Licenciatura em Engenharia Informática

e Computação

2021/2022 | 2nd Semester

Performance Evaluation of a Single Core

Parallel and Distributed Computing

Bruno Rosendo | up201906334@up.pt

João Mesquita | up201906682@up.pt

Rui Alves | up201905853@up.pt

27/03/2022

Problem Description

The objective of this project is to study the impact of memory hierarchy on the processor performance when accessing large amounts of data. For this purpose, the product of two matrices will be used, with different algorithms and respective implementations. In order to collect relevant performance indicators, such as execution time and Cache misses, the Performance API (PAPI) was used, along with the C++ and Java standard libraries.

Algorithms Explanation

Naive Multiplication

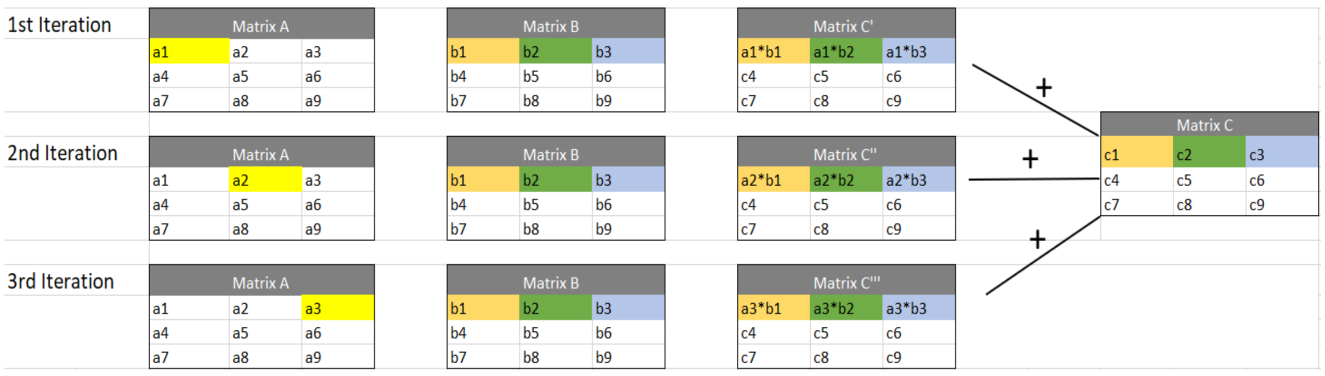
The Naive Multiplication of two matrices A and B consists of iterating the rows of the first matrix A and multiplying by each column of the matrix B, in the following manner:

```
for (i=1; i<n; i++)
  for (j=1; j<n; j++)
    for (k=1; k<n; k++)
      c[i,j]+= a[i,k]*b[k,j]
```

Changing the order of multiplications

This algorithm is almost identical to the previous one. However, the idea here is to switch the order of the last two *for cycles*, in order to optimize the use of the cache by the processor (this is further explained in the *Results and Analysis* section). This way, the multiplication is done by multiplying each element of the matrix A by the corresponding line of the matrix B. Therefore, the algorithm will look like this:

```
for (i=1; i<n; i++)
    for (k=1; k<n; k++)
        for (j=1; j<n; j++)
            c[i,j]+= a[i,k]*b[k,j]
```



1st Iteration

Matrix A		
a1	a2	a3
a4	a5	a6
a7	a8	a9

Matrix B		
b1	b2	b3
b4	b5	b6
b7	b8	b9

Matrix C'		
a1*b1	a1*b2	a1*b3
c4	c5	c6
c7	c8	c9

2nd Iteration

Matrix A		
a1	a2	a3
a4	a5	a6
a7	a8	a9

Matrix B		
b1	b2	b3
b4	b5	b6
b7	b8	b9

Matrix C''		
a2*b1	a2*b2	a2*b3
c4	c5	c6
c7	c8	c9

3rd Iteration

Matrix A		
a1	a2	a3
a4	a5	a6
a7	a8	a9

Matrix B		
b1	b2	b3
b4	b5	b6
b7	b8	b9

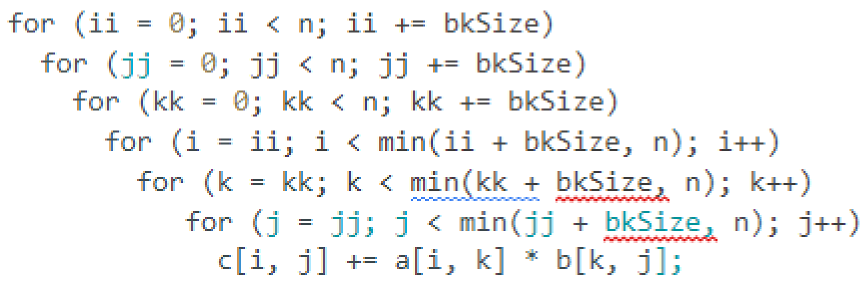
Matrix C'''		
a3*b1	a3*b2	a3*b3
c4	c5	c6
c7	c8	c9

