Performance Analysis of Lattice QCD Application with APGAS Programming Model

Koichi Shirahata[1], Jun Doi[2], Mikio Takeuchi[2]

1: Tokyo Institute of Technology

2: IBM Research - Tokyo

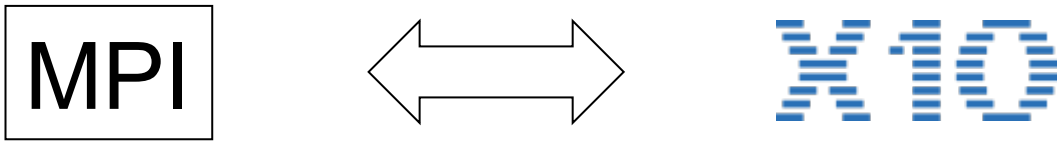
Programming Models for Exascale Computing

- Extremely parallel supercomputers
  - It is expected that the first exascale supercomputer will be deployed by 2020
  - Which programming model will allow easy development and high performance is still unknown
- Programming models for extremely parallel supercomputers
  - Partitioned Global Address Space (PGAS)
    - Global view of distributed memory
    - Asynchronous PGAS (APGAS)

→ Highly Scalable and Productive Computing using APGAS Programming Model

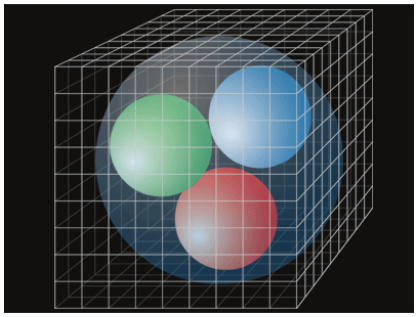
Problem Statement

- How is the performance of APGAS programming model compared with existing massage passing model?
  - Message Passing (MPI)
    - Good tuning efficiency
    - High programming complexity
  - Asynchronous Partitioned Global Address Space (APGAS)
    - High programming productivity, Good scalability
    - Limited tuning efficiency



Approach

- Performance analysis of lattice QCD application with APGAS programming model
  - Lattice QCD
    - one of the most challenging application for supercomputers
  - Implement lattice QCD in X10
    - Port C++ lattice QCD to X10
    - Parallelize using APGAS programming model
  - Performance analysis of lattice QCD in X10
    - Analyze parallel efficiency of X10
    - Compare the performance of X10 with MPI



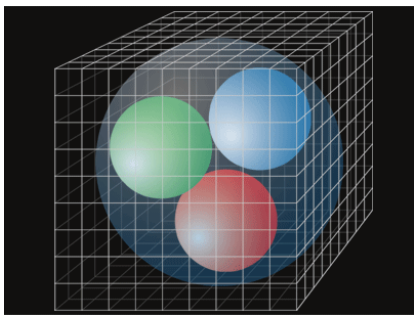
Goal and Contributions

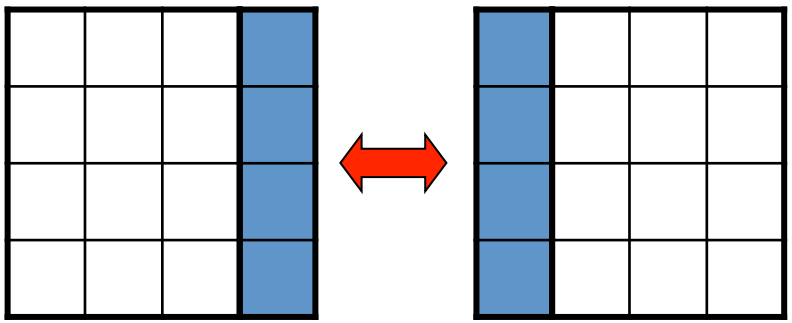
- Goal
  - Highly scalable computing using APGAS programming model
- Contributions
  - Implementation of lattice QCD application in X10
    - Several optimizations on lattice QCD in X10
  - Detailed performance analysis on lattice QCD in X10
    - 102.8x speedup in strong scaling
    - MPI performs 2.26x – 2.58x faster, due to the limited communication overlapping in X10

Table of Contents

Lattice QCD

- Lattice QCD
  - Common technique to simulate a field theory (e.g. Big Bang) of Quantum ChromoDynamics (QCD) of quarks and gluons on 4D grid of points in space and time
  - A grand challenge in high-performance computing
    - Requires high memory/network bandwidth and computational power
- Computing lattice QCD
  - Monte-Carlo simulations on 4D grid
  - Dominated by solving a system of linear equations of matrix-vector multiplication using iterative methods (etc. CG method)
  - Parallelizable by dividing 4D grid into partial grids for each place
    - Boundary exchanges are required between places in each direction



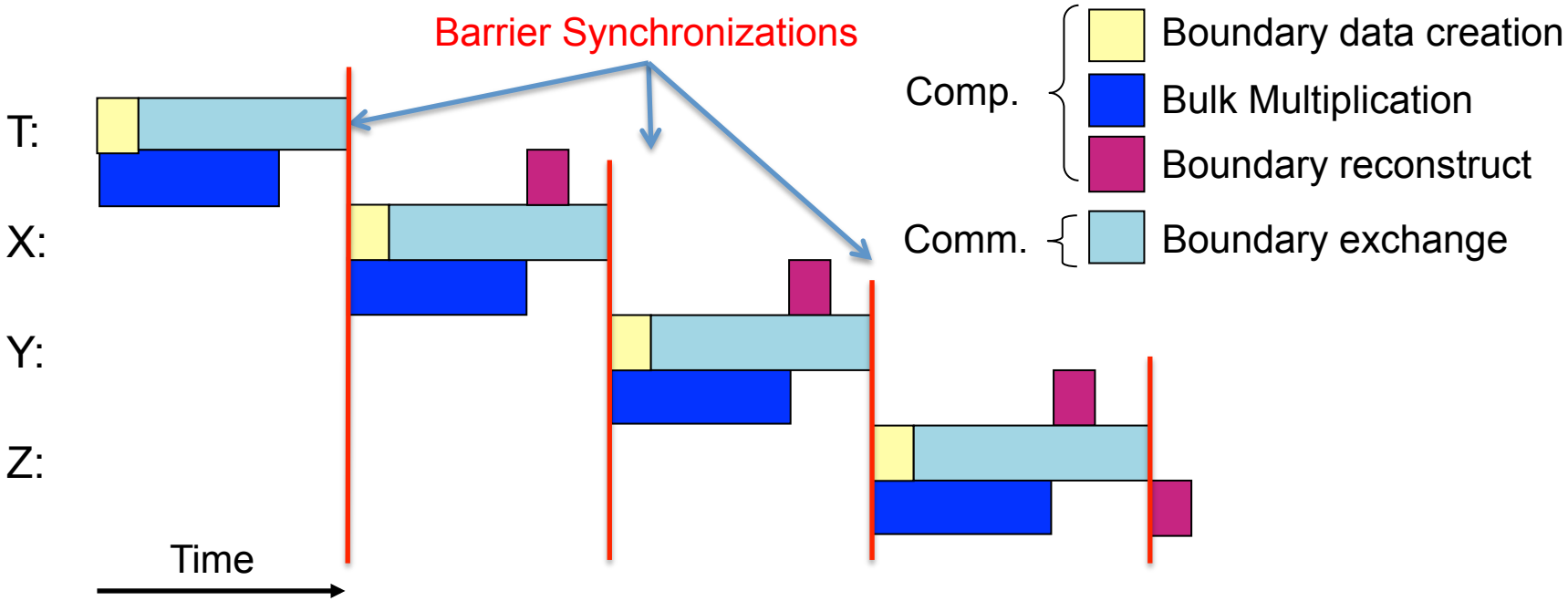


Implementation of lattice QCD in X10

- Fully ported from sequential C++ implementation
- Data structure
  - Use Rail class (1D array) for storing 4D arrays of quarks and gluons
- Parallelization
  - Partition 4D grid into places
    - Calculate memory offsets on each place at the initialization
    - Boundary exchanges using asynchronous copy function
- Optimizations
  - Communication optimizations
    - Overlap boundary exchange and bulk computations
  - Hybrid parallelization
    - Places and threads

Communication Optimizations

- Communication overlapping by using “asyncCopy” function
  - “asyncCopy” creates a new thread then copy asynchronously
  - Wait completion of “asyncCopy” by “finish” syntax
- Communication through Put-wise operations
  - Put-wise communication uses one-sided communication while Get-wise communication uses two-sided communication
- Communication is not fully overlapped in the current implementation
  - “finish” requires all the places to synchronize



Hybrid Parallelization

- Hybrid parallelization on places and threads (activities)
- Parallelization strategies for places
  - (1) Activate places for each parallelizable part of computation
  - (2) Barrier-based synchronization
    - Call “finish” for places at the beginning of CG iteration

  ⇨ We adopt (2) since calling “finish” for each parallelizable part of computation causes increase of synchronization overheads
- Parallelization strategies for threads
  - (1) Activate threads for each parallelizable part of computation
  - (2) Clock-based synchronization
    - Call “finish” for threads at the beginning of CG iteration

