CMSC416: Introduction to Parallel Computing

Topic: Advanced MPI + Performance Modeling, Analysis, and Tools
Date: February 20, 2024

Protocol for sending a message

Depending on the size of the message, MPI will use a certain protocol to send the message. They are eager, rendezvous, and short protocols.

The eager protocol is used for messages with a small amount of data. It assumes the receiving process can store that information.

The rendezvous protocol doesn't assume the receiving process has enough memory. It does a handshake (they agree on the buffer size) with the receiving protocol, then sends the message after agreement. The user doesn't need to create this, this is implemented within MPI and HPC.

The short protocol is used for messages with a very small amount of data. It sends the data in the message envelope. The message envelope contains all the command line argument information like rank, size, and tag packed together and sent. Usually, when larger amounts of data are sent, the message envelope and data are sent separately. When the data is small enough, MPI run time can use the short protocol and send the data within the message envelope.

MPI send will use lower-level routines to make the actual send call.

The figure on Slide 18 shows how MPI sends a message. First, MPI runtime will pack the data and create the message envelope the send to the Network Interface Card (NIC). This component is physically connected to the computer. From there, the data will be sent to the network. On the network, the data will have to go through several switches (or routers), so the message is copied several times. Finally, the message will reach the destination NIC connected to the destination node. MPI runtime will copy the data from the node to the local memory and then the receiving node will notify the runtime that it has received the data.

Other MPI send modes

MPI has three other send calls: MPI_Bsend, MPI_MPI_Ssend, and MPI_Rsend.

Weak versus Strong Scaling

Strong and weak scaling are two concepts used to understand how a program runs.

The goal of strong scaling is that with a fixed total problem size when we run on more processes, the total time would go down exponentially. An example situation would be sorting n numbers on 1 process, then 2 processes, then 4 processes, etc. The time should be going down as we use more processes.

The goal of weak scaling is that with a fixed problem size, as we increase the total problem size with more processes, the time should stay constant. The problem size per process should remain the same. An example situation would be sorting n numbers on 1 process, then sorting 2n numbers on 2 processes, then 4n numbers on 4 processes, etc. We want the plot to look like a horizontal line to show that the ratio between them stays constant.

Amdahl's Law

If you parallelize some part of the program, the serial parts become a sequential bottleneck for the whole program. When looking at the formula in Slide 4, the parallelized portion is f/p, where the serial portion is (1-f), where f represents the fraction of the code that can be parallelized. As p, the number of processes, goes to infinity, that fraction will approach 0, so the overall fraction will be 1/(1-f). This would be the time for the most efficient version of the program.

Performance Analysis

Performance analysis is the process of studying the performance of your code and determining where performance issues occur. There are many reasons for some parts being slow, including serial performance (non-parallelized serial code), serial bottlenecks in parallel (serial code that is being run in parallel), and communication overheads between processes.

There are two main methods to conduct performance analysis: modeling the performance and using measurement tools on the actual performance. We can use time complexity to get an analytical sense using the data size and number of processes. We can also use Isoefifciency analysis (Scalability analysis). We can also model different parts of the program (computation, communication I/O) and analyze their performances. For example, we can use LogP and alpha-beta model. Another approach would be using empirical performance analysis by conducting experiments and analyzing the data.

Parallel prefix sum

The goal of prefix sum is that for each number in the list, we sum all the numbers before it with the current one and set that to its current spot. When we parallelize the prefix sum computation, we communicate with the neighbor processes, and through each phase, we communicate with the neighbor of stride 1 away, then 2 away, then 4 away, etc. This would result in log p phases, where p is the number of processes.

Parallel prefix sum for n >> p

If we have n elements and p processes, we assign a block of size n/p elements to each process. First, we compute the prefix sum on each block locally (this will be a serial computation). The time complexity for this would be O(n/p) because we need to go through every element in the block. Then, we do the parallel algorithm with partial prefix sums. During each phase, the processes are sending data to the neighbor process on the right.

