Programming Models for SOCs in HPC
(A Play in 3 Acts)

Kathy Yelick
Lawrence Berkeley National Laboratory and UC Berkeley
Act I: SOC's in a Strange Land
NERSC is the Primary High Performance Computing Center for Office of Science Use

- DOE/SC allocates NERSC resources for their mission
- Over 5000 users and 700 projects run at NERSC
- They write about 2000 publications per year
- 2 Petascale systems today
  - NERSC-7: Hopper
  - NERSC-8: Edison

The workload is diverse and increasingly complex due to science workflows, integration of data, and demand for higher resolution and scale
NERSC technology leadership includes a path to Exascale

NERSC-8, 30 PF
NERSC-7, 2.6 PF
NERSC-6, 2.6 PF
NERSC-5+, 0.4PF
NERSC-5, 0.1TF
N6/Hopper: First Cray Aries (DARPA HPCS) system
N5/Franklin: First incremental upgrade
N7/Edison: First Intel/Cray System
N8/Cori: First Cray KNL System + Haswell partition configured for data
NERSC-9, 300 PF
NERSC-10 1+EF
Two Steps toward Exascale at NERSC

Cori: energy-efficient architecture on the exascale roadmap
- Over 9,300 Knights Landing compute nodes
- Self-hosted, up to 72 cores, 16 GB high bandwidth memory
- 1,600 Haswell nodes in data partition
- Cray Aries Interconnect

Wang Hall: New computing facility
- 12.5 MW initial capacity
- Expandable to 42MW
- Energy efficient design (PUE < 1.1)

Short walk from Berkeley Campus
Dedication on Nov 12
Materials and Chemistry are a Significant Fraction of the DOE/SC Computing Workload

- 10 codes make up 50% of the workload
- 25 codes make up 66% of the workload
- Edison (Cray with Intel IvyBridge) will be available until 2019/2020
Breakdown of Application Hours

Materials and Chemistry are a Significant Fraction of the DOE/SC Computing Workload

- 10 codes make up 50% of the workload
- 25 codes make up 66% of the workload
- Edison (Cray with Intel IvyBridge) will be available until 2019/2020
- NERSC Exascale Science Applications Program (NESAPAP)
  - New staff, training and partnerships with Intel for KNL
Titan, Mira and Edison represent 3 distinct architectures in SC
- Not performance portable across systems

APEX 2016 and CORAL @ ANL
- Xeon Phi, no accelerator

CORAL 2017
- IBM + NVIDIA

Best case #1: OpenMP4 absorbs accelerator features (likely), but code still requires a big ifdef

Best case #2: Architectures “converge” by 2023, perhaps with co-design help
Act II:
Don’t Fear the Compiler
A Compiler is Just a Translator

- Scientific computing relies heavily on libraries
  - LAPACK and FFTW are widely used at NERSC
- People use languages for their libraries
- Do we need a language? And a compiler?
  - If higher level syntax is needed for productivity
    - We need a language
  - If static analysis is needed to help with correctness
    - We need a compiler (front-end)
  - If static optimizations are needed to get performance
    - We need a compiler (back-end)
Autotuning: Write Code Generators

- **Two unsolved compiler problems:**
  - dependence analysis and
  - accurate performance models

- **Autotuners are code generators plus search**

Work by Williams, Oliker, Shalf, Madduri, Kamil, Im, Ethier,...
What we have and what we need

NERSC survey: what motifs do they use?

- Structured (30%)
- Sparse LA (20%)
- Spectral (30%)
- Particles (20%)
- Monte Carlo (20%)
- Dense LA (10%)
- Adaptive Unstructured (0%)

What code generators do we have?

<table>
<thead>
<tr>
<th>Dense Linear Algebra</th>
<th>Atlas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spectral Algorithms</td>
<td>FFTW, Spiral</td>
</tr>
<tr>
<td>Sparse Linear Algebra</td>
<td>OSKI</td>
</tr>
<tr>
<td>Structured Grids</td>
<td>TBD</td>
</tr>
<tr>
<td>Unstructured Grids</td>
<td></td>
</tr>
<tr>
<td>Particle Methods</td>
<td></td>
</tr>
<tr>
<td>Monte Carlo</td>
<td></td>
</tr>
</tbody>
</table>

Stencils are both the most important motifs and a gap in our tools
Approaches to Autotuning

How do we produce all of these (correct) versions?

- Using scripts (Python, perl, ML, C,..)
- Compiling annotated general-purpose language (X-Tune,...)
- Use preprocessor to generator code (Raja, Kokkos,TiDA)
- Compile a domain-specific language (D-TEC, Halide)
- Domain-specific compiler for domain-specific language (SEJITS)

Several Projects and PIs: Sam Williams, Mary Hall, Dan Quinlan, Armando Fox, Saman Amarsinghe, Armando Solar-Lezama, Jack Dongarra,
Approach #1: Compiler-Directed Autotuning

- **Two hard compiler problems**
  - Analyzing the code to determine legal transformations
  - Selecting the best (or close) optimized version

- **Approach #1: General-purpose compilers (+ annotations)**
  - Use *communication-avoiding optimizations* to reduce memory bandwidth
  - Apply *CHiLL compiler* technology with general polyhedral optimizations
  - Use autotuning to select optimized version

---

**Smooth**

- **All Optimizations**
- **+Fusion & Wavefront**
- **+Fusion & Partial Sums**
- **+Fusion**
- **Baseline**
- **Roofline Memory Bound**

---

**Results on Geometric Multigrid (miniGMG Smoother)**

- **Edison**
- **Hopper**

---

**Speedup**

- **Hopper**
- **Edison**

---

**MG-Solve**

- **smooth() on 64^3**
Approach #2: DSLs with General Purpose Compiler

- Generation of Complex Code for 10 Levels of Memory Hierarchy with SW managed cache
  - 4th order stencil computation from CNS Co-Design Proxy-App
  - Same DSL code can generate to 2, 3, 4, ... levels too

