Solved Problems, Unsolved Problems, and Non-Problems in Exascale Programming Models

Kathy Yelick
Associate Laboratory Director for Computing Sciences
Lawrence Berkeley National Laboratory
and
Professor of Electrical Engineering and Computer Science
UC Berkeley
Challenges from Above and Below

• Application challenges:
  the point is to do new science!

• Architectural challenges:
  for performance growth of any kind
Generic 2020+ Node Architecture

Memory Stacks on Package
Low Capacity, High Bandwidth, Software Control?

Lightweight Cores
(tiny, simple, massively parallel)
Throughput-Optimized

Bulky Cores
Latency Optimized

DRAM

DRAM

DRAM

NVRAM: Burst Buffers / rack-local storage (software control)

Based on slide from J. Shalf
1. **Lightweight cores** will have all/most of the system performance
   - Need fine-grained parallelism; avoid unnecessary synchronization
   - Cores not powerful enough for complex communication protocols?

2. **On-chip interconnect** offers opportunities for performance
   - New models of communication may be essential

3. **Hardware is heterogeneous**: no single ISA
   - Portability and performance portability are challenging

4. **New levels of memory hierarchy**, possibly software-controlled
   - Locality and communication-avoidance paramount

5. **Performance variability** may increase
   - Software or hardware control clock speeds

6. **Resilience** will be paramount at scale
   - Failures grow with the number of components and connections
Major Programming Model Research Areas

• **Performance Portability through Compilers and Autotuning**
  – Automatically generate GPU and CPU code & automatically tune
  – *E.g.*, Rose (D-TEC, LLNL), Halide (D-TEC, MIT), CHiLL (X-Tune, Utah), SEJITS (DEGAS, UCB), Legion (ExaCT, Stanford/LANL), SLEEC (Purdue)

• **Data Locality in Languages and Libraries**
  – Specify location of data (Partitioned Global Address Space)
  – *E.g.*, UPC/UPC++ (LBNL), CAF (Rice), TiDA (LBNL), RAJA (LLNL), KOKKOS (SNL)

• **Less Synchronous DAG Execution Models**
  – Static and dynamic DAG construction
  – *Examples*: OCR (Intel), HPX (XPRESS), Charm++ (UIUC), Legion (Stanford/LANL), Habanero (Rice)

• **Correctness**
  – Precimonious and OPR (Corvette/UCB)

• **Resilience Models and Technology**
  – Use of NVRAM (GVR, UChicago); Containment Domains (DEGAS/UTexas)

Funded by X-Stack, Co-Design and NNSA
OpenMP Loop Parallelism is the Wrong Level

- OpenMP is popular for its convenient loop parallelism

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

- Loop level parallelism is too coarse and too fine:
  - **Too coarse**: Implicit synchronization between loops adds overhead
  - **Too fine**: Need larger chunks of serial work across loops (fusion) to reduce synchronization costs and improve locality
Add modest performance variability (interior/border, physics, caches, DVS,...)

- **Dynamically scheduled DAGs**
  - Tolerates performance variability
  - Adds runtime overhead, e.g., synchronization rather than serial execution

- **SPMD**
  - Low overhead, uses serial execution rather than explicit synchronization
  - Too rigid under load imbalance; programmer does “bulkification”

- **Statically scheduled DAGs**
  - Low runtime overhead
  - Offline information may be insufficient and hard to analyze

*Note: This can be done in OpenMP*
Performance Portability
Where is Performance Portability?

