CMSC416: Introduction to Parallel Computing

Topic: Performance Issues
Date: April 4th, 2024

Performance Metrics

People use the following metrics to measure how fast the program runs.

- Time to solution: Time the whole program. Sometimes it might take a very long time for some physics or chemistry simulation to reach the solution, so instead people look at time per step.
- Time per step (iteration): If the program is designed with multiple iterations (steps), it is more efficient to record the time per iteration. For example, run 10 iterations and take the average.
- Science progress: Measure the specific quantity of interest per unit time, usually in science fields. For example, how much time does it take for a biomolecule to be simulated, or in epidemic simulations, how the number of days that can be simulated in a computer time (second).
- Floating point operations per second (flop/s): Many HPCs are doing floating point intensive jobs for most of their lives. Then how many floating-point operations per second will be a good metric for their performance, like how intensive the floating-point operation is, or is the program bounded by memory.
- Comparing multiple data points: Theoretically we use speedup and efficiency to compare the performance of different programs. Empirically we run the programs with different number of processes and compare the time to see whether we are scaling well.

Best performance

The following parameters are called by the professor as peak advertised numbers because they are usually given by the producers to advertise how good their products are. The numbers are theoretical upper bounds, in practice, reaching 20%-40% of them can already be reasonable for scientific computing jobs. For NN tasks involving simpler computations, 60% or more can be reached.

- Peak flop/s: For example, the world's top 2 HPC has the parameters:
  (Rmax = maximum performance, Rpeak = theoretical maximum performance)
  Frontier: Rmax = 119400 flop/s , Rpeak = 167982 flop/s
  Aurora: Rmax = 58534 flop/s, Rpeak = 109000 flop/s
  Even the top 2 super computers can only have 60% performance compared to the theoretical bounds.
- Peak memory bandwidth
- Peak network bandwidth

Why can't we reach the theoretical bounds? Because there are performance issues. Before knowing the issues, we first need to know what is happening in a program.

- Integer operations
- Floating-point operations
- Conditional instructions: if else statements.
- Loads/stores: load data from memory to registers.
- Data movement across the network (communications of messages + I\O): For example, in cuda, sending messages from GPU to GPU, or from CPU to GPU. In MPI, communications across processes. I/O: file reads/writes.

Best performance

After knowing what happens inside the computer, now we are going to see what issues we might meet.

- Serial code performance issues: Every problem that happens in the serial part that slows down the program.
- Load imbalance: Processes which run fast need to wait for those run slow.
- Communication issues / parallel overhead: More processes used will cause more communication overhead. I/O overhead can also happen.
- Algorithmic overhead / replicated work: The design of parallel algorithm might let each process do more additions (operations) than the serial case and thus produce overhead.
- Speculative loss: This is also replicate work but is labelled differently here in terms of classification. Speculative executions are something related to hardware which can help the programs perform well. If the program really needs it, then the program performs better. However sometimes the program doesn't need it, then it becomes excessive work.
- Critical paths: When you see a thread (process) is slow, it might not be the problem in itself. For example, it waited for another thread to send messages for a long time. Critical path just describes the long path of dependencies across threads/processes.
- Insufficient parallelism: Program has too many sequential part but very little parallel part.
- Bottlenecks: For example, all processes need to wait for the root to load and distribute data.

Serial code performance issues

People can use many performance tools to check the serial code performance issues. They can provide people with hardware metrics like floating-point operation, caches to tell you in different part of the hardware (cache, memory, floating-point unit), whether you are using them properly or not.
Solutions:

- Minimize data movement: Minimize the movement in data hierarchy. Bring data into registers and use them multiple times.
- Maximize data reuse: For example, using blocking in matrix multiplication.
- Optimize floating-point calculations: For example, calculating *0.2 will be faster than calculating /5.

Communication performance issues

- Overhead and grainsize: Decide how to divide data into processes.
  Corse grain: Send few large messages. This might cause overhead.
  Fine grain: Send many small messages. This needs better bandwidth.
- No overlap between communication and computation: The advantage of Isend and Irecv is that the processes can utilize the waiting to do computation. If a program does not have such techniques, then it is not efficient.
- Increasing amounts of communication as we run with more processes/threads: If the number of processes increase, it will be less efficient to let a process to still communicate with all the processes. A better way is to communicate within smaller groups of processes.
- Frequent global synchronization: Each global synchronization function will force every process to wait, e.g., Reduce. If these functions are called frequently, the waiting time will be long.

Critical paths

People want to find out what is the first bottleneck in a critical path, but this is difficult if the path is long.
Solutions:

- Eliminate the critical path with as few dependencies as possible, but usually this is not practical because communications are needed.
- Shorten the critical path: Try to compute asynchronously by using Isend and Irecv.
- Removing work: Optimize the number of processes involved.

Serial Bottleneck

- Use only one process for file I/O. Root 0 read and send messages to 999 processes. This can be solved by using parallel I/O.

Performance variability

While running the same executable many times, it performs differently.
This can lead to several problems:

- More time to complete science simulations: Usually it takes 3 months to complete, but now it takes 5-8 months, but people don't know.
- All the users are waiting in job queues. A slow task will force all the following jobs to be delayed.
- Inefficient use of machine time allocation: The number of jobs per computer time decreases, then some concern of energy usage or costs need to be considered.
- Debugging performance issues: If people revise the code and want to see an increase in performance, but there is no because of variability, then they will be confused.
- Cannot quantify the effect of various software changes on performance. For example, to check the changes made by using new software to calculate matrix multiplication.
- There is no reliable estimate time for running executables, so it is difficult to estimate time for a batch job or simulation.

Sources of performance variability

- Operating system (OS) noise/jitter: As OS runs, some daemons must tun. When daemons are running, all the other executables must stop and wait for them to finish. For example, a daemon checking whether there are new messages to process.
  If looking at the graph returned by measuring tools Fixed Work Quanta and Fixed Time Quanta, there are equal-space gaps between execution time. This phenomenon is just caused by daemon because the time for daemon to run is constant.
- Contention for shared resources