Parallel Computing

MPI

Christoph Beetz

Theoretical Physics I, Ruhr-Universität Bochum

September 7, 2010
Overview

- Introduction to Parallel Computing
- MPI
- How to parallelize your code
Why Parallel Computing

Calculation

- is too big to fit in memory of one machine
  - domain decomposition on several machines
- takes too much time with only one CPU
  - speed up with lots of CPU’s
Parallel Computer Architectures

SMP: Symmetric Multiprocessor

- Uses shared system resources

Diagram:

- Processor 1
- Processor 2
- Processor 3
- Processor 4

- BUS
- Memory
- IO
MPP: Massively Parallel Processors

consists of nodes connected by a high-speed network

Parallel Computer Architectures

Fast Interconnection Network

Processor 1
Memory
IO
BUS

Processor 2
Memory
IO
BUS

Processor 3
Memory
IO
BUS

Processor 4
Memory
IO
BUS
Mixed SMP and MPP

consists of nodes with several cores

Diagram of Mixed SMP and MPP showing nodes with processors and a fast interconnection network.
Parallel Computer Architectures

<table>
<thead>
<tr>
<th>Name</th>
<th>CPUs</th>
<th>CPU/Node</th>
<th>Mem/Node</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paddy</td>
<td>2</td>
<td>2</td>
<td>4 GB</td>
<td>portable</td>
</tr>
<tr>
<td>Euler</td>
<td>64</td>
<td>4</td>
<td>8 GB</td>
<td></td>
</tr>
<tr>
<td>DaVinci</td>
<td>128</td>
<td>8</td>
<td></td>
<td>4 x 516 GPUs/Node</td>
</tr>
<tr>
<td>BlueGene / P</td>
<td>294912</td>
<td>4</td>
<td>2 GB</td>
<td></td>
</tr>
<tr>
<td>RoadRunner</td>
<td>122400</td>
<td>4+4+32</td>
<td>2x16 GB</td>
<td>Opteron + Cell</td>
</tr>
<tr>
<td>Jaguar</td>
<td>224256</td>
<td>2x6</td>
<td>16 GB</td>
<td></td>
</tr>
</tbody>
</table>

Example: Heat transfer in 2 Dimensions

MPI

What is MPI
Basic MPI Instructions
Words of Wisdom
How to parallelize your program
Domain decomposition
Software

SMP
Multithreaded programs

Single thread

Parallel Computing
Introduction
Example: Heat transfer in 2 Dimensions
MPI
What is MPI
Basic MPI Instructions
Words of Wisdom
How to parallelize your program
Domain decomposition
Parallel Computing

Introduction

Example: Heat transfer in 2 Dimensions

MPI

What is MPI
Basic MPI Instructions
Words of Wisdom

How to parallelize your program
Domain decomposition

Software

SMP

Multithreaded programs

- Posix-Threads (http://en.wikipedia.org/wiki/POSIX_Threads)
- OpenMP (www.openmp.org)
MPP

Local serial programs with communication

Parallel Computing

Introduction

Parallel Computing
Example: Heat transfer in 2 Dimensions

MPI
What is MPI
Basic MPI Instructions
Words of Wisdom

How to parallelize your program
Domain decomposition
Software

MPP

Local serial programs with communication

- High Performance Fortran (http://hpff.rice.edu/)
- MPI (main focus of this lecture)
  - MPICH http://www-unix.mcs.anl.gov/mpi/mpich
  - Open MPI http://www.open-mpi.org
Problem

Solve heat transfer equation in 2 dimensions on a cartesian grid with periodic boundary conditions:

$$\partial_t u = a \cdot \Delta u$$ (1)
Parallel Computing

Introduction
Parallel Computing
Example: Heat transfer in 2 Dimensions

MPI
What is MPI
Basic MPI Instructions
Words of Wisdom
How to parallelize your program
Domain decomposition

Heat transfer

Serial program

program heat
implicit none
integer, parameter :: mx=32, my=32
real(kind=8) :: u(0:mx+1,0:my+1)
real(kind=8) :: unp1(0:mx+1,0:my+1)
integer :: i,j,ic
real(kind=8) :: x,y,xb,xe,yb,ye,dx,dy
real(kind=8) :: ax,ay,pi,time,dt
ax = 1
ay = 1
pi = 4.*atan(1.d0)
dt = 0.00025 ! critical value
dt = 0.0002
xb = 0.d0
xe = 1
yb = 0.d0
ye = 1
dx = (xe-xb)/mx
dy = (ye-yb)/my
write(*,'(a,e12.6)') 'cfl = ', dt/min(dx,dy)**2
u=0.
do j = -1,my+1
  y = yb + j*dy
  do i = -1,mx+1
    x = xb + i*dx
    if ((x-0.5)**2+(y-0.5)**2 .lt. 0.2) then
      u(i,j)= 1
    end if
  end do
end do
time = 0.
ic = 0
call VTKout(u,mx,my)
c --> simple Euler step
do while (time .le. 1000*dt)
c --> heat equation
  unp1(1:mx,1:my) = u(1:mx,1:my)
  & +dt*((u(2:mx+1,1:my)-2.*u(1:mx,1:my)
  & +u(0:mx-1,1:my))/dx**2
  & +(u(1:mx,2:my+1)-2.*u(1:mx,1:my)
  & +u(1:mx,0:my-1))/dy**2)
call boundary(unp1,mx,my)
u = unp1
time = time+dt
ic = ic+1
write(*,*) 'time = ',time
if (ic > 10) then
  ic = 0
  call VTKout(u,mx,my)end if
end do
return
end

