HANDLING LOAD IMBALANCE IN DISTRIBUTED & SHARED MEMORY

Presenters: Harshitha Menon, Seonmyeong Bak

PPL Group
Phil Miller, Sam White, Nitin Bhat,
Tom Quinn, Jim Phillips, Laxmikant Kale

MOTIVATION

INTEGRATED RTS MODEL

APPLICATIONS

INTRODUCTION

Application –

Machine –

Dynamic Variations

INTRODUCTION

Dynamic variations - Load imbalance

Persistent & Transient imbalance

Vast amounts of on-node parallelism

Can we leverage multi-core shared memory systems to handle transient and persistent load imbalance while maintaining locality with low overhead?

INTEGRATED RTS

New integrated run-time system that combines distributed programming model with concurrent tasks

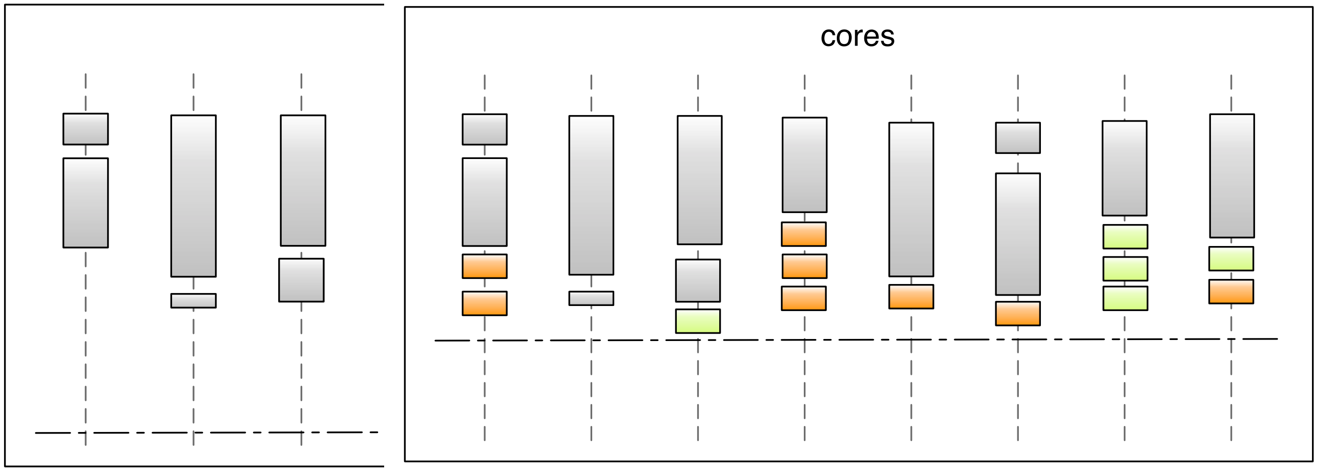
INTEGRATED RTS

Infrequent periodic balancing

+

Fine-grained work-sharing within the node

TRANSIENT & RESIDUAL IMBALANCE



INTEGRATED RTS MODEL

CHARM++ MODEL

Asynchronous Message-Driven Execution

Over-decomposition

Migratability

OVER-DECOMPOSITION

Decompose work & data units to many more pieces than execution units

OVER-DECOMPOSITION

Encapsulation of data and its computation inherently promotes data locality

MIGRATABILITY

Move work units to another execution unit at run time.

CHARM++ SMP MODEL

Takes advantage of multi-core processors

Launches one thread per core

CHARM++ SMP MODEL

Faster intra-node communication

Smaller memory footprint

Enables work-sharing within a node

INTEGRATED RTS

Combines Charm++ over-decomposition distributed memory model with concurrent tasks

