Charm++

Migratable Objects + Asynchronous Methods + Adaptive Runtime = Performance + Productivity

Laxmikant V. Kale*
Anshu Arya Nikhil Jain Akhil Langer Jonathan Lifflander
Harshitha Menon Xiang Ni Yanhua Sun Ehsan Totoni
Ramprasad Venkataraman* Lukasz Wesolowski

Parallel Programming Laboratory
Department of Computer Science
University of Illinois at Urbana-Champaign

*{kale, ramv}@illinois.edu

Benchmarks

Required

- 1D FFT
- Random Access
- Dense LU Factorization

Optional

- Molecular Dynamics
- Adaptive Mesh Refinement
- Sparse Triangular Solver

Metrics: Performance and Productivity

Our Implementations in Charm++

Productivity						Performance		
Code	C++	CI	Benchmark Subtotal	Driver	Total	Machine	Max Cores	Performance Highlight
1D FFT	54	29	83	102	185	IBM BG/P IBM BG/Q	64K 16K	2.71 TFlop/s 2.31 TFlop/s
Random Access	76	15	91	47	138	IBM BG/P IBM BG/Q	128K 16K	43.10 GUPS 15.00 GUPS
Dense LU	1001	316	1317	453	1770	Cray XT5	8K	55.1 TFlop/s (65.7% peak)
Molecular Dynamics	571	122	693	n/a	693	IBM BG/P IBM BG/Q	128K 16K	24 ms/step (2.8M atoms) 44 ms/step (2.8M atoms)
Triangular Solver	642	50	692	56	748	IBM BG/P	512	48x speedup on 64 cores with helm2d03 matrix
AMR	1126	118	1244	n/a	1244	IBM BG/Q	32k	22 timesteps/sec, 2d mesh, max 15 levels refinement

C++ Regular C++ code
CI Parallel interface descriptions and control flow DAG

Capabilities

Demonstrated Productivity Benefits

- Automatic load balancing
- Automatic checkpoints
- Tolerating process failures
- Asynchronous, non-blocking collective communication
- Interoperating with MPI

For more info

http://charm.cs.illinois.edu/

Capabilities: Automated Dynamic Load Balancing

- Measurement based fine-grained load balancing
  - Principle of persistence - recent past indicates near future.
  - Charm++ provides a suite of load-balancers.

Capabilities: Automated Dynamic Load Balancing

- Measurement based fine-grained load balancing
  - Principle of persistence - recent past indicates near future.
  - Charm++ provides a suite of load-balancers.
- How to use?
  - Periodic calls in application - `AtSync()`.
  - Command line argument - `+balancer Strategy`.

Capabilities: Automated Dynamic Load Balancing

- Measurement based fine-grained load balancing
  - Principle of persistence - recent past indicates near future.
  - Charm++ provides a suite of load-balancers.
- How to use?
  - Periodic calls in application - `AtSync()`.
  - Command line argument - `+balancer Strategy`.
- MetaBalancer - When and how to load balance?
  - Monitors the application continuously and predicts behavior.
  - Decides when to invoke which load balancer.
  - Command line argument - `+MetaLB`

Capabilities: Checkpointing Application State

- Checkpointing to disk for split execution
  `CkStartCheckpoint(callback)`
  - Designed for applications need to run for a long period, but cannot get all the allocation needed at one time.
- Restart applications from checkpoint on any number of processors

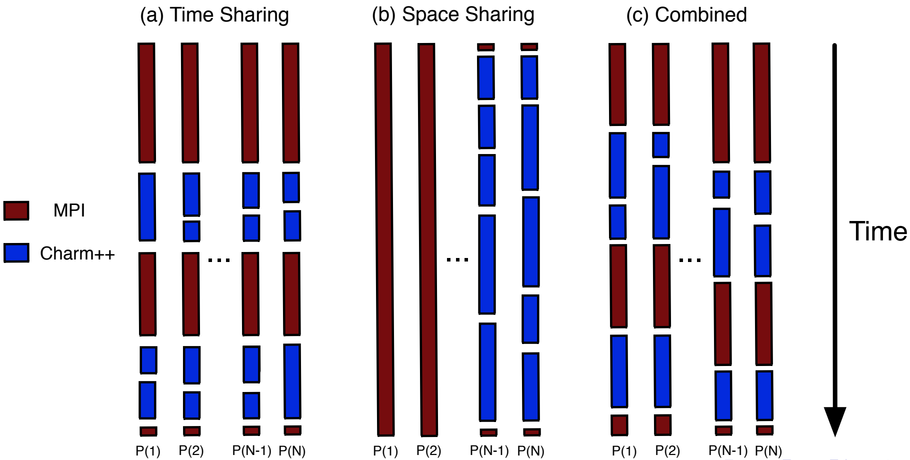
Capabilities: Tolerating Process Failures

