Programmation Parallèle (PRPA)

E. Carlinet {edwin.carlinet@lrde.epita.fr}

2021

EPITA Research & Development Laboratory (LRDE)

Agenda

Introduction to parallelism

Introducing parallelism

Study case

ILP and Caches

Vector processing (DLP)

Agenda

Agenda

1. *Introduction to parallelism*
2. *Instruction and data-level parallelism*
3. Thread level parallism
4. Modern C++ for concurrent programming
5. Data structure for concurrent programming

Introduction to parallelism

Why doing things in parallel ?

We want to have things *done* **quickly**.

Mobile dev.	Big data	Real time computing

- Mobile development: limited battery
- Big data analysis: huge data volume
- Real time system: has to provide a response in a bounded time

How to get things *done* quicker

1. Do less work
2. Do *some* work better (i.e. the one being the more time-consuming)
3. Do *some* work at the same time
4. Distribute work between different workers

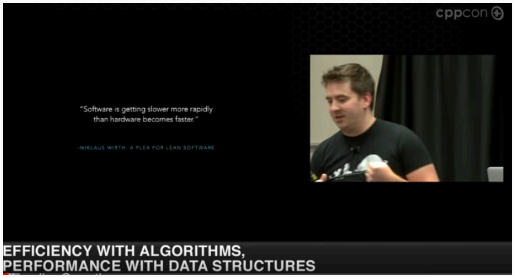
How to get things *done* quicker

1. Do less work
2. Do *some* work better (i.e. the one being the more time-consuming)
3. Do *some* work at the same time
4. Distribute work between different workers

- (1) Choose the most adapted **algorithms**, and avoid re-computing things
- (2) Choose the most adapted **data structures**
- (3,4) Parallelize

*Algorithms are for **efficiency***
*Data structures are for **performance***

Chandler Carruth
(Google C++ lead tech.)



Message

Concurrent programming appears in a process of **optimization** as a mean to get results faster.

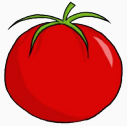
Should be considered after:

- benchmarking to identify bottlenecks
- when there is no other way to speed up bottlenecks

Introducing parallelism

The burger factory assembly line





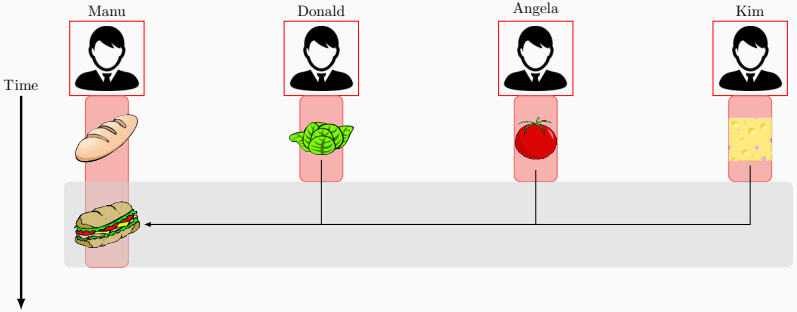
Get Bread & Cut	Get salad & Cut & Put in bread	Get tomatoe & Slice & Put in bread	Get cheese & Slice & Put in bread	Close

How to make several sandwiches as fast as possible ?

First strategy: multiple workers for one sandwich

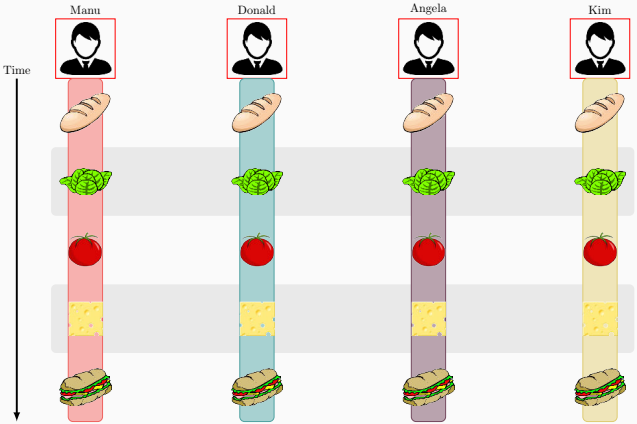
4 **super-workers** (4 CPU cores) collaborate to make 1 sandwich.

- Manu gets the bread and cuts and waits for the others
- Donald slices the salad
- Angela slices the the tomatoes
- Kim slices the cheeses



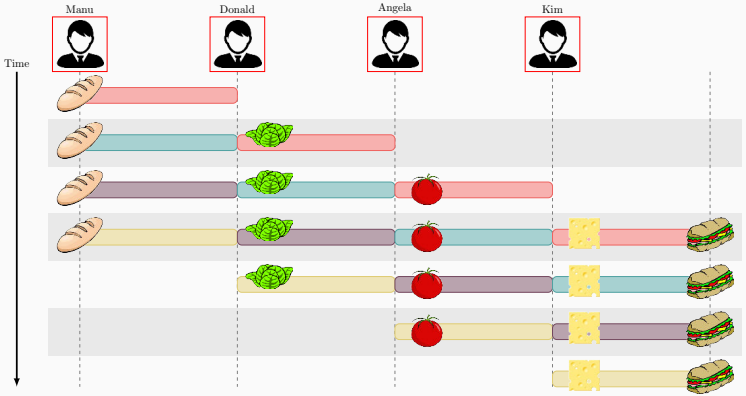
Second strategy: multiple workers for multiple sandwiches

- Manu makes sandwich 1
- Donald makes sandwich 2
- ...



Third strategy: multiple workers for one sandwich in a chain

- Manu cuts the bread
- Donald slices the salads
- Angela slices the tomatoes
- ...

