Background Knowledge

http://net.pku.edu.cn/~course/cs402

Peng Bo
School of EECS, Peking University
6/26/2008

Refer to Aaron Kimball’s slides

Background Topics

• Parallelization & Synchronization
• Fundamentals of Networking
• Search Engine Technology
  – Inverted index
  – PageRank algorithm

Parallelization & Synchronization

Parallelization Idea

• Parallelization is “easy” if processing can be cleanly split into n units:

```
/ \            
\  \ work     
\  \   |      
\  \  |      
\  \  |      
\  \  |      
\   v    Partition problem

thread  thread  thread
```

Parallelization Idea (2)

Spawn worker threads:

```
thread  thread  thread
```

In a parallel computation, we would like to have as many threads as we have processors. e.g., a four-processor computer would be able to run four threads at the same time.

Parallelization Idea (3)

```
thread  thread  thread
```

Workers process data
Parallelization Idea (4)

Report results

Parallelization Pitfalls

But this model is too simple!

- How do we assign work units to worker threads?
- What if we have more work units than threads?
- How do we aggregate the results at the end?
- How do we know all the workers have finished?
- What if the work cannot be divided into completely separate tasks?

What is the common theme of all of these problems?

Parallelization Pitfalls (2)

- Each of these problems represents a point at which multiple threads must communicate with one another, or access a shared resource.

- Golden rule: Any memory that can be used by multiple threads must have an associated synchronization system!

What is Wrong With This?

Thread 1:
```c
void foo() {
   x++;
   y = x;
}
```

Thread 2:
```c
void bar() {
   y++;
   x++;
}
```

If the initial state is $y = 0$, $x = 6$, what happens after these threads finish running?

Multithreaded = Unpredictability

- Many things that look like “one step” operations actually take several steps under the hood:

Thread 1:
```c
void foo() {
   eax = mem[x];
   inc eax;
   mem[x] = eax;
   ebx = mem[x];
   mem[y] = ebx;
}
```

Thread 2:
```c
void bar() {
   eax = mem[y];
   inc eax;
   mem[y] = eax;
   eax = mem[x];
   inc eax;
   mem[x] = eax;
}
```

- When we run a multithreaded program, we don’t know what order threads run in, nor do we know when they will interrupt one another.

Multithreaded = Unpredictability

This applies to more than just integers:

- Pulling work units from a queue
- Reporting work back to master unit
- Telling another thread that it can begin the “next phase” of processing

... All require synchronization!
Synchronization Primitives

- synchronization primitive
  - Semaphore / mutex
  - Condition variable
  - Barriers

Semaphores

- A semaphore is a flag that can be raised or lowered in one step
- Semaphores were flags that railroad engineers would use when entering a shared track

The “Corrected” Example

Thread 1:

```java
void foo() {
    sem.lock();
    x++;
    y = x;
    sem.unlock();
}
```

Thread 2:

```java
void bar() {
    sem.lock();
    y++;
    x++;
    sem.unlock();
}
```

Global var “Semaphore sem = new Semaphore();” guards access to x & y

Condition Variables

- A condition variable notifies threads that a particular condition has been met
- Inform another thread that a queue now contains elements to pull from (or that it’s empty – request more elements!)

The final example

Thread 1:

```java
void foo() {
    sem.lock();
    x++;
    y = x;
    fooDone = true;
    sem.unlock();
    fooFinishedCV.notify();
}
```

Thread 2:

```java
void bar() {
    sem.lock();
    while(!fooDone)
        fooFinishedCV.wait(sem);
    y++;
    x++;
    sem.unlock();
}
```

Global vars: Semaphore sem = new Semaphore(); ConditionVar fooFinishedCV = new ConditionVar(); boolean fooDone = false;

Barriers

- A barrier knows in advance how many threads it should wait for. Threads “register” with the barrier when they reach it, and fall asleep.
- Barrier wakes up all registered threads when total count is correct
- Pitfall: What happens if a thread takes a long time?
Too Much Synchronization?
Deadlock

Synchronization becomes even more complicated when multiple locks can be used.

Can cause entire system to "gel stuck".

