Basics of Supercomputing

MPI: Programming with the Message Passing Interface
Topics

• MPI Standard
• MPI-1 Model and Basic Calls
• MPI Communicators
• Point to Point Communication
• Trapezoidal Rule: a case study
• MPI Collective Calls
MPI Standard

- From 1992-1994, a community representing both vendors and users decided to create a standard interface to message passing calls in the context of distributed memory parallel computers
  - “Just” an API
  - FORTRAN77 and C bindings
  - Reference implementation (mpich) also developed
  - Vendors also kept their own internals (behind the API)
MPI Standard

• Since then
  – MPI-1.1
    • Fixed bugs, clarified issues
  – MPI-2
    • Extended MPI
      – New datatype constructors, language interoperability
    • New functionality
      – One-sided communication
      – MPI I/O
      – Dynamic processes
    • FORTRAN90 and C++ bindings

• Best MPI reference
Topics

• MPI Standard
• MPI-1 Model and Basic Calls
• MPI Communicators
• Point to Point Communication
• Trapezoidal Rule: a case study
• MPI Collective Calls
MPI : Basics

- Every MPI program must contain the preprocessor directive
  
  ```
  #include "mpi.h"
  ```

- The mpi.h file contains the definitions and declarations necessary for compiling an MPI program.

- mpi.h is usually found in the “include” directory of most MPI installations. For example on arete:

  ```
  [tim@n01 ~]$ ls /opt/packages/mpich-1.2.7p1/include/
  base11.h  mpe_graphics.h  mpeexten.h  mpi_errno.h  mpierr.h
  mpe.h     mpe_log.h      mpi.h       mpio.h      protofix.h
  [tim@n01 ~]$
  ```

  ```
  ...
  #include "mpi.h"
  ...
  MPI_Init(&Argc,&Argv);
  ...
  ...
  MPI_Finalize();
  ...
MPI: Initializing MPI Environment

Function: \texttt{MPI\_init()}

\begin{verbatim}
int MPI_Init(int *argc, char ***argv)
\end{verbatim}

Description:
Initializes the MPI execution environment. \texttt{MPI\_init()} must be called before any other MPI functions can be called and it should be called only once. It allows systems to do any special setup so that MPI Library can be used. \texttt{argc} is a pointer to the number of arguments and \texttt{argv} is a pointer to the argument vector. On exit from this routine, all processes will have a copy of the argument list.

\begin{verbatim}
...  
#include "mpi.h"
...
MPI_Init(&argc,&argv);
...
MPI_Finalize();
...
\end{verbatim}

MPI: Terminating MPI Environment

Function: MPI_Finalize()

Description:
Terminates MPI execution environment. All MPI processes must call this routine before exiting. MPI_Finalize() need not be the last executable statement or even in main; it must be called at some point following the last call to any other MPI function.

```c
...
#include "mpi.h"
...
MPI_Init(&argc,&argv);
...
MPI_Finalize();
...
```

MPI Hello World

• C source file for a simple MPI Hello World

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

int main( int argc, char *argv[])
{
    MPI_Init( &argc, &argv);
    printf("Hello, World!\n");
    MPI_Finalize();
    return 0;
}
```
Building an MPI Executable

- **Library version**
  
  ```bash
  cc -lheaderdir -Llibdir mpicode.c -lmpi
  ```
  
  - User knows where header file and library are, and tells compiler

- **Wrapper version**
  
  ```bash
  mpicc -o executable mpicode.c
  ```
  
  - Does the same thing, but hides the details from the user

- You can do **either one**, but don't try to do both!

- For our "Hello World" example:
  
  ```bash
  mpicc -o hello hello.c
  ```
Running an MPI Executable

- Some number of processes are started somewhere
  - Again, standard doesn’t talk about this
  - Implementation and interface varies
  - Usually, some sort of mpirun command starts some number of copies of an executable according to a mapping
  - Example:
    ```
    mpirun -np 2 ./a.out
    ```
    - Run two copies of ./a.out where the system specifies
  - Most production supercomputing resources wrap the mpi run command with higher level scripts that interact with scheduling systems such as PBS / LoadLeveler for efficient resource management and multi-user support
  - Sample PBS / Load Leveler job submission scripts:

**PBS File:**
```
#!/bin/bash
#PBS -l walltime=120:00:00,nodes=8:ppn=4
cd /home/cdekate/S1_L2_Demos/adc/
pwd
date
mpirun -np 32 -machinefile $PBS_NODEFILE ./padcirc
date
```

