Beyond UPC

Kathy Yelick
NERSC, Lawrence Berkeley National Laboratory
EECS Department, UC Berkeley
Berkeley UPC Team

Current UPC Team
- Filip Blagojevic
- Dan Bonachea
- Paul Hargrove (Runtime Lead)
- Steve Hofmeyer
- Costin Iancu (Compiler Lead)
- Seung-Jai Min
- Rajesh Nishtala
- Kathy Yelick (Project Lead)
- Yili Zheng

Former UPC Team Members
- Christian Bell
- Wei-Yu Chen
- Parry Husbands
- Michael Welcome
But Clock Frequency Scaling Replaced by Scaling Cores / Chip

15 Years of exponential growth ~2x year has ended

Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanović
This has Also Impacted HPC System Concurrency

Exponential wave of increasing concurrency for foreseeable future!
1M cores sooner than you think!
Is Exascale a Sure Thing?

Is there a Top500 Law?
Getting to Exascale

A back-of-the-envelope exascale design

• An exascale machine will be built from processors at roughly today’s clock rate
  – 1 GHz → 10^9 (within a factor of 4)
• An exascale machine therefore needs
  – 10^9-way concurrency
• That concurrency likely to be divided as
  – 10^6 chips plus 10^3 way concurrency (arithmetic units) on chip
• The 1K on-chip concurrency to be divided as
  – Independently executing cores with data parallelism
    • 16 cores each with 64-way vectors / GPU-warps
    • 128 cores each with 8-wide SIMD
  – Plus a 1-2 run the OS and other services

I only call them a “core” if they can execute a thread of instructions that are distinct.

There may be another 8-16 hardware threads per core if bandwidth is high enough that latency is still a problem.
Multicore vs. Manycore

- **Multicore: current trajectory**
  - Stay with current fastest core design
  - Replicate every 18 months (2, 4, 8... Etc...)
  - Advantage: Do not alienate serial workload
  - Examples: AMD Barcelona (4 cores), Intel Nehalem (4 cores),...

- **Manycore: converging in this direction**
  - Simplify cores (shorter pipelines, slower clocks, in-order processing)
  - Start at 100s of cores and replicate every 18 months
  - Advantage: easier verification, defect tolerance, highest compute/surface-area, best power efficiency
  - Examples: Cell SPE (8 cores), Nvidia G80 (128 cores), Intel Polaris (80 cores), Cisco/Tensilica Metro (188 cores)

- **Convergence: Ultimately toward Manycore**
  - Manycore: *if we can figure out how to program it!*
  - Hedge: Heterogenous Multicore (still must run PPT)

Slide source: John Shalf
Memory is Not Keeping Pace

Technology trends against a constant or increasing memory per core

- Memory density is doubling every three years; processor logic is every two.
- Storage costs (dollars/Mbyte) are dropping gradually compared to logic costs.

Question: Can you double concurrency without doubling memory?

Source: IBM
What’s Wrong with MPI Everywhere

- We can run 1 MPI process per core (flat model for parallelism)
  - This works now on dual and quad-core machines
- What are the problems?
  - **Latency:** some copying required by semantics
  - **Memory utilization:** partitioning data for separate address space requires some replication
    - How big is your per core subgrid? At 10x10x10, over 1/2 of the points are surface points, probably replicated
  - **Memory bandwidth:** extra state means extra bandwidth
  - **Weak scaling:** success model for the “cluster era;” will not be for the many core era -- not enough memory per core
  - **Heterogeneity:** MPI per CUDA thread-block?
- Easiest approach
  - MPI + X, where X is OpenMP, Pthreads, OpenCL, CUDA,…
PGAS Languages: Why use 2 Programming Models when 1 will do?

- **Global address space**: thread may directly read/write remote data
- **Partitioned**: data is designated as local or global

- Remote put and get: never have to say “receive”
  - Remote function invocation? See HPCS languages
- No less scalable than MPI! (see previous talks)
- Permits sharing, whereas MPI rules it out!
- One model rather than two, but if you insist on two:
  - Can call UPC from MPI and vice versa (tested and used)
What Heterogeneity Means to Me

• Case for heterogeneity
  – Many small cores or wide data parallelism needed for energy efficiency, etc.
  – Need one fat core (at least) for running the OS

• Local store, explicitly managed memory hierarchy
  – More efficient (get only what you need) and simpler to implement in hardware

• Co-Processor interface between CPU and Accelerator
  – Market forces push this: GPUs have been separate chips for specific domains, but they may move on-chip
  – Do we really have use this co-processor idea? Isn’t parallel programming hard enough
But….Optimizing for Multicore: Almost as Hard (if Not Harder)

