CSCI 6360: Parallel Computing Lecture Summary - 4

Anirban Das (dasa2@rpi.edu)

January 26, 2018

Explanation of the Master-Worker implementation of Matrix Multiplication using MPI_send/ MPI_recv function pair and the factors limiting its performance

This code implements a parallel way of matrix multiplication using MPI and master-slave (worker) architecture. There is a single master MPI process, denoted by `taskid == 0`. Workers are denoted by `taskid > 0`. The same program contains code block for the master and workers, since the same program is replicated in all the nodes, and the code blocks are executed based on conditional checking on `taskid`.

After initializing the MPI environment and setting the necessary `comm` variables, the **MASTER** has the job of diving the matrix data into chunks and sending them to **WORKERS** for execution. The dimensions of matrix A, B, C are already defined as `NRA, NCA, NCB`. The **MASTER** knows the number of workers from the value of `numworkers`. It distributes the number of rows of `a` more or less equally among all the workers, as the following: first uses `MPI_Send` to send the `offset`, number of rows 'rows' and the actual rows from matrix `a` and the whole `b` matrix to current worker, then updates the offset, moves on to the next worker. It then waits for the result from all the workers using `MPI_Recv`. This is a blocking communication, and the **MASTER** waits until receive from all **WORKERS** is complete. Then it reconstructs and prints the result.

The **WORKERS** receives their chunk of rows of `a` matrix, and `b` from the **MASTER** using `MPI_Recv`. It then performs the multiplication and send the corresponding rows of the `c` matrix to the **MASTER**.

The workflow between **MASTER-WORKER** can be thought of a SIMD paradigm where all **WORKER** nodes received same set of instructions but performing on different sets of data.

Factors limiting the code's performance:

1. The `MPI_Send`, `MPI_Recv` are blocking calls, hence there is always a potential for deadlock.
2. The master sends out the chunks of data in sequential order, and since the `MPI_Send` is blocking, the master will not proceed to send data to next worker unless the current one finishes. This by default causes the other nodes to sit idle, wasting CPU cycles, moreover the situation worsens if the send functions require a long time to finish.
3. Different workers will finish their computation in different wall clock time, but the master is waiting for them in a sequential order, might cause a huge delay. For example, if worker with `taskid=1` finishes the last, then the whole computation is delayed the most.
4. The master node just partitions the data on the worker node, then sits idle, without assisting computation. Depending on the system, even one more node helping in the computation might decrease latency.
5. No fault tolerance, if any 'node' goes corrupt, the result is fallacious.