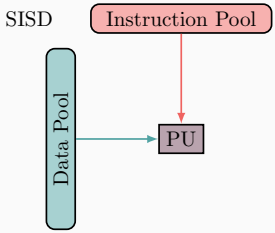


Fourth strategy: a worker makes several sandwitch

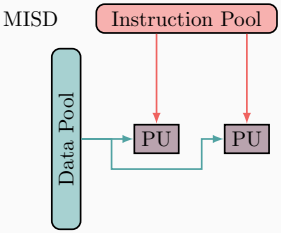
A worker has many arms and make 4 sandwiches at a time

Flynn's Taxonomy

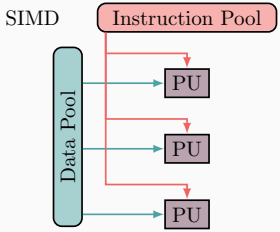
SISD = no parallelism



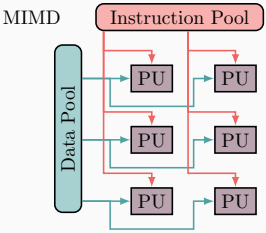
MISD = rare (fault-tolerant processor)



SIMD = same inst. on data (vector processing)



MIMD = usual parallel processor



Comparaison of the strategies (ideal case)

	1 sandwich	sandwich / min
4 workers for 1 sandwich	x4	x4
4 workers for 1 sandwich (pipeline)	x1	x4
4 workers for 4 sandwiches	x1	x4
1 worker for 1 sandwiches	x1	x4

Study case

Practical example (1/2)

The sum of string-encoded ints.

```
long sum_of_strings(const char* strings[], std::size_t n);
```

Practical example (1/2)

The sum of string-encoded ints.

```
long sum_of_strings(const char* strings[], std::size_t n);
```

Data generation:

```
const int N = 1e7;
vector<string>          data;
vector<const char*>     strings;
for (int i = 0; i < N; ++i)
{
  int number = randomint();
  data.push_back(to_string(number));
  strings.push_back(data[i].c_str());
}
shuffle(strings.begin(), strings.end());
```

Practical example (2/2)

The sum of string-encoded ints.

```
long myaoti(const char* str)
{
    long res = 0;
    bool is_negative = 1;
    if (*str == '-') {
        is_negative = true;
        str++;
    }
    for (; *str; str++)
        res = (res * 10) + (*str - '0');
    return is_negative ? -res : res;
}

long sum_string_vector_ref(const char* strings[], std::size_t n) {
    long sum = 0;
    for (std::size_t i = 0; i < n; ++i)
        sum += myatoi(strings[i]);
```

How to optimize this

Idea ?

How to optimize this

Idea ?

First bench and profile (demo time):

- Google Benchmark
- Perf
- Intel Studio

How to optimize this ?

Benchmark

```
-------------------------------------------------------------------
Benchmark                         Time           CPU Iterations
-------------------------------------------------------------------
bench/reference/1000              0 ms          0 ms      33696   45.8368M items/s
bench/reference/10000             0 ms          0 ms       1778   24.2266M items/s
bench/reference/100000           12 ms         12 ms         58   8.17872M items/s
bench/reference/1000000         121 ms        121 ms          6   7.89396M items/s
bench/reference/10000000       1215 ms       1211 ms          1   7.87629M items/s
```

How to optimize this ?

Profile

```
perf stat -d --delay 1600 ./bench --benchmark_filter=bench/reference/1000
```

```
Performance counter stats for './bench --benchmark_filter=reference':

   4050,301080      task-clock:u (msec)       #    0,503 CPUs utilized
             0      context-switches:u        #    0,000 K/sec
             0      cpu-migrations:u          #    0,000 K/sec
             2      page-faults:u             #    0,000 K/sec
13 957 938 750      cycles:u                  #    3,446 GHz                      (49,98%)
 5 365 733 100      instructions:u            #    0,38  insn per cycle           (62,50%)
   892 331 909      branches:u                #  220,312 M/sec                    (62,54%)
    27 022 813      branch-misses:u           #    3,03% of all branches          (62,52%)
   801 037 635      L1-dcache-loads:u         #  197,772 M/sec                    (62,51%)
   213 534 085      L1-dcache-load-misses:u   #   26,66% of all L1-dcache hits    (62,50%)
    98 515 722      LLC-loads:u               #   24,323 M/sec                    (49,98%)
    24 335 665      LLC-load-misses:u         #   24,70% of all LL-cache hits     (49,97%)

   8,051025821 seconds time elapsed
```

How to optimize this ?

Interpretation

Why the speed decreases as the number of element increases ?

How to optimize this ?

Interpretation

Why the speed decreases as the number of element increases ?

1000 items:

```
 6 262 464 828      L1-dcache-loads:u         #  698,141 M/sec                    (75,01%)
 1 238 441 970      L1-dcache-load-misses:u   #   19,78% of all L1-dcache hits    (75,02%)
   235 831 367      LLC-loads:u               #   26,291 M/sec                    (49,99%)
        32 622      LLC-load-misses:u         #    0,01% of all LL-cache hits     (49,98%)
```

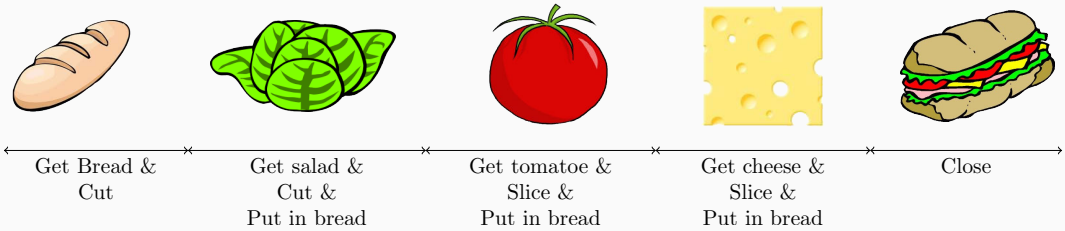
10 000 items:

```
4 099 865 672      L1-dcache-loads:u         #  370,714 M/sec                    (75,01%)
1 091 591 518      L1-dcache-load-misses:u   #   26,63% of all L1-dcache hits    (75,01%)
  523 902 195      LLC-loads:u               #   47,372 M/sec                    (49,99%)
    5 523 161      LLC-load-misses:u         #    1,05% of all LL-cache hits     (49,98%)
```

