Multi-threading on CPUs with OpenMP and Metrics for Performance Analysis of Applications

Sorbonne Université – Master SESI – MU5IN60 – Parallel Programming

Adrien CASSAGNE

October 9, 2023

Table of Contents

- **Nowadays, multi-core architecture is well spread** in High Performance Computing (HPC) and in embedded targets

- There are two main ways to use multi-core architectures
  1. Create **multiple processes** (= distributed memory model)
     → MPI standard, Unix inter-processes communications, sockets, ...
  2. Create **multiple threads** (= shared memory model)
     → Threads POSIX, OpenMP, ...

- In this session we will not talk about the multiple processes model

- And **we will go deeper into the multi-threaded model**

- OpenMP is a language dedicated to setup multi-threaded codes
- It is based on **compiler directives** (#pragma)
  — Those directives describe how to perform the parallelism
  — The main advantage of directives is to **not modify sequential code** (in theory...)
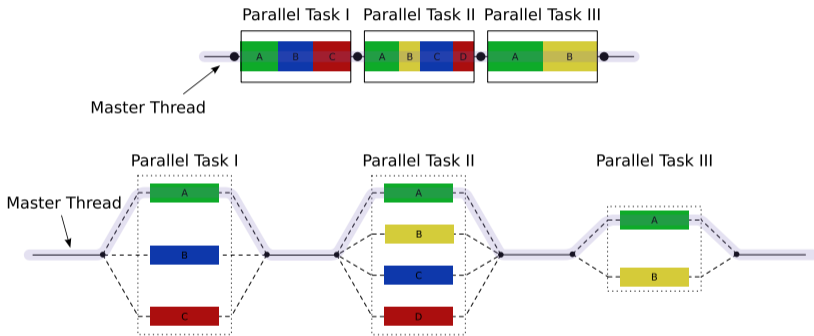
```
1 void add_vectors(const float* A, const float* B, float* C, const size_t n)
2 {
3 #pragma omp parallel // directive for the creation of a parallel zone (= threads creation)
4 { // <- beginning of the parallel zone
5 #pragma omp for // directive for distribution of for-loop indices among threads
6   for (size_t i = 0; i < n; i++)
7     C[i] = A[i] + B[i];
8 } // <- end of the parallel zone
9 }
```

Simple add_vectors OpenMP implementation
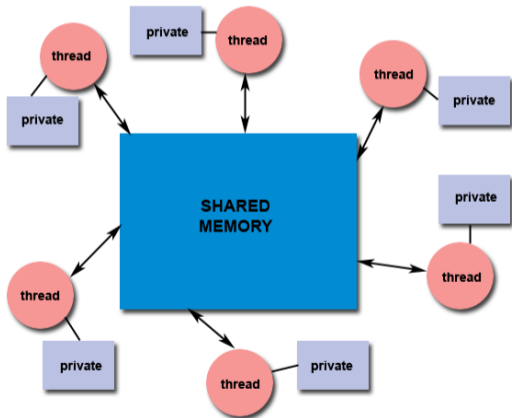
Fork-join Model

- When using `#pragma omp parallel` directive: threads are **created** (= **fork**)
- At the end of a parallel zone
  — Threads are **destroyed** (= **join**), except for the master thread
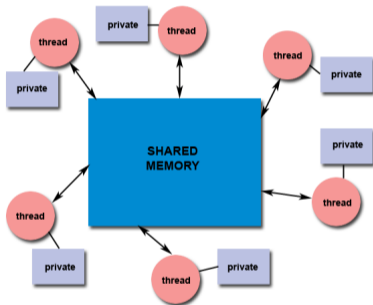 — There is an implicit barrier

- Each thread can access the global memory zone
  — This is called the shared memory (or the RAM of the CPUs)

- But threads also own private data
  — Not completely shared model
  — Very often, the key for achieving performance is to keep the memory private when possible...

```
1  void add_vectors(const float* A, const float* B, float* C,
2                   const size_t n)
3  {
4  #pragma omp parallel
5  {
6  #pragma omp for
7    // 'i' is private because it is declared
8    // after the omp parallel directive
9    for (size_t i = 0; i < n; i++) // <- 'n' is shared
10     // 'A', 'B' and 'C' are shared!
11     C[i] = A[i] + B[i];
12 }
13 }
```

- By default, variables that are **declared before a parallel zone** are **shared** (here A, B, C and n)

- And variables **declared inside a parallel zone** are **private** (here i)

- OpenMP provides data range control
  - **private**: local to the thread,
  - **firstprivate**: local to the thread and initialized
  - **shared**: shared by all the threads, in C/C++ this is the default behavior

