**ECE459: Programming for Performance** Winter 2020

# Lecture 18 — Data & Task Parallelism

*Patrick Lam* *2019-10-31*

## Data and Task Parallelism

There are two broad categories of paralellism: data parallelism and task parallelism. An analogy to data parallelism is hiring a call center to (incompetently) handle large volumes of support calls, *all in the same way*. Assembly lines are an analogy to task parallelism: each worker does a *different* thing.

More precisely, in data parallelism, multiple threads perform the *same* operation on separate data items. For instance, you have a big array and want to double all of the elements. Assign part of the array to each thread. Each thread does the same thing: double array elements.

In task parallelism, multiple threads perform *different* operations on separate data items. So you might have a thread that renders frames and a thread that compresses frames and combines them into a single movie file.

We'll continue by looking at a number of parallelization patterns, examples of how to apply them, and situations where they might apply.

### Data Parallelism with SIMD

The "typical" boring standard uniprocessor is Single Instruction Single Data (SISD) but since the mid-1980s we've had more options than that. We'll talk about single-instruction multiple-data (SIMD) later on in this course, but here's a quick look. Each SIMD instruction operates on an entire vector of data. These instructions originated with supercomputers in the 70s. More recently, GPUs; the x86 SSE instructions; the SPARC VIS instructions; and the Power/PowerPC AltiVec instructions all implement SIMD.

**Code.** Let's look at an application of SIMD instructions.

```
void vadd(double * restrict a, double * restrict b, int count) {
  for (int i = 0; i < count; i++)
    a[i] += b[i];
}
```

Compiling this without SIMD on a 32-bit x86 (`gcc -m32 -march=i386 -S`) might give this:

```
loop:
  fldl  (%edx)
  faddl (%ecx)
  fstpl (%edx)
  addl  8, %edx
  addl  8, %ecx
  addl  1, %esi
  cmp   %eax, %esi
  jle   loop
```

We can instead compile to SIMD instructions (`gcc -m32 -march=prescott -mfpmath=sse`) and get something like this:

```
loop:
  movupd (%edx),%xmm0
  movupd (%ecx),%xmm1
  addpd  %xmm1,%xmm0
  movpd  %xmm0,(%edx)
  addl   16,%edx
  addl   16,%ecx
  addl   2,%esi
  cmp    %eax,%esi
  jle    loop
```

The *packed* operations (p) operate on multiple data elements at a time (what kind of parallelism is this?) The implication is that the loop only needs to loop half as many times. Also, the instructions themselves are more efficient, because they're not stack-based x87 instructions.

SIMD is different from the other types of parallelization we're looking at, since there aren't multiple threads working at once. It is complementary to using threads, and good for cases where loops operate over vectors of data. These loops could also be parallelized; multicore chips can do both, achieving high throughput. SIMD instructions also work well on small data sets, where thread startup cost is too high, while registers are just there to use.

We'll see a case study that uses SIMD (also known as vector instructions) in Lecture 19. In [Lem18], Daniel Lemire argues that vector instructions are, in general, a more efficient way to parallelize code than threads. That is, when applicable, they use less overall CPU resources (cores and power) and can even run faster.

## Case Study on SIMD: Stream VByte

"Can you run faster just by trying harder?"

The performance improvements we've seen to date have been leveraging parallelism to improve throughput. Decreasing latency is trickier—it often requires domain-specific tweaks.

Sometimes it's classic computer science: Quantum Flow found a place where they could cache the last element of a list to reduce time complexity for insertion from $O(n^2)$ to $O(n \log n)$.

`https://bugzilla.mozilla.org/show_bug.cgi?id=1350770`

We'll also look at a more involved example of decreasing latency today, Stream VByte [LKR18]. Even this example leverages parallelism—it uses vector instructions. But there are some sequential improvements, e.g. Stream VByte takes care to be predictable for the branch predictor.

**Context.** We can abstract the problem to that of storing a sequence of small integers. Such sequences are important, for instance, in the context of inverted indexes, which allow fast lookups by term, and support boolean queries which combine terms.

Here is a list of documents and some terms that they contain:

| docid | terms |
|---:|---|
| 1 | dog, cat, cow |
| 2 | cat |
| 3 | dog, goat |
| 4 | cow, cat, goat |

The inverted index looks like this:

| term | docs |
|---|---|
| dog | 1, 3 |
| cat | 1, 2, 4 |
| cow | 1, 4 |
| goat | 3, 4 |

Inverted indexes contain many small integers in their lists: it is sufficient to store the delta between a doc id and its successor, and the deltas are typically small if the list of doc ids is sorted. (Going from deltas to original integers takes time logarithmic in the number of integers).

