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

Dan Bonachea, Mike Welcome, Christian Bell, Paul Hargrove

In conjunction with the joint UC Berkeley and LBL Berkeley UPC compiler development project
http://upc.lbl.gov
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), Compaq Alphaserver, Linux CLUMPS. etc.
GASNet Communication System- Goals (cont)

• 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 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
Progress to Date

• Designed & wrote the GASNet Specification
• Reference implementation of extended API
  – Written solely in terms of the core API
• Implemented a portable MPI-based core API
• Completed native (core&extended) GASNet implementations for several high-performance networks:
  – Quadrics Elan (Dan)
  – Myrinet GM (Christian)
  – IBM LAPI (Mike)
• Did initial public release of GASNet
• Implementation under-way for Infiniband (Paul)
  – other networks under consideration
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.

- AM previously implemented on a number of interconnects
  - MPI, LAPI, UDP/Ethernet, Via, Myrinet, and others

- Includes mechanism for explicit atomicity control in handlers
  - Even in the presence of interrupts & multithreading
  - Handler-safe locks & no-interrupt sections
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 *dest, int numbytes)
handle put_nb (int node, void *src, void *dest, int numbytes)
void put_nbi(int node, void *src, void *dest, int numbytes)

• "nb"/"nbi" = non-blocking with explicit/implicit handle
• Also have "value" forms that are register-memory, and "bulk" forms optimized for large memory transfers
• Extensibility of core API allows easily adding other more complicated access patterns (scatter/gather, strided, etc)
Extended API – Remote memory operations

• API for get/put synchronization:

• Non-blocking sync with explicit handles:
  
  ```c
  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 sync with implicit handles:

  ```c
  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()
  ```
Code Generation Tradeoffs

• Blocking vs. Non-blocking puts/gets
• Put/Get variety: non-bulk vs. bulk
  – optimized for small scalars vs large zero-copy
  – difference in semantics - put src, alignment
• Put/Get synchronization mechanism
  – expressiveness/complexity tradeoffs
  – explicit handle vs. implicit handle, access regions
• Work remains to explore these tradeoffs in the context of code generation
Performance Results
Experiments

- Micro-Benchmarks: ping-pong and flood

**Ping-pong round-trip latency test**

- Round-trip Latency = Total time / iterations

**Flood bandwidth test**

- Inv. throughput = Total time / iterations
- BW = msg size * iter / total time
GASNet Configurations Tested

• Quadrics (elan):
  – mpi-refext - AMMPI core, AM-based puts/gets
  – elan-elan - pure native elan implementation

• Myrinet (GM):
  – mpi-refext - AMMPI core, AM-based puts/gets
  – gm-gm - pure native GM implementation

• IBM SP (LAPI):
  – mpi-refext - AMMPI core, AM-based puts/gets
  – lapi-lapi - pure native LAPI implementation
System Configurations Tested

- Quadrics - falcon/colt (ORNL)
  - Compaq Alphaserver SC 2.0, ES40 Elan3, single-rail
  - 64-node, 4-way 667 MHz Alpha EV67, 2GB, libelan1.2/1.3, OSF 5.1
- Quadrics - lemieux (PSC)
  - Compaq Alphaserver SC, ES45 Elan3, double-rail (only tested w/single)
  - 750-node, 4-way 1GHz Alpha, 4GB, libelan1.3, OSF 5.1
- Quadrics - opus (PNL)
  - Itanium-2 Cluster, Elan3, double-rail (only tested w/single)
  - 128-node, 2-way 1GHz Itanium-2, 12GB, libelan1.4, Redhat Linux 7.2
- Myrinet - Alvarez (NERSC)
  - x86 Cluster, 33Mhz 64-bit Myrinet 2000 PCI64C, 200 MHz Lanai 9.2
  - 80-node, 2-way 866 Mhz P3, 1GB, GM 1.5.1, Redhat Linux 7.2
  - Empirical PCI bus bandwidth: 229MB/sec read, 245 MB/sec write
- LAPI - seaborg (NERSC)
  - IBM RS/6000 SP Power3, Colony-GX network
  - 380-node, 16-way 375MHz Power3, 64GB, 64KB L1, 8MB L2, AIX 5.1
GASNet Put/Get Roundtrip Latency (min over msg sz)

- put_nb
- get_nb

Microseconds

- mpi
- elan
- quadrics falcon
- quadrics colt
- quadrics opus
- quadrics lemieux
- myrinet alvarez
- Colony/GX seaborg

- mpi
- lapi
- lapi-poll
Quadrics elan-conduit

• Implementation based on elan-lib
  – the "portable" Quadrics API (will be supported on elan4)

• Core API
  – Polling-based implementation on elan queue API and TPORTS API
  – Uses zero-copy elan RDMA puts for AM Long msgs

• Extended API
  – Put/get implemented using zero-copy elan RDMA puts/gets in the common case
  – Some uncommon cases require bounce buffers or active messages as fallback
  – Barriers implemented using Quadrics hardware barrier for anonymous barriers, or broadcast/barrier for named
Empirical round-trip latency of hardware: ~3.4 us
Theoretical peak bandwidth of NIC hardware: 340 MB/sec
Quadrics elan-conduit: Future work

- Work-around or resolve some problems encountered in Quadrics elan-lib software
  - dual-rail operation
  - loopback on SMP nodes sharing a NIC
- Further performance tuning
  - based on feedback from app experience
  - implement split-phase barrier on NIC processor
- Continued maintenance with new versions of elanlib
  - new elan4 hardware expected soon
- We'd really like some Quadrics hardware of our own to play with! :}
Myrinet gm-conduit

- Work done by Christian Bell
- Initial Core API implementation took 2 weeks
  - AM implementation fairly straightforward over GM for Small/Medium AMs
  - Long/LongAsync AMs required more work for DMA support (addressed in extended API and Firehose algorithm)
  - Polling-based conduit (currently)
  - Under threaded GASNet configuration (PAR), allows for concurrent handler execution
Myrinet gm-conduit

• Extended API took 1 month
  – Proposed and published a new algorithm, Firehose algorithm, to improve performance of one-sided operations over pinning-based networks (GM, Infiniband) (to be presented at CAC ’03)
  – One-sided operations used for bulk and non-bulk puts
  – Gets currently use an AM with a one-sided put (GM 2.0 will add one-sided gets)

• Bootstrapping problem
  – Each Myrinet site must develop a custom bootstrapper or use 3rd-party solutions (Millennium nightmare)
  – GM conduit provides bootstrapping support for both dedicated (PBS) and non-dedicated (gexec) cluster configurations.
Empirical round-trip latency of hardware: \(~17\) us
Empirical peak bandwidth of hardware: ~210 MB/sec (puts only)
Myrinet gm-conduit

- **Future**
  - More efforts in tuning Firehose algorithm
  - Support for GM 2.0 and one-sided gets
  - Hooks for minimal interrupt support
  - Continued bootstrapping support
GASNet/LAPI for IBM SP

• Initial (non-optimized) implementation took 2 weeks
  – Use of GASNet conduit template provided simple implementation framework
  – GASNet PUT/GET Implemented using LAPI PUT/GET
  – GASNet AM Request/Reply and Barriers implemented using LAPI AMs
  – Non-blocking Sync methods implemented using LAPI counters
  – Handler Safe Locks implemented using Pthread mutex
  – No-Interrupt sections a No-op
  – No memory registration issues

• 3 weeks for Active Message optimizations

• LAPI Conduit can run in Interrupt or Polling mode
GASNet/LAPI: AM Optimizations

- Optimizations only apply to GASNet operations implemented using LAPI AM