  ⇨ We adopt (1) since we observed “finish” performs better scalability compared to the clock-based synchronization

Table of Contents

Evaluation

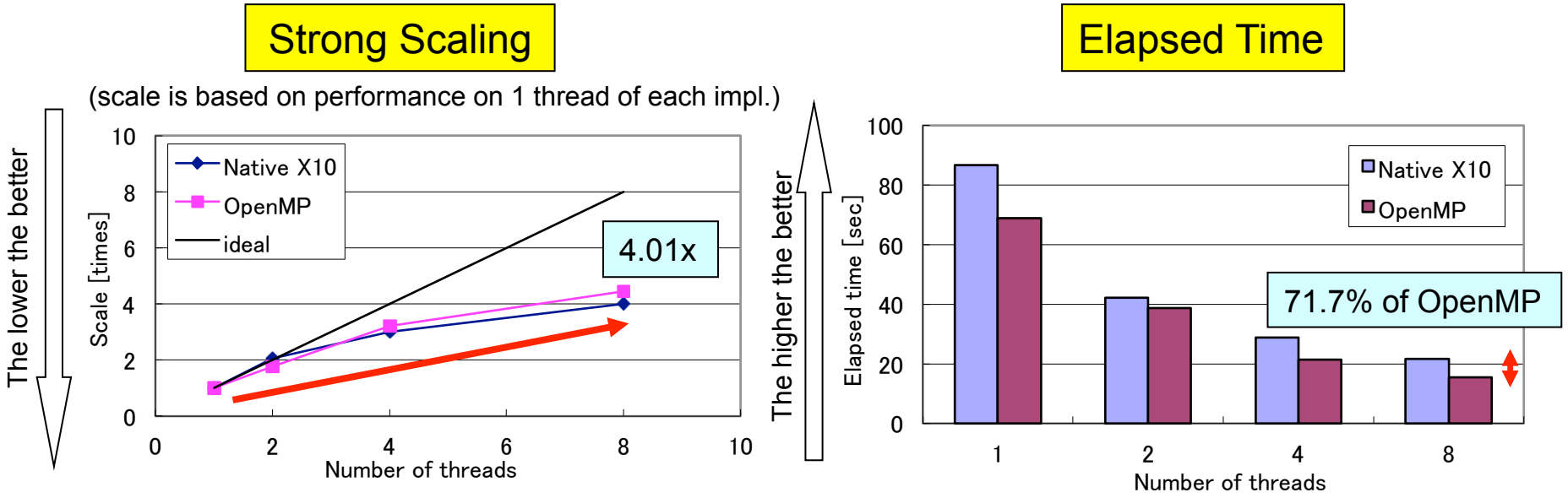
- Objective
  - Analyze parallel efficiency of our lattice QCD in X10
  - Comparison with lattice QCD in MPI
- Measurements
  - Effect of multi-threading
    - Comparison of multi-threaded X10 with OpenMP on a single node
    - Comparison of hybrid parallelization with MPI+OpenMP
  - Scalability on multiple nodes
    - Comparison of our distributed X10 implementation with MPI
    - Measure strong/weak scaling up to 256 places
- Configuration
  - Measure elapsed time of one convergence of CG method
    - Typically 300 to 500 CG iterations
  - Compare native X10 (C++) and MPI C

Experimental Environments

- IBM BladeCenter HS23 (Use 1 node for multi-threaded performance)
  - CPU: Xeon E5 2680 (2.70GHz, L1=32KB, L2=256KB, L3=20MB, 8 cores) x2 sockets, SMT enabled
  - Memory: 32 GB
  - MPI: MPICH2 1.2.1
  - g++: v4.4.6
  - X10: 2.4.0 trunk r25972 (built with "-Doptimize=true -DNO_CHECKS=true")
  - Compile option
    - Native X10: -x10rt mpi -O -NO_CHECKS
    - MPI C: -O2 -finline-functions –fopenmp
- IBM Power 775 (Use up to 13 nodes for scalability study)
  - CPU: Power7 (3.84 GHz, 32 cores), SMT Enabled
  - Memory: 128 GB
  - xlC_r: v12.1
  - X10: 2.4.0 trunk r26346 (built with "-Doptimize=true -DNO_CHECKS=true")
  - Compile option
    - Native X10: -x10rt pami -O -NO_CHECKS
    - MPI C: -O3 –qsmp=omp

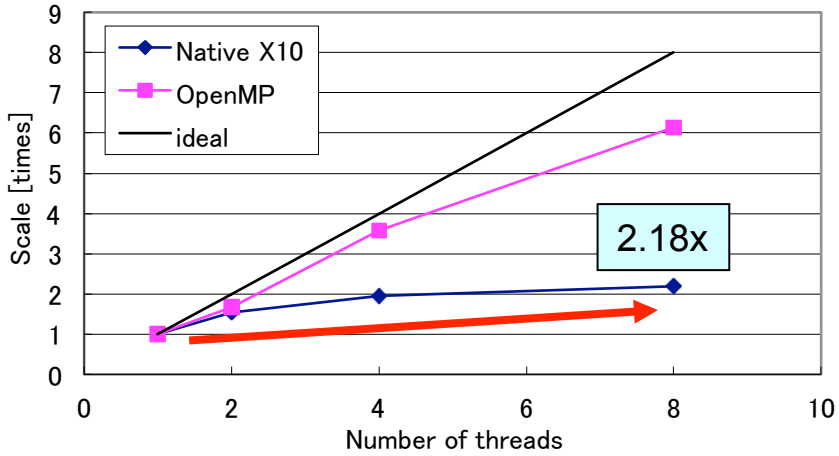
Performance on Single Place

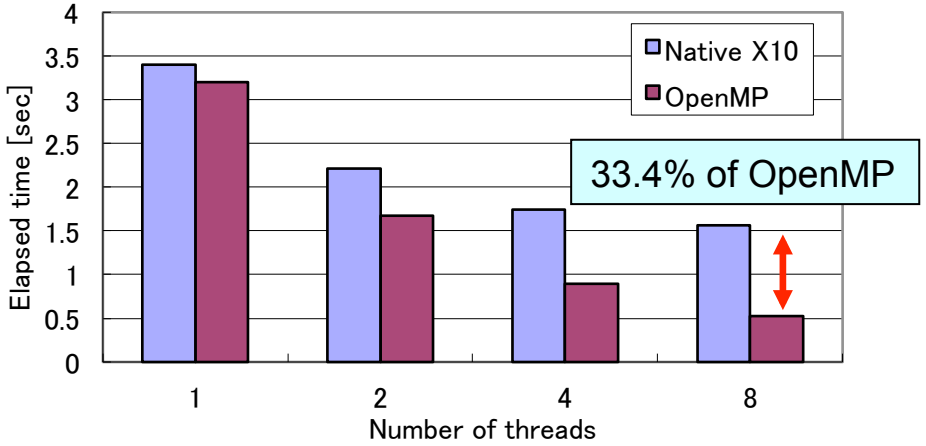
- Multi-thread parallelization (on 1 Place)
  - Create multiple threads (activities) for each parallelizable part of computation
  - Problem size: (x, y, z, t) = (16, 16, 16, 32)
- Results
  - Native X10 with 8 threads exhibits 4.01x speedup over 1 thread
  - Performance of X10 is 71.7% of OpenMP on 8 threads
  - Comparable scalability with OpenMP



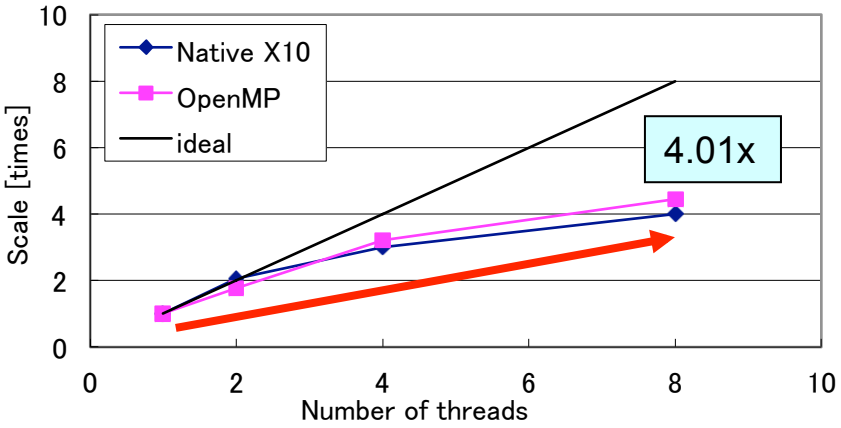
Performance on Difference Problem Sizes

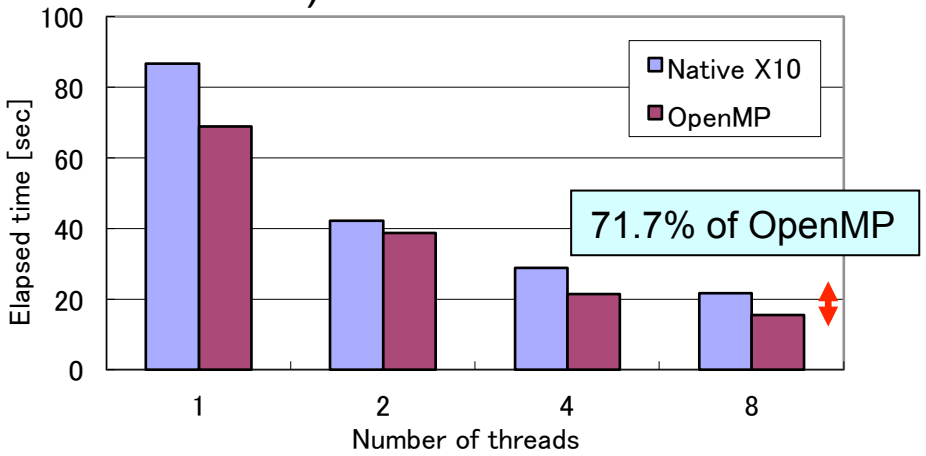
- Performance on (x,y,z,t) = (8,8,8,16)
  - Poor scalability on Native X10 (2.18x on 8 threads)





