The Berkeley UPC Project

Kathy Yelick

Christian Bell, Dan Bonachea,
Wei Chen, Jason Duell,
Paul Hargrove, Parry Husbands,
Costin Iancu, Wei Tu, Mike Welcome
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)
### 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 into 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 int y[2] [2*THREADS] /* blocked, with block size 2 */
  ```
- Shared variables may not occur in a function definition unless static.
Work Sharing with upc_forall()

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
  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 (as in Titanium)
  - Sharing inference
  - Automatic relaxation analysis and optimizations
Pointer-to-Shared Representation

- UPC has three different 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
• 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 has more communication events
- GAS languages can support both programming styles
Communication Optimizations

- Automatic optimizations of communication are key to
  - Usability of UPC: fine-grained programs with coarse-grained performance
  - Performance portability: make application performance less sensitive to the architecture

- Types of optimizations
  - Use of non-blocking communication (future work)
  - Communication code motion
  - Communication coalescing
  - Software caching (part of runtime)
  - Automatic relaxation: towards elimination of “relaxed”
    - Fundamental research problem for PGAS languages
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];

Unoptimized loop
```

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

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 vs. Bulk

- Message coalescing and bulk (manual) style code have comparable performance
  - For indefinite arrays 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
- Status: coalescing prototyped on 1D arrays
  - Needs full multi-D implementation and release
Automatic Relaxation

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

Initially flag = data = 0

<table>
<thead>
<tr>
<th>Proc A</th>
<th>Proc B</th>
</tr>
</thead>
<tbody>
<tr>
<td>data = 1;</td>
<td>while (flag!=1);</td>
</tr>
<tr>
<td>flag = 1;</td>
<td>... = ...data...;</td>
</tr>
</tbody>
</table>
Cycle Detection: Dependence Analog

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

\[
\begin{array}{ccc}
\text{write data} & \quad \text{read flag} & \quad \text{write flag} & \quad \text{read data} \\
\downarrow & \quad \times & \quad \downarrow & \quad \downarrow \\
\end{array}
\]

- 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

write x               write y               read y

read y                                      write x

- 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

![Diagram of cycle detection]

- The execution order of $D$ edge must be preserved; other $P$ edges may be reordered (modulo usual rules about serial code)
Cycle Detection Status

• For programs that do not require pointer or array analysis [Krishnamurthy & Yelick]:
  - Cycle detection is possible for small language
  - Synchronization analysis is critical: need to line up barriers to reduce analysis cost, improve accuracy
• Recent work [Chen, Krishnamurthy & Yelick 2003]
  - Improved running time $O(n^3)$ to $O(n^2)$
  - Array analysis extensions (3 types)
• Open: can this be done on complicated programs?
  - Implementation work and experiments needed
  - Pointer analysis will be needed: Titanium/Parry style distributed arrays
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
    – Recently added non-contiguous extensions
GASNet Performance Summary

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

- **put_nb**
- **get_nb**

<table>
<thead>
<tr>
<th>System</th>
<th>Roundtrip Latency (microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>quadrics</td>
<td></td>
</tr>
<tr>
<td>lemieux</td>
<td></td>
</tr>
<tr>
<td>quadrics</td>
<td></td>
</tr>
<tr>
<td>opus</td>
<td></td>
</tr>
<tr>
<td>myrinet</td>
<td></td>
</tr>
<tr>
<td>alvarez</td>
<td></td>
</tr>
<tr>
<td>myrinet</td>
<td></td>
</tr>
<tr>
<td>citrus</td>
<td></td>
</tr>
<tr>
<td>IA64</td>
<td></td>
</tr>
<tr>
<td>lepie</td>
<td></td>
</tr>
<tr>
<td>poll</td>
<td></td>
</tr>
<tr>
<td>mpi</td>
<td></td>
</tr>
<tr>
<td>lapi</td>
<td></td>
</tr>
<tr>
<td>pcp</td>
<td></td>
</tr>
<tr>
<td>(IA64)</td>
<td></td>
</tr>
<tr>
<td>(x86)</td>
<td></td>
</tr>
<tr>
<td>(PowerPC)</td>
<td></td>
</tr>
<tr>
<td>(x86 PCI-X)</td>
<td></td>
</tr>
<tr>
<td>myrinet</td>
<td></td>
</tr>
<tr>
<td>pc</td>
<td></td>
</tr>
<tr>
<td>(x86 PCI-X)</td>
<td></td>
</tr>
<tr>
<td>infiniband</td>
<td></td>
</tr>
<tr>
<td>pcp</td>
<td></td>
</tr>
<tr>
<td>(x86 PCI-X)</td>
<td></td>
</tr>
</tbody>
</table>
GASNet Performance Summary

