UPC: A Portable High Performance Dialect of C

Kathy Yelick

Christian Bell, Dan Bonachea, Wei Chen, Jason Duell, Paul Hargrove, Parry Husbands, Costin Iancu, Wei Tu, Mike Welcome
Parallelism on the Rise

- 1.8x annual performance increase
  - 1.4x from improved technology and on-chip parallelism
  - 1.3x in processor count for larger machines
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 differ: UPC (C), CAF (Fortran), Titanium (Java)
**UPC Design Philosophy**

- Unified Parallel C (UPC) is:
  - An explicit parallel extension of ISO C
  - A partitioned global address space language
  - Sometimes called a GAS language
- Similar to the C language philosophy
  - Concise and familiar syntax
  - Orthogonal extensions of semantics
  - Assume programmers are clever and careful
    - Given them control; possibly close to hardware
    - Even though they may get intro trouble
- Based on ideas in Split-C, AC, and PCP
A Quick UPC Tutorial
Virtual Machine Model

• Global address space abstraction
  - Shared memory is partitioned over threads
  - Shared vs. private memory partition within each thread
  - Remote memory may stay remote: no automatic caching implied
  - One-sided communication through reads/writes of shared variables

• Build data structures using
  - Distributed arrays
  - Two kinds of pointers: Local vs. global pointers (“pointers to shared”)
UPC Execution Model

- Threads work independently in a SPMD fashion
  - Number of threads given by `THREADS` set as compile time or runtime flag
    - `MYTHREAD` specifies thread index (0..`THREADS-1`)
    - `upc_barrier` is a global synchronization: all wait
- Any legal C program is also a legal UPC program

```c
#include <upc.h> /* needed for UPC extensions */
#include <stdio.h>
main() {
    printf("Thread %d of %d: hello UPC world\n", MYTHREAD, THREADS);
}
```
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 */
  ```

- Shared variables may not occur in a function definition unless static.
**Work Sharing with upc_forall()**

