PTfS Project report

2D Modeling of the steady-state heat equation

Sharon Mathew Panamparambil Reji – ak65okod / Matriculation number: 23474807
Shail Ranjit Jadhavrao - sy99qaqi / Matriculation number: 23462119
Subhas Chandra Dahal - te86hidy / Matriculation number: 23492472

September 29, 2024

Architecture of the system:

Command: **likwid-topology -g**

- CPU type: Intel Icelake SP processor
- Sockets: 2
- Cores per socket: 36
- Threads per core: 1
- Socket 0: 0-35
- Socket 1: 36-71
- L1 Cache: 48kB (individual for each core)
- L2 Cache: 1.25 MB (individual for each core)
- L3 Cache: 54MB (each socket has one)
- NUMA domains: 4
    - Numa 0: 0-17
    - Numa 1: 18-35
    - Numa 2: 36-53
    - Numa 3: 54-71

Running the code for the first time:

1. Run the perf file in cores 0-17 of Socket-0 with fixed frequency.
   Command: srun --cpu-freq=2000000-2000000:performance likwid-pin -C S0:0-17 ./perf 2000 20000

   Output:

   Total number of threads active = 18
   CG iterations = 20
   Performance CG = 146.998565 [MLUP/s]

PCG iterations = 20
Performance PCG = 55.209713 [MLUP/s]

FUNCTION	COUNTS	TOTAL TIME (s)	TIME/COUNT (s)
AXPBY	128	6.6106330	0.0516456
DOT_PRODUCT	109	4.6295130	0.0424726
APPLY_STENCIL	45	2.2681120	0.0504025
CG	1	5.4422300	5.4422300
GS_PRE_CON	21	7.8459340	0.3736159
PCG	1	14.4902040	14.4902040

2. Run the perf file with the entire 72 cores in fixed frequency.
Command: srun --cpu-freq=2000000-2000000:performance likwid-pin -C S0:0-17 ./perf 2000 20000

Output:
Total number of threads active = 72
CG iterations = 20
Performance CG = 132.986328 [MLUP/s]
PCG iterations = 20
Performance PCG = 53.351533 [MLUP/s]

FUNCTION	COUNTS	TOTAL TIME (s)	TIME/COUNT (s)
AXPBY	128	7.1211330	0.0556339
DOT_PRODUCT	109	5.0466500	0.0462995
APPLY_STENCIL	45	2.5016130	0.0555914
CG	1	6.0156560	6.0156560
GS_PRE_CON	21	7.8674520	0.3746406
PCG	1	14.9948830	14.9948830

Initial observation:

- All tests passed.
- No significant improvement in the performance when we quadrupled the number of cores.

Starting the Parallelisation:

After the successful run of the test program, I have run the *perf* file using the following command.

srun --cpu-freq=2000000-2000000:performance likwid-pin -C S0:0-17 ./perf 3000 30000

Which gave an insight about the methods that are more dominant in the run time. These were, **AXPBY, DOT_PRODUCT, APPLY_STENCIL** and **GS_PRE_CON.** These functions were parallelised. Changes made for each of these are given below,

APPLY STENCIL

```
    int blockSizeX = 62000;
    #pragma omp parallel for collapse(2) schedule(static)
    for (int j = 1; j < ySize-1; ++j) {
        for (int ii = 1; ii < xSize-1; ii += blockSizeX) {
            // Operate within the block
             for (int i = ii; i < std::min(ii + blockSizeX, xSize-1); ++i)
{
                (*lhs)(j,i) = w_c * (*x)(j,i)
                            - w_y * ((*x)(j+1,i) + (*x)(j-1,i))
                            - w_x * ((*x)(j,i+1) + (*x)(j,i-1));
            }
        }
    }
```

Here the outer loop was parallelised and the entire iteration is divided among the threads using the static schedule. **Static** ensures that the work is divided among the threads equally and is assigned to the threads in a round robin fashion. **Collapse(2)** clause ensures that the two nested loops below the directive should be collapsed into a larger iteration scheme.

Since this is a STENCIL scheme to ensure that data is coming from L3 cache when larger grid sizes are considered we have implemented the L3 blocking. The block size was computed using the formulae,

$$No.\ of\ Threads\ *\ 3\ *\ Jblock\ *\ 8B\ <=\ CacheSize/2$$

Using this formula we were able to find that **Jblock <= 62500.** Hence we took the block size as 62000. Our reasoning behind this implementation is that when dealing with larger grid sizes, spatial blocking ensures that the maximum possible amount of data will be loaded from Cache thus reducing the main memory traffic.

For smaller grid sizes, 2000*20,000 and 20,000*2000, since layer condition is satisfied there will be 1 STORE ***((*lhs)(j,i))*** and 2 LOAD ***((*lhs)(j,i), (*x)(j-1,i)).***

For higher grid size, 1000*400000 since layer condition is not satisfied there will be 1 STORE ***((*lhs)(j,i))*** and 4 LOAD ***((*lhs)(j,i), (*x)(j-1,i), (*x)(j+1,i), (*x)(j,i+1))***
Also there are 7 (3 MULT, 4ADD) Double Precision Floating Point operations for this method.

