IDA

Conveyors, an Abstraction for Message Aggregation

Bob Lucas[1,2], and Miller Maley[3]

[1]Institute for Defense Analyses/Center for Computing Sciences

[2]Livermore Software Technology, an Ansys company

[3]Institute for Defense Analyses/Center for Communications Research

Nov. 23, 2020

Institute for Defense Analyses

4850 Mark Center Drive ● Alexandria, Virginia 22311-1882

Collaborators

Bill Carlson, IDA/CCS

John Gilbert, UC Santa Barbara

Phil Merkey, IDA/CCS

Dan Pryor, IDA/CCS

Vivek Sarkar, Georgia Tech

Kathy Yelick, UC Berkeley

Outline

- Motivation
  - Sparse communication patterns are ubiquitous
  - Poor performance of small messages
    - Both MPI and PGAS (Parallel Global Address Space)
- Familiar message aggregation strategies
  - Collectives
  - Asynchronous messages
- Conveyors

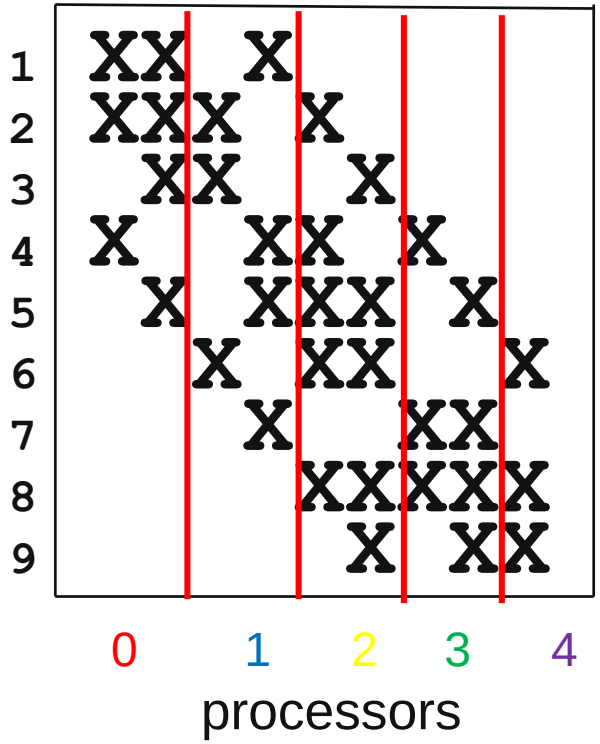
Sparse communication is ubiquitous

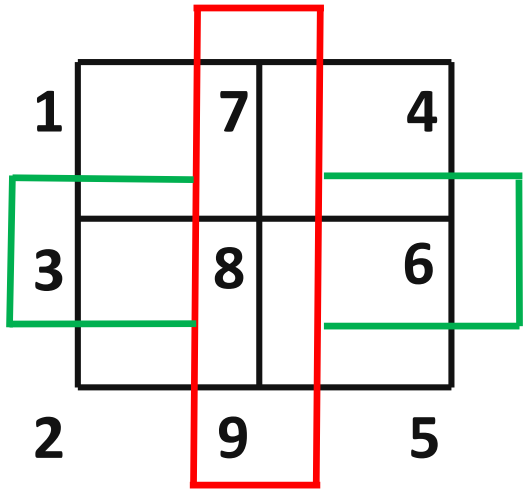
- Reordering for sparse matrix factorization
  - ParMetis “splits the graph” when it recurses
- Symbolic Analysis
  - Permute and redistribute the graph for concurrent analysis of the domains and separators
- Permute and redistribute the K and M matrices
  - One-time work at the start of a block shift invert Lanczos eigensolver
- Perform a multifrontal factorization
  - Scatter contribution blocks into the processors holding their parent’s frontal matrices
- Triangular solves
  - Subsets of the right-hand-side vector are exchanged between processors

Reordering and symbolic analysis

Nested dissection of toy problem

- User supplies columns of lower triangle on arbitrary processors
- ParMetis-style distribution

