3

MPI Parallel Programming

Lecture 1

Dr Abid Mehmood
(abid.mehmood@dsu.edu)

Dakota State University

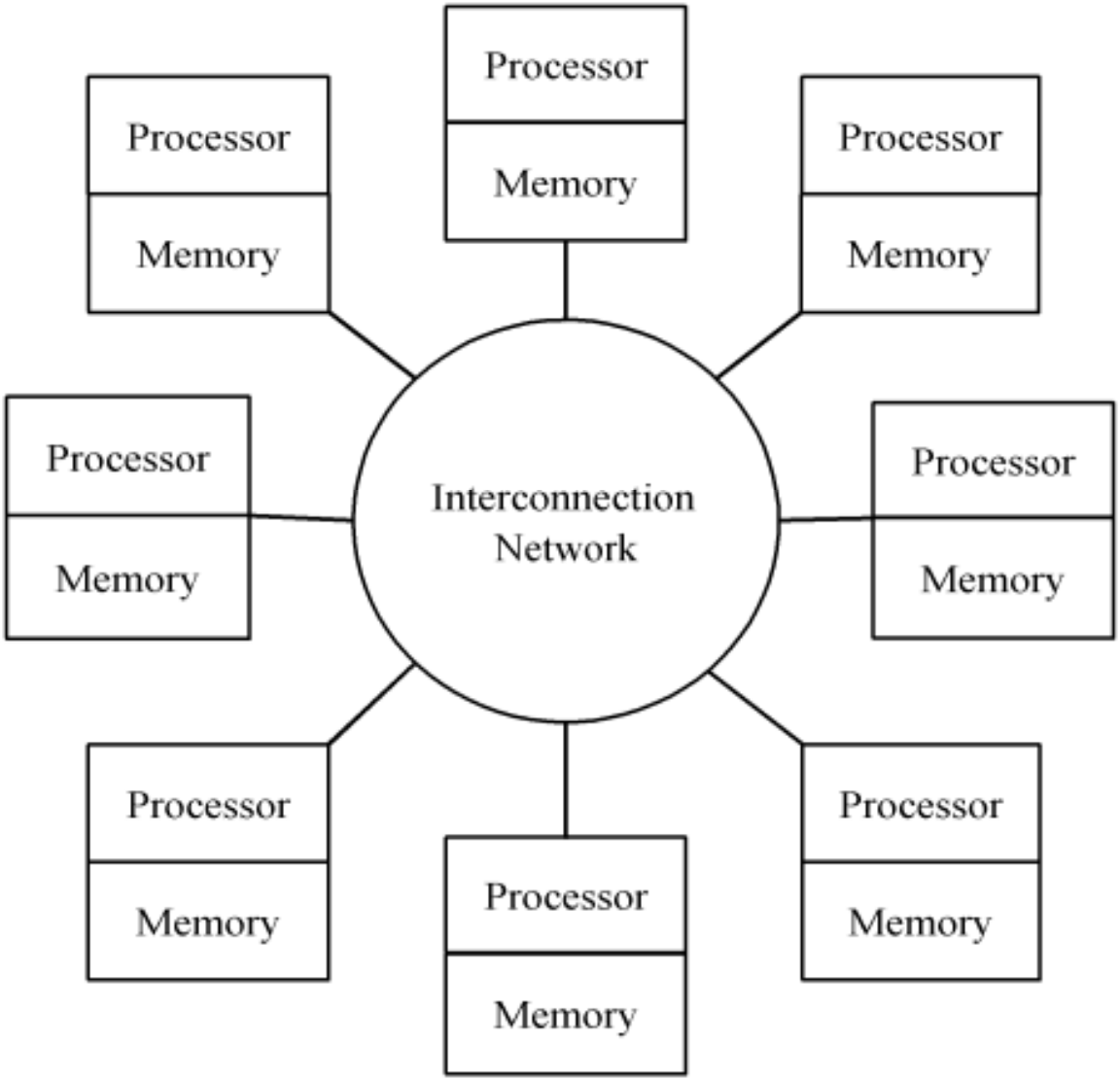
Learning Objectives

- Understanding how MPI programs execute
- Familiarity with fundamental MPI functions

