REPORT – Assignment 01

The main aim of Assignment 01 is to improve the Single-Core performance of any application. This can be achieved by a few methods, of which the assignment dealt with sequential optimization using compiler flags, improving I/O performance by generating binary files from text. The use of PAPI-Performance Application Programming Interface was learnt and implemented to measure metrics for the computation phase of our GCCG solver. The execution was conducted on the SuperMUC thin node and the results were visualized using Paraview.

1. Executing Environment

SuperMUC is the executing environment for our present assignment. SuperMUC is built of 18 islands with 512 nodes on each island. A single node is a shared memory system with 2 processors and 8 cores on each processor. The total computing power per processor is 172.8 GFlops.

Every node is a NUMA node with 2 QPI links. The Interconnection is achieved by Infinibands. The SuperMUC is also connected with Ethernet. The SuperMUC also uses a warm water cooling system to improve efficiency. There is also Autotuning to allow the tuning of applications for energy efficiency.

As mentioned, the Thin Nodes consist of 2 sockets and 8 cores per socket. We used the thin nodes for the purpose of executing the application.

Parameter (Thin Nodes)	Value
Number of Processors per Node	2
Number of Cores per Processor	8
Hyperthreading	Two-way Hyperthreading
Threads per Core	2
Number of Threads per Node	32
Peak Frequency per Core	3.5Ghz - Max Turbo Freq
Peak Performance (Mflops)per Core	21600 Mflops
Peak Performance (Mflops) per Node	345600 Mflops
Frequency Scaling	1.2 Ghz - 3.5 Ghz
L3 Cache per Processor (shared)	20 MB
L2 Cache per Core	256 KB
L1 Cache per Core	32 KB
Compiler	ICC
MPI Library	MPICC Intel - icc 14.0.3 20140422

Table 1. SuperMUC Properties

2. Ex 1.2 & Ex 1.3 - Performance Measurement using PAPI

This task involved the measurement of metrics for the computation phases of the GCCG solver for two input files, tjunc.dat and cojack.dat.

The performance was then compared for the –O1 (no vectorization) and –O3 optimization levels for both files. Later, only file cojack.dat was used for the measurement of the vectorization flags *–no-vec, -vec-report, -xhost.*

Properties:

-no-vec disables vectorization.
-vec-report generates a report for the vectorization of the compiled files.
-xhost – if the application is run on the processor type on which it was built, we use this flag. It enables aggressive vectorization on host processor.

For the performance measurements in our code we use PAPI. Our program contains two main gccg files.

In the file 'gccg.c' we measure **Mflops** and **execution time** with high-level API.
Mflops is number of floating point operations per second divided by one million.

One way of computing it with PAPI is to use low-level API and simply divide value in PAPI_FP_OPS counter with time passed (in microseconds).

However, in our case it is obtained directly with the High Level PAPI function call PAPI_flops(...).

In the file "gccg_cache.c" we measure **L2** and **L3 cache miss rates** with low-level API.

We obtain total L2 and L3 cache miss rates by computing the ratio between the total number of cache misses and total number of cache accesses. $\frac{Total\ Cache\ Misses}{Total\ Cache\ Accesses}$

For the respective measurements we use low-level PAPI.

TJUNC.DAT				
AVERAGE VALUES				
	Execution Time	**Mflops**	**L2 Cache Miss Rate %**	**L3 Cache Miss Rate %**
Flags				
-01	0.088103	1793.95	22.36287	0.001333
-03	0.077003	2020.784	38.21057	0.001

Table 2. Metrics for *tjunc.dat*
-O1 & -O3 Optimization Levels

COJACK.DAT				
AVERAGE VALUES				
	Execution Time	**Mflops**	**L2 Cache Miss Rate %**	**L3 Cache Miss Rate %**
Flags				
-01	4.54101	1550.679	39.2804	8.414367
-03	4.38346	1575.061	50.84217	10.67153

Table 3. Metrics for *cojack.dat*
-O1 & -O3 Optimization Levels

COJACK.DAT				
AVERAGE VALUES				
	Execution Time	Mflops	L2 Cache Miss Rate %	L3 Cache Miss Rate %
Flags				
-no-vec	13.9402	501.4176	39.36517	8.618867
-vec-report	4.54246	1553.072	50.8862	10.67897
-xhost	4.511665	1535.3701	51.95367	16.2857

Table 4. Metrics for *cojack.dat*
Various Optimization Flags

Observations & Conclusions

As we clearly see from Table 1 and Table 2, the use of –O3 optimization flag improves the execution time of the application. Both the files, tjunc.dat and cojack.dat have an improved performance when using –O3 flag as compared to the –O1 flag. The –O3 flag enables a more aggressive vectorization and this in turn causes more optimizations like unrolled loops and improves the execution time.

MFlops

During our first calculation of the flops, we calculated the flops using High Level PAPI, this resulted in the flops decreasing with the higher optimization level. This was contrary to the general idea that the flops should increase for higher optimization levels. The use of low-level PAPI with PAPI_DP_OPS counter lead us to corrected values. These values showed an increase in flops with higher optimization level.

The –xhost wasn't able to achieve the highest flops among all the results. We can attribute this cause to the excessive vectorization in the compilation which lead to decrease in efficiency.

In all our executions, the maximum performance achieved was with –O3 flag about 2 GFlops. In relation to the peak performance of a single node that provides 21.6GFlops, the performance achieved is only 10% of peak performance.

Cache Miss Rate

For the two files, the cache miss rate increased with higher level of optimization. This is due to the vectorization techniques applied to the code. The temporal and spatial locality of the cache is affected due to vectorization. And this results in the increasing cache misses.

