Introduction to the Message Passing Interface - MPI

Kevin Brown
Argonne National Laboratory

Donald Frederick
Lawrence Livermore National Laboratory

Lawrence Livermore National Laboratory

Tapia Conference 2022

LLNL-CONF-837957-DRAFT

This work was performed under the auspices of the U.S. Department of Energy by Argonne National Laboratory under Contract DE-AC02-06-CH11357, Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, Los Alamos National Laboratory under Contract DE-AC5206NA25396, and Oak Ridge National Laboratory under Contract DE-AC05-00OR22725.

Agenda

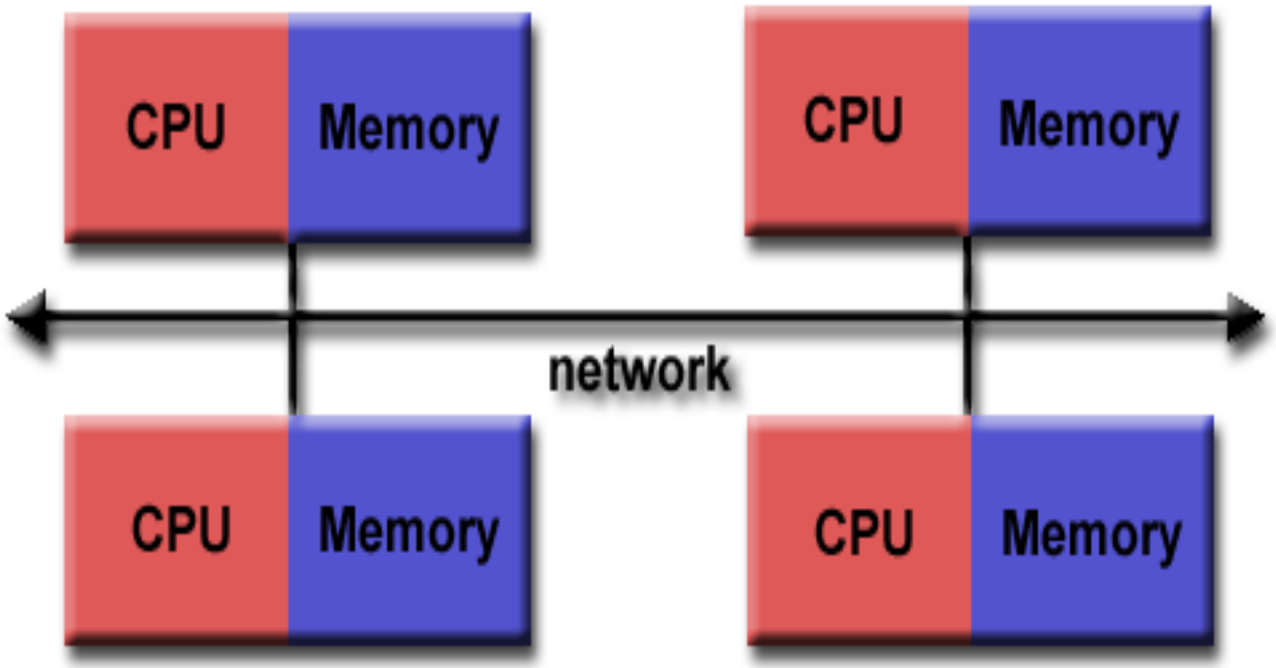
- What is MPI?
- Why does one use MPI?
- MPI References

What is MPI ?

- **An Interface Specification -M P I** = **M**essage **P**assing **I**nterface
- MPI is a specification for the developers and users of message passing libraries. It is the specification of what such a library should be.
- MPI *addresses the message-passing parallel programming model*: data is moved from the address space of one process to that of another process through cooperative, concurrent operations on each process.
- The goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs. The interface attempts to be:
  - Practical
  - Portable
  - Efficient
  - Flexible
- The MPI standard has gone through a number of revisions, with the most recent version being MPI-3.x
- Interface specifications have been defined for C and Fortran90 language bindings:
  - C++ bindings from MPI-1 are removed in MPI-3
- MPI-3 also provides support for Fortran 2003 and 2008 features
- Actual MPI library implementations differ in which version and features of the MPI standard they support.

Parallel Programming Model

Programming Model - Originally, MPI was designed for distributed memory architectures, which were becoming increasingly popular at that time (1980s - early 1990s).



