Performance and Optimization Issues in Multicore Computing

Minsoo Ryu

Department of Computer Science and Engineering
Hanyang University
Multicore Computing Challenges

- It is not easy to develop an efficient multicore program

- Three key aspects of multicore computing
  - Computation
  - Communication
  - Synchronization

- Some Challenges
  - Complex algorithms, multicore programming, memory issues, communication costs, ...
Outline

1. Computation Issues (Parallel Algorithms)
2. Memory Issues and Caching
3. Contention and Synchronization
4. Q & A
Parallel Algorithms

- The cost or complexity of serial algorithms is estimated in terms of the space (memory) and time (processor cycles) that they take.

- Parallel algorithms must take into account:
  - Parallelizability
  - Communication between different processors
  - Load balancing across processors

- Let’s consider two quicksort algorithms:
  - Sequential quicksort
  - Parallel quicksort
Sequential Quicksort

- Invented by Tony Hoare
  - Published in Comm. Of the ACM, 1961
  - 1980 Turing Award winner
  - Generally recognized as the fastest sorting algorithm, in the average case

- The quicksort algorithm
  - A divide and conquer algorithm which relies on a partition operation
  - To partition an array an element called a pivot is selected. All elements smaller than the pivot are moved before it and all greater elements are moved after it
  - The lesser and greater sublists are then recursively sorted
Example of Sequential Quicksort

- Examine the first, middle, and last entries

\[
\text{pivot} = \begin{array}{c}
57 & 70 & 97 & 38 & 63 & 21 & 85 & 68 & 76 & 9 & 81 & 36 & 55 & 79 & 74 & 85 & 16 & 61 & 77 & 49 & 24
\end{array}
\]

- Select 57 to be our pivot and move 24 into the first location

\[
\text{pivot} = 57
\]

- From both ends, search forward until we find 70 > 57 and search backward until we find 49 < 57

\[
\text{pivot} = 57
\]
Example of Sequential Quicksort

- **Swap 70 and 49**

  
  ```latex
  \text{pivot} = 57
  \begin{tabular}{cccccccccccc}
  24 & 49 & 97 & 38 & 63 & 21 & 85 & 68 & 76 & 9 & 81 & 36 & 55 & 79 & 74 & 85 & 16 & 61 & 77 & 70
  \end{tabular}
  ```

- **Search forward until we find 97 > 57 and search backward until we find 16 < 57**

  
  ```latex
  \text{pivot} = 57
  \begin{tabular}{cccccccccccc}
  24 & 49 & 97 & 38 & 63 & 21 & 85 & 68 & 76 & 9 & 81 & 36 & 55 & 79 & 74 & 85 & 16 & 61 & 77 & 70
  \end{tabular}
  ```

- **Swap 16 and 97**

  
  ```latex
  \text{pivot} = 57
  \begin{tabular}{cccccccccccc}
  24 & 49 & 16 & 38 & 63 & 21 & 85 & 68 & 76 & 9 & 81 & 36 & 55 & 79 & 74 & 85 & 97 & 61 & 77 & 70
  \end{tabular}
  ```
Example of Sequential Quicksort

- Repeat searching and swapping, and swap 68 and 9

  \[
  \text{pivot} = 57
  \]
  \[
  24 49 16 38 55 21 36 9 76 68 81 85 63 79 74 85 97 61 77 70
  \]

- Search forward and backward
  - The indices are out of order, so we stop

  \[
  \text{pivot} = 57
  \]
  \[
  24 49 16 38 55 21 36 9 76 68 81 85 63 79 74 85 97 61 77 70
  \]

- Move the larger item to the vacancy at the end of the array
  - Fill the empty location with the pivot, 57

  \[
  \text{pivot} = 57
  \]
  \[
  24 49 16 38 55 21 36 9 57 68 81 85 63 79 74 85 97 61 77 70 76
  \]
Example of Sequential Quicksort

- Now recursively call quick sort on the first and second half of the list
  - All entries < 57 are sorted
    ```
    pivot =
    24 49 16 38 55 21 36 9 57 68 81 85 63 79 74 85 97 61 77 70 76
    ```
  - All entries >= 57 are be sorted
    ```
    pivot =
    9 16 21 24 36 38 49 55 57 68 81 85 63 79 74 85 97 61 77 70 76
    ```
  - Finally, we arrive at an ordered list
    ```
    9 16 21 24 36 38 49 55 57 61 63 68 70 74 76 77 79 81 85 85 97
    ```
Parallel Quicksort