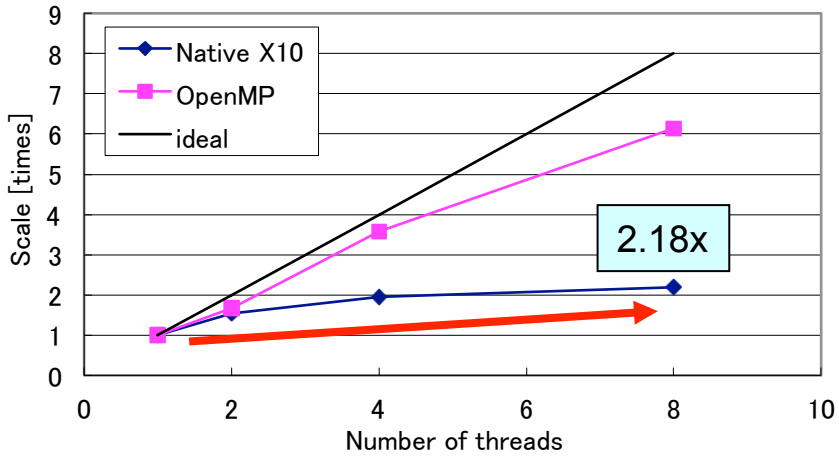
- Performance on (x,y,z,t) = (16,16,16,32)
  - Good scalability on Native X10 (4.01x on 8 threads)

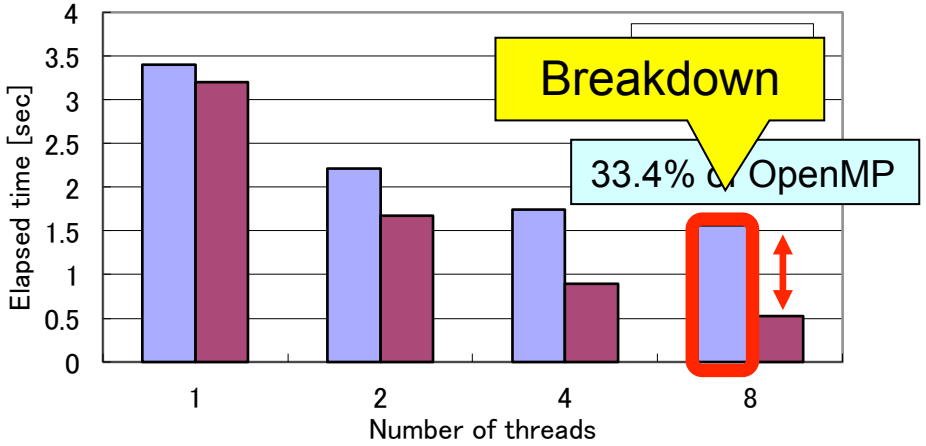




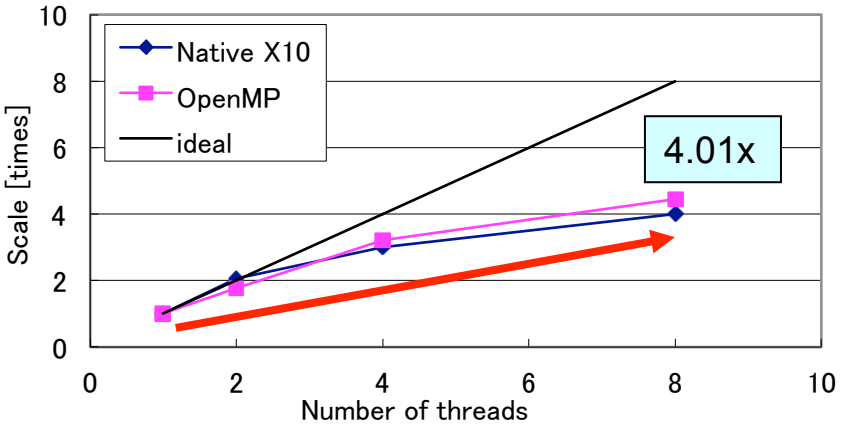
Performance on Difference Problem Sizes

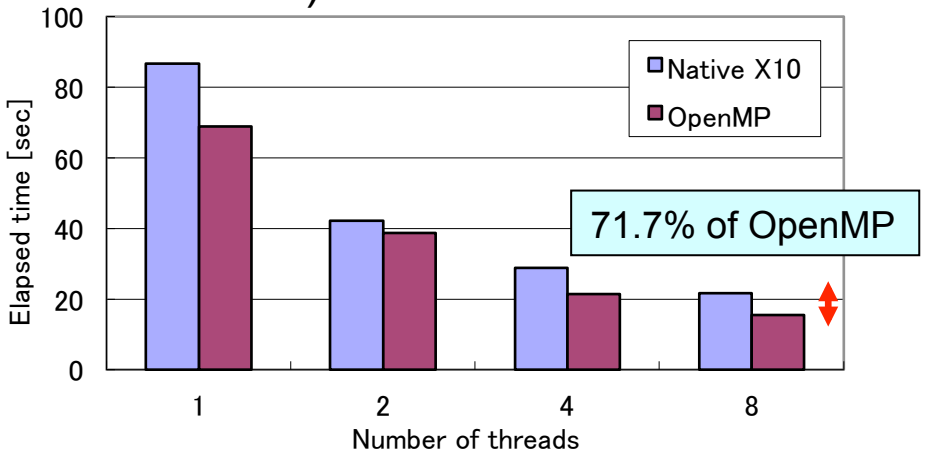
- Performance on (x,y,z,t) = (8,8,8,16)
  - Poor scalability on Native X10 (2.18x on 8 threads)





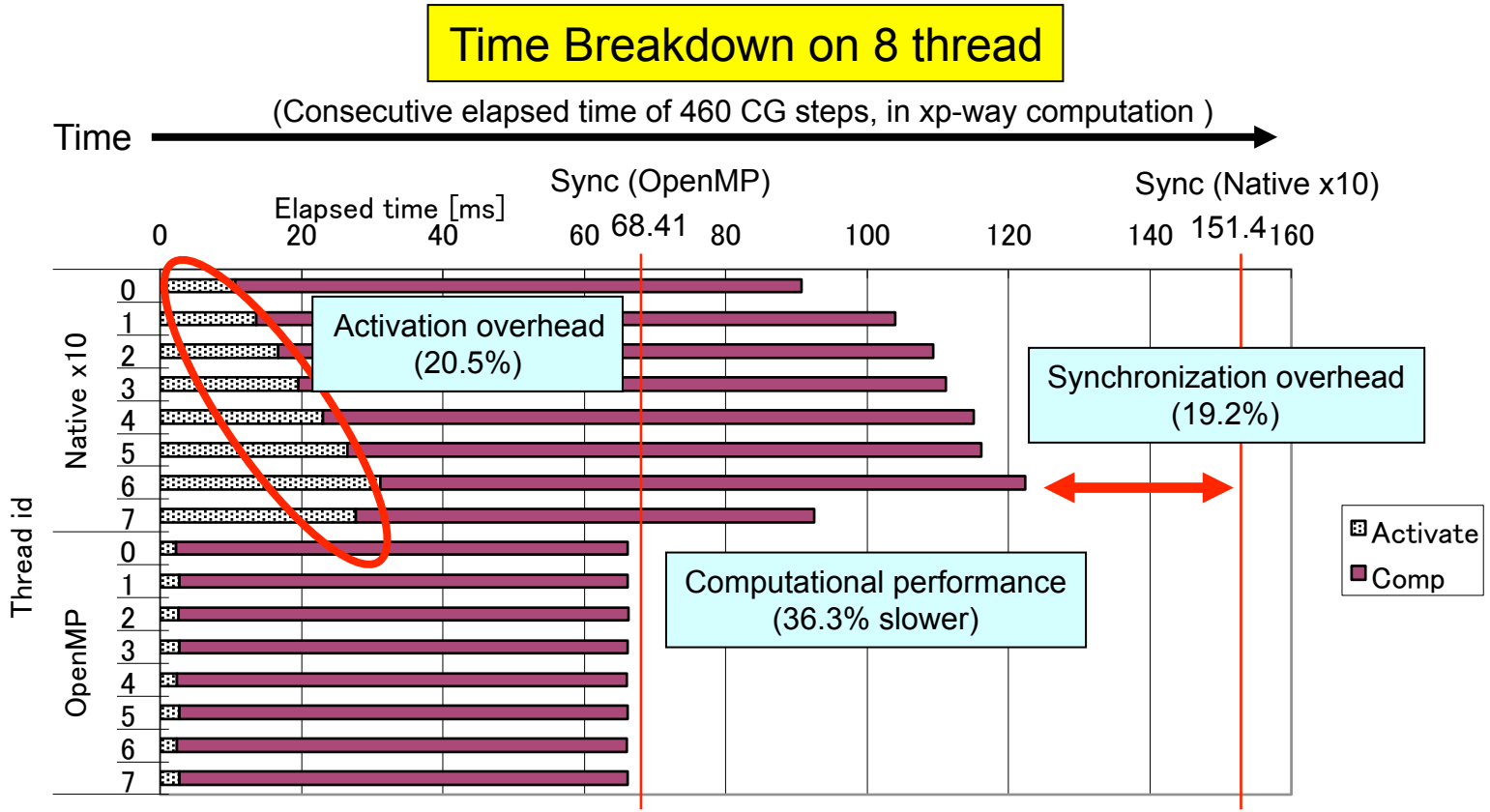
- Performance on (x,y,z,t) = (16,16,16,32)
  - Good scalability on Native X10 (4.01x on 8 threads)





