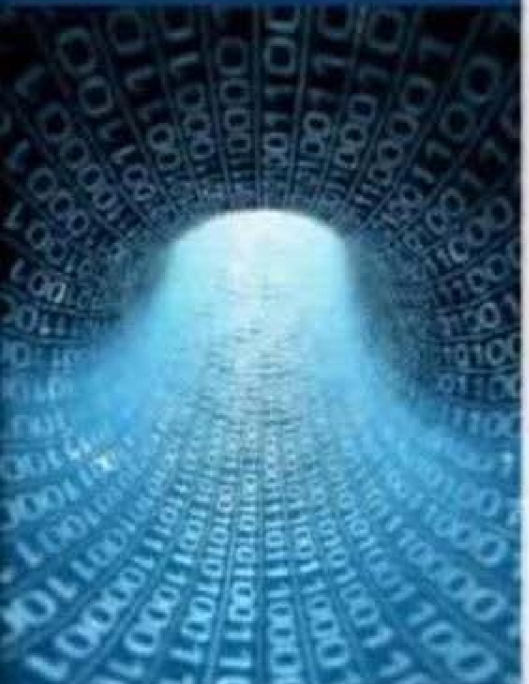


A Comparative Analysis of Asynchronous Many-Task Programming Models for Next Generation Platforms

Janine Bennett, Hemanth Kolla, Jeremiah Wilke, Ken Franko, Paul Lin, Greg Sjaardema, Nicole Slattengren, Keita Teranishi, Samuel Knight

SIAM CSE 2015
3/16/2015
MS 129 DAG-Based Efficient Scalable & Portable PDE Software

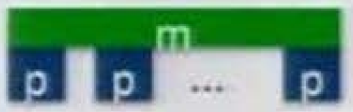
Sandia National Laboratories

Exceptional service in the national interest

ENERGY NNSA

Performance and programmability are achieved by targeting an underlying abstract machine model

Machine model: PRAM/SMP



Programming model: threads

Machine model:
Bulk Synchronous Model

Programming model: MPI

Machine model: Hybrid Candidate Type Architecture (CTA)

Programming model: Hybrid Bulk Synchronous MPI + X

Consider the abstract machine model of an exascale node

Overarching abstract machine model of an exascale node

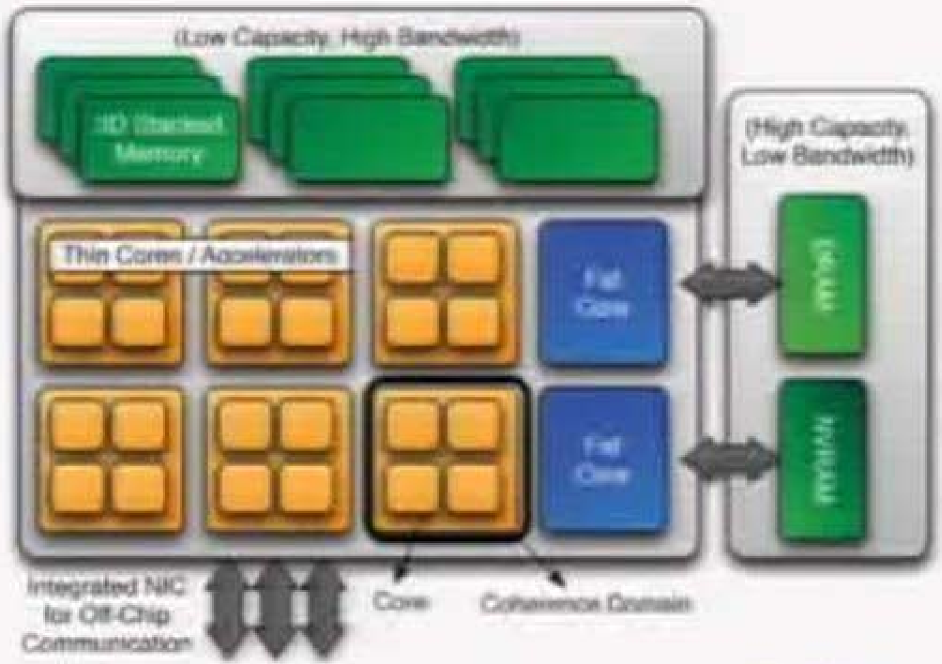


Image courtesy of www.cal-design.org

This new abstract machine model introduces significant complexities

Challenges
▪ Increases in concurrency
▪ Deep memory hierarchies
▪ Increased fail-stop errors
▪ Performance heterogeneity ▪ Accelerators ▪ Thermal throttling ▪ General system noise ▪ Responses to transient failures

