SAND2017-10960PE

DARMA: A C++ Portability Layer for Asynchronous Many-Task Runtimes

Dr. Robert L. Clay

Oct 2, 2017

RIKEN K-Computer Center

Kobe, Japan

The DARMA Development Team

- Janine C. Bennett (PI)
- Robert L. Clay (PM)
- David Hollman
- Hemanth Kolla
- Jonathan Lifflander
- Aram H. Markosyan
- Francesco Rizzi
- Nicole Slattengren
- Jeremiah J. Wilke

MOTIVATION

Extreme-scale HPC system architectures introduce a number of complexities

- Performance Heterogeneity
  - Accelerators
  - Thermal throttling
  - General system noise
  - Responses to transient failures
- Energy Constraints
- Decreased system reliability
- Deep memory hierarchies

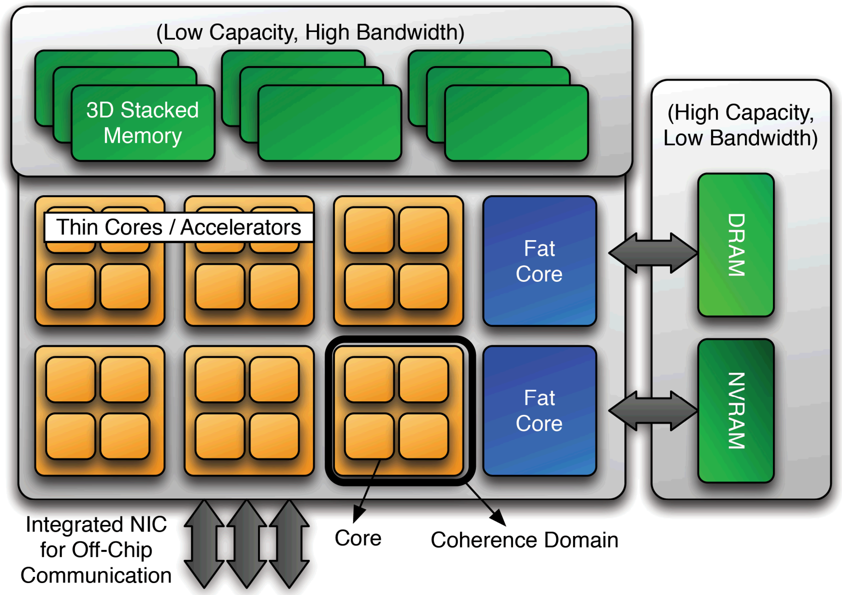

(Low Capacity, High Bandwidth)
3D Stacked Memory
(High Capacity, Low Bandwidth)
Thin Cores / Accelerators
Fat Core
Fat Core
DRAM
NVRAM
Integrated NIC for Off-Chip Communication
Core
Coherence Domain

Image courtesy of www.cal-design.org

Current imperative programming models and runtime systems require mitigation of challenges largely at application-developer level

What is the alternative?

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
```

Imperative vs declarative programming in a nutshell

What is the alternative?

Imperative

```
Get a piece of bread
If likes mustard
```

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