Performance Breakdown on Single Place

- Running on Native X10 with multi-threads suffers from significant overhead of
  - Thread activations (20.5% overhead on 8 threads)
  - Thread synchronizations (19.2% overhead on 8 threads)
  - Computation is also slower than that on OpenMP (36.3% slower)



Comparison of Hybrid Parallelization with MPI + OpenMP

- Hybrid Parallelization
  - Comparison with MPI
- Measurement
  - Use 1 node (2 sockets of 8 cores, SMT enabled)
  - Vary # Processes and # Threads s.t. (#Processes) x (# Threads) is constant
- Best performance when (# Processes, # Threads) = (4, 4) in MPI, and (16, 2) in X10
  - 2 threads per node exhibits best performance
  - 1 thread per node also exhibits similar performance as 2 threads

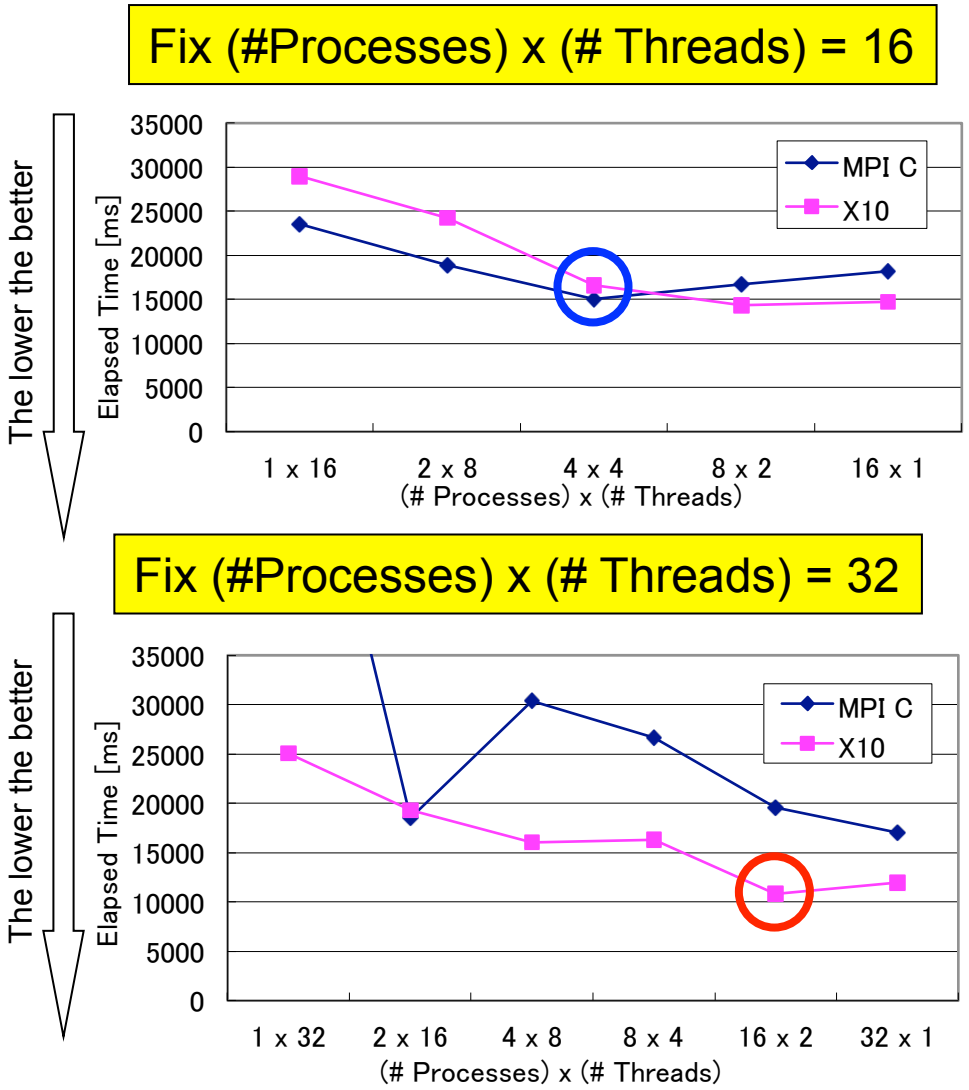
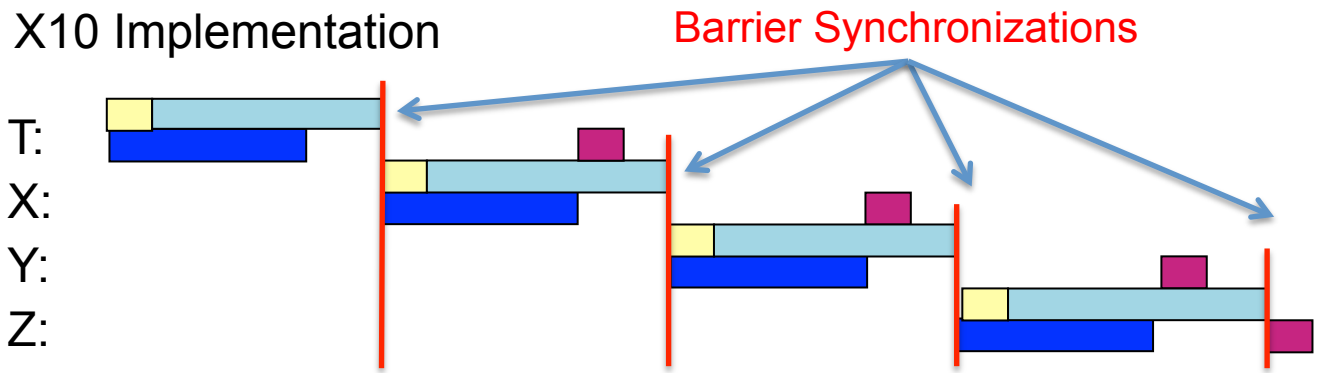


Table of Contents

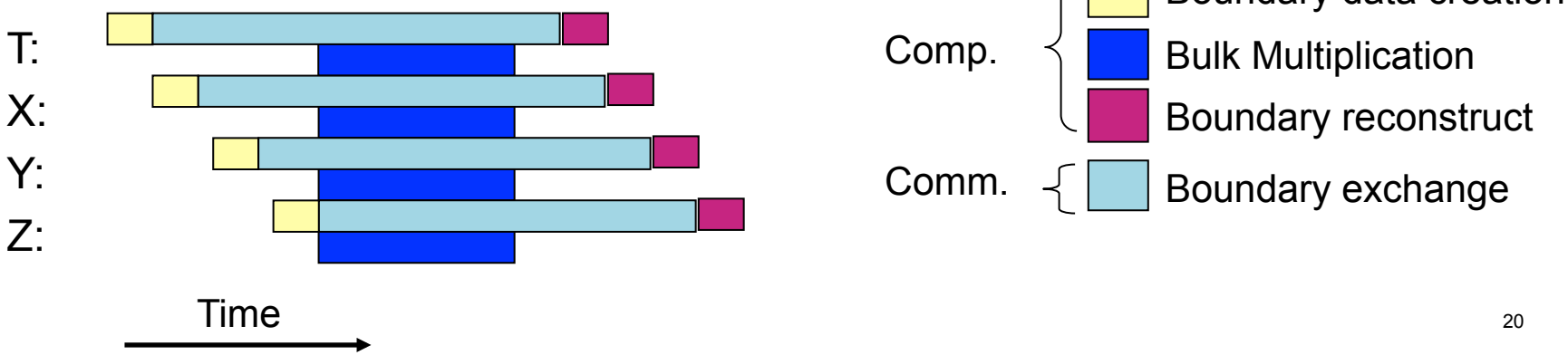
Comparison with MPI

- Compare the performance of X10 Lattice QCD with our MPI-based lattice QCD
  - Point-to-point communication using MPI_Isend/Irecv (no barrier synchronizations)

