Message Passing Interface (MPI) Programming

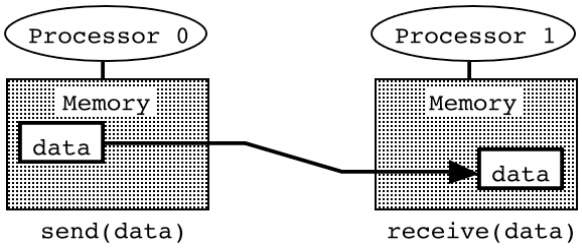
MPI (Message Passing Interface) is a standard message passing system that enables us to write and run applications on parallel computers. In 1992, MPI Forum was formed to develop a portable message passing system. The MPI standard was completed in 1994.[1] Now many vendors are supporting the standard, and there are several public domain implementations of the MPI. An example is the MPICH implementation from Argonne National Laboratory.[2]

USEFUL URL

- Argonne National Laboratory
  `http://www.mcs.anl.gov/mpi`

Message Passing Programming

- Distributed memory processes have access only to local data.
- The sender process issues a send call, and the receiver process issues a matching receive call.



POINT-TO-POINT MESSAGE PASSING

- Program `mpi_simple.c`

```
#include "mpi.h"
#include <stdio.h>
main(int argc, char *argv[]) {
  MPI_Status status;
  int myid;
  int n;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  if (myid == 0) {
    n = 777;
    MPI_Send(&n, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
  }
  else {
    MPI_Recv(&n, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
    printf("n = %d\n", n);
  }
  MPI_Finalize();
}
```

- **Single Program Multiple Data (SPMD) Model:** Identical copies of a program running in parallel. Each of processes begins execution at the same point in a common code image. The processes may each follow distinct flow of control. Processes are distinguished by their process ID's which are used for flow control and communication.

[1] Subsequently MPI1.2 and MPI2 standards were defined, see http://www.mcs.anl.gov/mpi.
[2] *Using MPI, 3rd Ed*. by W. Gropp, E. Lusk, and A. Skjellum (MIT Press, Cambridge, 2014).

Process 0

```
if (myid == 0) {
  n = 777;
  MPI_Send(&n,...);
}
else {
  MPI_Recv(&n,...);
  printf(...);
}
```

Process 1

```
if (myid == 0) {
  n = 777;
  MPI_Send(&n,...);
}
else {
  MPI_Recv(&n,...);
  printf(...);
}
```

MPI LIBRARY CALLS

- All C programs which call MPI library calls must include `mpi.h`.

`int MPI_Init(int *pargc, char ***pargv)`

Establishes the MPI environment. The call to `MPI_Init()` is required in every MPI program and must be the first MPI call. The arguments `MPI_Init()` are the addresses of the usual `main()` arguments `argc` and `argv`. The MPI system removes from the `argv` array any command-line arguments that should be processed by the MPI system before returning to the user program and to decrement `argc` accordingly. (`pargv` is a pointer to `char *argv[]`).

For example, executing `myprogram` as

```
> mpirun -np 4 myprogram -mpiversion x y z
```

the `-np 4` option is interpreted by the `mpirun` program. The `-mpiversion x y z` arguments are then passed to `myprogram`. The `MPI_Init()` call strips the `-mpiversion` argument so that after the call to `MPI_Init()` in the user program, the user program sees command arguments as if it has been called as

```
myprogram x y z
```

`MPI_Init()` returns the error condition. It returns `MPI_SUCCESS` (defined in `mpi.h`) if successful. Error codes are `MPI_ERR_xxx` where `xxx = TYPE` (for invalid data type argument), etc.

- **Dynamic process group:** If we start an MPI application as `mpirun -np 4 myprogram`, it will create 4 processes. MPI allows us to define a subset of these processes in run time using MPI library calls. Suppose a group consists of `n` processes. Processes in the group are numbered sequentially from `0` to `n-1`. This process ID in a group is called **rank**. Dynamic groups are useful, for example, when we want to broadcast a message only to a subset of the total processes.
- **Context:** In one application, we can create multiple groups with overlapping processes. Messages in different groups are never mixed. They are given by the MPI system unique IDs called context.
- **Communicator:** When creating a new group, a user associates it with a communicator variable in order to refer to the group later. A communicator is of type `MPI_Comm` (defined in `mpi.h`). **`MPI_COMM_WORLD`** is a predefined communicator referring to the entire processes.