  - Specifically GASNet AM and Barrier operations
  - Not needed for GASNet PUT/GET

- GASNet token caching and re-use to reduce allocation overhead

- Packing small message payload into LAPI AM Header Handler argument to reduce GASNet AM latency.

- Implementation of “Ready Queue” for quick execution of GASNet AM Request/Reply handlers
  - Eliminate 40-60 usec latency to schedule LAPI Completion Handler
  - “Ready” handlers executed by main thread while polling
LAPI AM: Execution Flow

**Origin Task**

1: `Amsend(tgt, HH, uhdr, payload...)`

... Data Data uhdr

18-20 us

**Target Task**

2: HH executes in Dispatcher
   * specify tgt addr for payload
   * register CH (optional)

3: After all data arrives Dispatcher will schedule CH (if registered)

4: CH executes in Completion Thread

40-60 us!!

5: Completion at Origin

Notes:
- HH is LAPI Header Handler
- CH is LAPI Completion Handler
- HH cannot block or issue comm calls
- uhdr is arg to HH (restricted size)
- Data Payload is optional
- CH registration is optional
- CH can execute arbitrary code

Time: 80-100 us (no data payload)
Empirical round-trip latency of hardware: ~42 us
Empirical peak bandwidth of hardware: ~350 MB/sec
GASNet/LAPI: Future Work

- Possible Future Optimizations:
  - Reduce/Eliminate locking overhead (costly on SP)
    - Token allocation
    - Access to “Ready Queue”
  - Improve Split-phase Barrier implementation
    - Broadcast Tree?
    - Implement as blocking barrier using LAPI_Gfence?
  - Throttle NB PUT/GET to avoid performance drop-off

- Future LAPI may allow restricted communication in HH
  - Would eliminate need for ready queue or CH for small message GASNet Request AM
  - NOTE: IBM will use this (future) LAPI version to re-implement MPI
Conclusions

GASNet provides a portable & high-performance interface for implementing GAS languages

• two-level design allows rapid prototyping & careful tuning for hardware-specific network capabilities

• We have a fully portable MPI-based implementation of GASNet, several native implementations (Myrinet, Quadrics, LAPI) and other implementations on the way (Infiniband)

• Performance results are very promising
  – Overheads of GASNet are low compared to underlying network
  – Interface provides the right primitives for use as a compilation target, to support advanced compiler communication scheduling
Future Work

• Further tune our native GASNet implementations
• Implement GASNet on new interconnects
  – Infiniband, Cray T3E, Dolphin SCI, SGI SHMEM, Cray X-1…
• Implement GASNet on other portable interfaces
  – UDP/Ethernet, ARMCI…
• Augment Extended API with other useful functions
  – Collective communication
    • broadcast, reduce, all-to-all
    • interface to be based on UPC Collective spec & Titanium collective ops
  – More sophisticated memory access ops
    • strided, scatter/gather (indexed put/get)
    • interface to be based on ARMCI and Titanium ops
• Network benchmarking based on GASNet (Paul)
More Future Work

• Collaborate with ARMCI effort
  – GASNet-over-ARMCI / or using ARMCI

• Potential External Collaborations
  – (Go)DIVA HPCS Darpa project, Quadrics, others..

• Implement some small, real applications directly on GASNet
  – Experiment with the interface to gain further insights into good code-generation strategies
  – Gather some app-level performance results
Extra Slides
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 (GAS)
  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
  – Examples: UPC, Titanium, Co-array Fortran, …
The Case for Portability

• Most 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
  – Want to re-use our investment in compiler technology across different platforms, networks and machine generations
  – Want to compare the effects of experimental parallel compiler optimizations across platforms
  – The existence of a fully portable compiler helps the acceptability of UPC as a whole for application writers
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

App. Thread

Time

lock acquire

Critical section

lock release

Async Interrupt

AM Handler

lock acquire

lock release

DEADLOCK

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 (interruptsDisabled bit, messageArrived bit)
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

Ping-pong round-trip test

Flood test

Round-trip Latency = Total time / iterations

Inv. throughput = Total time / iterations

BW = msg size * iter / total time
Latency (IBM SP, network depth = 8)

- Additional overhead of get/puts over AMMFI: 7 us
- Blocking and non-blocking get/puts equivalent
Bandwidth (IBM SP, network depth = 8)

- Non-blocking get/puts performed as well as AMMPP
- Non-blocking get/puts are benefited from overlap
Inv. Throughput (IBM SP, network depth = 8)

- Non-blocking get/puts performed as well as AMMPl
• Implies sender overhead.
• The difference from two round-trip latency can be used to estimate wire-delay and receiver overhead
GASNet Put/Get Bulk Bandwidth
(max over msg sz)

- put_nb_bulk
- get_nb_bulk

MB/sec

- mpi-refext
- elan-refext
- elan-elan
- mpi-refext
- elan-refext
- elan-elan
- mpi-refext
- gm-gm
- mpi-refext
- gm-gm

quadrics - falcon
quadrics - lemieux
myrinet - millennium
myrinet - alvarez
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.

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