PTfS Project:
Modelling 2D steady-state heat equation

Gokhul Shriram Ravi, Onkar Marathe, Youyi Zhang

September 12, 2024

Contents

1 Problem Description

Given is a 2D heat dissipation problem on a rectangular plate of diemnsion 1x1m. The goal is to find the steady state temperature distribution inside the plate.

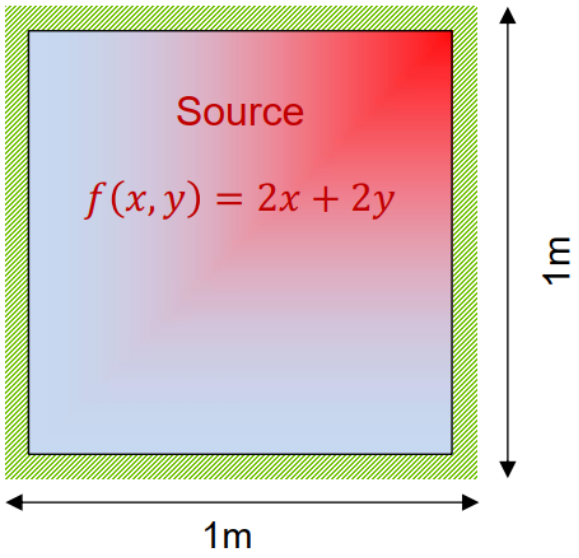


Figure 1.1: Heat dissipation on a rectangular plate

The governing equations are discretized and the linear system of equations thus obtained are converted into the form $Ax = b$ where, A is a banded matrix, u and b are column vectors. For solving this, we use a Conjugate Gradient and a Preconditioned Conjugate Gradient (PCG) with symmetric Gauss-Seidel preconditioning.

2 Hardware and Compiler Configuration and Specification

The configuration and hardware specifications used for running the codes and making the performance measurement are documented below.

- **Processor and cache specification**
  CPU name: Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz
  CPU type: Intel Icelake SP processor
  CPU stepping: 6
  Clock frequency: 2 GHz
  Sockets: 2
  Cores per socket: 36
  Threads per core: 1
  L1 cache size: 48 kB
  L2 cache size: 1.25 MB
  L3 cache size: 54 MB
  NUMA domains: 4
  Memory Bandwidth (for Roofline Prediction: 82GB/s)

- **Compilation directives for the tasks**
  - **Roofline Prediction - Experimental values**

```
LIKWID=on CXX=icpx make
salloc --nodes=1 --time=01:00:00
likwid-pin -q -c N:0-17 srun
    --cpu-freq=2000000-2000000:performance ./perf 2000
    20000
```

  - **Code Balance Measurement**

```
LIKWID=on CXX=icpx make --constraint=hwperf
salloc --nodes=1 --time=01:00:00
```

```
srun --cpu-freq=2000000-2000000:performance
    likwid-perfctr -c S0:0-17 -g MEM_DP -m ./perf 2000
    20000
```

– **Scalability of code on 72 threads**

```
LIKWID=on CXX=icpx make
salloc --nodes=1 --time=01:00:00
likwid-pin -q -c N:0-71 srun
    --cpu-freq=2000000-2000000:performance ./perf 2000
    20000
```

All the relevant job files, python scripts, etc have been attached separately to reproduce the results.

3 Tasks and Discussion

1 Loops fusions and possible optimisations

Firstly, all the .cpp files have been looked for potential places where loop fusion would be possible. In **Grid.cpp**, it is not possible to include loop fusion, without making any major changes to the code in itself. In the case of **PDE.cpp**, however, there was a potential for loop fusion; specifically in ***PDE::PDE***, where the Dirichlet boundary conditions are set. In **Solver.cpp** we have only while loops, so parallelization is also not possible. Therefore, it is only in **Grid.cpp** and **PDE.cpp**, that we have the potential to parallelize and increase performance.

It must also be noted however, that all functions in these files do not need to be parallelized. There are specific functions that have loops which have been parallelized, especially **applyStencil()**, **applyGSPreCon()**, **axpby()**, **dotproduct()** and **copy()** from these 2 files. On top of this, the constructors in **Grid.cpp** have to be parallelized.

