



On the Evolution of Communication in Parallel Systems

Marc Snir

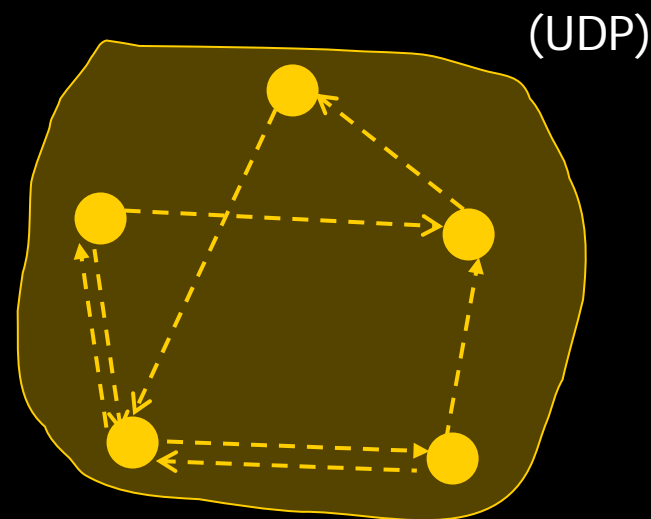
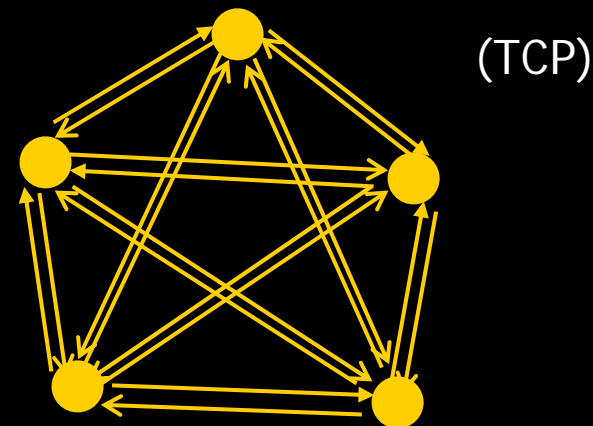
September 05

Focus of talk

- How do parallel programs express communication?
- Does this provide the communication hardware performance to the application?
- Does this fit application communication patterns?

60 second tutorial on communication protocols

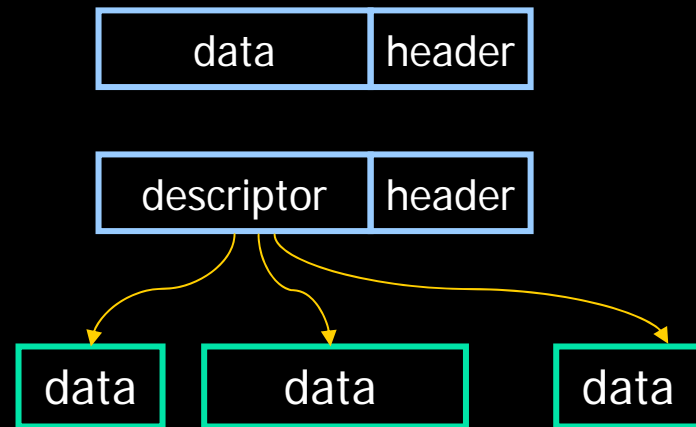
- Reliable vs. unreliable
 - need reliable
- User space vs. kernel space
 - parallel computing communication works better with user space
- Connection-oriented vs. connectionless
 - connection oriented does not scale well
 - connectionless needs congestion control
 - Less of a problem for parallel computing (?)



60 seconds advanced tutorial on communication protocols

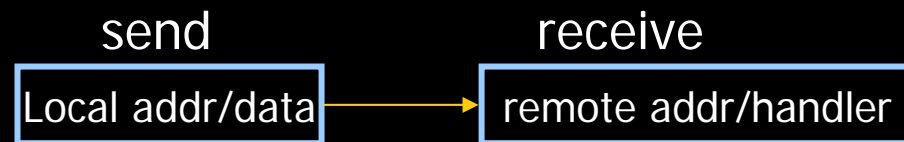
■ Immediate data vs. Buffer descriptor

- scatter/gather – how general?

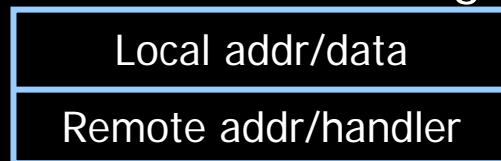


■ Two-sided vs. one-sided

- send-recv: need matching engine
- one sided: need smart comm coprocessor



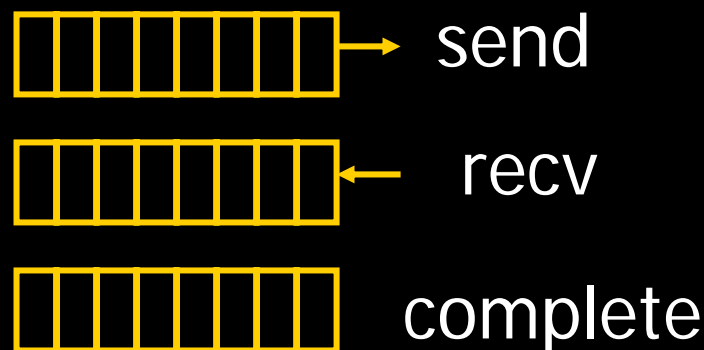
rDMA/active message



- Get (read)
- Put (write)
- Read-Modify-Write

Example: Infiniband

- Everything but the kitchen sink:
 - Reliable or unreliable
 - User space (or kernel)
 - Connectionless or connection-oriented
 - 2-sided or 1-sided
 - Immediate data or buffer descriptor
- 2507 pages standard
 - but no standard sw binding...
- Caveat: products do support only part of the standard
 - Usually not the parts needed for HPC...



