Tools for performance analysis
Optimization training at CINES

Adrien CASSAGNE
adrien.cassagne@inria.fr

2015/12/01
Contents

1 Basic concepts for a comparative analysis
2 Kernel performance analysis
3 Optimization strategy
Contents

1 Basic concepts for a comparative analysis
   - Restitution time
   - Speed up
   - Amdahl's law
   - Efficiency
   - Scalability

2 Kernel performance analysis

3 Optimization strategy

Tools for performance analysis
How to compare two versions of a code?

- The most simplest way is to compare the restitution time (alias the execution time) of the two versions
  - The faster one (shorter time) is the best
- This is simple but we have to remember it when we try to improve the performance of a code
- Be careful to always compare the same time
  - In scientific codes it is very common to have a pre-processing part and a solver part
  - Be sure to measure only the part in which you are interested
  - Otherwise, there is a chance that you will not see the effect of your modification
Measuring the performance of a parallel code

- Time is a basic tool for comparing two versions of a code
  - Consider that we have a time $t_1$ for the sequential version of code
  - If we put 2 cores we can hope to divide the time by 2 ($t_2 = \frac{t_1}{2}$)
  - If we put 3 cores we can hope to divide the time by 3 ($t_3 = \frac{t_1}{3}$)

- The table below shows the execution time of a code named **Code 1**
  - The real time refers to the measured restitution time of **Code 1**
  - The optimal time refers to the best theoretical time ($optiTime = \frac{seqTime}{nbCores}$)

<table>
<thead>
<tr>
<th>nb. of cores</th>
<th>real time</th>
<th>opti. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>98 ms</td>
<td>98.0 ms</td>
</tr>
<tr>
<td>2</td>
<td>50 ms</td>
<td>49.0 ms</td>
</tr>
<tr>
<td>3</td>
<td>35 ms</td>
<td>32.7 ms</td>
</tr>
<tr>
<td>4</td>
<td>27 ms</td>
<td>24.5 ms</td>
</tr>
<tr>
<td>5</td>
<td>22 ms</td>
<td>19.6 ms</td>
</tr>
<tr>
<td>6</td>
<td>18 ms</td>
<td>16.3 ms</td>
</tr>
</tbody>
</table>

Time in function of the number of cores for **Code 1**
Time graph

- The previous table is difficult to read for an analysis.
- It is easier to observe results with a graph.

This graph is not so bad but it is hard to see how far we are from the optimal time...
Introducing speed up

- An other way to compare performance is to compute the speed up.
- The standard is to use the sequential time as the reference time.
- The optimal speed up is always equal to the number of cores we use.

\[ sp = \frac{seqTime}{parallelTime}, \]

with \(seqTime\) the time measured from the 1 core version of the code and \(parallelTime\) the time measured from the parallel version of the code.

<table>
<thead>
<tr>
<th>nb. of cores</th>
<th>real time</th>
<th>speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>98 ms</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>50 ms</td>
<td>1.96</td>
</tr>
<tr>
<td>3</td>
<td>35 ms</td>
<td>2.80</td>
</tr>
<tr>
<td>4</td>
<td>27 ms</td>
<td>3.63</td>
</tr>
<tr>
<td>5</td>
<td>22 ms</td>
<td>4.45</td>
</tr>
<tr>
<td>6</td>
<td>18 ms</td>
<td>5.44</td>
</tr>
</tbody>
</table>

Time and speed up in function of the number of cores for Code 1.
Now, with the speed up, it is much easier to see how far we are from the optimal speed up!
Can we indefinitely put more cores and get better performances?