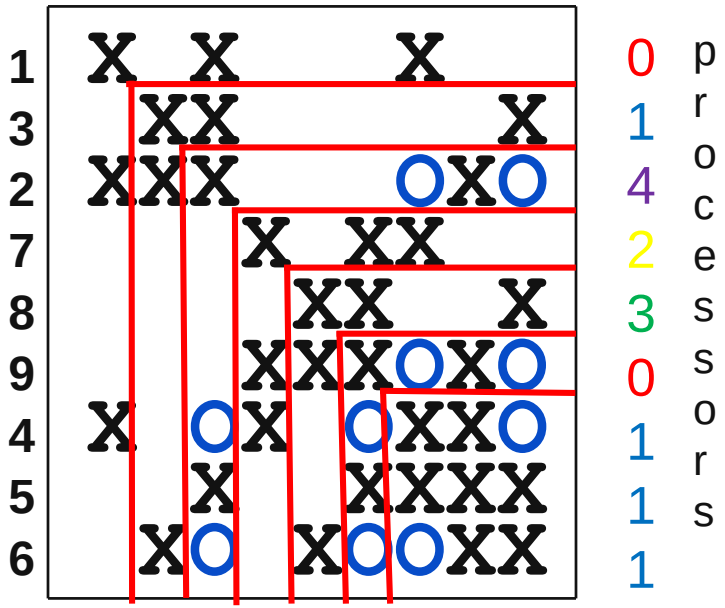




graph after two levels of nested dissection

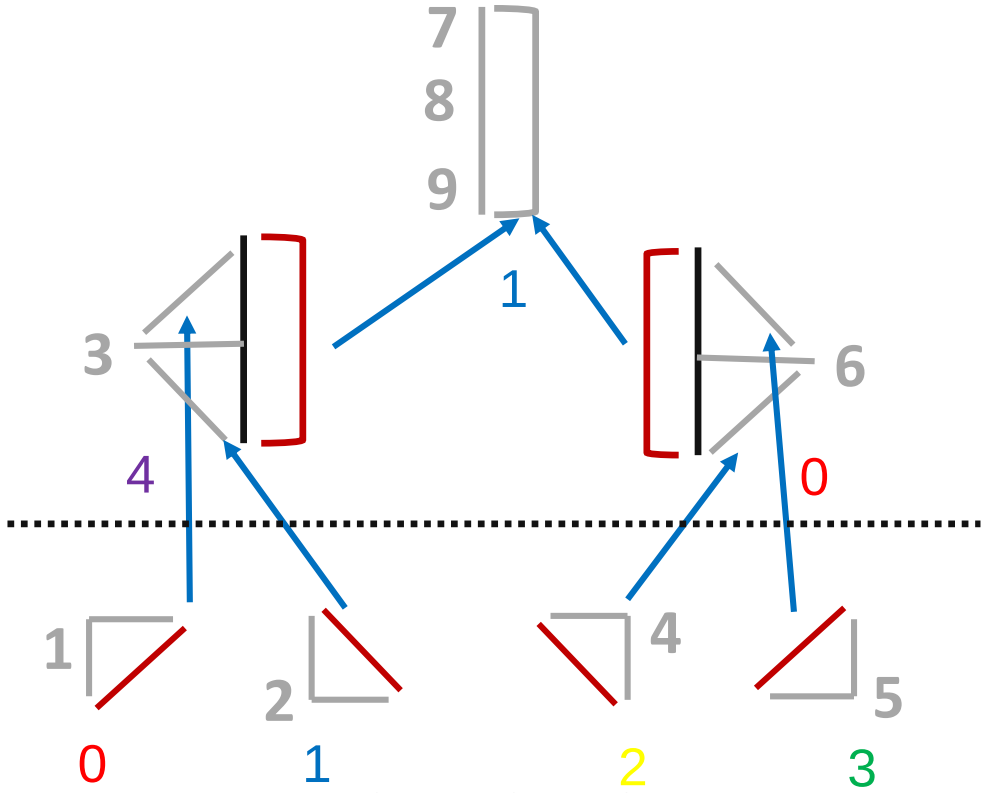
Reordering and symbolic analysis

Graph permuted and redistributed for parallel symbolic analysis with five processors



