BigSim Tutorial

Presented by
Gengbin Zheng and Eric Bohm

Charm++ Workshop 2007
Parallel Programming Laboratory
University of Illinois at Urbana-Champaign

Outline

- Overview
- BigSim Emulator
- Charm++ on the Emulator
- Simulation framework
  - Post-mortem simulation
  - Network simulation
- Performance analysis/visualization

Simulation-based Performance Prediction

- Extremely large parallel machines are being built with enormous compute power
  - Very large number of processors with petaflops level peak performance
- Are existing software environments ready for these new machines?
  - How to write a peta-scale parallel application?
  - What will be the performance like? Can these applications scale?

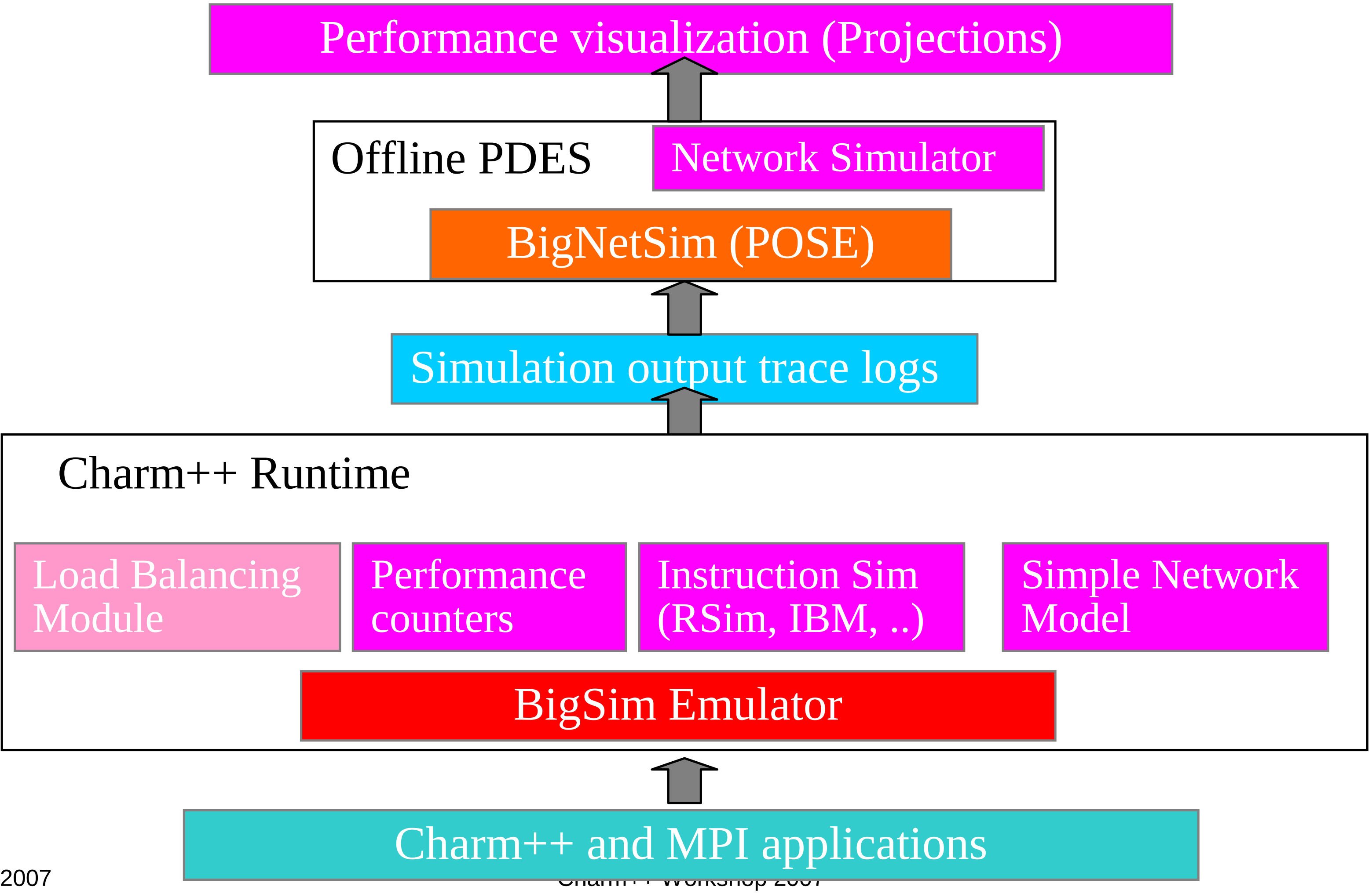
BigSim Simulation Toolkit

- BigSim emulator
  - Standalone emulator API
  - Charm++ on emulator
- BigSim simulator
  - Network simulator

Simulation-based Performance Prediction

- With focus on Charm++ and AMPI programming models
- Performance prediction is based on Parallel Discrete Event Simulation (PDES)
- Simulation is challenging, aims at different levels of fidelity
  - Processor prediction
  - Network prediction
- Two approaches
  - Direct execution (online mode)
  - Trace-driven (post-mortem mode)

Architecture of BigSim (postmortem mode)



Performance visualization (Projections)

Offline PDES

Network Simulator

BigNetSim (POSE)

Simulation output trace logs

Charm++ Runtime

Load Balancing Module

Performance counters

Instruction Sim (RSim, IBM, ..)