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

- Put NB Bulk
- Get NB Bulk

Bandwidth (MB/sec)
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
Problem Motivation

- Partitioned Global-address space (PGAS) languages
  - App performance tends to be sensitive to the latency & overhead
  - Total remotely accessible memory size limited only by VM space
  - Working set of memory being touched likely to fit in physical mem
- Implications for communication layer (GASNet)
  - Want high-bandwidth, zero-copy msgs for large transfers
  - Ideally all communication should be fully one-sided
- Pinning-based NIC's (e.g. Myrinet, Infiniband, Dolphin)
  - Provide one-sided RDMA transfer support, but…
  - Memory must be explicitly registered ahead of time
    - Requires explicit action by the host CPU on both sides
  - Memory registration can be VERY expensive!
    - Myrinet: 40 usecs to register one page, 6000 usecs to deregister
  - Want to reduce the frequency of registration operations and the need for two-sided synchronization
### Memory Registration Approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Zero-copy</th>
<th>One-sided</th>
<th>Full VM avail</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware-based (eg. Quadrics)</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>Hardware manages everything</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No handshaking or bookkeeping in software</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Hardware complexity and price, Kernel modifications</td>
</tr>
<tr>
<td>Pin Everything</td>
<td>✔️</td>
<td>✔️</td>
<td>X</td>
<td>Pin all pages at startup or when allocated (collectively)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Total usage limited to physical memory, may require a custom allocator</td>
</tr>
<tr>
<td>Bounce Buffers</td>
<td>X</td>
<td>X</td>
<td>✔️</td>
<td>Stream data through pre-pinned bufs on one/both sides</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Mem copy costs (CPU consumption/overhead, prevents comm. overlap), Messaging overhead (metadata and handshaking)</td>
</tr>
<tr>
<td>Rendezvous</td>
<td>✔️</td>
<td>X</td>
<td>✔️</td>
<td>Round-trip message to pin remote pages before each op</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Registration costs paid on every operation</td>
</tr>
<tr>
<td>Firehose</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>Common case: All the benefits of hardware-based</td>
</tr>
<tr>
<td></td>
<td>(common case)</td>
<td>(common case)</td>
<td>✔️</td>
<td>Uncommon case: Messaging overhead (metadata and handshaking)</td>
</tr>
</tbody>
</table>
Firehose: Conceptual Diagram

- Runtime snapshot of two nodes (A and C) mapping their firehoses to a third node (B).
- A and C can freely "pour" data through their firehoses using RDMA to/from anywhere in the buckets they map on B.
- Refcounts used to track number of attached firehoses (or local pins).
- Support lazy deregistration for buckets w/ refcount = 0 using a victim FIFO to avoid re-pinning costs.
- For details, see Firehose paper on UPC publications page (CAC'03).
Application Benchmarks

<table>
<thead>
<tr>
<th>App Name</th>
<th>Total Puts</th>
<th>Registration Strategy</th>
<th>Total Runtime</th>
<th>Average Put Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cannon Matrix Multiply</td>
<td>1.5 M</td>
<td>Rendezvous with-unpin</td>
<td>5460 s</td>
<td>5141 µs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rendezvous no-unpin</td>
<td>797 s</td>
<td>34 µs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Firehose (hit: 99.8%)</td>
<td>781 s</td>
<td>14 µs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(miss: 0.2%)</td>
<td></td>
<td>46 µs</td>
</tr>
<tr>
<td>Bitonic Sort</td>
<td>2.1 M</td>
<td>Rendezvous with-unpin</td>
<td>4740 s</td>
<td>522 µs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rendezvous no-unpin</td>
<td>289 s</td>
<td>33 µs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Firehose (hit: 99.98%)</td>
<td>255 s</td>
<td>15 µs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(miss: 0.02%)</td>
<td></td>
<td>54 µs</td>
</tr>
</tbody>
</table>

• Simple kernels written in Titanium - just want a realistic access pattern
  - 2 nodes, Dual PIII-866MHz, 1GB RAM, Myrinet PCI64C, 33MHz/64bit PCI bus
• Firehose misses are rare, and even misses often hit in victim cache
  - Firehose never needed to unpin anything in this case (total mem sz < phys mem)
Performance Results: "Best-case" Bandwidth

