MPI-2
Remote Memory Access

Based on notes by Sathish Vadhiyar, Rob Thacker, and David Cronk

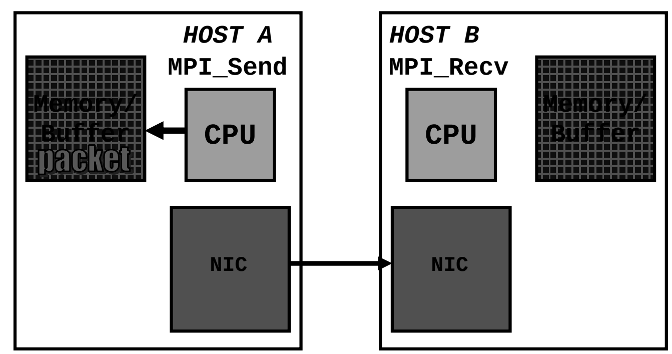
One Sided Communication

- One sided communication allows shmem style gets and puts
- Only one process need actively participate in one sided operations
- With sufficient hardware support, remote memory operations can offer greater performance and functionality over the message passing model
- MPI remote memory operations do not make use of a shared address space
- One sided comms are sensitive to OS/machine optimizations though

One Sided Communication

- By requiring only one process to participate, significant performance improvements are possible
  - No implicit ordering of data delivery
  - No implicit synchronization
- Some programs are more easily written with the remote memory access (RMA) model
  - Global counter

Standard message passing



Packet transmission is directly mitigated by the CPU's on both machines, multiple buffer copies may be necessary

Traditional message passing

- Both sender and receiver must cooperate
  - Send needs to address buffer to be sent
  - Sender specifies destination and tag
  - Recv needs to specify it's own buffer
  - Recv must specify origin and tag
- In blocking mode this is a very expensive operation
  - Both sender and receiver must cooperate and stop any computation they may be doing

Sequence of operations to `get' data

- Suppose process A wants to retrieve a section of an array from process B (process B is unaware of what is required)
  - Process A executes MPI_Send to B with details of what it requires
  - Process executes MPI_Recv from A and determines data required by A
  - Process B executes MPI_Send to A with required data
  - Process A executes MPI_Recv from B…
- 4 MPI-1 commands
- Additionally process B has to be aware of incoming message
  - Requires frequent polling for messages – potentially highly wasteful

Even worse example

- Suppose you need to read a remote list to figure out what data you need – sequence of ops is then:

```
Process A                       Process B
MPI_Send  (get list)           MPI_Recv (list request)
                               MPI_Send (list info)
MPI_Recv  (list returned)
MPI_Send  (get data)           MPI_Recv (data request)
                               MPI_Send (data info)
MPI_Recv  (data returned)
```


One Sided Communication

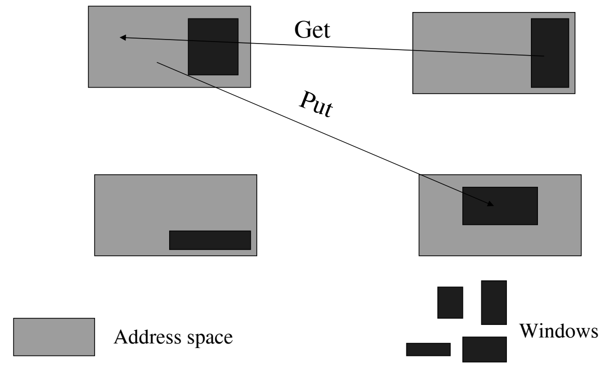
- RMA operations require 3 steps

1. Define an area of memory that can be used for RMA operations (window)
2. Specify the data to be moved and where to move it
3. Specify a way to know the data is available

One Sided Communication

- Memory Windows
  - A memory window defines an area of memory that can be used for RMA operations
  - A memory window must be a contiguous block of memory
  - Described by a base address and number of bytes
  - Window creation is collective across a communicator
  - A window object is returned. This window object is used for all subsequent RMA calls

