IBM Research

Is PGAS a viable programming model for exascale?

George Almási

Outline

- **Part 1: Assumptions about exascale; state of art**
- **Part 2: A critical examination of PGAS features**
- **Part 3: How HW can support PGAS**
- **Part 4: A biased/uninformed personal view of the future**

Basic assumptions about exascale architecture

- **Floating point oriented architecture**
  - Architecture crammed full of FPUs at expense of sanity
  - It's not ExaByte or ExaByte/s or ExaOp/s
  - Low BW/FP, memory/FP ratios
    - BLAS3 possible, BLAS2 broken, BLAS1 SOL
- **Deep memory & execution hierarchies**
  - Multiple levels of cache
  - Multiple memory domains (limited coherence?)
  - Multiple levels/types of execution units (!)
  - Multiple levels/types of network connections (!)

OpenMP+MPI

- **It is the law of the land**
- **Proven track record**
  - Millions of LOC
- **Well defined roles**
  - MPI for coarse grain
  - OpenMP for fine grain

Two solutions for two problems - funny interactions

- MPI thread funneling problem
- Linkage conflict:
  - MPI task-global
  - OMP thread-local
- Compromises MPI modularity (communicators)

How about accelerators?

Accelerator boards, GPU computing

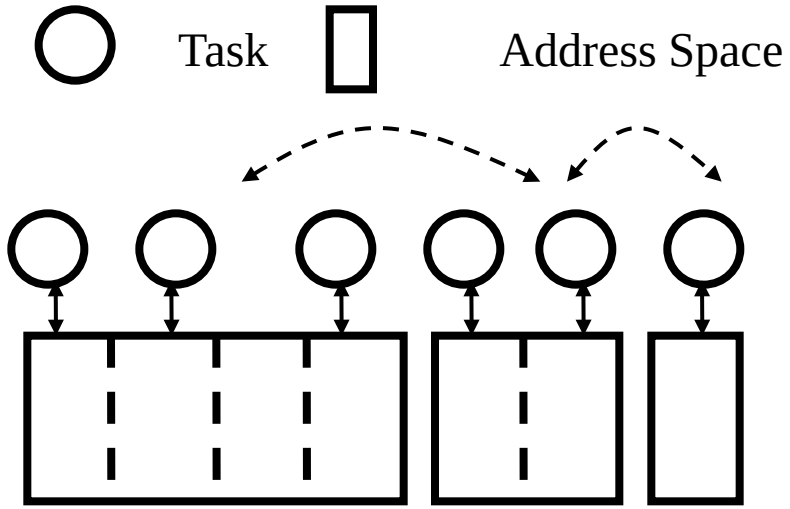
- **A few fat MPI nodes, many wimpy accelerators**
  - Enough cheap performance to lure developers into re-writing apps (already happening)
    - People willing to go back to single precision
    - People willing to coalesce mem access
  - This makes MPI+OpenMP look really good
- **Accelerator kernels only communicate with host**
  - Perfectly suitable for divide-and-conquer algorithms
  - Not so good for graph algorithms and chasing pointers across the system

Part 2: PGAS “Too little, too late?”

NO! A lot, and insane

What is Partitioned Global Address Space, really? Shared memory ... kinda, sorta

- **Global uniform shared memory is too expensive/power hungry to build**
  - Modern machines have network devices, NUMA shared memory
- **Provide partial illusion of shared memory**
  - Restricted to certain software constructs (“shared” vs “local”)
  - Explicit SW control of affinity (association b/w memory and task)
  - Encourage coarse grain communication: explicit one-sided communication primitives



Enough rope to hang yourself: programmer allowed to ignore affinity and hurt performance.

PGAS languages enlist compiler to mitigate effects of locality-naive code.

