Design and Engineering of Computer Systems

Lecture 35: Performance Scalability

Mythili Vutukuru
IIT Bombay
Performance scalability

• Performance measurement and analysis so far: how a system performs with a given configuration (CPU cores, memory, ...)

• Performance scalability: how performance improves if we give more resources to the system
  • A system with good performance scalability will improve its performance in proportion to the increase in resources

• After we have optimized performance of a system to the best possible extent, only way to improve performance further is by scaling:
  • Vertical scaling / scale-up: add more hardware resources (e.g., CPU cores / memory / whichever resource is bottleneck) to existing machine
  • Horizontal scaling / scale-out: add more replicas of bottleneck component and distribute load between replicas

• Cloud management systems provide auto-scaling: automatically identify when incoming load is beyond capacity, and scale bottleneck components
Multicore scalability

• One way to do vertical scaling for systems where CPU is performance bottleneck is to add more CPU cores to system
  • Cloud orchestration systems monitor CPU usage of components and dynamically assign more/less CPU cores based on utilization

• Multicore scalability: application performance increases in proportion to CPU cores assigned to application

• Applications that can be parallelized easily have good multicore scalability

• Common reasons for poor multicore scalability
  • Cache coherence overheads due to accessing shared memory via multiple CPU cores with private caches
  • Locking at application and OS serializes access to critical sections, reduces parallelism in application
bool isLocked = false
void acquire_lock() {
    while(test-and-set(isLocked, true) == true);
}

Cache coherence overhead

• When same memory location / variable is accessed from multiple CPU cores, multiple copies of cached data need to be kept in sync
  • Snooping or directory to keep track of which CPU core has cached which memory addresses
  • When one CPU core updates its private cache, other cores must update or invalidate their cached copy
  • Not just with true sharing, but also due to false sharing (CPU cores access different memory addresses located on same 64 byte cache line)

• Why do CPU cores access same memory location?
  • Multiple threads of process access same parts of memory image from different cores
  • Multiple processes in kernel mode can access same OS code/data from different cores
  • Variables like locks are accessed from multiple cores, resulting in cache line with lock variable bouncing across CPU cores during lock acquisition
Multicore speedup

- Perfect multicore scalability possible only if all threads/processes can execute independently in parallel on multiple CPU cores
- Sometimes, threads cannot execute in parallel, and must execute serially for some time, leading to poor multicore scalability
  - Example: only one thread at a time can execute critical section
  - Example: one thread in pipeline waits for previous thread to finish
- Amdahl’s law: estimate performance gains due to parallelism
  - Let \( T_1 \) = time required to perform a task on one CPU core
  - Let \( T_p \) = time required to perform task when running in parallel on “p” cores
  - Let \( \alpha \) = fraction of task that can be parallelized
  - We have \( T_p = (\alpha \cdot T_1 / p) + (1 - \alpha) \cdot T_1 \)
  - Speedup due to using multiple cores = \( T_1 / T_p \) (ideally \( p \) if \( \alpha = 1 \))
  - For large values of \( p \), speedup approx. \( 1 / (1 - \alpha) \)
  - If \( \alpha \) is small, speedup is small, poor multicore scalability
Techniques to improve multicore scalability

• Avoid sharing data across cores as far as possible: split application data into per-core / per-thread slices where possible
  • Split across cores at granularity of cache lines to avoid false sharing
• Use locks only when required, as locks cause cache coherence traffic and also serialize code execution
  • Modern lock implementations avoid excess cache coherence overheads when multiple threads on different cores contending for lock
• Lock-free application design, lock-free data structures
• OS designs evolving to scale well with multiple CPU cores, by splitting OS data structures into per-core slices where possible
• Modern OS have NUMA awareness: in NUMA systems (some CPU cores closer to some main memory), run process on CPU cores close to memory image
Horizontal scaling

• Suppose bottleneck component in a system cannot handle all incoming load, no matter how many optimizations are performed. What next?
  