One Sided Communication



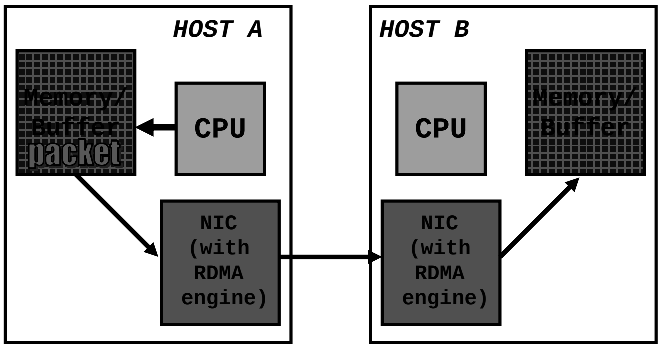
Coarse versus fine graining

- Expense of message passing implicitly suggests MPI-1 programs should be coarse grained
- Unit of messaging in NUMA systems is the cache line
  - What about API for (fast network) distributed memory systems that is optimized for smaller messages?
    - e.g. ARMCI http://www.emsl.pnl.gov/docs/parsoft/armci
  - Would enable distributed memory systems to have moderately high performance fine grained parallelism
  - A number of applications are suited to this style of parallelism (especially irregular data structures)
  - T3E and T3D both capable of performing fine grained calculations – well balanced machines
  - API's supporting fine grained parallelism have one-sided communication for efficiency – no handshaking to take processes away from computation

Puts and Gets in MPI-2

- In one sided communication the number of operations is reduced by (at least) a factor of 2
  - If communication patterns are dynamic and unknown then four MPI operations may be replaced by one MPI_Get/Put
- Circumvents the need to forward information directly to the remote CPU specifying what data is required
- MPI_Sends+MPI_Recv's are replaced by three possibilities
  - MPI_Get: Retrieve section of a remote array
  - MPI_Put: Place a section of a local array into remote memory
  - MPI_Accumulate: Remote update over operator and local data
- However, programmer must be aware of the possibility of remote processes changing local arrays!

RMA illustrated

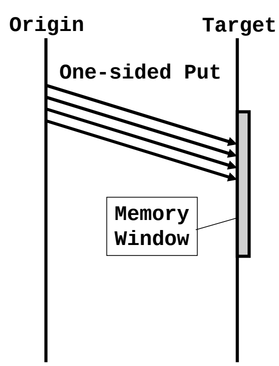


Benefits of one-sided communication

- No matching operation required for remote process
- All parameters of the operations are specified by the origin process
- Allows very flexible communcations patterns
  - Communication and synchronization are separated
    - Synchronization is now implied by the *access epoch*
- Removes need for polling for incoming messages
- Significantly improves performance of applications with irregular and unpredictable data movement

Windows: The fundamental construction for one-sided comms

- One sided comms may only write into memory regions (“windows”) set aside for communication
- Access to the windows must be within a specific access epoch
- All processes may agree on access epoch, or just a pair of processes may cooperate



Creating a window

- MPI_Win_create(base,size,disp_unit,info,comm,win,ierr)
  - Base address of window
  - Size of window in BYTES
  - Local unit size for displacements (BYTES, e.g. 4)
  - Info – argument about type of operations that may occur on window
  - Win – window object returned by call
- Should also free window using MPI_Win_free(win,ierr)
- Window performance is always better when base aligns on a word boundary

Options to info

- Vendors are allowed to include options to improve window performance under certain circumstances
- MPI_INFO_NULL is always valid
- If win_lock is not going to be used then this information can be passed as an info argument:

```
MPI_Info info;
MPI_Info_create(&info);
MPI_Info_set(info,"no_locks","true");
MPI_Win_create(…,info,…);
MPI_Info_free(&info);
```


Rules for memory areas assigned to windows

- Memory regions for windows involved in active target synchronization may be statically declared
- Memory regions for windows involved in passive target access epochs may have to be dynamically allocated
  - depends on implementation
  - For Fortran requires definition of Cray-like pointers to arrays
  - MPI_Alloc_mem(size,MPI_INFO_NULL,baseptr)
  - Must be associated with freeing call
    - MPI_Free_mem(baseptr)

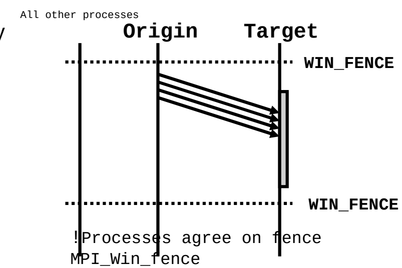
```
double *p

MPI_Alloc_mem(10*sizeof(double),
    MPI_INFO_NULL,&p)
…
…
…
call MPI_Free_mem(&p)
```