- Here `alpha` is a constant, we can put it in the private memory of each thread

- Efficient parallelism comes with minimal synchronizations
  - Shared data can generate a lot of synchronizations
  - Privacy increases thread independence

```c
void dot(const float* A,
               float* B,
         const float  alpha,
         const size_t n)
{
#pragma omp parallel      \
           shared(A, B) \
           firstprivate(alpha, n)
{
#pragma omp for
  // 'i' is still private because
  // it is declared after the
  // parallel zone
  for (size_t i = 0; i < n; i++) {
    B[i] = alpha * A[i];
  }
}
}
```

for-loop Indices Distribution

- for-loop indices distribution can be controlled by the **schedule** clause
  - **static**: indices distribution is precomputed (at compilation time), and the amount of indices is the same for each thread
  - **dynamic**: indices distribution is done in real time along the loop execution, work load balancing can be better than with the static scheduling but dynamic scheduling costs some additional resources in order to attribute indices at real time

- There are other types of scheduling but this is not a full OpenMP lesson

```
1 // ...
2 #pragma omp for schedule(static, 128) // we statistically attribute 128 per 128 indices to each threads
3   for(int i = 0; i < n; i++)
4     B[i] = alpha * A[i];
5 // ...
```

- Previous slides were a brief overview of the main OpenMP principles

- To have more precise informations you can take a look at the very good OpenMP reference card[1]

    — It could be a very good idea to print it and keep it ;-)

- In the next slides we will pay attention to some **OpenMP use cases**

---

[1] https://www.openmp.org/wp-content/uploads/OpenMPRef-5.0-111802-web.pdf

Table of Contents

Avoid False Sharing

- **False sharing** is a phenomena that occurs when threads write simultaneously data in a same cache line
  — Remember, the cache system works with lines of words: a line is the smallest element in caches coherence mechanism
  — If two or more threads are working on the same line they cannot write data simultaneously!
    ➜ Stores are serialized and we talk about false sharing

- To avoid false sharing, threads have to work on a bigger amount of data than the cache line size
  — Concretely we have to avoid (`static,1`) or (`dynamic,1`) scheduling
  — Cache lines are not very big (≈ 64 Bytes)
  — Just putting a (`static,16`) or (`dynamic,16`) often resolves the problem
    ○ Be aware that in some OpenMP implementations, the default scheduling pattern is (`static,1`)!

- In OpenMP there are a lot of **implicit barriers**, after each
  — #pragma omp parallel directive
  — #pragma omp for directive
  — #pragma omp single directive

- But not after #pragma omp master directive!

- If we are sure that there is no need to synchronise threads after the #pragma omp for directive, we can use the **nowait** clause

- Optimally we need only one #pragma omp parallel directive in a fully parallel code
  — OpenMP manages a pool of threads in order to reduce the cost of the #pragma omp parallel directive but this is not free, each time OpenMP has to reorganize the pool and wakes up the required threads

```
1  // A, B & C <- size = n, D <- size = 2n
2  void kernel_v1(const float *A, const float *B, const float *C,
3                 float *D, const float alpha, const size_t n) {
4  // overhead: threads creation and private variables creation
5  #pragma omp parallel shared(A, B, D) \
6                       firstprivate(alpha, n)
7  {
8  #pragma omp for schedule(static,16)
9  {
10   for (size_t i = 0; i < n; i++)
11     D[i] = alpha * A[i] + B[i];
12 } // implicit barrier
13 } // implicit barrier
14
15 // overhead: threads attribution and private variables creation
16 #pragma omp parallel shared(A, C, D) firstprivate(n)
17 {
18 #pragma omp for schedule(static,16)
19 {
20   for (size_t i = 0; i < n; i++)
21     D[n + i] = A[i] + C[i];
22 } // implicit barrier
23 } // implicit barrier
24 }
```

```
1  // A, B & C <- size = n, D <- size = 2n
2  void kernel_v1(const float *A, const float *B, const float *C,
3                 float *D, const float alpha, const size_t n) {
4  // overhead: threads creation and private variables creation
5  #pragma omp parallel shared(A, B, D) \
6                        firstprivate(alpha, n)
7  {
8  #pragma omp for schedule(static,16)
9  {
10   for (size_t i = 0; i < n; i++)
11     D[i] = alpha * A[i] + B[i];
12 } // implicit barrier
13 } // implicit barrier
14
15 // overhead: threads attribution and private variables creation
16 #pragma omp parallel shared(A, C, D) firstprivate(n)
17 {
18 #pragma omp for schedule(static,16)
19 {
20   for (size_t i = 0; i < n; i++)
21     D[n + i] = A[i] + C[i];
22 } // implicit barrier
23 } // implicit barrier
24 }
```

```
1  // A, B & C <- size = n, D <- size = 2n
2  void kernel_v2(const float *A, const float *B, const float *C,
3                 float *D, const float alpha, const size_t n) {
4  // overhead: threads creation and private variables creation
5  #pragma omp parallel shared(A, B, C, D) \
6                        firstprivate(alpha, n)
7  {
8  #pragma omp for schedule(static,16) nowait
9  {
10   for (size_t i = 0; i < n; i++)
11     D[i] = alpha * A[i] + B[i];
12 } // no implicit barrier (nowait clause)
13
14 #pragma omp for schedule(static,16)
15 {
16   for (size_t i = 0; i < n; i++)
17     D[n + i] = A[i] + C[i];
18 } // implicit barrier
19 } // implicit barrier
20 }
21
22 /* 'kernel_v2' is a better version than 'kernel_v1':
23  *   - only one parallel zone
24  *   - no barrier after the first loop (nowait clause) */
```