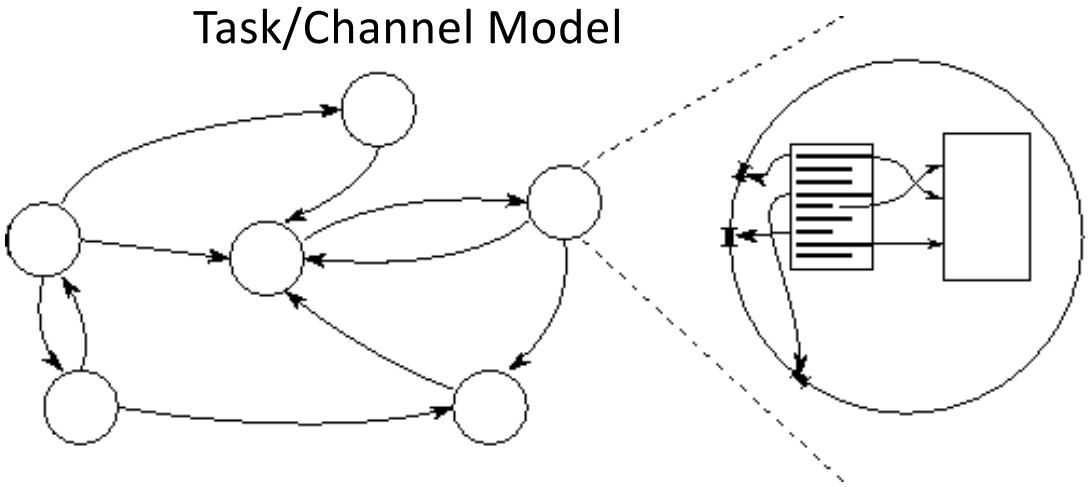
Outline

- Message-passing model
- Message Passing Interface (MPI)
- Coding MPI programs
- Compiling and running MPI programs
- Benchmarking MPI programs

Message-passing Model



Task/Channel vs. Message-passing



- Task/channel
  - Task
  - Explicit channels

- Message-passing
  - Process
  - Any-to-any communication

Process Characteristics in Message-Passing Model

- Number of processes is specified at launch time
- Remains constant throughout execution of program
- All processes execute the same program
- Each process has unique ID number
- Processes alternate between performing computations and communicating with each other.

Advantages of Message-Passing Model

- Gives programmer ability to manage the memory hierarchy
  - Maximize local computations while minimizing communications
- Portability to many architectures
- Easier to create a deterministic program
- Simplifies debugging

The Message Passing Interface

- Late 1980s: vendors had unique libraries
- 1989: Parallel Virtual Machine (PVM) developed at Oak Ridge National Lab
- 1992: Work on MPI standard began
- 1994: Version 1.0
- 1997: Version 2.0
- 2012: Version 3.0
- 2021: Version 4.0
- 2023: Version 4.1
- https://www.mpi-forum.org/

- The Open MPI Project is an open source MPI implementation developed and maintained by a consortium of academic, research, and industry partners.
- https://www.open-mpi.org/

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

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);

  printf("Process %d on %s out of %d\n", rank, processor_name, numprocs);

