Performance Analysis

Shuhao Zhang

Nanyang Technological University

shuhao.zhang@ntu.edu.sg

June 4, 2024

Recall: Testing for Concurrency

- Testing for correctness
  - Safety: nothing bad ever happens
  - Liveness: something good eventually happens (e.g., no deadlock)
- Testing for performance
  - Throughput: the rate at which a set of concurrent tasks is completed
  - Responsiveness: the delay between a request and completion of some action
  - ...

Performance: Two Viewpoints

A Claim

"Program X is **better** than Program Y"

- Better = Lower Response Time
  - The duration of processing one input is shorter
  - E.g., the time of performing $a = a + b$ in program X is shorter than in program Y.
- Better = Higher Throughput
  - More work can be done in the same duration
  - E.g., within the same amount of time, program X performs $a = a + b$ for more times than program Y.

Response Time in Sequential Systems

Response time of a program A:

- User CPU time: time CPU spends for executing program
- System CPU time: time CPU spends executing OS routines
- Waiting time: I/O waiting time and the execution of programs because of time sharing

We focus on analyzing and reducing User CPU time here.

waiting time: depends on the load of the computer system.
system CPU time: depends on the OS implementation.

User CPU time

- $Time_{user}(m) = N_{cycle}(m) * Time_{cycle}$
- $N_{cycle}(m)$=number of cycles needed for executing m instructions
- $Time_{cycle}$=Cycle time of CPU (depends on clock rate)

CPI

- Note that, instructions may have different execution times. CPI: cycles per instruction. So, we can estimate the user CPU time of a program only if we know the CPI of the program.
- For simplicity, we assume instructions have the same execution time and CPI=1 in the following discussion.

Latency

Latency: User CPU time of handling "one" input.

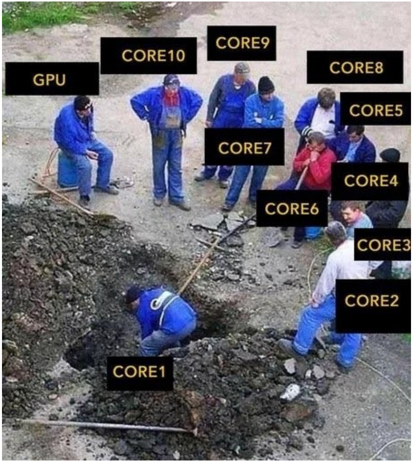
Throughput: number of instructions executed per unit of time

- Suppose a program needs to perform m instructions to handle a unit of input.
- And, it handles n unit of inputs (i.e., problem size) in T cycle time.
- Then, its throughput is $\frac{m*n}{T}$.

Performance vs Complexity

- One of the primary reasons to use threads is to improve performance.
- but techniques for improving performance also increase complexity and the likelihood of safety and liveness failures.

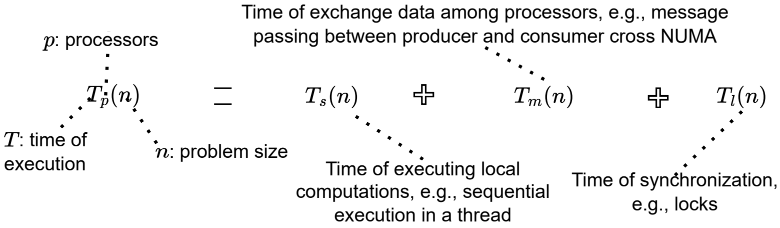
A Joke



Common case in real-life

You don't want your parallel program works like this.

Parallel Execution Time $T_p(n)$



Question

Can you explain the previous figure with the definition of $T_p(n)$?

Reduce $T_p(n)$

- $T_s(n)$ can be reduced by 1) smarter algorithm to bring down complexity, 2) better cache locality, 3) caching results, and so on.
- $T_m(n)$ can be reduced by 1) better program design (e.g., agglomeration, and thread reuse in executor framework), 2) better mapping strategy (e.g., NUMA-aware thread mapping), 3) data compression and so on.
- $T_l(n)$ can be reduced by 1) reduce critical section scope, 2) lock splitting/stripping, 3) replace locks and so on.

