Introduction to parallel computing, homework 1 redo

Anne-Marthe Hvammen Sikkerboel
245997
anne.sikkerboel@studenti.unitn.it
link to git repo: https://github.com/AnneMartheH/H1AmIPC.git

9. januar 2024

Array Multiplication

Solution implemented task1 array addition
In the main function array A,B and C are allocated statically and then initialized. Array A and B are initialized with random values between 0 and 1000, and array C is initialized with zeroes. The parameter n is decided by the user, controls the size of the arrays. These four parameters A,B,C and n is passed to the functions routine1, and routine2.

Routine1's task is to add array A with array B and place the result in array C. This is done through a for loop iterating through the elements of A and B and then adding them to C. In order to get more accurate timing this four loop is running ten times. When the time per loop is calculated the total time of 10 runs is divided on 10, to get the average run-time.

In routine 2 you have almost the same code as in routine1. The only difference is that two *__builtin_prefetch()* function is added. This is in order to prefetch the next values from array A and B, so that they are available in the cash. This prefetching also prefetches the next value when er are at the last value in the array. This is unnecessary and could have been fixed by an if statement.

Results
In this section the results of some of the tests is displayed. All tests has been run on an interactive session in the cluster.

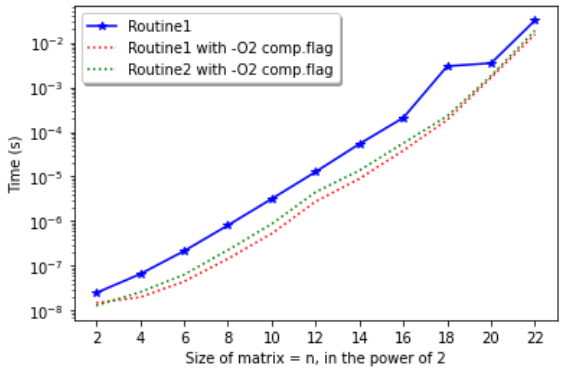
flags	**O0**	**O2**	**O3**	**Ofast**	**without**
wall clock time (s)	0,0225	0,00628	0,0119	0,0108	0,0177

Tabell 1: Average time used by routine1 in file task1.c with different flags. The average is calculated from four runs.

First, by observing table 1, one see that running the code with compilation flag -O0

n	routine 1	routine1/w -O2	routine 2 /w -O2
4	0.000000025	0.000000015	0.000000013
16	0.000000066	0.000000020	0.000000026
64	0.000000216	0.000000045	0.000000064
256	0.000000812	0.000000145	0.000000227
1024	0.000003183	0.000000532	0.000000872
4096	0.000012774	0.000002764	0.000004498
16384	0.000053991	0.000009001	0.000013733
65536	0.000207321	0.000038386	0.000056093
262144	0.002958778	0.000185751	0.000226347
1048576	0.003445900	0.001645652	0.001804951
4194304	0.031150598	0.015308349	0.018456620

Figur 1: Wall clock time used by the different routines in file task1.c, with n as the size of the matrix. Displayed in a table.



Figur 2: Wall clock time used by the different routines in file task1.c, with n as the size of the matrix. Displayed as a graph.

is the slowest. Except this compilation, all the other tested flags are faster than compiling without any additional flags. The fastest flag is -O2, which activates nearly all optimizations that does not involve space-speed trade off.

Second, looking at the plots from figure 2, one can see that the fastest runtime is for task1 with -O2 compilation flag. Then routine2 with -O2 flag is a bit slower. Both of these is quite a bit faster than routine1 without any compilation flags.

Discussion & Conclusion

From looking at the results above it seems that prefetching the next values from matrix A and B does not improve the run time. This can be because the next values are already in the cash. If this is the case prefetching these values again is unnecessary time. It would be reasonable that the next values already would be in the cash since the values are read from the arrays in an order, following the memory layout in C.

For the code in task1.c the fastest runtime is achived by running routine1 with -O2 compilation flag. This compilation flag focuses on optimizing without compromising the space-speed relationship.

Matrix copy via block reverse ordering

Solution implemented

This task consists of copying a matrix M into another matrix O block-wise with reversed order of the blocks. In routine1 this problem is solved by four nested for-loops. The two outer for-loops make sure the matrix M is sectioned into smaller blocks. Then the two inner for-loops iterates through these blocks of matrix M. In the two inner for-loops the variables k and l is incremented, in order to access the right position in the sub-matrices of O. Hence, the main work of the code is executed by these four for-loops, and this is also the timed section of the code.

