Parallel Programming Futures: What We Have and Will Have Will Not Be Enough

Michael Heroux

SOS 22
March 27, 2018

Exceptional service in the national interest

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.

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.

General Reality of Multicore Parallelism*

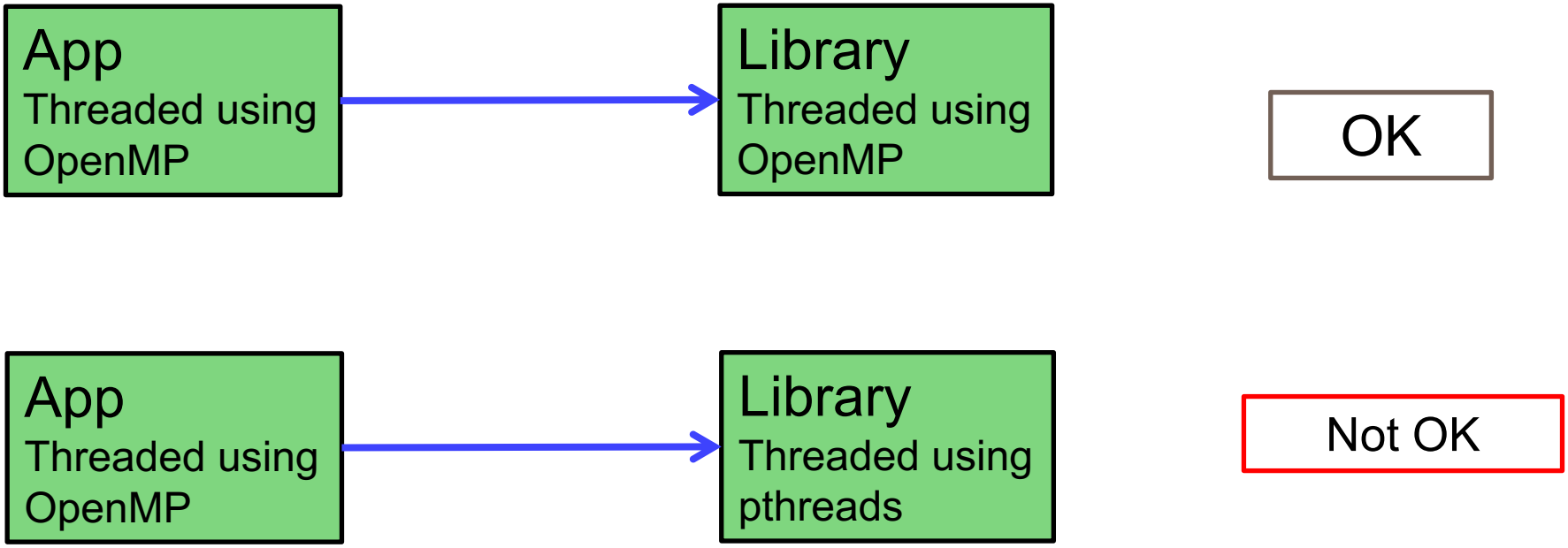
Sandia National Laboratories

- Best single shared memory parallel programming environment:
  - MPI.
- But:
  - Two level parallelism (MPI+X) is generally more effective.
- But, the best option for X (if explored at all) is:
  - MPI.
- Furthermore, for an (N_1 x N_2) MPI+X decomposition:

  "For a given number of core counts, the best performance is achieved with the smallest possible N_2 for both hybrid [MPI+OpenMP] and MPI [MPI+MPI] versions. As N_2 increases, the runtime also increases."

*Slide written 5 years ago, progress in the meantime but still true in many codes today.

Threading Multi-API Usage: Needs to work



- Problem: App uses all threads in one phase, library in another phase.
- Intel Sandy Bridge: Not OK **1.16 to 2.5 X slower than** OK .
- Intel Phi: Not OK **1.33 to 1.8 X slower than** OK .
- Implication:
  - Libraries must pick an API.
  - Or support many. Possible, but complicated.

Data Placement & Alignment

Sandia National Laboratories

- First Touch is not sufficient.
  - Happens at initialization. Hard to locate, control.
- Really need placement as first class concept.
  - Create object with specific layout.
  - Create objects compatible with existing object.
- Lack of support limits MPI+OpenMP.
  - OpenMP restricted to single UMA core set.