- Sometimes it is not possible to have a fully parallel code and some regions of the code remain intrinsically sequential

- In OpenMP we can specify this kind of region with the `#pragma omp critical` directive

- But we have to use this directive carefully
  - `#pragma omp critical` can be the main cause of slow down in OpenMP codes!

Scale `A` in `B` and find the minimum value of `B` in `min_val`

```
 1 float kernel_v1(const float *A, float *B, const size_t n) {
 2   float min_val = INF;
 3 #pragma omp parallel shared(A, B, min_val) firstprivate(n)
 4 {
 5 #pragma omp for schedule(static,16)
 6 {
 7   for (size_t i = 0; i < n; i++) {
 8     B[i] = 0.5f * A[i];
 9 #pragma omp critical // we want to be sure that only one
10 {                    // thread can modify min_val
11     if (B[i] < min_val)
12       min_val = B[i];
13 }
14   }
15 }
16 }
17   return min_val;
18 }
```

Scale `A` in `B` and find the minimum value of `B` in `min_val`

```
1  float kernel_v1(const float *A, float *B, const size_t n) {
2    float min_val = INF;
3  #pragma omp parallel shared(A, B, min_val) firstprivate(n)
4  {
5  #pragma omp for schedule(static,16)
6  {
7    for (size_t i = 0; i < n; i++) {
8      B[i] = 0.5f * A[i];
9  #pragma omp critical  // we want to be sure that only one
10 {                     //  thread can modify min_val
11     if (B[i] < min_val)
12       min_val = B[i];
13 }
14   }
15 }
16 }
17   return min_val;
18 }
19
20 /* This code is slow because each loop step contains a
21  * sequential part */
```

```
1  float kernel_v2(const float *A, float *B, const size_t n) {
2    float min_val = INF;
3  #pragma omp parallel shared(A, B, min_val) firstprivate(n)
4  {
5  #pragma omp for schedule(static,16)
6  {
7    for (size_t i = 0; i < n; i++) {
8      B[i] = 0.5f * A[i];
9      // no more threads synchro to perform the test
10     if (B[i] < min_val)
11 #pragma omp critical
12 {    // this is very important to re-do the test because
13      // an other thread may have modify the min_val value
14       if (B[i] < min_val)
15         min_val = B[i];
16 }
17   }
18 }
19 }
20   return min_val;
21 }
```

- In OpenMP 3 there is no optimal solution for search algorithms

- This kind of algorithm typically requires while-loops or do-while-loops

- However there is a tip to fix this lack in OpenMP 3 (see next slide)

- Latest versions of OpenMP (v4 and v5) provides better control of threads
  — We can terminate threads...
  — But this lead to more complex solutions, we will not see them today

Search if `val` element is in the `A` array of size `n`

```
1 bool search_val_v1(const float *A, const size_t n, float val)
2 {
3   bool found = false;
4 #pragma omp parallel shared(A, found) firstprivate(val)
5 {
6 #pragma omp for schedule(static,16)
7 {
8   for (size_t i = 0; i < n; i++) {
9     if (A[i] == val) {
10       found = true;
11       break; // not valid in OMP, the compilation
12              // will fail
13     }
14   }
15 }
16 }
17   return found;
18 }
```

