Alfa-SCAT
Scientific Computing Advanced Training

MPI Message Passing - Basics

Dr Mike Ashworth

Computational Science & Engineering Dept

CCLRC Daresbury Laboratory &

HPCx Terascaling Team

UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FISICAS Y MATEMATICAS
Departamento de Física

ESPCI

a project funded by EuropeAid

Outline

- Introduction
- Getting Started
- Point-to-Point Operations
- Collective Operations

Disclaimer

- MPI is usually the subject of a course running over (at least) several days
- This will be a very broad overview of the subject in two lectures with practical sessions

I will go quickly …

… there will be material missing …

… but hopefully it will give you a useful introduction

- You should follow up with in-depth courses or self-study

a project funded by EuropeAid

Message Passing Interface - Introduction

a project funded by EuropeAid

Before MPI

- In the early days (1980s) of MIMD distributed memory systems there were many message passing systems
- Some vendor-specific
  - Intel iPSC systems: isend, irecv
- Some portable
- PVM (Parallel Virtual Machine)
  - designed for networks of workstations
  - many advanced features
  - heterogenous – translation between different datatypes
  - process spawning and deletion
  - fast 'bufferless' send & receive for MPP systems
- PARMACS
  - Macro/library abstraction maps standard calls onto underlying message-passing system

What is MPI?

- The MPI is a *de facto* standard
  - cf. Fortran and C which are ISO & ANSI standards
- It was developed by a Consortium of users, software developers and hardware vendors
  - Message Passing Interface Forum
  - 40 organisations
- Version 1.0 - 5th May 1994
- Version 1.1 & 1.2 clarifications & corrections
- MPI-1 standard contains 128 subroutines
  - Bindings for Fortran 77 and C
  - Defines communicators – subsets of processes
  - Point-to-point message passing – send & receive
  - Variants for non-blocking sends & receives
  - Global operations
  - Derived datatypes

MPI-2

- MPI-2 was defined 18th July 1997
- MPI-2 includes MPI-1.2 and provides extensions for
  - Process creation and management
  - Single-sided communications
  - Extended collective operations
  - Parallel input/output
  - Extended language bindings – C++ & Fortran 90
- MPI-1 is still well-suited and sufficient for most applications

MPI implementations

- MPI defines how MPI should function and gives advice to implementers and to users
- The standard document is actually very readable
- Most parallel computer vendors have designed their own implementations, some of which take advantage of their special hardware features
- There is a reference or portable implementation called MPICH freely available from

http://www-unix.mcs.anl.gov/mpi/mpich/

MPI Resources

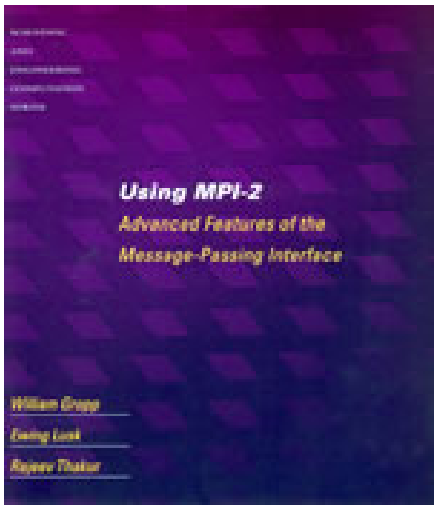
- MPI Home

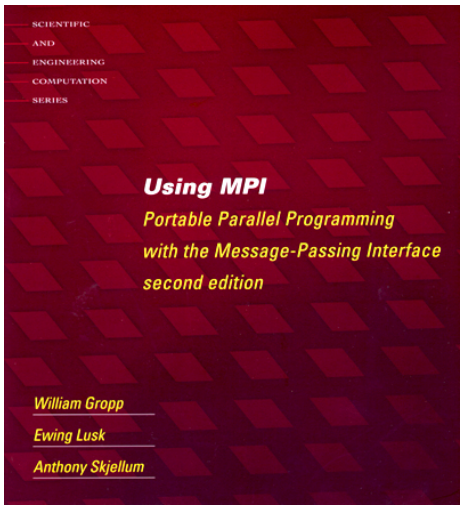
  http://www.mpi-forum.org/

  http://www-unix.mcs.anl.gov/mpi/

  - Contains MPI documentation (english)
- Google "MPI exercises" leads to several sites
- "The Book"

Using MPI-2 - Advanced Features of the Message Passing Interface, William Gropp, Ewing Lusk and Rajeev Thakur





