Date: Feb 20, 2024

Finishing up from last lecture

Protocols for sending messages

The algorithm used by the MPI runtime to send messages differs based on the size of the message being sent. There are three protocols used: **Eager**, **Rendezvous**, and **Short**.

Eager is used for small messages. The sender assumes that the receiver has enough space to store the incoming message and immediately sends it.

In contrast, **Rendezvous** does not assume the receiver has enough space to store the message. Instead, a handshake is first performed with the receiver that informs them of the incoming message's size. Once the receiver confirms that a buffer of that size has been posted, the sender will send the message. For this class we can assume that the handshake always completes.

Short is similar to eager and is used for very short messages. The difference is that the data is sent in the message envelope instead of in a separate send. The message envelope contains MPI runtime information for the send such as the datatype, tags, etc. This eliminates the separate sending of data from Eager and has potential performance considerations.

A similar performance consideration exists for Eager/Rendezvous, specifically the maximum size limit before MPI switches from Eager to Rendezvous, as Eager is faster than Rendezvous. This size limit for the Eager protocol can be changed by the user through environment variables that set the size limit.

Note that you as the user do not need to worry about calling these protocols. These "under the hood" protocols for sending messages are all handled by the MPI runtime, which will call the underlying libraries necessary for sending messages (i.e., UCX or PSM). The data to be sent is copied onto the Network Interface Card (NIC), which then copies it onto the network. The data goes through routers before reaching the destination NIC and is copied into the destination node's buffer, at which point MPI is notified that the buffer is ready for use.

Additionally, other MPI send modes exist but they won't be used in this course.

Performance modeling, analysis, and tools

Strong scaling vs. weak scaling revisited

To review, **strong scaling** is when the total problem size is fixed but the number of processes increases. An example of this is sorting *n* numbers on 1, 2, 4… processes, where *n* remains the same. The ideal graph for strong scaling has the computation time decreasing as the number of processes increases.

On the other hand, **weak scaling** has a fixed problem size per process, but the total problem size increases. Using the game of life as an example:

- 4 processes compute over a 256 x 256 board
- 8 processes compute over a 512 x 256 board
- 16 processes compute over a 512 x 512 board

Notice that the amount of cells each process handles does not change, we are just increasing the size of the problem. The ideal graph for weak scaling should be a horizontal line, where computation time stays constant even as the problem size and number of processes increases.

Amdahl's law revisited

Speedup is limited by the serial portion of the code (called the serial "bottleneck"). Assuming a fraction *f* of the code is parallelized on *p* processes, the speedup is calculated as:

$$\frac{1}{(1-f)+f/p}$$

Performance analysis

Performance analysis is the process of studying parallel code performance to identify why performance might be slow. Causes may include inefficient serial performance, serial bottlenecks when run in parallel, and/or communication overheads.

There are a variety of analysis methods we can use for performance analysis. Using **algebraic formulae** we can perform time complexity analysis using data size *n* and number of processes *p* (an example of this using parallel prefix sum follows this section). Another method is **scalability/isoefficiency analysis** (will be discussed later in the notes). Various ways to **model performance of operations** (like input and output) exist, such as the LogP model or alpha-beta model (also discussed later in the notes). And finally, **empirical performance analysis using tools** can be done to identify areas of improvement.

Example: **time complexity analysis** on the parallel prefix sum problem

Assume there are *n* total elements and *p* processes, where *n* >> *p*. This means each process is assigned a block of *n/p* elements.

The initial prefix sum that each process calculates on its own block requires *n/p* calculations (iterate over all *n/p* elements in the block and sum them together).

In the parallel algorithm, there are *log(p)* phases, as that is the number of strides needed to complete the algorithm. Each phase has *n/p* calculations, where the received prefix sum is added to all *n/p* elements in the block. Therefore each process does a total of $log(p) * (n/p)$ calculations. Note that we are assuming unit time for calculations - in a more detailed analysis, each calculation will need to be multiplied by a constant *C* for the time each operation takes. Lastly, each process sends a maximum of 1 message per phase, which takes *C* time to send, so the communication cost is $log(p) * 1 * C$.

Since each process is load balanced in this scenario, each process should finish around the same time, so the total time is the maximum time across all processes.

Communication can also be modeled using the **LogP model**, a simple model for communication on an interconnection network. **L** is the latency or delay on the network, **o** is the overhead (i.e., the processor is busy in communication), **g** is the gap (i.e., between successive sends/receives), and **P** is the number of processors or processes.

An even simpler model for communication is the **alpha + n * beta model** (or **alpha-beta model**). This calculates how long it takes to send a message across a network using the following equation: $T = \alpha + n * \beta$, where α is the latency (defined as the startup time to send a message on the network - this is different from the LogP definition of latency!), *n* is the size of the message being sent, and β is the bandwidth of the network.

Isoefficiency

Isoefficiency calculates the relationship between the problem size and the number of processors to maintain a certain level of efficiency. In other words, it answers the question "At what rate should we increase the problem size with respect to the number of processors to keep efficiency constant?" To ensure we can understand this analysis, let's review speedup and efficiency.

Speedup is the ratio of computation time on one process to the computation time on *p* processes. The formula for it is t_1/t_p. For example, if a single process takes 100 seconds to complete, and 4 processes take 25 seconds, the speedup is 100/25 = 4. But if 4 processes take 40 seconds, then the speedup is only 100/40 = 2.5. Efficiency is the speedup per process. The formula for it is $t_1/(t_p * p)$. The efficiency is between 0 and 1, where 1 is perfect efficiency (a good goal is 0.8 or 0.9). Following the above example, the efficiency when 4 processes take 25 seconds is 100 / (25 * 4) = 1, but if 4 processes take 40 seconds, then the efficiency is only 100 / (40 * 4) = 0.625.

Now let's look at efficiency in terms of overhead. The total time in all processes is equal to the time spent doing useful computation plus the overhead (extra computation + communication + idle time). Expressed another way, $p * t_p = t_1 + t_o$, where t_1 is the useful computation time and t_o is the overhead. If we look at parallel prefix sum, the total time is $p * O(n/p) * log(p) = O(n) * log(p)$ total time, which is more than the *O(n)* time from just the serial prefix sum. Now that we have an alternative expression for total time, we can rewrite the efficiency formula as follows: $t_1/(t_p * p) = t_1/(t_1 + t_o) = 1/(1 + (t_o/t_1))$. Given this, we can derive the isoefficiency function, as efficiency is constant if and only if t_o/t_1 is constant (*K*). $t_o = K * t_1$. What this means is that overhead should be a constant factor of the useful computation if the efficiency is to remain constant.

Example: **Isoefficiency analysis** on parallel 2D stencil problem

Given *n* elements and *p* processes, let's use isoefficiency analysis to compare 1D decomposition to 2D decomposition.

1D decomposition

Since there are *n* elements, the board is a $\sqrt{n} * \sqrt{n}$ board. Each process gets $\sqrt{n}/p$ rows, which equates to *n/p* elements per process. As such, each process does *n/p* computations per iteration - this is the value for t_1. Additionally, each process sends a maximum of $2\sqrt{n}$ elements to its neighbors - this is the value for t_o. Therefore, $t_o/t_1 = (2\sqrt{n})/(n/p) = (2p)/\sqrt{n}$

2D decomposition

Will be completed next class