Simple Network Model

BigSim Emulator

Charm++ and MPI applications

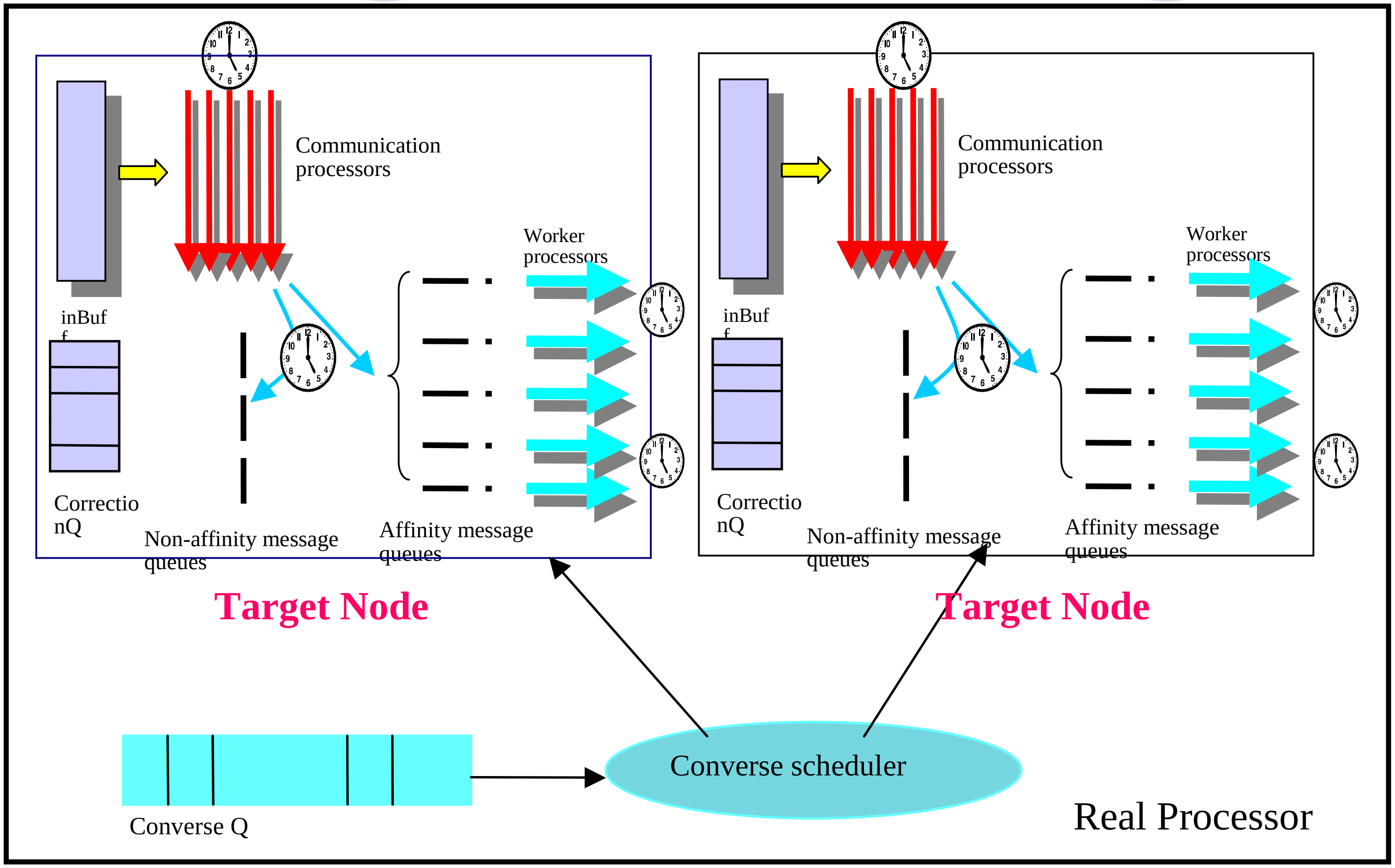
Outline

- Overview
- BigSim Emulator
- Charm++ on the Emulator
- Simulation framework
  - Online mode simulation
  - Post-mortem simulation
  - Network simulation
- Performance analysis/visualization

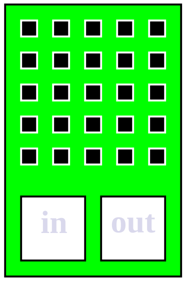
Emulator

- Emulate full machine on existing parallel machines
  - Actually run a parallel program with multi-million way parallelism
- Started with mimicking Blue Gene/C low level API
- Machine layer abstraction
  - Many multiprocessor (SMP) nodes connected via message passing