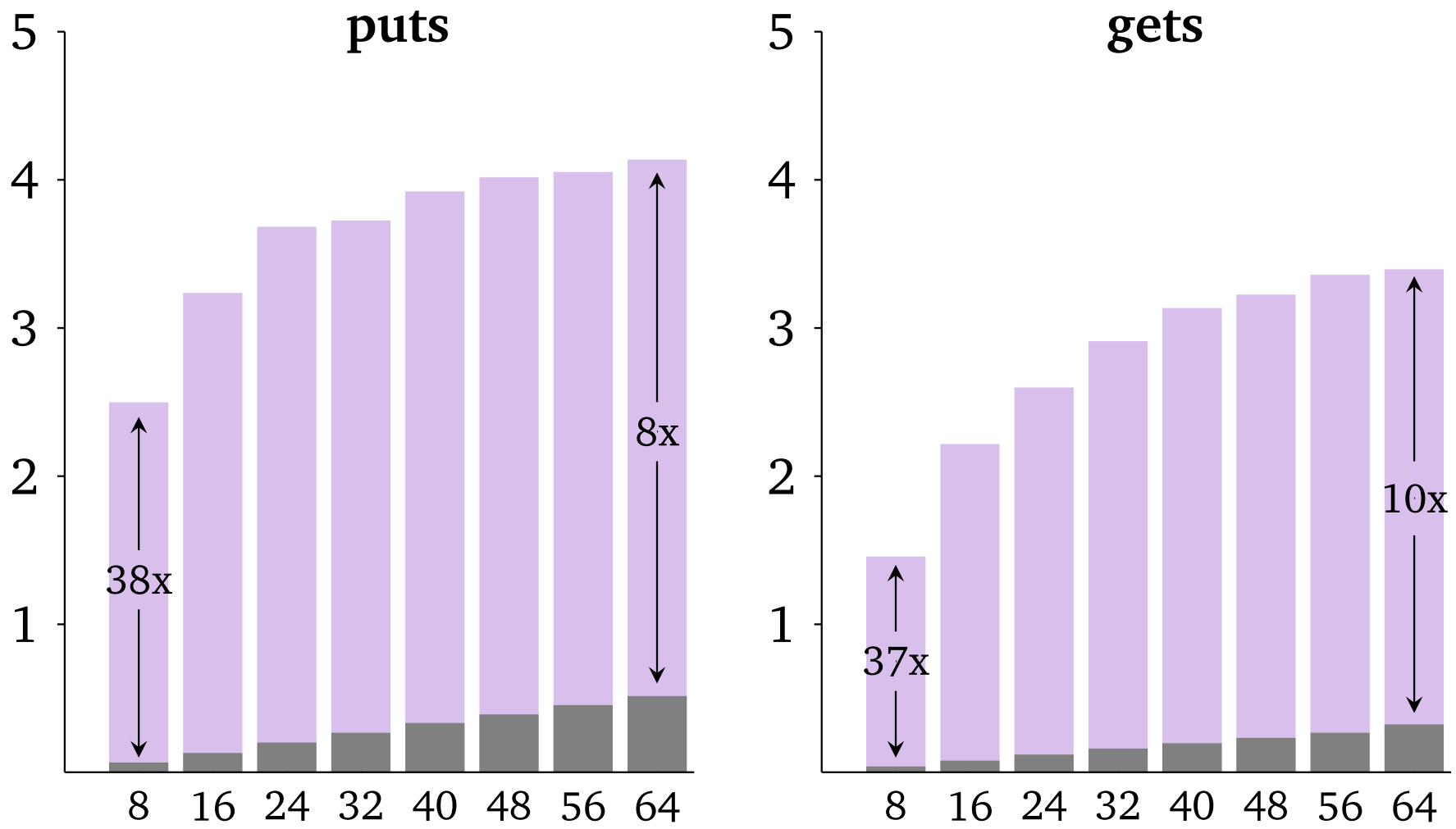
Chevrons for concurrent analysis of domains and separators

Elimination tree with three separators



Four independent domains

Messages with small payloads are slow



Payload bandwidth (GB/sec.) vs. size (bytes)
128 nodes, FDR Infiniband, 32 processes per node

MPI collective approach for redistributing a matrix

Traverse my subset of the input matrix

count those values that are needed by other processors

```
call MPI_ALLTOALL (send_cnt, 1, MPI_INTEGER,
.                  recv_cnt, 1, MPI_INTEGER,
.                  communicator, error)
```

Compute displacements and allocate send and receive buffer storage

Allocate additional storage (initial matrix, send_buf, recv_buf, and permuted matrix)

Load coefficient tuples (row,column,value) into send buffer

```
call MPI_ALLTOALLV (send_buf, send_cnt, send_dsp, MPI_INTEGER,
.                   recv_cnt, recv_cnt, recv_dsp, MPI_INTEGER,
.                   communicator, error)
```