PERSISTENT & TRANSIENT LOAD BALANCE

Node-aware load balancers

Fine-grained work-sharing within the node

LOAD BALANCING

Based on *principle of persistence*

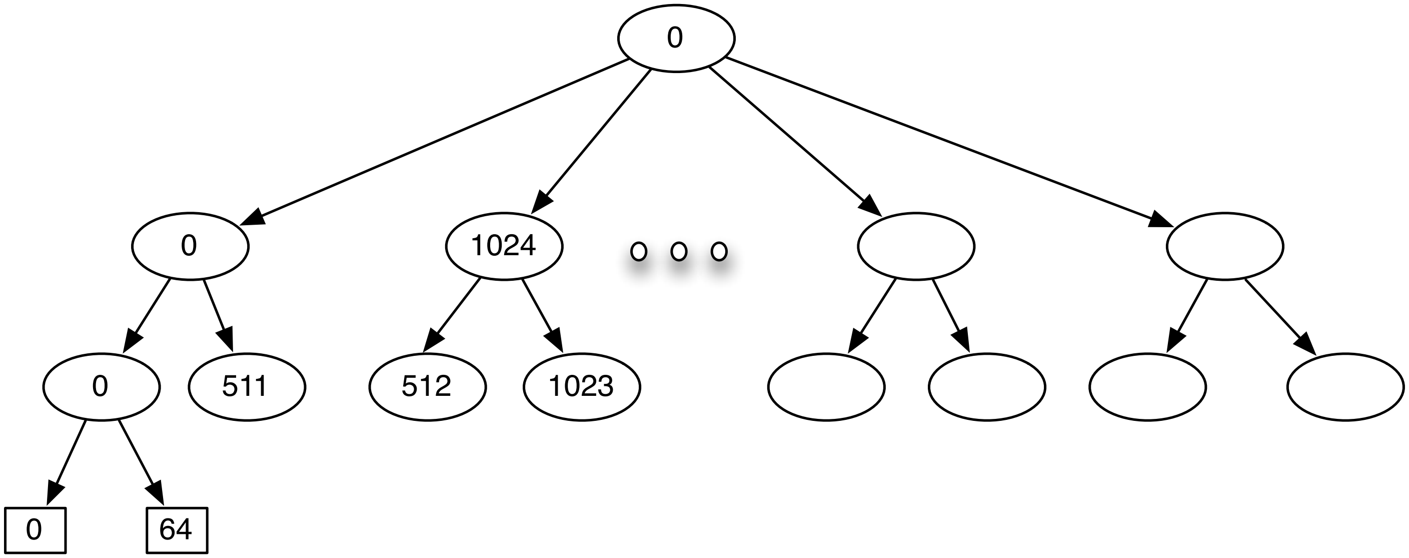
NODE-AWARE LOAD BALANCING

Hierarchical strategy

Coarsening to reduce memory and communication overhead

Different strategies at different levels

HIERARCHICAL LOAD BALANCER



AUTOMATIC LOAD BALANCING

RTS decides when to do load balancing

RTS decides which load balancer to use at each level

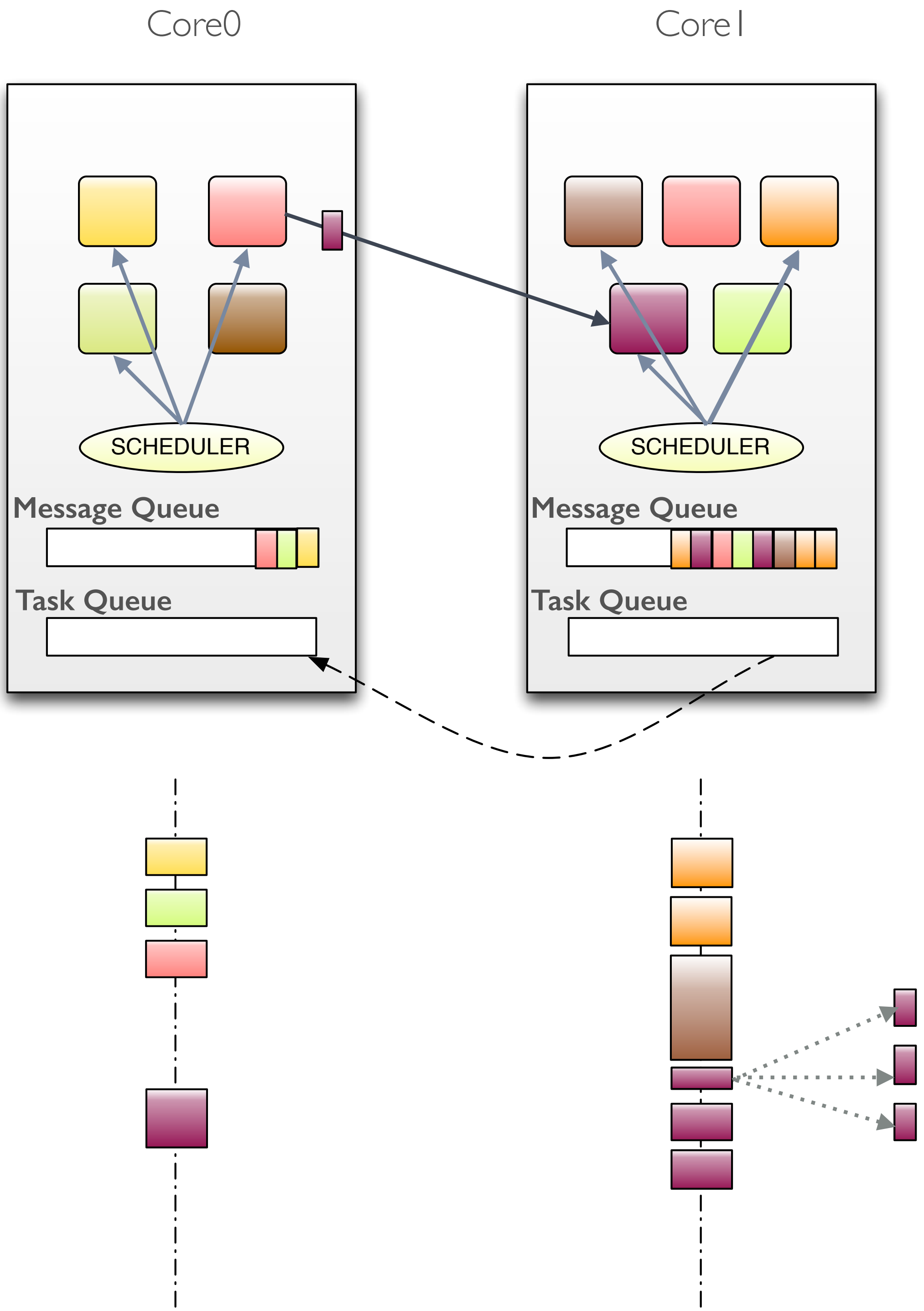
FINE-GRAINED WORK-SHARING

User specified tasks that can be executed on any core within a node

Work-stealing queue

With RTS support incurs lesser overhead

CkLoop - Previous work-sharing construct in Charm++



Core0

Core1

SCHEDULER

Message Queue

Task Queue

SCHEDULER

Message Queue

Task Queue

TASK CREATION

Charm++ task API

