Tuning MPI Applications for Peak Performance

William Gropp
Rusty Lusk
Rajeev Thakur

1

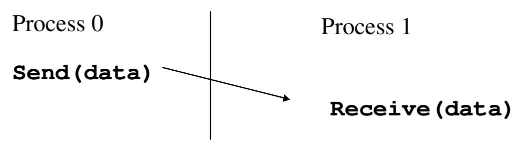
The Message-Passing Model

- A *process* is (traditionally) a program counter and address space.
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.
- Interprocess communication consists of
  - » Synchronization
  - » Movement of data from one process's address space to another's.

2

Cooperative Operations for Communication

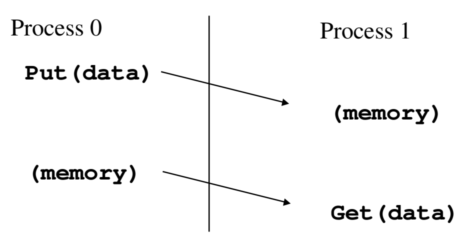
- The message-passing approach makes the exchange of data cooperative.
- Data is explicitly *sent* by one process and *received* by another.
- An advantage is that any change in the receiving process's memory is made with the receiver's explicit participation.
- Communication and synchronization are combined.



3

One-Sided Operations for Communication

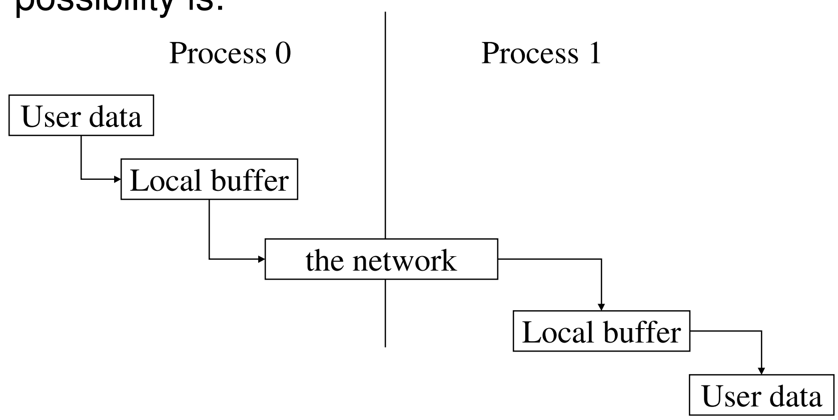
- One-sided operations between processes include remote memory reads and writes
- Only one process needs to explicitly participate.
- An advantage is that communication and synchronization are decoupled
- One-sided operations are part of MPI-2.



4

Buffers

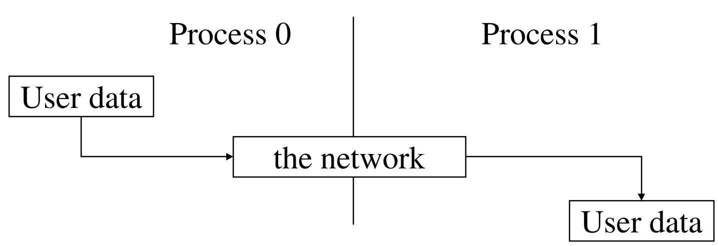
- When you send data, where does it go? One possibility is:



5

Avoiding Buffering

- It is better to avoid copies:



- This requires that **`MPI_Send`** wait on delivery, or that **`MPI_Send`** return before transfer is complete, and we wait later.

6

Blocking and Non-blocking Communication

- Blocking communication:
  - » `MPI_Recv` does not complete until the buffer is full (available for use).
  - » `MPI_Send` does not complete until the buffer is empty (available for use).
- Completion depends on size of message and amount of system buffering.

7

Sources of Deadlocks

- Send a large message from process 0 to process 1
  - » If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with this code?

Process 0	Process 1
Send(1)	**Send(0)**
Recv(1)	**Recv(0)**

- This is called "unsafe" because it depends on the availability of system buffers

8

Some Solutions to the "unsafe" Problem

- Order the operations more carefully:

Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

- Supply receive buffer at same time as send:

Process 0	Process 1
`Sendrecv(1)`	`Sendrecv(0)`

9

More Solutions to the "unsafe" Problem

- Supply own space as buffer for send

Process 0	Process 1
`Bsend(1)`	`Bsend(0)`
`Recv(1)`	`Recv(0)`

- Use non-blocking operations:

Process 0	Process 1
`Isend(1)`	`Isend(0)`
`Irecv(1)`	`Irecv(0)`
`Waitall`	`Waitall`

10

MPI's Non-blocking Operations

- Non-blocking operations return (immediately) “request handles” that can be tested and waited on.

```
MPI_Isend(start, count, datatype,
    dest, tag, comm, request)
MPI_Irecv(start, count, datatype,
    dest, tag, comm, request)
MPI_Wait(&request, &status)
```

- One can also test without waiting:

```
MPI_Test(&request, &flag, status)
```


Multiple Completions

- It is sometimes desirable to wait on multiple requests:

```
MPI_WAITALL(count, array_of_requests,
  array_of_statuses)
MPI_Waitany(count, array_of_requests,
  &index, &status)
MPI_Waitsome(count, array_of_requests,
  array_of indices, array_of_statuses)
```

- There are corresponding versions of **`test`** for each of these.

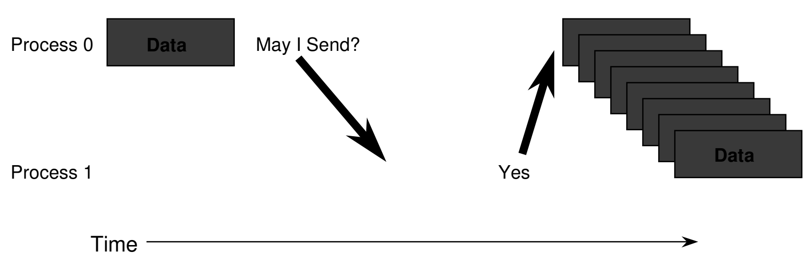
Communication Modes

- MPI provides multiple *modes* for sending messages:
  - » Synchronous mode (**`MPI_Ssend`**): the send does not complete until a matching receive has begun. (Unsafe programs deadlock.)
  - » Buffered mode (**`MPI_Bsend`**): the user supplies a buffer to the system for its use. (User allocates enough memory to make an unsafe program safe.
  - » Ready mode (**`MPI_Rsend`**): user guarantees that a matching receive has been posted.
    - – Allows access to fast protocols
    - – undefined behavior if matching receive not posted
- Non-blocking versions (**`MPI_Issend`**, etc.)
- **`MPI_Recv`** receives messages sent in any mode.

13

What is message passing?

- Data transfer plus synchronization



- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

14

Quick review of MPI Message passing

- Basic terms
  - » nonblocking - Operation does not wait for completion
  - » synchronous - Completion of send *requires* initiation (but not completion) of receive
  - » ready - *Correct* send requires a matching receive
  - » asynchronous - communication and computation take place simultaneously, ***not*** an MPI concept (implementations *may* use asynchronous methods)

15

Message protocols

- Message consists of "envelope" and data
  - » Envelope contains tag, communicator, length, source information, plus impl. private data
- Short
  - » Message data (message for short) sent with envelope
- Eager
  - » Message sent assuming destination can store
- Rendezvous
  - » Message not sent until destination oks

16

Special Protocols for DSM

- Message passing is a good way to use distributed shared memory (DSM) machines because it provides a way to express memory locality.
- Put
  - » Sender puts to destination memory (user or MPI buffer). Like Eager.
- Get
  - » Receiver gets data from sender or MPI buffer. Like Rendezvous.
- Short, long, rendezvous versions of these

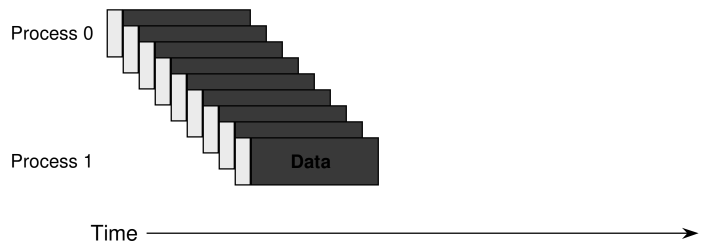
17

Message Protocol Details

- Eager not Rsend, rendezvous not Ssend resp., but related
- User versus system buffer space
- Packetization
- Collective operations
- Datatypes, particularly non-contiguous
  - » Handling of important special cases
    - – Constant stride
    - – Contiguous structures

18

Eager Protocol



- Data delivered to process 1
  - » No matching receive may exist; process 1 must then buffer and copy.