Intel Xeon (Clovertown)

AMD Opteron (Barcelona)

Sun Niagara2 (Victoria Falls)

Simplest possible problem: stencil computation: nearest neighbor relaxation on 3D Mesh

• For this simple code - all cache-based platforms show poor efficiency and scalability

• Could lead programmer to believe that approaching a resource limit
Fully-Tuned Performance

Intel Xeon (Clovertown)

AMD Opteron (Barcelona)

Sun Niagara2 (Victoria Falls)

Different optimizations have dramatic effects on different architectures

Largest optimization benefit seen for the largest core count

Optimizations include:

- NUMA-Aware
- Padding
- Unroll/Reordering
- Thread/Cache Blocking
- Prefetching
- SIMDization
- Cache Bypass
Stencil Results

Single Precision

Double Precision

Performance

Power Efficiency
PGAS Languages for Manycore

- PGAS memory are a good fit to machines with explicitly managed memory (local store)
  - Global address space implemented as DMA reads/writes
  - New “vertical” partition of memory needed for on/off chip, e.g., `upc_offchip_alloc`
  - Non-blocking features of UPC put/get are useful
- SPMD execution model needs to be adapted to heterogeneity

<table>
<thead>
<tr>
<th>l:</th>
<th>m:</th>
<th>Private on-chip</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 1</td>
<td>x: 5</td>
<td>x: 7</td>
</tr>
<tr>
<td>y:</td>
<td>y:</td>
<td>y: 0</td>
</tr>
</tbody>
</table>

- Shared partitioned on-chip
- Shared off-chip DRAM

Network

Computer Node
- CPU
- GPU
- CPU Memory

Computer Node
- CPU
- GPU
- CPU Memory
Radical (and Unappealing) Proposal

Adding teams to SPMD execution model
• These are needed for collectives in any case
• Uses separate teams for fat cores vs thin core teams

Execution model
• Execute SPMD code on either set
• Execute any code you want on each core
  – Careful: needs to be the same (data parallel execution) to run well
  – Or still use a different model (annotated loops) for SIMD parallelism
Features of Successful Languages

• Portability of applications
  – Multiple compilers, portable compilers, or both (UPC vs CAF to date)
UPC Compiler: Designed for Portability

- UPC Code
  - Compiler-generated code (C, asm)
  - Language Runtime system
  - GASNet Communication System
  - GASNet Core
  - Network Hardware

- Platform-independent
- Network-independent
- Language-independent
- Compiler-independent
Portability of GASNET

Original vision of conduit development progression
• Build GASNet core (Active Messages) with provided “reference implementation” of full API on core
• Incrementally develop native implementations of features (put/get, etc.) of full API

Alternative GASNet progression, use on Cray XT
• Pure MPI: mpi-conduit
  – “Runs everywhere, optimally nowhere”
• Hybrid: replaced put/get calls with Portals RDMA
• Pure Portals: Native Core API on Cray
• Firehose to reduce memory registration overheads

Easier parts of implementation first
Better time-to-solution for acceptable performance
Features of Successful Languages

• Portability of applications
  – Multiple compilers, portable compilers, or both (UPC vs CAF to date)
• Interoperability with other models
  – Calling MPI from UPC and vice versa
  – Necessary for incremental development
- 16 cores on 4 sockets: how many threads & processes?
- 8 cores with 2 hardware threads per core (hyperthreading)
- Processes intermix with MPI; Threads with OpenMP
- Performance tradeoffs unclear: Can we get shared memory with processes?
Performance improvement of ProcSM over Pthreads

- **IS**:
  - 4 threads: 0%
  - 8 threads: 10%
  - 16 threads: 20%
  - 32 threads: 30%

- **SP**:
  - 4 threads: 0%
  - 16 threads: 5%
  - 25 threads: 10%

- **BT**:
  - 4 threads: 0%
  - 16 threads: 10%
  - 25 threads: 20%

- **FT**:
  - 4 threads: 0%
  - 8 threads: 5%
  - 16 threads: 10%
  - 32 threads: 15%
Features of Successful Languages

• **Portability of applications**
  – Multiple compilers, portable compilers, or both

• **Interoperability with other models**
  – Calling MPI from UPC and vice versa
  – Necessary for incremental development

• **Performance comparable to or better than alternatives, including scalability**
  – This should be a selling point, not 2x slower