The src files were also checked for "redundant" computations, and one such instance is found in **Solver.cpp**. There is a repeated calculation of tol*tol in the while loops. However, there doesn't seem to be a visible change in the performance of the code on removing this "redundant"/repeated calculation.

The code is run before any parallelization and optimisation. First, the code is built by typing **make** and then the **./perf** is run for grid size 2000x20000. The performance of the code is around 150.6 MLUP/s for CG and 53 MLUP/s for PCG.

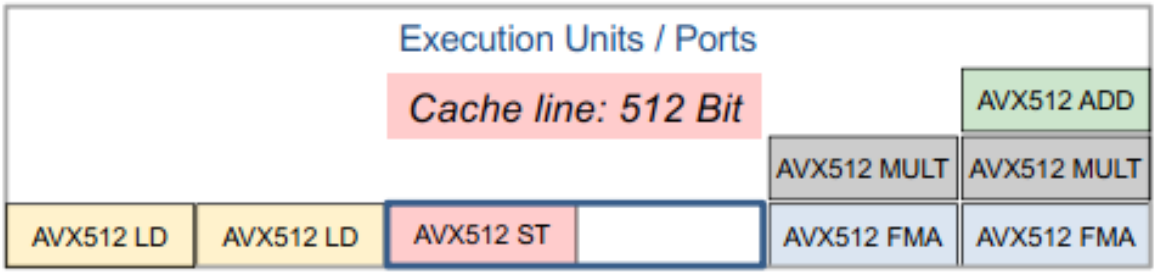
2 Roofline Predictions for CG and PCG in 1 ccNUMA domain

First, it is required to enumerate all the functions called by **CG** and **PCG** respectively.

CG
applystencil(v,p)
dotproduct(v,p)
axpby(x,x,p)
axpby(r,r,v)
dotproduct(r,r)
axpby(p,r,p)

Operation
applyStencil()
dotProduct()
axpby()
axpby()
dotProduct()
GSPreCon()
dotProduct()
axpby()

Next P_{max}^{core} for these functions have to be found out. The machine model for the **Intel(R) Xeon(R) Platinum 8360Y CPU** looks like this for AVX512 code;



with a maximum of 5 inst/cy.

- **applyStencil()**

  In this function, we have 1 LOAD, 1 STORE, 3 MULT and 4 ADD.

LD	ST	MULT	ADD	
		ADD	ADD	MULT
		FMA	ADD	

  1 AVX iteration = 3 cycles

  $\Rightarrow$ 1 AVX iteration $= 7 * 8/3 = 56/3$ $(FLOPs/cycle)$

- **dotProduct()**

  In this function, we have 2 LOAD, 1 MULT and 1 ADD.

LD	LD	ADD	MULT	

  2 AVX iteraion = 2 cycles

  $\Rightarrow$ 1 AVX iteration $= 2 * 8/2 = 8(FLOPs/cycle)$

LD	LD	MULT	MULT	
ADD	ST	LD	LD	MULT
MULT	ADD	ST		

– **axpby()**
In this function, we have 2 LOAD, 1 STORE, 2 MULT and 1 ADD.

2 AVX iteraion = 3 cycles
$\Rightarrow$ 1 AVX iteraion $= 3 * 8/3 = 8(FLOPs/cycle)$

– **GSPreCon() - forward**
In this function, we have 2 LOAD, 1 STORE, 3 MULT and 2 ADD.

LD	LD	MULT	MULT	
MULT	ADD			
ADD	STORE	LD	LD	MULT
MULT	ADD			
MULT	ADD	ST		

2 AVX iteraion = 5 cycles
$\Rightarrow$ 1 AVX iteraion $= 5 * 8/5 = 8(FLOPs/cycle)$

– **GSPreCon() - backward**
In this function, we have 2 LOAD, 1 STORE, 3 MULT and 2 ADD.

