SAND2017-9790PE

ASC ATDM Level 2 Milestone #6015: Asynchronous Many-Task Software Stack Demonstration

Final Review

August 9, 2017

Sandia National Laboratories

U.S. DEPARTMENT OF ENERGY
NNSA National Nuclear Security Administration

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND NO. 2011-XXXXP

The Milestone Team

- Janine C. Bennett
- Matt Bettencourt
- Robert Clay
- H. Carter Edwards
- Micheal Glass
- David Hollman
- Hemanth Kolla
- Jonathan Lifflander
- David Littlewood
- Aram H. Markosyan
- Stan Moore
- Stephen Olivier
- J. Antonio Perez
- Eric Phipps
- Francesco Rizzi
- Nicole Slattengren
- Dan Sunderland
- Jeremiah J. Wilke

The Review Committee

- Robert Armstrong, SNL (Committee Chair)
- Patricia Hough, SNL
- Michael Tupek, SNL
- David Daniel, LANL
- David Richards, LLNL
- Rajeev Thakur, ANL

Outline

- Motivation
- Milestone Overview
- AMT + DARMA Overview
- Milestone Results: Bottom Line Up front
- Deep Dive on Findings
  - Generality of the Backend API
  - Interoperability
  - Performance and Productivity
- Conclusions
- Future Work

Programmatic Drivers

- ASC/ATDM and the broader DOE Exascale Computing Project (ECP) are motivated by challenges presented by new computing architectures that are on the path to Exascale.
- There have been a number of “Exascale Challenges” workshops held in 2011/2012 as a build up to ATDM and ECP
- One such focus area was “Programming Challenges”
  - Asynchronous programming paradigms were proposed as a possible way to address some of the challenges – productivity and performance.
  - Asynchronous Many-Task Programming models have been a research area spread across many university (mostly) efforts.

ASC/ATDM Application Drivers

EMPIRE

- Focused on system qualification to hostile-ionizing radiation environments
- Coupled Source Region ElectroMagnetic Pulse (SREMP) to System Generated ElectroMagnetic Pulse (SGEMP) simulation.
- Physical spatial domain on the order of kilometers down to system geometry on the order of millimeters
- Embedded sensitivity analysis, uncertainty quantification and optimization

SPARC/SPARTA

- Virtual flight test simulations of re-entry vehicles from bus separation (exo-atmospheric) to target for normal and hostile environments
- SPARC
  - Time-accurate wall-modeled LES of high Reynolds number (100k-10M) hypersonic gas dynamics
- SPARTA
  - DSMC code to model low-density gas flow in the upper atmosphere
- Embedded sensitivity analysis, uncertainty quantification and optimization

ASC/IC Applications: Sierra and RAMSES

- Traditional engineering mechanics, electromagnetics, radiation effects, and circuit analysis

How Can AMT Impact Sandia's ASC Applications?

- Provide an abstraction layer for AMT programming models
- Influence the broader AMT community
- Overarching Questions
  - Is it portable across a variety of runtime system technologies?
  - Is it interoperable with Kokkos?
  - What is its performance and productivity compared to MPI?

Outline

- Motivation
- Milestone Overview
- AMT + DARMA Overview
- Milestone Results: Bottom Line Up front
- Deep Dive on Findings
  - Generality of the Backend API
  - Interoperability
  - Performance and Productivity
- Conclusions
- Future Work

Asynchronous Many-Task (AMT)
Software Stack Demonstration

- This milestone will evaluate a DARMA-compliant AMT runtime software stack comprising ATDM ASD software components and existing community AMT runtime technologies (e.g., Charm++).
- We will assess the performance and productivity of this software stack on kernels and proxy applications representative of the Sandia ATDM applications.
- As part of the effort to assess the perceived strengths and weaknesses of AMT models compared to more traditional approaches, experiments will be performed on test bed machines and one or more ATS-x system (target is Trinity).

Outcomes of milestone

- An initial DARMA-compliant AMT software stack
- A clear understanding of the strength and limitations of DARMA abstractions and DARMA-compliant software stack in the context of SNL’s ATDM codes
- Information to guide our future research and development in this area

Milestone Deliverables

1. DARMA-compliant AMT software stack on tests beds and one or more ATS-x system (intent is to use Trinity).
2. Implementation of ATDM application kernels and proxies developed for the AMT software stack.
3. An analysis of the productivity, performance, scalability, and dynamic load balancing capability for the DARMA-compliant runtime on those ATDM application kernels and proxies.
4. A report to inform the code development road map guiding the (Sandia) ASC code strategy

Outline

- Motivation
- Milestone Overview
- AMT + DARMA Overview
- Milestone Results: Bottom Line Up front
- Deep Dive on Findings
  - Generality of the Backend API
  - Interoperability
  - Performance and Productivity
- Conclusions
- Future Work

AMT research is focused on mitigating system complexities at the runtime system-level

- Abstractions provide a separation of concerns
- Removal of system-level specifics from application code
- Task parallelism
- Asynchrony, overlap of communication and computation
- Load balancing

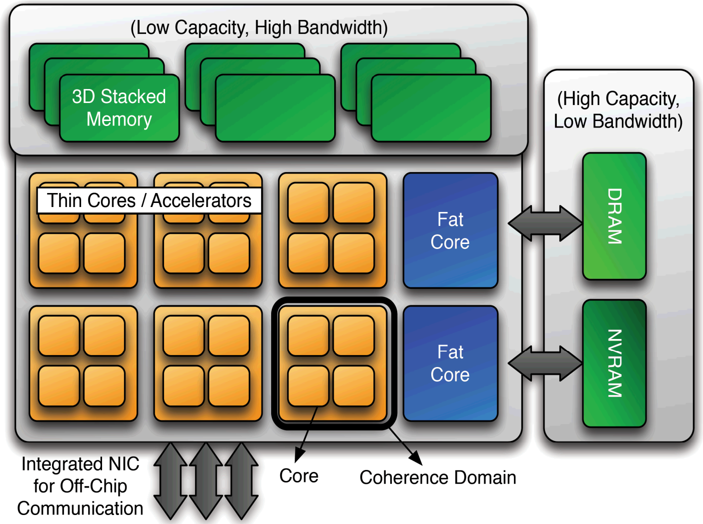

(Low Capacity, High Bandwidth)
3D Stacked Memory
(High Capacity, Low Bandwidth)
Thin Cores / Accelerators
Fat Core
Fat Core
DRAM
NVRAM
Integrated NIC for Off-Chip Communication
Core
Coherence Domain

Image courtesy of www.cal-design.org

COMPUTER ARCHITECTURE LABORATORY
EXASCALE DESIGN SPACE EXPLORATION

AMT models require a shift from an ***imperative to declarative*** programming paradigm

Imperative vs declarative programming: a simple example

Imperative

```
Get a piece of bread
If likes mustard
    Add mustard
If not vegetarian
    Add meat
Add cheese
Add veggies
Put more bread on top
Cut in half
```

Declarative

```
Make me a sandwich
```

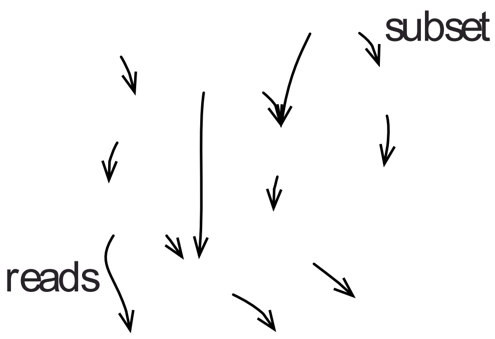
Programmer uses explicit statements to control program state and prescribe order of operations

Programmer expresses logic without prescribing control-flow

What is it about AMT models that enables a declarative programming approach?

- Directed acyclic graph (DAG) encodes data-task dependencies
- Enables a runtime system to reason about
  - Task and data parallelism
  - Overlapping communication and computation
  - Load balancing
  - When and where to execute work and move data

data-task graph

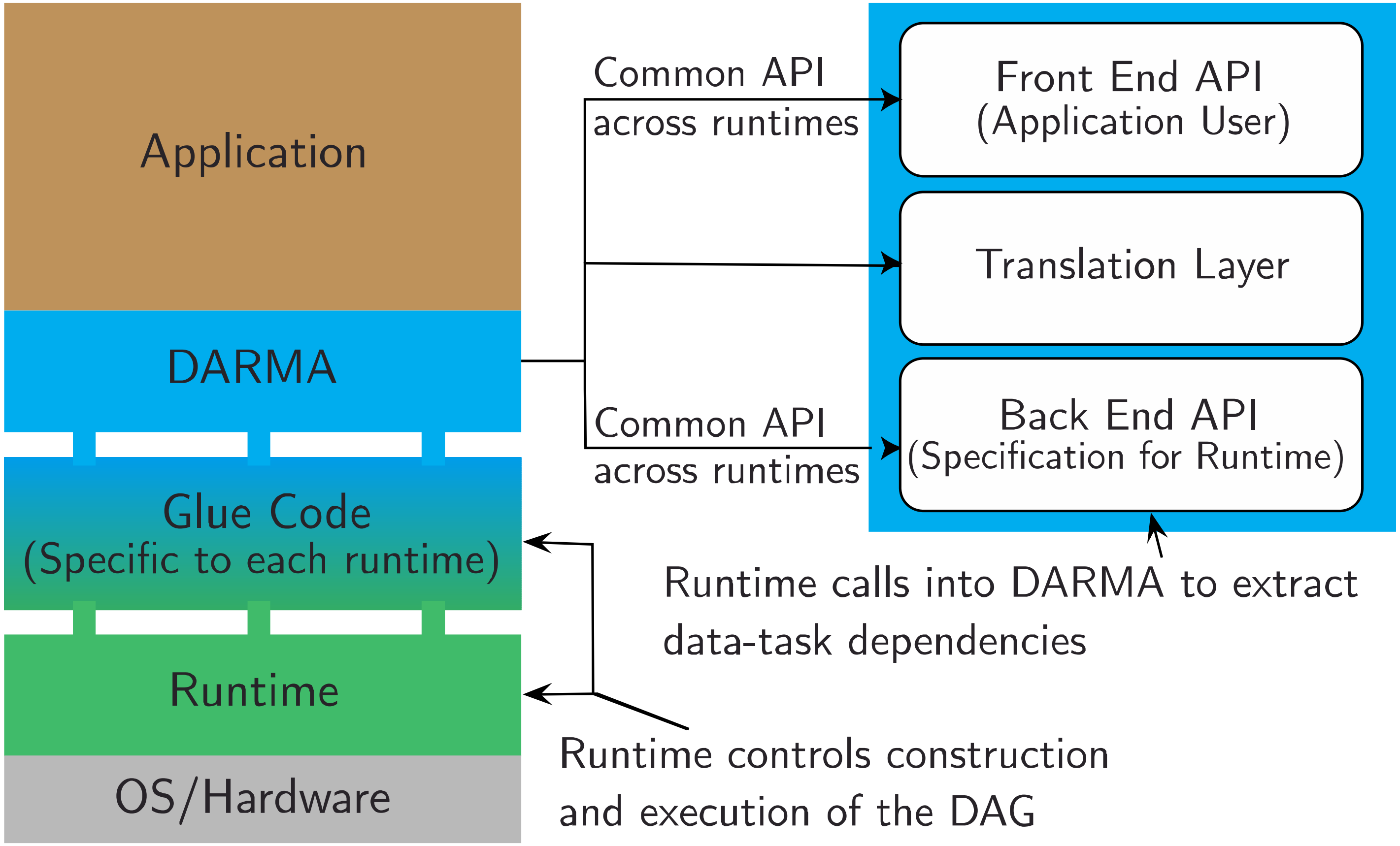


What is DARMA?

DARMA is a C++ abstraction layer for asynchronous many-task (AMT) runtimes.

It provides a set of abstractions to facilitate the expression of tasking that map to a variety of underlying AMT runtime system technologies.

What is DARMA?


Application
DARMA
Glue Code
(Specific to each runtime)
Runtime
OS/Hardware
Common API
across runtimes
Front End API
(Application User)
Translation Layer
Common API
across runtimes
Back End API
(Specification for Runtime)
Runtime calls into DARMA to extract data-task dependencies
Runtime controls construction and execution of the DAG

Outline

- Motivation
- Milestone Overview
- AMT + DARMA Overview
- Milestone Results: Bottom Line Up front
- Deep Dive on Findings
  - Generality of the Backend API
  - Interoperability
  - Performance and Productivity
- Conclusions
- Future Work

Milestone Deliverables

1. DARMA-compliant AMT software stack on tests beds and one or more ATS-x system (intent is to use Trinity).
2. Implementation of ATDM application kernels and proxies developed for the AMT software stack.
3. An analysis of the productivity, performance, scalability, and dynamic load balancing capability for the DARMA-compliant runtime on those ATDM application kernels and proxies.
4. A report to inform the code development road map guiding the (Sandia) ASC code strategy

Milestone Deliverable 1

A DARMA-compliant AMT software stack

- We have three different stacks in various stages of development
  - DARMA-Charm++: Fully distributed, focus of milestone performance analysis
  - DARMA-OnNode: Development tool
  - DARMA-HPX: Prototypes* (HPX3, HPX5)

including ATDM ASD components

- DARMA-OnNode+Kokkos (using OpenMP affinity layer for resource management)
- Ongoing research and development with Kokkos and Resource manager teams on DARMA-Charm+++Kokkos*

on NGP testbeds and one or more ATS-x system (intent is to use Trinity)

- Analyses were performed on Trinity (ATS-1) and Mutrino (Trinity testbed)

Milestone Deliverable 2

Implementation of ATDM application kernels and proxies developed for the AMT system

- Three benchmarks
  - Written by DARMA developers
  - Purpose: highlight benefits/limitations of the programming model and runtime
    - Jacobi: memory-bound computation, latency-bound communication to expose overheads
    - Molecular dynamics: compute-bound with more bandwidth-intensive communication to complement Jacobi
    - Simulated Imbalance: assess load balancing capabilities
- Three proxy applications
  - Written by application developers
  - Purpose: co-development of APIs, acquire subjective feedback, requirements
    - PIC: Direct collaboration with EMPIRE application team
      - SimplePIC, MiniPIC*
    - UQ*: Embedded analysis is a capability used by both applications
    - Multiscale*: Ties to IC/Sierra

Milestone Deliverable 3

<u>An analysis of the productivity, performance, scalability, and dynamic load balancing capability for the DARMA-compliant runtime on those ATDM application kernels and proxies</u>

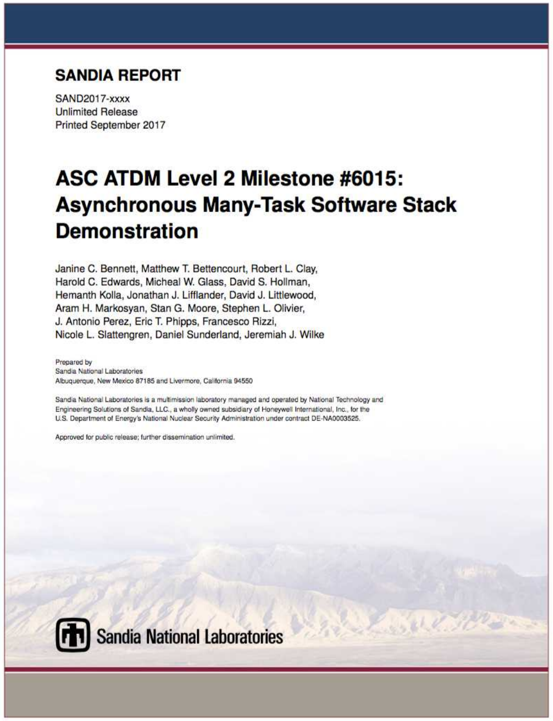
- 10000s of runs performed on ATS-1 systems (Haswell and KNL)
  - Mutrino: scaling and profiling studies up to 64 nodes (2K cores Haswell), (4K cores KNL)
  - Trinity KNL:
    - Scaling studies up to 2K nodes (131K cores)
    - Limited access precluded full scaling studies to 4K nodes (262K cores), spot runs only at 4K completed
  - Trinity Haswell: Some scaling results, given limited access we focused on KNL
- Two compilers: GCC6.3.0 and ICC18.0.0beta (ICC results NDA for now)
- Scaling studies, performance profiling to assess:
  - Overheads in balanced use cases with imperative baselines (MPI-only)
  - Scaling trends (focus on strong, weak scaling as time permitted)
  - Load-balancing capabilities for load-imbalanced use cases
- Subjective feedback from application developers on productivity
- Summary of semantic information gain in DARMA program specification

*exceeds criteria

Milestone Deliverable 4

A report to inform the code development road map guiding the (Sandia) ASC code strategy

- Currently in draft form
- ~130 pages so far with details regarding Deliverables 1, 2, and 3



SANDIA REPORT

SAND2017-xxxx
Unlimited Release
Printed September 2017

ASC ATDM Level 2 Milestone #6015: Asynchronous Many-Task Software Stack Demonstration

Janine C. Bennett, Matthew T. Bettencourt, Robert L. Clay,
Harold C. Edwards, Micheal W. Glass, David S. Hollman,
Hemanth Kolla, Jonathan J. Lifflander, David J. Littlewood,
Aram H. Markosyan, Stan G. Moore, Stephen L. Olivier,
J. Antonio Perez, Eric T. Phipps, Francesco Rizzi,
Nicole L. Slattengren, Daniel Sunderland, Jeremiah J. Wilke

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

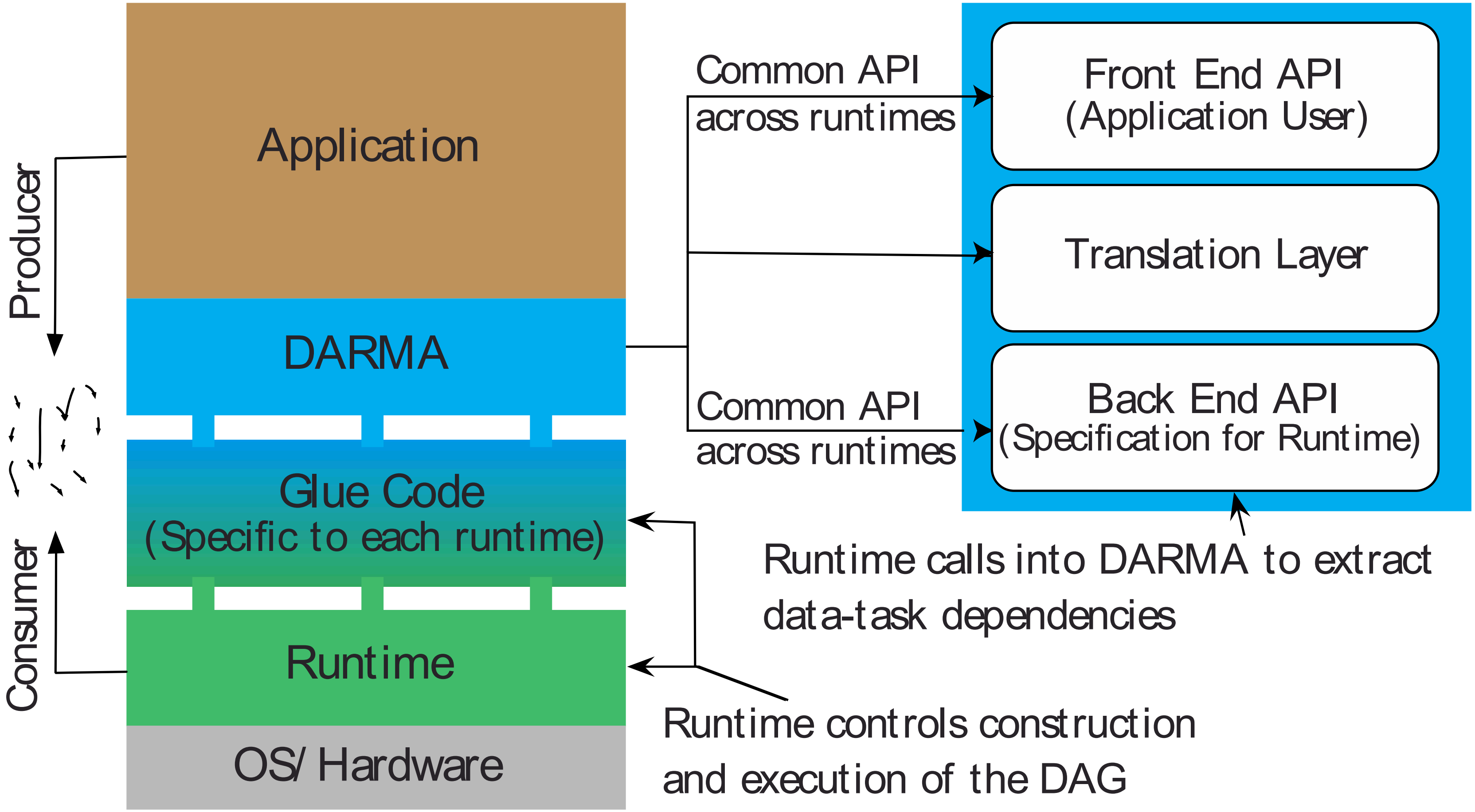
Conclusions

- **Productivity:**
  - **Easier to express communication overlap:** no Isend/wait pairs, communication progress not explicit in application code
  - **Easier to express tunable granularity:** data decomposition can mismatch execution resources (overdecomposition) without changing application code
  - **Easier to enable load balancing:** migratable data and work chunks can be transparently rebalanced without explicit bookkeeping and rebalancing in application code
- **Performance:**
  - **DARMA is scalable (weak and strong) up to 2K nodes**
  - **Load balancing shows major performance gains** with minimal effort from app developer
  - **Deferred execution and sequential task model have overheads** (~10% over MPI)
  - **Expect DARMA performance to improve as we tune the implementation**
- **Interoperability**: **It's complicated, but the initial results are promising; major focus in Q1 FY18**
- **Generality: declarative backend specification facilitates mapping to different technologies, development of "common components" across backend implementations**

Outline

- Motivation
- Milestone Overview
- AMT + DARMA Overview
- Milestone Results: Bottom Line Up front
- Deep Dive on Findings
  - Generality of the Backend API
  - Interoperability
  - Performance and Productivity
- Conclusions
- Future Work

By design DARMA captures a declarative specification of the application that does not prescribe control-flow



Runtime calls into DARMA to extract data-task dependencies

Runtime controls construction and execution of the DAG

DARMA’s Backend Runtime System Responsibilities

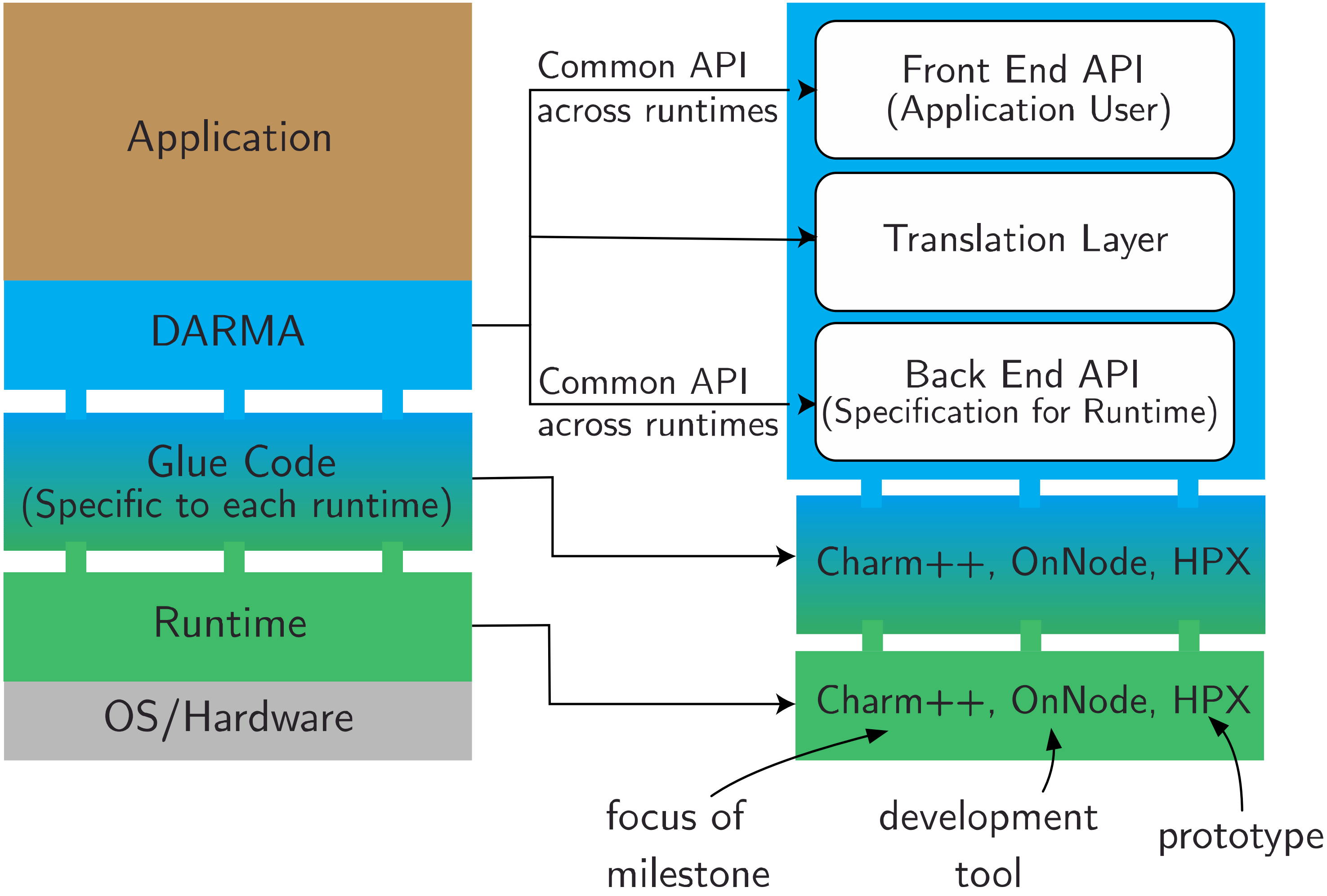
- **Manage data dependencies between tasks (data inputs and outputs)**
  - Exploit data usage (write/read/etc.) and sequencing information from the frontend to schedule tasks without data conflicts
  - Make scheduling decisions based on current state to copy, move, or stall data accesses to optimize performance and memory usage
- **Determine and track placement of data, tasks, and task collections across distinct memory spaces**
  - Distributed reference counting of data to determine task readiness and schedule appropriately
  - Manage location of task collection elements to efficiently transfer data for publishes (send) and fetches (receive) between elements
- **Coordinate data movement utilizing the underlying communication transport layer**
  - Use frontend interface to serialize/de-serialize arbitrarily typed objects to move C++ object across memory spaces
- **Implement collective operations (currently only reduce and all-reduce)**