Threading Futures

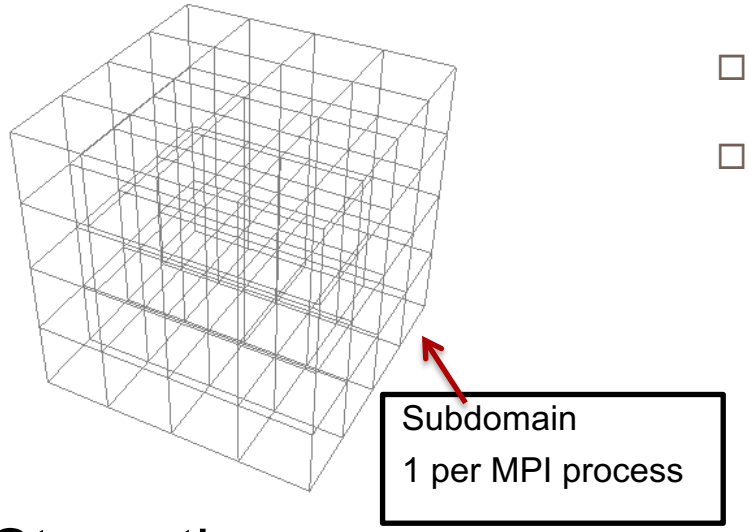
Sandia National Laboratories

- OpenMP may be all we need, if:
  - We move to task-on-patch design.
  - C++ native threads are compatible.
- Maybe threading challenges will not be an issue:
  - OpenMP vs CUDA vs PTX: Common GPU Runtime
  - Going forward accelerators provide most of the performance potential.
  - The threading challenges on CPUs may not matter.

Sandia National Laboratories

TASK-ON-PATCH/DATAFLOW DESIGN

Classic HPC Application Architecture

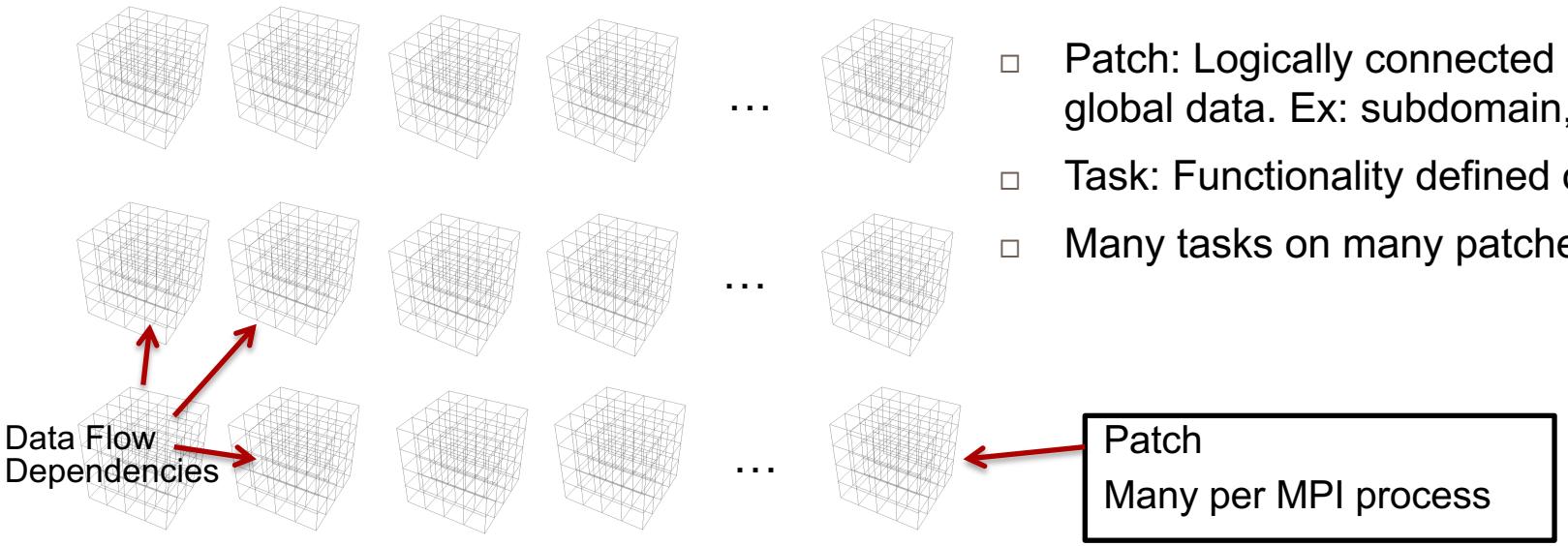


- Logically Bulk-Synchronous, SPMD
- Basic Attributes:
  - Halo exchange.
  - Local compute.
  - Global collective.
  - Halo exchange.