  MPI_Finalize();
}
```

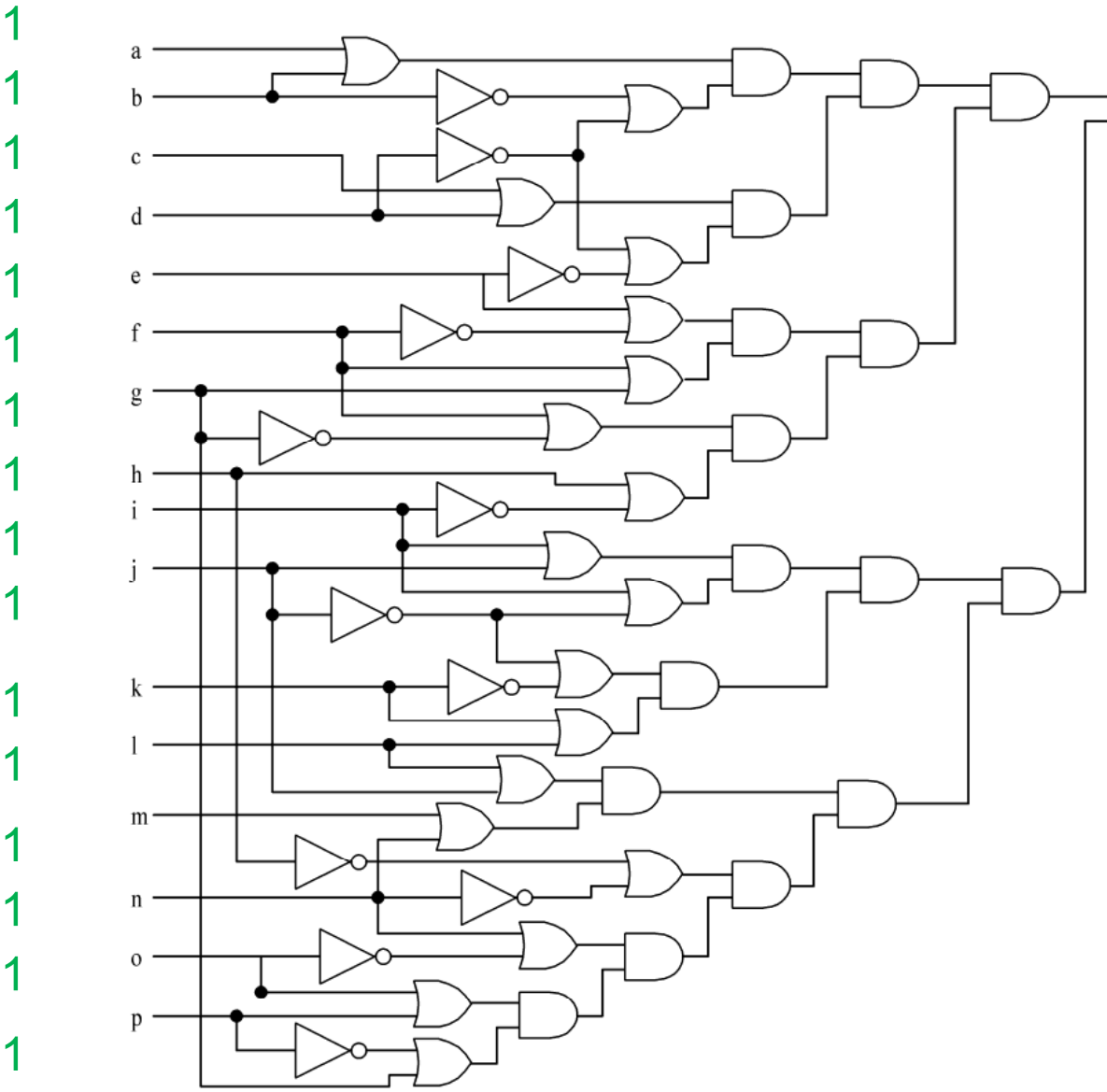
```
mpiuser@Washington:~/nfs/CSC718/openMPI/hello$ mpicc hello.c -o hello_mpi
mpiuser@Washington:~/nfs/CSC718/openMPI/hello$ mpirun -np 4 -machinefile machine
file.dsu ./hello_mpi
Process 0 on Lincoln out of 4
Process 2 on Roosevelt out of 4
Process 1 on Washington out of 4
Process 3 on Jefferson out of 4
mpiuser@Washington:~/nfs/CSC718/openMPI/hello$
```

Circuit Satisfiability Problem (CSAT) for MPI

- Demonstrates key MPI principles like process communication, parallel task distribution, and performance scaling.

- A well-known NP-complete problem
  - No known algorithms to solve in polynomial time
- You determine if there is a combination of inputs that causes the output of a Boolean circuit to be true (‘1’).
- Requires testing numerous input combinations.
- Exhaustive search for solutions is highly parallelizable:
  - Each possible input combination can be tested independently.
  - The problem fits naturally with MPI’s message-passing paradigm.

Circuit Satisfiability

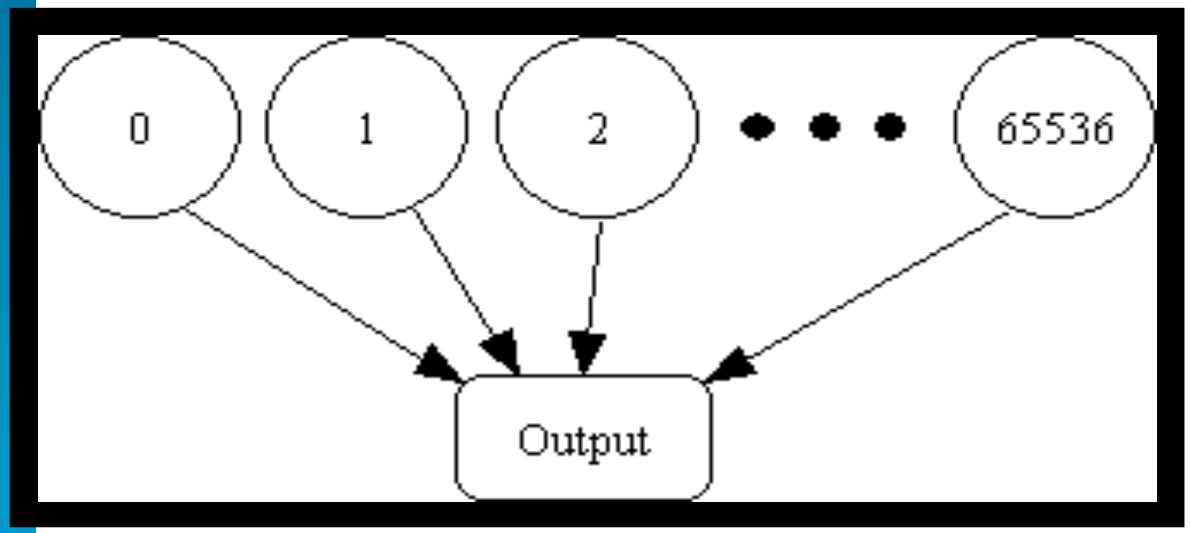


Not satisfied

To determine if some combination of inputs cause the output of the circuit to be 1

16 inputs ⇒ 65,536 combinations to test

Partitioning: Functional Decomposition



Embarrassingly parallel: No channels between tasks

Embarrassingly Parallel: Tasks that can be easily separated into independent sub-tasks.

Minimal communication between tasks.

Ideal for distributed systems and parallel programming.

Examples: Image processing, data encryption, large-scale simulations.

Agglomeration and Mapping in Parallel Algorithms

Agglomeration

- Grouping smaller tasks into larger units to minimize communication overhead and balance the workload.
- Properties of parallel algorithm
  - Fixed number of tasks (determined upfront)
  - No communications between tasks (no data exchange)
  - Time needed per task is variable (some may take longer)

Task mapping strategy

  - Decides how to distribute (map) tasks to processors
  - Use cyclic mapping to assign tasks to processors in a round-robin fashion to balance the workload.

Cyclic (Interleaved) Allocation

- Assume p processes and N tasks
- Each process gets every p-th piece of work
  - process *i* gets tasks numbered: *i*, *i* + *p*, *i* + *2p*, *i* + *3p*, … until all tasks are assigned
- Example: 5 processes and 12 pieces of work
  - P0: 0, 5, 10
  - P1: 1, 6, 11
  - P2: 2, 7
  - P3: 3, 8
  - P4: 4, 9

When the number of tasks is not perfectly divisible by the number of processes, some processes will end up with more tasks than others.

Exercise

- Assume n pieces of work, p processes, and cyclic allocation
- What is the most pieces of work any process has?
- What is the least pieces of work any process has?
- How many processes have the most pieces of work?

CSAT: Summary of Program Design

- Program will consider all 65,536 combinations of 16 boolean inputs
- Combinations allocated in cyclic fashion to processes
- Each process examines each of its combinations
- If it finds a satisfiable combination, it will output it

The example encapsulates key MPI ideas:

- ❑ *Initialization (MPI_Init)*
- ❑ *Rank-based task division*
- ❑ *Independent local computation*
- ❑ *Optional result communication*
- ❑ *Finalization (MPI_Finalize)*

Key MPI Terminology Summary

- **Process Rank:** Unique identifier for each process.
- Communicator: Group of processes involved in message passing.
- **Cyclic Allocation:** Distributes work evenly across processes.
- **Collective Communication:** Communication involving multiple processes.
- **Synchronization:** Ensuring processes are in sync.

DAKOTA STATE
THE BEACOM COLLEGE of
COMPUTER & CYBER SCIENCES

3

MPI Parallel Programming

Lecture 2

Dr Abid Mehmood
(abid.mehmood@dsu.edu)

Dakota State University

Objectives

- Parallelizing a task across multiple processes
- Enabling communication between processes
- Timing and performance in parallel programs

circuit1.c

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

int main (int argc, char *argv[])
{
    int i;
    int id;
    int p;
    void check_circuit (int, int);
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);
    for (i = id; i < 65536; i += p)
        check_circuit (id, i);

    printf ("Process %d is done\n", id);
    fflush (stdout);
    MPI_Finalize();
    return 0;
}
```