- Double in-memory checkpointing for online recovery
  `CkStartMemCheckpoint(callback)`
  - To tolerate the more and more frequent failures in HPC system.
- Injecting failure and automatically detection of failures
  `CkDieNow()`

Capabilities: Interoperability

Invoke Charm++ from MPI

- Callable like other external MPI libraries
- Use MPI communicators to enable the following modes



Capabilities: Interoperability

Trivial Changes to Existing Codes

- Initialize and destroy Charm++ instances
- Use interface functions to transfer control

```
//MPI_Init and other basic initialization
{ optional pure MPI code blocks }

//create a communicator for initializing Charm++
MPI_Comm_split(MPI_COMM_WORLD, peid%2, peid, &newComm);
CharmLibInit(newComm, argc, argv);

{ optional pure MPI code blocks }

//Charm++ library invocation
if(myrank%2)
  fft1d(inputData,outputData,data_size);

//more pure MPI code blocks
//more Charm++ library calls

CharmLibExit();
//MPI cleanup and MPI_Finalize
```

Capabilities: Asynchronous, Non-blocking Collective Communication

- Overlap collective communication with other work
- Topological Routing and Aggregation Module (TRAM)
  - Transforms point-to-point communication into collectives
  - Minimal topology-aware software routing
  - Aggregation of fine-grained communication
  - Recombining at intermediate destinations
- Intuitive expression of collectives through overloading constructs for point-to-point sends (e.g. broadcast)

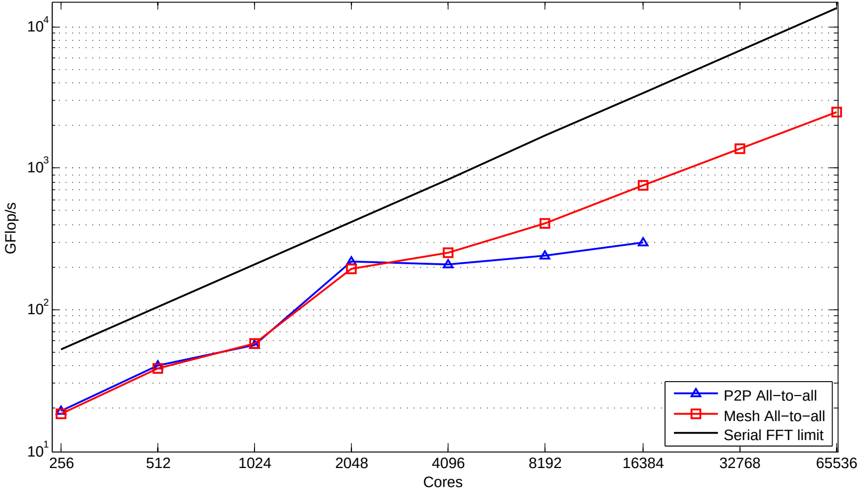
FFT: Parallel Coordination Code

doFFT()

```
for(phase = 0; phase < 3; ++phase) {
    atomic {
        sendTranspose();
    }
    for(count = 0; count < P; ++count)
        when recvTranspose[phase] (fftMsg *msg) atomic {
            applyTranspose(msg);
        }
    if (phase < 2) atomic {
        fftw_execute(plan);
        if(phase == 0)
            twiddle();
    }
}
```