- Strengths:
  - Portable to many specific system architectures.
  - Separation of parallel model (SPMD) from implementation (e.g., message passing).
  - Domain scientists write sequential code within a parallel SPMD framework.
  - Supports traditional languages (Fortran, C).
  - Many more, well known.

- Weaknesses:
  - Not well suited (as-is) to emerging systems.
  - Unable to exploit functional on-chip parallelism.
  - Difficult to tolerate dynamic latencies.
  - Difficult to support task/compute heterogeneity.

Task-centric/Dataflow Application Architecture



- Patch: Logically connected portion of global data. Ex: subdomain, subgraph.
- Task: Functionality defined on a patch.
- Many tasks on many patches.

- Strengths:
  - Portable to many specific system architectures.
  - Separation of parallel model from implementation.
  - Domain scientists write sequential code within a parallel framework.
  - Supports traditional languages (Fortran, C).
  - Similar to SPMD in many ways.

- More strengths:
  - Well suited to emerging systems.
  - Can exploit functional on-chip parallelism.
  - Can tolerate dynamic latencies.
  - Can support task/compute heterogeneity.

Task on a Patch

Sandia National Laboratories

- Patch: Small subdomain or subgraph.
  - Big enough to run efficiently once its starts execution.
    - CPU core: Need ~1 millisecond to cover overhead.
    - GPU: Give it big patches. GPU runtime does tasking very well on its own.
- Task code (Domain scientist writes most of this code):
  - Standard Fortran, C, C++ code.
  - E.g. FEM stiffness matrix setup on a “workset” of elements.
  - Should vectorize (CPUs) or SIMT (GPUs).
  - Should have small thread-count parallel (OpenMP)
    - Take advantage of shared cache/DRAM for UMA cores.
  - Source line count of task code should be tunable.
    - Too coarse grain task:
      - GPU: Too much register state, register spills.
      - CPU: Poor temporal locality. Not enough tasks for latency hiding.
    - Too fine grain:
      - Too much overhead or
      - Patches too small to keep task execution at 1 millisec.

Task-centric Status

Sandia National Laboratories

- Few application efforts in this area.
- Similar to the HPF days before explicit distribute memory programming.

Portable Task Coding Environment

Sandia National Laboratories

- **Task code must run on many types of cores:**
  - Multicore.
  - GPU (Nvidia).
  - Future accelerators.
- **Desire:**
  - Write single source, but be realistic.
  - Compile phase adapts for target core type.
- **Current approaches: Kokkos, OCCA, RAJA, …:**
  - Enable meta programming for multiple target core architectures.
- **Emerging and future: Fortran/C/C++ with OpenMP:**
  - Limited execution patterns, but very usable.
  - Like programming MPI codes today: Déjà vu for domain scientists.
- **Other future: C++ with Kokkos/OCCA/RAJA derivative in std namespace.**
  - Broader execution pattern selection, more complicated.

New Task Management Layer

Sandia National Laboratories

- **Async task launch:** latency hiding, load balancing.
- **Inter-task dependencies:**
  - Data read/write (Legion).
    - Task A writes to variable x, B depends on x. A must complete before B starts.
  - Futures:
    - Explicit encapsulation of dependency. Task B depends on A's future.
  - Alternative: Explicit DAG management (hard to do).
- **Temporal locality:**
  - Better to run B on the same core as A to exploit cache locality.
- **Data staging:**
  - Task should not be scheduled until its data are ready:
    - If B depends on remote data (retrieved by A).
- **Heterogeneous execution:** A on device 0, B on device 1.
- **Resilience:** If task A launched task B, A can relaunch B if B fails or times out.