19

Eager Features

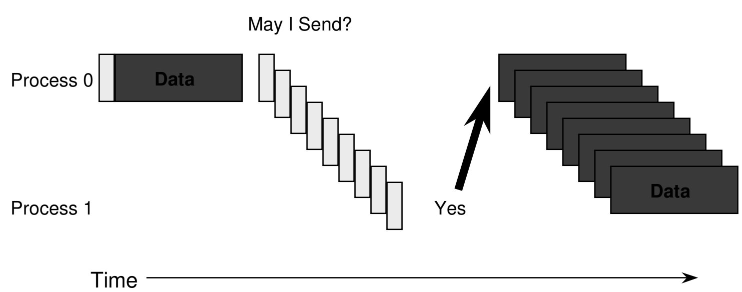
- Reduces synchronization delays
- Simplifies programming (just MPI_Send)
- Requires significant buffering
- May require active involvement of CPU to drain network at receiver's end
- May introduce additional copy (buffer to final destination)

20

How Scalable is Eager Delivery?

- Buffering must be reserved for arbitrary senders
- User-model mismatch (often expect buffering allocated entirely to “used” connections).
- Common approach in implementations is to provide same buffering for all members of MPI_COMM_WORLD; this is optimizing for non-scaleable computations
- Scaleable implementations that exploit message patterns are possible

Rendezvous Protocol



- Envelope delivered first
- Data delivered when user-buffer available
  - » Only buffering of envelopes required

Rendezvous Features

- Robust and safe
  - » (except for limit on the number of envelopes…)
- May remove copy (user to user direct)
- More complex programming (waits/tests)
- May introduce synchronization delays (waiting for receiver to ok send)

23

Short Protocol

- Data is part of the envelope
- Otherwise like eager protocol
- May be performance optimization in interconnection system for short messages, particularly for networks that send fixed-length packets (or cache lines)

24

User and System Buffering

- Where is data stored (or staged) while being sent?
  - » User’s memory
    - – Allocated on the fly
    - – Preallocated
  - » System memory
    - – May be limited
    - – Special memory may be faster

25

Implementing MPI_Isend

- Simplest implementation is to always use rendezvous protocol:
  - » MPI_Isend delivers a request-to-send control message to receiver
  - » Receiving process responds with an ok-to-send
    - – May or may not have matching MPI receive; only needs buffer space to store incoming message
  - » Sending process transfers data
- Wait for MPI_Isend request
  - » wait for ok-to-send message from receiver
  - » wait for data transfer to be complete on sending side

26

Alternatives for MPI_Isend

- Use a short protocol for small messages
  - » No need to exchange control messages
  - » Need guaranteed (but small) buffer space on destination for short message envelope
  - » Wait becomes a no-op
- Use eager protocol for modest sized messages
  - » Still need guaranteed buffer space for both message envelope and eager data on destination
  - » Avoids exchange of control messages

27

Implementing MPI_Send

- Can't use eager always because this could overwhelm the receiving process

```
if (rank != 0) MPI_Send( 100 MB of data )
else receive 100 MB from each process
```

- Would like to exploit the blocking nature (can wait for receive)
- Would like to be fast
- Select protocol based on message size (and perhaps available buffer space at destination)
  - » Short and/or eager for small messages
  - » Rendezvous for longer messages

28

Implementing MPI_Rsend

- Just use MPI_Send; no advantage for users
- Use eager always (or short if small)
  - » even for long messages

29

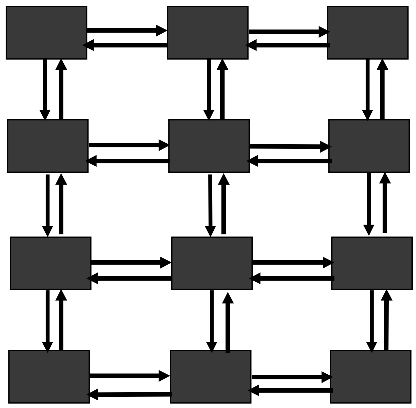
Unexpected Hot Spots

- Arises even in common grid exchange patterns
- Message passing illustrates problems present even in shared memory
  - » Blocking operations may cause unavoidable stalls

30

Mesh Exchange

- Exchange data on a mesh



31

Sample Code

- Do i=1,n_neighbors

```
Do i=1,n_neighbors
  Call MPI_Send(edge, len, MPI_REAL, nbr(i), tag,
                      comm, ierr)
Enddo
Do i=1,n_neighbors
  Call MPI_Recv(edge,len,MPI_REAL,nbr(i),tag,
                      comm,status,ierr)
Enddo
```

32

Deadlocks!

- All of the sends may block, waiting for a matching receive (will for large enough messages)
- The variation of
  if (has down nbr)
     Call MPI_Send(… down …)
  if (has up nbr)
     Call MPI_Recv(… up …)
  …
  sequentializes (all except the bottom process blocks)

Sequentialization

Start Send	Start Send	Start Send	Start Send	Start Send	Start Send	Send	Recv
					Send	Recv	
				Send	Recv		
			Send	Recv			
		Send	Recv				
	Send	Recv					
Send	Recv						

Fix 1: Use Irecv

- Do i=1,n_neighbors
```
   Call MPI_Irecv(edge,len,MPI_REAL,nbr(i),tag,
                  comm,requests(i),ierr)
Enddo
Do i=1,n_neighbors
   Call MPI_Send(edge, len, MPI_REAL, nbr(i), tag,
                 comm, ierr)
Enddo
Call MPI_Waitall(n_neighbors, requests, statuses, ierr)
```
- Does not perform well in practice. Why?

35

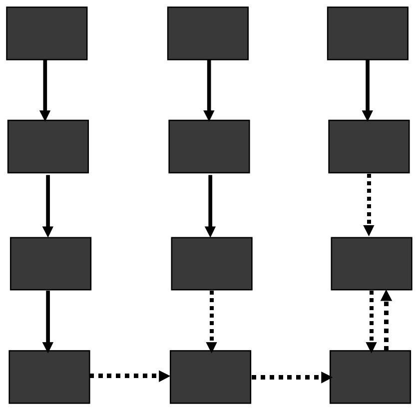
Timing Model

- Sends interleave
- Sends block (data larger than buffering will allow)
- Sends control timing
- Receives do not interfere with Sends
- Exchange can be done in 4 steps (down, right, up, left)

