github.com/SilvioBaratto/Lab-1

Report Lab 1

Silvio Baratto, Tobias Ganzmann

18 ottobre 2022

Indice

1 PERF (I)

1.1 Definition profiler

A profiler in software engineering, is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization, and more specifically, performance engineering. Profilers, which are also programs themselves, analyze target programs by collecting information on their execution. Based on their data granularity, on how profilers collect information, they are classified into event based or statistical profilers.

1.2 Code explanatin for benchmarking

The code given to do some performance analysis using perf as a profiler consist in a simple function that initialize a matrix of dimension 2000×2000 and fill all the matrix with a value equal to 1. The program repeat this operation 100 times.

1.3 Performance analysis tool (perf)

We are interested in see how the program behaves with different optimizers and code optimizations using perf to check the performance results. To do that with this instructions we want to look at:

a. CPU clock cycles at user level

b. Machine code instructions at user level

c. First-level data cache load hitst at user level

d. First-level data cache load misses at user level

e. First-level data cache store hits at user level

f. First-level data cache store misses at user level

g. Last-level cache load hits at user level

h. Last-level cache load misses at user level

i. Last-level cache store hits at user level

j. Last-level cache store misses at user level

The way how we obtain the following benchmark results is from this terminal command:

```
$ sudo perf stat -e cycles:u -e instructions:u \
                      -e L1-dcache-loads:u -e L1-dcache-load-misses:u \
                      -e L1-dcache-stores:u -e L1-dcache-store-misses:u \
                      -e LLC-loads:u -e LLC-load-misses:u \
                      -e LLC-stores:u -e LLC-store-misses:u ./task1
```

From the first run of the code we obtain the following results:

```
CPU = 5.893150 ms

 Performance counter stats for './task1':

     2,743,735,005      cycles:u                  (32.74%)
     3,199,246,694      instructions:u            (44.26%)
       402,366,995      L1-dcache-loads           (55.79%)
       803,099,963      L1-dcache-load-misses     (67.31%)
       799,527,269      L1-dcache-stores          (67.46%)
   <not supported>      L1-dcache-store-misses
         1,628,527      LLC-loads                 (67.46%)
             1,094      LLC-load-misses           (44.21%)
         2,479,174      LLC-stores                (21.69%)
               258      LLC-store-misses          (21.69%)

       0.590644181 seconds time elapsed

       0.590584000 seconds user
       0.000000000 seconds sys
```

Changing the code in such way that array is now traversed by rows instead of by columns, hence taking advantage of both the row major order used in C and the cache hierarchy we obtain the following results:

```
CPU = 1.423880 ms

 Performance counter stats for './task1':

       648,114,804      cycles:u                  (32.97%)
     1,624,440,721      instructions:u            (44.14%)
       396,820,828      L1-dcache-loads           (55.31%)
        12,204,130      L1-dcache-load-misses     (66.48%)
       383,946,246      L1-dcache-stores          (66.48%)
   <not supported>      L1-dcache-store-misses
             1,680      LLC-loads                 (66.48%)
                40      LLC-load-misses           (44.69%)
         6,988,534      LLC-stores                (22.35%)
                67      LLC-store-misses          (22.35%)

       0.143955657 seconds time elapsed

       0.135983000 seconds user
       0.007999000 seconds sys
```

From the graph below we can see how much changing to the row major perform better on cycles, instructions and cache.

1.4 Conclusion

2 PERF (II)

3 GPROF

a Functions with high "self ms/call" (shown in green) values and a big percentage of running time are considered our bottlenecks. The cannot be divided into smaller pieces and therefore not be parallelized.

b In contrast, functions with low "self ms/call" (shown in blue) values and frequent calls are already divided up in small, very managable chunks and can therefore be parallelized. In this case we divide the matrices in chunks and loop over them seperately in parallel (for example domain decomposition).

Time %	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name	total time ms
8.14	0.5	0.07	1	70	90.54	Label_Binary_Image	70
6.98	0.68	0.06	1	60	173.39	Binary_Dilation	60
6.98	0.62	0.06	2	30	30	BMP_Read_Color_Image	60
3.49	0.82	0.03	1	30	61.07	Or_Gray_Images	30
3.49	0.79	0.03	2	15	15	BMP_Write_Gray_Image	30
3.49	0.76	0.03	3	10	30.53	Binarize_Gray_Image	30
1.16	0.86	0.01	1	10	10	BMP_Write_Color_Image	10
5.81	0.73	0.05	7	7.14	7.14	BMP_Create_Gray_Image	49.98
1.16	0.85	0.01	2	5	35	Color_Image_to_Gray	10
22.09	0.19	0.19	234249700	0	0	BMP_Read_Gray_Pixel	0
9.3	0.27	0.08	103980960	0	0	BMP_Write_Gray_Pixel	0
9.3	0.35	0.08	51966720	0	0	BMP_Read_Color_Pixel	0
9.3	0.43	0.08	0	0	0	_init	0
6.98	0.56	0.06	13014556	0	0	BMP_Write_Color_Pixel	0
1.16	0.83	0.01	12991680	0	0	Read_Label	0
1.16	0.84	0.01	296827	0	0	Update_Lookup	0
0	0.86	0	1187308	0	0	Minimum_Label_Neighbor	0
0	0.86	0	1187308	0	0	Update_Lookup_Neighbor	0
0	0.86	0	296827	0	0	Label_Pixel	0
0	0.86	0	296827	0	0	Minimum_Label	0
0	0.86	0	7	0	0	BMP_Free_Gray_Image	0
0	0.86	0	3	0	0	BMP_Free_Color_Image	0
0	0.86	0	1	0	99.89	Add_Color_Images	0
0	0.86	0	1	0	0	BMP_Create_Color_Image	0
0	0.86	0	1	0	213.67	Binary_Closing	0
0	0.86	0	1	0	33.13	Binary_Erosion	0
0	0.86	0	1	0	0	Create_Label_Array	0
0	0.86	0	1	0	0	Destroy_Label_Array	0
0	0.86	0	1	0	0	Readjust_Lookup	0

4 OpenMP (Compilation and basic directives)

5 Optional assignment