BigSim Emulator: functional view



BigSim Programming API



- Machine initialization
  - Set/get machine configuration
  - Get node ID: (x, y, z)
- Message passing
  - Register handler functions on node
  - Send packets to other nodes (x,y,z) with a handler ID

User’s API

- BgEmulatorInit(), BgNodeStart()
- BgGetXYZ()
- BgGetSize(), BgSetSize()
- BgGetNumWorkThread(), BgSetNumWorkThread()
- BgGetNumCommThread(), BgSetNumCommThread()
- BgGetNodeData(), BgSetNodeData()
- BgGetThreadID(), BgGetGlobalThreadID()
- BgGetTime()
- BgRegisterHandler()
- BgSendPacket(), etc
- BgShutdown()

Examples

- charm/examples/bigsim/emulator
  - ring
  - jacobi3D
  - maxReduce
  - prime
  - octo
  - line
  - littleMD

BigSim application example - Ring

```
typedef struct {
  char core[CmiBlueGeneMsgHeaderSizeBytes];
  int data;
} RingMsg;

void BgNodeStart(int argc, char **argv)  {
     int x,y,z, nx, ny, nz;
    BgGetXYZ(&x, &y, &z);           nextxyz(x, y, z, &nx, &ny, &nz);
     if (x == 0 && y==0 && z==0)     {
          RingMsg msg = new RingMsg;                              msg->data =  888;
          BgSendPacket(nx, ny, nz, passRingID, LARGE_WORK, sizeof(RingMsg), (char *)msg);
     }
}

void passRing(char *msg)  {
     int x, y, z,  nx, ny, nz;
     BgGetXYZ(&x, &y, &z);            nextxyz(x, y, z, &nx, &ny, &nz);
     if (x==0 && y==0 && z==0)     if (++iter == MAXITER) BgShutdown();
     BgSendPacket(nx, ny, nz, passRingID, LARGE_WORK, sizeof(RingMsg), msg);
}
```

Emulator Compilation

- Emulator libraries implemented on top of Converse/machine layer:
  - libconv-bigsim.a
  - libconv-bigsim-logs.a
- Compile with normal Charm++ with “bigemulator” target
  - *./build bigemulator net-linux*
- Compile an application with emulator API
  - *charmc -o ring ring.C -language bigsim*

Execute Application on the Emulator

- Define machine configuration
  - Function API
    - BgSetSize(x, y, z), BgSetNumWorkThread(), BgSetNumCommThread()
  - Command line options
    - *+x +y +z*
    - *+cth +wth*
    - E.g.
      - *charmrun +p4 ring +x10 +y10 +z10 +cth2 +wth4*
  - Config file
    - *+bgconfig config*

Running with bgconfig file

- **+bgconfig ./bg_config**

```
x  10
y  10
z  10
cth 2
wth 4
stacksize  4000
timing  walltime
#timing  bgelapse
#timing  counter
#cpufactor   1.0
fpfactor    5e-7
traceroot     /tmp
log      yes
correct  no
network   bluegene
```

Ring Output

```
clarity>./ring 2 2 2 2 2
Charm++: standalone mode (not using charmrun)
BG info> Simulating 2x2x2 nodes with 2 comm + 2 work threads each.
BG info> Network type: bluegene.
alpha: 1.000000e-07     packetsize: 1024        CYCLE_TIME_FACTOR:1.000000e-03.
CYCLES_PER_HOP: 5       CYCLES_PER_CORNER: 75.
0 0 0 => 0 0 1
0 0 1 => 0 1 0
0 1 0 => 0 1 1
0 1 1 => 1 0 0
1 0 0 => 1 0 1
1 0 1 => 1 1 0
1 1 0 => 1 1 1
1 1 1 => 0 0 0

BG> BlueGene emulator shutdown gracefully!
BG> Emulation took 0.000265 seconds!
Program finished.
```

