Programmation Parallèle (PRPA)

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

2021

EPITA Research & Development Laboratory (LRDE)

Agenda

Thread-Level Parallelism

Parallelism in C++

C++ APIs for multi-threadings

Agenda

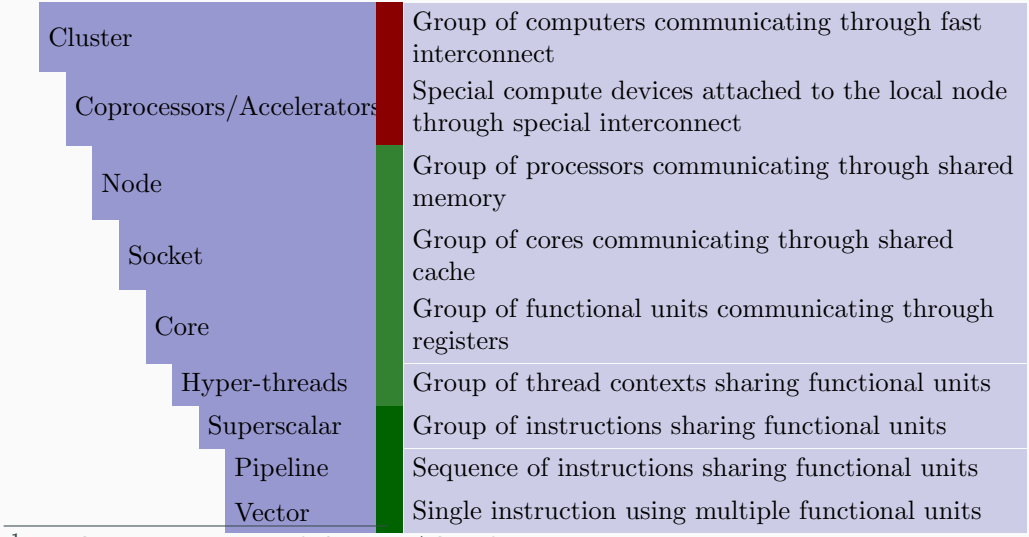
Agenda

1. Introduction to parallelism
2. Instruction and data-level parallelism
3. Thread level parallism
4. Parallel Design Patterns (with TBB)
5. C++ Memory model
6. Data structure for concurrent programming

Thread-Level Parallelism

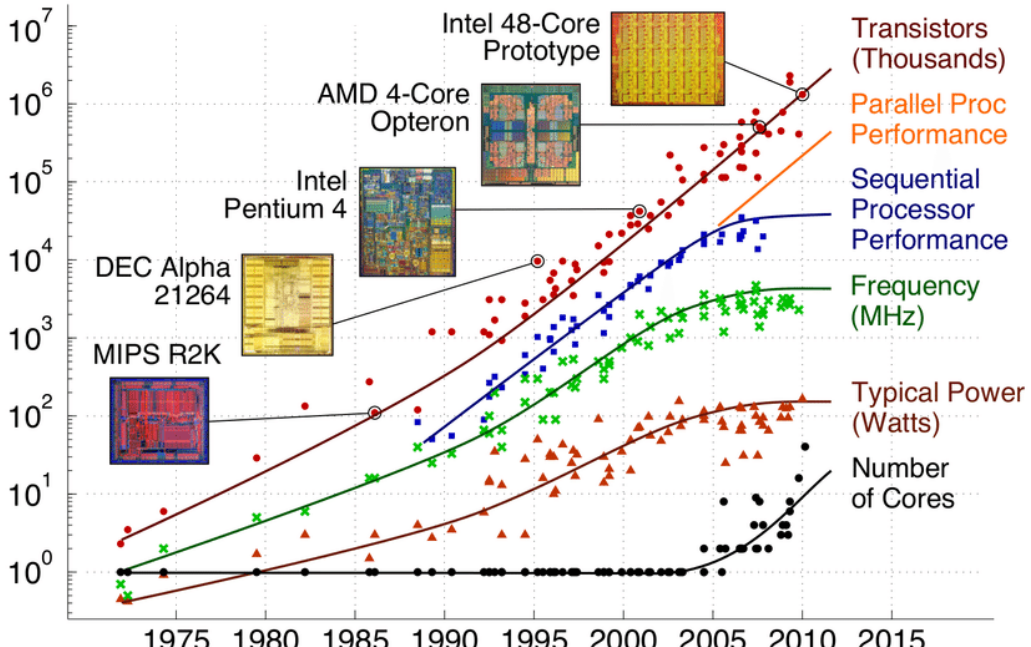
Remainder

Different level of parallelism[1]