100 000 items and more:

```
1 014 861 523      L1-dcache-loads:u         #  127,503 M/sec                    (75,00%)
  277 847 531      L1-dcache-load-misses:u   #   27,38% of all L1-dcache hits    (75,00%)
  125 825 296      LLC-loads:u               #   15,808 M/sec                    (50,00%)
   65 018 190      LLC-load-misses:u         #   51,67% of all LL-cache hits     (50,00%)
```

The sandwich assembly line

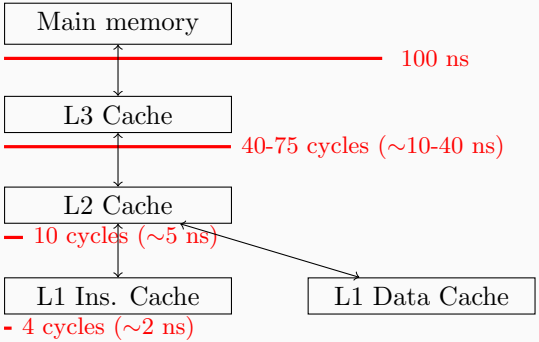


The **GET** operations is not as simple as you think:

- Get the tomatoe on the table (1s)
- Get the tomatoe in the basket near by the table (5s)
- Get the tomatoe in the storehouse (1 min)

The NUMA architecture

The memory access time depends on the memory location relative to the processor.



Making the code cache friendly

Data generation:

```
const int N = 1e7;
for (int i = 0; i < N; ++i)
{
  int number = randomint();
  data.push_back(to_string(number));
  strings.push_back(data[i].c_str());
}
```

```
- shuffle(strings.begin(), strings.end());
```

Do you think it will run faster ? How much ?

Making the code cache friendly

Contiguous data

Items	Time	CPU	Iterations	
1000	0 ms	0 ms	581258	79.1991M items/s
10000	0 ms	0 ms	55588	75.5323M items/s
100000	1 ms	1 ms	5389	73.4522M items/s
1000000	14 ms	14 ms	517	70.4993M items/s
10000000	136 ms	136 ms	52	70.1127M items/s

Random Data

Items	Time	CPU	Iterations	
1000	0 ms	0 ms	33696	45.8368M items/s
10000	0 ms	0 ms	1778	24.2266M items/s
100000	12 ms	12 ms	58	8.17872M items/s
1000000	121 ms	121 ms	6	7.89396M items/s
10000000	1215 ms	1211 ms	1	7.87629M items/s

Making the code cache friendly

For 10M items:

Now

```
 8 517 402 795      L1-dcache-loads:u         # 1053,734 M/sec
   371 840 023      L1-dcache-load-misses:u   #    4,37% of all L1-dcache hits
     3 466 740      LLC-loads:u               #    0,429 M/sec
     3 129 034      LLC-load-misses:u         #   90,26% of all LL-cache hits
```

Before

```
1 014 861 523      L1-dcache-loads:u         #  127,503 M/sec
  277 847 531      L1-dcache-load-misses:u   #   27,38% of all L1-dcache hits
  125 825 296      LLC-loads:u               #   15,808 M/sec
   65 018 190      LLC-load-misses:u         #   51,67% of all LL-cache hits
```

Much more L1 cache hit !

Making the code cache friendly

- How do you explain that `string` is cache friendy ?
- How do you explain the 90,26% LL-cache miss ?

Making the code cache friendly

- How do you explain that `string` is cache friendy ?
- How do you explain the 90,26% LL-cache miss ?

- C++ rocks ! Small buffer optimization for strings (no alloc => `char[16]`)
- Two much elements to be held in cache but few LL loads !

N° of items	1	10K	100K	1M	10M
L1 Cache Load	9G	9G	9G	9G	9G
L1 Cache Miss	4%	4%	4%	4%	4%
LL Cache Load	28K	2M	3.1M	3.5M	3.5M
LL Cache Miss	56%	1%	23%	84%	90%

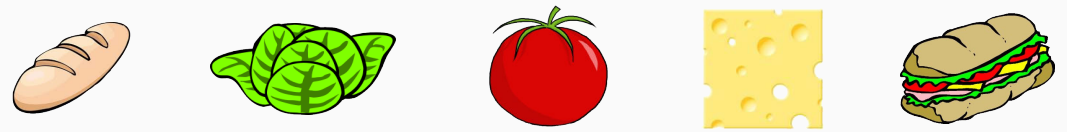
Take-home message

1. A program can be memory- or cpu- bandwith limited.
2. The first case occurs more often so use cache coherent data structures (if possible):
   - ~~list~~ -> `vector` or `deque`
   - ~~map~~, ~~unordered-map~~, ~~set~~, ~~unordered-set~~
3. Data and cache is of prime importance (even more in a multithreaded context -see later-).

