Parallel Computing

Why parallel computing?

Why parallel computing?

https://computing.llnl.gov/tutorials/parallel_comp/

Why parallel computing?

Please install Mesage Passing Interface MPI in your computer!

http://jetcracker.wordpress.com/2012/03/01/how-to

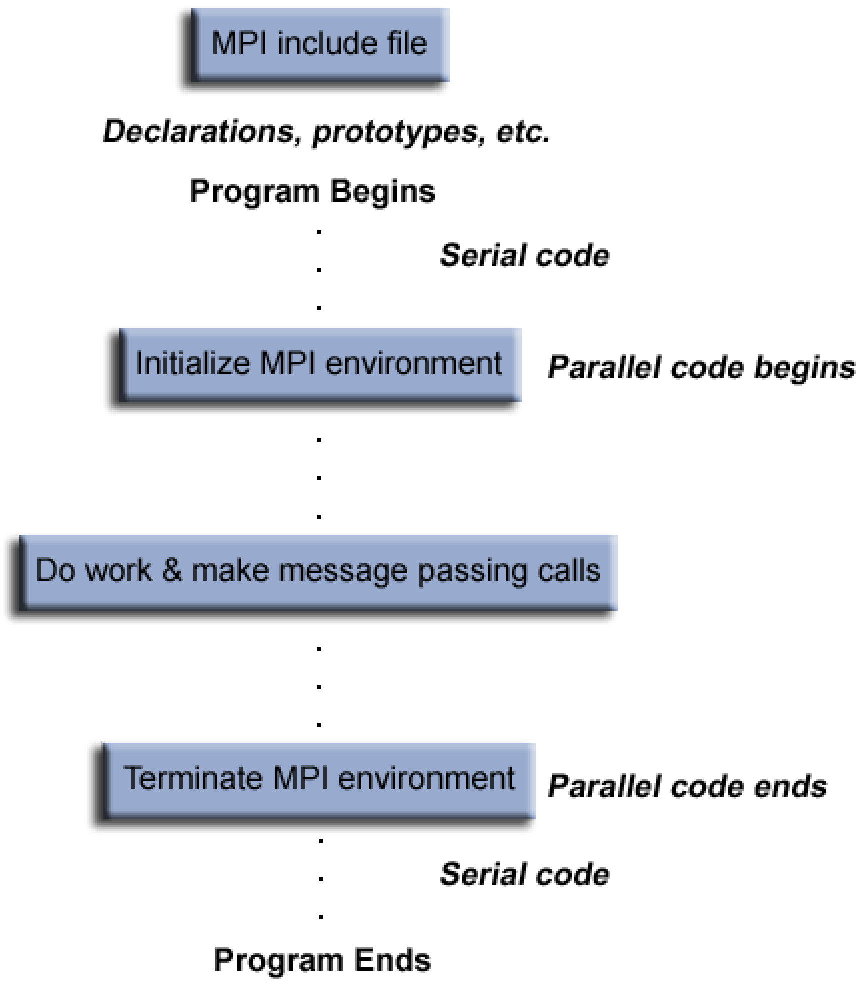
http://en.wikipedia.org/wiki/MPICH

Read more about MPI:

https://computing.llnl.gov/tutorials/mpi/

http://en.wikipedia.org/wiki/Message_Passing_Inte

Structure of the program



MPI

mpi-hello.c

mpi-example1.c

mpi-sumschem.c

mpi-sum.c

What is MPI?

- A *message-passing library specification*
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to provide access to advanced parallel hardware for
  - end users
  - library writers
  - tool developers

Why Use MPI?