Parallel Programming Model

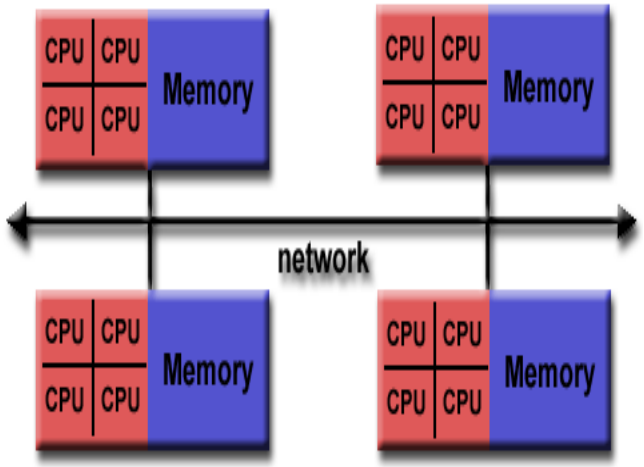
- As architecture trends changed, shared memory SMPs were combined over networks creating hybrid distributed memory / shared memory systems.
- MPI implementors adapted their libraries to handle both types of underlying memory architectures seamlessly. They also adapted/developed ways of handling different interconnects and protocols.
- Today, MPI runs on virtually any parallel hardware platform:
  - Distributed Memory
  - Shared Memory
  - Hybrid

MPI Basics

- Multiple, concurrent tasks launched – each with own control flow, memory
- Tasks communicate via messages – sent/received via a hardware communication channel.
- Messages contain data, can be used to coordinate concurrent tasks

Parallel Programming Model

This is the common platform for MPI-based parallel programming today.



Parallel Programming Model

Reasons for Using MPI

- **MPI allows concurrent use of multiple nodes/cpus/core in a linux cluster.**
- **Standardization** - MPI is the only message passing library that can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.
- **Portability** - There is little or no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.
- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance. Any implementation is free to develop optimized algorithms.
- **Functionality** - There are over 430 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.
  - NOTE: Most MPI programs can be written using a dozen or less routines
- **Availability** - A variety of implementations are available, both vendor and public domain.

MPI Components

- MPI Library – there are many library implementations
- MPI run-time infrastructure – implementation-dependent
  - MPI task launcher – varies with library implementation

  Often, ”mpirun” used as MPI task launcher

  Example -

  mpirun –n 64 ./a.out

  Launches 64 MPI tasks

  - MPI Environment variables – implementation-dependent

  Example -

MPI Is “Small” and “Big”

- Small

One can implement wide range of message-passing algorithms using just 6 basic MPI routines – send/receive messages.

- Big

Latest version (June 2021) of MPI Standard has over 300 routines

MPI – Program Structure

Most MPI program can be written with 6 basic MPI functions

- MPI Initialization

Define region of program executing on multiple MPI tasks

- Basic MPI Send/Receive messages

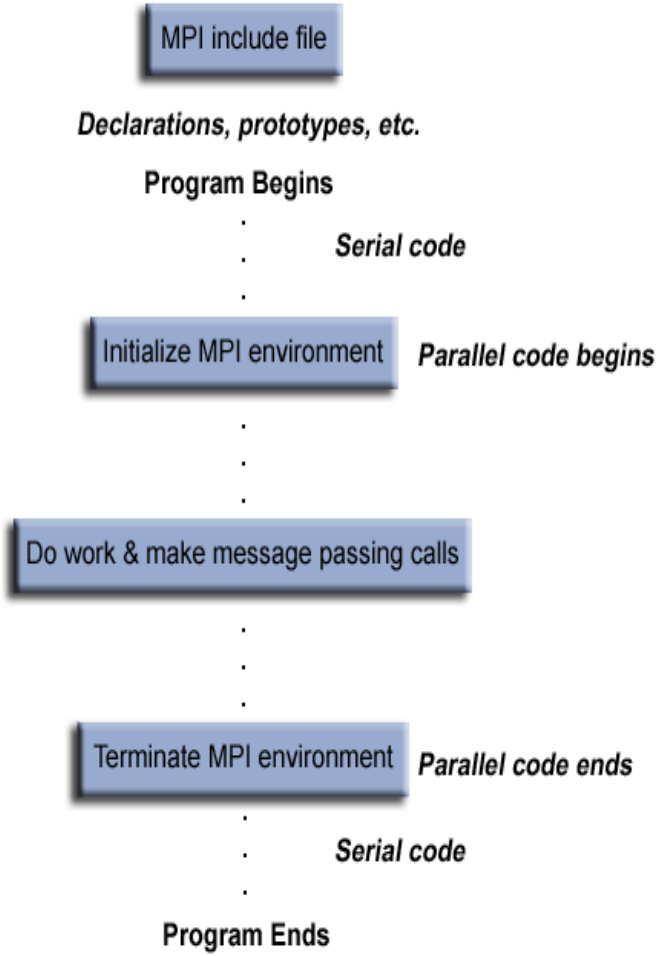
Sending/Receiving information - for computation, program control

