Charm++ Tutorial

Presented by:

Laxmikant V. Kale

Kumaresh Pattabiraman

Chee Wai Lee

PARALLEL PROGRAMMING LABORATORY

Overview

- Introduction
  - Developing parallel applications
  - Virtualization
  - Message Driven Execution
- Charm++ Features
  - Chares and Chare Arrays
  - Parameter Marshalling
  - Examples
- Tools
  - LiveViz
  - Parallel Debugger
  - Projections
- More Charm++ features
  - Structured Dagger Construct
  - Adaptive MPI
  - Load Balancing
- Conclusion

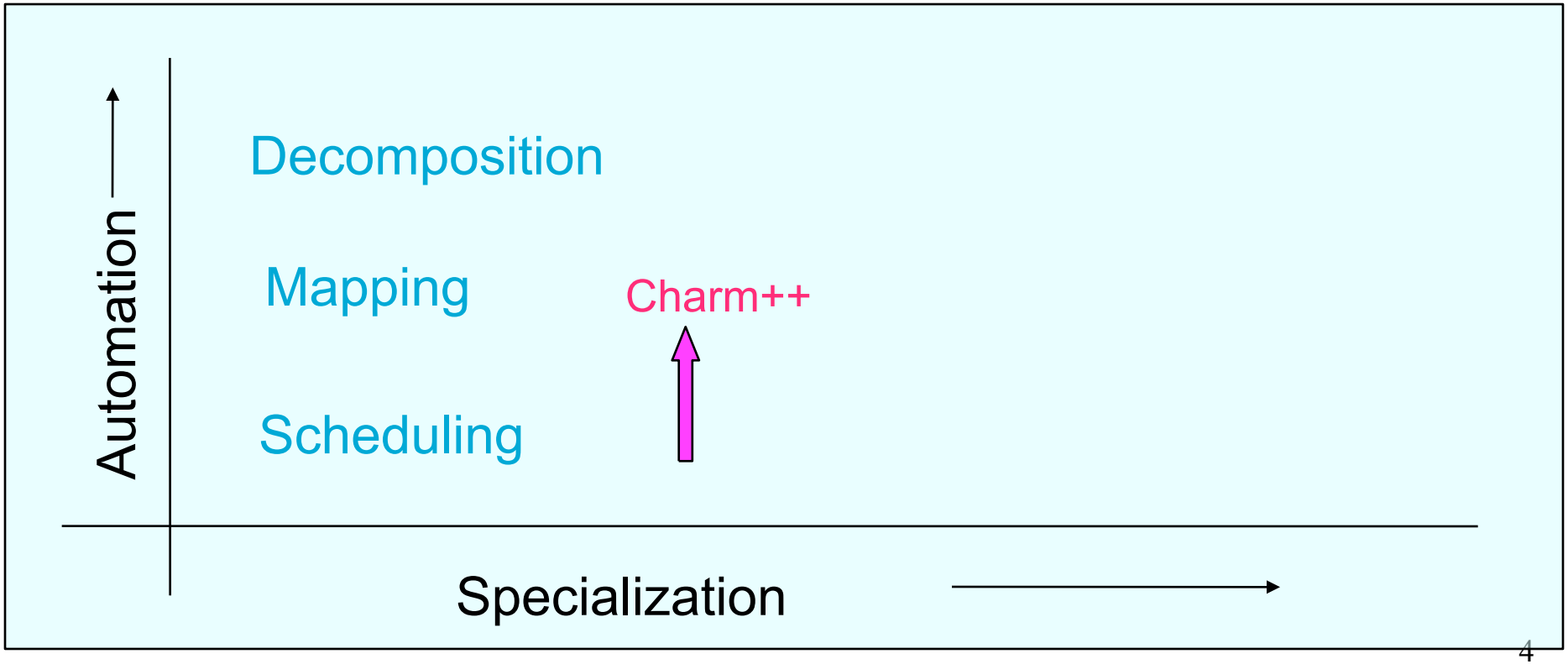
Outline

- Introduction
- Charm++ features
  - Chares and Chare Arrays
  - Parameter Marshalling
  - Examples
- Tools
  - LiveViz
  - Parallel Debugger
  - Projections
- More Charm++ Features
  - Structured Dagger Construct
  - Adaptive MPI
  - Load Balancing
- Conclusion

Developing a Parallel Application

Seek optimal division of labor between “system” and programmer

Decomposition done by programmer, everything else automated

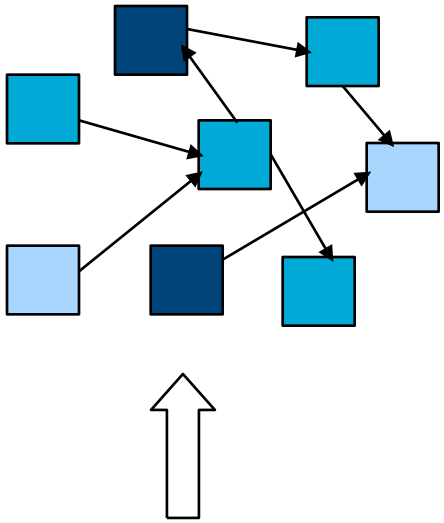


Virtualization: Object-based Decomposition

- Divide the computation into a large number of pieces
  - Independent of number of processors
  - Typically larger than number of processors
- Let the system map objects to processors