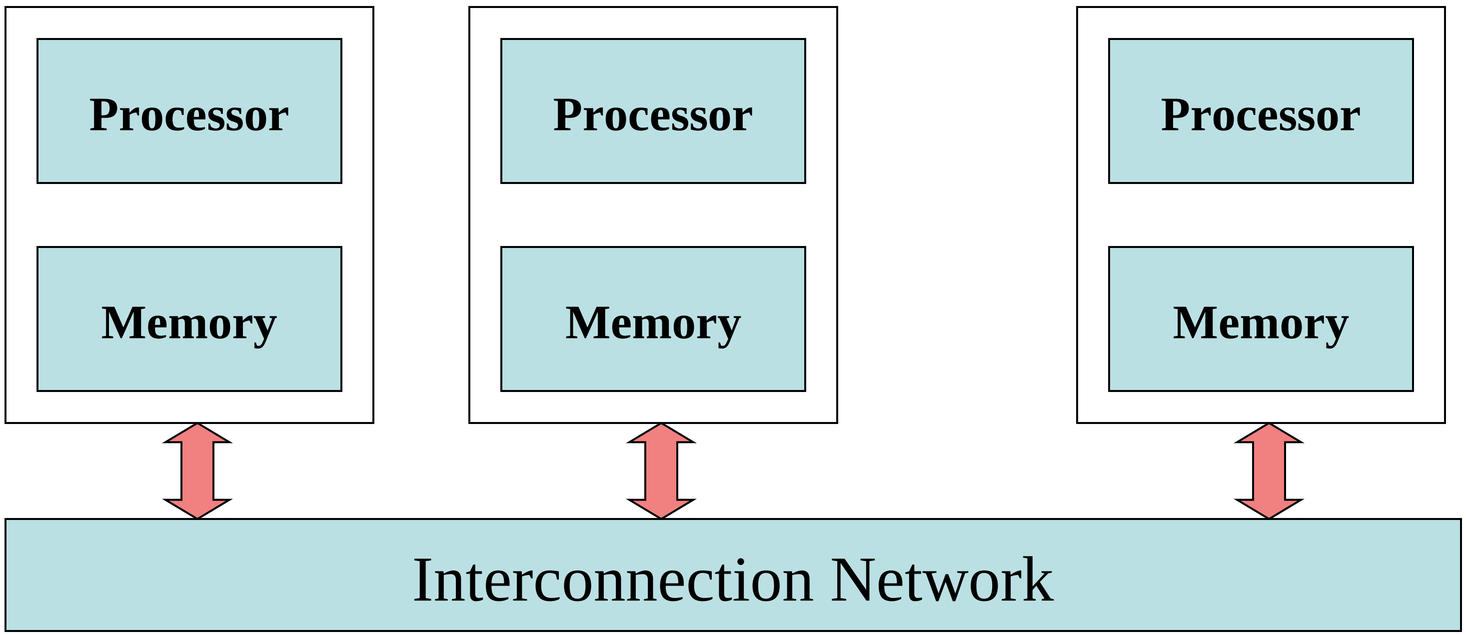
- MPI provides a powerful, efficient, and *portable* way to express parallel programs
- MPI was explicitly designed to enable libraries…
- … which may eliminate the need for many users to learn (much of) MPI

Why Use MPI?

- Standardization - MPI is the only message passing library which 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.
- Functionality - There are over 440 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.
- Availability - A variety of implementations are available, both vendor and public domain.