DARMA's Backend Runtime System Responsibilities

- **Manage data dependencies between tasks (data inputs and outputs)**
  - Exploit data usage (write/read/etc.) and sequencing information from the frontend to schedule tasks without data conflicts
  - Make scheduling decisions based on current state to copy, move, or stall data accesses to optimize performance and memory usage
- **Determine and track placement of data, tasks, and task collections across distinct memory spaces**
  - Distributed reference counting of data to determine task readiness and schedule appropriately
  - Manage location of task collection elements to efficiently transfer data for publishes (send) and fetches (receive) between elements
- **Coordinate data movement utilizing the underlying communication transport layer**
  - Use frontend interface to serialize/de-serialize arbitrarily typed objects to move C++ object across memory spaces
- **Implement collective operations (currently only reduce and all-reduce)**

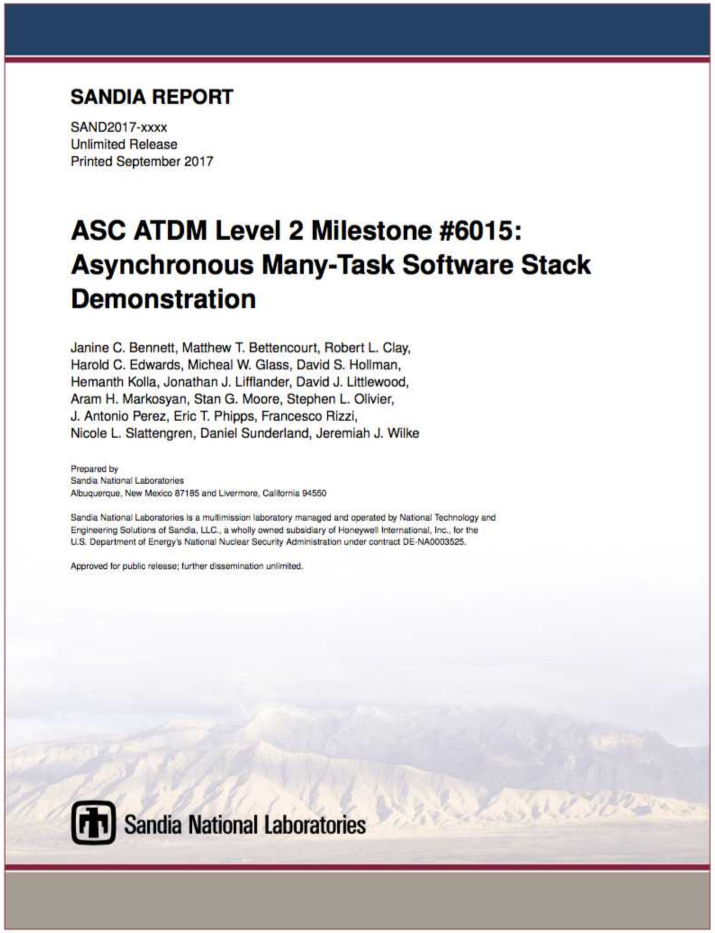
A runtime's level of native support for these capabilities is a contributing factor to the thickness of the "glue code"

Currently there are three back ends in various stages of development



Strategy and implementation details for backend mappings are included in the milestone report

- Details for current backends:
  - Charm++
  - OnNode (threads)
  - HPX3
  - HPX5
- Strategy for other backends:
  - REALM
  - Legion (Discussion of differences and similarities in programming model)
  - MPI



SANDIA REPORT

SAND2017-xxxx
Unlimited Release
Printed September 2017

ASC ATDM Level 2 Milestone #6015: Asynchronous Many-Task Software Stack Demonstration

Janine C. Bennett, Matthew T. Bettencourt, Robert L. Clay,
Harold C. Edwards, Micheal W. Glass, David S. Hollman,
Hemanth Kolla, Jonathan J. Lifflander, David J. Littlewood,
Aram H. Markosyan, Stan G. Moore, Stephen L. Olivier,
J. Antonio Perez, Eric T. Phipps, Francesco Rizzi,
Nicole L. Slattengren, Daniel Sunderland, Jeremiah J. Wilke

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

DARMA-Charm++ Overview

- **Manage data dependencies between tasks (data inputs and outputs)**
  - **Not a direct mapping:** implements local and distributed schedulers in Charm++ user-space to schedule and track DARMA data
- **Determine and track placement of data, tasks, and task collections across distinct memory spaces**
  - **Not a direct mapping:** utilizes Charm++'s groups, nodegroups, and chare arrays to manage DARMA tasks and data.
  - Carefully passes DARMA task collections to Charm++ chare arrays to utilize LB effectively
- **Coordinate data movement utilizing the underlying communication transport layer**
  - **Close mapping:** Uses Charm++'s native, platform-specific network layers (ugni, ibverbs, tcp/ip, mpi) to transfer data
  - **Close mapping:** Performs serialization/de-serialization by passing data to Charm++'s extensive PUP (Pack/UnPack) interface
- **Implement collective operations (currently only reduce and all-reduce)**
  - **Not a direct mapping:** Charm++ has a native reduce but not an all-reduce. Since Charm++ has vastly different collective semantics, reduce and all-reduce are re-implemented, but re-use Charm++ topological spanning trees

Outline

- Motivation
- Milestone Overview
- AMT + DARMA Overview
- Milestone Results: Bottom Line Up front
- Deep Dive on Findings
  - Generality of the Backend API
  - Interoperability
  - Performance and Productivity
- Conclusions
- Future Work

Interoperability: why is it important?

Sandia National Laboratories

- Interoperability: The ability of separate software components to efficiently share execution resources, share memory spaces, and exchange information
- Sandia has adopted a component-based approach to application development
- Interoperability of independent components is crucial to the success of the program

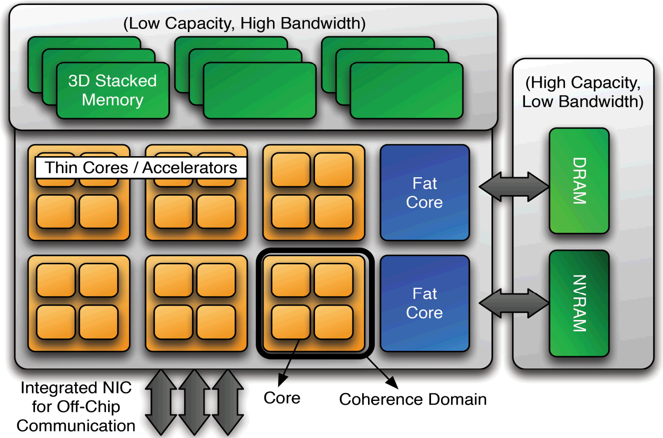

(Low Capacity, High Bandwidth)
3D Stacked Memory
(High Capacity, Low Bandwidth)
Thin Cores / Accelerators
Fat Core
Fat Core
DRAM
NVRAM
Integrated NIC for Off-Chip Communication
Core
Coherence Domain

Image courtesy of www.cal-design.org

COMPUTER ARCHITECTURE LABORATORY
EXASCALE DESIGN SPACE EXPLORATION

Trilinos ParaView OpenMP MPI PGAS OpenCL OpenACC Directives for Accelerators VTK NVIDIA CUDA

Interoperability: what is the underlying issue?

- Increases in system parallelism and heterogeneity drive increases in
  - Number of tools, runtimes, and languages aimed at gleaning performance from new architectures
  - Complexity of component based systems developed and deployed

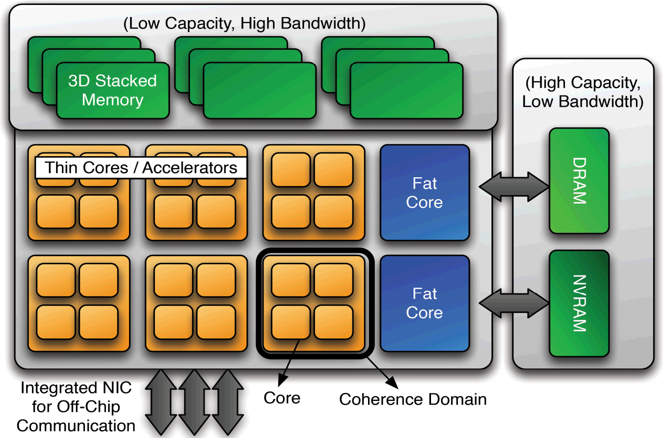

(Low Capacity, High Bandwidth)
3D Stacked Memory
(High Capacity, Low Bandwidth)
Thin Cores / Accelerators
Fat Core
Fat Core
DRAM
NVRAM
Integrated NIC for Off-Chip Communication
Core
Coherence Domain

Image courtesy of www.cal-design.org

- Challenge: Underlying assumption by many frameworks that all system resources are available for their use.

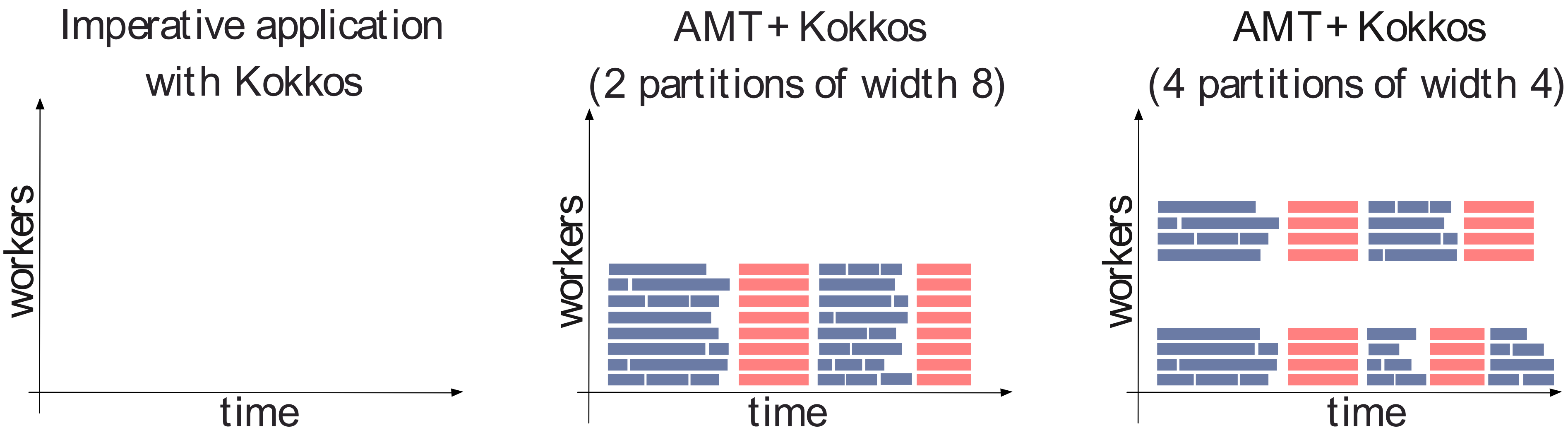
DARMA and its underlying runtime must be interoperable with node-level and network-level frameworks

- Node-level: Focus is interoperability with Kokkos
  - Kokkos provides performance portability across various architectures and has been adopted by Sandia's ASC applications
  - Interoperability research and development includes:
    - Execution-space interoperability
    - Memory-space/data management interoperability
  - Focus of this year's milestone: Execution Space Interoperability
- Network-level: Focus is interoperability with MPI
  - Handoff of network resources between imperative MPI codes and DARMA codes
  - Leading AMT runtimes all support this handoff mechanism, Charm++ included
  - DARMA abstractions naturally lend themselves to the handoff
  - Not in this year's milestone. Next year will see engineering work on this deliverable.

Kokkos: performance portability

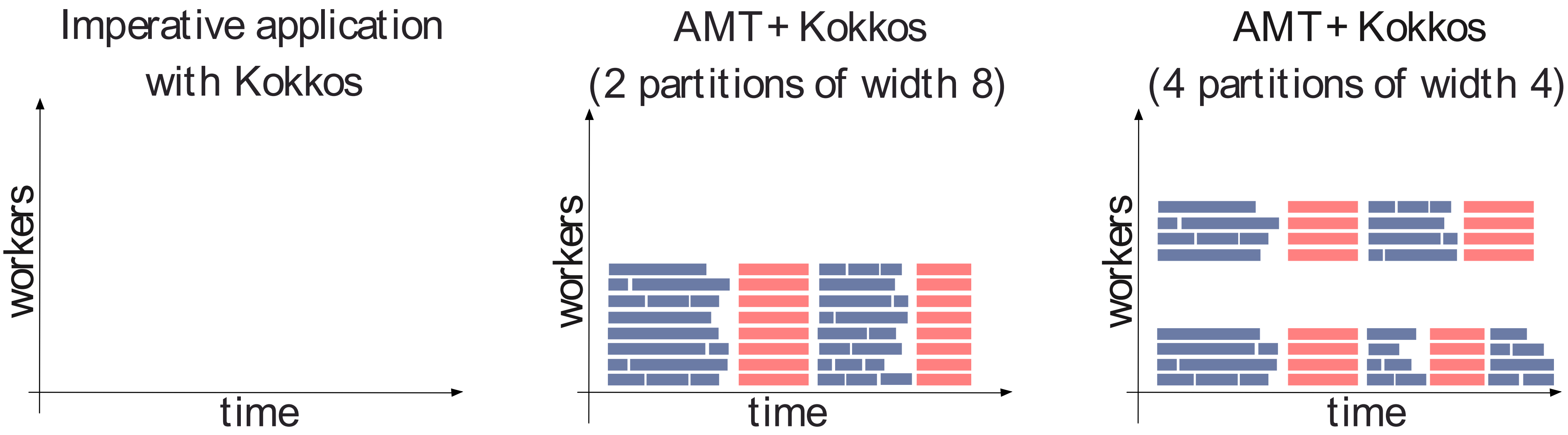
- Kokkos is a C++ library that provides node-level programming model abstractions to
  - Identify / encapsulate grains of data and parallelizable operations
  - Aggregate these grains with data structure and parallel patterns
  - Map aggregated grains onto memory and cores / threads
- Design is agnostic to inter-node parallelism
- Use case drivers initially focused on integration with MPI until recent AMT integration with Uintah

A visual comparison of Imperative+Kokkos and AMT+Kokkos highlights differences in use-cases



Serial work
Kokkos work
AMT work

A visual comparison of Imperative+Kokkos and AMT+Kokkos highlights differences in use-cases



Imperative application with Kokkos

AMT+ Kokkos (2 partitions of width 8)

AMT+ Kokkos (4 partitions of width 4)

Fork-join of resources

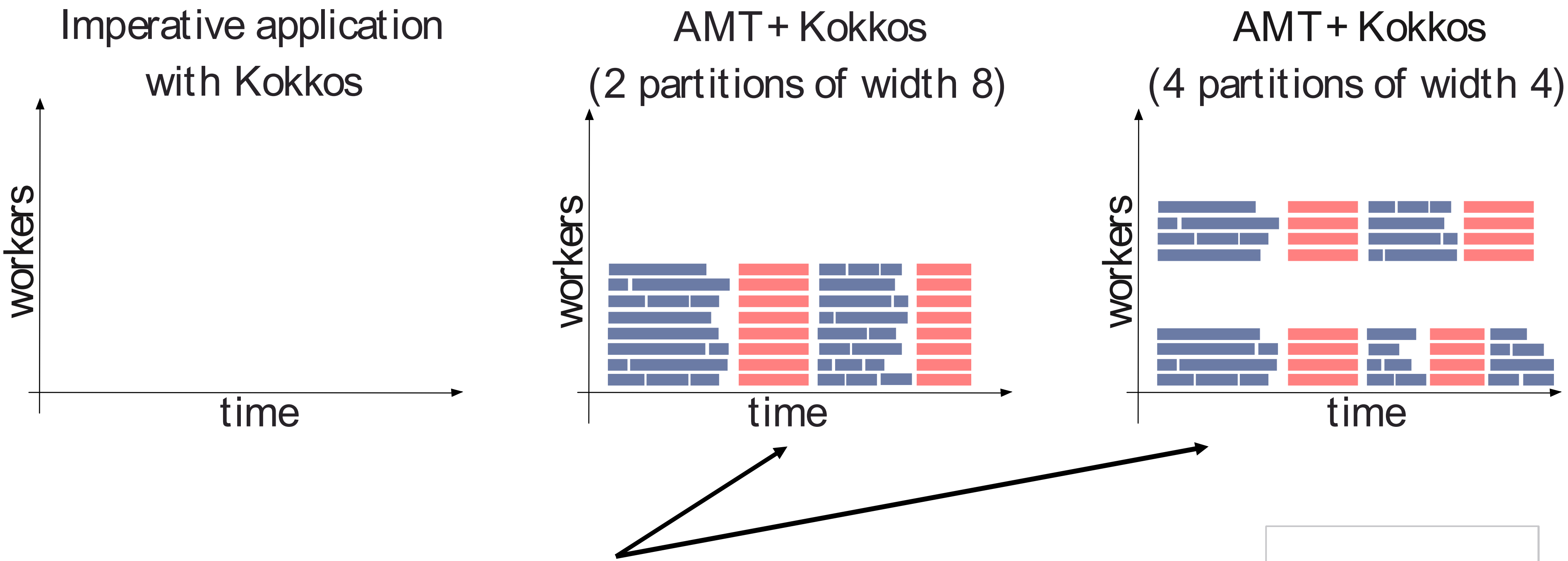
AMT runtimes are threaded with work scheduled dynamically, making
resource handoff more complicated.

Serial work

Kokkos work

AMT work

A visual comparison of Imperative+Kokkos and AMT+Kokkos highlights differences in use-cases


Imperative application with Kokkos
workers
time
AMT+ Kokkos (2 partitions of width 8)
workers
time
AMT+ Kokkos (4 partitions of width 4)
workers
time

Introduction of `Kokkos::partition_master` to partition node enables overlap of AMT and Kokkos work.

Serial work
Kokkos work
AMT work

Plan: Node Resource Manager (NoRMa) for execution resources management between Kokkos+DARMA

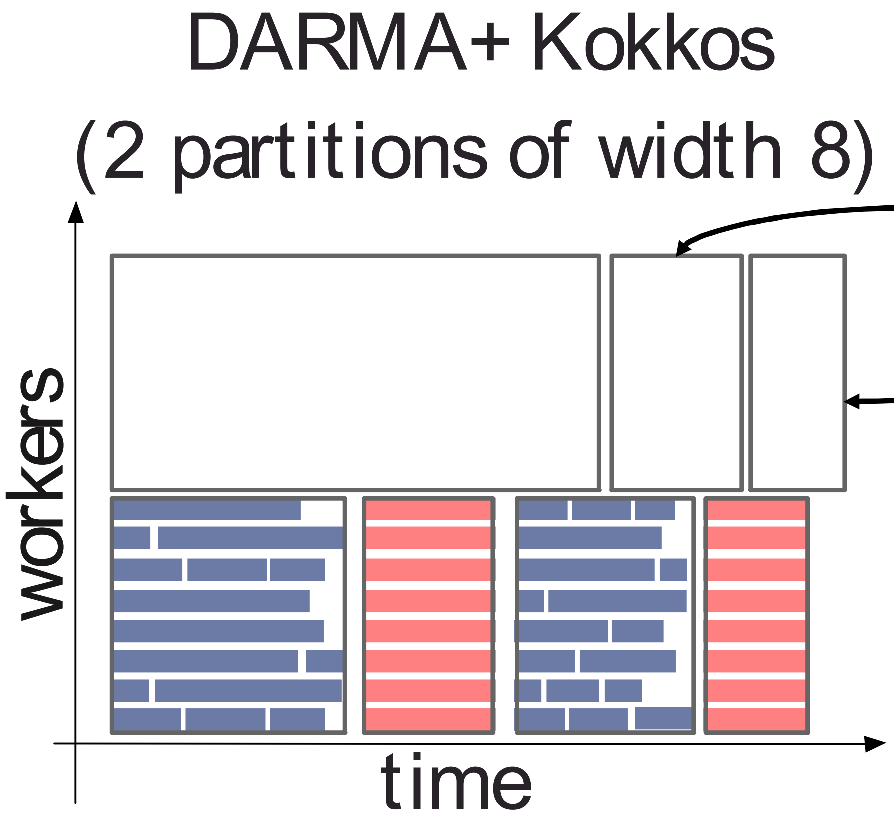
- Low-level interface: thin layer on-top of HWLOC
  - Maintains an inventory of available resources (cores and hardware threads)
  - Responds to requests from software components for those resources
  - Once resources are reserved, threads may be launched onto those resources directly by the owning component

- Optional high-level interface: C++ `std::threads`
  - Allows the “donation” of threads (bound to particular cores) between components (e.g. DARMA to Kokkos)

Ultimately the OpenMP affinity layer was chosen to manage execution resources between DARMA+Kokkos

- ATDM and IC application requirements regarding native vendor OpenMP libraries
  - Mixed Kokkos and OpenMP node programming (e.g., OpenMP-enabled Intel Math Kernel Library calls)
- Precedent of using OpenMP affinity layer in Uintah+Kokkos integration
- Charm++ runtime library’s interference with HWLOC controls

DARMA-OnNode+Kokkos prototype integration details

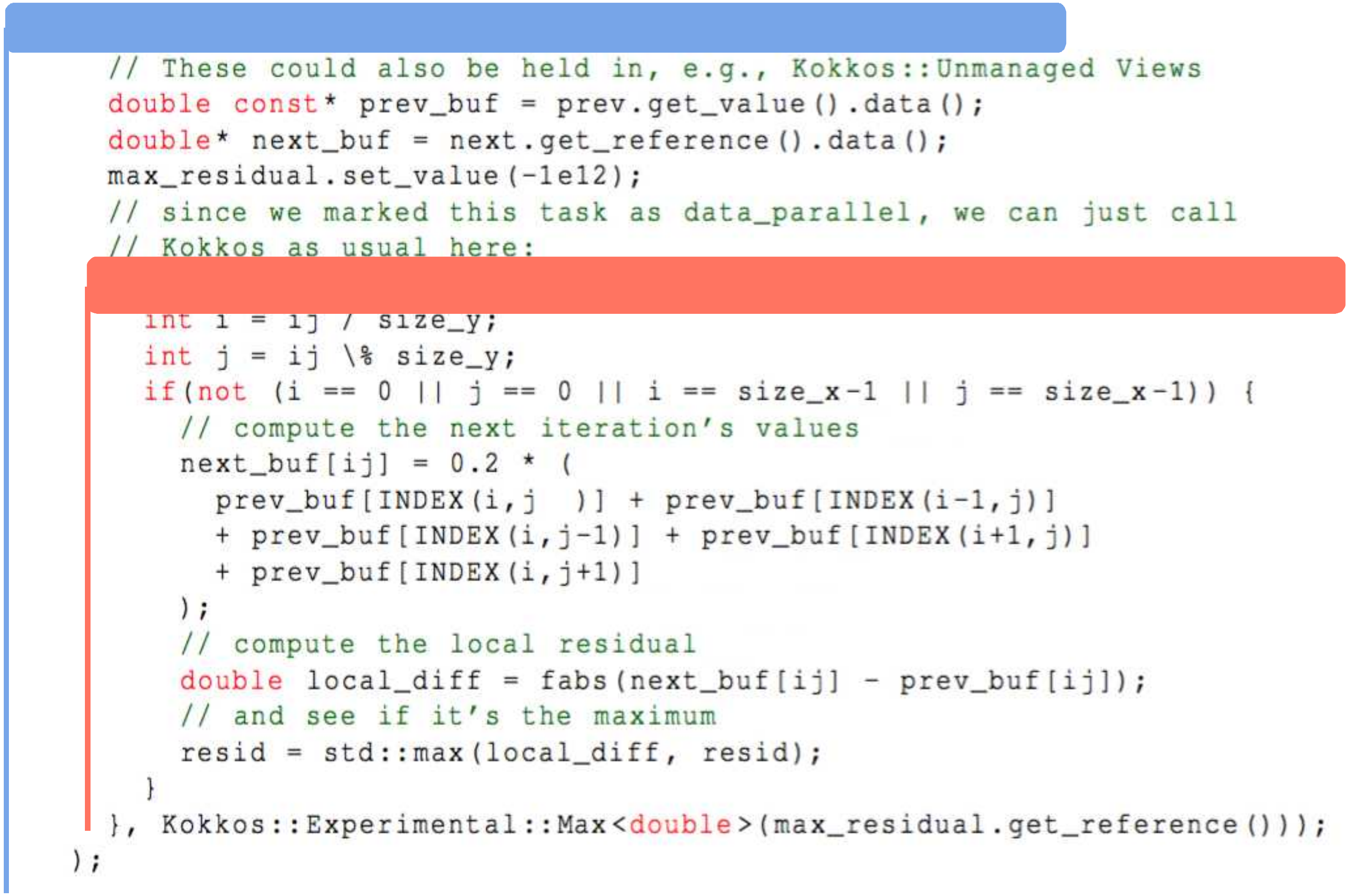


Synchronization via structured blocks of OpenMP parallel regions

Kokkos work
DARMA work

Exiting and re-entering of DARMA and Kokkos work queues happens at function boundaries