• **Take advantage of “best” hardware**
  – Best networks, multicore, etc.
Sharing and Communication Models: PGAS vs. MPI

- 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)
- 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)
GASNet vs. MPI Bandwidth on BG/P

- GASNet outperforms MPI on small to medium messages, especially when multiple links are used.
XT4 Performance

- Performance on Franklin, quad-core XT4 @ NERSC
  - NERSC development machine access for testing
  - Testing infrequently used code paths in Portals
- Native conduit outperforms GASNet-over-MPI by 2x
- Latency better than raw MPI
- Bandwidth equal to raw MPI
- Recent Firehose support increased performance by 4% to 8% in bandwidth (included)

[Bonachea, Hargrove, Welcome, Yelick, CUG ‘09]
UPC on BlueGene/P

- Faster dense linear algebra than PBLAS/ScalAPACK
  - Parallel matrix multiplication: 36% faster (256 cores)
  - Parallel Cholesky factorization: 9% faster (256 cores)
- Faster FFTs than MPI
- GASNet collectives up to 4x faster than previous release
- GASNet implemented on DCMF layer
Optimizing Collectives on Multicore

- Many algorithms even for barrier synchronization
- Dissemination based:
  - $O(T \log T)$ “messages”
  - Time: $L*(\log T)$ ($L$ = latency)
- Tree-based
  - $O(T)$ “messages”
  - Time: $2L*(\log T)$
Need for Autotuned Multicore Collectives

- "Traditional pthread barriers" yield poor performance
- Tree algorithms: best of structures, varying signaling [Nishtala+, HotPar’09]
Features of Successful Languages

• Portability of applications
  – Multiple compilers, portable compilers, or both

• Interoperability with other models
  – Calling MPI from UPC and vice versa
  – Necessary for incremental development

• Performance comparable to or better than alternatives, including scalability
  – This should be a selling point, not 2x slower

• Take advantage of “best” hardware
  – Best networks, multicore, etc.

• Easy to use for a broad set of applications
  – Are there applications that do not match UPC well?
Irregular Applications

• UPC originally for “irregular” applications
  – Many recent performance results are on “regular” ones (FFTs, NPBs, etc.); those also do well

• Does it really handle irregular ones? Which?
  – Irregular in data accesses:
    • Irregular in space (sparse matrices, AMR, etc.): global address space helps; needs compiler or language for scatter/gather
    • Irregular in time (hash table lookup, etc.): for reads, UPC handles this well; for write you need atomic operations
  – Irregular computational patterns:
    • High level independent tasks (ocean, atm, land, etc.): need teams
    • Non bulk-synchronous: use event-driven execution
    • Not statically load balanced (even with graph partitioning, etc.): need global task queue
Two Programming Model Questions

• What is the parallel control model?
  - Data parallel (single thread of control)
  - Dynamic threads
  - Single program multiple data (SPMD)

• What is the model for sharing/communication?
  - Store
  - Load
  - Receive
  - Send
  - Shared memory
  - Message passing
  - Implied synchronization for message passing, not shared memory
Complication of Work Sharing in Partitioned Memory

- If tasks are waiting for others to complete, then need to suspect tasks for fairness:
  - This can blow up the memory space
  - CILK and X10 results on “provably optimal space”: execute by functional call / stack semantics until you run out of work

- Run-to-completion:
  - Efficient and simpler to implement
  - But doesn’t always give the desired semantics

- Memory partitioning with work sharing: can run out of memory locally (GPUs and UPC)
Response of UPC to Challenges

- **Small memory per core**
  - Ability to directly access another core’s memory

- **Lack of UMA memory on chip**
  - Partitioned address space

- **Massive concurrency**
  - Good match for independent parallel cores
  - Not for data parallelism

- **Heterogeneity**
  - Need to relax strict SPMD with at least teams

- **Application generality**
  - Add atomics so remote writes work (not just reads)
A Brief History of Languages

• **When vector machines were king**
  – Parallel “languages” were loop annotations (IVDEP)
  – Performance was fragile, but there was good user support

• **When SIMD machines were king**
  – Data parallel languages popular and successful (CMF, *Lisp, C*, …)
  – Quite powerful: can handle irregular data (sparse mat-vec multiply); Irregular computation is less clear (search, sparse factorization)

• **When shared memory machines (SMPs) were king**
  – Shared memory models, e.g., OpenMP, Posix Threads, are popular

• **When clusters took over**
  – Message Passing (MPI) became dominant

• **When clusters of multicore take over…**
  – Will PGAS be the dominant programming model?

What does it take to make a programming language successful?