PARALLEL PROGRAMMING USING MPI – SPEEDUP AND MEMORY

KRISHNENDU GHOSH
GRADUATE STUDENT
ELECTRICAL ENGINEERING
UNIVERSITY AT BUFFALO

NOVEMBER 4, 2016

University at Buffalo
School of Engineering and Applied Sciences

BEFORE WE BEGIN

- **A FEW THINGS …**
  - *This is more of an introductory workshop on MPI*
  - *No prior background on MPI is needed*
  - *Some basic background on sequential programming would be assumed.*
  - *Examples will be shown in FORTRAN. The concepts are equally valid for C.*
  - *The codes for the examples are written by the speaker himself. Please be ‘aggressive’ to point out any ‘shabby’ coding habits that you come across.*
  - *The remarks and conclusions made hereafter are solely the speaker’s personal opinions. The speaker does not represent any of the official implementers of MPI.*

OUTLINE

- PARALLEL PROGRAMMING
- GETTING STARTED WITH MPI
- SPEEDUP
- MPI SHARED MEMORY
- SUMMARY

University at Buffalo
The State University of New York

OUTLINE

- **PARALLEL PROGRAMMING**
- GETTING STARTED WITH MPI
- SPEEDUP
- MPI SHARED MEMORY
- SUMMARY

WHY PARALLEL PROGRAMMING?

❑ <u>**THE BIG PICTURE**</u>

Growth of processor clock-rate is slowing down – the main reasons being :

- size limits of the processor components
- higher power consumption at higher clock rates.

What does that mean to us?

- **Slow execution of codes and more time to reach solution**
- **Limit on available memory**

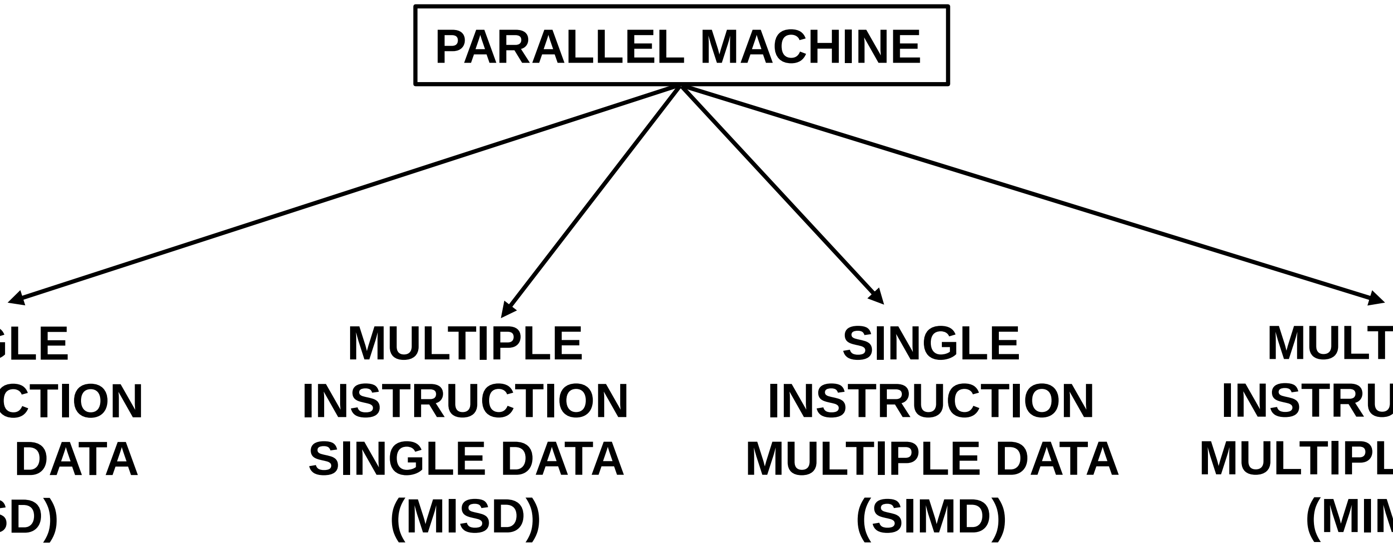
But that's limited by processor manufacturing. What do we programmers do?

- **Simple, use multiple processors**

PARALLEL COMPUTER ARCHITECTURE

T. Rauber and G. Runger, Parallel Programming, Springer 2013

- **FLYNN'S TAXONOMY**


PARALLEL MACHINE
SINGLE INSTRUCTION SINGLE DATA (SISD)
MULTIPLE INSTRUCTION SINGLE DATA (MISD)
SINGLE INSTRUCTION MULTIPLE DATA (SIMD)
MULTIPLE INSTRUCTION MULTIPLE DATA (MIMD)

Single processing element, single data memory.
Ex: a sequential computer

Multiple processing element ,single data memory.
Ex: No such commercial machine exists

Multiple processing elements - single instruction set (one program memory) and private data memories
Ex: single core computers

Multiple processing elements - private instruction sets and private data memories.
Ex: modern clusters.

