GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages

Dan Bonachea

Jaein Jeong

In conjunction with the joint UCB and NERSC/LBL UPC compiler development project

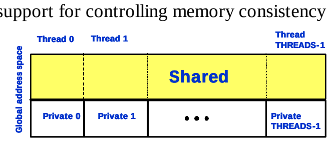
http://upc.nersc.gov

Introduction

- Two major paradigms for parallel programming
  - Shared Memory
    - single logical memory space, loads and stores for communication
    - ease of programming
  - Message Passing
    - disjoint memory spaces, explicit communication
    - often more scalable and higher-performance
- Another Possibility: Global-Address Space Languages
  - Provide a global shared memory abstraction to the user, regardless of the hardware implementation
  - Make distinction between local & remote memory explicit
  - Get the ease of shared memory programming, and the performance of message passing

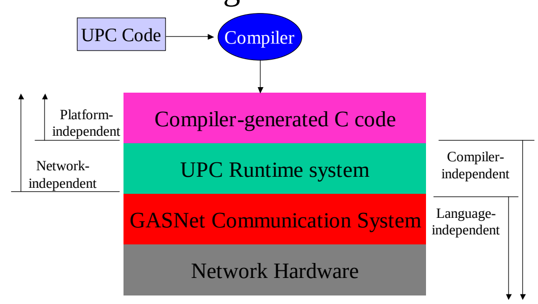
UPC (Unified Parallel C)

- A explicitly-parallel SPMD programming language with a Global-Address Space abstraction
  - Superset of the C programming language
  - Threads have a private memory area and share a global memory
  - Language support for data distribution using shared arrays
  - Language support for controlling memory consistency model



- Current UPC compiler implementations generate code directly for the target system
  - Requires compilers to be rewritten from scratch for each platform and network
  - We want a more portable, but still high-performance solution…

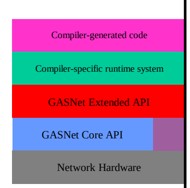
NERSC/UPC Runtime System Organization



GASNet Communication System- Goals

- Language-independence: Compatibility with several global-address space languages and compilers
  - UPC, Titanium, Co-array Fortran, possibly others..
  - Hide UPC- or compiler-specific details such as shared-pointer representation
- Hardware-independence: variety of parallel architectures & OS's
  - SMP: Origin 2000, Linux/Solaris multiprocessors, etc.
  - Clusters of uniprocessors: Linux clusters (myrinet, infiniband, via, etc)
  - Clusters of SMPs: IBM SP-2 (LAPI), Linux CLUMPS, etc.
- Ease of implementation on new hardware
  - Allow quick implementations
  - Allow implementations to leverage performance characteristics of hardware
- Want both portability & performance

GASNet Communication System- Architecture



- 2-Level architecture to ease implementation:
- Core API
  - Most basic required primitives, as narrow and general as possible
  - Implemented directly on each platform
  - 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

Our goals in this semester project (what we've done)

- Wrote the GASNet Specification
  - Included inventing a mechanism for safely providing atomicity in Active Message handlers
- Reference implementation of extended API
  - Written solely in terms of the core API
- Implemented a prototype core API for one platform (a portable MPI-based core)
- Evaluate the performance using micro benchmarks to measure bandwidth and latency
  - Focus on the additional overhead of using GASNet

Extended API – Remote memory operations

- Orthogonal, expressive, high-performance interface
  - Gets & Puts for Scalars and Bulk contiguous data
  - Blocking and non-blocking (returns a handle)
  - Also have a non-blocking form where the handle is implicit
- Non-blocking synchronization
  - Sync on a particular operation (using a handle)
  - Sync on a list of handles (some or all)
  - Sync on all pending reads, writes or both (for implicit handles)
  - Sync on operations initiated in a given interval
  - Allow polling (trysync) or blocking (waitsync)
- Useful for experimenting with a variety of parallel compiler optimization techniques

Extended API – Remote memory operations

- API for remote gets/puts:

```
void   get    (void *dest, int node, void *src, int numbytes)
handle get_nb (void *dest, int node, void *src, int numbytes)
void   get_nbi(void *dest, int node, void *src, int numbytes)

void   put    (int node, void *src, void *src, int numbytes)
handle put_nb (int node, void *src, void *src, int numbytes)
void   put_nbi(int node, void *src, void *src, int numbytes)
```