36

Mesh Exchange - Step 1

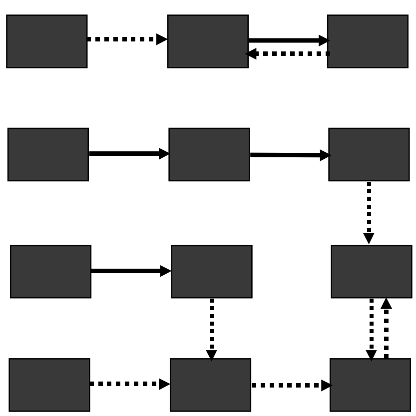
- Exchange data on a mesh



37

Mesh Exchange - Step 2

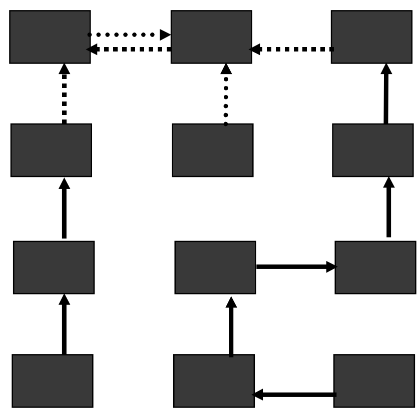
- Exchange data on a mesh



38

Mesh Exchange - Step 3

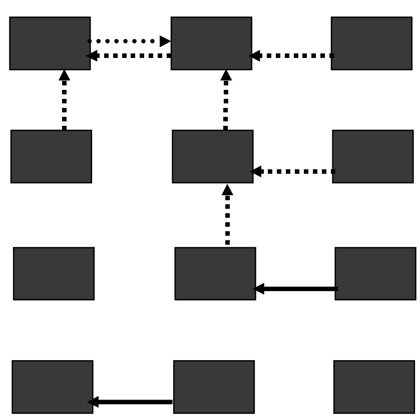
- Exchange data on a mesh



39

Mesh Exchange - Step 4

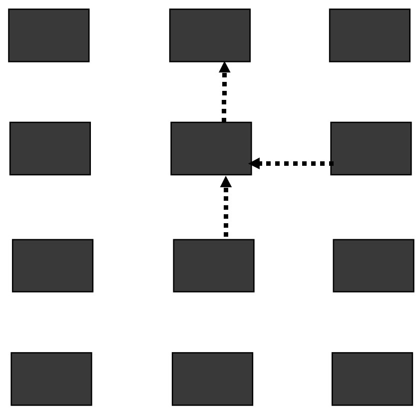
- Exchange data on a mesh



40

Mesh Exchange - Step 5

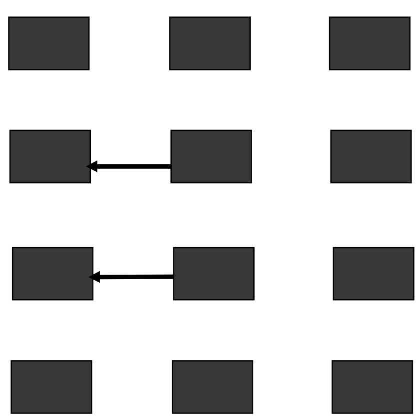
- Exchange data on a mesh



41

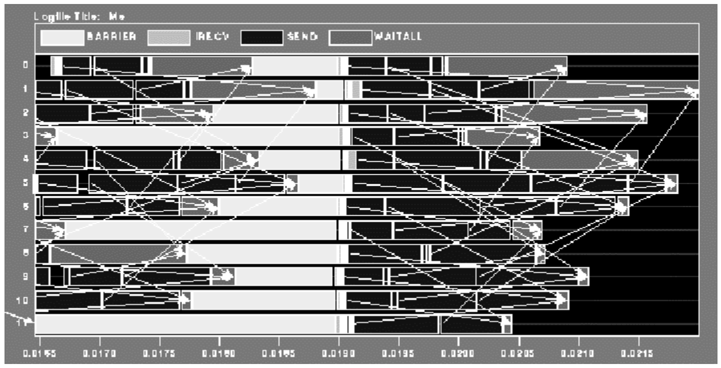
Mesh Exchange - Step 6

- Exchange data on a mesh



42

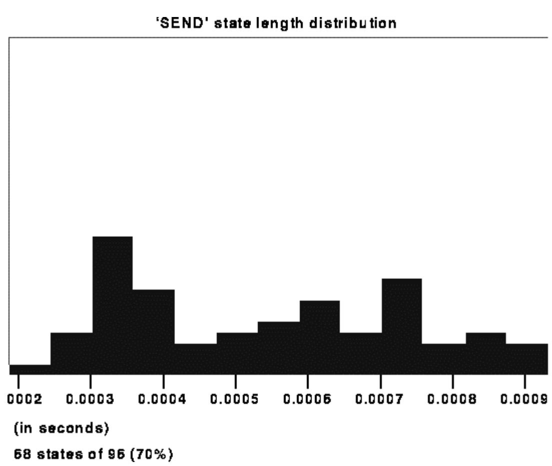
Timeline from IBM SP



• Note that process 1 finishes last, as predicted

43

Distribution of Sends



44

Why Six Steps?

- Ordering of Sends introduces delays when there is contention at the receiver
- Takes roughly twice as long as it should
- Bandwidth is being wasted
- Same thing would happen if using memcpy and shared memory

45

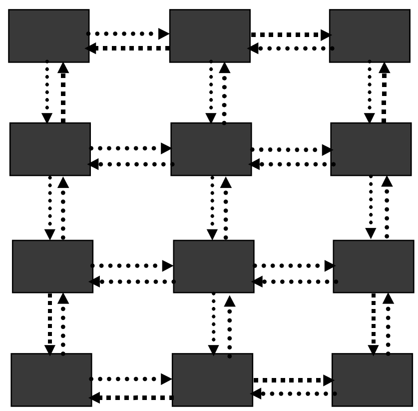
Fix 2: Use Isend and Irecv

- Do i=1,n_neighbors
```
Do i=1,n_neighbors
  Call MPI_Irecv(edge,len,MPI_REAL,nbr(i),tag,
                 comm,request(i),ierr)
Enddo
Do i=1,n_neighbors
  Call MPI_Isend(edge, len, MPI_REAL, nbr(i), tag,
                 comm, request(n_neighbors+i), ierr)
Enddo
Call MPI_Waitall(2*n_neighbors, request, statuses,
                 ierr)
```
- (We'll see later how to do even better than this)

46

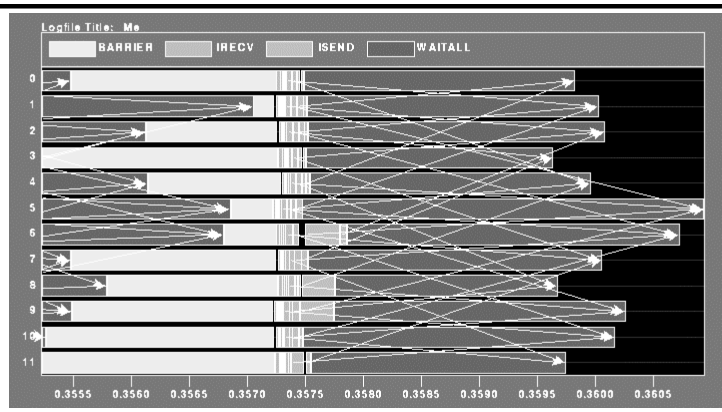
Mesh Exchange - Steps 1-4

- Four interleaved steps



47

Timeline from IBM SP



Note processes 5 and 6 are the only interior processors; these perform more communication than the other processors

48

Lesson: Defer Synchronization

- Send-receive accomplishes two things:
  - » Data transfer
  - » Synchronization
- In many cases, there is more synchronization than required
- Use nonblocking operations and MPI_Waitall to defer synchronization

49

MPI-2 Solution

- MPI-2 introduces one-sided operations
  - » Put, Get, Accumulate
  - » Separate data transfer from synchronization
- These are all nonblocking (blocking implies some synchronization)

50

One-sided Code

- Do i=1,n_neighbors
  Call MPI_Get(edge,len,MPI_REAL,nbr(i),
  edgedisp(i),len,MPI_REAL,win,ierr)
  Enddo
  Call MPI_Win_fence(0, win, ierr)
- MPI_Put may be preferable on some platforms
- Can avoid global synchronization (MPI_Win_fence) with MPI_Win_start/post/complete/wait
- Use MPI_Accumulate to move and add

51

MPI Datatypes

- The data in a message to sent or received is described by a triple (address, count, datatype), where
- An MPI *datatype* is recursively defined as:
  - » predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE_PRECISION)
  - » a contiguous array of MPI datatypes
  - » a strided block of datatypes
  - » an indexed array of blocks of datatypes
  - » an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, such an array of (int, float) pairs, or a row of a matrix stored columnwise.

52

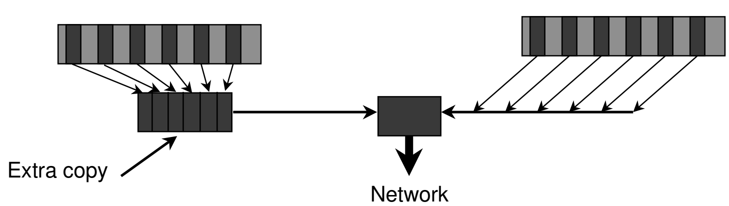
Why Datatypes?