subroutine boundary(u,mx,my)
implicit none
integer :: mx,my
real(kind=8) :: u(0:mx+1,0:my+1)
integer :: i,j
do i=1,mx
  u(i,0 ) = u(i,my)
  u(i,my+1) = u(i,1 )
end do
do j=0,my+1
  u(0,j ) = u(mx,j)
  u(mx+1,j) = u(1,j)
end do
return
end
real(kind=8) :: u(0:mx+1,0:my+1)
1. Step: Computation

\[
\begin{align*}
\text{unp1}(1:\text{mx}, 1:\text{my}) &= \text{u}(1:\text{mx}, 1:\text{my}) \\
&+ \text{dt} \cdot ((\text{u}(2:\text{mx}+1, 1:\text{my}) - 2 \cdot \text{u}(1:\text{mx}, 1:\text{my}) + \text{u}(0:\text{mx}-1, 1:\text{my})) / \text{dx}^2 \\
&+ (\text{u}(1:\text{mx}, 2:\text{my}+1) - 2 \cdot \text{u}(1:\text{mx}, 1:\text{my}) + \text{u}(1:\text{mx}, 0:\text{my}-1)) / \text{dy}^2)
\end{align*}
\]
call boundary(unp1, mx, my)
The Grid

```
serial ~ Parallel
```

```
mpirun -np 4 heatmpi

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, & nProcs, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, & myRank, ierr)
```

```
\begin{array}{|c|c|c|}
\hline
\text{Process 1} & \multicolumn{2}{c|}{\text{Process 3}} \\
\hline
my+1 & \multicolumn{2}{c|}{my} \\
\hline
my & \multicolumn{2}{c|}{my} \\
\hline
0 & 1 & \multicolumn{1}{c|}{mx} \multicolumn{1}{c|}{mx+1} \\
\hline
0 & \multicolumn{2}{c|}{Process 2} \\
\hline
\end{array}
```
Parallel Computing

The Grid

```
! GLOBAL FIELD DIMENSIONS
integer, parameter :: nx=1024, ny=1024
!
!
! LOCAL FIELD DIMENSIONS
integer :: mx,my
!
!
```

nProcsXY(1)=int(sqrt(dble(nProcs)))
nProcsXY(2)=nProcs/nProcsXY(2)
if mod(nx,nProcsXY(1).ne.0) then stop

```
mx=nx/nProcsXY(1)
my=ny/nProcsXY(2)
```

.fields must be allocatable now. Size (1:mx,1:my) depends on number of processes.
1. Step: Computation

For each process exactly the same computation as in serial case.

\[
\text{unp1}(1:mx,1:my) = u(1:mx,1:my) \\
&+dt*((u(2:mx+1,1:my)-2.*u(1:mx,1:my)+u(0:mx-1,1:my))/dx**2 \\
&+(u(1:mx,2:my+1)-2.*u(1:mx,1:my)+u(1:mx,0:my-1))/dy**2)
\]
- MPI absolutely necessary!
- Each process has to send to his upper and lower neighbour in each direction.
- In one direction the data to be send is not contiguous in memory.
What is MPI

MPI is

- a tool to consolidate what parallelization has separated
- a library for communication between processes
- available for Fortran / C / C++
- portable
Basic MPI Instructions

- Over 150 MPI subroutines
- Only need a dozen to parallelize program
- Classification
  - Environment Management
  - Collective Data Transfer
  - Point to Point Data Transfer
  - ...

Parallel Computing

Introduction
- Parallel Computing
- Example: Heat transfer in 2 Dimensions

MPI
- What is MPI
- Basic MPI Instructions
- Words of Wisdom

How to parallelize your program
- Domain decomposition
A minimal program in C++

```c++
#include <mpi.h>
void main(int* argc, char*** argv)
{
    MPI_Init(&argc, &argv);
    int nProcs, myRank;
    MPI_Comm_size(MPI_COMM_WORLD, &nProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    if (myRank==0) {
        std::cout << nProcs << ' ' ' processes started.' << std::endl;
    }
    std::cout << ' ' 'Hello from process ' ' ' << myRank << std::endl;
    MPI_Finalize();
}
```
A minimal program in C++

```cpp
#include <mpi.h>
void main(int* argc, char*** argv)
{
    MPI_Init(&argc, &argv);
    int nProcs, myRank;
    MPI_Comm_size(MPI_COMM_WORLD, &nProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    if (myRank==0) {
        std::cout << nProc << " processes started." << std::endl;
    }
    std::cout << "Hello from process " << myRank << std::endl;
    MPI_Finalize();
}
```