Unpack tuples to create permuted matrix

MPI asynchronous approach for redistributing a matrix

- Initialize send and receive buffers for other processors
  - call MPI_IRECV() to initialize the receive buffers
- Traverse my subset of the matrix
  - Determine who needs each coefficient
  - If not mine and the send buffer is available then
    - append this row and column
    - If its now full, call MPI_ISEND
  - else
    - Call MPI_WAITANY
    - If I received a message
      - unpack and store it
    - Loop back to check send buffer availability
  - end if
- Flush my send buffers
- Receive any last messages from the other processors
- Wait until the final sends complete

Asynchronous initialization

```
c
c     --  Begin by initializing the communication buffers.
c
      call MPI_Comm_Rank(world, mid, error)
      if (error .gt. 0) return
c
      do pe = 0, npes - 1
        send_cnt(pe)       = 0
        request(pe)        = MPP_REQUEST_NULL
        request(npes + pe) = MPP_REQUEST_NULL
        if (pe .ne. mid) then
          all_done    = .false.
          pe_done(pe) = .false.
          call MPI_IRecv(ibuf(1, 1, pe),    buf_len,
     1                   MPP_FLT,           pe,
     2                   0,                 world,
     3                   request(npes + pe), error)
        else
          pe_done(pe) = .true.
        end if
       end do
```

Main loop

```
      do j = 1, my_neq
        c = invi(my_map(j))
        do i = jk(j), jk(j + 1) - 1
          r = invi(ik(i))
          if (perm(r) .gt. perm(c)) then
            v = c
          else
            v = r
            r = c
          end if
          pe  = objOwn(location(v))
          if (pe .eq. mid) then
            p      = ptr(v)
            ia(p)  = r
            ptr(v) = p + 1
          else
10          if (request(pe .ne. MPI_COMM_NULL) then
            else
            end if
          end if
        end do
      end do
```

```
1     call MPI_Wait_Any((2 * npes), request, indx,
                        status,     error)
      if (error .ne. 0) return
      op = indx - 1
      if (op .lt. npes) then
        request(op) = MPP_REQUEST_NULL
      else
        op = op - npes
        call MPI_Get_Count(status, MPP_FLT,
1                          recvd,  error)
        if (error .ne. 0) return
        recvd = recvd / 3
        do k = 1, recvd
          call mf2UnTpl(ibuf(1, k, op), r2, c2, x)
          if (r2 .gt. 0) then
            p       = ptr(c2)
            ia(p)   = r2
            ptr(c2) = p + 1
          else
            pe_done(op) = .true.
          end if
        end do
        request(npes + op) = MPP_REQUEST_NULL
        if (.not. pe_done(op)) then
          call MPI_IRecv(ibuf(1, 1, op),     buf_len,
1                        MPP_FLT,            op,
2                        0,                  world,
3                        request(npes + op), error)
        end if
      end if
      go to 10
```

```
      sp = send_cnt(pe) + 1
      call mf2PkTpl(obuf(1, sp, pe), r, v, zero)
      if (sp .ge. tpl_len) then
        call MPI_ISend(obuf(1, 1, pe), buf_len,
                       MPP_FLT,        pe,
1                      0,              world,
2                      request(pe),    error)
3       sp = 0
      end if
      send_cnt(pe) = sp
```

Flush send buffers

```
      do pe = 0, npes - 1
        if (pe .ne. mid) then
30        if (request(pe) .ne. MPP_REQUEST_NULL) then
            call MPI_Wait_Any((2 * npes), request, indx,
     1                        status,     error)
            if (error .ne. 0) return
c
            op = indx - 1
            if (op .lt. npes) then
              request(op) = MPP_REQUEST_NULL
            else

            end if
            go to 30
          end if
          sp = send_cnt(pe) + 1
          r  = - 1
          c  = - 1
          call mf2PkTpl(obuf(1, sp, pe), r, c, zero)
c
          call MPI_ISend(obuf(1, 1, pe), (multplr * sp),
     1                   MPP_FLT,        pe,
     2                   0,              world,
     3                   request(pe),    error)
        end if
      end do
```

```
          op = op - npes
          call MPI_Get_Count(status, MPP_FLT,
     1                       recvd,  error)
          if (error .ne. 0) return
c
          recvd = recvd / 3
          do i = 1, recvd
            call mf2UnTpl(ibuf(1, i, op), r, c, x)
            if (r .gt. 0) then
              p      = ptr(c)
              ia(p)  = r
              ptr(c) = p + 1
            else
              request(npes + op) = MPP_REQUEST_NULL
              pe_done(op)        = .true.
            end if
          end do
c
          if (.not. pe_done(op)) then
            call MPI_IRecv(ibuf(1, 1, op),     buf_len,
     1                     MPP_FLT,            op,
     2                     0,                  world,
     3                     request(npes + op), error)
          end if
```