DOT_PRODUCT

```
    #pragma omp parallel for reduction(+:dot_res) schedule(static)
    for(int yIndex=shift; yIndex<x->numGrids_y(true)-shift; ++yIndex)
    {
        for(int xIndex=shift; xIndex<x->numGrids_x(true)-shift; ++xIndex)
        {
            dot_res += (*x)(yIndex,xIndex)*(*y)(yIndex,xIndex);
        }
    }
```

Here the outer loop was parallelised and the entire iteration is divided among the threads using the static schedule. Static ensures that the work is divided among the threads equally and is assigned to the threads in a round robin fashion. **Reduction** clause ensures that the value of the shared variable **dot_res** which will be present in all of the threads will be combined together in the end and the original value will be updated with this newly computed value.
In the programs there are situations where the grids passed on as arguments for the dotProduct are same and different. When they are the same (X=X), there will only be **1 LOAD** and when they are different (X!=X) there will be **2 LOADS.** Also there are 1 Double Precision Floating Point operations for this method.

AXPBY

```
    #pragma omp parallel for schedule(static)
    for(int yIndex=shift; yIndex<lhs->numGrids_y(true)-shift; ++yIndex)
    {
        // #pragma omp parallel for
                for(int  xIndex=shift;  xIndex<lhs->numGrids_x(true)-shift;
++xIndex)
        {
                        (*lhs)(yIndex,xIndex)  =  (a*(*x)(yIndex,xIndex))  +
(b*(*y)(yIndex,xIndex));
        }
    }
```

Here the outer loop was parallelised and the entire iteration is divided among the threads using the static schedule. Static ensures that the work is divided among the threads equally and is assigned to the threads in a round robin fashion.
There will be **1 STORE and 2 LOAD**. Because when the method is called in solver.cpp one of the grids is getting repeated all the time and for this only load will be needed. There are also 3 Double Precision Floating Point operations in this program.

GS_PRE_CON

```
    int n,t,jj,j,i;
    #pragma omp parallel private(n,t,jj,j,i)
    {
    #ifdef LIKWID_PERFMON
        LIKWID_MARKER_START("GS_PRE_CON");
    #endif
        n=omp_get_num_threads();
        t=omp_get_thread_num();
        int interval = (xSize-2)/n;
        int ifs = interval*t+1;
        int ife = (t==(n-1))?(xSize-2):(ifs+interval-1);
        //forward substitution
        for ( j=1; j<ySize-1+n-1; ++j)
        {
            jj = j - t;
            if(jj>=1 && jj<ySize-1)
            {
                // #pragma omp for
                for ( i=ifs; i<=ife; ++i)
                {
                        (*x)(jj,i)=w_c*((*rhs)(jj,i)+(w_y*(*x)(jj-1,i)+
w_x*(*x)(jj,i-1)));
                }
            }
            #pragma omp barrier
        }
        //backward substitution
        for ( j=ySize-2+n-1; j>0; --j)
        {
            jj = j - t;
            if(jj<ySize-1 && jj>=1){
                // #pragma omp for
                for ( i=ife; i>=ifs; --i)
                {
                        (*x)(jj,i) = (*x)(jj,i) + w_c*(w_y*(*x)(jj+1,i) +
w_x*(*x)(jj,i+1));
                }
            }
```

```
            #pragma omp barrier
        }
    #ifdef LIKWID_PERFMON
        LIKWID_MARKER_STOP("GS_PRE_CON");
    #endif
    }
    STOP_TIMER(GS_PRE_CON);
```

We have parallelised the entire forward and backward substitution region. Each thread is assigned a chunk of columns of size *ife*. Once the forward substitution is over threads wait at the explicit barrier for synchronisation. Similar explicit synchronisation region exists after the end of the backward substitution region also.

For **forward substitution**, when we use smaller grid sizes, 2000*20,000 and 20,000*2000, since layer condition is satisfied there will be 1 STORE `((*x)(jj,i))` and 2 LOAD `((*x)(jj,i), (*rhs)(jj,i))`.For higher grid size, 1000*400000 since layer condition is not satisfied there will be 1 STORE `((*x)(jj,i))` and 3 LOAD `((*x)(jj,i), (*rhs)(jj,i), (*x)(jj-1,i))`

Also there are 5 (3 MULT, 2ADD) Double Precision Floating Point operations for this method.

For **backward substitution**, when we use smaller grid sizes, 2000*20,000 and 20,000*2000, since layer condition is satisfied there will be 1 STORE `((*x)(jj,i))` and 1 LOAD `((*x)(jj,i))`.For higher grid size, 1000*400000 since layer condition is not satisfied there will be 1 STORE `((*x)(jj,i))` and 2 LOAD `((*x)(jj,i), (*x)(jj,i+1))`. Also there are 5 (3 MULT, 2ADD) Double Precision Floating Point operations for this method.

1. ***Calculate roofline predictions in [LUP/s] for CG and PCG on 1 ccNUMA domain (18 cores) of Fritz. Calculate for three grid sizes : 2000 × 20000, 20000 × 2000 and 1000 × 400000. The last dimension is in x-direction (innermost). Does the performance change? Why?***

   After parallelising the methods in CG and PCG solvers, we have tried to obtain the roofline performance predictions in LUP/s. We have run the code in a single socket of **FRITZ ICE LAKE Processor.** It was instructed to consider bandwidth as **82 GB/s**. The Grid sizes are 2000 × 20000, 20000 × 2000 and 1000 × 400000.