- "nb" = non-blocking with explicit handle
- "nbi" = non-blocking with implicit handle
- Also have "value" forms that are register-memory
- Recognize and optimize common sizes with macros
- Extensibility of core API allows easily adding other more complicated access patterns (scatter/gather, strided, etc)
- Names will all be prefixed by "gasnet_" to prevent naming conflicts

Extended API – Remote memory operations

- API for get/put synchronization:
- Non-blocking ops with explicit handles:

```
int  try_syncnb(handle)
void wait_syncnb(handle)

int  try_syncnb_some(handle *, int numhandles)
void wait_syncnb_some(handle *, int numhandles)
int  try_syncnb_all(handle *, int numhandles)
void wait_syncnb_all(handle *, int numhandles)
```

- Non-blocking ops with implicit handles:

```
int  try_syncnbi_gets()
void wait_syncnbi_gets()
int  try_syncnbi_puts()
void wait_syncnbi_puts()
int  try_syncnbi_all()  // gets & puts
void wait_syncnbi_all()
```

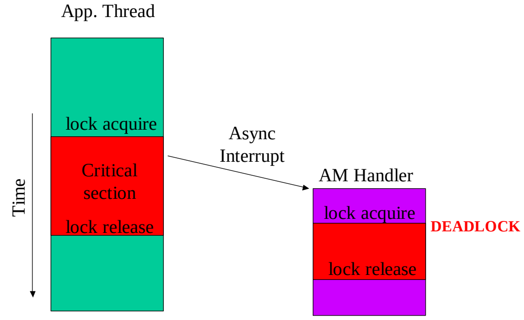
Core API – Active Messages

- Super-Lightweight RPC
  - Unordered, reliable delivery
  - Matched request/reply serviced by "user"-provided lightweight handlers
  - General enough to implement almost any communication pattern
- Request/reply messages
  - 3 sizes: short (<=32 bytes),medium (<=512 bytes), long (DMA)
- Very general - provides extensibility
  - Available for implementing compiler-specific operations
  - scatter-gather or strided memory access, remote allocation, etc.
- Already implemented on a number of interconnects
  - MPI, LAPI, UDP/Ethernet, Via, Myrinet, and others
- Started with AM-2 specification
  - Remove some unneeded complexities (e.g. multiple endpoint support)
  - Add 64-bit support and explicit atomicity control (handler-safe locks)

Core API – Atomicity Support for Active Messages

- Atomicity in traditional Active Messages:
  - handlers run atomically wrt. each other & main thread
  - handlers never allowed block (e.g. to acquire a lock)
  - atomicity achieved by serializing everything (even when not reqd)
- Want to improve concurrency of handlers
- Want to support various handler servicing paradigms while still providing atomicity
  - Interrupt-based or polling-based handlers, NIC-thread polling
  - Want to support multi-threaded clients on an SMP
  - Want to allow concurrency between handlers on an SMP
- New Mechanism: Handler-Safe Locks
  - Special kind of lock that is safe to acquire within a handler
    - HSL's include a set of usage constraints on the client and a set of implementation guarantees which make them safe to acquire in a handler
  - Allows client to implement critical sections within handlers

Why interrupt-based handlers cause problems



Analogous problem if app thread makes a synchronous network call (which may poll for handlers) within the critical section

Handler-Safe Locks

- HSL is a basic mutex lock
  - imposes some additional usage rules on the client
  - allows handlers to safely perform synchronization
- HSL's must always be held for a "bounded" amount of time
  - Can't block/spin-wait for a handler result while holding an HSL
  - Handlers that acquire them must also release them
  - No synchronous network calls allowed while holding
  - AM Interrupts disabled to prevent asynchronous handler execution
- Rules prevent deadlocks on HSL's involving multiple handlers and/or the application code
  - Allows interrupt-driven handler execution
  - Allows multiple threads to concurrently execute handlers

No-Interrupt Sections

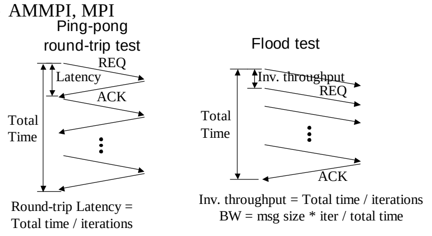
- Problem:
  - Interrupt-based AM implementations run handlers asynchronously wrt. main computation (e.g. from a UNIX signal handler)
  - May not be safe if handler needs to call non-signal-safe functions (e.g. malloc)
- Solution:
  - Allow threads to temporarily disable interrupt-based handler execution: hold_interrupts(), resume_interrupts()
  - Wrap any calls to non-signal safe functions in a no-interrupt section
  - Hold & resume can be implemented very efficiently using 2 simple bits in memory (interruptsEnabled bit, messageArrived bit)

