MPI-2: Extensions to the Message Passing Interface

Anthony Skjellum[1]
Purushotham Bangalore[1], Shane Hebert[1]
Rolf Rabenseifner[2]

Overview

- General Comments
- Features/Structure of MPI-1, MPI-2
- MPI 1.2 = Clarifications to MPI-1.1
- MPI-2 Functionality - Chapter by Chapter
- The Implementation Gap
- Summary
- Resources

General Comments

- About the tutorial presenters
- MPI-2 … a breadth-first introduction
- MPI-2 … who needs these extensions?
- What we cover in 2 hours
- What we don't cover in this time / further information
- Status of MPI-2 as a real programming environment, and changes expected
- Moving forward beyond this tutorial
- About [*comments — not part of the standard*]

Recapitulation of MPI 1.x

- Reliable Message Passing
- Static Process Model established at start
- Services
  - point-to-point transfers (blocking, non-blocking)
  - collective transfers (blocking)
  - communicators (safety, subsetting, bi-partite)
  - topologies for convenient programming
  - miscellaneous functionality (e.g. timing)
- Established by MPI Forum in 1993
- In wide use

MPI-2 Addresses // does not …

- Dynamically resizing the parallel application
- Parallel Client-Server Model
- Distributed shared memory (put/get) model
- Input-output to "devices," "parallel files"
- Active Messages (lightweight RPC)
- Virtual global shared memory support
- Classes of collective services: non-blocking, bi-partite, in-place options
- QOS, timeouts on operations
- Fault Tolerant Features
- Thread interoperability
- Full inter-language (C/Fortran/C++) interoperability
- Full exploitation of "persistent operations" for performance

MPI-2 Features

- Clarifications and corrections to MPI 1.1
- Process creation and management
- One-sided communications
- Extended collective communications
- Input/Output on files
- Miscellany
- External interfaces
- Language bindings

General rule: No modifications to MPI 1.1

MPI-2 Documents

- The MPI-2 Forum has released:
  - MPI-2: Extensions to the Message-Passing Interface (July 18,1997)
    - MPI 1.2 with
      - version handling
      - clarifications to MPI 1.1
    - MPI 2.0 with
      - new features
  - MPI-2 Journal of Development (July 18,1997)
    - ideas for further MPI standardization, e.g.
      - real-time MPI
      - cluster attributes for IMPI
      - general split collective communication
      - additional datatype constructors

7

MPI-1.2

- New function to obtain version of the MPI Standard implemented
- Compile time information
  - integer MPI_VERSION=1, MPI_SUBVERSION=2
- Runtime information
  - MPI_GET_VERSION(*version*, *subversion*)
- MPI_GET_VERSION can be called before MPI_INIT and after MPI_FINALIZE
- Clarifications and corrections to MPI-1.1
  - pointer to MPI-2 Chapter 10.2.2 *Problems with Fortran Bindings for MPI*

8

MPI 1.1 Clarifications, I.

- MPI_INITIALIZED
  - behavior not affected by MPI_FINALIZE
- MPI_FINALIZE
  - user must ensure the completion of all pending communications (locally) before calling finalize
  - is collective on MPI_COMM_WORLD
  - may abort all processes except "rank==0" in MPI_COMM_WORLD
- Status object after MPI_WAIT/MPI_TEST
- MPI_INTERCOMM_CREATE
- MPI_INTERCOMM_MERGE

9

MPI 1.1 Clarifications, II.

- Bindings for MPI_TYPE_SIZE
  - output argument in C is of type `int`
- MPI_REDUCE
  - the `datatype` and `op` for predefined operations must be same for all processes
- MPI_PROBE and MPI_IPROBE
- Attribute callback functions error behavior
- Other minor corrections
  - with nonblocking & persistent communications and MPI_Address:
    - Fortran problems with data copying and sequence association
    - Fortran problems with register optimization

10

Process Creation and Management

Motivation

- MPI-1 uses a static process model
  - all processes are present in `MPI_COMM_WORLD`
  - new processes can not be added
  - existing processes can not be removed
- Several applications require a dynamic process model
  - number of processors required is defined at runtime
  - all processes are not required all the time
  - client/server applications

12

MPI-2 Process Model

- Allows creation and termination of new processes from an existing MPI application
- Provides a mechanism for the newly created processes to communicate with the existing MPI application
- Allows two separate existing MPI applications to communicate with each other
- MPI-1 static model is considered as a special case of the MPI-2 process model

13

MPI_COMM_SPAWN

MPI_COMM_SPAWN(command, argv, maxprocs, info, root, comm, *intercomm, array_of_errcodes*)

- Creates `maxprocs` instances of `command` and returns `intercomm`
- Collective over `comm` (parents) and the spawned processes (children)
- Children have separate `MPI_COMM_WORLD`
- Children can optionally obtain `intercomm` using `MPI_COMM_GET_PARENT`
- `intercomm` for parents-children-communication
- MPI_COMM_SPAWN_MULTIPLE eg, if children on different hosts

14

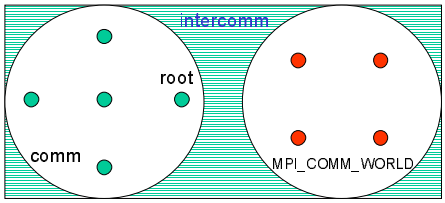
MPI_COMM_GET_PARENT

- Returns the parent intercommunicator for the process that was created using MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTIPLE
- Returns MPI_COMM_NULL
  - if the process was not spawned
  - the parent is freed or disconnected
- MPI_INTERCOMM_MERGE allows creation of intra-communicator over parent and children processes

SPMD for parent and children

15

Get the intercomm, I.

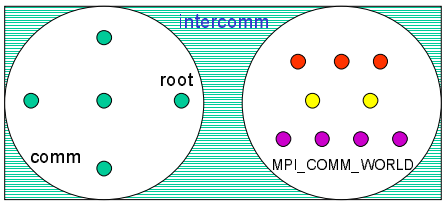


Parents:
MPI_COMM_SPAWN (……, root,comm, *intercomm*,…)

Children:
MPI_Init(…)
MPI_COMM_GET_PARENT (*intercomm*)

16

Get the intercomm, II.