We will now re-examine PGAS features

- **Sanity**
- **Interoperability with MPI**
- **Fit with machine hierarchy**
- **Implementability**

Variations on PGAS languages

- **One-sided access**
  - Modeled after shared memory
- **Global View & Array manipulation**
  - APL, Matlab, HPF
- **Pointers to shared**
  - Modeled after C, C++
- **Asynchronous execution**
  - Model: Scala
- **Collective communication**
  - Model: MPI
- **Lock synchronization**
  - Model: POSIX locks

Explicit one-sided data access

- **Good:**
  - Passive participant's CPU not interrupted
  - No handshake required
  - Looks like shared memory (w/ compiler to help)
  - (Almost) implementable with modern network HW

Bad:

Violates MPI data contract at a deep level

RAW, WAR etc. conflicts

“Strict” consistency just not practical with 1-10 usec sync overhead

Every PGAS language has its own version of ”relaxed” semantics

Truly scalable fence op very difficult to implement

Array languages and the Global View

- **Sequential operation on large blob of data:**
  - "multiply matrix A with matrix B" (A, B distributed)
  - "sum of A/eigenvalues of B/etc"
  - HPF, HTAs, Chapel, ZPL, elements of UPC and CAF
- **Easy to program**
- **Niche player (admittedly a large niche)**
  - Fantastic for regular problems
  - Useful but not great for irregular problems

Pointers to shared objects

- **Familiar from C programming**
  - Shared-memory flavor
  - Allows e.g. pointer chasing
- **Encourages worst C programming paradigms**
  - Strains type system (local->shared->local conversions)
  - Strains “array==pointer” dogma to breaking point
    - Encourages disregard of affinity through ops like “++”
  - Haunted by asynchrony and relaxed memory consistency
  - Encourages fine grained remote access
- **“Fat pointers” are very expensive to de-reference locally**
- **Prominent feature of UPC; disallowed by CAF**

Asynchronous (aka split-phase) remote access

- **Reason for existence:**
  - Mitigate latency (split-C)
  - Allow overlapping of communication with *anything*
  - Split-C, UPC extensions

The problem:

Split-phase transactions require programmer discipline

More discipline than most programmers have

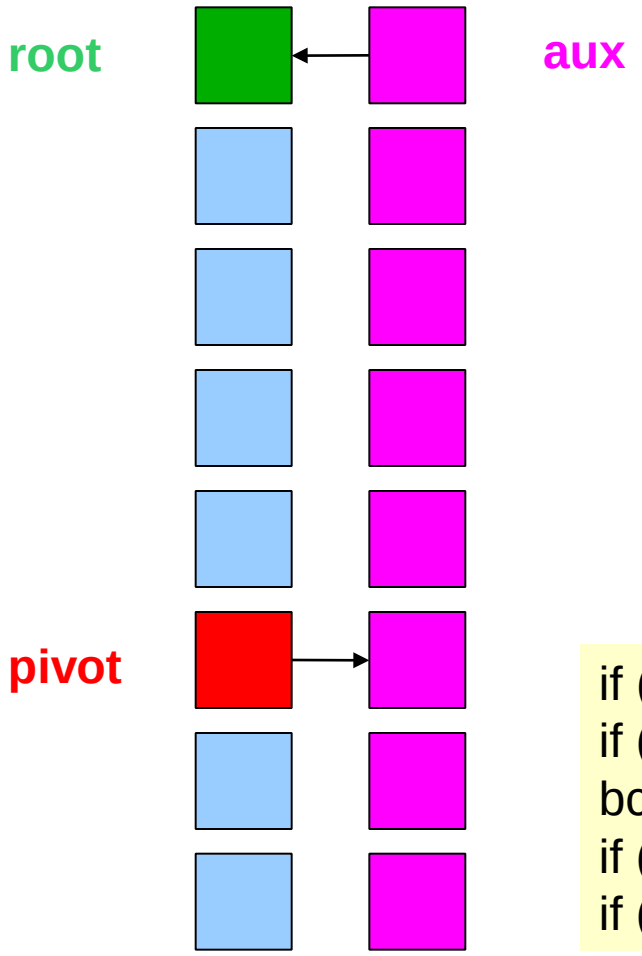
Subtle bugs abound

CAF solution: leave it to compiler!