- MPI “interrogation” functions

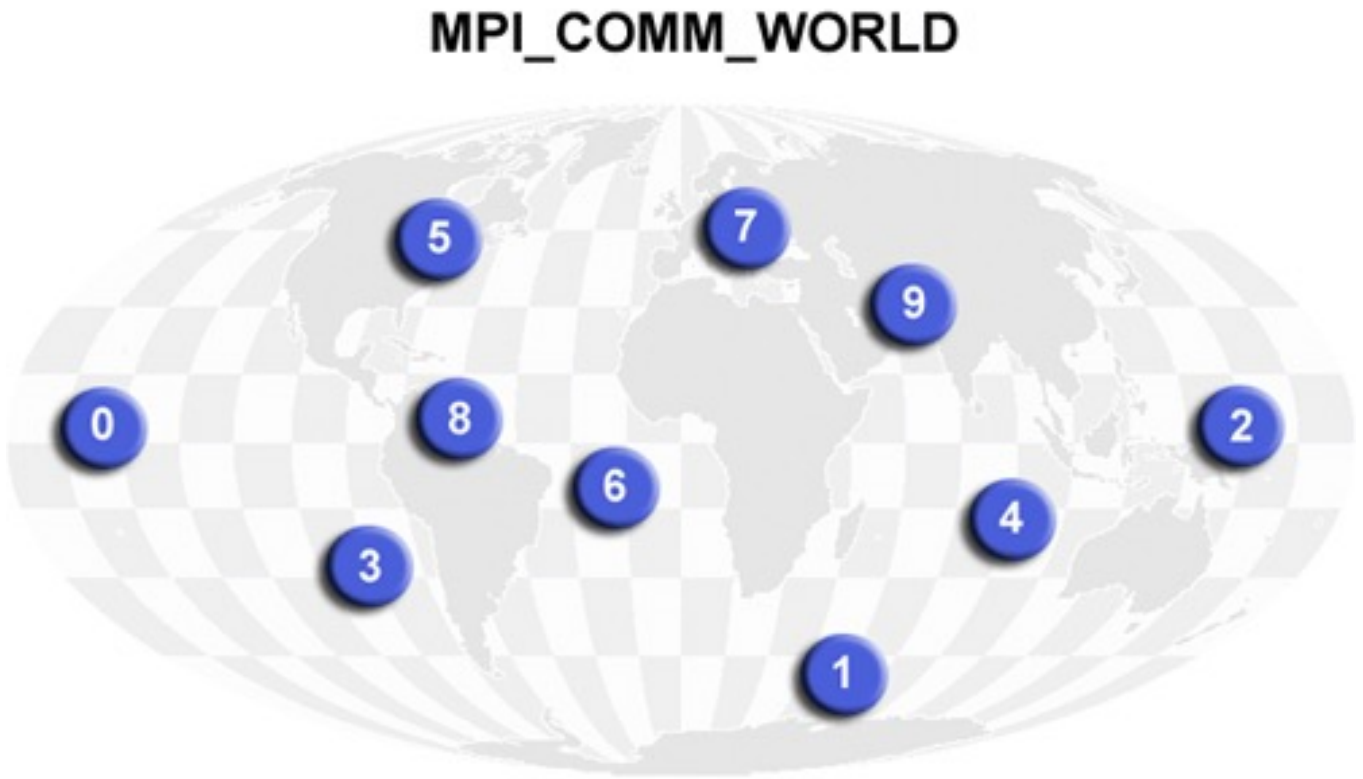
Information for program control

- MPI “finalization” – end MPI context

Program Structure - MPI



MPI Program Structure -



Header File:
Required for all programs that make MPI library calls.