University at Buffalo
The State University of New York

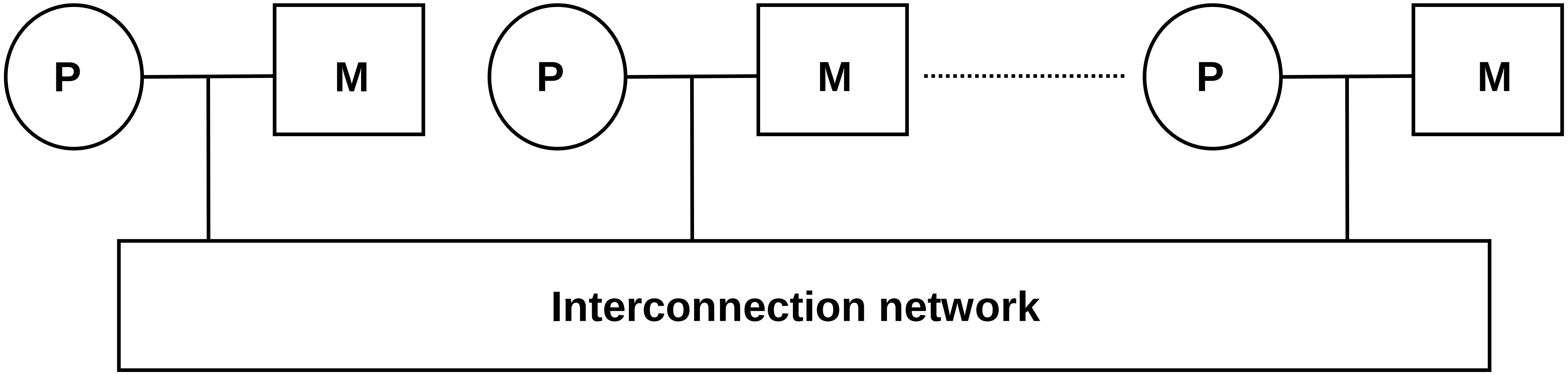
OUTLINE

- PARALLEL PROGRAMMING
- **GETTING STARTED WITH MPI**
- SPEEDUP
- MPI SHARED MEMORY
- SUMMARY

WHAT IS MPI?

W.P. Petersen and P. Arbenz, Introduction to Parallel Programming
Oxford University Press, 2004

- **<u>An MIMD system</u>**



- Message Passing Interface (MPI) establishes data access between nodes
- MPI is a standard message passing system on both shared and distributed systems
- A communicator group of processes perform a certain task
- Arbitrary master-slave relation

University at Buffalo
The State University of New York

GETTING ACCESS TO MPI

❑ **MPI IMPLEMENTATIONS**

MPICH (http://www.mpich.org/), Argonne National Laboratory

MVAPICH (http://mvapich.cse.ohio-state.edu/), Ohio State University

OpenMPI (https://www.open-mpi.org/), U of Tennessee, Los Alamos National Laboratory, Indiana University, and U of Stuttgart

INTEL MPI (https://software.intel.com/en-us/intel-mpi-library), Intel

❑ **MPI ON CCR AT UB**

```
module avail intel-mpi
module avail mvapich2
module avail openmpi
```

} Different modules available under each of these. Try to use the latest ones to get new features.

➢ Some implementations might have hardware interconnect specific modules. Load the one appropriate for you.

University at Buffalo
The State University of New York

FIRST PROGRAM

❑ HELLO WORLD WITH MPI

We want each proc to print its id

```
!-------------------------------------------
PROGRAM hello_world
!-------------------------------------------
USE mpi
!
IMPLICIT NONE
!
INTEGER       :: id, ierr, p
!
CALL mpi_init(ierr)
CALL mpi_comm_rank (MPI_COMM_WORLD, id, ierr)
CALL mpi_comm_size (MPI_COMM_WORLD, p, ierr)
!
PRINT '(" Total number of procs", i3, "  and my id is ", i3)',p, id
!
CALL mpi_finalize(ierr)
!
!
END PROGRAM hello_world
```

Initializes the MPI environment, ierr is almost ubiquitous in MPI FORTRAN, denotes the error code incase of an error

Gives id of a proc

Gives number of procs

Global Communicator, a predefined constant

Kills MPI

University at Buffalo
The State University of New York

BUILDING YOUR CODE WITH MPI

BUILD ➡

```
#!/bin/bash

module load intel/15.0
module load intel-mpi/5.0.2
which mpiifort
export MPIF90=mpiifort
$MPIF90 -traceback -g hello_world.f90 -o hello.x
```

LAUNCH ➡

```
# count the number of processors
np=`srun hostname -s | wc -l`
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so
srun -n $np hello.x
```

Integrates Intel-MPI to SLURM

OUTPUT ➡

```
Total number of procs  8  and my id is  0
Total number of procs  8  and my id is  1
Total number of procs  8  and my id is  2
Total number of procs  8  and my id is  3
Total number of procs  8  and my id is  4
Total number of procs  8  and my id is  5
Total number of procs  8  and my id is  6
Total number of procs  8  and my id is  7
```

BASIC MPI COMMANDS

We will only discuss a partial list of commands that are commonly used.

❑ **ENVIRONMENT MANAGEMENT**

4 such routines are used and explained in the hello_world program (slide 10)

❑ **POINT-TO-POINT COMMUNICATIONS**

MPI_SEND (buf, count, datatype, dest, tag, comm, ierr)

MPI_RECV (buf, count, datatype, source, tag, comm, status, ierr)

- **buf** is the data sent (received).
- **count** is the length of the data (like number of elements in an array)
- **dest** (**source**) is which proc id the data is going to (coming from)
- **tag** is an unique identification to a message. send tag = receive tag

  (wild card MPI_ANY_TAG will receive any message)