`int MPI_Comm_rank(MPI_Comm comm, int *rank)`

Obtain the node ID `rank` of the calling process in the range between `0` and `n-1` where `n` is the total number of processes in the group referred to by communicator `comm`.

`int MPI_Finalize()`

This call must be made by every process in an MPI computation. It terminates the MPI environment; no MPI calls may be made by a process after its call to `MPI_Finalize()`. It returns the error condition.

```
int MPI_Send(void *buffer, int count, MPI_Datatype datatype,
             int destination, int tag, MPI_Comm communicator)
```

Synchronous, (blocking) send of a message. It is safe to reuse the buffer when `MPI_Send()` returns (synchronous) (see p. 6). It may block until the message is received by the destination. The MPI standard leaves the decision to each implementation. However, correct programs are such that work even if `MPI_Send()` always blocks. It returns the error condition, `MPI_SUCCESS` if successful.

`buffer:`	refers to the buffer that contains the message to be sent. The buffer may be any legal type.
`count:`	a positive integer that specifies the number of elements to be sent.
`datatype:`	standard datatypes are **`MPI_INT`**, **`MPI_FLOAT`**, **`MPI_DOUBLE`**, **`MPI_CHAR`**, etc. User defined datatype is also supported.
`destination:`	a process ID (rank) where the message is to be sent.
`tag:`	an integer given by a user that identifies the label of the message.
`communicator:`	communicator to specify the group and context where the massage is to be sent.

```
int MPI_Recv(void *buffer, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm communicator, MPI_Status *status)
```

Synchronous, blocking receive of a message. Receive a message and wait for the receive operation to complete before proceeding. When the message is received, it is stored in buffer and the calling process resumes execution.

`buffer:`	refers to the buffer where the message is to be stored. The buffer may be any legal type.
`count:`	a positive integer that specifies the number of elements to be received.
`datatype:`	standard datatypes are **`MPI_INT`**, **`MPI_FLOAT`**, **`MPI_DOUBLE`**, **`MPI_CHAR`**, etc. User defined datatype is also supported.
`source:`	a process ID (rank) where the message is to be received from. **`MPI_ANY_SOURCE`** is a wildcard to accept a message from any source.
`tag:`	an integer given by a user that identifies the label of the message. **`MPI_ANY_TAG`** is a wildcard to accept a message with any tag.
`communicator:`	communicator to specify the group and context where the massage is to be received from.
`status:`	the status is filled in with information about the received message. For example, `status.MPI_SOURCE` is the source process rank `status.MPI_TAG` tag of the received message.

- **Datatype Constructors**: Define a user-defined datatype. Examples are,

  ```
  int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                          MPI_Datatype *newtype)
  ```

  Defines a new datatype that occupies contiguous memory cells consisting of count data elements of oldtype.

  ```
  int MPI_Type_struct(int count, int *array_of_blocklengths, int
                      *array_of_displacements, MPI_Datatype
                      *array_of_types, MPI_Datatype *newtype)
  ```

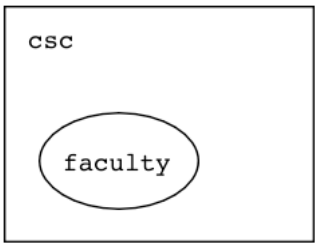
  Defines a new datatype that consists of blocks of memory cells occupied by different datatypes. The offset of each block is specified in bytes and stored in `array_of_displacements[]`.

COMMUNICATOR

PROCESS GROUP

Sometimes we want to perform global operations in a selected subset of all the processes.

- **Example:** `mail faculty`



In MPI, user programs can define new process groups at run time. In each group, member processes are sequentially numbered by `rank` from `0` to `n-1` where `n` is the number of processes in the group.

CONTEXT

Sometimes we do not want to mix two kinds of messages even in the same process group. This is true especially when we develop a library function. Messages sent in a library function must not be received outside that function.

```
main() {
  ...
  library_function();
  ...
  crecv(10,...);
  ...
}

library_function() {
  ...
  csend(10,...);
  ...
}
```

In MPI, context is implemented as a message ID allocated by the system. Context is a kind of message tag allocated by the system at run time in response to a user request. Message exchange occurs only when both user-defined tags as well as system-defined contexts match.