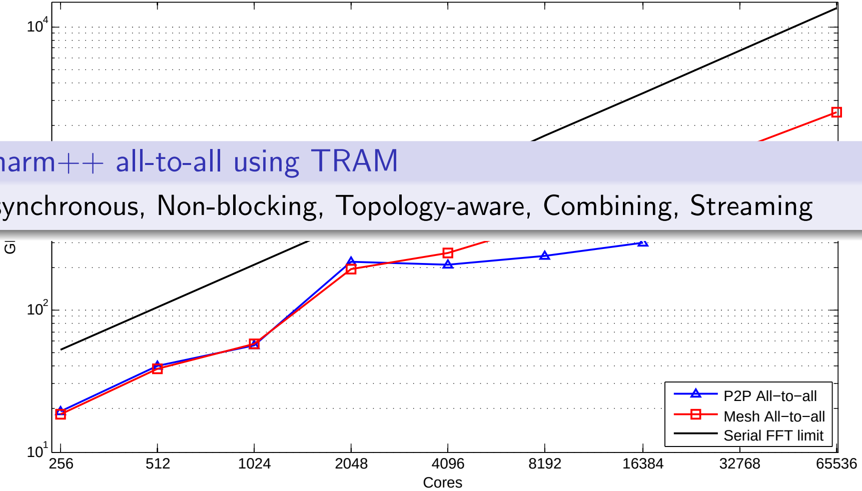
FFT: Performance

IBM Blue Gene/P (Intrepid), 25% memory, ESSL /w fftw wrappers



FFT: Performance

IBM Blue Gene/P (Intrepid), 25% memory, ESSL /w fftw wrappers



Charm++ all-to-all using TRAM

Asynchronous, Non-blocking, Topology-aware, Combining, Streaming

Random Access

Productivity

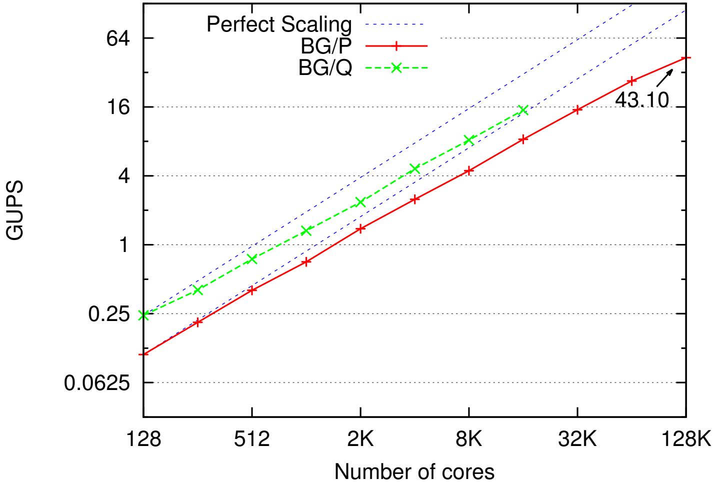
- Use point to point sends and let Charm++ optimize communication
- Automatically detect and adapt to network topology of partition

Performance

- Automatic communication optimization using TRAM
  - Aggregation of fine-grained communication
  - Minimal topology-aware software routing
  - Recombining at intermediate destinations

Random Access: Performance

IBM Blue Gene/P (Intrepid), BlueGene/Q (Vesta)



LU: Capabilities

- Composable library
  - Modular program structure
  - Seamless execution structure (interleaved modules)

LU: Capabilities

- Composable library
  - Modular program structure
  - Seamless execution structure (interleaved modules)
- Block-centric
  - Algorithm from a block's perspective
  - Agnostic of processor-level considerations

LU: Capabilities

- Composable library
  - Modular program structure
  - Seamless execution structure (interleaved modules)
- Block-centric
  - Algorithm from a block's perspective
  - Agnostic of processor-level considerations
- Separation of concerns
  - Domain specialist codes algorithm
  - Systems specialist codes tuning, resource mgmt etc