Example programs available on the Argonne website

Message Passing Interface – Getting Started

a project funded by EuropeAid

Communicators

- The MPI communicator is a powerful concept
- It is a handle referring to a set of tasks
- All communication operations take place in the context of a communicator
- Every MPI program starts with a pre-defined communicator MPI_Comm_World (via mpi.h/mpif.h) which refers to all the tasks started by the OS
  - Defines number of tasks, n, and each task has a rank, 0 to n-1
  - Can derive sub-communicators from MPI_Comm_World which have a different (smaller) number of tasks with different ranks
  - Can not extend MPI_Comm_World

MPI Init and Finalize

- Every legal MPI program must start with MPI_Init and end with MPI_Finalize …

```
call MPI_Init (ierr)
…
…
calls to other MPI subroutines
…
…
call MPI_Finalize(ierr)
```

calls to MPI routines are not allowed outside MPI_Init … MPI_Finalize (apart from MPI_Initialised inquiry)

MPI finding Size & Rank

- Usually the first things you do is find out the Size of MPI_Comm_World and your own Rank within it:

```
call MPI_Comm_Size (MPI_Comm_World, size, ierr)

call MPI_Comm_Rank (MPI_Comm_World, rank, ierr)
```

0 <= rank < size

- These can also be used for user-defined communicators
- The size of MPI_Comm_World is determined by how you launched the job e.g.

mpirun –np 32 a.out

MPI Hello World!

```
program hello
include ‘mpif.h’
integer ierr, rank, size
call MPI_Init (ierr)
call MPI_Comm_Size (MPI_Comm_World, size, ierr)
call MPI_Comm_Rank (MPI_Comm_World, rank, ierr)
write (*,*) ‘Hello World - I am process ‘,rank,’ of ‘,size
call MPI_Finalize (ierr)
stop
end
```


Basic send & receive

- Define data as a triplet:
  - Start address, buf
  - Number of elements, count
  - Datatype
- Define message source/destination as triplet:
  - rank (source or destination)
  - tag (arbitrary integer to label messages)
  - communicator (MPI_Comm_World or user-defined)

call MPI_Send (buf, count, datatype, dest, tag, comm, ierr)

call MPI_Recv (buf, count, datatype, source, tag, comm, status, ierr)

Six subroutines

- You can arguably construct any parallel program out of these six subroutines:
  - MPI_Init & MPI_Finalize
  - MPI_Comm_Size & MPI_Comm_Rank
  - MPI_Send & MPI_Recv
- If that were all there is to MPI
  - this course would be very short!
  - your program would be long-winded and inefficient
- But before we move on to look at MPI in more detail …

Let's try some exercises

UTFSM cluster

- You can find exercise 1 in
  /home/local/scat/parallel-course/ex1/
- Copy files into your own
  ~/parallel-course/ex1/
- There is a README
- makefile is provided using mpif77
- If you wish to use C change to mpicc
- Job script hello.job is provided
  run using ‘qsub hello.job’

Exercise 1

- Write a program hello.f that uses MPI and has each MPI process print

  Hello World - I am process i of n

- using the rank in MPI_Comm_World for i
- and the size of MPI_Comm_World for n

- You may want to use these MPI routines in your solution:

MPI_Comm_Rank, MPI_Comm_Size, MPI_Finalize, MPI_Init

- What order does the output appear in?
- Is it always the same?

Exercise 2

- How can “Hello World” be adapted to print messages in order?
- Add MPI_Send and MPI_Recv to the hello.f program so that the worker processes send their message to the master (rank 0) and the master does all the prints

Message Passing Interface – Point-to-Point

a project funded by EuropeAid

Defining data

- The send & receive buffers are defined by *count* items of type *datatype* starting at address *buf*
- Counting elements (not bytes) is machine independent
- Basic types are provided:

MPI datatype	Fortran datatype
MPI_Real	real
MPI_Integer	integer
MPI_Double_Precision	double precision
MPI_Complex	complex
MPI_Logical	logical
MPI_Character	character(1)
MPI_Byte	
MPI_Packed	

a similar (longer) list exists for C

others may be supported e.g. MPI_Real8 to match with real*8

a project funded by EuropeAid

Derived Datatypes

- In addition to the basic datatypes, it is possible to build derived datatypes
- They allow you
  - to send mixed data e.g. integers and reals as a single message
  - to send non-contiguous data
- MPI provides a large number of datatype constructors

to generate a datatype for equally spaced blocks

```
MPI_Type_Vector(count, blocklength, stride, oldtype, newtype)
```

e.g. for the row of a real array of dimension(n,m)

```
MPI_Type_Vector(m, 1, n, MPI_Real, rowtype)
```

then rowtype can be used in send/receive calls

```
MPI_Send(a(1,1),1,rowtype,...
```