• Titan, Mira and Edison represent 3 distinct architectures in DOE
  – 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, but code still requires a big ifdef

Best case #2: Architectures “converge” by 2023, perhaps with co-design help
Don’t Fear the Compiler

• When do we need a language and a compiler?
  – **Productivity**: higher level syntax
    • We need a language
  – **Correctness**: static analysis to eliminate some errors
    • We need a compiler (front-end)
  – **Optimization**: transformations
    • We need a compiler (back-end)

• Hard problems for compilers
  • Analyzing the code to determine legal transformations
    → Make this easier with a DSL
  • Selecting the best (or close) optimized version
    → Make this easier with autotuning

• A compiler is just a software translator
Approach #1: Full-Fledged Compiler and General Purpose Language (with annotation)

- **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

Results on Geometric Multigrid (miniGMG Smoother)

Protonu Basu, S. Williams, M. Hall
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

Jonathan Ragan-Kelly, Saman Amarsinghe, et al at MIT
Approach #3: DSLs with full-fledged compiler analysis

- 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></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

D. Quinlan et al, LLNL
Approach #4: DSL with DS-Compiler

- **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
    - 1400 lines of DSL-specific code; 1 undergrad over <2 months
  - Leverage Python introspection and symbolic / integer libraries

Graph showing Speedup of Kernel Fusion For Stencils with size of input and speedup.

<table>
<thead>
<tr>
<th>Size of Input</th>
<th>Speedup 2 Fused Kernels</th>
<th>Speedup 3 Fused Kernels</th>
<th>Speedup 4 Fused Kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>512</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1024</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2048</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4096</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Time (single core) for HPGMG:
- Python: 6s
- SEJITS: 4s
- HPGMG: 2 months effort, 1400 lines of domain-specific code generation

Armando Fox, M. Driscoll, N. Zhang, C. Markley, S. Williams, K. Yelick
Locality Control
subroutine vec_mult(p, v1, v2, N)
    real :: p(N), v1(N), v2(N)
    integer :: i
    call init(v1, v2, N)

!$omp target data map(to: v1, v2) map(from: p)
!$omp target
!$omp parallel do
    do i=1,N
        p(i) = v1(i) * v2(i)
    end do
!$omp end target
!$omp end target data
    call output(p, N)
end subroutine

And you have to do this for every loop!

Based on slide from J. Shalf
Data layouts can be used to improve locality (and find parallelism), e.g., CAF2, UPC++, Chapel, TiDA, Raja/Kokkos

- OpenMP allows a user to specify any of these layouts
- However, the code is different for GPUs vs CPUs.
- Several approaches pursued here as well

a) Logical Tiles (CPU)  b) Separated Tiles (GPU)  c) Regional Tiles (NUMA)

Separation of tiles with halos
Rethinking Communication
Lowering Overhead for Smaller Messages

Send/Receive

two-sided message
message id  data payload

one-sided put message
address  data payload

network interface
cores
memory

The + in MPI+X

MPI+X today:
• Communicate on one lightweight core
• Reverse offload to heavyweight core
Want to allow all cores to communicate (but keep the protocol simple!)

Lightweight communication is more important with lightweight cores

Khaled Ibrahim, P. Hargrove, C. Iancu, and the UPC group
• 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 (2\textsuperscript{nd} 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?
Avoiding Synchronization
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

- Bulk Synchronous
- Less Synchronous

LAPACK: removing barriers ~2x faster (PLASMA)

Accelerator Offload’

```
!$Acc data copy in(cix, ci1, ci2, ci3, ci4, ci5, ci6, ci7, ci8, ci9, ci10, ci11, ci12, ci13, ci14, &
   cix, ci12, ci13, ci14, r, b, uxyz, cell, rho, grad, index_max, index, &
   ciy, ci12, wet, mp, streaming_sbuf1, &
   !$Acc streaming_sbuf1, streaming_sbuf2, streaming_sbuf3, streaming_sbuf4, streaming_sbuf5, &
   !$Acc streaming_sbuf6, streaming_sbuf7, streaming_sbuf8, streaming_sbuf9, streaming_sbuf10, &
   !$Acc streaming_sbuf11, streaming_sbuf12, streaming_sbuf13, streaming_sbuf14, 
   &
   !$Acc streaming_rbuf1, streaming_rbuf2, streaming_rbuf3, streaming_rbuf4, streaming_rbuf5, &
   !$Acc streaming_rbuf6, streaming_rbuf7, streaming_rbuf8, streaming_rbuf9, streaming_rbuf10, 
   &
   !$Acc streaming_rbuf11, streaming_rbuf12, streaming_rbuf13, streaming_rbuf14, &
   !$Acc streaming_rbuf15, streaming_rbuf16, streaming_rbuf17, streaming_rbuf18, &
   !$Acc send_e, send_w, send_n, send_s, recv_e, recv_w, recv_n, recv_s)
```

