02258 Parallel Computer Systems Fall 2022

First Report

Tomás Estácio
Department of Applied
Mathematics and Computer
Science
Technical University of Denmark
Lyngby, Copenhagen, Denmark
s223187@dtu.dk

KEYWORDS

Cache, Performance, Speedup, Parallelization, SIMD, Intrinsic Function, HPC machines.

1. First Databar: Cache Performance

1.1 Observing Cache Performance

After logging on the HPC machines provided by DTU, downloading a program code written in C called *cachetest.c*, studying it, compiling, and executing it, there were two questions to be answered in the report about the problem in hand:

- How many and how large caches does the machine have and how did you arrive to that?
- How do you need to write your programs to suit the memory hierarchy?

For the first question, I started by analyzing the code that was given and tried to understand what its output would be. The purpose of it was to repeat the same instruction of accessing the cache memory by performing a load and a store of the same address and trying to obtain the time spent in that access to memory, using an empty loop performing a “dummy” code to subtract the overhead time associated to the instruction, managing to have a more real sample of the actual time for access to the cache memory. These attempts would be done and the time for each of them would be printed to the terminal. This execution would be all repeated because they were inside a for loop that would double the size of the cache for each iteration, until we have reached the maximum size of 16777216 bytes.

From the output of the program code, I understood that the first values of the measured time are negative, because the time measured for the “dummy” code is larger than the time measured for the instructions we actually want to measure, as it is not possible to perform prefetching in the “dummy” code since the code performs a change always in the same variable, but it is possible in the instructions we wanted to measure as it is always accessing different variables through the array. Then, I was able to verify that there are three significant changes in the measured time that are worthy of more careful analysis. The first occurrence appears in cache size of 65536 bytes, with a stride of 64 bytes to the access of memory, where the time increased from -2.11ns to -0.12ns, suggesting that there was an access to a new level of cache, the L2 cache level. That also means the size of the L1 cache level is close to 65536 bytes, most likely 64Kbytes. Normally, the L1 cache level is divided in the L1-I level and the L1-D level, one for instructions and the other for data, making the size of them 32Kbytes each.

The next moment where the was a sudden increase of the time was for cache size of 524288 bytes, when we had two increases: from -2.17ns to -1.24ns and from -1.24ns to 1.40ns, suggesting that we went from accessing L1 cache, to L2 cache and then to L3 cache, making it possible to understand that the size of the L2 cache is 256Kbytes or 512Kbytes, most likely the first option.

The final moment which we had a sudden increase of the measured time was for a cache size of 33554432 bytes, when we had these increases: from -2.13ns to -1.11ns, from -1.11ns to 1.46ns, from 6.46ns to 9.54ns and from 9.54ns to 12.12ns, with strides of 8, 16, 32, 64 and 128 bytes. From these results, we can conclude that, for this size of cache, there was a need to access L1, L2 and L3 levels of cache and the main memory. The size of the L3 level cache is most likely close to 25600Kbytes.

To confirm my conclusions, I used the command line *lscpu* in the terminal to confirm the number of cache levels and their sizes of the HPC machine that was accessing through ThinLinc.

For the second question, writing programs that need to access different levels of the memory hierarchy can result in lower performance of the system, so it is required to use several techniques to enhance it. Firstly, it is important to know how the memory of your specific machine is structured so that you can predict when you will need to access a lower level of the hierarchy and try to minimize those situations. Some techniques like memory allocation - managing memory in power of two increments, fixed and dynamic partitioning in Operating System are also great solutions to implement in your program to reduce the overhead of the accesses to the different levels of the memory.

1.2 Working with SIMD instructions

For this part of the Databar, the objectives were trying to understand and working with intrinsic functions and SIMD instructions, mainly the ones highlighted in the document published by Intel intitled “Intel Intrinsic Guide”.

After the introduction to the topics, there was two exercises to be resolved. The first one was to use intrinsic functions to vectorize a piece of code that performed the sum of all the elements of a vector. After writing the code using SIMD instructions, compiling it using the command line *gcc -O2 -std=gnu99 -msse4.1 sum.c* and executing it using *./a.out*, I was able to conclude that there was an enhancement in the performance of the code, comparing it to the sum_naive, because each register that was used had a size of 128 bits instead of 32 bits, making it possible to operate in 4 integers at a time for each instruction called. This is exactly the purpose of programming using SIMD, minimizing the number of instructions calls to improve the performance of the program.

To have a sample of the average speedup that was obtained in this exercise, I collected 15 execution times of the functions used and calculated the average speedup:

$$S = \frac{Texec(sum\_naive)}{Texec(sum\_vectorized)} = 2.9321 \quad (1)$$

The value obtained, theoretically, I expected to be closer to 4 how I explained before, however this was the real value obtained using the source code I wrote.