Parallel Program: Speedup

Measure the benefit of parallelism:

- $S_p(n) = \frac{T_{best\_seq}(n)}{T_p(n)}$
- Where, $T_p(n) = T_s(n) + T_m(n) + T_l(n)$
- Theoretically, $T_s(n) = \frac{T_{best\_seq}(n)}{p}$ in case of perfectly partitioning workloads among threads. In such as case, $S_p(n) = p$.
- However, due to $T_m(n) + T_l(n)$, it is usually $S_p(n) < p$. Hence, $S_p(n) \leq p$.
- Ideally, we aim to let $S_p(n) = p$ when designing parallel program.

Caution

In practice, $S_p(n) > p$ (superlinear speedup) is possible: it occurs when workload partitioning reduces total workloads, e.g., better cache utilization.

Parallel Program Cost: $C_p(n)$

- $C_p(n)=p*T_p(n)$, measures the total amount of work performed by all processors, i.e. processor-runtime product.
- A parallel program is **cost-optimal** if it executes the same total number of operations as the fastest sequential program, i.e., $C_p(n) = T_{best\_seq}(n)$.
  - However, the synchronization, message passing, waiting, etc add more operations to a parallel program.

Parallel Program: Efficiency

- Actual degree of speedup performance achieved compared to the maximum
- $E_p(n) = \frac{T_{best\_seq}(n)}{C_p(n)} = \frac{S_p(n)}{p} \leq 100\%.$

Understanding Scalability

- Interaction between the size of the problem and the size of the parallel computer (e.g., number of CPU cores)
  - Impact on load balancing, overhead, arithmetic intensity, locality of data access
  - Application dependent

Amdahl's Law

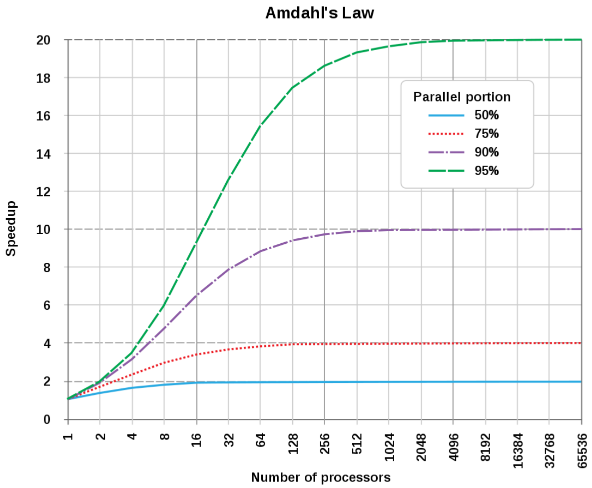
Amdahl's Law (1967)

Speedup of parallel execution is limited by the fraction of the algorithm that cannot be parallelized (f).

$$S_p(n) = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

- $f (0 \leq f \leq 1)$ is called the sequential fraction
- Also known as fixed-workload performance

Amdahl's Law Illustration



Understanding Scalability

- Manufacturers are discouraged from making large parallel computers
- More research attention was shifted towards developing parallelizing compilers that reduces sequential fraction

Really?

Amdahl's law assumes a fixed problem size.

Understanding Scalability

- However, f is not necessary a constant in many computing problems. For example, it can vary depending on the problem size n, e.g., how many requests need to process concurrently?
- As a result, f is often a function of n:

f is not a constant

$$\lim_{x \to \infty} f(n) = 0$$

Gustafson's Law (1988)

Gustafson's Law

Gustafson estimated the speedup $S_p(n)$ of a program gained by using parallel computing as follows:

$$\begin{aligned} S_p(n) &= f + (1-f) * p \\ &= p + (1-p) * f \end{aligned}$$

Implication

Gustafson's law instead proposes that programmers tend to increase the size of problems to fully exploit the computing power that becomes available as the resources improve.