No asynchronous access in language syntax

Consistency model allows async. impl. of remote accesses

The pitfalls of asynchronous access: an example



HPC Challenge code, HPL linpack

Operation:
- Broadcast pivot to everyone
- Copy root to pivot

Overlap solution:
- Copy pivot into a temporary (“aux”)
- Broadcast across “aux” array
- (Overlap) copy from root to pivot
- Copy aux->root

```
if (a_piv) memcpy (aux, pivot, blksize);
if (a_piv) v = memget_async  (pivot, root, blksize);
bcast (..., aux, blksize);
if (a_root) memcpy  (root, aux, blksize);
if (v) upc_waitsync(v);
```

Collective communication in global memory

- **Patterns for manipulating distributed instr & data flow**
  - Synchronization:
    - Barriers, fences, various forms of locks and atomic sections
  - Data exchanges:
    - Broadcast, scatter/gather, personalized communication
  - Distributed computation:
    - Reductions, prefix sums

MPI data contract:

Data is given to MPI primitive, returned at end of op; do not touch during operation

Interoperability trouble:

Difficult notion of “giving” non-local data to collective

Difficult to guarantee data integrity in the presence of remote one-sided data access

(UPC) profusion of unintelligible flags fail to control situation

Non-SPMD programming models

- **Asynchronous remote execution**
  - X10, Scala, “CAF 2.0”
- **Freedom from tyranny of SPMD**
  - End-run around memory consistency
    - All data is “local” or transferred via asyncs
- **Problems:**
  - Global termination detection
  - Niche player
  - No real thought has gone into integration with MPI (yet)
  - “Async” implementation is iffy (but not impossible)

The PGAS class wars: two populations of memory

- **Default shared:**
  - Closer to shared memory ideal
  - Global view objects are always shared
  - Encourages overuse of shared objects, makes performance & correctness more difficult
  - ZPL, Titanium

Default local:

Better (I didn't say good!) integration w/ MPI

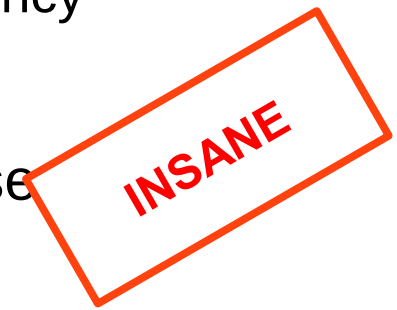
Tighter control on what can be accessed

Two classes of objects, sometimes necessitating extra local copies

UPC, CAF, HPF, X10 etc

Lock synchronization, atomic sections etc

- Tempting because familiar
- To pthreads programmers
- Utterly insane to think it scales across network
- Lock contention == network congestion
  - Ramifications beyond poor lock latency
- 3 orders of magnitude increase in lock latency
- Every PGAS program (that I have seen) that use
  has severe scalability issues
- Possible solution: limit scope of locks (somehow)



Part 3: PGAS and exascale hardware

- Non-coherent shared memory
- MPI on a network hierarchy
- No hierarchy funneling please
- Collectives on a network hierarchy
- The unbearable fine granularity of access
- Memory fences are hard to implement
- Symmetric allocation and short RDMAs

Non-coherent shared memory is insane

Memory

Main processor cannot touch

```
loop:
  ld …, buffer
  st …, network
  bdnz loop
```

Last iteration:
branch predictor predicts branch taken
ld executes speculatively

- cache miss causes first line of forbidden buffer area to be fetched into cache
- system executes branch, rolls back speculative loads
- **does not roll back cache line fetch (because it's nondestructive)**