Access epochs

- Although communication is mediated by GETs and PUTs they do not guarantee message completion
- All communication must occur within an access epoch
- Communication is only guaranteed to have completed when the epoch is finished
  - This is to optimize messaging – do not have to worry about completion until access epoch is ended
- Two ways of coordinating access
  - Active target: remote process governs completion
  - Passive target: Origin process governs completion

Access epochs : Active target

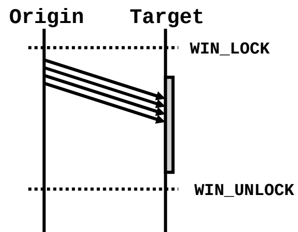
- Active target communication is usually expressed in a collective operation
- All processes agree on the beginning of the window
- Communication occurs
- Communication is then guaranteed to have completed when second WIN_Fence is called




```
!Processes agree on fence
MPI_Win_fence
!Put remote data
MPI_Put(..)
!Collective fence
MPI_Win_fence
!Message is guaranteed to
!complete after win_fence
!on remote process completes
```

Access epochs : Passive target

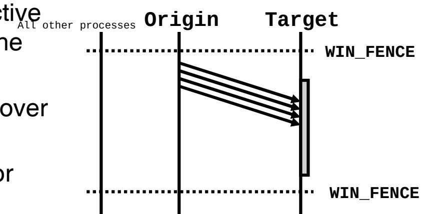
- For passive target communication, the origin process controls all aspects of communication
- Target process is oblivious to the communication epoch
- MPI_Win_(un)lock facilitates the communication




```
!Lock remote process window
MPI_Win_lock
!Put remote data
MPI_Put(..)
!Unlock remote process window
MPI_Win_unlock
!Message is guaranteed to
!complete after win_unlock
```


Non-collective active target

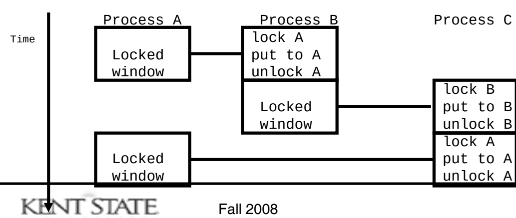
- Win_fence is collective over the comm of the window
- A similar construct over groups is available
- See Using MPI-2 for more details




```
!Processes agree on fence
Call MPI_Win_start(group,..)
Call MPI_Win_post(group,..)
!Put remote data
Call MPI_PUT(..)
!Collective fence
Call MPI_Win_complete(win)
Call MPI_Win_wait(win)
!Message is guaranteed to
!complete after waits finish
```


More on passive target access

- Closest idea to shared memory operation on a distributed system
- Very flexible communication model
- Multiple origin processes must negotiate on access to locks



Cray SHMEM – origin of many one-sided communication concepts

- On the T3E a number of variable types were guaranteed to occupy the same point in memory on different nodes:
  - Global variables/variables in common blocks
  - Local static variables
  - Fortran variables specified via !DIR$ SYMMETRIC directive
  - C variables specified by #pragma symmetric directive
  - Variables that are stack allocated, or dynamically on to the heap are not guaranteed to occupy the same address on different processors
- These variables could be rapidly retrieved/replaced via shmem_get/put
  - One sided operations
- Because these memory locations are shared among processors the library is dubbed “SHared MEMory” – SHMEM
  - It does not have a global address space (although you could implement one around this idea)
  - Similar idea to global arrays
- A lot of functionality from SHMEM is available in the MPI-2 one sided library (and was central in the design)

Shmem example

```
C
C Taken from Cray MPP Fortran Reference Manual
C
C Added CACHE_ALIGN directive to show how it should be done
C
C          Ken Steube - 3/11/96
C
C Each PE initializes array source() with the PE number,
C mype, and then gets the values of source from PE number
C mype-1. It checks to make sure the values it got the
C right values after receiving the data.
C
C This code calls shmem_get() to accomplish the task.
C Be aware that shmem_put() is significantly faster than
C shmem_get(), and so it should be used when possible.
C
      program   ring_of_PEs
      parameter (N=10 000)
      common /xxx/ target,source
      real      target(N)
      real      source(N)
CDIR$ CACHE_ALIGN target source
      integer   previous
      integer   shmem_get
      intrinsic my_pe
      data iflag /1/
C
      mype = my_pe()
C
       previous = mod(mype - 1 + N$PES, N$PES)
C
       do i = 1 , N ! Assign unique values on each PE
          source(i) = mype
       enddo
C
       call barrier()  ! All PEs initialize source
                       ! before doing the get
C
       iget = shmem_get(target, source, N,
     $      previous)
       do i = 1, N
          if (target(i) .ne. previous) then
             iflag = 0
             print*,'PE #',mype,': target(',i,')=',
     $          target(i),', should be ',previous
          endif
       enddo
C
       if (iflag .eq. 0) then
          print*,'Test failed on PE ',mype
       else
          print*,'Test passed on PE ',mype
       endif
C
       end
```


