



Software Design Considerations for Multicore CPUs

SMP dilemmas revisited

0.0

- The traditional recipe for rabbit stew begins ... first catch one rabbit.
- The recipe for successful HPC begins
... first get the data to the functional units.

1.0 TOPIC

- We will discuss some performance issues with modern multi-core CPUs.
- Not dumbing things down very much: a modern CPU chip is a bunch of CPUs inside one chip.
- Without a loss of generality, our discussion will focus on higher end CPU chips.

2.0 Definitions

- It is necessary to define our domain of discourse. Generally, the focus will be on higher end chips and boards.
- 2.1 Our Cores
- 2.2 Our Chip
- 2.3 Our Board
- 2.4 Summary

2.1 Our Cores

- When someone says core, instead think "computational vehicle".
- Just as a smart car and a 747 are different vehicles, not all cores are equal or even equivalent.
- Generally GPU cores are much less capable.
- Our high end cores are largely *INDEED* of each other with
 - MMU - memory management - arbitrating both main memory and cache accesses
 - FADD/FMUL or a fused FMADDER
 - one or more ALUs
 - instruction decoding/processing logic
 - with typical Caches:
 - L1 cache 16-32 Kb each Inst / Data latency 3-6 cycles (seamless)
 - L2 cache 128-512 Kb Data latency 8-12 cycles

2.2 Our Chip

- typically 4-24+ cores (whatever is latest)
- with an L3 cache of 6-16 MB latency 25-50 cycles
- ** Note that at this level we have what used to be called a
- symmetric multi-processor (SMP).
- For whatever reasons, this term has fallen out of usage.

2.3 Our Boards

- a board typically has 1, 2 or 4 chips (totalling 4 – 80+ cores)
- all the chips share a large RAM (many GB) with a latency 100-200 cycles
- **** note that a server board with 2 or 4 chips is a
- Non-uniform Memory Architecture (NUMA) machine.

2.4 Summary

- A modern CPU chip is notionally an SMP, a symmetric multi-processor, with a large secondary store (RAM)
- A modern server board with 2 or 4 chips is notionally a NUMA, non-uniform memory architecture machine, with a large memory (RAM)
- Over the past 20 years we observe that while microseconds became nanoseconds, costs in the millions of dollars became thousands of dollars, and a room of equipment shrank to a board, **the relative component speeds, sizes, and costs have not changed so much in 20 years!**
- One caveat is the impact of super linear memory/storage improvements.

3.0 Amdahl's Law

- *Execution time for a fixed task using n processors is*
- $T(n) = S + P/n;$
- *where:*
- *S is the scalar (1 processor only part of a task)*
- *P is parallel (sharable part of a task)*
- The maximum speedup is $S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}$

Amdahl's Law (easy way)

- Where the total work for a process is $S+P$
- and S is the scalar (1 core) part of a process
- and P is the parallelizable part of a process
- The minimum execution time is S and the maximum speedup is $S/(S+P)$
- The execution time using n processing elements is: $T(n) = S + P/n$