**LoadLeveler File:**
```
#!/bin/bash
#@ job_type = parallel
#@ job_name = SIMID
#@ wall_clock_limit = 120:00:00
#@ node = 8
#@ total_tasks = 32
#@ initialdir = /scratch/cdekate/
#@ executable = /usr/bin/poe
#@ arguments = /scratch/cdekate/padcirc
#@ queue
```
Running the Hello World example

- Using **mpirun**:
  ```
  mpirun -np 8 ./hello
  Hello, World!
  Hello, World!
  Hello, World!
  Hello, World!
  Hello, World!
  Hello, World!
  Hello, World!
  ```

- Using **PBS**:
  ```
  hello.pbs:
  
  #!/bin/bash
  #PBS -N hello
  #PBS -l walltime=00:01:00,nodes=2:ppn=4
  cd /home/cdekate/2008/17
  pwd
  date
  mpirun -np 8 -machinefile $PBS_NODEFILE ./hello
  date
  ```

more hello.o10030:
```
/home/cdekate/2008/17
Wed Feb  6 10:58:36 CST 2008
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Wed Feb  6 10:58:37 CST 2008
```
Topics

• MPI Standard
• MPI-1 Model and Basic Calls
• MPI Communicators
• Point to Point Communication
• Trapezoidal Rule: a case study
• Review of Basic MPI Calls
• MPI Collective Calls
MPI Communicators

- Communicator is an internal object
- MPI Programs are made up of communicating processes
- Each process has its own address space containing its own attributes such as rank, size (and argc, argv, etc.)
- MPI provides functions to interact with it
- Default communicator is MPI_COMM_WORLD
  - All processes are its members
  - It has a size (the number of processes)
  - Each process has a rank within it
  - One can think of it as an ordered list of processes
- Additional communicator(s) can co-exist
- A process can belong to more than one communicator
- Within a communicator, each process has a unique rank
MPI: Size of Communicator

Function: MPI_Comm_size()

int MPI_Comm_size ( MPI_Comm comm, int *size )

Description:
Determines the size of the group associated with a communicator (comm). Returns an integer number of processes in the group underlying comm executing the program. If comm is an inter-communicator, return the size of the local group. The comm in the argument list refers to the communicator-group to be queried, the result of the query (size of the comm group) is stored in the variable size.

```c
... #include "mpi.h"
...
int size;
MPI_Init(&Argc,&Argv);
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
err = MPI_Finalize();
...```

MPI: Rank of a process in comm

Function: MPI_Comm_rank()

int MPI_Comm_rank ( MPI_Comm comm, int *rank )

Description:
Returns the rank of the calling process in the group underlying the comm. If the \textit{comm} is an inter-communicator, the call MPI_Comm_rank returns the rank of the process in the local group. The first parameter \textit{comm} in the argument list is the communicator to be queried, and the second parameter \textit{rank} is the integer number rank of the process in the group of \textit{comm}.

```c
...
#include "mpi.h"
...
int rank;
MPI_Init(&Argc,&Argv);
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
err = MPI_Finalize();
...
```

Example: communicators

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

int main( int argc, char *argv[])
{
    int rank, size;
    MPI_Init( &argc, &argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank);
    MPI_Comm_size( MPI_COMM_WORLD, &size);
    printf("Hello, World! from %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Determines the rank of the current process in the communicator-group