Petersen and Arbenz, IPP, Oxford University Press, 2004

University at Buffalo
The State University of New York

BASIC MPI COMMANDS

❑ EXAMPLE – *send and receive*

We want to send the proc ids of each proc to all other procs by point-to-point communication

```
DO iproc = 1, np
  !
  src = iproc - 1
  !
  DO jproc = 1, np
    !
    dest = jproc - 1
    !
    IF (p_id .EQ. src .AND. .NOT. (dest.EQ.src)  )  &
      CALL MPI_SEND (p_id, 1, MPI_INTEGER, dest , 10, MPI_COMM_WORLD, ierr)
    !
    IF (p_id .EQ. dest .AND. .NOT. (dest.EQ.src)  ) THEN
       CALL MPI_RECV (p_id_, 1, MPI_INTEGER,  src , 10, MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
       PRINT '(" I am proc", i3, " and my buddy is proc", i3 )',p_id, p_id_
    ENDIF
    !
    CALL MPI_BARRIER(MPI_COMM_WORLD, ierr)
    !
  ENDDO
  !
ENDDO
```

src sends its p_id to dest

p_id is the id of the current proc

dest receives it

note both tags are equal

p_id_ is the id of the sending proc

Will come to this in next slide

OUTPUT ➡

```
Total number of procs is   2

I am proc  0 and my buddy is proc  1
I am proc  1 and my buddy is proc  0
```

University at Buffalo
The State University of New York

BASIC MPI COMMANDS

❑ <u>**COLLECTIVE COMMUNICATIONS – *broadcast, barrier, and reduction***</u>

MPI_BARRIER (comm, ierr)

Petersen and Arbenz, IPP, Oxford University Press, 2004

➢ Each proc in **comm** waits until all of them reach at this step

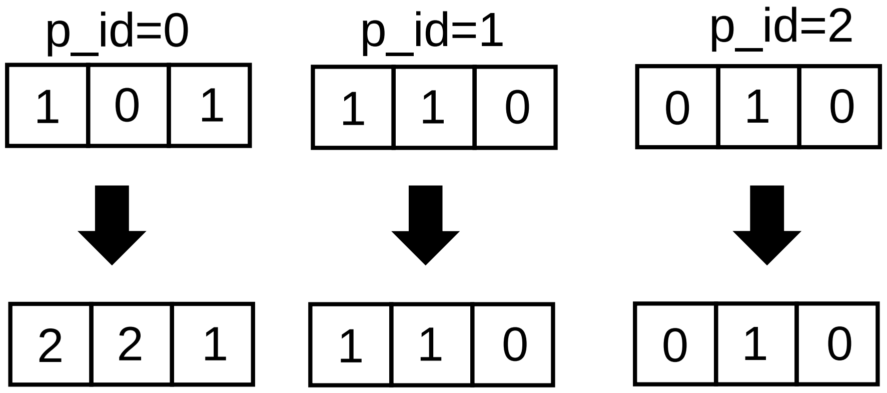
MPI_BCAST (data, count, datatype, root, comm, ierr)

➢ Proc **root** spreads **data** of length **count** to all other procs in **comm**.

MPI_REDUCE (segment, result, count, datatype, operator, root, comm, ierr)

➢ Proc dependent **segments** are merged as **result** in **root**. The merging is defined by **operator.**

MPI_REDUCE
(with operator = MPI_SUM and root=0)



University at Buffalo
The State University of New York

BASIC MPI COMMANDS

- **EXAMPLE – *broadcast and barrier***

We want one proc to read input and broadcast among others

```
! ---------------- Read input arguments --------------------
! --- Only one proc reads and then bcasts to others --------
!
IF (p_id.EQ.master) THEN
  !
  DO iiarg = 1, iargc()
    !
    CALL getarg( iiarg, inp_string(iiarg) )
    !
  ENDDO
  !
  READ ( inp_string(1), *) flag
  READ ( inp_string(2), *) m
  READ ( inp_string(3), *) n
  !
ENDIF
!
CALL mpi_barrier (MPI_COMM_WORLD, ierr)
CALL mpi_bcast(m,1,MPI_INTEGER,master,MPI_COMM_WORLD,ierr)
CALL mpi_bcast(n,1,MPI_INTEGER,master,MPI_COMM_WORLD,ierr)
CALL mpi_bcast(flag,10,MPI_CHARACTER,master,MPI_COMM_WORLD,ierr)
```

Only p_id = master does the i/o job

Other procs wait until master is done

Input data is broadcasted to other procs

University at Buffalo
The State University of New York

BASIC MPI COMMANDS

- **EXAMPLE – *reduction***

We want to compute $\{c\}_{n\times1} = [A]_{n\times m}\{b\}_{m\times1}$

```
! ----------- Split sample among procs --------------
!
n_per_proc = n / np
rest = MOD ( n , np)
IF (p_id .LE. (rest - 1)) n_per_proc = n_per_proc + 1
!
lower_bound = n_per_proc * p_id + 1
IF (p_id .GE. rest ) lower_bound = (n_per_proc + 1) * rest + n_per_proc * (p_id - rest) + 1
upper_bound = lower_bound + n_per_proc - 1
 ............
  ............

! ----------- Compute the proc wise split parts and then reduce to a single proc ---------
!
DO i = lower_bound, upper_bound
  !
  DO j = 1, m
     !
     c(i) = c(i) + A(i,j)*b(j)
     !
  ENDDO
  !
ENDDO
!
CALL mpi_barrier (MPI_COMM_WORLD, ierr)
CALL mpi_reduce (c,c_tot,n,MPI_FLOAT,MPI_SUM,master,MPI_COMM_WORLD,ierr)
```