• **Horizontal scaling**: instantiate multiple replicas of bottleneck component, distribute incoming load amongst replicas
  • Automatically done by cloud orchestration systems

• How do other components / clients contact multiple replicas?
  • Other components are told of multiple replicas explicitly (e.g., HTTP clients learn of multiple server replica IP addresses via DNS)
  • Or, all incoming traffic comes to a load balancer, which redirects traffic to replicas
  • Load balancer based design more popular as scaling is transparent to others

• **Load balancers** are special software/hardware components which redirect traffic to replicas as per some policy
  • Need to perform well to handle all incoming load without becoming bottleneck
  • Must adapt dynamically to changing load and changing number of replicas
Load balancer design

- Load balancer can operate at many layers of network stack
  - **Network layer load balancer**: only changes dst IP/port to redirect packets
    - All packets arrive to one (virtual) IP address/port number of server
    - Load balancer rewrites destination IP address/port number to redirect traffic to different server replicas
    - Does not perform any transport/application layer processing
  - **Application layer load balancer**: acts as application endpoint
    - Clients and other components connect to load balancer and not server replicas
    - Load balancer receives app requests (e.g., HTTP requests), makes a request again to server replica, fetches response, and sends it back to clients

- Application layer HTTP load balancers also serve other HTTP functions
  - Directly serve static content without contacting server replicas
  - Caching of responses from replicas, SSL termination, ...
  - Called reverse proxy servers (to differentiate from proxy servers at client side)
Load balancer policies

• How does load balancer distribute traffic to different replicas?
  • Note that traffic of one TCP/UDP connection should always go to same replica

• **Round robin:** assign connections to replicas in round robin manner
  • If first packet of a connection, pick one of the servers in round robin manner, store mapping from connection identifier (src/dst IP/port) to assigned server in a table
  • If packet of ongoing connection, redirect packet to previously assigned server

• **Hashing:** use hash of connection identifier to pick one of the server replicas
  • E.g., hash(src port, dst port, src IP, dst IP) modulo N, where N is number of servers
  • Problem: mappings of existing connections change when N changes, handle such changes carefully to not disrupt ongoing connections

• Other policies possible, e.g., pick least loaded server for a new connection
• What if requests of one user, coming on different connections, sent to different replicas? How is user state maintained correctly across replicas?
User stickiness in load balancing

• Some applications want to ensure “stickiness” of users or “sessions”
  • When user is purchasing product from e-commerce site, transaction happens over multiple TCP connections, which can go to different replicas
  • We would like all TCP connections of a user in one “session” to go to same replica

• Why stickiness? All data related to user’s session (e.g., shopping cart) is available in the same replica, instead of fetching from remote database frequently
  • Otherwise, every replica has to store/fetch session state in remote database often

• First connection of a session assigned to any replica using existing policy, e.g., round robin. Mapping from session to server stored. All subsequent connections of session assigned to same replica

• How is a user session identified? User source IP address, or HTTP cookies (special data in HTTP requests to identify users), ....

• With user stickiness, user data can be stored locally within components for faster access, need not store/fetch data in remote database servers for every request
Managing application state across replicas

- Application components store user state, e.g., current contents of user’s shopping cart. How to manage such state across multiple server replicas?
  - **Stateless design**: front end and app servers store no state, all state is stored/retrieved from backend databases for each request. Backend common to all replicas
    - High overhead due to remote access needed for every request
    - Easy to add server replicas and scale system horizontally; simple load balancer design
    - User level stickiness not needed, any replica can handle any user session
  - **Shared nothing stateful design**: each server replica locally stores a slice of application state for some users/sessions. User state is partitioned across replicas
    - Load balancer should ensure user level stickiness, redirect user traffic to replicas that have state
  - **Fully replicated stateful design**: all server replicas locally store application state of all users/sessions
    - Load balancer need not ensure user stickiness; any connection can be assigned to any server
    - Higher overhead than shared nothing design; servers must communicate with each other to keep all copies of user state consistent
Summary

• In this lecture:
  • Performance vs. scalability
  • Vertical scaling and multicore scalability
  • Horizontal scaling and load balancing

• Measure performance of any simple application/web server with increasing CPU cores. See if you get multicore scalability.