Load Balancing

Gengbin Zheng

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

PARALLEL PROGRAMMING LAB
DEPT. OF COMPUTER SCIENCE, UNIVERSITY OF ILLINOIS

Load Balancing

- Goal: **higher processor utilization**
- Object migration allows us to move the work load among processors easily
- Measurement-based Load Balancing
  - Principle of Persistence
  - Application independent
- Two major approaches to distributing work:
  - Centralized
  - Distributed

Migration

- Array objects can **migrate** from one processor to another
- Migration creates a new object on the destination processor while destroying the original
- Need a way of **packing** an object into a message, then **unpacking** it on the receiving processor

PUP framework

- PUP is a framework for packing and unpacking migratable objects into messages
- To migrate, must implement pack/unpack or *pup* method
- Pup method combines 3 functions
  - Data structure traversal : compute message size, in bytes
  - Pack : write object into message
  - Unpack : read object out of message

Writing a PUP Method

```
Class ShowPup {
   double a;      int x;
   char y;    unsigned long z;
   float q[3];    int *r; // heap allocated memory
public:
    void pup(PUP::er &p) {
      if (p.isUnpacking())
        r = new int[ARRAY_SIZE];
      p | a; p |x; p|y       // you can use | operator
      p(z); p(q, 3)          // or ()
      p(r,ARRAY_SIZE);
    }
};
```

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.

The Principle of Persistence

- Big Idea: the past predicts the future
- Patterns of communication and computation remain nearly constant

- By measuring these patterns we can improve our load balancing techniques

Centralized Load Balancing

- Uses information about activity on all processors to make load balancing decisions
- Advantage: **Global information** gives higher quality balancing
- Disadvantage: Higher **communication costs** and **latency**
- Algorithms: Greedy, Refine, Recursive Bisection (ORB), Metis

Neighborhood Load Balancing

- Load balances among a small set of processors (the neighborhood)
- Advantage: Lower communication costs
- Disadvantage: Could leave a system which is poorly balanced globally

- Algorithms: NeighborLB, WorkstationLB

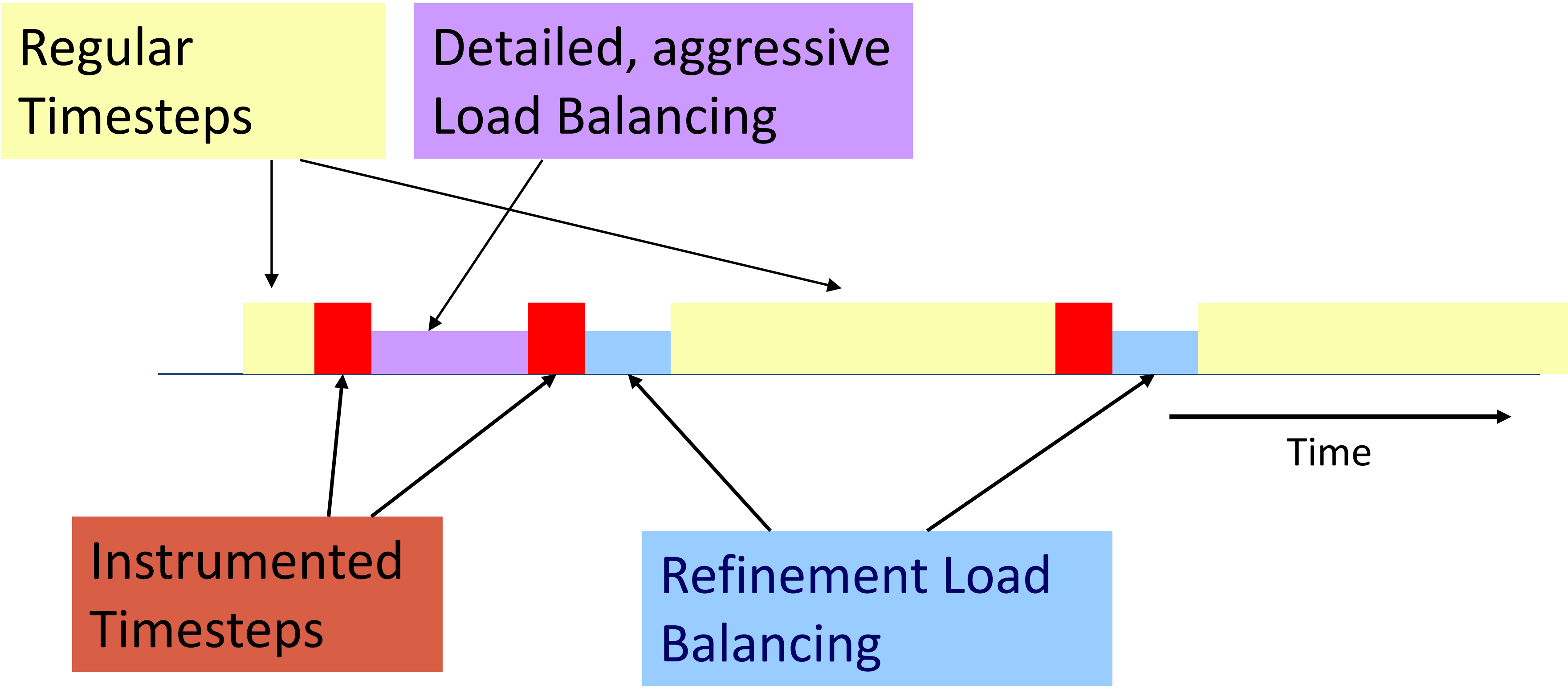
When to Re-balance Load?