n is split among np. Each proc gets n_per_proc

when mod (n, np) ≠ 0

index bounds on each proc

Each proc computes its own share

Total $\{c\}_{n\times1}$ reduced to master

University at Buffalo
The State University of New York

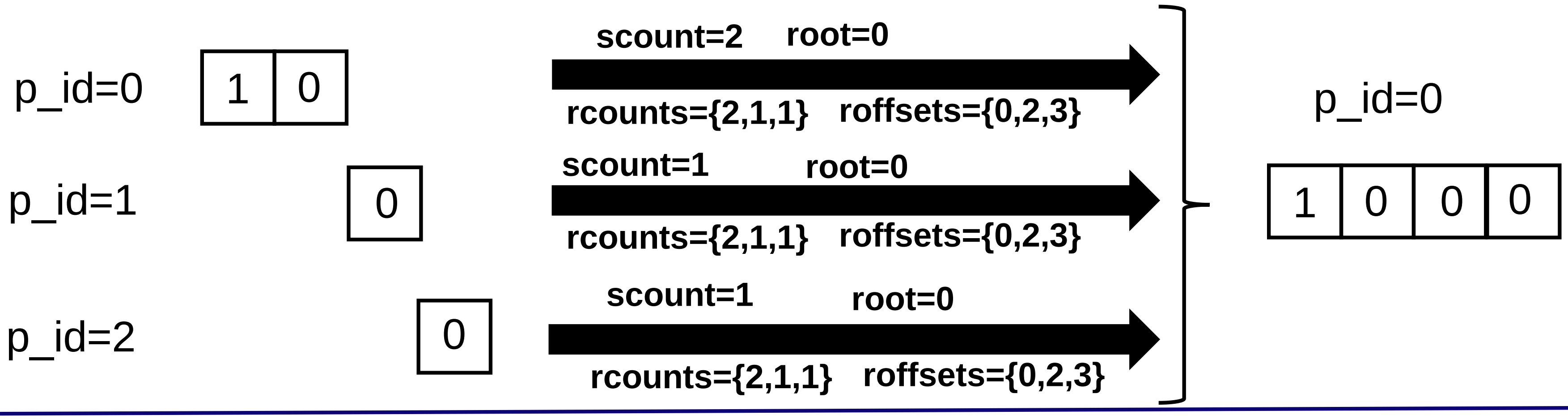
BASIC MPI COMMANDS

❑ <u>COLLECTIVE COMMUNICATIONS – *scatter(v) and gather(v)*</u>

- **MPI_SCATTER (MPI_GATHER)** – distribute (assemble) data segments
- **MPI_SCATTERV (MPI_GATHERV)** – additional features to non-uniformly distribute (assemble) data segments.
- We discuss the latter ones as that would clarify the former ones too.

Petersen and Arbenz, IPP, Oxford University Press, 2004

MPI_GATHERV (segment, scount, stype, result, rcounts, roffsets, rtype, root, comm, ierr)


p_id=0
1 0
scount=2 root=0
rcounts={2,1,1} roffsets={0,2,3}
p_id=1
0
scount=1 root=0
rcounts={2,1,1} roffsets={0,2,3}
p_id=2
0
scount=1 root=0
rcounts={2,1,1} roffsets={0,2,3}
p_id=0
1 0 0 0

University at Buffalo
The State University of New York

BASIC MPI COMMANDS

❑ **EXAMPLE – *gatherv***

Again we want to compute $\{c\}_{n\times 1} = [A]_{n\times m}\{b\}_{m\times 1}$

```
DO i = 1, n_per_proc
  !
  DO j = 1, m
     !
     c(i) = c(i) + A(i,j)*b(j)
     !
  ENDDO
  !
ENDDO
..............
counts(:) = 0
offsets(:) = 0
counts(p_id+1) = n_per_proc
offsets(p_id+1) = lower_bnd - 1
..............

CALL mpi_barrier(MPI_COMM_WORLD, ierr)
CALL mpi_allreduce (counts, all_counts, np, MPI_INTEGER, MPI_SUM, MPI_COMM_WORLD, ierr)
CALL mpi_allreduce (offsets, all_offsets, np, MPI_INTEGER, MPI_SUM, MPI_COMM_WORLD, ierr)
!
CALL mpi_gatherv(c,n_per_proc,MPI_FLOAT,c_tot,all_counts,all_offsets,MPI_FLOAT,master,&
                                                                    MPI_COMM_WORLD,ierr)
```

Each proc computes its share.

However, note that the size of the vector c in each proc is n_per_proc unlike the reduction case where it was n. Similar argument is valid for the matrix A

Each proc gets its own send counts and offsets

counts and offsets are merged in arrays using allreduce (all procs get the merged result)

Segments of c are gathered in c_tot

University at Buffalo
The State University of New York

MPI DATATYPES

❑ **COMMONLY USED ELEMENTARY MPI FORTRAN DATATYPES**

MPI TYPE	FORTRAN TYPE
MPI_INTEGER	INTEGER(KIND=4)
MPI_INTEGER8	INTEGER(KIND=8)
MPI_FLOAT	REAL (KIND=4)
MPI_DOUBLE	REAL (KIND=8)
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(LEN=1)

MPI 3.0 Standard http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

OUTLINE

University at Buffalo
The State University of New York

RULE OF THE GAME

T. Rauber and G. Runger, Parallel Programming, Springer 2013

- **AMDAHL'S LAW**

$$\boxed{S_p(n) = \frac{1}{f + \frac{1-f}{p}}}$$

- $S_p(n)$ is the limit in actual speedup of the entire code of a given size n
- p is the speed up of the code that benefits from the increase in resources.
- f is the fraction of the total execution time taken by the sequential part of the code

Take home message – *Overall speedup is limited by the percentage of the sequential part of the code*