```
Put more bread on top
Cut in half
```

Declarative

```
Make me a sandwich
```

Programmer expresses logic without prescribing control-flow

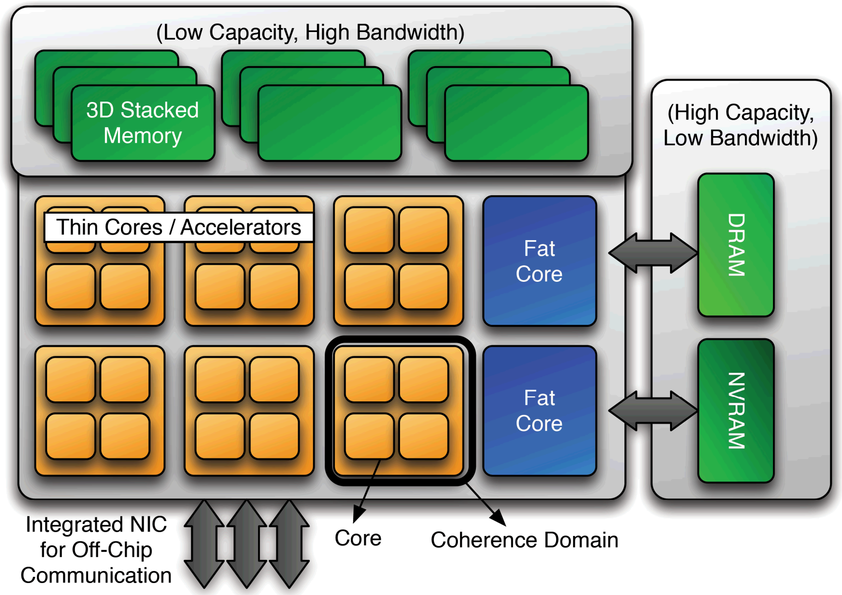
Imperative vs declarative programming in a nutshell

A declarative style of programming enables mitigation of challenges at the *runtime-level*

- Application developer specification of desired result

- Not a “magic bullet”: complexity must still be managed

- Separation of concerns: complexity management at runtime-level



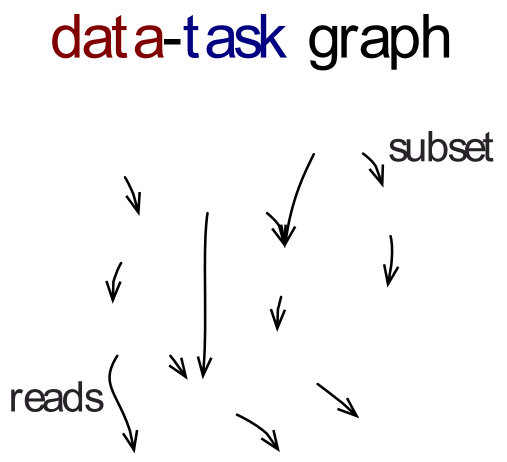
(Low Capacity, High Bandwidth)
3D Stacked Memory
(High Capacity, Low Bandwidth)
Thin Cores / Accelerators
Fat Core
Fat Core
DRAM
NVRAM
Integrated NIC for Off-Chip Communication
Core
Coherence Domain

Image courtesy of www.cal-design.org

COMPUTER ARCHITECTURE LABORATORY
EXASCALE DESIGN SPACE EXPLORATION

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
  - Dynamic load balancing
  - When and where to execute work and move data



Common misconception: AMT seeks to replace MPI

- MPI is a transport layer
- AMT runtimes can and do use MPI as their transport layer

Rather: AMT research focus is on developing abstractions to

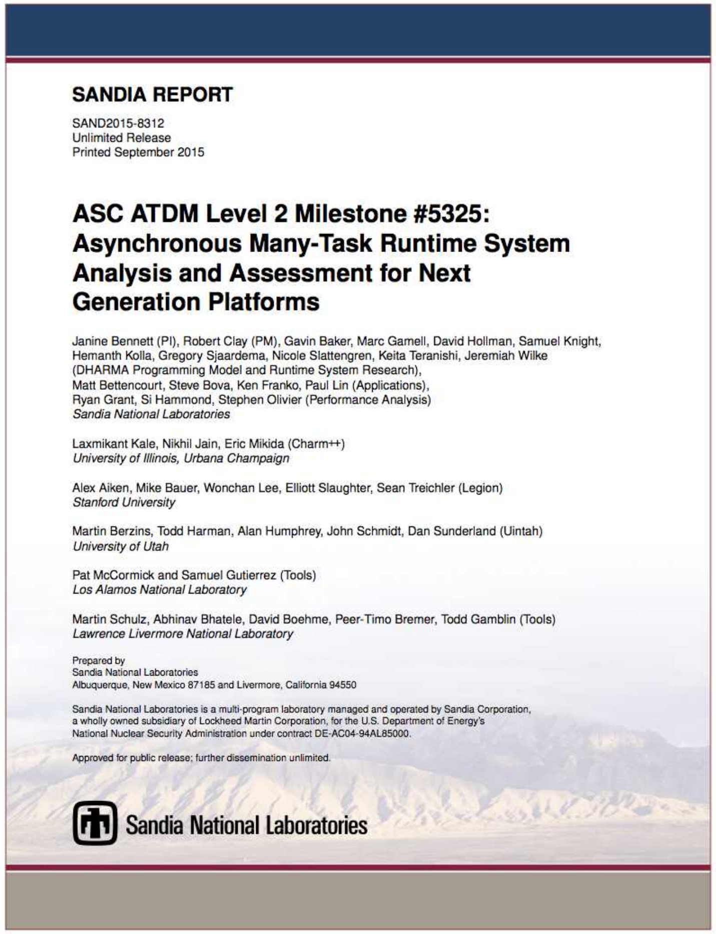
- Facilitate expression and management of asynchrony
- Express and manage task parallelism (in addition to data-parallelism)
- Capture semantic information that enables runtime-management of data movement and control-flow execution in complex memory and execution spaces
- Active research area
  - Habanero-UPC++, HPX, Legion, OCR, PaRSEC, SCIOTO, STAPL, Uintah, Charm++, StarPU, …

2015 study to assess leading AMT runtimes led to DARMA

Aim: inform Sandia's technical roadmap for next generation codes

- Broad survey of many AMT runtime systems
- Deep dive on Charm++, Legion, Uintah

- ***Programmability:*** Does this runtime enable efficient expression of ATDM workloads?
- ***Performance:*** How performant is this runtime for our workloads on current platforms and how well suited is this runtime to address future architecture challenges?
- ***Mutability:*** What is the ease of adopting this runtime and modifying it to suit our code needs?


SANDIA REPORT

SAND2015-8312
Unlimited Release
Printed September 2015

ASC ATDM Level 2 Milestone #5325: Asynchronous Many-Task Runtime System Analysis and Assessment for Next Generation Platforms