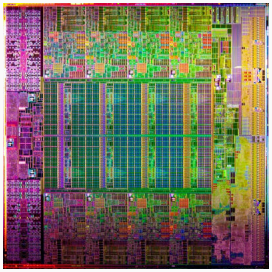


[1] From SIMD Vectorization with OpenMP / C. Terboven

Motivation

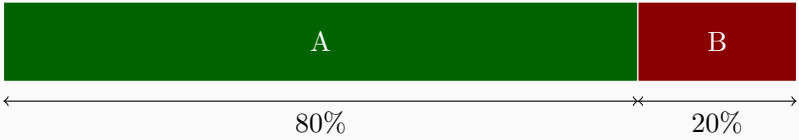

Intel 48-Core Prototype
AMD 4-Core Opteron
Intel Pentium 4
DEC Alpha 21264
MIPS R2K
Transistors (Thousands)
Parallel Proc Performance
Sequential Processor Performance
Frequency (MHz)
Typical Power (Watts)
Number of Cores
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1975 1980 1985 1990 1995 2000 2005 2010 2015

Motivation



	64bits Intel Xeon	Xeon 5100 series	Xeon 5500 series	Xeon 5600 series	Xeon E5 2600 series	Xeon Phi 7120P
Frequency	3.6 Ghz	3.0 Ghz	3.2 Ghz	3.3 Ghz	**2.7 Ghz**	**1.238 Ghz**
Cores	1	2	4	6	**12**	**61**
Threads	2	2	8	12	**24**	**244**
SIMD Width	128 bits (2 clocks)	128 bits (1 clock)	128 bits (1 clock)	128 bits (1 clock)	**256 bits (1 clock)**	**512 bits(1 clock)**

Optimizing with parallel programming

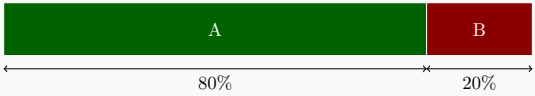


A program has two parts:

- `A` takes 80% of the wall time
- `B` takes 20% of the wall time

`A` can be parallelized, what is the maximum speed-up of the program ?

Amdahl's Law



- Let N be the number of threads
- Let B be the part of serial execution (ratio)
- Let T(N) be the execution time for N threads
- $t_1 = T(1)$ (time for a serial execution)

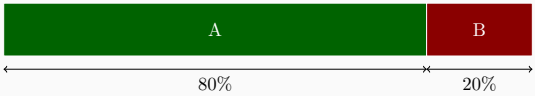
The new time processing time is:

$$T(N) = \underbrace{B.t_1}_{\text{Sequential time}} + \underbrace{\frac{(1-B)}{N} t_1}_{\text{Parallel time}}$$

The speedup S(N) is:

$$S(N) = \frac{t_1}{T(N)} = \left(B + \frac{1-B}{N} \right)^{-1}$$

Amdahl's Law



- With 4 threads, we have:

$$S(4) = (0.2 + 0.8/4)^{-1} = 2.5$$

- With an unlimited number of threads:

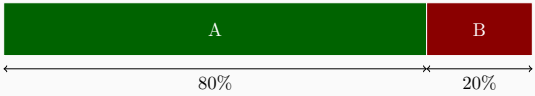
$$S(\infty) = \lim_{N \to \infty} (B + \frac{1 - B}{N})^{-1} = B^{-1} = 5$$

Amdahl's Law limitations

Strong scalability Augmenting the number of ressources = reducing time to process 1 job

- It applies only to the cases at *fixed problem size*
- If problem augments with the size of the data process, not applicable

Gustafson's law



Expressed in term of *workload* i.e. number of data processed in a *fixed execution time*.

- Let N be the number of threads
- Let B be the part of serial execution (ratio)
- Let W(N) be the supported workload for N threads
- W_1 is the workload before the ressources improve and have an execution time T

Workload for N threads:

$$W(N) = \underbrace{N.(1-B).W_1}_{\text{Parralel work}} + \underbrace{B.W_1}_{\text{Seq. Work}}$$

The speed up S(N) is then:

$$S(N) = \frac{T.W(N)}{T.W_1} = N.(1-B) + B$$

$$S(\infty) = \infty$$

Which rules apply

Weak scalability Augmenting the number of ressources allows an higher workload

Which rules apply

Weak scalability Augmenting the number of ressources allows an higher workload

- B does not depend on the dataset size (e.g. program startup)
  -> Gustafson's law
- B does depend on the dataset size (e.g. reduction operation)
  -> Amdahl's Law

- Amdahl's Law focus on *latency*
- Gustafson's law focuses on *throughput*

Latency vs. Throughput

- **Latency** (Délai): time to finish a task
- **Throughput** (Débit): number of tasks in a fixed time

Example.