Search if `val` element is in the `A` array of size `n`

```
 1 bool search_val_v1(const float *A, const size_t n, float val)
 2 {
 3   bool found = false;
 4 #pragma omp parallel shared(A, found) firstprivate(val)
 5 {
 6 #pragma omp for schedule(static,16)
 7 {
 8   for (size_t i = 0; i < n; i++) {
 9     if (A[i] == val)
10       found = true; // no more break, this is valid but
11                     // this is also slow, no more early
12                     // exit :-(
13   }
14 }
15 }
16   return found;
17 }
```

Search if `val` element is in the `A` array of size `n`

```
1  bool search_val_v1(const float *A, const size_t n, float val)
2  {
3    bool found = false;
4  #pragma omp parallel shared(A, found) firstprivate(val)
5  {
6  #pragma omp for schedule(static,16)
7  {
8    for (size_t i = 0; i < n; i++) {
9      if (A[i] == val)
10       found = true; // no more break, this is valid but
11                     // this is also slow, no more early
12                     // exit :-(
13   }
14 }
15 }
16   return found;
17 }
```

```
1  bool search_val_v2(const float *A, const size_t n, float val)
2  {
3    bool found = false;
4  #pragma omp parallel shared(A, found) firstprivate(val)
5  {
6  #pragma omp for schedule(static,16)
7  {
8    for (size_t i = 0; i < n; i++) {
9      if (!found) // we are doing nothing if we have found
10                 // the value in the array
11       if (A[i] == val)
12         found = true;
13   }
14 }
15 }
16   return found;
17 }
```

Search if `val` element is in the `A` array of size `n`

```
1  bool search_val_v1(const float *A, const size_t n, float val)
2  {
3    bool found = false;
4  #pragma omp parallel shared(A, found) firstprivate(val)
5  {
6  #pragma omp for schedule(static,16)
7  {
8    for (size_t i = 0; i < n; i++) {
9      if (A[i] == val)
10       found = true; // no more break, this is without
11                     // this is also slow, no more early
12                     // exit :-(
13   }
14 }
15 }
16   return found;
17 }
```

```
1  bool search_val_v2(const float *A, const size_t n, float val)
2  {
3    bool found = false;
4  #pragma omp parallel shared(A, found) firstprivate(val)
5  {
6  #pragma omp for schedule(static,16)
7  {
8    for (size_t i = 0; i < n; i++) {
9      if (!found) // we are doing nothing if we have found
10                 // the value in the array
11       if (A[i] == val)
12         found = true;
13   }
14 }
15 }
16   return found;
17 }
```

Promises not to modify the sequential code are not always kept...

Table of Contents

Execution Time
3 Parallel Code Analysis

How do you compare two versions of a code that does the same thing (from the functional point of view)?

- Compare the execution time of the two versions
  — The fastest program is the most efficient one
  — Intuitive and worth keeping in mind

- Be careful to compare the same times
  — Classic error: comparing the total execution time of one program with just a sub-part of another program's execution time
    → In this case, the two measured times are not comparable

- Let's consider $\mathcal{D}_1$ (or $\mathcal{D}_s$) the sequential time (time on 1 core) of a code
  - With 2 cores, we can hope to divide the time by 2 at best $(\mathcal{D}_2^m \geq \mathcal{D}_s/2)$
  - With 3 cores, we can hope to divide the time by 3 at best $(\mathcal{D}_3^m \geq \mathcal{D}_s/3)$

- The following table shows the execution times measured for a Code 1:

# core $(\mathcal{C})$	Measured time $(\mathcal{D}^m)$	Optimal time $(\mathcal{D}^o)$
1	98 ms	98.0 ms
2	50 ms	49.0 ms
3	35 ms	32.7 ms
4	27 ms	24.5 ms
5	22 ms	19.6 ms
6	18 ms	16.3 ms

  - Optimal time $= \mathcal{D}^o = \mathcal{D}_s/\mathcal{C}$

- The previous table is not easy to read
- Let's look at the results on a graph:
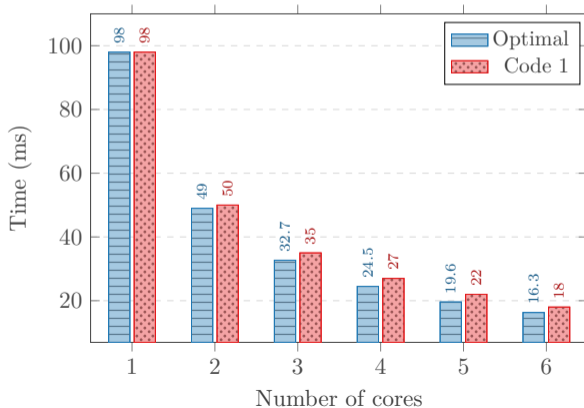
Speedup – Definition

$$\mathcal{S} = \mathcal{D}_s/\mathcal{D}_\mathcal{C},$$

with $\mathcal{D}_s$ the time measured for the 1-core version (= sequential version) of the code and $\mathcal{D}_\mathcal{C}$ the time measured for the parallel version with $\mathcal{C}$ cores.