▪ X10 Implementation



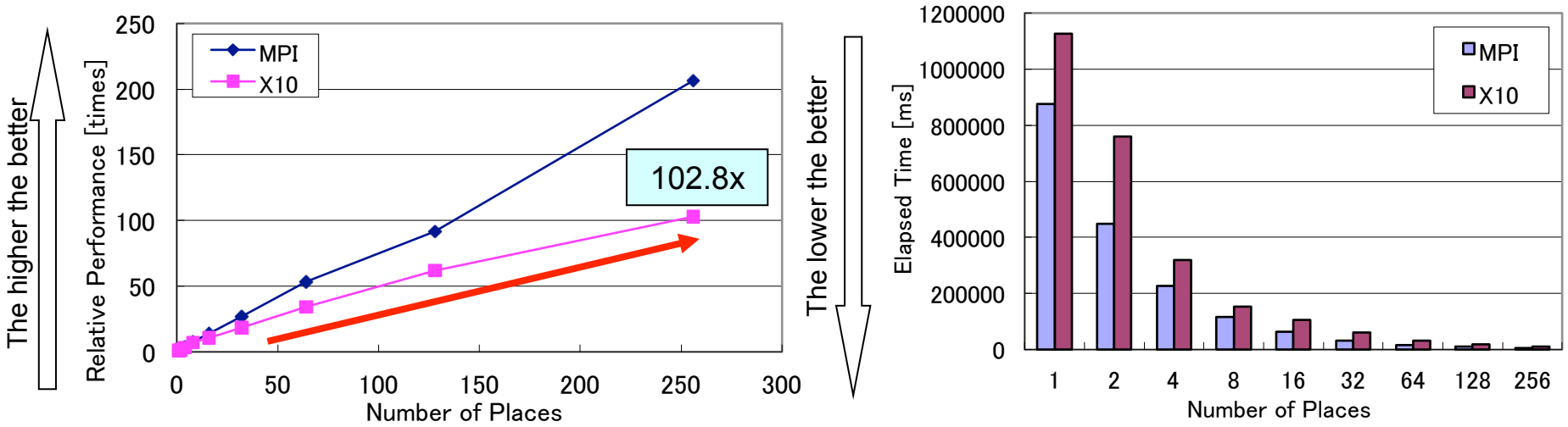
▪ MPI Implementation



Strong Scaling: Comparison with MPI

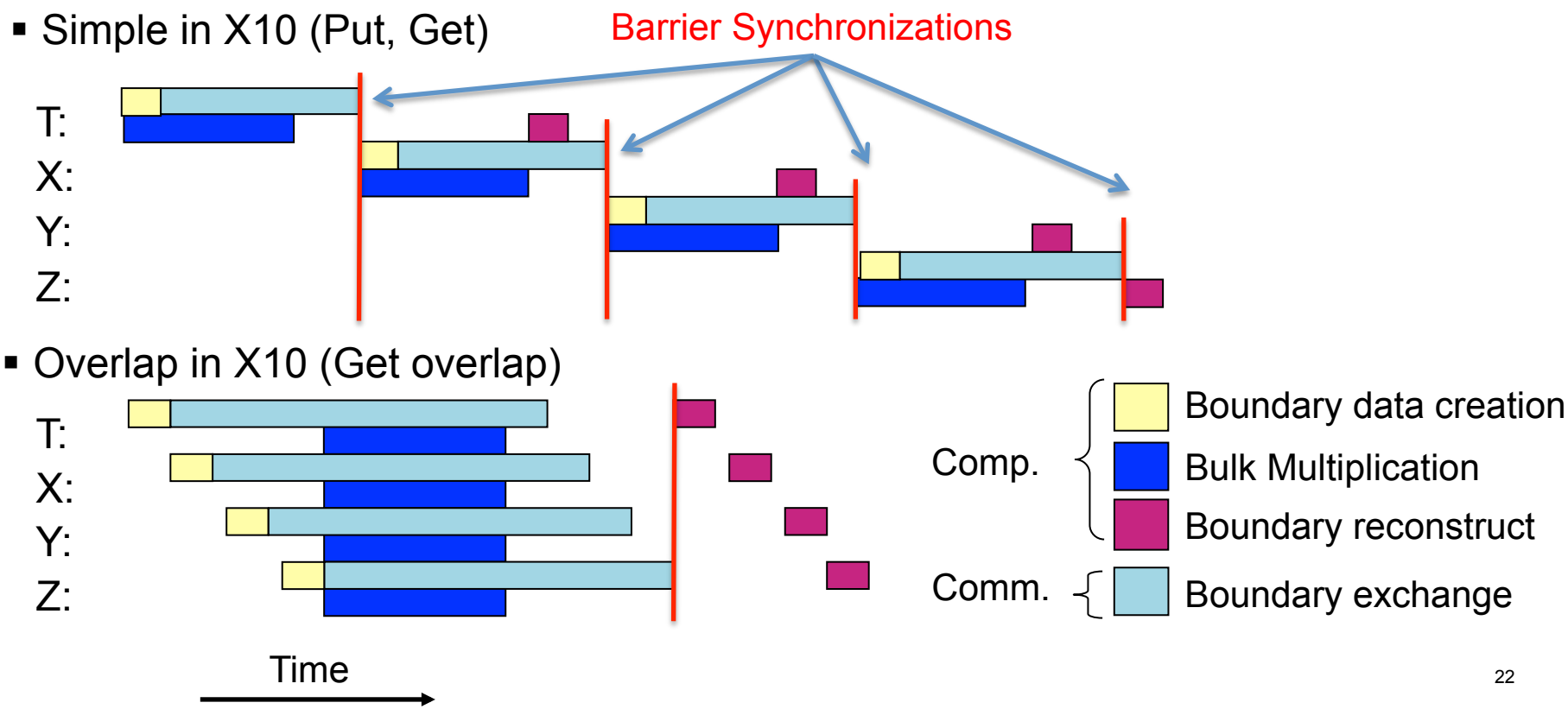
- Measurement on IBM Power 775 (using up to 13 nodes)
  - Increase #Places up to 256 places (19-20 places / node)
    - Not using hybrid parallelization
  - Problem size: (x, y, z, t) = (32, 32, 32, 64)
- Results
  - 102.8x speedup on 256 places compared to on 1 place
  - MPI exhibits better scalability
    - 2.58x faster on 256 places compared to X10

Strong Scaling Problem size: (x, y, z, t) = (32, 32, 32, 64) Elapsed Time



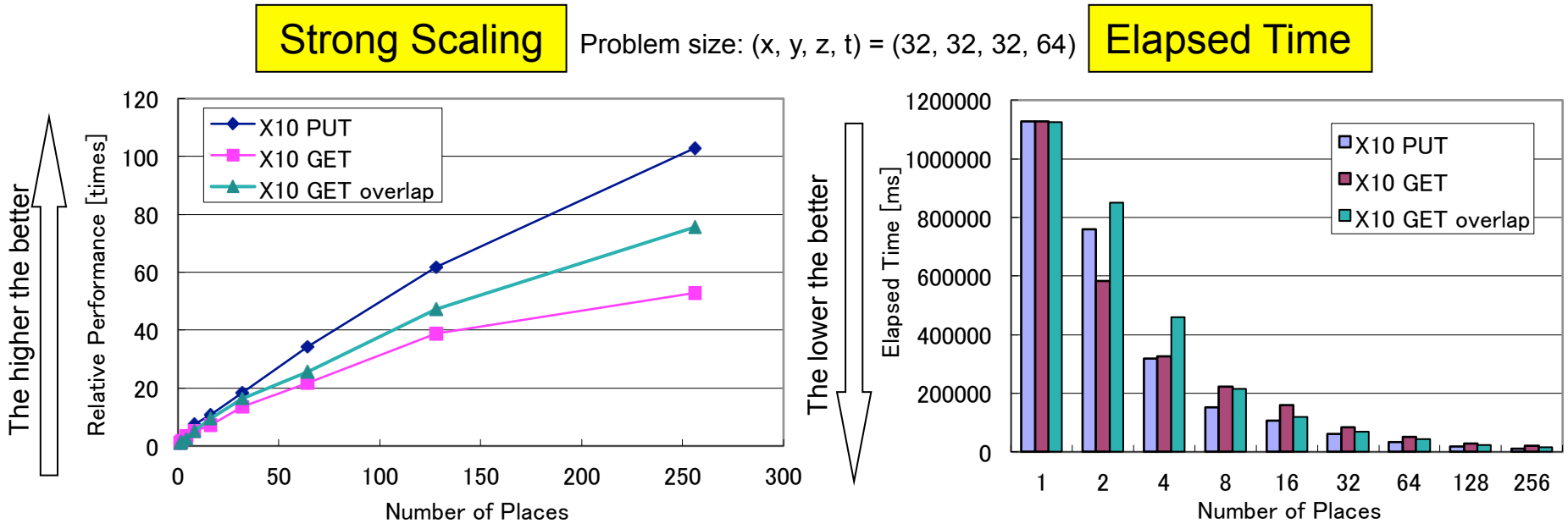
Effect of Communication Optimization

- Comparison of Put-wise and Get-wise communications
  - Put: “at” to source place, then copy data to destination place
  - Get: “at” to destination place, then copy data from source place
  - Apply communication overlapping (in Get-wise communication)
    - Multiple copies in a finish