MPI_Get/Put/Accumulate

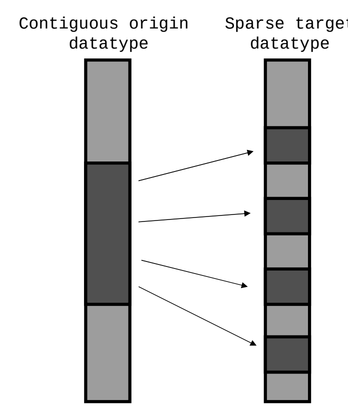
- Non-blocking operations
- MPI_Get(origin address,count,datatype,target,target displ,target count,target datatype,win,ierr)
  - Must specify information about both origin and remote datatypes – more arguments
  - No need to specify communicator – contained in window
  - Target displ is displacement from beginning of target window
  - Note remote datatype cannot resolve to overlapping entries
- MPI_Put has same interface
- MPI_Accumulate requires the reduction operator also be specified (argument before the window)
  - Same operators as MPI_REDUCE, but user defined functions cannot be used
  - Note MPI_Accumulate is really MPI_Put_accumulate, there is no get functionality (must do by hand)

MPI_Accumulate

- Extremely powerful operation "put+op"
- Question marks for implementations though
  - Who actually implements the "op" side of things?
  - If on remote node then there must be an extra thread to do this operation
  - If on local node, then accumulate becomes get followed by operation followed by put
- Many computations involve summing values into fields
  - MPI_Accumulate provides the perfect command for this
- For scientific computation it is frequently more useful than MPI_Put

Don't forget datatypes

- In one-sided comms datatypes play an extremely important role
  - Specify explicitly the unpacking on the remote node
  - Origin node must know precisely what the required remote data type is



Use PUTs rather than GETs

- Although both PUTs and GETs are non-blocking it is desirable to use PUTs whenever possible
  - GETs imply an inherent wait for data arrival and only complete when the message side has fully decoded the incoming message

MPI_Win_fence

- MPI_Win_fence(info,win,ierr)
  - Info allows user to specify constant that may improve performance (default of 0)
    - MPI_MODE_NOSTORE: No local stores
    - MPI_MODE_NOPUT: No puts will occur within the window (don't have to watch for remote updates)
    - MPI_MODE_NOPRECEDE: No earlier epochs of communication (optimize assumptions about window variables)
    - MPI_MODE_NOSUCCEED: No epochs of communication will follow this fence
    - NO_PRECEDE and NOSUCCEED must be called collectively
- Multiple messages sent to the same target between fences may be concatenated to improve performance

MPI_Win_(un)lock

- MPI_Win_lock(lock_type,target,info,win,ierr)
  - Lock_types:
    - MPI_LOCK_SHARED – use only for concurrent reads
    - MPI_LOCK_EXCLUSIVE – use when updates are necessary
- Although called a lock – it actually isn't (very poor naming convention)
  - "MPI_begin/end_passive_target_epoch"
  - Only on the local process does MPI_Win_lock act as a lock
  - Otherwise non-blocking
- Provides a mechanism to ensure that the communication epoch is completed
- Says nothing about order in which other competing message updates will occur on the target (consistency model is not specified)

Subtleties of *nonblocking* 'locking' and messaging

- Suppose we wanted to implement a fetch and add:

```
int one=1;
MPI_Win_create(…,&win);
…
MPI_Win_lock(MPI_LOCK_EXCLUSIVE,0,0,win);
MPI_Get(&value,1,MPI_INT,0,0,1,MPI_INT,win);
MPI_Accumulate(&one,1,MPI_INT,0,0,1,MPI_INT,MPI_SUM,win);
MPI_Win_unlock(0,win);
```

- Code is erroneous for two reasons:
  - 1. Cannot read and update same memory location in same access epoch
  - 2. Even if you could, communication is nonblocking and can complete in any order