mpi.h provides parameter like MPI_COMM_WORLD and variable types like MPI_INTEGER
A minimal program in C++

```cpp
#include <mpi.h>
void main(int* argc, char*** argv)
{
    MPI_Init(&argc, &argv);
    int nProcs, myRank;
    MPI_Comm_size(MPI_COMM_WORLD, &nProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    if (myRank==0) {
        std::cout << nProcs << " processes started."
    }
    std::cout << "Hello from process " << myRank << std::endl;
    MPI_Finalize();
}
```

MPI_Init has to be called once and only once, before calling any other MPI-subroutine to initialize the MPI environment.
A minimal program in C++

```c++
#include <mpi.h>

void main(int* argc, char*** argv)
{
    MPI_Init(&argc, &argv);
    int nProcs, myRank;
    MPI_Comm_size(MPI_COMM_WORLD, &nProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    if (myRank==0) {
        std::cout << nProcs << ' ' << ' processes started.' << std::endl;
    }
    std::cout << ' ' << 'Hello from process ' << myRank << std::endl;
    MPI_Finalize();
}
```

In C++ all parameters have to be passed as pointers.
A minimal program in C++

```cpp
#include <mpi.h>
void main(int* argc, char*** argv)
{
    MPI_Init(&argc, &argv);
    int nProcs,myRank;
    MPI_Comm_size(MPI_COMM_WORLD, &nProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    if (myRank==0) {
        std::cout << nProc << ‘‘ processes started.’’ << std::endl;
    }
    std::cout << ‘‘Hello from process ‘‘ << myRank << std::endl;
    MPI_Finalize();
}
```

With Communicators you can define groups of processes. MPI_COMM_WORLD is the Communicator for all processes.
Parallel Computing

Introduction
Example: Heat transfer in 2 Dimensions

MPI
What is MPI
Basic MPI Instructions
Words of Wisdom
How to parallelize your program
Domain decomposition

A minimal program in C++

```cpp
#include <mpi.h>
void main(int* argc, char*** argv)
{
    MPI_Init(&argc, &argv);
    int nProcs, myRank;
    MPI_Comm_size(MPI_COMM_WORLD, &nProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    if (myRank==0) {
        std::cout << nProc << ' ' processes started.' << std::endl;
    }
    std::cout << 'Hello from process ' << myRank << std::endl;
    MPI_Finalize();
}
```

MPI_Comm_size returns number of processes in variable nProcs.
Parallel Computing

Introduction

Example: Heat transfer in 2 Dimensions

MPI

What is MPI
Basic MPI Instructions
Words of Wisdom
How to parallelize your program
Domain decomposition

Environment Management

A minimal program in C++

```c++
#include <mpi.h>
void main(int* argc, char*** argv)
{
    MPI_Init(&argc, &argv);
    int nProcs,myRank;
    MPI_Comm_size(MPI_COMM_WORLD, &nProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    if (myRank==0) {
        std::cout << nProc << ‘‘ processes started.’’ << std::endl;
    }
    std::cout << ‘‘Hello from process ‘‘ << myRank << std::endl;
    MPI_Finalize();
}
```

MPI_Comm_rank returns the identification number of the process in variable myRank.
A minimal program in C++

```cpp
#include <mpi.h>
void main(int* argc, char*** argv)
{
    MPI_Init(&argc, &argv);
    int nProcs,myRank;
    MPI_Comm_size(MPI_COMM_WORLD, &nProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    if (myRank==0) {
        std::cout << nProc << ' ' ' processes started.' " " std::endl;
    }
    std::cout << ' ' 'Hello from process ' " " myRank << std::endl;
    MPI_Finalize();
}
```

All processes run the same code. Different behaviour can be achieved by using the rank of the processes.
A minimal program in C++

```c++
#include <mpi.h>
void main(int* argc, char*** argv)
{
    MPI_Init(&argc, &argv);
    int nProcs, myRank;
    MPI_Comm_size(MPI_COMM_WORLD, &nProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    if (myRank==0) {
        std::cout << nProc << ' ' ' processes started.' ' ' ' << std::endl;
    }
    std::cout << ' ' 'Hello from process ' ' ' ' << myRank << std::endl;
    MPI_Finalize();
}
```

MPI_Finalize terminates the MPI environment.
#include <mpi.h>
void main(int* argc, char*** argv)
{
    MPI_Init(&argc, &argv);
    int nProcs,myRank;
    MPI_Comm_size(MPI_COMM_WORLD, &nProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    if (myRank==0) {
        std::cout << nProcs << ' ' 'processes started.' << std::endl;
    }
    std::cout << 'Hello from process ' << myRank << std::endl;
    MPI_Finalize();
}