Overarching abstract machine model of an exascale node

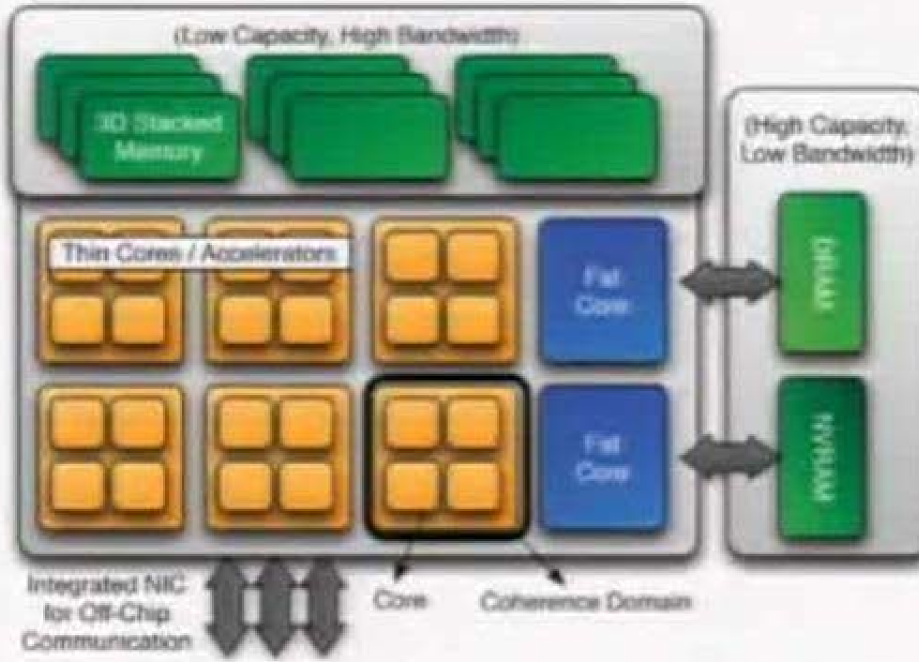
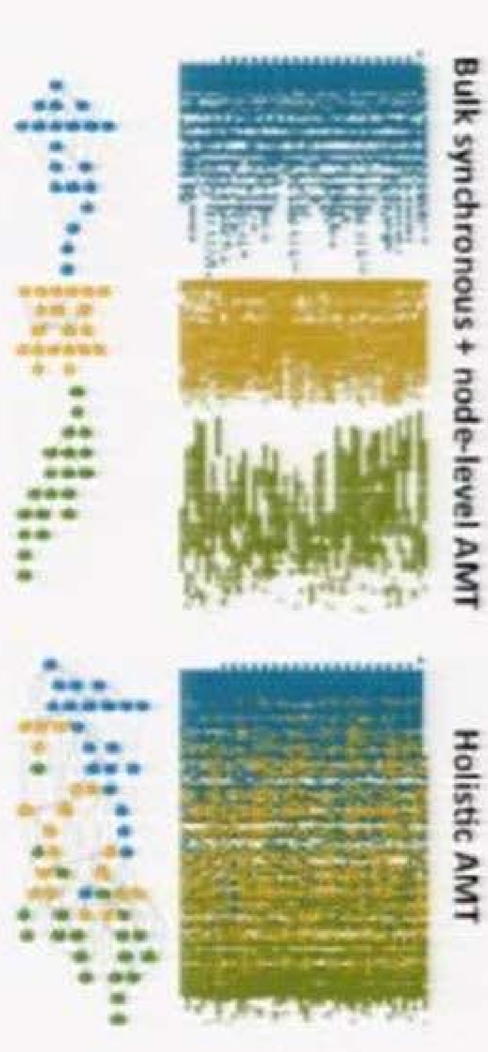


Image courtesy of www.cal-design.org

Asynchronous many-task (AMT) programming models show promise against exascale challenges

- Runtime systems show promise at sustaining performance despite node-degradation and failure
- Data flow programming model
  - Tasks are nodes in graph
  - Data dependencies are edges in graph
- Facilitate expression of task- and data-parallelism
- Has an active research community
  - Charm++, DHARMA, HPX, Legion, OCR, STAPL, Uintah, …



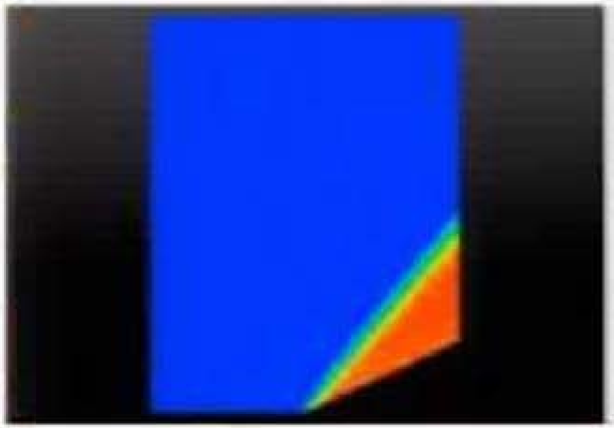
Images courtesy of Jack Dongarra

With so many variants, how do you know which is right for your application?