- Send commands (1 or 2 sided) queued in send queue
- Recv commands (for 2 sided comm) queued in recv queue
- Sends matched to receives in FIFO order
- Entry posted in complete queue once command is consumed

MPI (1)

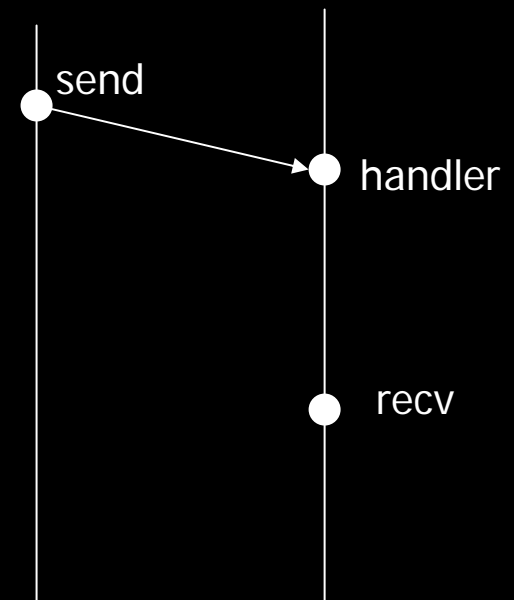
- Does MPI provide to parallel application the performance potential of Infiniband (or Quadrics, or Myricom, or...)?
- Does MPI express well common communication patterns?

MPI Performance

Why does MPI need $> 1,000$ instructions to transfer a byte from one processor to another?

- Not one reason: death by one thousands cuts;
 - the cost of generality

"Best case"



Short message protocol:
eager protocol

Where does the time go (send)

- `MPI_SEND(buf, count, datatype, dest, tag, comm)`
 - Check and interpret six parameters
 - Check for `MPI_PROC_NULL`
 - Check if data buffer is contiguous
 - Check for self-loop
 - Pick communication protocol according to message length
 - Allocate communication object
 - Initialize communication object
 - Invoke lower layer to push message (pass comm object)
 - Wait for lower layer completion
 - Free communication object

Where does the time go (handler)

■ Polling handler

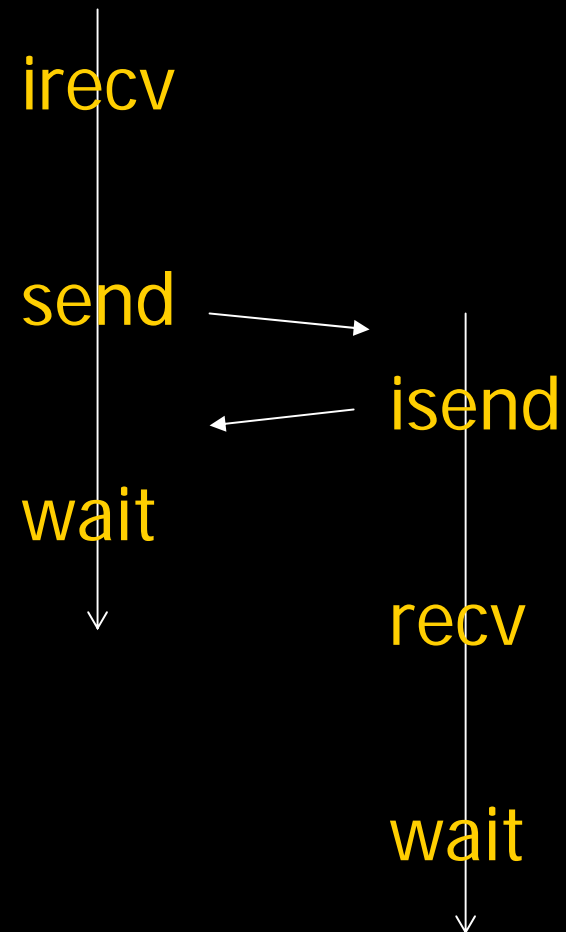
- invoke lower layer (pull message)
- wait for lower layer completion
- Unpack header
- If "eager send" then
 - Search queue of premature receives (linear search, 3 comparisons per item)
 - If not found then
 - allocate premature send object
 - initialize premature send object
 - enqueue in premature send queue

Where does the time go (recv)

- `MPI_RECV(buf, count, datatype, dest, tag, comm, status)`
 - Check and interpret seven parameters
 - Check for `MPI_PROC_NULL`
 - Check if data buffer is contiguous
 - Search queue of premature sends (linear search, 3 comparisons per item)
 - In found then
 - Dequeue object
 - Copy data to receive buffer
 - Set status object
 - Free object
 - Return

More complexities

- Locks, to ensure thread safety
- Side calls to polling handler to ensure progress
 - tradeoff on polling frequency
 - potential problem of endless recursion (in MPICH)
- MPI_CANCEL
- ...



MPI1 approach to performance

- Provide special case calls with lower overhead
 - Example: ready-mode send
 - can use eager send protocol for long messages (avoids one round-trip and two handler invocations)
 - Redefined as standard-mode send in MPICH

The classical vicious circle

Ready-mode does not seem efficient; I shall not spend time specializing my code to use this feature



Nobody uses Ready-mode; I shall not spend time on a faster implementation



MPI1 approach to performance (2)

- Example: persistent communication request
 - MPI_SEND_INIT, MPI_RECV_INIT, MPI_START
 - Saves the need to check and decode long parameter list
 - Can be (almost) used to create a channel and eliminate almost all MPI overhead (e.g., with Infiniband)



- Need to ensure that other receives cannot match the persistent send
- Need ready-mode, rather than standard mode

The classical screw-up

If we must speed-up MPI1...

- Shift some/all of MPI library code to communication co-processor
 - move queues management, matching and handling of unexpected sends to co-processor
 - Myrinet, Quadrics, Blue Gene\L (*)
 - offload main processor
 - saves context switches
 - use more specialized hw (network processor) and sw
- but NIC's are often behind main processor in raw speed

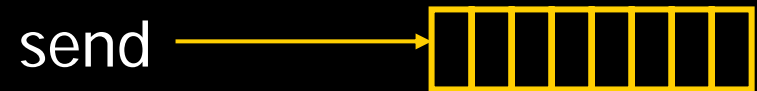
If we must speed-up MPI1 (2)

- Specialize and tune MPI code via preprocessor/compiler (or library designer if communication layer is encapsulated in library)
 - break the vicious circle...
- Need:
 - Local analysis (to inline, avoid parameter checking, preallocate objects...)
 - Global analysis (e.g., to replace standard mode with ready mode)
 - Recommended restricted programming style that avoids the “curse of generality”?
 - no premature sends, no dontcares...
 - Lower level, exposed communication layer

What should be this lower layer?