Programming with MPI

Message-passing Model



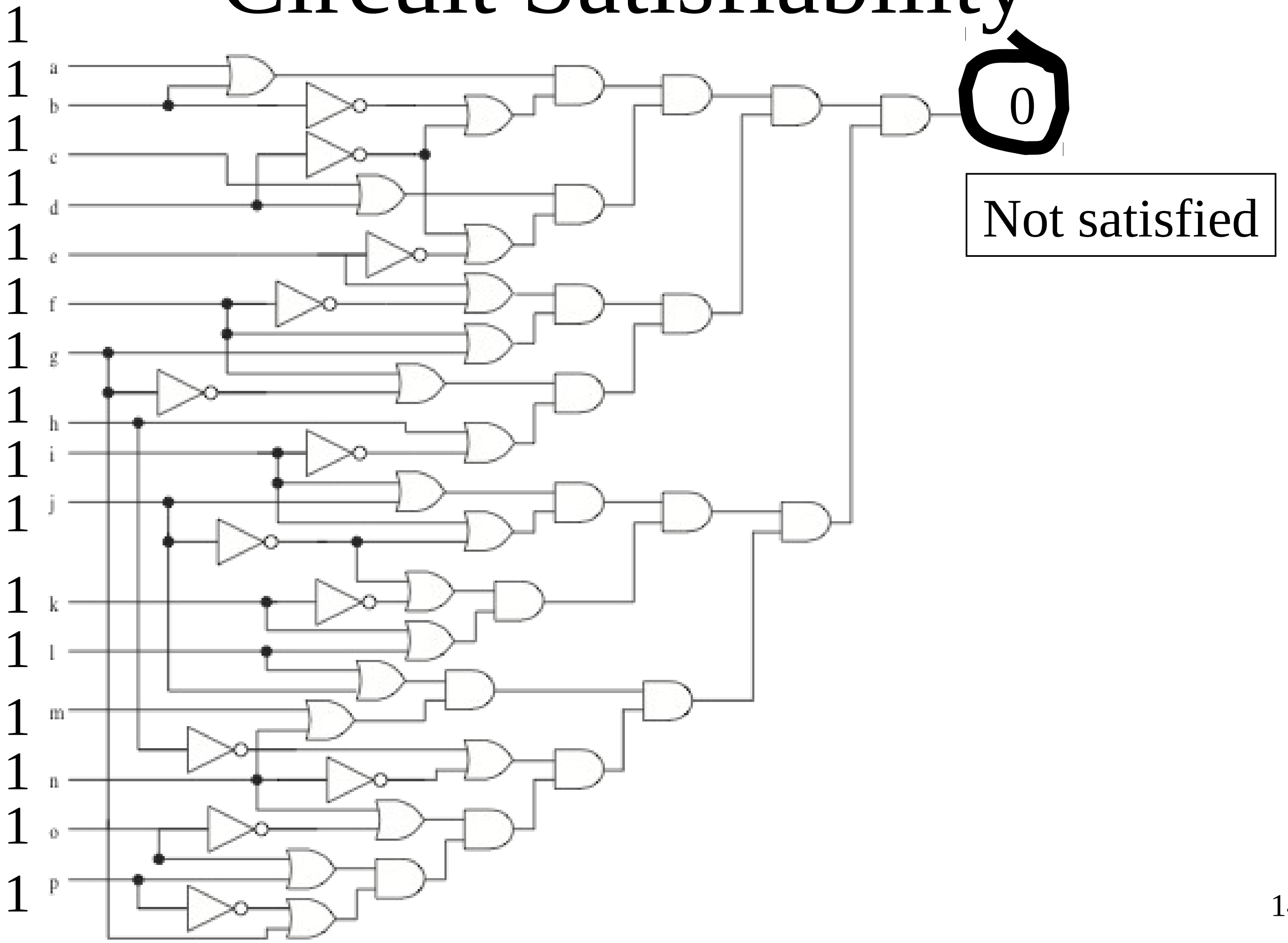
Processes

- Number is specified at start-up time.
  - Typically, fixed throughout the execution.
- All execute same program (SPMD).
  - Each distinguished by a unique ID number.
- Processes explicitly pass messages to communicate and to synchronize with each other.

Advantages of Message-passing Model

- Gives programmer ability to manage the memory hierarchy.
  - What’s local, what’s not?
- Portability to many architectures: can run both on shared-memory and distributed-memory platforms.

Circuit Satisfiability



```
/* 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 ("%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);
   }
}
```

Solution Method

- Circuit satisfiability is NP-complete.
- No known algorithms to solve in polynomial time.
- We seek all solutions.
- We find through exhaustive search.
- 16 inputs $\Rightarrow$ 65,536 combinations to test.

A parallel version: 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 print it.

Include Files

#include <mpi.h>

- MPI header file.

#include <stdio.h>

- Standard I/O header file.

Local Variables

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

- 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.