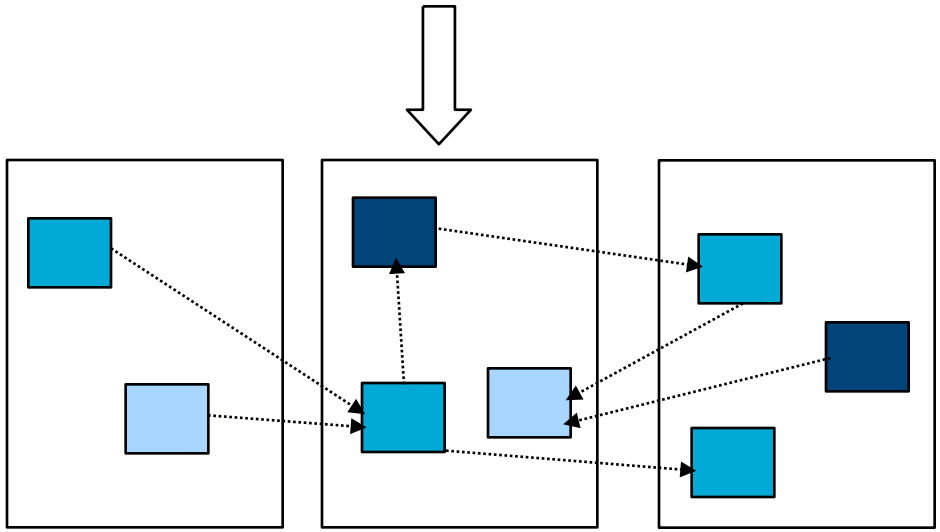
Object-based Parallelization

User is only concerned with interaction between objects



User View

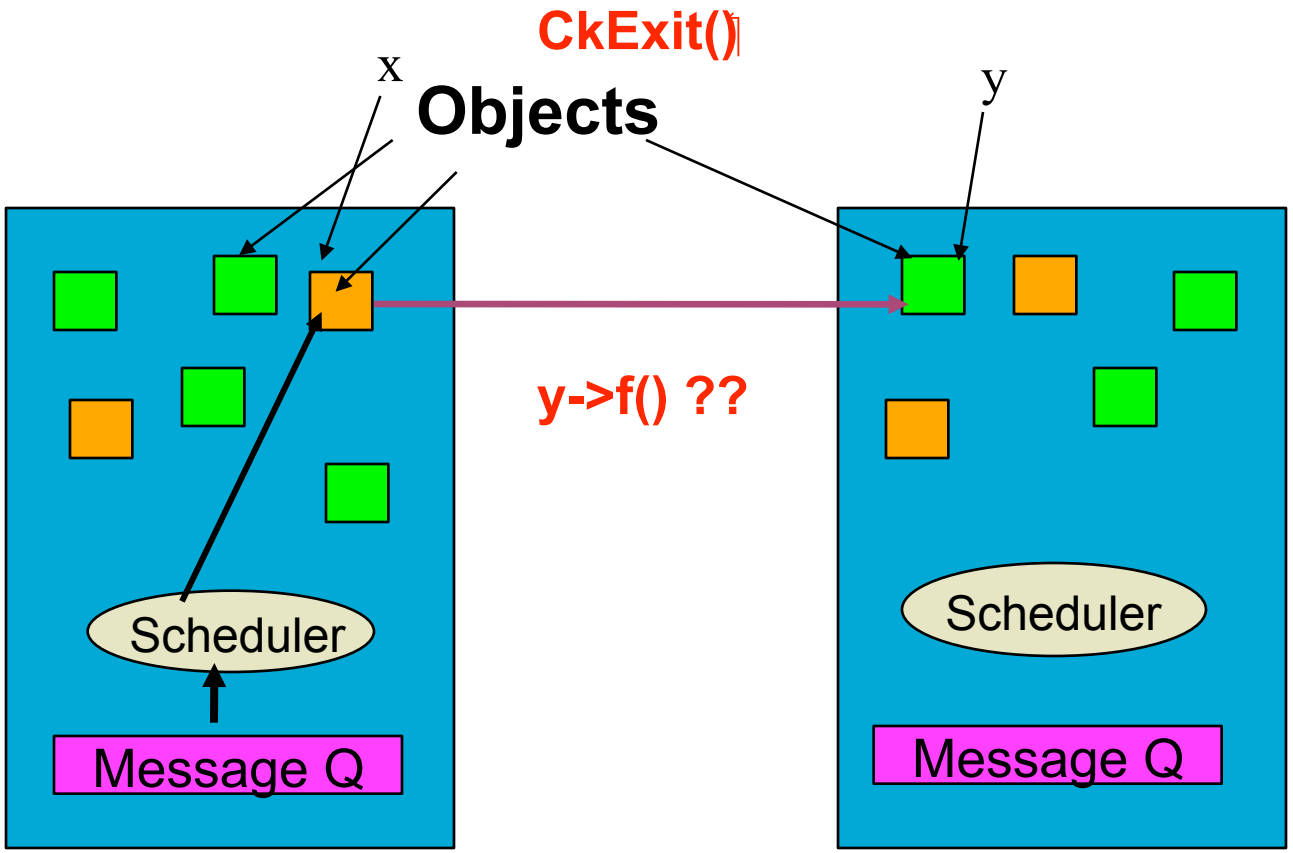
System implementation



Message-Driven Execution

- Objects communicate asynchronously through remote method invocation
- Encourages non-deterministic execution
- Benefits:
  - Communication latency tolerance
  - Logical structure for scheduling

Message-Driven Execution in Charm++



Other Charm++ Characteristics

- Methods execute one at a time
- No need for locks
- Expressing flow of control may be difficult

Outline

- Introduction
- Charm++ features
  - Chares and Chare Arrays
  - Parameter Marshalling
  - Examples
- Tools
  - LiveViz
  - Parallel Debugger
  - Projections
- More Charm++ Features
  - Structured Dagger Construct
  - Adaptive MPI
  - Load Balancing
- Conclusion

Chares – Concurrent Objects

- Can be dynamically created on any available processor
- Can be accessed from remote processors
- Send messages to each other asynchronously
- Contain "entry methods"

“Hello World”

```
// hello.ci

mainmodule hell
  mainchare myma
    entry mymai
  };
};
```

```
// hello.C file

#include “hello.decl.h”
class mymain : public Chare {
public:
      mymain(CkArgMsg *m)
      {
          ckout <<“Hello World”<<endl;
          CkExit();
      }

nclude “hello.def.h”
```

Generates:

hello.decl.h

hello.def.h

Compile and run the program

Compiling

- charmc <options> <source file>
- -o, -g, -language, -module, -tracemode

pgm: pgm.ci pgm.h pgm.C

Example Nodelist File:

group main ++shell ssh

host Host1

host Host2

rm++

To run a CH ... pgm" on four processors, type:

charmrun pgm +p4 <params>

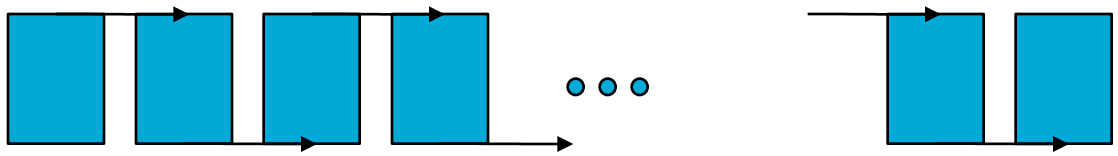
Nodelist file (for network architecture)

- list of machines to run the program
- host <hostname> <qualifiers>

Charm++ solution: Proxy classes

- Proxy class generated for each chare class
  - For instance, CProxy_Y is the proxy class generated for chare class Y.
  - Proxy objects know where the real object is
  - Methods invoked on this object simply put the data in an “envelope” and send it out to the destination
- Given a proxy p, you can invoke methods
  - p.method(msg);

Chare Arrays



- Array of Objects of the same kind
- Each one communicates with the next one
- Individual chares – cumbersome and not practical

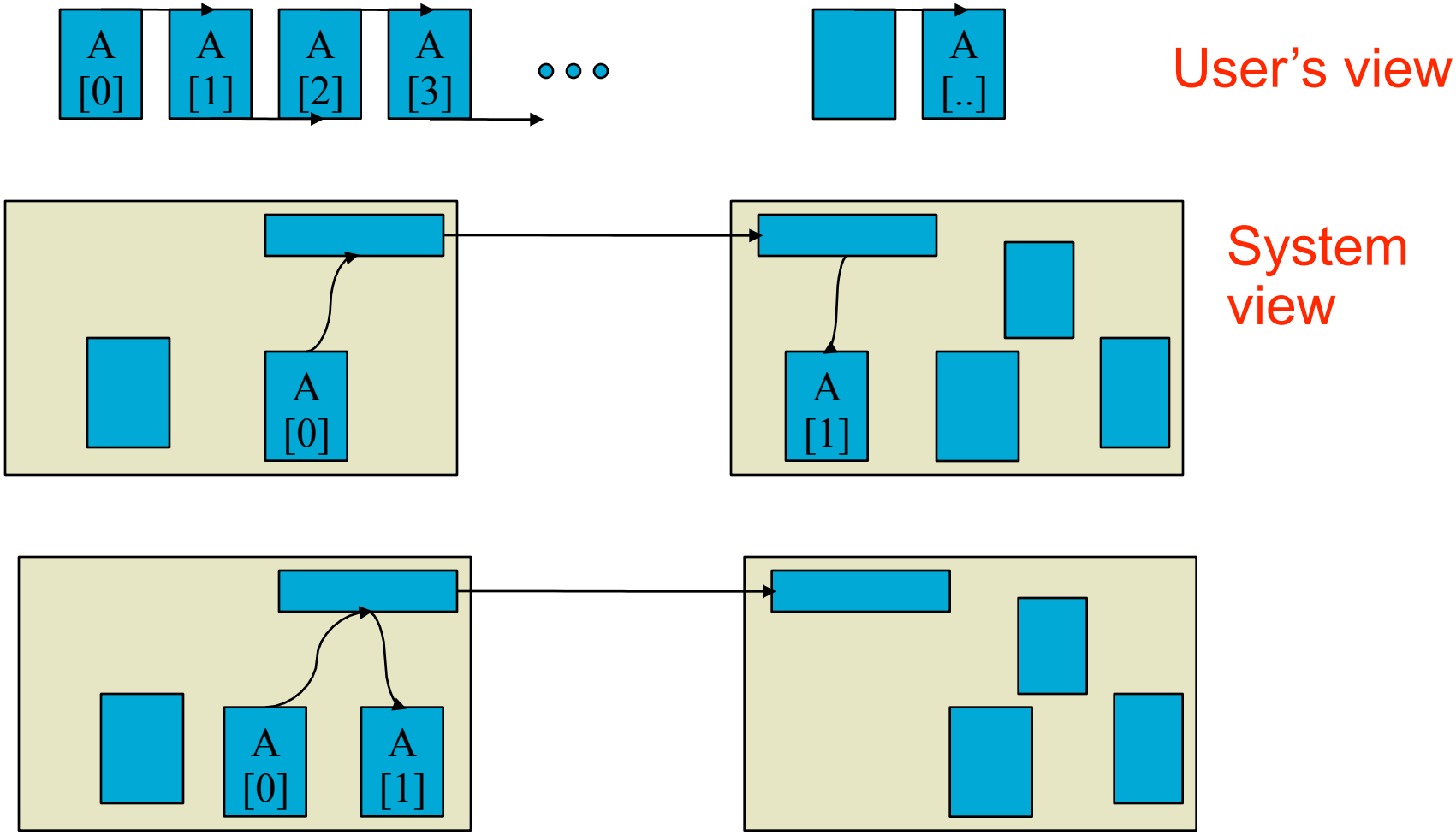
Chare Array:

– with a single global name for the collection

– each member addressed by an index

– mapping of element objects to processors handled by the system

Chare Arrays



Array Hello

```
mainmodule m {
  readonly CProxy_mymain
mainProxy;
  readonly int nElements;
  mainchare mymain { …. }
  array [1D] Hello {
    entry Hello(void);
    ent
  };
};
```

```
class Hello : public CBase_Hello
{
  public:
    Hello(CkMigrateMessage *m){}
    Hello();
```

```
Class mymain : public Chare
{
    mymain() {
        nElements=4;
        mainProxy = thisProxy;
        CProxy_Hello p =
                CProxy_Hello::ckNew(nElements);
        //Have element 0 say "hi"
        p[0].sayHi(12345);
    }
}
```

In mymain:: mymain()

Array Hello

```
void Hello::sayHi(int hiNo)
{
    ckout <<  hiNo <<"from element" << thisIndex
                                    << endl;
    if (thisIndex < nElements-1)
     //Pass the hello on:
      thisProxy[thisIndex+1].sayHi(hiNo+1);
    else
      //We've been around once-- we're done.
      mainProxy.done();
}
```

Element index

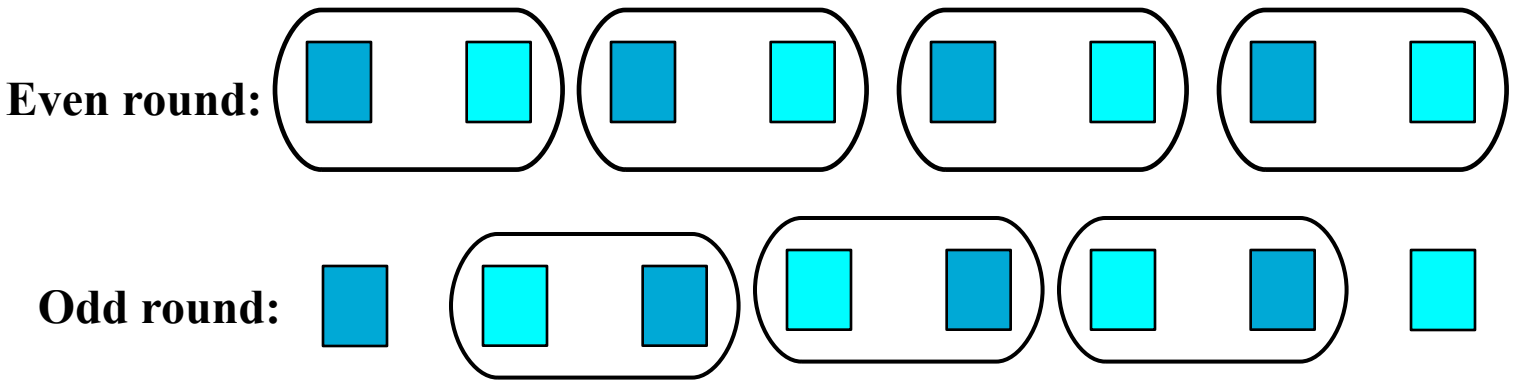
Array Proxy

Read-only

```
void mymain::done(void){
      CkExit();
}
```

Sorting numbers

- Sort n integers in increasing order.
- Create **n** chares, each keeping one number.
- In every **odd iteration** chares numbered 2i swaps with chare 2i+1 if required.
- In every **even iteration** chares 2i swaps with chare 2i-1 if required.
- After each iteration all chares report to the mainchare. After everybody reports mainchares signals next iteration. Sorting completes in **n** iterations.



Array Sort

sort.ci

```
mainmodule sort{
    readonly CProxy_myM
    readonly int nEleme

    mainchare myMain {
        entry myMain(Ck
        entry void swap
  };
  array [1D] sort{
     entry sort(void);
int
};
```

sort.h

```
class sort : public CBase_sort{
        private:
                int myValue;
        public:
                sort() ;
                sort(CkMigrateMessage *m);
                void setValue(int number);
                void swap(int round_no);
                                        (int from_index,
                                        t value);
```

myMain::myMain()

```
        swapcount=0;
        roundsDone=0;
        mainProxy = thishandle;
        CProxy_sort arr =
                CProxy_sort::ckNew(nElements);
        for(int i=0;i<nElements;i++)
                arr[i].setValue(rand());
        arr.swap(0);
```

Array Sort (continued ...)

```
void sort::swap(int roundno)
{
   bool sendright=false;
   if (roundno%2==0 && thisIndex%2==0|| roundno%2==1 && thisIndex%2==1)
      sendright=true;   //sendright is true if I have to send to right

   if((sendright && t
      mainProxy.swapd
   else{
       if(sendr
        thisPro
       else
        thisPro
   }
}
```

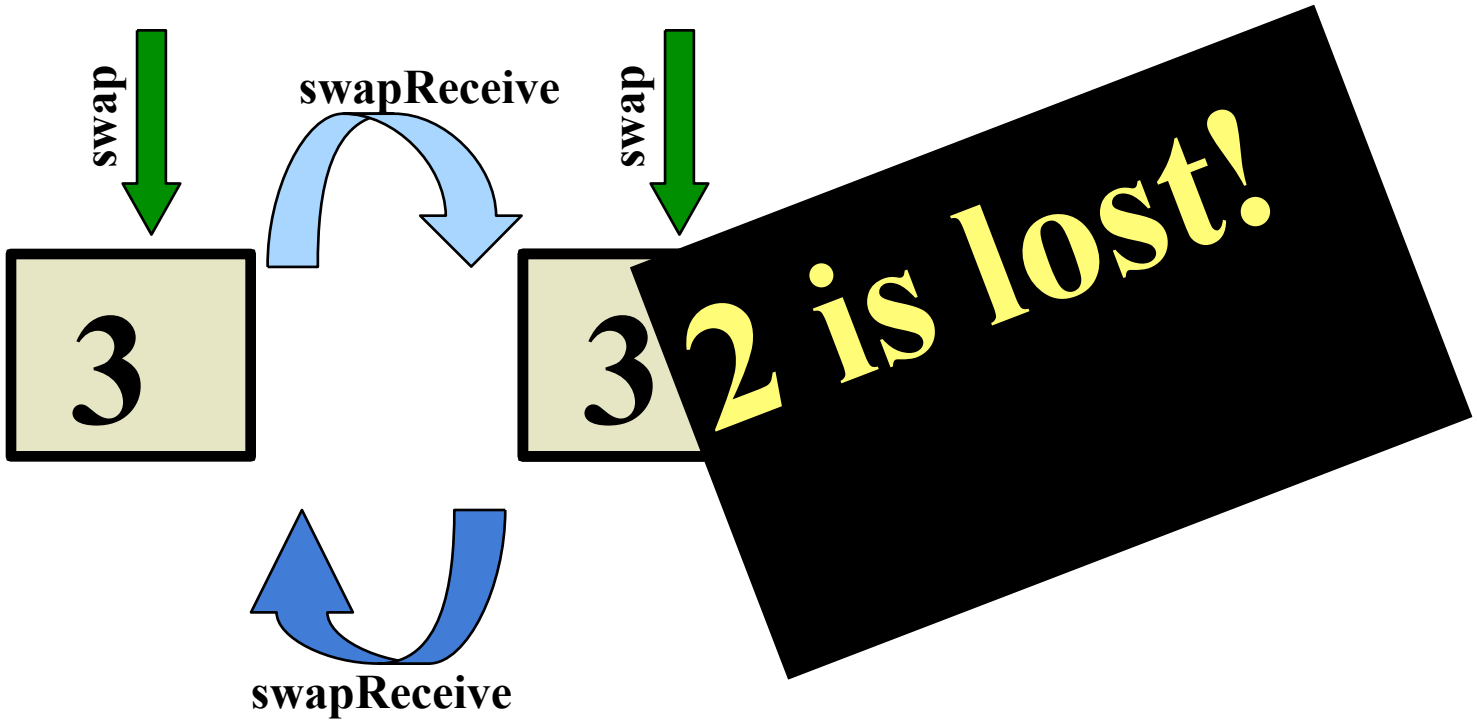
```
void sort::swapReceive(int from_index, int value)
{
      if(from_index==thisIndex-1 && value>myValue)

                                  && value<myValue)
```

```
void myMain::swapdone(void) {
    if (++swapcount==nElements) {
        swapcount=0;
        roundsDone++;
        if (roundsDone==nElements)
           CkExit();
        else
           arr.swap(roundsDone);
    }
}
```

Error!