Simple example

```
      subroutine exchng2( a, sx, ex, sy, ey, win,
     *   left_nbr, right_nbr, top_nbr, bot_nbr,
     *   right_ghost_disp, left_ghost_disp,
     *   top_ghost_disp, coltype, right_coltype, left_coltype )
      include 'mpif.h'
      integer sx, ex, sy, ey, win, ierr
      integer left_nbr, right_nbr, top_nbr, bot_nbr
      integer coltype, right_coltype, left_coltype
      double precision a(sx-1:ex+1,sy-1:ey+1)
C     This assumes that an address fits in a Fortran integer.
C     Change this to integer*8 if you need 8-byte addresses
      integer (kind=MPI_ADDRESS_KIND) right_ghost_disp,
     *    left_ghost_disp, top_ghost_disp, bot_ghost_disp
      integer nx

      nx = ex - sx + 1

      call MPI_WIN_FENCE( 0, win, ierr )
C Put bottom edge into bottom neighbor's top ghost cells
      call MPI_PUT( a(sx,sy), nx, MPI_DOUBLE_PRECISION, bot_nbr,
     *      top_ghost_disp, nx, MPI_DOUBLE_PRECISION,
     *      win, ierr )
C Put top edge into top neighbor's bottom ghost cells
      bot_ghost_disp = 1
      call MPI_PUT( a(sx,ey), nx, MPI_DOUBLE_PRECISION, top_nbr,
     *      bot_ghost_disp, nx, MPI_DOUBLE_PRECISION,
     *      win, ierr )
C Put right edge into right neighbor's left ghost cells
      call MPI_PUT( a(ex,sy), 1, coltype,
     *      right_nbr, left_ghost_disp, 1, right_coltype,
     *      win, ierr )
C Put left edge into the left neighbor's right ghost cells
      call MPI_PUT( a(sx,sy), 1, coltype,
     *      left_nbr, right_ghost_disp, 1, left_coltype,
     *      win, ierr )
      call MPI_WIN_FENCE( 0, win, ierr )
      return
      end
```

exchng2 for 2d poisson problem
No gets are required – just put your own data into other processes memory window.

Problems with passive target access

- Window creation must be collective over the comm
  - Expensive and time consuming
- MPI_Alloc_mem may be required
- Race conditions on a single window location under concurrent get/put must be handled by user
- Local and remote operations on a remote window cannot occur concurrently even if different parts of the window are being accessed at the same time
  - Local processes must execute MPI_Win_lock as well
- Multiple windows may have overlap, but must ensure concurrent operations to do different windows do not lead to race conditions on the overlap
- Cannot access (via MPI_get for example) and update (via a put back) the same location in the same access epoch (either between fences or lock/unlock)

Drawbacks of one sided comms in general (slightly dated)

- No evidence for advantage except on
  - SMP machines
  - Cray distributed memory systems (and Quadrics and now Infiniband)
    - Although advantage on these machines is significant – on T3E MPI latency is 16 μs, SHMEM latency is 2 μs
- Slow acceptance
  - Myrinet one sided comms “coming soon”
  - MPICH2 still not in full release
  - LAM supports only active target
- Unclear how many applications actually benefit from this model
  - Not entirely clear whether nonblocking normal send/recvs can achieve similar speed for some applications

Hardware – Reasons to be optimistic

- Newer network technologies (e.g. Infiniband, Quadrics) have a built in RDMA engine
  - RMA framework can built on top of the NIC library (“verbs”)
- 10 gigabit ethernet will almost certainly come with an RDMA engine
- Myrinet and SCI will both have one sided comms implemented very soon (after years of procrastination)
- Still in its infancy – number of software issues to work out
  - Support for non-contiguous datatypes is proving difficult – need efficient way to deal with the gather/scatter step
  - Many RDMA engines are designed for movement of contiguous regions – a comparatively rare operation in many situations
  - See http://nowlab.cis.ohio-state.edu/projects/mpi-iba/

Case Study: Matrix transpose

- See Sun documentation
- Need to transpose elements across processor space
  - Could do one element at a time (bad idea!)
  - Aggregate as much local data as possible and send large message (requires a lot of local data movement)
  - Send medium-sized contiguous packets of elements (there is some contiguity in the data layout)