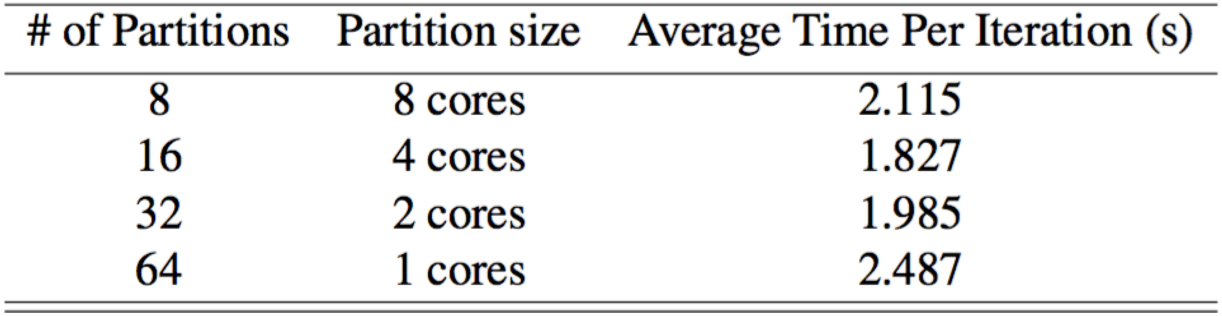
DARMA+Kokkos variant of the Jacobi benchmark used to demonstrate execution interoperability



```
 // These could also be held in, e.g., Kokkos::Unmanaged Views
 double const* prev_buf = prev.get_value().data();
 double* next_buf = next.get_reference().data();
 max_residual.set_value(-1e12);
 // since we marked this task as data_parallel, we can just call
 // Kokkos as usual here:

  int i = ij / size_y;
  int j = ij \% size_y;
  if(not (i == 0 || j == 0 || i == size_x-1 || j == size_x-1)) {
   // compute the next iteration's values
   next_buf[ij] = 0.2 * (
    prev_buf[INDEX(i,j  )] + prev_buf[INDEX(i-1,j)]
    + prev_buf[INDEX(i,j-1)] + prev_buf[INDEX(i+1,j)]
    + prev_buf[INDEX(i,j+1)]
   );
   // compute the local residual
   double local_diff = fabs(next_buf[ij] - prev_buf[ij]);
   // and see if it's the maximum
   resid = std::max(local_diff, resid);
  }
 }, Kokkos::Experimental::Max<double>(max_residual.get_reference()));
);
```

Preliminary experiments show that partition size impacts average iteration time on a KNL node



# of Partitions	Partition size	Average Time Per Iteration (s)
8	8 cores	2.115
16	4 cores	1.827
32	2 cores	1.985
64	1 cores	2.487

This prototype DARMA-OnNode+Kokkos integration feeds into ongoing DARMA-Charm++ + Kokkos development work

DARMA-Charm++ + Kokkos research and development is in progress and a major focus of FY18 efforts

- Similar to other AMT frameworks, Charm++ is centered on explicitly managed pthreads
  - OpenMP interoperability was via custom in-house libraries

- Work using vendor-supported OpenMP affinity layer in progress – joint with CharmWorks
  - Support directly within Charm++ runtime
  - Build and test support for integrated software stack

Outline

- Motivation
- Milestone Overview
- AMT + DARMA Overview
- Milestone Results: Bottom Line Up front
- Deep Dive on Findings
  - Generality of the Backend API
  - Interoperability
  - Performance and Productivity
- Conclusions
- Future Work

Performance and Productivity Outline

- DARMA productivity goals
- DARMA performance goals
- How do DARMA’s abstractions enable these goals?
- System target: ATS1 Overview
- Proxy results
- Benchmark results

DARMA productivity goals

- Application developers can focus on describing data decomposition and data effects, not managing execution resources (threads, network messages)
- Conflict-free programming model without explicit wait(…) calls
- Load balancing transparent to application, intrinsic to runtime
- Overlap of communication/computation transparent to application, intrinsic to runtime
- Ease transition from imperative to declarative programming style via deferred execution semantics

DARMA performance goals

- Deferred execution through C++ templates should be lightweight
- Overdecomposition should enable efficient pipelining of communication, overlap with computation
- Automatic, application-agnostic load balancers should achieve ~80-90% of the benefit of optimal load balancer

DARMA comprises abstractions for data and tasks

- Asynchronous smart pointers wrap user data and track meta-data used to build and annotate the DAG
  - `darma::AccessHandle<T>`
  - `darma::AccessHandleCollection<T>`
- Tasks are annotated via several interfaces
  - `darma::create_work`
  - `darma::create_concurrent_work`

How do DARMA’s abstractions enable these goals?

- Automatically capture dependencies and data effects through C++ metaprogramming
  - Visible code is just variables and functions, no tasks
  - Creating DAG directly in user code is tedious and error-prone
- Each data block/variable tracked by logical identifier in runtime
  - Enables automatic migration of data structures (data movement)
  - Enables automatic load balancing
- `create_concurrent_work` boundaries are natural locations for load balancing

How do DARMA’s abstractions enable these goals?

- Parallel algorithms are written to a data decomposition, not execution units (process, rank, thread)
  - Tunable granularity
  - Overdecomposition (communication overlap, load-balancing flexibility)

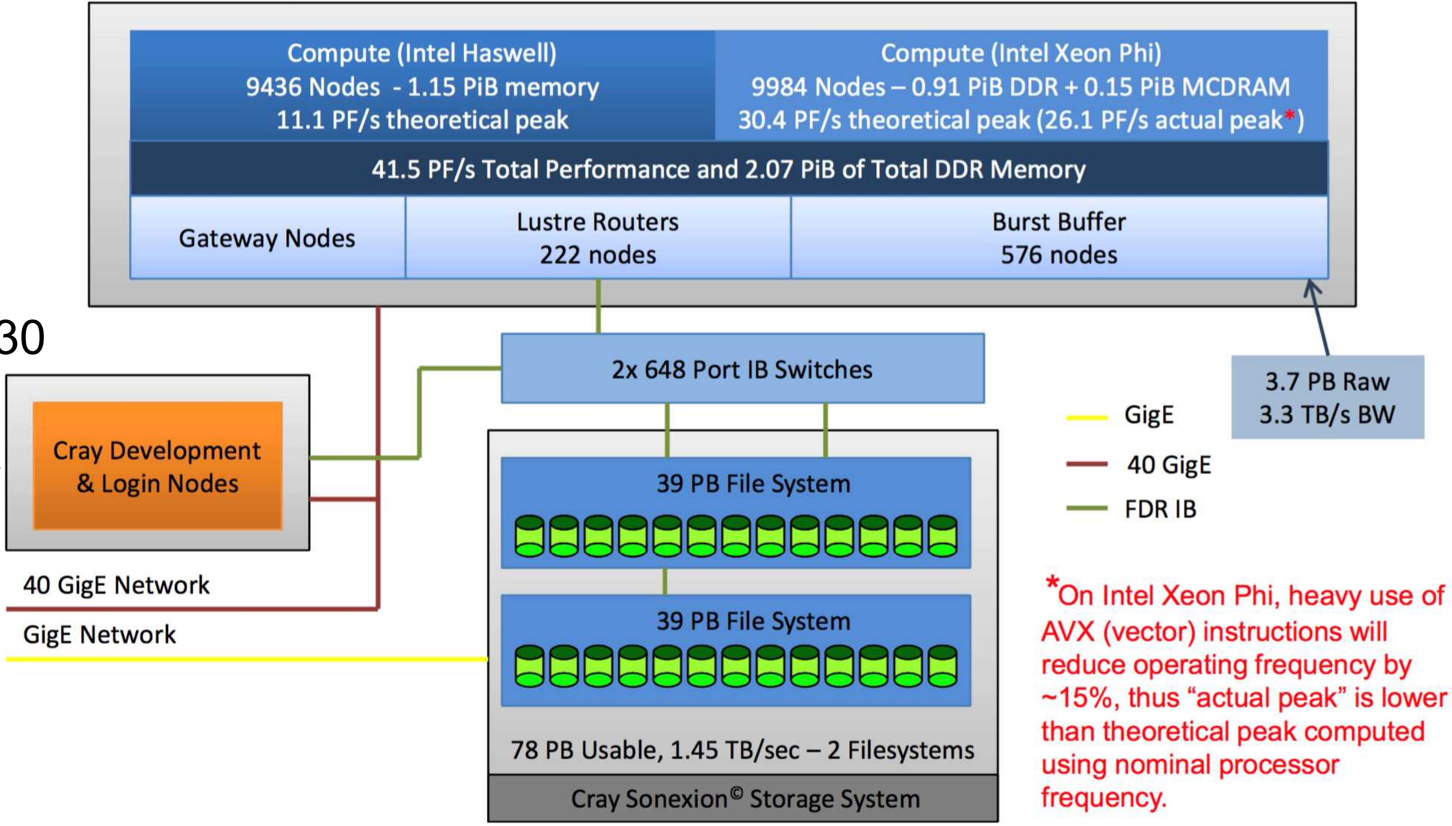
- Communication pattern automatically determined from data effects
  - Broadcast data if shared and read-only access
  - Streaming communication pattern (not yet implemented) if commutative access
  - Shared-memory optimizations for tasks/data in same process

Trinity/Advanced Technology Systems (ATS)-1 is the performance analysis target for the milestone

Haswell: enables support for current ASC/IC programs

KNL: enables emerging architecture, workflow, runtime system research

Cray XC30



Compute (Intel Haswell)
9436 Nodes - 1.15 PiB memory
11.1 PF/s theoretical peak

Compute (Intel Xeon Phi)
9984 Nodes – 0.91 PiB DDR + 0.15 PiB MCDRAM
30.4 PF/s theoretical peak (26.1 PF/s actual peak*)

41.5 PF/s Total Performance and 2.07 PiB of Total DDR Memory

Gateway Nodes

Lustre Routers
222 nodes

Burst Buffer
576 nodes

2x 648 Port IB Switches

3.7 PB Raw
3.3 TB/s BW

Cray Development & Login Nodes

40 GigE Network

GigE Network

39 PB File System

39 PB File System

78 PB Usable, 1.45 TB/sec – 2 Filesystems

Cray Sonexion© Storage System

GigE

40 GigE

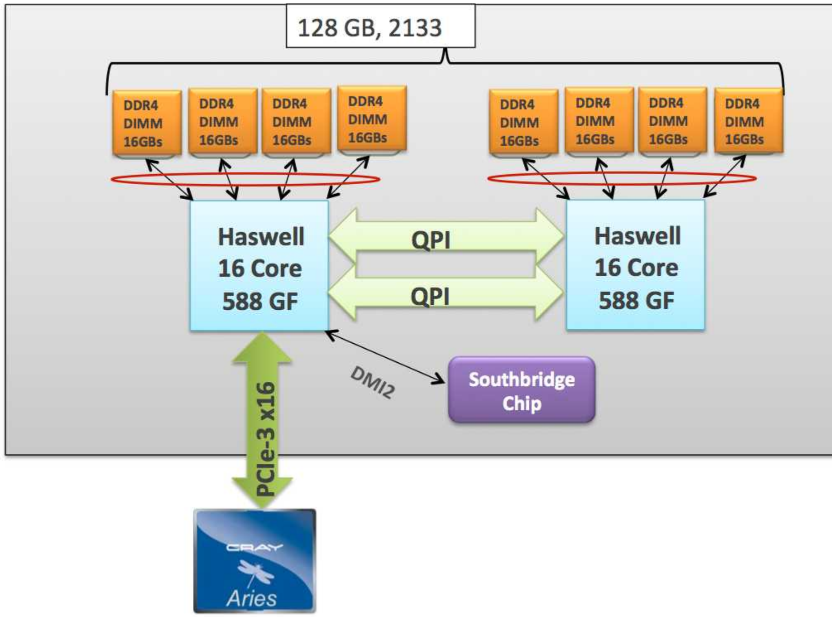
FDR IB

*On Intel Xeon Phi, heavy use of AVX (vector) instructions will reduce operating frequency by ~15%, thus "actual peak" is lower than theoretical peak computed using nominal processor frequency.

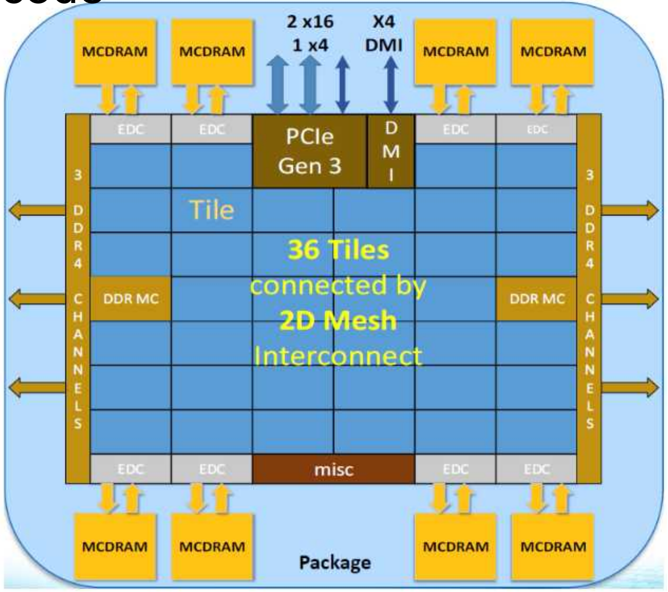
(Image courtesy of ACES)

Performance analysis results are captured for both Haswell and KNL architectures

Haswell should have better serial performance, and perform better on system tasks (e.g., communication)



KNL should do better on highly-parallel, numerically intensive code



(Images courtesy of ACES)

Proxy and benchmark overview

- Three benchmarks
  - Written by DARMA developers
  - Purpose: highlight benefits/limitations of the programming model and runtime
    - Jacobi: memory-bound computation, latency-bound communication to expose overheads
    - Molecular dynamics: compute-bound with more bandwidth-intensive communication to complement Jacobi
    - Simulated Imbalance: assess load balancing capabilities
- Three proxy applications
  - Written by application developers
  - Purpose: co-development of APIs, acquire subjective feedback, requirements
    - PIC: Direct collaboration with EMPIRE application team
      - SimplePIC, MiniPIC*
    - UQ*: Embedded analysis is a capability used by both applications
    - Multiscale*: Ties to IC/Sierra

*exceeds criteria

EMPIRE: ElectroMagnetic Plasma In Radiation Environments

- SNL is developing a new code base for plasma simulations
- Component based approach using the Trilinos framework
- The PIC component of Empire is the basis for our proxy app work
- Two sets on unknowns, mesh data and particles
  - Domain decomposition on the fields and the particles can be out of balance
  - Calculations are localized so colocation is important
  - Work can be created in one location and migrate to a different location
- Potential solution – overdecomposition
  - Overdecomposition breaks the problem up into more units than you have computational cores
  - Load balance at a middle level of work
  - Overlap computation and communication

From EMPIRE to MiniPIC and SimplePIC

- MiniPIC is an electrostatic PIC miniapp build on MPI+Kokkos.
- In the scope of this L2, a proxy app SimplePIC was developed
- SimplePIC is a particle move kernel from MiniPIC on a structured mesh
    - MPI based version of SimplePIC was developed for benchmark purposes.
- The current code design flow is: SimplePIC → MiniPIC → EMPIRE.

Co-Design Efforts

- In FY17 EMPIRE and DARMA teams hired Aram Markosyan
  - with computational plasma physics and numerical analysis background
  - shared postdoc
  - bi-weakly meetings with EMPIRE team
  - Tightly integrated with DARMA team
- Aram's role was to intensively communicate and represent the needs of EMPIRE team in the DARMA design processes
  - Designing and developing SimplePIC proxy app
  - SimplePIC and the DARMA backend were built up together this year
  - Every single new and experimental feature of DARMA was first tested on the SimplePIC (performance/productivity feedback)

Impact of tightly coupled collaboration

- Made DARMA a more performant, productive, feature rich and robust programming model

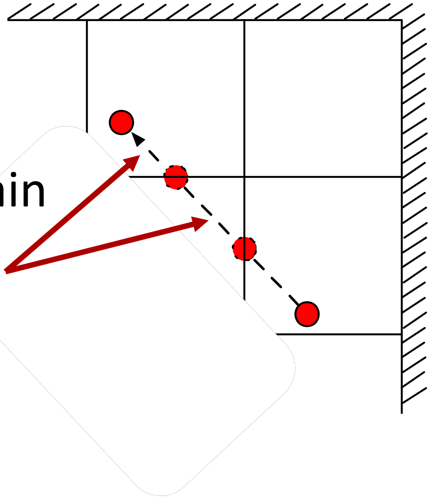
- Enabled app developer to look at PIC problem from completely new perspectives

SimplePIC Proxy Overview

- PIC method allows the statistical representation of general distribution functions in phase space
- It uses the fundamental equations retaining the full nonlinear effects
- SimplePIC includes only particle move kernel
- Domain Decomposition: 2-level 3D structured grid
  - $P_x \times P_y \times P_z$ grid of boxes (patches), $n_x \times n_y \times n_z$ grid within each box
- Computational costs:
  - $O(N_{particle})$ computation (memory bound), $O(N_{particle} \times patch_{surf}/patch_{vol})$ communication,
- Proxy goal: serve as test ground for PIC algorithm design and development on DARMA

SimplePIC Proxy Algorithm

- Decompose problem into patches and assign them to processing units
- For every patch initialize the swarm (particles on that patch)
- For each time step do (**iteration**)
  - For each particle in the swarm do
    - Advance particle until it reaches the patch interface or time expires
    - If time is not expired do
      - Put particle in the migrants (a buffer, corresponding to that patch interface)
      - Remove particle from swarm
  - Compute the total number of migrants in the entire domain
  - While total number of migrants > 0 do (**micro-iterations**)
    - For every patch interface exchange the migrants
    - For each interface do
      - For each particle in migrants do
        - Advance particle until it reaches the patch interface or time expires
        - If time expired add particle to swarm, otherwise put in migrants
    - Compute the total number of migrants



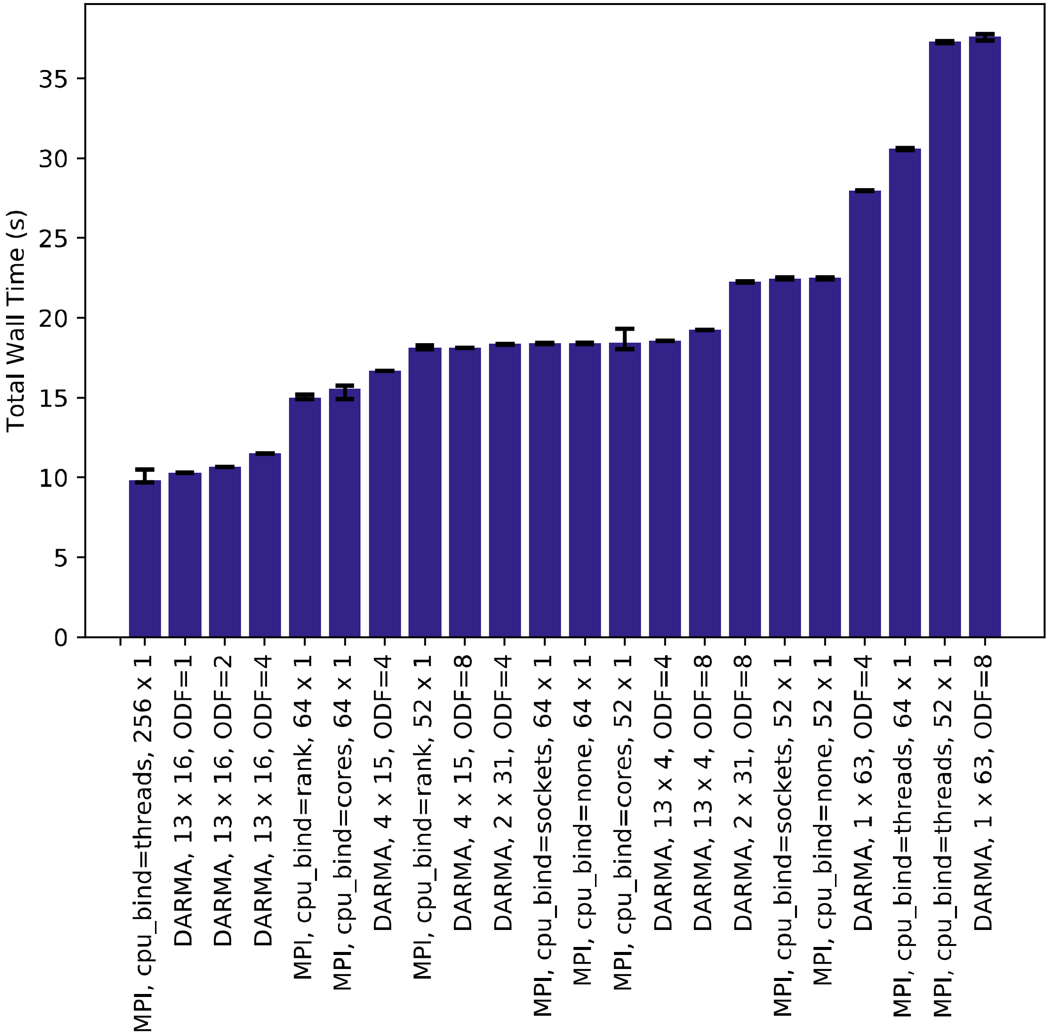
Balanced and Unbalanced SimplePIC Studies

- Balanced use case assesses overheads with respect to MPI-only implementation
  - Every computational cell has N randomly placed particles (5 - 30), with random velocities (|v| = const).
- Imbalanced use case assesses benefits of overdecomposition and load balancing
  - Initially place 80% of particles into the 20% of the domain creating load imbalance in the system.
  - The computational experiment was designed such that the system will reach to a fully balanced state in 500 iterations and come to the initial state in 1000 iterations.
- In all studies we kept CFL number to a value of 0.96, which translates into at most 2 micro-iterations per time step.

KNL architecture provides many possibilities for on-node parallelism

- Empirical exploration of cpu-binding and affinity tradeoffs
- Increasing number of communication threads/node
  - Fewer threads available for computation
  - Communication is driven forward more quickly
- Increasing number of hyperthreads/core
  - More threads actively computing
  - Potential cache conflicts
  - Weakened serial performance per thread
- CPU binding options
  - Binding tasks to physical cores only or to specific hyperthreads

A CPU binding and affinity study determined proper settings on KNL for SimplePIC



A variety of settings were tested for MPI and DARMA.