   Since we are working with stencils it is necessary to ensure that the **Layer conditions** are fulfilled for each of these dimensions. For that we use the following formulae based on L3 cache,

   $$No.\ of\ Threads\ *\ 3\ *\ xSize\ *\ 8B\ <=\ L3\ CacheSize/2$$

   Since we are running on one socket of the ccNUMA domain, No. of Threads will be 18 and L3 cache size for this machine is 54MB.

When **xSize = 20000** and applying layer condition we get,

$$18 * 3 * 20,000 * 8 <= (54 * 10^6)/2$$
$$20,000 <= 62,500$$

This ensures that the Layer Condition is satisfied and data fits into the L3 cache.
When **xSize = 2000** and applying layer condition we get,

$$18 * 3 * 2000 * 8 <= (54 * 10^6)/2$$
$$2000 <= 62,500$$

This ensures that the Layer Condition is satisfied and data fits into the L3 cache.
When **xSize = 400000** and applying layer condition we get,

$$18 * 3 * 400000 * 8 <= (54 * 10^6)/2$$
$$400000 > 62,500$$

This means that the Layer Condition is not satisfied and data will be loaded from main memory. This also means that for larger grid sizes most of the data will be loaded from main memory instead of cache for the stencils. This might increase the amount of data transferred thus decreasing performance.
Based on the architecture and the code, the arithmetic operations and load and store operations for each function will be mapped as follows,

ApplyStencil						Bottleneck: ADD
					ADD	1 AVX iter takes 4 cy
				MULT	MULT	8 lup takes 4 cy
LD	LD	ST	ST	FMA	FMA	**P = 8/4 = 2 LUP/cy**
LD	LD	ST		MULT	ADD	
LD	LD			MULT	ADD	
				MULT	ADD	
					ADD	

AXPBY						Bottleneck: MULT
					ADD	2 AVX iter takes 3 cy
				MULT	MULT	16 lup takes 3 cy
LD	LD	ST	ST	FMA	FMA	**P = 16/3 = 5.333 LUP/cy**
LD	LD	ST		MULT	ADD	
LD	LD	ST		MULT	MULT	
				MULT	ADD	

Dotproduct						Bottleneck: LD
					ADD	1 AVX iter takes 1 cy
				MULT	MULT	8 lup takes 1 cy
LD	LD	ST	ST	FMA	FMA	**P = 8/1 = 8 LUP/cy**
LD	LD			FMA		

GSPre Forward						Bottleneck: MULT
					ADD	1 AVX iter takes 3 cy
				MULT	MULT	8 lup takes 3 cy
LD	LD	ST	ST	FMA	FMA	**P = 8/3= 2.667 LUP/cy**
LD	LD	ST		MULT	ADD	
LD				MULT	ADD	
				MULT		

GSPre Backward						Bottleneck: MULT
					ADD	1 AVX iter takes 3 cy
				MULT	MULT	8 lup takes 3 cy
LD	LD	ST	ST	FMA	FMA	**P = 8/3 = 2.667 LUP/cy**
LD	LD	ST		MULT	ADD	
				MULT	ADD	
				MULT		

Now to find the roofline predictions of the functions in the CG and PCG solvers, we have made use of the following equations,

$$Pmax = (LUP / Cycle) * f * No.\ of\ Threads$$

(f - clockspeed in GHz, Unit of P_{max} will be GLUP/s)

Code balance is computed using the formulae,

$$Bc = No.\ of\ Bytes\ transferred\ /\ LUP$$

(LUP - Lattice Site Update, Unit of B_c will be B/LUP)

Then Computational Intensity can be computed as,

$$I = 1/Bc$$

(Unit of I will be LUP/B)

Then we can get the roofline performance prediction for each grid size as,

$$P = min(Pmax, I.Bs)$$

(Unit of P will be GLUP/s)

The GSPrecon actually had two separate loop elements. One for forward substitution and other for backward substitution. In this we had to find the P_{max} value separately for both loops and combine them together. For that we have made use of following method,

$$T_{FW} = 1/P_{FW}\ ,\ T_{BW} = 1/P_{BW}$$

$$T_{GSPRECON} = T_{FW} + T_{BW}$$

$$P_{GSPRECON} = 1/\ T_{GSPRECON}$$

Once the performance for each of the methods from the two solvers is found out, we can find out the total performance for both solvers using the following method.Let

$$\mathbf{P_{axpby}\ ,\ P_{applystencil},\ P_{dotproduct},\ P_{GSprecon}}$$

be the performance for each of the methods. Then We can find out the time taken by each of these functions using the formula.

$$\mathbf{T_{axpby}=1/P_{axpby},\ T_{applystencil}=1/P_{applystencil},\ T_{dotproduct}=1/P_{dotproduct},\ T_{GSprecon}=1/P_{GSprecon}}$$

In the CG solver there is one occurrence of Apply Stencil, three occurrences of AXPBY, One occurrence of Dot Product(**X**=/= **X**) and one occurrence of Dot product (**X**= **X**). Therefore total time taken in the CG solver both grid dimensions can be computed using the formula,

$$\mathbf{T_{CG} = 1*T_{applystencil} + 3*T_{axpby} + 1*T_{dotproduct(}\textit{diff}\mathbf{)} + 1*T_{dotproduct(}\textit{same}\mathbf{)}}$$