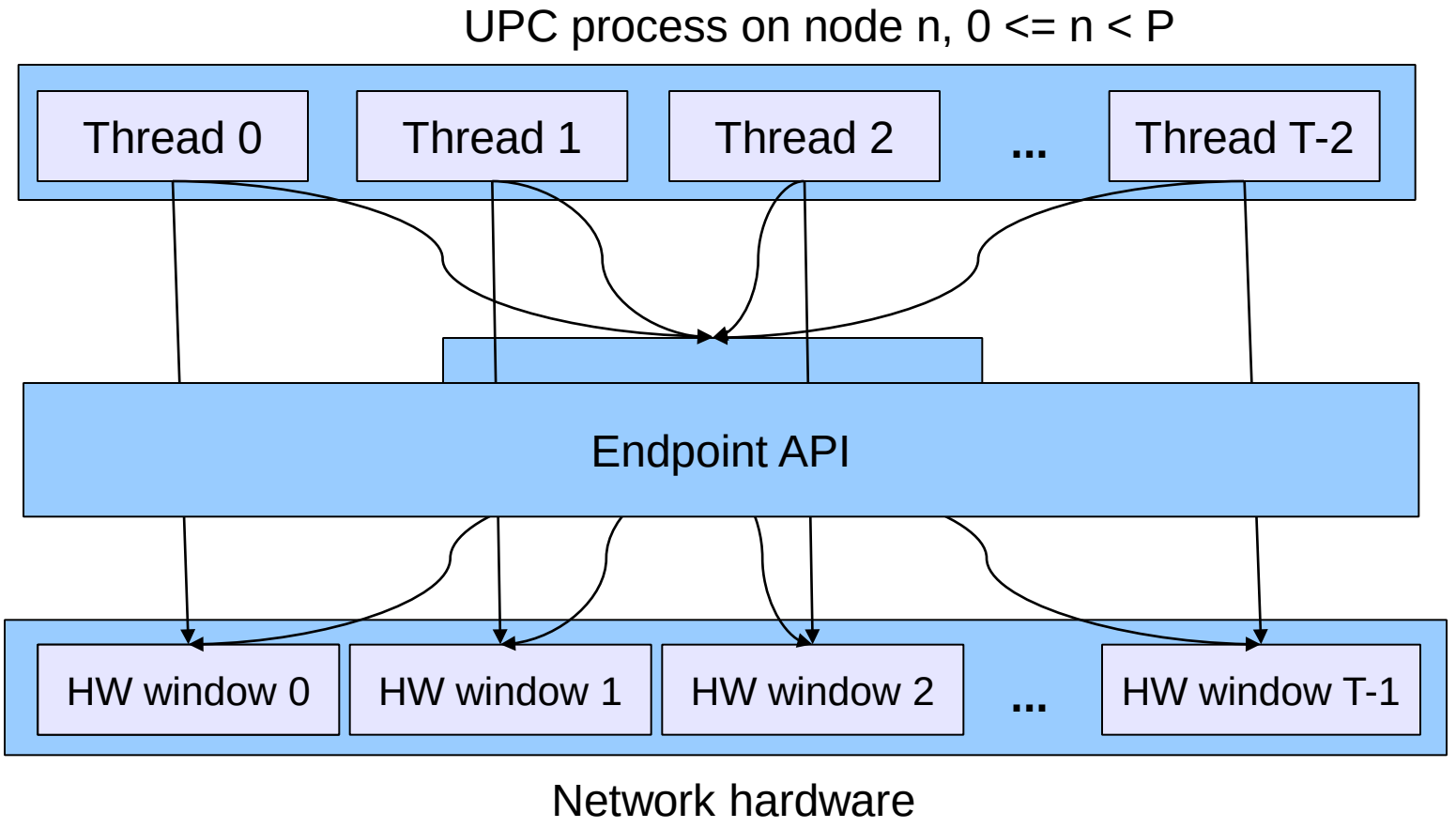
Conclusion: CPU 0 ends up with stale data in cache
But only when cache line actually survives before being used

Getting MPI to run on a hierarchical network architecture

- **Ironic: MPI assumes a flat network**
  - Is no better equipped to deal w/ exaflop than shmem is
- **MPI could deal with this:**
  - Multi-device implementations (available today)
  - Multiple levels of communicators (a la “EWORLD”)
  - MPI_THREAD_MULTI- like guarantees
- **Do we maintain fiction of any2any communication?**
  - Network hardware must be connectionless & reliable
    - Or SW state machine for each pt2pt connection!
      - $O(P^2)$ memory!

