Mid-semester Examination

CS633: Parallel Computing Date: 18th September 2019

Roll number _____

Name _____

- *Total marks is 100. Total time is 2 hours.*
- *There are 2 sections. All questions are compulsory.*
- *You **must** answer all questions on the question paper. Ample space has been provided below each question. You may use extra sheets for rough work only (will not be collected).*
- *Read the questions carefully before answering.*

Section-I

There are 8 questions in this section. Marks are in parenthesis.

I-1. There are 2 commands shown below. Both commands run a code on 4 processes. There are 3 senders and 1 receiver (the last rank) in both cases. The first argument is the number of elements (integers), and the second argument is the number of iterations (for sends/recvs). MPI_Sends and MPI_Recvs are used. Below, the first line is the command, the second line is the output (prints the maximum time). Assume all the hosts are connected to a 1Gb switch. Further, assume that all hosts have 8 logical cores (and 4 physical cores) each.

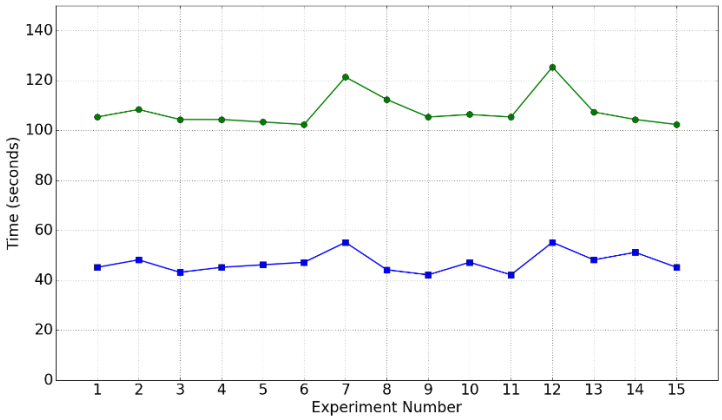
$ mpiexec -ppn 4 -hosts csews4,csews2,csews3,csews1 -np 4 ./sendrecv 10485760 10
Maximum time: A

$ mpiexec -ppn 1 -hosts csews4,csews2,csews3,csews1 -np 4 ./sendrecv 10485760 10
Maximum time: B

(a) Which rank (specify the rank number) is most likely to incur the maximum time and why?
(b) The maximum time incurred for one of the above commands is 1.2s and for the other it is 10.8s. Which time corresponds to which command, i.e. what is A and what is B most likely?
(c) Explain your answer to (b).

(3+3+4)

I-2.

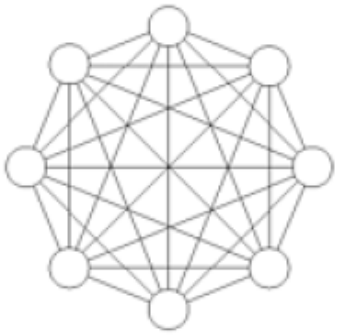


The above figure shows execution times of two different run configurations of the same parallel code in a supercomputer. Each execution was repeated 100 times per day on 15 different days (i.e. a total 1500 executions per configuration). Each data point in the graph is the average of those 100 runs on each day. The x-axis shows the experiment number corresponding to each of those 15 days. The y-axis shows the execution times. Answer the following:

(a) What may be the possible differences between the two run configurations?
(b) Why are the execution times not constant? Explain why are there peaks on certain days.

(5+5)

I-3.



For the network drawn on the left (fully connected graph of 8 nodes)

(a) What is the diameter? Explain your answer.

(b) Derive bisection bandwidth (Assume link bandwidth is 4 Gbps)

(answer specifically for the network drawn)

(3+7)

I-4. Assume a 1D array of N elements. The task is to add the square of each element of the array ‘a’. You may use something like below

```
sq       = a[i] * a[i]      – Line (I-4.1)
sqsum += sq                 – Line (I-4.2)
```

sqsum is the global sum of squares of every element, a[i] is the value of the i^{th} element. Assume that the execution times of the above lines (I-4.1 and I-4.2) are ‘c’ and ‘d’ units of time respectively.