Challenges for Task-on-Patch/Dataflow Strategies

Sandia National Laboratories

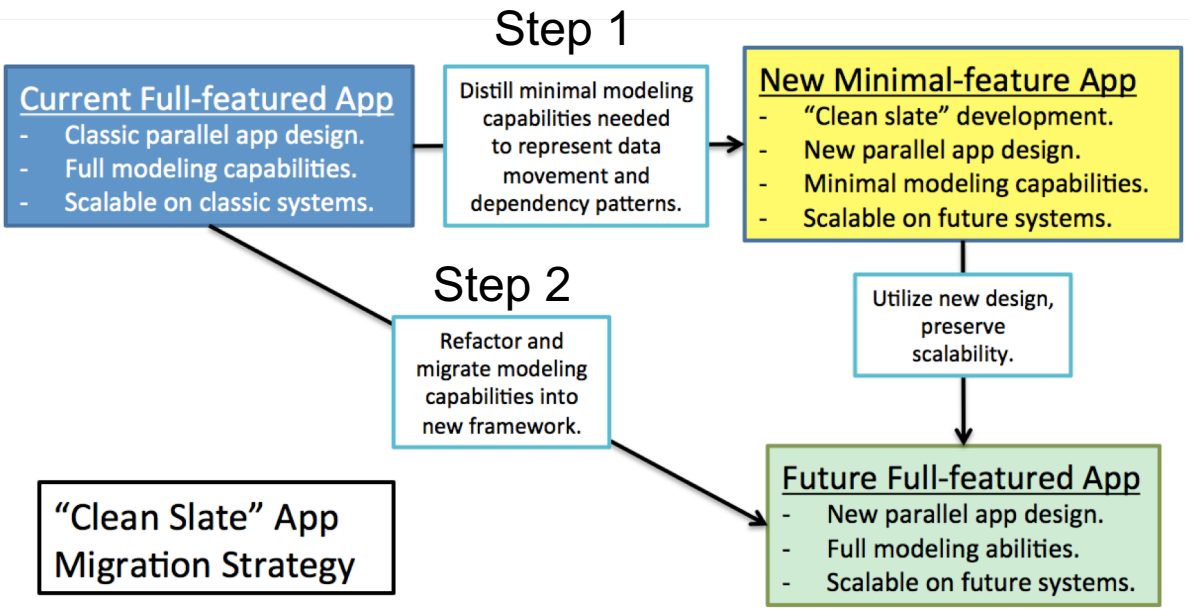
- **Functional vs. Data decomposition.**
  - Over-decomposition of spatial domain:
    - Clearly useful, challenging to implement.
  - Functional decomposition:
    - Easier to implement. Challenging to execute efficiently (temporal locality).
- **Dependency specification mechanism.**
  - How do apps specify inter-task dependencies?
  - Futures (e.g., C++, HPX), data addresses (Legion), explicit (Uintah).
- **Roles & Responsibilities:** App vs Libs vs Runtime vs OS.
- **Interfaces between layers.**
- **Huge area of R&D for many years.**

Data challenges:

- Read/write functions:
  - Must be task compatible.
  - Thread-safe, non-blocking, etc.
- **Versioning:**
  - Computation may be executing across multiple logically distinct phases (e.g. timesteps)
  - Example: Data must exist at each grid point and for all active timesteps.
- **Global operations:**
  - Coordination across task events.
  - Example: Completion of all writes at a time step.

Movement to Task-on-patch/Dataflow is Disruptive: Use Clean-slate strategies

- Best path to task-centric/dataflow.
- Stand up new framework:
  - Minimal, *representative* functionality.
  - Make it scale.
- Mine functionality from previous app.
  - May need to refactor a bit.
  - May want to refactor substantially.
- Historical note:
  - This was the successful approach in 1990s migration from vector multiprocessors (Cray) to distributed memory clusters.
  - In-place migration approach provided early distributed memory functionality. Failed long-term scalability needs.