Janine Bennett (PI), Robert Clay (PM), Gavin Baker, Marc Gamell, David Hollman, Samuel Knight, Hemanth Kolla, Gregory Sjaardema, Nicole Slattengren, Keita Teranishi, Jeremiah Wilke (DHARMA Programming Model and Runtime System Research),
Matt Bettencourt, Steve Bova, Ken Franko, Paul Lin (Applications),
Ryan Grant, Si Hammond, Stephen Olivier (Performance Analysis)
Sandia National Laboratories

Laxmikant Kale, Nikhil Jain, Eric Mikida (Charm++)
University of Illinois, Urbana Champaign

Alex Aiken, Mike Bauer, Wonchan Lee, Elliott Slaughter, Sean Treichler (Legion)
Stanford University

Martin Berzins, Todd Harman, Alan Humphrey, John Schmidt, Dan Sunderland (Uintah)
University of Utah

Pat McCormick and Samuel Gutierrez (Tools)
Los Alamos National Laboratory

Martin Schulz, Abhinav Bhatele, David Boehme, Peer-Timo Bremer, Todd Gamblin (Tools)
Lawrence Livermore National Laboratory

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

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.

Approved for public release; further dissemination unlimited.

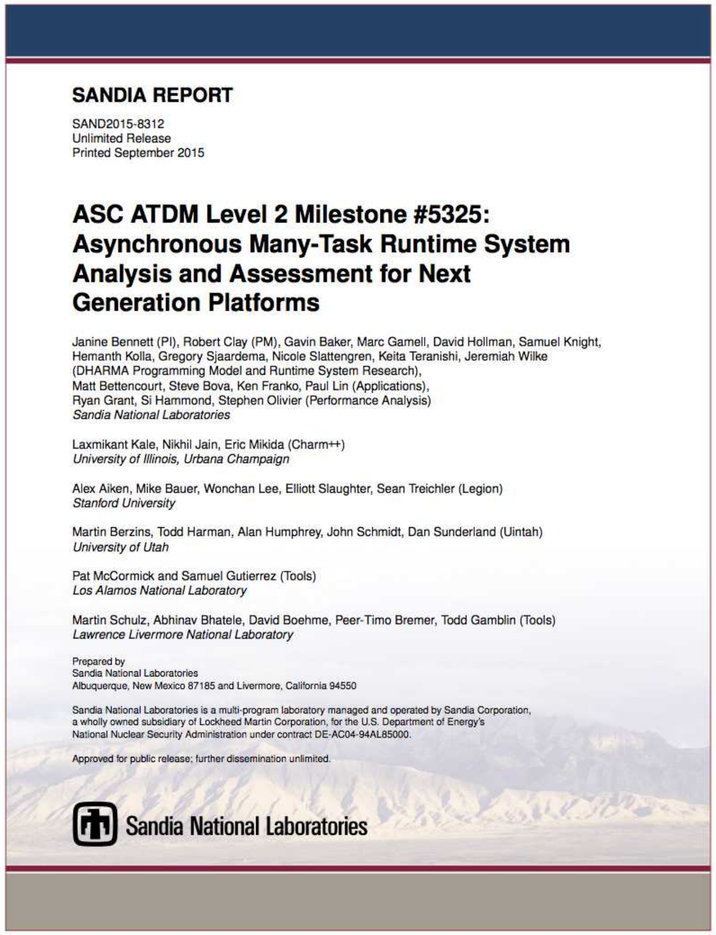
Sandia National Laboratories

2015 study to assess leading AMT runtimes led to DARMA

Aim: inform Sandia's technical roadmap for next generation codes

- ***Conclusions***
  - AMT systems show great promise
  - Gaps in requirements for Sandia applications
  - No common user-level APIs
  - Need for best practices and standards

- ***Survey recommendations led to DARMA***
  - C++ abstraction layer for AMT runtimes
  - Requirements driven by Sandia applications
  - A single user-level API
  - Support multiple AMT runtimes to begin identification of best practices



SANDIA REPORT

SAND2015-8312
Unlimited Release
Printed September 2015

ASC ATDM Level 2 Milestone #5325: Asynchronous Many-Task Runtime System Analysis and Assessment for Next Generation Platforms

Janine Bennett (PI), Robert Clay (PM), Gavin Baker, Marc Gamell, David Hollman, Samuel Knight, Hemanth Kolla, Gregory Sjaardema, Nicole Slattengren, Keita Teranishi, Jeremiah Wilke (DHARMA Programming Model and Runtime System Research),
Matt Bettencourt, Steve Bova, Ken Franko, Paul Lin (Applications),
Ryan Grant, Si Hammond, Stephen Olivier (Performance Analysis)
Sandia National Laboratories

Laxmikant Kale, Nikhil Jain, Eric Mikida (Charm++)
University of Illinois, Urbana Champaign

Alex Aiken, Mike Bauer, Wonchan Lee, Elliott Slaughter, Sean Treichler (Legion)
Stanford University

Martin Berzins, Todd Harman, Alan Humphrey, John Schmidt, Dan Sunderland (Uintah)
University of Utah

