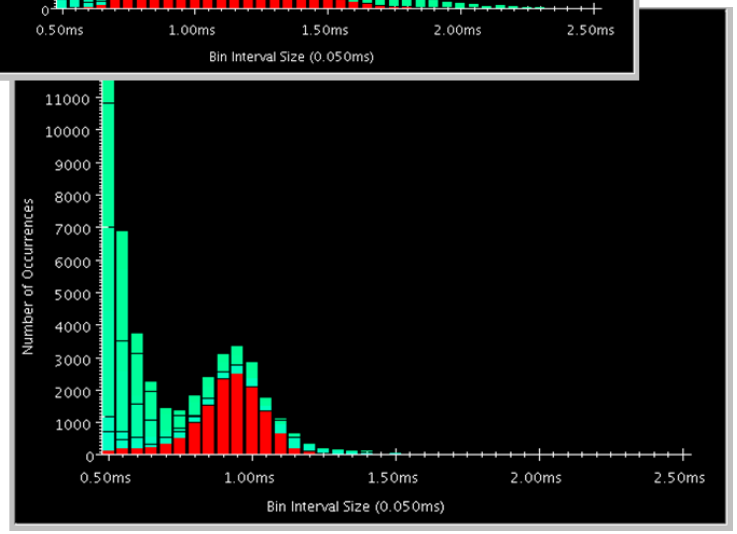


Example

NAMD 2-AwayX







CeNAT

Controlling Grainsize

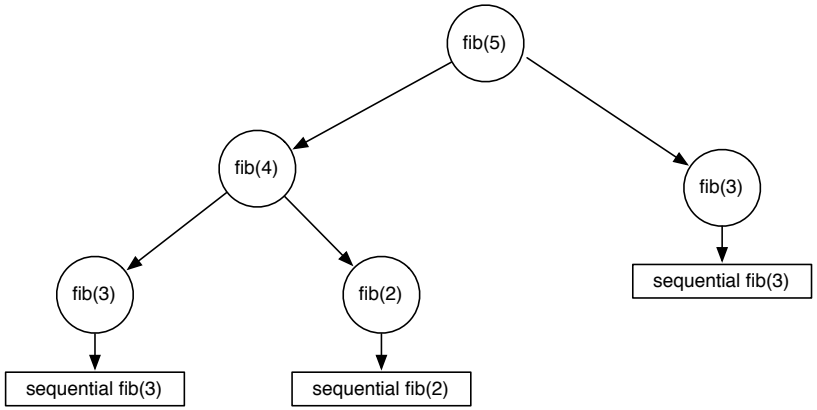
Rules of thumb for grainsize

- Make it as small as possible, as long as it amortizes the overhead
- More specifically, ensure:
  - *Average* grainsize is greater than kv (say $10v$), v is message overhead
  - No single grain should be allowed to be too large
    - Must be smaller than $\frac{T}{p}$, but actually we can be expressed as something smaller than kmv (say $100v$)
- Important corollary:
  - You can be at close to optimal grainsize without having to think about p, the number of processors
- $kv < g < mkv$ $(10v < g < 100v)$

Example

Grain size for Fibonacci

- Set a sequential threshold in the computational tree
  - Past this threshold (i.e. when $n < threshold$), instead of constructing two new chares, compute the fibonacci sequentially



- Setting the grainsize limit at 4 (which is too small, but good for illustration)
- The internal nodes of the tree do very little work, but coarser grains now amortize the cost of the fine-grained chares

CeNAT

Groups

One chare per PE

- Like a chare-array with one chare per PE
- Encapsulate processor local data
- May access the local member as a regular C++ object
- In .ci file,

```
group ExampleGroup {
  // Interface specifications as for normal chares
  // For instance, the constructor ...
  entry ExampleGroup(parameters1);
  // ... and an entry method
  entry void someEntryMethod(parameters2);
};
```

- No difference in .h and .C file definitions

Node Groups

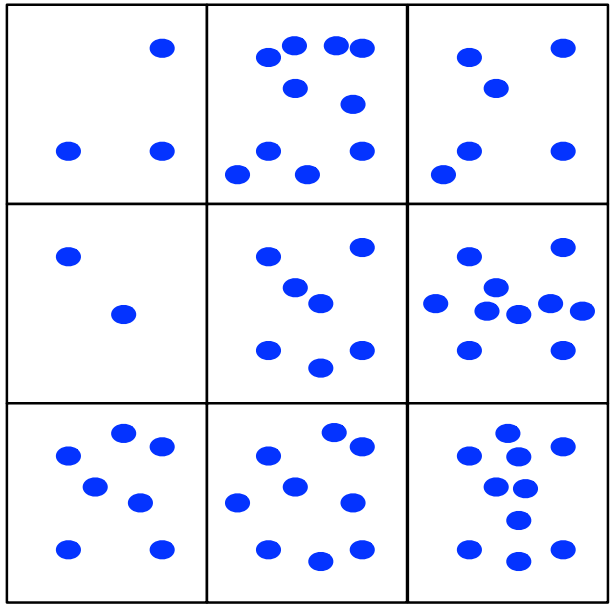
One chare per node

- A chare-array with one chare per node
  - In non-smp node groups and node groups are same
- No difference in .h and .C
- Creation and usage same as others
- An entry method on a node-group member may be executed on any PE of the node
- Concurrent execution of two entry methods of a node-group member may happen
  - Use `[exclusive]` for entry methods which are unsuitable for reentrance safety