Therefore, performance for the CG solver can be computed as,

$$\mathbf{P_{CG} = 1/T_{CG}\ [}\text{GLUP/s}\mathbf{]}$$

Now in the PCG solver there is one Apply Stencil, two Dot Products(**X**=/= **X**), one Dot product (**X**= **X**), three AXPBY, one GSPRECON. Therefore total time taken in the PCG solver both grid dimensions can be computed using the formula,

$$\mathbf{T_{PCG} = 1*T_{applystencil} + 3*T_{axpby} + 2*T_{dotproduct(}\textit{diff}\mathbf{)} + 1*T_{dotproduct(}\textit{same}\mathbf{)} + T_{GSprecon}}$$

Therefore, performance for the CG solver can be computed as,

$$P_{PCG} = 1/T_{PCG}\ [\text{GLUP/s}]$$

Based on the above formulas, we have computed roofline performance predictions for all methods under CG and PCG solvers for the three iteration schemes. This is attached below.

For **CG,**

CG (2000,20000)									
Function	**FLOPs**	**LD + ST**	**LUP /cy**	**Pmax in GLUP/s**	**Pmax*nCores in GLUP/s**	**Intensity in LUP/B**	**Bs in GB/sec**	**Intensity* Bs in [GLUP/s]**	**P_max = min(Pmax, I*Bs) in GLUP/s**
apply stencil	4ADD+ 3MULT	3LD+ 1ST	2	4	72	0.04	82.00	3.417	3.417
Dot product (X=/= X)	1FMA	1ST+ 1ST	8	16	288	0.0625	82.00	5.125	5.125
Dot product (X= X)	1FMA	1LD	8	16	288	0.125	82.00	10.25	10.25
axpby	1ADD+ 2MULT	2LD+ 1ST	5.33	10.67	192	0.04	82.00	3.42	3.42

CG (20000,2000)									
Function	**FLOPs**	**LD + ST**	**LUP /cy**	**Pmax in GLUP/s**	**Pmax*nCores in GLUP/s**	**Intensity in LUP/B**	**Bs in GB/sec**	**Intensity* Bs in [GLUP/s]**	**P_max = min(Pmax, I*Bs) in GLUP/s**
apply stencil	4ADD+ 3MULT	3LD+ 1ST	2	4	72	0.04	82.00	3.417	3.417
Dot product (X=/= X)	1FMA	1ST+ 1ST	8	16	288	0.0625	82.00	5.125	5.125
Dot product (X= X)	1FMA	1LD	8	16	288	0.125	82.00	10.25	10.25
axpby	1ADD+ 2MULT	2LD+ 1ST	5.33	10.67	192	0.04	82.00	3.42	3.42

CG (1000,400000)									
Function	**FLOPs**	**LD+ST**	**LUP/cy**	**Pmax in GLUP/s**	**Pmax[GLUP/s] * nCores**	**Intensity in LUP/B**	**Bs in GB/sec**	**Intensity* Bs in [GLUP/s]**	**P_max = min(Pmax, I*Bs) in GLUP/s**
apply stencil	4ADD+ 3MULT	3LD+ 1ST	2	4	72	0.03	82.00	2.050	2.050
Dot product (X=/= X)	1FMA	1ST+ 1ST	8	16	288	0.0625	82.00	5.125	5.125
Dot product (X= X)	1ADD+ 2MULT	1LD	8	16	288	0.125	82.00	10.25	10.25
axpby	1FMA	2LD+ 1ST	5.33	10.67	192	0.04	82.00	3.42	3.42

Based on these values, the total performance computed for the solver **CG** under the two grid dimensions are,

Total Performance of CG (2000,20000) & (20000,2000)			
Functions	**Occurrence**	**Perf[GLUP/s]**	**Time*Occurrence = (1/Perf)*Occurrence**
apply stencil	1	3.417	0.293
Dot product (X=/= X)	1	5.125	0.195
Dot product (X= X)	1	10.250	0.098
axpby	3	3.417	0.878
	Total time		1.463
	Total Perf[GLUP/s]	0.68	
	Total Perf[MLUP/s]	**683.33**	**Expected**
		652.53	**Measured(2000*20000)**
		653.64	**Measured(20000*2000)**

Total Performance of CG (1000,400000)			
Functions	**Occurance**	**Perf[GLUP/s]**	**Time = 1/Perf**
apply stencil	1	2.05	0.488

Dot product (X=/= X)	1	5.125	0.195
Dot product (X= X)	1	10.25	0.098
axpby	3	3.42	0.878
	Total time		1.659
	Total Perf[GLUP/s]	0.60	
	Total Perf[MLUP/s]	**602.94**	**Expected**
		594.10	**Measured**