- Car: speed = 100km/h, capacity 5
- Bus: speed = 80km/h, capacity 15

Transporting passengers 100km

	Latency (min)	Throughput (Passengers/Hour)
Car	60	5
Bus	75	12

Comparing performance

A is X times faster than B:

- Latency(A) = Latency(B) / X
- Throughput(A) = Throughput(B) * X

A is X% faster than B

- Latency(A) = Latency(B) / (1 + X)
- Throughput(A) = Throughput(B) * (1 + X)

i.e 100ms -> 75ms = 33% faster (not 25%)

Latency vs. Throughput

- Car: speed = 100km/h, capacity 5
- Bus: speed = 80km/h, capacity 15

	Latency (min)	Throughput (Passengers/Hour)
Car	60	5
Bus	75	12

Latency

- Car is `1.26` times faster than bus
- Car is 25% faster than bus

Throughput

- Bus is `2.4` times faster than car
- Bus is 140% faster than bus

Conclusion

- Single data (single same task) -> think **Latency**
- Multiple data stream -> think **throughput**

When you have a 20MPix image you're not interested in the time to process 1 pixel => MPix/s

Parallelism in C++

std::thread

`t = std::thread(fun, args...)`

- Thread `t` is spawn at construction with a function to be executed
- Before object destruction, a thread must be:
  - **joined** `t.join()`
  - or **detached** `t.detach()`

- **Join**: Blocks the current thread until the thread finishes its execution.
- **Detach**: Permits the thread to execute independently

(Note than you cannot return value from threads, passing by ref required).

```cpp
void f1(int n);
void f2(int& n);

int main()
{
  std::thread t1(f1, n + 1); // pass by value
  std::thread t2(f2, std::ref(n)); // pass by reference

  ... // do some stuff
  t1.join(); t2.join();
}
```

Parallel atoi

```
void foo(const char* strings[], std::size_t n, long& result) {
  for (std::size_t i = 0; i < n; ++i)
    result += myatoi_opt2(strings[i]);
}

long sum_string_vector_parallel_1(const char* strings[], std::size_t n)
{
  long sum = 0;
  std::size_t chunk_size = n / 4;

  std::thread t1(foo, strings + 0, chunk_size, std::ref(sum));
  std::thread t2(foo, strings + chunk_size, chunk_size, std::ref(sum));
  std::thread t3(foo, strings + 2*chunk_size, chunk_size, std::ref(sum));
  std::thread t4(foo, strings + 3*chunk_size, n - 3*chunk_size, std::ref(sum));

  t1.join(); t2.join(); t3.join(); t4.join();
  return sum;
}
```

Is this version ok ?

```
...
10734344625355923
10734344625355923
8051481392513753   <-----------------------------------------------------------------
bench/parallel1/10000000/real_time          8 ms          0 ms         85   1.16374G items/s
```

We do not get the good sum !

Parallel programming => use a thread sanitizer

- Compile with flags

```
clang++ -fsanitize=thread
```

- And run

```
WARNING: ThreadSanitizer: data race (pid=708)
  Write of size 8 at 0x7ffff93a9b58 by thread T2:
    #0 foo(char const**, unsigned long, unsigned long, long&)
    /home/edwin/lrde/cours/prpa/j1/atoi/atoi_parallel.cpp:17 (libimpl.so+0x346e)
    ...
    #6 std::error_code::default_error_condition() const ??:? (libstdc++.so.6+0xbc14e)

  Previous write of size 8 at 0x7ffff93a9b58 by thread T1:

  Location is stack of main thread.
```

Data race

When an evaluation of an expression ***writes*** *to* ***a memory location*** *and another evaluation* ***reads or modifies the same memory location****, the expressions are said to conflict. A program that has two conflicting evaluations has a* ***data race*** *[. . .]*

Solution 1

- Mark the variable **atomic**, meaning that all threads see the same value.
- Use += on atomic is equivalent to `fetch_add`

```
void foo_atomic(const char* strings[], std::size_t n, std::atomic<long>& result)
{
  for (std::size_t i = 0; i < n; ++i)
    result += myatoi_opt2(strings[i]);
}

long sum_string_vector_parallel_1(const char* strings[], std::size_t n)
{
  std::atomic<long> sum;
  std::size_t chunk_size = n / 4;

  std::thread t1(foo_atomic, strings + 0, chunk_size, std::ref(sum));
  std::thread t2(foo_atomic, strings + chunk_size, chunk_size, std::ref(sum));
  std::thread t3(foo_atomic, strings + 2*chunk_size, chunk_size, std::ref(sum));
  std::thread t4(foo_atomic, strings + 3*chunk_size, n - 3*chunk_size, std::ref(sum));

  t1.join(); t2.join(); t3.join(); t4.join();
  return sum;
}
```