COMMUNICATOR

The notions of group and context are combined in a single object called a communicator. Most communications are specified in terms of rank of the process in the group identified with the given communicator.

- **Example:** `mpi_comm.c`

```
#include "mpi.h"
#include <stdio.h>
#define N 64
main(int argc, char *argv[]) {
  MPI_Comm world, workers;
  MPI_Group world_group, worker_group;
  int myid, nprocs;
  int server, n = -1, ranks[1];
  MPI_Init(&argc, &argv);
  world = MPI_COMM_WORLD;
  MPI_Comm_rank(world, &myid);
  MPI_Comm_size(world, &nprocs);
  server = nprocs-1;
  MPI_Comm_group(world, &world_group);
  ranks[0] = server;
  MPI_Group_excl(world_group, 1, ranks, &worker_group);
  MPI_Comm_create(world, worker_group, &workers);
```

```
  MPI_Group_free(&worker_group);
  if (myid != server) {
    MPI_Allreduce(&myid, &n, 1, MPI_INT, MPI_SUM, workers);
    MPI_Comm_free(&workers);
  }
  printf("process %2d: n = %6d\n", myid, n);
  MPI_Finalize();
}
```

(For `MPI_Allreduce()`, see p. 9.)

```
> mpirun -np 4 mpi_comm
process  0: n =      3
process  1: n =      3
process  2: n =      3
process  3: n =     -1
```

MPI LIBRARY CALLS FOR MANAGING COMMUNICATORS

`MPI_Comm` is a data type to specify communicators.

`MPI_Group` is a data type to specify groups.

`MPI_Comm_size(MPI_Comm communicator, int *nprocs)`
Returns the number of processes `nprocs` in the group identified with `communicator`.

`MPI_Comm_group(MPI_Comm communicator, MPI_Group *group)`
Extracts the `group` information for the given `communicator`. The return value is the handle to the group of the communicator.

`MPI_Group_excl(MPI_Group old_group, int n_excl, int *ranks, MPI_Group *sub_group)`
Excludes `n_excl` members specified by ranks stored in `ranks[]` array from the group `old_group`, and then create `sub_group` with the smaller number of member processes. Excluded are a set of ranks, `{ranks[0], ..., ranks[n_excl-1]}`. This is one of the group constructors.

`MPI_Comm_create(MPI_Comm old_comm, MPI_Group sub_group, MPI_Comm *new_comm)`
Creates a new communicator `new_comm` consisting of `sub_group` of the parent communicator `old_comm`. This is a communicator constructor.

`MPI_Group_free(MPI_Group *group)`
Group destructor for deallocation; frees MPI system resources for `group`.

`MPI_Comm_free(MPI_Comm *communicator)`
Communicator destructor; frees MPI system resources associated with `communicator`.

Group Constructors

`int MPI_Group_incl(MPI_Group old_group, int n, int *ranks, MPI_Group *sub_group)`
Includes `n` members specified by ranks stored in `ranks[]` array from the group `old_group`, and then create `sub_group` with the smaller number of member processes.

`int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *new_group)`
`int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *new_group)`
`int MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *new_group)`
These functions apply set operations (union, intersection, and difference) to `group1` and `group2` to create a `new_group`. For example difference consists of all elements in `group1` not in `group2`.

(Example)

```
group1 = {a, b, c, d}, group2 = {d, a, e}
group1 ∪ group2 = {a, b, c, d, e}            (union)
group1 ∩ group2 = {a, d}                     (intersection)
group1 \ group2 = {b, c}                     (difference)
```

Message Passing Modes
(BY DR. WILLIAM SAPHIR OF NASA AMES RESEARCH CENTER)

- **Non-blocking:** A routine is non-blocking if it is guaranteed to complete regardless of external events (e.g., the other processors).

  Example: A send is non-blocking if it is guaranteed to return whether or not there is a matching receive.

- **Blocking:** A routine is blocking if its completion (return of control to the calling routine) may depend on an external event (an event that is outside the control of the routine itself).

  Example: A send is blocking if it does not return until there is a matching receive.

- **Asynchronous:** A routine is asynchronous if it initiates an operation that happens logically outside the flow of control of the calling process. The important practical distinction is whether the program may be required to check for completion of the operation before proceeding.

- **Synchronous:** A routine is synchronous if its operation happens within the flow of control of the calling process.

  Note that there is no agreement on terminology.