Outstanding receives

```
40    continue
        all_done = .true.
        do pe = 0, npes - 1
          if (.not. pe_done(pe)) all_done = .false.
        end do
       if (.not. all_done) then
        call mf2WaitAny((2 * npes), request, indx,
     1                  status,     error)
        if (error .ne. 0) return
c
        op = indx - 1
        if (op .lt. npes) then
          request(op) = MPP_REQUEST_NULL
        else

        end if
        go to 40
       end if
```

```
        op = op - npes
        call mf2GetCount(status, MPP_FLT,
     1                   recvd,  error)
        recvd = recvd / multplr
        do i = 1, recvd
          call mf2UnTpl(ibuf(1, i, op), r, c, x)
          if (r .gt. 0) then
            p      = ptr(c)
            ia(p)  = r
            ptr(c) = p + 1
          else
            request(npes + op) = MPP_REQUEST_NULL
            pe_done(op)        = .true.
          end if
        end do
c
        if (.not. pe_done(op)) then
          call mf2IRecv(ibuf(1, 1, op),     buf_len,
     1                  MPP_FLT,            op,
     2                  0,                  world,
     3                  request(npes + op), error)
        end if
```

Wait for outstanding sends

```
call MPI_Wait_All(npes, request, status, error)
if (error .gt. 0) return
```

Conveyors

- The journey at IDA
  - Shared memory programming model
    - Unified Parallel C on Cray T3E, “we were happy”
  - Shared memory programming on clusters
    - Distributed memory platforms with multicore nodes
    - UPC starts mimicking MPI, e.g., upc_allreducev
      - Programmers read and write blocks of shared memory
      - I consider this an abomination
  - Repeated my journey
    - Created a collectives-based communication library
    - Created an asynchronous communication library
- Abstraction and library for message aggregation
  - The conveyor library, authored by Miller Maley, was motivated by earlier work at IDA/CCS
    - Jason DeVinney, Phil Merkey, and Dan Pryor
  - Bill Carlson has also authored a Rust version

Conveyors API

Collective begin and reset functions

```
convey_begin (handle, packet size)
convey_reset (handle)
```

Local push, pull, and advance functions

```
convey_push (handle, object, destination)
convey_pull (handle, object, source)
convey_advance (handle, halt)
```

Matrix permutation with fake a Fortran conveyor, main loop

```
do j = 1, neq
  c = invi(my_map(j))
  do i = jk(j), jk(j + 1) - 1
    r = invi(ik(i))
    if (perm(r) .gt. perm(c)) then
      v = c
    else
      v = r
      r = c
    end if
    pe  = objOwn(location(v))
    if (pe .eq. mid) then
      p      = ptr(v)
      ia(p)  = r
      ptr(v) = p + 1
    else


    end if
  end do
end do
```

```
do while (.not. mf2CnvyPush(pe, r, v, x))
   if (c_status .ne. CONVEY_DONE) then
     do while (mf2CnvyPull(r2, c2, x2, c_status))
       p       = ptr(c2)
       ia(p)   = r2
       ptr(c2) = p + 1
     end do
   end if
 end do
```

