Programming Models for Petascale

Kathy Yelick
U.C. Berkeley and
Lawrence Berkeley National Laboratory
Moore’s Law is Alive and Well

2X transistors/Chip Every 1.5 years
Called “Moore’s Law”

Microprocessors have become smaller, denser, and more powerful.

Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

Slide source: Jack Dongarra
Clock Scaling Hits Power Density Wall

Scaling clock speed (business as usual) will not work

Source: Patrick Gelsinger, Intel®
Concurrency for Low Power

- Highly concurrent systems are more power efficient
  - Dynamic power is proportional to $V^2 f C$
  - Increasing frequency ($f$) also increases supply voltage ($V$): more than linear effect
  - Increasing cores increases capacitance ($C$) but has only a linear effect

- Hidden concurrency burns power
  - Speculation, dynamic dependence checking, etc.

- Push parallelism discovery to software to save power
  - Compilers, library writers and applications programmers

- Challenge: *Can you double the concurrency in your algorithms every 2 years?*
Revolution is Happening Now

- Chip density is continuing increase ~2x every 2 years
  - Clock speed is not
  - Number of processor cores may double instead
- There is little or no hidden parallelism (ILP) to be found
- Parallelism must be exposed to and managed by software

Source: Intel, Microsoft (Sutter) and Stanford (Olukotun, Hammond)
Petaflop with ~1M Cores by 2015?

Data from top500.org

1 PFlop system in 2008?

6-8 years

100 Pflop/s
10 Pflop/s
1 Pflop/s
100 Tflop/s
10 Tflops/s
1 Tflop/s
100 Gflop/s
10 Gflops/s
1 Gflop/s
10 MFlop/s

Petaflop with ~1M Cores

100 Pflop/s
10 Pflop/s
1 Pflop/s
100 Tflop/s
10 Tflops/s
1 Tflop/s
100 Gflop/s
10 Gflop/s
1 Gflop/s
10 MFlop/s

On your desk in 2025?

1 PFlop system in 2009

6-8 years
8-10 years
Need a Fundamentally New Approach

• Rethink hardware
  – What limits performance
  – How to build efficient hardware

• Rethink software
  – Massive parallelism
  – Eliminate scaling bottlenecks replication, synchronization

• Rethink algorithms
  – Massive parallelism and locality
  – Counting Flops is the wrong measure
Rethink Hardware

(Ways to Waste $50M)
Waste #1: Ignore Power Budget

Power is top concern in hardware design

• Power density within a chip
  – Led to multicore revolution

• Energy consumption
  – Always important in handheld devices
  – Increasingly so in desktops
  – Soon to be significant fraction of budget in large systems

• One knob: increase concurrency
Optimizing for Serial Performance Consumes Power

- **Power5 (Server)**
  - 389 mm²
  - 120 W @ 1900 MHz
- **Intel Core2 sc (Laptop)**
  - 130 mm²
  - 15 W @ 1000 MHz
- **PowerPC450 (BlueGene/P)**
  - 8 mm²
  - 3 W @ 850 MHz
- **Tensilica DP (cell phones)**
  - 0.8 mm²
  - 0.09 W @ 650 MHz

Each core operates at 1/3 to 1/10th efficiency of largest chip, but you can pack 100x more cores onto a chip and consume 1/20 the power!
Power Demands Threaten to Limit the Future Growth of Computational Science

Looking forward to Exascale (1000x Petascale)

• DOE E3 Report
  – Extrapolation of existing design trends
  – Estimate: 130 MW

• DARPA Exascale Study
  – More detailed assessment of component technologies
    • Power-constrained design for 2014 technology
    • 3 TF/chip, new memory technology, optical interconnect
  – Estimate:
    • 40 MW plausible target, not business as usual
    • Billion-way concurrency with cores and latency-hiding

• NRC Study
  – Power and multicore challenges are not just an HPC problem
Waste #2: Buy More Cores than Memory can Feed

- Required bandwidth depends on the algorithm
- Need hardware designed to algorithmic needs
Are We Already Bandwidth-Limited?