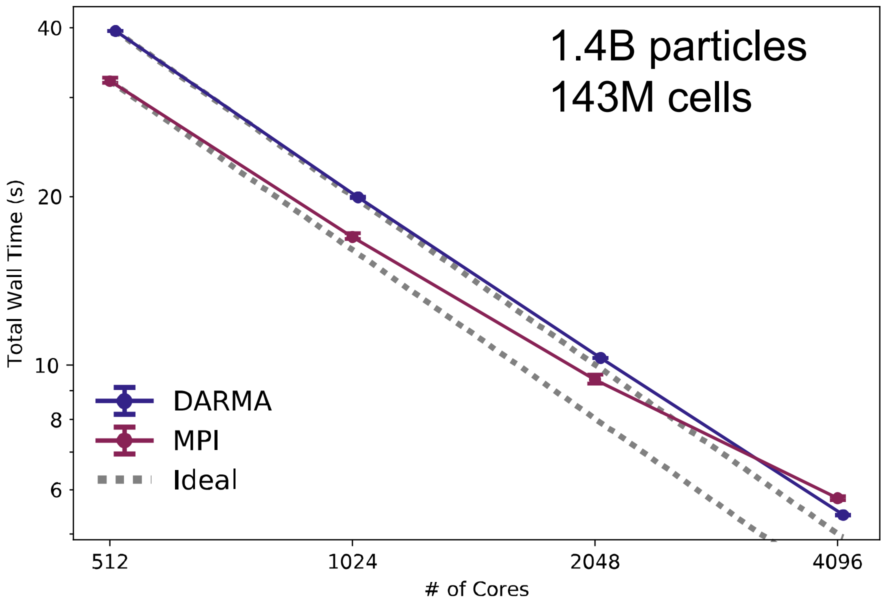
Optimal settings:
MPI: 4-way hypertheading with cpu_bind = threads

DARMA: 13 processes per node, each with

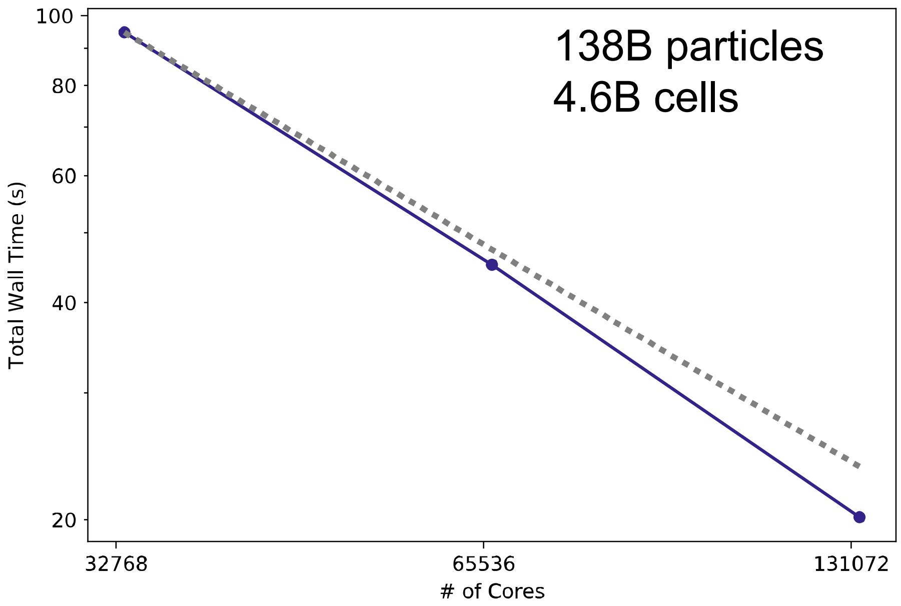
- 16 compute threads (4 compute cores)
- 1 communication thread

Strong scaling of balanced SimplePIC up to 131K cores/2K nodes (KNL)

Mutrino (KNL, 4K cores)



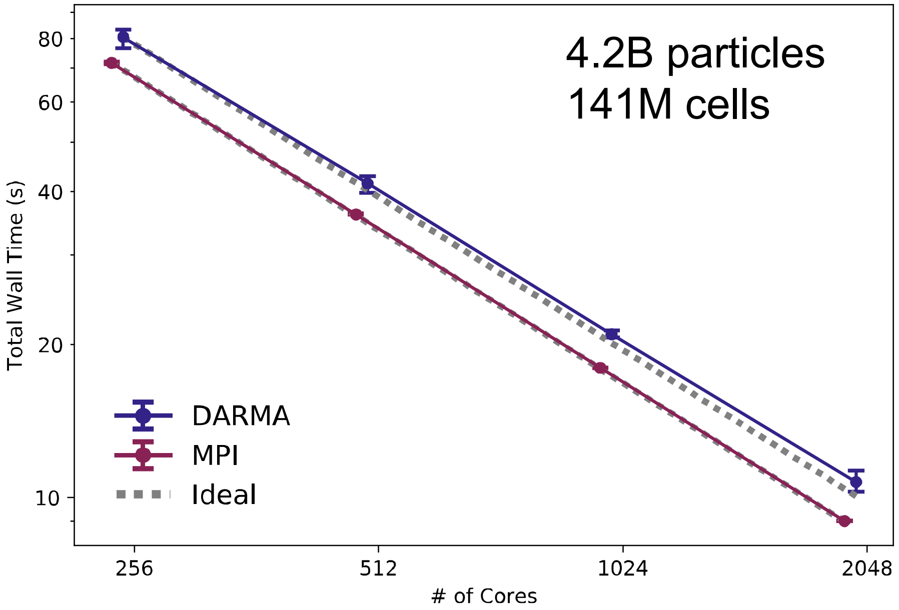
Trinity (KNL, 131K cores)



- DARMA overhead with respect to MPI is -5-24%.
- On 2K cores, grain size is too small and, hence, degraded scaling.
- MPI scaling degradation is likely due to MPI only launch on KNL.
- DARMA scales super-linearly up to 131K cores.

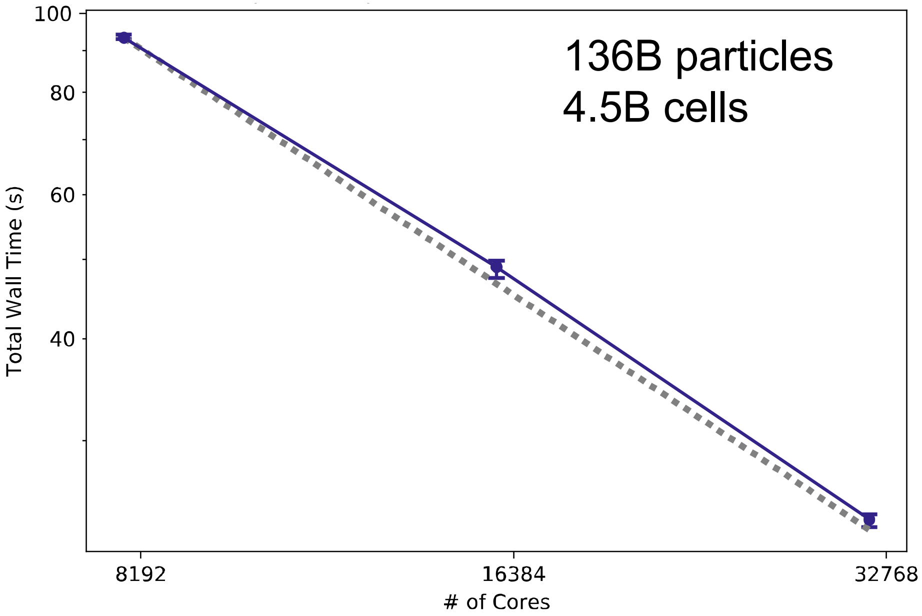
Strong scaling of balanced SimplePIC up to 32K cores/2K nodes (Haswell)

Mutrino (Haswell, 2K cores)



- DARMA overhead with respect to MPI is 12-19%.
- On 2K cores, grain size is too small and, hence, DARMA does not have perfect linear scaling.
- MPI scales ideally on up to 2K cores.

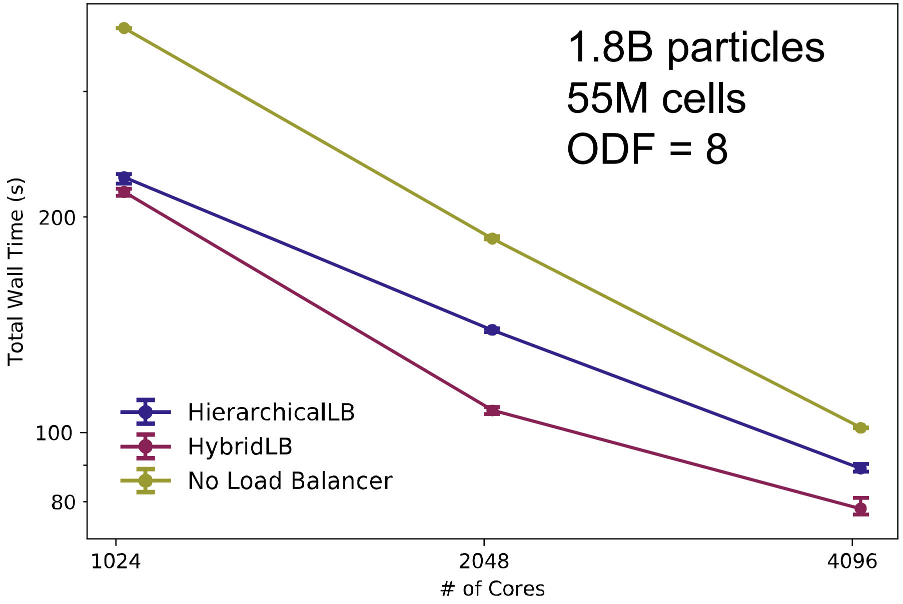
Trinity (Haswell, 32K cores)



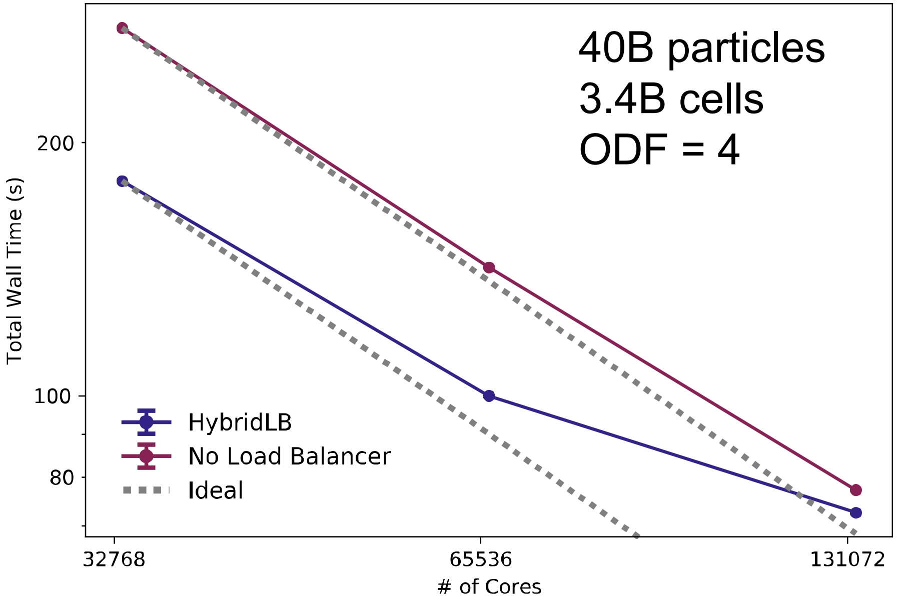
- DARMA scales consistently good on up to 32K cores.
- Slight overheads can be explained by the small problem size on higher core counts.

DARMA Strong scaling of imbalanced SimplePIC up to 131K cores/2K nodes (KNL)

Mutrino (KNL, 2K cores)



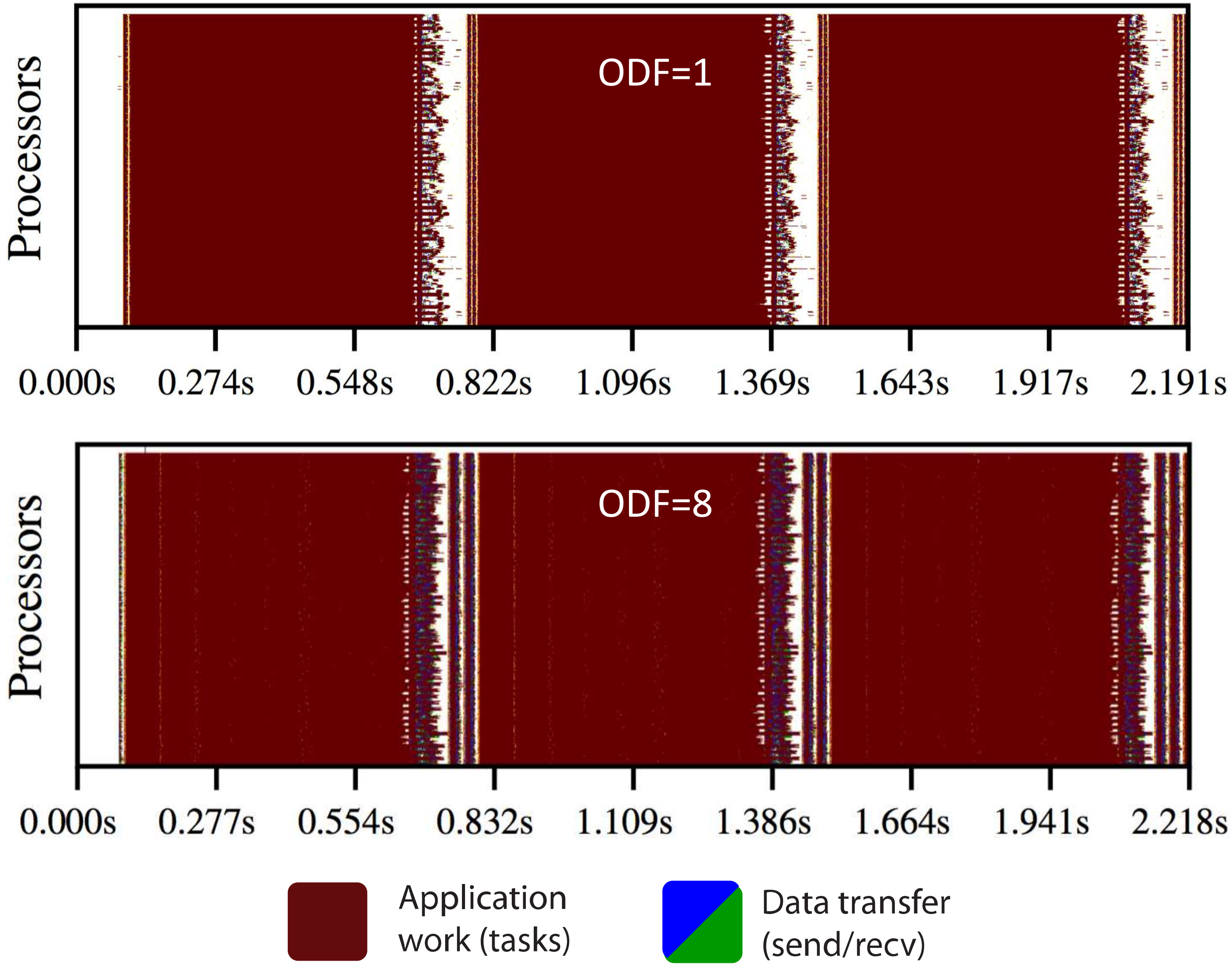
Trinity (KNL, 131K cores)



- For lower core counts, load balancing provides around 50% speedup.
- For higher core counts, at least at this overdecomposition level, speed up due to a load balancer is 20%.
- These trends are similar for Haswell.

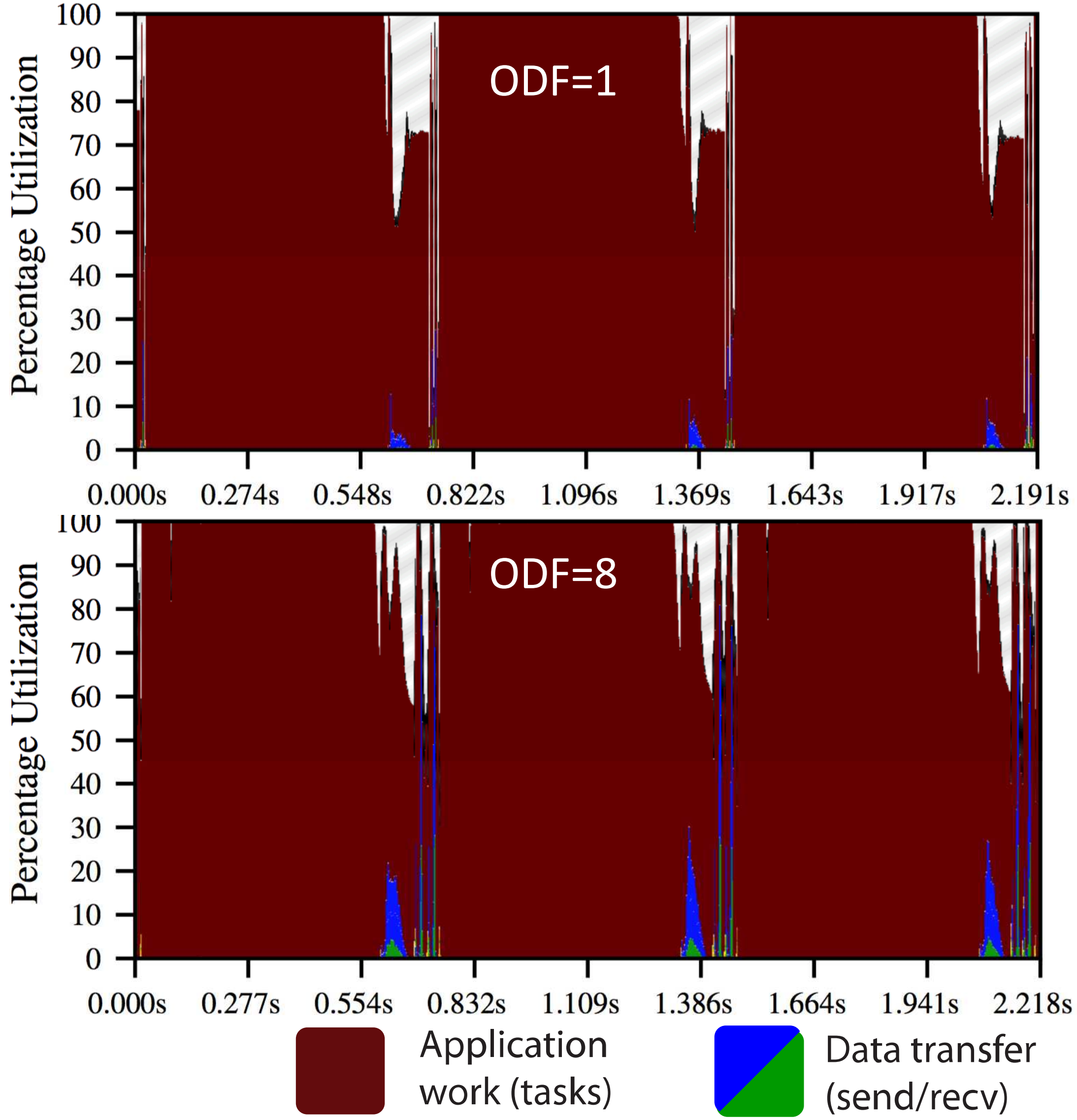
- Similar trends are present on Trinity at these higher scales.

DARMA Time Profile Graph of Balanced SimplePIC on 2k Cores/64 nodes (Haswell) for 3 Iterations



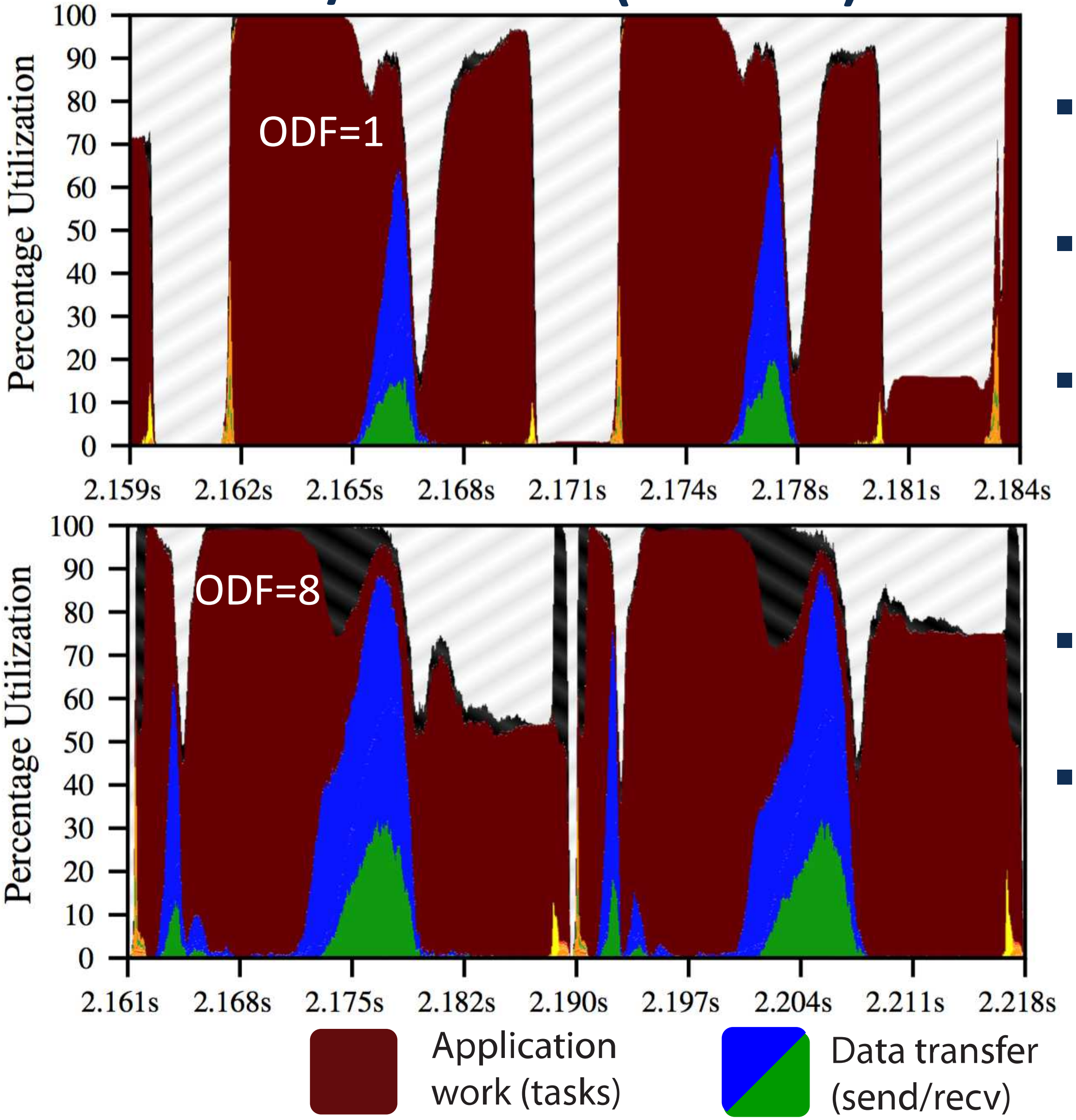
- x-axis is time and y-axis are different cores
- Most of the time is spent executing application tasks
- There is a small amount of idle time (white) at the end of each iteration

DARMA Percentage Utilization Graph of Balanced SimplePIC on 2k Cores/64 nodes (Haswell) for 3 Iterations



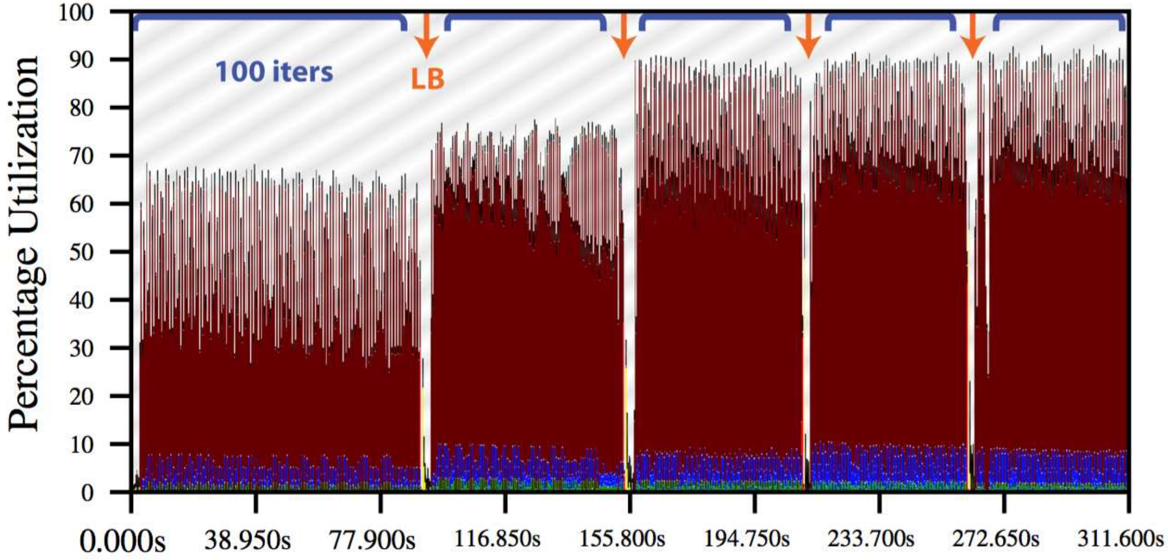
- x-axis is time and y-axis is the proportional aggregate of work type spent across the worker cores
- With an overdecomposition factor of 8 (ODF=8) the data transfer time is slightly increased
- The idle time at the end of the iteration is slightly reduced with ODF=8 because the system is able to overlap communication with computation

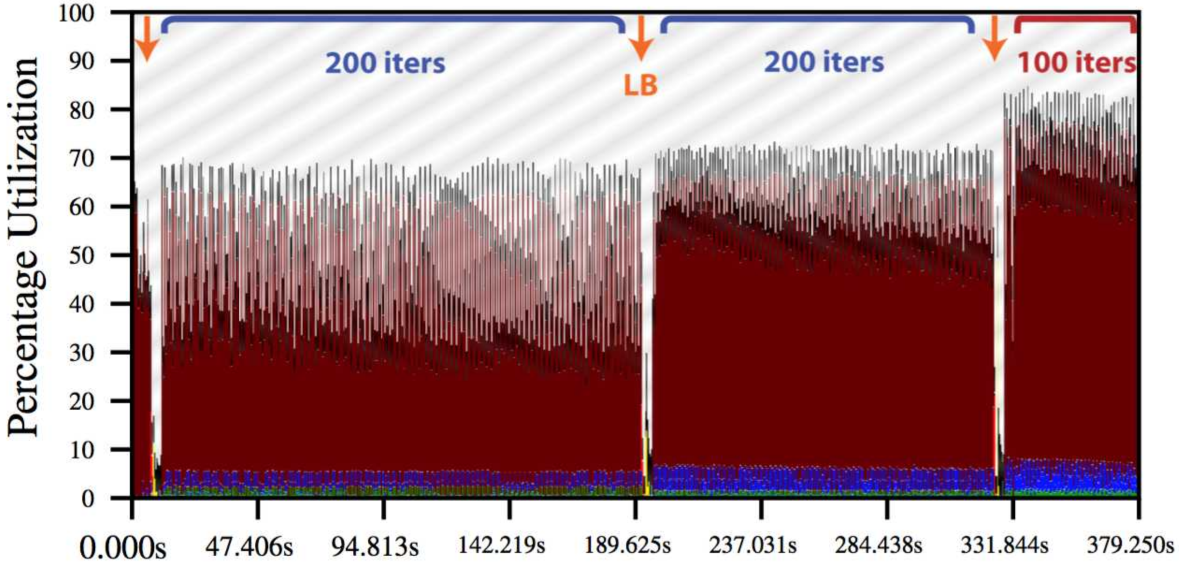
DARMA Time Profile Graph of <u>Balanced</u> SimplePIC on 2k Cores/64 nodes (Haswell) for last 2 micro iterations



- Processor utilization for 2 micro iterations
- Note the scale: this is 25 milliseconds
- Overdecomposition increases the execution time because data transfer is increased (note the increase in green and blue area)
- More particles must cross the boundaries with smaller boxes
- Overall processor utilization is increased because there is more overlap with communication

DARMA Projection views of imbalanced SimplePIC on 2K cores (Haswell)





- Significant improvement in load imbalance with more frequent calls to load balancer.
- The overhead (cost) of load balancer is essentially constant.
- Over 50% CPU utilization increase after the first load balancer call (in both cases).

