CMSC416: Introduction to Parallel Computing

Topic: Advanced MPI
Date: 02/15/2024

Waiting for MPI Requests:

- The **MPI_Request** object is opaque, resides in the MPI system memory, and is managed by MPI.
    - It is associated with a particular communication operation, and it links a posted non-blocking operation to its completion.
    - To use it, we declare an MPI_Request object and send a pointer to it as an argument to MPI routines.

Other MPI Calls:

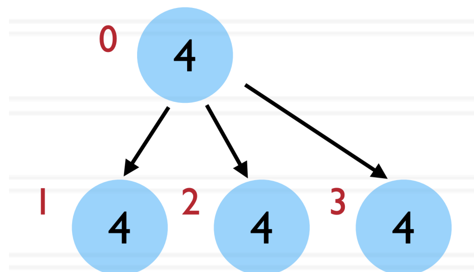
- **MPI_Test** returns immediately and sets the parameter flag to true if the parameter request has completed.
- **MPI_Waitall** waits for all requests in the parameter array to complete.
- **MPI_Waitany** waits for one or more requests in the parameter array to complete.

Point-to-Point Operations:

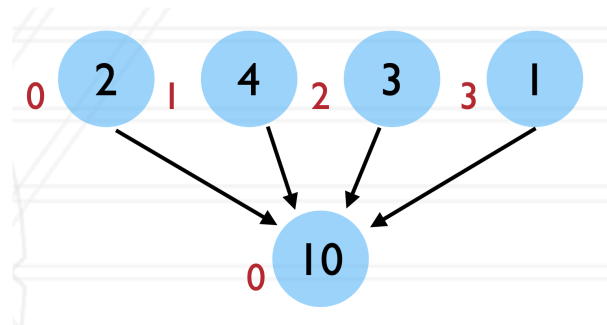
- **Point-to-point operations** are between two processes.

Collective Operations:

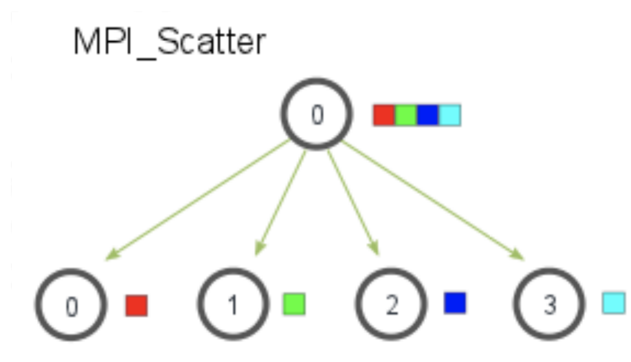
- **Collective operations** are between all processes in a communicator.
- **MPI_Barrier** blocks until all processes in the communicator have reached this routine.
    - It helps you synchronize across programs.
- **MPI_Bcast** sends data from the root process to all other processes in the communicator.
    - If the current process is the root, the buffer stores the data to be sent. Else, the buffer stores the data that has been received.
    - SPMD ensures that a single variable declaration can be used in both the root process and the receiving processes.
    - The implementation of this routine is abstracted from the user.



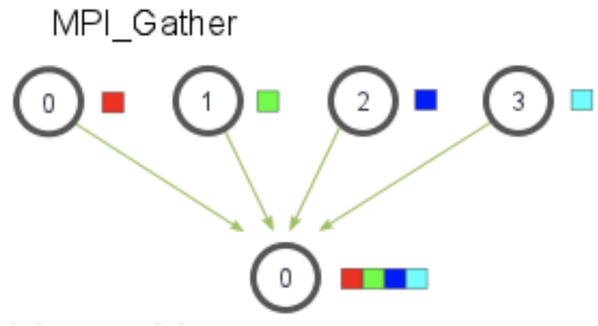
- **MPI_Reduce** collects data from all processes, aggregates it, and stores the result in the root process.
    - The function signature is **int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)**.
    - **sendbuf** is a buffer that stores the data to be sent, and it should be valid on all processes.
    - **recvbuf** is a buffer that stores the result, and it only needs to exist on the root process.
    - **op** is a predefined MPI operation (ex. **MPI_SUM**).



- **MPI_Allreduce** sends the result of the aggregation back to all processes rather than only to the root process.
- **MPI_Scatter** sends data from the root process to all other processes.
    - The function signature is **int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)**.
    - Based on **sendcount** and **recvcount**, MPI will split **sendbuf** and send different parts to different processes.
    - This function sends the same amount of data to each process.



- **MPI_Gather** gathers data from all processes and stores the result (without aggregating it) in the root process.



- **MPI_Scan** computes the scan (partial reductions) of data on a collection of processes[1].

Other MPI Calls:
- **MPI_Wtime** returns the time elapsed since epoch.
    - To compute a program's elapsed time, call this routine at the start and end of the program, then compute the difference.

```
{
  double starttime, endtime;
  starttime = MPI_Wtime();

  ....  code region to be timed  ...

  endtime = MPI_Wtime();
  printf("Time %f seconds\n",endtime-starttime);
}
```

Calculating the Value of π:
- To compute π programmatically, we divide a circle into tiny line segments and compute the integral over them. When this is done for a large number of iterations, π is approximated.
- The serial algorithm is as follows:

[1] https://www.mpich.org/static/docs/v3.3/www3/MPI_Scan.html

```c
int main(int argc, char *argv[])
{
    ...

    n = 10000;

    h   = 1.0 / (double) n;
    sum = 0.0;

    for (i = 1; i <= n; i += 1) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x * x));
    }
    pi = h * sum;

    ...
}
```

- The for loop approximates the integral. During each iteration, we calculate a partial value that is stored in x, which is then used in an expression that is added to a sum. After the for loop ends, this sum is then used in an expression that approximates π.
- This is an embarrassingly parallel problem. No loop iteration depends on any other loop iteration, and therefore, we can split the loop iterations across several processes to parallelize this serial algorithm. To divide the work across processes, we can use a **round robin strategy** (ex. 0, 1, 2, 3, 0, 1, 2, 3, …) or a **block division strategy** (ex. 0, 0, 0, 0, 1, 1, 1, 1, ….).
- The parallel algorithm is as follows:

```c
int main(int argc, char *argv[])
{
    ...

    n = 10000;
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    h   = 1.0 / (double) n;
    sum = 0.0;

    for (i = myrank + 1; i <= n; i += numranks) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x * x));
    }
    pi = h * sum;

    MPI_Reduce(&pi, &globalpi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    ...
}
```

- This parallel implementation uses a round-robin strategy to divide the work (ex. process 1 performs loop iterations 2, 6, 10, …).

- The **MPI_Bcast** operation is not required but demonstrates that if only one process creates the number of iterations n, it can broadcast it to all other processes.
- The **MPI_Reduce** operation is necessary to aggregate the partial sums computed by each process.
    - This ensures that for all inputs n, the serial and parallel algorithms compute the same result.
    - The root process stores a variable **globalpi**, which stores the result of the reduction. **globalpi** is then output.