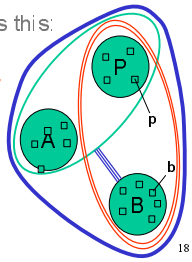
Parents:
MPI_COMM_SPAWN _MULTIPLE (3, ……, root,comm, *intercomm*,…)

Children:
MPI_Init(…)
MPI_COMM_GET_PARENT (*intercomm*)

17

Multi-merging, a Challenge

- If a comm. P spawns A and B sequentially, how can P, A and B communicate in a single intracomm?
- The following sequence supports this:
  - P+A merge to form intracomm PA
  - P+B merge to form intracomm PB
  - PA and B create intercomm PA+B [using PB as peer, with p, b as leaders]
  - PA+B merge to form intracomm PAB
- This is not very easy, but does work



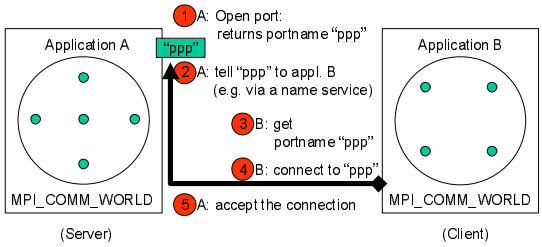
18

MPI_Info Object

- An `MPI_Info` is an opaque object that consists of a set of (key,value) pairs
  - both key and value are strings
  - a key should have a unique value
  - several keys are reserved by standard / implementation
  - portable programs may use `MPI_INFO_NULL` as the info argument, or sets of vendor keys
  - Several sets of vendor-specific keys may be used
- Allows applications to pass environment-specific information
- Several functions provided to manipulate the info objects
- Used in: *Process Creation*, *Window Creation*, *MPI-I/O*

19

Establishing Communication



Result:
An intercommunicator between both original communicators

20

Another way

- Another way to establish MPI communication
- MPI_COMM_JOIN(fd, *intercomm*)
- joins by an intercommunicator
- two independent MPI processes
- that are connected with Berkley Sockets of type SOCK_STREAM

21

Singleton INIT

- High quality MPI's will allow single processes to start, call MPI_INIT(), and later join in with other MPI programs
- This approach supports
  parallel plug-ins to sequential APPs
  other transparent uses of MPI
- Provides a means for using MPI without having to have the "main" program be MPI specific.

22

MPI-2 "Spawn" does not cover

- no resource management issues
- no process management (kill, notify)
- fault tolerance (if a process dies)

23

MPI-Spawn: Summary

- Provides functionality
  - to create new processes and establish communication with the newly created process
  - to establish communication between two existing MPI applications
- Whenever possible start all processes at once instead of using spawn
- Calling spawn multiple times is not equivalent to calling spawn_multiple
- Info object provides optimization hints to the implementation
- [*never available on Cray T3E ???*]

24

One-sided Communications

Synchronization Taxonomy

- Message Passing:
  explicit transfer, implicit synchronization, implicit cache operations

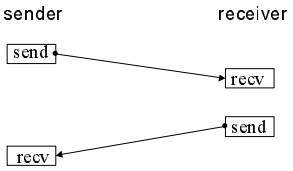
Access to other processes' memory:

- **1-sided**
  explicit transfer, explicit synchronization, implicit cache operations (problem!)
- Shared Memory
  implicit transfer, explicit synchronization, implicit cache operations
- shmem interface
  explicit transfer, explicit synchronization, explicit cache operations

26

Cooperative Communication

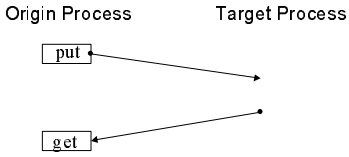
- MPI-1 supports cooperative or 2-sided communication
- Both sender and receiver processes must participate in the communication



27

One-sided Communication

- Communication parameters for both the sender and receiver are specified by one process (origin)
- User must impose correct ordering of memory accesses



28

One-sided Operations

- Initialization
  - MPI_ALLOC_MEM, MPI_FREE_MEM
  - MPI_WIN_CREATE, MPI_WIN_FREE
- Remote Memory Access (RMA, nonblocking)
  - MPI_PUT
  - MPI_GET
  - MPI_ACCUMULATE
- Synchronization
  - MPI_WIN_FENCE (like a barrier)
  - MPI_WIN_POST / MPI_WIN_START / MPI_WIN_COMPLETE / MPI_WIN_WAIT
  - MPI_WIN_LOCK / MPI_WIN_UNLOCK

29

Window Creation

- Specifies the region in memory (already allocated) that can be accessed by remote processes
- Collective call over all processes in the intracommunicator
- Returns an opaque object of type `MPI_Win` which can be used to perform the remote memory access (RMA) operations

MPI_WIN_CREATE(base_address, win_size, disp_unit, info, comm, *win*)

30

MPI_Put

- Performs an operation equivalent to a send by the origin process and a matching receive by the target process
- The origin process specifies the arguments for both the origin and target process
- The target buffer is at address target_addr = win_base + target_disp * disp_unit

```
MPI_PUT(origin_address, origin_count, origin_datatype,
target_rank, target_disp, target_count, target_datatype,
win)
```

Use only basic datatypes or derived datatypes without byte-length displacements!

31

MPI_Get

- Similar to the put operation, except that data is transferred from the target memory to the origin process
- To complete the transfer a synchronization call must be made on the window involved
- The local buffer should not be accessed until the synchronization call is completed

```
MPI_GET(origin_address, origin_count, origin_datatype,
target_rank, target_disp, target_count, target_datatype,
win)
```

32

MPI_Accumulate

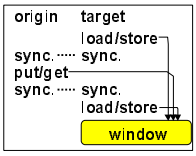
- Accumulates the contents of the origin buffer to the target area specified using the predefined operation op
- User-defined operations cannot be used
- Accumulate is atomic: many accumulates can be done by many origins to one target
  -> [*may be very expensive*]

```
MPI_ACCUMULATE(origin_address, origin_count,
origin_datatype, target_rank, target_disp, target_count,
target_datatype, op, win)
```

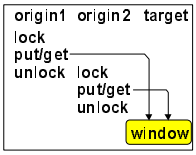
33

Synchronization Calls

- Active target communication
  - communication paradigm similar to message passing model
  - target process participates only in the synchronization
  - fence or post-start-complete-wait



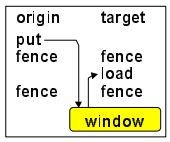
- Passive target communication
  - communication paradigm closer to shared memory model
  - only the origin process is involved in the communication
  - lock/unlock



34

MPI_Win_fence

- Synchronizes RMA operations on specified window
- Collective over the window
- Like a barrier
- Should be used before and after calls to put, get, and accumulate
- The `assert` argument is used to provide optimization hints to the implementation
- Used for active target communication

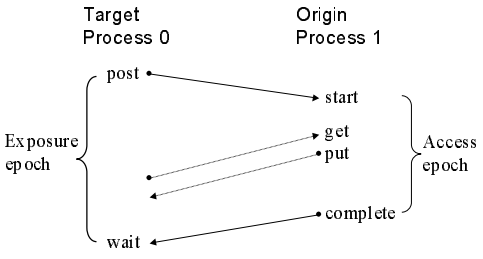



```
MPI_WIN_FENCE(assert, win)
```

35

Start/Complete and Post/Wait, I.