Phased Migration from Intra-Node to Inter-Node

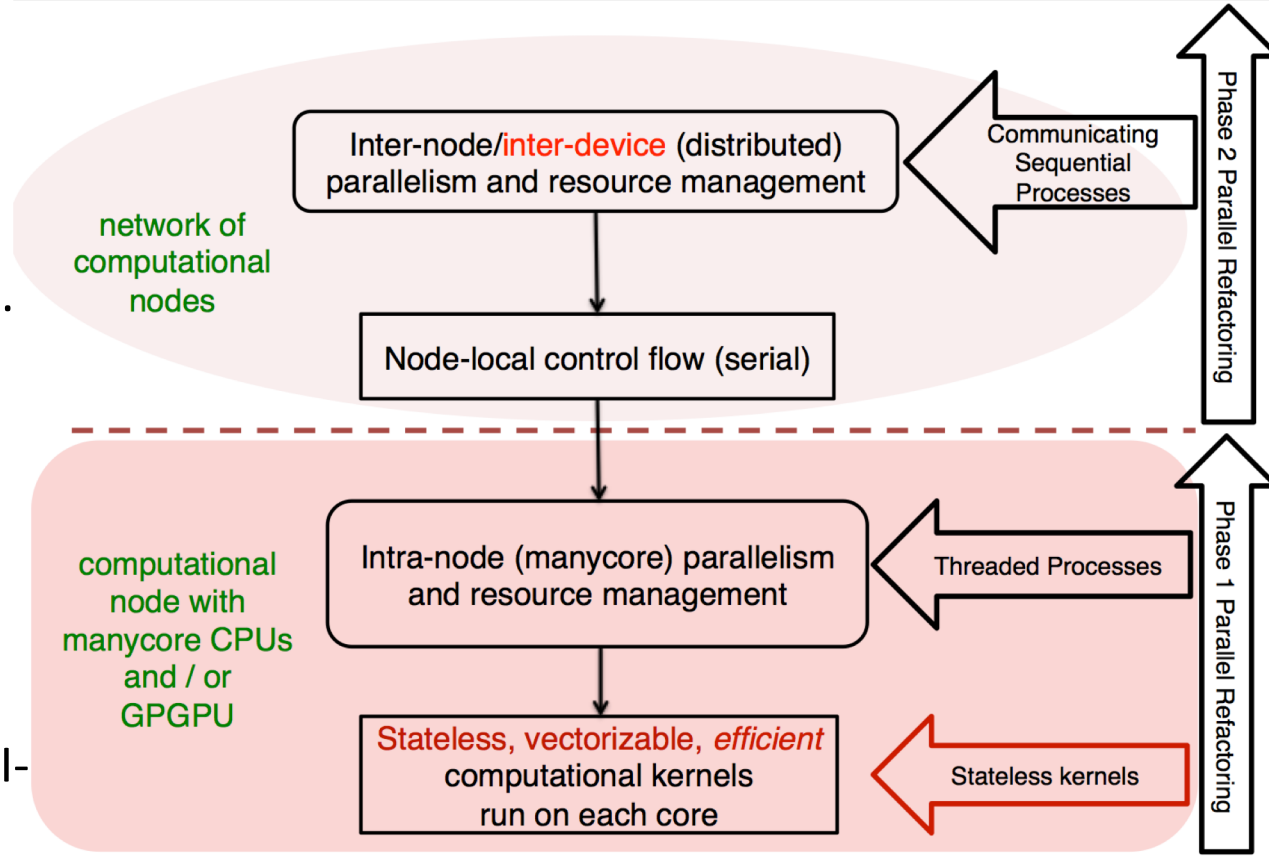
Sandia National Laboratories

Phase 1: Intra-node

- Most urgent need.
- Different algorithms possible:
  - Shared address space.
  - Shared write via atomics.
- Shared goal general computing community.

Phase 2: Inter-node

- Must first understand MPI-based approaches.
- Apps have knowledge of cost models: when migration can work.



15

Sandia National Laboratories

SIMULTANEOUS HETEROGENEOUS EXECUTION: SPECIALIZE HARDWARE

Simultaneous heterogeneous execution

Sandia National Laboratories

- HPCG on Trinity
- 9380 Haswell, 9984 KNL compute nodes.
  - Haswell
    - Processor dimensions: 27x42x17
    - Local grid dimensions: 160x160x112
  - KNL
    - Processor dimensions: 27x42x34
    - Local dimensions: 160x160x152
- HPCG result: 546 TF/s (3rd).
  - Previous 180 TF/s for Haswell only.
- Key Point: For sparse codes, it's about the memory system.

HPCG on SIERRA (Power9's + 4 Voltas):
- About 10% of performance is from Power9's
- Summit 6 GPUs: Power9's less important.
- Both:
  - Code complexity challenging.
  - Runtime system complexity (MPI).
  - Work partitioning.

2-phase porting strategy: TaihuLight

- Management Processing Element (MPE)
  - 64-bit RISC core
  - support both user and system modes
  - 256-bit vector instructions
  - 32 KB L1 i-cache, 32 KB L1 data, both 4 way set associative.
  - 256 KB L2 cache, 8 way set associative
- Computing Processing Element (CPE)
  - 8x8 CPE mesh
  - 64-bit RISC core
  - support only user mode
  - 256-bit vector instructions
  - 64 KB Scratch Pad Mem (1 private per CPE)
    - Can be configured as explicit local mem or SW managed $.
  - Each CPE has own 16KB direct mapped i-cache.