Sample Output:
2 processes started
Hello from process 1
Hello from process 0
Environment Management

A minimal program in Fortran

```fortran
PROGRAM hello
IMPLICIT NONE
include 'mpif.h'
INTEGER ierror, nProcs, myRank
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nProcs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myRank, ierr)
IF (myRank==0) PRINT *, nProcs, ' processes started'
PRINT *, 'Hello from process ', myRank
CALL MPI_FINALIZE(ierr)
END PROGRAM hello
```

use Fortran include file.
Parallel Computing

Introduction

Example: Heat transfer in 2 Dimensions

MPI

What is MPI

Basic MPI Instructions

Words of Wisdom

How to parallelize your program

Domain decomposition

Environment Management

A minimal program in Fortran

```fortran
PROGRAM hello
IMPLICIT NONE
include ‘mpif.h’
INTEGER ierror, nProcs, myRank
CALL MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nProcs, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myRank, ierr)
IF (myRank==0) PRINT *,nProcs,’” processes started’’
PRINT ‘,’”Hello from process ‘’,myRank
CALL MPI_FINALIZE(ierr)
END PROGRAM hello
```

Return code is given in the last argument. Note, that the wrong number of arguments will result in segmentation fault during run-time.
Parallel Computing

Example: Heat transfer in 2 Dimensions

MPI

What is MPI

Basic MPI Instructions

Words of Wisdom

How to parallelize your program

Domain decomposition

Environment Management

**MPI\_BARRIER**

Blocks all processes until the last one calls this routine.

CALL MPI\_BARRIER(communicator, ierror)

Can be used to serialize parallel programs. e.g. to write in a file in order of processes.

DO (i=0,nProcs)
   IF (i==myRank) THEN
      do some IO-Routines
   END IF
   CALL MPI\_BARRIER(MPI\_COMM\_WORLD,ierror)
END DO
Collective Communication

**MPI_BROADCAST**

Broadcasts data from one process to all processes in the communicator.

CALL MPI_BCAST(buffer, count, datatype, root, communicator, ierror)

- **buffer**: First element of send / receive data
- **count**: Number of elements to be send
- **datatype**: MPI datatype of elements in buffer
- **root**: Which process sends the data
- **comm**: Communicator of Processes data will be send to
- **ierror**: Return value
MPI_BROADCAST

Broadcasts data from one process to all processes in the communicator.

IF (myRank==0) buffer=42 ELSE buffer=0
CALL MPI_BCAST(buffer, 1, MPI_INTEGER,
               0, MPI_COMM_WORLD, ierr)

```
INT buffer;
if (MPI_Rank == 0) buffer = 42; else buffer = 0;
MPI_Bcast(buffer, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr);
```
Collective Communication

**MPI_GATHER**

Collects individual messages from each process at the root process.

```
CALL MPI_GATHER(sendbuf, sendcnt, sendtype, 
               recvbuf, recvcnt, recvtype, root, comm, ierror)
```

- **sendbuf**: First element of send data
- **sendcnt**: Number of elements to be send
- **sendtype**: MPI datatype of elements in sendbuf
- **recvbuf**: Buffer, where data is collected. Cannot overlap sendbuf
- **recvcnt**: Number of elements in recvbuf
- **recvtype**: MPI datatype of elements in recvbuf
- **root**: Which process collects the data
- **comm**: Communicator of Processes
- **ierror**: Return value
Collective Communication

**MPI_GATHER**
Collects individual messages from each process at the root process

```fortran
INTEGER :: irecv(3)
INTEGER :: isend

isend=myRank+1
CALL MPI_GATHER(isend, 1, MPI_INTEGER,
                 irecv, 3, MPI_INTEGER, 2, MPI_COMM_WORLD, ierr)
```
**MPI_REDUCE**

Reduces data from all processes to one structure via data reduction operation.

```call
CALL MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, communicator, ierror)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sendbuf</code></td>
<td>First element of send data buffer</td>
</tr>
<tr>
<td><code>recvbuf</code></td>
<td>First element of receive data buffer</td>
</tr>
<tr>
<td><code>count</code></td>
<td>Number of elements to be send</td>
</tr>
<tr>
<td><code>datatype</code></td>
<td>MPI datatype of elements in buffer</td>
</tr>
<tr>
<td><code>op</code></td>
<td>Reduction operation</td>
</tr>
<tr>
<td><code>root</code></td>
<td>Which process receives the data</td>
</tr>
<tr>
<td><code>comm</code></td>
<td>Communicator of Processes data will be reduced from</td>
</tr>
<tr>
<td><code>ierror</code></td>
<td>Return value</td>
</tr>
</tbody>
</table>
Collective Communication

**MPI_REDUCE**

Reduces data from all processes to one structure via data reduction operation

CALL MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, communicator, ierror)

Reduction Operations: Write

- `MPI_SUM` the sum
- `MPI_PROD` the product
- `MPI_MAX` the maximum
- `MPI_MIN` the minimum
- ...
- ...

of `sendbuf` of all processes to `recvbuf` of process `root`.
Collective Communication

**MPI_REDUCE**

Reduces data from all processes to one structure via data reduction operation.