# cores ($\mathcal{C}$)	Time ($\mathcal{D}^m$)	Speedup ($\mathcal{S}$)
1	98 ms	1.00
2	50 ms	1.96
3	35 ms	2.80
4	27 ms	3.63
5	22 ms	4.45
6	18 ms	5.44

- Sequential time is used as reference time

- Visually very simple to see if `Code 1` is close to the the optimal
- Optimal speedup is equal to the number of cores used (no more!)

Amdahl's Law
3 Parallel Code Analysis

- Can we increase parallelism indefinitely to speedup our codes?
  — Amdahl said no!
  — To be more precise, it depends on the characteristics of the code...
  — If the code is fully parallelizable: speedup is infinite
  — If the code is NOT fully parallelizable: there's a limit

$$\mathcal{S}_{\max} = \frac{1}{1 - f\mathcal{D}_p},$$

with $\mathcal{S}_{\max}$ the maximum achievable speedup and $f\mathcal{D}_p$ the fraction of parallel time in the code ($0 \leq f\mathcal{D}_p \leq 1$).

Amdahl's Law – Example
3 Parallel Code Analysis

- Let's take a code composed of two parts:
  - 20 % is intrinsically sequential
  - 80 % can be parallelized

- What is the maximum achievable speedup?

$$\mathcal{S}_{\max} = \frac{1}{1 - f\mathcal{D}_p} = ...$$

Amdahl's Law – Example
3 Parallel Code Analysis

- Let's take a code composed of two parts:
  — 20 % is intrinsically sequential
  — 80 % can be parallelized

- What is the maximum achievable speedup?

$$\mathcal{S}_{\max} = \frac{1}{1 - f\mathcal{D}_p} = \frac{1}{1 - 0.8} = \frac{1}{0.2} = 5.$$

Amdahl's Law – Example
3 Parallel Code Analysis

- Let's take a code composed of two parts:
  — 20 % is intrinsically sequential
  — 80 % can be parallelized

- What is the maximum achievable speedup?

$$\mathcal{S}_{\max} = \frac{1}{1 - f\mathcal{D}_p} = \frac{1}{1 - 0.8} = \frac{1}{0.2} = 5.$$

That's not much when you consider architectures with dozens of CPU cores!
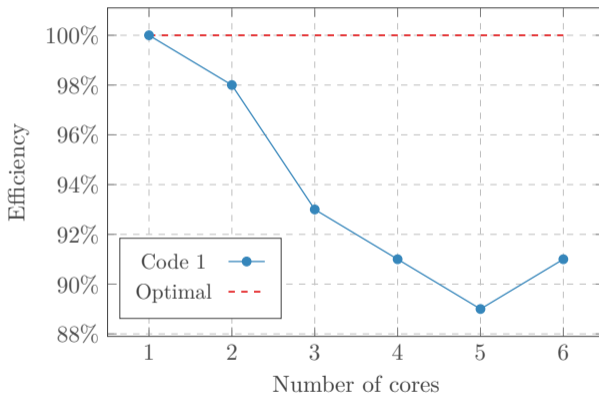
Efficiency – Definition

- Several ways to define the efficiency ($\mathcal{E}$) of a code
  — From the speedup: $\mathcal{E} = \mathcal{S}^m/\mathcal{S}^o$
  — From the execution time: $\mathcal{E} = \mathcal{D}^m/\mathcal{D}^o$

- Efficiency is a ratio: $0\% < \mathcal{E} \leq 100\%$

- For `Code 1`, the efficiency as a function of the number of cores:

# cores ($\mathcal{C}$)	Time ($\mathcal{D}^m$)	Speedup ($\mathcal{S}$)	Efficiency ($\mathcal{E}$)
1	98 ms	1.00	100%
2	50 ms	1.96	98%
3	35 ms	2.80	93%
4	27 ms	3.63	91%
5	22 ms	4.45	89%
6	18 ms	5.44	91%

Efficiency – Visualization
3 Parallel Code Analysis

- Equivalent to the speedup, at least for now...