- Amdahl said no!
- Or, to be more precise, it depends on the characteristics of the code...
- If the code is fully parallel we can indefinitely put more cores and get better performances
- If not, there is a limitation on the maximal speed up we can reach

\[ sp_{\text{max}} = \frac{1}{1 - ft_p}, \]

with \( sp_{\text{max}} \) the maximal speed up reachable and \( ft_p \) the parallel fraction of time in the code \((0 \leq ft_p \leq 1)\).
Amdahl law: example

- If we have a code composed of two parts:
  - 20% is intrinsically sequential
  - 80% is parallel
- What is the maximal reachable speed up?

\[
sp_{\text{max}} = \frac{1}{1-ft_p} = \ldots
\]
Amdahl law: example

- If we have a code composed of two parts:
  - 20% is intrinsically sequential
  - 80% is parallel
- What is the maximal reachable speed up?

\[
sp_{\text{max}} = \frac{1}{1 - ft_p} = \frac{1}{1 - 0.8} = \frac{1}{0.2} = 5.
\]

- We have to try hard to limit the sequential part of the code
- It is essential to reach a good speed up
- In many cases, the sequential part remains in the pre-processing part of the code but also in IOs and communications...
The efficiency is the relation between the real version of a code and the optimal version.

There are many ways to define the efficiency of a code:

- With the speed up: \( eff = \frac{realSp}{optiSp} \)
- With the restitution time: \( eff = \frac{optiTime}{realTime} \)
- Etc.

The efficiency can be expressed as a percentage: \( 0\% < eff \leq 100\% \)

<table>
<thead>
<tr>
<th>nb. of cores</th>
<th>real time</th>
<th>speed up</th>
<th>efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>98 ms</td>
<td>1.00</td>
<td>100%</td>
</tr>
<tr>
<td>2</td>
<td>50 ms</td>
<td>1.96</td>
<td>98%</td>
</tr>
<tr>
<td>3</td>
<td>35 ms</td>
<td>2.80</td>
<td>93%</td>
</tr>
<tr>
<td>4</td>
<td>27 ms</td>
<td>3.63</td>
<td>91%</td>
</tr>
<tr>
<td>5</td>
<td>22 ms</td>
<td>4.45</td>
<td>89%</td>
</tr>
<tr>
<td>6</td>
<td>18 ms</td>
<td>5.44</td>
<td>91%</td>
</tr>
</tbody>
</table>

Time, speed up and efficiency in function of the number of cores for Code 1.
How far we are from the optimal code becomes very clear with the efficiency!
The scalability of a code is its capacity to be efficient when we increase the number of cores.

A code is scalable when it can use a lot of cores.

But, how do we measure the scalability of a code? How do we know when a code is no more scalable?

In fact, there is no easy answer.

However, there are two well-known models for qualifying the scalability of a code:

- Strong scalability
- Weak scalability
Strong scalability

- In this model we measure the code execution time each time we add a core.
- And we keep the same problem size each time: the problem size is a constant.

<table>
<thead>
<tr>
<th>nb. of cores</th>
<th>problem size</th>
<th>real time</th>
<th>speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>98 ms</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>50 ms</td>
<td>1.96</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>35 ms</td>
<td>2.80</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>27 ms</td>
<td>3.63</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>22 ms</td>
<td>4.45</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>18 ms</td>
<td>5.44</td>
</tr>
</tbody>
</table>

Problem size, time and speed up in function of the number of cores for Code 1.
Strong scalability graph

- This is the same graph presented before for the speed up: it represents an analysis of the strong scalability of Code 1.

![Strong scalability of Code 1 (problem size = 100)](chart)

- We can see that the strong scalability of Code 1 is pretty good for 6 cores: we reach a 5.4 speed up, this is not so far from the optimal speed up!
Now we introduce Code 2

Measurements of this code are presented below

<table>
<thead>
<tr>
<th>nb. of cores</th>
<th>problem size</th>
<th>real time</th>
<th>speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>98 ms</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>50 ms</td>
<td>1.96</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>35 ms</td>
<td>2.80</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>32 ms</td>
<td>3.06</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>30 ms</td>
<td>3.27</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>33 ms</td>
<td>2.97</td>
</tr>
</tbody>
</table>

Problem size, time and speed up in function of the number of cores for Code 2
Strong scalability of **Code 2** (graph)

- We can see that **Code 2** has a bad strong scalability.
- But this is not a sufficient reason to put it in the trash!
- What about its weak scalability?
Weak scalability