For PCG,

PCG (2000,20000)									
Function	**FLOPs**	**LD+ST**	**LUP /cy**	**Pmax[GLUP/s] (*10^9)**	**Pmax[GLUP/s]*No.Cores(18)**	**Intensity [LUP/B]**	**Bs[GB/sec]**	**Intensity* Bs [GLUP/sec]**	**P_max [min(Pmax, I*Bs)]**
apply stencil	4ADD+ 3MULT	3LD+ 1ST	2	4	72	0.04	82.00	3.417	3.417
Dot product (X=/= X)	1FMA	1ST+ 1ST	8	16	288	0.0625	82.00	5.125	5.125
Dot product (X= X)	1FMA	1LD	8	16	288	0.125	82.00	10.25	10.25
axpby	1ADD+ 2MULT	2LD+ 1ST	5.33	10.67	192	0.04	82.00	3.42	3.42
GSPre (forward)	2ADD+ 3MULT	2LD+ 1ST	2.67	5.33	96	0.04	82.00	3.42	3.42
GSPre (backward)	2ADD+ 3MULT	1LD+ 1ST	2.67	5.33	96	0.06	82.00	5.13	5.13
GSPre (combined)									2.05

PCG (20000,2000)

Function	FLOPs	LD+ST	LUP/cy	Pmax[GLUP/s] *10^9	Pmax[GLUP/s]*No.Cores(18)	Intensity [LUP/B]	Bs[GB/sec]	Intensity *Bs [GLUP/sec]	Pstencil [min(Pmax, I*Bs)]
apply stencil	4ADD+ 3MULT	3LD+ 1ST	2	4	72	0.04	82.00	3.417	3.417
Dot product (X=/= X)	1FMA	1ST+ 1ST	8	16	288	0.0625	82.00	5.125	5.125
Dot product (X= X)	1ADD+ 2MULT	1LD	8	16	288	0.125	82.00	10.25	10.25
axpby	1FMA	2LD+ 1ST	5.33	10.67	192	0.04	82.00	3.42	3.42
GSPre (forward)	2ADD+ 3MULT	2LD+ 1ST	2.67	5.33	96	0.04	82.00	3.42	3.42
GSPre (backward)	2ADD+ 3MULT	1LD+ 1ST	2.67	5.33	96	0.06	82.00	5.13	5.13
GSPre (combined)									2.05

PCG (1000,400000)									
Function	**FLOPs**	**LD + ST**	**LUP/cy**	**Pmax[GLUP/s] *10^9**	**Pmax[GLUP/s]*No.Cores(18)**	**Intensity [LUP/B]**	**Bs[GB/sec]**	**Intensity* Bs [GLUP/sec]**	**Pstencil [min(Pmax, I*Bs)]**
apply stencil	4ADD+43MULT	3LD+ 1ST	2	4	72	0.025	82	2.05	2.05
Dot product (X=/= X)	1FMA	1ST+ 1ST	8	16	288	0.06	82.00	5.125	5.125
Dot product (X= X)	1ADD+ 2MULT	1LD	8	16	288	0.125	82.00	10.25	10.25
axpby	1FMA	2LD+ 1ST	5.33	10.67	192	0.04	82.00	3.417	3.417
GSPre (forward)	2ADD+ 3MULT	2LD+ 1ST	2.67	5.33	96	0.03	82.00	2.56	2.56
GSPre (backward)	2ADD+ 3MULT	1LD+ 1ST	2.67	5.33	96	0.04	82.00	3.42	3.42
GSPre (combined)									1.46

Based on these values, the total performance computed for the solver **CG** under the two grid dimensions are,

Total Performance of PCG (2000,20000) & (20000,2000)			
Functions	**Occurrence(n)**	**Perf[GLUP/s]**	**Time*n = (1/Perf)*n**
apply stencil	1	3.417	0.293
Dot product (X=/= X)	2	5.125	0.390
Dot product (X= X)	1	10.25	0.098
axpby	3	3.42	0.878
GSPreCon	1	2.05	0.488
	Total time		1.659
	Total Perf[GLUP/s]	0.603	
	Total Perf[MLUP/s]	**602.94**	**Expected**
		400.57	**Measured(2000*20000)**
		267.21	**Measured(20000*2000)**

Total Performance of PCG (1000,400000)			
Functions	**Occurrence(n)**	**Perf[GLUP/s]**	**Time*n = (1/Perf)*n**
apply stencil	1	2.05	0.487804878
Dot product (X=/= X)	2	5.125	0.3902439024
Dot product (X= X)	1	10.25	0.09756097561
axpby	3	3.417	0.8780487805
GSPreCon	1	1.46	0.6829268294
	Total time		1.853658537
	Total Perf[GLUP/s]	0.5394736842	
	Total Perf[MLUP/s]	**539.4736842**	**Expected**
		398.671348	**Measured**

Then the expected performance for solvers can be denoted as,

GridSize	2000*20,000	20,000*2000	1000*4,00,000
CG (MLUP/s)	**683.33**	**683.33**	**602.9411765**
PCG (MLUP/s)	**602.94**	**602.94**	**539.4736842**

Performance is expected to drop, because xSize = 400000 violates the layer condition and therefore, more data needs to be loaded from the main memory.

2. ***Check whether you attain the roofline performance by running the code on 1 ccNUMA domain of Fritz for the three dimensions given above. Timings are already included. Run the code using the following command. ./perf num_grids_y num_grids_x***

To run the code on one ccNUMA domain of the Fritz cluster, we first requested for a single interactive node using the following command,

salloc --nodes=1 --time=1:00:00 -p singlenode -C hwperf

Then the we requested for the Intel and likwid modules using the command,

module load intel
module load likwid

Afterwards we compiled the code using the **make** command.
One ccNUMA domain contains 18 cores. Hence we decided to run the code in socket S0. It was mentioned to fix the clockspeed at 2GHz. Hence to measure the performance we ran the following commands,

srun --cpu-freq=2000000-2000000 likwid-pin -C S0:0-17 ./perf 2000 20000
srun --cpu-freq=2000000-2000000 likwid-pin -C S0:0-17 ./perf 20000 2000
srun --cpu-freq=2000000-2000000 likwid-pin -C S0:0-17 ./perf 1000 400000

Here **likwid-pin -C** is used to pin the threads to the 18 cores of socket 0.