Shutting Down MPI

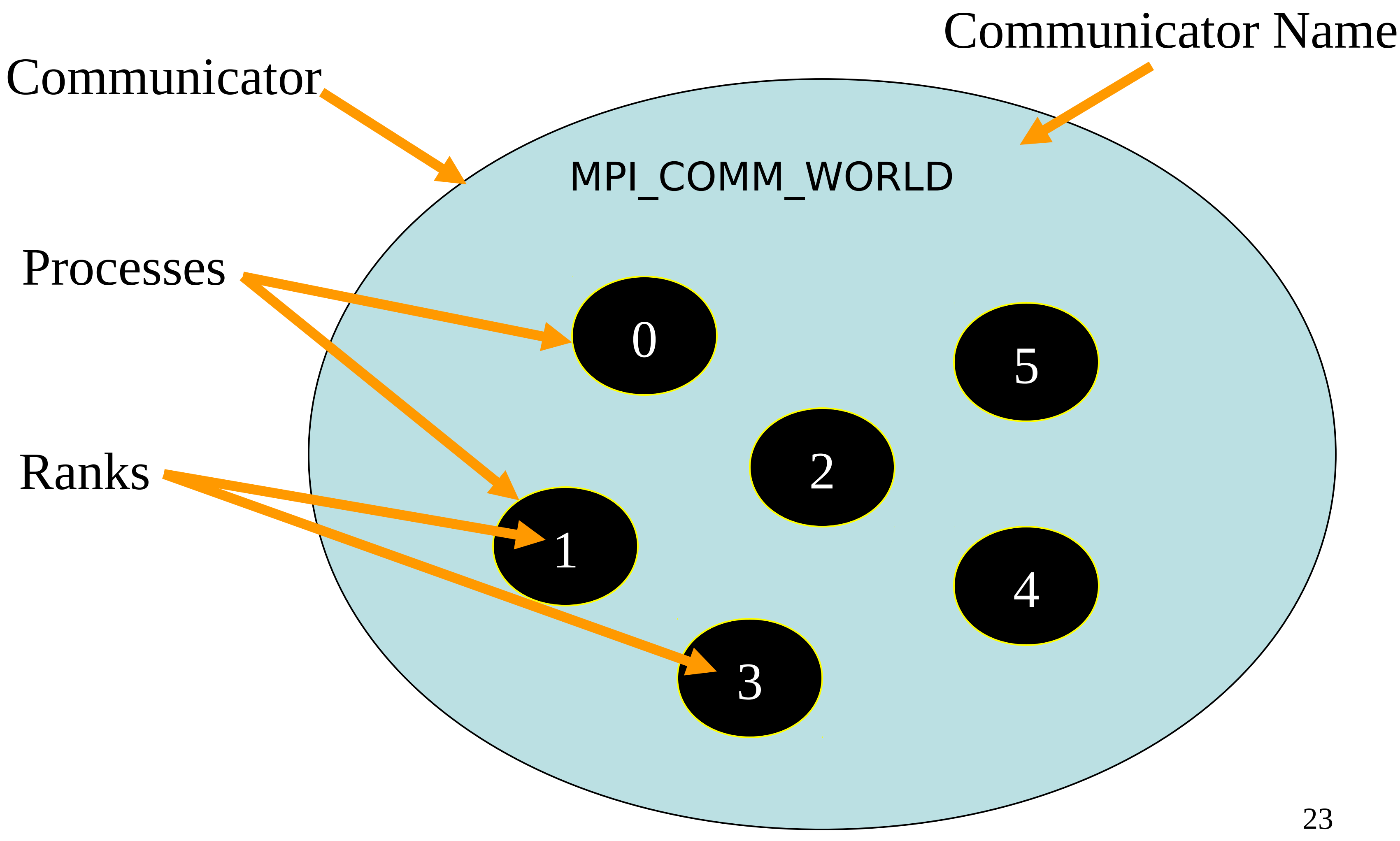
MPI_Finalize();

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

Communicators

- Communicator: opaque object that provides message-passing environment for processes.
- MPI_COMM_WORLD
  - Default communicator.
  - Includes all processes that participate the run.
- It’s possible to create new communicators by user.
  - Always a subset of processes defined in the default communicator.