(a) Derive the serial time required for this problem.
(b) What are the different ways in which you can parallelize this problem on ‘p’ processes? Explain your approaches and derive the parallel times and speedups for each of your ways. Assume that you cannot use any MPI collective function, you may use point-to-point communications only.

(3+7)

I-5. Aggregate bandwidth (AB) of a network = total available bandwidth when each node concurrently sends/receives messages to/from its neighbors. AB represents the sum of bandwidths available at all nodes. Derive the aggregate bandwidth for a n X n X n mesh (3D) network. Assume bidirectional links, and assume that each node is able to send/receive at the same time. Bandwidth per link = B.

(10)

I-6.

```
#include <stdio.h>
#include "mpi.h"

int main (int argc, char *argv[])
{
  int rank, size, color, N=3, newrank, newsize;
  MPI_Comm newcomm;

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  color = rank%N;
  MPI_Comm_split (MPI_COMM_WORLD, color, rank, &newcomm);
  MPI_Comm_rank (newcomm, &newrank);

  MPI_Bcast (&color, 1, MPI_INT, 0, newcomm);
  if (newrank == 0)
    printf ("Leader %d (%d) is %d\n", rank, newrank, color);

  if (newrank == size/N - 1)
     printf ("New color of %d (%d) is %d\n", rank, newrank, color);

  MPI_Finalize();
  return 0;
}
```

What is the output (any order) of the code above for the following commands?

(a) mpiexec –np 6 –hosts host1,host2,host3 ./exe
(b) mpiexec –np 7 –hosts host1,host2,host3 ./exe

(5+5)

I-7.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main (int argc, char *argv[])
{

 int numElements = atoi (argv[1]);
 int arr[numElements], myrank, size, len;

 MPI_Status status;
 int count, recvarr[numElements];

 MPI_Init (&argc, &argv);

 MPI_Comm_rank ( MPI_COMM_WORLD, &myrank );
 MPI_Comm_size ( MPI_COMM_WORLD, &size );

 if (myrank%2 == 0)
   MPI_Send (arr, numElements, MPI_BYTE, myrank+1, myrank, MPI_COMM_WORLD);

 else
   MPI_Recv (recvarr, numElements, MPI_BYTE, myrank+1, myrank, MPI_COMM_WORLD, &status);

 MPI_Finalize ();
 return 0;

}
```

Find the MPI-related bug(s) in the above code. Assume that we are executing the code on even number of processes. Explain your answer.

(5)

I-8.

```
#include <stdio.h>
#include "mpi.h"

int main (int argc, char *argv[])
{

  int rank, size, color, root=1;
  int sendbuf, recvbuf=9;

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  sendbuf = rank;
  MPI_Reduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_MAX, root, MPI_COMM_WORLD;
  color = recvbuf;
  recvbuf = rank;
  printf ("%d: color=%d recvbuf=%d\n", rank, color, recvbuf);

  MPI_Finalize();
  return 0;

}
```

What is the output (any order) of the above code for the command
mpiexec –np 5 ./exe

(5)

Section-II

Encircle the correct option(s). There are 15 questions in this section. Every question has 2 marks.

II-1. Parallel programmers use the MPI library for parallel communications because

- The library hides many communication complexities from the programmer
- The library sets up all the communication channels required for messaging
- The programmer can rely on the library to auto-parallelize a sequential code
- It helps with domain decomposition and process assignment

II-2. Consider the following C code snippet, when run on 2 processes. Eager limit is 128 KB.

```
if (rank == 0) {
   int a = 10, b = 100;
   MPI_Send (&a, 1, MPI_INT, 1, 1, MPI_COMM_WORLD); // Send 1
   MPI_Send (&b, 1, MPI_INT, 1, 2, MPI_COMM_WORLD); // Send 2
}