Remember :

- ✓ Message passing is **asynchronous**.
- ✓ Messages can be delivered **out of order**.



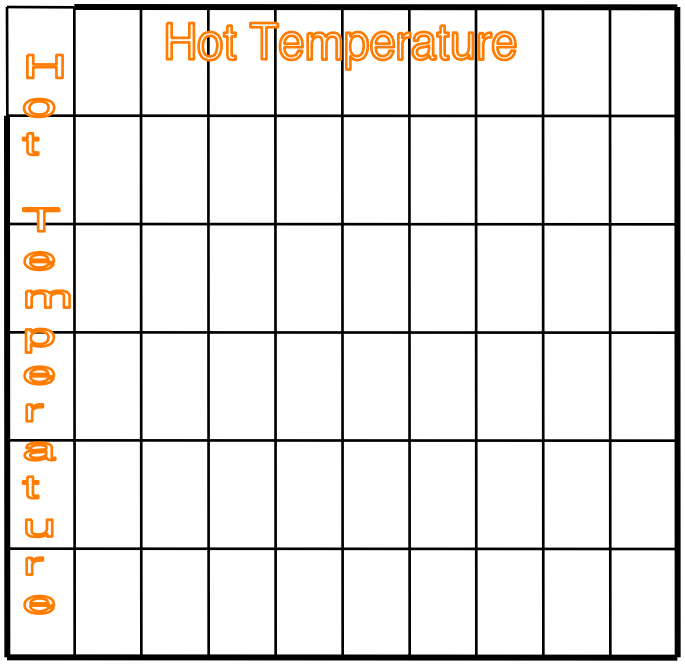
Array Sort (correct)

```
void sort::swap(int roundno)
{
   bool sendright=false;
   if (roundno%2==0 && thisIndex%2==0|| roundno%2==1 && thisIndex%2==1)
      sendright=true;   //sendright is true if I have to send to right
   if ((s
      mai
   } else
      if
   }
}
```

```
void sort::swapReceive(int from_index, int value) {

   if (from_index==thisIndex-1) {
      if (value>myValue) {
                                                    ve(thisIndex, myValue);

                                                    ve(thisIndex, value);
```

```
void myMain::swapdone(void) {
   if (++swapcount==nElements) {
      swapcount=0;
      roundsDone++;
      if (roundsDone==nElements)
         CkExit();
      else
         arr.swap(roundsDone);
   }
}
```

Example: 5-Point 2-D Stencil

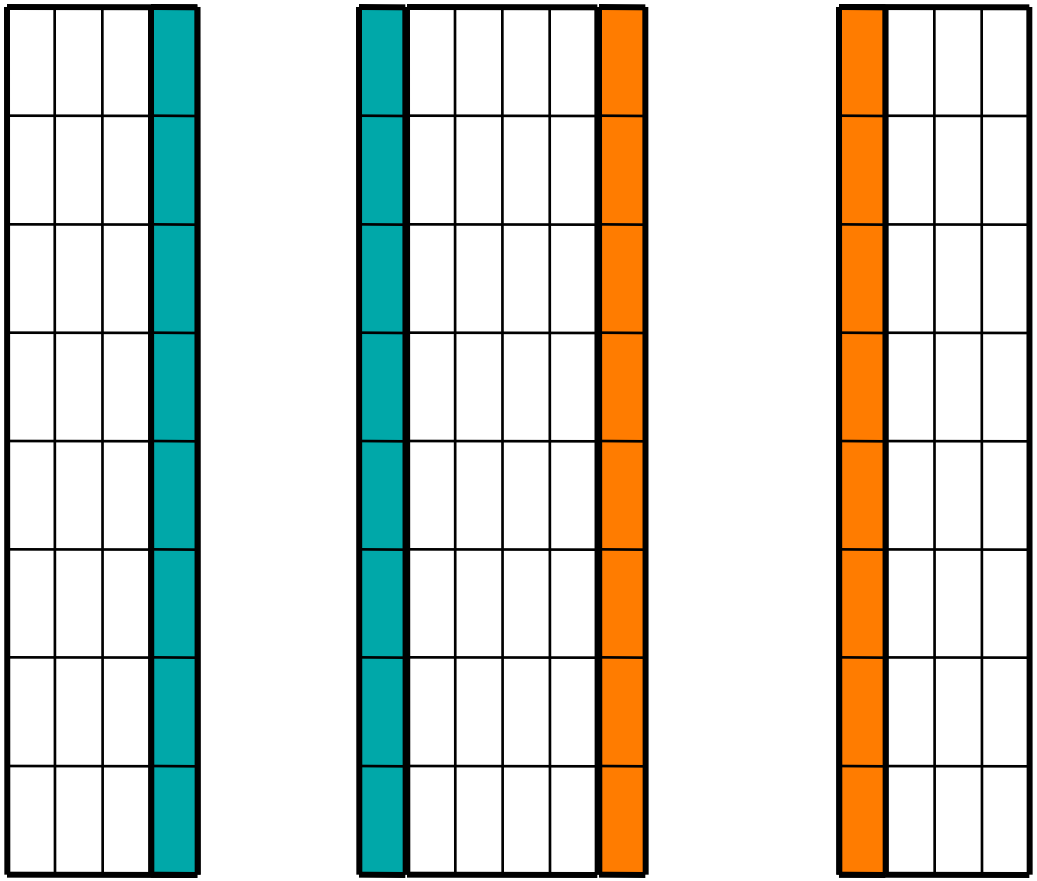


Hot temperature on two sides will slowly spread across the entire grid.

Example: 5-Point 2-D Stencil

- Input: 2D array of values with boundary condition
- In each iteration, each array element is computed as the average of itself and its neighbors (5 points)
- Iterations are repeated till some threshold difference value is reached

Parallel Solution!



Parallel Solution!

- Slice up the 2D array into sets of columns
- Chare = computations in one set
- At the end of each iteration
  - Chares exchange boundaries
  - Determine maximum change in computation
- Output result at each step or when threshold is reached

Arrays as Parameters

- Array cannot be passed as pointer
- Specify the length of the array in the interface file
  - entry void bar(int n,double arr[n])
  - n is size of arr[]

Stencil Code

```
void Ar1::doWork(int sendersID, int n, double arr[])
{
    maxChange = 0.0;
    if (sendersID == thisIndex-1)
    {  leftmsg = 1; }
    //set boolean to indicate we received the left message
    else if (sendersID == thisIndex+1)
    {  rightmsg = 1; }
    //set boolean to indicate we received the right message
    // Rest of the code on a following slide
    …
}
```

Reduction

- Apply a single operation (add, max, min, ...) to data items scattered across many processors
- Collect the result in one place
- Reduce *x* across all elements
  - **contribute(sizeof(***x***), &***x***, CkReduction::***sum_int***);**
- Must create and register a callback function that will receive the final value, in main chare

Types of Reductions

- Predefined Reductions – A number of reductions are predefined, including ones that
  - Sum values or arrays
  - Calculate the product of values or arrays
  - Calculate the maximum contributed value
  - Calculate the minimum contributed value
  - Calculate the logical and of integer values
  - Calculate the logical or of contributed integer values
  - Form a set of all contributed values
  - Concatenate bytes of all contributed values
- Plus, you can create your own

Code (continued …)

```
void Ar1::doWork(int sendersID, int n, double arr[n])
{
   //Code on previous slide
   …
   if (((rightmsg == 1) && (leftmsg == 1)) || ((thisIndex == 0)
   &&
   (rightmsg == 1)) || ((thisIndex ==K-1) && (leftmsg == 1)))
   {
     // Both messages have been received and we can now
         compute the new values of the matrix
        …
     // Use a reduction to find determine if all of the maximum
        errors on each processor had a maximum change that
        is below our threshold value.

   contribute(sizeof(double), &maxChange,
                                        CkReduction::max_double);
   }
}
```