MPI – Program Structure – Components: Include File

C include file	Fortran include file
#include "mpi.h"	include 'mpif.h'

MPI – Program Structure

Format of MPI Calls:

C Binding	
Format:	**rc = MPI_Xxxxx(parameter, ...)**
Example:	**rc = MPI_Bsend(&buf,count,type,dest,tag,comm)**
Error code:	Returned as "rc". MPI_SUCCESS if successful
Fortran Binding	
Format:	**CALL MPI_XXXXX(parameter,..., ierr)** **call mpi_xxxxx(parameter,..., ierr)**
Example:	**CALL MPI_BSEND(buf,count,type,dest,tag,comm,ierr)**
Error code:	Returned as "ierr" parameter. MPI_SUCCESS if successful

MPI Environment Management/Interrogation Functions

MPI_Init

- Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program. For C programs, MPI_Init may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent.

MPI_Init (&argc,&argv)

MPI_INIT (ierr)

MPI Environment Management/Interrogation Functions

MPI_Comm_size

- Returns the total number of MPI processes in the specified communicator, such as MPI_COMM_WORLD. If the communicator is MPI_COMM_WORLD, then it represents the number of MPI tasks available to application.

MPI_Comm_size (comm,&size)

MPI_COMM_SIZE (comm,size,ierr)

MPI Environment Management/Interrogation Functions

MPI_Comm_rank

- Returns the rank of the calling MPI process within the specified communicator. Each MPI process is assigned a unique integer rank between 0 and number of tasks - 1 within the communicator MPI_COMM_WORLD. This rank is often referred to as a task ID.

MPI_Comm_rank (comm,&rank)

MPI_COMM_RANK (comm,rank,ierr)

MPI Environment Management/Interrogation Functions

MPI_Finalize

- Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.

MPI_Finalize ()

A Simple MPI Program – Hello World

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define  MASTER                 0

int main (int argc, char *argv[])
{
int   numtasks, taskid, len;
char hostname[MPI_MAX_PROCESSOR_NAME];
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
MPI_Get_processor_name(hostname, &len);
printf ("Hello from task %d on %s!\n", taskid,
hostname);
if (taskid == MASTER)
  printf("MASTER: Number of MPI tasks is:
%d\n",numtasks);
MPI_Finalize();
}
```

A Simple MPI Program – Hello World Output – Run on One Node

mpirun –n6 hello.x

- Task launcher mpirun launches 6 mpi tasks

MASTER: Number of MPI tasks is: 6

Hello from task 1 on ruby238!

Hello from task 4 on ruby238!

Hello from task 5 on ruby238!

Hello from task 3 on ruby238!

Hello from task 2 on ruby238!

Hello from task 0 on ruby238!

In this case, all 6 MPI tasks ran on one node. However, MPI allows running across **multiple** nodes.

A Simple MPI Program – Hello World Output – Run Across 2 Nodes