- Parallelization is worth-less if the data layout is not adapted
- You can get more speed-up with a proper data layout

ILP and Caches

The assembly line revisited (1/2)



Pipeline:

Instructions	
1. Get Bread	7. Slice tomatoes
2. Cut Bread	8. Put tomatoes in bread
3. Get Salad	9. Get cheese
4. Cut Salad	10. Slice cheese
5. Put Salad in bread	11. Put cheese in bread
6. Get Tomatoes	12. Close

The assembly line revisited (1/2)

- We have dependency between some instructions e.g. `Get Tomatoe` before `Slice tomatoe`.
- We can reorder instructions as soon as dependancies are not broken:

Instructions	
1. Get Bread	7. Slice tomatoes
2. Cut Bread	8. Get Cheese
3. Get Salad	9. Put tomatoes in bread
4. Cut Salad	10. Slice cheese
5. Get Tomatoes	11. Put cheese in bread
6. Put Salad in bread	12. Close

Modern processors can reorder their pipeline; this is:

Out of order execution (OoO)

The assembly line revisited (2/2)

Suppose we have we have three arms usable in the same time:

- Unit 1. can grab products
- Unit 2. can cut/slice stuffs
- Unit 3. can do other stuffs

OoO becomes useful

The assembly line revisited (2/2)

Cycle	Load Unit	Slice/Cut Unit	Generic Unit
1	1. Get Bread		
2	3. Get Salad	2. Cut Bread	
3.	5. Get Tomatoes	4. Cut Salad	
4.	8. Get Cheese	7. Cut tomatoes	6. Put Salad in bread
5.		10. Cut cheese	9. Put Tomatoes in bread
6.			11. Put cheese in bread
7.			12. Close

The assembly line revisited (2/2)

Cycle	Load Unit	Slice/Cut Unit	Generic Unit
1	1. Get Bread		
2	3. Get Salad	2. Cut Bread	
3.	5. Get Tomatoes	4. Cut Salad	
4.	8. Get Cheese	7. Cut tomatoes	6. Put Salad in bread
5.		10. Cut cheese	9. Put Tomatoes in bread
6.			11. Put cheese in bread
7.			12. Close

This is **Instruction Level Parallelism (ILP)**

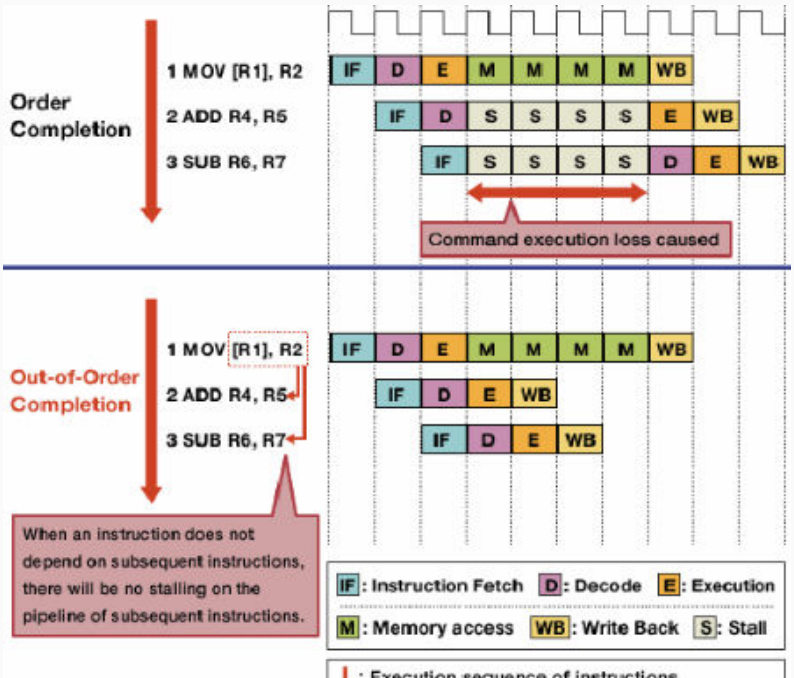
Here the ILP is `1.7 inst./cycle`

Hardware Parallelism: Pipeline

Pipeline RISC:

- IF: Instructions are serially fetched
- ID: Instructions are decoded (ID) and register are affected
- EX: Instruction Executed by an Unit (EX):
- MEM: LOAD or STORE from memory
- WB: Store a result in a register

Hardware Parallelism: Pipeline



Hardware Parallelism: Feeding the pipeline

1. Data-dependent instructions is not ILP-friendly.
2. Branching (JUMP/IF) in not ILP-friendly (a branch-miss implies a pipeline flush)

What the processor can do ?

1. Use out-of-order execution
2. Do speculative-branching and branch-prediction.

What you (or your compiler) can do:

1. You can break data-dependant instructions
2. You can avoid branching

ATOI is back

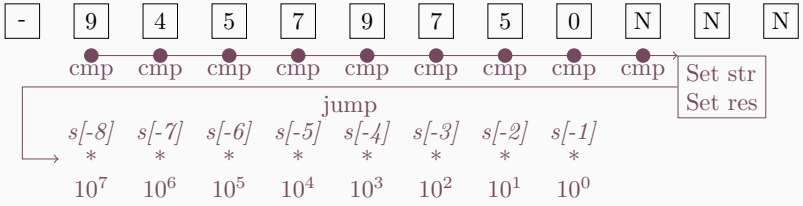
Note: strings are `NULL`-terminated, have `NULL` beyond the end, and have SBO.