➢ Algorithm
   ▪ Each process holds a segment of the unsorted list
     • The unsorted list is evenly distributed among the processes
   ▪ Processes exchange their data
   ▪ The list segment stored on each process is sorted
     • The last element on process i’s list is smaller than the first element on process i + 1’s list
Example of Parallel Quicksort

(1) $\begin{align*}
P_0 & : 75, 91, 15, 64, 21, 8, 88, 54 \\
P_1 & : 50, 12, 47, 72, 65, 54, 66, 22 \\
P_2 & : 83, 66, 67, 0, 70, 98, 99, 82 \\
P_3 & : 20, 40, 89, 47, 19, 61, 86, 85
\end{align*}$

(2) $\begin{align*}
P_0 & : 75, 91, 15, 64, 21, 8, 88, 54 \\
P_1 & : 50, 12, 47, 72, 65, 54, 66, 22 \\
P_2 & : 83, 66, 67, 0, 70, 98, 99, 82 \\
P_3 & : 20, 40, 89, 47, 19, 61, 86, 85
\end{align*}$

Process $P_0$ chooses and broadcasts randomly chosen pivot value

(3) $\begin{align*}
P_0 & : 75, 91, 15, 64, 21, 8, 88, 54 \\
P_1 & : 50, 12, 47, 72, 65, 54, 66, 22 \\
P_2 & : 83, 66, 67, 0, 70, 98, 99, 82 \\
P_3 & : 20, 40, 89, 47, 19, 61, 86, 85
\end{align*}$

Exchange “lower half” and “upper half” values

(4) $\begin{align*}
P_0 & : 75, 15, 64, 21, 8, 54, 66, 67, 0, 70 \\
P_1 & : 50, 12, 47, 72, 65, 54, 66, 22, 20, 40, 47, 19, 61 \\
P_2 & : 83, 98, 99, 82, 91, 88 \\
P_3 & : 89, 86, 85
\end{align*}$

After exchange step
Example of Parallel Quicksort

Processes P0 and P2 choose and broadcast randomly chosen pivots

Exchange values

Each processor sorts values it controls
Performance Issues with Parallel Quicksort

- **Performance concerns**
  - The parallel quicksort algorithm is likely to do a poor job of load balancing
    - If the pivot value is not the median value, we will not divide the list into two equal sublists
    - Finding the median value is prohibitively expensive on a parallel computer
  - The communication cost for data exchange across processors could be too high

- **There are other quicksort algorithms**
  - Hyperquicksort
  - Parallel sorting by regular sampling
Task Dependency Graph

- Task dependency graph
  - One task cannot start until some other tasks finish
  - The output of one task is the input to another task
    - Often useful for data flow processing

- What are the latency and throughput?

![Diagram of Task Dependency Graph]

- Start
  - Thread A: 15 msec
  - Thread D: 30 msec

- Thread A to Thread B: 20 msec
- Thread C: 15 msec
- Thread E: 10 msec
- Thread F: 5 msec

- End
Worst Case Latency

➢ Find the critical path

\[ A + B + E = 45 \]
\[ A + C + E = 40 \]
\[ A + C + F = 35 \]
\[ D + C + E = 55 \]
\[ D + C + F = 50 \]
Maximum Throughput

➢ Find the slowest stage

Max(A, B, C, D, E, F) = 30 (= D)
Bottleneck Identification and Optimization

- Thread D is the common bottleneck for both latency and throughput
- Reduce the execution time of D to 25ms (⇐ 30ms)
  - Worst case latency: 50 msec (⇐ 55 msec)
  - Throughput: 25 msec (⇐ 30 msec)
Memory Issues and Caching
Memory Performance

- **DRAM Memory**
  - 30-50 nanoseconds to get data from memory into CPU
  - CPU can do 2-4 operations every 300 picoseconds
    - 100+ times slower than CPU!