Pat McCormick and Samuel Gutierrez (Tools)
Los Alamos National Laboratory

Martin Schulz, Abhinav Bhatele, David Boehme, Peer-Timo Bremer, Todd Gamblin (Tools)
Lawrence Livermore National Laboratory

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

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.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

WHAT IS DARMA?

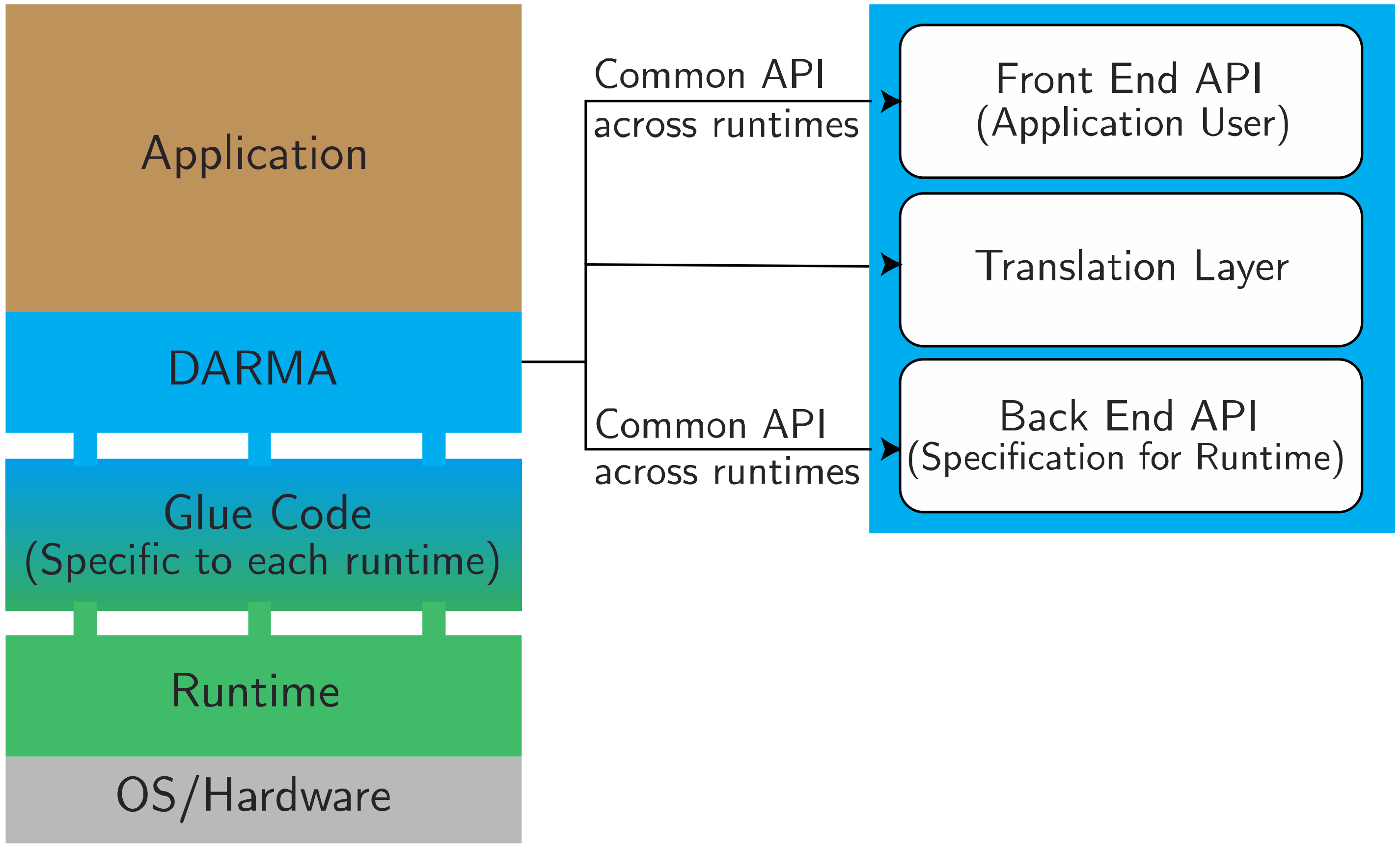
What is DARMA?

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.

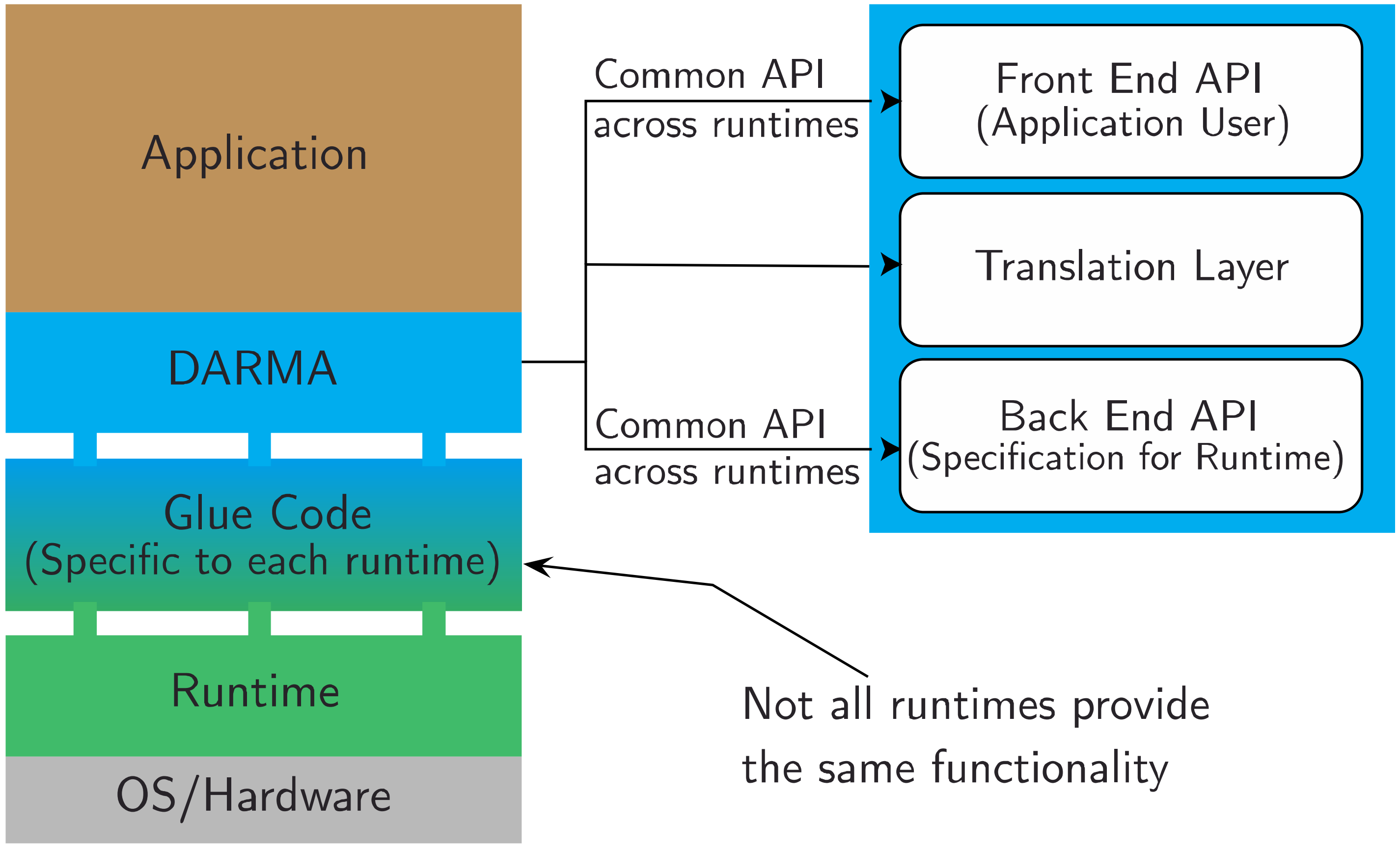
Sandia is using DARMA to inform its technical roadmap for next generation codes.

DARMA provides a unified API to application developers to specify tasks

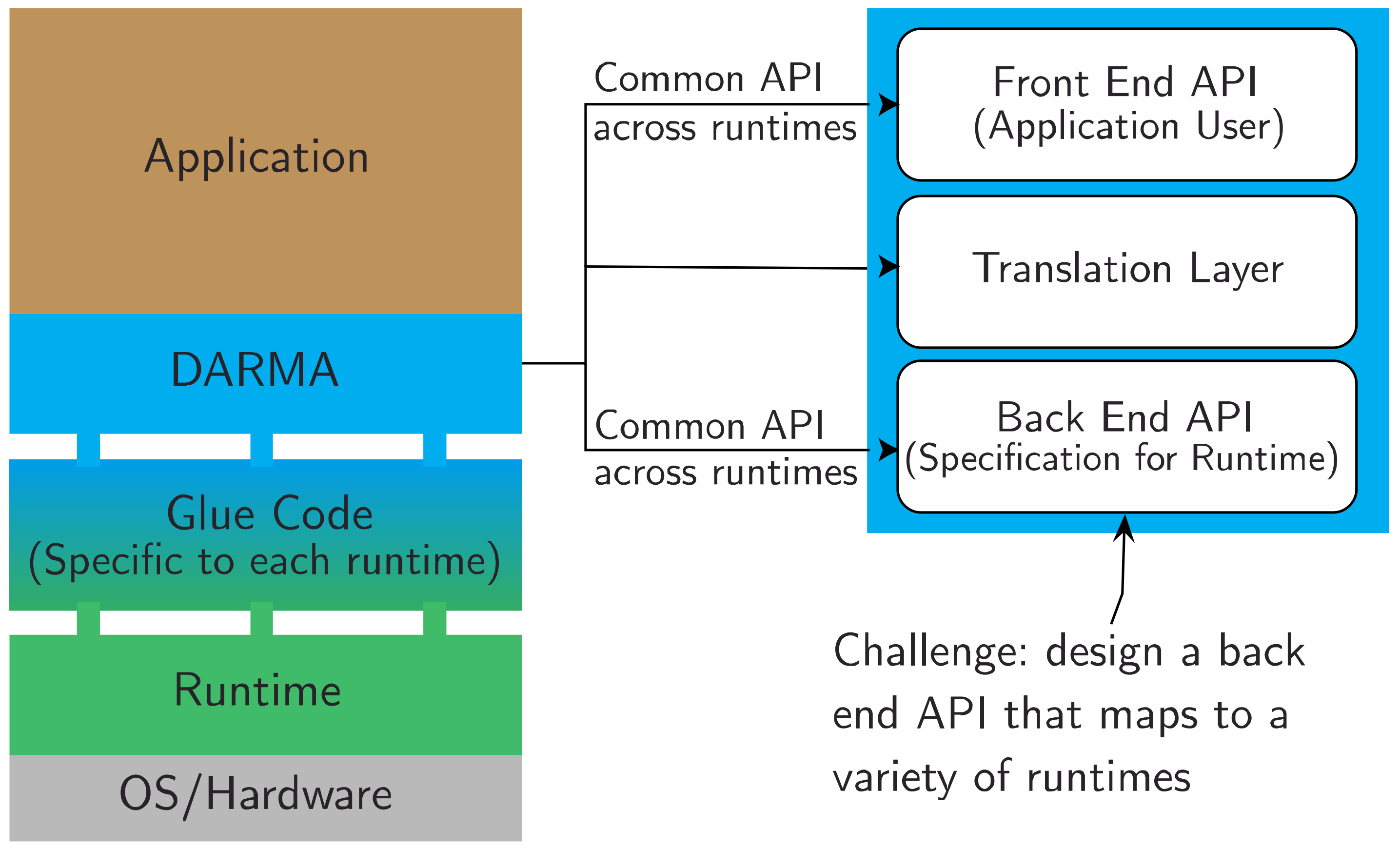


Application
DARMA
Glue Code
(Specific to each runtime)
Runtime
OS/Hardware
Common API across runtimes
Common API across runtimes
Front End API
(Application User)
Translation Layer
Back End API
(Specification for Runtime)

Application code is translated into a series of backend API calls to an AMT runtime



Application code is translated into a series of backend API calls to an AMT runtime

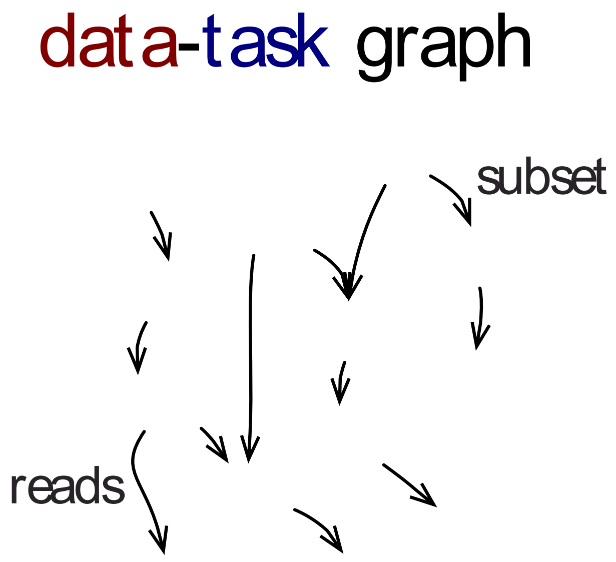


Application
DARMA
Glue Code
(Specific to each runtime)
Runtime
OS/Hardware
Common API across runtimes
Common API across runtimes
Front End API
(Application User)
Translation Layer
Back End API
(Specification for Runtime)

Challenge: design a back end API that maps to a variety of runtimes

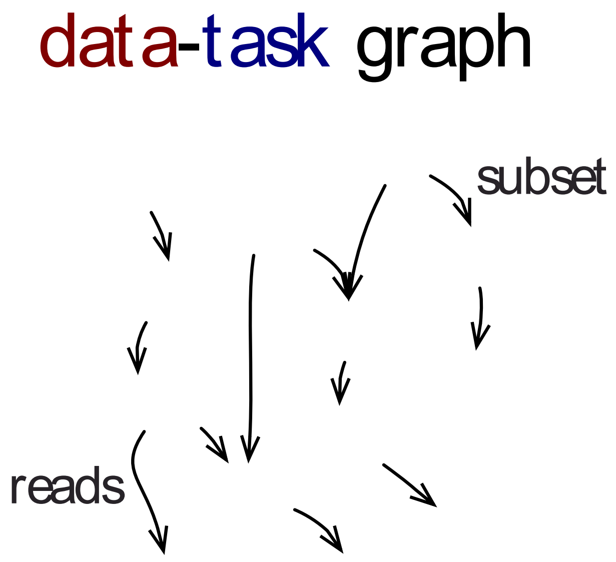
Considerations when developing a backend API that maps to a variety of runtimes

- AMT runtimes often operate with a directed acyclic graph (DAG)
  - Captures relationships between application data and inter-dependent tasks
- DAGs can be annotated to capture additional information
  - Tasks' read/write usage of data
  - Task needs a subset of data

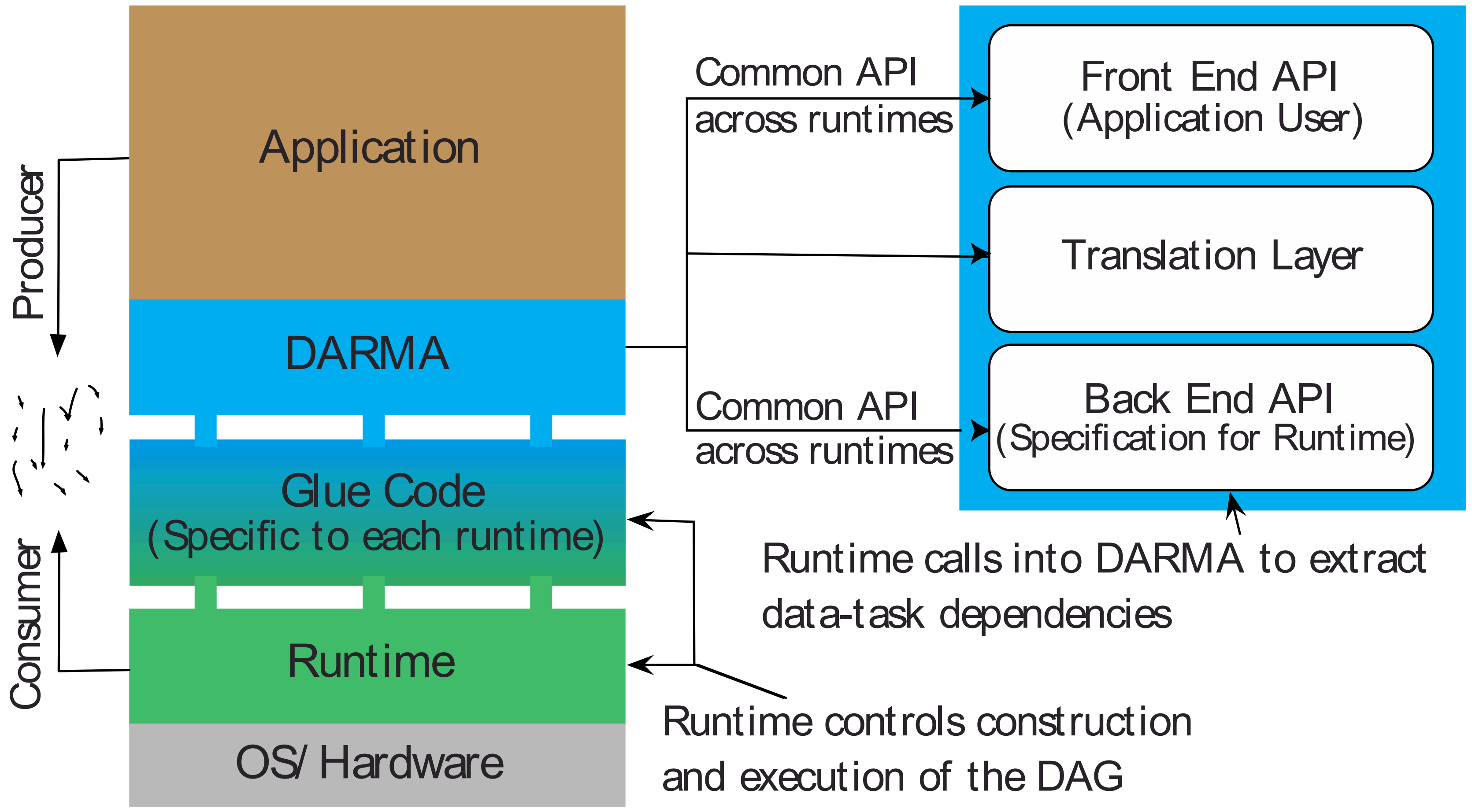


Considerations when developing a backend API that maps to a variety of runtimes

- AMT runtimes often operate with a directed acyclic graph (DAG)
  - Captures relationships between application data and inter-dependent tasks
- DAGs can be annotated to capture additional information
  - Tasks' read/write usage of data
  - Task needs a subset of data
- Additional information enables runtime to reason more completely about
  - When and where to execute a task
  - Whether to load balance
- Existing runtimes leverage DAGs with varying degrees of annotation



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