Conclusions on SimplePIC Performance Study

- Balanced SimplePIC study stressed DARMA overheads with respect to MPI. In the worst cases we are off by 25%.

- Balanced SimplePIC also showed excellent scalability on 131K cores (2K KNL nodes).

- Imbalanced SimplePIC demonstrated the benefits of overdecomposition and load balancing on 131k cores (2K KNL nodes), while maintaining strong scalability.

Lessons learned on productivity for SimplePIC proxy

- “Manual (dynamic) overdecompositon and load balancing in MPI can be very tedious and error prone task even for structured PIC. For unstructured case, the situation is very complex.”
- “Data decomposition in DARMA provides intuitive mechanisms for work load balancing, while runtime handles scheduling.”
- ”DARMA abstractions are fairly intuitive and provide a productive environment for code design and development.”

Quotes from application developer

From SimplePIC to MiniPIC (and to EMPIRE)

- As designed, SimplePIC serves as a test ground for a algorithmic exploration for MiniPIC (EMPIRE).

- MiniPIC DARMA work is in progress
  - Move kernel DARMA-tized
  - DSMC kernel in progress
  - FY18 efforts will focus on full DARMA+Kokkos PIC kernels in MiniPIC for uptake into EMPIRE

Uncertainty Quantification (UQ)

- UQ is identified as one of the main application-driven exascale targets due to:
  - growing importance and impact on predictive modeling and simulations, e.g. reliability analysis
  - challenges due to atypical workloads

- Both ATDM applications have an emphasis on embedded UQ capabilities (e.g., embedded sampling, sensitivity analysis, V&V)

- Sampling-based UQ methods are the most common approaches currently used for exploring UQ in large application codes.

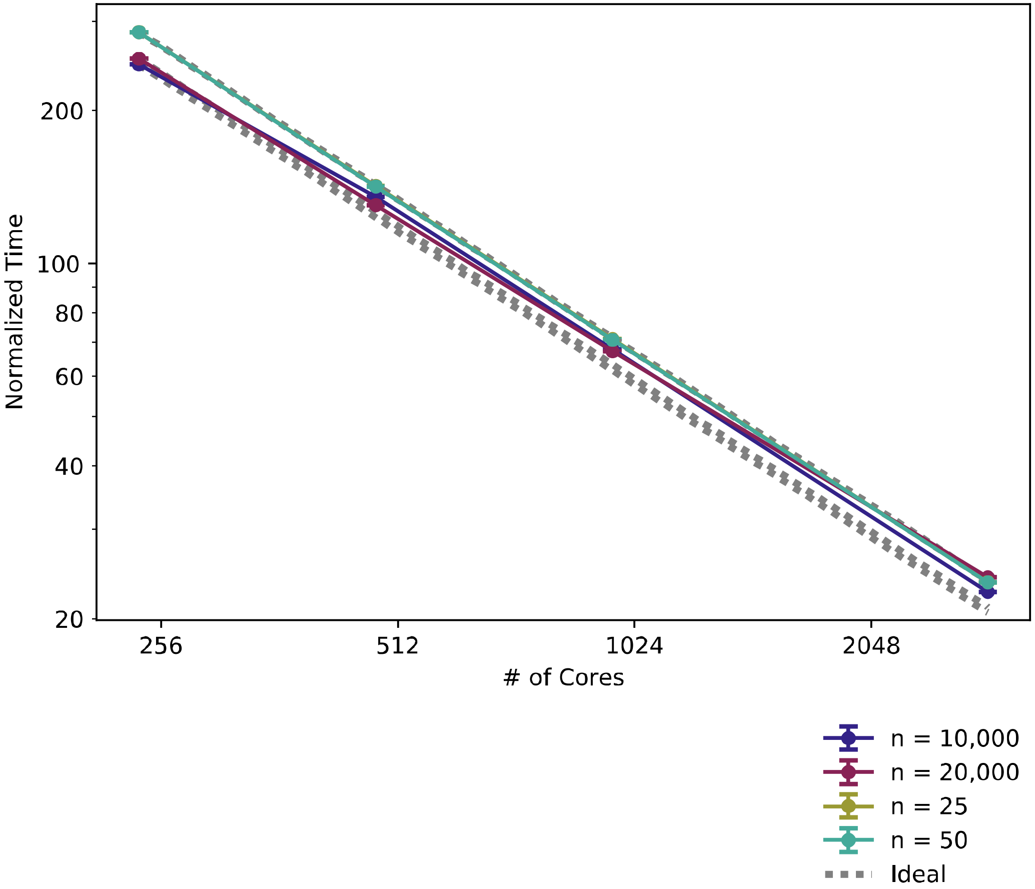
- As part of FY17, we have started exploring how to tackle UQ in DARMA.

Two Demonstrators Developed within UQ

- Monte Carlo Analysis for 1D stochastic diffusion equation
  - Challenges and features:
    - Scheduling/mapping of O(million) independent tasks
    - No point-to-point communication involved except for global collectives

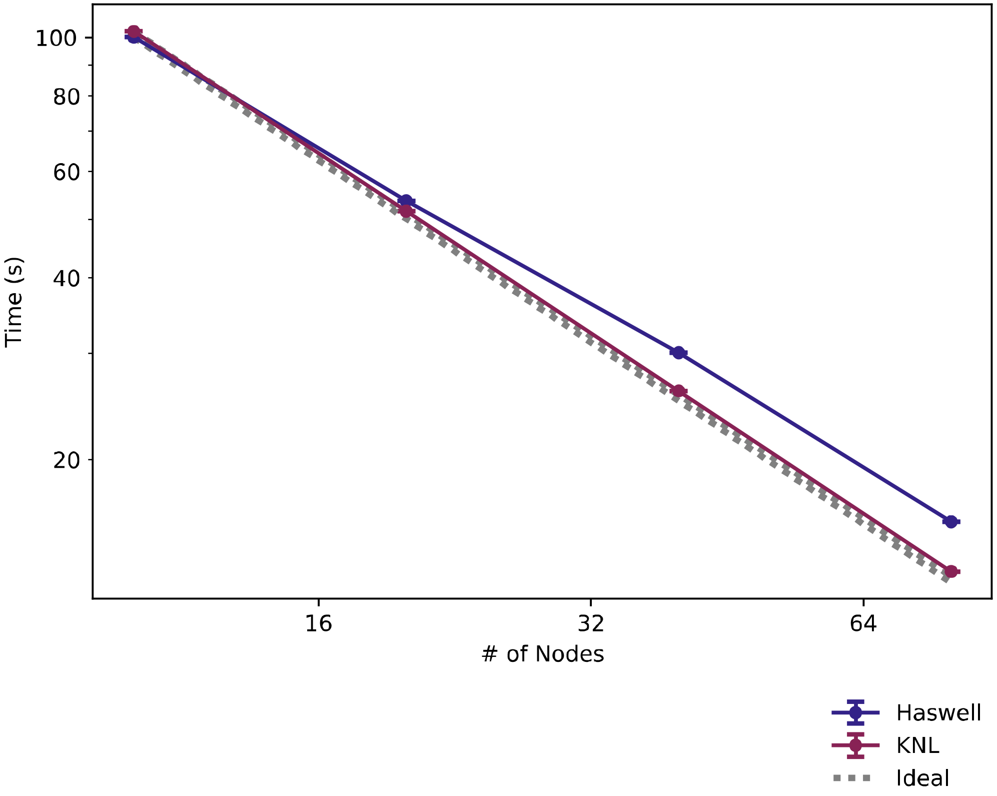
- Multi Level Monte Carlo Analysis for 1D stochastic PDE
  - Challenges and features:
    - Multiple sets of independent tasks of varying computational cost
    - No communication involved except for global collectives
    - Dynamic addition of new levels based
    - Dynamically changing number of samples per level
    - Potentially highly imbalanced application due to inhomogeneous convergence time of PDE solves

DARMA Strong Scaling for Monte Carlo up to 64 nodes (2K cores) Haswell



- n: number of PDE samples per DARMA index
- Total number of samples: N=n*2880, where 2880 is tot # of threads (96 nodes * 30 threads/node).
- Each PDE solve = linear system of 4,194,304 degrees of freedom.
- Good scaling (as expected). Cache/memory effects appear for larger problems.

DARMA Strong Scaling for Multi Level Monte Carlo up to 96 Nodes (Haswell and KNL)



- Adaptive MLMC starting from 4 fixed initial number of levels. Coarsest level has 4096 grid points.
- Workload varies from O(billion) ``small'' tasks for coarse level, to O(100) for finest level.
- Good scaling. Not enough runs to pinpoint the causes behind Haswell trend.

Lessons Learned from UQ Studies

- “Forward problems are typically characterized by/treated with many independent samples, making them a natural fit for AMT models.”
- “Task/data reusability can be a key feature to leverage for sampling methods. E.g.: use as initial condition the final solution of other tasks to potentially accelerate convergence. This is a feature that we have not explored yet, but are planning to within the next fiscal year.”
-
- “Dynamic workload and load balancing for MLMC is a good feature to explore and for testing work scheduling and speculative execution techniques. “
- “Future work will involve more heterogeneous problems: impact of load balancing, dynamic parallelism, optimal task mapping.”

Quotes from application developer

Multiscale Solid Mechanics Proxy

Purpose

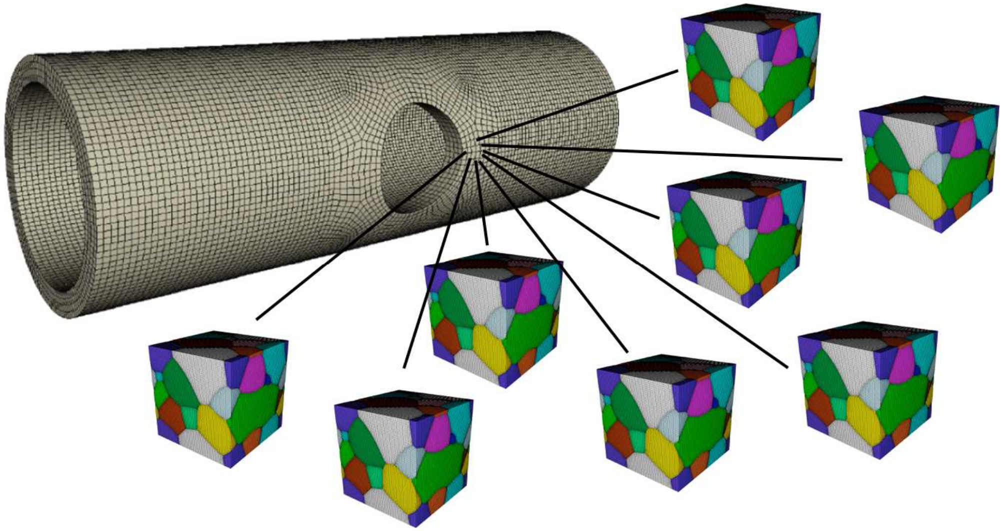
- Investigate the application of DARMA within an engineering analysis code that is representative of ASC IC codes (e.g., *Sierra/SolidMechanics*)
- Evaluate the performance of DARMA against other parallelization strategies (serial, MPI, Tpetra) under a variety of load imbalance scenarios
- Standard geometric partition schemes will lead to large load imbalance

Modeling capabilities

- Lagrangian finite element code for the solution of dynamic solid mechanics problems on non-uniform meshes
- Single-scale and multiscale capabilities (like FE^2)

Multiscale Solid Mechanics Proxy

- FE squared (FE^2) multiscale approach
- Representative volume elements (RVEs) are associated with material points in the macroscale model
- RVE models acts as high-fidelity constitutive models
    - RVE models are solved as independent finite element problems
    - Information exchange between macroscale model and the RVE models



DARMA has the potential to effectively manage the computational complexity of the multiscale problem

Overview of Multiscale Solid Mechanics Proxy

- Application code written from scratch in C++
- Minimal use of third-party libraries, no application-level MPI
- Compartmentalize data structures and procedures for use with DARMA concurrent work paradigm
- Virtually all data stored in simple, serializable containers

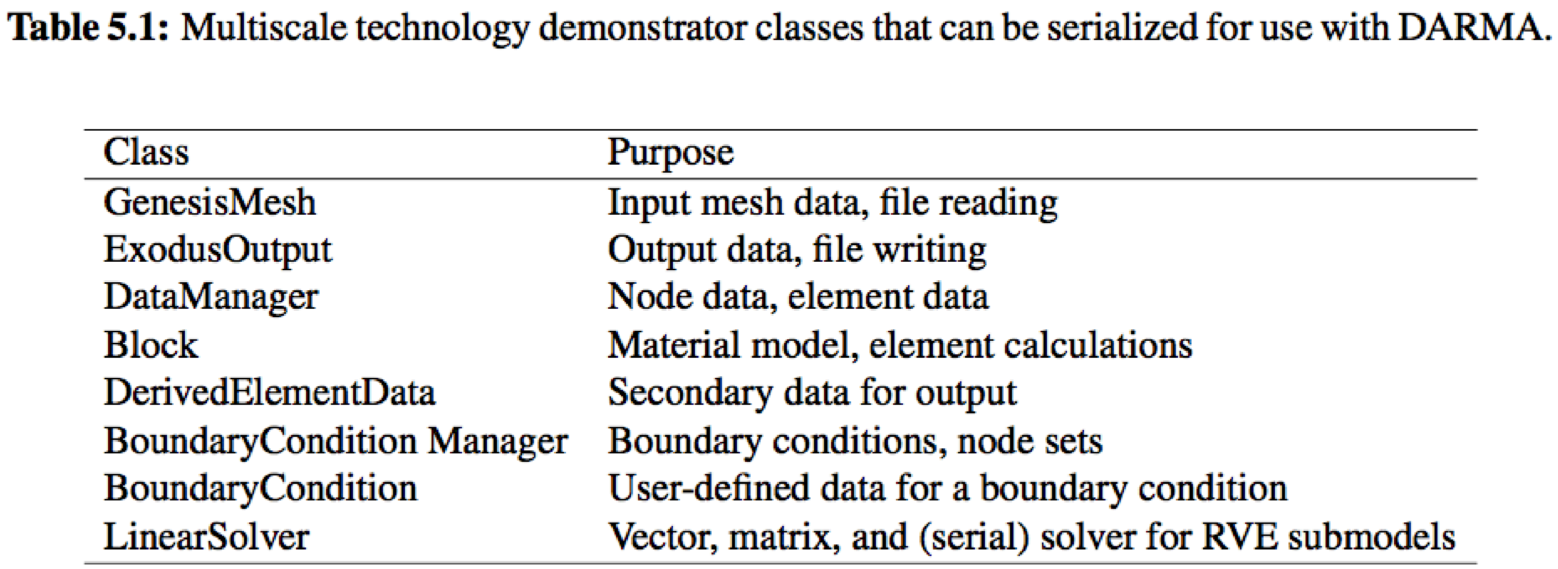


Table 5.1: Multiscale technology demonstrator classes that can be serialized for use with DARMA.

Class	Purpose
GenesisMesh	Input mesh data, file reading
ExodusOutput	Output data, file writing
DataManager	Node data, element data
Block	Material model, element calculations
DerivedElementData	Secondary data for output
BoundaryCondition Manager	Boundary conditions, node sets
BoundaryCondition	User-defined data for a boundary condition
LinearSolver	Vector, matrix, and (serial) solver for RVE submodels

Lessons Learned in Multiscale Solid Mechanics Proxy

- “Adoption of DARMA requires a shift is developer mindset and a revamping of conventional architecture for engineering analysis codes”
- “Moving toward an AMT runtime is best achieved by conceptualizing the application software as a set of tasks with well-defined dependencies”
- Division of labor:
  - “Application developers are responsible for the design and implementation of compartmentalized tasks and data containers”
  - “DARMA is responsible for the execution of tasks, parallelization, and many aspects of code performance”

Quotes from application developer

Summary of quotes on productivity from our application developers

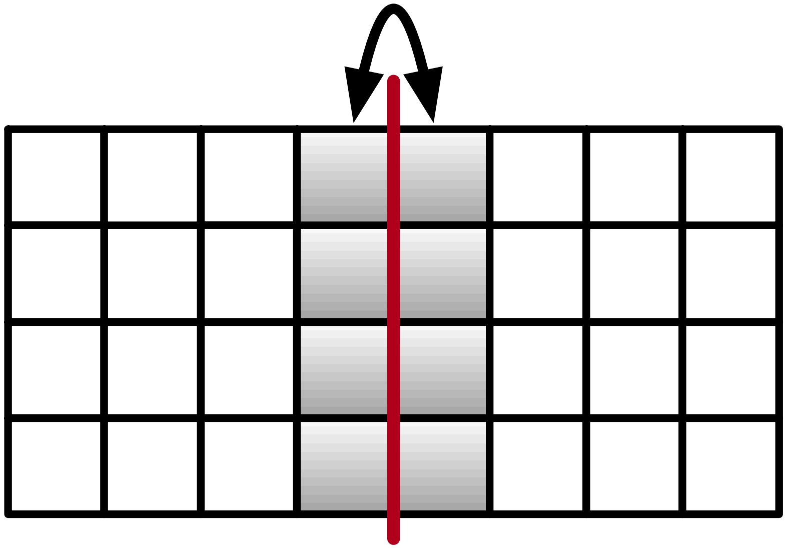
- “DARMA provides an intuitive means to reason about your problem in an AMT way.”
- “Deferred semantics is a significant help for those who are used to imperative programming only.”
- “Moving toward an AMT runtime is best achieved by conceptualizing the application software as a set of tasks with well-defined dependencies”
- “Future work should include focus on documentation and productivity tools (timers, performance profilers, debuggers)”

2D Jacobi Linear Solve Benchmark

- Solve Ax = b derived from basic 2D heat equation

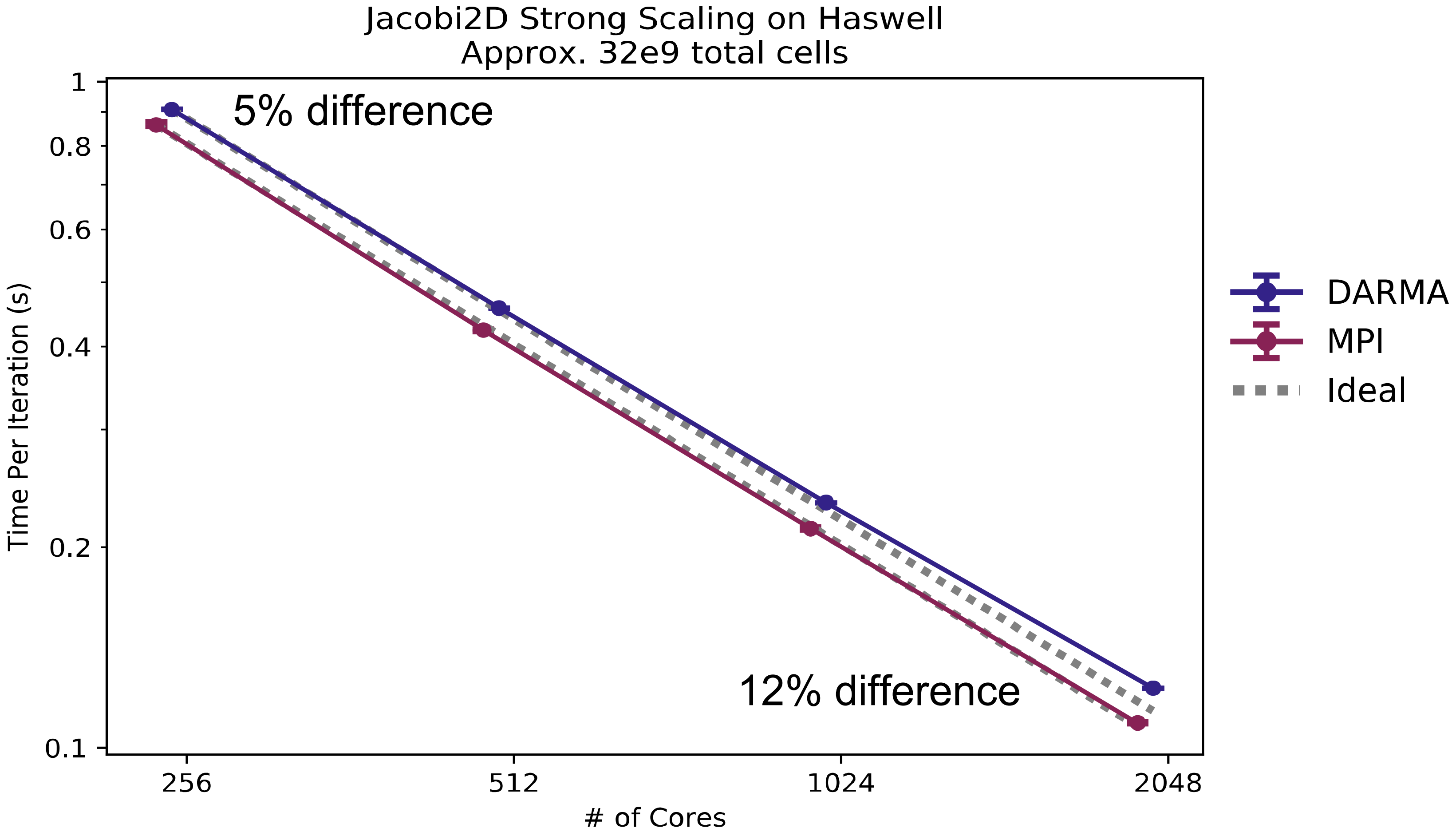
$$x_{i,j}^{(k+1)} = \frac{1}{A_{ij}}\left(x_{i+1,j}^{(k)} + x_{i,j+1}^{(k)} + x_{i-i,j}^{(k)} + x_{i,j-1}^{(k)}\right)$$

- Nearest-neighbor communication of halo region on 2D grid
- 2D stencil computation within a patch