```
MPI_COMM_WORLD
```

Determines the size of the communicator-group

```
MPI_COMM_WORLD
```

Hello, World! from 1 of 8
Hello, World! from 0 of 8
Hello, World! from 5 of 8
...
Example : Communicator & Rank

• Compiling:

  mpicc -o hello2 hello2.c

• Result:

  Hello, World! from 4 of 8
  Hello, World! from 3 of 8
  Hello, World! from 1 of 8
  Hello, World! from 0 of 8
  Hello, World! from 5 of 8
  Hello, World! from 6 of 8
  Hello, World! from 7 of 8
  Hello, World! from 2 of 8
Topics

• MPI Standard
• MPI-1 Model and Basic Calls
• MPI Communicators
• Point to Point Communication
• Trapezoidal Rule: a case study
• Review of Basic MPI Calls
• MPI Collective Calls
MPI : Point to Point Communication primitives

- A basic communication mechanism of MPI between a pair of processes in which one process is sending data and the other process receiving the data, is called “point to point communication”
- Message passing in MPI program is carried out by 2 main MPI functions
  - MPI_Send – sends message to a designated process
  - MPI_Recv – receives a message from a process
- Each of the send and recv calls is appended with additional information along with the data that needs to be exchanged between application programs
- The message envelope consists of the following information
  - The rank of the receiver
  - The rank of the sender
  - A tag
  - A communicator
- The source argument is used to distinguish messages received from different processes
- Tag is user-specified int that can be used to distinguish messages from a single process
Message Envelope

• Communication across processes is performed using messages.
• Each message consists of a fixed number of fields that is used to distinguish them, called the Message Envelope:
  – Envelope comprises source, destination, tag, communicator
  – Message comprises Envelope + data
• Communicator refers to the namespace associated with the group of related processes

Source: process0
Destination: process1
Tag: 1234
Communicator: MPI_COMM_WORLD

Source: process0
Destination: process1
Tag: 1234
Communicator: MPI_COMM_WORLD
MPI: (blocking) Send message

Function: MPI_Send()

```
int MPI_Send(
    void                 *message,
    int                    count,
    MPI_Datatype datatype,
    int                    dest,
    int                    tag,
    MPI_Comm    comm )
```

Description:
The contents of message are stored in a block of memory referenced by the first parameter `message`. The next two parameters, count and datatype, allow the system to determine how much storage is needed for the message: the message contains a sequence of `count` values, each having `MPI` type `datatype`. MPI allows a message to be received as long as there is sufficient storage allocated. If there isn't sufficient storage an overflow error occurs. The `dest` parameter corresponds to the rank of the process to which message has to be sent.

### MPI : Data Types

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>

You can also define your own (derived datatypes), such as an array of ints of size 100, or more complex examples, such as a struct or an array of structs.
**MPI: (blocking) Receive message**

<table>
<thead>
<tr>
<th>Function:</th>
<th><code>MPI_Recv()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>int <code>MPI_Recv</code>(</td>
<td></td>
</tr>
<tr>
<td>void *message,</td>
<td></td>
</tr>
<tr>
<td>int count,</td>
<td></td>
</tr>
<tr>
<td><code>MPI_Datatype</code> datatype,</td>
<td></td>
</tr>
<tr>
<td>int source,</td>
<td></td>
</tr>
<tr>
<td>int tag,</td>
<td></td>
</tr>
<tr>
<td><code>MPI_Comm</code> comm,</td>
<td></td>
</tr>
<tr>
<td><code>MPI_Status</code> *status )</td>
<td></td>
</tr>
</tbody>
</table>

**Description:**

The contents of message are stored in a block of memory referenced by the first parameter `message`. The next two parameters, count and datatype, allow the system to determine how much storage is needed for the message: the message contains a sequence of `count` values, each having `MPI` type `datatype`. MPI allows a message to be received as long as there is sufficient storage allocated. If there isn’t sufficient storage an overflow error occurs. The `source` parameter corresponds to the rank of the process from which the message has been received. The `MPI_Status` parameter in the `MPI_Recv()` call returns information on the data that was actually received. It references a record with 2 fields – one for the source and one for the tag [http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Recv.html](http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Recv.html)
### MPI_Status object

**Object:** MPI_Status

**Example usage:**

```c
MPI_Status status;
```

**Description:**

The MPI_Status object is used by the receive functions to return data about the message, specifically the object contains the id of the process sending the message (MPI_SOURCE), the message tag (MPI_TAG), and error status (MPI_ERROR).

```c
#include "mpi.h"
...
MPI_Status status; /* return status for */
...
MPI_Init(&argc, &argv);
...
if (my_rank != 0) {
...
    MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}
else { /* my rank == 0 */
    for (source = 1; source < p; source++) {
        MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
    ...
    MPI_Finalize();
...
```
/* hello world, MPI style */

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

int main(int argc, char* argv[]) {
    int my_rank;       /* rank of process */
    int p;             /* number of processes */
    int source;        /* rank of sender */
    int dest;          /* rank of receiver */
    int tag=0;         /* tag for messages */
    char message[100]; /* storage for message */
    MPI_Status status; /* return status for receive */

    /* Start up MPI */
    MPI_Init(&argc, &argv);

    /* Find out process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!", my_rank);
        dest = 0;
        /* Use strlen+1 so that \0 gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    } else { /* my rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    printf("Greetings from process %d!\n", my_rank);

    /* Shut down MPI */
    MPI_Finalize();

} /* end main */

Src : Prof. Amy Apon
Communication map for the example.

