15-853: Algorithms in the Real World

Parallel Algorithms: Lecture 1
Nested parallelism
Cost model
Parallel techniques and algorithms
Why Parallelism: Machines

Intel Xeon Eight-Core E5-2660 2.2GHz 8.0GT/s 20MB LGA2011 Processor without Fan, Retail BX80621E52660
by Intel

Be the first to review this item

Price: $137.99 & FREE Shipping

Get $40.00 off instantly: Your cost could be $97.99 upon approval for the Amazon.com Store Card. Learn more

Note: Not eligible for Amazon Prime. Available with free Prime shipping from other sellers on Amazon.

Only 13 left in stock.
Get it as fast as Thursday, Oct. 13.
Ships from and sold by Galactics.

- Model: Intel Xeon Processor E5-2660
- Core Count: 8
- Clock Speed: 2.2 GHz
- Cache: 20 MB
- Max Memory Bandwidth: 51.2 GB/s
- Socket: LGA2011

Used & new (29) from $47.95 + $6.44 shipping
Why Parallelism: Machines

by Rob Williams — Sunday, June 11, 2017

Intel 28-Core Xeon Platinum 8176 Dual-Socket Server Rocks Cinebench Benchmark With 112 Threads

Data Center » Servers

AMD does an Italian job on Intel, unveils 32-core, 64-thread 'Naples' CPU

Claims to be two times faster than Chipzilla's latest data centre processor

By Chris Mellor 8 Mar 2017 at 12:35

59 ☰ SHARE ▼
112 core 4-chip server

Up to 6TByte memory
Xeon Phi: Knights Landing (64 cores)
Qualcomm readies up 48-core Centriq 2400 ARM server chip
by Zak Killian — 12:51 PM on December 9, 2016

Maybe 2017 will be the year that ARM servers finally become a thing. After demoing a 24-core server chip a little more than a year ago, Qualcomm's Datacenter Technologies subsidiary has announced the Centriq 2400 CPU. This new chip is a 48-core ARMv8 processor based on a new in-house CPU core design called Falkor, and it's compliant with ARM's Server Base System Architecture specification. Earlier in the week, Qualcomm showed off the new hardware running "a typical datacenter application" comprising Linux with Java and Apache Spark.
4992 "cuda" cores

Nvidia Tesla K80 24GB GPU Accelerator passive cooling 2x Kepler GK210 900-22080-0000-000

by NVIDIA

Price: $4,295.95 + $11.55 shipping

Note: Not eligible for Amazon Prime.

In Stock.
Ships from and sold by eServer PRO.

Estimated Delivery Date: Aug. 27 - Sept. 1 when you choose Expedited at checkout.

- Nvidia Tesla K80 GPU: 2x Kepler GK210
- Memory size (GDDR5): 24GB (12GB per GPU)
- CUDA cores: 4992 (2496 per GPU)
- Memory bandwidth: 480 GB/sec (240 GB/sec per GPU)
- 2.91 Tflops double precision performance with NVIDIA GPU Boost - See more at: http://www.nvidia.com/object/tesla-servers.html#shtml.1F5LVwFq.dpuf

4 new from $4,135.00

Upgrading to a Solid-State Drive?
Learn how to install an SSD with Amazon Tech Shorts. Learn more
Up to 300K servers
LG Optimus 2X: first dual-core smartphone launches with Android, 4-inch display, 1080p video recording

Samsung Galaxy S IV to feature Exynos 28nm quad-core processor?
Written by Andre Yoskowitz @ 01 Nov 2012 18:02

It has been a few weeks but there is a new rumor regarding the upcoming Samsung Galaxy S IV.

According to reports, Samsung will pack next year's flagship device with its "Adonis" Exynos processor, a quad-core ARM 15 beast that uses efficient 28nm tech.

Samsung is supposedly still testing the application processor, but mass production is scheduled for the Q1 2013 barring any delays.
Lenovo Announces First Octa-Core Smartphone, The Vibe X2

10-core MediaTek Helio X20 is official

May 2015
MediaTek and TSMC trialing new 7nm smartphone processor with mad CPU core count

Posted: 09 Mar 2017, 06:53, by Luis D.