Scaling Constraints

- Application-oriented scaling
  - Distribute one client request to one core.
  - Split computation into multiple phases and distribute each phase to one core.
  - Problem/application dependent.
- Resource-oriented scaling
  - Problem constrained scaling (PC): use a parallel computer to solve the same problem faster, e.g., divide the problem into #Cores pieces.
  - Time constrained scaling (TC): completing more work in a fixed amount of time.
  - Memory constrained scaling (MC): run the largest problem possible without overflowing main memory

Takehome Question

- We have previously studied a number of parallel program patterns.
- We can utilize those patterns to better design multithread program rather than from scratch. Furthermore,
- **Think about it:** If a program follows a certain pattern, say the Producer-Consumer pattern, we can then analysis its efficiency rather easily.
  - Why so? Give it a thought.

Performance Analysis Challenges

- Experiment with writing and tuning your own parallel programs
  - Many times, we obtain misleading results or tune code for a workload that is not representative of real-world use cases
- Start by setting your application performance goals
  - Response time, throughput, speedup?
  - Determine if your evaluation approach is consistent with these goals
- Try the simplest parallel solution first and measure performance to see where you stand

How really "good" your program is?

- Identify appropriate test scenarios – how the class is used
- Sizing empirically for various bounds, e.g., number of threads, buffer capabilities, etc.

Performance Measurement: Identify appropriate test scenarios

- E.g., if it's a shared queue, you may want to test insert and delete concurrently.
- If it's a shared stack, test pop and push concurrently.
- ...

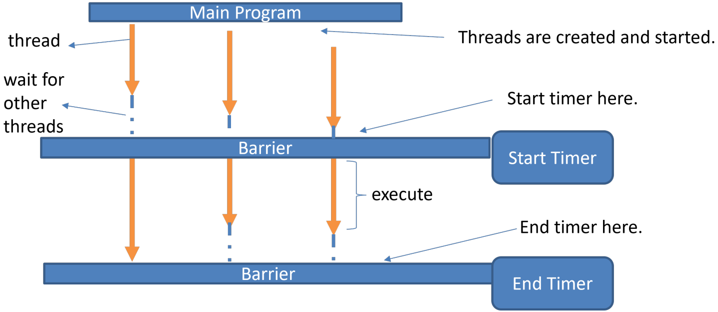
Performance Measurement: Sizing empirically

```
for (int cap = 1; cap <= 1000; cap *= 10)
    System.out.println("Capacity: " + cap);
    for (int pairs = 1; pairs <= 128; pairs *= 2)
        System.out.print("Pairs: " + pairs + "\t");
```

Running Time Measurement

- Measuring program running time for sequential program is easy.
  - Start Timer(); Program executes(); End Timer().
- It is challenging for metering concurrent program.
- Ideally, we shall have Start Timer(); Multithreads are running (); End Timer();
- But, each thread runs independently.
- To ensure measurement correctness, we need to synchronize those threads.

General idea of benchmarking concurrent program



CyclicBarrier

- A CyclicBarrier supports an optional Runnable command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released.

```
//when all parties ready, start/end the
    BarrierTimer.
final CyclicBarrier cb = new CyclicBarrier
    (3, BarrierTimer);
```

Example

See the BarrierTimer.java

BarrierTimer

```
public class BarrierTimer implements Runnable {
    private boolean started;
    private long startTime, endTime;

    public synchronized void run() {
        long t = System.nanoTime();
        if (!started) {
            started = true;
            startTime = t;
        } else
            endTime = t;
    }

    public synchronized void clear() {
        started = false;
    }

    public synchronized long getTime() {
        return endTime - startTime;
    }
}
```

Not Java?

We can implement our own barrierTimer in other programming language to achieve the same goal.