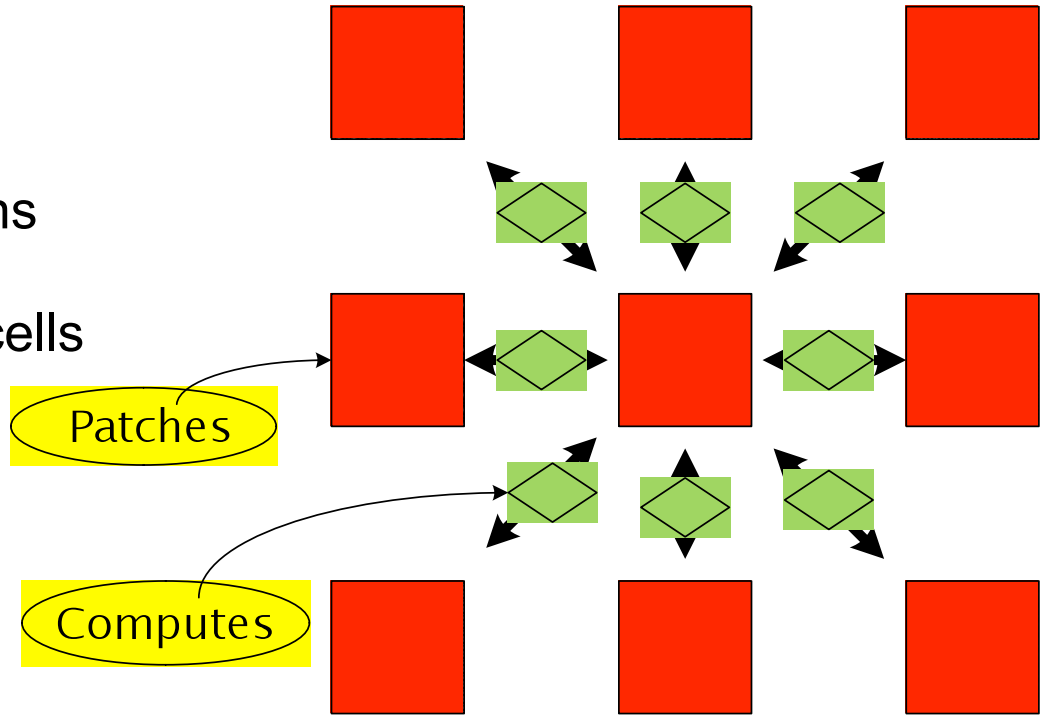
Callbacks

- A generic way to transfer control to a chare after a library(such as reduction) has finished.
- After finishing a reduction, the results have to be passed to some chare's entry method.
- To do this, create an object of type $CkCallback$ with chare's ID & entry method index
- Different types of callbacks
- One commonly used type:

CkCallback cb(<chare's entry method>,<chare's proxy>);

A Molecular Dynamics Example

- 2D Simulation space
  - Broken into a 2DArray of chares
- Called Patches (or) Cells
  - Contains particles
- Computes (or) Interactions
  - Interactions between particles in adjacent cells
- Periodic!



One time step of computation

- **Cells ------- *Vector<Particles>* ------> Interaction**
- One interaction object for each pair of Cells
  - Interaction object computes the particle interaction between the two vectors it receives
- **Interaction ------- *Resulting Forces* ------> Cells**
- Each cell receives forces from all its 8 surrounding interaction objects
  - Cells compute resultant force on its particles
  - Finds which particles need to migrate to other cells
- **Cells ------ *Vector<Migrating_Particles>* -----> Cells**

Now, some code..

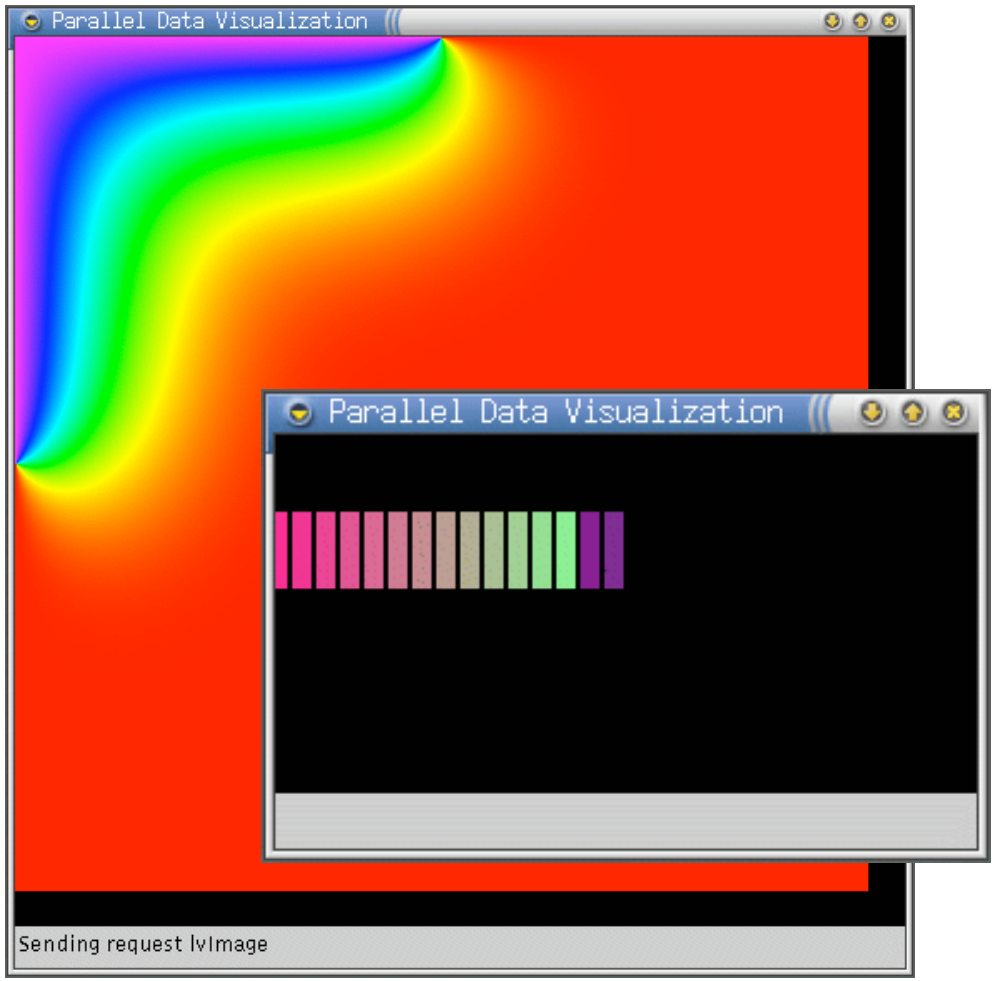
```
// cell.ci

module cell {

  array [2D] Cell {
    entry Cell();
    entry void start();
    entry void updateForces(CkVec<Particle> particles);
    entry void updateParticles(CkVec<Particle> updates);
    entry void requestNextFrame(liveVizRequestMsg *m);
  };

  array [4D] Interaction {                    // Sparse Array
    entry Interaction();
    entry void interact(CkVec<Particle>, int i, int j);
  };
};

-----------------------------------------------------------
Spare Array - Insertion

For each pair of adjacent cells (x1,y1) and (x2,y2)

  interactionArray( x1, y1, x2, y2 ).insert( /* proc number
*/ );
```

Outline

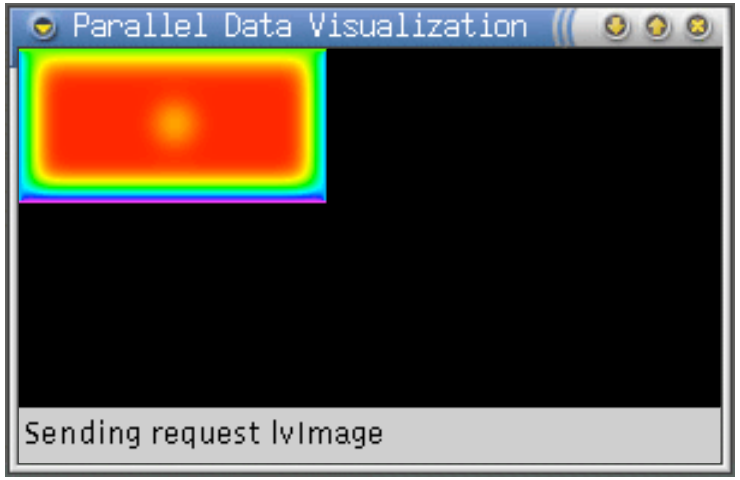
- Introduction
- Charm++ features
  - Chares and Chare Arrays
  - Parameter Marshalling
  - Examples
- Tools
  - LiveViz
  - Parallel Debugger
  - Projections
- More Charm++ Features
  - Structured Dagger Construct
  - Adaptive MPI
  - Load Balancing
- Conclusion

LiveViz – What is it?