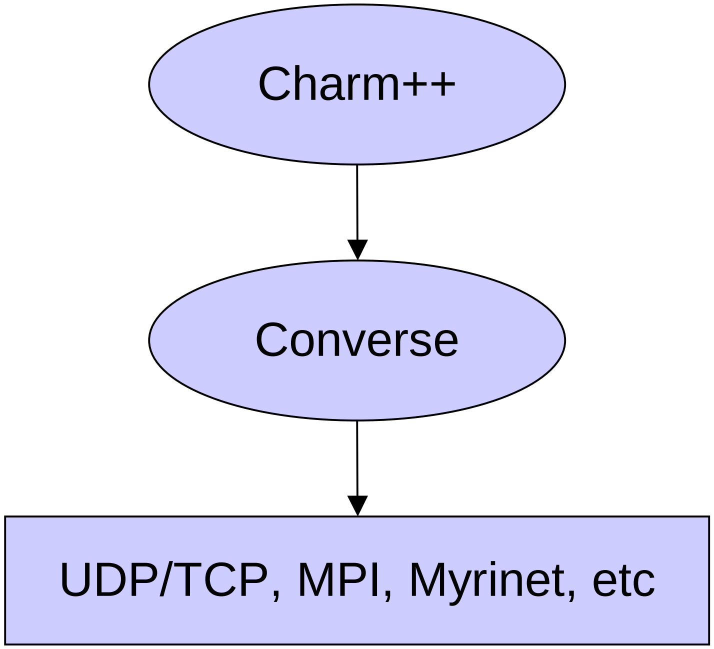
Outline

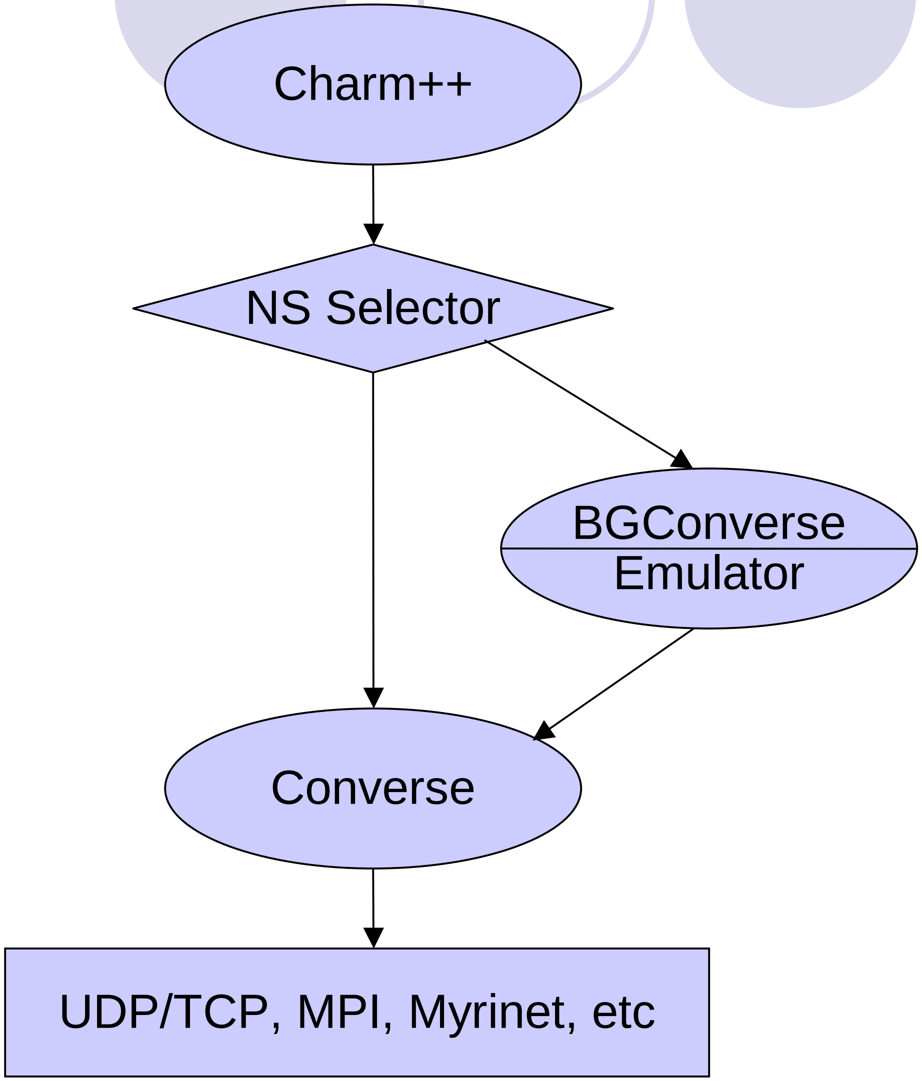
- Overview
- BigSim Emulator
- Charm++ on the Emulator
- Simulation framework
  - Online mode simulation
  - Post-mortem simulation
  - Network simulation
- Performance analysis/visualization

BigSim Charm++/AMPI

- Charm++/AMPI implemented on top of BigSim emulator, using it as another machine layer
- Support frameworks and libraries
  - Load balancing framework
  - Communication optimization library (comlib)
  - FEM
  - Multiphase Shared Array (MSA)

BigSim Charm++





Build Charm++ on BigSim

- Compile Charm++ on top of BigSim emulator
  - Build option “bigemulator”
  - E.g.
    - Charm++:

      ./build charm++ net-linux bigemulator
    - AMPI:

      ./build AMPI net-linux bigemulator

Running Charm++/AMPI Applications

- Compile Charm++/AMPI applications
  - Same as normal Charm++/AMPI
  - Just use charm/net-linux-bigsim/bin/charmc
- Running BigSim Charm++ applications
  - Same as running on emulator
    - Use command line option, or
    - Use bgconfig file

Example – AMPI Cjacobi3D

- *cd charm/net-linux-bigemulator/examples/ampi/Cjacobi3D*
- Make
  - *charmc -o jacobi jacobi.o -language ampi -module EveryLB*