- In this model we measure the execution time depending on the number of cores.
- And we change the problem size in proportion to the number of cores!
- We cannot compute the speed up because we do not compare same problem sizes.
- But we can compute an efficiency: \( eff = \frac{optiTime}{parallelTime} = \frac{seqTime}{parallelTime} \)

<table>
<thead>
<tr>
<th>nb. of cores</th>
<th>problem size</th>
<th>real time</th>
<th>efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>98 ms</td>
<td>100%</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>100 ms</td>
<td>98%</td>
</tr>
<tr>
<td>3</td>
<td>300</td>
<td>101 ms</td>
<td>97%</td>
</tr>
<tr>
<td>4</td>
<td>400</td>
<td>105 ms</td>
<td>93%</td>
</tr>
<tr>
<td>5</td>
<td>500</td>
<td>109 ms</td>
<td>90%</td>
</tr>
<tr>
<td>6</td>
<td>600</td>
<td>111 ms</td>
<td>88%</td>
</tr>
</tbody>
</table>

Problem size, time and speed up in function of the number of cores for Code 2.
The weak scalability of Code 2 is pretty good (∼ 90% of efficiency with 6 cores)

So, why the strong scalability was so bad?
- Perhaps because the problem size was too small...
- Remember Amdahl’s law, perhaps the parallel fraction of time was not big enough with a problem size of 100
Let’s redo the strong scalability test for Code 2
But with a bigger problem size (600)!

<table>
<thead>
<tr>
<th>nb. of cores</th>
<th>problem size</th>
<th>real time</th>
<th>speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>600</td>
<td>611 ms</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>600</td>
<td>308 ms</td>
<td>1.98</td>
</tr>
<tr>
<td>3</td>
<td>600</td>
<td>210 ms</td>
<td>2.91</td>
</tr>
<tr>
<td>4</td>
<td>600</td>
<td>162 ms</td>
<td>3.77</td>
</tr>
<tr>
<td>5</td>
<td>600</td>
<td>133 ms</td>
<td>4.59</td>
</tr>
<tr>
<td>6</td>
<td>600</td>
<td>111 ms</td>
<td>5.50</td>
</tr>
</tbody>
</table>

Problem size, time and speed up in function of the number of cores for Code 2
With a bigger problem size the strong scalability is much better!

- Strong scalability results are much more dependent on the problem size than for weak scalability
- But it is not always possible to perform a complete weak scalability test
- This is why the two models are complementary to estimate the scalability of a code
Contents

1 Basic concepts for a comparative analysis

2 Kernel performance analysis
   - Flop/s
   - Peak performance
   - Arithmetic intensity
   - Operational intensity
   - Roofline model

3 Optimization strategy
In the previous section, we saw how to compare different versions of a code (tools for a comparative analysis)

But we did not speak about concepts to analyse the performance of the code itself

The number of floating-point operations is an important characteristic of an algorithm

Well-spread in the High Performance Computing world

```
float sum(float *values, int n)
{
    float sum = 0.f;
    // total flops = n * 1
    for(int i = 0; i < n; i++)
        sum = sum + values[i]; // 1 flop because of 1 addition
    return sum;
}
```

Counting flops in a basic `sum` kernel
Floating-point operations per second

- Number of floating-point operations alone is not very interesting
- But with this information we can compute the number of floating-point operations per second (flop/s)!
  - Flop/s is very useful because we can directly compare this value with the peak performance of a CPU
  - With flop/s we can know if we are making a good use of the CPU
  - Today CPUs are very fast and we will use Gflop/s as a standard (1 Gflop/s = 10^9 flop/s)
The peak performance is the maximal computational capacity of a processor.