OpenMP integration

TASK GENERATION & SCHEDULING

Recursive

Broadcast task message

Only when idle

History

RECURSIVE TASKS

Loop iterations split into half forms a task

Work-stealing queue to share work

BROADCAST TASK MESSAGE

All the cores within a node receive broadcast message for a task

Atomically increments a variable to obtain the next chunk

ONLY WHEN IDLE

An atomic counter to determine number of idle cores within a node

Selectively creates tasks depending on that counter

Adaptively control number of tasks generated

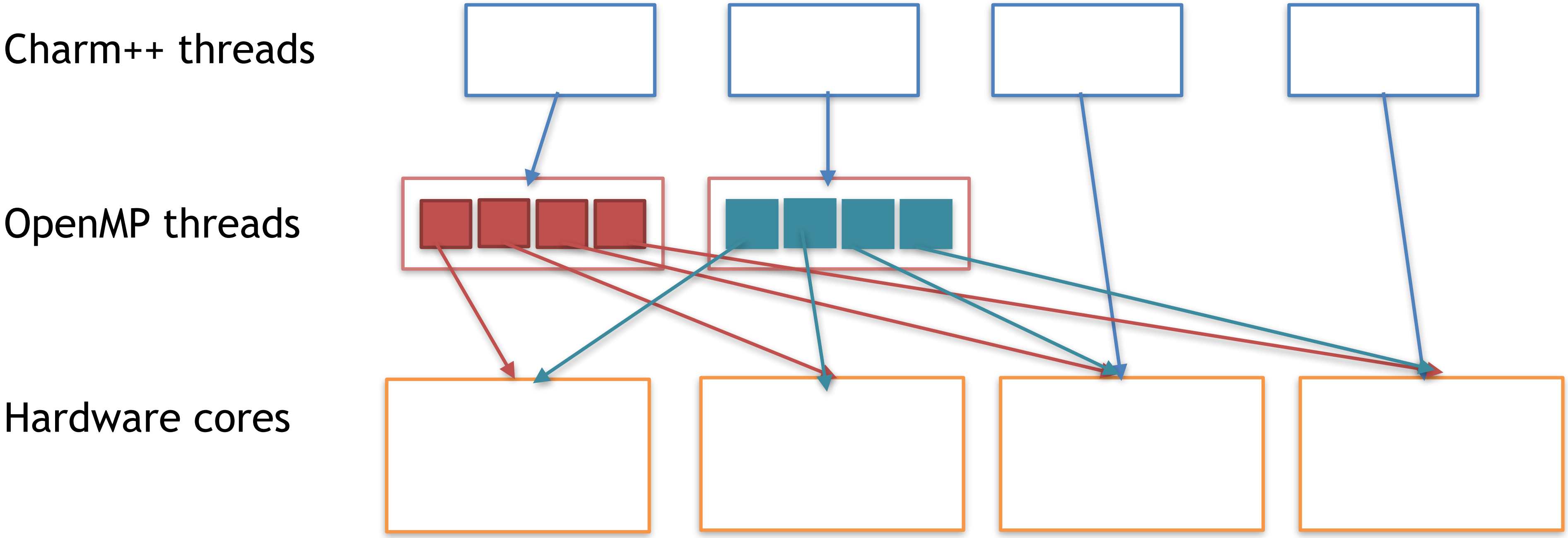
HISTORICAL DATA

Historical data on fraction of the tasks executed locally

Use this data to determine number of tasks to be created

Issues in the OpenMP interoperation with Charm++

- Resource contention
  - Oversubscription problem by separate Charm++ and OpenMP thread pool