- Most real applications run at <10 of peak
- Assumption is they are bandwidth-limited
- NERSC SSP Scientific Applications
  - Run on 1 (SC) and 2 (DC) cores; 2 Core version is 2 copies of the code
  - If bandwidth is the bottleneck, time should go up with 2 cores
  - It does, but closer than expected
Waste #3: Ignore Little’s Law—Latency also Matters

Sparse Matrix-Vector Multiply (2flops / 8 bytes) should be BW limited

<table>
<thead>
<tr>
<th>Name</th>
<th>Intel</th>
<th>AMD</th>
<th>IBM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chips*Cores</td>
<td>2*4 = 8</td>
<td>2*2 = 4</td>
<td>1*8 = 8</td>
</tr>
<tr>
<td>Architecture</td>
<td>4-/3-issue, 2-/1-SSE3, OOO, caches, prefetch</td>
<td>2-VLIW, SIMD, local RAM, DMA</td>
<td></td>
</tr>
<tr>
<td>Clock Rate</td>
<td>2.3 GHz</td>
<td>2.2 GHz</td>
<td>3.2 GHz</td>
</tr>
<tr>
<td>Peak MemBW</td>
<td>21.3 GB/s</td>
<td>21.3</td>
<td>25.6 GB/s</td>
</tr>
<tr>
<td>Peak GFLOPS</td>
<td>74.6 GF</td>
<td>17.6 GF</td>
<td>14.6 (DP Fl. Pt.)</td>
</tr>
<tr>
<td>Naïve SpMV (median of many matrices)</td>
<td>1.0 GF</td>
<td>0.6 GF</td>
<td>--</td>
</tr>
<tr>
<td>Efficiency %</td>
<td>1%</td>
<td>3%</td>
<td>--</td>
</tr>
</tbody>
</table>
Why is the STI Cell So Efficient?
(Latency Hiding with Software Controlled Memory)

- Performance of Standard Cache Hierarchy
  - Cache hierarchies are insufficient to tolerate latencies
  - Hardware prefetch prefers long unit-stride access patterns (optimized for STREAM)
- Cell “explicit DMA”
  - Cell software controlled DMA engines can provide nearly flat bandwidth
  - Performance model is simple and deterministic (much simpler than modeling a complex cache hierarchy),
    \[
    \min\{\text{time for memory ops}, \text{time for core exec}\}
    \]
Waste #4: Unnecessarily Synchronize Communication

- MPI Message Passing is two-sided: each transfer is tied to a synchronization event (message received)
- One-sided communication avoids this
A one-sided put/get message can be handled directly by a network interface with RDMA support
- Avoid interrupting the CPU or storing data from CPU (preposts)

A two-sided messages needs to be matched with a receive to identify memory address to put data
- Offloaded to Network Interface in networks like Quadrics
- Need to download match tables to interface (from host)
Rethinking Programming Models

for

Massive concurrency

Latency hiding

Locality control
Parallel Programming Models

• Parallel software is still an unsolved problem!

• Most parallel programs are written using either:
  – Message passing with a SPMD model
    • for scientific applications; scales easily
  – Shared memory with threads in OpenMP, Threads, or Java
    • non-scientific applications; easier to program

• Partitioned Global Address Space (PGAS) Languages
  – global address space like threads (programmability)
  – SPMD parallelism like MPI (performance)
  – local/global distinction, i.e., layout matters (performance)
Partitioned Global Address Space Languages

- Explicitly-parallel programming model with SPMD parallelism
  - Fixed at program start-up, typically 1 thread per processor
- Global address space model of memory
  - Allows programmer to directly represent distributed data structures
- Address space is logically partitioned
  - Local vs. remote memory (two-level hierarchy)
- Programmer control over performance critical decisions
  - Data layout and communication
- Performance transparency and tunability are goals
  - Initial implementation can use fine-grained shared memory
- Base languages UPC (C), CAF (Fortran), Titanium (Java)
- New HPCS languages have similar data model, but dynamic multithreading
Partitioned Global Address Space

- **Global address space:** any thread/process may directly read/write data allocated by another
- **Partitioned:** data is designated as local or global

By default:
- Object heaps are shared
- Program stacks are private

- **3 Current languages:** UPC, CAF, and Titanium
  - All three use an SPMD execution model
  - Emphasis in this talk on UPC and Titanium (based on Java)