Communicator



Determine Number of Processes

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

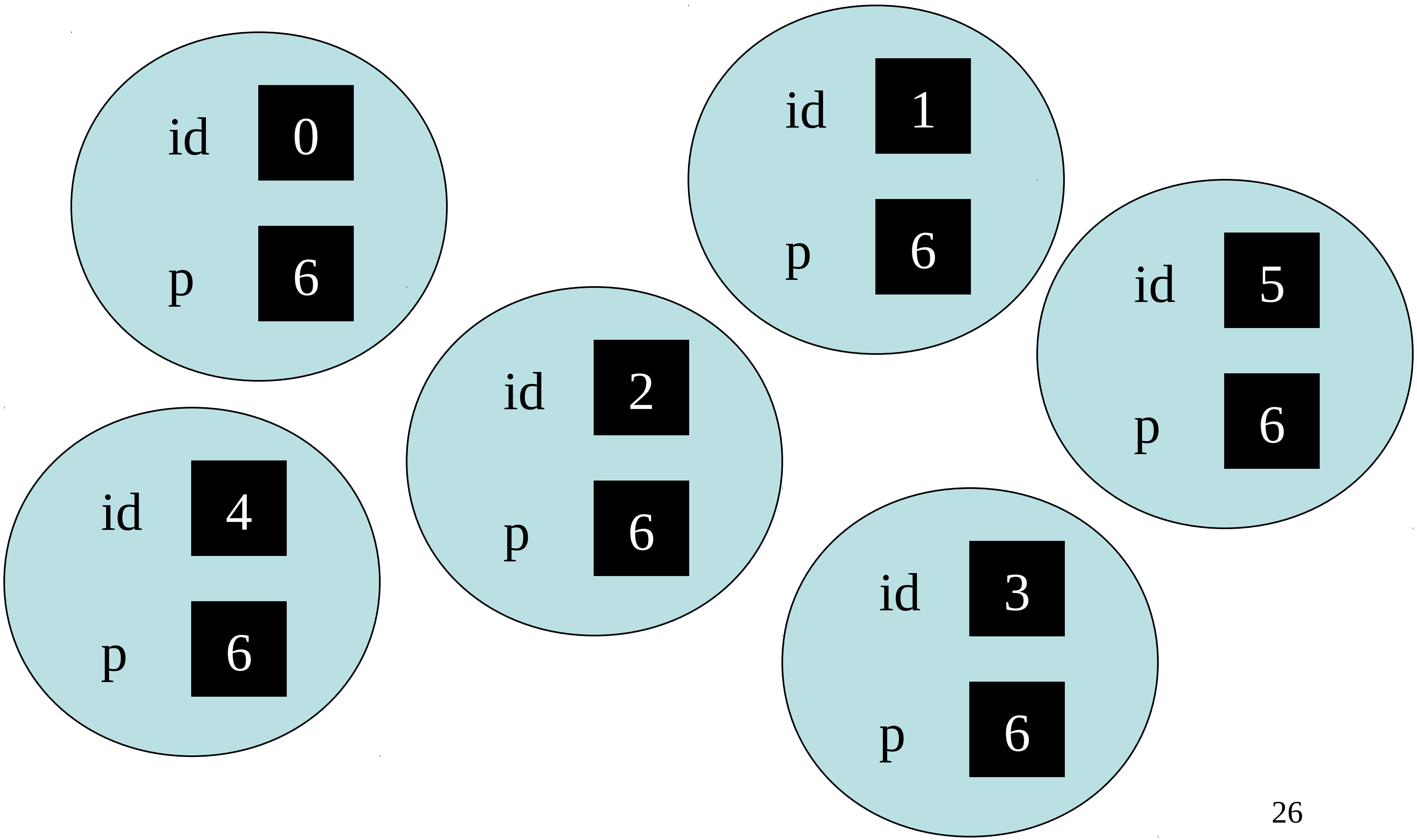
- First argument is communicator,
- Number of processes returned through second argument.