if (rank == 1) {
   MPI_Recv (&b, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status1); // Recv 1
   MPI_Recv (&a, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status2); // Recv 2
}
```

- Send 1 will match Recv1, Send2 will match Recv2
- Send 1 may match Recv2, Send2 may match Recv1
- In rank 1, a = 10, b = 100 after the two receives
- In rank 1, a = 100, b = 10 after the two receives

II-3. MPI_Bcast of 40 bytes on 4 processes from rank 0 can be implemented as

- MPI_Scatter of 10 bytes to each process from rank 0, followed by MPI_Allgather
- MPI_Bcast of first 20 bytes, followed by MPI_Bcast of next 20 bytes
- MPI_Scatter from rank 0, followed by MPI_Reduce
- MPI_Send from rank 0 to ranks 1, 2, 3 and MPI_Recv at ranks 1, 2, 3

II-4. Assume sizeof(double) = 8 bytes, sizeof(int) = 4 bytes.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main (int argc, char *argv[]) {

 int N = 100, myrank, count;
 double data[N];
 int recvdata[2*N];
 MPI_Status status;

 MPI_Init (&argc, &argv);
 MPI_Comm_rank (MPI_COMM_WORLD, &myrank);

 if (myrank == 0)        /* code for process 0 */
    MPI_Send (data, N, MPI_DOUBLE, 1, 99, MPI_COMM_WORLD);
 else if (myrank == 1)  /* code for process 1 */
{
   MPI_Recv (recvdata, 2*N, MPI_INT, 0, 99, MPI_COMM_WORLD, &status);
   MPI_Get_count (&status, MPI_INT, &count );
   printf ("%d\n", count);
}

MPI_Finalize ();
return 0;

}
```

Output for mpiexec –np 2 ./exe

- 8
- 100
- Will not run as the number of bytes received is more than what is allocated at the receiver
- 200
- Datatype mismatch error
- None of the above

II-5. The number of processes spawned as a result of the command “mpirun –np 4 –hosts host1:2,host2:2,host3:2 ./exe” is

- 6
- 4
- Either 4 or 6
- This command will give “Number of processes mismatch” error

II-6. A unique number is stored in every process. Every process wants to know its relative positioning with respect to others based on this unique number (lower the number, lower the order number). We can compute this using

- MPI_Gather followed by sorting, followed by MPI_Bcast
- MPI_Gather followed by sorting, followed by MPI_Scatter
- MPI_Allgather followed by sorting, followed by MPI_Reduce
- MPI_Allgather followed by sorting

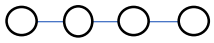
II-7. With respect to the binomial algorithm for MPI_Gather:

- Time taken by this algorithm is O(L log P + N*(P-1)/(P*B))
- A tree-like network topology may be more suitable for this algorithm
- A ring network topology may be better than tree-like topology for this algorithm
- The time taken is independent of the network topology because this algorithm is robust

II-8. Assume host1 and host2 have 2 cores each (core1 and core2). The process to core mapping for the command “mpiexec –hosts host1,host2 –np 4 ./exe” (using MPICH) may be

- Rank 0 on host1, core1, Rank 2 on host1, core2, Rank 3 on host2, core1, Rank 4 on host2, core2
- The hosts are randomly decided by the Hydra process manager at the time of spawning the processes
- Rank 0 and 1 on host1 and Rank 2 and 3 on host2
- Rank 0 and 2 on host1 and Rank 1 and 3 on host2

II-9. The maximum number of hops for the ring algorithm for MPI_Allgather on 4 ranks placed on 4 consecutive nodes on the linear array network shown on the right is



- 3
- 2
- 1
- Either 1 or 3, depends on the execution of the algorithm

II-10. With respect to the parallelization steps - decomposition, assignment and orchestration:

- They are useful to determine how one can expose more concurrency in the problem
- Used to determine which sub-domain will be executed on which process
- Which process will run on which processor
- How many processes will be used to solve a problem

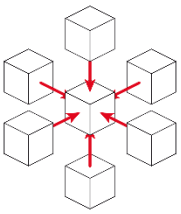
II-11. With respect to communications in a parallel code:

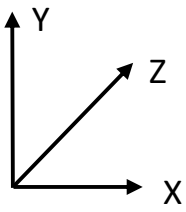
- Communications are always necessary in a parallel code
- Communication always takes more time than computation in a parallel code
- Computation is done on a few cores, and few cores are set aside for communications
- The data for communication resides in a special region of memory, so that data may be accessed faster

II-12. With respect to network topology:

- Communications are greatly impacted by the topology
- A parallel algorithm must always be designed specific to a topology
- A parallel algorithm may be modified to cater to a specific network topology
- If the network is high-bandwidth (e.g. Fibre optics), network topology does not play any role in the parallel code performance

II-13. A 3D grid (data domain) of size nx X ny X nz is decomposed on P processes. Assume a 7-point 3D stencil (shown in the figure) code. Assume XYZ order in case of 3D decomposition and XY order in case of 2D decomposition. Which of the following is/are true?