Exercise 2.3

Particle interaction

Implement a particle interaction code on a 2D space:



Complete the code in directory:

`code/particle/skeleton`

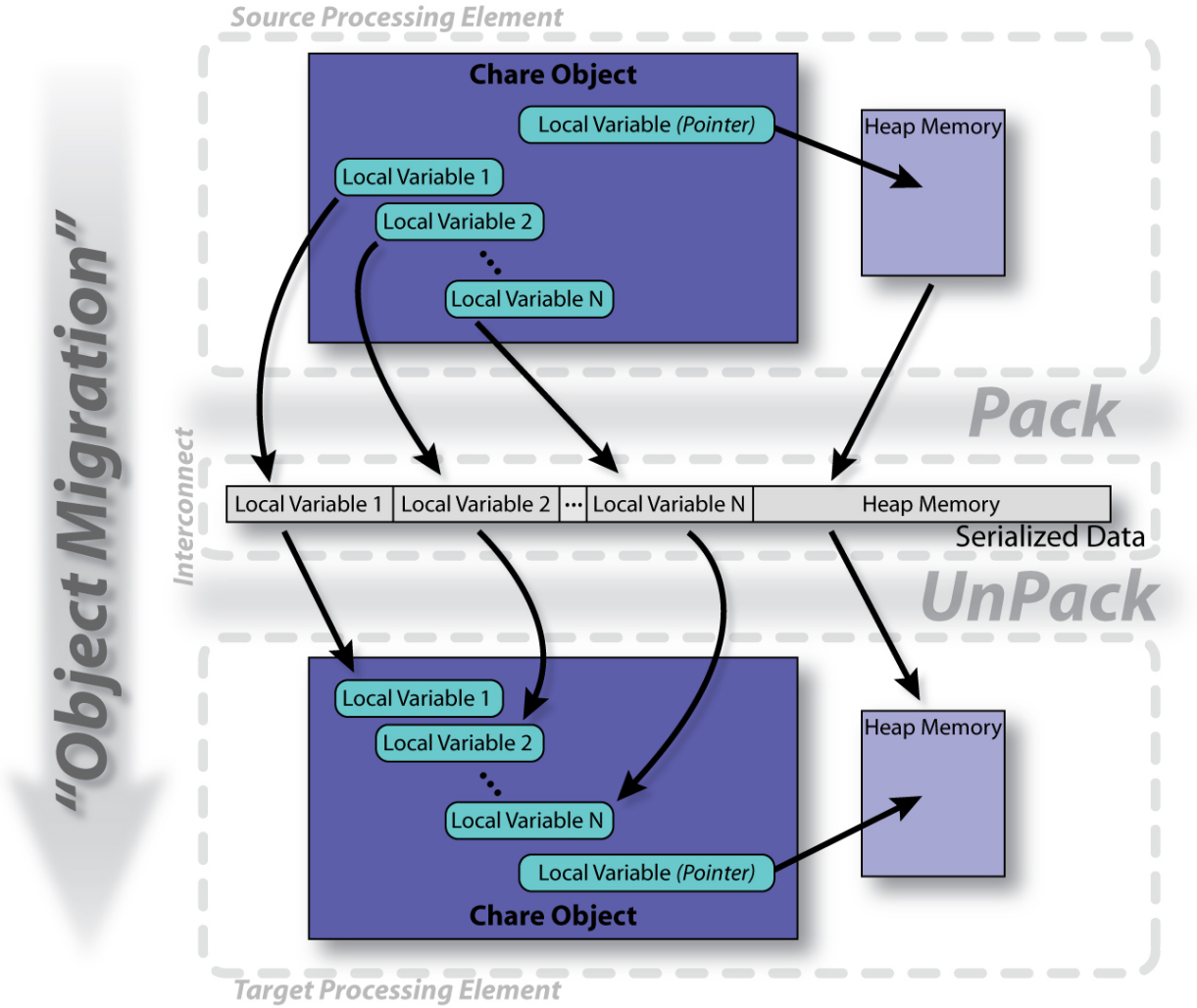
of the class repository:

```
git clone https://github.com/emenesesrojas/parallel_objects.git
```