```bash
mpirun -np 8 -machinefile hosts ./hello3
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
Greetings from process 4!
Greetings from process 5!
Greetings from process 6!
Greetings from process 7!
Greetings from process 0!
Writing logfile....
Finished writing logfile.
[cdekate@celeritas l7]$
```
Topics

• MPI Standard
• MPI-1 Model and Basic Calls
• MPI Communicators
• Point to Point Communication
• Trapezoidal Rule: a case study
• Review of Basic MPI Calls
• MPI Collective Calls
Approximating Integrals: Definite Integral

- Problem: to find an approximate value to a definite integral 

\[ \int_{a}^{b} f(x) \, dx. \]

- A definite integral from \( a \) to \( b \) of a non negative function \( f(x) \) can be thought of as the area bound by the x-axis, the vertical lines \( x=a \) and \( x=b \), and the graph of \( f(x) \).
Approximating Integrals: Trapezoidal Rule

Approximating area under the curve can be done by dividing the region under the curve into regular geometric shapes and then adding the areas of the shapes.

In Trapezoidal Rule, the region between $a$ and $b$ can be divided into $n$ trapezoids of base $h = (b-a)/n$

The area of a trapezoid can be calculated as $\frac{h(b_1 + b_2)}{2}$

In the case of our function the area for the first block can be represented as

$$\frac{h(f(a) + f(a + h))}{2}$$

The area under the curve bounded by $a,b$ can be approximated as:

$$\left[ \frac{h(f(a) + f(a + h))}{2} \right] + \left[ \frac{h(f(a + h) + f(a + 2h))}{2} \right] + \left[ \frac{h(f(a + 2h) + f(a + 3h))}{2} \right] + \left[ \frac{h(f(a + 3h) + f(b))}{2} \right]$$
Parallelizing Trapezoidal Rule

• One way of parallelizing Trapezoidal rule:
  – Distribute chunks of workload (each workload characterized by its own subinterval of \([a,b]\) to each process)
  – Calculate \(f\) for each subinterval
  – Finally add the \(f\) calculated for all the subintervals to produce result for the complete problem \([A,B]\)

• Issues to consider
  – Number of trapezoids \((n)\) are equally divisible across \((p)\) processes (load balancing).
  – First process calculates the area for the first \(n/p\) trapezoids, second process calculates the area for the next \(n/p\) trapezoids and so on

• Key information related to the problem that each process needs is the
  – Rank of the process
  – Ability to derive the workload per processor as a function of rank

Assumption: Process 0 does the summation
Parallelizing Trapezoidal Rule

- Algorithm
  Assumption: Number of trapezoids \( n \) is evenly divisible across \( p \) processors
  
  \[
  h = \frac{(b - a)}{n}
  \]

  - Each process calculates its own workload (interval to integrate)
    - local number of trapezoids (local\_n) = \( \frac{n}{p} \)
    - local starting point (local\_a) = \( a + (\text{process\_rank} \times \text{local\_n} \times h) \)
    - local ending point (local\_b) = (local\_a + local\_n \times h)
  
  - Each process calculates its own integral for the local intervals
    - For each of the local\_n trapezoids calculate area
    - Aggregate area for local\_n trapezoids

- If PROCESS\_RANK == 0
  - Receive messages (containing sub-interval area aggregates) from all processors
  - Aggregate (ADD) all sub-interval areas

- If PROCESS\_RANK > 0
  - Send sub-interval area to PROCESS\_RANK(0)

Classic SPMD: all processes run the same program on different datasets.
Parallel Trapezoidal Rule

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

main(int argc, char** argv) {
    int my_rank; /* My process rank */
    int p; /* The number of processes */
    float a = 0.0; /* Left endpoint */
    float b = 1.0; /* Right endpoint */
    int n = 1024; /* Number of trapezoids */
    float h; /* Trapezoid base length */
    float local_a; /* Left endpoint my process */
    float local_b; /* Right endpoint my process */
    int local_n; /* Number of trapezoids for my calculation */
    float integral; /* Integral over my interval */
    float total; /* Total integral */
    int source; /* Process sending integral */
    int dest = 0; /* All messages go to 0 */
    int tag = 0;
    MPI_Status status;

Trapezoidal Example Adapted from Parallel Programming in MPI P.Pacheco Ch 4
float Trap(float local_a, float local_b, int local_n, float h); /* Calculate local integral */