Version 1 (remainder)

1. Test if negative
2. Advance char pointer till `NULL`, computing `x = x * 10 + k`
3. Test if negative and returns `-x` or `x`

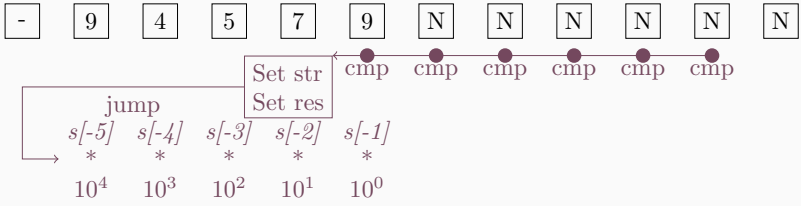
```
long res = 0;
bool negative = false;
if (*str == '-') { negative = true; str++; }

for (; *str; str++)
  res = res * 10 + (*str - '0');

return negative ? (-res) : res;
```

ATOI is back (Vers. 2)



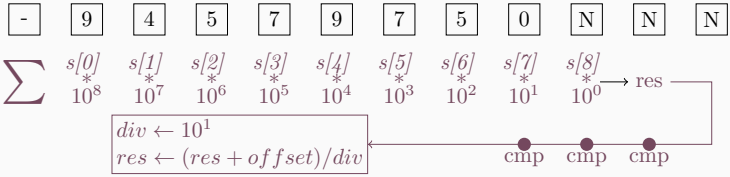

```
bool negative = false;  if (*str == '-') { negative = true; str++; }

long res;
if (str[1] == 0)      { str += 1;  res = '0' * -1L;          goto case_len_1;  }
else if (str[2] == 0) { str += 2;  res = '0' * -11L;         goto case_len_2;  }
else if (str[3] == 0) { str += 3;  res = '0' * -111L;        goto case_len_3;  }
...
else if (str[9] == 0) { str += 9;  res = '0' * -111111111L;  goto case_len_9;  }
else                  { str += 10; res = '0' * -1111111111L; goto case_len_10; }

case_len_10: res += str[-10] * (long)1e9;
case_len_9:  res += str[-9] *  (long)1e8;
...
```

ATOI is back (Vers. 3)



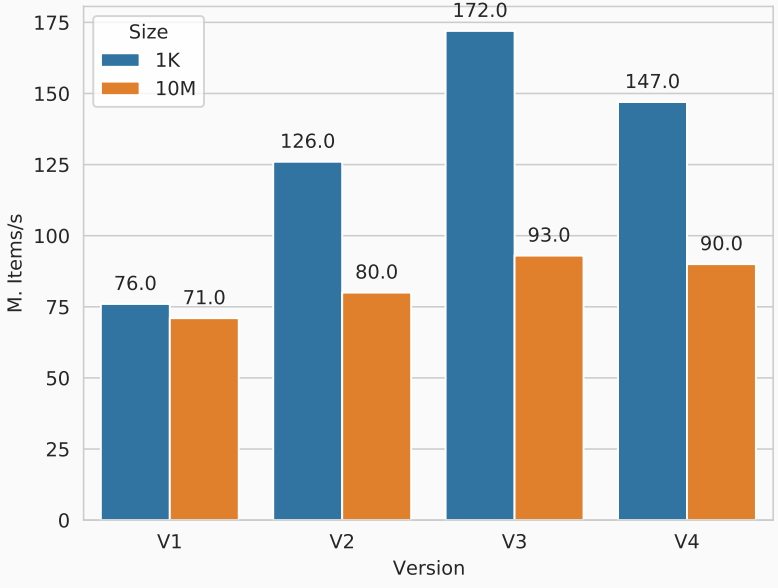

```
bool negative = false; if (*str == '-') { negative = true; str++; }

long res;
if (str[9] != 0)      { str += 10; res = '0' * -1111111111L; goto case_len_10; }
else if (str[8] != 0) { str += 9;  res = '0' * -111111111L;  goto case_len_9;  }
else if (str[7] != 0) { str += 8;  res = '0' * -11111111L;   goto case_len_8;  }
...
else if (str[1] != 0) { str += 2;  res = '0' * -11L;         goto case_len_2;  }
else                  { str += 1;  res = '0' * -1L;          goto case_len_1;  }

case_len_10: res += str[-10] * (long)1e9;
case_len_9:  res += str[-9] *  (long)1e8;
...
```

ATOI is back (Vers. 4)




```
bool negative = false;
if (*str == '-') { negative = true; str++; }

long res = 0;
res += str[8] * (long)1e0;
res += str[7] * (long)1e1;
...
res += str[0] * (long)1e8;

if (str[9] != 0)       { res = res * 10 + str[9]; res += ('0' * -1111111111L); }
else if (str[8] != 0) { res = (res + ('0' * -111111111L)) / 1e0; }
else if (str[7] != 0) { res = (res + ('0' * -11111111L)) / 1e1; }
...
else if (str[1] != 0) { res = (res + ('0' * -11L)) / 1e7; }
else                  { res = (res + ('0' * -1L)) / 1e8; }
return negative ? -res : res;
```

Which one is the fastest ?



The fastest version is **from 31% to 2X** faster than the naïve one. How do you explain that ?

Hint 1

Algorithm 1 computes:

```
((((s[0] - '0') * 10) + (s[1] - '0')) * 10 + (s[2] - '0')) * 10 + (s[3] - '0')...
```

We have data dependency for each multiplication.

The others compute:

```
k0 * s[0] + k1 * s[1] + k2 * s[2]...
```

No dependencies (10%-60% speed up).

Hint 2

Numbers are uniformly sampled on 32bits ints.

Int Range	Int length	Propability
$1e^9 - UINT31\_MAX$	10	53%
$1e^8 - (1e^8 - 1)$	9	42%
$0 - (1e^8 - 1)$	<9	5%

- Algorithm 2 tests length increasingly: 7+ comparisons in 95% of cases
- Algorithm 3/4 test length decreasingly: 1 or 2 comparisons in 95% of cases

Hint 2

Numbers are uniformly sampled on 32bits ints.

Int Range	Int length	Propability
$1e^9 - UINT31\_MAX$	10	53%
$1e^8 - (1e^8 - 1)$	9	42%
$0 - (1e^8 - 1)$	<9	5%

- Algorithm 2 tests length increasingly: 7+ comparisons in 95% of cases
- Algorithm 3/4 test length decreasingly: 1 or 2 comparisons in 95% of cases

Even if branch prediction of algo 2 will be true in most cases, it remains much less tests (10% - 15% speed up).