Solution 2

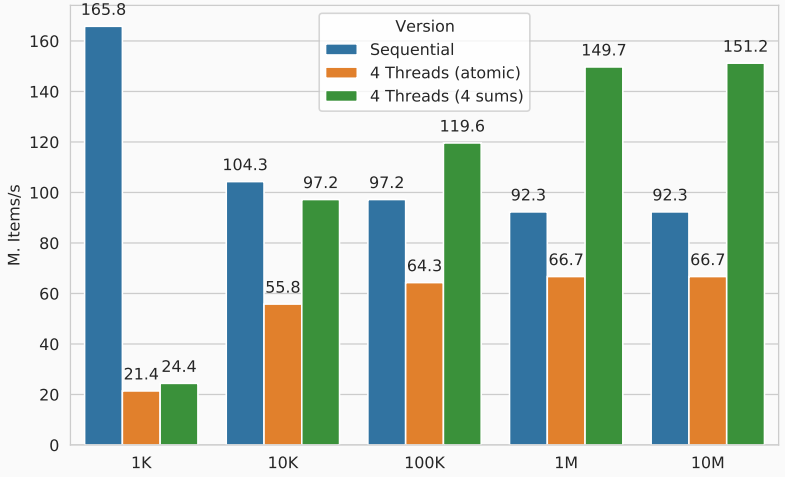
Make the sum on distinct **locations**

```
long sum_string_vector_parallel_2(const char* strings[], std::size_t n)
{
  long sum[4] = {0, 0, 0, 0};
  std::size_t chunk_size = n / 4;

  std::thread t1(foo, strings + 0, chunk_size, std::ref(sum[0]));
  std::thread t2(foo, strings + chunk_size, chunk_size, std::ref(sum[1]));
  std::thread t3(foo, strings + 2*chunk_size, chunk_size, std::ref(sum[2]));
  std::thread t4(foo, strings + 3*chunk_size, n - 3*chunk_size, std::ref(sum[3]));

  t1.join(); t2.join(); t3.join(); t4.join();
  return sum[0] + sum[1] + sum[2] + sum[3];
}
```

Performance



- Version 1 is **slower** than the sequential version
- Version 2 is **slower** than the sequential version for small datasets and barely 1.5x faster on 10M items. :'(

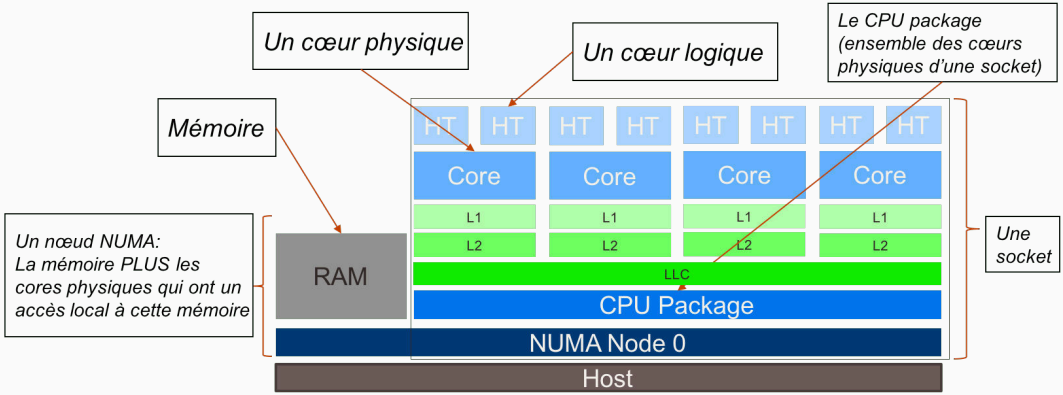
Parallel programming is hard !!

Parallel programming is hard[2]:

- Data races: Invalid program
- Race conditions: Invalid program
- Contention: Poor scaling (e.g. accessing a shared ressource)
- False sharing: Poor scaling
- Load imbalance: Poor scaling
- Poor locality: Bad performance
- Communication overhead: Bad everything

[2] P. McKenney, M. Michael & M. Wong "Is Parallel Programming still hard?

NUMA revisited

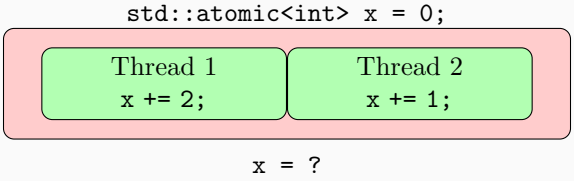


3

- Hyper-Thread share L1 and L2 caches
- Physical cores share the LLC (and RAM)

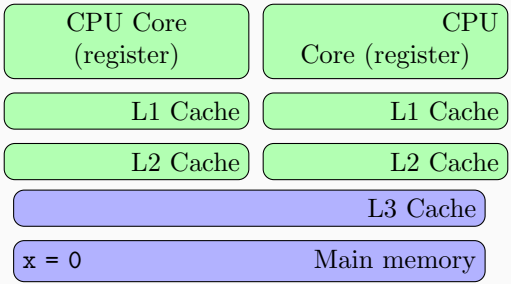
[3] http://blog.enioka.com/post/2017/07/06/topologie-cpu-vmware-vsphere/

NUMA revisited

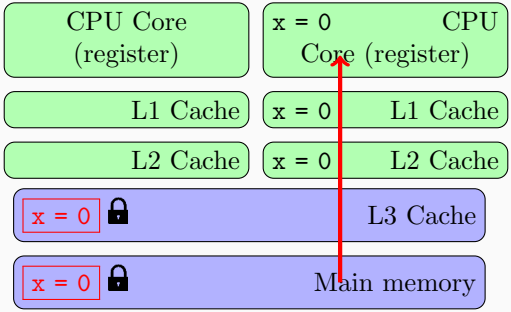


- Read modify write operation:
  - Read `x` from memory
  - Add something to `x`
  - Write `x` to memory

What's going on ?