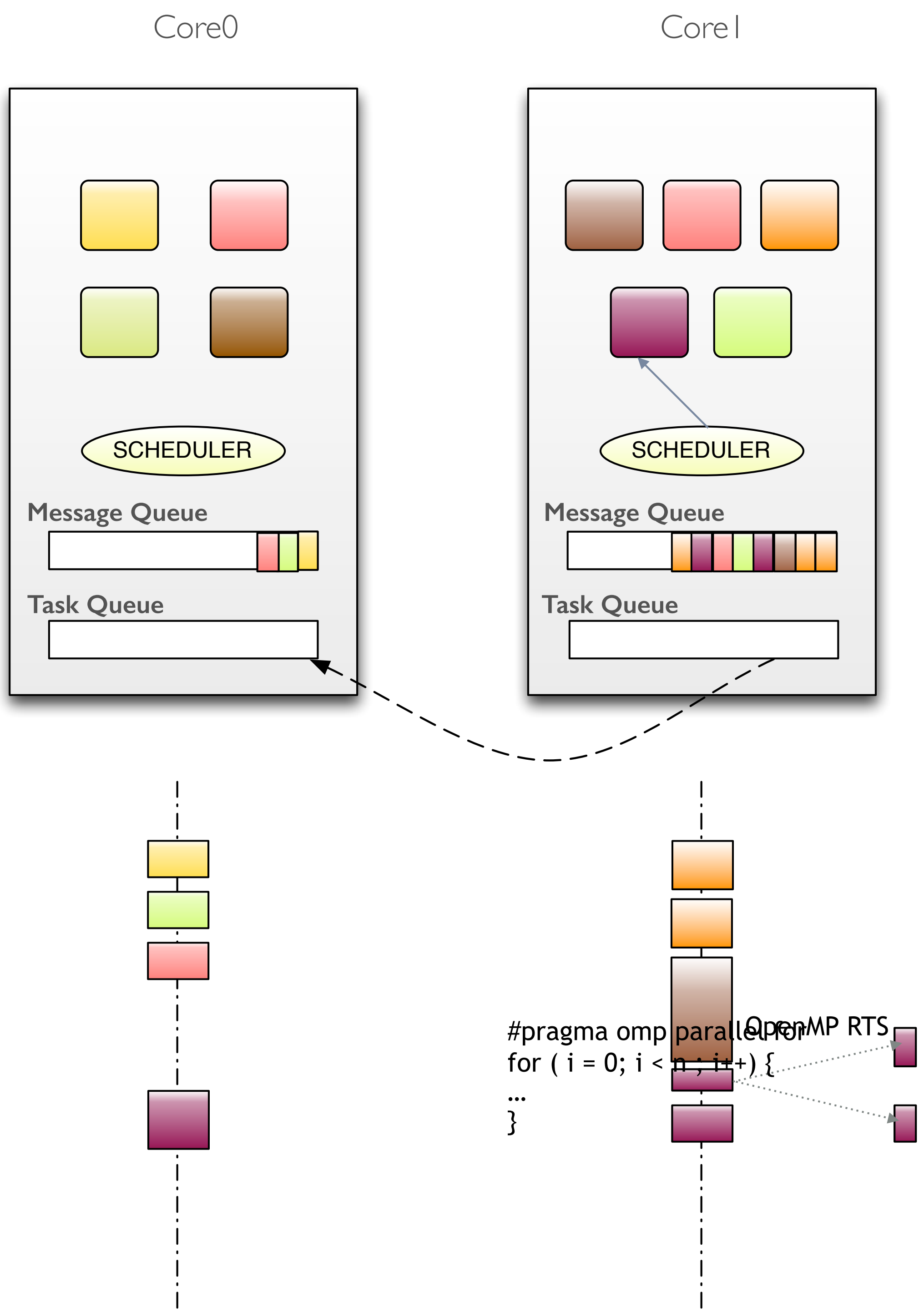
# of hw cores = 4 < # of threads = 10 -> **oversubscription**

OpenMP integration into Charm++

- Use threads on Charm++ Runtime for OpenMP
  - Works on SMP mode of Charm++

- Each OpenMP task become a Charm++ runtime message
  - OpenMP tasks can be migrated among cores within a node
  - Used for transient load balancing within a node

- Modified GNU OpenMP 4.0, forked from GCC 4.9.3



```
#pragma omp parallel for
for ( i = 0; i < n ; i++) {
...
}
```

Issues in naïve OpenMP integration

- Overheads of message creation
  - Too many messages are created
  - OpenMP tasks are created even when there is no idle thread on a node

- Applied some optimizations to solve this overheads

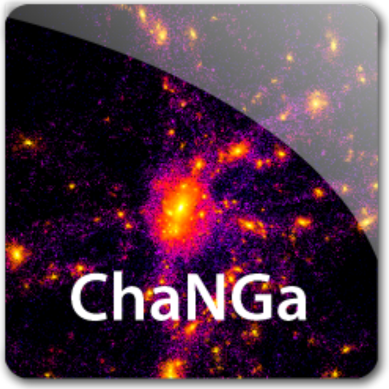
Optimizations to solve the overhead of naïve OpenMP integration

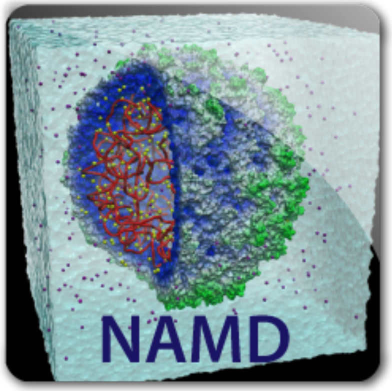
- Use an atomic counter
  - keep track of the number of idle threads within a node
- Use a history vector
  - keep record of how many of the OpenMP tasks have been stolen and executed by the other idle threads
- Combine these two heuristics to determine the number of messages to be created

APPLICATIONS

Application Study

- Applications
  - ChaNGa
    - Charm++, Cosmology Simulation
    - Cosmo25 dataset is used
(Highly clustered 2 billion particles dark simulation)