circuit1.c

```c
/* Return 1 if 'i'th bit of 'n' is 1; 0 otherwise */
#define EXTRACT_BIT(n,i) ((n&(1<<i))?1:0)

void check_circuit (int id, int z) {
   int v[16];        /* Each element is a bit of z */
   int i;

   for (i = 0; i < 16; i++) v[i] = EXTRACT_BIT(z,i);

   if ((v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3])
      && (!v[3] || !v[4]) && (v[4] || !v[5])
      && (v[5] || !v[6]) && (v[5] || v[6])
      && (v[6] || !v[15]) && (v[7] || !v[8])
      && (!v[7] || !v[13]) && (v[8] || v[9])
      && (v[8] || !v[9]) && (!v[9] || !v[10])
      && (v[9] || v[11]) && (v[10] || v[11])
      && (v[12] || v[13]) && (v[13] || !v[14])
      && (v[14] || v[15])) {
      //printf("satified for number: %d\n", z);
      printf ("%d) %d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d\n", id,
         v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7],v[8],v[9],
         v[10],v[11],v[12],v[13],v[14],v[15]);
      fflush (stdout);
   }
}
```

Local Variables

```
int main (int argc, char *argv[]) {
    int i;
    int id; /* Process rank */
    int p;  /* Number of processes */
    void check_circuit (int, int);
```

- Include **`argc`** and **`argv`**: they are needed to initialize MPI
- One copy of every variable for each process running this program

Initialize MPI

```
MPI_Init (&argc, &argv);
```

- First MPI function called by each process
- Not necessarily first executable statement
- Allows system to do any necessary setup