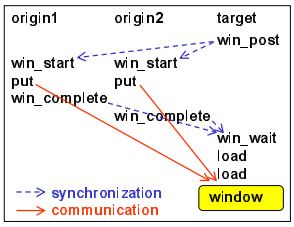
- Used for active target communication with weak synchronization



36

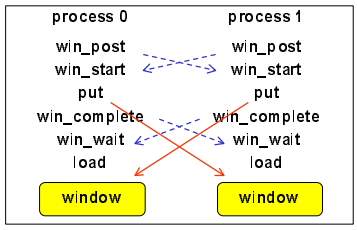
Start/Complete and Post/Wait, II.

- RMA (put, get, accumulate) are finished
  - locally after win_complete
  - at the target after win_wait
- local buffer must not be reused before RMA call locally finished
- communication partners must be known
- no atomicity for overlapping "puts"
- assertions may improve efficiency
  --> give all information you have



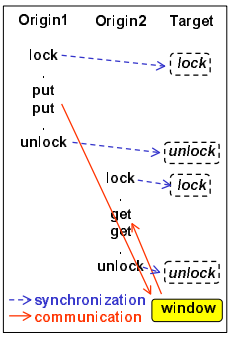
Start/Complete and Post/Wait, III.

- symmetric communication possible, only win_start and win_wait may block



Lock/Unlock

- Does not guarantee a sequence
- agent may be necessary on systems without (virtual) shared memory
- Portable programs can use lock calls to windows in memory allocated **only** by **`MPI_ALLOC_MEM`**
- RMA completed at both origin and target



MPI_ALLOC_MEM

MPI_ALLOC_MEM (size, info, *baseptr*)

MPI_FREE_MEM (base)

```
REAL A
POINTER (P, A(100))   ! no memory is allocated
INTEGER (KIND=MPI_ADDRESS_KIND) Size
INTEGER Lng_real, Win, IERR)
CALL MPI_TYPE_EXTENT(MPI_REAL, Lng_real, IERR)
Size = 100*Lng_real
CALL MPI_ALLOC_MEM(Size, MPI_INFO_NULL, P, IERR)
CALL MPI_WIN_CREATE(A, Size, Lng_real, MPI_INFO_NULL,
                    MPI_COMM_WORLD, Win, IERR)
...
CALL MPI_WIN_FREE(Win, IERR)
CALL MPI_FREE_MEM(A, IERR)
```


Fortran Problems with 1-Sided

Source of Process 1	Source of Process 2	Executed in Process 2
bbbb = 777	buff = 999	register_A := 999
call MPI_WIN_FENCE	call MPI_WIN_FENCE	
call MPI_PUT(bbbb into buff of process 2)		stop application thread buff := 777 in PUT handler continue application thread
call MPI_WIN_FENCE	call MPI_WIN_FENCE	
	ccc = buff	ccc := register_A

- Fortran register optimization
- Result ccc=999, but expected ccc=777
- How to avoid: (see MPI-2, Chap. 6.7.3)
  - window memory declared in COMMON blocks
    i.e. MPI_ALLOC_MEM cannot be used
  - declare window memory as VOLATILE
    (non-standard, disables compiler optimization)

One-sided: Summary

- Three one-sided communication primitives provided
- Several synchronization options supported
- User must ensure that there are no conflicting accesses
- For better performance **assertions** should be used with fence/start/post operations

Extended Collective Operations

New Features

- Two communicator construction operations extended to intercommunicators
  - MPI_COMM_CREATE
  - MPI_COMM_SPLIT
- Collective communication extended to intercommunicators

- Provision to specify "in place" buffers for collective operations on intracommunicator
- Two new collective routines added
  - generalized all-to-all
  - exclusive scan

44

Non-blocking Collective???

- True non-blocking collective was proposed in MPI-2, but dropped
- Two-phase (aka two-face) operations were proposed, but then only supported with the parallel I/O chapter
- Arguments were made that non-blocking collective message passing not needed
- This seems to be an incorrect argument, but it leaves MPI-2 without non-blocking collective message passing…

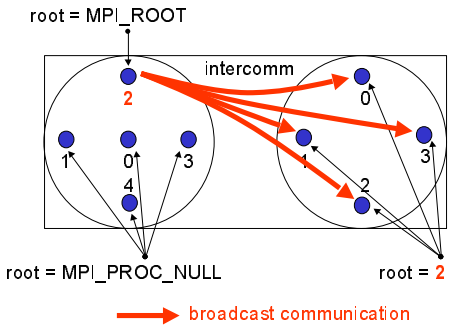
45

Intercommunicators

- Extending communicator constructors
  - create intercommunicator as sub-communicator of an existing intercommunicator
  - split an intercommunicator
- Extended for intercommunicators:
  - all collective communication routines, except MPI_SCAN and MPI_EXSCAN
  - examples on the following slides for
    - rooted operations
    - all-to-all operations

46

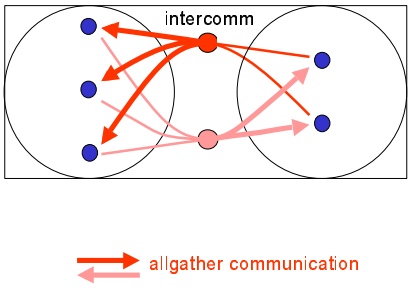
MPI_Bcast on **inter**comm.



47

MPI_Allgather on **inter**comm.

remember: allgather = gather+broadcast



48

"In place" Buffer Specification, I.