Use Case Study 1

- Let's try to use the barrierTimer to setup a proper testing framework.
- We are going to meter the efficiency of different counter implementations.

CasCounterTest

```
public class CasCounterTest {
    private BarrierTimer timer = new BarrierTimer();
    protected static final ExecutorService pool = Executors.
        newCachedThreadPool();
    //set up the data object here
    LockCounter lockCounter = new LockCounter();
    protected final int nTrials, nThreads;
    protected CyclicBarrier barrier;
    protected final int nIncrements = 10000;
    ...
```

Setup the testing framework: the timer and barrier.

CasCounterTest

```
class LockCounter {
    private int value;

    public synchronized int getValue() {
        return value;
    }

    public synchronized int increment() {
        return value++;
    }
}
public class CasCounterTest {
    ...
    public CasCounterTest(int nThreads, int trials) {
        this.nThreads = nThreads;
        this.nTrials = trials;
        barrier = new CyclicBarrier(nThreads + 1, timer);
    }
    ...
```

The implementation of lock based counter that we are interested at.

CasCounterTest

```
public class CasCounterTest {
    ...
    public void test() {
        try {
            timer.clear();
            for (int i = 0; i < nThreads; i++) {
                pool.execute(new Runnable() {
                    public void run() {
                        try {
                            barrier.await();
                            for (int i = 0; i < nIncrements; i++) {
                                //perform the data operation
                                lockCounter.increment();
                            }
                            barrier.await();
                        } catch (InterruptedException |
                             BrokenBarrierException e) {
                            e.printStackTrace();
                        }
                    }
                });
            }
            barrier.await();//start execution of all threads
            barrier.await();//wait for all to finish execution
            System.out.print("Total Time: " + timer.getTime());
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
```

CasCounterTest

```
public class CasCounterTest {
    ...
    public static void main(String[] args) throws Exception {
        int tpt = 100000;
        for (int nThreads = 32; nThreads <= 100; nThreads += 10) {
            CasCounterTest t = new CasCounterTest(nThreads, tpt);
            System.out.print("number of threads: " + nThreads + "\t");
            t.test();
            System.out.println();
            Thread.sleep(1000);
        }
        CasCounterTest.pool.shutdown();
    }
    ...
```

Changing the thread setting and metering.

Profiling

- Sometimes, we are not only interested at knowing how fast the program is,
- We want to know where the bottleneck is.

Performance analysis strategy

Determine what limits performance:

- Computation
- Memory bandwidth (or memory latency)
- Synchronization

Establish the bottleneck

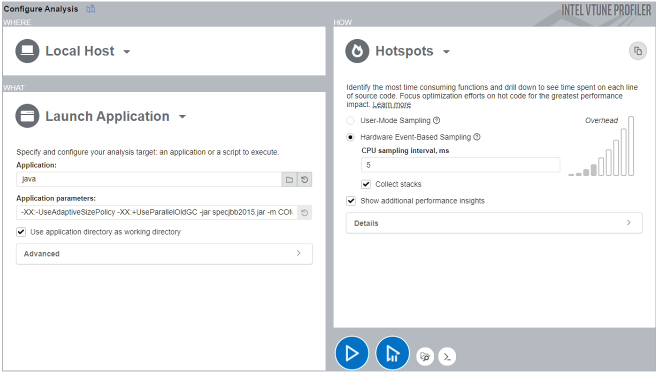
Possible Bottlenecks

- Instruction-rate limited: add "math" (non-memory instructions)
  - Does execution time increase linearly with operation count as math is added?
- Memory bottleneck: remove almost all math, but load same data
  - How much does execution-time decrease?
- Locality of data access: change all array accesses to A[0]
  - How much faster does your code get?
- Sync overhead: remove all atomic operations or locks
  - How much faster does your code get? (provided it still does approximately the same amount of work)