```fortran
CALL MPI_REDUCE(sendbuf, recvbuf, 1, MPI_INTEGER, MPI_SUM, 
                 1, MPI_COMM_WORLD, ierr)
```

<table>
<thead>
<tr>
<th></th>
<th>Proc0</th>
<th>Proc1</th>
<th>Proc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendbuf</td>
<td>12</td>
<td>11</td>
<td>19</td>
</tr>
<tr>
<td>recvbuf</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

```
MPI_REDUCE (.,MPI_SUM,.)
```

<table>
<thead>
<tr>
<th></th>
<th>Proc0</th>
<th>Proc1</th>
<th>Proc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendbuf</td>
<td>12</td>
<td>11</td>
<td>19</td>
</tr>
<tr>
<td>recvbuf</td>
<td>0</td>
<td>42</td>
<td>0</td>
</tr>
</tbody>
</table>
Collective Communication

**MPI_ALLREDUCE / MPI_ALLGATHER**

Same as MPI_REDUCE and MPI_GATHER but result is stored on every process

CALL MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, communicator, ierror)

<table>
<thead>
<tr>
<th>Proc0</th>
<th>Proc1</th>
<th>Proc2</th>
</tr>
</thead>
<tbody>
<tr>
<td>recvbuf</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sendbuf</td>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

| recvbuf | 42 | 42 | 42 |
| sendbuf | 7 | 3 | 2 |

**MPI_ALLREDUCE**

(.,MPI_PROD,..)
Collective Communication

**MPI_ALLTOALL**

Send a distinct message from each process to every process. Process i sends j-th block of sendbuffer to process j. Process j stores data from process i in i-th block of recvbuffer.

CALL MPI_ALLTOALL(sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, comm, ierror)

- sendbuf: First element of data to be send
- sendcount: Number of elements to be send to each process
- sendtype: Data type
- recvbuf: Starting address of data to be received
- recvcount: Number of elements received from any process
- recvtype: Data type
- comm: Communicator of Processes data will be send to
- ierror: Return value
Parallel Computing

**MPI_ALLTOALL**

Send a distinct message from each process to every process. Process i sends j-th block of sendbuffer to process j. Process j stores data from process i in i-th block of recvbuffer.

CALL MPI_ALLTOALL(sendbuf, 3, MPI_INTEGER, recvbuf, 3, MPI_INTEGER, MPI_COMM_WORLD, ierror)
Point to Point Communication

**MPI_SEND, MPI_RECV**

-used to send data to one specific process-

**CALL MPI_SEND(buffer, count, datatype, dest, tag, communicator, ierror)**

- **buffer**: First element of data to be send
- **count**: Number of elements in **buffer**
- **datatype**: MPI datatype of elements in buffer
- **dest**: Rank of destination process
- **tag**: Message tag. Any positive integer number.
- **comm**: Communicator of Processes data will be send to
- **ierror**: Return value
Point to Point Communication

**MPI_SEND, MPI_RECV**

Used to receive data from one specific process

CALL MPI_RECV(buffer, count, datatype, source, tag, communicator, status, ierror)

- **buffer**: Received data will be stored here
- **count**: Number of elements to be received
- **datatype**: MPI datatype of elements in buffer
- **source**: Rank of sending process
- **tag**: Tag of the message or MPI_ANY_TAG
- **comm**: Communicator of Processes data will be send to
- **status**: Integer(MPI_STATUS_SIZE)
- **ierror**: Return value
Parallel Computing

Point to Point Communication

**MPI_SEND, MPI_RECV**

Send an INTEGER value from Proc0 to Proc1

```fortran
INTEGER :: B
INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status

... B=0
IF(myRank==0) THEN
    B=42
    CALL MPI_SEND(B, 1, MPI_INTEGER, 1, 0, MPI_COMM_WORLD, ierror)
ELSE IF (myRank==1) THEN
    CALL MPI_RECV(B, 1, MPI_INTEGER, 0, MPI_ANY_TAG,MPI_COMM_WORLD, status, ierror)
END IF

... 
```

```
<table>
<thead>
<tr>
<th>Proc0</th>
<th>Proc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>42</td>
<td>42</td>
</tr>
</tbody>
</table>
```