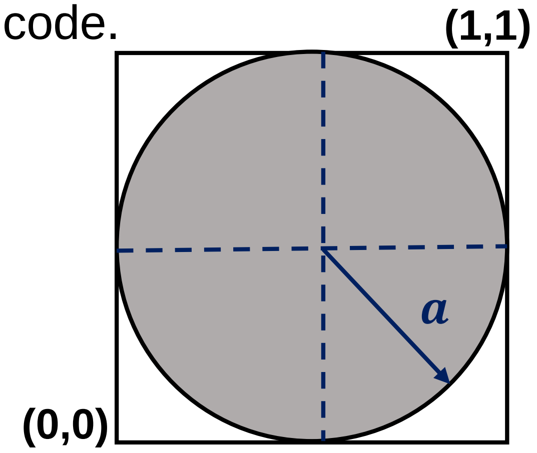
SPEEDUP – EXAMPLE

- **Goal** – Improve scalability/speedup through optimizing code.

- **Task** – Compute the value of π stochastically.

- **Method –**

  - Generate random points (x, y) inside an unit square.
  - Count the fraction of them falling inside an unit circle.
  - $\pi = 4 \times \frac{Points\ inside\ unit\ circle}{Total\ number\ of\ points}$
  - This is referred to as the dartboard algorithm **[Message Passing Interface, B. Barney, LLNL]**



Square area = $4a^2$

Circle area = πa^2

✓ ***We would compute this in parallel using MPI in different ways and compare the speedup and accuracy of the calculation.***

SPEEDUP – EXAMPLE

❑ SEQUENTIAL VERSION

```
c_ser = 0
DO i = 1,n
   !
   CALL RANDOM_NUMBER(r1)
   CALL RANDOM_NUMBER(r2)
   !
   a = (r1*r1 + r2*r2)
   IF(a.LT.(1.0)) c_ser = c_ser + 1
   !
END DO
!
PRINT'("Serially calculated Pi  = ",f16.10)',4.0D0*float(c_ser)/float(n)
```

In-built pseudo random number generator (RNG) with a period of $2^{1024} – 1$ (https://gcc.gnu.org/) Based on XORshift logic.

Output 3.1400

✓ A higher value of the sample size (n) will produce more accurate result

University at Buffalo
The State University of New York

SPEEDUP – EXAMPLE

❑ **PARALLEL VERSION 1**

Same program run on split samples and reduced at the end

```
DO i = 1, n_per_proc
  !
  CALL RANDOM_NUMBER(r1)
  CALL RANDOM_NUMBER(r2)
  !
  a = r1*r1 + r2*r2
  IF(a.LT.(1.0)) c_par = c_par + 1
  !
END DO
!
CALL mpi_barrier (MPI_COMM_WORLD, ierr)
CALL mpi_allreduce (c_par,c_par_tot,1,MPI_INTEGER,MPI_SUM,MPI_COMM_WORLD,ierr)
```

Each proc gets n_per_proc number of samples

all_reduce merges the results

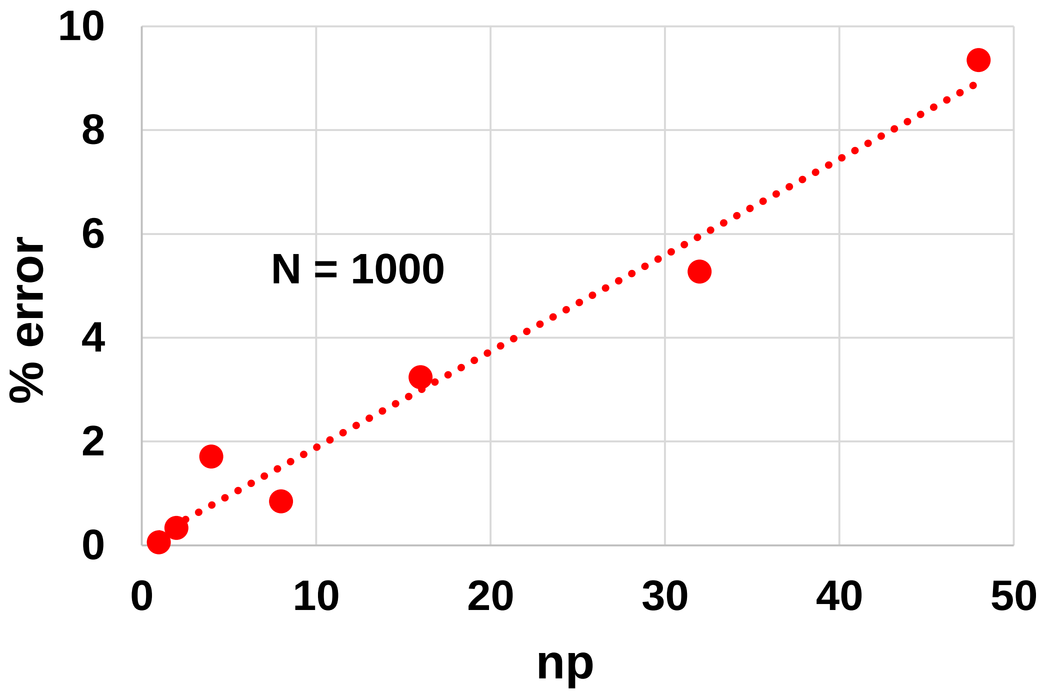
Output **NOT ACCURATE** and
ACCURACY DROPS WITH INCREASING NUMBER OF PROCS !