- Since all data is labeled by type, an MPI implementation can support communication between processes on machines with very different memory representations and lengths of elementary datatypes (heterogeneous communication).
- Specifying application-oriented layout of data in memory
  - » can reduce memory-to-memory copies in the implementation
  - » allows the use of special hardware (scatter/gather) when available

53

Non-contiguous Datatypes

- Provided to *allow* MPI implementations to avoid copy



  - » Not widely implemented *yet*
- Handling of important special cases
  - » Constant stride
  - » Contiguous structures

54

Potential Performance Advantage in MPI Datatypes

- Handling non-contiguous data
- Assume must pack/unpack on each end
  - » *cn + (s + r n) + cn = s + (2c + r)n*
- Can move directly
  - » *s + r' n*
  - » *r'* probably > r but < *(2c+r)*
- MPI implementation must copy data anyway (into network buffer or shared memory); having the datatype permits removing 2 copies

Performance of MPI Datatypes

- Test of 1000 element vector of doubles with stride of 24 doubles (MB/sec).
  - » MPI_Type_vector
  - » MPI_Type_struct (MPI_UB for stride)
  - » User packs and unpacks by hand

	SP2	SP3	SGI	Origin	T3D
Vector	4.0	5.4	6.7	8.0	**12.7**
Struct	3.0	4.3	3.0	3.2	0.4
User	**15**	**7.9**	**40**	**47**	10.8

- Performance very dependent on implementation; should improve with time

Performance of MPI Datatypes II

- Performance of MPI_Type_Vector
  - » 1000 doubles with stride 24
  - » Performance in MB/sec

Platform	Vendor MPI	MPICH	User
		Equivalent Type_Struct	
IBM SP Blue Pacific	3.79 2.94	6.71 1.02	9.52
SGI O2000 Blue Mountain	7.63 3.09	20.2 2.6	43.8
Intel Tflops Red	32.3 3.29	MPICH *is* Vendor MPI	34.7

57

Datatype Abstractions

- Standard Unix abstraction is “block of contiguous bytes” (e.g., readv, writev)
- MPI specifies datatypes recursively as
  - » count of (type,offset)
    where offset may be relative or absolute
  - » MPICH uses a simplified form of this for MPI_Type_vector (and so outperforms vendor MPIs)
- More general form would provide superior performance for most user-defined datatypes
  - » MPI implementations can improve

58

Working With MPI Datatypes

- An MPI datatype defines a *type signature*:
  - » sequence of pairs: (basic type,offset)
  - » An integer at offset 0, followed by another integer at offset 8, followed by a double at offset 16 is
    - – (integer,0), (integer,4), (double,16)
  - » Offsets need not be increasing:
    - – (integer,64),(double,0)
- An MPI datatype has an extent and a size
  - » *size* is the number of bytes of the datatype
  - » *extent* controls how a datatype is used with the *count* field in a send and similar MPI operations
  - » extent is a misleading name

59

What does extent do?

- Consider
  MPI_Send(buf, count, datatype, …)
- What actually gets sent?
- MPI defines this as

```
do i=0,count-1
   MPI_Send(buf(1+i*extent(datatype)),1,
                 datatype,…)
```

  (buf is a byte type like integer*1)
- *extent* is used to decide where to send from (or where to receive to in MPI_Recv) for count > 1
- Normally, this is right after the last byte used for (i-1)

60

Changing the extent

- MPI-1 provides two special types, MPI_LB and MPI_UB, for changing the extent of a datatype
  - » This doesn't change the *size*, just how MPI decides what addresses in memory to use in offseting one datatype from another.
- Use MPI_Type_struct to create a new datatype from an old one with a different extent
  - » Use MPI_Type_create_resized in MPI-2

61

Sending Rows of a Matrix

- From Fortran, assume you want to send a row of the matrix
  A(n,m),
  that is, A(row,j), for j=1,…, m
- A(row,j) is not adjacent in memory to A(row,j+1)
- One solution: send each element separately:

```
Do j=1,m
   Call MPI_Send( A(row,j), 1, MPI_DOUBLE_PRECSION, …)
```

- Why not?

62

MPI Type vector

- Create a single datatype representing elements separated by a constant distance (*stride*) in memory
  - » m items, separated by a stride of n:
  - » MPI_Type_vector(m, 1, n,
    MPI_DOUBLE_PRECISION, newtype, ierr)
    MPI_Type_commit(newtype, ierr)
  - » Type_commit required before using a type in an MPI communication operation.
- Then send one instance of this type
  MPI_Send(a(row,1), 1, newtype, ….)

63

Test your understanding of Extent

- How do you send 2 rows of the matrix? Can you do this:
  MPI_Send(a(row,1),2,newtype,…)
- Hint: Extent(newtype) is distance from the first to last byte of the type
  - » Last byte is a(row,m)
- Hint: What is the first location of A that is sent after the first row?

64

Sending with MPI_Vector

- Extent(newtype) is ((m-1)*n+1)*sizeof(double)
  - » Last element sent is A(row,m)
- do i=0,1
  MPI_Send(buf(1+i*extent(datatype)),1,
  datatype,…)
  becomes
- MPI_Send(A(row,1:m),…) (i=0)
  MPI_Send(A(row+1,m:2m-1),…) (i=1)
- ! The second step is not
  MPI_Send(A(row+1,1:m),…)

65

Solutions for Vectors

- MPI_Type_vector is for very specific uses
  - » rarely makes sense to use count other than 1
- Two send two rows, simply change the blockcount:
  MPI_Type_vector(m, 2, n,
  MPI_DOUBLE_PRECISION, newtype, ierr)
- Stride is still relative to basic type

66

Sending Vectors of Different Sizes

- How would you send A(i,2:m) and A(i+1,3:m) with a single MPI datatype?
  - » Use "count" to select the number, as in
    MPI_Send(A(i,2),m-1,type,…)
    MPI_Send(A(i+1,3),m-2,type,…)
- Hint: Use an extent of n elements

Striding Type

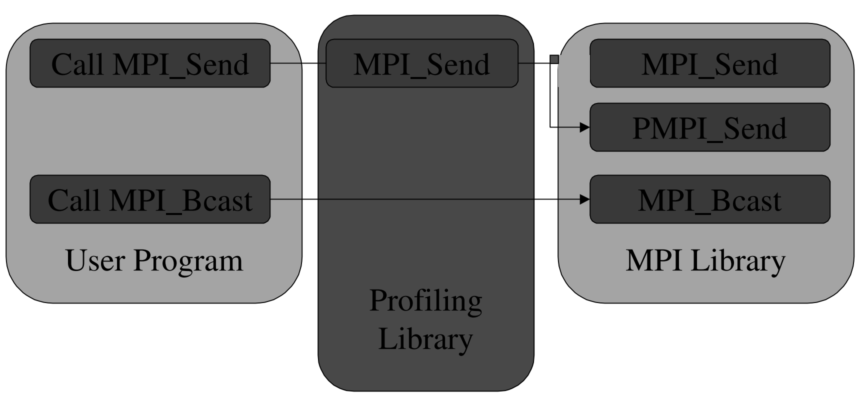
- Create a type with an extent of a column of the array:
  - »
```
types(1) = MPI_DOUBLE_PRECISION
types(2) = MPI_UB
displs(1) = 0
displs(2) = n * 8   ! Bytes!
blkcnt(1) = 1
blkcnt(2) = 1
call MPI_Type_struct( 2, blkcnt, displs, types,
                      newtype, ierr )
```
- MPI_Send(A(i,2),m-1,newtype,…)
  sends the elements A(i,2:m)

Using PMPI routines

- PMPI allows selective replacement of MPI routines at link time (no need to recompile)
- Some libraries already make use of PMPI
- Some MPI implementations have PMPI bugs
  - IBM — PMPI_Wtime() returns 0
  - Cray — PMPI in a separate library that some installations have not installed

69

Profiling Interface



70

Example Use of Profiling Interface

```
static int nsend = 0;

int MPI_Send( void *start, int count,
  MPI_Datatype datatype, int dest,
  int tag, MPI_Comm comm )
{
    nsend++;
    return PMPI_send(start, count, datatype,
                     dest, tag, comm);
}
```

