Parallelism beyond MapReduce: Threads on Multicores

Kai Shen

MapReduce Pro and Con

Pros:
- Easy programming interface;
- The underlying system can automatically support parallelism, data movement, load balancing, and fault-tolerance.

Cons:
- Programming interface is still restrictive for some applications;
- Too much automation, too little direct control for performance optimization;
- Not optimal for parallelism in single shared-memory multicore machine.

Threads

- Parallel programming model in a single shared-memory multi-core machine; without support for data move and management
- Programming model:
  - Create bunch of threads, each running something.
  - Multiple threads run in parallel on a multi-core.
  - All threads share the memory space (each thread can directly access any data in the memory space). [No direct data sharing in MapReduce]
  - Very flexible and fairly simple without synchronization.

Synchronization on Shared Data

- Two threads that operate on a shared counter:
  - One increases the counter by one;
  - The other decreases the counter by one.
- Counter increase may be implemented as:
  ```
  register1 = counter;
  register1 = register1 + 1;
  counter = register1;
  ```
- Counter decrease may be implemented as:
  ```
  register2 = counter;
  register2 = register2 - 1;
  counter = register2;
  ```
- Each counter operation must be protected from concurrent update by the other (synchronization by mutex locks).
Synchronization to Preserve Task Ordering (Dependencies)

- Task ordering needed in applications:
  - Word counting from individual documents need to complete before the aggregation of the counts;
  - In each iteration of K-means, the cluster assignment of all samples must happen before new cluster centers are computed and the next iteration commences.
- Dependencies can be preserved by synchronization (wait/signal).

When to Use Threads?

- When all data fit into memory, and
  - Threads is more flexible (than MapReduce) in allowing data sharing and controlling task ordering
  - Threads has less overhead (than MapReduce)
- When threads run in each multi-core machine of a large MapReduce cluster

Threads Programming Steps

- Decompose application into tasks
  - Like identify map tasks for MapReduce
  - Formalize the dependencies (more flexible than MapReduce)
  - How good is the decomposition?
    - Exposed parallelism, load balance, required synchronization
- Add synchronization
  - Preserve the task dependencies
  - Protect shared data structure for concurrent accesses
- Fine-grain vs. coarse-grained decomposition
  - Parallelism vs. challenges in control

Performance Objectives

- Fast speed
  - Assume the sequential is already optimized, we want high speedup, ideally equaling the number of CPU cores
Application: Word Counting

- How to support word counting with threads?
  - Step 1: Each thread counts a subset of the document
  - Step 2: Counts are aggregated after all documents are counted
- Parallelism is step 1; dependency between steps 1 and 2
- Is it going to work well (good speedup)?
  - Good parallelism, reasonable load balancing, little dependency, but too small compute-to-data ratio

Application: K-means

- How to support K-means with threads?
  - Within each iteration, parallelize the sample clustering
  - Synchronization between iterations
- Is it going to work well (good speedup)?
  - Some parallelism, good load balancing, more dependency (compared to word count), but better compute-to-data ratio (with a large number of iterations).

Application: PageRank Computation

- A linear system of equations (N variables, N linear equations)
- Solved iteratively with a matrix-vector multiplication at each step
- How to support PageRank with threads?
  - Within each iteration, parallelize the matrix-vector multiplication
  - Synchronization between iterations
- Is it going to work well (good speedup)?
  - Good parallelism, good load balancing, some dependency, good compute-to-data ratio (with a large number of iterations)
- But iterative method doesn’t produce precise solution

Application: Gaussian Elimination

- Simplification – ignore final solving step and pivoting
- Reduce an equation matrix into an equivalent upper-diagonal

\[ \begin{align*}
  A & \rightarrow X \\
  p^*A_{c,c} + A_{r,c} & \rightarrow X \\
  X & = r_1 + p^*r_2 \\
  \text{for } c=1 \text{ to } N \\
  \text{for } r=c+1 \text{ to } N \\
  \text{zero out } A_{r,c} \text{ by adding } p^*A_{r,c} \text{ to } A_{r,r} 
\end{align*} \]
**Application: Gaussian Elimination**

\[
p^* A_{c,1} + A_{c,1}^* + A_{c,2}^* = R_{c,2} + p^* c_1
\]

for \(c=1\) to \(N\)
for \(r=c+1\) to \(N\)
zero out \(A_{c,r}\) by adding \(p^* A_{c,1}\) to \(A_{c,r}\)

- **How to parallelize? Dependencies:**
  - Outer loop instances (e.g., \(c=2\) depends on \(c=1\))?  
  - Inner loop instances (e.g., \(r=3\) depends on \(r=2\))?  
  - Within one inner loop instance?

- **Task decomposition:**
  - Row, column, 2-dimensional

**Irregular Parallelism**

- Real problems contain large, sparse matrices
- Solve them as dense matrices waste time on zero-element operations
- Sparse matrix computation
  - Load imbalance
  - Managing nonzero fillins

**Static vs. Dynamic Task Assignment**

- **Static task assignment**
  - No. of tasks = No. of threads = No. of CPU cores
  - Each task gets a dedicated thread/CPU and runs to the end

- **Dynamic task assignment**
  - No. of tasks > No. of threads
  - Maintain a queue of ready tasks, protected by mutex lock
  - Each thread grabs a task and runs it; grabs another one when the current task completes
  - Advantage: good load balancing
  - Disadvantage: complex implementation, may hurt data locality

**Example Speedup Results**

- The speed ratio over the best sequential run
Performance Considerations

- Load balancing
- Synchronizations
  - your application synchronization and synchronization in the OS
- Memory bandwidth
  - is the performance bounded by reading from the memory?
- I/O
  - is the performance bounded by reading from the storage?
- Data locality
  - cache-efficient algorithms (block-based computation)
- Contention on shared resources

Multicore Architecture

- Last-level cache becomes a significant part of the chip
  ⇒ Multicore: add another processor core on the chip (sharing a single last-level cache)
  - Low cost (manufacturing, power)

- Additional benefit: faster processor-to-processor sharing

Contention on Shared Resources

- Shared resources: cache space, memory bandwidth
- What am I not seeing the speedup I expect?
- Load balancing implication