Do not forget to define variable `CHARMDIR` in the Makefile

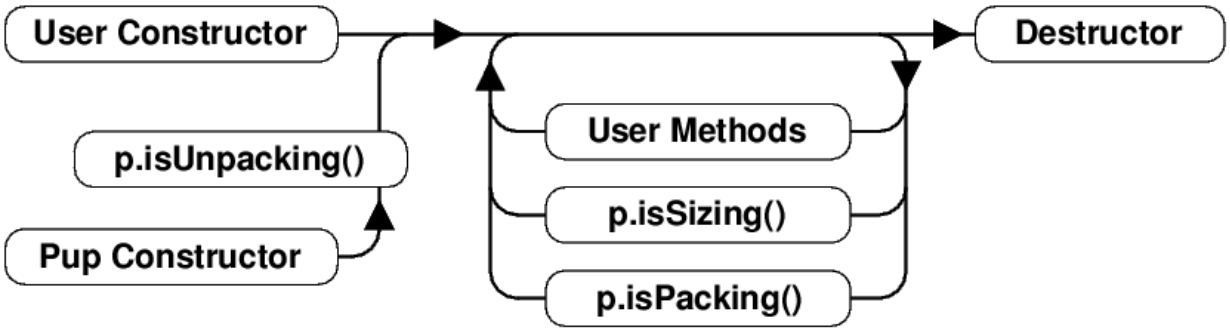
The Pack-unpack (PUP) Process

Base of migratable objects



Packing and Unpacking

PUP usage sequence



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

Object Serialization

Writing a PUP routine

```
class MyChare : public
        CBase_MyChare {
   int a;
   float b;
   char c;
   float localArray[LOCAL_SIZE];
};
```

```
void pup(PUP::er &p) {
    CBase_MyChare::pup(p);
    p | a;
    p | b;
    p | c;
    p(localArray, LOCAL_SIZE);
}
```

Serializing Dynamic Memory

Writing a PUP routine with pointers

```
class MyChare : public
      CBase_MyChare {
  int heapArraySize;
  float* heapArray;
  MyClass *pointer;
};
```

```
void pup(PUP::er &p) {
    CBase_MyChare::pup(p);
    p | heapArraySize;
    if (p.isUnpacking()) {
      heapArray = new float[
            heapArraySize];
    }
    p(heapArray, heapArraySize);
    bool isNull = !pointer;
    p | isNull;
    if (!isNull) {
      if (p.isUnpacking()) pointer
            = new MyClass();
      p | *pointer;
    }
  }
}
```

PUP Framework

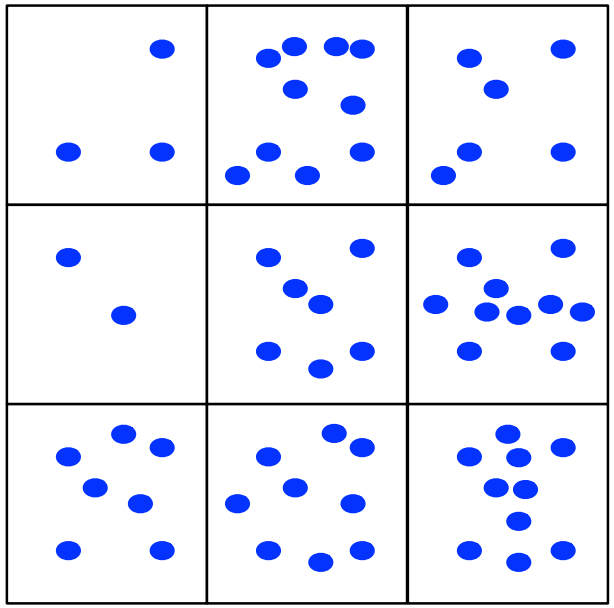
Concerns

- 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 variables in the same order
- Test PUP routines with +balancer RotateLB

Exercise 2.4

Particle interaction

Implement the PUP methods in the particle interaction code on a 2D space:



Complete the code in directory:

`code/particle/skeleton`

of the class repository:

```
git clone https://github.com/emenesesrojas/parallel_objects.git
```