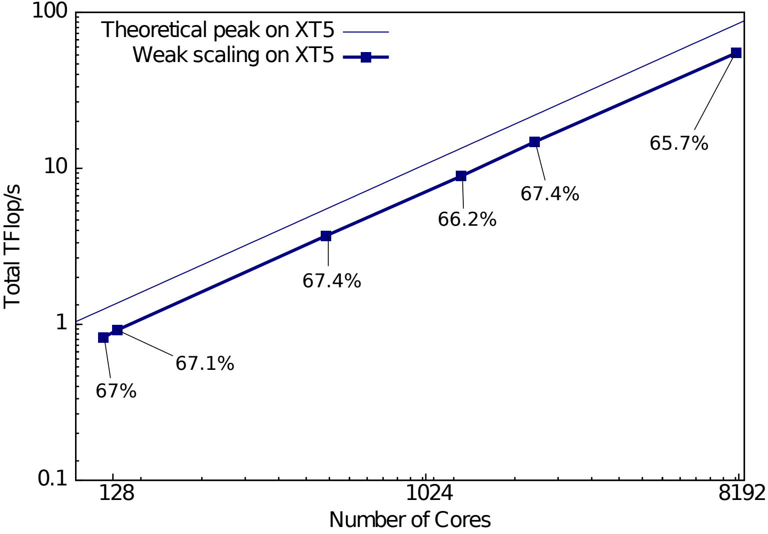
	Lines of Code			Module-specific Commits	
	CI	C++	Total		
Factorization	517	419	**936**	472/572	83%
Mem. Aware Sched.	9	492	501	86/125	69%
Mapping	10	72	82	29/42	69%

LU: Capabilities

- Flexible data placement
  - Experiment with data layout
- Memory-constrained adaptive lookahead

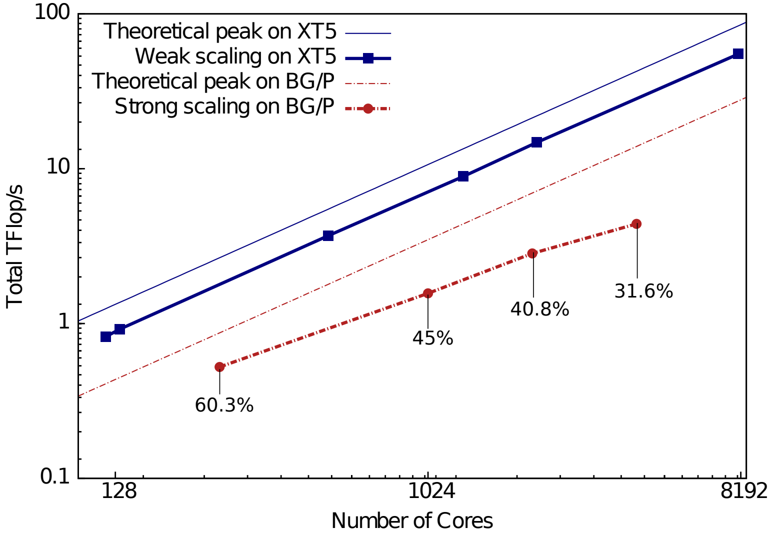
LU: Performance

Weak Scaling: (N such that matrix fills 75% memory)



LU: Performance

... and strong scaling too! (N=96,000)



Optional Benchmarks

Why MD, AMR and Sparse Triangular Solver

- Relevant scientific computing kernels
- Challenge the parallelization paradigm
  - Load imbalances
  - Dynamic communication structure
- Express non-trivial parallel control flow

LeanMD

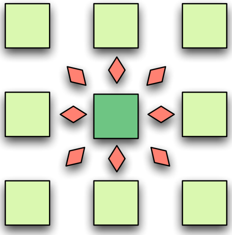
SLOC: 693

1. Mimics short-range force calculation in NAMD
2. Resembles *miniMD* of Mantevo project (SLOC≈3000)
3. Advanced features:

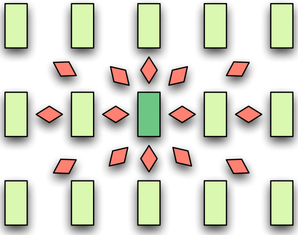
   Metabalancer: automated dynamic load balancing

   Fault tolerance: in-memory checkpointing based restart

   Split Execution: checkpoint on x cores, restart on y cores



1 Away Decomposition



2 AwayX Decomposition

Code for FT and LB

```
if (stepCount % ldbPeriod == 0) {
    serial { AtSync(); }
    when ResumeFromSync() { }
}
```