	Algorithm 2	Algorithm 3	Algorithm 4
Branches / Iter	128 946 625	58 994 570	63 804 420
Branch-misses	3.52%	8.03%	7.5%

About Version 4

Can you figure out why the case `length 10` is handle differently ?

```
if (str[9] != 0)      { res = res * 10 + str[9]; res += ('0' * -1111111111L); }
else if (str[8] != 0) { res = (res + ('0' * -111111111L)) / 1e0; }
else if (str[7] != 0) { res = (res + ('0' * -11111111L)) / 1e1; }
...
else if (str[1] != 0) { res = (res + ('0' * -11L)) / 1e7; }
else                  { res = (res + ('0' * -1L)) / 1e8; }
```

About Version 4

Can you figure out why the case `length 10` is handle differently ?

```
if (str[9] != 0)      { res = res * 10 + str[9]; res += ('0' * -1111111111L); }
else if (str[8] != 0) { res = (res + ('0' * -111111111L)) / 1e0; }
else if (str[7] != 0) { res = (res + ('0' * -11111111L)) / 1e1; }
...
else if (str[1] != 0) { res = (res + ('0' * -11L)) / 1e7; }
else                  { res = (res + ('0' * -1L)) / 1e8; }
```

- We avoid the integer division that would be required in 47% of cases
- We favor an integer multiplication required in 53% of cases, the division in only required in 5% of cases.

Conclusion about ILP

You can get huge speed-up by organizing your program to favor ILP:

- Avoid data dependancy
- Avoid branches as much as possible
- Organize branches w.r.t the data:
  - Profiled-guided optimization can do that
  - Some compiler annotation can be used .e.g. `__builtin_expect(...)`

Vector processing (DLP)

Processors are super-scalar

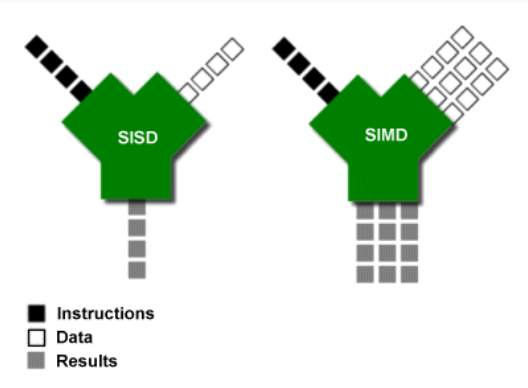
- Processors have Intructions-Level Parallism with OoO.
- Most processors provide SIMD extensions:

Processor	Instructions	Register size
x86	SSE	128
	AVX(2)	256
	AVX512	512
ARM	NEON	128
IBM POWER	AltiVec	128
	QBX	256
SPARC	HPC-ACE	128
	HPC-ACE2	256

Note that GPU follows the SIMD model

GPU programming is out-of-scope, but structuring a program for SIMD is similar to structuring for GPU.

Programming with SIMD



1. We must perform the same set of instructions for all data.
2. Data must be contiguous

Consequences:

1. Code should be branchless (but branches can be rewritten with branchless instructions)
2. You should prepare your data for SIMD

Programming with SIMD

- Explicit SIMD using SIMD intrasic instructions
  `<mmintrin.h>` MMX, `<xmmintrin.h>` SSE, `<emmintrin.h>` SSE2, `<pmmintrin.h>` SSE3. Ref

```
__m64 _mm_abs_pi16 (__m64 a)
__m64 _mm_abs_pi32 (__m64 a)
__m64 _mm_abs_pi8 (__m64 a)
```

- Explicit SIMD
  - Use SIMD types from compiler extensions and use normal operators.

```
typedef float float4 __attribute__((ext_vector_type(4)));
typedef float float2 __attribute__((ext_vector_type(2)));

float4 x, y;
x += y
```

Programming with SIMD

- Use SSE abstractions libraries
- Add OpenMP instructions

```
#pragma omp simd
for (int i = 0; i < N; ++i)
    a[i] += s * b[i];
```

- **Trust the compiler and its auto-vectorization** but help it.

The aliasing issue

```
void multadd(const int a[], const int b[], int c[], std::size_t n)
{
  for (std::size_t i = 0; i < n; ++i)
     c[i] = b[i] * a[i] + c[i];
}
```

Suppose someone is calling with:

```
int* tab = new int[n+2];
multadd(tab, tab + 1, tab + 2);
// or
multadd(tab + 2, tab + 1, tab);
```

The compiler has to generate valid code for this case (no vectorization possible).

The aliasing issue

```
void multadd(const int a[], const int b[], int c[], std::size_t n)
{
  for (std::size_t i = 0; i < n; ++i)
     c[i] = b[i] * a[i] + c[i];
}
```

Suppose someone is calling with:

```
int* tab = new int[n+2];
multadd(tab, tab + 1, tab + 2);
// or
multadd(tab + 2, tab + 1, tab);
```

The compiler has to generate valid code for this case (no vectorization possible).