- MPI-1 requires separate send and receive buffers for collective operations
- In MPI-2 collective operations can use the same buffer for send and receive by specifying `MPI_IN_PLACE` as the send or receive buffer argument
- The problem with argument aliasing in Fortran is eliminated by using this special value
- Some collective operations do not support the "in place" option (e.g., Alltoall, Alltoallv)

49

"In place" Buffer Specification, II.

The `MPI_IN_PLACE` has two meanings:

- to prohibit the local copy:
  - GATHER(V), SCATTER(V) at root node
  - ALLGATHER(V) at any node
- to overwrite input buffer with the result: (sendbuf=`MPI_IN_PLACE`, input is taken from recvbuf, which is then overwitten)
  - REDUCE at root
  - ALLREDUCE, REDUCE_SCATTER, SCAN at any node

50

Generalized All-to-all

- The most general form of all-to-all
- Allows separate specification of count, displacement, and datatype
- Displacement is specified in terms of no. of bytes to allow maximum flexibility
- Useful for matrix transpose and corner-turn operations

MPI_Alltoallw(sendbuf, sendcounts, sdispls, **sendtypes**, *recvbuf*, recvcounts, rdispls, **recvtypes**, comm)

recvtypes, sendtypes now both arrays

51

Input/Output

Motivation, I.

- Many parallel applications need
  - coordinated parallel access to a file by a group of processes
  - simultaneous access
  - all processes may read/write many (small) non-contiguous pieces of the file, i.e. the data may be distributed amongst the processes according to a partitioning scheme
  - all processes may read the same data
- Efficient collective I/O based on
  - fast physical I/O by several processors, e.g. striped
  - distributing (small) pieces by fast message passing

53

Motivation, II.

- Analogy: writing / reading a file is like sending/receiving a message
- Handling parallel I/O needs
  - handling groups of processes -> MPI groups
  - collective operations -> file handle defined like communicators
  - non-blocking operations to overlap computation & I/O -> MPI_I..., MPI_Wait, ... & new **split** collective interface
  - non-contiguous access -> MPI derived datatypes

54

I/O Features

- Provides a high-level interface to support
  - data file partitioning among processes
  - transfer global data between memory and files (collective I/O)
  - asynchronous transfers
  - strided access
- MPI derived datatypes used to specify common data access patterns for maximum flexibility and expressiveness

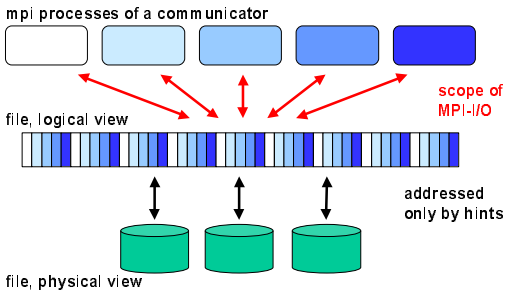
55

MPI-I/O, Principles

- MPI file contains elements of a single MPI datatype (etype)
- partitioning the file among processes with an access template (filetype)
- all file accesses transfer to/from a contiguous or non-contiguous user buffer (MPI datatype)
- non-blocking / blocking and collective / individual read / write routines
- individual and shared file pointers, explicit offsets
- automatic data conversion in heterog. systems
- file interoperability with external representation

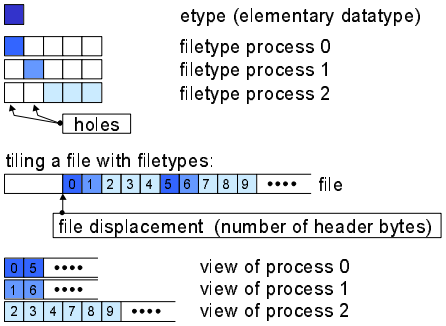
56

Logical view / Physical view



57

Definitions



58

Comments on Definitions

file - an ordered collection of typed data items

etypes - is the unit of data access and positioning / offsets
- can be any basic or derived datatype (with non-negative, monotonically non-decreasing, non-absolute displacem.)
- generally contiguous, but need not be
- typically same at all processes

filetypes - the basis for partitioning a file among processes
- defines a template for accessing the file
- different at each process
- the etype or derived from etype (displacements: non-negative, monoton. non-decreasing, non-abs., multiples of etype extent)

view - each process has its own view, defined by: a displacement, an etype, and a filetype.
- The filetype is repeated, starting at **displacement**

offset - position relative to current view, in units of etype

59

Opening an MPI File

- MPI_FILE_OPEN is collective over comm
- filename's namespace is implementation-dependent!
- filename must reference the same file on all processes
- process-local files can be opened by passing MPI_COMM_SELF as comm
- returns a file handle *fh* [*represents the file and the process group of* **comm**]

```
MPI_FILE_OPEN(comm, filename, amode, info, fh)
```

60

Access Modes

- same value of **amode** on all processes in MPI_FILE_OPEN
- Bit vector OR of integer constants (Fortran 77: +)
  - MPI_MODE_RDONLY - read only
  - MPI_MODE_RDWR - reading and writing
  - MPI_MODE_WRONLY - write only
  - MPI_MODE_CREATE - create if file doesn't exist
  - MPI_MODE_EXCL - error creating a file that exists
  - MPI_MODE_DELETE_ON_CLOSE - delete on close
  - MPI_MODE_UNIQUE_OPEN - file not opened concurrently
  - MPI_MODE_SEQUENTIAL - file only accessed sequentially: mandatory for sequential stream files (pipes, tapes, ...)
  - MPI_MODE_APPEND - all file pointers set to end of file [*caution: reset to zero by any subsequent MPI_FILE_SET_VIEW*]

61

File Info: Reserved Hints

- Argument in MPI_FILE_OPEN, MPI_FILE_SET_VIEW, MPI_FILE_SET_INFO
- reserved key values:
  - collective buffering
    - "collective_buffering": specifies whether the application may benefit from collective buffering
    - "cb_block_size": data access in chunks of this size
    - "cb_buffer_size": on each node, usually a multiple of block size
    - "cb_nodes": number of nodes used for collective buffering
  - disk striping (only relevant in MPI_FILE_OPEN)
    - "striping_factor": number of I/O devices used for striping
    - "striping_unit": length of a chunk on a device (in bytes)
- MPI_INFO_NULL may be passed

62

Closing and Deleting a File

- Close: collective

```
MPI_FILE_CLOSE(fh)
```

- Delete:
  - automatically by MPI_FILE_CLOSE if **amode=MPI_DELETE_ON_CLOSE** | ... was specified in MPI_FILE_OPEN
  - deleting a file that is not currently opened:

```
MPI_FILE_DELETE(filename, info)
```