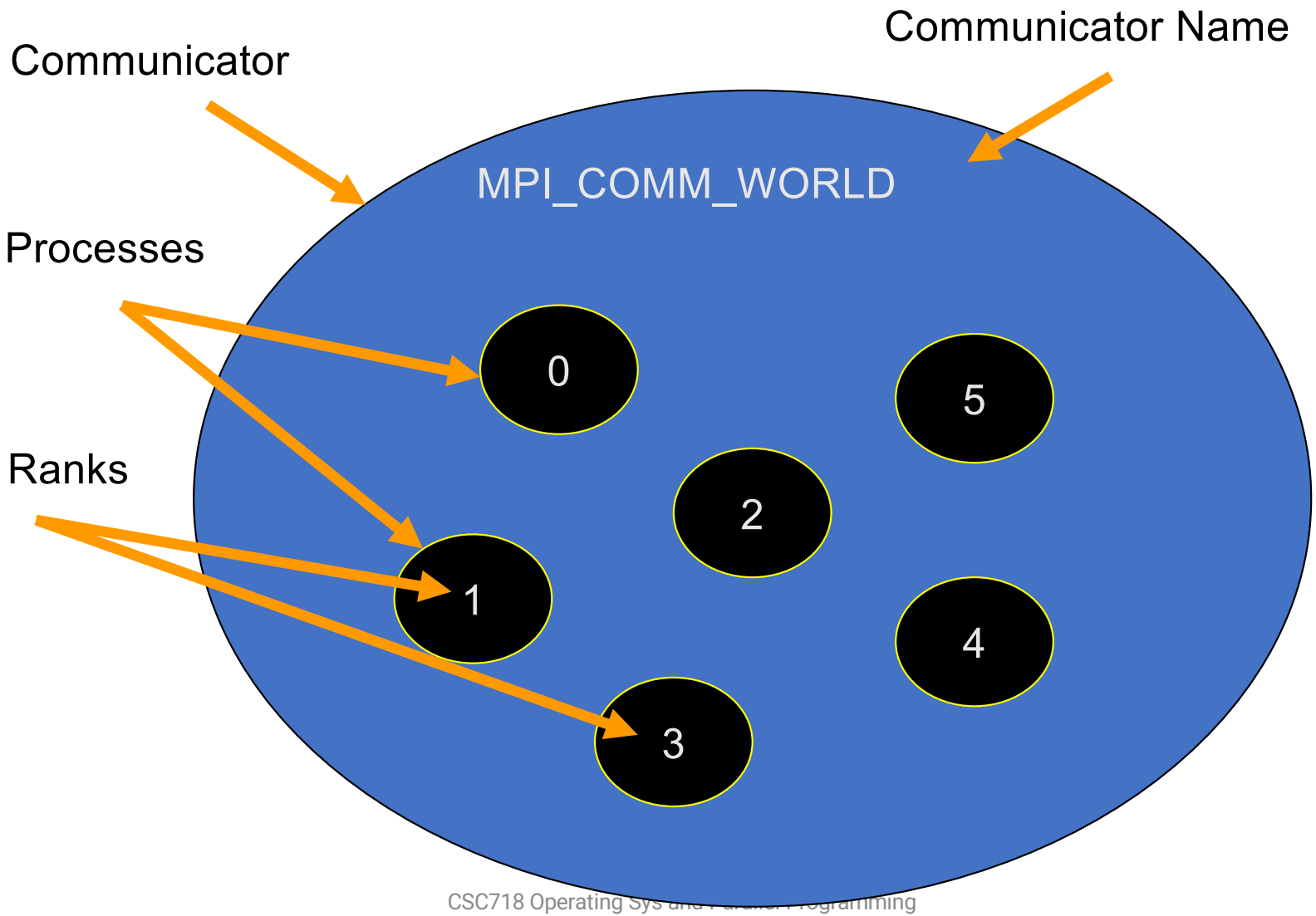
Types of Communication in Parallel Programming

- Point-to-point communication: Direct communication between two processes.
  - MPI functions: MPI_Send, MPI_Recv.
- Collective communication: Involves groups of processes.
  - Examples: Broadcast, reduce, scatter, and gather.
  - MPI functions: MPI_Bcast, MPI_Reduce, MPI_Scatter, MPI_Gather.
- Synchronization: Ensuring processes are aligned before continuing.
  - MPI function: MPI_Barrier.

Communicator and MPI_COMM_WORLD

- Communicator: Object that provides message-passing environment for processes
- Every communicator in MPI has a name.
- MPI_COMM_WORLD
  - Default communicator
  - Includes all processes
- Each process knows its rank within a communicator
  - Could have different ranks in different communicators
- Possible to create new communicators
  - Will cover later in course

Communicator and MPI_COMM_WORLD



Determine Number of Processes

```
MPI_Comm_size (MPI_COMM_WORLD, &p);
```

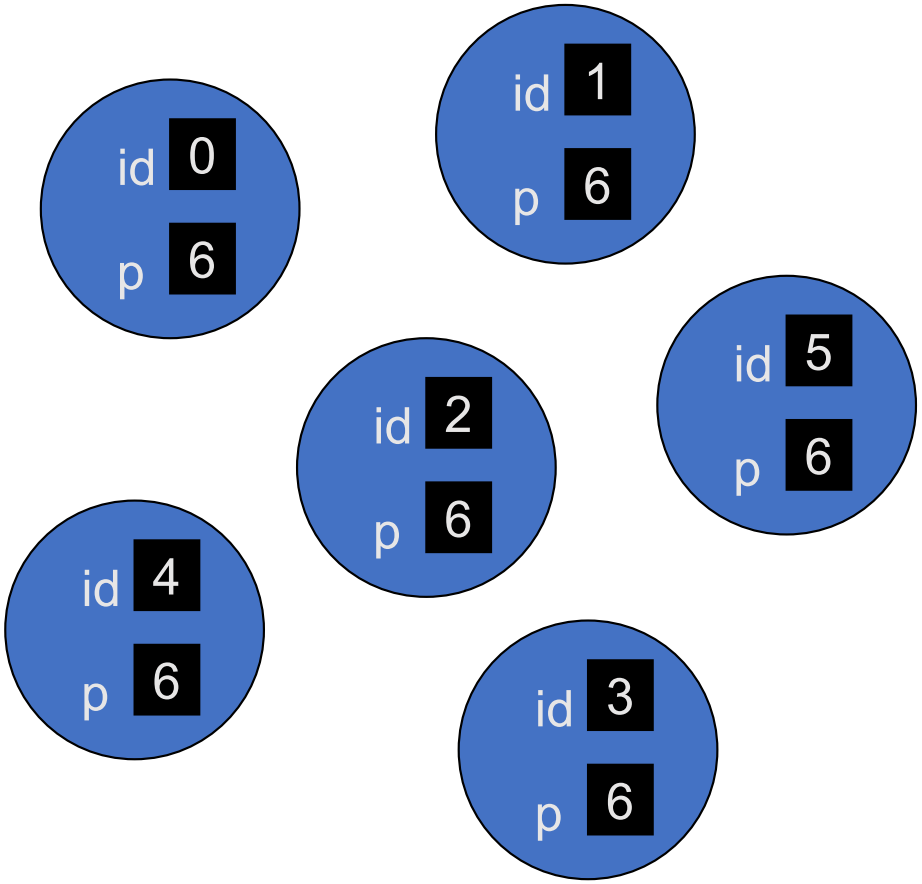
- First argument is communicator
- Number of processes returned through second argument