Producer
Application
DARMA
Glue Code
(Specific to each runtime)
Runtime
OS/ Hardware
Consumer
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

DARMA’s Backend Runtime System Responsibilities

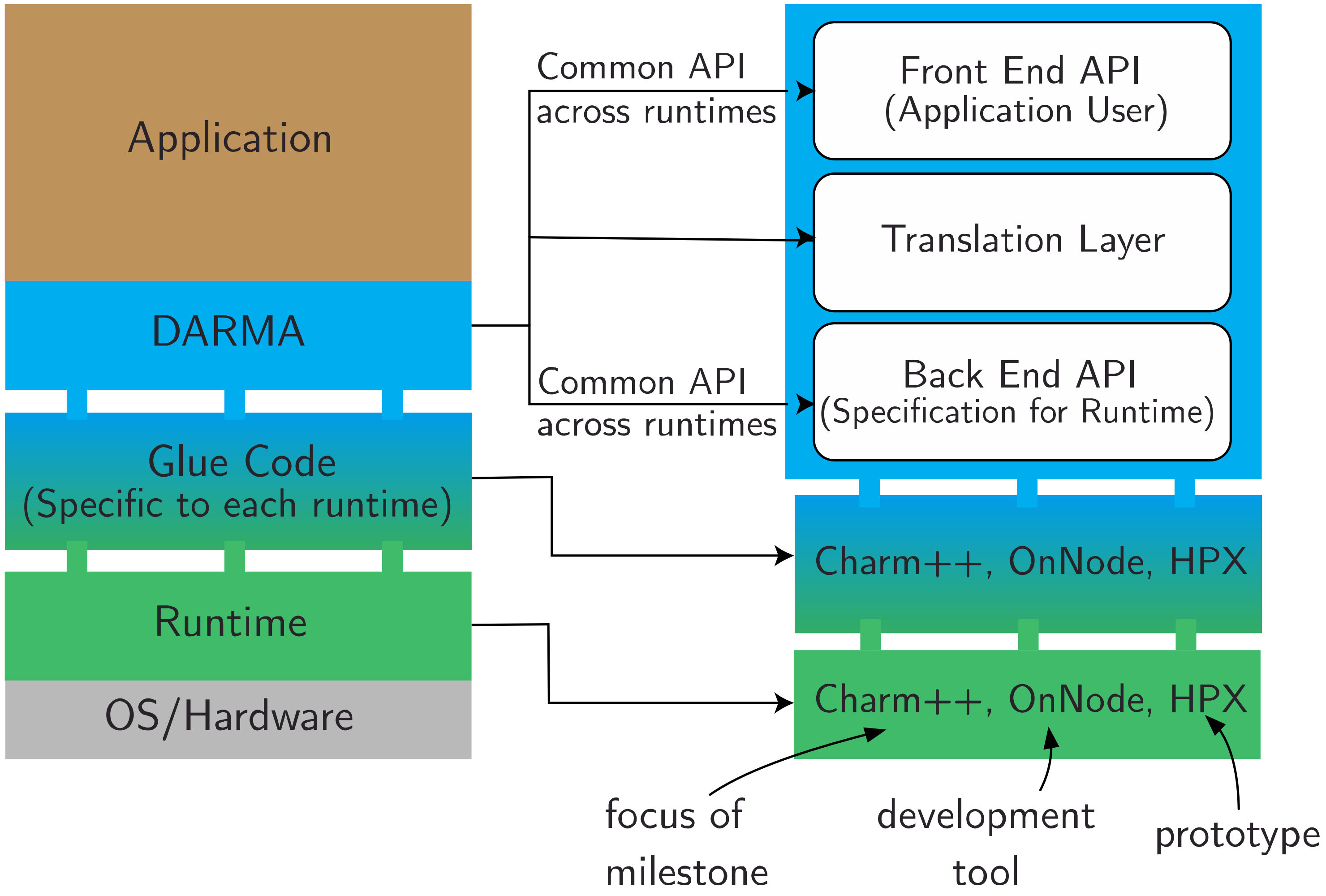
- **Manage data dependencies between tasks (data inputs and outputs)**
  - Exploit data usage (write/read/etc.) and sequencing information from the frontend to schedule tasks without data conflicts
  - Make scheduling decisions based on current state to copy, move, or stall data accesses to optimize 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)**

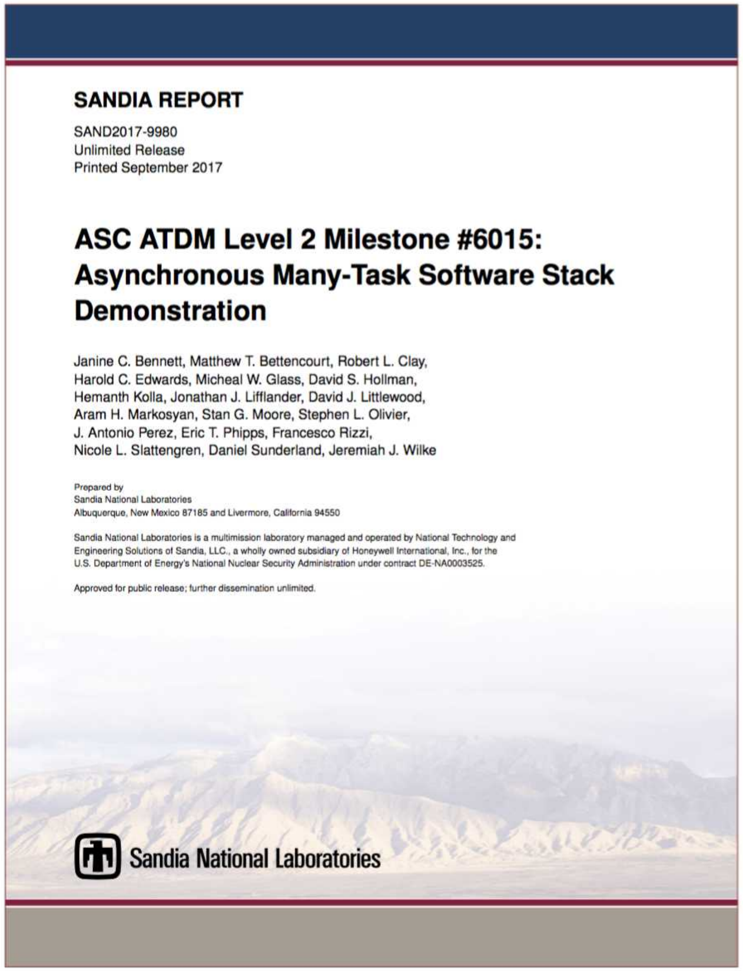
A runtime’s level of native support for these capabilities is a contributing factor to the thickness of the “glue code”

Currently there are three back ends in various stages of development


Application
DARMA
Glue Code (Specific to each runtime)
Runtime
OS/Hardware
Common API across runtimes
Common API across runtimes
Front End API (Application User)
Translation Layer
Back End API (Specification for Runtime)
Charm++, OnNode, HPX
Charm++, OnNode, HPX
focus of milestone
development tool
prototype

Strategy and implementation details for backend mappings are included in a recent tech 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

SAND2017-9980
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.

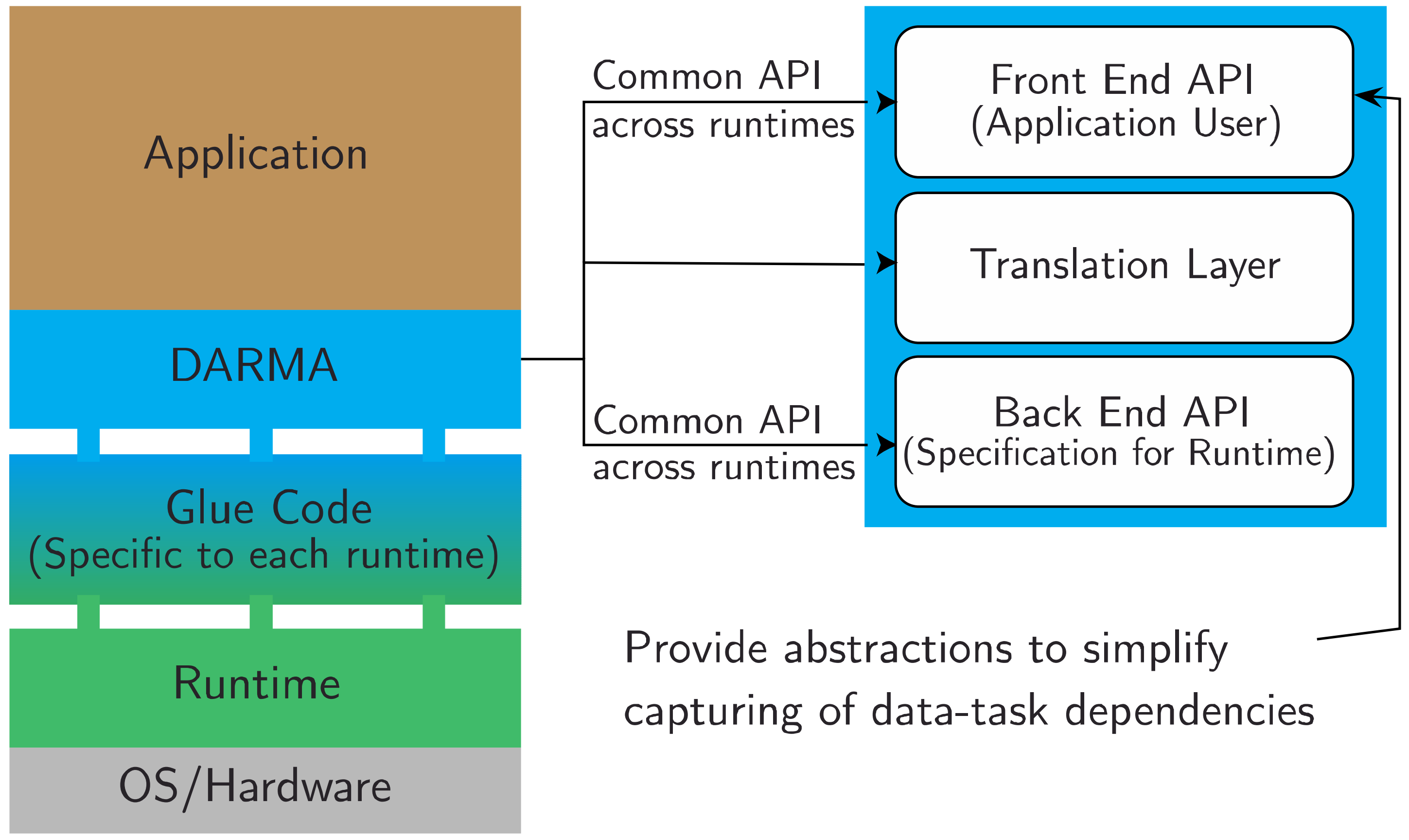
Approved for public release; further dissemination unlimited.

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

DARMA front end abstractions for data and tasks are co-designed with Sandia application scientists


Application
DARMA
Glue Code (Specific to each runtime)
Runtime
OS/Hardware
Common API across runtimes
Common API across runtimes
Front End API (Application User)
Translation Layer
Back End API (Specification for Runtime)
Provide abstractions to simplify capturing of data-task dependencies

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`

DARMA's abstractions provide the application developer with productivity and performance benefits

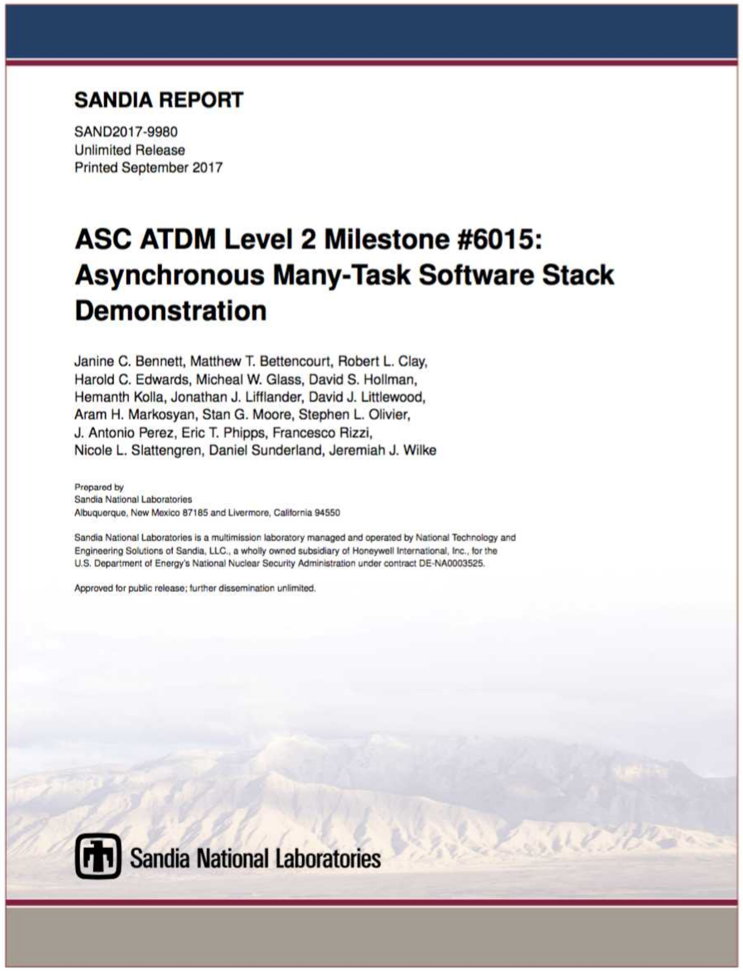
- Automatically capture dependencies and data effects through C++ metaprogramming
  - Visible code is just variables and functions, no tasks
  - Creating DAG directly in user code is tedious and error-prone
- Each data block/variable tracked by logical identifier in runtime
  - Enables automatic migration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