  Example: `pvm_send()` in PVM and `csend()` in NX have almost exactly the same semantics but the documentation says differently.

  `pvm_send()`
  "The pvm_send routine is **asynchronous**. Computation on the sending processor resumes as soon as the message is safely on its way to the receiving processor. This is in contrast to synchronous communication, during which computation on the sending processor halts until the matching receive is executed by the receiving processor."

  `csend()`
  "This is a **synchronous** system call. The calling process waits (blocks) until the send completes. Completion of the send does not mean that the message was received, only that the message was sent and that the send buffer can be reused."

  We will call both calls **nonblocking, synchronous**.

  Why Use Asynchronous Message Passing?

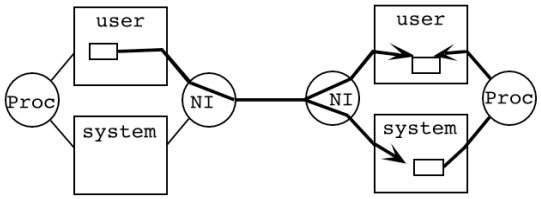
  Answer: To overlap communication with computation.

SYNCHRONOUS MESSAGE PASSING

`MPI_Send()`

Semantics: (blocking), synchronous

- Safe to modify original data immediately after the `MPI_Send()` call.
- Depending on implementation, it may return whether or not a matching receive has been posted, or it may block (especially if no buffer space available). Programmer should assume that it is blocking.



Implementation

- May or may not buffer messages at source and/or destination. (*cf.* The following is the Intel NX implementation to demonstrate the concept of buffering.)
- If a receive has been posted, it delivers the message directly to the user buffer.
- If not, it buffers the message in system space on destination node.
- Does not return until message has been transferred out of the sending user buffer.

`MPI_Recv()`

Semantics: blocking, synchronous

- Blocks for message to arrive.
- Safe to use data on return.

Implementation

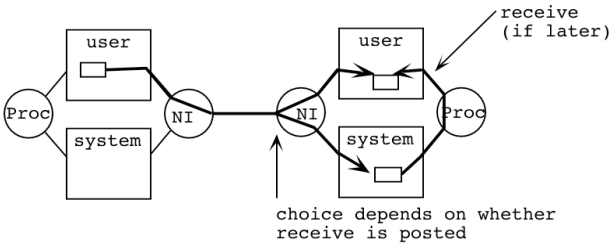
- If a matching message has been buffered, copies messages into user space and returns.
- Posts receive for data.
- Waits for data to arrive.
- Does not return until message has been transferred into the receiving user buffer.

ASYNCHRONOUS MESSAGE PASSING

`MPI_Isend()`

Semantics: non-blocking, asynchronous

- Returns whether or not a matching receive has been posted.
- Not safe to modify original data immediately (use `MPI_Wait()` system call).



Implementation

- May or may not buffer. (*cf.* The following is the Intel NX implementation to demonstrate the concept of buffering.)
- If a receive has been posted, it delivers the message directly to the user buffer.
- If not, it buffers the message in system space on destination node.
- Returns "immediately" before message has been transferred out of the sending user buffer.

`MPI_Irecv()`

Semantics: non-blocking, asynchronous

- Does not block for message to arrive.
- Cannot use data before checking for completion with `MPI_Wait()`.

Implementation (Intel NX)

- If a matching message has arrived (and is buffered), copies messages into user space and returns.
- Posts receive and returns.

Asynchronous communication enables the overlap of computation & communication. (*cf.* An alternative approach is multi-threading integrated with the communication subsystem.)

`int MPI_Irecv(void *buffer, int count, MPI_Datatype datatype,`
`int source, int tag, MPI_Comm comm, MPI_Request *request)`

Posts an asynchronous receive that initiates a receipt of a message. It immediately returns a handle (an ID given by the system) which will be used by `MPI_Wait()`.

`buffer:` refers to the buffer where the received message will be stored.
`count:` the number of elements in the message buffer.
`datatype:` datatype of each receive buffer entry.
`source:` rank of source.
`tag:` an integer given by a user that identifies the label of the message.
`comm:` communicator.
`request:` request handle.

`int MPI_Wait(MPI_Request *request, MPI_Status *status)`

Waits for completion of an asynchronous send or receive operation. When the message is complete, it returns and the associated message buffer is available for reuse, in the case of a send operation, or the buffer contains valid data, in the case of receive.