Results: Effect of Communication Optimization

- Comparison of PUT with GET in communication
  - PUT performs better strong scaling
    - Underlying PUT implementation in PAMI uses one-sided communication while GET implementation uses two-sided communication



Weak Scaling: Comparison with MPI

- Measurement on IBM Power 775
  - Increase #Places up to 256 places (19-20 places / node)
  - Problem size per Place: 131072 ((x, y, z, t) = (16, 16, 16, 32))
- Results
  - 97.5x speedup on 256 places
  - MPI exhibits better scalability
    - 2.26x on 256 places compares with X10
    - MPI implementation performs more overlapping of communication

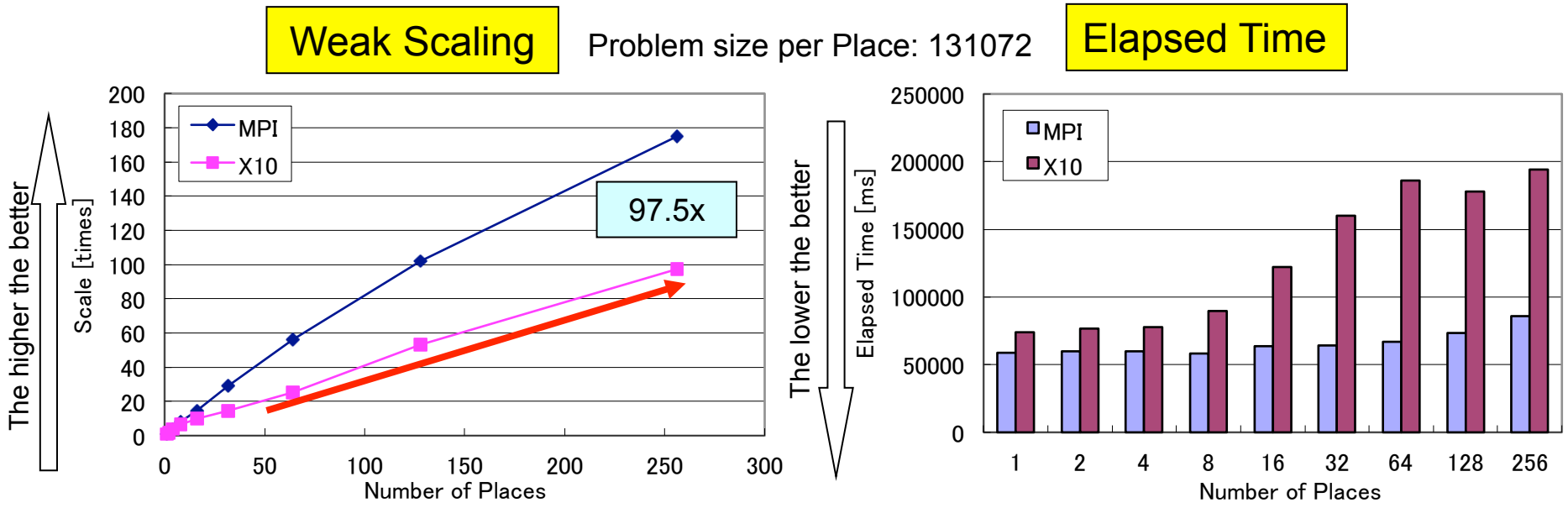


Table of Contents

Related work

- Peta-scale lattice QCD on a Blue Gene/Q supercomputer [1]
  - Fully overlapping communication and applying node-mapping optimization for BG/Q
- Performance comparison of PGAS with MPI [2]
  - Compare the performance of Co-Array Fortran (CAF) with MPI on micro benchmarks
- Hybrid programming model of PGAS and MPI [3]
  - Hybrid programming of Unified Parallel C (UPC) and MPI, which allows MPI programmers incremental access of a greater amount of memory by aggregating the memory of several nodes into a global address space

[1]: Doi, J.: Peta-scale Lattice Quantum Chromodynamics on a Blue Gene/Q supercomputer
[2]: Shan, H. et al.:A preliminary evaluation of the hardware acceleration of the Cray Gemini Interconnect for PGAS languages and comparison with MPI
[3]: Dinan, J. et al: Hybrid parallel programming with MPI and unified parallel C

Conclusion

- Summary
  - Towards highly scalable computing with APGAS programming model
  - Implementation of lattice QCD application in X10
    - Include several optimizations
  - Detailed performance analysis on lattice QCD in X10
    - 102.8x speedup in strong scaling, 97.5x speedup in weak scaling
    - MPI performs 2.26x – 2.58x faster, due to the limited communication overlapping in X10

- Future work
  - Further optimizations for lattice QCD in X10
    - Further overlapping by using point-to-point synchronizations
    - Accelerates computational parts using GPUs
  - Performance analysis on supercomputers
    - IBM BG/Q, TSUBAME 2.5

Source Code of Lattice QCD in X10 and MPI

- Available from the following URL
  - https://svn.code.sourceforge.net/p/x10/code/applications/trunk/LatticeQCD/

Backup

APGAS Programming Model

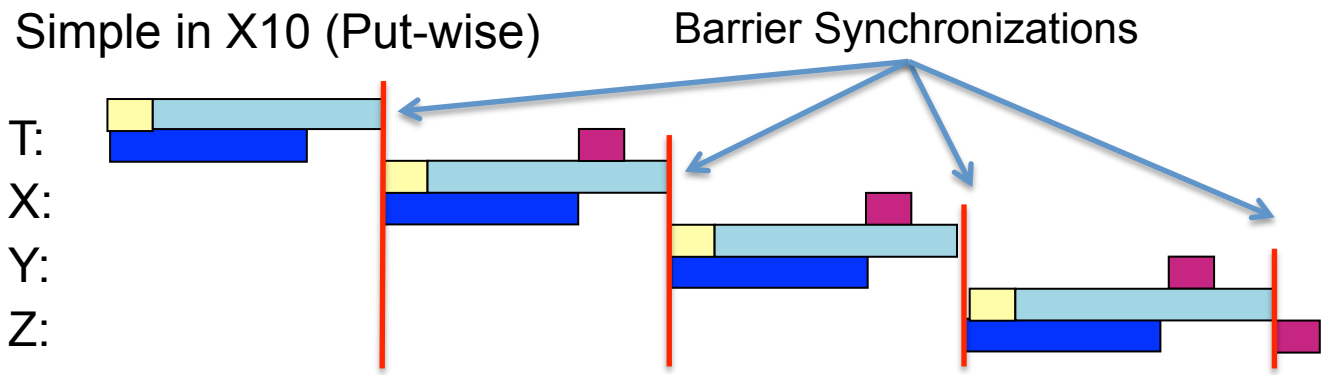
- PGAS (Partitioned Global Address Space) programming model
  - Global Address Space
    - Every thread sees entire data set
  - Partitioned
    - Global address space is divided to multiple memories
- APGAS programming model
  - Asynchronous PGAS (APGAS)
    - Threads can be dynamically created under programmer control
  - X10 is a language implementing APGAS model
    - New activity (thread) is created dynamically using “async” syntax
    - PGAS memories are called Places (Processes)
      - Move to other memory using “at” syntax
    - Threads and places are synchronized by calling “finish” syntax

Existing Implementation of Lattice QCD in MPI

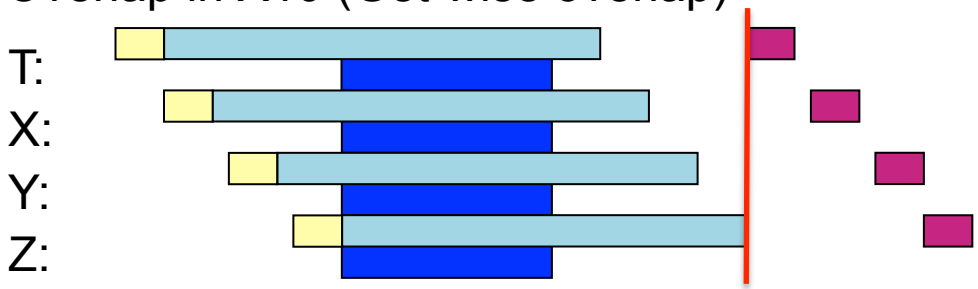
- Partition 4D grid into MPI processes
  - Boundary data creation => Boundary exchange => Bulk update
- Computation and communication overlap
  - Overlap Boundary exchange and Boundary/ Bulk update
- Hybrid parallelization
  - MPI + OpenMP