LD	LD	MULT	MULT	
MULT	ADD			
ADD	STORE	LD	LD	MULT
MULT	ADD			
MULT	ADD	ST		

2 AVX iteraion = 5 cycles
$\Rightarrow$ 1 AVX iteraion $= 5 * 8/5 = 8(FLOPs/cycle)$

The roofline is calculated as

$$P = min(P_{peak}, I * b_s)$$

where, P_{peak} is Peak performance, b_s is Peak memory bandwidth and I is Computational Intensity. P_{peak} and b_s are hardware characteristics while I is code dependent. Another consideration to keep in mind, is if the grid fits into the cache. This can be found out using the Layer condition, given by,

$$\text{numthreads} * 3 * \text{layersize} * 8B < \text{CacheSize}/2$$

For the case of 2000 * 20000, we have,

$$3 * 18 * 20000 * 8 < 27/2MB$$

$\Rightarrow$ Layer condition satisfied $\Rightarrow$ only one cache miss per iteration

For the case of 20000 * 2000, we have,

$$3 * 18 * 2000 * 8 < 27/2MB$$

$\Rightarrow$ Layer condition satisfied $\Rightarrow$ only one cache miss per iteration

But for the case of 1000 * 400000 however,

$$3 * 18 * 400000 * 8! < 27/2MB$$

$\Rightarrow$ Layer condition not satisfied $\Rightarrow$ only four cache misses per iteration

Based on the cache calculation, we observe that only two grid sizes adhere to the execution port figures defined earlier: 2000*20000 20000*2000 On the other hand, the 1000*400000 grid does not follow the same pattern due to cache misses.

From this, we have

CG - 2000x20000							
kernel	Perf [flops/cy]	P_max [flops/s]	Code Balance	Perf [flops/cy]	P [flops/cy]	Grid Size	Time (s)
applyStencil(v,p)	18.66666667	6.72E+11	16	3.588E+10	3.59E+10	4E+07	0.00780488
dotProduct(v,p)	8	2.88E+11	16	1.025E+10	1.03E+10	4E+07	0.00780488
axpby(x,x,p)	8	2.88E+11	24	1.025E+10	1.03E+10	4E+07	0.01170732
axpby(r,r,v)	8	2.88E+11	24	1.025E+10	1.03E+10	4E+07	0.01170732
dotProduct(r,r)	8	2.88E+11	8	2.05E+10	2.05E+10	4E+07	0.00390244
axpby(p,r,p)	8	2.88E+11	24	1.025E+10	1.03E+10	4E+07	0.01170732
							0.05463415

From this, we can say that the total performance for CG P_{total} is given by

$$P_{total} = \frac{4 * 10^7}{0.05463} = 7.32 * 10^8 LUP/s$$

PCG - 2000x20000							
kernel	Perf [flops/cy]	P_max [flops/s]	Code Balance	Perf [flops/cy]	P [flops/cy]	Grid Size	Time (s)
applyStencil(v,p)	18.66666667	6.72E+11	16	3.588E+10	3.59E+10	4E+07	0.00780488
dotProduct(v,p)	8	2.88E+11	16	1.025E+10	1.03E+10	4E+07	0.00780488
axpby(x,x,p)	8	2.88E+11	24	1.025E+10	1.03E+10	4E+07	0.01170732
axpby(r,r,v)	8	2.88E+11	24	1.025E+10	1.03E+10	4E+07	0.01170732
dotProduct(r,r)	8	2.88E+11	8	2.05E+10	2.05E+10	4E+07	0.00390244
axpby(p,r,p)	8	2.88E+11	24	1.025E+10	1.03E+10	4E+07	0.01170732
dotProduct(v,z)	8	2.88E+11	16	1.025E+10	1.03E+10	4E+07	0.00780488
GSPrecon(Forward)	8	2.88E+11	24	1.708E+10	1.71E+10	4E+07	0.01170732
GSPrecon(Backward)	8	2.88E+11	16	2.563E+10	2.56E+10	4E+07	0.00780488
							0.08195122