Wildcards

- MPI messages are non-overtaking
  - if one process send two messages to another, then they will be received in the order they were sent
- MPI_ANY_SOURCE
  - A receive may use MPI_ANY_SOURCE as the source rank
  - This matches with a message from any rank
- MPI_ANY_TAG
  - A receive may use MPI_ANY_TAG as the message tag
  - This matches with a message with any tag
- Use only when necessary and beneficial
- It is much safer to specify the source and tag when you know them

Deadlock

- Process 0

call MPI_Send (…, 1, …)

call MPI_Recv (…, 1, …)

- Process 1

call MPI_Send (…, 0, …)

call MPI_Recv (…, 0, …)

- What happens?

Unless MPI_Send/MPI_Recv is buffered

(and this depends on the MPI implementation)

DEADLOCK

Avoiding deadlock

- We can sometimes re-organise the communications to avoid deadlock
  - First even processes send odd processes receive
  - Then odd processes send even processes receive
- But this serialises the communications into two stages which is inefficient
- Better to use MPI_SendRecv, which combines send and receive in a single deadlock-free call
- Another good solution is to use non-blocking communications …

Communications modes

- The standard MPI_Send and MPI_Receive are blocking
  - MPI_Send does not return until it is safe to re-use the send buffer – this may need to wait until the receive is complete!
  - MPI_Receive does not return until the data is ready in the receive buffer
- MPI provides other modes of send:
  - buffered; user-provided buffer space allows send to complete irrespective of whether a receive has been posted
  - synchronous; completion indicates that the receive has started
  - ready; send completes because the receive must be posted

  MPI_BSend, MPI_SSend, MPI_RSend

- These may be used to ensure deadlock free operation, but non-blocking communication is much better

Non-blocking communications I

- A non-blocking send/receive initiates the operation, but does not complete it
- With suitable hardware the data transfer can proceed in parallel with local computation
- Users should be aware that it is not safe to re-use send/receive buffers before completion
- Call provides a request handle
- Operation is completed with a MPI_Wait

MPI_ISend, MPI_IBSend, MPI_IRSend, MPI_ISSend

MPI_IRecv

Non-blocking communications II

- Example

```
MPI_ISend (a,1,sendhalodt,north,tag,MPI_Comm_World,sendrq,ierr)
MPI_IRecv (a,1,recvhalodt,south,tag,MPI_Comm_World,recvrq,ierr)
...
...
intervening computation
...
...
MPI_Wait (sendrq,sendstatus,ierr)
MPI_Wait (recvrq,recvstatus,ierr)
```

Only now can you be sure that the data in a has been sent/received

- This is very highly recommended

Message Passing Interface – Collectives

Introduction

- Collective communication involves a group of processes
- MPI provides the following collectives
  - Barrier
  - Broadcast
  - Gather
  - Scatter
  - AllGather (all processes receive the result)
  - AllToAll (complete exchange)
  - Reduce (sum, max, min, or user-defined)
  - AllReduce (as Reduce but all processes receive result)
  - Scan

Barrier

- Simplest example of a collective operation is MPI_Barrier

```
MPI_Barrier (MPI_Comm_World,ierr)
```

- Operates as a barrier – processes wait here until all of them have arrived
- Barriers are BAD
- Collective operations must be executed by all processes in the communicator
- Collective operations are blocking

Collective examples

- MPI_Broadcast

  MPI_Broadcast (buf,count,datatype,root,MPI_Comm_World,ierr)

- MPI_Gather

  MPI_Gather (sendbuf,sendcount,sendtype,
  recvbuf,recvcount,recvtype,root,MPI_Comm_World,ierr)

- MPI_Reduce

  MPI_Reduce (sendbuf,recvbuf,count,datatype,
  op,MPI_Comm_World,ierr)

  where op is one of MPI_Sum, MPI_Max, MPI_Min, etc

Exercise 3

- You can find exercise 3 in

  /usr/local/scat/parallel-course/ex3/

- This is a very primitive Jacobi iteration to solve the Laplace equation in two dimensions with finite differences
- Serial code jacobi_serial.f
- Parallel version jacobi.f
  - But the communications is missing!