Endpoints, aka system software is frequently neglected

Hybrid mode execution: P nodes x T threads/node



Collectives on heterogeneous architectures

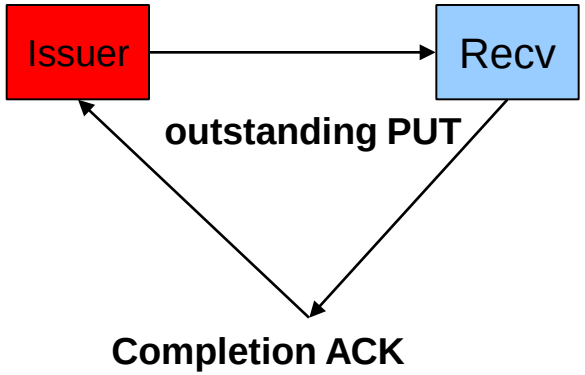
- **State of the art:**
  - HW acceleration for intra-node
    - Reductions, broadcasts, barriers
  - Not much of anything done for in-node
- **PERCS:**
  - 2^7 threads in memory coherence domain -> trouble!
  - 2^{14} nodes in intranode domain -> CAU
  - Extrapolate: more, and deeper, domains
- **BTW, how does one chain collectives across networks?**
  - Extra credit: how does one chain non-blocking collectives across networks?

Fine-grain one-sided communication

- **True one-sided communication (no CPU involvement)**
  - Remote load/store support (very, very expensive)
  - RDMA for very short messages
    - Network hardware has to snoop TLBs, inject into cache; huge risk for HW designers
  - HW/SW to assure symmetry in allocations
    - Or we build our own allocator (bad!)
- **HW support for active messages:**
  - Fire a thread on remote end to execute function
  - Trouble with context switches & resource allocation
    - Suppose fired thread wants to communicate?

How to not implement PGAS fence support

- **Chain of "reasonable" decisions leads to bad performance**
- **Initial implementation of UPC fences take 200ms**
  - when asymmetric communication pattern followed by upc barrier



PAMI fence implementation:
Issuer counts outstanding messages
Receivers issue ACKs
ACKs retire outstanding messages
Fence complete when outstanding == 0

Default PAMI behavior:
Fence ACKs ride piggyback
on reliability layer (HAL) ACKs

Default HAL behavior:
Coalesced packet ACKs
Lazy delivery of ACKs
Mitigated by timer interrupt

Part 4: Predicting winners & losers

Why is PGAS not taking over the world?

- **Performance equivalence:**
  - Necessary; not sufficient
- **Better productivity:**
  - Not proven until used enough
  - Does anyone actually care about productivity?
- **Portability:**
  - Platforms x compilers
  - Performance portability!

Not backward compatible

Interoperability with MPI is ill-defined

Contortions on both sides

Winning strategies:

Better productivity? ... **no**

Higher pain treshold for business as usual? ... **yes, but** ...

Enable business as usual? **YES**

New functionality? ... **not sure**

Interlude: the many faults of UPC

- **Complicated:**
  - Block-cyclic index computation
  - Most programmers use cyclic (BF=1), blocked (BF=N/Threads) or indefinite (all indices on 1 thread)
  - There has been talk of abolishing blocking factors
- **Internal consistency:**
  - Type casts are messy
  - This is the factor that trips up most UPC programmers

Performance:

Pointers-to-shared are the graves of performance

UPC pointer arithmetic

2 integer divs + 2 modulo ops /index

Mitigated by compiler in loops

Interoperability

UPC threads vs MPI tasks

UPC has two classes of objects. MPI does not.

Mixing UPC shared access with MPI 2-sided comm. leads to chaos.

UPC shared arrays not compatible with MPI communicators

Is any of PGAS going to make it?

1. The MPI forum needs to be involved

- PGAS language has to be “advice to users”
- We cannot invent yet another language and hope it sticks

2. Composability/interoperability is key

- New language has to feel like a natural extension of MPI
- No awkward matches, no incomplete fits
- Original vision of MPI was to ne RT library, enable compilers

3. The illusion of shared memory is valuable

- Need to come to grips with 2 memory populations

4. MPI has failed as a runtime library

- The real reason why IBM is working on PAMI

My own list of favorites (1 of 2)

- **Global view is a winner**
  - Real progress in last few years
  - Co-indices yes, blocking factors no, distributions yes
  - Do not go overboard with syntax! (HPF lurks)
  - By default let compiler deal with split phase assignment
- **Leave pointers-to-shared, split phase in the mix**
  - Like “goto”: considered harmful but necessary
- **Compiler deals with shared data**
  - MPI can touch any data anywhere

My list of favorites (2 of 2)

- **Lose Java. Keep C++/Python.** **Lose Fortran ...**
  - OO framework with remote method invocation
- **Build a strong, portable standard library**
  - It can make or break a language
  - People distrust compiler magic; willing to trust libraries