The file tjunc.dat can completely fit into the L3 cache of our environment. Thus, there is little or no effect of vectorization. However, the file cojack.dat was much bigger than the L3 cache and thus, an increased cache miss rate was observed.

3. Ex 2.3 Binary & Text Files

A binary file is a computer file that stores data in binary form for computer storage and processing purposes. When a regular file is created, it is encoded with a certain standard. This standard affects the size and performance of the file. The conversion of such files into computer readable format leads to extra storage and performance time. A code was written to convert the .dat file into a .bin file. This file can be easily read by the processor as it does not have to translate the binary form. Hence, the execution time and performance of this type of file is always better than the files other formats.

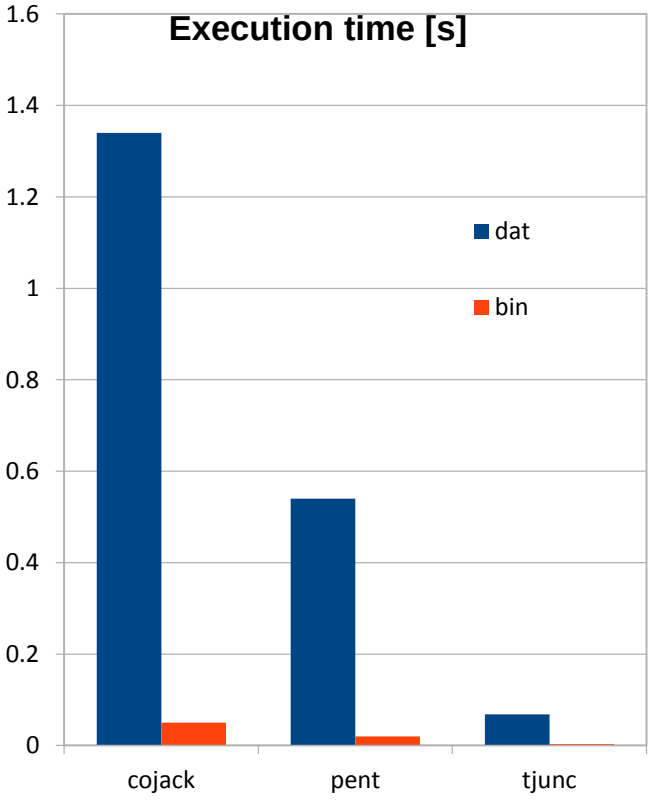
A comparative graph has been plotted to visualize the difference in reading time and file sizes. We made use of the PAPI module to calculate the reading time for different files. A code snippet has been shown below.

```
if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT) {
printf("Error: PAPI_library_init!");
exit(1);
}
t1 = PAPI_get_real_usec(); // Starts calculating time

// read-in the input file
int f_status = read_formatted(file_in, format, &*nintci, &*nintcf, &*nextci, &*nextcf, &*lcc, &*bs,
&*be, &*bn, &*bw, &*bl, &*bh, &*bp, &*su);

t2 = PAPI_get_real_usec(); // Stops time count
```

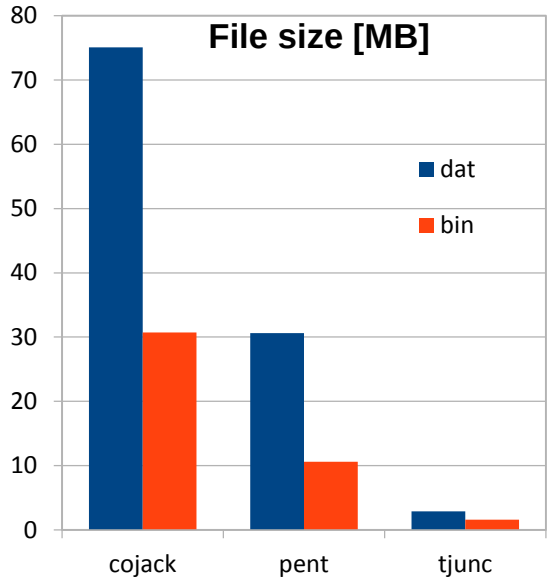
Fig 1. Code snippet of PAPI for reading time



Time[s]	dat	bin	ratio
cojack	1.34	0.05	26.8
pent	0.54	0.02	27.0
tjunc	0.068	0.003	22.7

Fig 2. & Table 5. – Execution time for *Text vs Binary* files.

The reading time for the two files is drastically different. The *.dat* takes nearly 25 times the time taken for reading a *.bin* file. This shows the superiority of the binary file and its contribution to optimization.



Size[MB]	dat	bin	ratio
cojack	75.1	30.7	2.4
pent	30.6	10.6	2.9
tjunc	2.9	1.6	1.8

Fig 3. & Table 6. – File Size for *Text vs Binary* files.

The output file generated by our *text2bin()* code has reduced the file size to approx. 2 times the usual size. This is a significant drop for huge files involved in solution of large equations.

REMARKS:

A1/code/ -- contains the application with an additional file gccg_cache.c (used for the measurement of cache miss rates). The binary conversion can be done by a command

```
make binconv
./binconv    cojack.dat    binary_cojack
```

The entire application can be made by the commands

```
make
./gccg    text    tjunc.dat    text_tjunc
```

The output (.vtk files and summary.out) will be available in this folder.

A1/plots/ -- the solution of the input file pent.dat has been visualized. The .vtk outputs and the screenshot for the same are provided.

A1/data/ -- the SuperMUC-properties.xls performance sheet has been completed. The other file, 'Data.xlsx' includes the performance parameters of 3 runs as discussed in solution to Ex1.2 & Ex1.3.

A1/scripts/ -- includes the job file used for performing tasks on SuperMUC. The file extension has been modified to .txt to allow the attachment to be sent.