- **3 Emerging languages:** X10, Fortress, and Chapel
• Many common concepts, although specifics differ
  – Consistent with base language, e.g., Titanium is strongly typed
• Both private and shared data
  – int x[10]; and shared int y[10];
• Support for distributed data structures
  – Distributed arrays; local and global pointers/references
• One-sided shared-memory communication
  – Simple assignment statements: x[i] = y[i]; or t = *p;
  – Bulk operations: memcpy in UPC, array ops in Titanium and CAF
• Synchronization
  – Global barriers, locks, memory fences
• Collective Communication, IO libraries, etc.
Private vs. Shared Variables in UPC

- C variables and objects are allocated in the private memory space
- Shared variables are allocated only once, in thread 0’s space
  ```
  shared int ours;
  int mine;
  ```
- Shared arrays are spread across the threads
  ```
  shared int x[2*THREADS] /* cyclic, 1 element each, wrapped */
  ```
- Heap objects may be in either private or shared space
PGAS Language for Multicore

• PGAS languages are a good fit to shared memory machines, including multicore
  – Global address space implemented as reads/writes
  – Current UPC and Titanium implementation uses threads

• Alternative on shared memory is OpenMP or threads
  – PGAS has locality information that is important on multi-socket SMPs and may be important as #cores grows
  – Also may be exploited for processor with explicit local store rather than cache, e.g., Cell processor
PGAS Languages for Clusters

• PGAS languages are a good fit to distributed memory machines and clusters of multicore
  – Global address space uses fast one-sided communication
  – UPC and Titanium use GASNet communication
• Alternative on distributed memory is MPI
  – PGAS partitioned model scaled to 100s of nodes
  – Shared data structure are only in the programmer’s mind in MPI; cannot be programmed as such
One-Sided vs. Two-Sided: Practice

- InfiniBand: GASNet vapi-conduit and OSU MVAPICH 0.9.5
- Half power point ($N^{1/2}$) differs by one order of magnitude
- This is not a criticism of the implementation!

Joint work with Paul Hargrove and Dan Bonachea

NERSC Jacquard machine with Opteron processors
Communication Strategies for 3D FFT

• Three approaches:
  – Chunk:
    • Wait for 2\textsuperscript{nd} dim FFTs to finish
    • Minimize # messages
  – Slab:
    • Wait for chunk of rows destined for 1 proc to finish
    • Overlap with computation
  – Pencil:
    • Send each row as it completes
    • Maximize overlap and
    • Match natural layout

Joint work with Chris Bell, Rajesh Nishtala, Dan Bonachea
NAS FT Variants Performance Summary

- Slab is always best for MPI; small message cost too high
- Pencil is always best for UPC; more overlap

Best MFlop rates for all NAS FT Benchmark versions

- Myrinet 64
- InfiniBand 256
- Elan3 256
- Elan3 512
- Elan4 256
- Elan4 512

M Flops per Thread

 Chunk (NAS FT with FFTW)
 Best MPI (always slabs)
 Best UPC (always pencils)

.5 Tflops

#procs 64 256 256 512 256 512

M Flops per Thread
Making PGAS Real: Applications and Portability
Arrays in a Global Address Space

- **Key features of Titanium arrays**
  - Generality: indices may start/end and any point
  - Domain calculus allow for slicing, subarray, transpose and other operations without data copies
- **Use domain calculus to identify ghosts and iterate:**
  ```
  foreach (p in gridA.shrink(1).domain()) ...
  ```
- **Array copies automatically work on intersection**
  ```
  gridB.copy(gridA.shrink(1));
  ```

Useful in grid computations including AMR

Joint work with Titanium group
Languages Support Helps Productivity

**C++/Fortran/MPI AMR**
- Chombo package from LBNL
- Bulk-synchronous comm:
  - Pack boundary data between procs
  - All optimizations done by programmer

**Titanium AMR**
- Entirely in Titanium
- Finer-grained communication
  - No explicit pack/unpack code
  - Automated in runtime system
- General approach
  - Language allow programmer optimizations
  - Compiler/runtime does some automatically

