MPI

Michael Puthawala

With lots of things taken from

https://computing.llnl.gov/tutorials/mpi/

https://computing.llnl.gov/tutorials/parallel_comp/
Parallel vs serial

Serial process

Parallel process
Shared memory vs Distributed memory

Uniform Memory Access (UMA)

Distributed Memory
Distributed memory model

- In the distributed model, processes churn away on their work, and only communicate to each other over the network.
- This leads to the natural question, how should the separate processes coordinate?
- Enter, **Message Passing Interface (MPI)**
- MPI provides a specification for how messages can be passed between independent processes.
Distributed memory model

- In the distributed model, processes churn away on their work, and only communicate to each other over the network.
- This leads to the natural question, how should the separate processes coordinate?
- Enter, Message Passing Interface (MPI)
- MPI provides a specification for how messages can be passed between independent processes.
Distributed memory model

- In the distributed model, processes churn away on their work, and only communicate to each other over the network.
- This leads to the natural question, how should the separate processes coordinate?
- Enter, Message Passing Interface (MPI)
- MPI provides a specification for how messages can be passed between independent processes.
Distributed memory model

- In the distributed model, processes churn away on their work, and only communicate to each other over the network.
- This leads to the natural question, how should the separate processes coordinate?
- Enter, Message Passing Interface (MPI)
- MPI provides a specification for how messages can be passed between independent processes.
Example MPI program

```c
int main(int argc, char *argv[]) {
    int numtasks, rank, len, rc;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    rc = MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(hostname, &len);
    printf("Number of tasks= %d My rank= %d Running on %s\n", numtasks, rank, hostname);
    MPI_Finalize();
}
```
Example MPI program

```c
int main(int argc, char *argv[]) {
    int numtasks, rank, len, rc;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    rc = MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(hostname, &len);
    printf("Number of tasks= %d My rank= %d Running on %s\n",
        numtasks, rank, hostname);
    MPI_Finalize();
}
```
Example MPI program

```c
int main(int argc, char *argv[]) {
    int numtasks, rank, len, rc;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    rc = MPI_Init(&argc, &argv); // Initializes the MPI environment
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(hostname, &len);
    printf("Number of tasks= %d My rank= %d Running on %s\n",
           numtasks, rank, hostname);
    MPI_Finalize();
}
```
Example MPI program

```c
int main(int argc, char *argv[]) {
    int numtasks, rank, len, rc;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    rc = MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks); // returns size of network
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(hostname, &len);
    printf("Number of tasks= %d My rank= %d Running on %s\n", 
        numtasks, rank, hostname);
    MPI_Finalize();
}
```
int main(int argc, char *argv[]) {
    int numtasks, rank, len, rc;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    rc = MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // returns this process' rank
    MPI_Get_processor_name(hostname, &len);
    printf("Number of tasks= %d My rank= %d Running on %s\n", numtasks, rank, hostname);
    MPI_Finalize();
}
Example MPI program

```c
int main(int argc, char *argv[]) {
    int numtasks, rank, len, rc;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    rc = MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(hostname, &len); // Identifies the work station
    printf("Number of tasks= %d My rank= %d Running on %s\n", numtasks, rank, hostname);
    MPI_Finalize();
}
```
Example MPI program

```c
int main(int argc, char *argv[]) {
    int numtasks, rank, len, rc;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    rc = MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(hostname, &len);
    printf("Number of tasks= %d My rank= %d Running on %s\n",
            numtasks, rank, hostname);
    MPI_Finalize(); // Terminates the MPI environment
}
```
Example MPI program

```c
int main(int argc, char *argv[]) {
    int numtasks, rank, len, rc;
    char hostname[MPI_MAX_PROCESSOR_NAME];

    rc = MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(hostname, &len);
    printf("Number of tasks= %d My rank= %d Running on %s\n",
           numtasks, rank, hostname);
    MPI_Finalize();
}
```

Communication between processes

- So how do processes send messages to each other?
- They call any number of functions, the 6 most common of which are:
  - MPI_Send
  - MPI_Recv
  - MPI_Bcast
  - MPI_Scatter
  - MPI_Gather
  - MPI_Reduction
Suppose that I wanted to send the number 13 from process 0, to process 1, that might look like:

Note that the data is sent in a buffer, i.e. an array of ints. If you wanted to send more data, just increase the size of the buffer.

```c
int buf[1];
MPI_Status status;
if (rank == 0) {
    buf[0] = 13;
    MPI_Send(buf, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
} else if (rank == 1) {
    MPI_Recv(buf, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
} // MPI_Send (&buf,count,datatype,dest,tag,comm)
// MPI_Recv (&buf,count,datatype,source,tag,comm,&status)
```
When distributed parallelization the answer?

- If you had a problem that can be split up into many smaller programs, each of which requires a lot more computation than communication between problems.

- Examples include:
  - Stochastic integration
  - 3D projections for graphics.
  - Orbital integrals

- Such problems are called “embarrassingly parallel”
My project

- My project this quarter was to parallelize the orbital integral computation.

- Specifically, given a set of basis functions \( \{ \phi_i \}_{1}^{n} \), we had to compute
  \[
  \int \phi_i(y)\phi_j(y) \int \frac{\phi_k(x)\phi_l(x)}{|x - y|} \, dx \, dy = \langle \phi_i \phi_j, V \phi_k \phi_l \rangle \text{ for } i, j, k, l = 1, \ldots, n
  \]

- We don’t actually compute the inner integral, instead we solve the equation
  \[
  \Delta \Phi_{kl} = -4\pi \phi_k \phi_l. \text{ This takes comparatively much longer than computing}
  \int \phi_i \phi_j \Phi_{kl} \, dy
  \]
The problem is naturally symmetric

1. \( \langle \phi_i \phi_j, V \phi_k \phi_l \rangle = \langle \phi_j \phi_i, V \phi_k \phi_l \rangle = \langle \phi_i \phi_j, V \phi_l \phi_k \rangle \)

2. \( \langle \phi_i \phi_j, V \phi_k \phi_l \rangle = \langle \phi_k \phi_l, V \phi_i \phi_j \rangle \)

Eliminate redundancies of the first kind by requiring that \( i \geq j, k \geq l \)

Eliminate redundancies of the second kind by requiring that \( i + (n + 1)j \geq k + (n + 1)l \)

Combined, these allow you to reduce the work by about a factor of 8.
I parallelized the task, of computing $O_{ijkl} = \langle \phi_i \phi_j, V \phi_k \phi_l \rangle$ by assigning worker nodes to compute $\int \frac{\phi_k(x) \phi_l(x)}{|x-y|} dx$ once for a given $l^*, k^*$ and then they computed $O_{ijk^*l^*}$ for each $1 \leq i,j \leq n$.
if (rank == ROOT) {
    for (size_t k = 0; k < kRange; k++) {
        for (size_t l = 0; l <= k; l++) {
            // Wait for a worker to request some work. When a request is received, perform a handshake, and send the worker their k,l
        }
    }
    for (size_t i = 0; i < numtasks - 1; i++) {
        // Once all of the work is done, tell each worker to halt work
    }
} else {
    while (send_request) {
        // Ask the root node for more work. If a k,l is received, compute all O_i,j,k,l for every i and j, given k,l
        // Otherwise stop.
    }
}
Results, for n = 4

- Total time for 1 workers, 52739(ms)
- Total time for 2 workers, 26425.9(ms)
- Total time for 4 workers, 16563.5(ms)
- Total time for 8 workers, 16076.2(ms)
- Total time for 16 workers, 13980.8(ms)