Programming of Supercomputers

Assignment 1

Team 12:
Mahyar Valizadeh
Ajay Kunuthur

Introduction

The objective of this assignment is analyzing sequential optimization and it's done by applying different compiler flags and measuring performance using PAPI library and getting to know the running environment which we will use throughout the course.

As it is asked we use given source code of an GCCG solver, for further details check assignment's presentation and tasks declaration.

Getting to know the environment

As the first task it was required to know the running environment of our application which is the supercomputer located at Leibniz-Rechenzentrum, famously called as "SuperMUC". Using Secure Shell (ssh) from trusted network, the environment can be accessed but how can we check the specification and characteristics of that is using "$ lscpu" command for getting the number of processor on a node and frequency of a core and number of core in a CPU and the memory and cache level sizes. Another command "$ less /proc/cpuinfo" is used to get the peak frequency and check whether hyper threading is enabled and last command is "$ grep -E '^model name|^cpu MHz' /proc/cpuinfo" to check whether frequency scaling is also enabled by checking current CPUs clock speed. The result of these commands are provided in attached Data sheet in data folder.
Another section in this part is getting to know the library/interface which can be used to measure performance of the application in question.
Therefore, the basic usage of papi libraries has been achieved. These libraries allow the access to the hardware counters of the system in a runtime environment.
On Supermuc, we understand that the job must be submitted to a queue, where it allocates the job to the complete node, when the node is free.
We have accessed the hardware info of the cpu but the results in the interactive job mode is different from the job submit mode.

Cpu info	Interactive job	Job cmd file (llsubmit)
No. of sockets in a node	2	4
No. of cores in a socket	8	10
Multithreads in a core	2	2
Cpu's in a socket	16	20
Total cpu's in a node	32	80

In the above table, the sockets in the nodes were given as zero, but from the other values, we have identified them to the values above. So, the load leveler has allocated extra processing units for the job.

Calculation of the metrics

As mentioned above using PAPI we can access some hardware counters and using these counters some important aspect of performance measuring can be achieved. The following counters have been accessed.

PAPI_L2_TCM – The total level 2 cache misses.
PAPI_L2_TCA – The total level 2 cache accesses.
PAPI_L3_TCM – The total level 3 cache misses.
PAPI_L3_TCA – The total level 3 cache accesses.

These counters have been estimated for each of the phases described in the source code. Hence, the cache miss rate

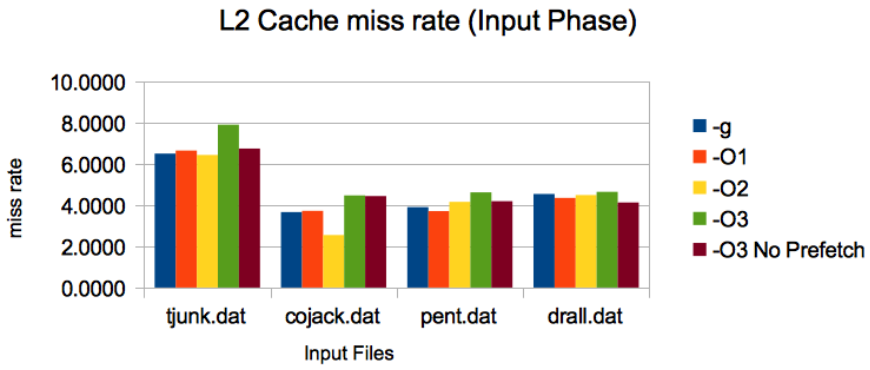
L2_cache_miss_rate = (PAPI_L2_TCM/PAPI_L2_TCA) * 100;
L3_cache_miss_rate = (PAPI_L3_TCM/PAPI_L3_TCA) * 100;

Through papi_timers, the time of execution of each phase has been calculated using the calling routine PAPI_get_real_usec(). The execution time has been calculated in micro seconds. And with another counter “PAPI_FP_OPS” the total floating point operations have been counted.
Hence, mflops = Total floating point operations / Ececution time in microseconds.

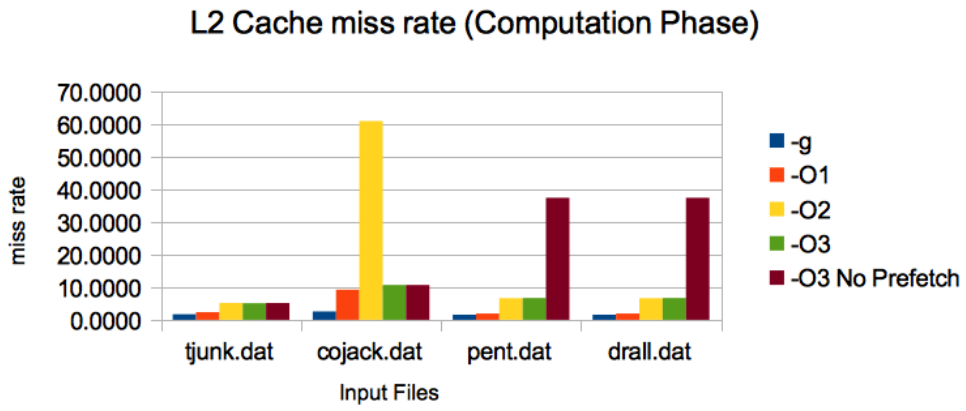
Comparison with ideal performance of the Supermuc