- Charm++ (UIUC)
- DHARMA (SNL)
- HPX (IU/LSU)
- Legion (Stanford)
- OCR (Intel/Rice/...)
- STAPL (Texas A&M)
- Uintah (U. Utah)
- ...

Sandia ASC-funded comparative analysis study

- Overarching goal: Provide guidance to the code development road map for Sandia ASC (Advanced Simulation and Computing) codes, based on in-depth exploration using realistic proxies
- Starting with MiniAero
  - Fully 3D unstructured finite volume
  - Runge-Kutta 4th order time marching
  - 1st or 2nd order in space
  - Inviscid Roe Flux and Newtonian Viscous fluxes
  - Boundary Conditions: Supersonic inflow, supersonic outflow, and tangent flow
  - ~3800 lines of C++ code (> 850 in mesh generation)
  - Minimal dependencies (Kokkos)
  - Data-parallel not task-parallel
- Given time/resources: MiniPIC, MiniFE, MiniContact



Comparative study (work in progress)

- Initial MiniAero implementations in Charm++, Legion, Uintah nearly complete
  - OCR implementation to begin in April
  - MiniAero implementations will be made available at Mantevo.org
- Tight coupling of Sandia runtime developers, application developers, and University/Industry contacts
- Assessing the *programmability*, *mutability*, and *performance* of various runtimes in the context of ASC workloads

Assessing programmability

- Does this programming model and runtime system support the natural expression and execution of the ASC applications of interest?

- Planned activities:
  - Gather qualitative feedback from application developers
    - Rate abstractions, APIs, ease of use, etc.
  - Collect quantitative data
    - Size of code, length of time to code/optimize, etc.

Assessing performance

- What are the scaling properties and performance of the mini app in this runtime system before and after performance optimization?
- How do they compare with the bulk-synchronous implementation?
- How does the scaling of the mini app in this runtime system change with task granularity and different levels of over-decomposition?
- How does this runtime system provide support for dynamic load balancing?
- Can the application scientist directly control load balancing and/or provide load-balancing hints (e.g., physics/domain specific knowledge)?
- How well does the runtime system support fault containment and recovery?
- How does this runtime system facilitate code coupling (e.g. in situ analysis and visualization, multi-physics)?
- How do the implementations compare from a power/energy perspective?

Assessing performance

- Planned activities:
  - Weak and strong scaling studies
  - Work-granularity studies
    - Data: over-decomposition levels
    - Task: granularity (how much code is in the task)
  - Load balancing studies
    - System-induced imbalance
    - Application-induced imbalance
  - Given sufficient time/resources
    - Fault tolerance experiments
    - Gather power/energy usage

Overarching design decisions

Charm++	• Interacting collections of over-decomposed objects (Chares) • Asynchronous methods invoked on remote objects • Adaptive runtime system optimizes performance
Legion	• Logical regions: expressive relational data model • Understanding of data automates task-graph and movement • Decouple code specification from mapping to system
OCR	• Fine-grained, event-driven, moveable tasks • Elastic runtime with flexible distribution • Open source community involvement
Uintah	• Runtime development driven by application needs at scale • Application code runs "unchanged" from 600 to 600K cores • Asynchronous out-of-order execution, work stealing

Additional detail can be found in summary slides from Supercomputing 2014 BOF:
"Asynchronous Many-Task Programming Models for Next Generation Platforms"

Many issues and open research questions remain

- Need to characterize runtime system performance for broad classes of algorithms and architectures
  - What is the right granularity of work?
    - What is the right level of over-decomposition?
    - How much work should a task comprise?
    - How do these numbers differ for load-balancing intra- & inter-node?
  - Need to be careful regarding use of Mini Apps – they don't tell the entire story

- Need continued increased engagement/feedback from application developer community in comparative studies
  - ExMatEx summer schools, this study are a start but not sufficient

Many issues and open research questions remain

- Need for increased investment in debuggers, performance optimization, compiler support
- Need for algorithmic (applied mathematics) research
  - Develop new techniques that leverage increased runtime system asynchrony
- Standardization -- at a minimum we need community agreement regarding definitions of terms