71

Using the Profiling Interface from Fortran

```
Block data
common /mycounters/ nsend
data nsend/0/
end

subroutine MPI_Send( start, count, datatype, dest,&
                     tag, comm, ierr )
integer start(*), count, datatype, dest, tag, comm
common /mycounters/ nsend
save   /mycounters/
nsend = nsend + 1
call PMPI_send(start, count, datatype, &
                   dest, tag, comm, ierr )
end
```

72

Finding Unsafe uses of MPI_Send

```
subroutine MPI_Send( start, count, datatype, dest,
                     tag, comm, ierr )
integer start(*), count, datatype, dest, tag, comm
call PMPI_Ssend(start, count, datatype,
                   dest, tag, comm, ierr )
end
```

- MPI_Ssend will not complete until the matching receive starts
- MPI_Send can be implemented as MPI_Ssend
- At some value of *count*, MPI_Send will act like MPI_Ssend (or fail)

73

Reporting on Unsafe MPI_Send

```
subroutine MPI_Send( start, count, datatype, dest, tag, comm,
                     ierr )
integer start(*), count, datatype, dest, tag, comm
include 'mpif.h'
integer request,  status(MPI_STATUS_SIZE)
double precision t1, t2
logical flag

call PMPI_Issend(start, count, datatype, dest, tag, comm,
                 request, ierr )
flag = .false.
t1   = MPI_Wtime()
Do while (.not. flag .and. t1 + 10 .gt. MPI_Wtime())
    call PMPI_Test( request, flag, status, ierr )
Enddo
if (.not. Flag) then
    print *, 'MPI_Send appears to be hanging'
    call MPI_Abort( MPI_COMM_WORLD, 1, ierr )
endif
end
```

74

Logging and Visualization Tools

- Upshot and MPE tools
  http://www.mcs.anl.gov/mpi/mpich
- Pallas VAMPIR
  http://www.pallas.com/
- IBM VT
  http://www.rs6000.ibm.com/software/sp_products/pe.html
- Pablo http://www-pablo.cs.uiuc.edu/Projects/Pablo/pablo.html
- Paragraph
  http://www.ncsa.uiuc.edu/Apps/MCS/ParaGraph/ParaGraph.html
- Paradyn http://www.cs.wisc.edu/~paradyn
- Other vendor tools

75

Contents of MPI-2

- Extensions to the message-passing model
  - Dynamic process management
  - One-sided operations
  - Parallel I/O
- Making MPI more robust and convenient
  - C++ and Fortran 90 bindings
  - External interfaces, handlers
  - Extended collective operations
  - Language interoperability
  - MPI interaction with threads

76

Dynamic Process Management

- Issues
  - » maintaining simplicity, flexibility, and correctness
  - » interaction with operating system, resource manager, and process manager
  - » connecting independently started processes
- Spawning new processes is *collective,* returning an intercommunicator.
  - » Local group is group of spawning processes.
  - » Remote group is group of new processes.
  - » New processes have own **`MPI_COMM_WORLD`**.
  - » **`MPI_Comm_get_parent`** lets new processes find parent communicator.

77

One-Sided Operations: Issues

- Balancing efficiency and portability across a wide class of architectures
  - » shared-memory multiprocessors
  - » NUMA architectures
  - » distributed-memory MPP's
  - » Workstation networks
- Retaining "look and feel" of MPI-1
- Dealing with subtle memory behavior issues: cache coherence, sequential consistency
- Synchronization is separate from data movement.

78

Extended Collective Operations

- In MPI-1, collective operations are restricted to ordinary (intra) communicators.
- In MPI-2, most collective operations apply also to intercommunicators, with appropriately different semantics.
- E.g, Bcast/Reduce in the intercommunicator resulting from spawning new processes goes from/to root in spawning processes to/from the spawned processes.
- In-place extensions

79

External Interfaces

- Purpose: to ease extending MPI by layering new functionality portably and efficiently
- Aids integrated tools (debuggers, performance analyzers)
- In general, provides portable access to parts of MPI implementation internals.
- Already being used in layering I/O part of MPI on multiple MPI implementations.

80

C++ Bindings

- C++ binding alternatives:
  - » use C bindings
  - » Class library (e.g., OOMPI)
  - » “minimal” binding
- Chose “minimal” approach
- Most MPI functions are member functions of MPI classes:
  - » example: `MPI::COMM_WORLD.send( ... )`
- Others are in MPI namespace
- C++ bindings for both MPI-1 and MPI-2

Fortran Issues

- “Fortran” now means Fortran-90.
- MPI can’t take advantage of some new Fortran (-90) features, e.g., array sections.
- Some MPI features are incompatible with Fortran-90.
  - » e.g., communication operations with different types for first argument, assumptions about argument copying.
- MPI-2 provides “basic” and “extended” Fortran support.

Performance Issues in MPI-2

- Threads
  - » Use of threads requires thread-safe code; may require expensive locks inside the implementation
  - » MPI_INIT_THREAD options
- Dynamic Processes
  - » Adding processes with MPI_Spawn may not provide peak performance
- One-sided
  - » New ways to defer synchronization
  - » Efficient implementation on non-shared memory hardware is challenging.
- Parallel I/O
  - » Implementations exist…

83

I/O Performance Issues

84

Outline

- Unix I/O API and its performance implications
- MPI-IO and its solution to Unix I/O performance problems
- The ROMIO MPI-IO implementation and performance results

85

Unix I/O API

- Most parallel file systems have a Unix-like API
  - » *open* Open a file
  - » *lseek* Move the file pointer to a particular offset in the file
  - » *read* Read *n* bytes starting from the current location of the file pointer
  - » *write* Write *n* bytes starting from the current location of the file pointer
  - » *close* Close the file

86

Performance

- Open
  - » usually very expensive
  - » can take more than 1 second
- Seek
  - » performance depends on implementation
  - » very expensive on Paragon, inexpensive on SP
- Read/Write
  - » Performs well if I/O size is large -- 1MB or more
  - » Performs very poorly if I/O size is small -- 8KB or less
- Close
  - » not as expensive as open

87

Some Other I/O Functions

- *readv, writev*
  - » Reads into/writes from multiple buffers in memory
  - » In the file, however, data is assumed to be contiguously located
  - » Most implementations have a limit of 16 on the number of distinct memory buffers in one call
  - » Of limited use because users need to specify noncontiguity in the file more often than in memory

88

Asynchronous I/O

- *aio_read, aio_write* in POSIX
  - » nonblocking: user can test for completion or have the implementation issue a signal
  - » most implementations have a limit on the maximum number of concurrently active asynchronous operations; the limit is a small number, but not predefined
  - » performance is not good enough yet

89

List I/O

- *lio_listio* in POSIX
  - » user can specify a list of operations at a time
  - » can be a mixture of reads and writes
  - » each operation is internally performed as a separate *aio_read*, *aio_write*
  - » user can specify if *lio_listio* should return immediately (nonblocking) or return after completion of all requests
- Drawbacks:
  - » it doesn't treat the list as a *single* request
  - » implementation can't perform data sieving
  - » no notion of collective I/O
  - » not implemented everywhere

90

I/O Access Patterns in Parallel Applications

- Different from those in sequential programs
- Sequential programs typically access data in contiguous chunks
- In many parallel programs, each process may need to access several noncontiguous pieces of data from a file
  - » e.g., bytes 0--10, 20--30, 40--50, ...
- In addition, groups of processes may need to access the file simultaneously, and the accesses of different processes may be interleaved in the file

91

Problems with Unix API for Parallel I/O

- Noncontiguous accesses in the file cannot be expressed in a single function call
- Each contiguous piece must be accessed separately, resulting in too many system calls and poor performance
- No notion of collective I/O

92

Why MPI is a Good Setting for Parallel I/O

- Writing is like sending a message and reading is like receiving
- Any parallel I/O system will need a mechanism to:
  - » specify groups of processes for collective operations
  - » specify noncontiguous accesses
  - » test/wait for completion of nonblocking operations
- That is, it needs lots of MPI-like machinery