Do not forget to define variable `CHARMDIR` in the Makefile

Automatic Dynamic Load Balancing

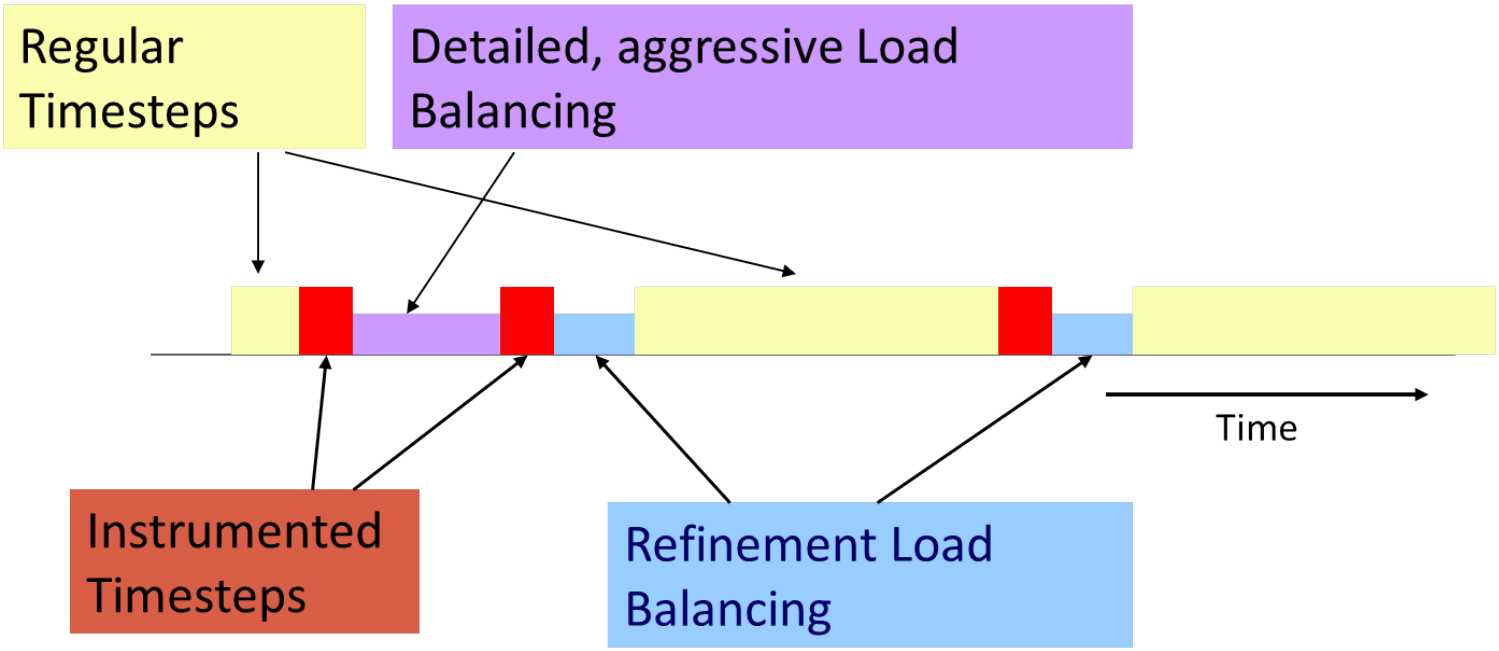
Letting RTS shuffle objects around to get performance

- 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

CeNAT

Load Balancing Protocol

Typical load balancing steps



CeNAT

Making Objects Migratable

Code to use load balancing

- Insert `usesAtSync = true;` into object's constructor
- Write PUP method to serialize the state of a chare
- Insert `if (myLBStep) AtSync();` call at natural barrier
- Implement `ResumeFromSync()` to resume execution
  - Typical ResumeFromSync contribute to a reduction

Migrating Objects

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`

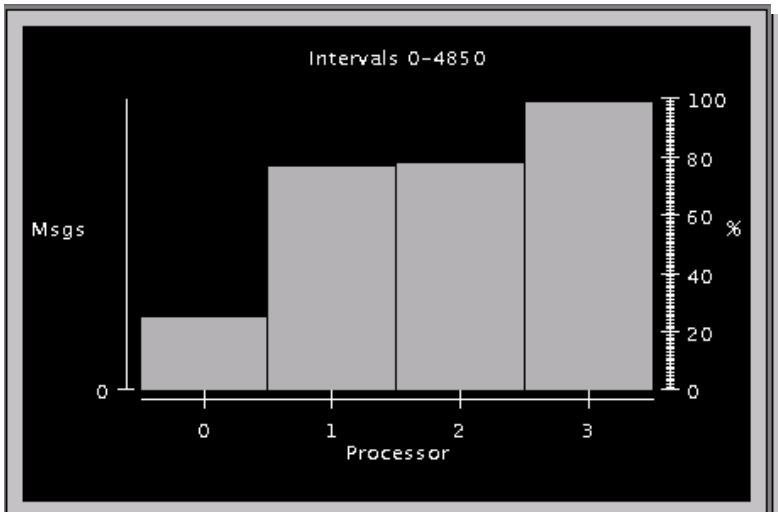
Example

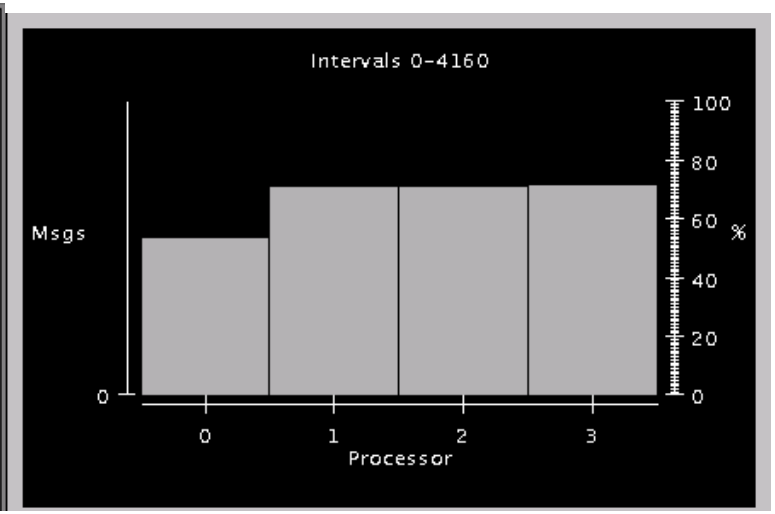
Stencil

```
while (!converged) {
  serial {
    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])
    serial { updateBoundary(d, w, h, b); }
  }
  serial {
    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) { serial { AtSync(); } when ResumeFromSync() {} }
  if (++i % 5 == 0) {
    when checkConverged(bool result) serial {
      if (result) { mainProxy.done(); converged = true; }
    }
  }
}
```

Example

Stencil performance





Detecting Bottlenecks

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

Reduce max load, not dispersion in load

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 $= max_i$(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.