Determine Process Rank

```
MPI_Comm_rank (MPI_COMM_WORLD, &id);
```

- First argument is communicator
- Process rank (in range 0, 1, …, p-1) returned through second argument

Replication of Automatic Variables



Each process has its own copy of variables declared within a function (automatic variables).

Automatic variables are not shared among processes.

Recall that MPI programs are for distributed memory systems, not shared memory.

What about External Variables?

```
int total;

int main (int argc, char *argv[]) {
    int i;
    int id;
    int p;
    …
```

Global variable

- Where is variable `total` stored?

What about External Variables?

```
int total;
```

Global variable

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

…

- Global variables are accessible to all functions within the same source file (or other files if declared as extern)
- Each process has its own copy of global variables (not shared).
- To share, MPI communication function will be used (e.g. MPI_Send, MPI_Bcast)

Cyclic Allocation of Work

```
for (i = id; i < 65536; i += p)
    check_circuit (id, i);
```

- The loop determines **which combinations** each process will check.
- Parallelism is outside the function **`check_circuit`**
- It can be an ordinary, sequential function.
- The MPI logic only handles which data each process should process, not how the computation itself is done.

Shutting Down MPI

```
MPI_Finalize();
```

- Call after all other MPI library calls
- Allows system to free up MPI resources

Compiling MPI Programs

```
mpicc -O -o foo foo.c
```

- **mpicc:** script to compile and link C+MPI programs
- Flags: same meaning as C compiler
  - `-O` —— optimize
  - `-o <file>` —— where to put executable

Running MPI Programs

- **`mpirun -np <p> <exec> <arg1>`** …
  - **`-np <p>`** — number of processes
  - **`<exec>`** — executable
  - **`<arg1>`** … — command-line arguments

Specifying Host Processors

- `mpirun –np 2 –machinefile machinefile.dsu hello`
- (default) File .mpi-machines in home directory lists host processors in order of their use
- Example machinefile contents
  - band01.cs.ppu.edu
  - band02.cs.ppu.edu
  - band03.cs.ppu.edu
  - band04.cs.ppu.edu

If -np exceeds the number of listed machines, MPI cycles through the list again (depending on configuration).

Enabling Remote Logins

- MPI needs to be able to initiate processes on other processors without supplying a password
- Each processor in group must list all other processors in its .rhosts file [hostname username] e.g.,
  `band01.cs.ppu.edu student`
  - `band02.cs.ppu.edu student`
  - `band03.cs.ppu.edu student`
  - `band04.cs.ppu.edu student`

Deciphering Output

- Output order only partially reflects order of output events inside parallel computer
- If process A prints two messages, first message will appear before second (guaranteed intra-process ordering)
- If process A calls **`printf`** before process B, there is no guarantee process A's message will appear before process B's message

Impl Tip: If ordered output is required for readability (e.g., by process rank):

- Use MPI_Barrier() before printing, or
- Have only the root process print results after gathering them.

```
(base) abid@MacBookAir circuit % mpirun -np 1 circuit1
0) 1010111110011001
0) 0110111110011001
0) 1110111110011001
0) 1010111111011001
0) 0110111111011001
0) 1110111111011001
0) 1010111110111001
0) 0110111110111001
0) 1110111110111001
Process 0 is done
```

```
(base) abid@MacBookAir circuit % mpirun -np 2 circuit1
0) 0110111110011001
0) 0110111111011001
0) 0110111110111001
1) 1010111110011001
1) 1110111110011001
1) 1010111111011001
1) 1110111111011001
1) 1010111110111001
1) 1110111110111001
Process 0 is done
Process 1 is done
```

```
(base) abid@MacBookAir circuit % mpirun -np 3 circuit1
0) 0110111110011001
0) 1110111111011001
0) 1010111110111001
2) 1010111110011001
2) 0110111111011001
2) 1110111110111001
1) 1110111110011001
1) 1010111111011001
1) 0110111110111001
Process 0 is done
Process 1 is done
Process 2 is done
```

```
(base) abid@MacBookAir circuit % mpirun -np 4 circuit1
3) 1110111110011001
3) 1110111111011001
3) 1110111110111001
1) 1010111110011001
1) 1010111111011001
1) 1010111110111001
2) 0110111110011001
2) 0110111111011001
2) 0110111110111001
Process 0 is done
Process 3 is done
Process 2 is done
Process 1 is done
```

Enhancing the Program

- We want to find total number of solutions
- Incorporate sum-reduction into program
- Reduction is a collective communication