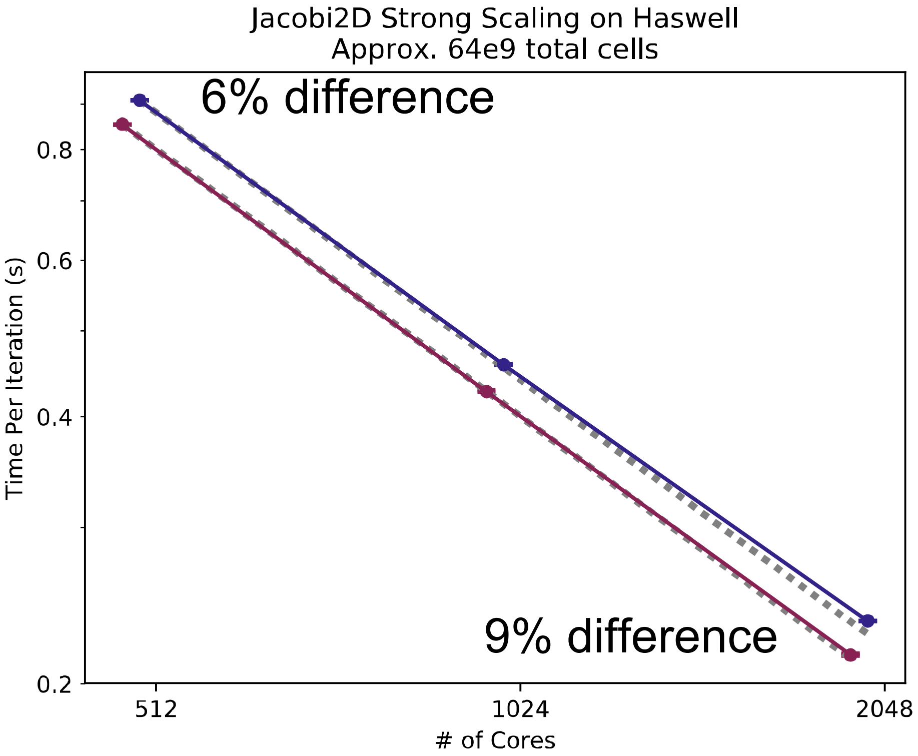
- Byte/flop ratio very high, only a few flops per grid point
- Only small halo region communicated
- Jacobi is *memory-bound* computation, with *latency-bound* communication

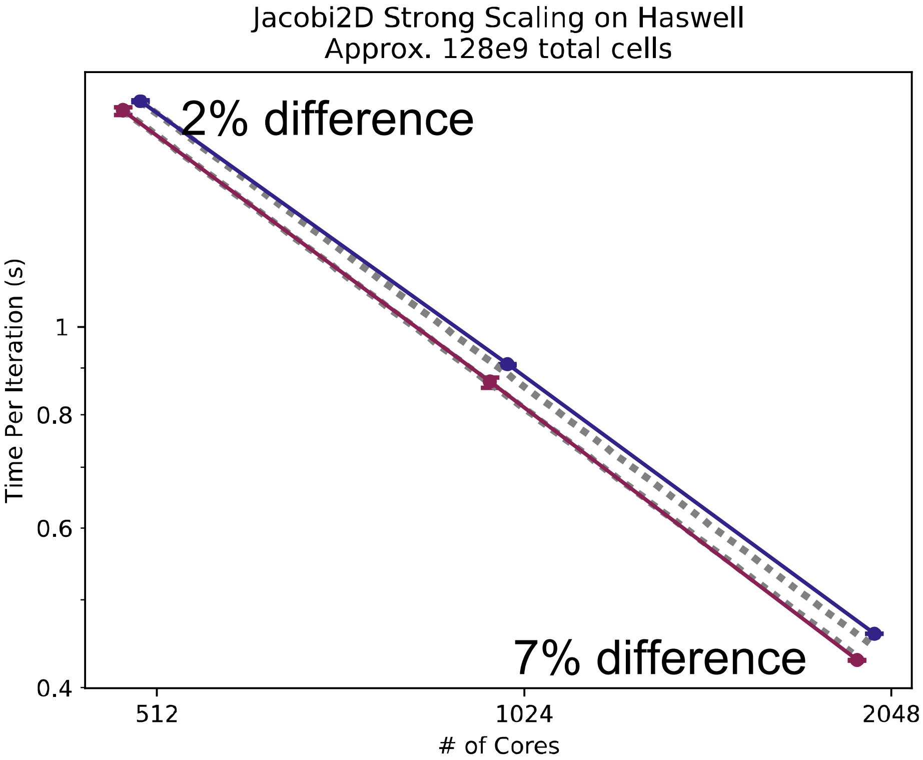
Haswell on Mutrino (64 nodes) shows good strong scaling, relative overhead increases at scale



- Constant overhead (performance difference) relative to MPI
- Overhead percent increases at larger scales

Jacobi2D Haswell (up to 64 nodes) strong scaling trends are consistent across problem sizes

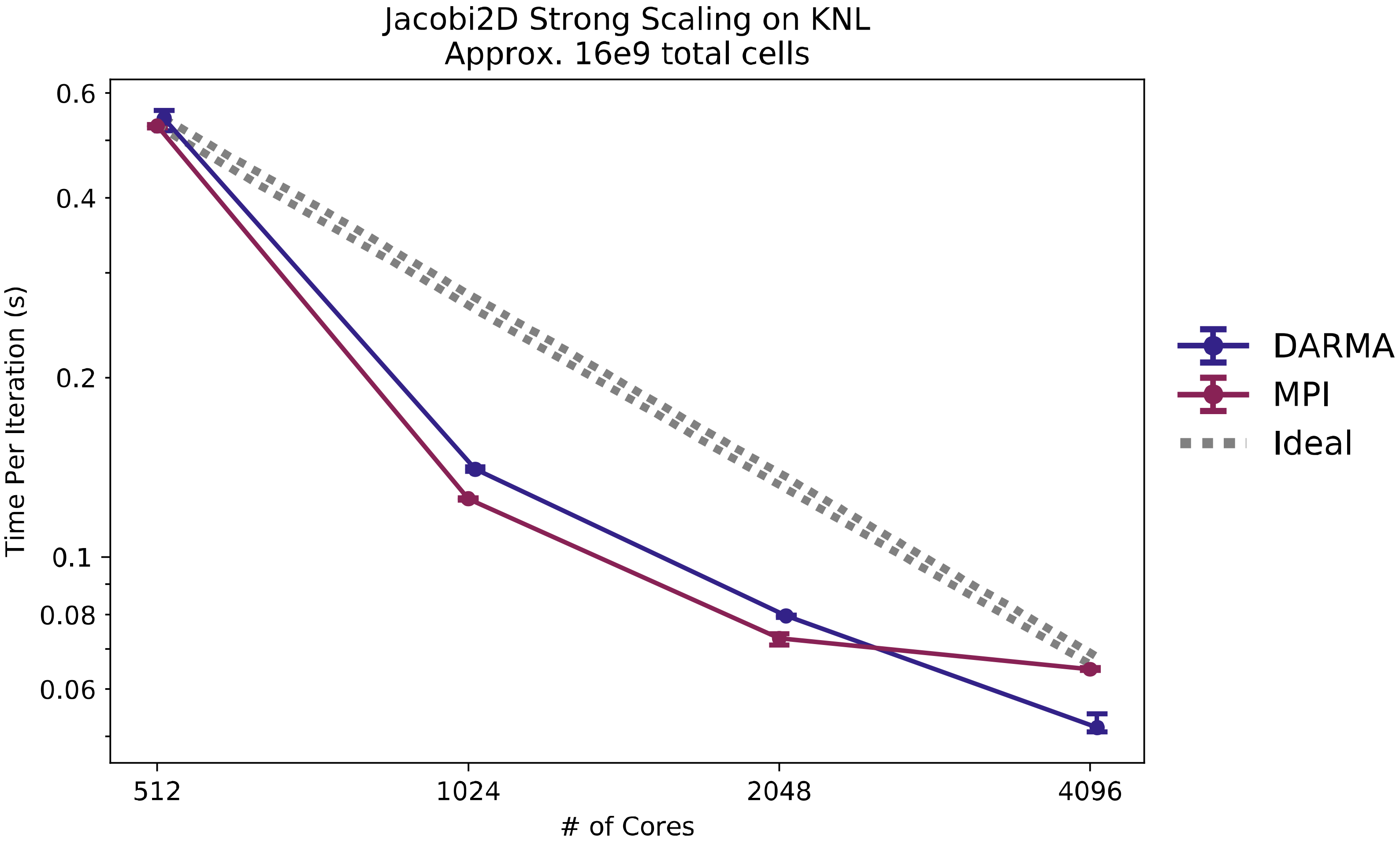




- Percent difference decreases with larger problem sizes (task sizes)
- Larger task sizes better amortize scheduling overheads

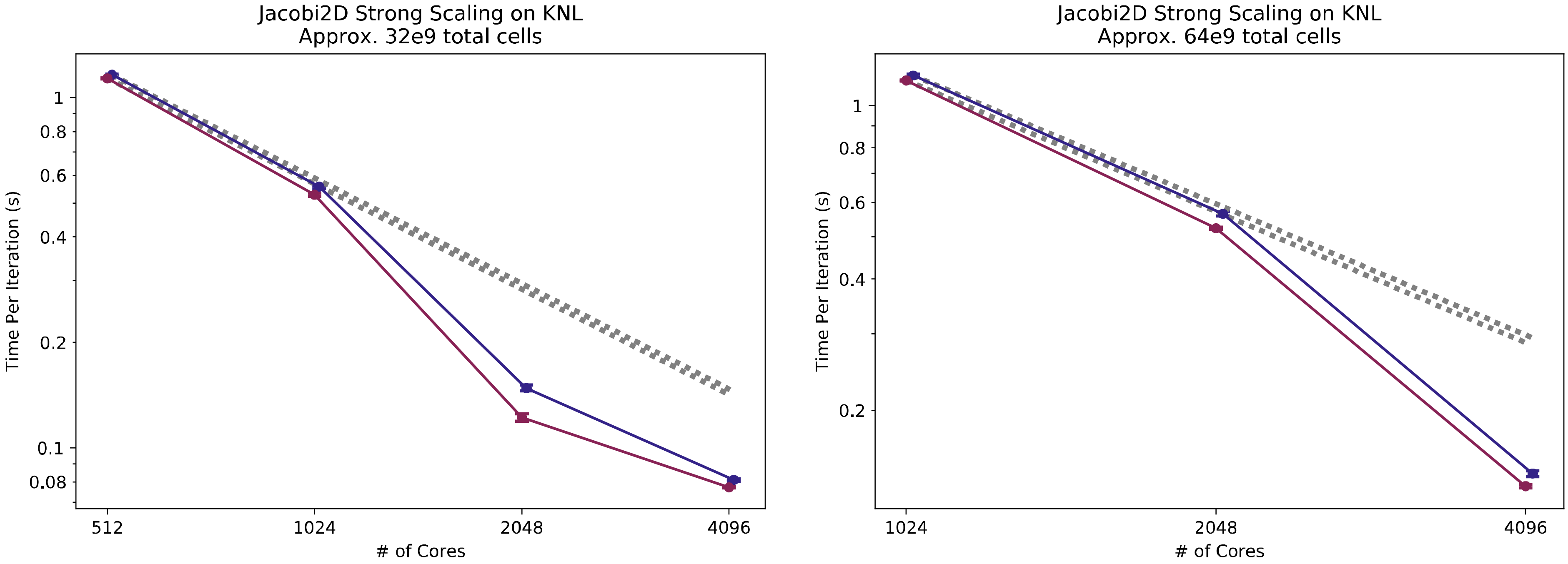
● DARMA
● MPI
■ ■ ■ Ideal

Jacobi2D KNL (up to 64 Nodes/4K cores) shows super-linear strong scaling for both MPI and DARMA



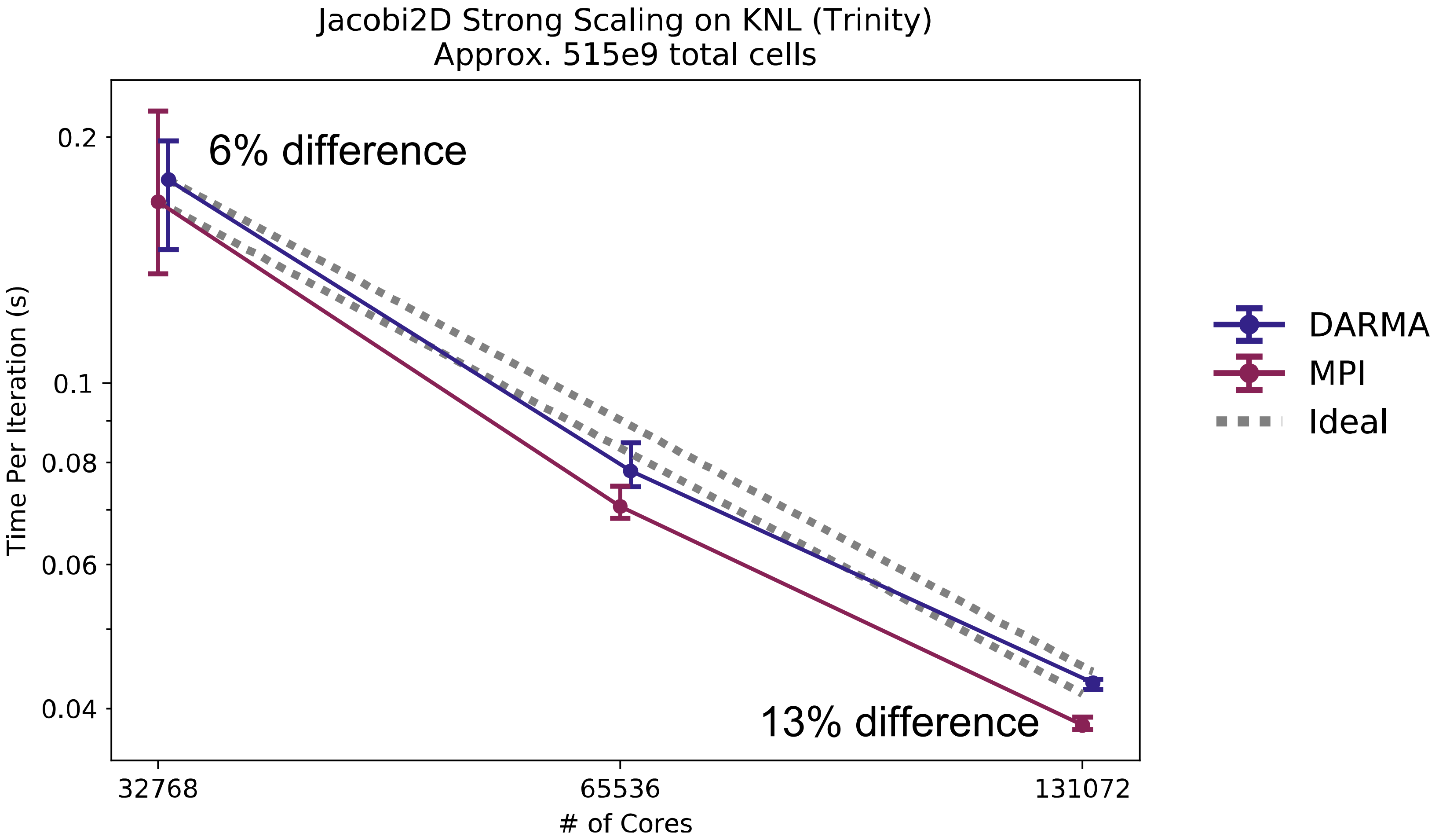
- Memory footprint decreases as problem size per node shrinks at larger scales
- Super-linear effects likely related to better cache/MCDRAM usage

Jacobi2D KNL (up to 64 Nodes/4K cores) strong scaling trends are consistent across problem sizes



- Superlinear discontinuity delayed at larger problem sizes
- Discontinuity corresponds exactly to 16GB threshold for MCDRAM capacity

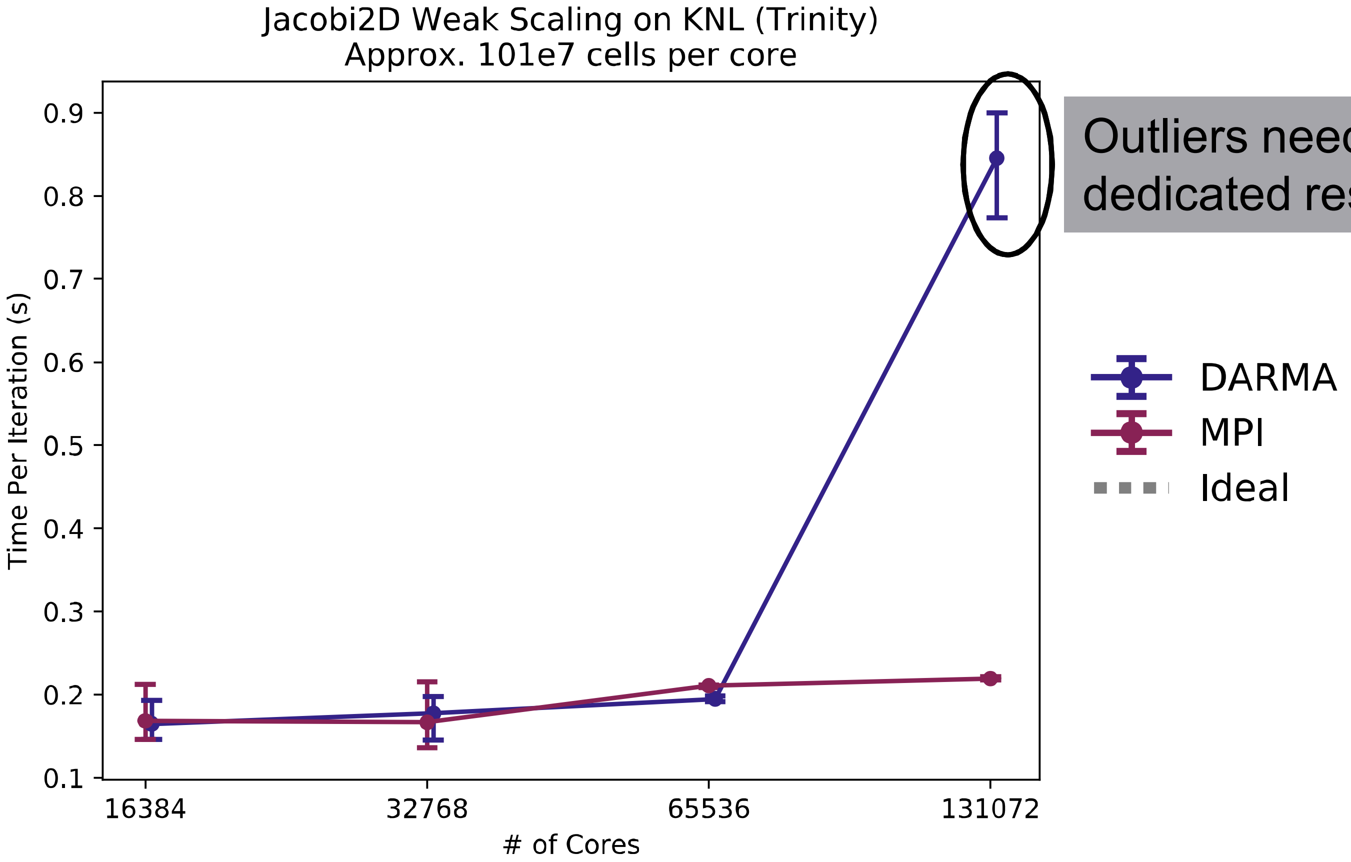
Jacobi2D KNL (up to 2048 nodes/131K cores) also has super-linear strong scaling, relatively constant overhead



6% difference

13% difference

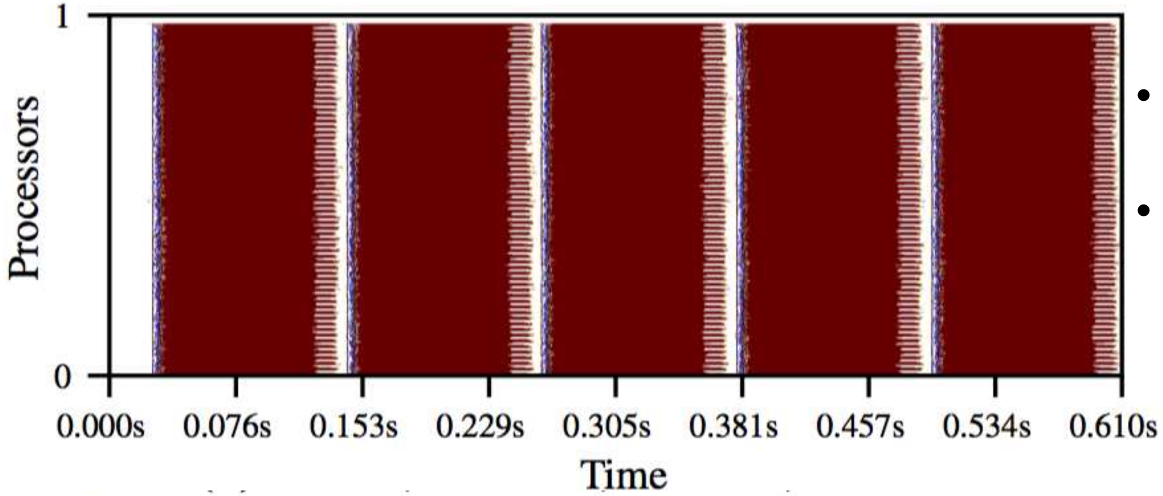
Jacobi2D KNL (up to 2048 nodes/131K cores) weak scaling results consistent with MPI, but with outlier



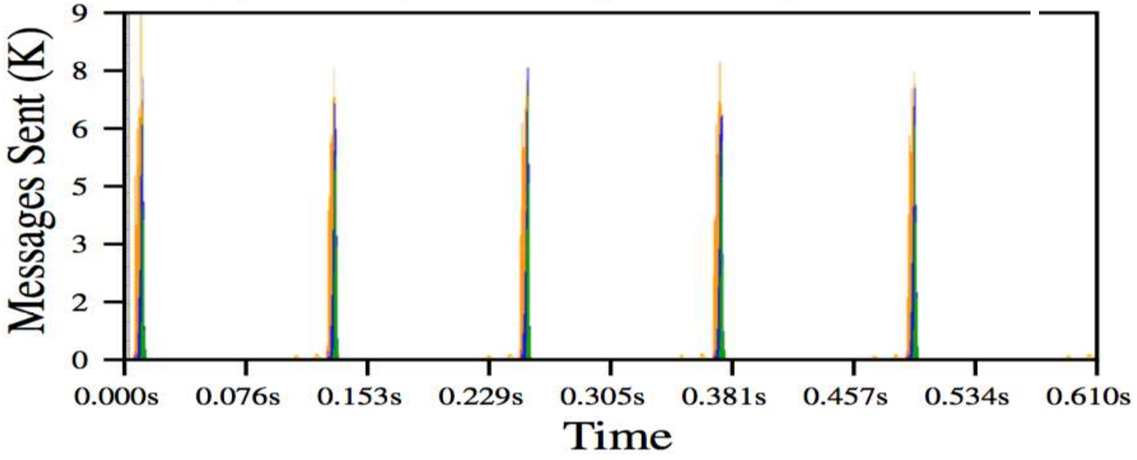
Outliers need additional runs on dedicated reservation

- Outliers were run on congested KNL session on Trinity
- DARMA performance stays relatively flat
- No superlinear benefit from MCDRAM as in strong scaling

All-reduce every iteration limits asynchronous execution, blocks scheduling new computation/communication



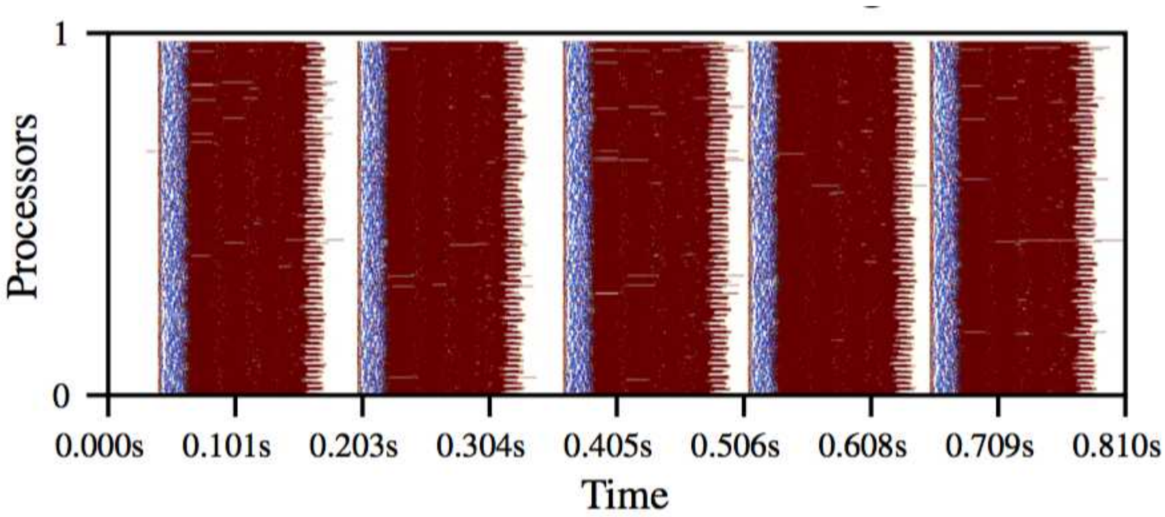
- Red is active computation, white is idle time
- Execution shows intermittent stalls between iterations