```
Hello from task 33 on ruby138!
Hello from task 64 on ruby139!
Hello from task 68 on ruby139!
Hello from task 34 on ruby138!
Hello from task 32 on ruby138!
Hello from task 18 on ruby138!
Hello from task 21 on ruby138!
Hello from task 19 on ruby138!
Hello from task 20 on ruby138!
Hello from task 28 on ruby138!
Hello from task 35 on ruby138!
Hello from task 24 on ruby138!
Hello from task 36 on ruby139!
Hello from task 65 on ruby139!
Hello from task 69 on ruby139!
Hello from task 66 on ruby139!
Hello from task 67 on ruby139!
Hello from task 10 on ruby138!
Hello from task 25 on ruby138!
Hello from task 29 on ruby138!
Hello from task 26 on ruby138!
Hello from task 49 on ruby139!
Hello from task 9 on ruby138!
Hello from task 48 on ruby139!
Hello from task 17 on ruby138!
Hello from task 58 on ruby139!
Hello from task 60 on ruby139!
Hello from task 70 on ruby139!
Hello from task 40 on ruby139!
Hello from task 42 on ruby139!
Hello from task 50 on ruby139!
Hello from task 51 on ruby139!
Hello from task 61 on ruby139!
Hello from task 63 on ruby139!
Hello from task 71 on ruby139!
Hello from task 8 on ruby138!
Hello from task 46 on ruby139!
Hello from task 41 on ruby139!
Hello from task 30 on ruby138!
Hello from task 22 on ruby138!
Hello from task 31 on ruby138!
Hello from task 37 on ruby139!
Hello from task 52 on ruby139!
Hello from task 27 on ruby138!
Hello from task 23 on ruby138!
Hello from task 47 on ruby139!
Hello from task 16 on ruby138!
Hello from task 4 on ruby138!
Hello from task 53 on ruby139!
Hello from task 5 on ruby138!
Hello from task 11 on ruby138!
Hello from task 39 on ruby139!
Hello from task 45 on ruby139!
Hello from task 44 on ruby139!
```

…

…

Hello from task 43 on ruby139!

MASTER: Number of MPI tasks is: 72

In this case, each ruby node has 36 cores and 1 MPI task ran on each core

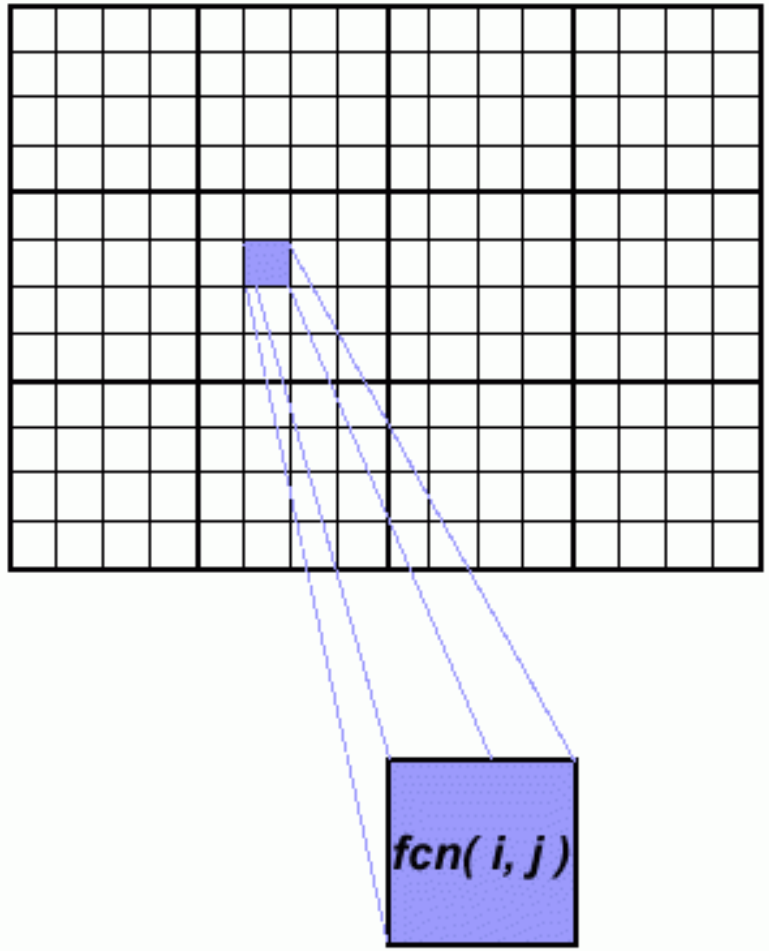
A Simple MPI Program 2-D Array Assignment Example

This example demonstrates calculations on 2-dimensional array elements;

- a function is evaluated on each array element.
- The computation on each array element is independent from other array elements.
- The problem is computationally intensive
- The serial program calculates one element at a time in sequential order.

Serial code for this may look like:

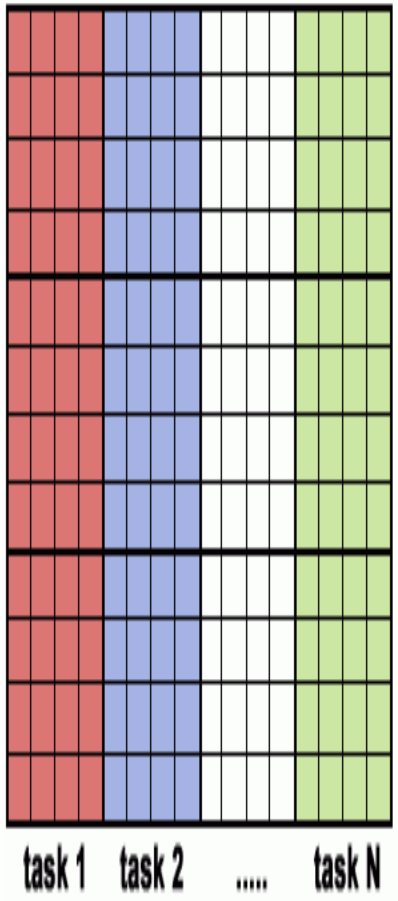
```
do j = 1,n
 do i = 1,n a(i,j) = fcn(i,j)
end do
end do
```



A Simple MPI Program 2-D Array Assignment Example – Paralleism

The calculation of elements is independent of one another - leads to an embarrassingly parallel solution.

- Arrays elements are evenly distributed so that each process owns a portion of the array (subarray).
- Distribution scheme is chosen for efficient memory access; e.g. unit stride (stride of 1) through the subarrays.
- Unit stride maximizes cache/memory usage.
- Independent calculation of array elements ensures there is no need for communication or synchronization between tasks.
- After the array is distributed, each task executes the portion of the loop corresponding to the data it owns.

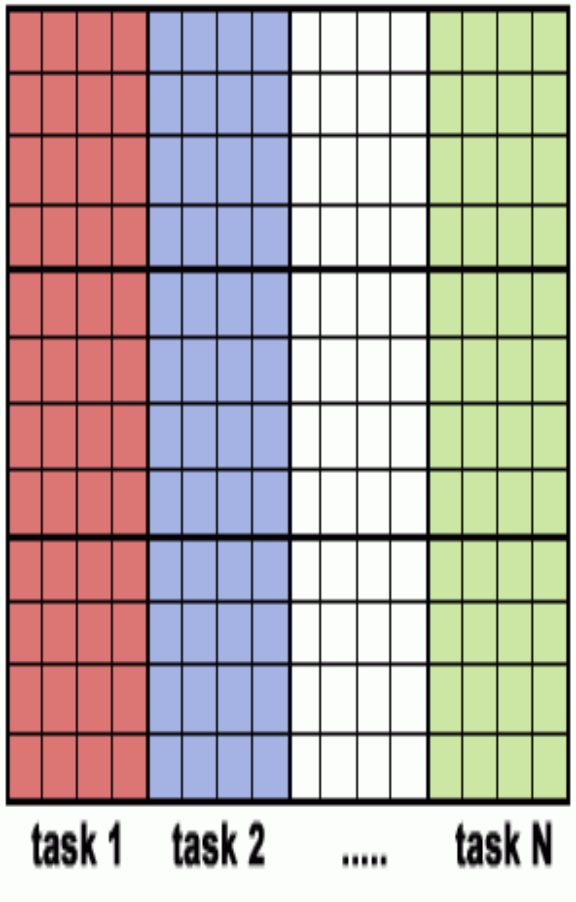


A Simple MPI Program 2-D Array Assignment Example - Parallelism

Pseudocode example showing changes for parallel implementation

in red