The peak performance of the SuperMUC is given by around 3 peta flops however the peak performance of a core is 21.6 GFlops. Maximum performance achieved in our case, is given by 3533 MFlops which compared to one core is around 16%. Based on the data provided in the Data sheet the average performance of the code was around 1100 MFlops which is really small.
The max performance can never be achieved, since there are always latencies in the system. However through a perfect optimization and good data distribution, ideally a good 60-70 % of the peak performance is desired.

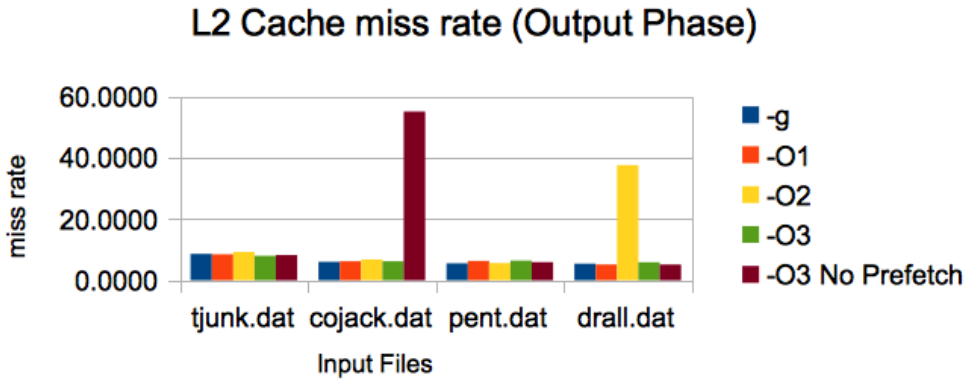
Discussion on the measured data



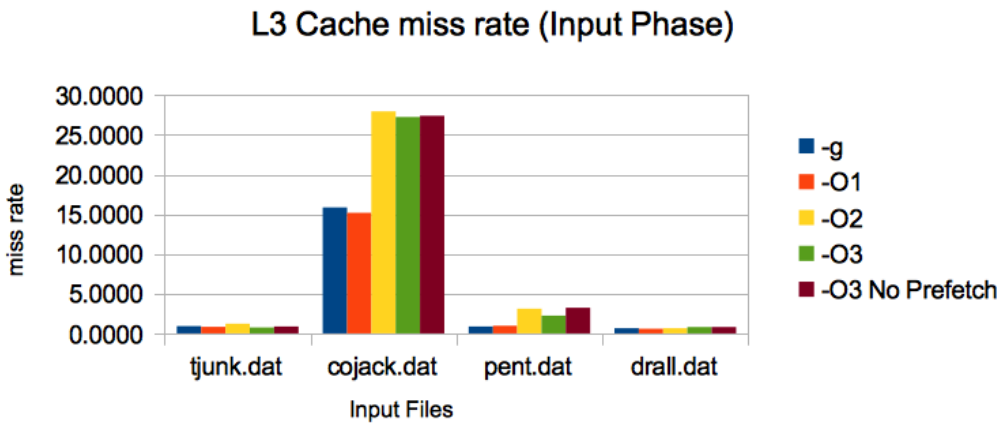
Here you can see the L2 cache miss rate for different scenarios are sketched and the level of optimization is compared. For input or output section optimizing is not changing the cache miss rate