Dynamic Load Balancing Scenarios

Particle interaction and mesh refinement

- Examples representing typical classes of situations
  - Particles distributed over simulation space
    - Dynamic: because particles move
    - Cases:
      Highly non-uniform distribution (cosmology)
      Relatively uniform distribution
- Structured grids, with dynamic refinements/coarsening
- Unstructured grids with dynamic refinements/coarsening

Load Balancing Strategies

Algorithms for an NP-hard problem

- 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

Example

Particle-interaction code

Orthogonal Recursive Bisection (ORB)

- At each stage: divide particles equally
- Processor don't 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

Periodic Load Balancing

Introspective runtime system

Centralized strategies:

- Charm RTS collects data (on one processor) about:
  - computational load
  - communication for each pair of objects
- Partition the graph of objects across processors
  - Take communication into account
    - Point-to-point, as well as multicast over a subset of objects
    - 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

Object Partitioning Strategies

Dominating computation component

- 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

Dominating communication component

When communication cost is significant:

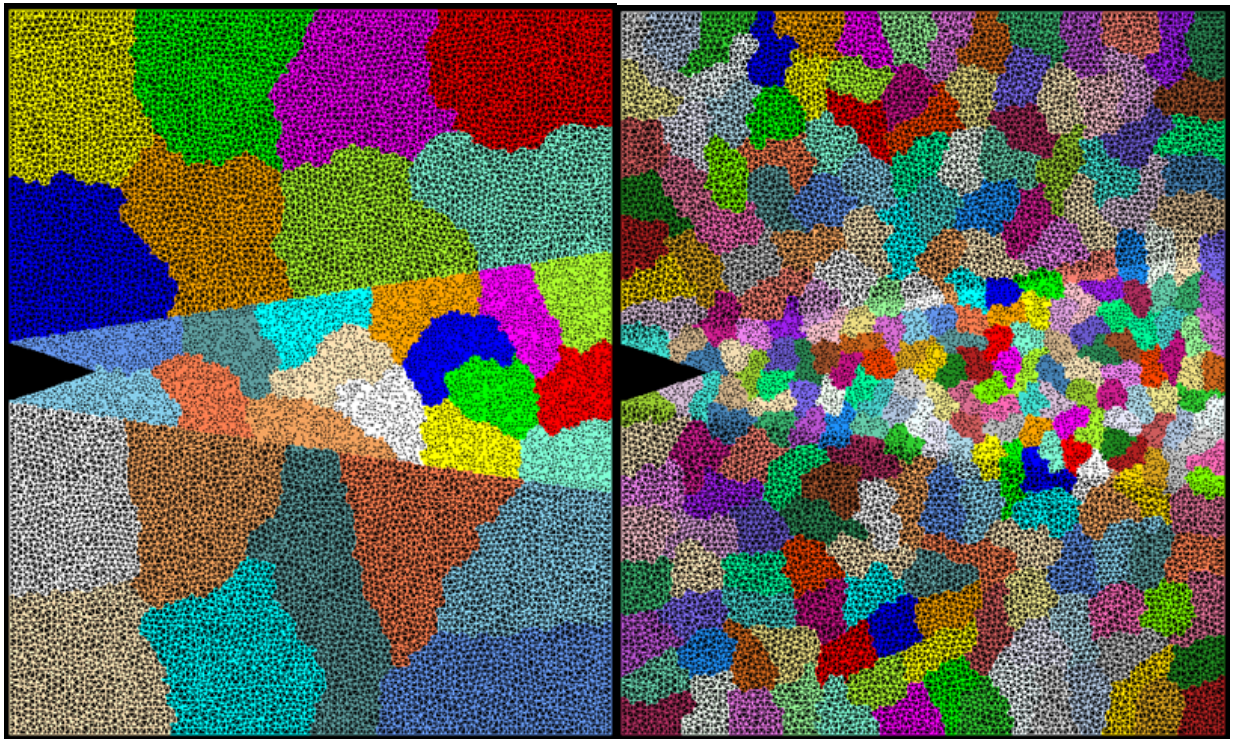
- Still use greedy strategy, but:
  - At each assignment step, choose between assigning O to least loaded processor and the processor that already has objects that communicate most with O.
    - Based on the degree of difference in the two metrics
    - Two-stage assignments:
      In early stages, consider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