Work by Tong Wen and Philip Colella; Communication optimizations joint with Jimmy Su
Particle/Mesh Method: Heart Simulation

- Elastic structures in an incompressible fluid.
  - Blood flow, clotting, inner ear, embryo growth, ...
- Complicated parallelization
  - Particle/Mesh method, but “Particles” connected into materials (1D or 2D structures)
  - Communication patterns irregular between particles (structures) and mesh (fluid)

Joint work with Ed Givelberg, Armando Solar-Lezama, Charlie Peskin, Dave McQueen

2D Dirac Delta Function

<table>
<thead>
<tr>
<th>Code Size in Lines</th>
<th>Fortran</th>
<th>Titanium</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8000</td>
<td>4000</td>
</tr>
</tbody>
</table>

Note: Fortran code is not parallel
Immersed Boundary Method Performance

Joint work with Ed Givelberg, Armando Solar-Lezama, Charlie Peskin, Dave McQueen
Dense and Sparse Matrix Factorization

Completed part of U

A(i,j)  A(i,k)

A(j,i)  A(j,k)

Completed part of L

Trailing matrix to be updated

Panel being factored

Blocks 2D block-cyclic distributed

Matrix-matrix multiplication used here. Can be coalesced

Panel factorizations involve communication for pivoting

Joint work with Parry Husbands and Esmond Ng
Matrix Factorization in UPC

- UPC factorization uses a highly multithreaded style
  - Used to mask latency and to mask dependence delays
  - Three levels of threads:
    - UPC threads (data layout, each runs an event scheduling loop)
    - Multithreaded BLAS (boost efficiency)
    - User level (non-preemptive) threads with explicit yield
  - No dynamic load balancing, but lots of remote invocation
  - Layout is fixed (blocked/cyclic) and tuned for block size
- Same framework being used for sparse Cholesky
- Hard problems
  - Block size tuning (tedious) for both locality and granularity
  - Task prioritization (ensure critical path performance)
  - Resource management can deadlock memory allocator if not careful

Joint work with Parry Husbands
• Comparable to MPI HPL (numbers from HPCC database)
• Faster than ScalAPACK due to less synchronization
• Large scaling of UPC code on Itanium/Quadrics (Thunder)
  • 2.2 TFlops on 512p and 4.4 TFlops on 1024p

Joint work with Parry Husbands
Program Synthesis

- Needs extensive tuning knobs for writing basic code
- Don’t do this by hand: tools for tuning

**Spec:** simple implementation
(3 loop 3D stencil)

**Sketch:** optimized skeleton
(5 loops, missing some index/bounds)

**Optimized code** (tiled, prefetched, time skewed)

**Autotuning:** self-tuning code
- Can select from algorithms/data structures changes not producible by compiler transform
Tools for Efficiency: Autotuning

- **Automatic performance tuning**
  - Use machine time in place of human time for tuning
  - Search over possible implementations
  - Use performance models to restrict search space
  - Autotuned libraries for dwarfs (up to 10x speedup)

- Spectral (FFTW, Spiral)
- Dense (PHiPAC, Atlas)
- Sparse (Sparsity, OSKI)
- Stencils/structured grids

- Are these compilers?
  - Don’t transform source
  - There are compilers that use this kind of search
  - But not for the sparse case (transform matrix)

Optimization:
- 1.5x more entries (zeros)
- 1.5x speedup
- Compilers won’t do this!
And don’t think running MPI process per core is good enough for performance.
LBMHD: Structure Grid Application

- Plasma turbulence simulation
- Two distributions:
  - momentum distribution (27 components)
  - magnetic distribution (15 vector components)
- Three macroscopic quantities:
  - Density
  - Momentum (vector)
  - Magnetic Field (vector)
- Must read 73 doubles, and update(write) 79 doubles per point in space
- Requires about 1300 floating point operations per point in space
- Just over 1.0 flops/byte (ideal)
- No temporal locality between points in space within one time step
Pthread Implementation

- Not naïve
  - fully unrolled loops
  - NUMA-aware
  - 1D parallelization

- Always used 8 threads per core on Niagara2
Pthread Implementation

- Not naïve
  - fully unrolled loops
  - NUMA-aware
  - 1D parallelization

- Always used 8 threads per core on Niagara2
Autotuned Performance
(+SIMDization, including non-temporal stores)