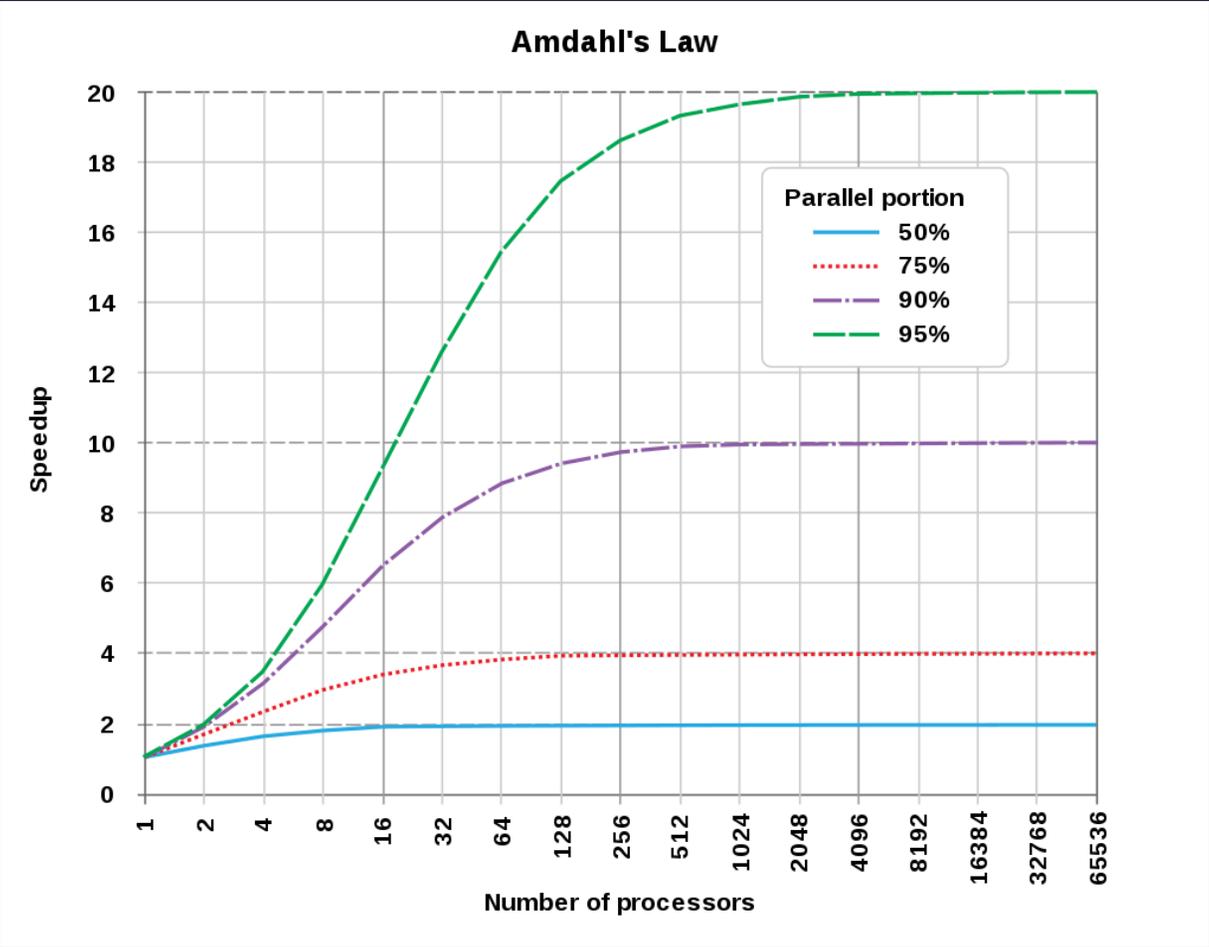
Example with

$S=10.0$ $P=90.0$

Max 10X speedup

n	T(n)	Speedup	S = 10.0	P/n = 90.0/n
1	100.0	1.0X	10.0	90.0
10	19.0	5.263X	10.0	9.0
20	14.5	6.896X	10.0	4.5
100	10.9	9.174X	10.0	0.90
1000	10.09	9.911X	10.0	0.09
10000	10.009	9.991X	10.0	0.009

Amdahl's Law (graph)



Amdahl (my way)

- Try to fit $T(n) = S + P/n$. Frequently doesn't work.
- A better equation is: $T(n) = S + A * P / [n - k(n)]$;
- A is the overhead of converting to parallel code. 1.05 is a good value
- $k(n)$ is the overhead of managing n processing elements, PEs.
- $k(n) = [0.25, 1.0]$ is a reasonable range for moderate n

“You can never be better than S”

- Instantiating for the sake of concreteness, we will take a 100 hour job where $S+P=100$, regardless of how much we reduce S.
- This is often realistic as reducing one time (scalar) work often requires compensating work in the parallel components.
- $T(n) = S + P / n$

S	T(16)		T(32)		T(64)	
10.0	15.625		12.813		11.406	
5.0	10.937		7.969		6.484	
2.0	8.125		5.063		3.531	
1.0	7.187		4.094		2.547	
0.5	6.71		3.609		2.055	
0.25	6.484		3.367		1.809	

Cache Latency Review

	Latency (cycles)	Typical size
L1	3-6	16-32KiB Inst+Data
L2	8-12	128-1024KiB
L3	25-50	6-18 MiB
RAM	100-200	8+ GiB

Cache Address Mapping

- We assume 1 MB cache buffer (2^{20}) and 8 KB (2^{13}) memory page sizes
- 1) cache is mapped based on *physical* memory addresses
- *** not the contiguous logical user address space
- A 1 MB cache assigns placement based on the low 20 bits of the physical address.
- Pathology: if all user physical pages are 1 MB aligned, then only a single 8 KB cache page would be used of the 1 MB cache.

Correlation function

- $C[i] = \text{sum} (A[i + j] * B[j]); \quad i=0:n, \quad j=0:k$
- Implemented to exceed 98% of machine theoretic (for reasonable n & k)
- *** by malicious alignment of A, B, C SLOWED by a factor of 96
- Both function invocations had
 - 1) identical floating point (FP) operation counts
 - 2) 100 % CPU utilization

I say cache, you think trash.

- Least Recently Used (LRU) is the most frequently used cache replacement algorithm.
- Consider a data cache of 100 words. If we are looping on 100 words or less, then the first iteration runs slowly as we load the data at (say) 100 cycles/word. Subsequent iterations access the data at 3-6 cycles/word.
- Consider a loop over 101 words. Words 1:100 load normally, then where does word 101 go? Word 1 is LRU and word 101 replaces it. Thus ends the first iteration.
- Next iteration starts with word 1. Load from memory and it replaces word 2 which is now LRU.
- Pathology: 100-200 cycles per word instead of 3-6 cycles.