./charmrun +p2 ./jacobi 2 2 2 +vp8 +bgconfig ~/bg_config +balancer GreedyLB +LBDebug 1

```
[0] GreedyLB created
iter 1 time: 1.022634 maxerr: 2020.200000
iter 2 time: 0.814523 maxerr: 1696.968000
iter 3 time: 0.787009 maxerr: 1477.170240
iter 4 time: 0.825189 maxerr: 1319.433024
iter 5 time: 1.093839 maxerr: 1200.918072
iter 6 time: 0.791372 maxerr: 1108.425519
iter 7 time: 0.823002 maxerr: 1033.970839
iter 8 time: 0.818859 maxerr: 972.509242
iter 9 time: 0.826524 maxerr: 920.721889
iter 10 time: 0.832437 maxerr: 876.344030
[GreedyLB] Load balancing step 0 starting at 11.647364 in PE0
n_obj:8 migratable:8 ncom:24
GreedyLB: 5 objects migrating.
[GreedyLB] Load balancing step 0 finished at 11.777964
[GreedyLB] duration 0.130599s memUsage: LBManager:800KB CentralLB:0KB
iter 11 time: 1.627869 maxerr: 837.779089
iter 12 time: 0.951551 maxerr: 803.868831
iter 13 time: 0.960144 maxerr: 773.751705
iter 14 time: 0.952085 maxerr: 746.772667
iter 15 time: 0.956356 maxerr: 722.424056
iter 16 time: 0.965365 maxerr: 700.305763
iter 17 time: 0.947866 maxerr: 680.097726
iter 18 time: 0.957245 maxerr: 661.540528
iter 19 time: 0.961152 maxerr: 644.421422
iter 20 time: 0.960874 maxerr: 628.564089

BG> Bigsim mulator shutdown gracefully!
BG> Emulation took 36.762261 seconds!
```