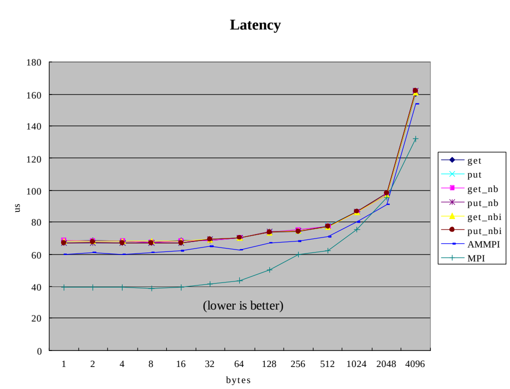
Jaein's part

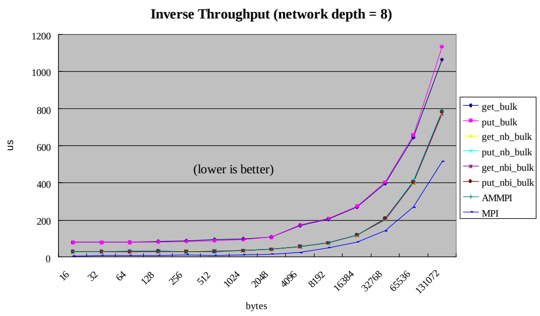
Performance Benchmarking of prototype MPI-based GASNet core (built on pre-existing AM-MPI)

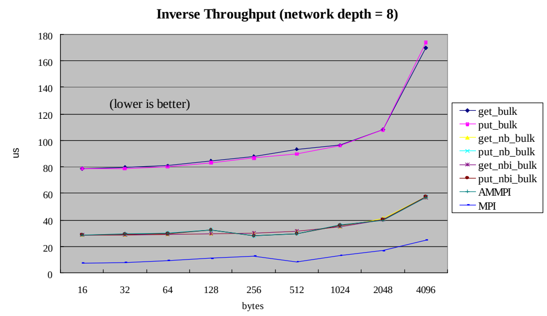
Experiments

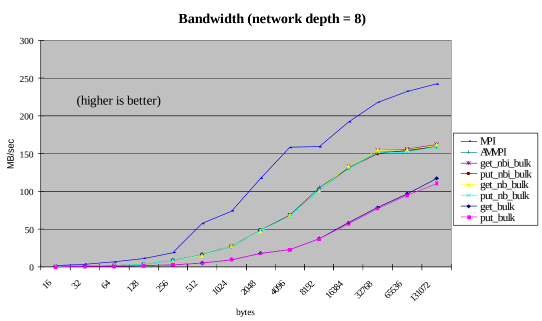
- Experimental Platform: IBM SP Seaborg
- Micro-Benchmarks: ping-pong and flood
- Comparison
  - blocking get/put, non-blocking get/put (explicit and implicit)
  - AMMPI, MPI











Results

- Explicit and implicit non-blocking get/put performed equally well
- Latency was good but can be tuned further
  – blocking and non-blocking I/O had 7 us overhead over AMMPI
- Bandwidth and throughput were satisfactory
  – Non-blocking I/O performed as well as AMMPI.
- Overall performance is dominated by AMMPI implementation
- Expect better GASNet performance on a native AM implementation

	Blocking	Non-blocking	AMMPI	MPI
Latency (ping-pong round trip)	67 us	67 us	60 us	39 us
Inv throughput (flood: at 16bytes)	79 us	29 us	29 us	8 us
Bandwidth (flood: at 128KB)	113 MB/sec	160 MB/sec	159 MB/sec	242 MB/sec

Conclusions & Future Work

- MPI is not a good match for implementing global-address space languages
  – Semantic mismatch between non-blocking get/put accesses and msg send/recv
- Atomicity for active message handlers
  – Handler-safe locks allow handler concurrency & interrupt-based handlers
- Future Work:
  – Implement GASNet on other interconnects
    - LAPI, GM, Quadrics, Infiniband …
  – Tune AMMPI for better performance on specific platforms
  – Augment Extended API with other useful functions
    - Collective communication (broadcast, reductions)
    - More sophisticated memory access ops (scatter/gather)