The Wide Finder Project

An overview of Tim Bray's Wide Finder project by

Peter Robinson

Mar 19, 2010

What is a Wide Finder?

- Apache log files (ASCII) can be huge (250+MB in our test case).
- Is there a way to process these files (FIND) that is scalable with multiple cores (WIDE)?

Sample Input

```
host-24-225-218-245.patmedia.net - - [01/Oct/2006:06:33:45 -0700] "GET /ongoing/ongoing.atom HTTP/1.1" 304 - "-" "NetNewsWire/
2.0b37 (Mac OS X; Lite; http://ranchero.com/netnewswire/)"
72-48-42-121.dyn.grandenetworks.net - - [01/Oct/2006:06:33:45 -0700] "GET /ongoing/ongoing.atom HTTP/1.1" 200 44877 "-"
"Onfolio/2.02"
c529d19fd.cable.wanadoo.nl - - [01/Oct/2006:06:33:45 -0700] "GET /ongoing/ongoing.rss HTTP/1.0" 301 310 "-" "BitTorrent/4.0.0"
c529d19fd.cable.wanadoo.nl - - [01/Oct/2006:06:33:45 -0700] "GET /ongoing/ongoing.rss HTTP/1.0" 301 315 "-" "BitTorrent/4.0.0"
c529d19fd.cable.wanadoo.nl - - [01/Oct/2006:06:33:46 -0700] "GET /ongoing/ongoing.atom HTTP/1.0" 304 - "-" "BitTorrent/4.0.0"
lj602070.inktomisearch.com - - [01/Oct/2006:06:33:46 -0700] "GET /ongoing/When/200x/2004/04/18/Persuasion HTTP/1.0" 304 - "-"
"Mozilla/5.0 (compatible; Yahoo! Slurp; http://help.yahoo.com/help/us/ysearch/slurp)"
72-48-42-121.dyn.grandenetworks.net - - [01/Oct/2006:06:33:49 -0700] "GET /ongoing/ongoing.atom HTTP/1.1" 200 44877 "-"
"Onfolio/2.02"
cuscon24086.tstt.net.tt - - [01/Oct/2006:06:33:49 -0700] "GET /ongoing/ongoing.atom HTTP/1.1" 304 - "-" "Mozilla/5.0 (Windows; U;
Windows NT 5.1; en-US; rv:1.8.0.7) Gecko/20060909 Firefox/1.5.0.7"
fj5022.inktomisearch.com - - [01/Oct/2006:06:33:50 -0700] "GET /ongoing/When/200x/2004/05/03/Pedroni HTTP/1.0" 200 13140 "-"
"Mozilla/5.0 (compatible; Yahoo! Slurp; http://help.yahoo.com/help/us/ysearch/slurp)"

+ 1,000,000 more lines or so
```

What we're trying to identify...

host-24-225-218-245.patmedia.net - - [01/Oct/2006:06:33:45 -0700] "GET /ongoing/ongoing.atom HTTP/1.1" 304 - "-" "NetNewsWire/2.0b37 (Mac OS X; Lite; http://ranchero.com/netnewswire/)"
72-48-42-121.dyn.grandenetworks.net - - [01/Oct/2006:06:33:45 -0700] "GET /ongoing/ongoing.atom HTTP/1.1" 200 44877 "-" "Onfolio/2.02"
c529d19fd.cable.wanadoo.nl - - [01/Oct/2006:06:33:45 -0700] "GET /ongoing/ongoing.rss HTTP/1.0" 301 310 "-" "BitTorrent/4.0.0"
c529d19fd.cable.wanadoo.nl - - [01/Oct/2006:06:33:45 -0700] "GET /ongoing/ongoing.rss HTTP/1.0" 301 315 "-" "BitTorrent/4.0.0"
c529d19fd.cable.wanadoo.nl - - [01/Oct/2006:06:33:46 -0700] "GET /ongoing/ongoing.atom HTTP/1.0" 304 - "-" "BitTorrent/4.0.0"
lj602070.inktomisearch.com - - [01/Oct/2006:06:33:46 -0700] "GET /ongoing/When/200x/2004/04/18/Persuasion HTTP/1.0" 304 - "-" "Mozilla/5.0 (compatible; Yahoo! Slurp; http://help.yahoo.com/help/us/ysearch/slurp)"
72-48-42-121.dyn.grandenetworks.net - - [01/Oct/2006:06:33:49 -0700] "GET /ongoing/ongoing.atom HTTP/1.1" 200 44877 "-" "Onfolio/2.02"
cuscon24086.tstt.net.tt - - [01/Oct/2006:06:33:49 -0700] "GET /ongoing/ongoing.atom HTTP/1.1" 304 - "-" "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.7) Gecko/20060909 Firefox/1.5.0.7"
fj5022.inktomisearch.com - - [01/Oct/2006:06:33:50 -0700] "GET /ongoing/When/200x/2004/05/03/Pedroni HTTP/1.0" 200 13140 "-" "Mozilla/5.0 (compatible; Yahoo! Slurp; http://help.yahoo.com/help/us/ysearch/slurp)"

+ 1,000,000 more lines or so

Desired Output

Elapsed Time

User Time

Count Breakdown

```
0.787870883942 0.797733
          2006/01/08/No-New-XML-Languages = 600
                    2003/02/04/Construction = 600
                  2004/04/27/RSSticker = 600
                    2003/06/23/SamsPie = 700
                      2003/09/18/NXML = 800
                    2003/10/16/Debbie = 800
               2006/01/31/Data-Protection = 800
                  2003/07/25/NotGaming = 1300
                  2006/07/28/Open-Data = 2000
               2006/09/29/Dynamic-IDE = 8900
```

Sample Ruby Code

```
counts = {}
counts.default = 0

ARGF.each_line do |line|
  if line =~ %r{GET /ongoing/When/\d\d\dx/(\d\d\d\d/\d\d/\d\d/
[^ .]+) }
    counts[$1] += 1
  end
end

keys_by_count = counts.keys.sort { |a, b| counts[b] <=> counts
[a] }
keys_by_count[0 .. 9].each do |key|
  puts "#{counts[key]}: #{key}"
end
```

Need to ...

search the file and match to a reg exp.

output a breakdown for the number of matches for each key

Fundamental steps we need to take...