From this, we can say that the total performance for PCG P_{total} is given by

$$P_{total} = \frac{4 * 10^7}{0.081951} = 4.88 * 10^8 LUP/s$$

CG - 20000x2000							
kernel	Perf [flops/cy]	P_max [flops/s]	Code Balance	Perf [flops/cy]	P [flops/cy]	Grid Size	Time (s)
applyStencil(v,p)	18.66666667	6.72E+11	16	3.59E+10	3.59E+10	4E+07	0.00780488
dotProduct(v,p)	8	2.88E+11	16	1.03E+10	1.03E+10	4E+07	0.00780488
axpby(x,x,p)	8	2.88E+11	24	1.03E+10	1.03E+10	4E+07	0.01170732
axpby(r,r,v)	8	2.88E+11	24	1.03E+10	1.03E+10	4E+07	0.01170732
dotProduct(r,r)	8	2.88E+11	8	2.05E+10	2.05E+10	4E+07	0.00390244
axpby(p,r,p)	8	2.88E+11	24	1.03E+10	1.03E+10	4E+07	0.01170732
							0.05463415

From this, we can say that the total performance for CG P_{total} is given by

$$P_{total} = \frac{4 * 10^7}{0.05463} = 7.32 * 10^8 LUP/s$$

PCG - 20000x2000							
kernel	Perf [flops/cy]	P_max [flops/s]	Code Balance	Perf [flops/cy]	P [flops/cy]	Grid Size	Time (s)
applyStencil(v,p)	18.66666667	6.72E+11	16	3.588E+10	3.59E+10	4E+07	0.00780488
dotProduct(v,p)	8	2.88E+11	16	1.025E+10	1.03E+10	4E+07	0.00780488
axpby(x,x,p)	8	2.88E+11	24	1.025E+10	1.03E+10	4E+07	0.01170732
axpby(r,r,v)	8	2.88E+11	24	1.025E+10	1.03E+10	4E+07	0.01170732
dotProduct(r,r)	8	2.88E+11	8	2.05E+10	2.05E+10	4E+07	0.00390244
axpby(p,r,p)	8	2.88E+11	24	1.025E+10	1.03E+10	4E+07	0.01170732
dotProduct(v,z)	8	2.88E+11	16	1.025E+10	1.03E+10	4E+07	0.00780488
GSPrecon(Forward)	8	2.88E+11	24	1.708E+10	1.71E+10	4E+07	0.01170732
GSPrecon(Backward)	8	2.88E+11	16	2.563E+10	2.56E+10	4E+07	0.00780488
							0.08195122

From this, we can say that the total performance for PCG P_{total} is given by

$$P_{total} = \frac{4 * 10^7}{0.081951} = 4.88 * 10^8 LUP/s$$

CG - 1000x400000							
kernel	Perf [flops/cy]	P_max [flops/s]	Code Balance	Perf [flops/cy]	P [flops/cy]	Grid Size	Time (s)
applyStencil(v,p)	18.66666667	6.72E+11	32	1.794E+10	1.79E+10	4E+08	0.15609756
dotProduct(v,p)	8	2.88E+11	16	1.025E+10	1.03E+10	4E+08	0.07804878
axpby(x,x,p)	8	2.88E+11	24	1.025E+10	1.03E+10	4E+08	0.11707317
axpby(r,r,v)	8	2.88E+11	24	1.025E+10	1.03E+10	4E+08	0.11707317
dotProduct(r,r)	8	2.88E+11	8	2.05E+10	2.05E+10	4E+08	0.03902439
axpby(p,r,p)	8	2.88E+11	24	1.025E+10	1.03E+10	4E+08	0.11707317
							0.62439024