MPI-IO Background

- Marc Snir (IBM Watson) paper exploring MPI as context for parallel I/O (1994, after MPI-1)
- MPI-IO e-mail discussion group led by Jean-Pierre Prost (IBM) and Bill Nitzberg (NASA)
- MPI-IO group joins MPI-2 Forum in June 1996
- MPI-2 standard released in July 1997

Introduction to MPI-IO

- MPI-IO can be considered as Unix I/O plus (lots of) other stuff
- Basic operations:
  `MPI_File_{open, close, read, write, seek}`
- Parameters to these operations match Unix, aiding straightforward port from Unix I/O to MPI-IO
- However, to get performance and portability, more advanced features must be used

95

MPI-IO Features

- Noncontiguous access in both memory and file
- Use of explicit offset
- Individual and shared file pointers
- Nonblocking I/O
- Collective I/O
- File interoperability
- Portable data representation
- Mechanism for providing hints applicable to a particular implementation and I/O environment (e.g. number of disks, striping factor): **`info`**

96

MPI Datatypes

- In MPI-1, the data to be sent or received is described by a triple (address, count, datatype)
- Datatypes in MPI are of two kinds: basic and derived
- Basic datatypes are those that correspond to the basic datatypes in the host programming language, e.g., MPI_INT, MPI_FLOAT
- Derived datatypes can be constructed out of multiple copies of basic datatypes with gaps

97

Derived Datatype Constructors

- Datatype constructor functions can be used recursively to create
  - » a contiguous array of datatypes
  - » a strided array of blocks of datatypes
  - » an indexed array of blocks of datatypes
  - » an arbitrary structure of datatypes
  - » datatypes describing portions of matrices
- Therefore, any noncontiguous access pattern can be represented in terms of an MPI derived datatype

98

Noncontiguous Accesses

- Noncontiguous data in both memory and file is specified using MPI datatypes, both predefined and derived
- Data layout in memory specified on each call, as in message-passing
- Data layout in file is defined by a file *view*
- A process can access data only within its view
- Views can be changed

99

Basic Data Access Functions

- Individual file pointer: **`MPI_File_read`**
- Explicit file offset: **`MPI_File_read_at`**
- Shared file pointer: **`MPI_File_read_shared`**
- Nonblocking I/O: **`MPI_File_iread`**
- Similarly for writes

100

Collective I/O Operations

- Blocking:

  `MPI_File_read_all`

- Non-blocking:

  `MPI_File_read_all_begin`

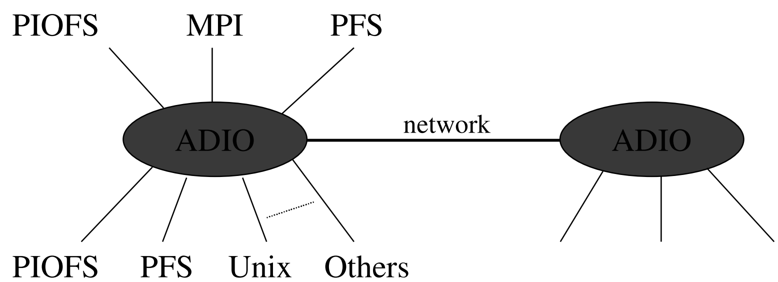
  `MPI_File_read_all_end`

Some Other MPI-IO Features

- Multiple file data representations
  - native
  - internal
  - portable
- Built-in file type constructors for distributed arrays and subarrays.

ROMIO - A Portable Implementation of MPI-IO

- Implementation strategy: an abstract device for I/O (ADIO)
- Tested for low overhead
- Can use any MPI implementation (MPICH, vendor)



103

Two-Phase Collective I/O

- ROMIO has an optimized implementation of two-phase collective I/O
- I/O is done in two phases: an I/O phase and a communication phase
- In the I/O phase, data is read/written in large chunks to minimize I/O latency
- Message-passing among compute nodes is used to redistribute data as needed

104

Current Status of ROMIO

- Latest version, 1.0.1, released in July 1998
- A substantial portion of the standard has been implemented:
  - » noncontiguous accesses in memory and file
  - » collective I/O
  - » hints
  - » asynchronous I/O
- Supports large files --- greater than 2 Gbytes
- Works with MPICH and vendor MPI implementations (SGI, HP, NEC)
- Included as part of HP MPI and MPICH
- Web site: http://www.mcs.anl.gov/romio

105

Hints used in ROMIO

- cb_buffer_size
- cb_nodes
- striping_unit
- striping_factor

MPI-2 predefined hints

- ind_rd_buffer_size
- ind_wr_buffer_size

New Algorithm Parameters

- start_iodevice
- pfs_svr_buf

Platform-specific hints

106

Using MPI-IO Functions

- Any application as a particular “I/O access pattern” based on its I/O needs
- The same access pattern can be presented to the I/O system in different ways depending on what I/O functions are used and how
- We classify the different ways of expressing I/O access patterns in MPI-IO into four “levels”, level 0 -- level 3. (SC98 paper)
- Each level represents increasing amounts of data per I/O request

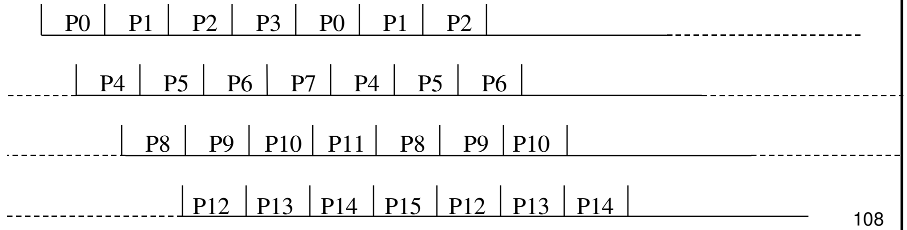
Example: Distributed Array Access

Large array distributed among 16 processes

P0	P1	P2	P3
P4	P5	P6	P7
P8	P9	P10	P11
P12	P13	P14	P15

Each square represents a subarray in the memory of a single process

Access Pattern in the file



Level-0 Access

- Each process makes one independent read request for each row in the local array (as in Unix)

```
MPI_File_open(..., file, ..., &fh)
for (i=0; i<n_local_rows; i++) {
      MPI_File_seek(fh, ...)
      MPI_File_read(fh, row[i], ...)
}
MPI_File_close(&fh)
```

109

Level-1 Access

- Similar to level 0, but each process uses collective I/O functions

```
MPI_File_open(MPI_COMM_WORLD, file, ..., &fh)
for (i=0; i<n_local_rows; i++) {
      MPI_File_seek(fh, ...)
      MPI_File_read_all(fh, row[i], ...)
}
MPI_File_close(&fh)
```

110

Level-2 Access

- Each process creates a derived datatype to describe the noncontiguous access pattern, defines a file view, and calls independent I/O functions

```
MPI_Type_create_subarray(..., &subarray, ...)
MPI_Type_commit(&subarray)
MPI_File_open(..., file, ..., &fh)
MPI_File_set_view(fh, ..., subarray, ...)
MPI_File_read(fh, localarray, ...)
MPI_File_close(&fh)
```

111

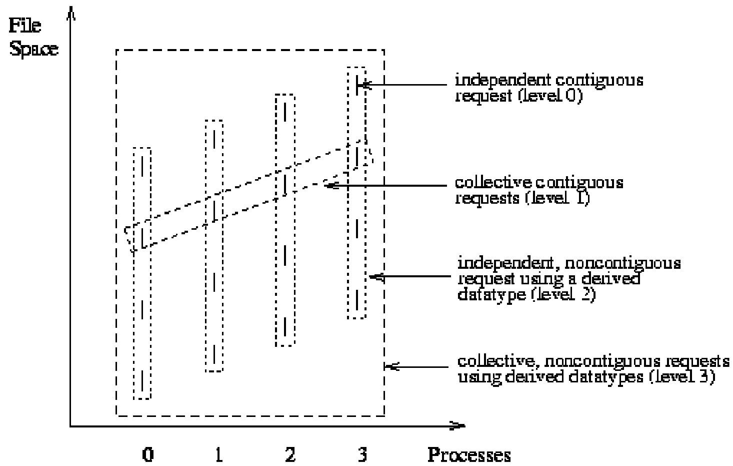
Level-3 Access

- Similar to level 2, except that each process uses collective I/O functions

```
MPI_Type_create_subarray(..., &subarray, ...)
MPI_Type_commit(&subarray)
MPI_File_open(MPI_COMM_WORLD, file, ..., &fh)
MPI_File_set_view(fh, ..., subarray, ...)
MPI_File_read_all(fh, localarray, ...)
MPI_File_close(&fh)
```