- Need two things:
 - Simple reliable datagram service
 - connectionless messaging
 - short messages received in strict arrival order
 - RDMA – remote put
- Strategy used for MPI on T3E (EPCC), Infiniband (Ohio), etc.
- Can achieve x4 reduction in sw overhead (IBM 96)
- Usually need to “fake” datagram service.
- Need to virtualize, for effective support of migration and load balancing

Eager protocol



Rendezvous protocol



Should one directly use rDMA?

- Communication:
 - matches src id, src addr, dest id, dest addr
- Send-recv:
 - source provides dest id, src addr; destination provides src id, dest addr; each side decides when transfer can occur on its side
- Put:
 - source provides all parameters and decides when transfer can occur on both sides
- Put can be used, rather than send-recv whenever
 - association of src address to dest address is persistent
 - synchronization is global and separated from communication
 - **This is a very frequent scenario!**

MPI2 One-sided

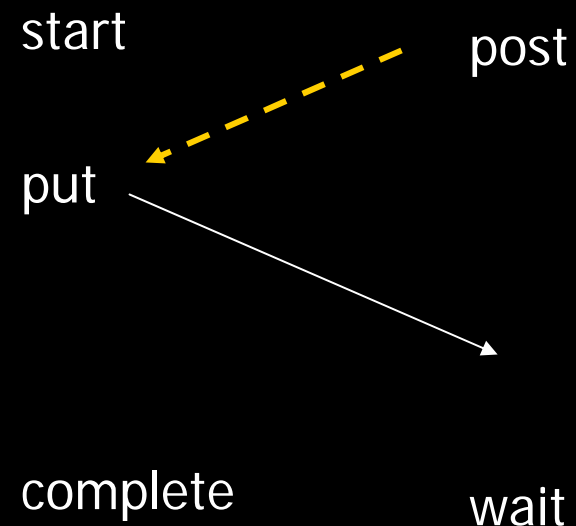
- Aimed at exploiting rDMA within MPI
 - PUT, GET, ACCUMULATE (similar to shmem)
 - Consistent with MPI syntax and semantics
- Often seems much slower than SHMEM on systems that have hardware supported rDMA [Luecke, Spanoyannis, Kraeva 2004]
 - up to x300 difference!

Issues with MPI2 one-sided

- Some difference due to more general interface
 - `MPI_PUT(origin_addr, origin_count, origin_datatype, target_rank, target_disp, target_count, target_datatype, win)`
 - `shmem_X_put(target, source, len, pe)`
 - could be avoided by preprocessing?
- Some (most?) difference due to lax implementation and obscurity of standard

Apparent inefficiency of MPI2 one-sided

- MPI2 requires that data not be put in remote memory before "post" executes
 - additional handshake (global barrier or rendezvous)
- Shmem moves this responsibility to the user
- MPI2 provides an option to bypass check (MPI_MODE_NOCHECK)
 - not used in paper comparing MPI2 to shmem!
 - either not implemented or not understood



Real MPI2 one-sided issues

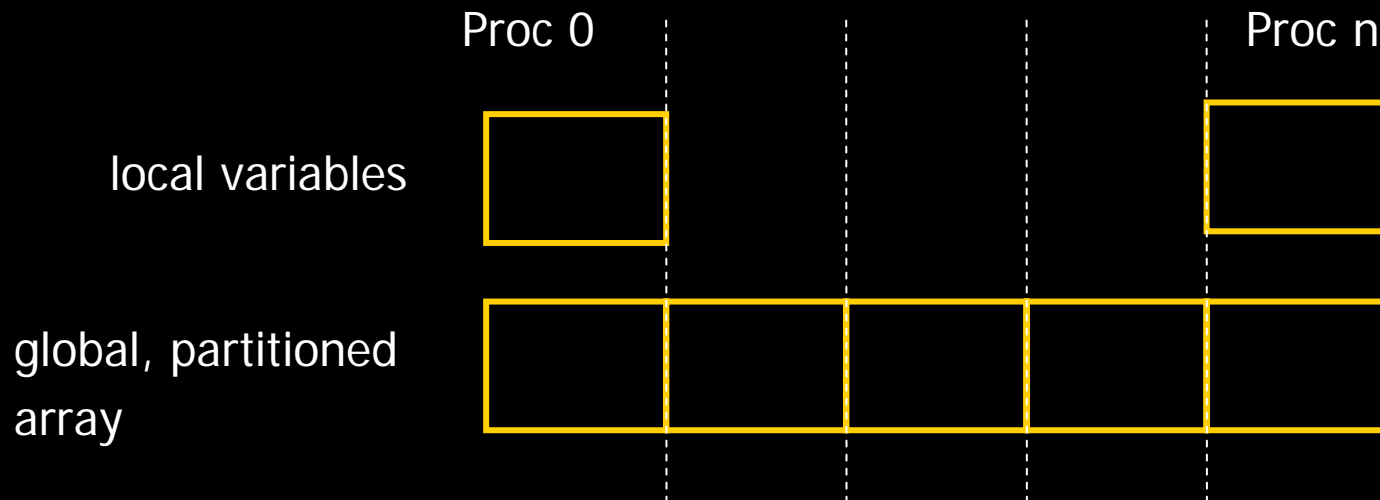
- Too complicated (hard to understand)
- No real fence call (MPI2 fence is a barrier)
- No yet implemented well
- Tried to accommodate too many requirements!
- **Time to reconsider?**
 - change default to `MPI_MODE_NOCHECK` – shift handshake to user code
 - provide true fence
 - restrict and simplify

Should communication be encapsulated in a library?

