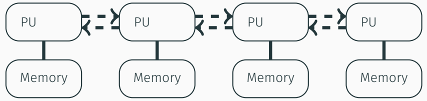
Introduction to MPI

Jeremy Iverson

College of Saint Benedict & Saint John’s University

background



- A standard for explicit distributed memory parallel computation.
- Many implementations available, both open-source and proprietary.

1. We are using an implementation of MPI called Open-MPI

execution model

- Uses the *SPMD* model of parallelism
  - All processes execute the same program.
    - Different processes carry out different actions by conditional execution of code based on processes' rank.
    - Processes can communicate with each other by sending explicit messages

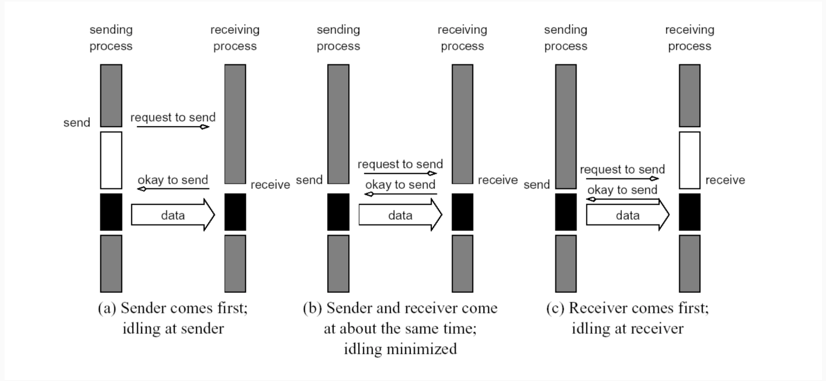
library api

- Requires inclusion of `mpi.h` header file.

- `MPI_Init()`
- `MPI_Finalize()`
- `MPI_Comm_size()`
- `MPI_Comm_rank()`
- `MPI_Send()`
- `MPI_Recv()`

point-to-point communication

- MPI uses *communicators* to organize processes. Processes can only communicate with other processes in the same communicator. The base communicator to which all processes belong is called `MPI_COMM_WORLD`.
- Programs can deadlock due to improperly ordered or unmatched point-to-point communications.


sending process
receiving process
send
request to send
okay to send
receive
data
(a) Sender comes first; idling at sender
(b) Sender and receiver come at about the same time; idling minimized
(c) Receiver comes first; idling at receiver

collective communication

- Represent regular *communication patterns* that are performed by parallel algorithms.
- Include groups of processes, not just two.
- Can be implemented to take advantage of underlying network characteristics and thus improve performance compared to simple point-to-point equivalents.
- Most parallel libraries provide functions to perform them (`omp parallel for reduction(+:sum)`)

communication patterns

```c
if (0 == rank) {
  for (int r = 1; r < p; r++) {
    MPI_Send(&x, 1, MPI_INT, r, 0,
      MPI_COMM_WORLD);
  }
} else {
  MPI_Recv(&x, 1, MPI_INT, 0, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

communication patterns

```
if (0 == rank) {
  for (int r = 1; r < p; r++) {
    MPI_Send(&x, 1, MPI_INT, r, 0,
      MPI_COMM_WORLD);
  }
} else {
  MPI_Recv(&x, 1, MPI_INT, 0, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

```
MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

communication patterns

```
if (0 == rank) {
  for (int r = 1; r < p; r++) {
    MPI_Send(rating + r * n * m, n * m,
      MPI_DOUBLE, r, 0, MPI_COMM_WORLD);
  }
} else {
  MPI_Recv(rating, n * m, MPI_DOUBLE, 0, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

communication patterns

```
if (0 == rank) {
  for (int r = 1; r < p; r++) {
    MPI_Send(rating + r * n * m, n * m,
      MPI_DOUBLE, r, 0, MPI_COMM_WORLD);
  }
} else {
  MPI_Recv(rating, n * m, MPI_DOUBLE, 0, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

```
MPI_Scatter(rating, n * m, MPI_DOUBLE,
  rating, n * m, MPI_DOUBLE, 0,
  MPI_COMM_WORLD);
```

communication patterns

```
if (0 == rank) {
  for (int r = 1; r < p; r++) {
    size_t const rn = (r + 1) * base > n
                    ? n - r * base : base;
    MPI_Send(rating + r * base * m, rn * m,
      MPI_DOUBLE, r, 0, MPI_COMM_WORLD);
  }
} else {
  MPI_Recv(rating, ln * m, MPI_DOUBLE, 0, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

communication patterns

```
if (0 == rank) {
  for (int r = 1; r < p; r++) {
    size_t const rn = (r + 1) * base > n
                    ? n - r * base : base;
    MPI_Send(rating + r * base * m, rn * m,
      MPI_DOUBLE, r, 0, MPI_COMM_WORLD);
  }
} else {
  MPI_Recv(rating, ln * m, MPI_DOUBLE, 0, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

```
MPI_Scatterv(...);
```