```c
shared int v1[N], v2[N], sum[N];
void main() {
    int i;
    upc_forall (i=0; i<N; i++; &v1[i])
        sum[i] = v1[i] + v2[i];
}
```

- This owner computes idiom is common, so UPC has
  
  ```c
  upc_forall (init; test; loop; affinity) statement;
  ```

- Programmer indicates the iterations are independent
  - Undefined if there are dependencies across threads
  - Affinity expression indicates which iterations to run
    - **Integer**: `affinity%THREADS is MYTHREAD`
    - **Pointer**: `upc_threadof(affinity) is MYTHREAD`

i would also work
Memory Consistency in UPC

- Shared accesses are strict or relaxed, designed by:
  - A pragma affects all otherwise unqualified accesses
    - #pragma upc relaxed
    - #pragma upc strict
    - Usually done by including standard .h files with these
  - A type qualifier in a declaration affects all accesses
    - int strict shared flag;
  - A strict or relaxed cast can be used to override the current pragma or declared qualifier.

- Informal semantics
  - Relaxed accesses must obey dependencies, but non-dependent access may appear reordered by other threads
  - Strict accesses appear in order: sequentially consistent
Other Features of UPC

• Synchronization constructs
  - Global barriers
    - Variant with labels to document matching of barriers
    - Split-phase variant (**upc_notify** and **upc_wait**)
  - Locks
    - **upc_lock**, **upc_lock_attempt**, **upc_unlock**

• Collective communication library
  - Allows for asynchronous entry/exit
    ```c
    shared [] int A[10];
    shared [10] int B[10*THREADS];
    // Initialize A.
    upc_all_broadcast(B, A, sizeof(int)*NELEMS,
                       UPC_IN_MYSYNC | UPC_OUT_ALLSYNC);
    ```

• Parallel I/O library
The Berkeley UPC Compiler
Goals of the Berkeley UPC Project

• Make UPC Ubiquitous on
  - Parallel machines
  - Workstations and PCs for development
  - A portable compiler: for future machines too

• Components of research agenda:
  1. Runtime work for Partitioned Global Address Space (PGAS) languages in general
  2. Compiler optimizations for parallel languages
  3. Application demonstrations of UPC
Berkeley UPC Compiler

- Compiler based on Open64
  - Multiple front-ends, including gcc
  - Intermediate form called WHIRL
- Current focus on C backend
  - IA64 possible in future
- UPC Runtime
  - Pointer representation
  - Shared/distribute memory
- Communication in GASNet
  - Portable
  - Language-independent
Optimizations

• In Berkeley UPC compiler
  - Pointer representation
  - Generating optimizable single processor code
  - Message coalescing (aka vectorization)

• Opportunities
  - forall loop optimizations (unnecessary iterations)
  - Irregular data set communication (Titanium)
  - Sharing inference
  - Automatic relaxation analysis and optimizations
UPC has three difference kinds of pointers:
- Block-cyclic, cyclic, and indefinite (always local)
A pointer needs a “phase” to keep track of where it is in a block
- Source of overhead for updating and de-referencing
- Consumes space in the pointer

<table>
<thead>
<tr>
<th>Address</th>
<th>Thread</th>
<th>Phase</th>
</tr>
</thead>
</table>

Our runtime has special cases for:
- Phaseless (cyclic and indefinite) – skip phase update
- Indefinite – skip thread id update
- Some machine-specific special cases for some memory layouts

Pointer size/representation easily reconfigured
- 64 bits on small machines, 128 on large, word or struct
Performance of Pointers to Shared

- Phaseless pointers are an important optimization
  - Indefinite pointers almost as fast as regular C pointers
  - General blocked cyclic pointer 7x slower for addition
- Competitive with HP compiler, which generates native code
  - Both compiler have improved since these were measured
Generating Optimizable (Vectorizable) Code

- Translator generated C code can be as efficient as original C code
- Source-to-source translation a good strategy for portable PGAS language implementations
NAS CG: OpenMP style vs. MPI style

- GAS language outperforms MPI+Fortran (flat is good!)
- Fine-grained (OpenMP style) version still slower
  - shared memory programming style leads to more overhead (redundant boundary computation)
- GAS languages can support both programming styles
Message Coalescing

- Implemented in a number of parallel Fortran compilers (e.g., HPF)
- Idea: replace individual puts/gets with bulk calls
- Targets bulk calls and index/strided calls in UPC runtime (new)
- Goal: ease programming by speeding up shared memory style