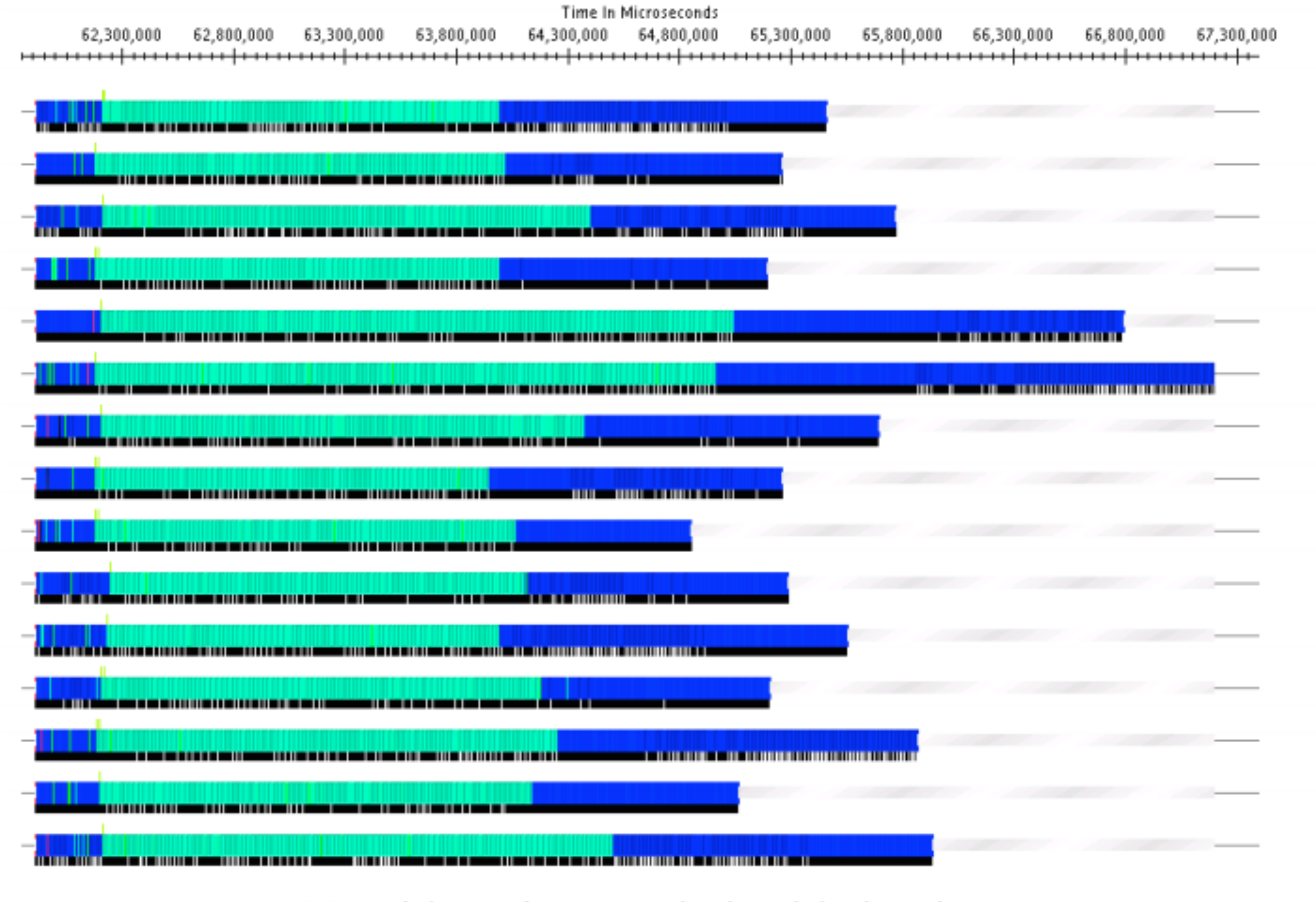


  - NAMD
    - Charm++, Molecular Dynamics simulation
    - Energy minimization run using collective variable module
(270,000 atoms and 200,000 bonds)
  - Kripke
    - MPI, Deterministic Particle Transport proxy application
    - Benefit: multiple MPI ranks per node can each parallelize OMP regions across entire node if imbalanced
    - Described in the next talk on AMPI