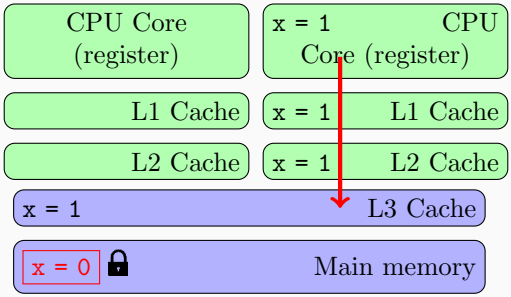


What's going on ?



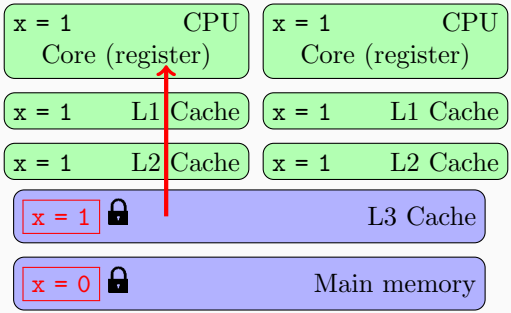
- Thread 2 executes
- It fetches x from caches to main memory
- It locks x address (by hardware impl.)
- Thread 1 executes
  - It try to fetch a x which is locked (goes to by hardware impl.)

What's going on ?



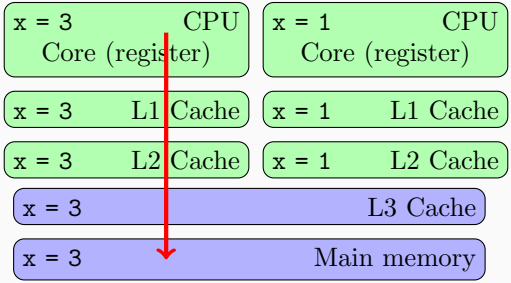
- Thread 2 keeps executing
- It updates x
- It write back x to caches
- It *unlocks* x address

What's going on ?



- Thread 1 executes
  - It fetches `x`
  - It locks `x` address

What's going on ?

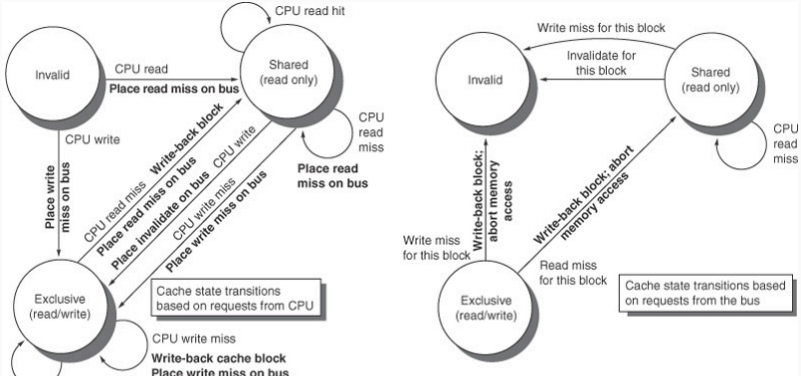


- Thread 1 keeps executing
  - It updates x
  - It write back x address
  - It unlocks x address

What's going on in hardware (MESI) ?

x can be in one of the MESI states (handle by hardware):

State	Description
Modified (M)	x is only in the current cache, and is *dirty* (modified)
Exclusive (E)	x is only in the current cache and *clean*.
Shared (S)	x is stored in many caches and *clean*
Invalid (I)	x is unused



Why is it so slow ? Version 1

```
void foo_atomic(const char* strings[], std::size_t n, std::atomic<long>& result)
{
  for (std::size_t i = 0; i < n; ++i)
    result += myatoi_opt2(strings[i]);
}


long sum_string_vector_parallel_1(const char* strings[], std::size_t n)
{
  std::atomic<long> sum;
  std::size_t chunk_size = n / 4;

  std::thread t1(foo_atomic, strings + 0, chunk_size, std::ref(sum));
  std::thread t2(foo_atomic, strings + chunk_size, chunk_size, std::ref(sum));
  ...
  t1.join(); t2.join(); t3.join(); t4.join();
  return sum;
}
```

Why is it so slow ? Version 1

```
void foo_atomic(const char* strings[], std::size_t n, std::atomic<long>& result)
{
  for (std::size_t i = 0; i < n; ++i)
    result += myatoi_opt2(strings[i]);
}


long sum_string_vector_parallel_1(const char* strings[], std::size_t n)
{
  std::atomic<long> sum;
  std::size_t chunk_size = n / 4;

  std::thread t1(foo_atomic, strings + 0, chunk_size, std::ref(sum));
  std::thread t2(foo_atomic, strings + chunk_size, chunk_size, std::ref(sum));
  ...
  t1.join(); t2.join(); t3.join(); t4.join();
  return sum;
}
```