Initial port:
- Vanilla MPI, 1 rank per MPE
- 23.2 GF/s /core
- 4 vector FMA
- 2 pipes
- 16 Flops/cycle FMA
- Peak: 2/65 of node peak

Subsequent optimization:
- Offload any work to CPEs
- 11.6 GF/s /core
- 4 vector FMA
- 1 pipe
- 8 Flops/cycle FMA

CAM-SE to TaihuLight: 2017 Gordon Bell Finalist

Sandia National Laboratories

- CAM-SE: Spectral Element Atmospheric dynamical core
  - Reported:
    - 754,129 SLOC.
    - 152,336 SLOC modified for TaihuLight (20%).
    - 57,709 SLOC added (8%).
    - 12+ team members.
  - Challenges:
    - Reusability of code seems low: Much of the optimization is specific to Sunway CPE processor.
    - Translation effort difficult to accomplish while still delivery science results: Disruptive.
  - Other notable example: Uintah (see Dec 2017 ASCAC talk)
    - Separation of runtime concerns seems to really help, but app-specific.

Sandia National Laboratories

Some Observations from these Efforts

- Even the simplest simultaneous heterogeneous execution is difficult.
  - But maybe most apps won't care: Sequential heterogeneous execution may be sufficient.
  - But some probably will: Hard to support.
- MPI-backbone approach is very attractive.
  - Initial app port to host backbone, hotspot optimization.
  - Investment in portable programming expressions seems essential.
  - Separation of functionality expression and work/data mapping seems essential.

Vectorization and SIMT

Sandia National Laboratories

- Essential commodity curve.
- Reorganizing for vector/SIMT:
  - Common complaint: Need better integer/address performance.
  - Response: Reorganize for unit stride across finite element/volume formulations.
- Challenges:
  - Difficult to abstract away from domain scientist-programmer.
  - Maintaining vector code is difficult: Brittle.
  - Compilers struggle with C++: Both a maturity and language issue.

Sandia National Laboratories

Algorithms and Resilience

Our Luxury in Life (wrt FT/Resilience)

The privilege to think of a computer as a *reliable, digital* machine.

Conjecture: This privilege will persist through Exascale and beyond.

Reason: Vendors will not give us a unreliable system until we are ready to use one, and we will not be ready any time soon.

Take away message

Sandia National Laboratories

If we want unreliable systems,
we must work harder on resilience.

Four Resilient Programming Models

Sandia National Laboratories

Relaxed Bulk Synchronous (rBSP)

Skeptical Programming. (SP)

Local-Failure, Local-Recovery (LFLR)

Selective (Un)reliability (SU/R)

Toward Resilient Algorithms and Applications
Michael A. Heroux arXiv:1402.3809v2 [cs.MS]

Resilience Priorities

Sandia National Laboratories

Focused effort on alternatives to global checkpoint-restart.

Holistic, beginning-to-end resilient application code:

- Tolerates node failure.
- Reduces risk from silent data corruption.

Final Take-Away Points

Sandia National Laboratories

- We are expecting magic from our parallel programming tools.
  - Expect: Memory system efficiency, Massive concurrency.
  - Expect: Ease of expression, Performance portability.
  - Without doing the hard work of refactoring our apps and libraries.
  - We are in the HPF days of highly concurrent node programming.
- The hard work is re-design and refactoring apps.
- Explicit data patch and task partitioning underneath MPI:
  - Better use of spatial locality.
    - Single MPI partition: Pages, cache lines mean too much incidental traffic, pollution of fast memory resources.
    - Patch partitioning: Bring in only the data needed, eliminate false sharing.
  - Better use of temporal locality.
    - Pipelining of functions across a patch: Temporal locality through a sequence of functions.
    - Ability to for cache/register sizes.

Final Take-Away Points

Sandia National Laboratories

Heterogeneous systems:

- Likely strategy in the presence of design constraints.
- Challenging to program.
- Architecture based on “Linux Cluster” backbone attractive.
- Complete thread scalability is a lot of work, difficult to fund.
- Task-on-patch/data centric programming: Still an urgent need.

We still need new features:

- Robust SIMD/SIMT compilation.
- Rapid adaptability to new devices, aka, performance portability.
- Better memory preparation and description: alignment, access contracts.
- Transition of HPX, Kokkos, others: Features into C++ standard.
- Resilience: For when when the reliable machine can’t be delivered.
- Futures: Powerful, simple concept understandable by domain scientists.

Parallel Programming Futures

Sandia National Laboratories

What we have and will have will not be enough:

- Because we don't have enough well-architected apps:
  - Our parallel programming efforts are addressing issues that are not issues.
  - And they are not addressing the real issues well enough.
- If we want better parallel programming capabilities:
  - We need more better-architected apps.
  - Focused efforts to address the needs of these apps.