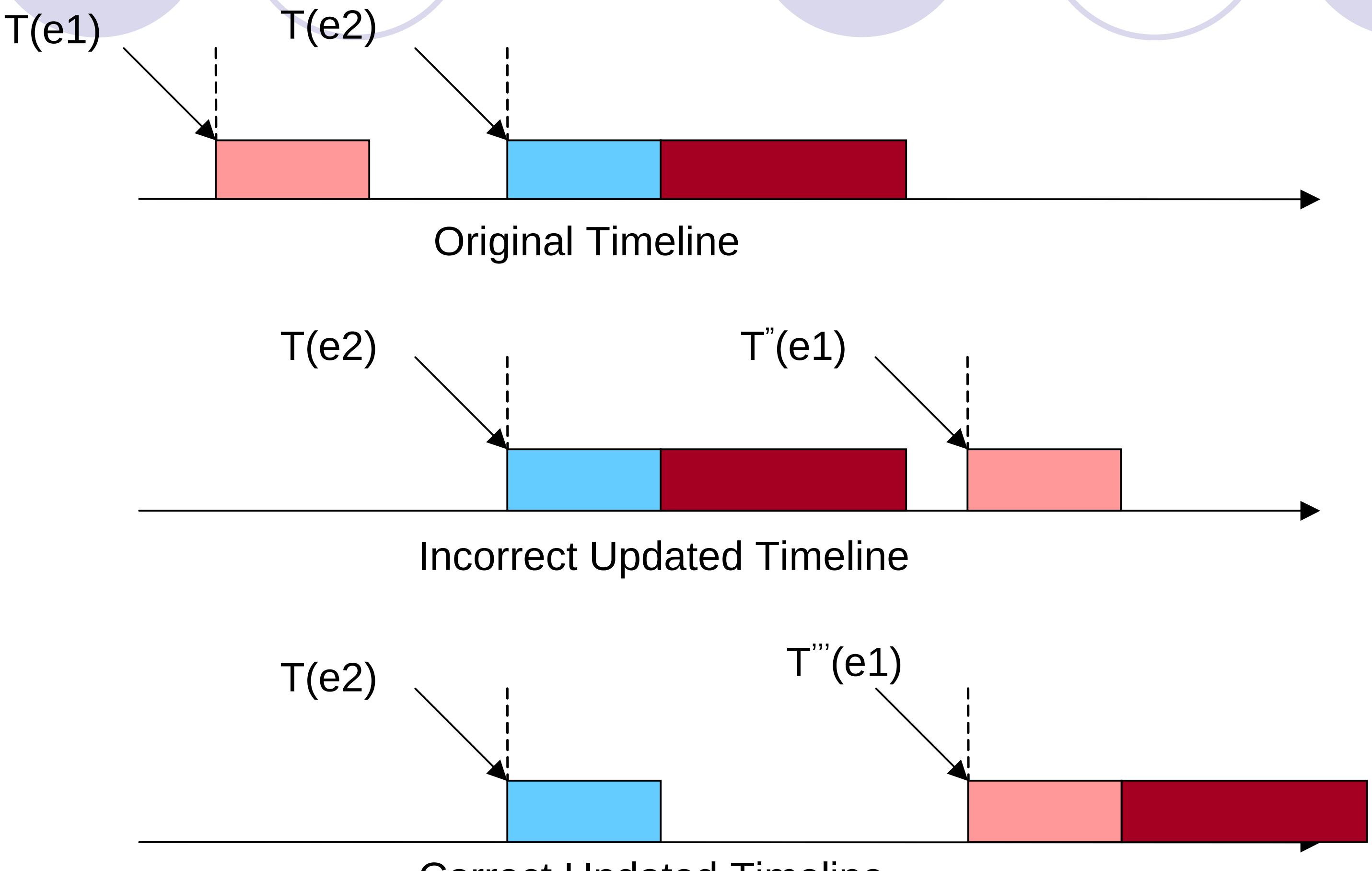
Performance Prediction

- How to predict performance?
  - Different levels of fidelity
  - Sequential portion:
    - User supplied timing expression
    - Wall clock time
    - Performance counters
    - Instruction level simulation
  - Message passing:
    - Simple latency-based network model
    - Contention-based network simulation

How to Ensure Simulation Accuracy

- The idea:
  - Take advantage of inherent determinacy of an application
  - Don’t need rollback - same user function then is executed only once
  - In case of out of order delivery, only timestamps of events are adjusted

Timestamp Correction (Jacobi1D)



LEGEND:

🟥	getStripFromRight (e1)
🟦	getStripFromLeft (e2)
■	doWork

Structured Dagger (Jacobi1D)

```
entry void jacobiLifeCycle()
{
  for (i=0; i<MAX_ITER; i++)
  {
    atomic {sendStripToLeftAndRight();}
    overlap
    {
      when getStripFromLeft(Msg *leftMsg)
         { atomic { copyStripFromLeft(leftMsg); } }
      when getStripFromRight(Msg *rightMsg)
         { atomic { copyStripFromRight(rightMsg); } }
    }
    atomic{ doWork(); /* Jacobi Relaxation */ }
  }
}
```

Sequential time - BgElapse

- **BgElapse**

```
entry void jacobiLifeCycle()
{
  for (i=0; i<MAX_ITER; i++)
  {
    atomic {sendStripToLeftAndRight();}
    overlap
    {
      when getStripFromLeft(Msg *leftMsg)
         { atomic { copyStripFromLeft(leftMsg); } }
      when getStripFromRight(Msg *rightMsg)
         { atomic { copyStripFromRight(rightMsg); } }
    }
    atomic{ doWork(); BgElapse(10e-3);}
  }
}
```

Sequential Time – using Wallclock

- Wallclock measurement of the time can be used via a suitable multiplier (scale factor)
- Run application with +bgwalltime and +bgcpufactor, or
- +bgconfig ./bgconfig:

```
timing  walltime
cpufactor   0.7
```