University at Buffalo
The State University of New York

SPEEDUP – EXAMPLE

❑ **PARALLEL VERSION 1**

- ➢ Aliasing among the RNGs in different procs.
- ➢ Effective sample size drops
- ➢ Poor statistics.
- ➢ For very large number of procs result becomes unreliable.



- ➢ For this particular problem you can get away with any significant error simply choosing a very high N
- ➢ But think about large problem sizes where you cannot afford to have a high N.

University at Buffalo
The State University of New York

SPEEDUP – EXAMPLE

❑ PARALLEL VERSION 2

```
IF (ip.EQ.master) THEN
...........
  DO i = 1, n
     !
     CALL RANDOM_NUMBER(x)
     r1_all(i) = x
     CALL RANDOM_NUMBER(x)
     r2_all(i) = x
     !
  ENDDO
  !
ENDIF
!
CALL mpi_barrier (MPI_COMM_WORLD, ierr)
...........
```

RNG is called by only one proc to generate all the random numbers to avoid aliasing.

Random numbers are scattered across procs

```
CALL mpi_scatterv ( r1_all, all_counts, all_offsets, MPI_DOUBLE, &
                   r1, n_per_proc, MPI_DOUBLE, master, MPI_COMM_WORLD, ierr)
...........
 DO i = 1, n_per_proc
   !
   a = (r1(i)*r1(i) + r2(i)*r2(i))
   IF(a.LT.(1.0)) c_par = c_par + 1
   !
 END DO
 !
 CALL mpi_barrier (MPI_COMM_WORLD, ierr)
 CALL mpi_allreduce (c_par,c_par_tot,1,MPI_INTEGER,MPI_SUM,MPI_COMM_WORLD,ierr)
```

‘area’ calculation in parallel

Output

Good accuracy ☺
Poor (no) speedup ☹

Amdahl’s law in action !

University at Buffalo
The State University of New York

SPEEDUP – EXAMPLE

❑ **PARALLEL VERSION 3**

```
CALL RANDOM_SEED(PUT=arr)
!
DO i = 1, n_per_proc
  !
  CALL RANDOM_NUMBER(r1)
  CALL RANDOM_NUMBER(r2)
  !
  a = r1*r1 + r2*r2
  IF(a.LT.(1.0)) c_par = c_par + 1
  !
END DO
!
CALL mpi_barrier (MPI_COMM_WORLD, ierr)
CALL mpi_allreduce (c_par,c_par_tot,1,MPI_INTEGER,MPI_SUM,MPI_COMM_WORLD,ierr)
```

arr is dependent on the proc_id

Each proc gets its distinct own seed to start the RNG.

This ensures each proc gets a distinct number stream although they run in parallel

Avoids aliasing. Choose the initial seed carefully to minimize correlation among streams