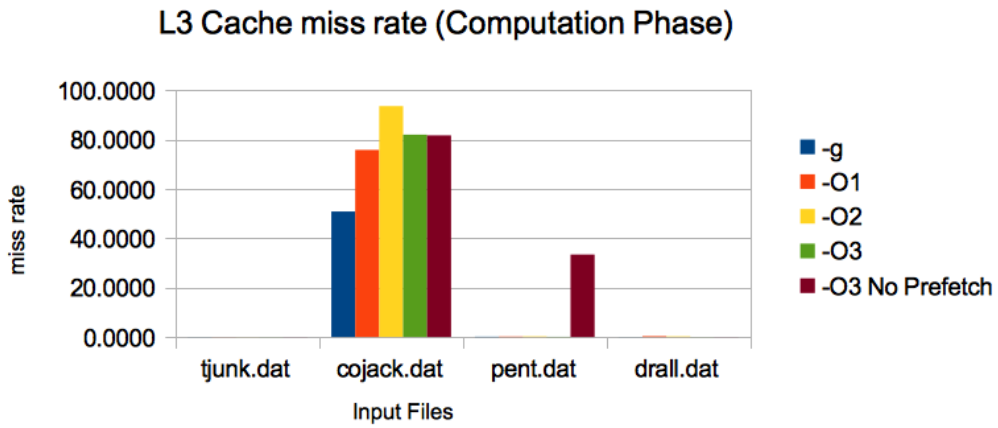
significantly. However, for computation phase ,as you can see, without prefetching the cache misses are much higher in computation phase due to availability of data. There is also probably one case which is not agreeing on usual trend which may be the problem with reliability of working

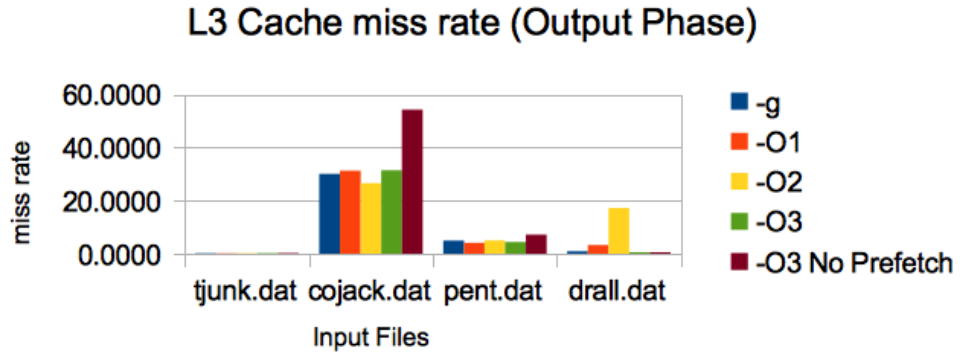


environment. Therefore basically it may happened that the machine doesn't work properly or something would happen during execution which may be an error under run in superMUC.
Moreover the trend shows that the optimization makes the cache misses for L2 higher in computation phase but it may uses L3 more so that in general optimized code would run faster and for that we need to check the execution time which will be discussed later on.

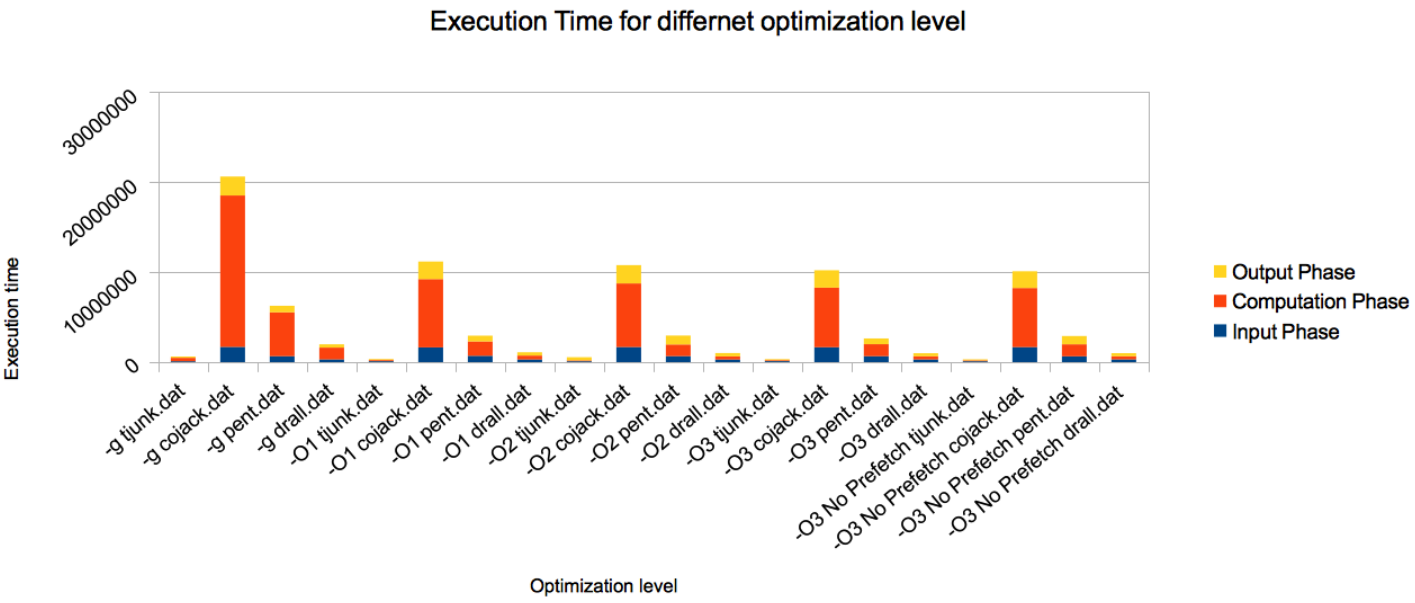


Again for L3 cache miss rate it can be seen some optimization will increase cache miss rate but it is possible it can give us faster result anyway.