```fortran
BEGIN
  ...
  B=0
  IF(myRank==0) THEN
    B=42
    CALL MPI_SEND(B, 1, MPI_INTEGER, 1, 0, MPI_COMM_WORLD, ierror)
  ELSE IF (myRank==1) THEN
    CALL MPI_RECV(B, 1, MPI_INTEGER, 0, MPI_ANY_TAG,MPI_COMM_WORLD, status, ierror)
  END IF
  ...
END
```
MPI_SENDRECV

Send and receive data in a combined operation

CALL MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status, ierror)

Parameter of MPI_SEND
MPI_SENDRECV

Send and receive data in a combined operation

CALL MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status, ierror)

Parameter of MPI_RECV
Parallel Computing

Introduction
Example: Heat transfer in 2 Dimensions
MPI
What is MPI
Basic MPI Instructions
Words of Wisdom
How to parallelize your program
Domain decomposition

Point to Point Communication

**MPI_SENDRECV**

Send and receive data in a combined operation

```
src=myRank-1
dest=myRank+1
if (myRank==0) src=nProcs-1
if (myRank==nProcs-1) dest=0

call MPI_SENDRECV(sendbuf, 1, MPI_INTEGER, dest, 1,
                 recvbuf, 1, MPI_INTEGER, src, MPI_ANY_TAG,
                 MPI_COMM_WORLD, stat, ierror)
```
Point to Point Communication

What happens on Send and Receive

<table>
<thead>
<tr>
<th>Send</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Copy data into user buffer</td>
</tr>
<tr>
<td>▶ Call MPI_SEND</td>
</tr>
<tr>
<td>▶ MPI copies user buffer into system buffer</td>
</tr>
<tr>
<td>▶ System buffer is send to receiving process</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Receive</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Call MPI_RECV</td>
</tr>
<tr>
<td>▶ Receive data in system buffer</td>
</tr>
<tr>
<td>▶ MPI copies data into user buffer</td>
</tr>
<tr>
<td>▶ Use data in user buffer</td>
</tr>
</tbody>
</table>
Point to Point Communication

Blocking/ Nonblocking

Blocking: procedure blocks program until copy to system buffer has completed

- all collective operations
- send, recv

Nonblocking: procedure returns immediately

- isend, irecv
- exceedingly effective with communication co-processor

Blocking: procedure blocks program until copy to system buffer has completed

- all collective operations
- send, recv

Nonblocking: procedure returns immediately

- isend, irecv
- exceedingly effective with communication co-processor
Parallel Computing

MPI_ISEND (buffer, count, datatype, dest, tag, communicator, request, ierror)

MPI_IRECVC (buffer, count, datatype, source, tag, communicator, request, ierror)

MPI_WAIT (request, status, ierror)

MPI_WAITALL (count, request, status, ierror)

- MPI_WAIT blocks the program until the request-operation is finished.
- MPI_ISEND/MPI_IRECVC followed immediately by MPI_WAIT is the same as a blocking operation.
- You can mix blocking with nonblocking operations. e.g.: send with MPI_ISEND and receive with MPI_RECV
- MPI_WAITALL waits for several ISEND/IRECV. request has to be an array then.
Parallel Computing

**Example: Heat transfer in 2 Dimensions**

**MPI**

**What is MPI**

**Basic MPI Instructions**

**Words of Wisdom**

**How to parallelize your program**

**Domain decomposition**

---

**MPI_IRECV / MPI_ISEND / MPI_WAIT / MPI_WAITALL**

```fortran
left=myRank-1
right=myRank+1
if (myRank==0) left=nProcs-1
if (myRank==nProcs-1) right=0

call MPI_IRECV(lRank, 1, MPI_INTEGER, left, MPI_ANY_TAG, MPI_COMM_WORLD, req(1), ierror)
call MPI_IRECV(rRank, 1, MPI_INTEGER, right, MPI_ANY_TAG, MPI_COMM_WORLD, req(2), ierror)
call MPI_ISEND(myRank, 1, MPI_INTEGER, right, myRank, MPI_COMM_WORLD, req(3), ierror)
call MPI_ISEND(myRank, 1, MPI_INTEGER, left, myRank, MPI_COMM_WORLD, req(4), ierror)
call MPI_WAITALL(4, req, stat, ierror)
```

---

MyRank= 0 left Rank= 3 right Rank= 1
MyRank= 1 left Rank= 0 right Rank= 2
MyRank= 2 left Rank= 1 right Rank= 3
MyRank= 3 left Rank= 2 right Rank= 0
Bidirectional Communication

Three cases of bidirectional communication

- Both processes call send routine first, then receive
- Both processes call receive routine first, then send
- One process calls receive first then send and the other calls in opposite order
Point to Point Communication

First Send then Receive

- **Blocking**
  - Program is returning from send when the copy is finished.
  - Can cause deadlocks, if system buffer is smaller than sendbuffer

- **Nonblocking**
  - Both processes: ISEND(.., req, ...), RECV, WAIT(..,req,..)
  - free from deadlock
Point to Point Communication

First Receive then Send

- **Blocking**
  - Blocking receive returns when copy from system buffer completed
  - Deadlock on any machine

- **Nonblocking**
  - Both processes: IRECV(..., req, ...), SEND, WAIT(...,req,..)
  - free from deadlock
Point to Point Communication

Receive / Send and Send / Receive

- Always save
- Use blocking or non-blocking subroutines
- Considering performance and the avoidance of deadlocks, use