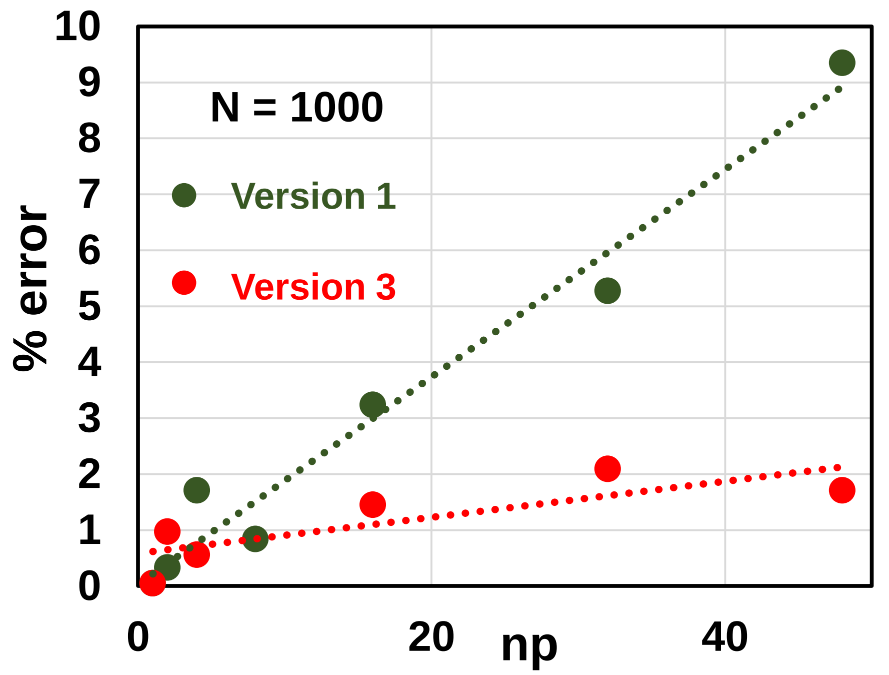
Output

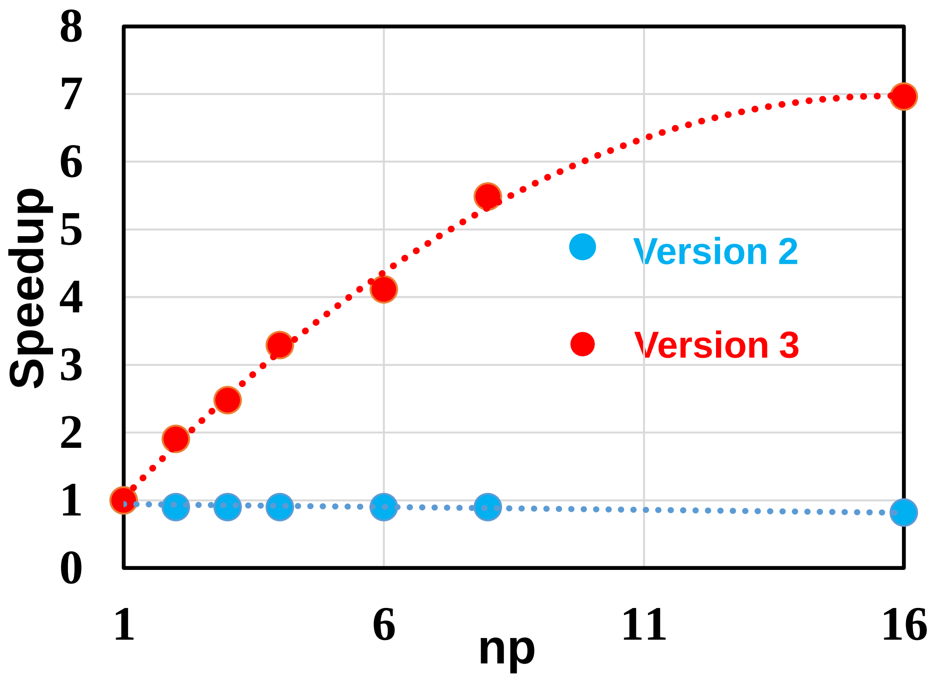
Good accuracy ☺
Good speedup ☺

University at Buffalo
The State University of New York

SPEEDUP – EXAMPLE

❑ PERFORMANCE COMPARISON





- The sample size, N, for speedup comparison is taken to be much higher so that the problem size becomes reasonable to compare the speedup.

- I use INTEL MKL (https://software.intel.com/en-us/intel-mkl) RNGs where separate generators can be used for each proc.

University at Buffalo
The State University of New York

OUTLINE

WHY SHARED MEMORY (SHM)?

- **PROBLEM WITH DISTRIBUTED MEMORY (PURE MPI)**
  - Each proc has its own copy of all the variables.
  - This becomes a problem as array sizes grow and is redundant for SHM procs
  - Total memory requirement exceeds the given node’s memory capacity.
  - Plus, unnecessary MPI calls among the SHM procs.
- **SHARING THE MEMORY AMONG PROCS (HYBRID)**
  - Allocate a shared memory window in each proc that is accessible to all procs
  - Split the variable among procs, one proc can access other’s share.
  - This hybrid approach drastically cuts down the memory requirement
  - Different ways – MPI+OpenMP, MPI+MPI-3, etc.

Let’s see how this can be done using MPI. You need to use MPI-3 for SHM

SHM ALLOCATION

MPI_WIN_ALLOCATE_SHARED (size, disp_unit, info, comm, baseptr, win, ierr)

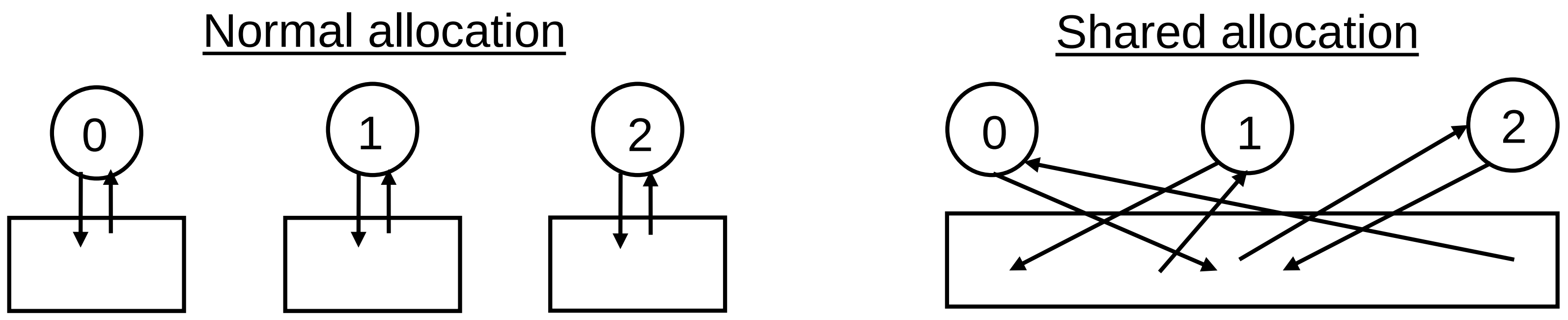
locally allocated size in bytes

local unit size displacement

how memory should be allocated

pointer to locally allocated segment

window object for future reference



- Programmer's responsibility to make sure that all procs under comm can have a shared memory segment. Will see how.
- info = MPI_INFO_NULL allocates contiguous memory; segments are next to each other. Use ALLOC_SHARED_NONCONTIG for non-contiguous memory.

MPI 3.0 Standard http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

University at Buffalo
The State University of New York

SHM ACCESS

$$\textbf{MPI\_WIN\_SHARED\_QUERY (}\underbrace{\textbf{win, rank,}}_{\text{i/p}}\ \underbrace{\textbf{size, disp\_unit, baseptr}}_{\text{o/p}}\textbf{)}$$

- Rank should be the local rank of the target proc
- baseptr points to the local segment in the target proc
- Often the storing and loading of data from remote target procs are referred to as *Remote Memory Access (RMA)*
- <u>FORTRAN only</u>

USE, INTRINSIC :: ISO_C_BINDING

CALL C_F_POINTER(baseptr, a, shape)

C pointer
type (c_ptr)

Fortran pointer

Shape of a

Or use Cray pointers. Some compliers may need separate option for Cray pointers during compilation.