The second exercise was to also vectorize a piece of code that performed a loop unrolling in a vector to perform the sum of its elements.

After writing the function sum_vectorized_unrolled, compiling the program, and executing it, the conclusions obtained were very similar to the ones in the previous exercise. Using the same logic, I was able to perform the same number of instructions that were done in the sum_unrolled function but managed to deal with 4 times more data in each iteration of the loop, all thanks to the use of intrinsic functions, that allow the user to use 128 bits registers.

Again, I collected 15 values of the execution times of the functions used and calculated the average speedup:

$$S = \frac{Texec(sum\_unrolled)}{Texec(sum\_vectorized\_unrolled)} = 3.8906 \quad (2)$$

In this case, I was able to obtain a value closer to 4 for the speedup, which is more realistic because, in each iteration of the loop inside the function sum_vectorized_unrolled, I work with 4 times more data than in the function sum_unrolled.

2. Second Databar: Performance Counters

2.1 Working with performance counters

In this Databar, I was challenged to use hardware performance counters to check the execution of four different programs: a-v1.c, a-v2.c, b-v1.c and b-v2.c. The counters used, specifically four sub-commands of the Linux *perf* tool, like *perf list, perf stat, perf record* and *perf report*, are able to give an insight into the execution and the performance of each program.

The first Linux sub-command *perf list* gives a list of the events available in the DTU’s HPC machine accessed, like CPU-cycles, instructions, cache-references, cache-misses and some more. The *perf stat* sub-command allows to run a program with one or more events counted during the execution. In the end, it prints out the number of times each selected event occurred. The *perf record* sub-command records both the number of events that occurred and the location where those events happened as the program executed. Finally, the *perf report* sub-command gives the user the possibility to view the results of the execution, navigating by function names, seeing which instructions are associated with which fractions of events.

Using these perf tools, it became possible to have a clearer understanding of the differences between the performance in a-v1.c and a-v2.c programs. Analyzing the two different program codes, it was clear to understand that the difference between them was that in the program a-v2.c there is a call to the function qsort that sorts the array, from smallest to biggest element, making it a lot more predictable to the processor, before accessing all his elements. This was the start to trying to understand the performance difference between them, but the *perf* tools gave a lot clearer understanding of the cause.

Firstly, I started by checking the number of instructions of both programs and it looked like the program a-v2.c had a slightly higher number of instructions than the program a-v1.c, despite the execution time of version 2 is smaller. So, I decided to look to the statistics related to the CPU cycles, and there it showed that the a-v2.c had a lot less CPU cycles than a-v1.c, which makes sense considering version 2 runs faster than version 1. Finally, I analyzed the number of branches and branches misses for both the programs, and arrived at the conclusion that, in a-v1.c there is a 24.43% of branch-misses but only a 0.032% of branch-misses in a-v2.c, making the execution time of the a-v1.c program higher because of the penalty time, harming the performance.

Secondly, performing the *perf record -e cycles* followed with the executable program, we were able to see the number of samples of data written in each of the programs, being the value of the version 1 (11031 samples) much larger than the version 2 (3912 samples).

Finally, performing the *perf report* on the version 2 program shows that 99.34% of instructions were performed inside the main function and then, inside it, a large percentage of the cycles are performed inside the two for loops, one that goes from 0 to ITERATIONS-1 and other that goes from 0 to ARRAY_SIZE-1, where the code performs the sum of the elements inside the array data, if that element if bigger or equal than 128. For the version 1 program, the sub-command shows that 99.96% of instructions were performed inside the main function, a very close percentage to the one obtained in other program, which does not constitute any surprise. However, analyzing the percentage of cycles performed in different parts of the program, it is possible to conclude that the compare instruction takes less cycles in the version 1 of the program, but the instruction related to accessing memory addresses take up more percentage than in the version 2 program, being that the probable cause for the high percentage of branch-misses. This conclusion checks out with the fact that the array that was not sorted from smallest to biggest element, with the use of the function qsort, making it less predictable to the processor fetching the next value of the element.

In the part B of this Databar, I was challenged to use the same hardware performance counters to compare to similar programs that have very different performance.

Before using the *perf* tools, I decided to look at both codes of the programs and figure out the main differences between them. The main difference was in the choice of which element of the array would be chosen to be summed: in the program b-v1.c, the index starts at 0 and it becomes the value of the element in the previous index, making the access to the elements more unpredictable and, because of that, with a worse performance comparing to the program b-v2.c, which index goes from 0 to ARRAY_SIZE-1, one by one, being a lot more predictable and so with a better performance.