Routine2 is identical to routine1, with the exception that the next values from matrix M is being prefetched. The prefetching is not optimal since it does not take consideration to the edge cases, like when one is at the last value of one sub matrix.

Results

compflag test n = 2^12, b = 2^8		
	routine1	routine2
O0	0,0841034	0,0800749
O2	0,0251803	0,0190895
O3	0,061856	0,0394995
Ofast	0,0272813	0,0206838
without	0,0845558	0,0804653

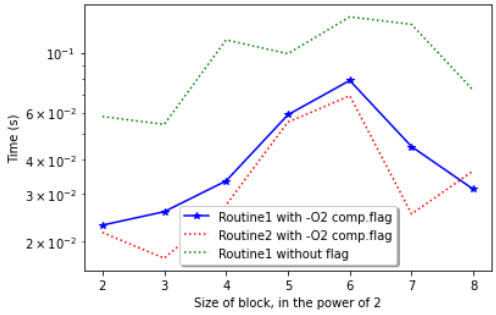
Figur 3: Compilation flag test with different flags on routine1 and routine2 from file task2.c. With the matrix size $n = 2^{12}$ and $b = 2^8$.

wallclock time test				
b = 2^	b =	routine 1	routine1 W/O2	routine2 w/O2
2	4	0.058019658	0,022914482	0,0215274
3	8	0.054259368	0,025716896	0,0172629
4	16	0.111762644	0,033447672	0,0272247
5	32	0.099280140	0,058896322	0,0553252
6	64	0.136105744	0,078982478	0,0692895
7	128	0.127558420	0,044856242	0,0252127
8	256	0.072537580	0,031134472	0,0364911

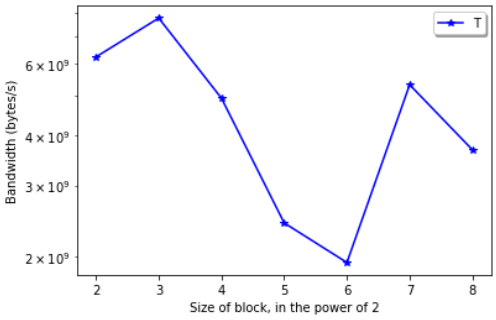
Figur 4: Wallclock time on different routines from file task2.c. With different b = blok sizes and $n = 2^{12}$. result displayed in a table.

As in task1.c in task2.c the compilation flag -O2 is the fastest, and compilation with O0 is the slowest.

By looking at the plots from figure 5 and figure 4. One can observe that the fastest runtime is for task2 with -O2 compilation flag. Then routine1 with -O2 flag is a bit slower. Both of these is quite a bit faster than routine1 without any compilation flags.



Figur 5: Wallclock time on different routines from file task2.c. With different b = blok sizes. result displayed as a graph.



Figur 6: Bandwidth for routine2 with compilation flag O2. blockSizes is given in the power of 2. From file task2.c

The b = blockSize which make the code fastest is dependent on the toutine and how it is compiled. For routine2 with O2 compilation flag and routine1 without compilation flags the code is fastest with $b = 2^3$. However for routine1 with O2 compilation flag it is fastest when $b = 2^2$. For all the routines independent on the compilation $b = 2^6$ is the blockSize that makes the routines slowest. However the runtimes is quite good again for $b = 2^7 \& b = 2^8$.

From figure 6, you can see that the bandwidth for routine2 with O2 compilation flag has the highest value at $blockSize = 2^3$. The lowest bandwidth is at $blockSize = 2^6$.

Discussion & Conclusion
From looking at the results above it seems that prefetching the next values from matrix A and B does improve the run time. Since this task iterates through the matrix M blockwise, it may be difficult for the compiler to understand which value comes next. Hence, by prefetching the next into values into the cash, we are preventing cash misses.

The blockSize affects the runtime a lot. It seems that either big ($n > 2^6$) or small($n < 2^4$) block sizes is preferable.

Compiling the code with the O2 flag improves the run time, both for routines with and without prefetching.

Bibliography

here are some of the links that have been used as inspiration or to get facts to this project:

https://caiorss.github.io/C-Cpp-Notes/compiler-flags-options.html

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

https://www.daemon-systems.org/man/__builtin_prefetch.3.html