- Peak bandwidth - puts to same location with increasing message sz
- Firehose beats Rendezvous no-unpin by eliminating round-trip handshaking msgs
- Firehose gets 100% hit rate - fully one-sided/zero-copy transfers
Performance Results: "Worst-case" Put Bandwidth

- 64 KB puts, uniform randomly distributed over increasing working set size
  - worst-case temporal and spatial locality
- Note graceful degradation of Firehose beyond 400 MB working set

Rendezvous no-unpin exceeds physical memory and crashes at 400MB
Firehose Status and Conclusions

- Firehose algorithm is an ideal registration strategy for PGAS languages on pinning-based networks
  - Performance of Pin-Everything (without the drawbacks) in the common case, degrades to Rendezvous-like behavior for the uncommon case
  - Exposes one-sided, zero-copy RDMA as common case

- Recent work on firehose
  - Generalized Firehose for Infiniband/VAPI-GASNet (region-based), prepared for use in Dolphin/GASNet
  - Algorithmic improvements for better scaling
  - Improving pthread-safe implementation of Firehose
Berkeley UPC Runtime

• UPC-specific layer above GASNet

• Code gen target for our compiler and Intrepid
Pthreaded UPC

- Pthreaded version of the runtime
  - Our current strategy for SMPs and clusters of SMPs
- Implementation challenge: thread-local data.
  - Different solution for binary vs. source-to-source
- Has exposed issues in UPC specification:
  - Global variables in C vs. UPC
  - Misc. standard library issues: rand() behavior
- Plan for the future
  - System V shared memory implementation
  - Benefit: many scientific libraries are not pthread-safe.
  - But: bootstrapping issues, limits on size of shared regions
GCCUPC (Intrepid) support

- GCCUPC can now use Berkeley UPC runtime
  - Generates binary objects that link with our library.
- GCCUPC previously only for shared memory: now able to use any GASNet network
  - Myrinet, Quadrics, Infiniband, MPI, Ethernet
- Demonstrates flexibility of our runtime
  - Primary obstacle: inline functions
  - Current solution:
    - GCCUPC generates performance-critical logic (ptr manipulation, MYTHREAD, etc.) directly
    - Convert other inline functions into regular functions
  - Future: extra inlining pass
    - Read our inline function definitions & generate binary code from them for shared accesses
    - Would give GCCUPC our platform-specific shared pointer representations
C++/Fortran/MPI Interoperability

- Experiment came out of GCCUPC work
  - Needed to publish an explicit initialization API
  - Made sure C++/MPI could use it, so we wouldn’t have to change interface later.

- Motivation: “2nd Front” for UPC acceptance
  - Allow UPC to benefit existing C++/Fortran/MPI codes
  - Allow UPC code to use C++/Fortran/MPI libraries
  - Optimize critical sections of code
  - Communication, CPU overlap
  - Easier to implement certain algorithms
  - Easier to use than GASNet
  - Provide transparently in existing libraries (SuperLU)
C++/MPI Interoperability

• Note: “This is not UPC++”
  - We’re not supporting C++ constructs within UPC
  - C++/MPI can call UPC functions like regular C functions
  - UPC code can call C functions in C++/MPI code
  - UPC functions can return regular C pointers to local shared data, then convert them back to shared pointers to do communication

• Status:
  - Working in both directions: {C++/MPI} --> UPC, and vice versa
  - Tested with IBM xIC, Intel ecc, HP cxx, GNU g++, and their MPI versions.
UPC as a Library Language

• Major limitation: can’t share arbitrary data
  - Can’t share arbitrary global/stack/heap memory: must allocate shared data from UPC calls (local_alloc, etc.)
  - This problem would exist for UPC++, too.

• “Shared everything” UPC
  - Regular dynamic/heap memory: easy (hijack malloc)
  - Stack/global data: harder (but firehose allows)
  - Optional UPC extensions?
    - UPC_SHARED_EVERYTHING
    - Allow pointer casts from local --> shared.