- Charm++ library
- Visualization tool
- Inspect your program's current state
- Java client runs on any machine
- You code the image generation
- 2D and 3D modes



LiveViz – Monitoring Your Application



- LiveViz allows you to watch your application's progress
- Doesn't slow down computation when there is no client

LiveViz Setup

```
#include "liveViz.h"

Main::Main(. . .) {
  /* Do misc initilization stuff */

CkCallback c(CkIndex_Cell::requestNextFrame(0),cellArray);

liveVizConfig cfg(liveVizConfig::pix_color,
                  /* animate image */ true);

liveVizInit(cfg,cellArray,c); // Initialize the library

}
```

Adding LiveViz to Your Code

```
void Cell::requestNextFrame(liveVizPollRequestMsg *m) {
  // Compute the dimensions of the image piece we'll send
  i.e myWidthPx and myHeightPx.

  // Color pixels of particles and draw doundaries of cell
  // For greyscale it's 1 byte, for color it's 3

  // Finally, return the image data to the library
  liveVizPollDeposit(m, sx, sy, myWidthPx, myHeightPx,
intensity,this, imageBits);
}
```

Link With The LiveViz Library

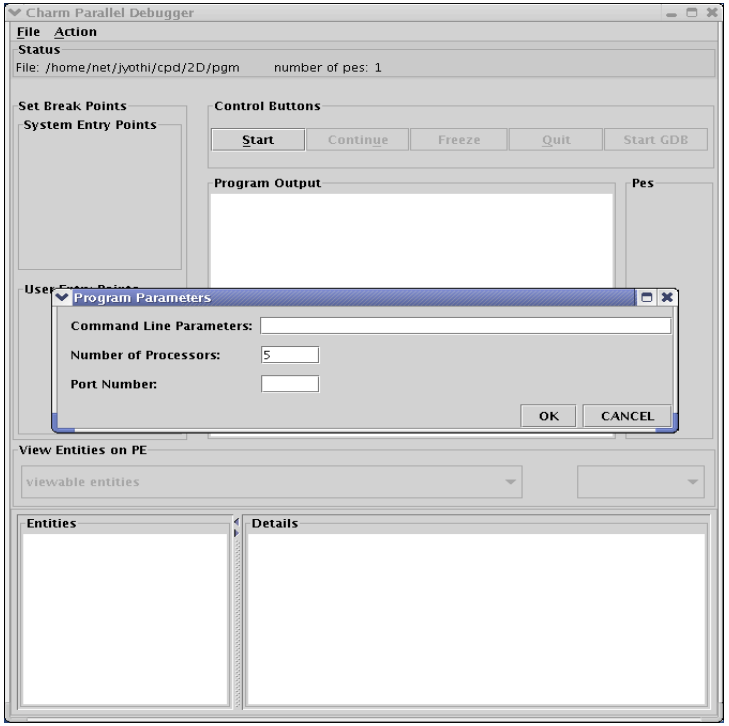
```
OPTS=-g
CHARMC=charmc $(OPTS)

all: molecular

molecular: main.o cell.o
        $(CHARMC) -language charm++ \
                  -o molecular main.o cell.o \
                  -module liveViz

...
...
```

LiveViz Summary

- Easy to use visualization library
- Simple code handles any number of clients
- Doesn’t slow computation when there are no clients connected
- Works in parallel, with load balancing, etc.

Parallel debugging support

- Parallel debugger (*charmdebug*)
- Allows programmer to view the changing state of the parallel program
- Java GUI client



Debugger features

- Provides a means to easily access and view the **major programmer visible entities**, including objects and messages in queues, during program execution
- Provides an interface to **set and remove breakpoints** on remote entry points, which capture the major programmer-visible control flows

Debugger features (contd.)

- Provides the ability to freeze and unfreeze the execution of selected processors of the parallel program, which allows a **consistent snapshot**
- Provides a way to **attach** a sequential debugger (like **GDB**) to a specific subset of processes of the parallel program during execution, which keeps a manageable number of sequential debugger windows open

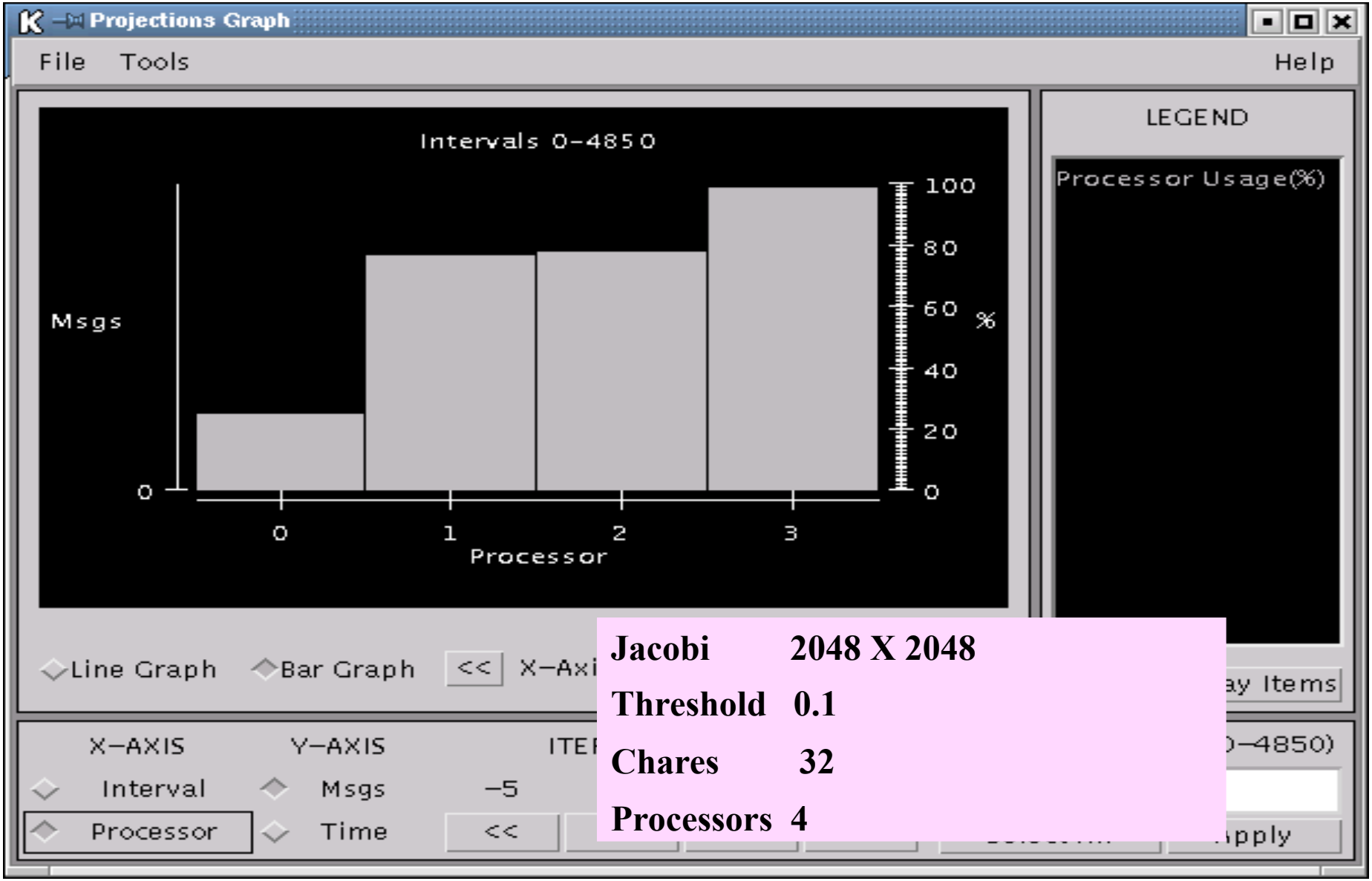
Alternative debugging support

- Uses gdb for debugging
  - Runs each node under gdb in an xterm window, prompting the user to begin execution
- Charm program has to be compiled using ‘-g’ and run with ‘++debug’ as a command-line option.