Effect of Communication Optimization

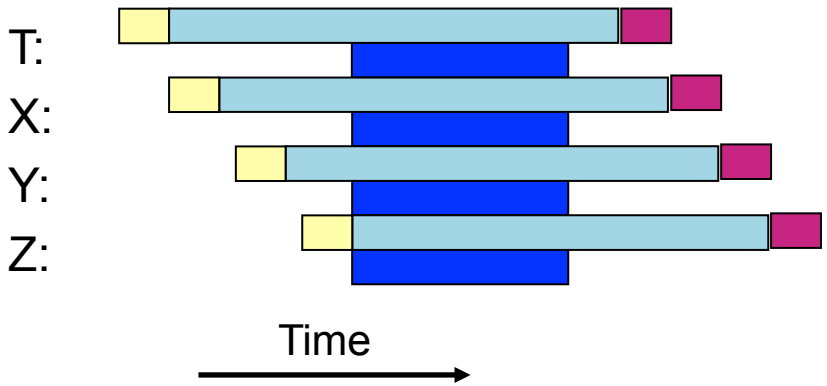
- Simple in X10 (Put-wise)



- Overlap in X10 (Get-wise overlap)

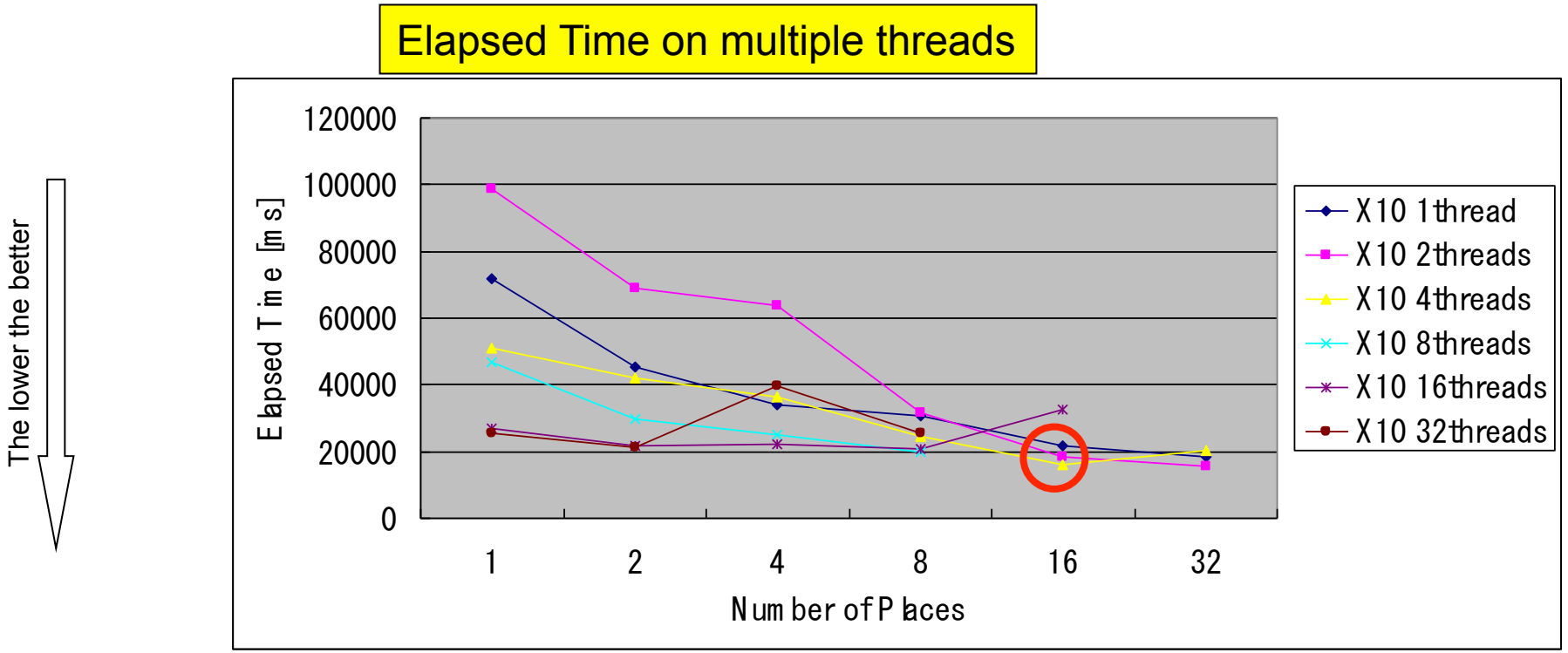


- Overlap in MPI



Hybrid Performance on Multiple Nodes

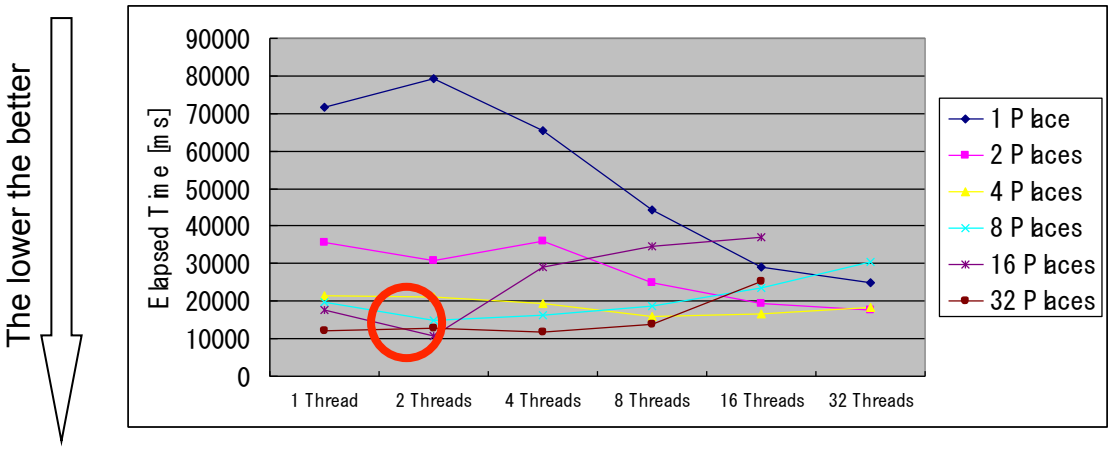
- Hybrid Parallelization on multiple nodes
- Measurement
  - Use up to 4 nodes
  - (# Places, # Threads) = (32, 2) shows best performance
    - (# Places, # Threads) = (8, 2) per node



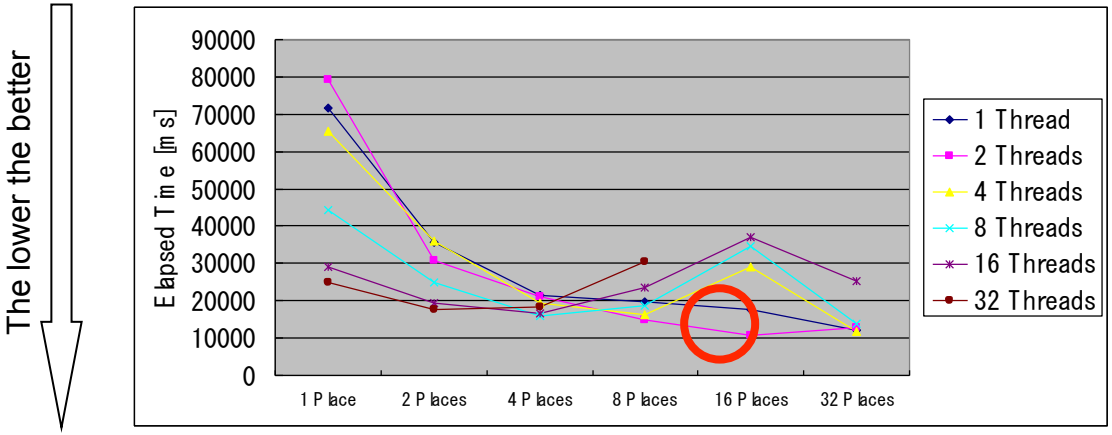
Performance of Hybrid Parallelization on Single Node

- Hybrid Parallelization
  - Places and Threads
- Measurement
  - Use 1 node (2 sockets of 8 cores, HT enabled)
  - Vary # Places and # Threads from 1 to 32 for each
- Best performance when (# Places, # Threads) = (16, 2)
- Best balance when (# Places) x (# Threads) = 32

Thread Scalability (Elapsed Time)



Place Scalability (Elapsed Time)