Code for FT and LB

```
if (stepCount % ldbPeriod == 0) {
    serial { AtSync(); }
    when ResumeFromSync() { }
}
```

```
if (stepCount % checkptFreq == 0) {
    serial {
        //coordinate to start checkpointing
        contribute(CkCallback(CkReductionTarget(Cell,startCheckpoint),thisProxy(0,0,0)));
    }
    if (thisIndex.x == 0 && thisIndex.y == 0 && thisIndex.z == 0) {
        when startCheckpoint() serial {
            CkCallback cb(CkReductionTarget(Cell,recvCheckPointDone), thisProxy);
            if (checkptStrategy == 0) CkStartCheckpoint(logs.c_str(), cb);
            else CkStartMemCheckpoint(cb);
        }
    }
    when recvCheckPointDone() { }
}
```

Code for FT and LB

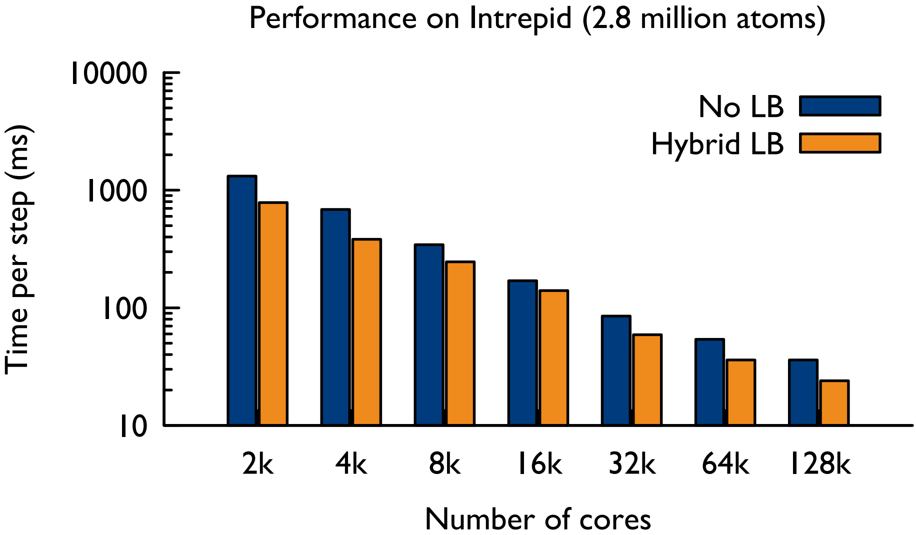
```
if (stepCount % ldbPeriod == 0) {
    serial { AtSync(); }
    when ResumeFromSync() { }
}
```

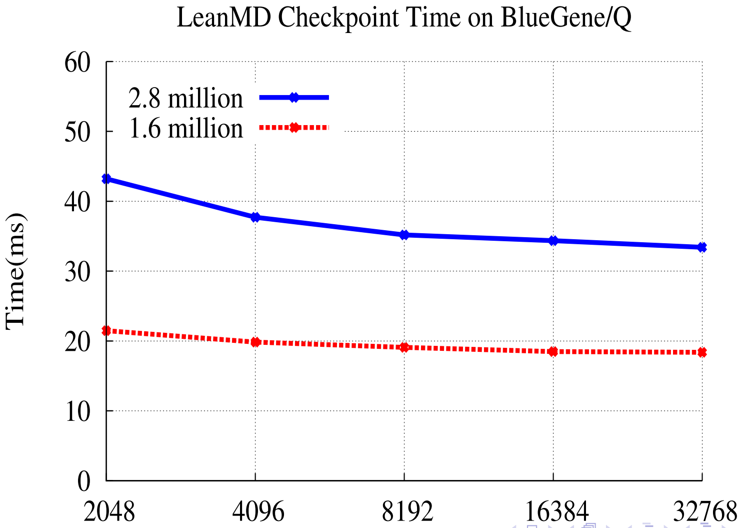
```
if (stepCount % checkptFreq == 0) {
    serial {
        //coordinate to start checkpointing
        contribute(CkCallback(CkReductionTarget(Cell,startCheckpoint),thisProxy(0,0,0)));
    }
    if (thisIndex.x == 0 && thisIndex.y == 0 && thisIndex.z == 0) {
        when startCheckpoint() serial {
            CkCallback cb(CkReductionTarget(Cell,recvCheckPointDone), thisProxy);
            if (checkptStrategy == 0) CkStartCheckpoint(logs.c_str(), cb);
            else CkStartMemCheckpoint(cb);
        }
    }
    when recvCheckPointDone() { }
}
```

```
//kill one of processes to demonstrate fault tolerance
if (stepCount == 30 && thisIndex.x == 1 && thisIndex.y == 1 && thisIndex.z == 0) serial {
    if (CkHasCheckpoints()) {
        CkDieNow();
    }
}
```

