Unified Parallel C (UPC)

Katherine Yelick
NERSC Director, Lawrence Berkeley National Laboratory
EECS Professor, 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

Current NERSC PModels Team
• John Shalf
• Alice Koniges
• Hongzhang Shan
Moore’s Law is Alive and Well

Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanović
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
Performance Has Also Slowed, Along with Power

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

Sum of the # of cores in top 15 systems (from top500.org)

Exponential wave of increasing concurrency for foreseeable future!
1M cores sooner than you think!
Programming Models Challenges

• Programming models is the most pervasive research problem facing computing today due to:
  – The shift single core to multicore processors
  – Other forms of explicit intra-node parallelism, e.g., SIMD units, GPUs, and other forms or accelerators
• It is a major concern for general-purpose desktop, laptop, and embedded system developers not just HPC
• Past improvements came form a combination of:
  – Hidden concurrency (e.g., ILP)
  – Clock speed scaling
  – Explicit concurrency (more processors + SIMD)
• Future improvements will from explicit concurrency alone
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
- Memory costs (dollars/Mbyte) are dropping gradually compared to logic costs

Question: Can you double concurrency without doubling memory?
Software Issues at Scale

• Need for 1K-way chip concurrency by 2019
  – Exascale system will require billion-way concurrency with O(1K) way per chip
  – Departmental scale (1 PF) systems also need this

• Notion of a “core” is changing
  – Concurrency can come from cores, SIMD, etc.
  – Count 64-bit floating point units rather than cores

• The memory/core ratio will drop significantly
  – Strong scaling will be essential

• Flat “MPI Everywhere” model is not workable
  – Cannot afford to partition/replicate data
  – Need fine-grained concurrency
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

- UPC is one example of a PGAS language
- Remote put and get: never have to say “receive”
  - Remote function invocation? See HPCS languages
- As scalable as MPI (not cache-coherent shared memory)!
- Permits sharing, whereas MPI rules it out!
What is UPC?

• Threads working independently in a SPMD fashion
  – Number of threads is `THREADS`, executing thread is `MYTHREAD`
  – `upc_barrier` is a global synchronization: all wait

• Any legal C program is also a legal UPC program
  
  ```c
  printf("Thread %d of %d: hello\n", MYTHREAD, THREADS);
  ```

• Shared and private variables

  ```c
  shared int ours; /* lives on P0 */
  int mine; /* a copy per thread */
  shared int * ptr = (shared int *) upc_alloc(sizeof(int));
  ```

• Shared arrays spread over (distributed) memory

  ```c
  shared double a1[100]; /* cyclic */
  shared double [4] a2 [100]; /* 4-elt blocks */
  ```

• Collective operations (broadcast, reduce, etc.)
Why use UPC between nodes?
UPC Compiler: Designed for Portability

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

Platform-independent
Network-independent
Language-independent
Compiler-independent
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]
GASNet: Portability and High-Performance

GASNet better for latency across machines
GASNet excels at mid-range sizes: important for overlap
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
Why Use UPC on Multicore?
What’s wrong with Threads / OpenMP?

- Shared memory programming often considered convenient, but not scalable
- Threads and OpenMP:
  - OpenMp encourages serial thinking between loops
  - No control over locality
  - Dynamic threading model can overwhelm runtime
  - Don’t interoperate with flat MPI
- UPC
  - SPMD model allows simple, efficient runtime (1 / core)
  - Allows locality controlled (incremental development)
  - Always-parallel thinking
Performance of UPC on SMPs

- Result inconclusive so far
- Biggest factors: quality of benchmark writer
- Second biggest: quality of serial code generation
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: $2L*(\log T)$
“Traditional pthread barriers” yield poor performance

Tree algorithms: best of structures, varying signaling [Nishtala+, HotPar’09]
Multicore Utilization

• Existing systems are asymmetric in terms of architecture (accelerators), performance (Nehalem) and software (OS service cores, cloud virtualization)
• Static assignment of SPMD threads is not always optimal
• Mechanisms for resource allocation and policies for control
  – Speed balancing, user level load balancer for scientific apps
  – Distributed, scalable soft state
  – Study: NAS-EP on 16 cores
  – “SPEED” Outperforms (Linux, BSD) and fair processor scheduling algorithms (DWRR)
Beyond the SPMD Model: Dynamic Threads

• UPC uses a static threads (SPMD) programming model
  – No dynamic load balancing built-in
  – Berkeley UPC model extends basic memory semantics (remote read/write) with active messages
  – AM have limited functionality (no messages except acks) to avoid deadlock in the network

• A more dynamic runtime would have many uses
  – Application load imbalance, OS noise, fault tolerance

• Two extremes are well-studied
  – Dynamic load balancing (e.g., random stealing) without locality
  – Static parallelism (with threads = processors) with locality

• Can we combine both in a general-purpose way?

Joint work with Parry Husbands
Parallel LU Factorization: Case Study

- Panel factorizations involve communication for pivoting.
- Completed part of $L$: $A(i,j), A(i,k)$, Panel being factored.
- Completed part of $U$: $A(i,j), A(i,k)$.
- Trailing matrix to be updated: $A(j,i), A(j,k)$.
- Blocks 2D block-cyclic distributed.
- Matrix-matrix multiplication used here. Can be coalesced.
Event Driven Execution of Dense LU

- Ordering needs to be imposed on the schedule
- Critical operation: Panel Factorization
  - need to satisfy its dependencies first
  - perform trailing matrix updates with low block numbers first
  - “memory constrained” lookahead
- General issue: dynamic scheduling in partitioned memory
  - Can deadlock memory allocator!

some edges omitted
DAG Scheduling of LU in UPC + Multithreading

- UPC uses a static threads (SPMD) programming model
  - Multithreading 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
[Husbands & Yelick, SC07]
UPC for Accelerators?
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
Data Tiling for GPU Computing

```c
shared [2][2] float A[4][4], B[4][4], C[4][4];
accelmem Ag[2][2], Bg[2][2], Cg[2][2])
upc_memput (&Ag, &A[i][k], 4*sizeof(float));
```
Execution Models

- Synchronous model
- Virtual GPU model
- Hybrid model
Lessons Learned

• One-sided communication is faster than 2-sided
  – Microbenchmarks and FFTs

• Global address space can ease programming
  – E.g., for shared random access structures, hash tables

• PGAS languages are promising for multicore
  – Gives locality control for scalability
  – Allow shared state, which avoids replication

• PGAS language ideas may adapt to accelerators
  – Partitioned memory model is a natural fit
  – SPMD execution model needs to be extended

• Current PGAS languages are not the final answer