```
for i (i = mystart; i < myend; i++)
{   for j (j = 0; j < n; j++)
    {   a(i,j) = fcn(i,j);
}
}
```



A Simple MPI Program 2-D Array Assignment Example – Pseudocode for Parallel case

```
find out if I am MASTER or
WORKER

if I am MASTER

  initialize the array

  send each WORKER info on
part of array it owns

  send each WORKER its
portion of initial array
```

```
else if I am WORKER
  receive from MASTER info on part of array I own
receive from MASTER my portion of initial array
calculate my portion of array
do j = my first column,my last column
  do i = 1,n a(i,j) = fcn(i,j)
 end do
end do
send MASTER results
endif
```

A Simple MPI Program – Vector Dot Product

Vector Dot Product (scalar), of two vectors, A, B

$$A*B = \sum a_i * b_i$$

MPI solution – partition work among "N" mpi tasks so that each does partial sum for total of M elements of the vectors

$$A*B = \sum a_i * b_i = \sum \quad (\sum_i^M a_i * b_i)$$

A Simple MPI Program – Vector Dot Product Pseudocode

```
Am I BOSS or WORKER ?

if I am BOSS
   initialize the array
   send each WORKER info on part of array
it owns
   send each WORKER its portion of initial
array (psum)
 receive from each WORKER
results = psum
DotProd = Sum of psums

else if I am WORKER

   receive from BOSS info on part of
array I own

   receive from BOSS  my portion of
initial array

   # calculate my portion of array

      do j = my first column,my last

         psum = a(i)* b(i))

      end do

send BOSS  psum

 endif
```

A Simple MPI Program – Vector Dot Product - Output

mpirun – n16 ./aout Launch 16 MPI tasks for DoProduct example

Output for example –

```
MPI task 0 has started...
Initialized array serial dot product = 3.200000e+07
Sent 4000000 elements to task 1 offset= 4000000
Sent 4000000 elements to task 2 offset= 8000000
Sent 4000000 elements to task 3 offset= 12000000
Task 0 mysum = 8.000000e+06
MPI task 1 has started...
Task 1 mysum = 8.000000e+06
MPI task 2 has started...
Task 2 mysum = 8.000000e+06
MPI task 3 has started...
Task 3 mysum = 8.000000e+06
…
…