This value can be calculated from the maximum number of floating-point operations per clock cycle, the frequency and the number of cores:

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

with $nOps$ the number of floating-point operations that can be achieved per clock cycle, $freq$ the processor’s frequency and $nCores$ the number of cores in the processor.
Peak performance of a processor: example

<table>
<thead>
<tr>
<th>CPU name</th>
<th>Core i7-2630QM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Sandy Bridge</td>
</tr>
<tr>
<td>Vect. inst.</td>
<td>AVX-256 bit (4 double, 8 simple)</td>
</tr>
<tr>
<td>Frequency</td>
<td>2 GHz</td>
</tr>
<tr>
<td>Nb. cores</td>
<td>4</td>
</tr>
</tbody>
</table>


The peak performance in simple precision:

\[
peakPerf_{sp} = nOps \times freq \times nCores = (2 \times 8) \times 2 \times 4 = 128 \text{ Gflop/s}
\]

The peak performance in double precision:

\[
peakPerf_{dp} = nOps \times freq \times nCores = (2 \times 4) \times 2 \times 4 = 64 \text{ Gflop/s}
\]

- \(nOps = 2 \times \text{vectorSize}\) because with the Sandy Bridge architecture we can compute 2 vector instructions in one a cycle (add and mul)
Previously we have seen how to compute the Gflop/s of our code and how to compute the peak performance of a processor.

Sometime the measured Gflop/s are far away from the peak performance:
- It could be because we did not optimize well our code.
- Or simply because it is not possible to reach the peak performance.
- In many cases both previous statements are true!

So, with the arithmetic intensity we consider more than just computational things: we add the memory accesses/operations.

\[
AI = \frac{\text{flops}}{\text{memops}}
\]
Arithmetic intensity: example

```c
float sum(float *values, int n)
{
    float sum = 0.f; // we did not count sum as a memop
    // because it is probably a register

    // total flops = n * 1  ||  total memops = n * 1
    for(int i = 0; i < n; i++)
        sum = sum + values[i]; // 1 flop because of 1 addition
        // 1 memop because of 1 access
        // in an wide array (values)

    return sum;
}
```

Counting flops and memops in a basic `sum` kernel

- The arithmetic intensity of `sum` function is: \( AI_{\text{sum}} = \frac{n \times 1}{n \times 1} = 1 \)
- The higher the arith. intensity is, the more the code is limited by the CPU
- The lower the arith. intensity is, the more the code is limited by the RAM
Operational intensity

- Compare to the arithmetic intensity, the operational intensity is slightly different because it also depends on the size of data

\[
OI = \frac{flops}{memops \times sizeOfData} = \frac{AI}{sizeOfData}
\]

- \(sizeOfData\) depends on the type of data we use in our code, \texttt{int} and \texttt{float} are 4 bytes, \texttt{double} is 8 bytes.

- In the previous code (\texttt{sum}) we worked with \texttt{float} so the operational intensity is: \(OI_{sum} = \frac{n \times 1}{(n \times 1) \times 4} = \frac{1}{4}\)

- Like the arithmetic intensity:
  - The higher the ope. intensity is, the more the code is limited by the CPU
  - The lower the ope. intensity is, the more the code is limited by the RAM
Operational intensity

A basic `sum1` kernel in simple precision

```c
// AI = 1 || OI = 1/4
float sum1(float *values, int n)
{
    float sum = 0.f;
    for(int i = 0; i < n; i++)
        sum = sum + values[i];
    return sum;
}
```

A basic `sum2` kernel in double precision

```c
// AI = 1 || OI = 1/8
// this code is more limited by RAM than sum1 code
double sum2(double *values, int n)
{
    double sum = 0.0;
    for(int i = 0; i < n; i++)
        sum = sum + values[i];
    return sum;
}
```
The Roofline model

- The Roofline is a model witch has be made in order to limit the maximal reachable performance
- This model takes into consideration two things
  - Memory bandwidth
  - Peak performance of the processors
- Depending on the operational intensity, the code is limited by memory bandwidth or by peak performance
- Be careful, this model is relevant when the size of data is bigger than the CPU cache sizes!

\[
Attainable \text{ Gflop/s} = \min \begin{cases} 
\text{Peak floating point performance}, \\
\text{Peak memory bandwidth} \times \text{OI}.
\end{cases}
\]
We know how to calculate the CPU peak performance and the operational intensity of a code but have not spoken about the memory bandwidth.

