Message Passing Interface
MPI

Robert Whitten Jr

Props to my BF@W – Rebecca Hartman-Baker

Terrorist Raccoon attacks 280MW substation

Shortly after 2:00am, August 3, a determined raccoon penetrated one of two on-site substations at ORNL, and entered a 13,800V switchgear without the correct safety training and without the correct personal protective equipment (PPE). The result was not pretty. The raccoon's next of kin have been notified. The downstream effect caused two 13,800V, 600A service lines to the computing facility to trip. The electrical service outage affected large portions of the computing facility, including the file systems and compute systems as well as the infrastructure services in a separate area.

Utilities crews, after finishing a short snack of roasted raccoon, are executing an incident investigation to determine what circumstances were in play that would simultaneously affect more than a single 13,800V feeder. We will review the after-action report for possible corrective actions. As this occurred at the distribution/ substation level, any physical changes described in the after action report would be coupled with other large full-site/multi-site maintenance events, and would be scheduled far in advance.

Cray and ORNL crews were onsite in the early hours of the morning restoring service.

Introduction

- Message Passing Interface (MPI)
- Standard – not a language
- Fortran, C, C++ bindings (libraries)
- Industry standard
- Standard created by user community
  - MPI-1 – original standard
  - MPI-2 – add C++ and more advanced functions
- Programming models supported
  - Single Program, Multiple Data (SPMD)
  - Multiple Programs, Multiple Data (MPMD)