- ✓ A compiler can do many of the optimizations we mentioned
- ✗ Parallel languages have a bad reputation
 - Shared memory languages (e.g. OpenMP) perform badly on clusters
 - do not provide user control of communication
 - Distributed memory languages (e.g. HPF) never matured
 - HPF1 was too restrictive, HPF2 never happened
- Latest attempt: Partitioned Global Address Space (PGAS)
 - UPC, CAF, Titanium

PGAS Languages

- Fixed number of processes, each with one thread of control
- Global partitioned arrays that can be accessed by all processes
 - Global arrays are syntactically distinct
 - compiler generates communication code for each access
- Limited number of global synchronization calls



Co-Array Fortran

- Global array \equiv one extra dimension
 - `integer a[*]` - one copy of `a` on each process
 - `real b(10)[*]` - one copy of `b(10)` on each process
 - `real c(10)[3,*]` – one copy of `c(10)` on each process; processes indexed as 2D array
- SPMD
 - code executed by each process independently
 - communication by accesses to global arrays
 - split barrier synchronization

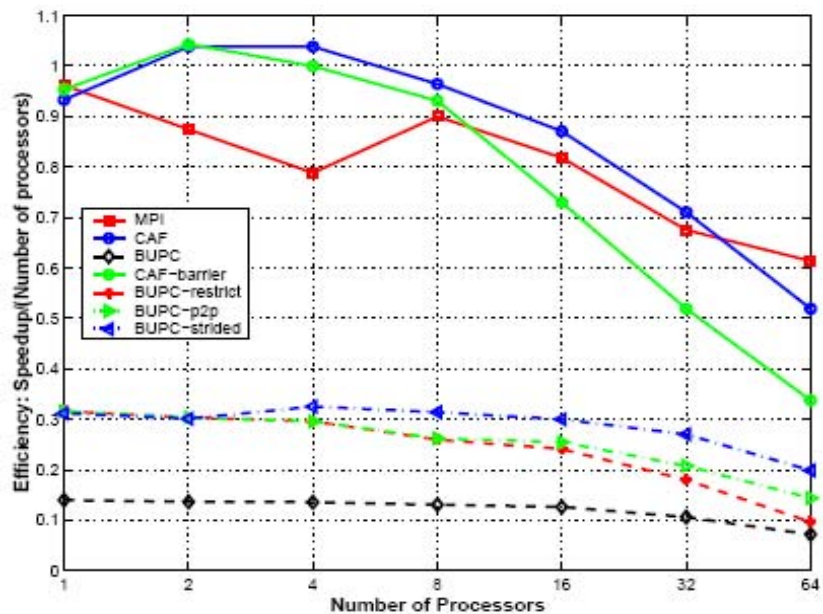
`notify_team(team)` `sync_team(team)`

Unified Parallel C

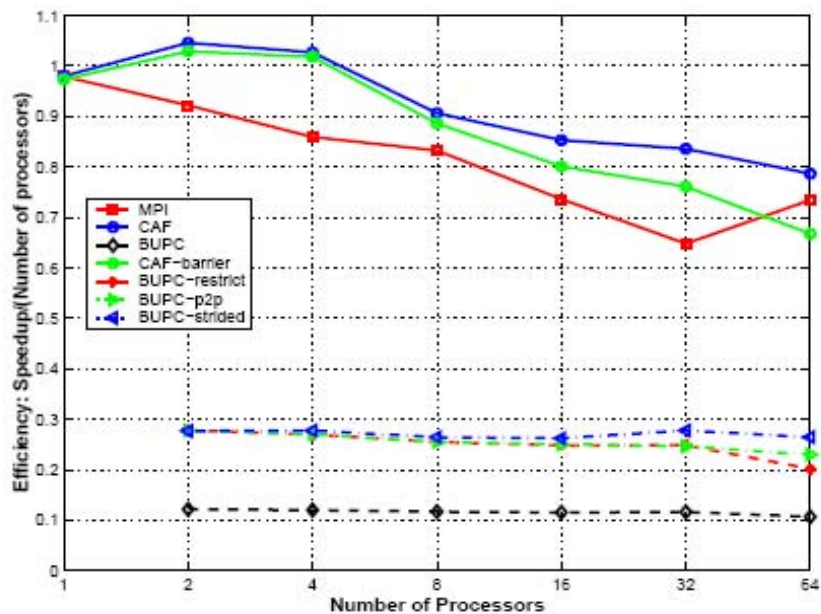
- (Static) global array is declared with qualifier **shared**
 - **shared int q[100]** – array of size 100 distributed round-robin
 - **shared [*] int q[100]** – block distribution
 - **shared [3] int q[100]** – block-cyclic distribution
 - **shared int* q** – local pointer to shared
- SPMD model
 - code executed by each process independently
 - communication by accesses to global arrays
 - global barrier or global split barrier
 - **upc_barrier, upc_notify, upc_wait**
 - simple **upc_forall**: each iteration is executed on process specified by affinity expression

Not too far from message-passing

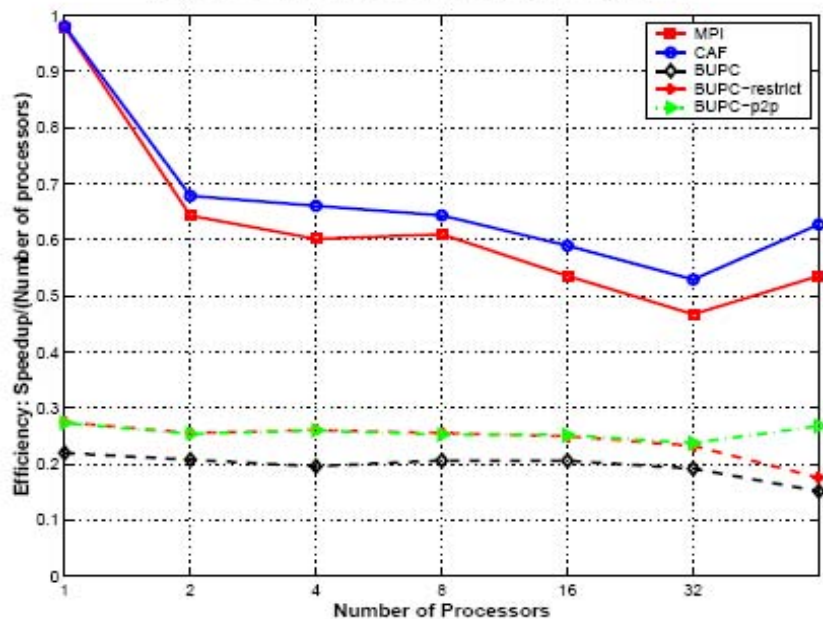
- MPI Fortran (resp. C) code with encapsulated communication layer can be recoded into CAF (resp. UPC) by recoding communication layer only
- Such code can achieve similar or better performance than MPI on NAS kernels [Coarfa, Dotsenko, Mellor-Crummey, Cantonnet, El-Ghazawi, Mohanti, Yao, Chavarria-Miranda, PPOPP June 05]