Mar 2017, 12 core
Parallelism is here... And Growing!

Parallelism for the Masses
“Opportunities and Challenges”

Andrew Chien, 2008-853
Outline (draft)

Concurrency vs. Parallelism
Concurrency example
Quicksort example
Nested Parallelism
  - fork-join and parallel loops
Cost model: work and span
Techniques:
  - Using collections: inverted index
  - Divide-and-conquer: merging, mergesort, kd-trees, matrix multiply, matrix inversion, fft
  - Contraction: quickselect, list ranking, graph connectivity, suffix arrays
Parallelism vs. Concurrency

Parallelism: using multiple processors/cores running at the same time. Property of the machine.

Concurrency: non-determinacy due to interleaving threads. Property of the application.

<table>
<thead>
<tr>
<th>Parallelism</th>
<th>Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>sequential</td>
</tr>
<tr>
<td>parallel</td>
<td>concurrent</td>
</tr>
<tr>
<td>serial</td>
<td>Traditional programming</td>
</tr>
<tr>
<td>parallel</td>
<td>Traditional OS</td>
</tr>
<tr>
<td>parallel</td>
<td>Deterministic parallelism</td>
</tr>
<tr>
<td>parallel</td>
<td>General parallelism</td>
</tr>
</tbody>
</table>
Nested Parallelism

nested parallelism =
  arbitrary nesting of parallel loops + fork-join

- Assumes no synchronization among parallel tasks except at joint points.
- Deterministic if no race conditions

Advantages:
- Good schedulers are known
- Easy to understand, debug, and analyze
Nested Parallelism: parallel loops

cilk_for (i=0; i < n; i++)
    B[i] = A[i]+1;

Parallel.ForEach(A, x => x+1);

B = {x + 1 : x in A}

#pragma omp for
for (i=0; i < n; i++)
    B[i] = A[i] + 1;

Cilk

Microsoft TPL (C#, F#)

Nesl, Parallel Haskell

OpenMP
Nested Parallelism: fork-join

cobegin {
   S1;
   S2;}

coinvoke(f1,f2)
Parallel.invoke(f1,f2)

#pragma omp sections
{
   #pragma omp section
   S1;
   #pragma omp section
   S2;
}

Dates back to the 60s. Used in dialects of Algol, Pascal

Java fork-join framework
Microsoft TPL (C#, F#)

OpenMP (C++, C, Fortran, ...)

Dates back to the 60s. Used in dialects of Algol, Pascal

Java fork-join framework
Microsoft TPL (C#, F#)

OpenMP (C++, C, Fortran, ...)

15-853
Nested Parallelism: fork-join

```plaintext
spawn S1;
S2;
sync;

cilk, cilk+

(exp1 || exp2)

Various functional languages

plet
  x = exp1
  y = exp2
in
  exp3

Various dialects of ML and Lisp
```

15-853
Serial Parallel DAGs

Dependence graphs of nested parallel computations are series parallel.

Two tasks are parallel if not reachable from each other. A data race occurs if two parallel tasks are involved in a race if they access the same location and at least one is a write.
Cost Model

Compositional:

Work: total number of operations
- costs are added across parallel calls

Depth: span/critical path of the computation
- Maximum span is taken across forked calls

Parallelism = Work/Depth
- Approximately # of processors that can be effectively used.
Combining costs

Combining for parallel for:

\[
\text{parfor } i \text{ in } [0:n] \\
\text{\hspace{1cm}} f(i); \\
\text{\hspace{1cm}} w = \exp(p\text{for ...}); \\
\text{\hspace{1cm}} W_{\text{pexp}}(\text{pfor ...}) = \sum_{i=0}^{n-1} W_{\exp}(f(i)) \quad \text{work} \\
\text{\hspace{1cm}} D_{\text{pexp}}(\text{pfor ...}) = \max_{i=0}^{n-1} D_{\exp}(f(i)) \quad \text{span}
\]
A Formal Model: The MP-RAM

Start with the sequential Random Access Machine (RAM) model

Add a fork(n) instruction:
1. Forks n identical child copies of the process, the i-th one has i in a special index register
2. Suspend the parent
3. When all children finish, restart the parent with 0 in the index register.