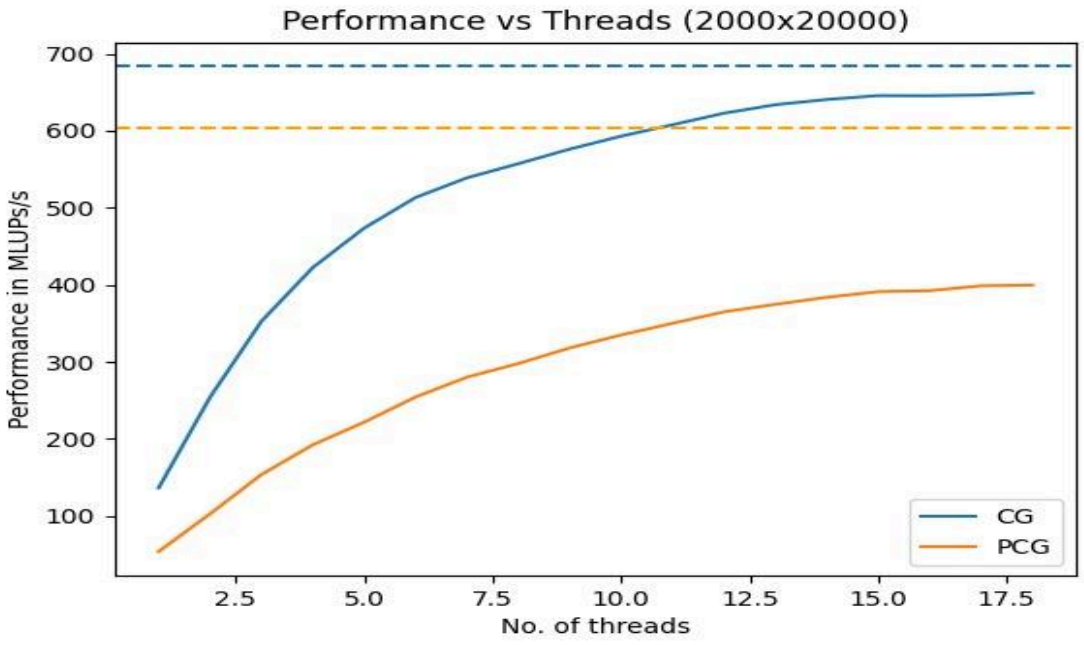
We got the following performance measurements while running the code,

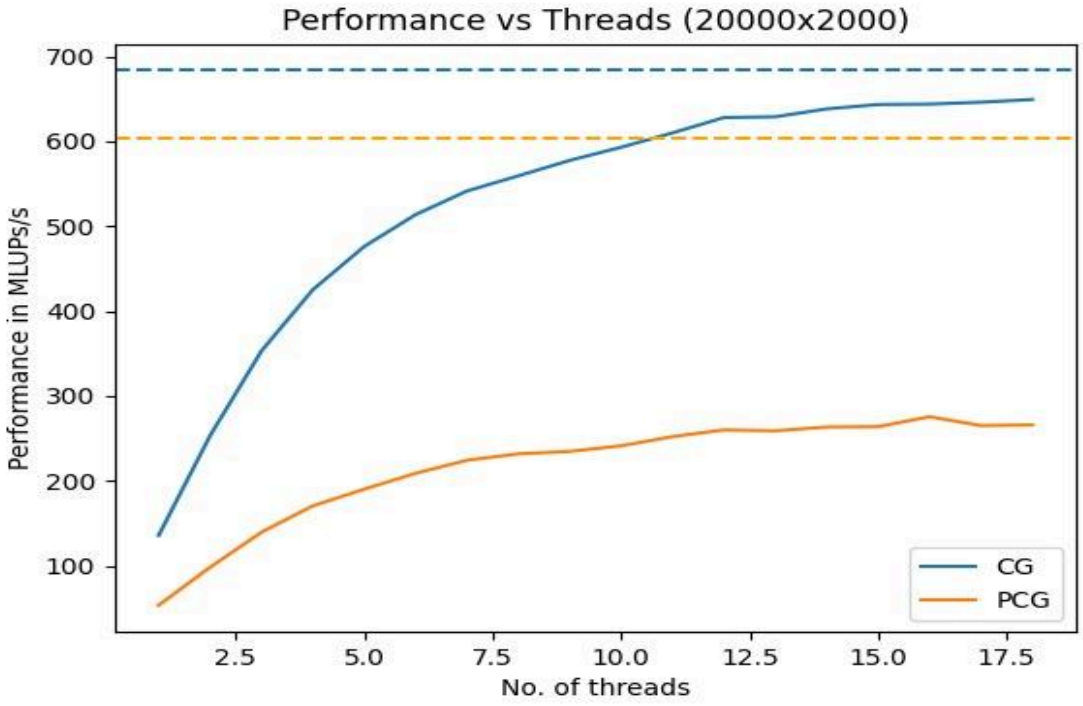
GridSize	2000*20,000	20,000*2000	1000*4,00,000
CG (MLUP/s)	652.53	653.64	594.09839
PCG (MLUP/s)	400.57	267.21	398.671348