- Code size of autogenerated code

<table>
<thead>
<tr>
<th>Memory Hierarchy</th>
<th>2 Level</th>
<th>3 Level</th>
<th>4 Level</th>
<th>...</th>
<th>10 level</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSL Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>Auto Generated Code</td>
<td>446</td>
<td>500</td>
<td>553</td>
<td></td>
<td>819</td>
</tr>
</tbody>
</table>

Use of Rose/PolyOpt to apply DSLs to large applications and collaboration on AMR
Approach #3: Domain-Specific (but not too specific)
Languages used by other markets

Developed for Image Processing

- 10+ FTEs developing Halide
- 50+ FTEs use it; > 20 kLOC

HPGMG (Multigrid on Halide)

- Halide Algorithm by domain expert
- Halide Schedule either
  - Auto-generated by autotuning with opentuner
  - Or hand created by an optimization expert

Halide performance

- Autogenerated schedule for CPU
- Hand created schedule for GPU
- No change to the algorithm

![Graph showing execution time comparison between Original and Halide CPU for different problem sizes (1^3 to 256^3).]
Approach #4: Small Compiler for Small Language

- **SEJITS**: Selected Embedded Just-In-Time Specialization:
  - General optimization framework (Ctree)
  - Currently implemented part of HPGMG benchmark in stencil DSL
    - Within 50% of hand-optimized code
    - ~1000 lines of DSL-specific code; 1 undergrad over <2 months
Act III:
Overhead Can’t be Tolerated
• **LogGP: no overlap**

  - $O_{send}$
  - $L$
  - $O_{recv}$

  ![Diagram](Diagram1)

• **Observed: overheads can overlap: $L$ can be negative**

  - $O_{send}$
  - $O_{recv}$

  ![Diagram](Diagram2)

  **EEL:** end to end latency (instead of transport latency $L$)
  
  - $g$: minimum time between small message sends
  - $G$: additional gap per byte for larger messages
Communication and Manycore: the problem is the “+”

- MPI + X today:
  - Communicate on one lightweight core
  - Reverse offload to heavyweight core
- MPI stack may not run well on lightweight cores
- Issues preventing efficient interoperability:
  - Addressability: can’t name remote threads?
  - Separability: How to manage communication resources for independent paths
- More feasible for 1-sided than 2-sided

Ideal hybrid programming

Default hybrid programming

Khaled Ibrahim, ICS 2014
Avoid Latency and Implicit Synchronization

- Two-sided message passing (e.g., send/receive in MPI) requires matching a send with a receive to identify memory address to put data
  - Couples data transfer with synchronization, which is sometimes what you want

- Using global address space decouples synchronization
  - Pay for what you need!
Bandwidths on Cray XE6 (Hopper)

- Berkeley UPC
- Cray UPC
- Cray MPI

Bandwidth (MB/s)

Msg. size

UPC/MPI
Lightweight Communication for Lightweight Cores

• DMA (Put/Get)
  – Blocking and non-blocking (completion signaled on initiator)
  – Single word or Bulk
  – Strided (multi-dimensional), Index (sparse matrix)

• Signaling Store
  – All of the above, but with completion on receiver
  – What type of “signal”?
    • Set a bit (index into fixed set of bits 😞)
    • Set a bit (second address sent 😊)
    • Increment a counter (index into fixed set of counters 😋)
    • Increment a counter (second address for counter 😍)
    • Universal primitives: compare-and-swap (2nd address + value), fetch-and-add handy but not sufficient for multi/reader-writers 😊

• Remote atomic (see above) – should allow for remote enqueue

• Remote invocation
  – Requires resources to run: use dedicated set of threads?

DEGAS Overview  23
Technology Transfer Paths

• Languages
  – Adoption into popular programming models
    • One-sided into MPI (again)
    • Locality control into OpenMP
  – Adoption by a compiler community (Chemistry DSL)

• Compilers
  – Leverage mainstream compilers (LLVM)
  – Leverage another existing “domain-specific” language
  – Small compilers for small languages

• Next phase
  – Focus on application partnerships
  – Partnerships with library and framework developers
  – Collaborate with vendors on hardware desires and constraints
Thank you!
Sources of Unnecessary Synchronization

**Loop Parallelism**

```
!$OMP PARALLEL DO
  DO I=2,N
    B(I) = (A(I) + A(I-1)) / 2.0
  ENDDO
!$OMP END PARALLEL DO
```

“Simple” OpenMP parallelism implicitly synchronized between loops

**Abstraction**

LAPACK: removing barriers ~2x faster (PLASMA)

**Libraries**

<table>
<thead>
<tr>
<th>Analysis</th>
<th>% barriers</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto</td>
<td>42%</td>
<td>13%</td>
</tr>
<tr>
<td>Guided</td>
<td>63%</td>
<td>14%</td>
</tr>
</tbody>
</table>

NWChem: most of barriers are unnecessary (Corvette)

**Accelerator Offload**

The transfer between host and GPU can be slow and cumbersome, and may (if not careful) get synchronized
Random Access to Large Memory

Meraculous Assembly Pipeline

Perl to PGAS: Distributed Hash Tables
• Remote Atomics
• Dynamic Aggregation
• Software Caching (sometimes)
• Clever algorithms and data structures (bloom filters, locality-aware hashing)

→ UPC++ Hash Table with “tunable” runtime optimizations

Human: 44 hours to 20 secs
Wheat: “doesn’t run” to 32 secs

Grand Challenge: Metagenomes

Productivity: Enabling a New Class of Applications?