- **Hard drive**
  - A typical 7200 RPM desktop HDD has a "disk-to-buffer" data transfer rate up to 1030 Mbit/s

- **Solution**
  - Rely on caching based on the memory hierarchy
  - If data fits in cache, you get Gigaflops performance per processor
Memory Hierarchy

Latency | Bandwidth | Managed by
--- | --- | ---
1 cyc | 3-10 words/cycle < 1KB | compiler managed
1-3 cyc | 1-2 words/cycle 32KB - 1MB | hardware managed
5-10 cyc | 1 word/cycle 1MB - 4MB | hardware managed
30-100 cyc | 0.5 words/cycle 64MB - 4GB | OS managed
$10^6$-$10^7$ cyc | 0.01 words/cycle 4GB+ | OS managed
User-Level Caching of Data and Code

- **Data caching**
  - Maintains a free list for each type of user-defined data structure
  - Allocate one from the free list, use it, and return it back to the list
    - Recall the slab allocator in the Linux kernel
  - Data content can also be cached for possible reuse

- **Code caching**
  - Cache translated code and reuse it
  - One application is the binary translation in full virtualization
False Sharing with Hardware Cache

- **False sharing**
  - Multiple threads are accessing items of data held on a single cache line
  - The cache line is constantly being bounced between processors due to the cache coherency mechanism

- **Solution**
  - It is easy to solve false sharing by padding the accessed structures so that the variable used by each thread resides on a separate cache line
Cache Coherence

- Cache coherence is the consistency of shared resource data stored in multiple local caches
- Two cache coherency protocols
  - Snoopy protocol
  - Directory-based protocol
False Sharing and Its Solution

- Each thread accesses to the counter structure
  - A. The size of counter is 4 bytes
  - B. Each counter is located at 64-byte intervals

A. thread code with false sharing
(64-byte cache line size)

B. thread code without false sharing
(64-byte cache line size)
Thread Pooling (Thread Cache)

- The overhead of dynamic thread creation and termination is high
  - If the amount of work known in advance use static thread creation otherwise use dynamic thread creation

- Thread pool
  - Create a number of threads
  - Jobs are organized in a queue
  - A new job is dispatched to a free thread in the pool
Thread Stack Size

- Each thread requires a separate stack area
  - Reading the stack size for a POSIX thread

```c
int main()
{
    size_t stacksize;
pthread_attr_t attributes;
pthread_attr_init( &attributes );
pthread_attr_getstacksize( &attributes, &stacksize );
printf( "Stack Size = %i\n", stacksize);
pthread_attr_destroy( &attributes );
}
```

- Running it on Ubuntu produces the result shown in Listing 5.11, indicating that the default stack size is 8MB

```
$ gcc stack.c -lpthread
$ ./a.out
Stack Size = 8388608
```
Thread Stack Size

- If each thread is allocated an 8MB stack, then there can be at most 512 threads on 32-bit HW
  - \( 512 \times 8 \text{ MB} = \text{entire 4GB address space} \)
  - The default size of stack depends on the underlying OS

- It is possible to change the default stack size by the following option
  - `ulimit -s <stacksize>`
  - Thus, it can be a good idea to assess how much memory is actually required for the stack of each child thread and limit the default stack size
Contention and Synchronization
Lock Contention

- Lock contention between different threads cause significant performance degradation
  - Hot locks (hot spots)

- General solutions (to reduce the lock holding time)
  - Narrow the lock scope
  - Reduce lock granularity
Lock Holder Preemption

- Lock holder preemption describes the situation when a CPU is preempted inside a critical section
  - Other threads waiting for the lock must wait indefinitely

- Well known problems
  - A VCPU (virtual CPU) is preempted inside the guest kernel while holding a spinlock (in OS virtualization)
  - Priority inversion problem (in real-time systems)
Shared Data Contention

- Contention on shared data may seriously limit performance as well as causing locking overhead
  - Only one thread can occupy the bus and access the memory at a time

- Two approaches
  - Data replication
  - Data partitioning
Data Replication and Consistency Control

- **Data replication is used for two purposes**
  - **To increase the reliability of a system**
    - The system can continue working after one replica crashes
  - **To improves the performance**
    - Processors can access its own copy simultaneously
    - Performs well especially when the data is mostly read

- **Consistency control**
  - One issue of data replication is consistency problems
  - Whenever a copy is modified, modifications have to be carried out on all copies
    - Analogous to the cache coherence problem
Data Partitioning

- Data is divided across \( n \) processors, and threads are routed to the partitions that contain the data they need to access.

- Data partitioning is often used for database systems on multicore HW.
  - Some applications are “perfectly partitionable”
    - Every transaction can be executed in its entirety at a single partition
  - Some applications are “imperfectly partitionable”
    - Many applications have some transactions that span multiple partitions
    - For these transactions, some form of concurrency control is needed
Rule of Thumb

- Always keep in mind
  - “Computation is FAST”
  - “Communication is SLOW”

- If you can “do extra work in an initialization step” to reduce the work done in each time step, it is generally well worth the effort

- Collect all of your “communication at one point” in your program