[same implementation-dependent rules as in MPI_FILE_OPEN]

63

Other File Manipulation Routines

- Pre-allocating space for a file [*may be expensive*]
  MPI_FILE_ PREALLOCATE(fh, size)
- Resizing a file [*may speed up first writing on a file*]
  MPI_FILE_SET_SIZE(fh, size)
- Querying file size
  MPI_FILE_GET_SIZE(filename, *size*)
- Querying file parameters
  MPI_FILE_GET_GROUP(fh, *group*)
  MPI_FILE_GET_AMODE(fh, *amode*)
- File info object
  MPI_FILE_SET_INFO(fh, info)
  MPI_FILE_GET_INFO(fh, *info_used*)

64

File Views

- Provides a set of data visible and accessible from an open file
- A separate view of the file is seen by each process through triple := (displacement, etype, filetype)
- User can change a view during the execution of the program - but collective operation
- A linear byte stream, represented by the triple (0, MPI_BYTE, MPI_BYTE), is the default view

65

Set/Get File View

- Set view
  - changes the process's view of the data
  - local and shared file pointers are reset to zero
  - collective operation
  - etype and filetype must be committed
  - datarep argument is a string that specifies the format in which data is written to a file: "native", "internal", "external32", or user-defined
  - same etype extent and same datarep on all processes
- Get view
  - returns the process's view of the data

```
MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)
MPI_FILE_GET_VIEW(fh, disp, etype, filetype, datarep)
```

66

Data Representation, I.

- "native"
  - data stored in file identical to memory
  - on homogeneous systems no loss in precision or I/O performance due to type conversions
  - on heterogeneous systems loss of interoperability
  - no guarantee that MPI files accessible from C/Fortran
- "internal"
  - data stored in implementation specific format
  - can be used with homogeneous or heterogeneous environments
  - implementation will perform type conversions if necessary
  - no guarantee that MPI files accessible from C/Fortran

67

Data Representation, II.

- "external32"
  - follows standardized representation (IEEE)
  - all input/output operations are converted from/to the "external32" representation
  - files can be exported/imported between different MPI environments
  - due to type conversions from (to) native to (from) "external32" data precision and I/O performance may be lost
  - "internal" may be implemented as equal to "external32"
  - can be read/written also by non-MPI programs
- user-defined

68

5 Aspects of Data Access

- Direction: Read / Write
- Positioning [realized via routine names]
  - explicit offset (_AT)
  - individual file pointer (no positional qualifier)
  - shared file pointer (_SHARED or _ORDERED)
    (different names used depending on whether non-collective or collective)
- Coordination
  - non-collective
  - collective (_ALL)
- Synchronism
  - blocking
  - non-blocking (I) and split collective (_BEGIN, _END)
- Atomicity, [realized with a separate API: MPI_File_set_atomicity]
  - atomic
  - non-atomic

69

All Data Access Routines

Positioning	Synchronization	Non-collective	Collective
Explicit offsets	blocking	READ_AT WRITE_AT	READ_AT_ALL WRITE_AT_ALL
	non-blocking & split collective	IREAD_AT IWRITE_AT	READ_AT_ALL_BEGIN READ_AT_ALL_END WRITE_AT_ALL_BEGIN WRITE_AT_ALL_END
Individual file pointers	blocking	READ WRITE	READ_ALL WRITE_ALL
	non-blocking & split collective	IREAD IWRITE	READ_ALL_BEGIN READ_ALL_END WRITE_ALL_BEGIN WRITE_ALL_END
Shared file pointers	blocking	READ_SHARED WRITE_SHARED	READ_ORDERED WRITE_ORDERED
	non-blocking & split collective	IREAD_SHARED IWRITE_SHARED	READ_ORDERED_BEGIN READ_ORDERED_END WRITE_ORDERED_BEGIN WRITE_ORDERED_END

Read e.g. **MPI_FILE**_READ_AT

70

Explicit Offsets

e.g. MPI_FILE_READ_AT(fh,offset,*buf*,count,datatype,*status*)

- attempts to read `count` elements of `datatype`
- starting `offset` * units of `etype`
  from begin of view (= `displacement`)
- the sequence of basic datatypes of `datatype`
  (= signature of `datatype`)
  must match
  contiguous copies of the `etype` of the current view
- EOF can be detected by noting that the amount of data read is less than `count`
  - i.e. EOF is no error!
  - use MPI_GET_COUNT(status,datatype,*recv_count*)

71

Individual File Pointer, I.

e.g. MPI_FILE_READ(fh, *buf*,count,datatype,*status*)

- same as *"Explicit Offsets"*, except:
- the offset is the current value of the **individual file pointer** of the calling process
- the individual file pointer is updated by
  new_fp = old_fp + $\frac{\text{elements(datatype)}}{\text{elements(etype)}}$ * count
  i.e. it points to the next etype after the last one that will be accessed
  (*formula is not valid if EOF is reached*)

72

Individual File Pointer, II.

MPI_FILE_SEEK(fh, offset, whence)

- set individual file pointer fp:
  - set fp to offset – if whence=MPI_SEEK_SET
  - advance fp by offset – if whence=MPI_SEEK_CUR
  - set fp to EOF+offset – if whence=MPI_SEEK_EOF

MPI_FILE_GET_POSITION(fh, *offset*)
MPI_FILE_GET_BYTE_OFFSET(fh, offset, disp)

- to inquire offset
- to convert offset into byte displacement
  [*e.g. for `disp` argument in a new view*]

Shared File Pointer, I.

- same view at all processes mandatory!
- the offset is the current, *global* value of the **shared file pointer** of `fh`
- multiple calls [*e.g. by different processes*] behave as if the calls were serialized
- non-collective, e.g.

MPI_FILE_READ_SHARED(fh, *buf*, count, datatype, *status*)

- collective calls are *serialized* in the **order** of the processes' ranks, e.g.:

MPI_FILE_READ_ORDERED(fh,*buf*,count,datatype,*status*)

Shared File Pointer, II.

MPI_FILE_SEEK_SHARED(fh, offset, whence)
MPI_FILE_GET_POSITION_SHARED(fh, *offset*)
MPI_FILE_GET_BYTE_OFFSET(fh, offset, disp)

- same rules as with individual file pointers

Collective Data Access

- Explicit offsets / individual file pointer:
  - same as non-collective calls by all processes "of `fh`"
  - ***chance for best speed!!!***
- shared file pointer:
  - accesses are ordered by the ranks of the processes
  - optimization chance:
    - first, locations within the file for all processes can be computed
    - then parallel physical data access by all processes

Application Scenery, I.