Example

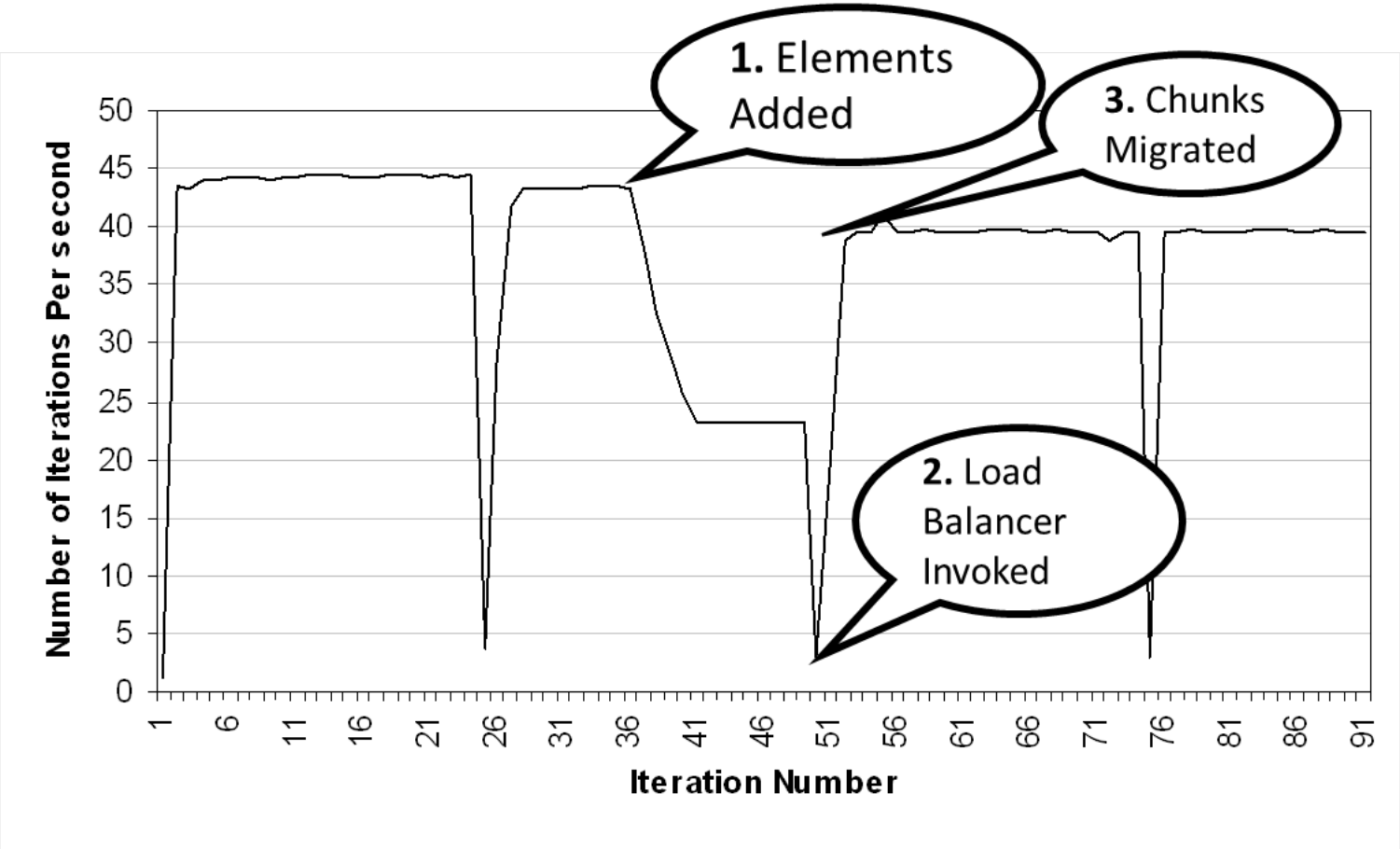
Crack Propagation



Decomposition into 16 chunks (left) and 128 chunks, 8 for each PE (right). The middle area contains cohesive elements. Both decompositions obtained using Metis. As computation progresses, crack propagates, and new elements are added, leading to more complex computations in some chunks.

Load Balancing Example

Crack propagation



CeNAT

Distributed Load Balancing

Avoid bottleneck in information collection

- Centralized strategies:
  - Still fine for 3000 processors for NAMD
- Distributed balancing is needed when:
  - Number of processors is large
  - 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
  - Achieving global effects by local action

Load Balancing on Large Machines

Scaling load balancing

- Centralized load balancing strategies don't 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