*** Final parallel dot  product = 3.200000e+07 ***
```

A Simple MPI Program – Matrix- Matrix Multiply

Matrix-Matrix multiply (matrix), of two matrices, A, B – produces another matrix, C

A * B = C

Or matrix elements of C, $C_{ij} = \sum A_{ik} B_{kj}$

Where the sum is over the k elements of A, B

Calculating PI – Using Monte Carlo Method

- Inscribe a circle with radius **r** in a square with side length of ***2r***
- The area of the circle is **Πr2** and the area of the square is **4r2**
- The ratio of the area of the circle to the area of the square is:
  Πr2 / 4r2 = Π / 4

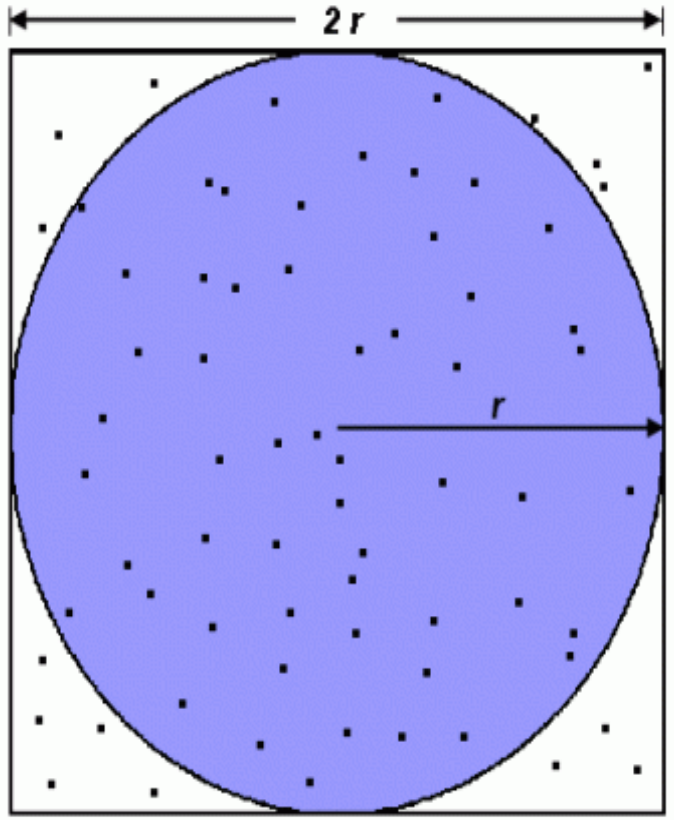
If One randomly generates **N** points inside the square, approximately

N * Π / 4 of those points (**M**) should fall inside the circle.

Π is then approximated as:

N * Π / 4 = M
Π / 4 = M / N
Π = 4 * M / N

Note that increasing the number of points generated improves the approximation.



$$A_S = (2r)^2 = 4r^2$$

$$A_C = \pi r^2$$

$$\pi = 4 \times \frac{A_C}{A_S}$$

Calculating PI – Using Monte Carlo Method

Pseudocode for serial algorithm