```c
if (myrank==0) then
    call MPI_ISEND(sendbuf,...,ireq1,...)
    call MPI_Irecv(recvbuf,...,ireq2,...)
elseif (myrank==1) then
    call MPI_IRECV(recvbuf,...,ireq1,...)
    call MPI_ISEND(sendbuf,...,ireq2,...)
endif
call MPI_WAIT(ireq1,...)
call MPI_WAIT(ireq2,...)
```
Parallel Computing

Introduction
Example: Heat transfer in 2 Dimensions

MPI
What is MPI
Basic MPI Instructions
Words of Wisdom
How to parallelize your program
Domain decomposition

Words of Wisdom

- Whenever possible use collective operations
- Mind the succession of Send and Recvs
- Don’t reuse buffer, until operation is finished
- Don’t mix nonblocking ’point to point’ with collective operations
- Care about scalability

IF (myrank==0) THEN
  MPI_SEND(myrank, sum, myrank+1)
  MPI_RECV(sum, nproc-1)
  PRINT *,''The sum is '',sum
ELSE IF (myrank==nproc-1) THEN
  MPI_RECV(sum, myrank-1)
  sum=sum+myrank
  MPI_SEND(sum, 0)
ELSE
  MPI_RECV(sum, myrank-1)
  sum=sum+myrank
  MPI_SEND(sum, myrank+1)
END IF

CALL MPI_REDUCE(myrank, sum, MPI_SUM, 0)

IF (myrank==0)
  PRINT *,''The sum is '',sum
Words of Wisdom

- Whenever possible use collective operations
- Mind the succession of Send and Recvs
- Don’t reuse buffer, until operation is finished
- Don’t mix nonblocking 'point to point' with collective operations
- Care about scalability

CPU0
MPI_ISEND(.,cpu1,.,tag1,.)
MPI_ISEND(.,cpu1,.,tag2,.)
MPI_ISEND(.,cpu1,.,tag3,.)
MPI_ISEND(.,cpu1,.,tag4,.)

CPU1
MPI_RECV(.,cpu0,.,tag4,.)
MPI_RECV(.,cpu0,.,tag3,.)
MPI_RECV(.,cpu0,.,tag2,.)
MPI_RECV(.,cpu0,.,tag1,.)

Will allocate huge amount of memory. System crash possible.
Words of Wisdom

- Whenever possible use collective operations
- Mind the succession of Send and Recvs
- Don’t reuse buffer, until operation is finished
- Don’t mix nonblocking ’point to point’ with collective operations
- Care about scalability

MPI_ISEND(buffer,..,req,..)  
....
buffer(0)=something  
....
MPI_WAIT(req,ierror)  

MPI_Irecv(buffer,..,req,..)  
....
z=buffer(0)  
....
MPI_WAIT(req,ierror)  

Results in data race and wrong results.
Parallel Computing

Introduction
Example: Heat transfer in 2 Dimensions

MPI
What is MPI
Basic MPI Instructions
How to parallelize your program
Domain decomposition

Words of Wisdom

- Whenever possible use collective operations
- Mind the succession of Send and Recvs
- Don’t reuse buffer, until operation is finished
- Don’t mix nonblocking 'point to point' with collective operations
- Care about scalability

CPU0
MPI_ISEND(buffer,..,req,..)

CPU1
MPI_RECV(buffer,...)

On the ragged edge of legality. Will your MPI survive it?
Words of Wisdom

- Whenever possible use collective operations
- Mind the succession of Send and Recvs
- Don’t reuse buffer, until operation is finished
- Don’t mix nonblocking 'point to point' with collective operations
- Care about scalability

```
CPU1 to CPU_n
MPI_SEND(buffer,..,CPU0,..)