- Good for predicting a larger machine using a fraction of the machine

Sequential Time – performance counters

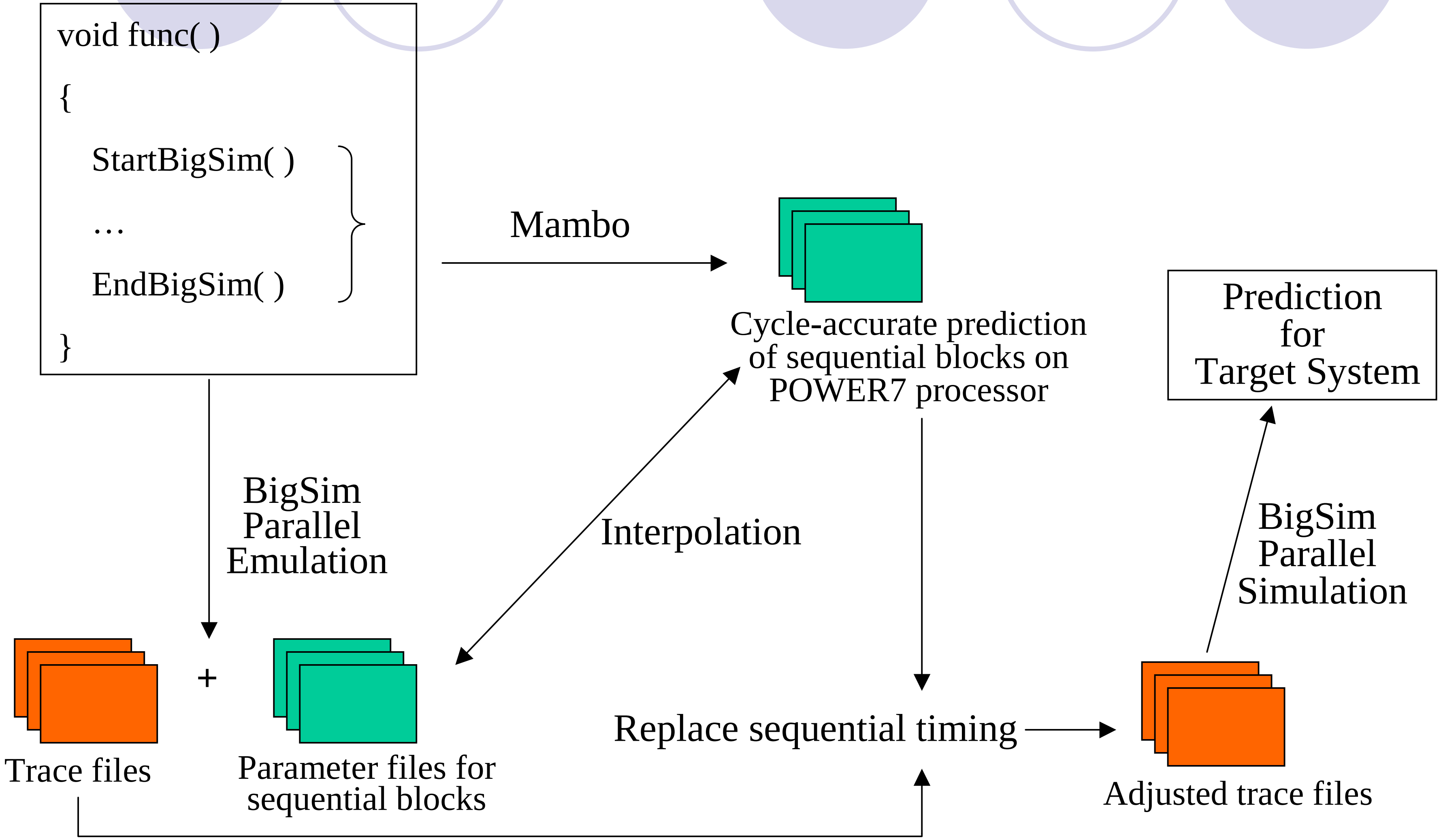
- Count floating-point, integer, memory and branch instructions (for example) with hardware counters
  - with a simple heuristic, use the expected time for each of these operations on the target machine to give the predicted total computation time.
- Cache performance and the memory footprint effects can be approximated by percentage of memory accesses and cache hit/miss ratio.
- Perfex and PAPI are supported
- Example of use, for a floating-point intensive code:

```
+bgconfig ./bg_config
 timing  counter
 fpfactor    5e-7
```

Sequential Time – Instruction level simulation

- Run instruction-level simulator separately to get accurate timing information (sampling)
- An interpolation-based scheme
  - Use result of a smaller scale instruction level simulation to interpolate for large dataset
    - do a least-squares fit to determine the coefficients of an approximation polynomial function

Case study: BigSim / Mambo



```
void func( )
{
   StartBigSim( )
   …
   EndBigSim( )
}
```

Mambo

Cycle-accurate prediction of sequential blocks on POWER7 processor

Prediction for Target System

BigSim Parallel Emulation

Interpolation

BigSim Parallel Simulation

Trace files + Parameter files for sequential blocks

Replace sequential timing

Adjusted trace files

Using interpolation tool

- Compile interpolation tool
  - Install GSL, the GNU Scientific Library
  - cd charm/examples/bigsim/tools/rewritelog
  - Modify the file interpolatelog.C to match your particular tastes.

    OUTPUTDIR specifies a directory for the new logfiles

    CYCLE_TIMES_FILE specifies the file which contains accurate timing information
  - Make
- Modify source code
  - Insert startTraceBigSim() call before a compute kernel. Add an endTraceBigSim() call after the kernel. Currently the first call takes between 0 and 20 parameters describing the computation.

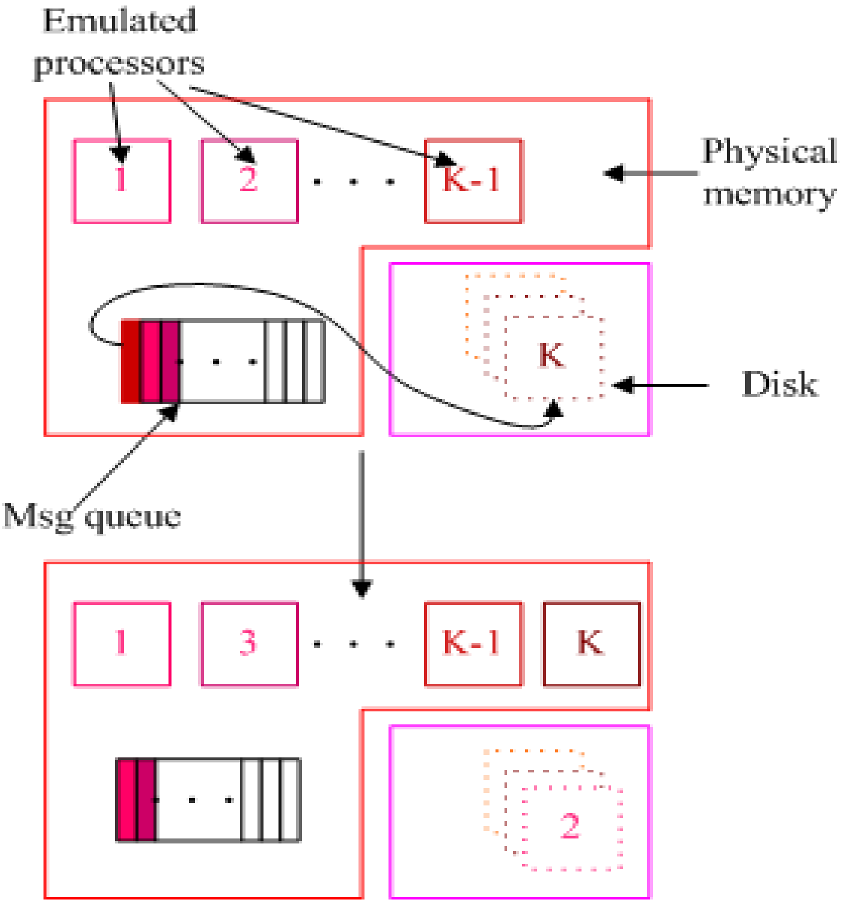
```
startTraceBigSim(param1, param2, param3, …);
// Some serial computational kernel goes here
endTraceBigSim("EventName");
```