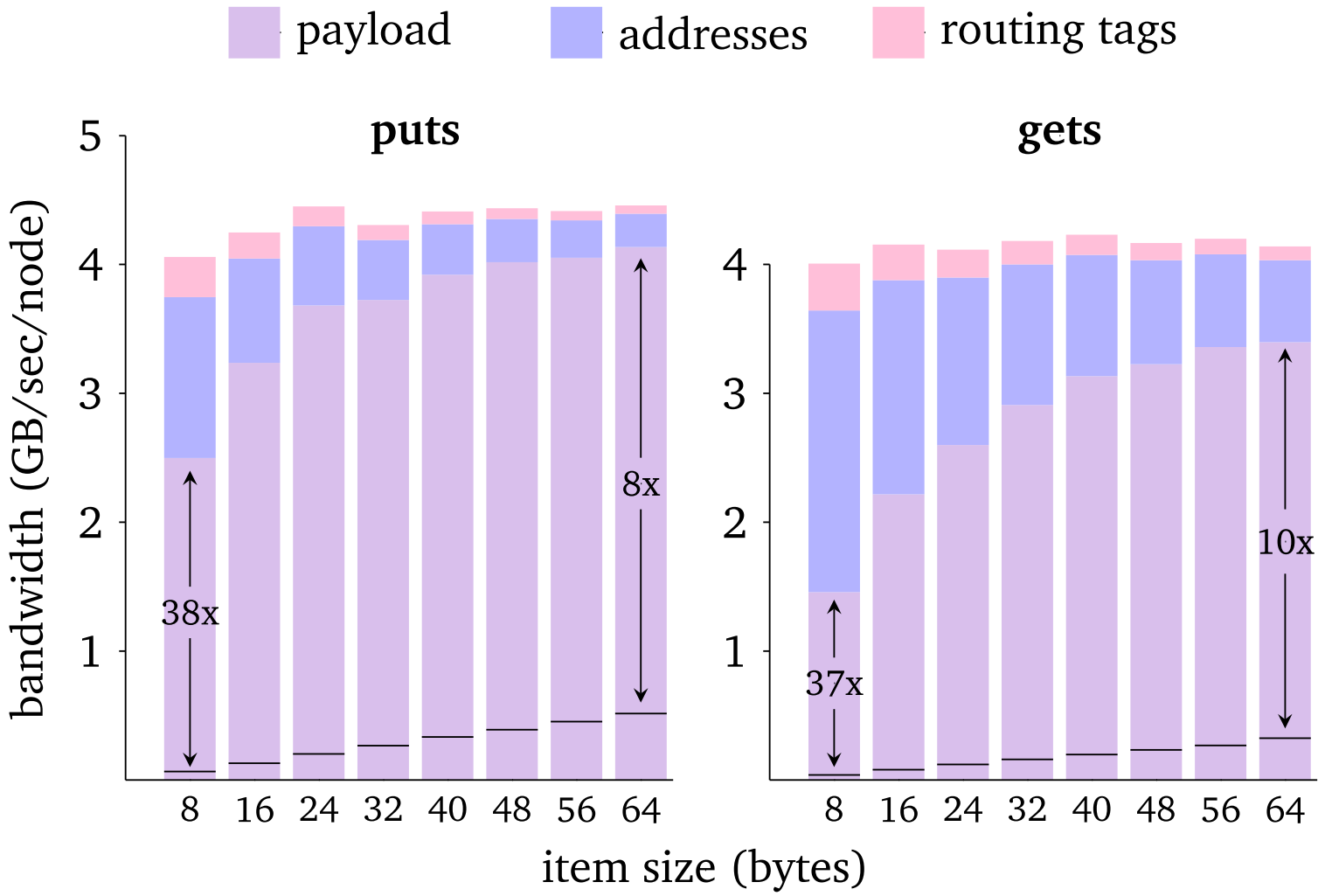
Clean up revisited (fake Fortran conveyor)

```
c
c       --  Flush.
c
        call CnvyAdvance (error)
        if (error .gt. 0) return
c
c       --  Receive any outstanding values from others.
c
        do while (c_status .ne. CONVEY_DONE)
          if (CnvyPull(r, c, x, c_status)) then
            p      = ptr(c)
            ia(p)  = r
            ptr(c) = p + 1
          end if
        end do
```

IDA's high-performance implementation

- Simple conveyors (ISEND/IRECV) scale poorly
  - Buffer size scales with the number of processors
- Use multi-hop conveyor communication pattern
  - Exploits fast intranode communication for local targets
  - Multi-hop algorithm for remote targets
    - Hop to unique local processor that talks to the target node
    - Hop across the network to the target node
    - Hop locally to target processor
  - Same as Priest, Steil, Sanders, Pearce, "You've got mail (YGM)", IPDPSW, 2019
- With P processors on each of N nodes:
  - Buffer size is $O(\sim 8P + 4N/P^2)$
- Message size is autotuned