112

The Four Levels of Access



113

Optimizations

- Given complete access information, an implementation can perform optimizations such as:
  - » Data Sieving: Read large chunks and extract what is really needed
  - » Collective I/O: Merge requests of different processes into larger requests
  - » Improved prefetching and caching

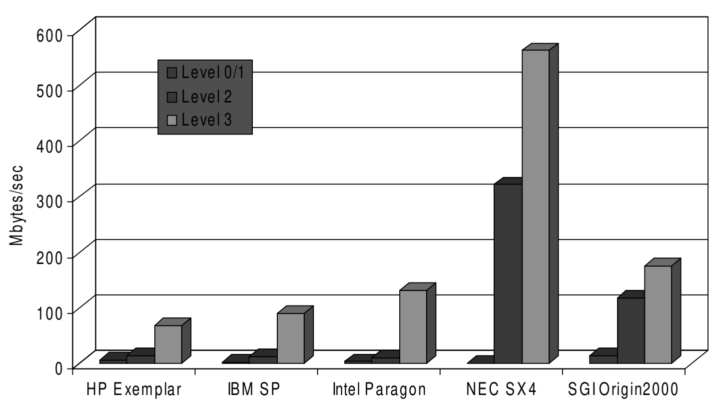
114

Performance Results

- Distributed array access
- Unstructured code from Sandia
- On five different parallel machines:
  - » HP Exemplar
  - » IBM SP
  - » Intel Paragon
  - » NEC SX-4
  - » SGI Origin2000

115

Distributed Array Access: Read Bandwidth

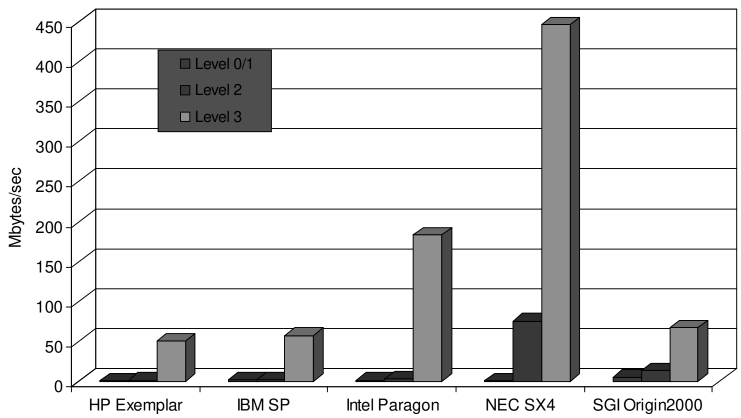


64 procs | 64 procs | 256 procs | 8 procs | 32 procs

Array size: 512 x 512 x 512

116

Distributed Array Access: Write Bandwidth

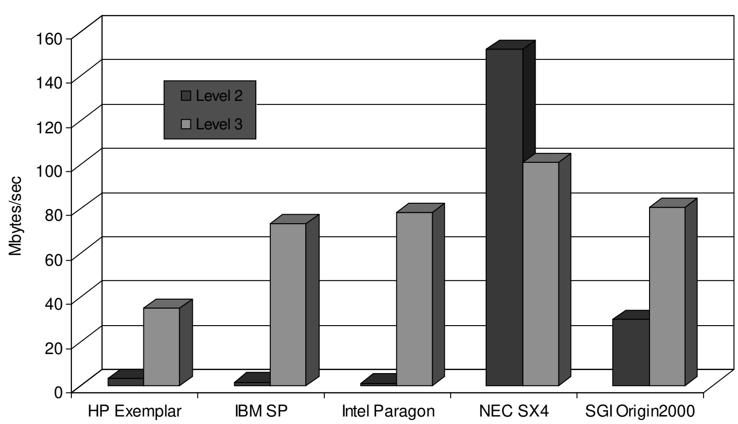


64 procs 64 procs 256 procs 8 procs 32 procs

Array size: 512 x 512 x 512

117

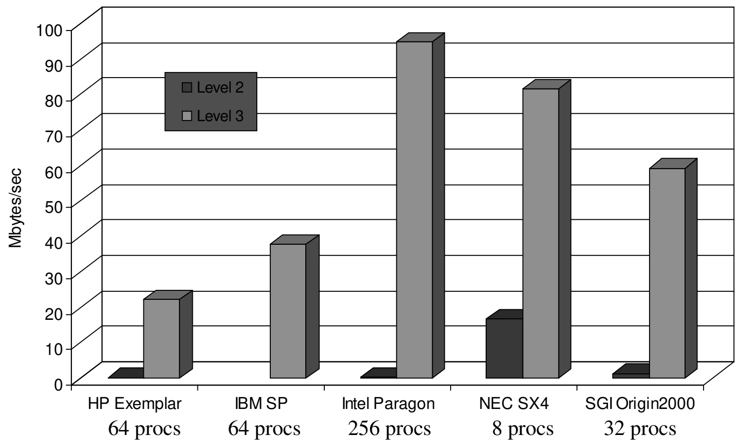
Unstructured Code: Read Bandwidth



64 procs 64 procs 256 procs 8 procs 32 procs

118

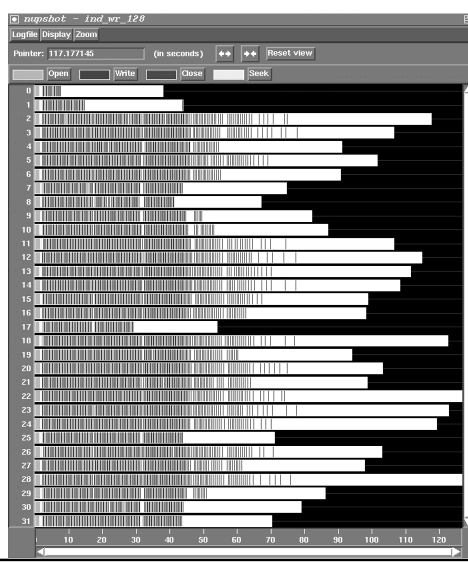
Unstructured Code: Write Bandwidth



119

Independent Writes

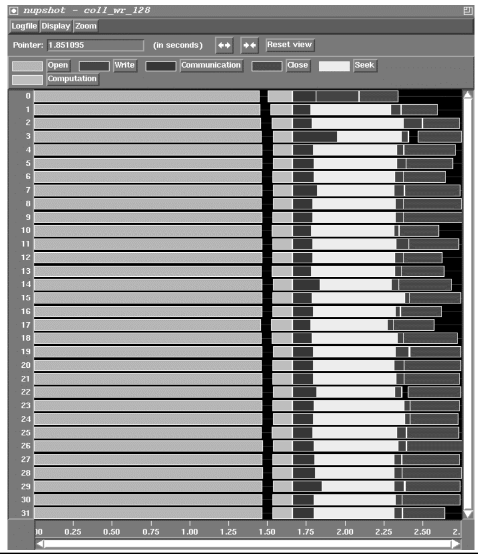
- On Paragon
- Lots of seeks and small writes
- Time shown = 130 seconds



20

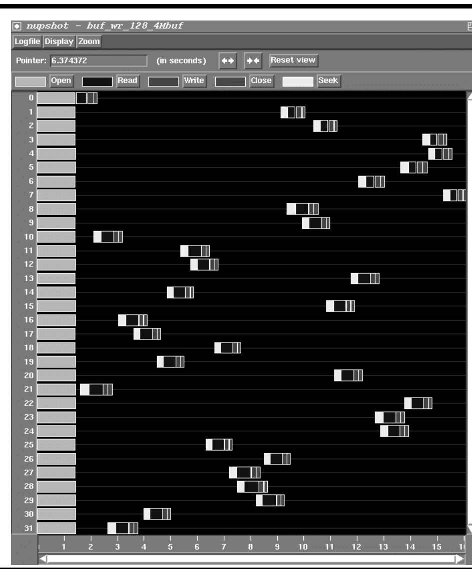
Collective Write

- On Paragon
- Computation and communication precede seek and write
- Time shown = 2.75 seconds