Thread A:
semaphore1.lock();
semaphore2.lock();
/* use data guarded by
sempahores */
semaphore1.unlock();
semaphore2.unlock();

Thread B:
semaphore2.lock();
semaphore1.lock();
/* use data guarded by
sempahores */
semaphore1.unlock();
semaphore2.unlock();

Image: RPI CSCI.4210 Operating Systems notes

And if you thought I was joking…

The Moral: Be Careful!

• Synchronization is hard
  – Need to consider all possible shared state
  – Must keep locks organized and use them consistently and correctly
• Knowing there are bugs may be tricky; fixing them can be even worse!
• Keeping shared state to a minimum reduces total system complexity

Fundamentals of Networking

Sockets: The Internet = tubes?

• A socket is the basic network interface
• Provides a two-way "pipe" abstraction between two applications
• Client creates a socket, and connects to the server, who receives a socket representing the other side

Ports

• Within an IP address, a port is a sub-address identifying a listening program
• Allows multiple clients to connect to a server at once
Example: Web Server (1/3)

1) Server creates a socket attached to port 80

The server creates a listener socket attached to a specific port. 80 is the agreed-upon port number for web traffic.

Example: Web Server (2/3)

2) Client creates a socket and connects to host

The client-side socket is still connected to a port, but the OS chooses a random unused port number. When the client requests a URL (e.g., “www.google.com”), its OS uses a system called DNS to find its IP address.

Example: Web Server (3/3)

3) Server accepts connection, gets new socket for client

Server chooses a randomly-numbered port to handle this particular client. Listener is ready for more incoming connections, while we process the current connection in parallel.

Example: Web Server (2/3)

4) Data flows across connected socket as a “stream”, just like a file

What makes this work?

- Underneath the socket layer are several more protocols
- Most important are TCP and IP (which are used hand-in-hand so often, they’re often spoken of as one protocol: TCP/IP)

IP: The Internet Protocol

- Defines the addressing scheme for computers
- Encapsulates internal data in a “packet”
- Does not provide reliability
- Just includes enough information for the data to tell routers where to send it

TCP: Transmission Control Protocol

- Built on top of IP
- Introduces concept of “connection”
- Provides reliability and ordering
Why is This Necessary?

- Not actually tube-like “underneath the hood”
- Unlike phone system (circuit switched), the packet switched Internet uses many routes at once

Networking Issues

- If a party to a socket disconnects, how much data did they receive?
- … Did they crash? Or did a machine in the middle?
- Can someone in the middle intercept/modify our data?
- Traffic congestion makes switch/router topology important for efficient throughput

Search Engine Technology

- Built to encourage public search work
  - Open-source, w/pluggable modules
  - Cheap to run, both machines & admins
- Goal: Search more pages, with better quality, than any other engine
  - Pretty good ranking
  - Has done ~ 200M pages, more possible
- Hadoop is a spinoff

Meta-details

- Built to encourage public search work
  - Open-source, w/pluggable modules
  - Cheap to run, both machines & admins
- Goal: Search more pages, with better quality, than any other engine
  - Pretty good ranking
  - Has done ~ 200M pages, more possible
- Hadoop is a spinoff
Moving Parts

- Acquisition cycle
  - WebDB
  - Fetcher
- Index generation
  - Indexing
  - Link analysis (maybe)
- Serving results

WebDB

- Contains info on all pages, links
  - URL, last download, # failures, link score, content hash, ref counting
  - Source hash, target URL
- Must always be consistent
- Designed to minimize disk seeks
  - 19ms seek time x 200m new pages/mo
  - ~44 days of disk seeks!
- Single-disk WebDB was huge headache

Fetcher

- Fetcher is very stupid. Not a “crawler”
- Pre-MapRed: divide “to-fetch list” into $k$ pieces, one for each fetcher machine
- URLs for one domain go to same list, otherwise random
  - “Politeness” w/o inter-fetcher protocols
  - Can observe robots.txt similarly
  - Better DNS, robots caching
  - Easy parallelism
- Two outputs: pages, WebDB edits

WebDB/Fetcher Updates

URL: http://www.yahoo/index.html
ContentHash: MD5_balboglerropewolefbag
LastUpdated: Today!

URL: http://www.flickr.com/index.html
ContentHash: MD5_toewkekqmekkalekaa
LastUpdated: Today!