/* Let the system do what it needs to start up MPI */
MPI_Init(&argc, &argv);

/* Get my process rank */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

/* Find out how many processes are being used */
MPI_Comm_size(MPI_COMM_WORLD, &p);

h = (b-a)/n; /* h is the same for all processes */
local_n = n/p; /* So is the number of trapezoids */

/* Length of each process' interval of integration = local_n*h. So my interval * starts at: */
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
integral = Trap(local_a, local_b, local_n, h);

Trapezoidal Example Adapted from Parallel Programming in MPI P.Pacheco Ch 4
Parallel Trapezoidal Rule

/* Add up the integrals calculated by each process */
if (my_rank == 0) {
    total = integral;
    for (source = 1; source < p; source++) {
        MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
                  MPI_COMM_WORLD, &status);
        total = total + integral;
    }
} else {
    MPI_Send(&integral, 1, MPI_FLOAT, dest, tag,
             MPI_COMM_WORLD);
}
/* Print the result */
if (my_rank == 0) {
    printf("With n = %d trapezoids, our estimate
           \n", n);
    printf("of the integral from %f to %f = %f\n",
           a, b, total);
}
/* Shut down MPI */
MPI_Finalize();
} /* main */

Trapezoidal Example Adapted from Parallel Programming in MPI P. Pacheco Ch 4
Parallel Trapezoidal Rule

float Trap(
    float local_a  /* in */,
    float local_b  /* in */,
    int    local_n /* in */,
    float  h       /* in */) {

    float integral;  /* Store result in integral */
    float x;
    int i;

    float f(float x);  /* function we're integrating */

    integral = (f(local_a) + f(local_b))/2.0;
    x = local_a;
    for (i = 1; i <= local_n-1; i++) {
        x = x + h;
        integral = integral + f(x);
    }
    integral = integral*h;
    return integral;
} /* Trap */

float f(float x) {
    float return_val;
    /* Calculate f(x). */
    /* Store calculation in return_val. */
    return_val = x*x;
    return return_val;
} /* f */
Parallel Trapezoidal Rule

[cdekate@celeritas l7]$ mpirun -np 8 ... trap
With n = 1024 trapezoids, our estimate
of the integral from 2.000000 to 25.000000 = 5205.667969
Writing logfile....
Finished writing logfile.

[cdekate@celeritas l7]$ ./serial
Enter a, b, and n
2 25 1024
With n = 1024 trapezoids, our estimate
of the integral from 2.000000 to 25.000000 = 5205.666016
[cdekate@celeritas l7]$
Topics

• MPI Standard
• MPI-1 Model and Basic Calls
• MPI Communicators
• Point to Point Communication
• Point to Point Communication In-depth
• Trapezoidal Rule: a case study
• MPI Collective Calls
Collective Calls

• A communication pattern that encompasses all processes within a communicator is known as collective communication

• MPI has several collective communication calls, the most frequently used are:
  – Synchronization
    • Barrier
  – Communication
    • Broadcast
    • Gather & Scatter
    • All Gather
  – Reduction
    • Reduce
    • AllReduce
MPI Collective Calls: Barrier

Function: MPI_Barrier()

```c
int MPI_Barrier ( MPI_Comm comm )
```

Description:
Creates barrier synchronization in a communicator group `comm`. Each process, when reaching the MPI_Barrier call, blocks until all the processes in the group reach the same MPI_Barrier call.

Example: MPI_Barrier()

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

int main (int argc, char *argv[])
{
    int rank, size, len;
    char name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc, &argv);

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(name, &len);

    MPI_Barrier(MPI_COMM_WORLD);

    printf("Hello world! Process %d of %d on %s\n", rank, size, name);
    MPI_Finalize();
    return 0;
}

[cdekate@celeritas collective]$ mpirun -np 8 barrier
Hello world! Process 0 of 8 on celeritas.cct.lsu.edu
Writing logfile....
Finished writing logfile.
Hello world! Process 4 of 8 on compute-0-3.local
Hello world! Process 1 of 8 on compute-0-0.local
Hello world! Process 3 of 8 on compute-0-2.local
Hello world! Process 6 of 8 on compute-0-5.local
Hello world! Process 7 of 8 on compute-0-6.local
Hello world! Process 5 of 8 on compute-0-4.local
Hello world! Process 2 of 8 on compute-0-1.local
[cdekate@celeritas collective]$