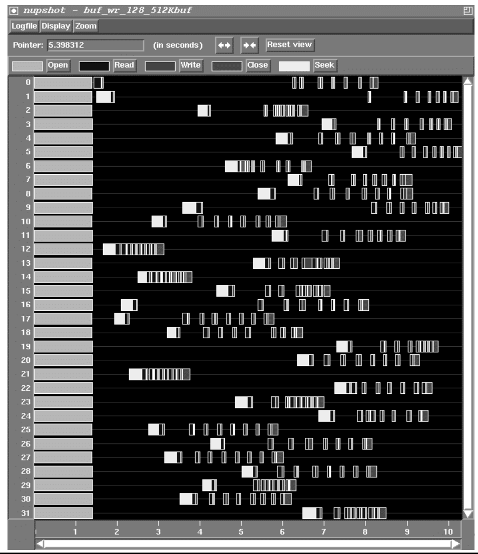
Independent Writes with Data Sieving

- On Paragon
- Use large blocks, write multiple “real” blocks plus “gaps”
- Requires lock, read, modify, write, unlock for writes
- 4 MB blocks
- Time = 16 seconds



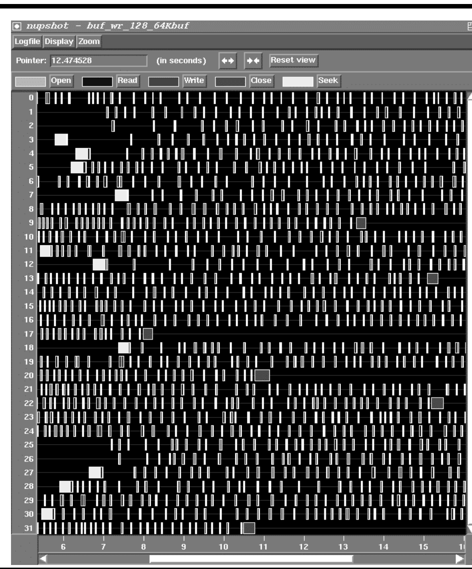
Changing the Block Size

- Smaller blocks mean less contention, therefore more parallelism
- 512 KB blocks
- Time = 10.2 seconds



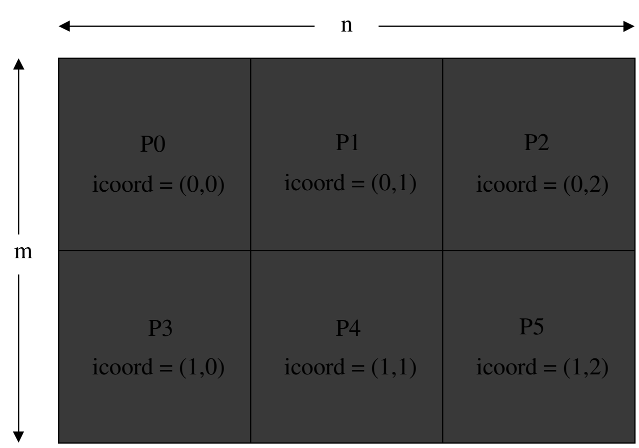
Data Sieving with Small Blocks

- If the block size is too small, however, the increased parallelism doesn't make up for the many small writes
- 64 KB blocks
- Time = 21.5 seconds



Detailed Example: Distributed Array Access

- Global array size: *m* x *n*
- Array is distributed on a 2D grid of processes: *nproc(1) x nproc(2)*
- Coordinates of a process in the grid: *icoord(1), icoord(2)*   (0-based indices)
- Local array stored in file in a layout corresponding to global array in column-major (Fortran) order
- Two cases
  - » local array is stored contiguously in memory
  - » local array has “ghost area” around it in memory

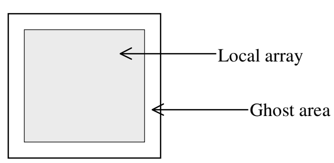


nproc(1) = 2, nproc(2) = 3

CASE I: LOCAL ARRAY CONTIGUOUS IN MEMORY

```
iarray_of_sizes(1) = m
iarray_of_sizes(2) = n
iarray_of_subsizes(1) = m/nproc(1)
iarray_of_subsizes(2) = n/nproc(2)
iarray_of_starts(1) = icoord(1) * iarray_of_subsizes(1)      ! 0-based, even in Fortran
iarray_of_starts(2) = icoord(2) * iarray_of_subsizes(2)      ! 0-based, even in Fortran
ndims = 2

call MPI_TYPE_CREATE_SUBARRAY(ndims, iarray_of_sizes, iarray_of_subsizes,
          iarray_of_starts, MPI_ORDER_FORTRAN, MPI_REAL, ifiletype, ierr)
call MPI_TYPE_COMMIT(ifiletype, ierr)

call MPI_FILE_OPEN(MPI_COMM_WORLD, '/home/thakur/test',
          MPI_MODE_CREATE + MPI_MODE_RDWR, MPI_INFO_NULL, ifh, ierr)

call MPI_FILE_SET_VIEW(ifh, 0, MPI_REAL, ifiletype, 'native', MPI_INFO_NULL, ierr)

ilocal_size = iarray_of_subsizes(1) * iarray_of_subsizes(2)
call MPI_FILE_WRITE_ALL(ifh, local_array, ilocal_size, MPI_REAL, istatus, ierr)

call MPI_FILE_CLOSE(ifh, ierr)
```

CASE II: LOCAL ARRAY HAS GHOST AREA OF SIZE 2 ON EACH SIDE




```
!  Create derived datatype describing layout of local array in memory

iarray_of_sizes(1) = m/nproc(1) + 4
iarray_of_sizes(2) = n/nproc(2) + 4
iarray_of_subsizes(1) = m/nproc(1)
iarray_of_subsizes(2) = n/nproc(2)
iarray_of_starts(1) = 2                        ! 0-based, even in Fortran
iarray_of_starts(2) = 2                        ! 0-based, even in Fortran
ndims = 2

call MPI_TYPE_CREATE_SUBARRAY(ndims, iarray_of_sizes, iarray_of_subsizes,
          iarray_of_starts, MPI_ORDER_FORTRAN, MPI_REAL, imemtype, ierr)
call MPI_TYPE_COMMIT(imemtype, ierr)
```

Continued...

CASE II continued...

```
! Create derived datatype to describe layout of local array in the file

iarray_of_sizes(1) = m
iarray_of_sizes(2) = n
iarray_of_subsizes(1) = m/nproc(1)
iarray_of_subsizes(2) = n/nproc(2)
iarray_of_starts(1) = icoord(1) * iarray_of_subsizes(1)      ! 0-based, even in Fortran
iarray_of_starts(2) = icoord(2) * iarray_of_subsizes(2)      ! 0-based, even in Fortran
ndims = 2

call MPI_TYPE_CREATE_SUBARRAY(ndims, iarray_of_sizes, iarray_of_subsizes,
          iarray_of_starts, MPI_ORDER_FORTRAN, MPI_REAL, ifiletype, ierr)
call MPI_TYPE_COMMIT(ifiletype, ierr)

! Open the file, set the view, and write

call MPI_FILE_OPEN(MPI_COMM_WORLD, '/home/thakur/test',
          MPI_MODE_CREATE + MPI_MODE_RDWR, MPI_INFO_NULL, ifh, ierr)
call MPI_FILE_SET_VIEW(ifh, 0, MPI_REAL, ifiletype, 'native', MPI_INFO_NULL, ierr)
call MPI_FILE_WRITE_ALL(ifh, local_array, 1, imemtype, istatus, ierr)
call MPI_FILE_CLOSE(ifh, ierr)
```

Summary

- MPI-IO can help users achieve better I/O performance
- On their part, users must use some of the special features of MPI-IO
- In particular, when accesses are noncontiguous, users must create derived datatypes, define file views, and use the collective I/O functions

130