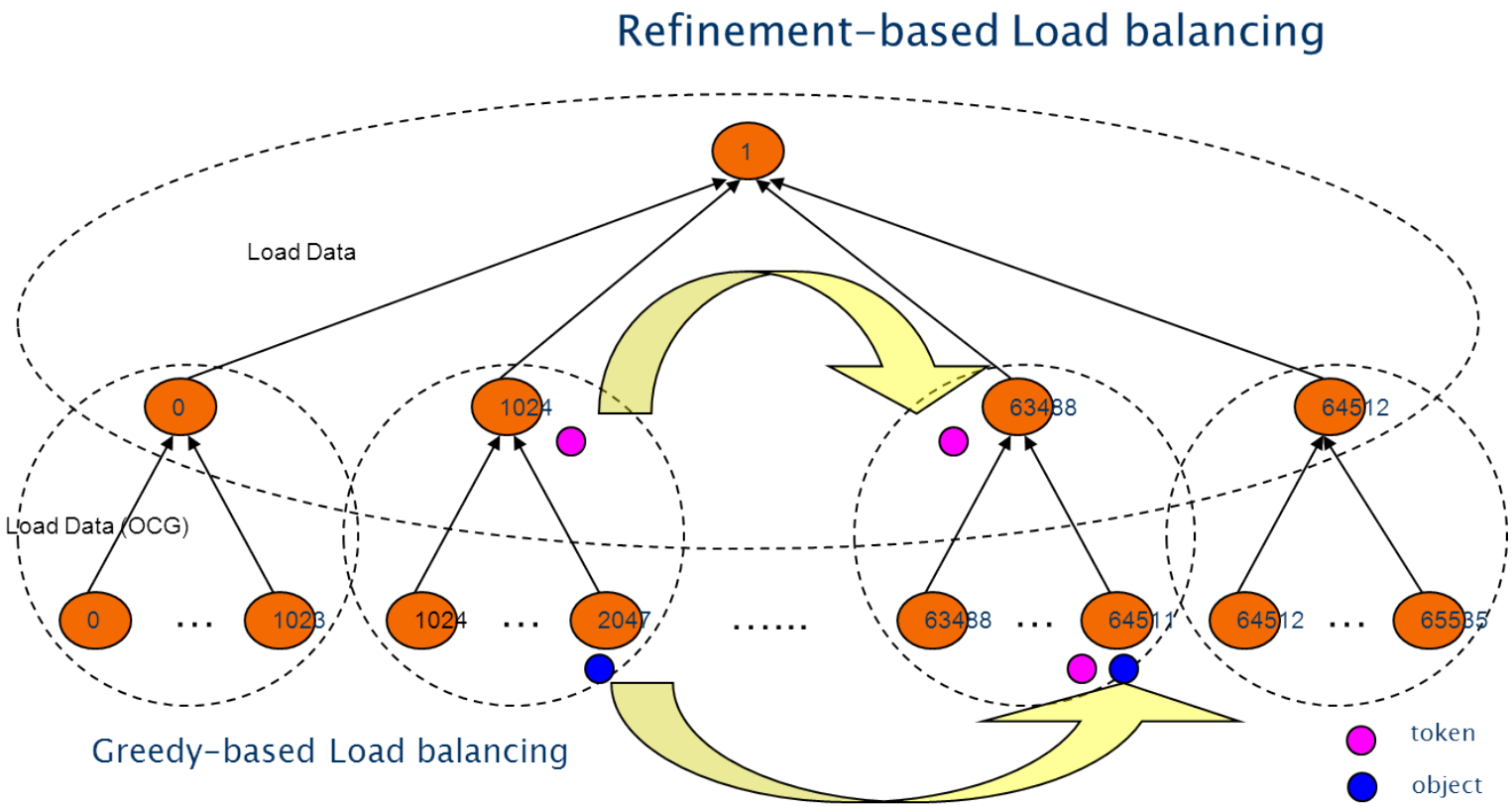
Hierarchical Load Balancers

Multi-level approach

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

A Hybrid Scheme

Mixing load balancing strategies



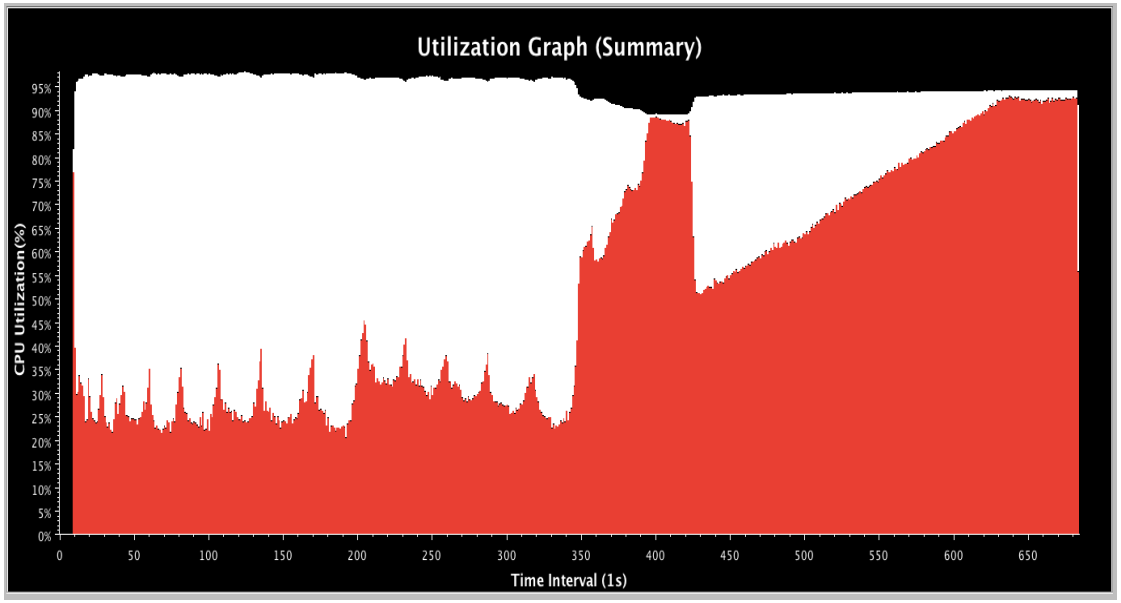
MetaBalancer

When and how to load balance?