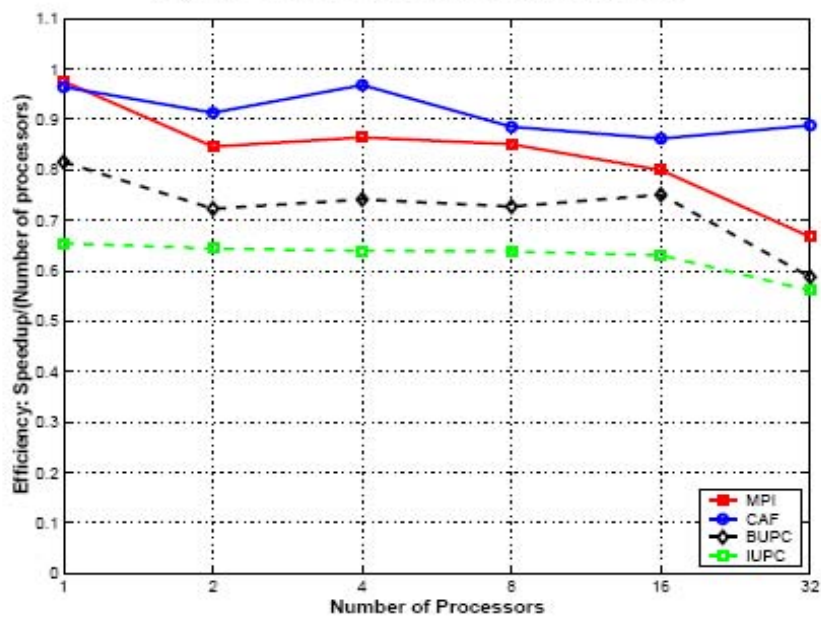
(a) MG class A on Itanium2+Myrinet



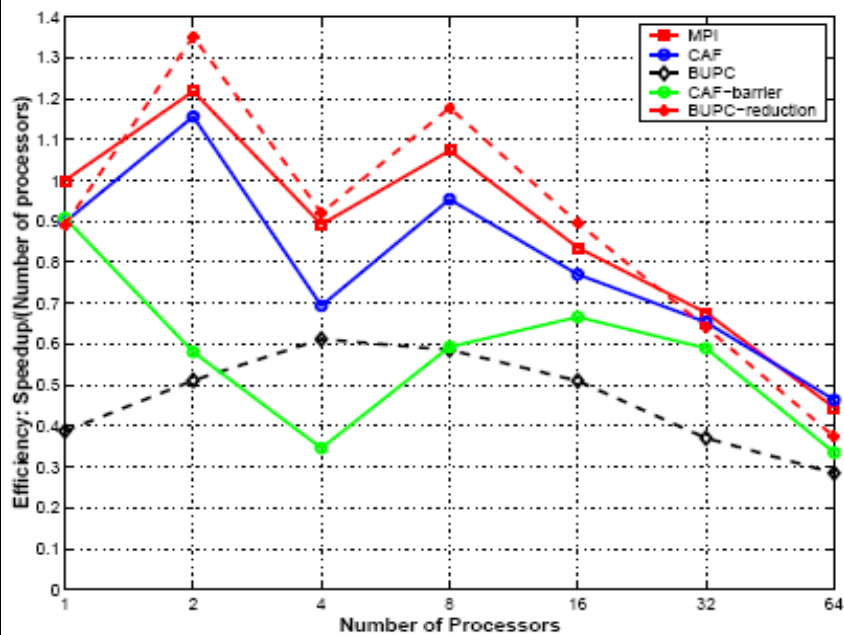
(b) MG class C on Itanium2+Myrinet



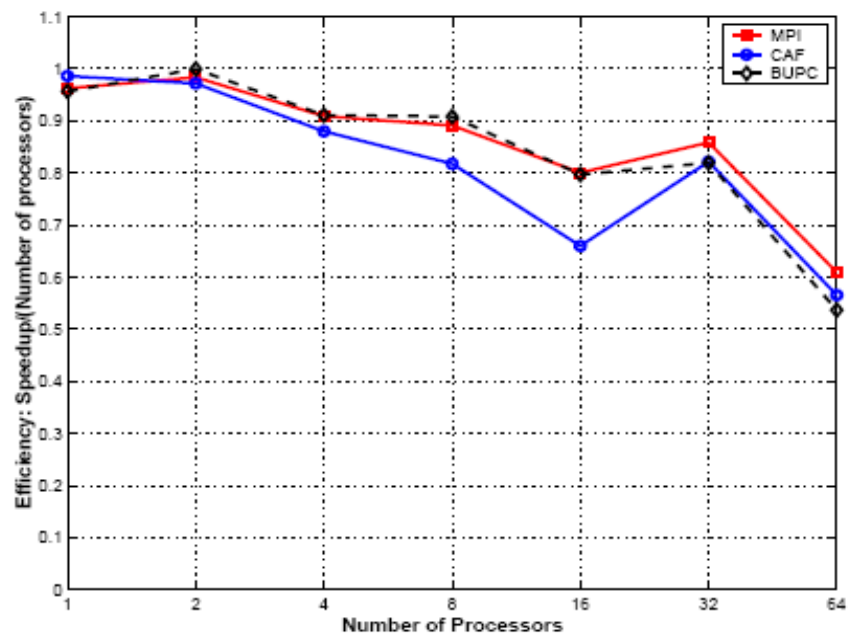
(c) MG class B on Altix 3000



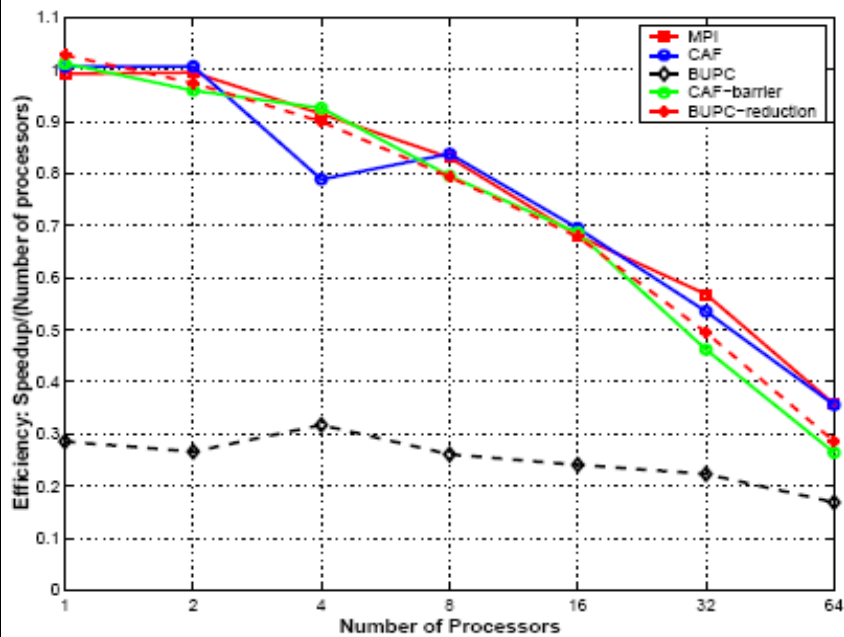
(d) MG class B on Origin 2000



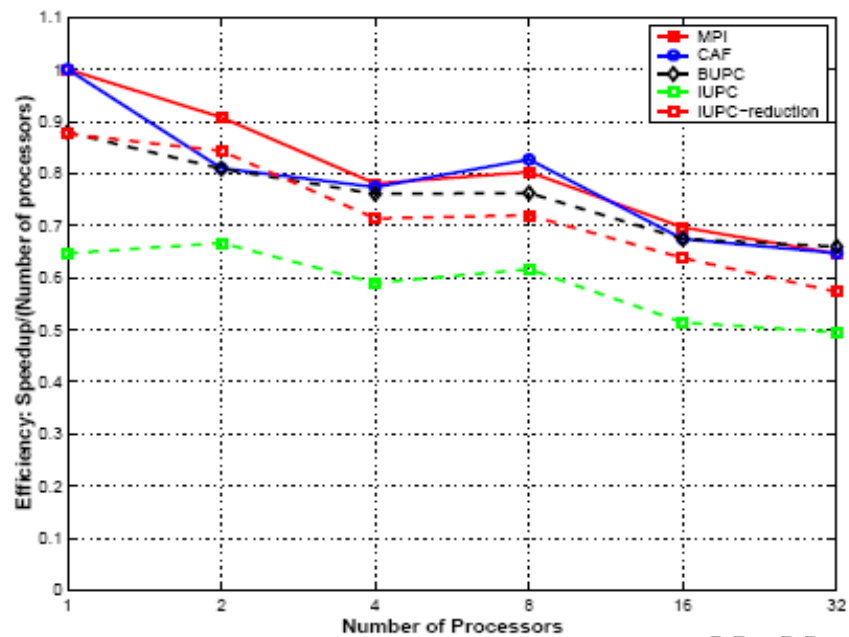
(a) CG class C on Itanium2+Myrinet



(b) CG class B on Alpha+Quadrics

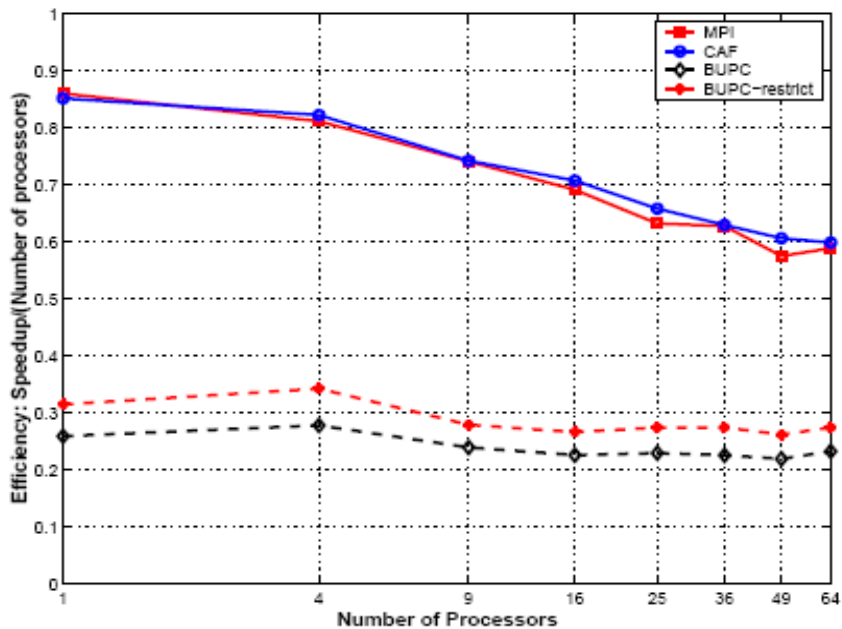


(c) CG class C on Altix 3000

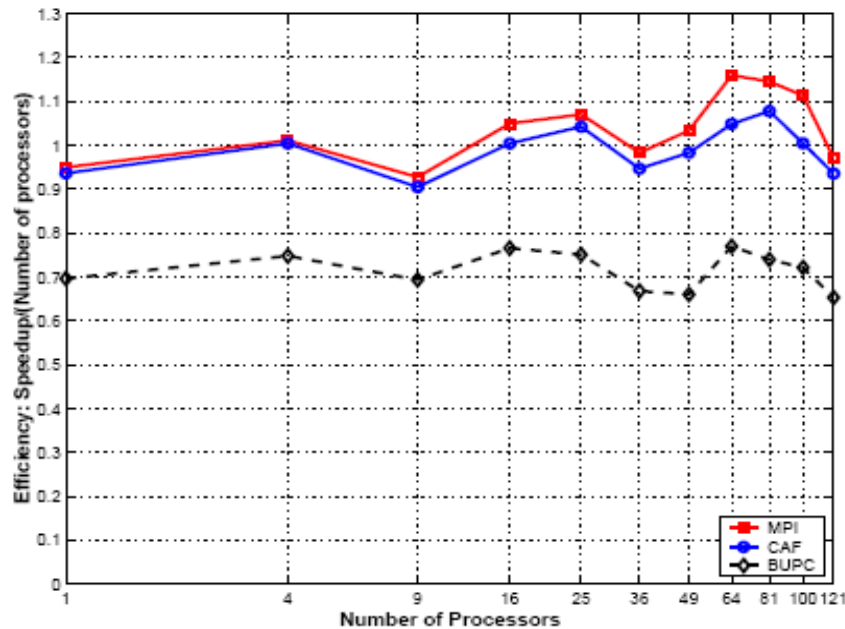


(d) CG class B on Origin 2000

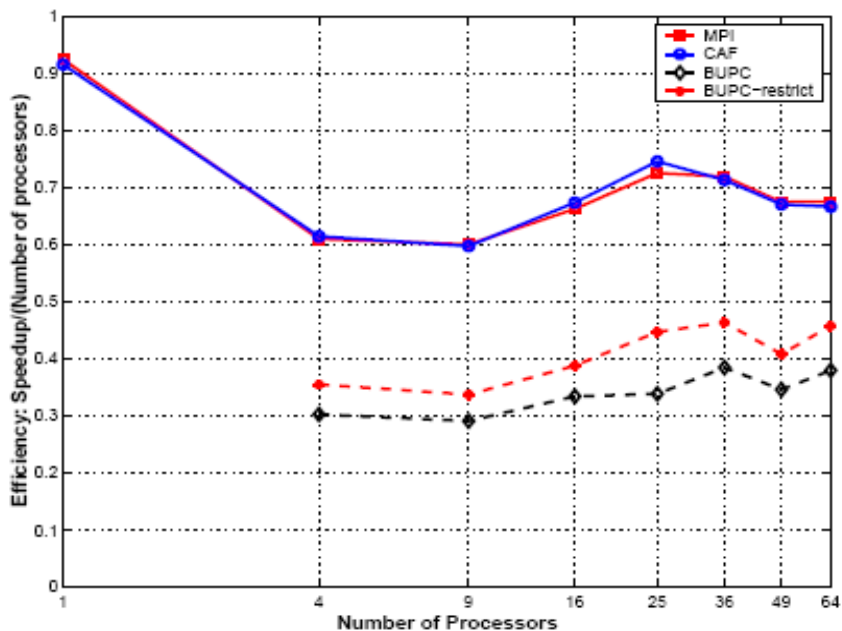
PPOPP05



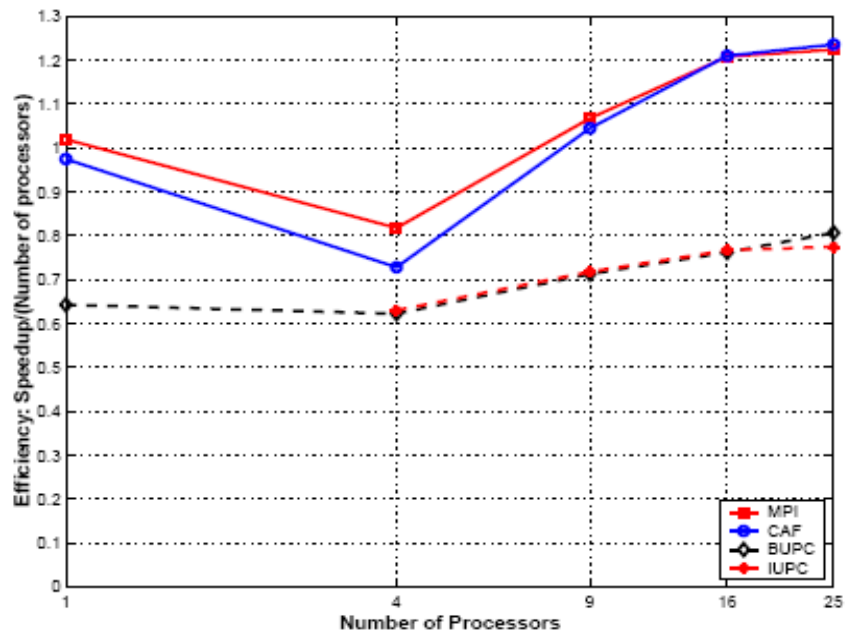