- Compilers (gcc & icc) failed at exploiting SIMD.
- Expanded the code generator to use SIMD intrinsics.
- Explicit unrolling/reordering was extremely valuable here.
- Exploited movntpd to minimize memory traffic (only hope if memory bound)
- Significant benefit for significant work

Sun Niagara2 (Huron) vs. IBM Cell Blade

Cell version was not autotuned

Legend:
- +SIMDization
- +SW Prefetching
- +Unrolling
- +Vectorization
- +Padding
- Naïve+NUMA
Autotuned Performance
(Cell/SPE version)

• First attempt at cell implementation.
• VL, unrolling, reordering fixed
• Exploits DMA and double buffering to load vectors
• Straight to SIMD intrinsics.
• Despite the relative performance, Cell’s DP implementation severely impairs performance

Intel Clovertown

AMD Opteron

Sun Niagara2 (Huron)

IBM Cell Blade*

*collision() only
Autotuned Performance
(Cell/SPE version)

**Intel Clovertown**
- 7.5% of peak flops
- 17% of bandwidth

**AMD Opteron**
- 42% of peak flops
- 35% of bandwidth

**Sun Niagara2 (Huron)**
- 59% of peak flops
- 15% of bandwidth

**IBM Cell Blade**
- 57% of peak flops
- 33% of bandwidth

- +SW Prefetching
- +Unrolling
- +Vectorization
- +Padding
- Naïve+NUMA

*collision() only*
Rethink Algorithms
Latency and Bandwidth-Avoiding

• Many iterative algorithms are limited by
  – Communication latency (frequent messages)
  – Memory bandwidth

• New optimal ways to implement Krylov subspace methods on parallel and sequential computers
  – Replace $x \rightarrow Ax$ by $x \rightarrow [Ax,A^2x,\ldots,A^kx]$
  – Change GMRES, CG, Lanczos, … accordingly

• Theory
  – Minimizes network latency costs on parallel machine
  – Minimizes memory bandwidth and latency costs on sequential machine

• Performance models for 2D problem
  – Up to 7x (overlap) or 15x (no overlap) speedups on BG/P

• Measure speedup: 3.2x for out-of-core
Locally Dependent Entries for \([x, Ax, \ldots, A^8x]\), A tridiagonal

Can be computed without communication
k=8 fold reuse of A
Remotely Dependent Entries for $[x, Ax, \ldots, A^8x]$, A tridiagonal

One message to get data needed to compute remotely dependent entries, not $k=8$

Price: redundant work
Fewer Remotely Dependent Entries for \([x, Ax, \ldots, A^8x]\), A tridiagonal

Type (2) Remote Dependencies for k=8

Reduce redundant work by half
Latency Avoiding Parallel Kernel for $[x, Ax, A^2x, \ldots, A^kx]$

- Compute **locally dependent entries** needed by neighbors
- Send data to neighbors, receive from neighbors
- Compute remaining **locally dependent entries**
- Wait for receive
- Compute **remotely dependent entries**
Can use Matrix Power Kernel, but change Algorithms

Work by Demmel and Hoemmen
Predictions and Conclusions

- Parallelism will explode
  - Number of cores will double every 18-24 months
  - Petaflop (million processor) machines will be common in HPC by 2015 (all top 500 machines will have this)

- Performance will become a software problem
  - Parallelism and locality are fundamental; can save power by pushing these to software

- Locality will continue to be important
  - On-chip to off-chip as well as node to node
  - Need to design algorithms for what counts (communication not computation)

- Massive parallelism required (including pipelining and overlap)
Conclusions

• Parallel computing is the future

• Re-think Hardware
  – Hardware to make programming easier
  – Hardware to support good performance tuning

• Re-think Software
  – Software to make the most of hardware
  – Software to ease programming
  • Berkeley UPC compiler: http://upc.lbl.gov
  • Titanium compiler: http://titanium.cs.berkeley.edu

• Re-think Algorithms
  – Design for bottlenecks: latency and bandwidth