Data contention

The shared variable `sum` causes many **data cache bouncing**

Why is it so slow ? Version 2

```
long sum_string_vector_parallel_2(const char* strings[], std::size_t n)
{
  long sum[4] = {0, 0, 0, 0};
  std::size_t chunk_size = n / 4;

  std::thread t1(foo, strings, 0, chunk_size, std::ref(sum[0]));
  std::thread t2(foo, strings, chunk_size, 2 * chunk_size, std::ref(sum[1]));
  std::thread t3(foo, strings, 2*chunk_size, 3*chunk_size, std::ref(sum[2]));
  std::thread t4(foo, strings, 3*chunk_size, n, std::ref(sum[3]));

  t1.join(); t2.join(); t3.join(); t4.join();
  return sum[0] + sum[1] + sum[2] + sum[3];
}
```

Why is it so slow ? Version 2

```
long sum_string_vector_parallel_2(const char* strings[], std::size_t n)
{
  long sum[4] = {0, 0, 0, 0};
  std::size_t chunk_size = n / 4;

  std::thread t1(foo, strings, 0, chunk_size, std::ref(sum[0]));
  std::thread t2(foo, strings, chunk_size, 2 * chunk_size, std::ref(sum[1]));
  std::thread t3(foo, strings, 2*chunk_size, 3*chunk_size, std::ref(sum[2]));
  std::thread t4(foo, strings, 3*chunk_size, n, std::ref(sum[3]));

  t1.join(); t2.join(); t3.join(); t4.join();
  return sum[0] + sum[1] + sum[2] + sum[3];
}
```

False sharing

- The sum variables are on the **same cacheline**
- The processor invalidates and exchange whole cacheline

Solutions ?

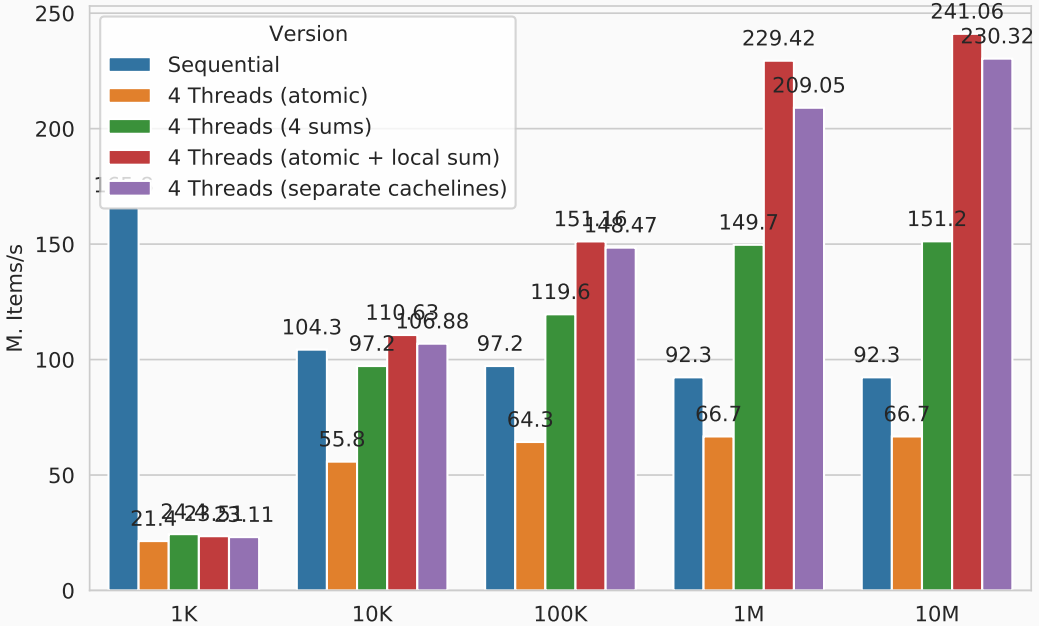
- Push variable on different cachelines (v1)

```
alignas(128) long sum0 = 0;
alignas(128) long sum1 = 0;
alignas(128) long sum2 = 0;
alignas(128) long sum3 = 0;
```