Default: Load balancer will migrate periodically

- Programmer Control: **AtSync** load balancing

  AtSync method: enable load balancing at specific point
  - Object ready to migrate
  - Re-balance if needed
  - ***AtSync()*** called when your chare is **ready** to be load balanced
    - load balancing may not start right away
  - ***ResumeFromSync()*** called when load balancing for this chare has **finished**

Load Balancing Steps



Using a 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***

Load Balancing in Jacobi2D

```
Main:
   Setup worker array, pass data to them

Workers:
   Start looping

   Send messages to all neighbors with ghost rows

   Wait for all neighbors to send ghost rows to me

   Once they arrive, do the regular Jacobi relaxation

   Calculate maximum error, do a reduction to compute
     global maximum error

   If timestep is a multiple of 64, load balance the
     computation. Then restart the loop.
```

Example is at: charm/tests/charm++/loadbalancing/jacobi2D

Load Balancing in Jacobi2D (cont.)

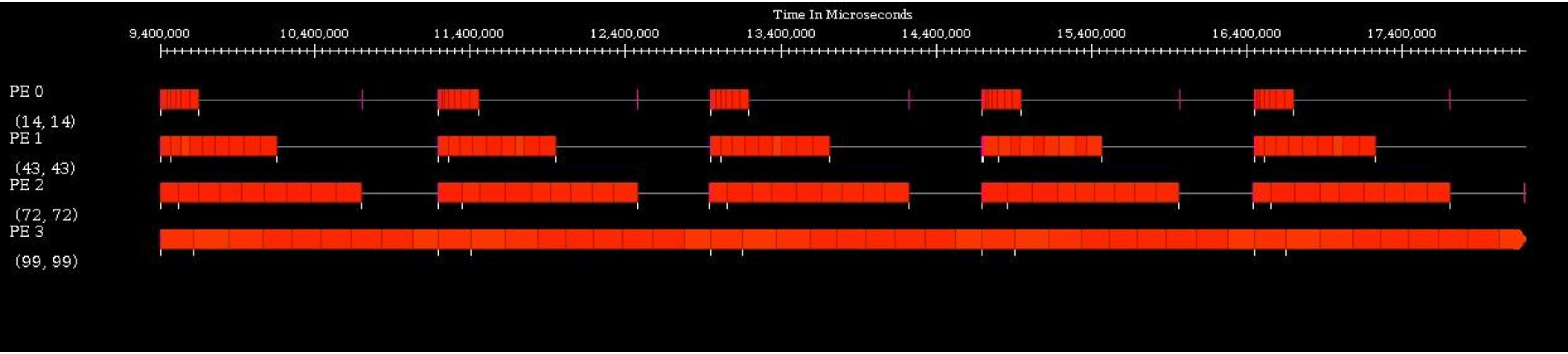
```
JacobiChunk::JacobiChunk(void) {
      //Initialize other parameters
      usesAtSync=CmiTrue;
```

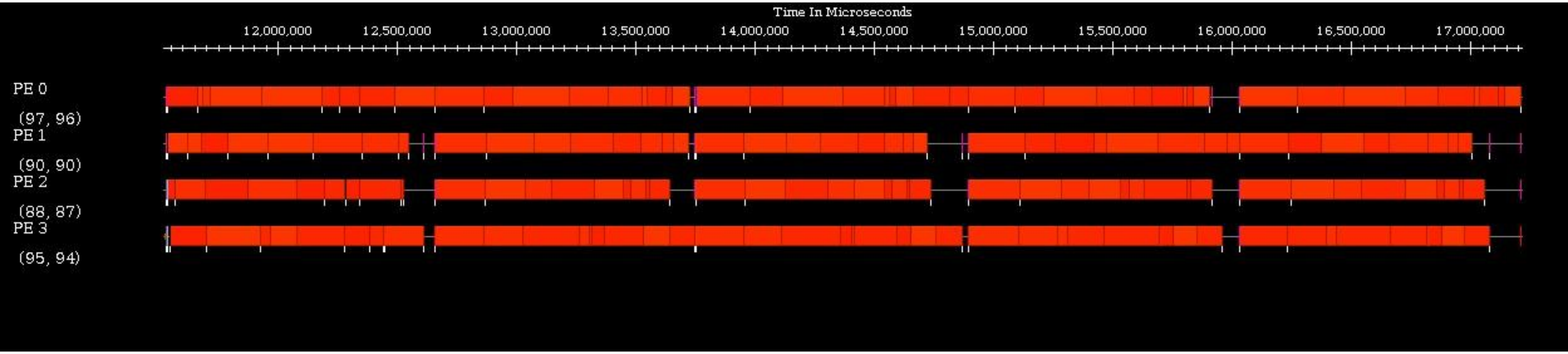
```
Void JacobiChunk::refine(void){
  // do all the jacobi computation
   ....
  cont
}
```

```
Void JacobiChunk::refine(void){
  numIters++;
  if(numIters%10==5)    AtSync();
  else    thisProxy.startNextIter();
}
```

```
void JacobiChunk::ResumeFromSync(void){
 startNextIter();
}
```