MD: Performance

1 million atoms. IBM Blue Gene/P (Intrepid)

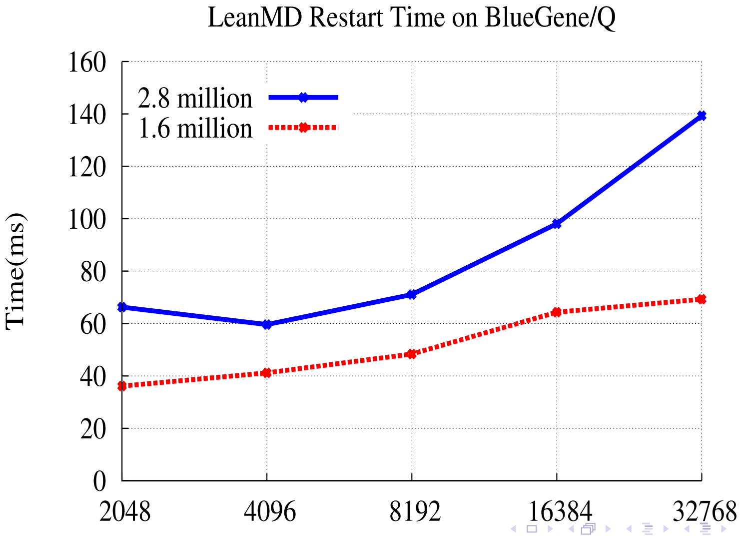




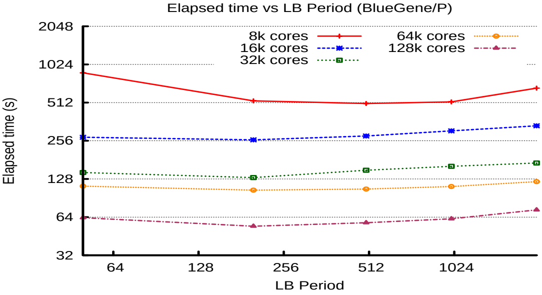
LeanMD Checkpoint Time on BlueGene/Q
Time(ms)
0
10
20
30
40
50
60
2048
4096
8192
16384
32768
2.8 million
1.6 million

Fault Tolerance Support for MD

MD: Restart Time



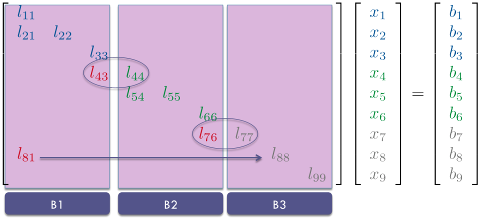
Meta-Balancer vs Periodic Load Balancing



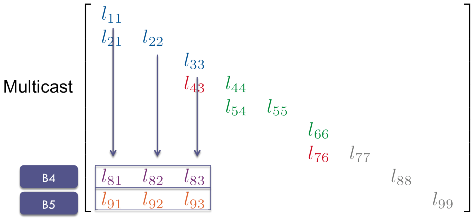
Cores	No LB (s)	Periodic LB (s)	Meta-Balancer (s)
8k	666	504	413
16k	336	260	277
32k	171	131	131
64k	122	104	100
128k	73	54	52

- Frequent load balancing increases total execution time
- Infrequent load balancing leads to load imbalance and results in no gains
- Meta-Balancer adaptively performs load balancing to obtain best total execution time

Sparse Triangular Solver- Matrix Decomposition



Column Decomposition



Dense Parts Decomposition

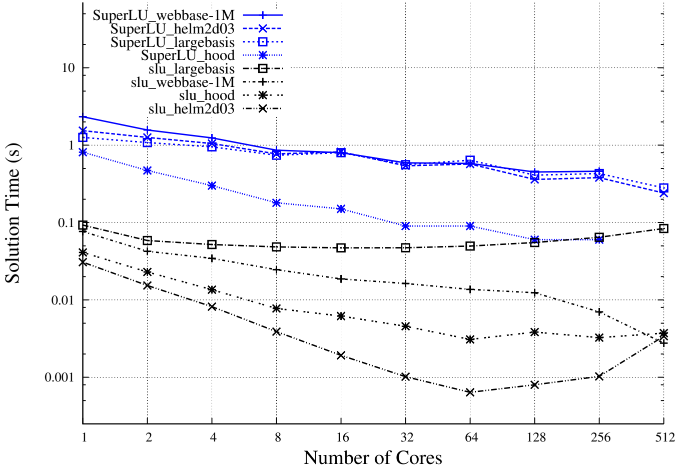
Sparse Triangular Solver- Parallel Algorithm