URL: http://www.cnn.com/index.html
ContentHash: MD5_sdflkjweroiwelksd
LastUpdated: 4/07/05

URL: http://www.about.com/index.html
ContentHash: MD5_balboglerropewolefbag
LastUpdated: Today!

Fetcher edits

Indexing

- How to retrieve some information from a large document set efficiently?
Document Collection

User Information Need

• Search inside this news site for articles talks about Culture between China and Japan, and doesn't talk about students abroad.
• QUERY:
  - “中国 日本 文化 -留学生”

How to do it?

• Could grep all of Web Pages for “中国”、“文化”and “日本”, then strip out pages containing “留学生”?  
  - Slow (for large corpora)
  - NOT “留学生” is non-trivial
  - Other operations (e.g., find “中国 NEAR “日本””) not feasible

Document Representation

• Bag of words model
• Document-term incidence matrix

<table>
<thead>
<tr>
<th>Term</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
<th>D6</th>
<th>D7</th>
</tr>
</thead>
<tbody>
<tr>
<td>中国</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>文化</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>日本</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>留学生</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>教育</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>北京</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Retrieval

• Search inside this news site for articles talks about Culture between China and Japan, and doesn’t talk about students abroad.
• To answer query:
  - take the vectors for “中国”, “文化”, “日本”, “留学生”(complemented) bitwise AND
  - 101110 AND 110010 AND 011011 AND 100011
  = 000010

Incidence Vector

• Transpose the Document-term incidence matrix
• So we have a 0/1 vector for each term.

1 if page contains word, 0 otherwise
Let’s build a search system!

- Consider \( N = 1 \text{ million} \) documents, each with about 1K terms.
- Avg 6 bytes/term include spaces/punctuation
  - 6GB of data in the documents.
- Say there are \( M = 500K \) distinct terms among these.

Can’t build the matrix

- 500K x 1M matrix has half-a-trillion 0’s and 1’s.
- But it has no more than one billion 1’s.
  - matrix is extremely sparse.
- What’s a better representation?
  - We only record the 1 positions.

Inverted index

- For each term \( T \): store a list of all documents that contain \( T \).
- Do we use an array or a list for this?

Inverted index construction

Documents to be indexed.

Token stream.

Linguistic modules

Modified tokens.

Indexer

Inverted index.

Sorted by docID (more later on why).
Indexer steps

- Sequence of (Modified token, Document ID) pairs.

I did enact Julius Caesar. I was killed in the Capitol; Brutus killed me.

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious.

Core indexing step

- Multiple term entries in a single document are merged.
- Frequency information is added.

The result is split into a Dictionary file and a Postings file.

Query processing

- Consider processing the query: 中国 AND 文化
  - Locate 中国 in the Dictionary;
  - Retrieve its postings.
  - Locate 文化 in the Dictionary;
  - Retrieve its postings.
  - "Merge" the two postings:
The merge

- Walk through the two postings simultaneously, in time linear in the total number of postings entries.

If the list lengths are \( x \) and \( y \), the merge takes \( O(x+y) \) operations.

**Crucial:** postings sorted by docID.

Indexing in Nutch

- Iterate through all \( k \) page sets in parallel, constructing inverted index.
- Creates a "searchable document" of:
  - URL text
  - Content text
  - Incoming anchor text
- Other content types might have a different document fields
  - Eg. email has sender/receiver
  - Any searchable field end-user will want
- Uses Lucene text indexer

Link analysis

- A page’s relevance depends on both intrinsic and extrinsic factors
  - Intrinsic: page title, URL, text
  - Extrinsic: anchor text, link graph
- **PageRank** is most famous of many
- Others include:
  - HITS
  - OPIC
  - Simple incoming link count

PageRank Algorithm

We assume page \( i \) has pages \( T_i \) to which it points \( (i.e., \text{are citations}) \). The parameter \( d \) is a damping factor which can be set between 0 and 1. We usually set \( d \) to 0.85. There are more details about \( d \) in the next section. Also \( C(i) \) is defined as the number of links going out of page \( i \). The PageRank of a page \( i \) is given as follows:

\[
PR(i) = \left(1-d\right) + d \left(PR(T1)/C(T1) + \ldots + PR(Tn)/C(Tn)\right)
\]

Note that the PageRanks form a probability distribution over web pages, so the sum of all web pages’ PageRanks will be one.
Link analysis in Nutch

