CMSC416: Introduction to Parallel Computing

Topic: Performance Issues
Date: April 4th, 2024

Performance Metrics

There are multiple ways to record performance including: time to solution, time per step, science progress, and FLOPS. Time to solution looks at the time it takes for the entire program to reach a solution. This metric is used in Fourier solutions. Time per step looks at a smaller or more specific section of the code which results in a shorter total time than time to solution. An example for science progress could be in epidemic diffusion where we are trying to measure how fast a disease can spread in X number of days. FLOPS just mean floating point operations per second. This is useful in programs which have a lot of computation happening. The more FLOPS a program has, the faster it tends to be since it can compute more per second.

Serial Code Performance

Serial code performance issues can be identified using performance tools such as HPC toolkit. One possible solution to this type of issue is to try to maximize data reuse. For example, think back to matrix multiply with blocking. Another possible solution could be to optimize your floating-point calculations. An example of this could be replacing a division by 5 with a multiply by .2. Another example could be saving the decimal of a square root that you will use frequently as a variable instead of calculating it each time. The last possible solution is to minimize data movement.

Communication Performance Issues

Communication performance issues can be caused by overhead and grainsize. Fine grain involves a lot of tiny messages, and coarse grain is fewer, but larger messages being sent. There can also be communication performance issues if there are increased amounts of communication with more processes and threads. Frequent global synchronization can also be an issue. For example, if a program has too many barriers, this will slow down the performance. A possible solution to fix overlap between communication and computation is to use non-blocking calls. This allows for the program to continue computation without stopping for communication.

Critical Path

A critical path is a long chain of operations with consecutive dependencies. This can occur across multiple processes. A good goal to aim for is to identify the length of the path. Once this path is identified, we should try to eliminate it completely if that is possible. If this is not possible, we could try to shorten the path by having less communication or processes involved and by removing work.

Serial Bottlenecks

Serial bottle necks occur only on one process. An example of a serial bottleneck is having one process responsible for input and output. While this is all that we have seen in class so far, sometimes this may not be ideal. For example, if we have 128 processes that we need to send information to, this will take some time to execute the communication. A solution to this is to try and parallelize the I/O. A way to do this is to have each process read in a different piece of data. Some other serial bottlenecks include reducing to one process and then broadcasting and having one process responsible for assigning work to others. Similar to the I/O solution, the best way to counteract serial bottlenecks is to try and parallelize as much as possible. Recall Amdahl's law which states that speedup is limited by the serial portion of the code. Therefore, the more we parallelize, the less serial bottlenecks we should have.

Performance Variability

Performance variability can be caused by OS noise and contention for shared resources. OS noise happens when a computation is interrupted by the OS. This can happen from a daemon. Some daemons need to occur such as the OS checking the status of the hardware. Other daemons such as applications opening when you turn on the computer are avoidable. Daemons also have different execution times depending on what the daemon is for. To reduce OS noise, we could turn off unnecessary daemons or try to reduce the number of daemons. The reason why we care to do this is because these interruptions will impact the performance predictability. Performance variability can cause increased wait times in job queues and take more time to complete simulations. Imagine logging on to execute a program one day and it takes 1 minute and then you try to execute the next day and it takes 10 minutes to execute. This can cause increased energy usage which usually leads to an increase in cost. The software development cycle can be affected by performance variability because it is now more difficult to debug performance issues. Without proper intuition, the programmer may not know if it is the actual code that is not running as efficient as it could, or if it is the machine not running the code as efficient as it could.