- Scenery A:
  - Task: Each process has to read the whole file
  - Solution: MPI_FILE_READ_ALL
    = collective with individual file pointers,
    with same view (displacement+etype+filetype)
    on all processes
    [*internally: striped-reading by several process, only once from disk, then distributing with bcast*]
- Scenery B:
  - Task: The file contains a list of tasks,
    each task requires different compute time
  - Solution: MPI_FILE_READ_SHARED
    = non-collective with a shared file pointer
    (same view is necessary for shared file p.)

Application Scenery, II.

- Scenery C:
  - Task: The file contains a list of tasks,
    each task requires **the same** compute time
  - Solution: MPI_FILE_READ_ORDERED
    = **collective** with a **shared** file pointer
    (same view is necessary for shared file p.)
  - or: MPI_FILE_READ_ALL
    = **collective** with **individual** file pointers,
    different views: *filetype* with
    MPI_TYPE_CREATE_SUBARRAY(1,nproc,
    1, myrank, ..., datatype_of_task, *filetype*)
    [*internally: both may be implemented the same and equally with following scenery D*]

Application Scenery, III.

- Scenery D:
  - Task: The file contains a matrix, block partitioning, each process should get a block
  - Solution: generate different filetypes with MPI_TYPE_CREATE_DARRAY, the view on each process represents the block that should be read by this process, MPI_FILE_READ_AT_ALL with offset=0 (= collective with explicit offsets) reads the whole matrix collectively
    [*internally: striped-reading of contiguous blocks by several process, then distributed with "alltoall"*]

79

Non-blocking Data Access

e.g. MPI_FILE_IREAD(fh, *buf*, count, datatype, *request*)
MPI_WAIT(request, *status*)
MPI_TEST(request, *status*)

- analogous to MPI-1 non-blocking
- **non-standard** interface if ROMIO is used and not integrated:

MPIO_Request request
MPI_FILE_IREAD(fh, *buf*, count, datatype, *request*)
MPIO_WAIT(request, *status*)
MPIO_TEST(request, *status*)

80

Split Collective Data Access, I.

- the collective operation is **split** into two parts:
  - start the split collective operation

e.g. MPI_FILE_READ_ALL_BEGIN(fh, *buf*, count, datatype)

  - complete the operation and return the `status`

MPI_FILE_READ_ALL_END(fh, *buf*, *status*)

81

Split Collective Data Access, II.

- Rules and Restrictions:
  - the MPI_...BEGIN calls are collective
  - the MPI_...END calls are collective, too
  - only one active (pending) split or regular collective operation per file handle at any time
  - split collective does not match ordinary collective
  - same `buf` argument in MPI_...BEGIN and ..._END call
- Chance to overlap file I/O and computation
- but also a valid implementation:
  - does all work within the MPI_...BEGIN routine, passes status in the MPI_...END routine
  - passes arguments from MPI_...BEGIN to MPI_...END, does all work within the MPI_...END routine

82

Scenery – Split Collective

- Scenery A:
  - Task: Each process has to read the whole file
  - Solution:
    - MPI_FILE_READ_ALL_BEGIN
      = collective with individual file pointers, with same view (displacement+etype+filetype) on all processes
      [*internally: starting asynchronous striped-reading by several process*]
    - then computing some other initialization,
    - MPI_FILE_READ_ALL_END.
      [*internally: waiting until striped-reading finished, then distributing the data with bcast*]

83

MPI I/O Error Handling

- File handles have their own error handler
- Default is MPI_ERRORS_RETURN, i.e. **non-fatal**
  [vs message passing: MPI_ERRORS_ARE_FATAL]
- Default is associated with MPI_FILE_NULL
  [vs message passing: with MPI_COMM_WOLRD]
- MPI is *undefined* after first erroneous MPI call
- but a **high quality** implementation will support I/O error handling facilities

84

Example with Subarray, I.

```
!       real garray(20,30)
!HPF$ PROCESSORS  P(2, 3)
!HPF$ DISTRIBUTE garray(BLOCK,BLOCK)
        real larray(10,10) ;      integer (kind=MPI_OFFSET_KIND) disp ;  disp=0

        ndims=2 ; psizes(1)=2 ; period(1)=.false. ; psizes(2)=3 ; period(2)=.false.
        call MPI_CART_CREATE(MPI_COMM_WORLD, ndims, psizes, period,
                                                          .TRUE., comm, ierror)
        call MPI_COMM_RANK(comm, rank, ierror)
        call MPI_CART_COORDS(comm, rank, ndims, coords, ierror)

        gsizes(1)=20 ;  lsizes(1)= 10 ;  starts(1)=coords(1)*lsizes(1)
        gsizes(2)=30 ;  lsizes(2)= 10 ;  starts(2)=coords(2)*lsizes(2)
        call MPI_TYPE_CREATE_SUBARRAY(ndims, gsizes, lsizes, starts,
                        MPI_ORDER_FORTRAN, MPI_REAL, subarray_type, ierror)
        call MPI_TYPE_COMMIT(subarray_type , ierror)

        call MPI_FILE_OPEN(comm, 'exa_subarray_testfile', MPI_MODE_CREATE +
                                MPI_MODE_RDWR, MPI_INFO_NULL, fh, ierror)
        call MPI_FILE_SET_VIEW (fh, disp, MPI_REAL, subarray_type, 'native',
                                MPI_INFO_NULL, ierror)
        call MPI_FILE_READ_ALL(fh, larray, lsizes(1)*lsizes(2), MPI_REAL,
                                status, ierror)
```

85

Example with Subarray, II.