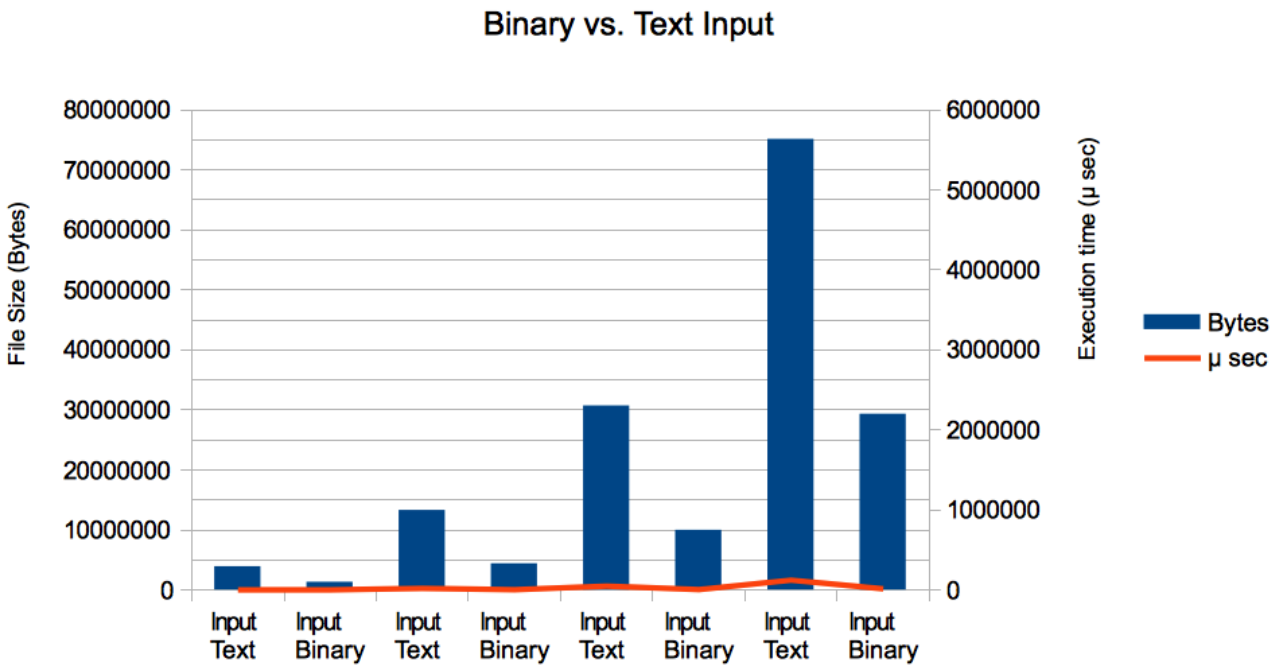
The main and more obvious thing here is for measuring performance is by checking the execution times. Here is the result for different level of parallelism.
It can be observed from this graph even with losing performance due to higher cache misses still higher optimization level will give us faster and less execution time. Another thing is even for different execution stages we have gained performance due to optimization.



It can observed from the Data sheet provided, that for most of the cases more percentage of the work is in calculation part and thus execution of this part takes longer than input or output and here cache miss is really important. In conclusion, cache miss here is then plays the important role in total execution time but fortunately by exploiting it we gain higher performance.
Another subject from graph above is that execution time with respect to level of parallelism would be less with more optimized level.

On the comparison of Text vs binary input files

Generally speaking both the binary and ASCII (text) files are both the same. But the ASCII file stores a byte in terms of an ASCII code, which is actually read as a 1's and 0's by the processor. But the binary file can give you the direct access to the number at the file stream, in terms of number of bytes of data you are expecting the value to store. Hence, ASCII has some kind of inherent little overhead, but both files are regularly used.



Performance Estimation – As expected, reading a binary files which has direct access to the value in terms of bytes so much faster than the ASCII text files. In the data sheet, we can see that the size of the binary file is (1/3) of the size of the text files. Sometimes there is a wastage of space in ASCII files vowing to the fact that it requires 7 bits to represent each character and on overall consuming more space on the system. As you can see from the picture above the execution time and size of binary files is less than the text ones.