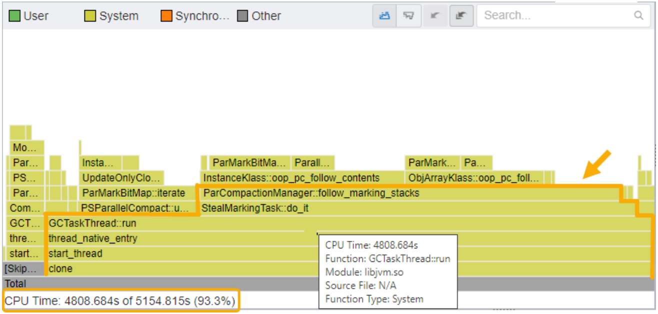
Instrumentation Tools

- Modify the source code, executable or runtime environment to understand the performance
- And, it can be tedious suppose we want to know the detailed performance of every portion of the program
  - Say, the program contains 10 methods, how much running time attribute to each of the method?
  - Of course, we can manually insert time measurement codes like we do before to measure each method, but it is very tedious.
  - There are tools to help.

Intel Vtune

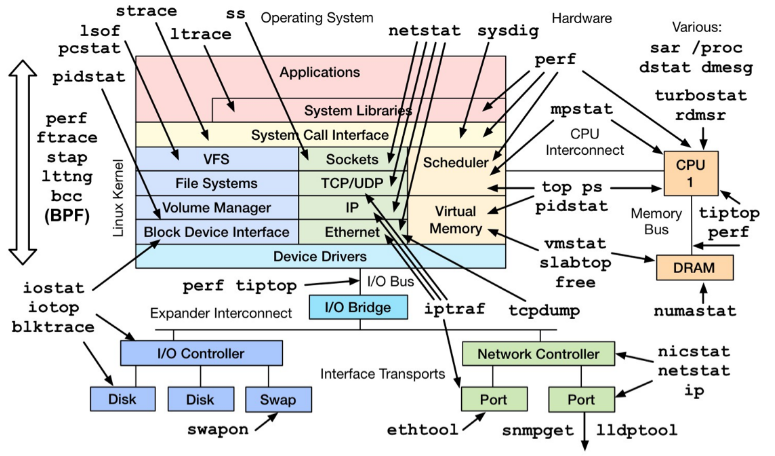

Configure Analysis
INTEL VTUNE PROFILER
WHERE
Local Host
HOW
Hotspots
WHAT
Launch Application
Identify the most time consuming functions and drill down to see time spent on each line of source code. Focus optimization efforts on hot code for the greatest performance impact. Learn more
User-Mode Sampling
Overhead
Hardware Event-Based Sampling
CPU sampling interval, ms
5
Collect stacks
Show additional performance insights
Details
Specify and configure your analysis target: an application or a script to execute.
Application:
java
Application parameters:
-XX:-UseAdaptiveSizePolicy -XX:+UseParallelOldGC -jar specjbb2015.jar -m COM
Use application directory as working directory
Advanced

Intel Vtune



- A Flame Graph is a visual representation of the stacks and stack frames.
- The width of each box in the graph indicates the percentage of the function CPU time to total CPU time.

Linux Performance Observation Tools



http://www.brendangregg.com/linuxperf.html 2016

Perf

- Modern architectures expose performance counters
  - Cache misses, branch mispredicts, IPC, etc
- Perf tool provides easy access to these counters
  - perf list – list counters available on the system
  - perf stat – count the total events
  - perf record – profile using one event
  - perf report – Browse results of perf record

Contents skipped

- Parallel computing is a complex subject and one module can hardly cover everything about it.
- Other related topics that are not covered but are still very important include:
  - "Parallel Algorithms"
  - "Thread Safety Design" and
  - "Parallel Program Correctness Theoretical Analysis"
- You are encouraged to do more self-study on those matters, a new MPE may be proposed to cover them if enough attentions.
- Next week, we will have the quiz 1 and we will offer some materials about "parallel hardware" for self-study as the last topic covered for the first half of the course.
- And, the second half of the course will focus on "distributed memory parallel computing", covered by another faculty.