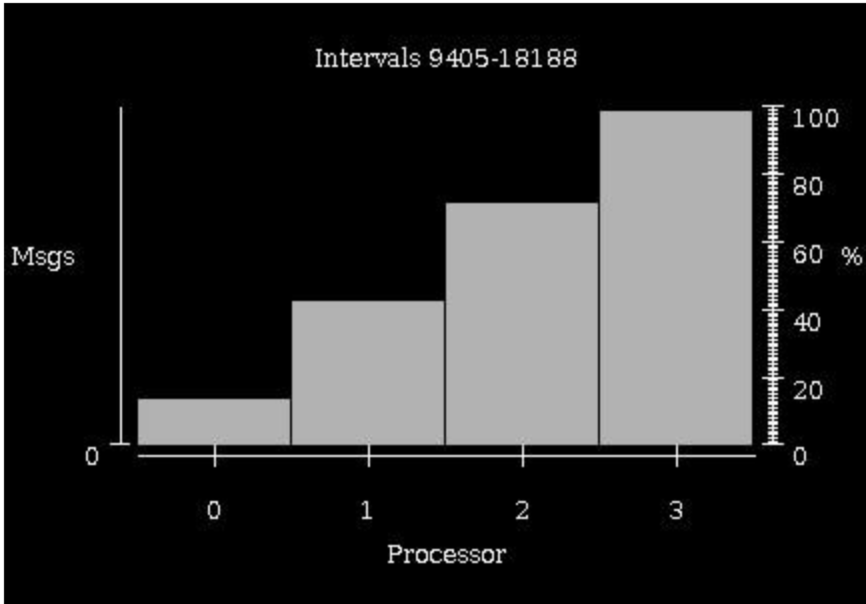
Timelines: Before and After Load Balancing

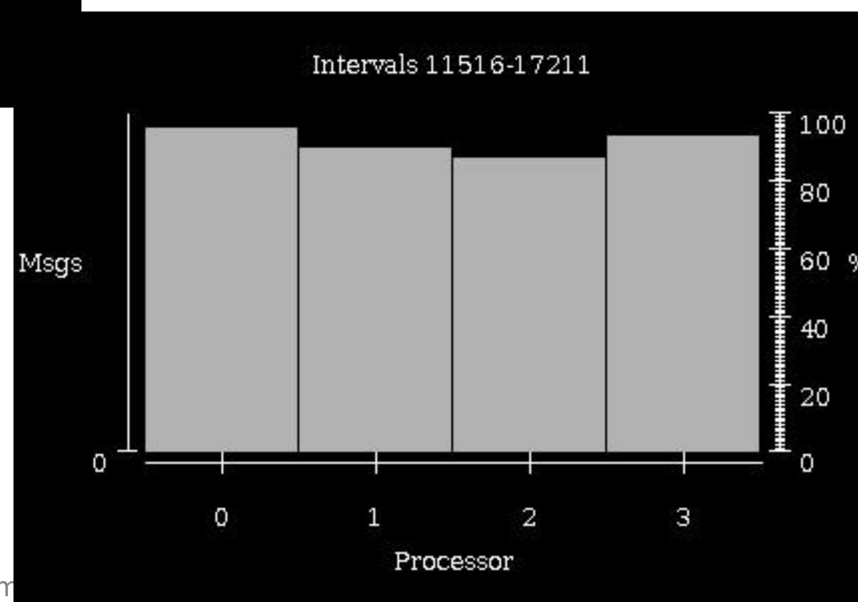




6x6 chunks running on 4 processors
Each chunk is a 64x64 array
Artificial load imbalance

Processor Utilization: After Load Balance





6x6 chunks running on 4 processors
Each chunk is a 64x64 array
Artificial load imbalance

AMPI

- Same idea, with MPI extention:
  - MPI_Migrate()
- Migrate stack data:
  - Isomalloc
- Migrate heap data
  - Isomalloc, or
  - MPI_Register(void *, MPI_PupFn)

- Example at: charm/examples/ampi/Cjacobi3D