`request:` request handle.
`status:` received message status object.

- Program `irecv_mpi.c`

```
#include "mpi.h"
#include <stdio.h>
#define N 1000
main(int argc, char *argv[]) {
  MPI_Status status;
  MPI_Request request;
  int send_buf[N], recv_buf[N];
  int send_sum = 0, recv_sum = 0;
  long myid, left, Nnode, msg_id, i;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &Nnode);
  left  = (myid + Nnode - 1) % Nnode;
  for (i=0; i<N; i++) send_buf[i] = myid*N + i;
  /* Post a receive */
  MPI_Irecv(recv_buf, N, MPI_INT, MPI_ANY_SOURCE, 777, MPI_COMM_WORLD,
            &request);
  /* Perform tasks that don't use recv_buf */
  MPI_Send(send_buf, N, MPI_INT, left, 777, MPI_COMM_WORLD);
  for (i=0; i<N; i++) send_sum += send_buf[i];
  /* Complete the receive */
  MPI_Wait(&request, &status);
  /* Now it's safe to use recv_buf */
  for (i=0; i<N; i++) recv_sum += recv_buf[i];
  printf("Node %d: Send %d Recv %d\n", myid, send_sum, recv_sum);
  MPI_Finalize();
}
```

```
> mpirun -np 3 irecv_mpi
Node 0: Send  499500 Recv 1499500
Node 1: Send 1499500 Recv 2499500
Node 2: Send 2499500 Recv  499500
```

Note that 0+1+…+999 = (0+999)×1000/2 = 499500; 1000+1001+…1999 = (1000+1999)×1000/2 = 1499500; 2000+2001+…2999 = (2000+2999)×1000/2 = 2499500.

Global Operations

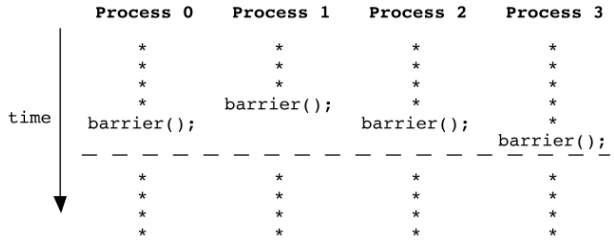
BARRIER SYNCHRONIZATION

- **Barrier:** A point in the program where parallel processes wait for each other. After all the processes have reached the barrier statement, they are all released to continue execution.

```
<A>;
barrier();
<B>;
```

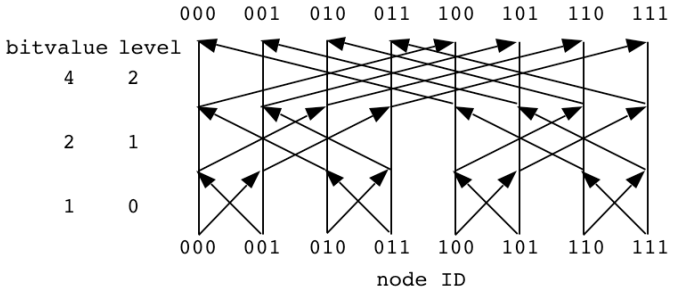
`MPI_Barrier(MPI_Comm communicator)`

Waits until all other nodes in the group have called `MPI_Barrier()` before continuing to execute the next line.



ALL-TO-ALL REDUCTION

- **All-to-all reduction:** Each process contributes a partial value to obtain the global summation. In the end, all the processes will receive the calculated summation. It is also called **multiple aggregation**.
- **Hypercube algorithm:** Communication requirements of a reduction operation can be structured as a series of pairwise exchanges, one with each neighbor in a hypercube (**buttefly**) structure. This structure allows a computation requiring all-to-all communication among `p` processes to be performed in $\log_2 p$ steps.




```
   a000 + a001 + a010 + a011 + a100 + a101 + a110 + a111
=  ((a000 + a001) + (a010 + a011))
+  ((a100 + a101) + (a110 + a111))
```

At each level l, a process exchanges messages with a partner whose node ID differs only at the l-th bit position.

`MPI_Allreduce(void *sendbuf, void *recvbuf, int count,`
`MPI_Datatype datatype, MPI_Op op, MPI_Comm communicator)`

All-to-all reduction operation applied to data pointed to by `sendbuf`. The result is returned to the address `recvbuf` in all the processes in the group specified by the `communicator`. Valid operations for `op` are: **`MPI_SUM, MPI_MIN, MPI_MAX`**, etc.