Tell the compiler that pointers do no alias:

```
void multadd(const int* __restrict a, const int* __restrict b,
             int* __restrict c, std::size_t n)
```

The alignment issue

- On some processor, loading a chunk of data is cheaper if the address is aligned to the vector size (some just do not allow unaligned loads).
- Avoid loading too much cachelines

Force alignment allocation:

```
void* ptr = std::aligned_alloc(16, n); //128-bits alignment
```

Tell the compiler that pointers are aligned:

```
__assume_aligned(ptr, 16); // Intel
```

or with OpenMP:

```
#pragma omp simd aligned(ptr : 16)
```

The function call issue

```
for (std::size_t i = 0; i < n; ++i)
    a[i] = foo(5, a[i])
```

Solution:

- Mark the function inline so that the compiler can try inlining
- Generate simd version of the function with OpenMP:

```
#pragma omp declare simd uniform(a) linear(1: b)
void foo(float a, const int* b);
```

Compiler auto-vectorization

Even if you do not tell the compiler about *aliasing* and *alignment* it will try to vectorize:

Loop vectorizer:

1. Test if pointers can alias, if so go to 4.
2. Run the scalar loop till reaching vector alignment
3. Run the vectorized loop till there are not more enough items
4. Run the scalar loop with remaining items

Atoi is back (again again). How to vectorize `atoi`?

Version 4:

```
long res = 0;
res += str[8] * (long)pow10[0];
res += str[7] * (long)pow10[1];
...
res += str[1] * (long)pow10[7];
res += str[0] * (long)pow10[8];

if (str[9] != 0)      { res = res * 10 + str[9]; res += ('0' * -1111111111L); }
else if (str[8] != 0) { res = (res + ('0' * -111111111L)) / pow10[0]; }
else if (str[7] != 0) { res = (res + ('0' * -11111111L)) / pow10[1]; }
...
else if (str[1] != 0) { res = (res + ('0' * -11L)) / pow10[7]; }
else                  { res = (res + ('0' * -1L)) / pow10[8]; }
```

Atoi is back (again)

```
long res = 0;
res += str[8] * (long)pow10[0];
res += str[7] * (long)pow10[1];
...
res += str[1] * (long)pow10[7];
res += str[0] * (long)pow10[8];
```

Becomes:

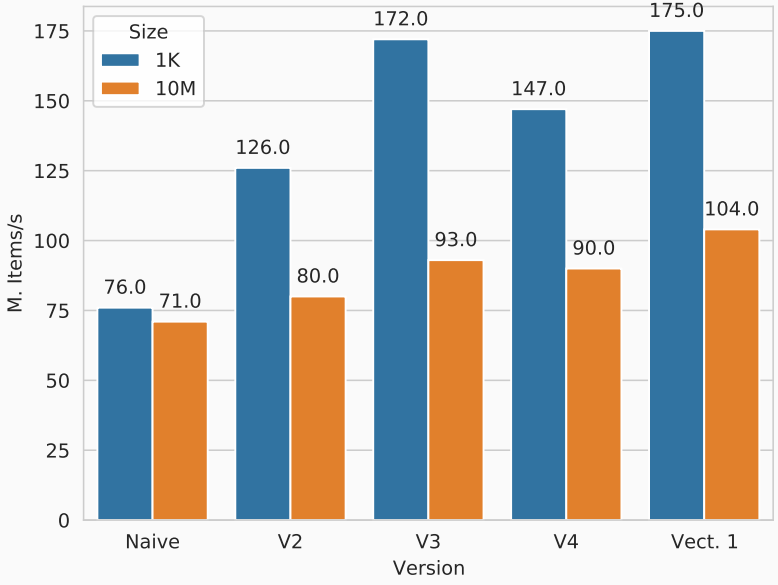
```
using vec_uint_t = unsigned int __attribute__((ext_vector_type(8)));
using vec_char_t = char  __attribute__((ext_vector_type(8)));
vec_char_t y = { str[1], str[2], str[3], ..., str[8]};
vec_uint_t x = { pow10[7], pow10[6], pow10[5], ... pow10[0] };
x *= __builtin_convertvector(y, vec_uint_t);
x = _mm256_hadd_epi32(x, x); // Horizontal reduction
x = _mm256_hadd_epi32(x, x); // Horizontal reduction
long res = x[0] + x[4] + str[0] * (long)pow10[8];

...
```

Remark

- No compiler has been able to vectorize this way:
  This is SLP (superword-level parallelism) vectorization, not a loop vectorization.
  It combines similar independent instructions into vector instructions.

- **Yuck !!** Hard to code, compiler intrinsics, hard for maintenance.

What about performances



Again a 10% speed-up but again... **yuck** !

What about doing real LOOP vectorization

Current data layout:

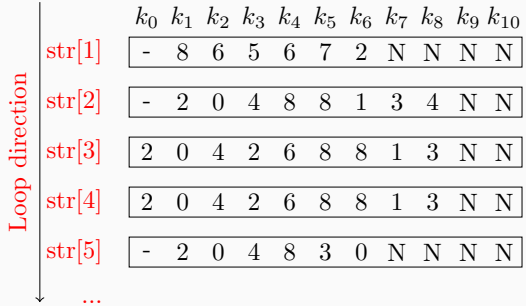


Figure 1 – The data layout

If we want a loop oriented vectorization, we must:

1. Change the data layout orientation
2. Do the same operations for each character of the strings

Loop oriented data layout

- We replace the array of 16 chars (`char[n][16]`) by 16 arrays of chars (`char[16][n]`)
- We force the first character to be `'-'` or `'+'`
- We left-padded strings with `'0'`

Before:

Loop direction ↓

	k_0	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9	k_{10}
str[1]	-	8	6	5	6	7	2	N	N	N	N
str[2]	-	2	0	4	8	8	1	3	4	N	N
str[3]	2	0	4	2	6	8	8	1	3	N	N
str[4]	2	0	4	2	6	8	8	1	3	N	N
str[5]	-	2	0	4	8	3	0	N	N	N	N
...											

Now:

Loop direction ↓

	k_0	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9	k_{10}
str[1]	-	0	0	0	0	8	6	5	6	7	2
str[2]	-	0	0	2	0	4	8	8	1	3	4
str[3]	+	0	2	0	4	2	6	8	8	1	3
str[4]	+	0	2	0	4	2	6	8	8	1	3
str[5]	-	0	0	0	0	2	0	4	8	3	0
...	...	...	...	...	...	...	...	...	...	...	...