- A cubic grid per process may give lower communication times in case of 3D decomposition
- Total communication volume per process depends on the decomposition strategy
- In general, 2D decomposition is better than 3D decomposition because of better mapping
- In order to do 3D decomposition for this, we need a perfect cube number of processes

II-14. Which of the following is/are true?

- Non-blocking broadcasts can be replaced with non-blocking sends and receives
- MPI_Send/Recv must be carefully used to ensure a safe code
- Programmers must ensure that tags for MPI_Reduce must always match
- MPI_Bcast will take less time if the internal implementation of broadcast algorithm uses MPI_Ssend instead of MPI_Send

II-15. Parallel programming is hard, in general because one

- has to worry about how to parallelize
- must know the problem domain well in order to better parallelize
- must take care of many factors such as -- minimize idling, understand how well the communications map to the network, decide an optimal number of processes and maximize efficiency
- must know the communication pattern well in order to better decompose the domain

MPI FUNCTIONS (PROTOTYPES)

1) int **MPI_Comm_rank**(MPI_Comm comm, int *rank)
2) int **MPI_Comm_size**(MPI_Comm comm, int *size)
3) int **MPI_Send**(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
4) int **MPI_Recv**(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
5) int **MPI_Barrier**(MPI_Comm comm)
6) int **MPI_Bcast**(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
7) int **MPI_Gather**(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
8) int **MPI_Scatter**(const void *sendbuf, int sendcount, MPI_Datatype sendtype,void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
9) int **MPI_Allgather**(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype,MPI_Comm comm)
10) int **MPI_Reduce**(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
11) int **MPI_Allreduce**(const void *sendbuf, void *recvbuf, int count,MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
12) int **MPI_Alltoall**(const void *sendbuf, int sendcount, MPI_Datatype sendtype,void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
13) int **MPI_Gatherv**(const void *sendbuf, int sendcount, MPI_Datatype sendtype,void *recvbuf, const int *recvcounts, const int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)
14) int **MPI_Scatterv**(const void *sendbuf, const int *sendcounts, const int *displs, MPI_Datatype sendtype, void *recvbuf, int recvcount,MPI_Datatype recvtype, int root, MPI_Comm comm)
15) int **MPI_Allgatherv**(const void *sendbuf, int sendcount, MPI_Datatype sendtype,void *recvbuf, const int *recvcounts, const int *displs,MPI_Datatype recvtype, MPI_Comm comm)
16) int **MPI_Alltoallv**(const void *sendbuf, const int *sendcounts, const int *sdispls, MPI_Datatype sendtype, void *recvbuf, const int *recvcounts, const int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)
17) int **MPI_Isend**(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
18) int **MPI_Irecv**(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)
19) int **MPI_Ibcast**(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm, MPI_Request *request)
20) int **MPI_Igather**(const void *sendbuf, int sendcount, MPI_Datatype sendtype,void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)
21) int **MPI_Iscatter**(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)
22) int **MPI_Iallgather**(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
23) int **MPI_Ireduce**(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm, MPI_Request *request)
24) int **MPI_Iallreduce**(const void *sendbuf, void *recvbuf, int count,MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Request *request)
25) int **MPI_Ialltoall**(const void *sendbuf, int sendcount, MPI_Datatype sendtype,void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)
26) int **MPI_Igatherv**(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)
27) int **MPI_Iscatterv**(const void *sendbuf, const int sendcounts[], const int displs[],MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)
28) int **MPI_Iallgatherv**(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int displs[], MPI_Datatype recvtype,MPI_Comm comm, MPI_Request *request)
29) int **MPI_Ialltoallv**(const void *sendbuf, const int sendcounts[], const int sdispls[], MPI_Datatype sendtype, void *recvbuf, const int recvcounts[], const int rdispls[], PI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)