Analysis

<table>
<thead>
<tr>
<th>Libraries</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)

Communication

Analysis

<table>
<thead>
<tr>
<th>Libraries</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>

The transfer between host and GPU can be slow and cumbersome, and may (if not careful) get synchronized
Latency Hiding

Many causes of idle time to avoid/hide:

- Application load imbalance
- Communication latency
- Synchronizing shared access (especially remote)
- Dependencies (especially remote)

Asynchronous Runtimes

- Single latency-hiding mechanism, threads: run “unrelated” work while waiting
- Dynamic load balancing

UPC++

- Non-blocking operations for communication hiding
- Static/dynamic load balancing for imbalance
- Threads for dependencies
HPX Asynchronous Runtime Performs on Manycore

LibGeoDecomp - Weak Scaling - Distributed (Host Cores)

Cores 0 3K 6K 9K 13K 16K
GFLOPS 0 100000 200000 300000 400000 500000 600000 700000

HPX MPI Theoretical Peak

Matrix Transpose (12kx12k doubles) on the Xeon Phi

Babbage

Higher is Better

Credit: Harmut Kaiser, LSU and HPX team
Legion Programming Model & Runtime

• Dynamic task-based
  – Data-centric – tasks specify what data they access and how they use them (read-only, read-write, exclusive, etc.)
  – Separates task implementation from hardware mapping decisions
  – Latency tolerant

• Port of S3D complete
  – Currently programmed at the runtime layer (Realm)

• Declarative specification of task graph in Legion
  – Serial program
  – Read/Write effects on regions of data structures
  – Determine maximum parallelism

Legion team from Stanford, ExaCT Co-Design Center
Application Drivers
Data Analysis: 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

All metagenomes

Productivity: Enabling a New Class of Applications?

Mixed Observation + Simulation: Data Fusion

• Seismic modeling for energy applications "fuses" observational data into simulation
• With UPC++, can solve larger problems

Scott French, Y. Zheng, B. Romanowicz, K. Yelick

Distributed Matrix Assembly
• Remote asyncs with user-controlled resource management
• Team idea to divide threads into injectors / updaters
• 6x faster than MPI 3.0 on 1K nodes

→ Improving UPC++ team support
Load Imbalance and Distributed Variable Blocked Arrays

- Hartree Fock calculation for Chemistry
  - Inherent load imbalance
  - Locality still critical
- UPC++ Solution
  - Hybrid static + dynamic (nearest neighbor work stealing) with fast atomic operations
  - New distributed array abstraction enabled productivity and performance
- Impact
  - *20% faster than the best existing solution (GTFock with Global Arrays)*

<table>
<thead>
<tr>
<th>Distributed Array</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>User can easily define a rectangular domain in the global index space</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Strong Scaling of UPC++ HF Compared to GTFock with Global Arrays on NERSC Edison

Adaptive Meshing with Irregular Ghost Updates

• Adaptive Mesh Refinement (AMR) in Boxlib
  • Communication and metadata costs make flat MPI impractical
• UPC++: Hybrid algorithm; One model
  • PGAS within a node for sharing data in memory while enabling data locality
  • Light-weight one-sided communication
  • Interoperability with MPI and OpenMP
• Impact
  • *Improved BoxLib communication performance by 15% compared to MPI and MPI+OpenMP*
  • *Expect higher benefits with increased core count per node*
  • *All applications built on BoxLib Benefit!*

Weiquin Zhang, Y. Zheng

BoxLib software: https://github.com/BoxLib-Codes/BoxLib
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