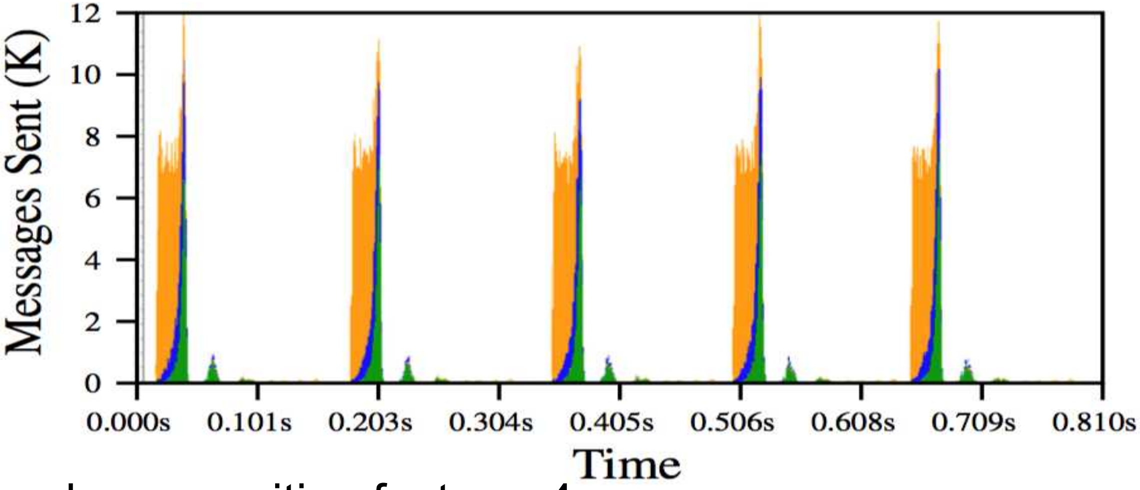
- Peaks show messages sent in a time interval
- Communication becomes bursty as messages wait for convergence check

Overdecomposition factor = 1

Overdecomposition improves bursty communication, makes idle times worse performance



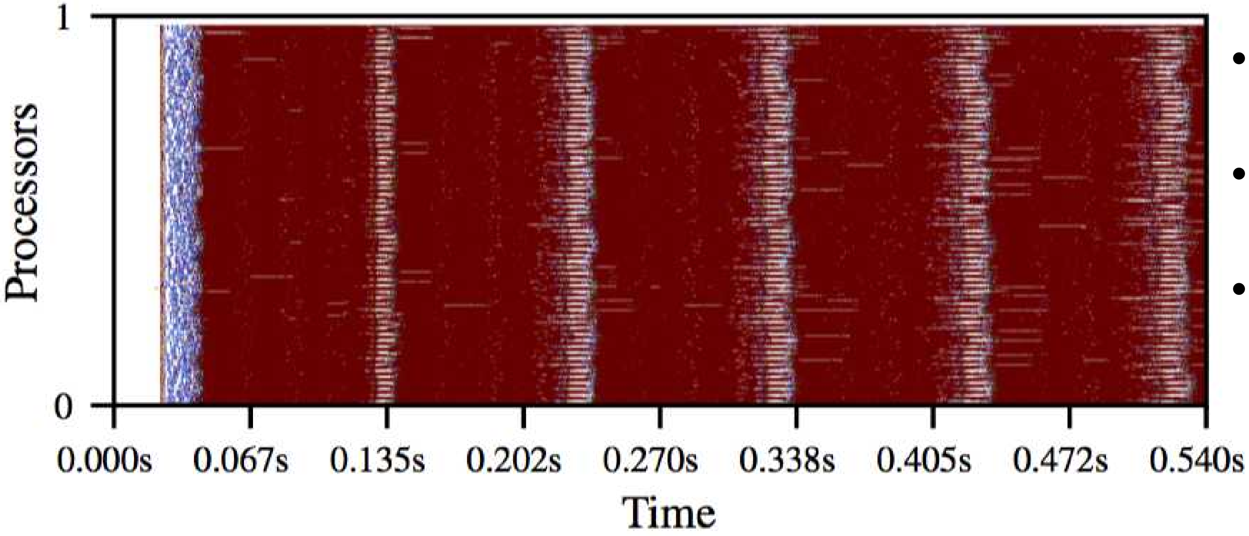
- Red is active computation, white is idle time
- Execution shows intermittent stalls between iterations
- Overdecomposition makes all-reduce more expensive, increasing idle time



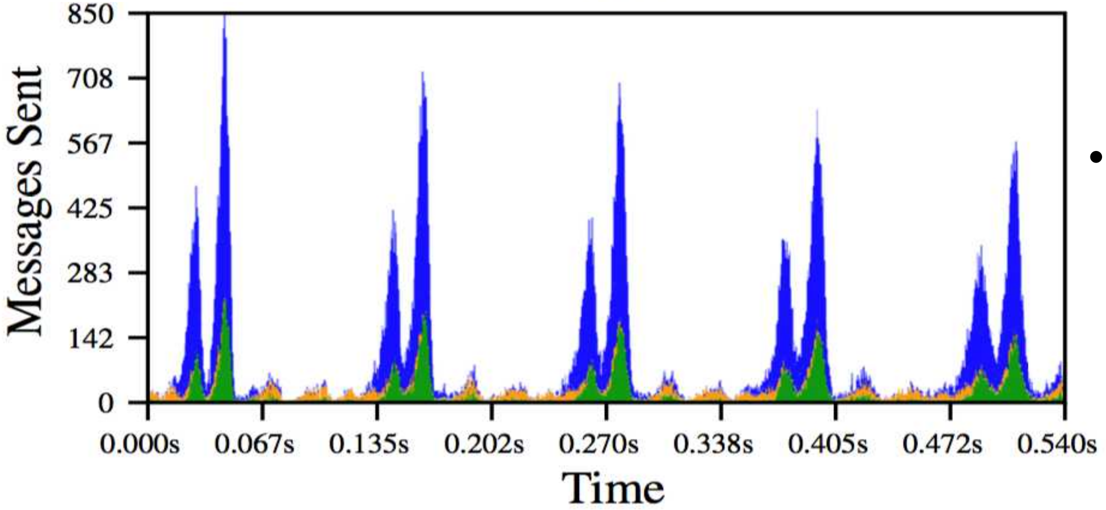
- Communication overlap with overdecomposition makes less bursty messages
- Cannot compensate synchronization cost of all-reduce

Overdecomposition factor = 4

Simulate "speculative execution", perform all-reduce every N iterations. Idle time shrinks significantly.



- Red is active computation, white is idle time
- All-reduce only done every 10 iterations
- Stalls in execution is much less pronounced



- Communication overlap best when combining less frequent all-reduces with overdecomposition

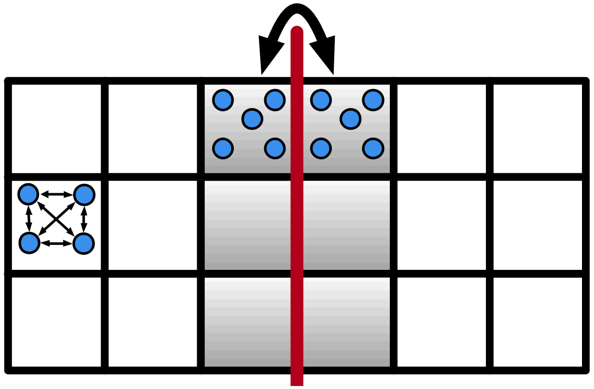
Overdecomposition factor = 4

Lessons learned on productivity for Jacobi benchmark

- Tunable granularity, overdecomposition, communication overlap occurs naturally in DARMA with no additional work
  - MPI without overdecomposition is *not* difficult to write, but lacks any significant overlap of communication and computation
  - MPI with overdecomposition is *very* difficult to write, requires error-prone or inefficient use of MPI_Test or MPI_Waitany
- Jacobi memory-bound, needs *tiling* more than communication overlap
  - Some tiling naturally occurs in DARMA with tunable granularity, but auto-tuning tiling optimizations to Mutrino not performed
  - Tiling to L1 cache size is fine-grained, difficult for DARMA to do but theoretically possible without app changes given declarative model
- Non-optimized collectives combined with conditionals (while loop) limited DARMA scheduler performance
  - Forced DARMA code changes (check convergence every N iterations)
  - Developers should not be refactoring to optimize code with conditionals
  - Better collectives, speculative execution needed in runtime

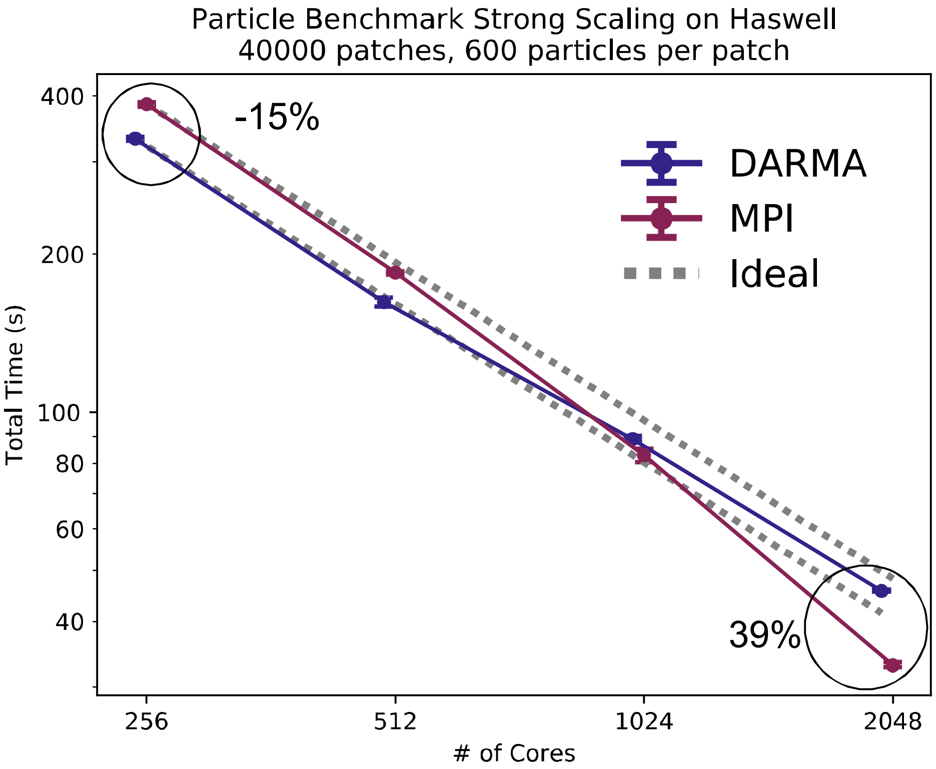
Molecular Dynamics Benchmark Overview

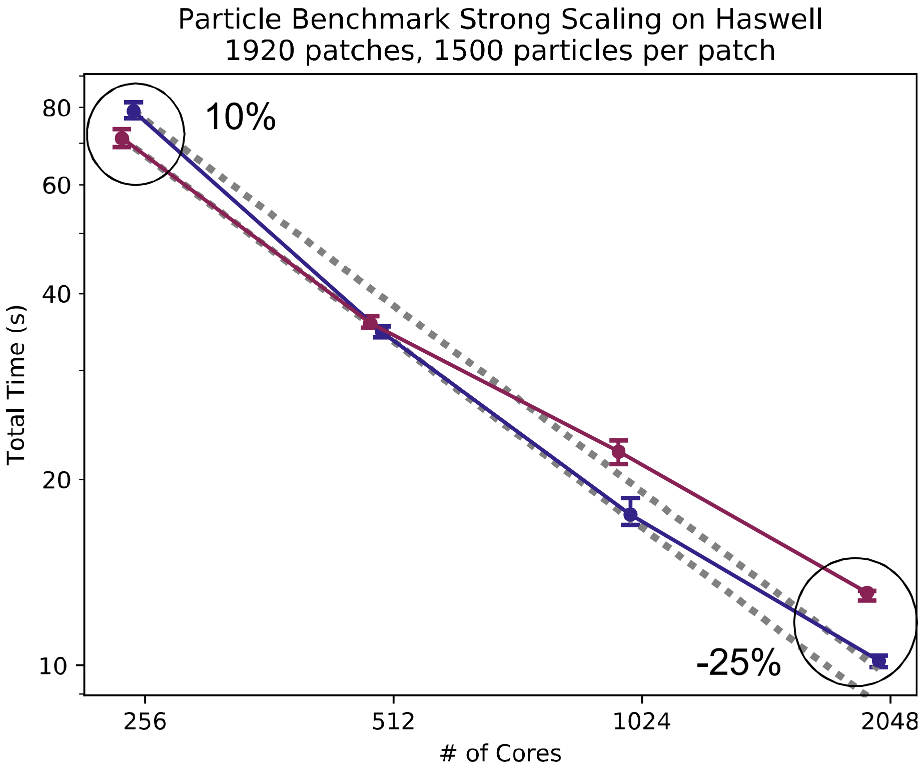
- Stages:
  - Exchange particles in neighboring cells (communication)
  - Compute pairwise forces (Lennard-Jones potential) between all neighboring particles
  - Accelerate particles and update atom positions
  - Migrate particles that move outside their original cell (communication)



- N particles per box: O(N) communication O(N^2) computation
- Benchmark goal: compute-bound with more bandwidth-intensive communication to complement Jacobi2D

Haswell (up to 64 nodes/2K cores) shows strong scaling with overheads dependent on work-grain size

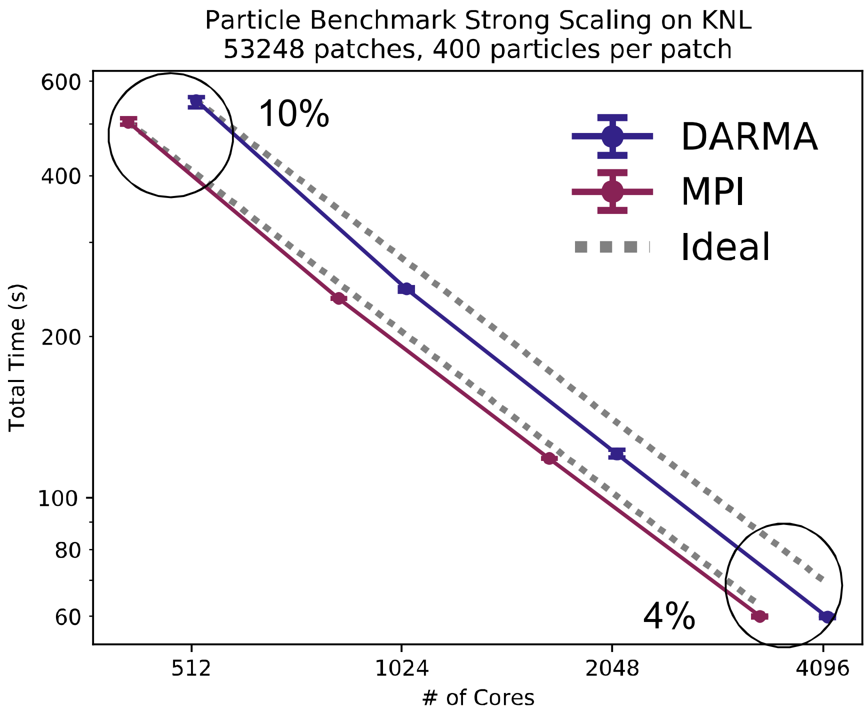


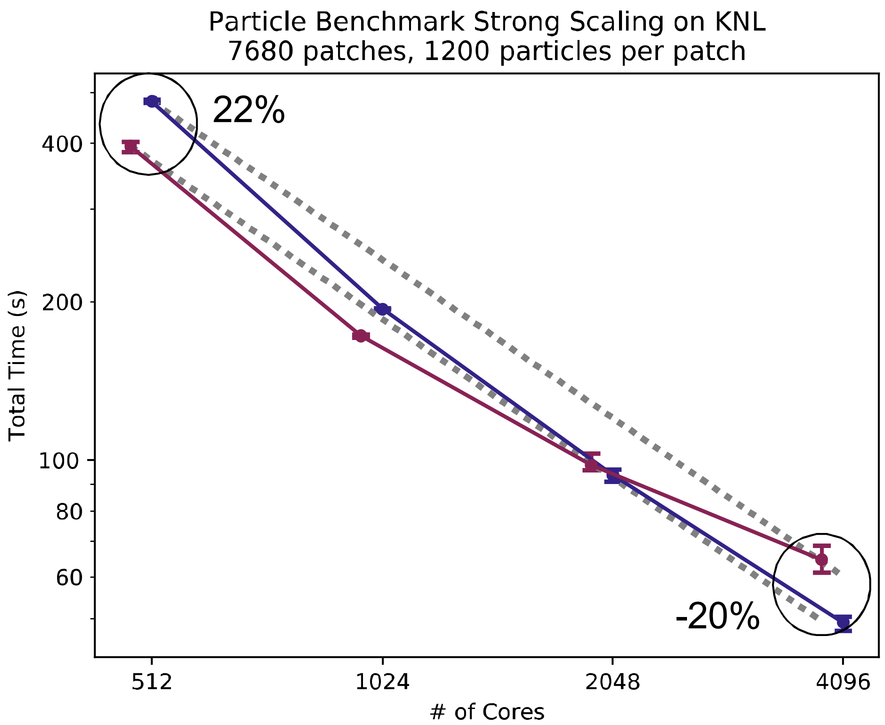


- More patches with fewer particles stresses communication system, particularly for MPI
- MPI shows superlinear scaling with fewer patches per core

- DARMA performs better with larger patch sizes (more particles)
- MPI struggles with extra data movement in larger patches

KNL (up to 64 nodes) shows similar trends with decent strong scaling, overheads depend on work-grain size

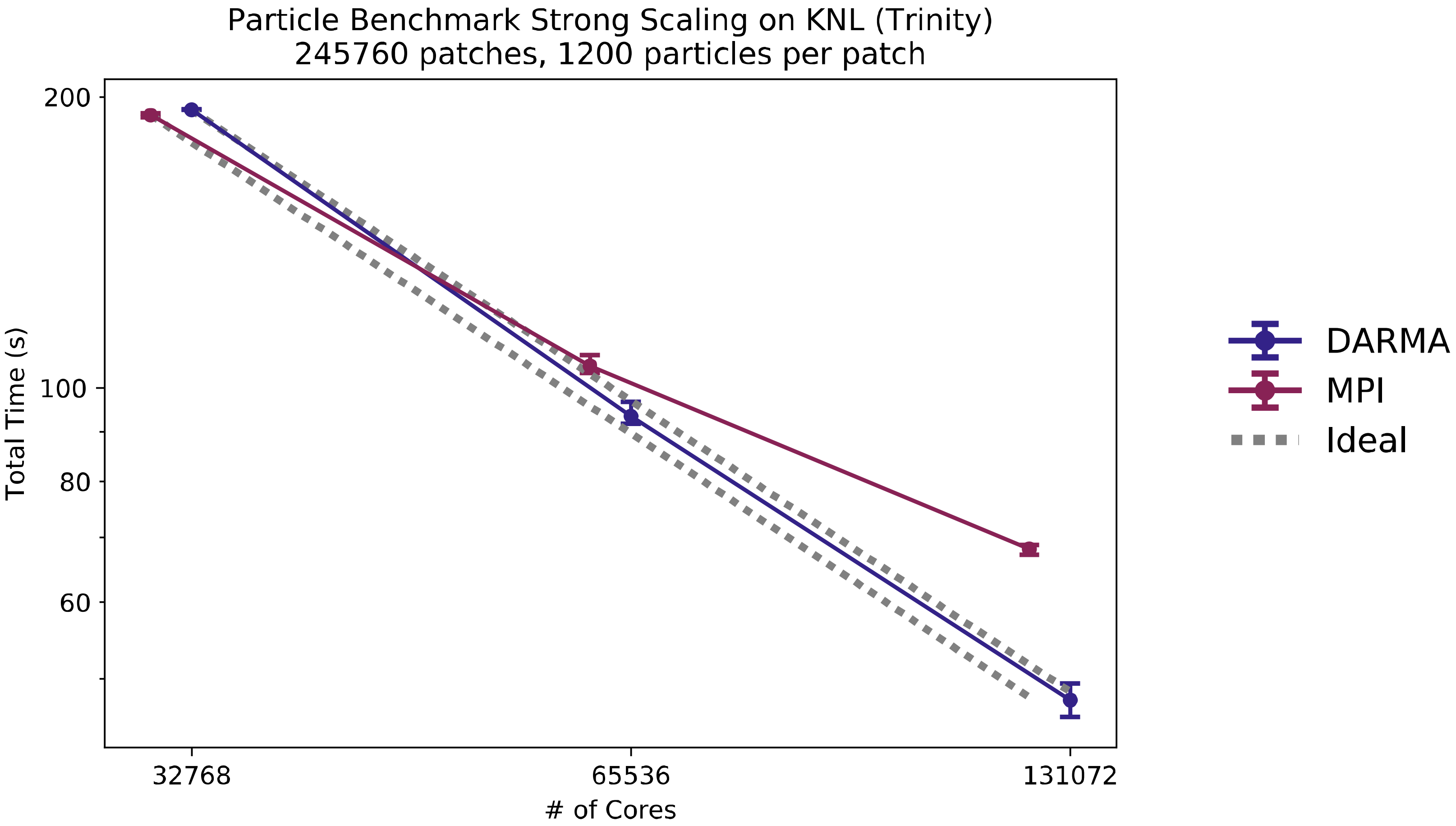




- Superlinear scaling likely due to lower memory footprint, MCDRAM

- MPI struggles with extra data movement in larger patches

KNL (up to 2048 nodes/131K cores) highlights strong scaling for DARMA-Charm++ backend



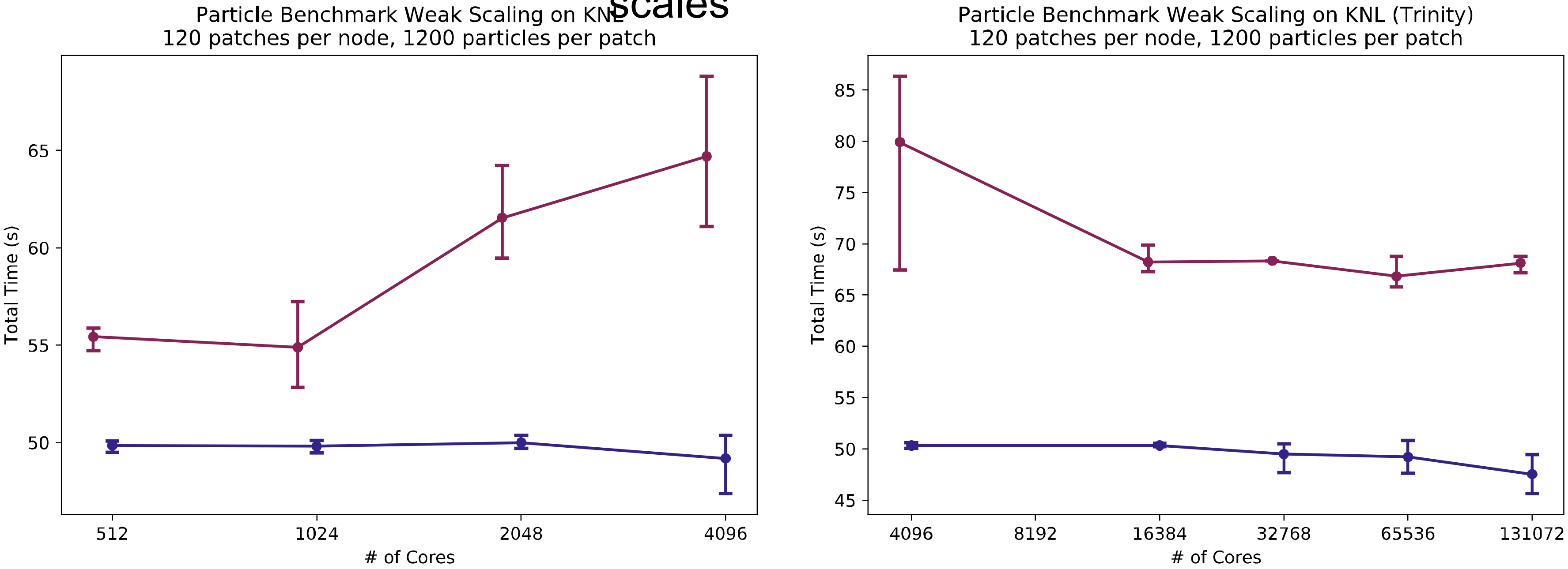
- Consistent with Mutrino KNL, MPI struggles with communication at larger scales
- MPI implementation ``best initial attempt'' at overdecomposition in MPI
- Difficult to identify as DARMA performing well or MPI performing poorly