Scalability of a Code – Definition
3 Parallel Code Analysis

- The scalability of a code is its capacity to be efficient when the number of cores increases
  — A code scales if it is able to benefit from the power of several cores

- How do we measure a code's scalability? How do we know if a code doesn't scale?
  — No simple answer

- 2 widely used models for characterizing the scalability of parallel code:
  — The "strong" scalability
  — The "weak" scalability

Strong Scalability – `Code 1` Example

- Measures execution time as a function of the number of cores

- With **a constant problem size**

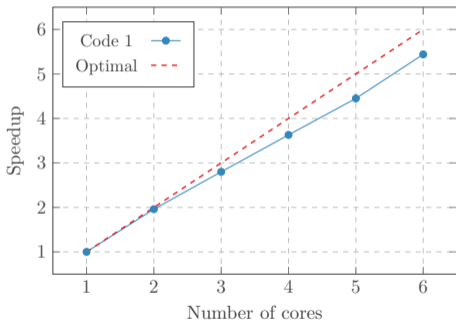- For example, for `Code 1`, with a problem of size 100:

# cores	Size	Time	Speedup
1	100	98 ms	1.00
2	100	50 ms	1.96
3	100	35 ms	2.80
4	100	27 ms	3.63
5	100	22 ms	4.45
6	100	18 ms	5.44

Strong scalability is generally observed on an speedup graph:

Here, for 6 cores, `Code 1` achieves a speedup of 5.4, so we can conclude that this code scales well up to 6 cores.

- Let's consider a new `Code 2`
- Here are the measurements for this code:

# cores	Size	Time	Speedup
1	100	98 ms	1.00
2	100	50 ms	1.96
3	100	35 ms	2.80
4	100	32 ms	3.06
5	100	30 ms	3.27
6	100	33 ms	2.97

- We can see that the strong scalability of `Code 2` is poor

- Above a certain number of cores, parallelism can no longer speedup code :-(

- This model considers the execution time as a function of the number of cores
- **And the problem size increases in proportion to the number of cores!**
- Compute the *speedup* makes no sense if the problem size is not constant
- BUT it is possible to compute the efficiency: $\mathcal{E} = \mathcal{D}^s/\mathcal{D}^m$

Intuition: if we can't compute a problem of a given size any faster, can we compute a bigger problem in the same time?

Most of the time, yes, and it's easier! This is what happens most of the time in high performance computing: scientific models become more and more refined = the size of the problem increases.
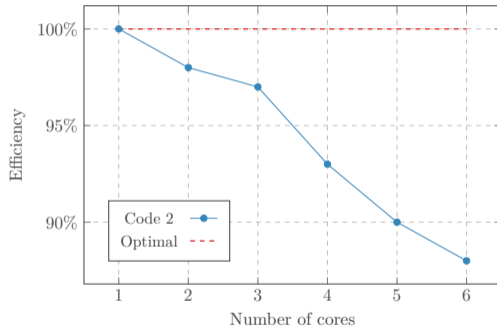
Weak Scalability – `Code 2` Example
3 Parallel Code Analysis

Measures for `Code 2`:

# cores	Size	Time	Efficiency
1	100	098 ms	100%
2	200	100 ms	98%
3	300	101 ms	97%
4	400	105 ms	93%
5	500	109 ms	90%
6	600	111 ms	88%

- The weak scalability of `Code 2` is good ($\approx 90\%$ for 6 cores)
- Why is strong scalability bad?
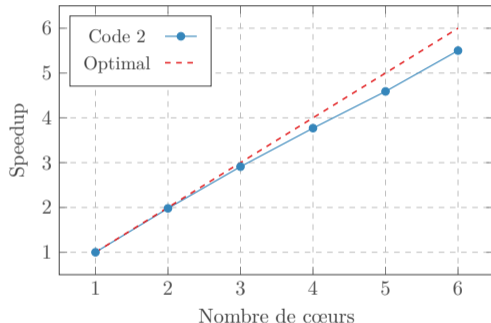   — Amdahl's law: not enough parallelism for a small problem size

- New problem size: 600

# cores	Size	Time	Speedup
1	600	611 ms	1.00
2	600	308 ms	1.98
3	600	210 ms	2.91
4	600	162 ms	3.77
5	600	133 ms	4.59
6	600	111 ms	5.50

- For larger problem sizes, strong scalability is good
- Not always possible to test for strong scalability: lack of time
- Not always possible to test for low scalability: impossible to have ever larger problem sizes

Table of Contents

4 Kernel Performance Analysis

- The number of arithmetic operations in a code is an **important characteristic**

- Example of the number of float operations (*flops*) in a kernel that performs a sum:

```
1 float sum(float *values, size_t n) {
2   float sum = 0.f;
3
4   // total flops = n * 1
5   for (size_t i = 0; i < n; i++)
6     sum = sum + values[i]; // 1 flop because of the addition
7
8   return sum;
9 }
```