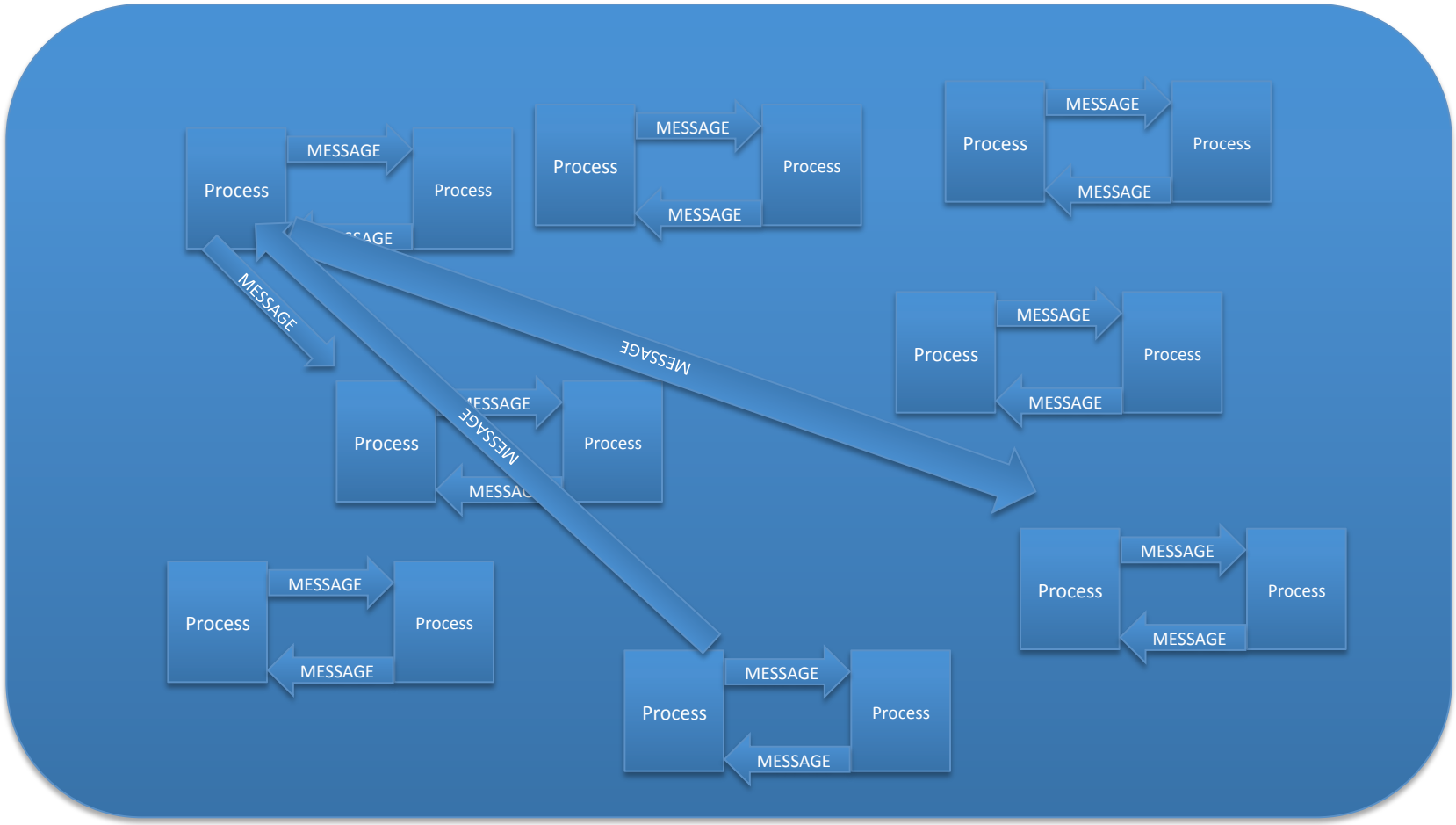
Message Passing

- Processes pass data back and forth
  - Data = message
- Items needed to send a message
  - Sender & receiver
  - Data location at sender and receiver
  - Size of data sent and received

MPI Concepts

- Workflow:
  - Initialize environment -> Setup group -> Distribute data -> do work -> Close environment
- Some terms:
  - Ranks = process ID
  - Size = how many processes
  - Comm = group of processes
  - Tag = message ID
  - Datatypes = MPI definition of types
    - MPI_INT, MPI_CHAR, user created, etc.

MPI Comm World



Six MPI commands you can't live without

- `int MPI_Init(int *argc, char **argv)`
- `int MPI_Finalize(void)`
- `int MPI_Comm_size(MPI_Comm comm, int *size)`
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`

Initiation and Termination

- `MPI_Init(int *argc, char **argv)` initiates MPI
  - Place in body of code after variable declarations and before any MPI commands
- `MPI_Finalize(void)` shuts down MPI
  - Place near end of code, after last MPI command

Environmental Inquiry

- `MPI_Comm_size(MPI_Comm comm, int *size)`
  - Find out number of processes
  - Allows flexibility in number of processes used in program
- `MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - Find out identifier of current process
  - 0 ≤ `rank` ≤ `size`-1

Message Passing: Send

- `MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - Send message of length `count` bytes and datatype `datatype` contained in `buf` with tag `tag` to process number `dest` in communicator `comm`
  - E.g. `MPI_Send(&x, 1, MPI_DOUBLE, manager, me, MPI_COMM_WORLD)`

Message Passing: Receive

- `MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - Receive message of length `count` bytes and datatype `datatype` with tag `tag` in buffer `buf` from process number `source` in communicator `comm` and record status `status`
  - E.g. `MPI_Recv(&x, 1, MPI_DOUBLE, source, source, MPI_COMM_WORLD, &status)`

Message Passing

- WARNING! Both standard send and receive functions are *blocking*
- `MPI_Recv` returns only after receive buffer contains requested message
- `MPI_Send` may or may not block until message received (usually blocks)
- Must watch out for deadlock

Deadlocking Example (Always)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
    int me, np, q, sendto;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (np%2==1) return 0;
    if (me%2==1) {sendto = me-1;}
    else {sendto = me+1;}
    MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD, &status);
    MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    printf("Sent %d to proc %d, received %d from proc %d\n", me,
        sendto, q, sendto);
    MPI_Finalize();
    return 0;
```

Deadlocking Example (Sometimes)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
    int me, np, q, sendto;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (np%2==1) return 0;
    if (me%2==1) {sendto = me-1;}
    else {sendto = me+1;}
    MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD, &status);
    printf("Sent %d to proc %d, received %d from proc %d\n", me,
        sendto, q, sendto);
    MPI_Finalize();
    return 0;
```

Deadlocking Example (Safe)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
    int me, np, q, sendto;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (np%2==1) return 0;
    if (me%2==1) {sendto = me-1;}
    else {sendto = me+1;}
    if (me%2 == 0) {
        MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
        MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD, &status);
  } else {
        MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD, &status);
        MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    }
    printf("Sent %d to proc %d, received %d from proc %d\n", me, sendto, q,
  sendto);
    MPI_Finalize();
    return 0;
}
```

Explanation: Always Deadlock Example

- Logically incorrect
- Deadlock caused by blocking `MPI_Recvs`
- All processes wait for corresponding `MPI_Sends` to begin, which never happens

Explanation: Sometimes Deadlock Example

- Logically correct
- Deadlock could be caused by `MPI_Sends` competing for buffer space
- Unsafe because depends on system resources
- Solutions:
  - Reorder sends and receives, like safe example, having evens send first and odds send second
  - Use non-blocking sends and receives or other advanced functions from MPI library (beyond scope of this tutorial)

Other useful MPI Commands

- MPI_Bcast(void* buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm, int ierror)
  - Broadcast to every process
- MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, int ierror)
  - Pull data from all processes in group
- MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, int ierror)
  - Similar to broadcast put only to a particular group
- MPI_Reduce(void* sendbuf, void* sendbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
  - Reduce to a singe value (min, max, etc)

Questions?

http://www.olcf.ornl.gov