Modifications

- Modify function **`check_circuit`**
  - Return 1 if circuit satisfiable with input combination
  - Return 0 otherwise
- Each process keeps local count of satisfiable circuits it has found
- Perform reduction after **`for`** loop

New Declarations and Code

```
int count;  /* Local sum */
int global_count; /* Global sum */
int check_circuit (int, int);

count = 0;
for (i = id; i < 65536; i += p)
    count += check_circuit (id, i);
```

Definition of MPI_Reduce()

```
int MPI_Reduce(
   void *operand,   /* address of send buffer */
   void *result,    /* address of receive buffer (only at root) */
   int count,       /* number of elements */
   MPI_Datatype type, /* data type of elements */
   MPI_Op operator, /* reduction operation */
   int root,        /* rank of process receiving result */
   MPI_Comm comm    /* communicator */
);
```

```
double local_sum, global_sum;
MPI_Reduce(&local_sum, &global_sum, 1, MPI_DOUBLE,
MPI_SUM, 0, MPI_COMM_WORLD);
```

All processes send their local double values, and process 0 receives the combined sum.

MPI_Datatype Options

- MPI_CHAR
- MPI_DOUBLE
- MPI_FLOAT
- MPI_INT
- MPI_LONG
- MPI_LONG_DOUBLE
- MPI_SHORT
- MPI_UNSIGNED_CHAR
- MPI_UNSIGNED
- MPI_UNSIGNED_LONG
- MPI_UNSIGNED_SHORT

MPI_Op Options

- MPI_MAX maximum
- MPI_MIN minimum
- MPI_SUM sum
- MPI_PROD product
- MPI_LAND logical and
- MPI_BAND bit-wise and
- MPI_LOR logical or
- MPI_BOR bit-wise or
- MPI_LXOR logical xor
- MPI_BXOR bit-wise xor
- MPI_MAXLOC max value and location (rank)
- MPI_MINLOC min value and location

Our Call to MPI_Reduce()

```
MPI_Reduce (&count,
            &global_count,
            1,
            MPI_INT,
            MPI_SUM,
            0,
            MPI_COMM_WORLD);
```

Process 0 gets the result

```
if (!id) printf ("There are %d different solutions\n",
   global_count);
```

circuit2.c

```
4   int main(int argc, char *argv[])
5   {
6       int i;
7       int id;
8       int p;
9
10      int count;  /* Local sum */
11      int global_count; /* Global sum */
12      int check_circuit (int, int);
13
14      MPI_Init (&argc, &argv);
15      MPI_Comm_rank (MPI_COMM_WORLD, &id);
16      MPI_Comm_size (MPI_COMM_WORLD, &p);
17
18      count = 0;
19      for (i = id; i < 65536; i += p)
20          count += check_circuit (id, i);
```

circuit2.c

```
21      /* aggregate data from all processes in a communicator and
22      reduce it to a single result*/
23      MPI_Reduce (&count, //send buffer (local data)
24              &global_count, // receive buffer (global data)
25              1, //how many elements per process are to be reduced (1: count)
26              MPI_INT, // type of element to be reduced
27              MPI_SUM, // op to be performed on data
28              0, // in which process data will be stored
29              MPI_COMM_WORLD);
30
31      printf ("Process %d is done\n", id);
32      fflush (stdout);
33      MPI_Finalize();
34
35      if (!id) printf ("There are %d different solutions\n",
36         global_count);
37
38      return 0;
39  }
40
41  /* Return 1 if 'i'th bit of 'n' is 1; 0 otherwise */
42  #define EXTRACT_BIT(n,i) ((n&(1<<i))?1:0)
```

```c
int check_circuit (int id, int z) {
   int v[16];        /* Each element is a bit of z */
   int i;

   for (i = 0; i < 16; i++) v[i] = EXTRACT_BIT(z,i);

   if ((v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3])
      && (!v[3] || !v[4]) && (v[4] || !v[5])
      && (v[5] || !v[6]) && (v[5] || v[6])
      && (v[6] || !v[15]) && (v[7] || !v[8])
      && (!v[7] || !v[13]) && (v[8] || v[9])
      && (v[8] || !v[9]) && (!v[9] || !v[10])
      && (v[9] || v[11]) && (v[10] || v[11])
      && (v[12] || v[13]) && (v[13] || !v[14])
      && (v[14] || v[15])) {
      printf ("%d) %d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d\n", id,
         v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7],v[8],v[9],
         v[10],v[11],v[12],v[13],v[14],v[15]);
      fflush (stdout);
      return 1;
   }

   return 0;
}
```

Other MPI Collective Communication Functions

- **MPI_Gather:** Collects values from all processes to a root process.
- **MPI_Scatter:** Distributes chunks of data from the root process to all others.
- **MPI_Bcast:** Broadcasts data from the root process to all other processes.

circuit2.c

```
(base) abid@MacBookAir circuit % mpirun -np 3 circuit2
2) 1010111110011001
2) 0110111111011001
2) 1110111110111001
0) 0110111110011001
0) 1110111111011001
0) 1010111110111001
1) 1110111110011001
1) 1010111111011001
1) 0110111110111001
Process 1 is done
Process 2 is done
Process 0 is done
There are 9 different solutions
```

Benchmarking the Program

- **MPI_Barrier** — process synchronization, blocks until all processes have reached this routine.
- **MPI_Wtick** — timer resolution, returns the resolution of MPI_Wtime
- **MPI_Wtime** — in seconds, returns an elapsed time on the calling processor.