```
shared [0] int * r;
...
for (i = L; i < U; i++)
  exp1 = exp2 + r[i];
```

```
int lr[U-L];
...
upcr_memget(lr, &r[L], U-L);
for (i = L; i < U; i++)
  exp1 = exp2 + lr[i-L];
```

Unoptimized loop  Optimized Loop
Message Coalescing vs. Fine-grained

- One thread per node
- Vector is 100K elements, number of rows is 100*threads
- Message coalesced code more than 100X faster
- Fine-grained code also does not scale well
  - Network overhead
• Message coalescing and bulk style code have comparable performance
  - For indefinite array the generated code is identical
  - For cyclic array, coalescing is faster than manual bulk code on elan
    - memgets to each thread are overlapped
    - Points to need for language extension
Automatic Relaxation

- Goal: simplify programming by giving programmers the illusion that the compiler and hardware are not reordering.
- When compiling sequential programs:
  \[
  x = \text{expr1}; \\
  y = \text{expr2};
  \]
  \[
  y = \text{expr2}; \\
  x = \text{expr1};
  \]
  Valid if y not in expr1 and x not in expr2 (roughly).
- When compiling parallel code, not sufficient test.

Initially flag = data = 0

Proc A          Proc B
data = 1;       while (flag!=1);
flag = 1;       ... = ...data...;
Cycle Detection: Dependence Analog

- Processors define a “program order” on accesses from the same thread
  \[ P \text{ is the union of these total orders} \]
- Memory system define an “access order” on accesses to the same variable
  \[ A \text{ is access order (read/write & write/write pairs)} \]

- A violation of sequential consistency is cycle in \( P \cup A \).
- Intuition: time cannot flow backwards.
Cycle Detection

• Generalizes to arbitrary numbers of variables and processors

• Cycles may be arbitrarily long, but it is sufficient to consider only cycles with 1 or 2 consecutive stops per processor
Static Analysis for Cycle Detection

- Approximate P by the control flow graph
- Approximate A by undirected “dependence” edges
- Let the “delay set” D be all edges from P that are part of a minimal cycle

write z          read x
down            down
write y          read x
  up            up
read y          write z

- The execution order of D edge must be preserved; other P edges may be reordered (modulo usual rules about serial code)
- Conclusions:
  - Cycle detection is possible for small language
  - Synchronization analysis is critical
  - Open: is pointer/array analysis accurate enough for this to be practical?
GASNet: Communication Layer for PGAS Languages
GASNet Design Overview - Goals

- Language-independence: support multiple PGAS languages/compilers
  - UPC, Titanium, Co-array Fortran, possibly others.
  - Hide UPC- or compiler-specific details such as pointer-to-shared representation
- Hardware-independence: variety of parallel arch., OS's & networks
  - SMP's, clusters of uniprocessors or SMPs
  - Current networks:
    - Native network conduits: Myrinet GM, Quadrics Elan, Infiniband VAPI, IBM LAPI
    - Portable network conduits: MPI 1.1, Ethernet UDP
    - Under development: Cray X-1, SGI/Cray Shmem, Dolphin SCI
  - Current platforms:
    - CPU: x86, Itanium, Opteron, Alpha, Power3/4, SPARC, PA-RISC, MIPS
    - OS: Linux, Solaris, AIX, Tru64, Unicos, FreeBSD, IRIX, HPUX, Cygwin, MacOS
- Ease of implementation on new hardware
  - Allow quick implementations
  - Allow implementations to leverage performance characteristics of hardware
  - Allow flexibility in message servicing paradigm (polling, interrupts, hybrids, etc)
- Want both portability & performance
GASNet Design Overview - System Architecture

- 2-Level architecture to ease implementation:
- Core API
  - Most basic required primitives, as narrow and general as possible
  - Implemented directly on each network
  - Based heavily on active messages paradigm
- Extended API
  - Wider interface that includes more complicated operations
  - We provide a reference implementation of the extended API in terms of the core API
  - Implementors can choose to directly implement any subset for performance - leverage hardware support for higher-level operations
  - Currently includes:
    - blocking and non-blocking puts/gets (all contiguous), flexible synchronization mechanisms, barriers
    - Just recently added non-contiguous extensions (coming up later)
GASNet Performance Summary

GASNet Put/Get Roundtrip Latency (min over msg sz)

- put_nb
- get_nb

Roundtrip Latency (microseconds)

- mpi
- elan
- mpi
- gm
- mpi
- gm
- mpi
- lapi
- lapi-poll
- mpi
- gm
- mpi
- vapi

 Platforms:
- quadrics
- opus
- myrinet alvarez (x86)
- myrinet citrus (IA64)
- Colony/GX seaborg (PowerPC)
- myrinet pcp (x86 PCI-X)
- infiniband pcp (x86 PCI-X)
GASNet Performance Summary

GASNet Put/Get Bulk Flood Bandwidth (max over msg sz)

- put_nb_bulk
- get_nb_bulk

Bandwidth (MB/sec)

 mpi elan mpi elan mpi gm mpi gm
- quadrics lemieux (Alpha)
- quadrics opus (IA64)
- myrinet alvarez (x86)
- myrinet citris (IA64)
- mpi lapi lapi-poll