Other MPI Library Calls

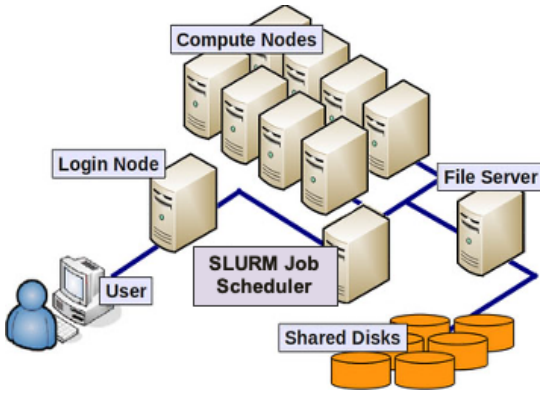
`double MPI_Wtime()`

Returns the elapsed wall-clock time in seconds since some time in the past. The "time in the past" is guaranteed not to change during the lifetime of the process.

How to Run an MPI Application on the CARC Cluster at USC

(This part of the lecture is machine specific and may be supplemented by an appropriate documentation on the computer to be used.)

Center for Advanced Research Computing (CARC) at the University of Southern California (USC) hosts several computing platforms. One of the computing clusters, named Discovery, is a distributed memory system. The login nodes, `discovery.usc.edu` and `discovery2.usc.edu`, are available for user logins in order to edit, compile and submit batch jobs to the compute nodes. Submitted jobs will be scheduled to run on ~500 compute nodes by Slurm (Simple Linux Utility for Resource Management) job scheduling software; see the figure below.



URL

- USC CARC Computing Resource
  `https://carc.usc.edu/user-information/user-guides/hpc-basics/discovery-resources`

Compilation

1. Log in to `discovery.usc.edu` using the secure shell.
   `> ssh <login_id>@discovery.usc.edu`
2. To use the MPI library for message passing, append the following lines in the `.bahshrc` file in your home directory (if you are using the Bash shell interface).
   ```
   module purge
   module load usc
   ```
3. Put your MPI source code, *e.g.*, `mpi_simple.c`, in your directory.
4. Create a file named `makefile`, the content of which is the following, to compile `mpi_simple.c`:
   ```
   mpi_simple: mpi_simple.c
   [TAB]mpicc -O -o mpi_simple mpi_simple.c
   ```
5. Compile the application program: The following will create an executable, `mpi_simple`.
   `discovery1: make mpi_simple`

Execution

1. Create a script file (named, *e.g.*, `mpi_simple.sl`) to submit an MPI job using the Slurm (Simple Linux Utility for Resource Management), the content of which is (note the executable, `mpi_simple`, needs be placed in the same directory where you submit the script file):

```
#!/bin/bash                          // Interpret the following with the bash command interpreter
#SBATCH --nodes=1                    // Request 1 node
#SBATCH --ntasks-per-node=2          // Request 2 processors per node
#SBATCH --time=00:00:10              // Maximum wall-clock time for the job is 10 seconds
#SBATCH --output=mpi_simple.out      // Standard output will be returned in file mpi_simple.out
#SBATCH -A anakano_429               // Charge the computation to CSCI 596 class account
```

```
mpirun -n $SLURM_NTASKS ./mpi_simple     // Run the job on 2 (= 1 × 2) processors allocated by Slurm
```

2. Submit a Slurm job:

```
discovery1: sbatch mpi_simple.sl
Submitted batch job 63695                // Slurm has given the job ID 63695
```

- You can check the status of your Slurm job using the `squeue` command.

```
discovery1: squeue -u <login_id>
JOBID PARTITION        NAME     USER ST       TIME  NODES  NODELIST(REASON)
63695      main  mpi_simple  anakano PD       0:00      1  (Resources)
```

After the job is completed, you can see the result (standard output and error if any) in the file, `mpi_simple.out`, in the working directory specified in your script file.

```
discovery1: more mpi_simple.out
n = 777
```

- You can kill a Slurm job using the `scancel` command, specifying its Slurm job ID.

```
discovery1: scancel 63695
```

Further Information

For further information about running jobs on Discovery, please see:
`https://carc.usc.edu/user-information/user-guides/hpc-basics/getting-started-discovery`