Program 1

```
include "mpif.h"
real(8), allocatable, dimension(:) ::
a, b, c, d
real(8) t0, t1, t2, t3
! initialize parameters
call init(me,np,n,nb)
! allocate matrices
allocate(a(nb*np*nb))
allocate(b(nb*nb*np))
allocate(c(nb*nb*np))
allocate(d(nb*np*nb))
! initialize matrix
call initialize_matrix(me,np,nb,a)
! timing
do itime = 1, 10
  call MPI_Barrier(MPI_COMM_WORLD,ier)
  t0 = MPI_Wtime()
```

```
! first local transpose
  do k = 1, nb
  do j = 0, np - 1
    ioffa = nb * (   j   + np * (k-1) )
    ioffb = nb * ( (k-1) + nb *   j   )
    do i = 1, nb
      b(i+ioffb) = a(i+ioffa)
    enddo
  enddo
  enddo
  t1 = MPI_Wtime()
  ! global all-to-all
call MPI_Alltoall(b, nb*nb, MPI_REAL8, &
                  c, nb*nb, MPI_REAL8, MPI_COMM_WORLD, ier)
  t2 = MPI_Wtime()
  ! second local transpose
  call dtrans(`o', 1.d0, c, nb, nb*np, d)
  call MPI_Barrier(MPI_COMM_WORLD,ier)
  t3 = MPI_Wtime()

if ( me .eq. 0 ) &
  write(6,'(f8.3," seconds; breakdown on proc 0 =  ",3f10.3)') &
  t3 - t0, t1 - t0, t2 - t1, t3 - t2
enddo
! check
call check_matrix(me,np,nb,d)
deallocate(a)
deallocate(b)
deallocate(c)
deallocate(d)
call MPI_Finalize(ier)
```

This code aggregates data locally and uses the two-sided Alltoall collective Operation. Data is then rearranged using a subroutine called DTRANS()

Version 2 – one sided

```
include "mpif.h"
integer(kind=MPI_ADDRESS_KIND) nbytes
integer win
real(8) c(*)
pointer (cptr,c)
real(8), allocatable, dimension(:) :: a, b, d
real(8) t0, t1, t2, t3
! initialize parameters
call init(me,np,n,nb)
! allocate matrices
allocate(a(nb*np*nb))
allocate(b(nb*nb*np))
allocate(d(nb*np*nb))
nbytes = 8 * nb * nb * np
call MPI_Alloc_mem(nbytes, MPI_INFO_NULL, cptr, ier)
if ( ier .eq. MPI_ERR_NO_MEM  ) stop
! create window
call MPI_Win_create(c, nbytes, 1, MPI_INFO_NULL, MPI_COMM_WORLD,
 win, ier)
! initialize matrix
call initialize_matrix(me,np,nb,a)
! timing
```

```
do itime = 1, 10
  call MPI_Barrier(MPI_COMM_WORLD,ier)
  t0 = MPI_Wtime()
  t1 = t0
  ! combined local transpose with global all-to-all
  call MPI_Win_fence(0, win, ier)
  do ip = 0, np - 1
  do ib = 0, nb - 1
    nbytes = 8 * nb * ( ib + nb * me )
    call MPI_Put(a(1+nb*ip+nb*np*ib), nb, MPI_REAL8, ip, nbytes, &
                                       nb, MPI_REAL8, win, ier)
  enddo
  enddo
  call MPI_Win_fence(0, win, ier)
  t2 = MPI_Wtime()
  ! second local transpose
  call dtrans(`o', 1.d0, c, nb, nb*np, d)
  call MPI_Barrier(MPI_COMM_WORLD,ier)
  t3 = MPI_Wtime()
  if ( me .eq. 0 ) &
    write(6,'(f8.3," seconds; breakdown on proc 0 = ",3f10.3)') &
    t3 - t0, t1 - t0, t2 - t1, t3 - t2
enddo
! check
call check_matrix(me,np,nb,d)
! deallocate matrices and stuff
call MPI_Win_free(win, ier)
deallocate(a)
deallocate(b)
deallocate(d)
call MPI_Free_mem(c, ier)
call MPI_Finalize(ier)
end
```

No local aggregation is used, and communication is mediated via MPI_Puts. Data is then rearranged using a subroutine called DTRANS()

Performance comparison

Version	Total	Local Aggregation	Communication	Dtrans call
1	2.109	0.585	0.852	0.673
2	1.177	0.0	0.43	0.747

- One sided version is twice as fast on this machine (Sun 6000 SMP)
- Net data movement is slightly over 1.1 Gbyte/s, which is about ½ the net bus bandwidth (2.6 Gbyte/s)
- Big performance boost from getting rid of aggregation and the fast messaging using shorter one sided messages

Summary

- One sided comms can reduce synchronization and thereby increase performance
- They indirectly reduce local data movement
- The reduction in messaging overhead can simplify programming