- Schedule independent computation
- High priority: Process messages
- Low priority: do independent computation

- Highest priority: receive solution values
- High priority: Process messages

```
if (onDiagonalChare) {
  serial { thisProxy[thisIndex].indepCompute(...) }
  overlap {
    while (!finished) {
      when recvData(int len, double data[len], int rows[len])
        serial {if(len>0) diagReceiveData(len, data, rows);}
    }
    when indepCompute(int a) serial {myIndepCompute();}
  }
} else {
  when getXvals(xValMsg* msg) serial {nondiag_compute();}
  while (!finished) {
    when recvData(int len, double data[len], int rows[len])
    serial {nondiagReceiveData(len, data, rows);}
  }
}
```

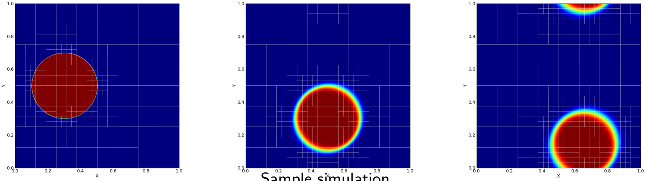
Sparse Triangular Solver- Performance vs. SuperLU_DIST



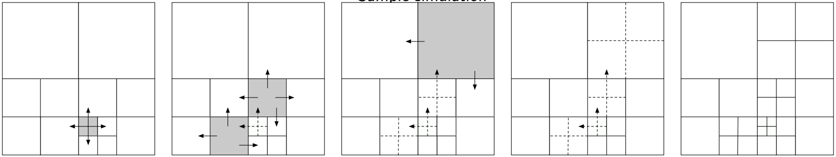
Sparse Triangular Solver- Productivity in Charm++

- More complicated (higher performance) algorithm in 692 total SLOCs
  - vs. 897 SLOCs of SuperLU_DIST triangular solver
- Overdecomposition (with Round-Robin mapping) is essential
  - Communication computation overlap, load balance
- Creation of parallel units dynamically
  - Distributing dense regions
- Message-driven nature and priorities
  - No need for something like MPI_Iprobe