- Difficult to find the optimum load balancing period:
  - Depends on the application characteristics
  - Depends on the machine the application is run on
  - Frequent load balancing leads to high overhead and no benefit
  - Infrequent load balancing leads to load imbalance and results in no gains
- Monitors the application continuously and predicts behavior
- Decides when to invoke which load balancer
- Command line argument: `+MetaLB`

Example

Fractography with no load balancing

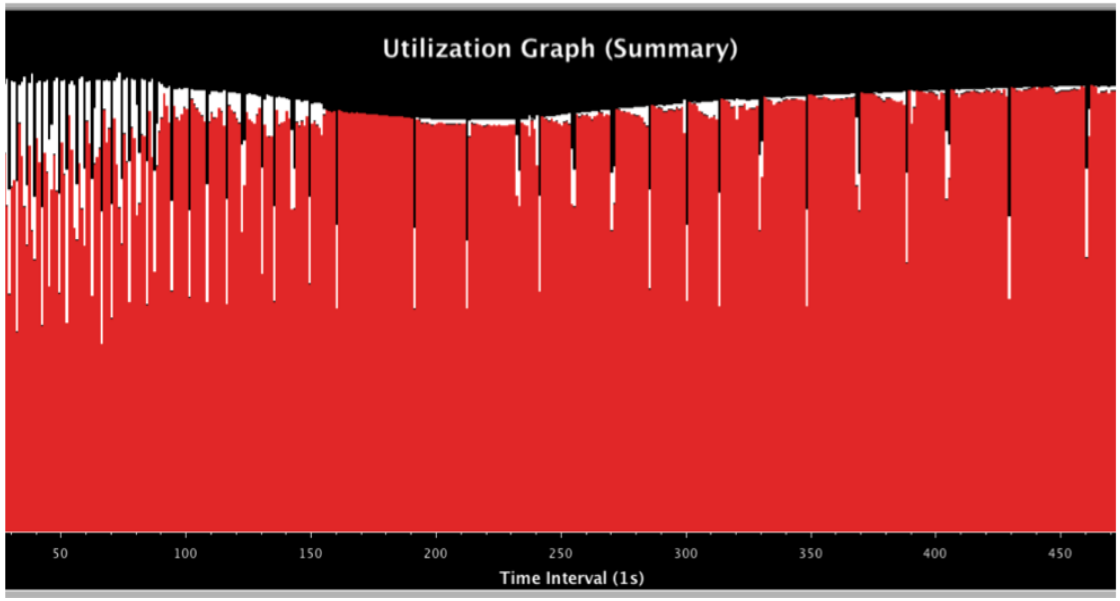


- Large variation in processor utilization
- Low utilization leading to resource wastage

CeNAT

Example

Fractography with Meta-Balancer



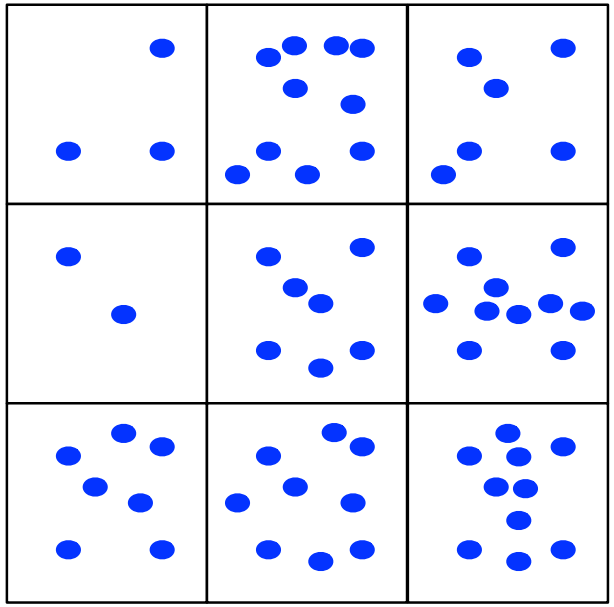
- Identifies the need for frequent load balancing in the beginning
- Frequency of load balancing decreases as load becomes balanced
- Increases overall processor utilization and gives gain of 31%

CeNAT

Exercise 2.5

Particle interaction

Run the particle interaction code on a 2D space:



Use different load balancer with the code in directory:

`code/particle/skeleton`

of the class repository:

```
git clone https://github.com/emenesesrojas/parallel_objects.git
```

Do not forget to define variable `CHARMDIR` in the Makefile

Concluding Remarks

- Migratability empowers load balancing framework
- Periodic load balancer based on principle of persistence
- Several algorithms for solving load imbalance on an HPC system

Thank You!
Q&A