Projections: Quick Introduction

- Projections is a tool used to **analyze** the **performance** of your application
- The tracemode option is used when you build your application to enable tracing
- You get one log file per processor, plus a separate file with global information
- These files are read by Projections so you can use the Projections views to analyze performance

Screen shots – Load imbalance



Timelines – load imbalance

Outline

- Introduction
- Charm++ features
  - Chares and Chare Arrays
  - Parameter Marshalling
  - Examples
- Tools
  - LiveViz
  - Parallel Debugger
  - Projections
- More Charm++ Features
  - Structured Dagger Construct
  - Adaptive MPI
  - Load Balancing
- Conclusion

Structured Dagger

- Motivation:
  - Keeping flags & buffering manually can complicate code in charm++ model.
  - Considerable overhead in the form of thread creation and synchronization

Advantages

- Reduce the complexity of program development
  - Facilitate a clear expression of flow of control
- Take advantage of adaptive message-driven execution
  - Without adding significant overhead

What is it?

- A coordination language built on top of Charm ++
  - Structured notation for specifying intra-process control dependences in message-driven programs
- Allows easy expression of dependences among messages, computations and also among computations within the same object using various structured constructs

Structured Dagger Constructs

To Be Covered in Advanced Charm++ Session

- *atomic* {code}
- *overlap* {code}
- *when* <entrylist> {code}
- if/else/for/while
- foreach

Stencil Example Using Structured Dagger

```
stencil.ci
 array[1D] Ar1 {
…
entry void GetMessages () {
    when rightmsgEntry(), leftmsgEntry() {
        atomic { CkPrintf("Got both left and right messages \n");
                 doWork(right, left); }
    }
};

entry void rightmsgEntry();
entry void leftmsgEntry();
…
};
```

AMPI = Adaptive MPI

- Motivation:
  - Typical MPI implementations are not suitable for the new generation parallel applications
    - Dynamically varying: load shifting, adaptive refinement
  - Some legacy codes in MPI can be easily ported and run fast in current new machines
  - Facilitate those who are familiar with MPI

What is it?

- An MPI implementation built on Charm++ (MPI with virtualization)
- To provide benefits of Charm++ Runtime System to standard MPI programs
  - Load Balancing, Checkpointing, Adaptability to dynamic number of physical processors

Sample AMPI Program
Also a valid MPI Program

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char** argv){
  int ierr, rank, np, myval=0;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ierr = MPI_Comm_size(MPI_COMM_WORLD, &np);

  if(rank < np-1) MPI_Send(&myval, 1, MPI_INT, rank+1,1,MPI_COMM_WORLD);
  if(rank > 0) MPI_Recv(&myval,1, MPI_INT, rank-1,1,MPI_COMM_WORLD, &status);

  printf("rank %d completed\n", rank);
  ierr = MPI_Finalize();
}
```

AMPI Compilation

Compile:

charmc sample.c -language ampi -o sample

Run:

charmrun ./sample +p16 +vp 128 [args]

Instead of Traditional MPI equivalent:

mpirun ./sample -np 128 [args]

Comparison to Native MPI

- AMPI Performance
  - Similar to Native MPI
  - Not utilizing any other features of AMPI(load balancing, etc.)

- AMPI Flexibility
  - AMPI runs on any # of Physical Processors (eg 19, 33, 105). Native MPI needs cube #.

Current AMPI Capabilities

- Automatic checkpoint/restart mechanism
  - Robust implementation available
- Load Balancing and “process” Migration
- MPI 1.1 compliant, Most of MPI 2 implemented
- Interoperability
  - With Frameworks
  - With *Charm*++
- Performance visualization

Load Balancing

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

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

- 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);
    }
};
```

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, 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 when needed

- 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**

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.
```

Load Balancing in Jacobi2D (cont.)

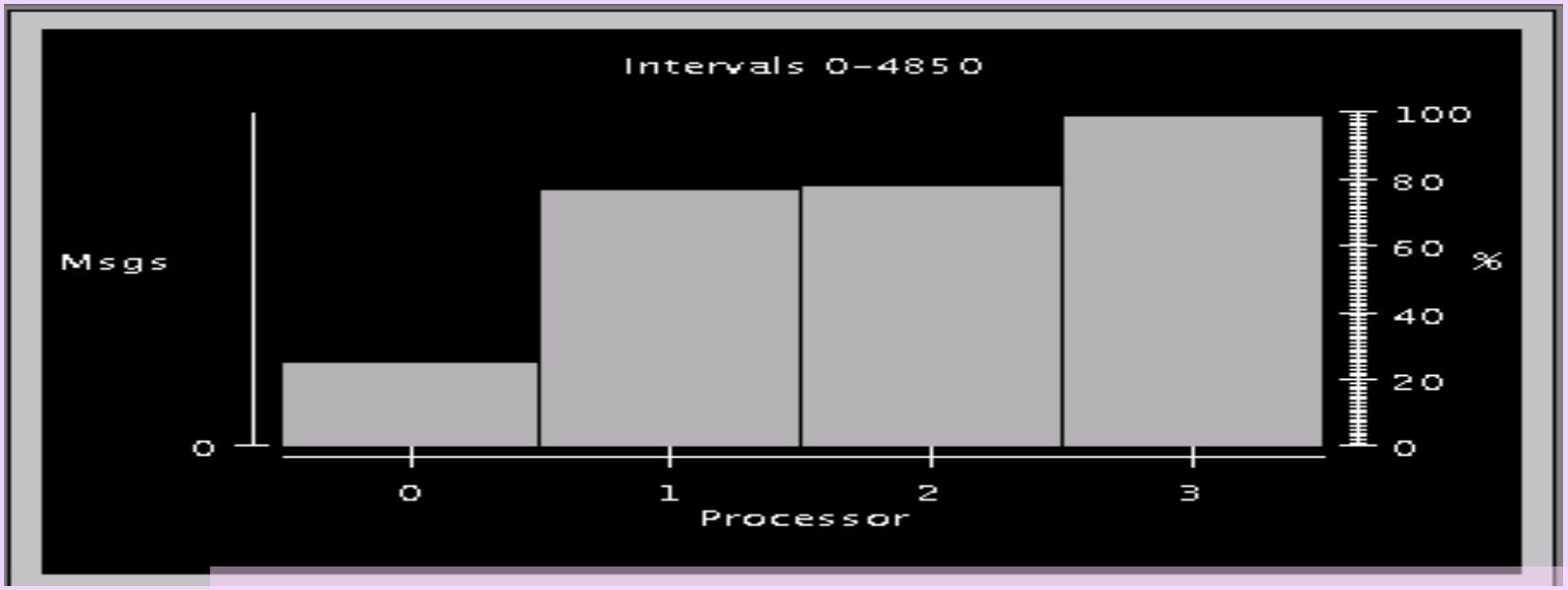
```
worker::worker(void) {
     //Initialize other parameters
```

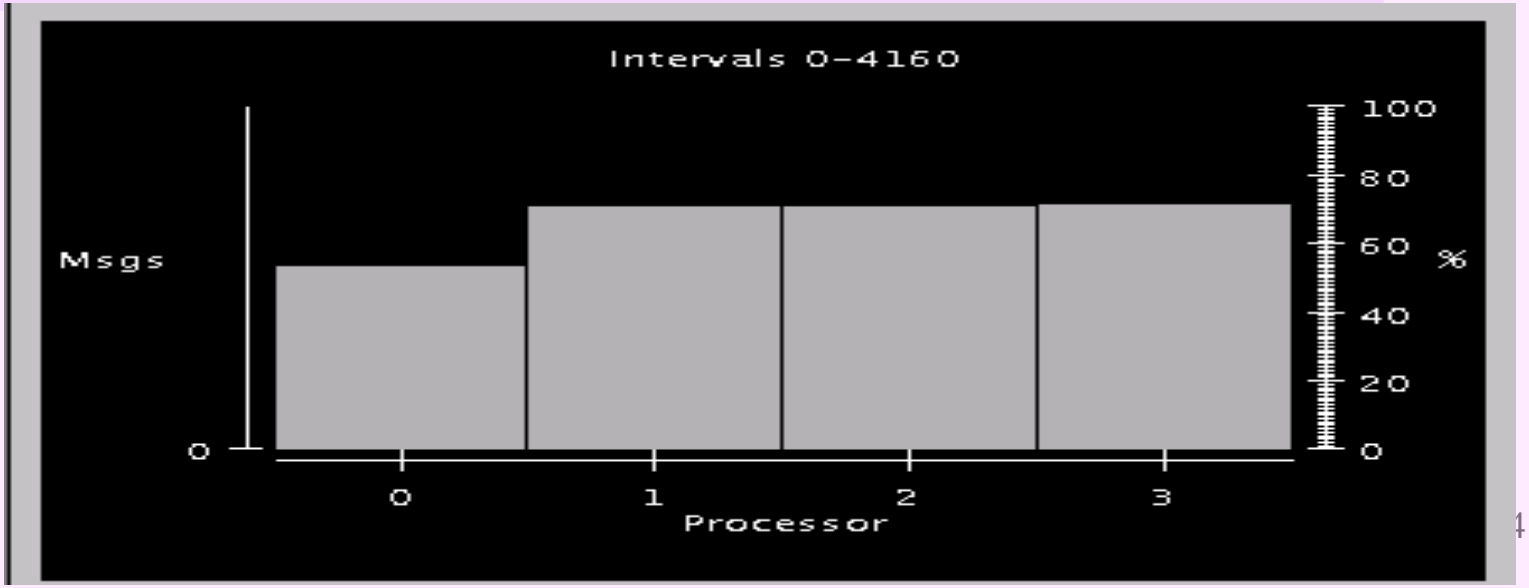
```
Void worker::doCompute(void){
  // do all the jacobi computation
  syncCount++;
  if(syncCount%64==0)
    AtSync();
  else
```