- Open the file and read it
- Search the data and find all matches to a regular expression
- Output counts of each unique key match

The Challenge

- Can this problem be scaled to multiple cores?

What ensued

- When Tim Bray posed this challenge on his blog Ongoing, many people stepped up to the plate. Although Tim Bray argues that this is fundamentally a parallel IO problem, all solutions posted did no true parallel IO, but instead minimized the amount of IO required, in addition to providing parallel implementations of the actual calculations required.
- What is perhaps surprising is how much room for improvement there was on this front.
- The benchmarking was done using at test file of about 250 MB, which was easily small enough to fit in file cache. All results posted were done with the cache warmed up, so essentially no reads from disk during timing.
- We will focus on the techniques used to optimize the text processing/ regular expression matching inherent to this problem.

Cold vs. Hot cache

Cold

```
7.0644030571 1.413628
                    2005/07/27/Atomic-RSS = 600
                  2003/02/04/Construction = 600
                    2004/04/27/RSSticker = 600
                      2003/06/23/SamsPie = 700
                         2003/09/18/NXML = 800
                       2003/10/16/Debbie = 800
                2006/01/31/Data-Protection = 800
                      2003/07/25/NotGaming = 1300
                      2006/07/28/Open-Data = 2000
                   2006/09/29/Dynamic-IDE = 8900
```

Hot

```
0.785873889923 0.796048
                    2005/07/27/Atomic-RSS = 600
                  2003/02/04/Construction = 600
                    2004/04/27/RSSticker = 600
                      2003/06/23/SamsPie = 700
                         2003/09/18/NXML = 800
                       2003/10/16/Debbie = 800
                2006/01/31/Data-Protection = 800
                      2003/07/25/NotGaming = 1300
                      2006/07/28/Open-Data = 2000
                   2006/09/29/Dynamic-IDE = 8900
```

The difference between the first time a 250 MB file is read and subsequent reads is huge (7.06 seconds vs. 0.785 elapsed time for this highly optimized implementation) due to the file being cached in main memory. Comparisons on the Wide Finder website all are using the cached file, not reading from disk. Thus, we're not really doing IO at all, just reading large blocks from memory.

Parallel IO

- True concurrency in IO requires specialized hardware and file systems for this task (RAID,striping, multiple disks, etc.)
- Will always be hardware limited.

Dealing with Reg Expressions

```
ARGF.each_line do |line|
  if line =~ %r{GET /ongoing/When/\d\d\dx/(\d\d\d\d/\d\d/\d\d/
[^ .]+) }
  end
end
```

- What all happens in this line of code?
  - First, the regular expression needs to get compiled.
  - Then, we actually need to test the expression against lines in the file.

Dealing with Reg Expressions

```
pat = r"GET /ongoing/When/\d\d\dx/(\d\d\d\d/\d\d/\d\d/[^ .]+) "
search = re.compile(pat).search

matches = (search(line) for line in file("o10k.ap"))
```

- We can move the compilation of the reg exp out of the loop.
- We can eliminate lines that can't match "GET /ongoing/When/"
  - String matching can actually be sublinear (Boyer-Moore algorithm)
- Both of these create noticeable speedup.

Block Processing

- Instead of reading line by line we can read our data in blocks, and process each block.
- Reduces the number of system file calls that have a large amount of overhead.
- Makes our algorithm amenable to parallelism by assigning blocks to different threads.

Block Processing

- One major issue:

  We care about where lines end, and they're not going to end neatly along block boundaries. Need to deal with this issue somehow. Workable and doable, but adds complexity to our code.

Memory Mapping

- This technique maps the file directly into memory. This avoids future calls into the filesystem, leading to better performance.
- After this is done, no need to scan a file to a particular line, have random access.
- Allows “file access” to be concurrent via shared memory parallelism. Especially useful combined with block processing.

A bit of Code Philosophy

- Tim Bray consistently asks the question "what should programmer have to do to achieve parallelism/efficient IO?
- If memory mapping is the way to go for file reads, why don't we do that by default? (The Multix operating system, for instance, only provided memory mapping).
- For embarrassingly parallel situations (i.e. for loops where order doesn't matter), shouldn't things be dead simple for the programmer? (Why aren't OpenMP style pragmas or something similar more prevalent in parallel libraries?)

A bit of Code Philosophy

- Bray points out that most Wide Finder programmers sought to avoid/ reduce the use of regular expressions in their solutions. As a strong proponent of regular expressions, he clearly is made uncomfortable by this, and points out that many of the default reg exp engines in languages have lots of room for optimization. He even argues that a factor of 2-3 loss in performance may be worth the sacrifice if it saves the programmer time due to the simplicity of reg expressions.

Summary

- Since the industry is clearly moving to "wider" chips (more cores), we need to be shifting to mindsets that take advantage of this.
- IO is an important area, and the default line-by-line reading of a file seems to be very inefficient.

Summary

- Memory mapping combined with block processing provides good speedup and benefits from multiple cores.
- Combining the data at the end of your calculation is important too - the best way you do this will likely depend on the language/parallelism package combination you choose.

Any Questions?