Matrix C		
c1	c2	c3
c4	c5	c6
c7	c8	c9

Image 1: Flow of the second multiplication algorithm

Block Matrix Multiplication

With the goal of achieving a better reuse of data in the local memory (cache), the previous algorithms, which refer to individual elements of the matrices, can be replaced by one that operates on submatrices, called **blocks**. The advantage of this method is that the small blocks can be moved into the faster layers of cache, depending on their size (bkSize), and their elements can be repeatedly reused. The computations on the blocks are made the same way they were in the second algorithm, to try and achieve the best reuse of the processor's cache. Hence, this approach will look like the following:



```
for (ii = 0; ii < n; ii += bkSize)
  for (jj = 0; jj < n; jj += bkSize)
    for (kk = 0; kk < n; kk += bkSize)
      for (i = ii; i < min(ii + bkSize, n); i++)
        for (k = kk; k < min(kk + bkSize, n); k++)
          for (j = jj; j < min(jj + bkSize, n); j++)
            c[i, j] += a[i, k] * b[k, j];
```

$$A = \left[\begin{array}{cc|cc} 1 & 2 & 2 & 7 \\ 1 & 5 & 6 & 2 \\ \hline 3 & 3 & 4 & 5 \\ 3 & 3 & 6 & 7 \end{array}\right] \quad <=> \quad \begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix}$$

$$B = \left[\begin{array}{cc|cc} 3 & 1 & 1 & 4 \\ 7 & 4 & 5 & 3 \\ \hline 2 & 4 & 2 & 4 \\ 3 & 1 & 8 & 5 \end{array}\right] \quad <=> \quad \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix}$$

$$C = A * B = \begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix} = \begin{bmatrix} A11*B11+ & A11*B12+ \\ A12*B21 & A12*B22 \\ A21*B11+ & A21*B12+ \\ A22*B21 & A22*B22 \end{bmatrix}$$

Image 2: Block Matrix Multiplication Algorithm

Performance Metrics

In order to successfully study the impact of the memory hierarchy on the processor performance when accessing large amounts of data, it is important to choose a set of performance metrics that lets us compare the efficiency of the different algorithms and data sizes. For this purpose, the metrics used in this study were the following:

- **Execution time:** Real execution time of the matrix multiplication, measured by the C++ or Java standard libraries. An important metric which estimates the impact of the algorithm in real life applications.
- **GFlops:** A derived measure, calculated with the formula ***(2 * N^3) / T***, where N is the number of lines of the matrix (NxN) and T is the execution time. This is a standard measure of computer performance, which evaluates the number of floating point operations per second.

- **Data Cache Misses (DCM):** Since we're studying the influence of the cache on the processor performance, it is important to compare the frequency of cache misses between the algorithms. For that purpose, the number of misses is directly measured by PAPI, which is only available for the C/C++ language.
- **Data Cache Hits (DCH):** This metric is useful to calculate the percentage of Data Cache Misses, relative to the total number of accesses. To do that, these accesses are directly measured by PAPI, which is only available for the C/C++ language.
- **Percentage of Data Cache Misses (% DCM):** A derived measure, calculated with the formula ***(DCM / (DCM + DCH)) * 100***, where *DCM* and *DCH* are the metrics explained above. This is important to directly compare the efficiency of the cache usage by the different algorithms.

Results and Analysis

Using the algorithms and performance metrics explained above, it was possible to measure and analyze the following results. It should be noted that the L1 cache's miss percentage was not possible to obtain, since none of the group's elements owned a processor permissive of reading the cache's data hits or accesses.

Naive Multiplication

C++:

Matrix Size	Time (s)	GFlops / s	DCM L1	DCM L2	DCH L2	DCM L2 (%)
600x600	0.223	1.937	244 560 243	7 915 044	238 833 002	3.21
1000x1000	1.474	1.357	1 231 990 925	76 969 658	1 061 103 479	6.76
1400x1400	6.263	0.876	3 446 520 842	352 133 818	2 747 871 324	11.36
1800x1800	25.860	0.451	9 084 128 288	780 837 346	5 808 340 864	11.85
2200x2200	59.384	0.359	1 7645 799 085	1 537 694 373	10 507 321 345	12.77
2600x2600	99.954	0.352	30 895 382 483	3 226 483 656	16 766 181 399	16.14
3000x3000	161.460	0.334	50 316 669 133	8 111 919 195	23 242 070 966	25.87

Table 1: Naive Matrix Multiplication C++ Results

Java:

Matrix Size	Time (s)	GFlops / s
600x600	0.878	0.492
1000x1000	4.167	0.480
1400x1400	16.183	0.339
1800x1800	36.078	0.323
2200x2200	70.971	0.300
2600x2600	126.327	0.278
3000x3000	191.421	0.282

Table 2: Matrix Multiplication Java Results

As we can see, in the naive algorithm, the performance (time and GFlops) and percentage of cache misses greatly increases with the matrix size, since the order of the inner loops of the matrix operations does not optimize the use of the cache, resulting in a greater number of misses. This was true for both C++ and Java. The latter showed worse performance because of the nature of Java, which has a higher abstraction level.