VByte is one of a number of schemes that use a variable number of bytes to store integers. This makes sense when most integers are small, and especially on today's 64-bit processors.

VByte works like this:

- $x$ between 0 and $2^7 - 1$, e.g. $17 = 0b10001$: $0xxxxxxx$, e.g. $00010001$;
- $x$ between $2^7$ and $2^{14} - 1$, e.g. $1729 = 0b11011000001$: $1xxxxxxx/0xxxxxxx$, e.g. $11000001/00001101$;
- $x$ between $2^{14}$ and $2^{21} - 1$: $0xxxxxxx/1xxxxxxx/1xxxxxxx$;
- etc.

That is, the control bit, or high-order bit, is 0 if you have finished representing the integer, and 1 if more bits remain. (UTF-8 encodes the length, from 1 to 4, in high-order bits of the first byte.)

It might seem that dealing with variable-byte integers might be harder than dealing fixed-byte integers, and it is. But there are performance benefits: because we are using fewer bits, we can fit more information into our limited RAM and cache, and even get higher throughput. Storing and reading 0s isn't an effective use of resources. However, a naive algorithm to decode VByte also gives lots of branch mispredictions.

Stream VByte is a variant of VByte which works using SIMD instructions. Science is incremental, and Stream VByte builds on earlier work—masked VByte as well as VARINT-GB and VARINT-G8IU. The innovation in Stream VByte is to store the control and data streams separately.

Stream VByte's control stream uses two bits per integer to represent the size of the integer:

| | | | |
|---|---|---|---|
| 00 | 1 byte | 10 | 3 bytes |
| 01 | 2 bytes | 11 | 4 bytes |

Each decode iteration reads a byte from the control stream and 16 bytes of data from memory. It uses a lookup table over the possible values of the control stream to decide how many bytes it needs out of the 16 bytes it has read, and then uses SIMD instructions to shuffle the bits each into their own integers. Note that, unlike VByte, Stream VByte uses all 8 bits of each data byte as data.

For instance, if the control stream contains $0b1000\ 1100$, then the data stream contains the following sequence of integer sizes: $3, 1, 4, 1$. Out of the 16 bytes read, this iteration will use 9 bytes; it advances the data pointer by 9. It then uses the SIMD "shuffle" instruction to put the decoded integers from the data stream at known positions in the 128-bit SIMD register; in this case, it pads the first 3-byte integer with 1 byte, then the next 1-byte integer with 3 bytes, etc. Let's say that the input is `0xf823 e127 2524 9748 1b.. .... .... ....`. The 128-bit output is `0x00f8 23e1/0000 0027/2524 9748/0000/001b`, with the /s denoting separation between outputs. The shuffle mask is precomputed and, at execution time, read from an array.

The core of the implementation uses three SIMD instructions:

```
uint8_t C = lengthTable[control];
__m128i Data = _mm_loadu_si128 ((__m128i *) databytes);
__m128i Shuf = _mm_loadu_si128(shuffleTable[control]);
Data = _mm_shuffle_epi8(Data, Shuf);
databytes += C; control++;
```

**Discussion.** The paper [LKR18] includes a number of benchmark results showing how Stream VByte performs better than previous techniques on a realistic input. Let's discuss how it achieves this performance.

- control bytes are sequential: the processor can always prefetch the next control byte, because its location is predictable;
- data bytes are sequential and loaded at high throughput;
- shuffling exploits the instruction set so that it takes 1 cycle;
- control-flow is regular (executing only the tight loop which retrieves/decodes control and data; there are no conditional jumps).

We're exploiting SIMD, so this isn't quite strictly single-threaded performance. Considering branch prediction and caching issues, though, certainly improves single-threaded performance.

## References

[Lem18] Daniel Lemire. Multicore versus SIMD instructions: the "fasta" case study, 2018. Online; accessed 03-January-2018. URL: `https://lemire.me/blog/2018/01/02/multicore-versus-simd-instructions-the-fasta-case-study/`.

[LKR18] Daniel Lemire, Nathan Kurz, and Christoph Rupp. Stream vbyte: Faster byte-oriented integer compression. *Information Processing Letters*, 130(Supplement C):1 – 6, 2018. URL: `http://www.sciencedirect.com/science/article/pii/S0020019017301679`.