Number of Operations per Second

- Metric widely used in high performance computing, particularly the number of **floating-point** operation per second (flop/s)
    - The same is defined for integer operation (iop/s)
    - Or simply the number of operation per second (op/s), or *Million Instructions Per Second* (MIPS)
        → This metrics is used in the kernel Linux (`cat /proc/cpuinfo`)

- The flop/s ratio can be directly compared with the peak performance of a computing architecture

- A metric to estimate the good use (or not) of the hardware architecture

Processor Peak Performance

- **The processor's maximum computational capacity**

- It can be deduced from our knowledge of the hardware architecture:

$$peakPerf = nOps \times freq \times nCores,$$

with $nOps$ the number of operations the architecture can achieve in one cycle (ILP), $freq$ the processor frequency and $nCores$ the number of processor cores.

Processor Peak Performance – Example
4 Kernel Performance Analysis

CPU name	Core i7-2630QM
Architecture	Sandy Bridge
Vect. inst.	AVX-256 bit (4 double, 8 single)
Frequency	2 GHz
Nb. cores	4

Peak performance in floating-point single precision:

$$peakPerf_{sp} = nOps \times freq \times nCores = (2 \times 8) \times 2 \times 4 = 128 \; Gflop/s$$

Peak performance in floating-point double precision:

$$peakPerf_{dp} = nOps \times freq \times nCores = (2 \times 4) \times 2 \times 4 = 64 \; Gflop/s$$

- $nOps = 2 \times vectorSize$ because the architecture back-end allows to issue 2 instructions par cycle (vadd et vmul)

- Sometimes (even often) measured op/s are far from peak performance
  — Code is poorly optimized
  — Peak performance cannot be achieved
  — In most cases, both are true...

- With **arithmetic intensity** we take into account **memory accesses**:

$$AI = \frac{ops}{memops}.$$

```
1 float sum(float *values, size_t n) {
2   float sum = 0.f; // do not count the memory acces to 'sum' because it will be optimized in register
3   // total flops = n * 1  ||  total memops = n * 1
4   for (size_t i = 0; i < n; i++)
5     sum = sum + values[i]; // 1 flop because of the addition, 1 memop because of one acces in
6                            // the 'values' array
7   return sum;
8 }
```

- Arithmetic intensity of `sum` is: $AI_{\texttt{sum}} = \frac{n \times 1}{n \times 1} = 1$

- The higher the arithmetic intensity, the more the code is limited by computational units

- The lower the arithmetic intensity, the more the code is limited by memory accesses

Operational Intensity

- Compared to arithmetic intensity, operational intensity takes into account **the size of the data in memory**:

$$OI = \frac{flops}{memops \times sizeOfData} = \frac{AI}{sizeOfData},$$

$sizeOfData$ depends on the datatype, `int` and `float` use 4 bytes, `long long int` and `double` use 8 bytes.

- In the previous code (`sum`), the memory accesses are made on `float` and the operational intensity is: $OI_{\texttt{sum}} = \frac{n \times 1}{(n \times 1) \times 4} = \frac{1}{4}$

Operational Intensity – Example
4 Kernel Performance Analysis

Sum in single precision:

```
1 // AI = 1  ||  OI = 1/4
2 float sum1(float *values, size_t n) {
3   float sum = 0.f;
4   for (size_t i = 0; i < n; i++)
5     sum = sum + values[i];
6   return sum;
7 }
```

Sum in double precision:

```
1 // AI = 1  ||  OI = 1/8
2 double sum2(double *values, size_t n) {
3   double sum = 0.0;
4   for (size_t i = 0; i < n; i++)
5     sum = sum + values[i];
6   return sum;
7 }
```

- sum1 and sum2 kernels have the same arithmetic intensity

- The operational intensity of sum1 is higher than that of sum2
  — sum2 kernel is more limited by memory access than sum1 kernel

- Roofline[1] is model that limits **the maximum achievable performance**

- Takes into account
  — Memory bandwidth (RAM)
  — Processor peak performance

- Depending on operational intensity, code is limited either by memory bandwidth or by processor peak performance

$$\text{Attainable op/s} = \min \begin{cases} \text{Peak CPU op/s performance,} \\ \text{Peak memory bandwidth} \times OI. \end{cases}$$

---

[1]S. Williams, A. Waterman, and D. Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *ACM Communications* 52.4 (Apr. 2009), pp. 65–76. DOI: 10.1145/1498765.1498785.