DARMA's abstractions provide the application developer with productivity and performance benefits

- 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

A recent report captures a detailed assessment of the overall DARMA approach

- Uses proxy applications and benchmarks representative of Sandia applications
- Performance assessment on Trinity Supercomputer
- Feedback from application developers
- Assessment of
  - interoperability challenges
  - generality of backend API



SANDIA REPORT

SAND2017-9980
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

APPLICATION DRIVERS

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

EMPIRE: ElectroMagnetic Plasma In Radiation Environments

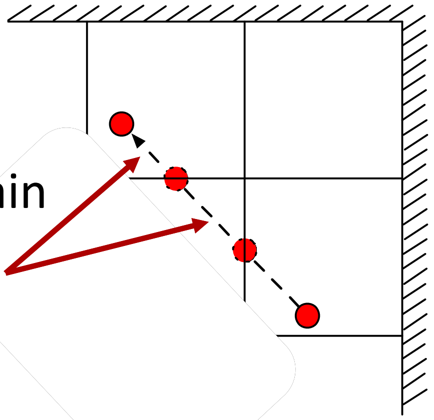
- SNL is developing a new code base for plasma simulations
- Component based approach using the Trilinos framework
- The PIC component of Empire is the basis for our proxy app work
- Two sets on unknowns, mesh data and particles
  - Domain decomposition on the fields and the particles can be out of balance
  - 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

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



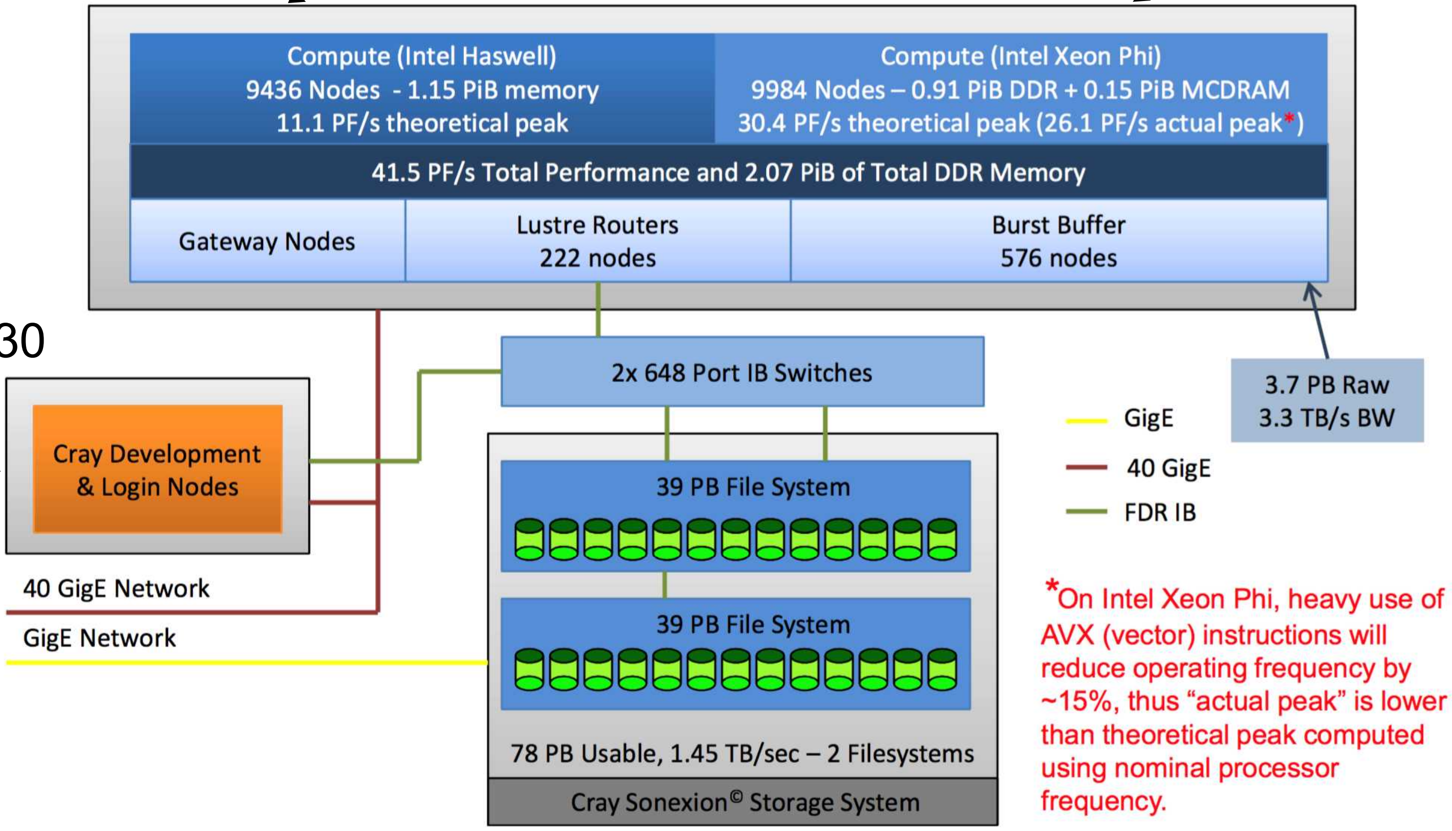
MAPPING TO TRINITY

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

Haswell: enables support for current applications

KNL: enables emerging architecture, workflow, runtime system research

Cray XC30



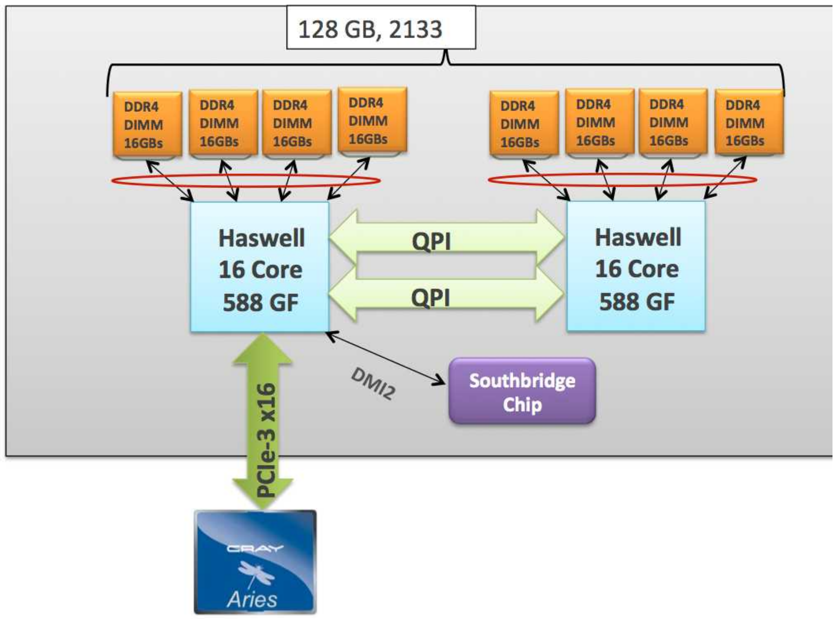
Compute (Intel Haswell)
9436 Nodes - 1.15 PiB memory
11.1 PF/s theoretical peak
Compute (Intel Xeon Phi)
9984 Nodes – 0.91 PiB DDR + 0.15 PiB MCDRAM
30.4 PF/s theoretical peak (26.1 PF/s actual peak*)
41.5 PF/s Total Performance and 2.07 PiB of Total DDR Memory
Gateway Nodes
Lustre Routers 222 nodes
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.

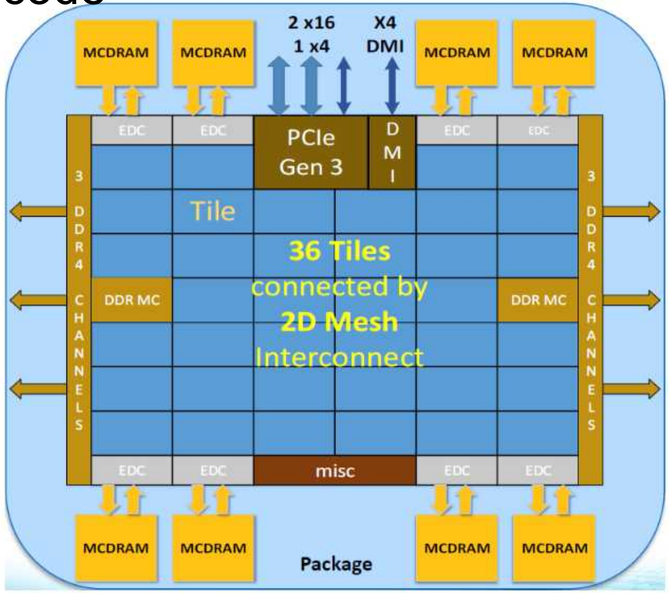
(Image courtesy of ACES)

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)

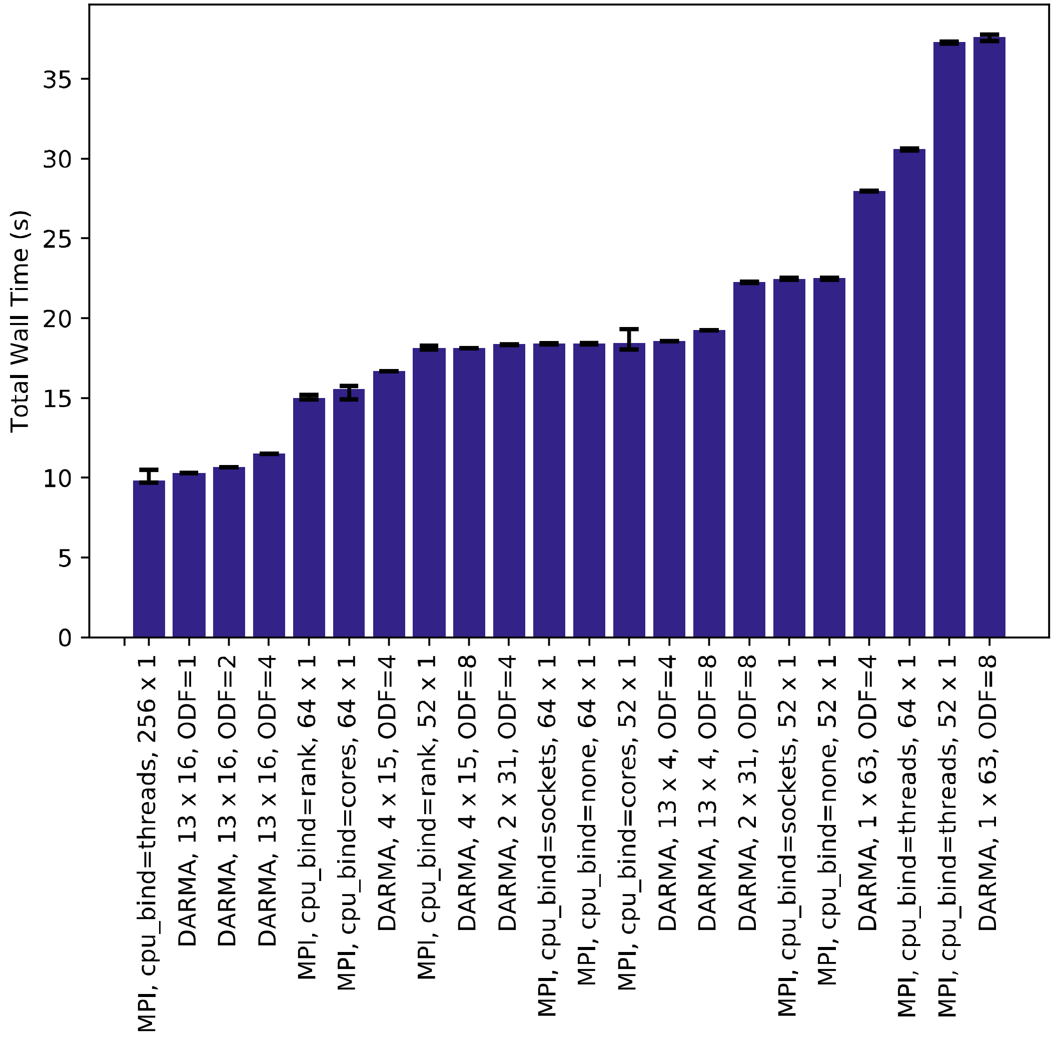
Balanced and Unbalanced SimplePIC Studies

- 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

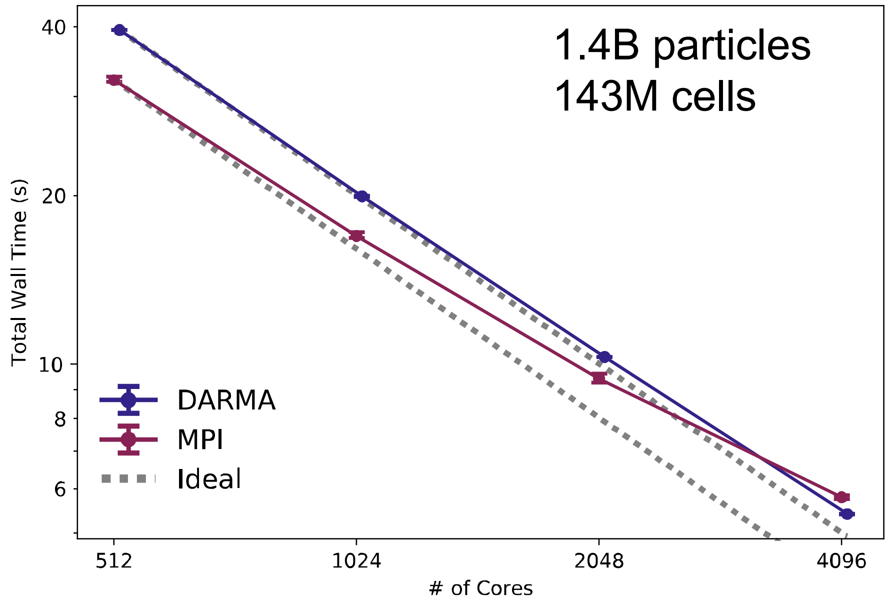
DARMA: 13 processes per node, each with

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

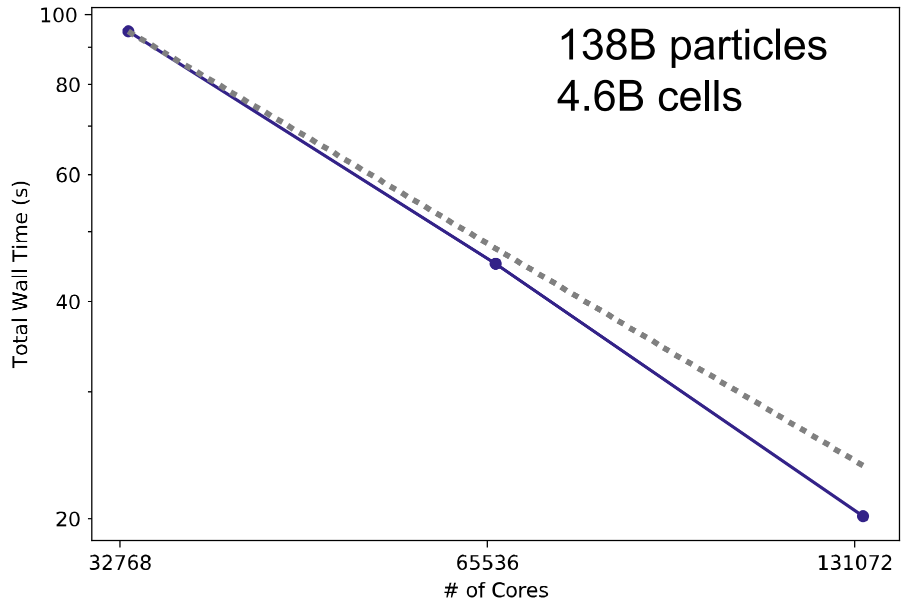
DYNAMIC LOAD BALANCING

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

Mutrino (KNL, 4K cores)



Trinity (KNL, 131K cores)

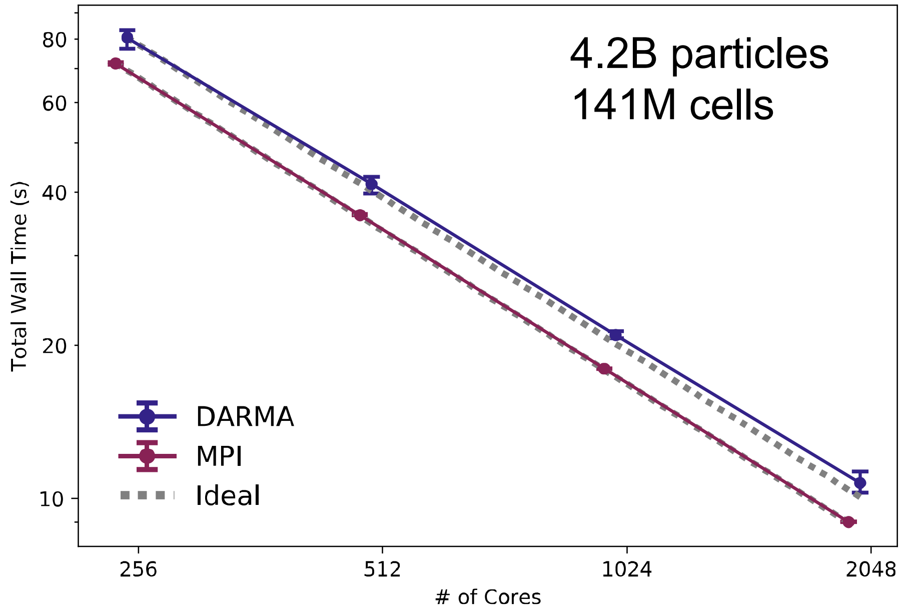


- DARMA overhead with respect to MPI is -5-24%.