- Colony/GX seaborg (PowerPC)
- myrinet pcp (x86 PCI-X)
- mpi vapi
- infiniband pcp (x86 PCI-X)
GASNet vs. MPI on Infiniband

Roundtrip Latency of GASNet vapi-conduit and MVAPICH 0.9.1 MPI

OSU MVAPICH widely regarded as the "best" MPI implementation on Infiniband
MVAPICH code based on the FTG project MVICH (MPI over VIA)
GASNet wins because fully one-sided, no tag matching or two-sided sync.overheads
MPI semantics provide two-sided synchronization, whether you want it or not
GASNet vs. MPI on Infiniband

Bandwidth of GASNet vapi-conduit and MVAPICH 0.9.1 MPI

GASNet significantly outperforms MPI at mid-range sizes - the cost of MPI tag matching.
Yellow line shows the cost of naïve bounce-buffer pipelining when local side not prepinned -
memory registration is an important issue.
Applications in PGAS Languages
PGAS Languages Scale

- Use of the memory model (relaxed/strict) for synchronization
- Medium sized messages done through array copies

NAS MG in UPC (Berkeley Version)

MFlops

# Procs
Performance Results
Berkeley UPC FT vs MPI Fortran FT

NAS FT 2.3 Class A - NERSC Alvarez Cluster

80 Dual PIII-866MHz Nodes running Berkeley UPC
(gm-conduit /Myrinet 2K, 33Mhz-64Bit bus)
Challenging Applications

• Focus on the problems that are hard for MPI
  - Naturally fine-grained
  - Patterns of sharing/communication unknown until runtime

• Two examples
  - Adaptive Mesh Refinement (AMR)
    - Poisson problem in Titanium (low flops to memory/comm)
    - Hyperbolic problems in UPC (higher ratio, not adaptive so far)
    - Task parallel view (first)
  - Immersed boundary method simulation
    - Used for simulating the heart, cochlea, bacteria, insect flight,..
    - Titanium version is a general framework
      - Specializations for the heart and cochlea
    - Particle methods with two structures: regular fluid mesh + list of materials
Ghost Region Exchange in AMR

- Ghost regions exist even in the serial code
  - Algorithm decomposed as operations on grid patches
  - Nearest neighbors (7, 9, 27-point stencils, etc.)
- Adaptive mesh organized by levels
  - Nasty meta-data problem to find neighbors
  - May exists only at a different level
Distributed Data Structures for AMR

- This shows just one level of the grid hierarchy
- Not an distributed array in any of the languages that support them
- Note: Titanium uses this structure even for regular arrays
Programmability Comparison in Titanium

- Ghost region exchange in AMR
  - 37 lines of Titanium
  - 327 lines of C++/MPI, of which 318 are MPI-related

- Speed (single processor, full solve)
  - The same algorithm, Poisson AMR on same mesh
  - C++/Fortran Chombo: 366 seconds
  - Titanium by Chombo programmer: 208 seconds
  - Titanium after expert optimizations: 155 seconds
    - The biggest optimization was avoiding copies of single element arrays, which required domain/performance expert to find/fix
  - Titanium is faster on this platform!
Heart Simulation in Titanium

- Programming experience
  - Code existed in Fortran for vector machines
  - Complete rewrite in Titanium
    - Except fast FFTs
    - 3 GSR years + 1.5 postdoc year
  - # of numerical errors found along the way
    - About 1 every 2 weeks
    - Mostly due to missing code
  - # of race conditions: 1
Scalability

- $512^3$ in < 1 second per timestep not possible
- 10x increase in bisection bandwidth would fix this
Those Finicky Users

- How do we get people to use new languages?
  - Needs to be incremental
  - Start at the bottom, not at the top of the software stack
- Need to demonstrate advantages
  - Performance is the easiest: comes from ability to use great hardware
  - Productivity is harder:
    - Managers may be convinced by data
    - Programmers will vote by experience
    - Wait for programmer turnover
- Key: language must run well everywhere
  - As well as the hardware allows
PGAS Languages are Not the End of the Story

- Flat parallelism model
  - Machines are not flat: vectors, streams, SIMD, VLIW, FPGAs, PIMs, SMP nodes,…
- No support for dynamic load balancing
  - Virtualize memory structure → moving load is easier
  - No virtualization of processor space → taskqueue library
- No fault tolerance
  - SPMD model is not a good fit if nodes fail frequently
- Little understanding of scientific problems
  - CAF and Titanium have multiD arrays
  - A matrix and grid are both arrays, but they’re different
  - Next level example: Immersed boundary method language
To Virtualize or Not to Virtualize

• Why to virtualize
  - Portability
  - Fault tolerance
  - Load imbalance

  If we design for these problems, need to beat MPI and HPC will always be a niche

• Why to not virtualize
  - Deep memory hierarchies
  - Expensive system overhead
  - Some problems match the hardware; don’t want to pay overhead

• PGAS languages virtualize memory structure but not processor number
• Can we provide virtualized machine, but still allow for control in mapping (separate code)?