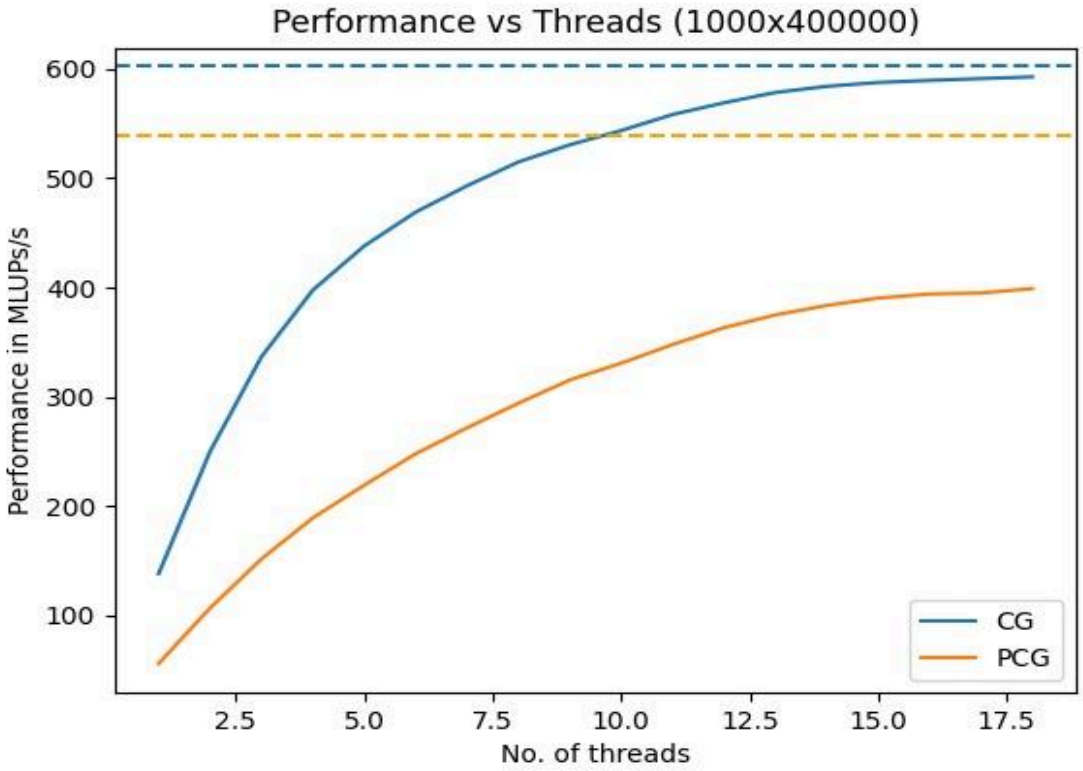
Compiling the expected and measured performance values together,

GridSize	2000*20,000		20,000*2000		1000*4,00,000	
	Expected	Measured	Expected	Measured	Expected	Measured
CG (MLUP/s)	683.33	652.53	683.33	653.64	602.9412	594.0984
PCG (MLUP/s)	602.94	400.57	602.94	267.21	539.4737	398.671348

It can be inferred that the measured performances are well within the roofline predictions. In no instance the measured performance is seen to be above expected and in most of the cases measured performance is close to the expected performance.
We have also plotted performance against the number of threads to find how it is scaling.




Performance vs Threads (20000x2000)
Performance in MLUPs/s
No. of threads
CG
PCG
700
600
500
400
300
200
100
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Performance vs Threads (1000x400000)
Performance in MLUPs/s
No. of threads
CG
PCG
600
500
400
300
200
100
2.5
5.0
7.5
10.0
12.5
15.0
17.5

From the graph it is evident that, for **CG**,performance is saturating below the expected limit that we have computed using roofline modelling thus confirming our findings. For **PCG** also the performance is saturating as we reach towards 18 cores. But the maximum performance code can achieve is less than the theoretical roofline limit.

3. ***Measure the code balance in [bytes/LUP] of 'applyStencil' and 'GSPreCon' kernels on 1 ccNUMA domain (18 threads) for the three grid dimensions and comment whether it agrees with your model. You can use LIKWID for the measurement. • To switch on LIKWID measurement set the LIKWID flag to 'on', i.e., LIKWID=on CXX=icpx make***

To get the LIKWID measurements we have ran the program using the following commands,

srun --cpu-freq=2000000-2000000 likwid-perfctr -C S0:0-17 -g MEM -m ./perf 2000 20000
srun --cpu-freq=2000000-2000000 likwid-perfctr -C S0:0-17 -g MEM -m ./perf 20000 2000
srun --cpu-freq=2000000-2000000 likwid-perfctr -C S0:0-17 -g MEM -m ./perf 1000 4000000

The output gives several measurements from which we only need **data volume** and **call count.**
Using these measurements we can calculate the code balance using following formulae,

$$Bc = data\ volume\ /\ (call\ count\ *\ xSize\ *\ ySize)$$

Apply Stencil

GridSize	Data Volume (GB)	CallCount	xSize	ySize	Bc (measured) B/LUP	Bc(expected) B/LUP
2000*20000	39.8578	45	20000	2000	22.14322222	25
20000*2000	39.9344	45	2000	20000	22.18577778	25
1000*400000	624.4936	45	400000	1000	34.69408889	40

From the table we can infer that the measured code balance is less than what we expected. Especially when grid size is 2000*20000 and 20000*2000, measured and expected values are close hence it is possible to say that the expected values are almost accurate. But when grid size is 1000*400000, when we compare expected code balance with measured we can find that it is close but not fully accurate.