```
 npoints = 10000
circle_count = 0
do j = 1,npoints
        generate 2 random numbers between 0 and 1
        xcoordinate = random1
        ycoordinate = random2
        if (xcoordinate, ycoordinate) inside circle
        then circle_count = circle_count + 1
end do
PI = 4.0*circle_count/npoints
```

Calculating PI – Using Monte Carlo Method

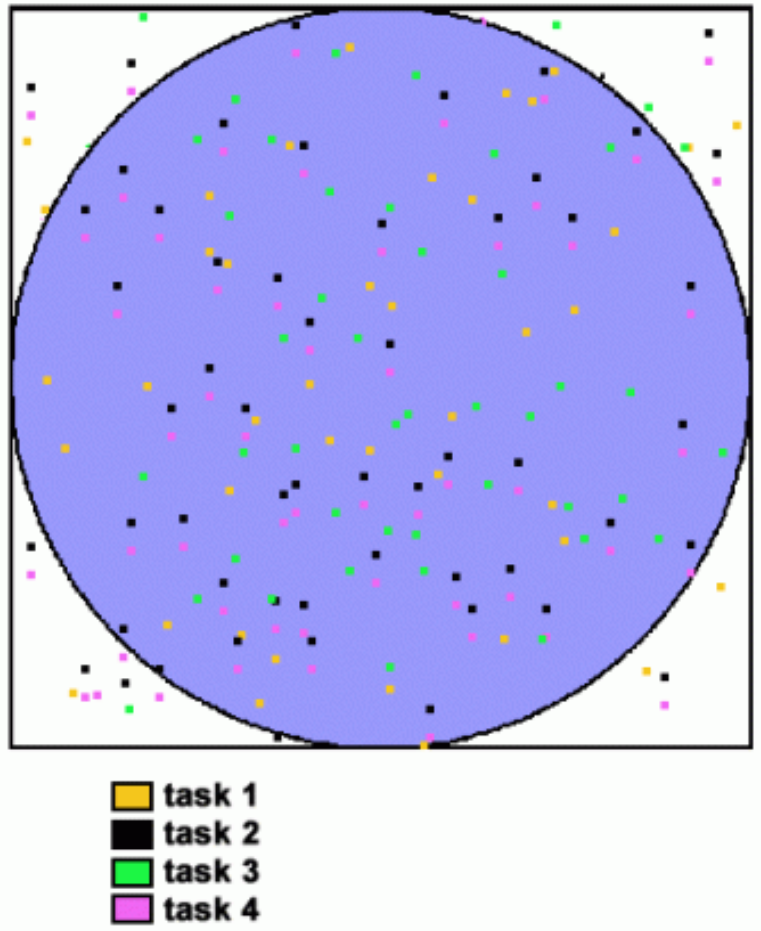
A problem that's easy to parallelize:

- All point calculations are **independent**; no data dependencies
- Work can be evenly divided; no load balance concerns
- No need for communication or synchronization between tasks

Parallel strategy:

- Divide the loop into equal portions that can be executed by the pool of tasks
- Each task independently performs its work
- A SPMD model is used
- One task acts as the master to collect results and compute the value of PI

Pseudo code solution: red highlights changes for parallelism.



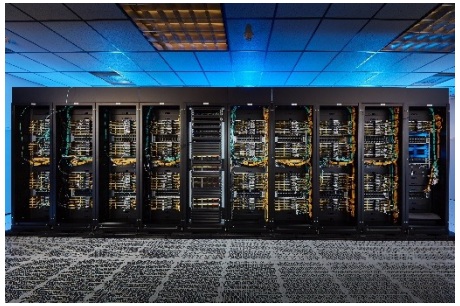
Calculating PI – Using Monte Carlo Method - Pseudocode

```
npoints = 10000

circle_count = 0

p = number of tasks

num = npoints/p
```

```
find out if I am MASTER or WORKER
do j = 1,num
  generate 2 random numbers between 0 and 1
xcoordinate = random1
ycoordinate = random2
if (xcoordinate, ycoordinate) inside circle
then circle_count = circle_count + 1
end do
 if I am MASTER
    receive from WORKERS their circle_counts
    compute PI (use MASTER and WORKER calculations)
else if I am WORKER
   send to MASTER circle_count
endif
```

Systems for Running MPI - Examples

Lassen



Corona

Sierra

El Capitan

Quartz

Ruby

Today

On Your Laptop

4 MPI Exercises

Examples have been built and tested using gcc/10.2.1, mvapich2/2.3

- MPI Hello World example
- MPI Array assignment example
- MPI Dot Product of Two Vectors
- MPI Multiply Two Matrices
- MPI Compute Pi using Monte Carlo Method

Building examples – after modifying makefile for your specific compiler MPI library

- make mpi_hello
- make mpi_array
- make mpi_DotProd
- make mpi_mm
- make mpi_pi_send

5 MPI Exercises

If “a.out” is the MPI executable, run with

(for 4 MPI tasks)

mpirun –np 4 ./a.out

More MPI Information – References

- LLNL Livermore Computing MPI Tutorial: **https://hpc-tutorials.llnl.gov/mpi/**
- MPI Standard documents: http://www.mpi-forum.org/docs/
- MPI Tutorials: www.mcs.anl.gov/research/projects/mpi/tutorial
- MPI libraries that can be downloaded and installed on laptop
  - MPICH MPI Library - https://www.mpich.org/
  - MVAPICH MPI Library – https://mvapich.cse.ohio-state.edu/
  - OpenMPI MPI Library – https://www.open-mpi.org/

HPC References for MPI - Books

- ”Parallel and High Performance Computing”, Robey and Zamora
- “An Introduction to Parallel Computing”, Pacheco and Malensek

Lawrence Livermore
National Laboratory