From this, we can say that the total performance for CG P_{total} is given by

$$P_{total} = \frac{4 * 10^8}{0.62439} = 6.41 * 10^8 LUP/s$$

PCG - 1000x400000							
kernel	Perf [flops/cy]	P_max [flops/s]	Code Balance	Perf [flops/cy]	P [flops/cy]	Grid Size	Time (s)
applyStencil(v,p)	18.66666667	6.72E+11	32	1.794E+10	1.79E+10	4E+08	0.15609756
dotProduct(v,p)	8	2.88E+11	16	1.025E+10	1.03E+10	4E+08	0.07804878
axpby(x,x,p)	8	2.88E+11	24	1.025E+10	1.03E+10	4E+08	0.11707317
axpby(r,r,v)	8	2.88E+11	24	1.025E+10	1.03E+10	4E+08	0.11707317
dotProduct(r,r)	8	2.88E+11	8	2.05E+10	2.05E+10	4E+08	0.03902439
axpby(p,r,p)	8	2.88E+11	24	1.025E+10	1.03E+10	4E+08	0.11707317
dotProduct(v,z)	8	2.88E+11	16	1.025E+10	1.03E+10	4E+08	0.07804878
GSPrecon(Forward)	8	2.88E+11	24	1.708E+10	1.71E+10	4E+08	0.11707317
GSPrecon(Backward)	8	2.88E+11	16	2.563E+10	2.56E+10	4E+08	0.07804878
							0.89756098

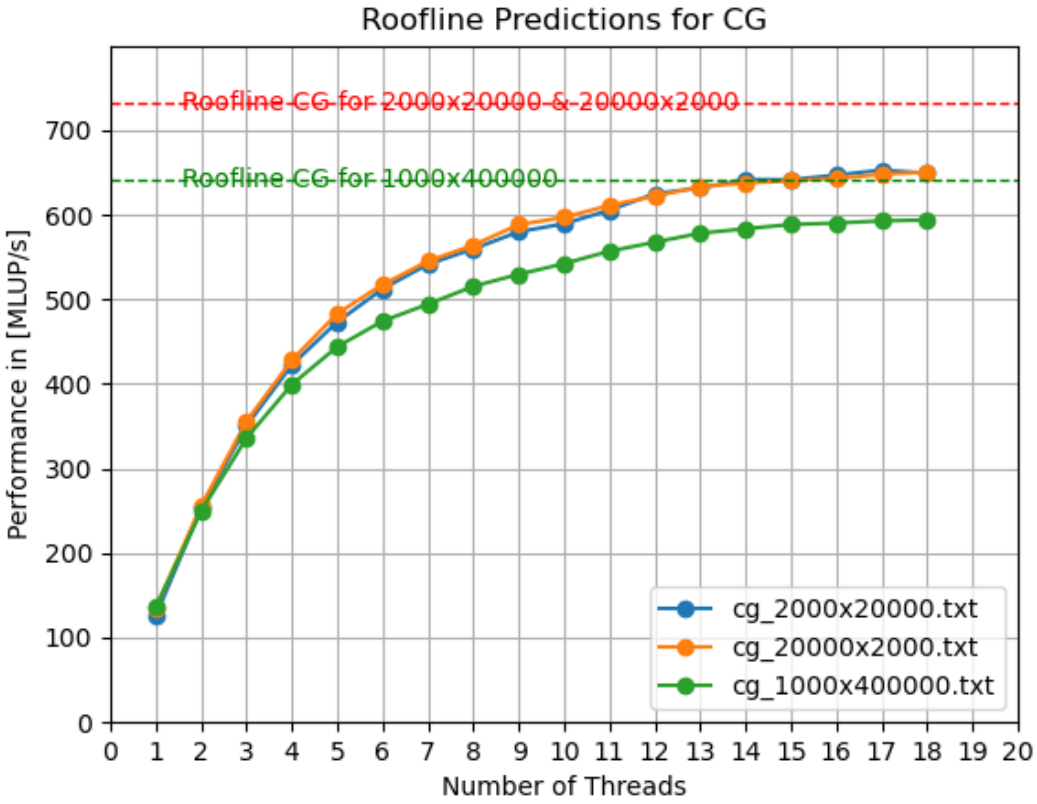
From this, we can say that the total performance for PCG P_{total} is given by