Memory Bandwidth Measurement
4 Kernel Performance Analysis

- Memory bandwidth or memory throughput represents the number of bytes that can be read/written from RAM in one second (o/s or GB/s)

- How to know the memory bandwidth?
  - It is possible to compute its theoretical value
  - But we often prefer to use a micro-benchmark program: "STREAM"[1] or "bandwidth"[2]

- STREAM (University of Virginia) is a small, relatively simple code for measuring memory bandwidth → C code

- bandwidth (Sorbonne University) targets the same features as STREAM but in a more friendly-user and accurate way → C++ code

---

[1]STREAM: https://www.cs.virginia.edu/stream/

[2]bandwidth: https://github.com/alsoc/bandwidth

Let's take the same processor as before:

CPU name	Core i7-2630QM
Peak perf sp	128 GFlop/s
Peak perf dp	64 GFlop/s
Mem. bandwidth	17.6 GB/s

Single precision sum:

```
1 // AI = 1  ||  OI = 1/4
2 float sum1(float *values, size_t n) {
3   float sum = 0.f;
4   for (size_t i = 0; i < n; i++)
5     sum = sum + values[i];
6   return sum;
7 }
```

Double precision sum:

```
1 // AI = 1  ||  OI = 1/8
2 double sum2(double *values, size_t n) {
3   double sum = 0.0;
4   for (size_t i = 0; i < n; i++)
5     sum = sum + values[i];
6   return sum;
7 }
```

Peak perf sp	128 GFlop/s
Peak perf dp	64 GFlop/s
Mem. bandwidth	17.6 GB/s

For `sum1`, operational intensity is: $OI_{sum1} = 1/4$.
Let's apply the Roofline model:

$$\text{Attainable Gflop/s} = \min \begin{cases} \text{Peak floating point performance,} \\ \text{Peak memory bandwidth} \times OI. \end{cases}$$

$$\Rightarrow$$

$$\text{Attainable Gflop/s}_{\texttt{sum1}} = \min \begin{cases} 128 \text{ Gflop/s,} \\ 17.6 \times \frac{1}{4} \text{ Gflop/s.} \end{cases} = 4.4 \text{ Gflop/s}$$

Peak perf sp	128 GFlop/s
Peak perf dp	64 GFlop/s
Mem. bandwidth	17.6 GB/s

For `sum2`, operational intensity is: $OI_{sum2} = 1/8$.
Let's apply the Roofline model:

$$\text{Attainable Gflop/s} = \min \begin{cases} \text{Peak floating point performance,} \\ \text{Peak memory bandwidth} \times OI. \end{cases}$$
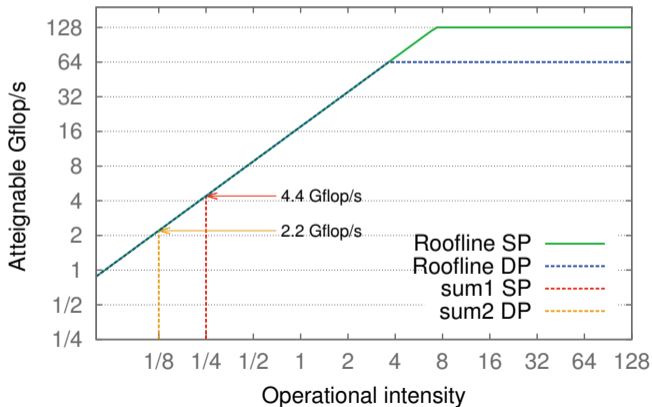
$$\Rightarrow$$

$$\text{Attainable Gflop/s}_{\text{sum2}} = \min \begin{cases} 64 \text{ Gflop/s,} \\ 17.6 \times \frac{1}{8} \text{ Gflop/s.} \end{cases} = 2.2 \text{ Gflop/s}$$
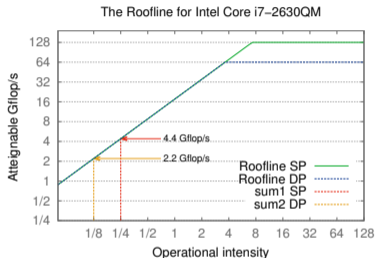
The Roofline for Intel Core i7−2630QM

Roofline Model – Example – Visualization

4 Kernel Performance Analysis

The Roofline for Intel Core i7–2630QM

- There are **two different *Rooflines***
  - One for computations in single precision
  - The other for computations in double precision

- Here it is clear that sum1 and sum2 are **limited by the memory bandwidth**

Q&A

Thank you for listening!
Do you have any questions?

Bibliography

[1] S. Williams, A. Waterman, and D. Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *ACM Communications* 52.4 (Apr. 2009), pp. 65–76. DOI: 10.1145/1498765.1498785.