Benchmarking Code

```
double elapsed_time;
…
MPI_Init (&argc, &argv);
MPI_Barrier (MPI_COMM_WORLD);
elapsed_time = - MPI_Wtime(); //start timer
…
MPI_Reduce (…);
elapsed_time += MPI_Wtime(); //stop timer
```

circuit3.c

```
4   int main(int argc, char *argv[])
5   {
6       int i;
7       int id;
8       int p;
9
10      double elapsed_time;
11      int count;  /* Local sum */
12      int global_count; /* Global sum */
13      int check_circuit (int, int);
14
15      MPI_Init (&argc, &argv);
16      MPI_Barrier (MPI_COMM_WORLD);
17      elapsed_time = - MPI_Wtime();
18
19      MPI_Comm_rank (MPI_COMM_WORLD, &id);
20      MPI_Comm_size (MPI_COMM_WORLD, &p);
21
22      count = 0;
23      for (i = id; i < 65536; i += p)
24          count += check_circuit (id, i);
25
```

circuit3.c

```
26      MPI_Reduce (&count,
27                  &global_count,
28                  1,
29                  MPI_INT,
30                  MPI_SUM,
31                  0,
32                  MPI_COMM_WORLD);
33      elapsed_time += MPI_Wtime();
34
35      // printf ("Process %d is done.\n", id);
36      // fflush (stdout);
37      MPI_Finalize();
38
39      if (!id)
40      {
41          printf ("Elapsed time: %f.\n", elapsed_time);
42          fflush (stdout);
43          printf ("There are %d different solutions\n", global_count);
44      }
45
46      return 0;
47  }
48
49  /* Return 1 if 'i'th bit of 'n' is 1; 0 otherwise */
50  #define EXTRACT_BIT(n,i) ((n&(1<<i))?1:0)
51
```

circuit3.c

```
52  int check_circuit (int id, int z) {
53      int v[16];          /* Each element is a bit of z */
54      int i;
55
56      for (i = 0; i < 16; i++) v[i] = EXTRACT_BIT(z,i);
57
58      if ((v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3])
59          && (!v[3] || !v[4]) && (v[4] || !v[5])
60          && (v[5] || !v[6]) && (v[5] || v[6])
61          && (v[6] || !v[15]) && (v[7] || !v[8])
62          && (!v[7] || !v[13]) && (v[8] || v[9])
63          && (v[8] || !v[9]) && (!v[9] || !v[10])
64          && (v[9] || v[11]) && (v[10] || v[11])
65          && (v[12] || v[13]) && (v[13] || !v[14])
66          && (v[14] || v[15])) {
67          // printf ("%d) %d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d\n", id,
68          //   v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7],v[8],v[9],
69          //   v[10],v[11],v[12],v[13],v[14],v[15]);
70          // fflush (stdout);
71          return 1;
72      }
73
74      return 0;
75  }
```

Benchmarking Results

```
mpiuser@Roosevelt:~/nfs/CSC718/openMPI/circuit$ mpirun -np 4 -machinefile machinefile.dsu ./circuit3
Elapsed time: 0.001894.
There are 9 different solutions
mpiuser@Roosevelt:~/nfs/CSC718/openMPI/circuit$ mpirun -np 4 ./circuit3
Elapsed time: 0.027015.
There are 9 different solutions
mpiuser@Roosevelt:~/nfs/CSC718/openMPI/circuit$
```

0.001894 seconds (~1.894 milliseconds) on 4 processes across **multiple machines**.

0.027015 seconds (~27.015 milliseconds) on 4 processes on the **same machine**.

Interpreting Benchmarking Results

- Speedup: How much faster a parallel algorithm performs compared to a sequential one.

  Speedup = Sequential Time / Parallel Time.

- Efficiency: How effectively parallelization is utilized.

  Efficiency = Speedup / Number of Processors.

- Scalability: How well performance improves as more processors are added.

  Scalability = Time with 1 processor / Time with n processors

Speedup Example

```
mpiuser@Roosevelt:~/nfs/CSC718/openMPI/circuit$ mpirun ./circuit3
Elapsed time: 0.003307.
There are 9 different solutions
mpiuser@Roosevelt:~/nfs/CSC718/openMPI/circuit$ mpirun -np 4 -machinefile machinefile.dsu ./circuit3
Elapsed time: 0.002584.
There are 9 different solutions
mpiuser@Roosevelt:~/nfs/CSC718/openMPI/circuit$
```

1 process: 0.003307 seconds (~3.307 ms)

4 processes: 0.002584 seconds (~2.584 ms)

Speedup= 0.003307 / 0.002584 = 1.279

Parallel execution is ~1.279 times faster.

% Improvement=(Speedup – 1) x 100 =~ 27.9%

Practical Applications of MPI

- **Scientific Simulations:** Weather modeling, climate prediction (e.g., NASA climate models).
- **Data Analytics:** Processing big data in machine learning systems.
- **Engineering Simulations:** Used in research for fluid dynamics, material science, etc.
- **High-Performance Computing:** MPI-based simulations run on systems like CERN or NASA for particle physics.