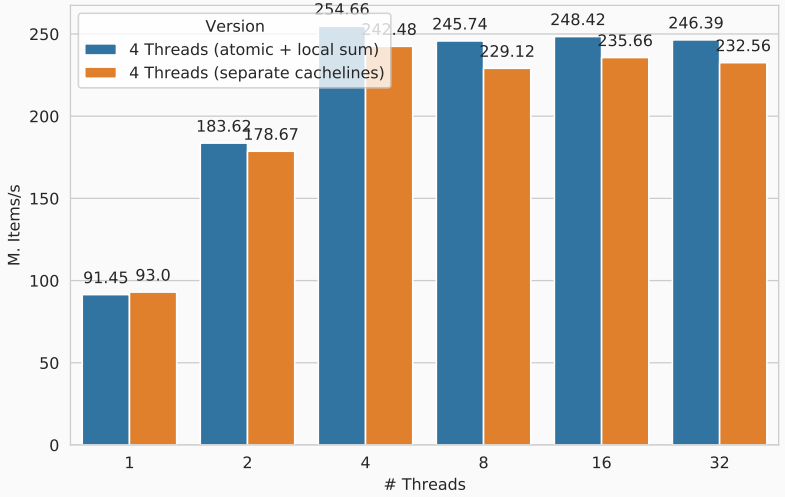
- Use a local variable for summing and reduction on the shared variable at the end (v2)

```
void foo_atomic(const char* strings[], std::size_t n, std::atomic<long>& result)
{
  long tmp = 0;
  for (std::size_t i = 0; i < n; ++i)
    tmp += myatoi_opt2(strings[i]);
  result += tmp;
}
```

Results


Version
Sequential
4 Threads (atomic)
4 Threads (4 sums)
4 Threads (atomic + local sum)
4 Threads (separate cachelines)
M. Items/s
250
200
150
100
50
0
1K
10K
100K
1M
10M
104.3
55.8
97.2
110.63
106.88
97.2
64.3
119.6
151.16
148.47
92.3
66.7
149.7
229.42
209.05
92.3
66.7
151.2
241.06
230.32

Results (as a number of threads)



Best speed-up:

2.5X with 4 threads on core-i7 (2 physical cores + 2 logical cores)

C++ APIs for multi-threadings

More than way to do it

- Asynchronous C++ API `std::thread` -> `std::future`
- Parallel Libraries
  - Intel TBB
  - Parallel STL
- OpenMP

Note: in course 4 we see more about each of them.

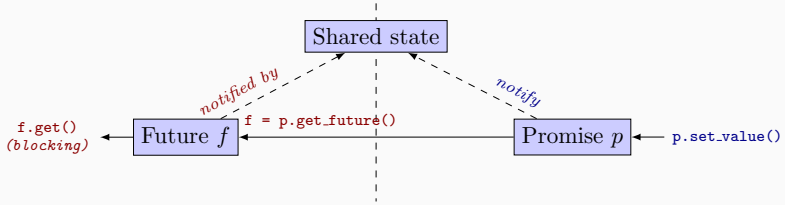
Async Standard C++ Library

Problems with `std::thread`:

- Threads do not return value (pass by ref)
- Data races
- Context switching
- Synchronisation overhead

std::async

```
std::future std::async(Function foo, args...);
```



Main Thread	Thread 1
Creates a *promise* `p`	
Get the *future* `f`	
Spawn thread 1	Execute `x = foo(args)`
Do stuffs	*execute*
`y = p.get()` *blocking*	*execute*
idle	*end executing*
idle	`p.set_value(x)`
y is ready	*Cleanup*
Continue	

std::async

```
long local_sum(const char *strings[], std::size_t n)
{
  long sum = 0;
  for (std::size_t i = 0; i < n; ++i)
    sum += myatoi_opt2(strings[i]);
  return sum;
}

long sum_string_parallel_async(const char* strings[], std::size_t n, int nthread)
{
  std::size_t chunk_size = n / nthread + 1;
  std::array<std::future<long>, 64> results;

  for (int i = 0; i < nthread; ++i, n -= chunk_size, strings += chunk_size)
    results[i] = std::async(local_sum, strings, std::min(n, chunk_size));

  long sum = 0;
  for (int i = 0; i < nthread; ++i)
    sum += results[i].get();
  return sum;
}
```

More than way to do it

- Asynchronous C++ API `std::thread` -> `std::future`
- Parallel Libraries
  - Intel TBB
  - Parallel STL
- OpenMP

Library approaches

High level approach, thread management is delegated to the library.

```
template<typename Range, typename Value, typename Func, typename Reduction>
Value parallel_reduce(const Range& range, const Value& identity,
                      const Func& func, const Reduction& reduction,
                      [, partitioner[, task_group_context& group]] );
```

- Range: a TBB range of values
- Identity: Identity value for the reduction
- Func: *Map* function to execute on each sub-range
- Reduction: *Join* function to execute on two sub-ranges
- Partionner: a way to say how to split ranges

Parallel atoi with TBB

```
long sum_string_tbb(const char* strings[], std::size_t n, int nthread)
{
  return
    tbb::parallel_reduce(
      tbb::blocked_range<const char**>(strings, strings + n),
      0L,
      [](const tbb::blocked_range<const char**>& rng, long init) {
        return init + local_sum(rng.begin(), rng.size());
      },
      std::plus<long>(),
      tbb::static_partitioner()
      );
}
```