Changing the order of multiplications

C++:

Matrix Size	Time (s)	GFlops / s	DCM L1	DCM L2	DCH L2	DCM L2 (%)
600x600	0.100	4.320	286 796	597 491	25 621 827	2.28
1000x1000	0.634	3.155	1 760 833	3 521 665	79 474 344	4.24
1400x1400	1.765	2.109	4 922 082	9 651 142	207 070 103	4.54
1800x1800	3.918	2.977	11 130 866	20 612 714	444 836 992	4.43
2200x2200	9.090	2.996	19 381 091	24 226 364	1 509 393 472	1.54
2600x2600	11.591	3.032	40 222 447	39 824 205	3 505 284 114	1.12
3000x3000	24.264	3.054	62 074 365	61 459 796	5 411 154 262	1.12

Table 3: Different order of multiplications C++ Results

Java:

Matrix Size	Time (s)	GFlops / s
600x600	0.212	2.038
1000x1000	1.230	1.626
1400x1400	3.334	1.646
1800x1800	7.104	1.642
2200x2200	8.763	2.430
2600x2600	15.660	2.245
3000x3000	23.102	2.337

Table 4: Different order of multiplications Java Results

This time, with a small change, the performance of the operations was much better than with the naive algorithm, which confirms the hypothesis of a better use of the cache. This can also be seen in the decrease of cache misses, relatively to the naive algorithm.

Block Matrix Multiplication

Matrix Size	Time (s)	GFlops / s	DCM L1	DCM L2	DCH L2	DCM L2 (%)
4096x4096	33.435	4.110	1 498 905 128	4 788 834 276	6 464 354 587	42.56
6144x6144	99.179	4.677	2 715 124 754	8 758 466 947	30 056 258 949	22.56
8192x8192	267.490	4.110	9 735 629 596	31 405 256 761	59 120 213 902	34.69
10240x10240	487.566	4.404	22 103 581 644	69 073 692 638	105 467 801 014	39.57

Table 5: Block matrix multiplication 128 Block Size - Results

Matrix Size	Time (s)	GFlops / s	DCM L1	DCM L2	DCH L2	DCM L2 (%)
4096x4096	27.124	5.067	953 662 498	2 384 156 244	7 530 893 028	24.05
6144x6144	144.712	3.205	3 524 470 964	8 811 177 410	22 425 485 730	28.208
8192x8192	467.726	2.351	10 023 014 374	23 864 319 938	32 900 534 165	42.04
10240x10240	530.264	4.050	16 024 370 227	39 083 829 821	109 240 769 865	26.35

Table 6: Block matrix multiplication 256 Block Size - Results

Matrix Size	Time (s)	GFlops / s	DCM L1	DCM L2	DCH L2	DCM L2 (%)

4096x4096	41.296	3.328	632 787 617	1 346 356 633	6 789 839 416	16.55
6144x6144	162.378	2.857	1 741 446 907	3 705 206 186	14 803 000 278	20.01
8192x8192	381.558	2.882	4 829 854 449	10 499 683 584	37 587 241 348	21.83
10240x10240	688.802	3.112	7 889 107 870	16 785 335 893	89 461 344 299	31.18

Table 7: Block matrix multiplication 512 Block Size - Results

By applying the *Block Matrix Algorithm*, there's a noticeable improvement in the execution time and GFlops of the program. This is thanks to the separation in smaller blocks, which theoretically reduces the amount of cache misses. Surprisingly, the DCM rate in L2 proved to be higher than expected. We believe the main reason for this increment in DCM in L2 is thanks to an increasing amount of hits in L1 cache. Hence, the algorithm is more efficient, because it uses the upper layer of cache more efficiently, but, as a consequence, the number of times that a L1 miss results in a L2 miss increases, resulting in a higher percentage of data cache misses in the second layer. Unfortunately, we weren't able to confirm this since we didn't have access to a computer that was capable of reading the L1 cache's data hits or accesses.

It should be noted that, in many of the measurements, there were more data cache misses in the L2 layer than in the L1 layer. This phenomenon was more noticeable as the matrix size increased. A possible reason for this is optimizations performed in the processor's cache management, that could be skipping the L1 cache lookup, knowing the matrix elements were previously prefetched into L2, since the prefetching is stronger in this layer.

Conclusions

Memory hierarchy is separated into different levels, distinguished by their response time, complexity and capacity, meaning they have different performances. Using the matrix multiplication example, it is possible to conclude that not only does the memory hierarchy influence performance but also the use of that hierarchy by the programmer.

References

- Materials from the Parallel and Distributed Computing classes
- Block algorithms: Matrix Multiplication as an example
- Compiler Optimizations effect on FLOPs and L2/L3 Cache Miss Rate using PAPI
- Prefetching data at L1 and L2