GSPreCon

GridSize	Data Volume (GB)	CallCount	xSize	ySize	Bc (measured) B/LUP	Bc(expected) B/LUP
2000*20000	32.6008	21	2000	20000	38.81047619	40
20000*2000	32.9392	21	20000	2000	39.21333333	40
1000*400000	337.5788	21	400000	1000	40.18795238	56

From the table we can infer that the measured code balance is close to what we expected. Especially when grid size is 2000*20000 and 20000*2000, measured and expected values are close hence it is possible to say that the expected values are accurate.
But when grid size is 1000*400000, there is a considerable difference between expected and measured code balance. While computing the expected code balance for GSPreCon with Grid size 1000*400000, we assumed that there were 5 loads and 2 stores taking into account write allocate. We assume that the compiler did not perform the write allocate and we considered that as extra, hence the difference in code balance.

4. ***How does the CG and PCG code scale from 1 ccNUMA domain to 1 node (4 ccNUMA domains, 72 cores) of Fritz? Does it scale perfectly? If not, why? Can you fix it?***

In order to measure if there is scaling of performance from 1 ccNUMA domain(18 cores) to 4 ccNUMA domain (72 cores) we have requested for a dedicated node of FRITZ using the command,

salloc --nodes=1 --time=1:00:00 -p singlenode -C hwperf

Afterwards we have loaded intel and likwid modules using,

module load intel
module load likwid

We have fixed the clockspeed at 2GHz,
In order to get the performance on 1 ccNUMA domain we have run the following codes,

```
srun --cpu-freq=2000000-2000000 likwid-pin -C S0:0-17 ./perf 2000 20000
srun --cpu-freq=2000000-2000000 likwid-pin -C S0:0-17 ./perf 20000 2000
srun --cpu-freq=2000000-2000000 likwid-pin -C S0:0-17 ./perf 1000 400000
```

In order to get the performance on 4 ccNUMA domain we have run the following codes,

```
srun --cpu-freq=2000000-2000000 likwid-pin -C S0:0-35@S1:36-71 ./perf 2000 20000
srun --cpu-freq=2000000-2000000 likwid-pin -C S0:0-35@S1:36-71 ./perf 20000 2000
srun --cpu-freq=2000000-2000000 likwid-pin -C S0:0-35@S1:36-71 ./perf 1000 400000
```

We have received the following performance predictions,

Dimension	Solver	18 cores	36 cores	54 cores	72 cores
2000*20000	**CG**	652.53	664.76	599.76	566.69
	PCG	400.57	415.98	357.53	335.26
20000*2000	**CG**	653.64	663.62	599.60	569.88
	PCG	267.21	253.37	186.18	160.81
1000*400000	**CG**	594.10	602.65	557.07	536.56
	PCG	398.67	421.50	366.87	343.75

It is evident from the table that performance is not scaling. After saturating around 36 cores it further decreases after 54 cores.
However we have also tried to measure performance with a *numactl* option to ensure pages (data) are distributed across the nodes in a round robin fashion (pinning). To measure performance scaling we have tried to run the program in 18, 36, 54, 72 cores for the three dimensions. We have used the following argument for that,

srun --cpu-freq=2000000-2000000:performance numactl --interleave=all likwid-pin -C S0:0-17 ./perf ySize xSize
srun --cpu-freq=2000000-2000000:performance numactl --interleave=all likwid-pin -C S0:0-35./perf ySize xSize
srun --cpu-freq=2000000-2000000:performance numactl --interleave=all likwid-pin -C S0:0-35@S1:36-53 ./perf ySize xSize
srun --cpu-freq=2000000-2000000:performance numactl --interleave=all likwid-pin -C S0:0-35@S1:36-71 ./perf ySize xSize

We have received the following performance predictions,

extra parameters	Dimension	Solver	18 cores	36 cores	54 cores	72 cores
numactl --interleave=all	2000*20000	**CG**	1192.36	1223.08	1582.34	1853.04
		PCG	599.18	677.51	791.92	864.30
	20000*2000	**CG**	1205.09	1222.90	1588.27	1884.94
		PCG	328.76	322.08	218.78	217.20
	1000*400000	**CG**	1000.74	1042.36	1360.82	1558.23
		PCG	610.43	726.61	920.46	1030.78

From the table, we can infer that performance is increasing when we increase the number of cores. That is there is a scaling of performance. We assume that this increase is due to

efficient cache usage. There is only one exception for this trend, which is the PCG solver for the grid size 20000*2000. We assume that performance decreased because the column size (iteration space) was not large enough to suppress the overhead contributed by explicit barrier and blocking.