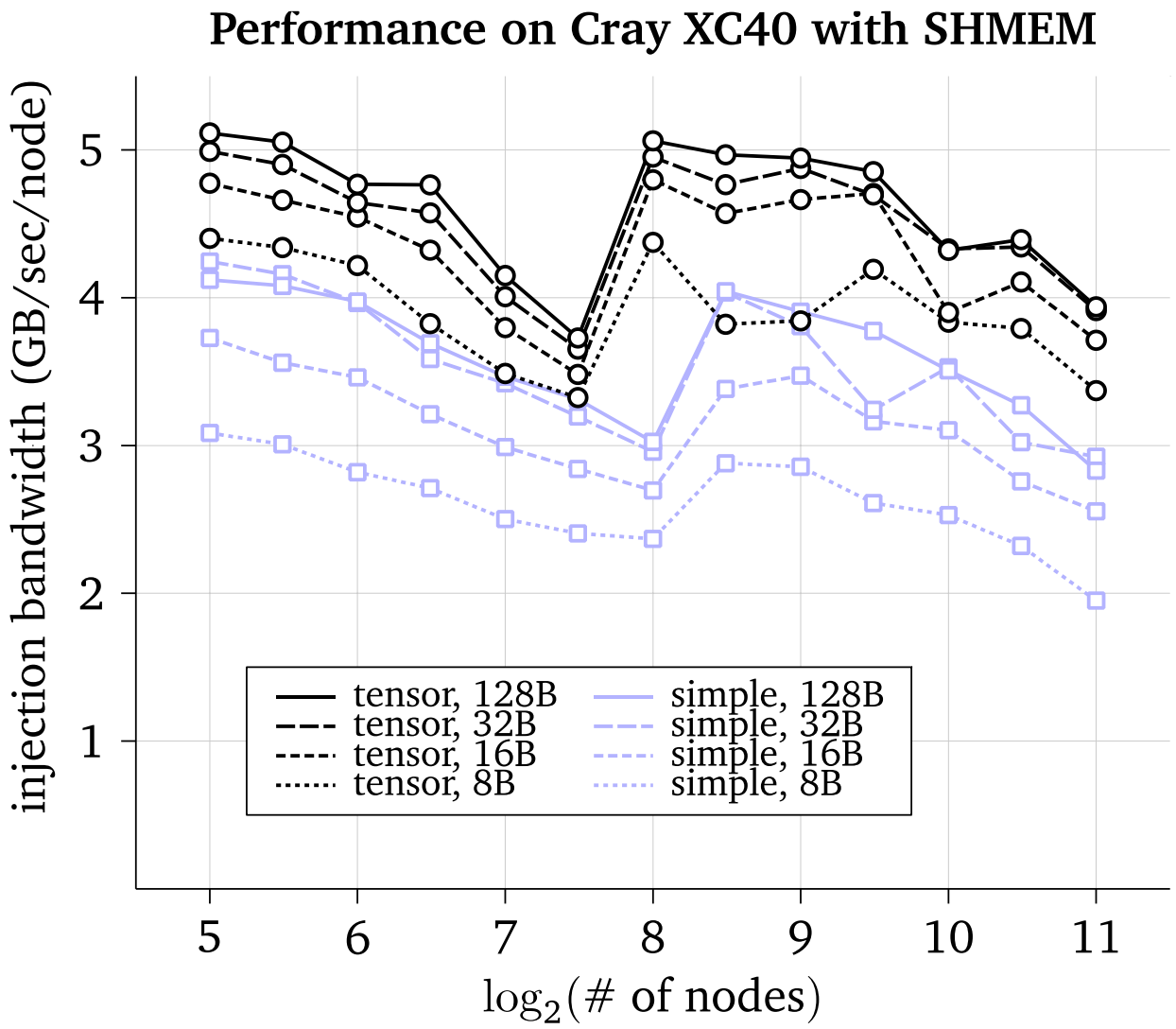
Put/Get microbenchmark



Payload bandwidth (GB/sec.) vs. size (bytes)
128 nodes, FDR Infiniband, 32 processes per node

Bandwidth versus payload size

- Simple conveyors here are the equivalent to my Fortran conveyors
  - SHMEM instead of MPI
- Tensor conveyors are production quality implementation
  - On github



Summary

- Conveyors provide a convenient abstraction for message aggregation
  - Probably improves code performance
  - Certainly improves programmer productivity
- A high-performance, open source implementation is available
  - jdevinney/bale on github
  - Can run on top of UPC, SHMEM, and MPI
- Ongoing research project, so it'll only get better