The number of calculations in each phase is n/p because when we get the single data value from the left neighbor, we add it to each element in the block. So, the total number of calculations log(p) * n/p (number of phases times the calculation time per block).

For communication, in each phase of each process, it is sending 1 call to its neighbor. So, the total communication time would be log (p) * 1 * C (C is the constant amount of time needed for the call) for each process.

Modeling communication: LogP model

This model is used to model the communication in the network. There are several variables needed:

L represents the latency or delay, which is how long it takes for a message to travel on the network.
O represents the overhead, which is when sending a message from the processor, the amount of time the processor is busy in communication.
G represents the gap, which is the time between successive sends and receives. This also depends on the bandwidth (1/G). Bandwidth represents at what rate can you remove data from the network.
P represents the number of processes.

Using these variables, we can try to model the amount of time to send a message.

Alpha + n * beta model

This model focuses on two costs when sending a message. There is a fixed start-up cost for each message (latency), which is represented as alpha, and the bandwidth term (how fast the network is), which is represented as beta. N represents the size of the message. This model focuses on the amount of time to send a message of that size across the network. If we want to find the total time for several messages, we can find the time for one message, and then multiply by the number of messages.

Isoefficiency

Isoefficiency is the ability of a program to be able to maintain a certain level of efficiency by increasing the total problem size. It creates a relationship between the total problem size and the number of processes. It uses speedup and efficiency equations to derive another equation to determine isoefficiency.

Speedup and efficiency

There are two equations that are needed for the isoefficiency calculation.

Speedup is the ratio of time spent on one process (t1) over the time spent on p processes (tp). Suppose the time spent on one process is 100 seconds and the time spent on 4 processes is 25 seconds, then the speedup would be 100/25 = 4.

Efficiency is similar to speedup, but instead looks at speedup's value and divides it by the number of processes used. This score would be in the range of 0 to 1 (1 would be perfect efficiency). The goal is to have a high efficiency value.

Efficiency in terms of overhead

In a sequential program, the code will be intentional and useful, so the time will not be wasted. In a parallel program, however, there were will some MPI calls to be sent, which adds to the program's overhead (anything that is needed to make the code run in parallel is part of the program's overhead).

The total time spent across processes can be broken down into separate parts: useful computation (elapsed time for if the code was serial) and overhead (all of the extra computation involved in parallel programs, communication between processes, and idle time (receiving processes have to wait for the message) that would be needed to make the program run in parallel). For example, when looking at the parallel prefix sum problem, if we were to do this sequentially, there would be O(n) additions. In parallel, there are O(n/p) additions per log(p) phases over p processes. So, this would be O(n * log(p)) additions. In this case, the useful computation would be in O(n) part and the overhead would be in O(n * log(p)).

The formula for efficiency is shown in Slide 13, where t1 represents the useful computation time and t0 is the overhead time. The total time is the sum of these two values. When looking at the ratio in the denominator of the equation, if t0 is positive (there is overhead), we cannot achieve perfect efficiency.

Isoefficiency function

If we want the isoefficiency to be constant, then t0/t1 has to be constant. This can be represented using K. So, the formula would be t0 = K * t1. This shows the relationship between

the overhead and the sequential computation. We want to make sure that as we increase the general computation, the overhead should scale appropriately and not take over.

Isoefficiency analysis

Now let's look at Isoefficiency analysis on Game of Life 1D decomposition and 2D decomposition.

When looking at the 1D diagram in Slide 15, since the dimensions of the board are square_root(n) by square_root(n), then each process will look at square_root(n)/p. So, each process will have a computation of n/p because square_root(n)/p rows * square_root(n) elements per row. This value would be t0. In terms of communication, we are sending 2 messages per process. The total volume of data being sent is a good approximation of how much time is sent for communication, so in this case, it would be 2 messages * square_root(n) per process since we are sending 2 rows of data. This value would be t1. The ratio would then be t0/t1 = (2 * p)/square_root(n).