- All MPI coordinates and indices start with 0, even in Fortran, i.e. with MPI_ORDER_FORTRAN
- MPI indices (here `starts`) may differ (/) from Fortran indices
- Block distribution on 2*3 processes:

rank = 0 coords = (0, 0) starts = (0, 0) garray(1:10, 1:10) ≅ larray (1:10, 1:10)	rank = 1 coords = (0, 1) starts = (0, 10) garray(1:10, 11:20) ≅ larray (1:10, 1:10)	rank = 2 coords = (0, 2) starts = (0, 20) garray(1:10, 21:30) ≅ larray (1:10, 1:10)
rank = 3 coords = (1, 0) starts = (10, 0) garray(11:20, 1:10) ≅ larray (1:10, 1:10)	rank = 4 coords = (1, 1) starts = (10, 10) garray(11:20, 11:20) ≅ larray (1:10, 1:10)	rank = 5 coords = (1, 2) starts = (10, 20) garray(11:20, 21:30) ≅ larray (1:10, 1:10)

86

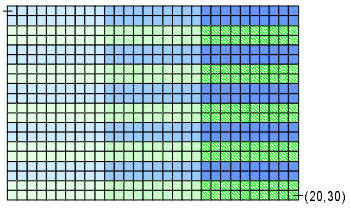
Example with Darray, I.

```
!       real garray(20,30)
!HPF$ PROCESSORS  P(2, 3)
!HPF$ DISTRIBUTE garray(CYCLIC(2),BLOCK)
        real larray(10,10) ;      integer (kind=MPI_OFFSET_KIND) disp ;  disp=0

        call MPI_COMM_SIZE(comm, size, ierror)
        ndims=2 ; psizes(1)=2 ; period(1)=.false. ; psizes(2)=3 ; period(2)=.false.
        call MPI_CART_CREATE(MPI_COMM_WORLD, ndims, psizes, period,
                                                          .TRUE., comm, ierror)
        call MPI_COMM_RANK(comm, rank, ierror)
        call MPI_CART_COORDS(comm, rank, ndims, coords, ierror)

        gsizes(1)=20 ;  distribs(1)= MPI_DISTRIBUTE_CYCLIC;  dargs(1)=2
        gsizes(2)=30 ;  distribs(2)= MPI_DISTRIBUTE_BLOCK;   dargs(2)=
                                                    MPI_DISTRIBUTE_DFLT_DARG
        call MPI_TYPE_CREATE_DARRAY(size, rank, ndims, gsizes, distribs, dargs,
                psizes, MPI_ORDER_FORTRAN, MPI_REAL, darray_type, ierror)
        call MPI_TYPE_COMMIT(darray_type , ierror)

        call MPI_FILE_OPEN(comm, 'exa_subarray_testfile', MPI_MODE_CREATE +
                                MPI_MODE_RDWR, MPI_INFO_NULL, fh, ierror)
        call MPI_FILE_SET_VIEW (fh, disp, MPI_REAL, darray_type, 'native',
                                MPI_INFO_NULL, ierror)
        call MPI_FILE_READ_ALL(fh, larray, 10*10, MPI_REAL, istatus, ierror)
```

87

Example with Darray, II.

- Cyclic distribution in first dimension with strips of length 2
- Block distribution in second dimension
- distribution of global garray onto the larray in each of the 6 processes
- garray:



88

Example with Darray, III.

- Cyclic distribution in first dimension with strips of length 2
- Block distribution in second dimension
- Processes' tasks:

rank = 0 coords = (0, 0) garray(1: 2, 5: 6, 9:10, 13:14, 17:18, 1:10) ≅ larray (1:10, 1:10)	rank = 1 coords = (0, 1) garray(1: 2, 5: 6, 9:10, 13:14, 17:18, 11:20) ≅ larray (1:10, 1:10)	rank = 2 coords = (0, 2) garray(1: 2, 5: 6, 9:10, 13:14, 17:18, 21:30) ≅ larray (1:10, 1:10)
rank = 3 coords = (1, 0) garray(3: 4, 7: 8, 11:12, 15:16, 19:20, 1:10) ≅ larray (1:10, 1:10)	rank = 4 coords = (1, 1) garray(3: 4, 7: 8, 11:12, 15:16, 19:20, 11:20) ≅ larray (1:10, 1:10)	rank = 5 coords = (1, 2) garray(3: 4, 7: 8, 11:12, 15:16, 19:20, 21:30) ≅ larray (1:10, 1:10)

89

Implementation-Restrictions, I.

- MPICH 1.1.2 , mpt.1.3.0.1 on CRAY-T3E, HP MPI 1.4, ...
  - with ROMIO 1.0.1
- ROMIO 1.0.1
  - without shared file pointer routines
  - without MPI_MODE_SEQUENTIAL (as amode in MPI_FILE_OPEN)
  - without split collective routines
  - "status" is not filled in any function
  - EOF is not detected while reading and file pointer is set behind EOF
  - i.e. no chance to detect EOF via status or MPI_FILE_GET_POSITION
  - nonblocking with MPIO_Request, MPIO_Wait, MPIO_Test
  - returns only ierror = MPI_SUCCESS or MPI_ERR_UNKNOWN
  - only "native" data representation
  - without registering user-defined representations

90

Implementation-Restrictions, II.

- mpt.1.3.0.1 on T3E
  - only with I/O chapter (based on ROMIO)
  - problems with MPI_TYPE_CREATE_DARRAY
    - empty blocks allowed but erroneous implementation
  - without parallel, striped file I/O --> only ~30 MB/s RAID
  - Work around for empty blocks and striped file I/O:
    - see www.hlrs.de/mpi/mpi_t3e.html#StripedIO
    - up to 200 MByte/sec parallel striped file I/O but only for applications using parallel collective I/O on large files (> 3 Mbyte / process)
    - with benchmark example exa_block.f www.hlrs.de/mpi/exa_block.f

91

Implementation-Restrictions, III.

- MPI-2 on the Fujitsu VPP
  - needs extra PE for a server process
    - to handle shared file pointers
    - to run on MPIFS caching file system
  - does not support Unix file system — implementation is underway
  - collective file access — optimization is underway

92

MPI-I/O: Summary

- Rich functionality provided to support various data representation and access
- MPI I/O routines provide flexibility as well as portability
- Collective I/O routines can improve I/O performance
- Initial implementations of MPI I/O available (eg. ROMIO from Argonne)

93

Other MPI-2 Functionality

Process Startup

- Current MPI implementations provide the "mpirun" as a startup command which is not standard and not portable
- MPI-2 specifies an "mpiexec" startup command (recommended)

```
mpiexec {-n <maxprocs> -soft <      >
        -host <      > -arch <      >
        -wdir <      > -path <      >
        -file <      > ……. Command
        } : {……} : {……}
```

95

Null Values

- MPI-2 allows user to pass NULL for argc and argv arguments in MPI_Init
- MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE can be passed as the status argument to inform implementation not to fill in the status field

```
MPI-1:
        MPI_Init(&argc, &argv);
        MPI_Wait( ……., &status);
MPI-2:
        MPI_Init(NULL, NULL);
        MPI_Wait(……., MPI_STATUS_IGNORE);
```

96

C-Macros

- Macro Implementation in C, is now allowed for
  - (P)MPI_WTIME [*this is the only change to MPI 1.2]*
  - (P)MPI_WTICK [*ditto.*]
  - new handle-conversion functions, e.g. MPI_Group_f2c
  - no others
- Rationale
  - performance on some systems
  - lower overhead from timing
  - no less predictability

97

Other Functionality, I.