The operations are the same:

$$res = \sum_{j=0}^{n} \sum_{i=1}^{10} str[11-i][j] * 10^{10-i}$$

Version 1

```
long res = 0;
#pragma clang loop vectorize(disable)
for (std::size_t i = 0; i < n; ++i)
{
  unsigned int current = 0;
  for (int k = 1; k < MAX_INT32_STRING_LENGTH; ++k)
    current += (str[k][i] - '0') * pow10[MAX_UINT31_STRING_LENGTH - k];

  long val = (str[0][i] == '-') ? -(long)current : (long)current;
  res += val;
}
return res;
```

Version 2 (hand-made explicit vectorization)

```
using v8i = int __attribute__((ext_vector_type(8)));
using v8c = char __attribute__((ext_vector_type(8)));
long res = 0;
for (; n >= 8; n -= 8)
{
  v8i local_sum(0);
  for (int k = 1; k < MAX_INT32_STRING_LENGTH; ++k)
  {
    v8c x0 = *(v8c*)(str[k]);
    v8i x = __builtin_convertvector(x0, v8i);
    x -= v8i('0');
    x *= v8i(-pow10[MAX_UINT31_STRING_LENGTH-k]);
    local_sum += x;
  }
  // Negate
  v8c x = *((v8c*)str[0]);
  auto mask = x - v8c(44);
  v8i mask_v8i = __builtin_convertvector(mask, v8i);
  local_sum = _mm256_sign_epi32(local_sum, mask_v8i);

  // Horizontal reduction
  for (int k = 0; k < 8; ++k) res += local_sum[k];

  // Advance pointers
  for (int k = 0; k < MAX_INT32_STRING_LENGTH; ++k) str[k] += 8;
}
// Fallback to normal loop for remaining chars
```

Benchmarks

1. Version 1 with `#pragma clang loop vectorize(disable)`
2. Version 2 with hand-made explicit vectorization
3. Version 1 with `#pragma clang loop vectorize(enable)` (auto-vectorizer)

```
atoi.cpp:230:3: remark: the cost-model indicates that interleaving is not benefici
  for (std::size_t i = 0; i < n; ++i)
  ^
atoi.cpp:230:3: remark: vectorized loop (vectorization width: 4, interleaved count
```

Benchmarks

1. Version 1 with `#pragma clang loop vectorize(disable)`
2. Version 2 with hand-made explicit vectorization
3. Version 1 with `#pragma clang loop vectorize(enable)` (auto-vectorizer)

```
atoi.cpp:230:3: remark: the cost-model indicates that interleaving is not benefici
  for (std::size_t i = 0; i < n; ++i)
  ^
atoi.cpp:230:3: remark: vectorized loop (vectorization width: 4, interleaved count
```

Results

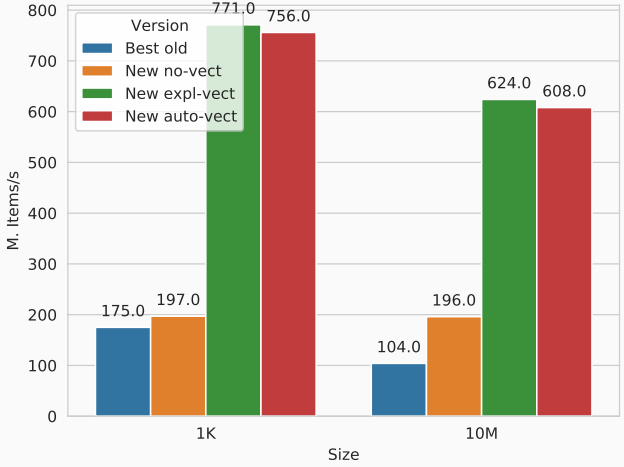
Number of items	1K	10K	100K	1M	10M
Version 1 (no vect.)	197 M/s	199 M/s	200 M/s	194 M/s	196 M/s
Version 2 (explicit vect.)	771 M/s	782 M/s	765 M/s	651 M/s	624 M/s
Version 1 (auto vect.)	756 M/s	745 M/s	721 M/s	637 M/s	608 M/s

Conclusion

- Vectorization brings a 3X to 4X speed-up
- Hand-made vectorization brings only 2% speed-up: Compilers rock

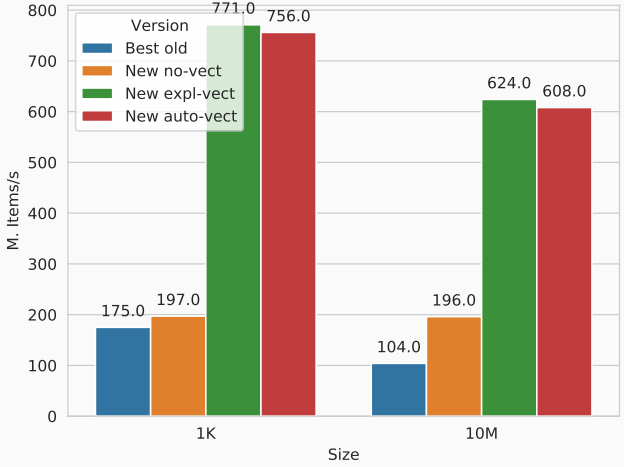
Conclusion

- Vectorization brings a 3X to 4X speed-up
- Hand-made vectorization brings only 2% speed-up: Compilers rock



Conclusion

- Vectorization brings a 3X to 4X speed-up
- Hand-made vectorization brings only 2% speed-up: Compilers rock



So far, we have only used a single CPU :)

Take-home message

Structuring data for processing is always the key !

Questions ?