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



Cyclic Allocation of Work

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

- Parallelism is outside function **`check_circuit`**
- It can be an ordinary, sequential function

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

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


   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;
}
```

Put `fflush()` after every `printf()`

Compiling MPI Programs

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

- **mpicc**: script to compile and link MPI programs.
- Flags: same meaning as C compiler.
  - **`-O`** — optimization level.
  - **`-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.

Execution on 1 CPU

```
% mpirun -np 1 sat
0) 1010111110011001
0) 0110111110011001
0) 1110111110011001
0) 1010111111011001
0) 0110111111011001
0) 1110111111011001
0) 1010111110111001
0) 0110111110111001
0) 1110111110111001
Process 0 is done
```

Execution on 2 CPUs

```
% mpirun -np 2 sat
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
```

Execution on 3 CPUs

```
% mpirun -np 3 sat
0) 0110111110011001
0) 1110111111011001
2) 1010111110011001
1) 1110111110011001
1) 1010111111011001
1) 0110111110111001
0) 1010111110111001
2) 0110111111011001
2) 1110111110111001
Process 1 is done
Process 2 is done
Process 0 is done
```

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.
- If process A calls **`printf`** before process B, there is no guarantee process A’s message will appear before process B’s message.

Enhancing the Program

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

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 */

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

Prototype of `MPI_Reduce()`

```
int MPI_Reduce (
   void *operand, /* addr of 1st reduction element */
   void *result,    /* addr of 1st reduction result */
   int count,        /* reductions to perform */
   MPI_Datatype type,  /* type of elements */
   MPI_Op operator,      /* reduction operator */
   int  root,         /* process getting result(s) */
   MPI_Comm comm       /* communicator */
);
```

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_BAND**
- **MPI_BOR**
- **MPI_BXOR**
- **MPI_LAND**
- **MPI_LOR**
- **MPI_LXOR**
- **MPI_MAX**
- **MPI_MAXLOC**
- **MPI_MIN**
- **MPI_MINLOC**
- **MPI_PROD**
- **MPI_SUM**

Our Call to `MPI_Reduce()`

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

Only process 0 will get the result

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

Execution of Second Program

```
% mpirun -np 3 seq2
0) 0110111110011001
0) 1110111111011001
1) 1110111110011001
1) 1010111111011001
2) 1010111110011001
2) 0110111111011001
2) 1110111110111001
1) 0110111110111001
0) 1010111110111001
Process 1 is done
Process 2 is done
Process 0 is done
There are 9 different solutions
```

Benchmarking the Program

- **`MPI_Barrier`** — barrier synchronization.
- **`MPI_Wtick`** — timer resolution.
- **`MPI_Wtime`** — current time.

Benchmarking Code

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

MPI Functions So Far

- **Basic Functions:**
  - **MPI_Init**
  - **MPI_Comm_rank**
  - **MPI_Comm_size**
  - **MPI_Finalize**
- **More:**
  - **MPI_Reduce**
  - **MPI_Barrier**
  - **MPI_Wtime**
  - **MPI_Wtick**

Two More Functions to Conquer the World

```
int MPI_Send(
  void* buf,                    /* addr of 1st element to send */
  int count,                    /* number of elements */
  MPI_Datatype datatype,        /* type of elements */
  int dest,                     /* destination process id */
  int tag,                      /* message tag */
  MPI_Comm comm                 /* communicator */
);

int MPI_Recv(
 void* buf,                     /* addr of 1st element to receive */
  int count,                    /* number of elements */
  MPI_Datatype datatype,        /* type of elements */
  int source,                   /* source process id */
  int tag,                      /* message tag */
  MPI_Comm comm,                /* communicator */
  MPI_Status* status            /* info about message received */
);
```

mpi-block.c

More on **MPI_Recv**

- **MPI_Recv** is a blocking function.
- Use wildcards:
  - **MPI_ANY_SOURCE**.
  - **MPI_ANY_TAG**.
- You can check the status of the received message:
  - MPI_Status.**MPI_SOURCE**
  - MPI_Status.**MPI_TAG**