```
c
}
```

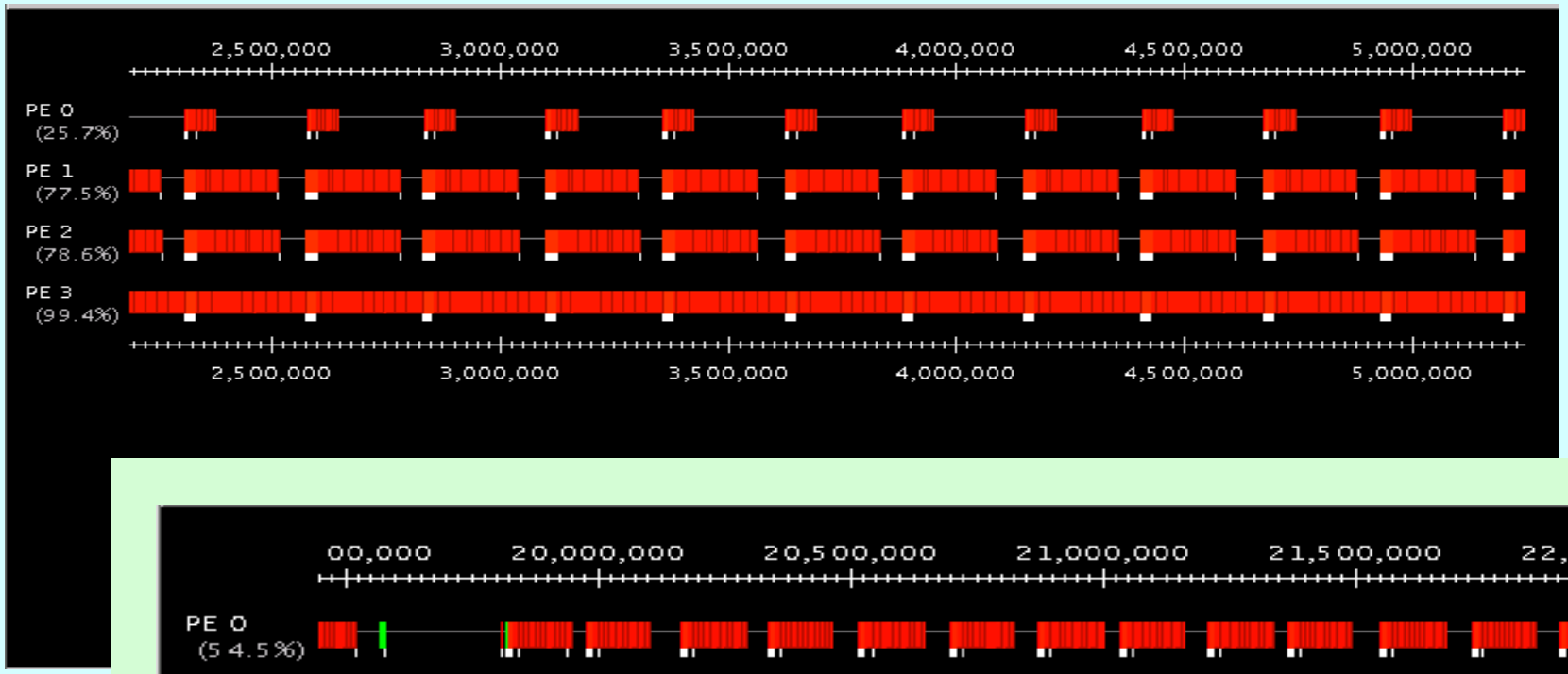
```
void worker::ResumeFromSync(void){
 contribute(1*sizeof(float),&errorMax,CkReduction::max_float);
}
```

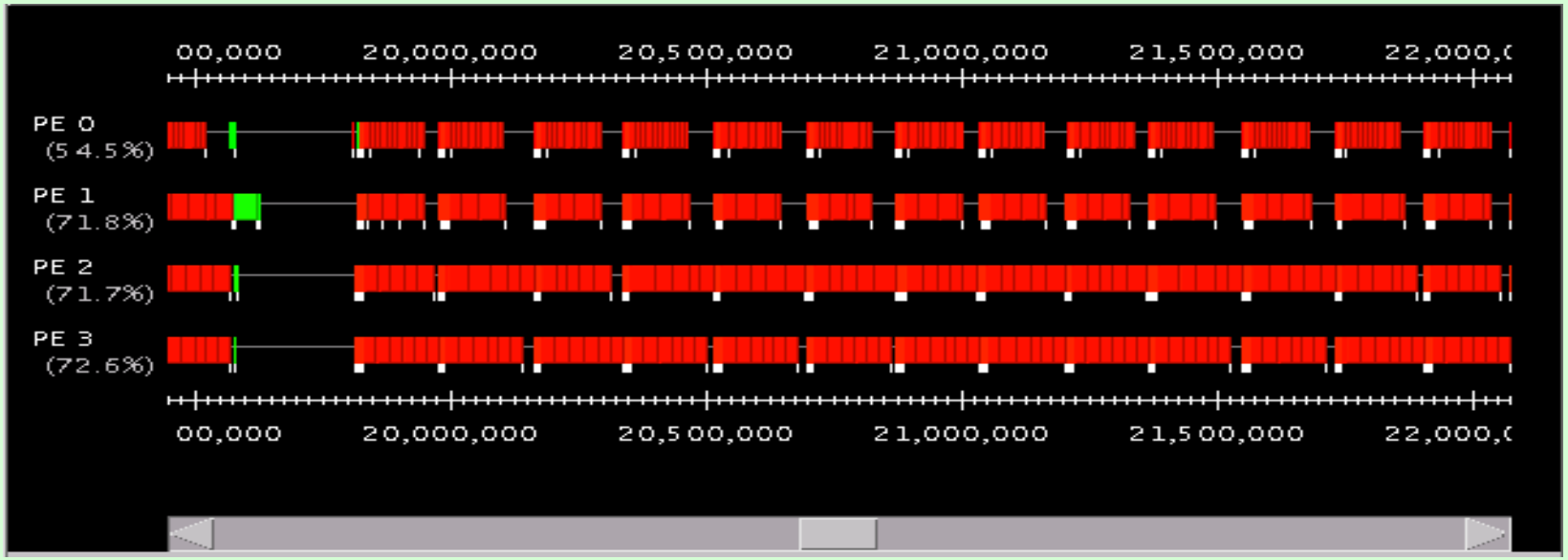
Processor Utilization: After Load Balance





Timelines: Before and After Load Balancing





Advanced Features

- Groups
- Node Groups
- Priorities
- Entry Method Attributes
- Communications Optimization
- Checkpoint/Restart

Conclusions

- Better Software Engineering
  - Logical Units decoupled from number of processors
  - Adaptive overlap between computation and communication
  - Automatic load balancing and profiling
- Powerful Parallel Tools
  - Projections
  - Parallel Debugger
  - LiveViz

More Information

- http://charm.cs.uiuc.edu
  - Manuals
  - Papers
  - Download files
  - FAQs
- ppl@cs.uiuc.edu