The memory bandwidth is the number of bytes (8 bits) that memory can bring to the processor in one second (B/s or GB/s).

How to know what is memory bandwidth?
- We could theoretically calculate this value.
- But we prefer to measure the bandwidth with a micro benchmark: STREAM.

STREAM is a little code specially made in order to compute the memory bandwidth of a computer.
- It gives good and precise results.
- This is better than the theoretical memory bandwidth because there is always a difference between the theory and the reality...
The Roofline model: example

Here is an example (same as before) of the specifications of a processor with the measured memory bandwidth:

<table>
<thead>
<tr>
<th>CPU name</th>
<th>Core i7-2630QM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Sandy Bridge</td>
</tr>
<tr>
<td>Vect. inst.</td>
<td>AVX-256 bit (4 double, 8 simple)</td>
</tr>
<tr>
<td>Frequency</td>
<td>2 GHz</td>
</tr>
<tr>
<td>Nb. cores</td>
<td>4</td>
</tr>
<tr>
<td>Peak perf sp</td>
<td>128 GFlop/s</td>
</tr>
<tr>
<td>Peak perf dp</td>
<td>64 GFlop/s</td>
</tr>
<tr>
<td>Mem. bandwidth</td>
<td>17.6 GB/s</td>
</tr>
</tbody>
</table>

Specifications from http://ark.intel.com/products/52219
The Roofline model: example

We only keep the needed specifications for the Roofline model:

<table>
<thead>
<tr>
<th>CPU name</th>
<th>Core i7-2630QM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak perf sp</td>
<td>128 GFlop/s</td>
</tr>
<tr>
<td>Peak perf dp</td>
<td>64 GFlop/s</td>
</tr>
<tr>
<td>Mem. bandwidth</td>
<td>17.6 GB/s</td>
</tr>
</tbody>
</table>

We will take the previous sum1 and sum2 codes as an example for the Roofline model.
The Roofline model: example

A basic `sum1` kernel in simple precision

```c
// AI = 1 || OI = 1/4
float sum1(float *values, int n)
{
    float sum = 0.0f;
    for(int i = 0; i < n; i++)
        sum = sum + values[i];
    return sum;
}
```

A basic `sum2` kernel in double precision

```c
// AI = 1 || OI = 1/8
// this code is more limited by RAM than sum1 code
double sum2(double *values, int n)
{
    double sum = 0.0;
    for(int i = 0; i < n; i++)
        sum = sum + values[i];
    return sum;
}
```
The Roofline model: example

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak perf sp</td>
<td>128 GFlop/s</td>
</tr>
<tr>
<td>Peak perf dp</td>
<td>64 GFlop/s</td>
</tr>
<tr>
<td>Mem. bandwidth</td>
<td>17.6 GB/s</td>
</tr>
</tbody>
</table>

We will take the previous `sum1` and `sum2` codes as an example for the Roofline model:

- The `sum1` operational intensity is $\frac{1}{4}$
- The `sum2` operational intensity is $\frac{1}{8}$

Let’s see what is the attainable performance with the Roofline model:

\[
\text{Attainable Gflop/s} = \min \left\{ \text{Peak floating point performance, } \right. \\
\left. \text{Peak memory bandwidth } \times \text{OI.} \right\}
\]

\[
\Rightarrow \quad \text{Attainable Gflop/s}_{\text{sum1}} = \min \left\{ 128 \text{ Gflop/s, } \right. \\
\left. 17.6 \times \frac{1}{4} \text{ Gflop/s. } \right\} = 4.4 \text{ Gflop/s}
\]
The Roofline model: example

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Peak perf sp</strong></td>
<td>128 GFlop/s</td>
</tr>
<tr>
<td><strong>Peak perf dp</strong></td>
<td>64 GFlop/s</td>
</tr>
<tr>
<td><strong>Mem. bandwidth</strong></td>
<td>17.6 GB/s</td>
</tr>
</tbody>
</table>

We will take the previous `sum1` and `sum2` codes as an example for the Roofline model:

- The `sum1` operational intensity is $\frac{1}{4}$
- The `sum2` operational intensity is $\frac{1}{8}$

Let’s see what is the attainable performance with the Roofline model:

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

$$\Rightarrow$$

$$\text{Attainable Gflop/s}_{\text{sum2}} = \min \left\{ \begin{array}{l}
64 \text{ Gflop/s,} \\
17.6 \times \frac{1}{8} \text{ Gflop/s.}
\end{array} \right. = 2.2 \text{ Gflop/s}$$
The Roofline model: example on a graph

- The graph below represents the Roofline for the previous processor.
- There are two different Rooflines:
  - One for the simple precision floating-point computations.
  - One for the double precision floating-point computations.

Here, it is clear that the `sum1` and `sum2` codes are limited by the memory bandwidth.
Contents

1 Basic concepts for a comparative analysis

2 Kernel performance analysis

3 Optimization strategy
   - Optimization process
   - Code bottleneck
   - Profilers
The optimization process

- Optimize a code is an iterative process
  - Firstly we have to measure or to profile the code
  - And secondly we can try optimizations (taking the profiling into consideration)
In the profiling part we have to determine the code bottlenecks:
- Memory bound
- Compute bound

We can use the previous the Roofline model to do that:
- This is a very good way to understand the code limitations and the code itself!

But sometimes the code is too big and we cannot apply the Roofline model everywhere (too much time consuming):
- We can use a profiler in order to detect hotspots in the code
- When we know hotspot zones we can apply the Roofline model on them!
Some profilers

- There are a lot of profilers
  - gprof
  - Tau
  - Vtune
  - Vampir
  - Scalasca
  - Valgrind
  - Paraver
  - PAPI
  - Etc.

- The most important feature of a profiler is to easily see which part of the code is time consuming
  - It is that part of the code we will try to optimize

- Of course we can do much more than that with a profiler but this is not in the range of this lesson
### gprof example

**Flat profile:**

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>time</th>
<th>% cumulative self</th>
<th>time</th>
<th>% cumulative self</th>
<th>time</th>
<th>% cumulative self</th>
<th>time</th>
<th>% cumulative self</th>
<th>time</th>
<th>% cumulative self</th>
<th>time</th>
<th>% cumulative self</th>
<th>time</th>
<th>% cumulative self</th>
<th>time</th>
<th>% cumulative self</th>
<th>time</th>
<th>% cumulative self</th>
<th>time</th>
<th>% cumulative self</th>
<th>time</th>
<th>% cumulative self</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.94</td>
<td>1.01</td>
<td>6.81</td>
<td>1.47</td>
<td>5.84</td>
<td>1.87</td>
<td>5.77</td>
<td>2.26</td>
<td>5.62</td>
<td>2.64</td>
<td>3.70</td>
<td>2.89</td>
<td>3.55</td>
<td>3.13</td>
<td>3.55</td>
<td>3.37</td>
<td>3.11</td>
<td>3.58</td>
<td>2.96</td>
<td>3.78</td>
<td>2.81</td>
<td>3.97</td>
</tr>
<tr>
<td>1.01</td>
<td>1.01</td>
<td>0.46</td>
<td>0.46</td>
<td>0.40</td>
<td>0.40</td>
<td>0.39</td>
<td>0.39</td>
<td>0.38</td>
<td>0.38</td>
<td>0.25</td>
<td>0.25</td>
<td>0.24</td>
<td>0.24</td>
<td>0.21</td>
<td>0.21</td>
<td>0.20</td>
<td>0.20</td>
<td>0.19</td>
<td>0.19</td>
<td>0.18</td>
<td>0.18</td>
</tr>
</tbody>
</table>

**Remarks:**
- __intel_new_memcpy__ very typical syndrome in C++ codes
- pass2__ most time consuming code routine
- Complexe::Complexe(...) related to __intel_new_memcpy
- Complex::operator=(...) related to __intel_new_memcpy
- Complexe::operator=(...) probably an external call
- _ZN8ComplexeC9Edd probably an external call
- second most time consuming routine

**gprof flat profiling of a code**