(b) SP class C on Itanium2+Myrinet



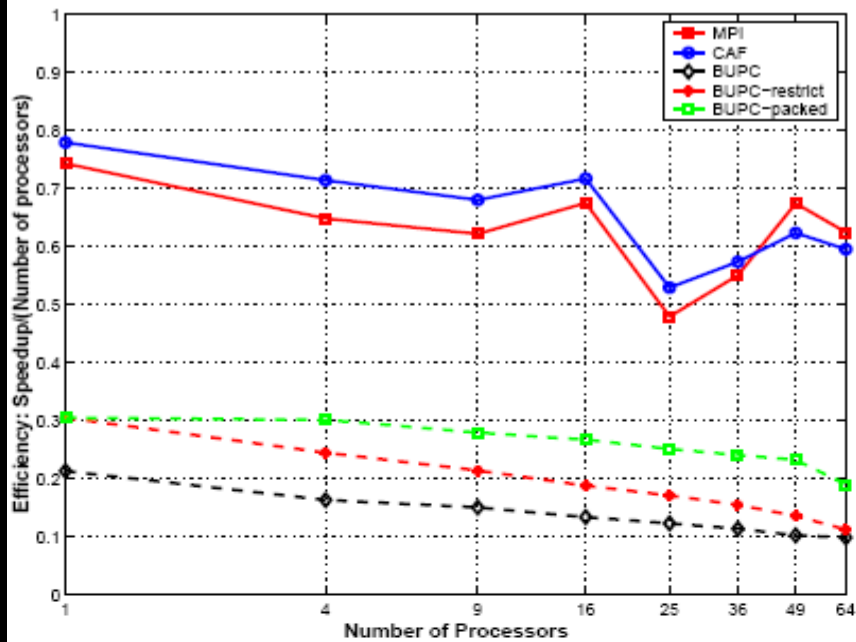
(a) SP class C on Alpha+Quadrics



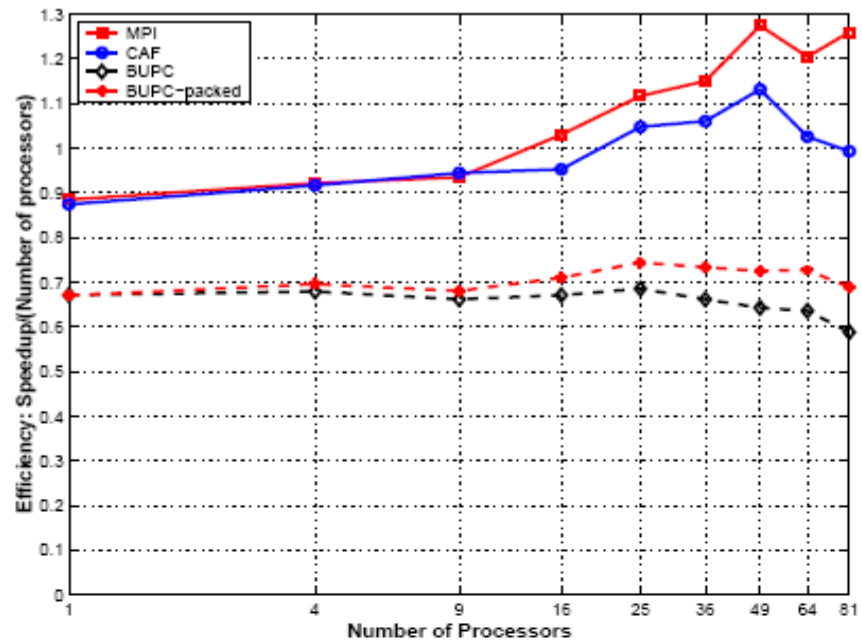
(c) SP class C on Altix 3000



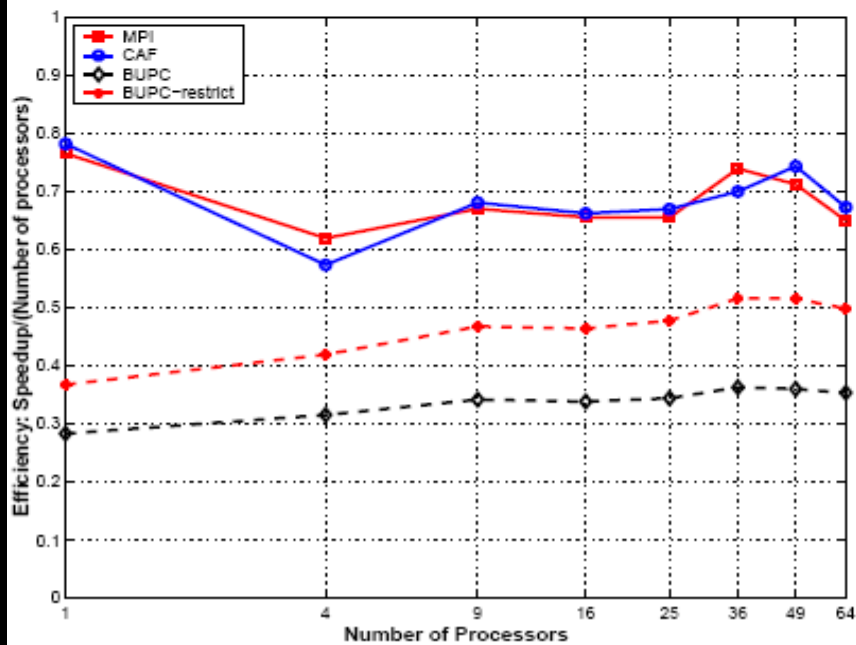
(d) SP class B on Origin 2000 PPOPP05



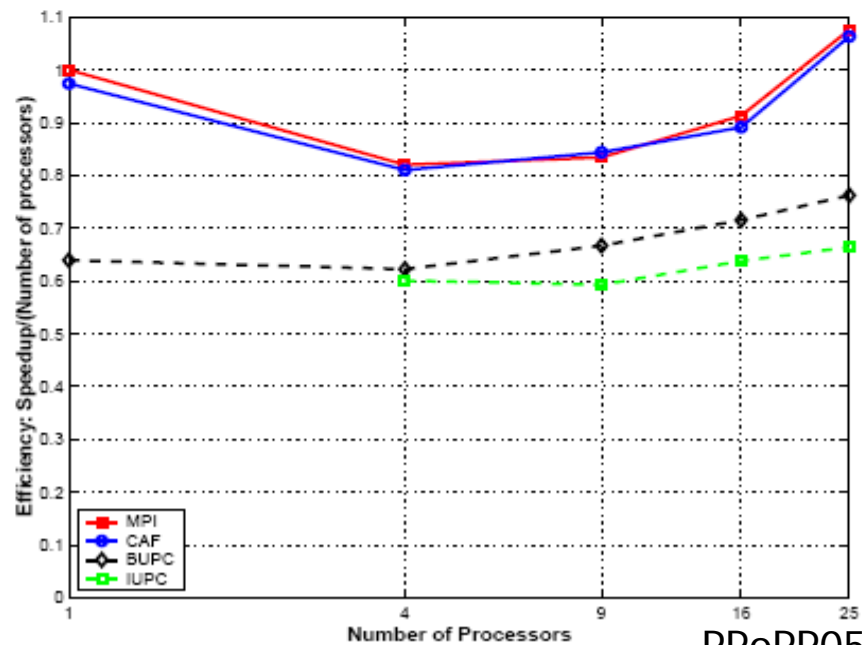
(a) BT class C on Itanium2+Myrinet



(b) BT class B on Alpha+Quadrics



(c) BT class B on Altix 3000



(d) BT class A on Origin 2000

Will MPI be replaced by PGAS languages?

- There is still work to be done
 - Simple to fix issues:
 - F90 array notation allows for bulk transfer, but UPC misses such notation
 - global synchronization too restrictive
 - compiler optimizations of communication not very sophisticated
 - communication coalescing, split-phase communication,...
 - More significant issues:
 - Not obvious what happens with more dynamic, irregular codes
 - **Both CAF and UPC need better encapsulation**
 - support for “communicators” (they only have “MPI_COMM_WORLD”)
 - support for OO

Meantime DARPA is forging ahead...

- High Productivity Computing Systems program:
Cray, IBM, Sun
- Chapel, X10, Fortress
 - Chapel: distributions (HPF) + control parallelism and atomic transactions (multithreading) + OO + generic programming
 - X10: Java + cluster memory model + remote asynchronous invocations + clocks + atomic blocks
 - Fortress: focus on abstraction & type inference
- Several more years of research are needed
- No concrete plans for convergence yet
- No portability/compatibility solutions
- Economic model is not obvious

Summary

- We can extract better performance from MPI
- We can fix MPI2, esp. one-sided, to improve usability
- PGAS languages are not yet ready for prime-time, but could get there in a few years
 - will improve performance, but will not significantly change programming model
- The HPCS languages could be a significant game changer
 - but will not happen for a while