Adaptive Mesh Refinement

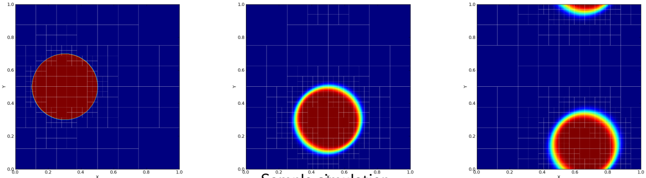


Sample simulation

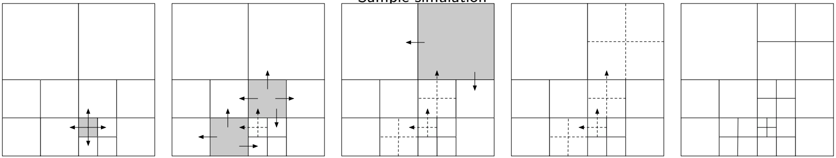


Propagation of refinement decision messages

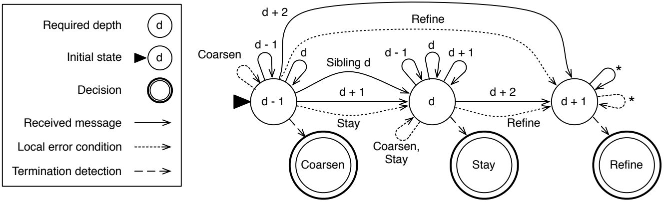
Adaptive Mesh Refinement



Sample simulation



Propagation of refinement decision messages



Finite state machine for each block's decision update

Adaptive Mesh Refinement

Charm++ Implementation

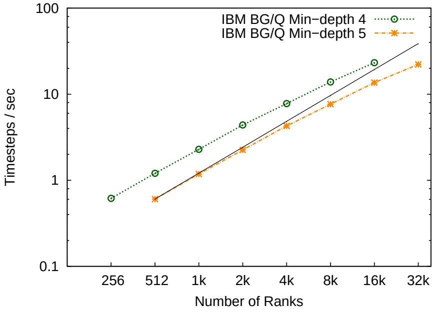
- Blocks as virtual processors instead of each process containing many blocks
  - simplifies implementation
- Blocks addressed with bit vector indices
  - CharmRTS handles physical locations
- Dynamic distributed load balancing

Algorithmic Improvements[1]

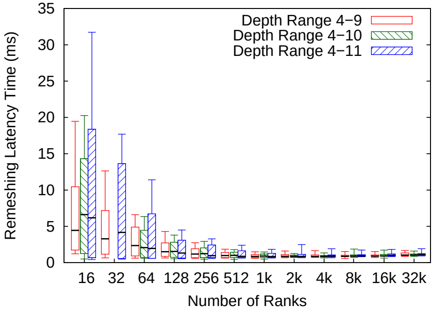
- $O(\#blocks/P)$ **vs** $O(\#blocks)$ memory per process
- 2 system quiesence states **vs** $\#level$ reductions for mesh restructuring
- $O(1)$ **vs** $O(\log P)$ time neighbor lookup

[1] Langer, et.al. Scalable Algorithms for Distributed Memory Adaptive Mesh Refinement. In 24th International Conference on Computer Architecture and High Performance Computing (SBAC-PAD) 2012, NY, USA.

Adaptive Mesh Refinement



Timesteps per second strong scaling on IBM BG/Q with a max depth of 15.



The non-overlapped delay of remeshing in milliseconds on IBM BG/Q. The candlestick graphs the minimum and maximum values, the 5th and 95th percentile, and the median.

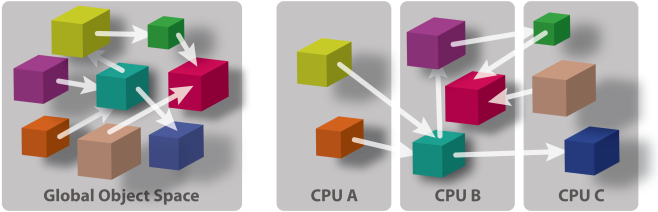
Charm++ at SC12

Temperature-aware load balancing Tue @ 2:00 pm

NAMD at 200K+ cores Thu @ 11:00 am

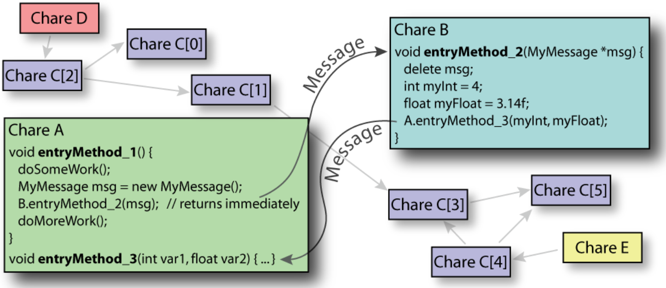
For more info

http://charm.cs.illinois.edu/



Object-based Express logic via indexed collections of interacting objects (both data *and* tasks)

Over-decomposed Expose more parallelism than available processors



Runtime-Assisted	scheduling, observation-based adaptivity, load balancing, composition, etc.
Message-Driven	Trigger computation by invoking remote *entry* methods
Non-blocking, Asynchronous	Implicitly overlapped data transfer

- Regular C++ code
  - No special compilers

- Regular C++ code
  - No special compilers
- Small parallel interface description file
  - Can contain control flow DAG
  - Parsed to generate more C++ code

- Regular C++ code
  - No special compilers
- Small parallel interface description file
  - Can contain control flow DAG
  - Parsed to generate more C++ code
- Inherit from framework classes to
  - Communicate with remote objects
  - Serialize objects for transmission
- Exploit modern C++ program design techniques (OO, generics etc)

TRAM: Message Routing and Aggregation