- Nutch performs analysis in WebDB
  - Emit a score for each known page
  - At index time, incorporate score into inverted index
- Extremely time-consuming
  - In our case, disk-consuming, too (because we want to use low-memory machines)
- Link analysis is sexy, but importance generally overstated, so fast and easy:
  - \(0.5 \times \log(\# \text{ incoming links})\)

Administering Nutch

- Admin costs are critical
  - It’s a hassle when you have 25 machines
    - Google has >100k, probably more
- Files
  - WebDB content, working files
  - Fetchlists, fetched pages
  - Link analysis outputs, working files
  - Inverted indices
- Jobs
  - Emit fetchlists, fetch, update WebDB
  - Run link analysis
  - Build inverted indices

Administering Nutch (2)

- Admin sounds boring, but it’s not!
  - Really
    - I swear
- Large-file maintenance
  - Google File System (Ghemawat, Gobioff, Leung)
    - Nutch Distributed File System
- Job Control
  - Map/Reduce (Dean and Ghemawat)
    - Pig (Yahoo Research)
- Data Storage (BigTable)

Nutch Distributed File System

- Similar, but not identical, to GFS
- Requirements are fairly strange
  - Extremely large files
  - Most files read once, from start to end
  - Low admin costs per GB
- Equally strange design
  - Write-once, with delete
  - Single file can exist across many machines
  - Wholly automatic failure recovery

NDFS (2)

- Data divided into blocks
- Blocks can be copied, replicated
- Datanodes hold and serve blocks
- Namenode holds meta info
  - Filename \(\rightarrow\) block list
  - Block \(\rightarrow\) datanode-location
- Datanodes report in to namenode every few seconds
NDFS File Read

1. Client asks datanode for filename info
2. Namenode responds with blocklist, and location(s) for each block
3. Client fetches each block, in sequence, from a datanode

NDFS Replication

1. Always keep at least k copies of each blk
2. Imagine datanode 4 dies; blk 90 lost
3. Namenode loses heartbeat, decrements blk 90’s reference count. Asks datanode 5 to replicate blk 90 to datanode 0
4. Choosing replication target is tricky

Map/Reduce

- Map/Reduce is programming model from Lisp (and other places)
  - Easy to distribute across nodes
  - Nice retry/failure semantics
- map(key, val) is run on each item in set
  - emits key/val pairs
- reduce(key, vals) is run for each unique key emitted by map()
  - emits final output
- Many problems can be phrased this way

Map/Reduce (2)

- Task: count words in docs
  - Input consists of (url, contents) pairs
  - map(key=url, val=contents):
    - For each word w in contents, emit (w, "1")
  - reduce(key=word, values=uniq_counts):
    - Sum all "1"s in values list
    - Emit result "(word, sum)"

Map/Reduce (3)

- Task: grep
  - Input consists of (url+offset, single line)
  - map(key=url+offset, val=line):
    - If contents matches regexp, emit (line, "1")
  - reduce(key=line, values=uniq_counts):
    - Don’t do anything; just emit line
- We can also do graph inversion, link analysis, WebDB updates, etc

Map/Reduce (4)

- How is this distributed?
  1. Partition input key/value pairs into chunks, run map() tasks in parallel
  2. After all map()s are complete, consolidate all emitted values for each unique emitted key
  3. Now partition space of output map keys, and run reduce() in parallel
- If map() or reduce() fails, reexecute!
Map/Reduce Job Processing

1. Client submits "grep" job, indicating code and input files
2. JobTracker breaks input file into \( k \) chunks, and assigns work to trackers. (In this case 6.) Assigns work to trackers.
3. After map(), tasktrackers exchange map-output to build reduce() keyspace.
4. JobTracker breaks reduce() keyspace into \( m \) chunks (in this case 6). Assigns work.
5. Reduce() output may go to NDFS.

Nutch & Hadoop

- NDFS stores the crawl and indexes
- MapReduce for indexing, parsing, WebDB construction, even fetching
  - Broke previous 200M/mo limit
  - Index-serving?
- Required massive rewrite of almost every Nutch component

Summary

- Parallelization & Synchronization
- Fundamentals of Networking
- Search Engine Technology
  - Inverted index
  - PageRank algorithm

Readings


Q&A