Parallel STL

- Introduced in C++17 (not yet available in all compilers)
- But many implementations available (including one with Intel TBB)

```
return std::transform_reduce(std::execution::par,
    strings,              // First iterator of the range
    strings + n,          // Past-the-end iterator of the range
    0L,                   // Identity element
    std::plus<long>(),    // Reduction op
    myatoi_opt2);         // Map op
```

More than way to do it

- Asynchronous C++ API `std::thread` -> `std::future`
- Parallel Libraries
  - Intel TBB
  - Parallel STL
- OpenMP

OpenMP approach

- Use annotations to describe a parallel section

```
#pragma omp parallel for [clauses]
```

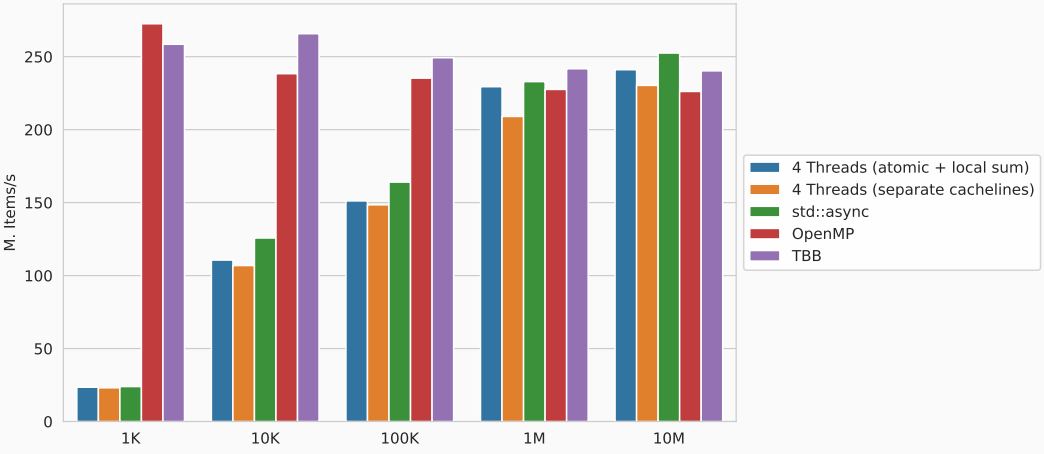
Clause	
`schedule(type, [size])`	See below
`reduction(operator : variables)`	Reduction variables
`num_threads(n)`	Number of threads used
`ordered`	Process in order (Thread *n+1* waits for *n* to finish)

Schedule type	
static	Divided in *chunk_size* and assigned to threads statically
dynamic	Divided in *chunk_size* and assigned to available threads
guided	Like *dynamic* with decreasing size of chunks
runtime	Decision deferred until runtime

OpenMP approach

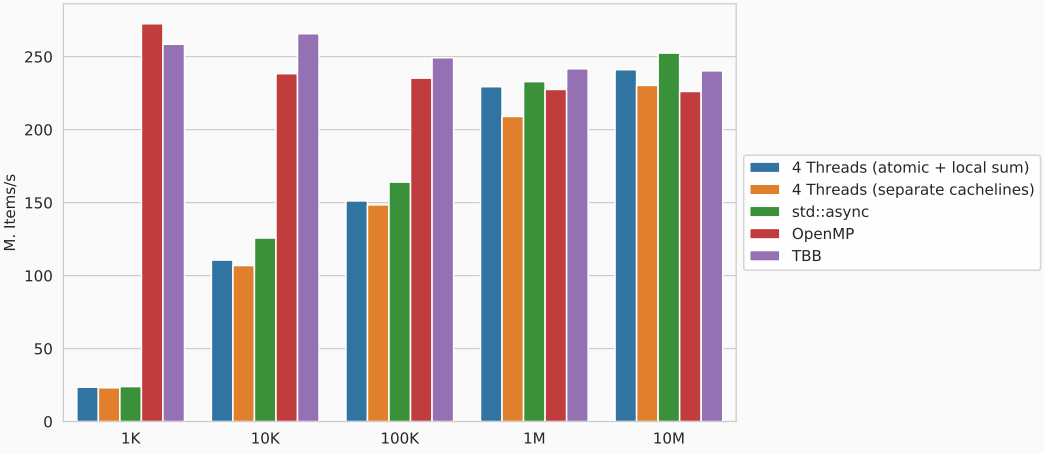
```
long sum_string_openmp(const char* strings[], std::size_t n, int nthread)
{
  long sum = 0;
  #pragma omp parallel for reduction(+:sum) schedule(static) num_threads(nthread)
  for (std::size_t i = 0; i < n; ++i)
    sum += myatoi_opt2(strings[i]);
  return sum;
}
```

Results with 4 threads



Why do we get bad results at 1K with `std::thread` and `std::async` ?

Results with 4 threads



Why do we get bad results at 1K with `std::thread` and `std::async` ?

- Creating threads has a cost
- OpenMP / TBB create a thread pool at startup

Questions ?