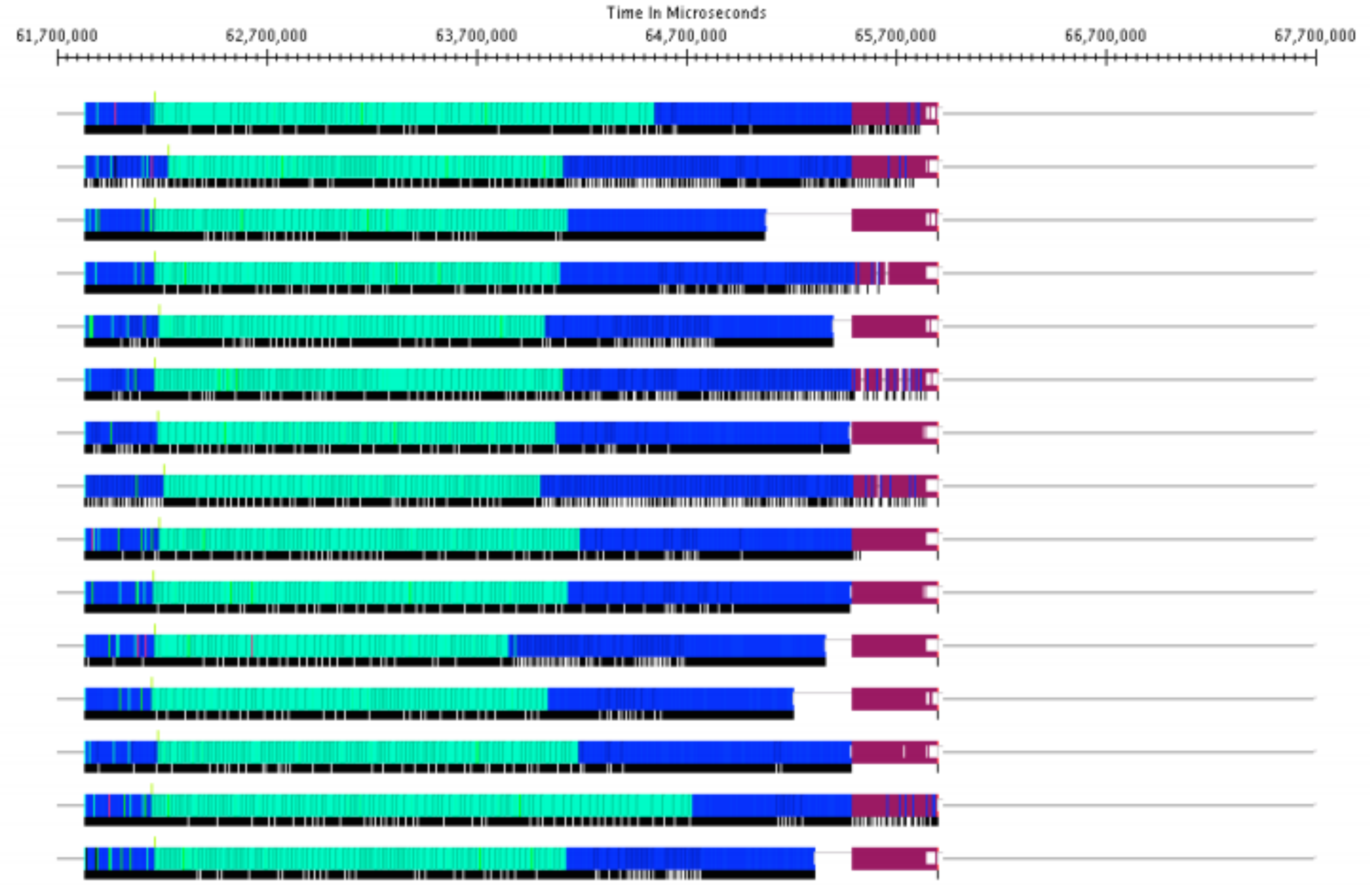
Application Study

- Machines used for evaluation
  - ANL Vesta, Bluegene Q
    - 2048 nodes, PowerPC A2 1.6Ghz (16 cores, 64 threads)
  - NCSA Blue Waters, Cray XE/XK hybrid
    - 22,640 nodes for Cray XE
    - AMD interlagos 6276 (16 cores, 32 threads)