To verify my previous conclusions, I used the *perf stat -e cycles ./b-v1* and the *perf stat -e cycles ./b-v2* and verified that the number of CPU cycles executed for each program are very different, being a lot bigger for b-v1.c than b-v2.c. Following that sub-command, I used *perf record* and *perf report* to help with more arguments to my conclusions, where I noticed that the number of samples of data written on version 1 of the program (6309 samples) were bigger than on version 2 (3953 samples) and that the percentage of cycles in the version 1 of the program is focused on the following instructions: 33.65% of the CPU cycles are spent on the addition of the variables SUM and data[index], 24.90% of the CPU cycles are spent on *index = data[index]* line code and 25.05% of the CPU cycles are spent on the jump if index not equal to -1 instruction, and all these values show that the first version of the program on changing the value of index to the value of the several elements of the array, comparing index to -1 and doing the addition of the SUM variable and each of the elements of the data array. For the version 2 of the program, the focus is primarily on the sum of index and 1, with 58.17% of the CPU cycles, the addition of the variable SUM with the elements of the data array with 29.23% and moving the result of that addition to the variable SUM, with 12.43%. These results make us conclude that the version 1 is less efficient than version 2 because it takes more CPU cycles to go through the data array, spending a higher percentage on the *index = data[index]* line and on the jump if not equal instruction, which uses the variable changed in the previous instruction, and because version 2 has the CPU focus on mostly arithmetic instructions that take less time to perform.

In conclusion, the issue of performance for these programs are best discovered using this performance tools, specifically the use of Linux commands, and a close analysis to the program code.

3. Third Databar: Thinking parallel

3.1 Parallelism in code

For this Databar, the task was to conceptually write a parallel version of the program calc.c, but there was no need to write code, just analyze it and figure out which parts of the code can be made parallel and which parts must remain serial.

From analyzing the code, it was easy to comprehend that the first for loop that sets every element in the *from* array to the value of 0.0 and the for loop that goes through every element of the *to* and *from* array, except the first and last, and performs the following instructions:

to[i] = from[i] + 0.1(from[i-1] – 2*from[i] + from[i+1]*

To perform parallelism in these pieces of code, we would need to divide the for loop in parts so that the number of processors that we want to use could work on them at the same time, improving the overall performance of the program.

However, there are specific parts of the program that cannot really become parallel, like the following piece of code:

double tmp = from;*
from = to;
to = tmp;

These lines of code can't be parallelized because of mutual dependences they have to be performed in that specific order, one after another. For the same reasons, the for loop in the code where there's a printf function cannot also be parallelized because we must ensure that it is made in a certain order. This is a problem that limits the performance of the program, but the best option here is just leave them as serial code and use parallelism to the previous referenced parts of the program, which already makes the performance of the program a lot better.

The next question in the Databar instructions was about the measurement of time in specific parts of the program that have very short execution time, which normally is made by using the function available in the C library <*sys/time.h*> that's called *gettimeofday*, which we would need to call two times, in the beginning of the instructions we want to measure and, in the end, and just subtract the values to obtain the time the instructions take. The tricky part about using this function is that it returns both the number of seconds and microseconds in separate long int variables, so that to obtain the total time including microseconds you need to sum both together accordingly.

To calculate the value of the performance of serial program we use the formula

$$P_f^s = \frac{s+p}{Tserial} \qquad (3)$$

where s represents the serial part of the program, p represents the parallel part of the program and Tserial represents the runtime of the serial code. To calculate the value of the performance of the parallel program we use the formula

$$P_f^p = \frac{1}{s+\frac{1-s}{N}} \qquad (4)$$

where N represents the number of processors used in the parallel code. Finally, to calculate the speedup obtained by introducing parallelism in the program we use the formula

$$S_f = \frac{Pfp}{Pfs} = \frac{1}{s+\frac{1-s}{N}} \qquad (5)$$

After being able to do these calculations, I expect that the code would increase its performance depending on the number of processors working at the same time. From my calculations and analysis, the conclusion I get is that the performance would increase N/2 times, because, by using the *perf report* Linux sub-command, I was able to check that the part of the code that can be parallelized takes about half of the percentage of CPU cycles and the other half is occupied by serial code that does not receive any growth in performance. So, by that logic, I expect a speedup of about N/2, being N, the number of processors used for the program.

Regarding using the Amdahl's law to predict the performance of using parallelism in the program, it will always give us an upper bound of the value of speedup our program can have. However, there are several other costs that is needed to overcome when dealing with parallelism, like each task start up and termination time, synchronization problems when using multiple processors, costs of communication between multiple tasks, overhead of libraries, parallel compiler and supportive OS and the complexity associated with the design, the code the debugging and the maintenance of the program.

REFERENCES

[1] Chapman & Hall/CRC, *Introduction to High Performance Computing for Scientists and Engineers*
[2] Grinnell College, *Lab 11: Hardware Performance Counters*
[3] Berkeley, *Lab 8: SSE Instructions and Loop unrolling*
[4] Intel, *Intel Intrinsics Guide*