MPI 3.0 Standard **http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf**

University at Buffalo
The State University of New York

SPLITING THE COMMUNICATOR

❑ **MAKING SURE PROCS HAVE A SHM SEGMENT**

➢ First job in any SHM program is to split the communicator group into SHM communicator groups (shmcomm)

➢ This ensures all procs under shmcomm can have a shared memory segment.

➢ If we work on a single node (unusual for big codes), MPI_COMM_WORLD and shmcomm are same.

❑ **EXAMPLE**

```
!
! *********** Split global comm into SMP comms; the third argument 0 says that SMP ranks ********
! ****************************** should be consistent with global ranks **************************
!
CALL MPI_COMM_SPLIT_TYPE (MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, shmcomm, ierr)
CALL mpi_comm_rank (shmcomm, my_rank, ierr)
CALL mpi_comm_size (shmcomm, my_size, ierr)
!
```

Rank within shmcomm
Size of shmcomm

University at Buffalo
The State University of New York

SHM – EXAMPLE

- **ALLOCATE SHARED MEMORY**

We want to store a 2D array $A_{mk \times nk}$ inside a shared window

```
! ********* Allocate shared memory. Change MPI_INFO_NULL to ALLOC_SHARED_NONCONTIG if you *******
! ****************************** want noncontiguous memory segments. *****************************
!
shape1 = (/loc_size, nk/)
size1 = DP_size * shape1(1) * shape1(2)
!
IF (p_id.LT.my_size) PRINT '( " My rank is ", i3, " and local size is ", i3)' , my_rank, loc_size
!
CALL MPI_WIN_ALLOCATE_SHARED (size1, 1 , MPI_INFO_NULL, shmcomm, p, smwin, ierr)
!
CALL C_F_POINTER(p, a, shape1)  ! to assign C pointer target to FORTRAN pointer
!
 ............
 ............
 CALL DD(a)
 CALL MPI_Win_fence (0, smwin, ierr)
 CALL DD(a)
 !
 DO i = 1, loc_size
   !
   DO j = 1, nk
      !
      a(i, j) = FLOAT ( ( p_id + my_rank + 2) * ( i + j ) )
      !
   ENDDO
   !
 ENDDO
 !
 CALL DD(a)
 CALL MPI_Win_fence (0, smwin, ierr)
 CALL DD(a)
```

loc_size is mk split over np
size1 is the actual size in bytes

fence. Different from MPI_BARRIER. Ensures no wrong loading of data during the RMA operation.

Fortran only, DD is just an user defined empty routine to make sure register copies of *a* are written back to memory. See MPI 3.0 Standard (page 638) for more details

SHM – EXAMPLE

❑ **SHARED QUERY**

Next we want to remote access some elements of the array

```
DO iproc = 1, my_size - 1
   !
   IF ( imk.GT.all_offsets(iproc) .AND. imk.LE.all_offsets(iproc+1) ) THEN
      !
      ipool = iproc - 1
      ipos  = imk - all_offsets(iproc)
      PRINT '( (/1x), "proc address and local positions are ", 2i3, (/1x) )', ipool, ipos
      EXIT
      !
   ELSEIF ( imk.GT.all_offsets(my_size)) THEN
      !
      ipool = my_size - 1
      ipos  = imk - all_offsets(ipool)
      PRINT '( (/1x), "proc address and local positions are ", 2i3, (/1x))', ipool, ipos
      EXIT
      !
   ENDIF
   !
ENDDO
!
! ************ Ask for remote proc attributes *****
!
CALL MPI_WIN_SHARED_QUERY (smwin, ipool, size1, disp1, p1, ierr)
rm_shape (2) = shape1(2)
rm_shape (1) = size1 / ( DP_size * rm_shape(2) )
!
CALL C_F_POINTER(p1, a1, rm_shape)
```

First find out which proc the asked element belongs to and what is its position inside that proc

Next access that element through shared query

University at Buffalo
The State University of New York

SHM – EXAMPLE

❑ **OUTPUT**

```
My rank is   0 and local size is  14
My rank is   1 and local size is  13
My rank is   2 and local size is  13
```

Let's take mk=40, nk=40 with mk being split among 3 procs

Suppose one of the proc (say p_id=0) wants to access A (21, :)

```
  32.0
  36.0
  40.0
  44.0
  48.0
  .........
  .........
  176.0
  180.0
  184.0
  188.0
```

Here we double-check the functionality of the code –

Definition of A : $A(i,j) = (p\_id + my\_rank + 2) * (i + j)$

Now imk=21 belongs to p_id=1 and my_rank=1

Local position of imk=21 in p_id=3 should be ipos=(21-14) = 7

Hence the array should be $A(21,j) = 4 * (7 + j)\ \forall\ j \in [1,40]$

This is what we see at the output.

OUTLINE

SUMMARY AND ACKNOWLEDGEMENT

❑ <u>SUMMARY</u>

- Parallel programming becomes more important and powerful in scientific computing every passing year.
- MPI is an extremely useful and efficient standard to build parallel codes.
- Speedup in parallel codes is limited by the sequential percentage of the code.
- MPI shared memory provides excellent features to deal with large memory.

❑ <u>ACKNOWLEDGEMENT</u>

- **Advisor : Dr. Uttam Singisetti**
- **Center for Computational Research (CCR) at UB**
- **Science & Engineering Node Services (SENS) at UB**
- **Electrical Engineering (EE) Department at UB**

THANK YOU ALL!

HAPPY PARALLEL PROGRAMMING!

… Enjoy responsibly ☺

Cheers,
Krishnendu Ghosh
kghosh3@buffalo.edu
https://sites.google.com/site/ghoshportal/