- On 2K cores, grain size is too small and, hence, degraded scaling.
- MPI scaling degradation is likely due to MPI only launch on KNL.

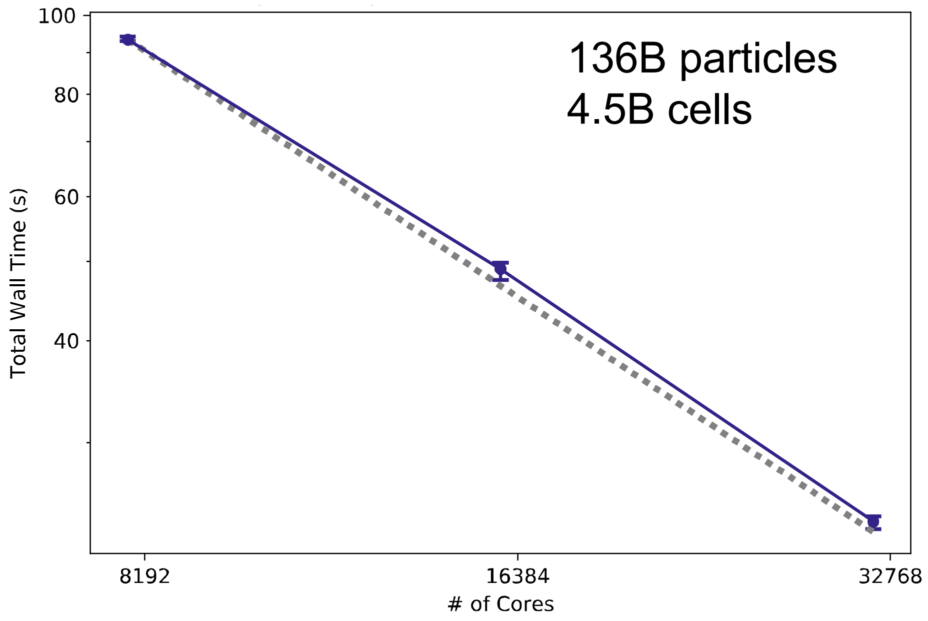
- DARMA scales super-linearly up to 131K cores.

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

Mutrino (Haswell, 2K cores)



Trinity (Haswell, 32K cores)

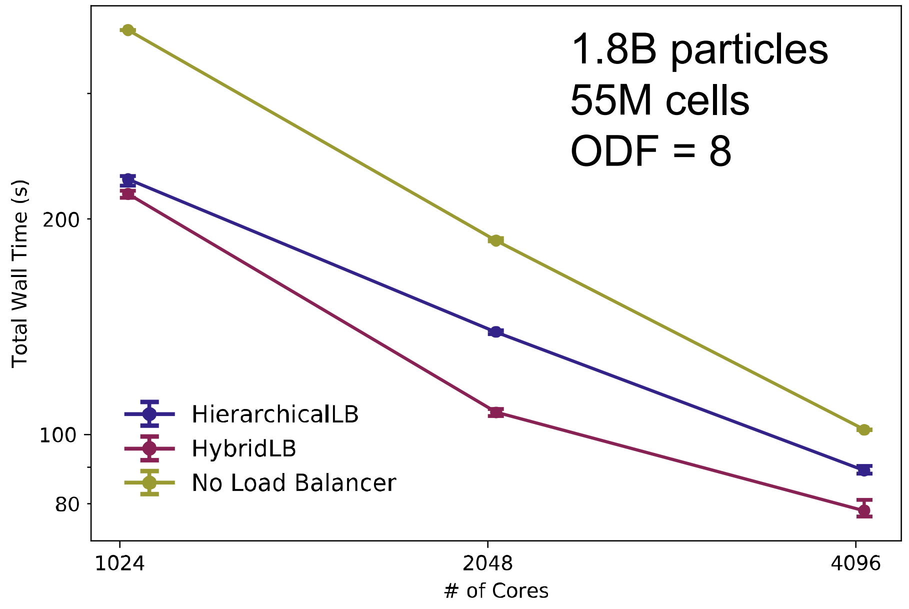


- DARMA overhead with respect to MPI is 12-19%.
- On 2K cores, grain size is too small and, hence, DARMA does not have perfect linear scaling.
- MPI scales ideally on up to 2K cores.

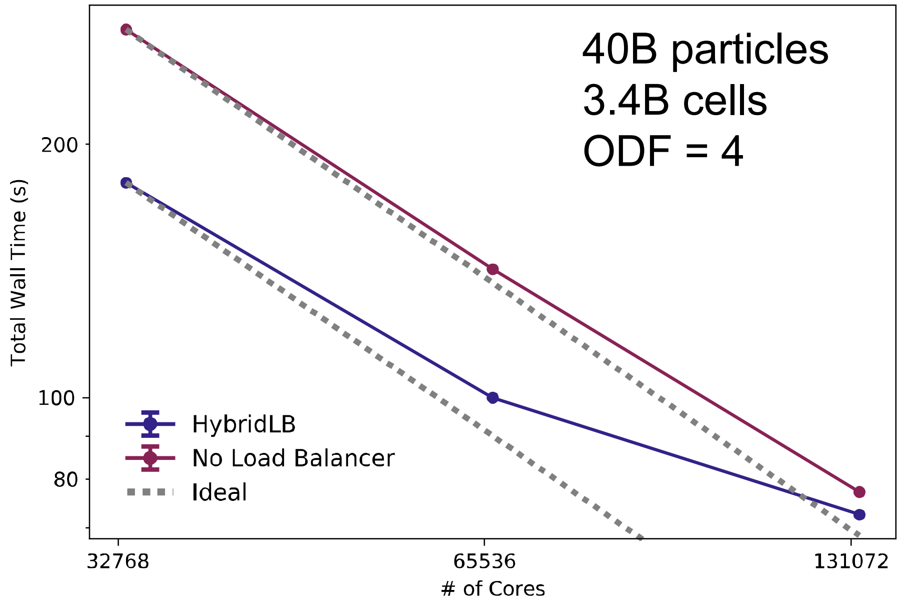
- DARMA scales consistently good on up to 32K cores.
- Slight overheads can be explained by the small problem size on higher core counts.

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

Mutrino (KNL, 2K cores)



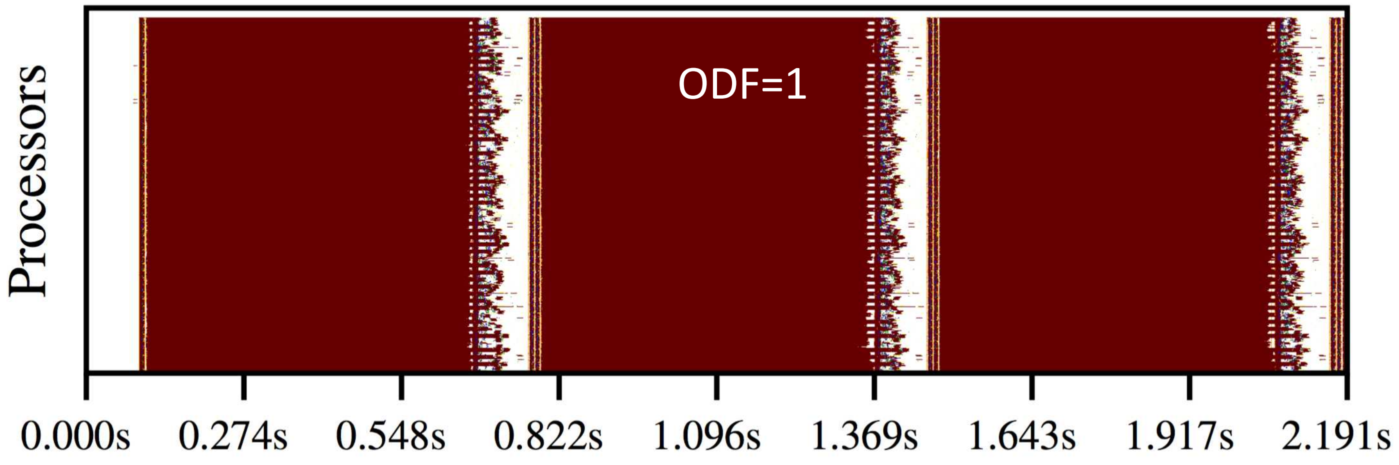
Trinity (KNL, 131K cores)

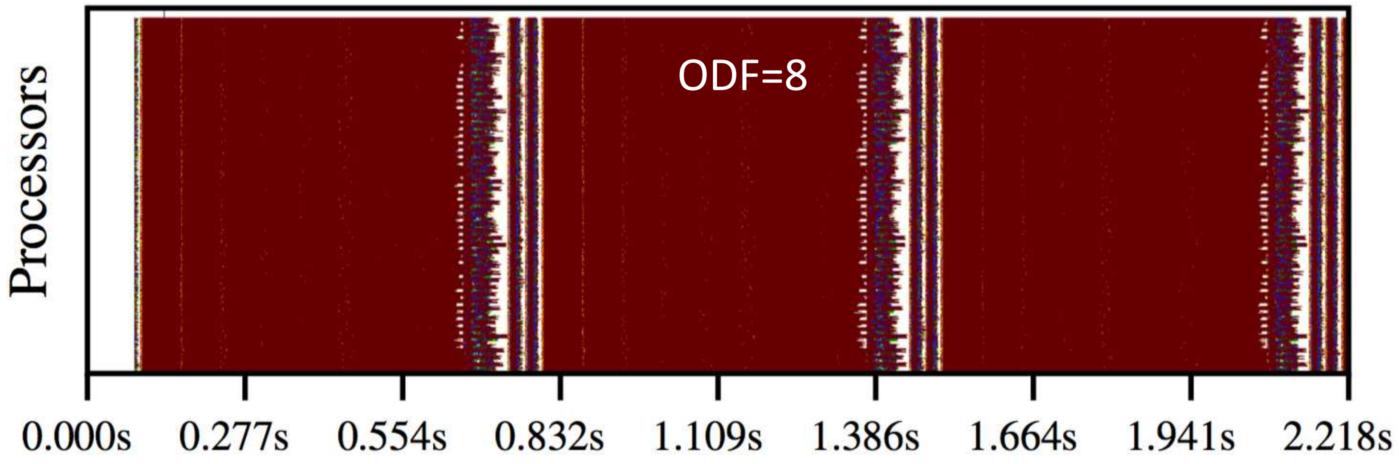


- For lower core counts, load balancing provides around 50% speedup.
- For higher core counts, at least at this overdecomposition level, speed up due to a load balancer is 20%.
- These trends are similar for Haswell.

- 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



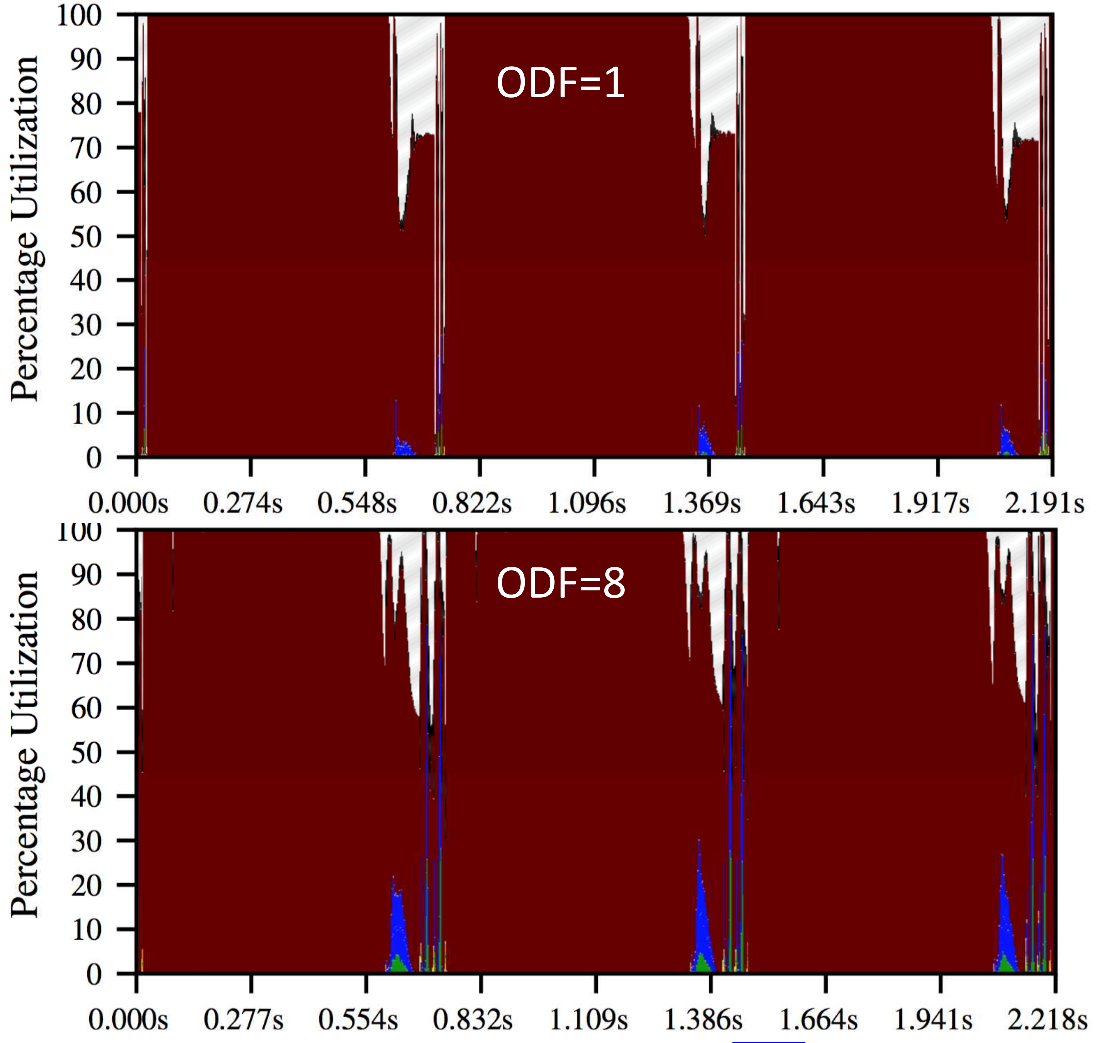


🟥 Application work (tasks)

🟦 Data transfer (send/recv)

- 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 <u>Balanced</u> SimplePIC on 2k Cores/64 nodes (Haswell) for 3 Iterations

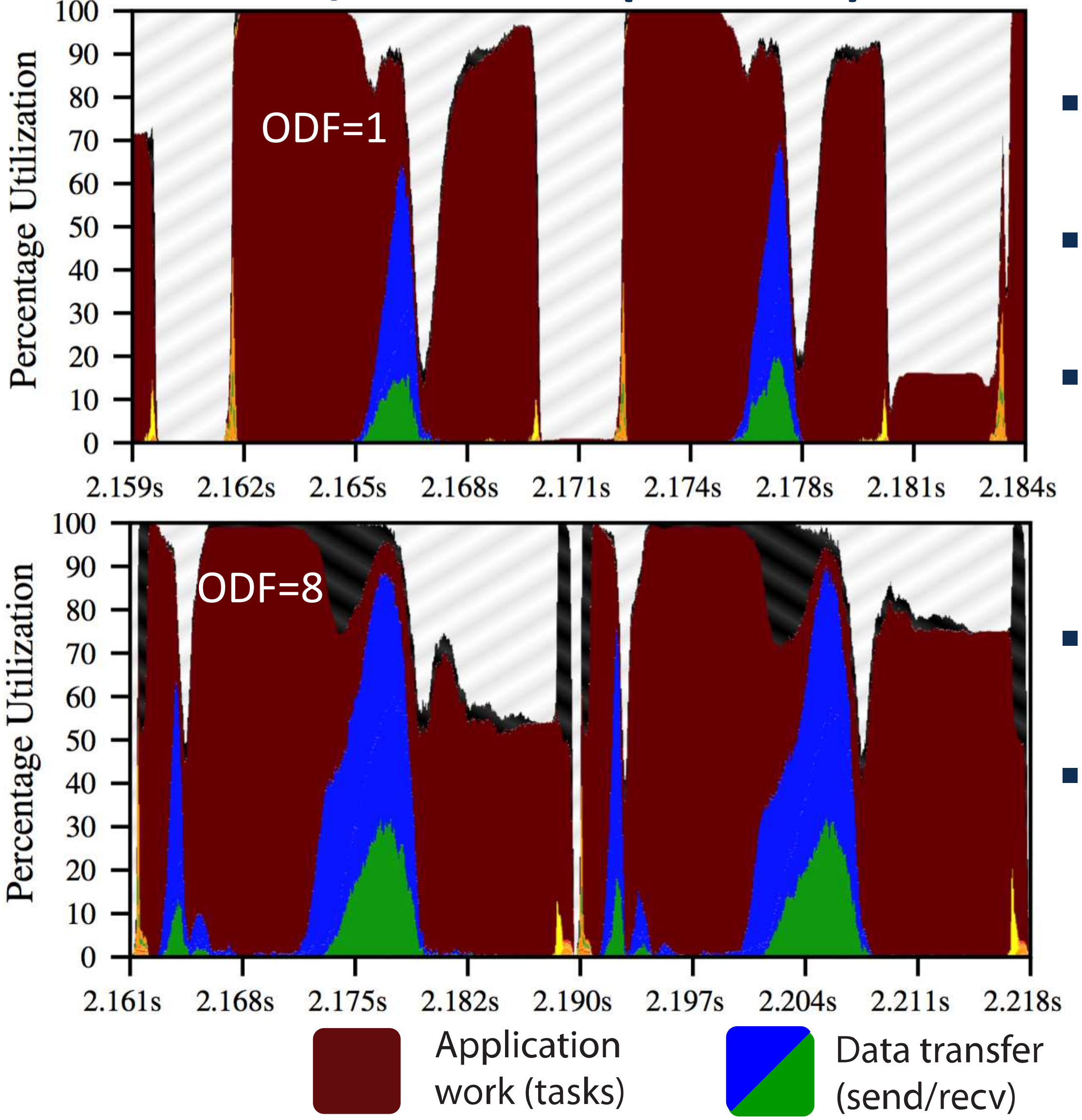


🟥 Application work (tasks)

🟦 Data transfer (send/recv)

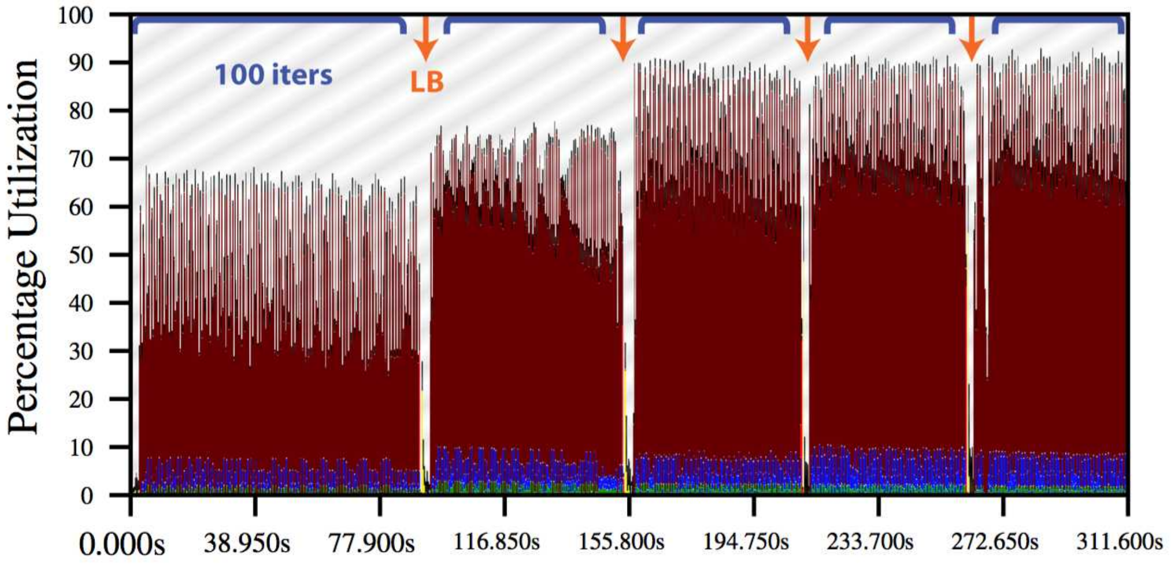
- x-axis is time and y-axis is the proportional aggregate of work type spent across the worker cores
- With an overdecomposition factor of 8 (ODF=8) the data transfer time is slightly increased
- The idle time at the end of the iteration is slightly reduced with ODF=8 because the system is able to overlap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

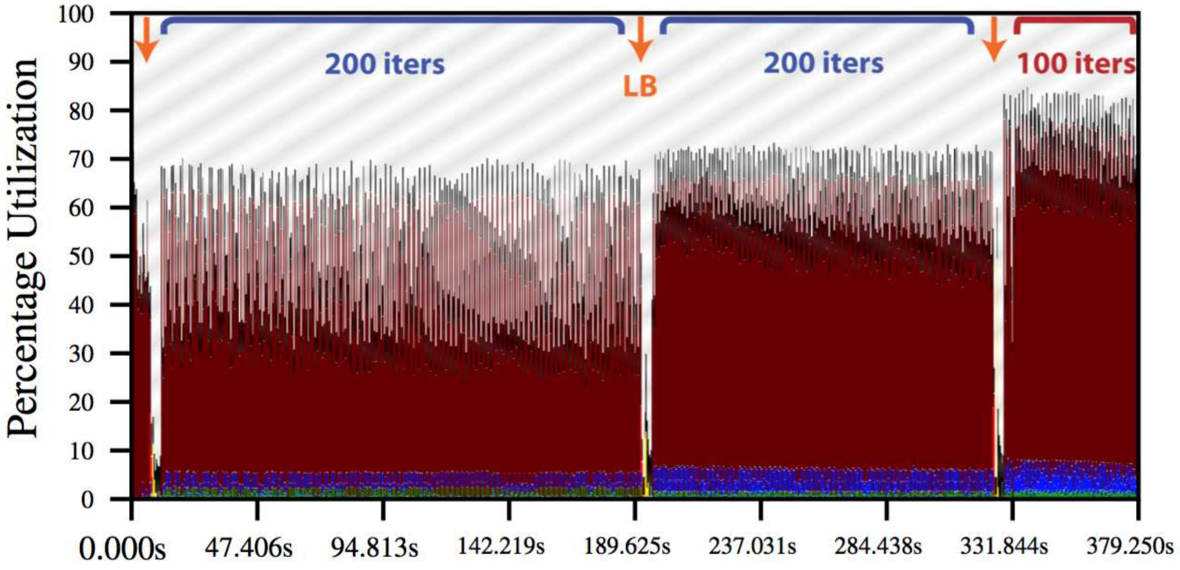
DARMA Time Profile Graph of Balanced SimplePIC on 2k Cores/64 nodes (Haswell) for last 2 micro iterations



- 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

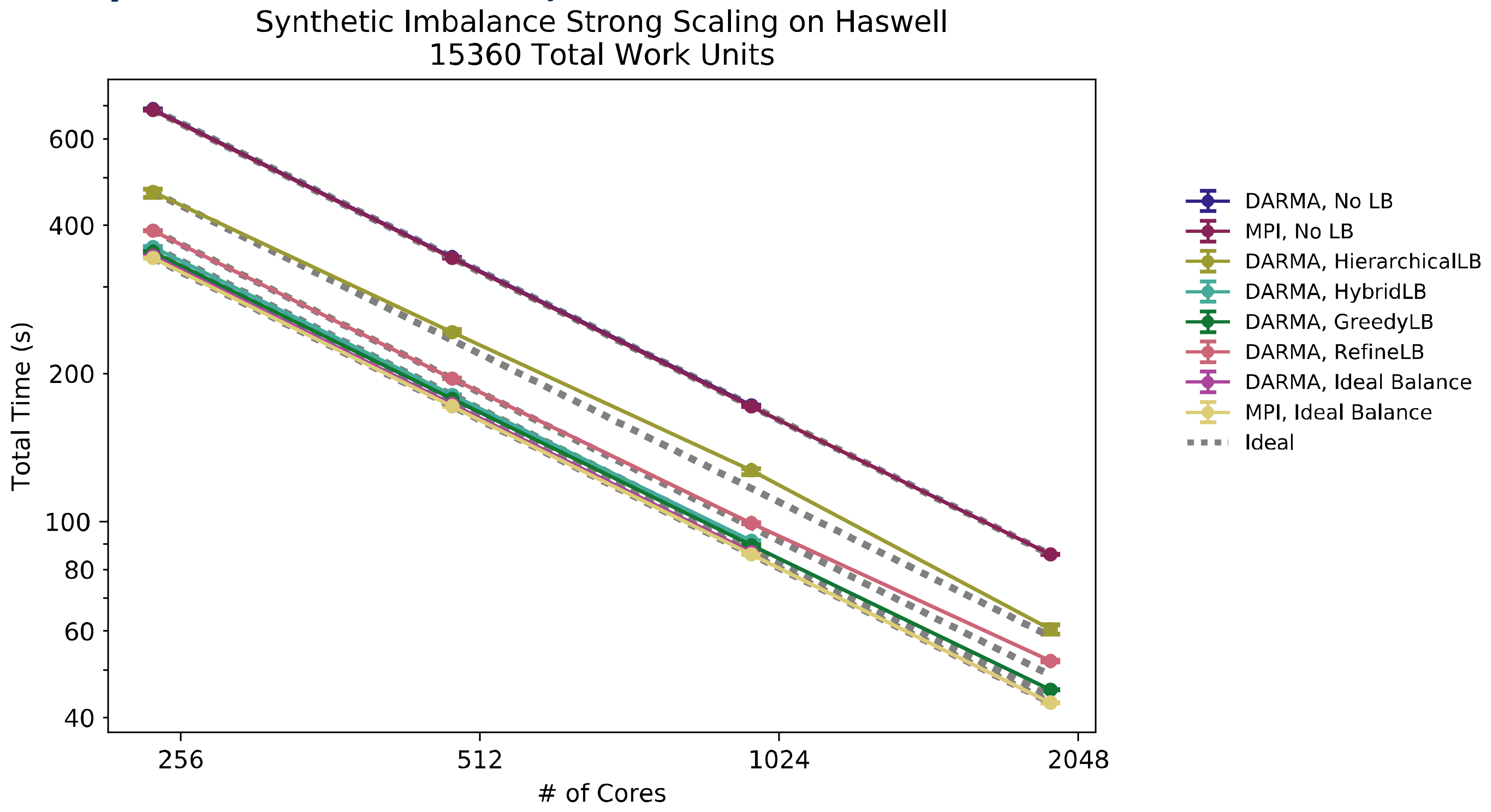
Summary of quotes on productivity from our application developers

- "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)"

Different load balancers have cost, scaling, and optimality tradeoffs

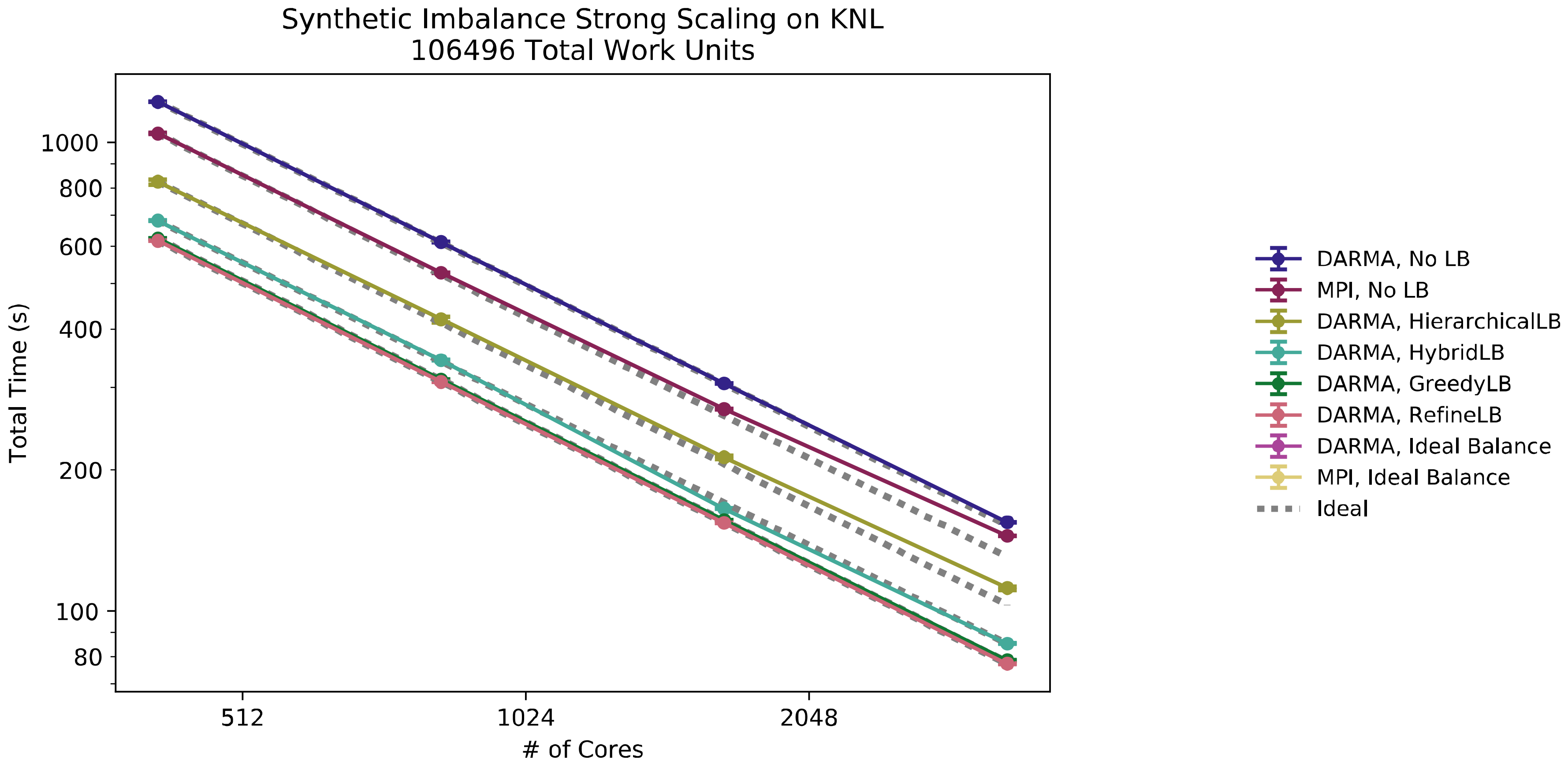
LB Type	LB Name	Description	Benefits	Drawbacks