$$P_{total} = \frac{4 * 10^8}{0.89756} = 4.46 * 10^8 LUP/s$$

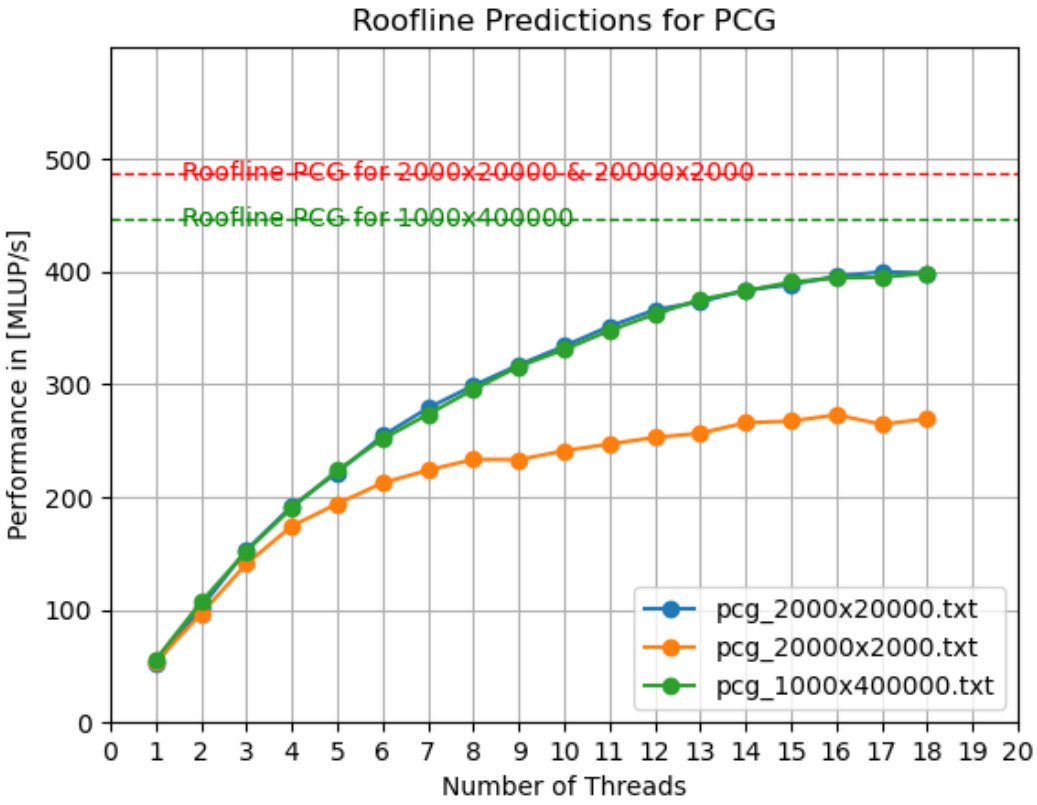
It is interesting to note that for all the functions considered here, the memory performance was lower than the compute performance.

3 Comparison of Theoretical Roofline Performance with actual Performance figures

The code is made and run on one ccNUMA domain (i.e. 18 threads) of Fritz. The following plots were made for the CG and PCG solvers.



The experimental value is very close to the theoretical roofline value, with no significant difference observed. This indicates that the experimental setup and implementation are consistent with theoretical expectations.



The performance comparison plot reveals that the actual performance closely aligns with the theorized roofline for grid sizes 2000x20000 and 1000x400000. However, a notable deviation from the predicted roofline is observed for the grid size 20000x2000. The wavefront parallelization strategy in GSPreCon() is identified as the primary cause of this performance anomaly. The mismatch between the larger outer loop and smaller inner loop significantly compromises the performance.

4 Code Balance of applyStencil() and GSPreCon()

Code Balance(B_c) is given by,

$$B_c = \frac{\text{Data Transfer}}{\text{Amount of Work}} = \frac{\text{MEM\_DP * (FLOPS/LUP)}}{\text{FLOPS\_DP}}$$

where, MEM_DP represents memory bandwidth in MB/s, FLOPS/LUP represents the number of floating point operations per LUP and FLOPS_DP represents the number of floating point operations in MFLOP/s. The code is made using appropriate flags to enable usage of likwid-perfctr. The code is then run on 1 ccNUMA domain. The performance values were noted down and they have been tabulated below:

		GSPreCon		ApplyStencil	
2000x20000	Mem	50789.6185	3.856530352	76375.9432	3.22511
	Flops	13169.7702		23681.656	
20000x2000	Mem	19838.2542	3.88287692	75396.6362	3.212869
	Flops	5109.1638		23467.0762	
1000x400000	Mem	62824.4162	3.995213394	78074.6038	4.957581
	Flops	15724.9213		15748.5276	

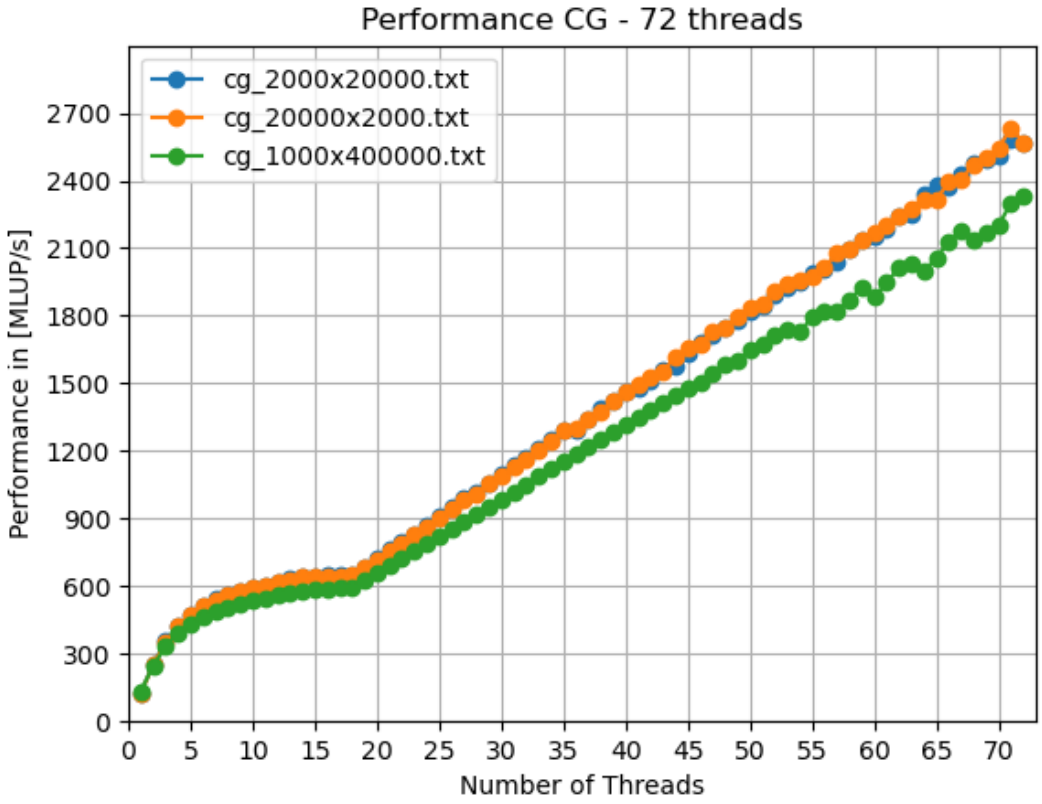
The highlighted values represent the values retrieved from likwid-perfctr. The units of measurement are consistent with the ones specified above. The formula for B_c is then applied and the final code balance metrics are calculated.