Purely nested. Can be viewed as series-parallel DAG. Work and Depth as usual
**MP-PRAM: ordering**

Any serialization (topological sort of DAG) is valid, and memory operations have normal semantics under this ordering.

Results can depend on ordering

**Definitions (pairs of instructions):**
- **Conflict**: access the same location, at least one of which is a write.
- **Concurrent**: are unordered in the DAG
- **Race**: conflict and concurrent

Race free programs always return the same result
Why Work and Span

Simple measures that give us a good sense of efficiency (work) and scalability (span).

Can schedule in $O(W/P + D)$ time on $P$ processors. This is within a constant factor of optimal.

Goals in designing an algorithm

1. Work should be about the same as the sequential running time. When it matches asymptotically we say it is work efficient.

2. Parallelism $(W/D)$ should be polynomial $O(n^{1/2})$ is probably good enough.
Example: Quicksort

function quicksort(S) =
if (#S <= 1) then S
else let
    a = S[rand(#S)];
    S1 = {e in S | e < a};
    S2 = {e in S | e = a};
    S3 = {e in S | e > a};
    R = {quicksort(v) : v in [S1, S3]};
in R[0] ++ S2 ++ R[1];

Partition
Recursive calls

How much parallelism?
Quicksort Complexity

Sequential Partition and appending
Parallel calls

Work = $O(n \log n)$

Span = $O(n)$

Parallelism = $O(\log n)$

Not a very good parallel algorithm

*All randomized with high probability*
Quicksort Complexity

Now let's assume the partitioning and appending can be done with:

- Work = $O(n)$
- Span = $O(\log n)$

but recursive calls are made sequentially.
Quicksort Complexity

Parallel partition
Sequential calls

Span $= O(n)$

Work $= O(n \log n)$
Parallelism $= O(\log n)$

Not a very good parallel algorithm

*All randomized with high probability*
Quickstart Complexity

Parallel partition
Parallel calls

Span = $O(lg^2 n)$

Span = $O(lg n)$

Work = $O(n \log n)$

Parallelism = $O(n/\log n)$

A good parallel algorithm

*All randomized with high probability*
Quicksort Complexity

Caveat: need to show that depth of recursion is $O(\log n)$ with high probability
Parallel selection

\{e \text{ in } S \mid e < a\};

S = [2, 1, 4, 0, 3, 1, 5, 7]
F = S < 4 = [1, 1, 0, 1, 1, 1, 0, 0]
I = addscan(F) = [0, 1, 2, 2, 3, 4, 5, 5]

where F
R[I] = S = [2, 1, 0, 3, 1]

Each element gets sum of previous elements.
Seems sequential?
Scan

- Sum: [2, 1, 4, 2, 3, 1, 5, 7]
- Recurse: [0, 3, 6, 4, 12]
- Interleave: [2, 7, 12, 18]

Result: [0, 2, 3, 7, 9, 12, 13, 18]
function addscan(A) =
if (#A <= 1) then [0]
else let
  evens = addscan(sums);
  odds = {evens[i] + A[2*i] : i in [0:#a/2]};
in interleave(evens,odds);

W(n) = W(n/2) + O(n) = O(n)
D(n) = D(n/2) + O(1) = O(log n)
Parallel Techniques

Some common themes in “Thinking Parallel”

1. Working with collections.
   - map, selection, reduce, scan, collect

2. Divide-and-conquer
   - Even more important than sequentially
   - Merging, matrix multiply, FFT, ...

3. Contraction
   - Solve single smaller problem
   - List ranking, graph contraction

4. Randomization
   - Symmetry breaking and random sampling