- Datatype constructor
  - MPI_TYPE_CREATE_INDEXED_BLOCK
- Check if MPI had finalized
  - MPI_FINALIZED
- C/C++/Fortran Language interoperability support
  - between languages in same processes
  - messages transferred from one language to another

98

Other Functionality, II.

- New datatype manipulation functions
  - type constructors with explicit addresses
  - extents, true-extents, bounds of datatypes
  - subarray datatype constructor
  - distributed array datatype constructor
- New predefined datatypes
  - to support wide characters - MPI_WCHAR
  - to support reduction on characters - MPI_SIGNED_CHAR
  - since unsigned long long is part of ISO C9X - MPI_UNSIGNED_LONG_LONG

99

Other Functionality, III.

- Canonical Pack/Unpack
  - MPI_PACK_EXTERNAL
  - MPI_UNPACK_EXTERNAL
  - MPI_PACK_EXTERNAL_SIZE
- New values VERSION=2, SUBVERSION=0
- New error class for Invalid Keyval: MPI_ERR_KEYVAL
- User function at process termination via attributes and delete callback function on MPI_COMM_SELF

100

Deprecated Names/Functions

Deprecated	MPI-2 Replacement
MPI_ADDRESS	MPI_GET_ADDRESS
MPI_TYPE_HINDEXED	MPI_TYPE_CREATE_HINDEXED
MPI_TYPE_HVECTOR	MPI_TYPE_CREATE_HVECTOR
MPI_TYPE_STRUCT	MPI_TYPE_CREATE_STRUCT
MPI_TYPE_EXTENT	MPI_TYPE_GET_EXTENT
MPI_TYPE_UB	MPI_TYPE_GET_EXTENT
MPI_TYPE_LB	MPI_TYPE_GET_EXTENT
MPI_UB	MPI_TYPE_CREATE_RESIZED
MPI_LB	MPI_TYPE_CREATE_RESIZED
.....	.....
.....	.....

Motivation: Incorrect Fortran interface for Aint

101

External Interfaces

- Generalized requests to allow user to define new non-blocking operations
- Associating information with status object
- Datatype decoding functions
- Interaction with threads
- New attribute caching and naming functions for communicators, windows, and types
- New error handling routines for communicators, windows, and files
- mainly for library writers

102

C++ Language Bindings

- C++ bindings match the new C bindings
- MPI objects are C++ objects
- MPI functions are methods of C++ classes
- User must use MPI create and free functions instead of default constructors and destructors
- Uses shallow copy semantics
  (except MPI::Status objects)
- C++ exceptions used instead of returning error code
- declared within an MPI namespace (`MPI::...`)
- C++/C mixed-language interoperability

Fortran 90 Support, I.

- MPI-2 Fortran bindings are Fortran 90 bindings that are "Fortran 77 friendly" (most cases)
- Fortran 90 and MPI have several incompatibilities
  - strong typing vs. choice arguments
  - data copying vs. sequence association
  - special values vs. special constants
  - .....
- MPI-2 standard documents the "do's" and "don'ts" while using Fortran 90 features
  - *Chap. 10.2.2* "Problems with Fortran Bindings for MPI"

Fortran 90 Support, II.

- Different support levels:
  - Basic Fortran support
    - mpif.h is valid with free-form and fixed-form
  - Extended Fortran support
    - an **`mpi`** module
    - datatype generation routines for KIND-parameterized Fortran types:
      - `MPI_TYPE_CREATE_F90_INTEGER`
      - `MPI_TYPE_CREATE_F90_REAL`
      - `MPI_TYPE_CREATE_F90_COMPLEX`
    - alternative concept
      [*not appropriate for heterogeneous platforms*]:
      - `MPI_SIZEOF, MPI_TYPE_MATCH_SIZE`

Comments

I/O	*most important chapter* *if we get high-performance implementations*
Fortran	*please read the section 10.2.2* "Problems with Fortran Bindings for MPI" *also if you only use Fortran 77 subset*
1-sided	*never better than point-to-point ???*
spawn	*does it help any PVM user to come to MPI?*
misc	*useful* ***`MPI_TYPE_SIMPLE_STRUCT`*** *went to JOD*
collective	*nonblocking "split" interface only for I/O-routines!*
Thread	*for coexistence of OpenMP & MPI enough???*
Interoperability	*between different MPI implementations is not part of the standard, but may come with IMPI*

Summary

- MPI-2 standard document available since July 1997
- Provides extensions to MPI-1,
  does not replace MPI-1
- Provides a wide variety of functionality,
  some still untested
- Implementation of parts of the MPI-2 standard are available
- Full implementations may be available by mid-2000

References/Resources

- The MPI Forum is documented officially at
  www.mpi-forum.org
- Gropp et al have a book on MPI-2, based on the standard, from MIT Press, entitled
  "MPI - The Complete Reference, V. 2"
  MIT Press, 1998
- News group: comp.parallel.mpi (!?!?!)
- ROMIO subset of MPI-2 I/O: *www.mcs.anl.gov/romio*
- Other web pages: *www.mcs.anl.gov/mpi/mpich/*
  www.mpi.nd.edu/lam/
- Authors' MPI pages: *www.erc.msstate.edu/mpi/*
  www.hlrs.de/mpi/

Index

Introduction	pp	2 - 7	(6 pages)	10’
MPI 1.2	pp	8 - 10	(3 pages)	5’
Process Management	pp	11 - 24	(14 pages)	15’
				[10’]
One-sided Communication	pp	25 - 42	(18 pages)	15’
Extended Collective Comm.	pp	43 - 51	(9 pages)	5’
				[10’]
MPI File Input/Output	pp	52 - 93	(42 pages)	45’
				[10’]
Other MPI-2 Functionality	pp	94 - 105	(12 pages)	10’
Summary and References	pp	106 - 109	(4 pages)	15’
& Discussion				2 h