Using interpolation tool (cont.)

- Run the application through emulator, generating trace logs (bgTrace*)and parameter files (param.*)
- Run the same application with instruction-level simulator, get accurate timing indexed by parameters
- Run interpolation tool under bgTrace dir:
  - ./interpolatelog

Out-of-core Emulation

- Motivation
  - Physical memory is shared
  - VM system would not handle well
- Message driven execution
  - Peek msg queue => what execute next? (prefetch)



Using Out-of-core

- Compile an application with bigemulator
- Run the application through the emulator, and command line option:
  - *+ooc 512*

Simple Network Model

- No contention modeling
  - Latency and topology based
- Built-in network models for
  - Quadrics (Lemieux)
  - Blue Gene/C
  - Blue Gene/L

Choose Network Model at Run-time

- Command line option:
  - *+bgnetwork bluegenel*
- BigSim config file:
  - *+bgconfig ./bg_config*

```
network   bluegenel
```

How to Add a New Network Model

- Inherit from this base class defined in blue_network.h:

```
class BigSimNetwork
{
protected:
  double alpha;       // cpu overhead of sending a message
  char *myname;    // name of this network
public:
  inline double alphacost() { return alpha; }
  inline char *name() { return myname; }
  virtual double latency(int ox, int oy, int oz, int nx, int ny, int nz, int
    bytes) = 0;
  virtual void print() = 0;
};
```

How to Obtain Predicted Time

- BgGetTime()
  - Print to stdout is not useful actually
  - Because the printed time at execution time is not final.
  - Final timestamp can only be obtained after timestamp correction (simulation) finishes.

How to Obtain Predicted Time (cont.)

- BgPrint (char *)
  - Bookmarking events
  - E.g.

    BgPrint(“start at %f\n”);
  - Output to bgPrintFile.0 when simulation finishes
    - Look back these bookmarks
    - Replace “%f” with the committed time

Running Applications with Online Network Simulator

- Two modes
  - With simple network model (timestamp correction)
    - *+bgcorrect*
  - Partial prediction only (no timestamp correction)
    - *+bglog*
    - Generate trace logs for post-mortem simulation

With bgconfig

+bgconfig ./bg_config

```
x  64
y  32
z  32
cth 1
wth 1
stacksize  4000
timing  walltime
#timing  bgelapse
#timing  counter
cpufactor   1.0
#fpfactor    5e-7
traceroot     /tmp
log       yes
correct  no
network   bluegene
```

BigSim Trace Log

- Execution of messages on each target processor is stored in trace logs (binary format)
  - named bgTrace[#], # is simulating processor number.
- Can be used for
  - Visualization/Performance study
  - Post-mortem simulation with different network models
- Loadlog tool
  - Binary to human readable ascii format conversion
  - *charm/examples/bigsim/tools/loadlog*

ASCII Log Sample

```
[22]  0x80a7a60 name:msgep (srcnode:0 msgID:21) ep:1
[[  recvtime:0.000498 startTime:0.000498 endTime:0.000498 ]]
backward:
forward: [0x80a7af0 23]

[23]  0x80a7af0 name:Chunk_atomic_0 (srcnode:-1 msgID:-1) ep:0
[[  recvtime:-1.000000 startTime:0.000498 endTime:0.000503 ]]
msgID:3 sent:0.000498 recvtime:0.000499 dstPe:7 size:208
msgID:4 sent:0.000500 recvtime:0.000501 dstPe:1 size:208
backward: [0x80a7a60 22]
forward: [0x80a7ca8 24]

[24]  0x80a7ca8 name:Chunk_overlap_0 (srcnode:-1 msgID:-1) ep:0
[[  recvtime:-1.000000 startTime:0.000503 endTime:0.000503 ]]
backward: [0x80a7af0 23]
forward: [0x80a7dc8 25] [0x80a8170 28]
```