Associative Cache

- N-way associative caches mitigate cache physical address pathology.
- As an example, consider a 4 MB cache implemented as 4x1MB physically addressed caches.
- A physical address is mapped by the low 20 bits (1 MB) and assigned to one of the 4 lines using LRU logic.
- This allows 4 different elements with the same 1 MB alignment to be concurrently cache resident.

Cache Line Coherence

- If one PE modifies a memory location, then any copies of that data “must” be updated in the other PEs and kept consistent with the RAM. This is called cache coherence. Coherence is atomically maintained at the cache line level (not at the byte). Cache lines are typically 16-128 bytes.
- Cache lines have a left and right half that are read and written consecutively. Consider a cache line of 8 words. The left half is words 0:3 and the right half is words 4:7. If you load word 5, the right half is loaded and then the left half is loaded immediately after.
- Pathology: Consider if PE1 keeps a counter in word 0, PE2 in word 1 ... PE8 in word 7. Because coherence is maintained at the line level, this generates a horrendous wave of memory activity likely slowing the machine to turtle speeds.

Cache Pathology Summary

- 1) There are as many cache replacement algorithms as can be imagined. Nearly all of them are too expensive to be feasible. The only real cost effective competitor to LRU is pseudo-random. The trade off is utilization versus replacement logic complexity.
- 2) **** You can not eliminate cache pathology ****

You can hide cache pathology or make it difficult to predict, but you can NEVER eliminate it.

I prefer to keep it where I can see it.

Finally: a kernel example Ver 1 is S L O W

```
// simple code for 1000 outputs
```

```
// sum 10 elements then multiply by a weight
```

```
C[i] = sum (A[j][i]); i=0:999, j=0:9
```

```
C[i] = w[i] * C[i]; i=0:999
```

Finally a kernel example Ver 2 is F A S T

```
// simple code for 1000 outputs  
// sum 10 elements then multiply by a weight  
// ver 2 is 100 times faster than Ver 1
```

```
C[i] = sum (A[i][j]); j=0:9, i=0:999  
C[i] = w[i] * C[i]; i=0:999
```

Finally a kernel example Ver 3 is F A S T E R

```
// simple code for 1000 outputs  
// sum 10 elements then multiply by a weight  
// Ver 3 is 10% faster than Ver 2
```

```
C[i] = sum (w[i]*A[i][j]); j=0:9, i=0:999
```

Analysis

- Ver 1 & 2 10000 FADDS 1000 FMULS
- Ver 3 10000 FADDS 10000 FMULS

- Ver 1 100 seconds
- Ver 2 1.1 seconds
- Ver 3 1.0 seconds

- WHY?

End run on S by Concurrency

- You've done what you can reduce S by improving efficiency and shifting work into the parallel parts.
- Consider a subtask with timing $T(n)=5+95/n$ minutes and $n=16$
- Task time is $5+95/16 = 10.937$ minutes
- about a 9X speedup for 16 PEs

Software Pipelining (simplified)

- 15 PEs will do the parallel part in $95/15 = 6.333$ minutes
- 1 PE does the scalar part in 5 minutes
- Given a set of similar tasks $A[0:n]$
- Defining $S(A[i])$ to be the scalar part and $P(A[i])$ is the parallelizable part

- Time per subtask 6.33 minutes

- Speedup 15.79X

Time (minutes)	PE 0 (Scalar work)	PE 1:15 (Parallel work)
0.00	S(A[0])	
6.33	S(A[1])	P(A[0])
12.67	S(A[2])	P(A[1])
19.00	S(A[3])	P(A[2])
		P(A[3])

Binding operations

- For our example $S=5$ $P=95$, suppose $S=5=3+2=\text{read}+\text{write}$
- Maximum concurrency is $\max (T[\text{read}], T[\text{write}], T(P/n))$
- Concurrency is:

Read $A[i+2]$

Compute $A[i+1]$

Output $A[i]$

- read is the binding operation at 3 minutes and $95/31 = 3.065$
- We can fully employ up to 32 PEs gives 32.62X speedup

Memory Organization

- Contiguous accesses
- Separate read/write buffers

Stranger and stranger

- These things really (albeit rarely) happen!
- 1) You drop the code on a 16 PE processor and the speedup is 150X.
- 2) The run time is roughly the *square* of the number of processors.

# processors	Run time (hours)
1	2.0
2	1.3
4	4.2
8	14.6
16	Forget it

Caveats

- Pro level HPC is harder than it looks
- Software pipelining is difficult to get right and harder to debug
- Multi PE debugging problems can be horrific,
like finding a needle in a matrix of haystacks

Combined the problems and life gets VERY interesting.

My codes already scales, Chris. (F U)

- Without any special consideration, perhaps it does.
- The most common situation is that the timing equation is
- $T(n) = s + 10 * P/n$ instead of $T(n) = S + P/n$
- Interesting prospect to have the task speeded up by a factor of 10
- --- then scale poorly.