Centralized	**GreedyLB**	Heap-based, considers all tasks for redistribution	Provides high quality distribution	Not scalable, expensive in memory and space
Centralized	**RefineLB**	Heap-based, considers only tasks above threshold	Fast for centralized load 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



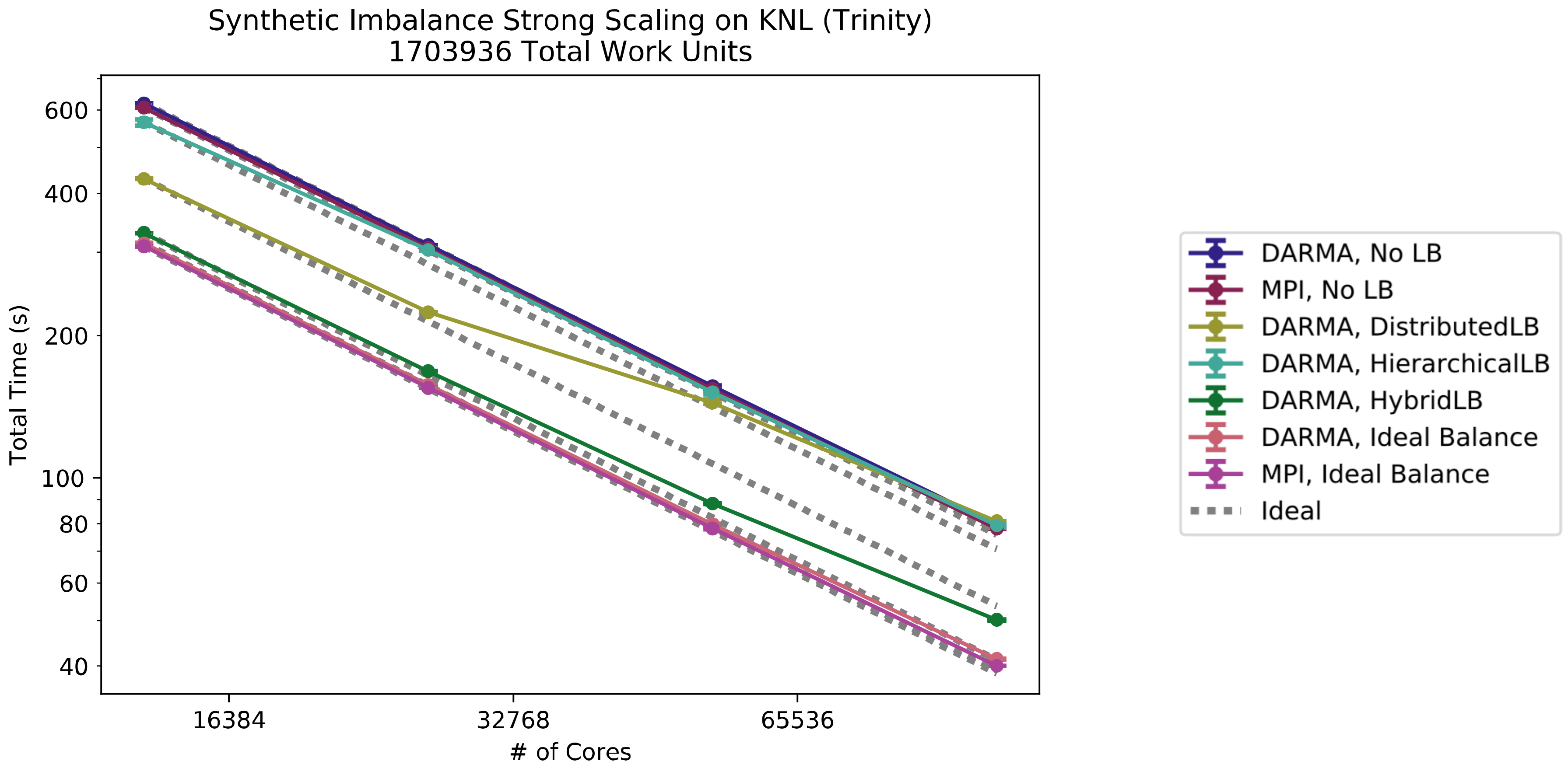
- Only Greedy, Hybrid load balancers competitive with optimal balance baseline
- All load balancers relatively scalable up to 64 nodes, different quality solutions though
- All load balancers better than worst-case baseline with no load balancing

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



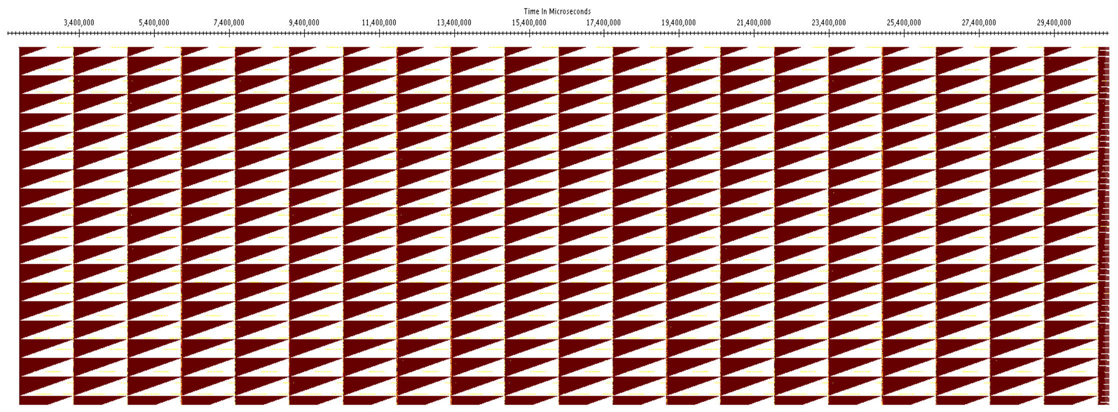
- Only Greedy, Hybrid load balancers competitive with optimal balance baseline
- All load balancers relatively scalable up to 64 nodes, different quality solutions
- All load balancers better than worst-case baseline with no load balancing

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



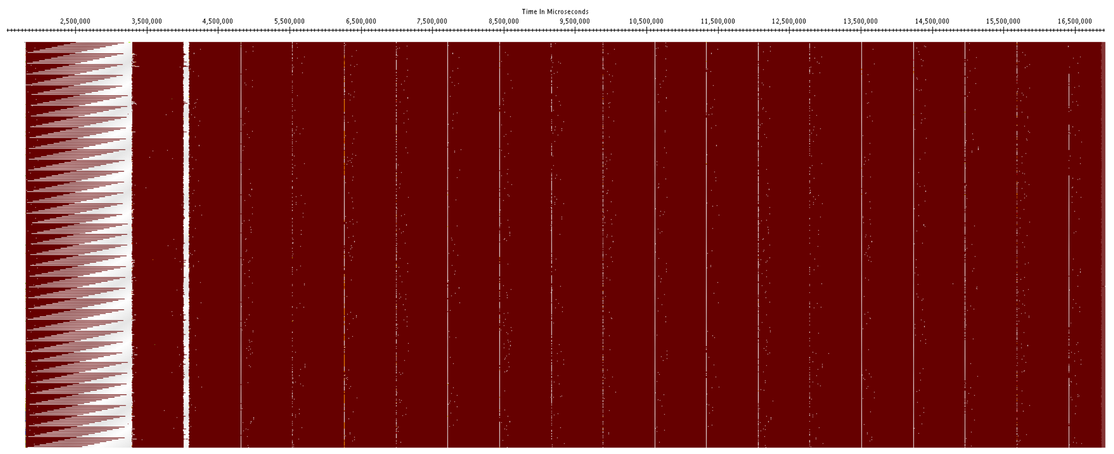
- Refine load balancers skipped, could not finish in 30 minute time cutoff
- All load balancers still relatively scalable, Hierarchical has best scalability but worse quality of load balance
- Only hybrid load balancer gets near optimal balance with low load balance overheads

Load balancers redistribute work, shrink idle time between iterations



No load balancing

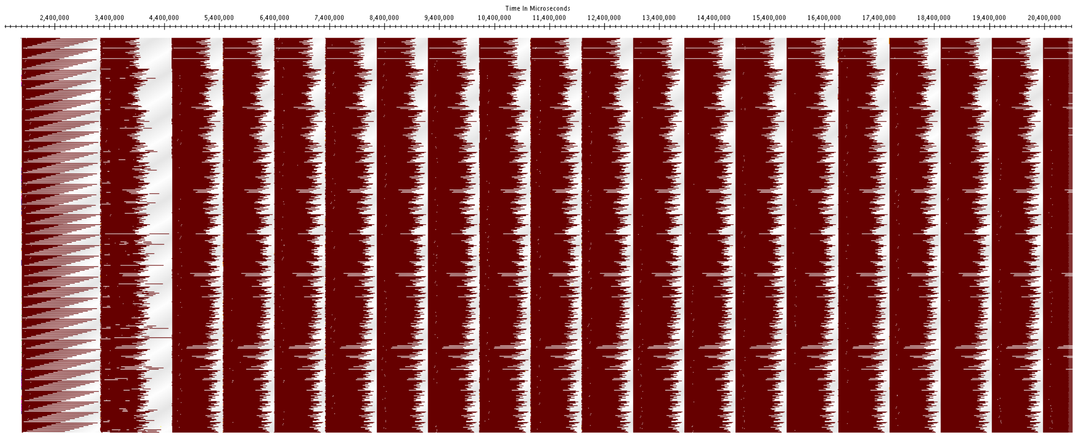
- Red is active computation, white is idle time for each thread
- Execution shows certain threads idling while large tasks finish



Greedy load balancer

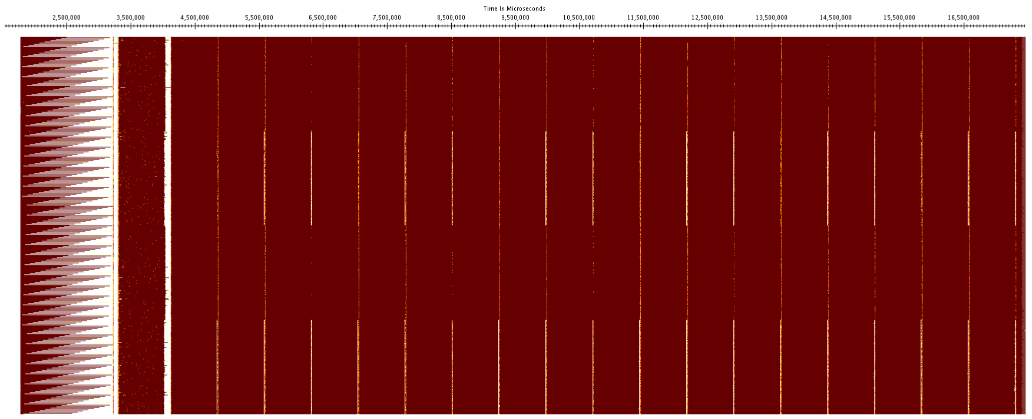
- First iteration imbalanced, but idle time shrinks as load balancer finds nearly optimal solution on second iteration

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

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:**
  - **Experiments up to 2K nodes show that DARMA is scalable (weak and strong)**
  - **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**
- **Generality: declarative backend specification facilitates mapping to different technologies, development of "common components" across backend implementations**

Future Work

- Focus of DARMA team next year
  - Interoperability with node-level libraries
  - Hardening/Tuning
  - Productivity tools (timers, performance profilers, debugging aides)
  - Development of MPI backend
- Bigger picture/longer term efforts
  - Best practices and standards-based runtime solutions

SUN