CPU0
do i=1,n
    MPI_RECV(buffer,..,CPU_i,..)
```

Not scalable! Don’t write such code.
Parallelization Strategy

Speed up

- Increase fraction of program that can be parallelized
- Balance the workload. Sometimes hard but worth it.
- Minimize communication time by
  - decreasing amount of data transmitted
  - decreasing number of times data is transmitted
- Minimize cache misses in loops
Decrease amount of data transmitted

Splitting in

- **one** dimension:
  communication
  \[ n^2 \times 2 \times w \times 1 \]

- **two** dimensions:
  communication
  \[ n^2 \times 2 \times w \times 2 / p^{1/2} \]

- **three** dimensions:
  communication
  \[ n^2 \times 2 \times w \times 3 / p^{2/3} \]

- \( w \) = width of halo
- \( n^3 \) = size of matrix
- \( p \) = number of processors
- cyclic boundary
- \( \rightarrow \) two neighbors in each direction

Optimal for \( p > 11 \)
Decrease number of times data is transmitted

- prefer domain decomposition with contiguous boundary
- copy uncontiguous memory to contiguous buffer and send it in one operation
- use derived data type, but it may be not optimal
Minimize cache misses

do j=jstart, jend
  do i=1, n
    a(i,j)=b(i,j)+c(i,j)
  end do
end do

minimize cache misses.
Minimize cache misses

\[
\begin{align*}
\text{do } & j = j_{\text{start}}, j_{\text{end}} \\
\text{do } & i = 1, n \\
& a(i, j) = b(i, j) + c(i, j) \\
\text{end do} \\
\text{end do}
\end{align*}
\]

minimize cache misses.

\[
\begin{align*}
\text{do } & j = 1, n \\
\text{do } & i = i_{\text{start}}, i_{\text{end}} \\
& a(i, j) = b(i-1, j) + b(i+1, j) \\
\text{end do} \\
\text{end do}
\end{align*}
\]

but, communication latency is much larger than memory latency.
Virtual topologies

**MPI_CART_CREATE**

Makes a new communicator to which topology information of an cartesian grid has been attached.

\[
\text{MPI_CART_CREATE ( comm, dim, procperdim, periods, reorder, newComm, ierr )}
\]

- **comm**: Input communicator
- **dim**: Number of dimensions of cartesian grid
- **procperdim**: Number of processes in each dimension, integer(dim)
- **periods**: Grid is periodic (true) or not (false), bool(dim)
- **reorder**: Ranking of processes may be reordered (true) or not (false)
- **newComm**: New communicator
- **ierror**: Return value
Virtual topologies

**MPI_CART_CREATE**

Makes a new communicator to which topology information of an cartesian grid has been attached.

\[
\text{procs}=\langle 4, 2, 3 \rangle \\
\text{periods}=\langle 1, 1, 1 \rangle \\
\text{MPI_CART_CREATE} (\text{MPI_COMM_WORLD}, 3, \text{procs, periods}, \& \\
\text{.true.}, \text{comm3d, ierror} )
\]

![Diagram of 3D grid with RANK=23 and (0,0,0) coordinate](attachment:image.png)
## MPI_CART_SHIFT

Returns shifted source and destination ranks.

\[
\text{MPI_CART_SHIFT (comm, direction, displ, source, dest, ierror )}
\]

- **comm**: Communicator with cartesian structure
- **direction**: Coordinate dimension of shift
- **displ**: Displacement (\(>0\): upwards, \(<0\) downwards)
- **source**: Rank of source process
- **dest**: Rank of destination process
- **ierror**: Return value

If the dimension specified by direction is non-periodic, off-end shifts result in the value MPI_PROC.NULL.
Virtual topologies

### MPI_CART_COORDS

Translates process rank in a communicator into cartesian coordinates.

\[
\text{MPI_CART_COORDS}(\text{comm, rank, maxdims, coords, ierror});
\]

- **comm**: Communicator with cartesian structure.
- **rank**: Rank of process within group comm.
- **dim**: Number of dimensions of cartesian grid.
- **coords**: Cartesian coordinates of the process.
- **ierror**: Return value
Virtual topologies

Example: Dimension=3, 24 Processes

call MPI_Cart_create(MPI_COMM_WORLD, dim, nproc, periods, 0, &comm3d, ierror)
call MPI_Cart_coords(comm3d, myRank, dim, coords,ierror)
call MPI_Cart_rank(comm3d, coords, rank3d, ierror)
call MPI_Cart_shift(comm3d, 0, 1, left ,right,ierror)
call MPI_Cart_shift(comm3d, 1, 1, down, up,ierror)
call MPI_Cart_shift(comm3d, 2, 1, south,north,ierror)

I am proc 13: coords(1)= 2
I am proc 13: coords(2)= 0
I am proc 13: coords(3)= 1
I am proc 13: left = 7
I am proc 13: right = 19
I am proc 13: down = 16
I am proc 13: up = 16
I am proc 13: north = 14
I am proc 13: south = 12
MPE

Method 0

- copy to buffer
- irecv, isend, waitall
Parallel Computing

Example: Heat transfer in 2 Dimensions

MPI

What is MPI

Basic MPI Instructions

Words of Wisdom

How to parallelize your program

Domain decomposition

MPI analysis tool

MPE

Method 1

- copy to buffer
- (irecv,isend / isend,irecv) , waitall
MPI analysis tool

MPE

Method 2
- copy to buffer
- sendrecv

Parallel Computing
Introduction
Example: Heat transfer in 2 Dimensions
MPI
What is MPI
Basic MPI Instructions
Words of Wisdom
How to parallelize your program
Domain decomposition
Scaling of a magnetohydrodynamic turbulence simulation

Speedup: how much faster is the computation on \( x \) processors than on 256