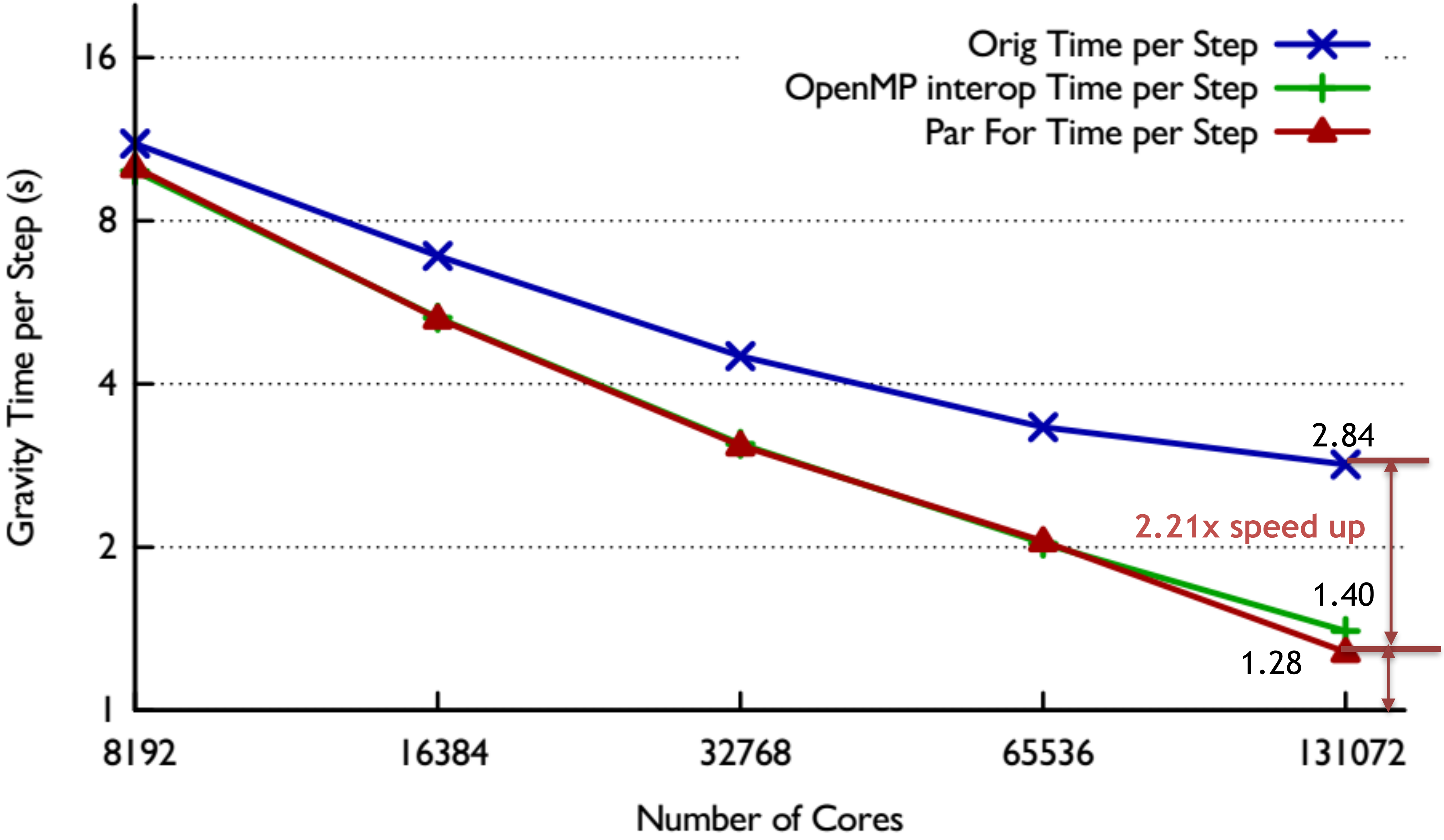
Projection of ChaNGa with cosmo25 on Blue Waters, Cray XE6 (128K cores)



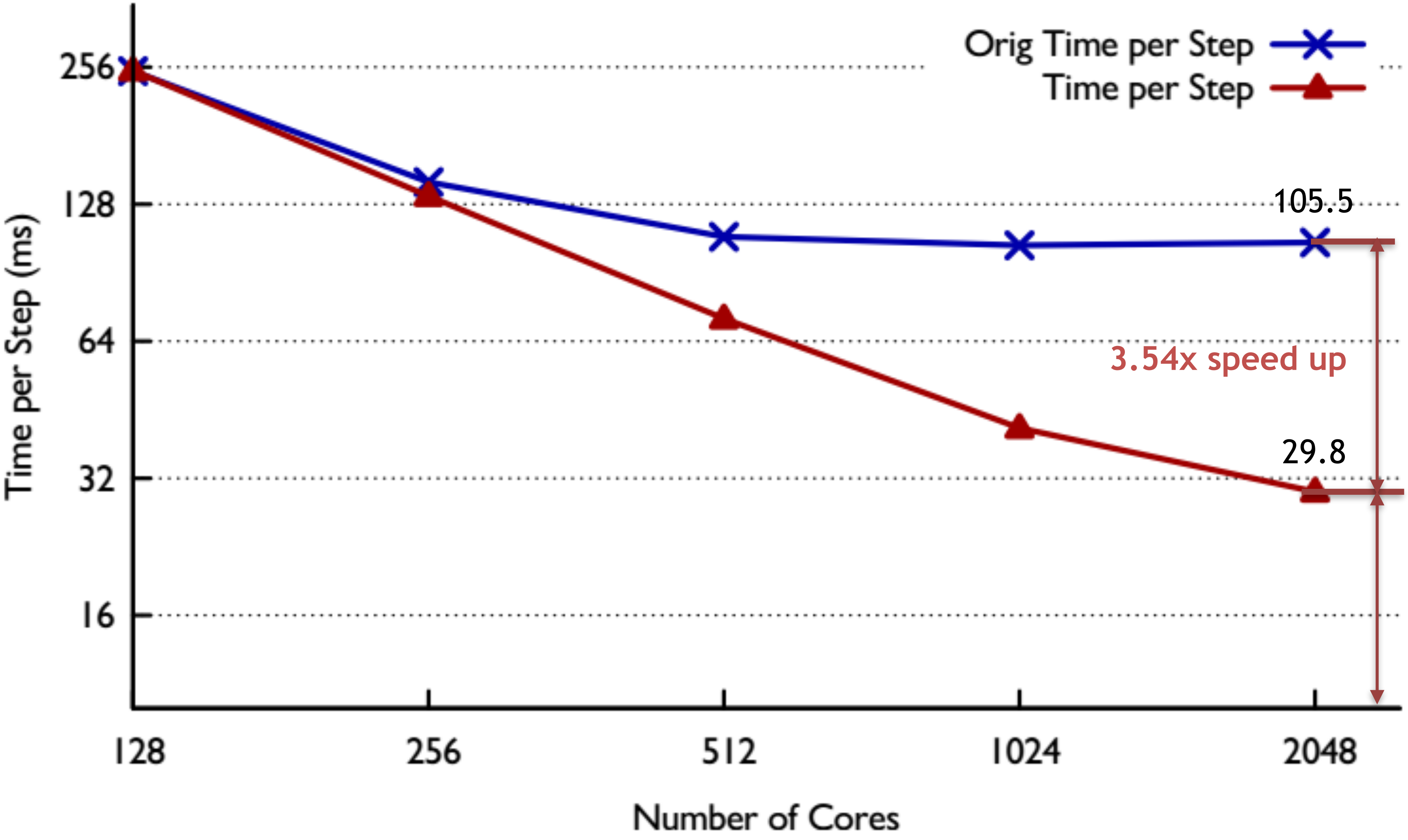
Projection of ChaNGa with cosmo25 on Blue Waters, Cray XE6 (128K cores)