On KNL, DARMA consistently outperforms MPI

Mutrino

Note differences in scales

Trinity



120 patches/node and 1200 particles/patch

- MPI implementation ``best initial attempt'' at overdecomposition in MPI
- Difficult to identify as DARMA performing well or MPI performing poorly

● DARMA

● MPI

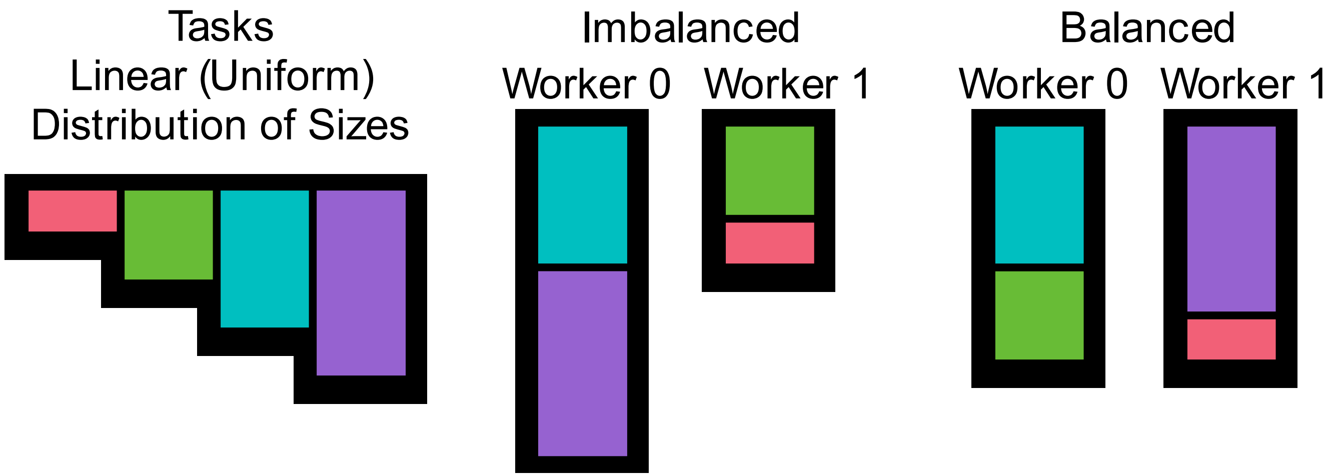
▪▪▪ Ideal

Lessons learned on productivity for molecular dynamics

- Some MPI codes overdecompose (many boxes per rank), but still aggregate all messages (box sends) to a given neighbor
- Message aggregation blocks computation until all particles are sent
- Avoiding message aggregation in MPI and pipelining communication was error prone, tedious tag matching of box send with box receive of same size
- Overdecomposition natural in DARMA with focus on data decomposition in application, runtime handles scheduling
- No need for message aggregation or tedious tag matching schemes

Simulated Load Imbalance Benchmark Overview

- Generates adversarial imbalanced work distribution with known optimal solution

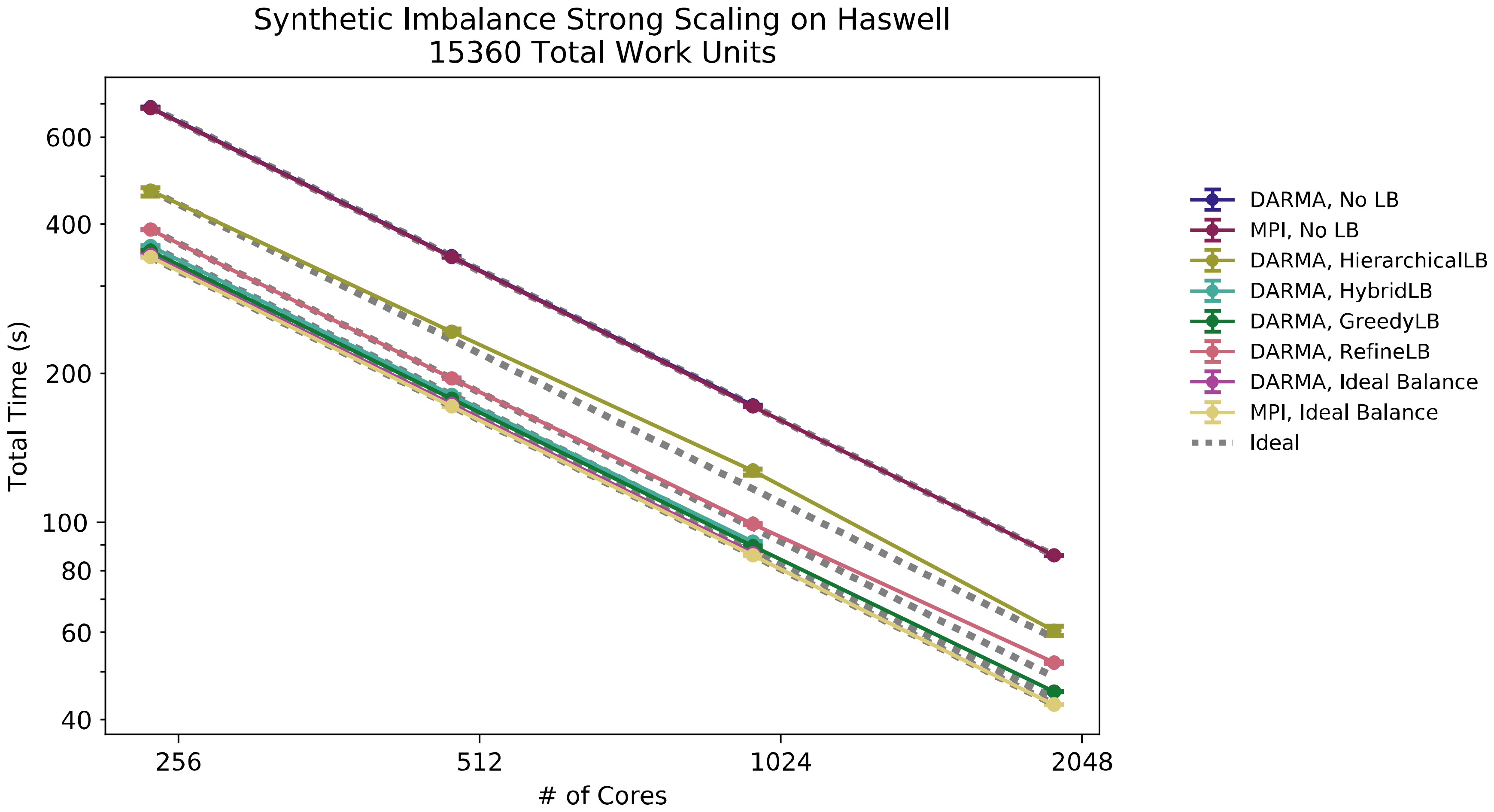


- Benchmark can be run in three modes
  - Perfectly balanced known optimal distribution (best case)
  - Adversarial imbalance with no load balancing (worst case)
  - Adversarial imbalance with load balancing enabled
- Benchmark goal: Assess interplay of load balancing overheads and quality of load balancing

Different load balancers have cost, scaling, and optimality tradeoffs

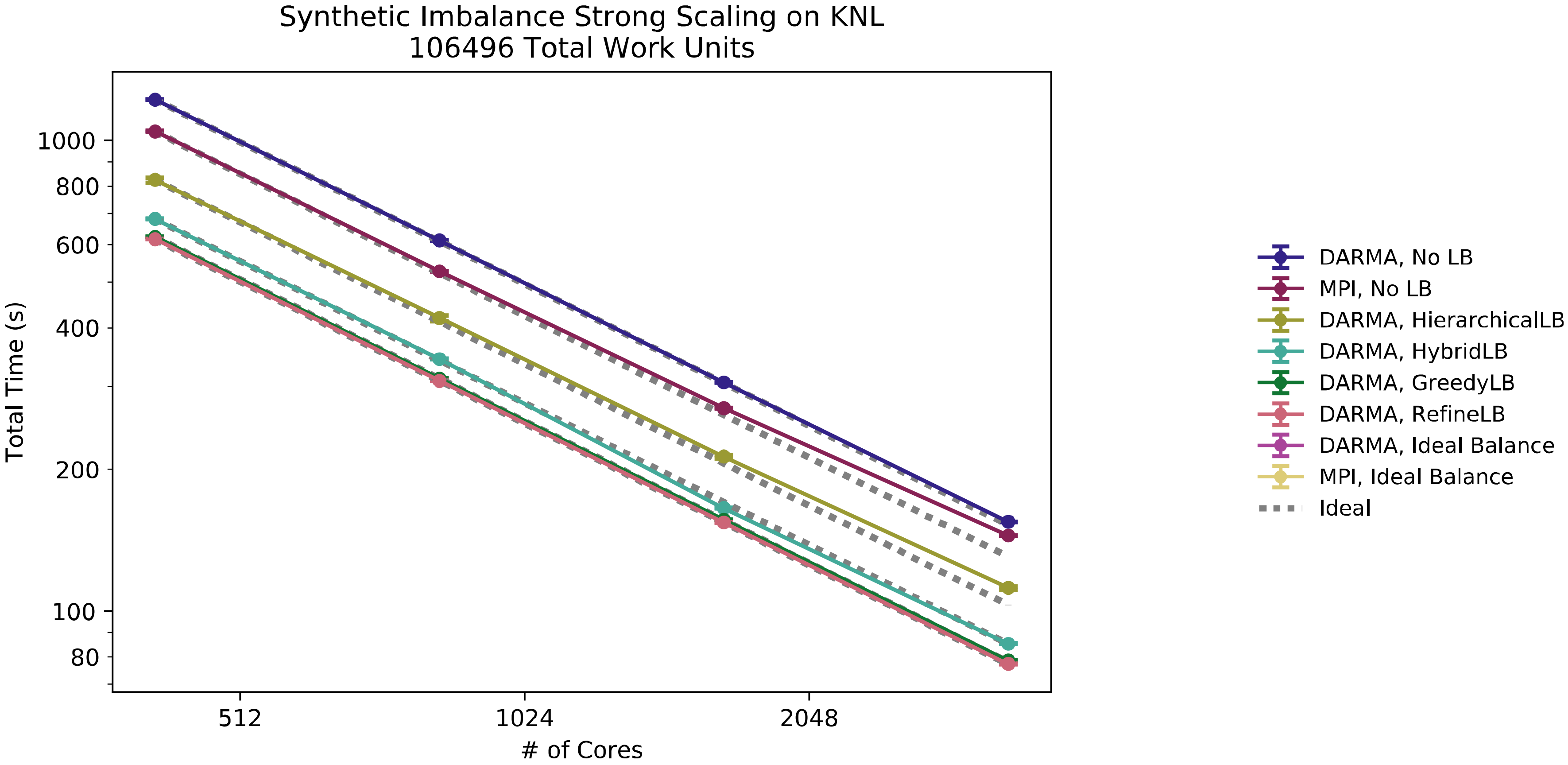
LB Type	LB Name	Description	Benefits	Drawbacks
Centralized	**GreedyLB**	Heap-based, considers all tasks for redistribution	Provides high quality distribution	Not scalable, expensive in memory and space
Centralized	**RefineLB**	Heap-based, considers only tasks above threshold	Fast for centralized load balancer	Not scalable, quality might be low
Distributed, gossip-based	**DistributedLB**	Gossip-based, probabilistic transfer	Extremely fast, fully decentralized	Quality may be low
Distributed, tree-based	**HierarchicalLB**	Tree-based, hierarchical transfer	Fast, typically provides high quality	Greedy algorithm may not be aggressive
Distributed, group-based	**HybridLB**	Creates subgroups of processors and applies centralized	Can reuse centralized LB schemes	May be expensive and slow with large groups

Synthetic imbalance on Haswell (up to 64 nodes/2K cores) shows overheads, scalabilities of each balancer



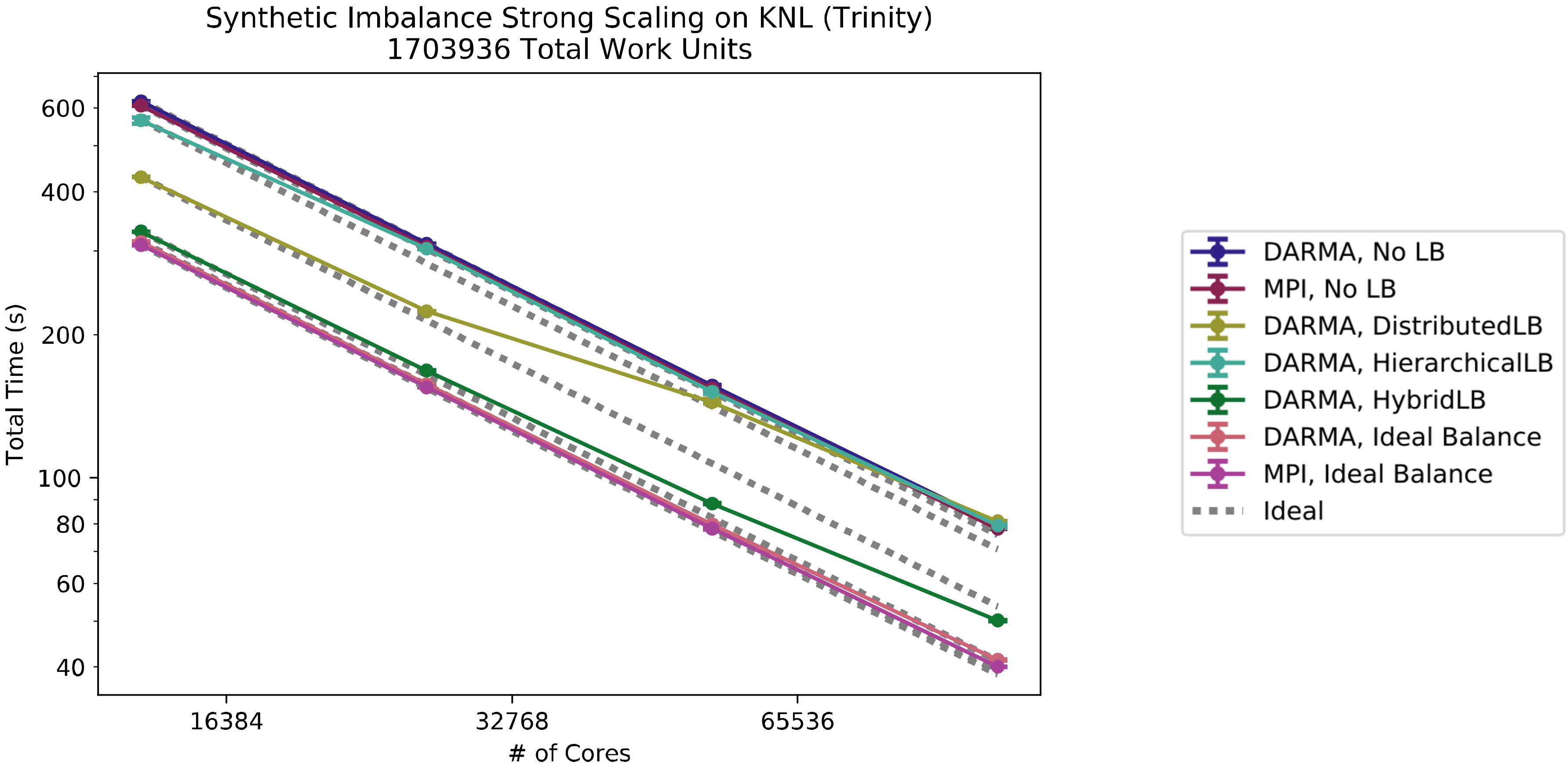
- Only Greedy, Hybrid load balancers competitive with optimal balance baseline
- All load balancers relatively scalable up to 64 nodes, different quality solutions though
- All load balancers better than worst-case baseline with no load balancing

Synthetic imbalance on KNL (up to 64 nodes/2K cores) shows overheads, scalabilities of each balancer



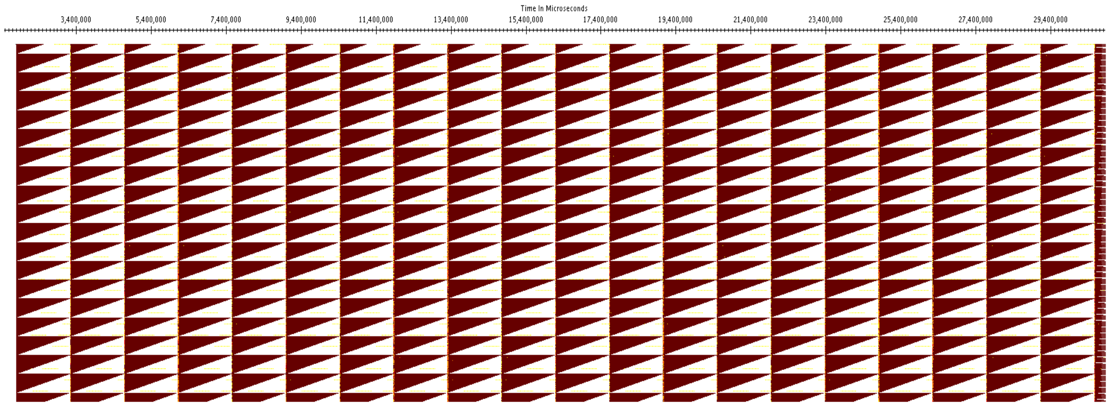
- Only Greedy, Hybrid load balancers competitive with optimal balance baseline
- All load balancers relatively scalable up to 64 nodes, different quality solutions
- All load balancers better than worst-case baseline with no load balancing

Large runs on Trinity (up to 2K nodes) highlight scalability differences between load balancers (KNL)



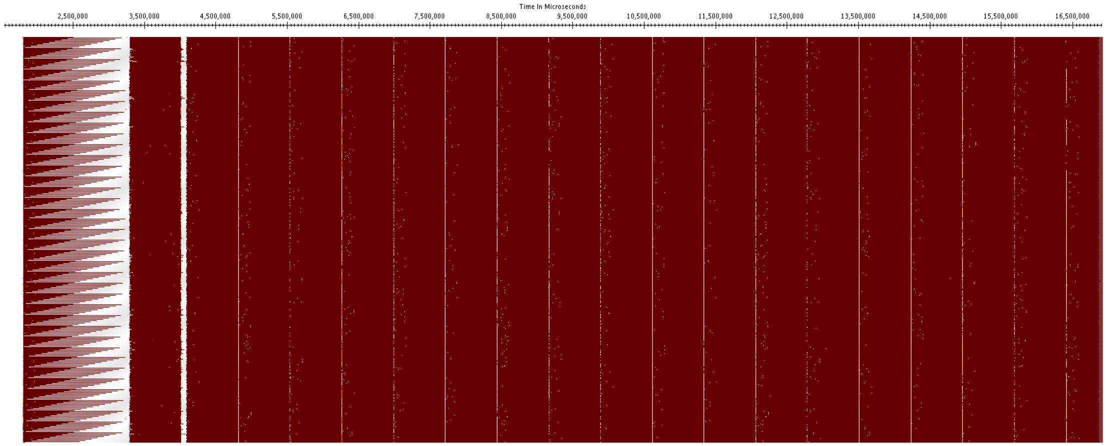
- Refine load balancers skipped, could not finish in 30 minute time cutoff
- All load balancers still relatively scalable, Hierarchical has best scalability but worse quality of load balance
- Only hybrid load balancer gets near optimal balance with low load balance overheads

Load balancers redistribute work, shrink idle time between iterations



No load balancing

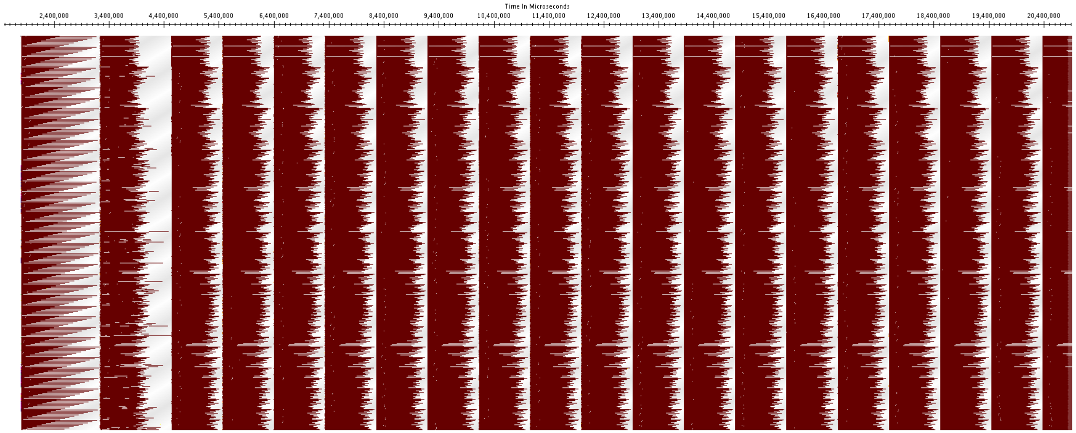
- Red is active computation, white is idle time for each thread
- Execution shows certain threads idling while large tasks finish



Greedy load balancer

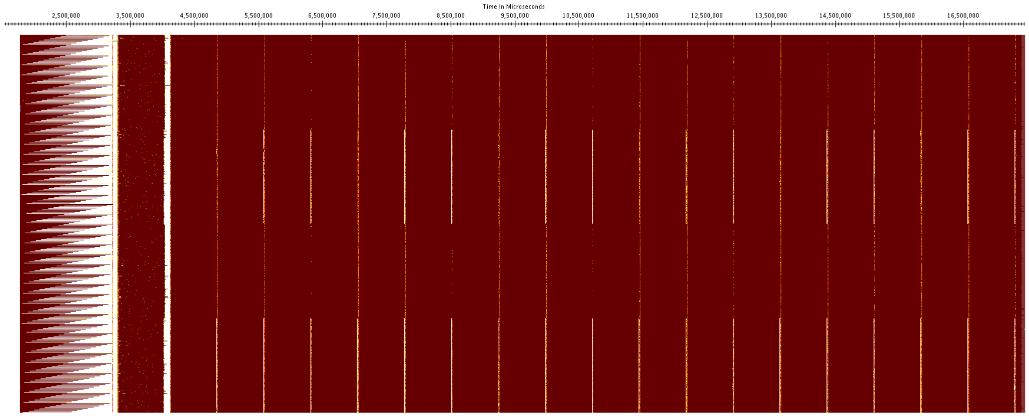
- First iteration imbalanced, but idle time shrinks as load balancer finds nearly optimal solution on second iteration

Some load balancers improve results, but solution is not optimal



Hierarchical load balancer

- Red is active computation, white is idle time for each thread
- Execution shows certain threads idling while large tasks finish



- First iteration imbalanced, but idle time shrinks as load balancer finds nearly optimal solution on second iteration

Hybrid load balancer

Lessons learned on productivity for synthetic imbalance benchmark

- Even for very basic linear imbalance problem, there is no direct mapping to a scalable MPI collective, routine to derive optimal task distribution
- MPI_Gather-Sort-MPI_Scatter could be easily implemented for balancing, but is not scalable
- Ad hoc implementation of app-specific load balancers would be tedious and error-prone
- Load balancing handled transparently in DARMA-Charm++ application, although some tuning may be required to select best load balancer for each application
- Hybrid balancer seems a good universal starting choice

Outline

- Motivation
- Milestone Overview
- AMT + DARMA Overview
- Milestone Results: Bottom Line Up front
- Deep Dive on Findings
  - Generality of the Backend API
  - Interoperability
  - Performance and Productivity
- Conclusions
- Future Work

Conclusions

- **Productivity:**
  - **Easier to express communication overlap:** no Isend/wait pairs, communication progress not explicit in application code
  - **Easier to express tunable granularity:** data decomposition can mismatch execution resources (overdecomposition) without changing application code
  - **Easier to enable load balancing:** migratable data and work chunks can be transparently rebalanced without explicit bookkeeping and rebalancing in application code
- **Performance:**
  - **DARMA is scalable (weak and strong) up to 2K nodes**
  - **Load balancing shows major performance gains** with minimal effort from app developer
  - **Deferred execution and sequential task model have overheads** (~10% over MPI)
  - **Expect DARMA performance to improve as we tune the implementation**
- **Interoperability**: **It's complicated, but the initial results are promising; major focus in Q1 FY18**
- **Generality: declarative backend specification facilitates mapping to different technologies, development of "common components" across backend implementations**

Outline

- Motivation
- Milestone Overview
- AMT + DARMA Overview
- Milestone Results: Bottom Line Up front
- Deep Dive on Findings
  - Generality of the Backend API
  - Interoperability
  - Performance and Productivity
- Conclusions
- Future Work

Future Work

- Focus of DARMA team next year
  - Interoperability
  - Hardening/Tuning
  - Productivity tools (timers, performance profilers, debugging aides)
  - Devops, documentation, and testing
  - Focus on Empire and SPARC requirements
  - Continued engagement with UQ, Multiscale teams
- Bigger picture/longer term efforts
  - ATS-2
  - Best practices and standards-based runtime solutions