• Interoperability with MPI Communicators
  - subgroup collectives, I/O

• UPC libraries: static vs. dynamic threads
Usability/Stability improvements

- Nightly build of runtime on many configurations:

<table>
<thead>
<tr>
<th>System</th>
<th>Architecture</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>x86/IA64</td>
<td>GM/VAPI</td>
</tr>
<tr>
<td>Tru64</td>
<td>Alpha</td>
<td>Elan/MPI</td>
</tr>
<tr>
<td>AIX</td>
<td>Power 3</td>
<td>LAPI/MPI</td>
</tr>
<tr>
<td>OS X</td>
<td>Power 5</td>
<td>IB/MPI</td>
</tr>
<tr>
<td>SGI Altix</td>
<td>IA64</td>
<td>pthread/MPI</td>
</tr>
<tr>
<td>T3E</td>
<td>Alpha</td>
<td>MPI</td>
</tr>
</tbody>
</table>

- Test suite now contains 250+ test cases
  - works with IBM, Quadrics, PBS batch systems
  - Nightly tests: 20 configurations, including all network types (both single/multi-threads, optimized/debug)
upc_trace
Performance Analysis Tool

• Included in Berkeley UPC 2.0

• Plugs into the existing GASNet tracing facilities
  - records detailed statistics and traces of all GASNet communication activities

• Provides convenient summarization of a GASNet trace file
  - helps you understand the communication behavior of your UPC program
  - helps to find communication "leaks"
  - diagnose load imbalance
upc_trace
Performance Analysis Tool

• Usage is very simple - analogous to gprof
  upcc-trace MG.upc  compile with tracing enabled
  upcrun -trace -n 4 -p 2 MG  enable run w/trace output

• Features:
  - displays all put/get traffic
    - with message size statistics
    - distinguishes shared-remote and shared-local accesses
  - displays all barriers with wait times and notify/wait interval
  - all information is correlated to a source line in your UPC program

• Future plans:
  - Increase Speed and hide internals
  - Features: Track memory allocation & usage, lock/unlock, collectives

• Separate barriers by thread (instead of node)
  - Data analysis services
    - Distribution of resources used in put/get reports
    - Auto detect load imbalance in barrier reports
Applications in PGAS Languages
• Use of the memory model (relaxed/strict) for synchronization
• Medium sized messages done through array copies

**Graph:**

**NAS MG in UPC (Berkeley Version)**

- **Y-axis:** MFlops
- **X-axis:** # Procs

- 0 to 250,000 MFlops range
- 128, 256, 512, 1024 # Procs points
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)
  - Sparse direct solvers
    - Irregular data structures, dynamic/asynchronous communication
    - Small messages
  - Mesh generator
    - Delauney
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
Parallel Triangulation in UPC

- Implementation of a projection-based algorithm (Blelloch, Miller, Talmor, Hardwick)
- Points and processors recursively divided
  - Uses parallel convex hull algorithm (also divide & conquer) to decide on division of points into two sets
  - Each set is then processed by ½ of the processors
- Lowest level of recursion (when we have one processor) performed by Triangle (Shewchuk)
- UPC feedback
  - Teams should really be in the language
  - Non-blocking bulk operations needed
  - May need some optimization guarantees or pragmas (e.g. for vectorization)
Preliminary Timing Numbers

- Time for 1 million points in a sphere

667MHz Alpha/Elan @ MTU (HP UPC)  
*Caveat: Using “optimistic” median scheme*

<table>
<thead>
<tr>
<th>Threads</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14.6</td>
</tr>
<tr>
<td>2</td>
<td>12.61</td>
</tr>
<tr>
<td>4</td>
<td>7.50</td>
</tr>
<tr>
<td>8</td>
<td>5.19</td>
</tr>
</tbody>
</table>
Sparse Solvers

• Sparse matrices arise in many application domains
• Direct solvers are even more challenging than iterator
• Investigating SuperLU on the X1 in collaboration with X. Li
• SuperLU factors a matrix; after factoring triangular solves (one or many) follow
• Sparse Triangular Solve (SpTS).
  - Solve for $x$ in $Tx = b$ where $T$ is a lower triangular sparse
  - Used after sparse Cholesky or LU factorization to solve sparse linear systems
• Irregularity arises from dependence
• Hard to parallelize
  - dependence structures only known at runtime
  - must effectively build dependence tree in parallel
Performance

- Linear scaling is not expected or achieved
- Plan to compare to MPI implementation
Summary

• Berkeley UPC compiler has made UPC ubiquitous
  - PCs, desktops, cluster, SMPs, supercomputers
• Performance portability is the next challenge
  - Compiler optimizations for communication
  - Runtime optimizations (caching)
• Language questions remain
  - Consistency model: can we simplify it through better compiler analysis?
  - Are explicit non-blocking primitives?
• Better tool support is key
  - Debugging as well as performance tools
  - Present high level information--the vector super model