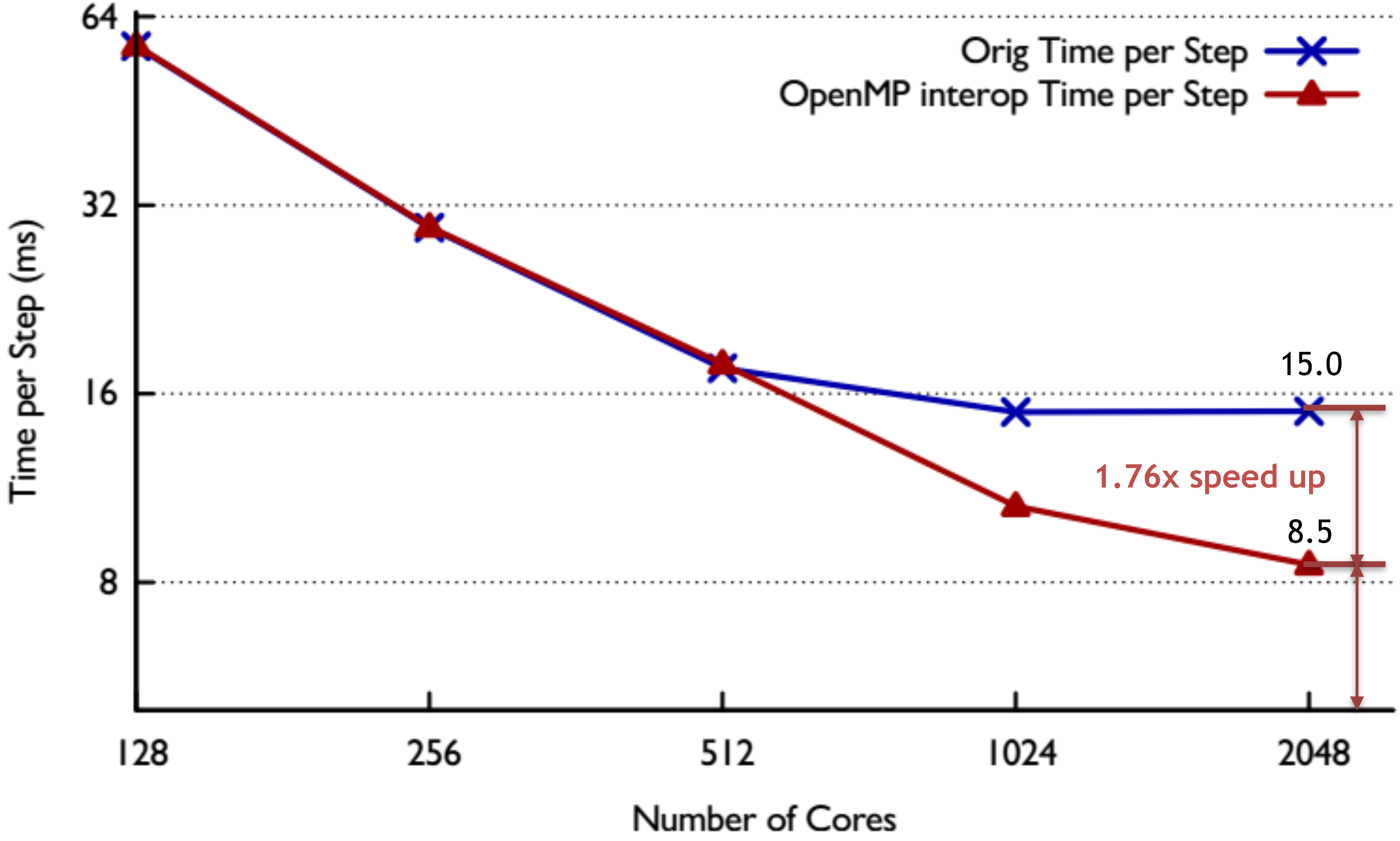
ChaNGa with cosmo25 on Blue Waters, Cray XE6



NAMD with colvar module on ANL Vesta, Bluegene Q



NAMD with colvar module on Blue Waters, Cray XE6



Summary

- We proposed new parallel constructs and OpenMP integration with Charm++
- Solved load imbalance in intra-node level significantly
- In NAMD and ChaNGa, load imbalance in intra-node level is mitigated significantly
- Even MPI application can benefit from this work with AMPI

THANK YOU!