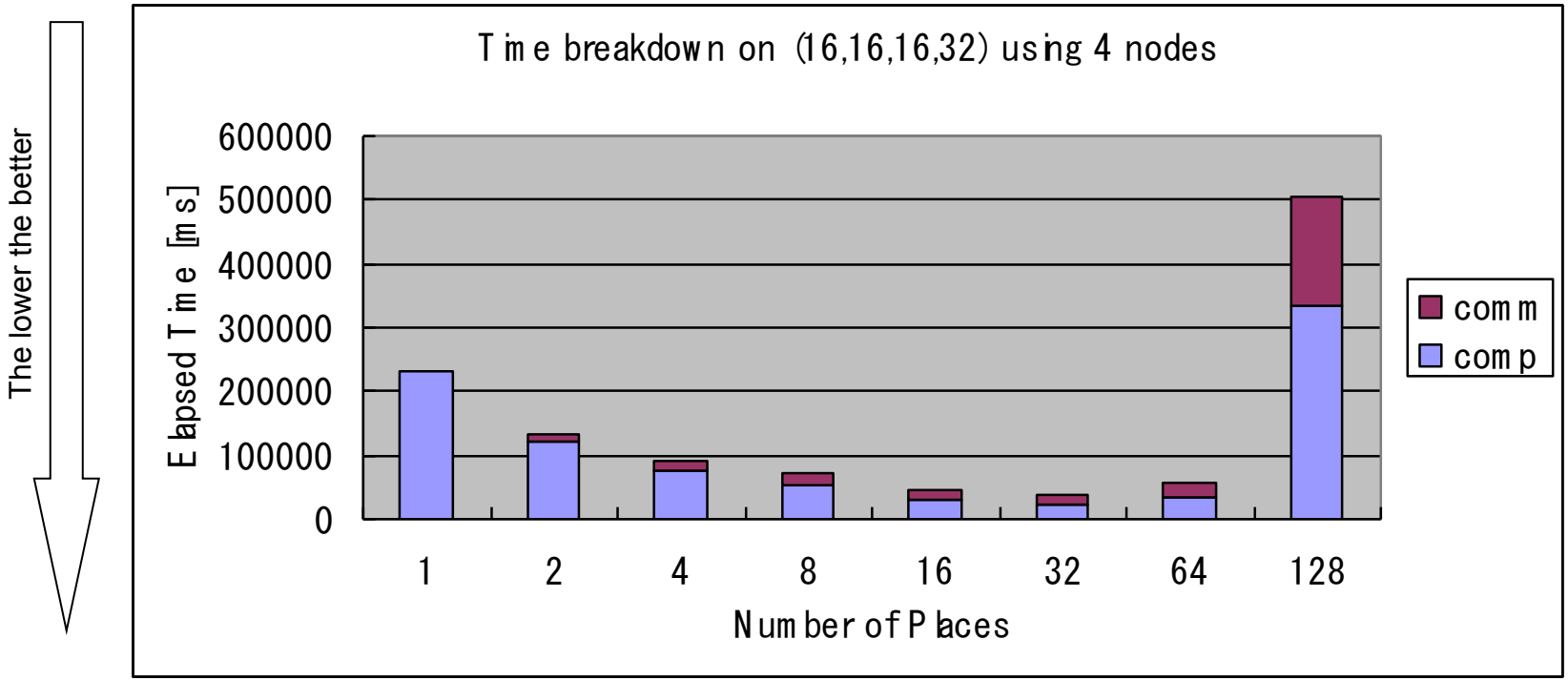


Xeon(Sandy bridge) E5 2680 (2.70GHz, L1=32KB, L2=256KB, L3=20MB, 8 cores, HT ON) x2
DDR-3 32GB, Red Hat Enterprise Linux Server 6.3 (2.6.32-279.el6.x86_64)
X10 trunk r25972 (built with "-Doptimize=true -DNO_CHECKS=true")
g++: 4.4.7
Compile option for native x10: -x10rt mpi -O -NO_CHECKS

Lattice QCD (Koichi)

- Time breakdown of Lattice QCD (non-overlapping version)
- Communication overhead causes the performance saturation
  - Communication overhead increases in proportion to the number of nodes
  - Communication ratio increases in proportion to the number of places

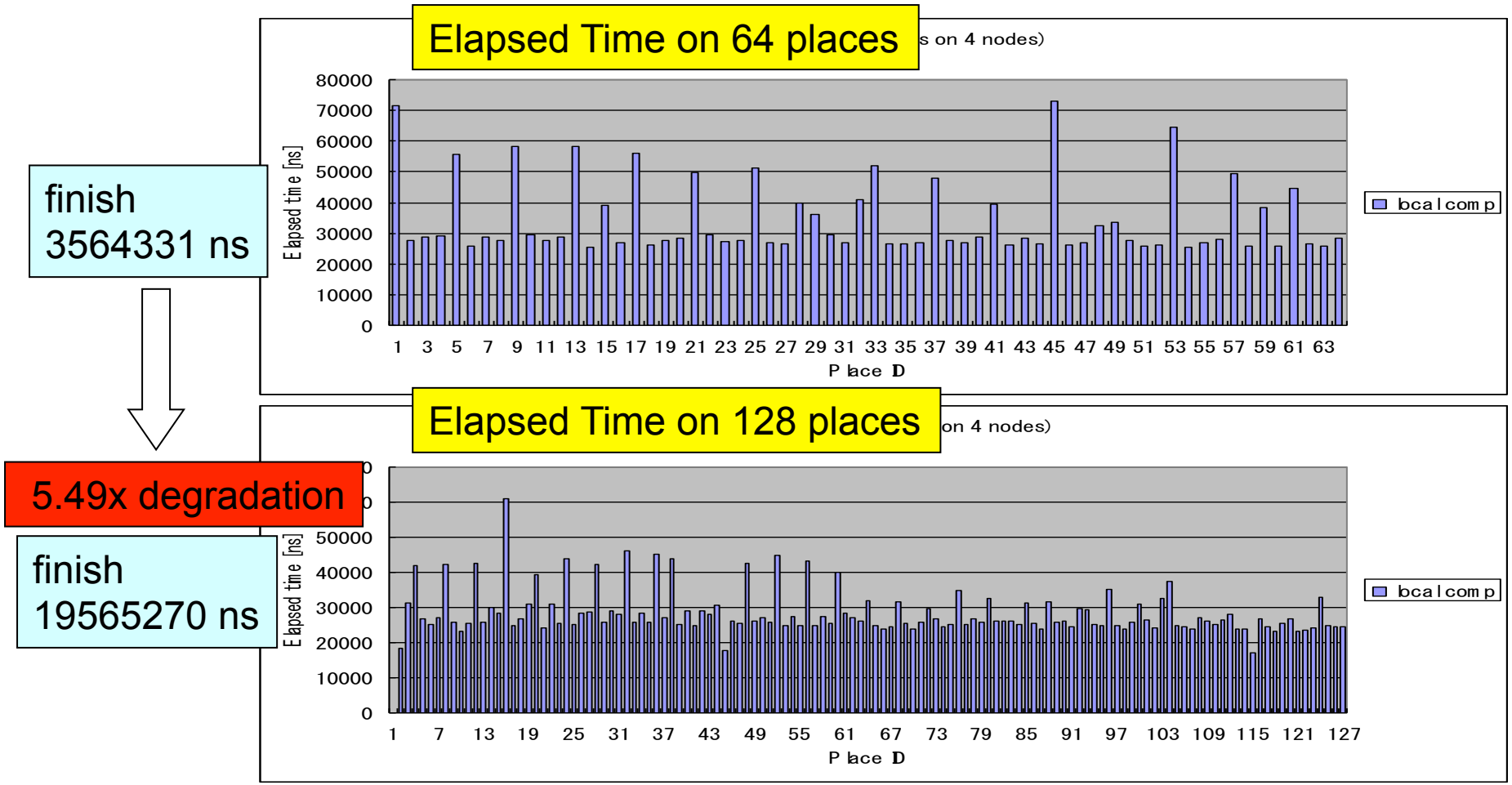
Elapsed Time



Xeon(Sandy bridge) E5 2680 (2.70GHz, L1=32KB, L2=256KB, L3=20MB, 8 cores, HT ON) x2
DDR-3 32GB, Red Hat Enterprise Linux Server 6.3 (2.6.32-279.el6.x86_64)
X10 trunk r25972 (built with "-Doptimize=true -DNO_CHECKS=true")
g++: 4.4.7
Compile option for native x10: -x10rt mpi -O -NO_CHECKS

Lattice QCD (Koichi)

- Time breakdown of a part of computation in a single finish (64, 128 places)
  - Significant degradation on 128 places compared to 64 places
  - Similar behavior on each place between using 64 places and 128 places
  - Hypothesis: invocation overhead of "at async" and/or synchronization overhead of "finish"



Xeon(Sandy bridge) E5 2680 (2.70GHz, L1=32KB, L2=256KB, L3=20MB, 8 cores, HT ON) x2
DDR-3 32GB, Red Hat Enterprise Linux Server 6.3 (2.6.32-279.el6.x86_64)
X10 trunk r25972 (built with "-Doptimize=true -DNO_CHECKS=true")
g++: 4.4.7
Compile option for native x10: -x10rt mpi -O -NO_CHECKS

Lattice QCD (Kaichi)

- at v/s asyncCopy on data transfer
  - asyncCopy performs 36.2% better when using 2 places (1 place / node)