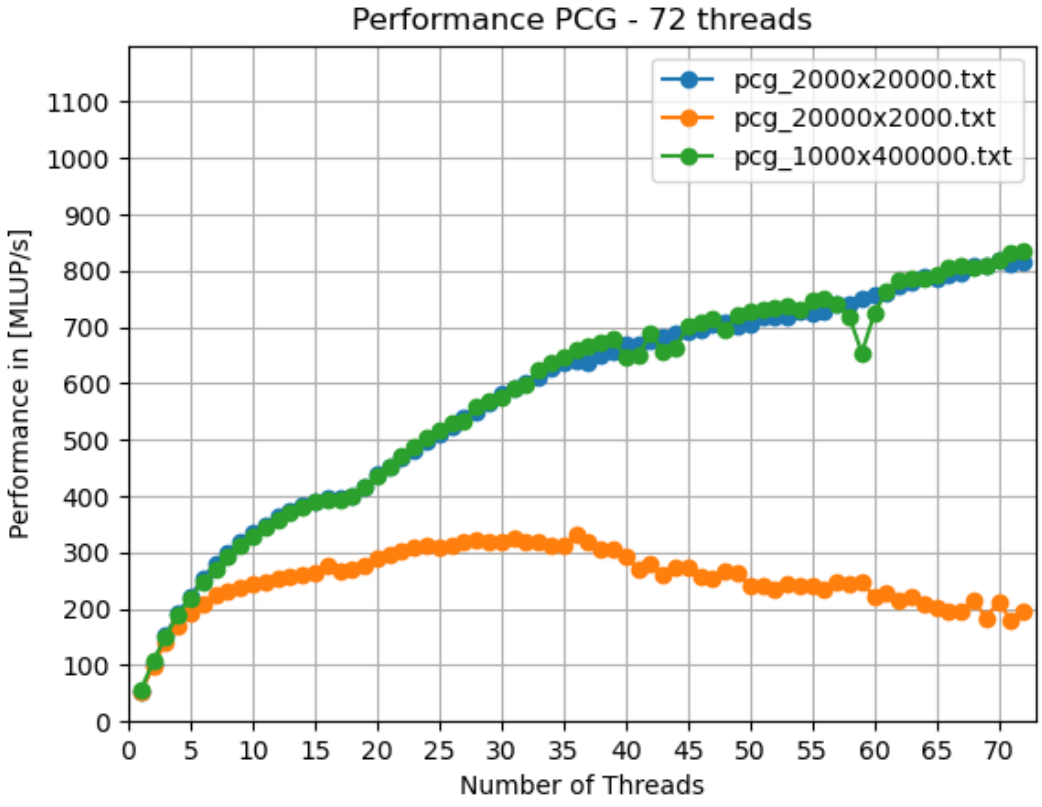
	2000x20000	20000x2000	1000x400000
GSPreCon	38.5653035	38.8287692	39.95213394
ApplyStencil	22.5757693	22.4900814	34.70306815

It can be observed that the value of B_c vary from the ones used in making the Roofline model. It must be kept in mind that the memory bandwidth was kept constant at 82 GB/s for 1 ccNUMA domain.

5 Scaling of the code on 72 threads

The code was made and run on all 72 threads of Fritz at 2GHz. The results of the same are plotted below:





We can see for CG that the performance is scaling perfectly. There is a saturation observed at about 18 threads (1ccNUMA domain) and from thereon out, it scales perfectly.

However, it is obvious by looking at the plots of the PCG solver that the performance is not scaling perfectly. A possible reason for this, particularly for the 20000x2000 grid size, is that performance saturates after using a certain number of threads. This likely stems from parallelizing the inner loop, which has fewer iterations compared to the outer loop, causing poor load balancing across threads. As a result, some threads complete their tasks much faster while others are still processing, leading to an unbalanced workload distribution and limiting further scalability, and in this case, a decrease in performance as well.

We notice a distinct performance plateau for the two grid sizes beyond a certain point. Prior to this, the workload distribution is relatively balanced, allowing for reasonable scaling, similar to the Conjugate Gradient method. However, as we exceed 36 threads, the scaling efficiency deteriorates. The primary culprit behind this suboptimal scaling is likely the increased synchronization overhead, which arises from the growing contention for shared resources among threads. This, in turn, leads to more frequent synchronization, resulting in significant overhead that escalates with the thread count. As a consequence, thread management and coordination become major bottlenecks, causing performance gains to dwindle. Furthermore, excessive synchronization can lead to thread idle time, ultimately limiting scalability.

There are potential workarounds that might work in fixing this performance scaling issue. One of them is making the load more balanced, so that there are no significant synchronization overheads and idling threads. Using **schedule(guided)** could potentially help in distributing the load more equally. Another workaround, might be removing the wavefront parallelization, and implementing a red-black Gauss Seidel iteration instead, to entirely avoid the synchronization overheads.