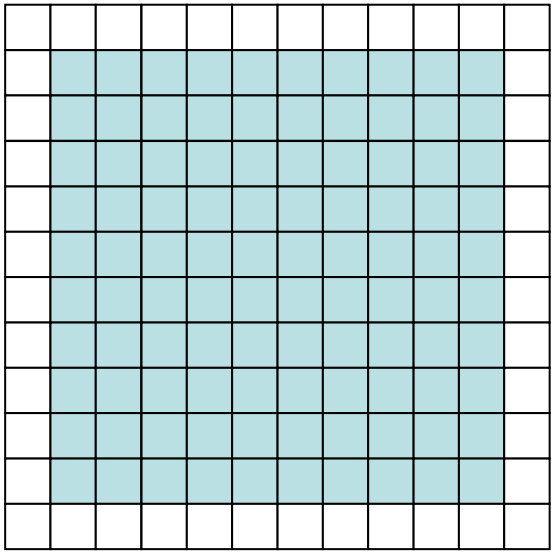
Checklist for parallelisation

- Partition the problem domain
  - set up new indexes for the bounds of the sub-domain
- Assign sub-domains to processors
- Change loop bounds to run over the local sub-domain
  - ensure the *owner computes* rule is obeyed
- Determine the communications dependencies
  - examine the relationship between LHS assigns and RHS references to non-local data
- Insert communications calls
- Test correctness
  - Results should be bit-wise identical with number of processors
- Test performance
  - Scaling of performance with number of processors

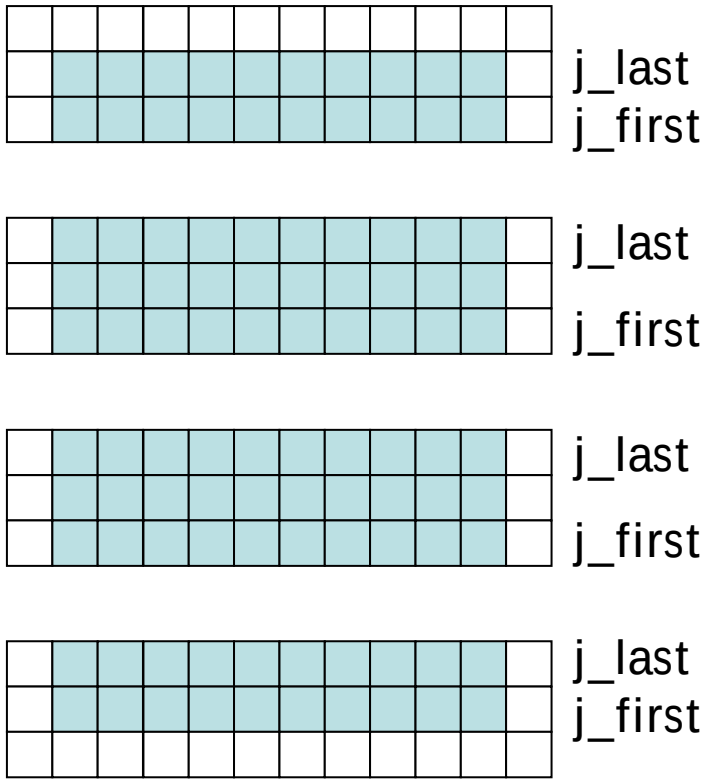
Partitioning

12x12 problem size

outer rows/column are boundary data

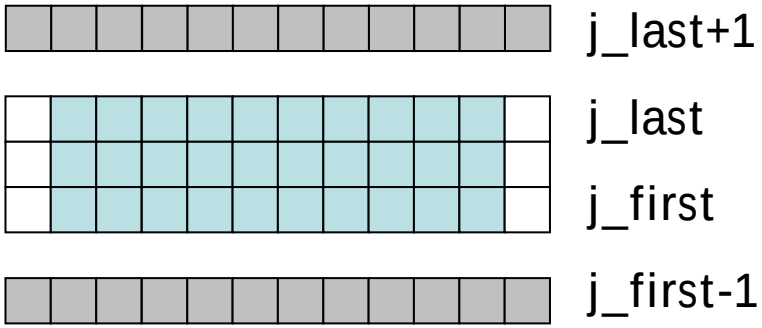


1D partitioning in j for 4 processors



a project funded by EuropeAid

Halo exchange



Send from j_last

Receive into j_last+1

Send from j_first

Receive into j_first-1

That's enough help
Off you go!

- Do the serial and parallel program produce the same answers?
- Does the parallel program run faster?
- How can you improve its performance?