Chapter 1

Preface

In this project we are supposed to parallelize the existing Steady state heat transfer simulation. along with

- Calculating the Roof line performance for 1 ccNUMA
- run the parallelized on 1 ccNUMA domain and compare with roofline performance.
- measure the code balance and compare with our predicted code balance
- scale the code to work on 4 ccNUMA domains.

the code is going to be run on a single node of fritz and its specifications are as follow

- Processor: Intel® Xeon® Platinum 8360Y Processor (Ice Lake)
- Clock Frequency: 2.39 GHz (fixed at (f_{cpu}) 2.0 GHz for this project)
- Memory Bandwidth (b_s): 82 GB/s (given)
- L3 Cache Size (1 socket-36 cores): 54 MB
- L2 Cache Size (per core): 1.25 MB
- L1 Cache Size (per core): 48 KB
- ccNUMA Domains: 4
- Sockets: 2
- No. of Cores (per socket): 36
- 2 AVX-512 FMA units (source: NHR FAU website)
- Assumption No write allocates considered for Fritz

We have written few scripts that would build and run the code and generate the proper graphs and information for post processing their description and overview are

- script_testing.sh : used to build and run the test of our code.
- script_performance.sh is used to build and run the performance on 18 cores for different grid sizes of our code.
- script_1ccnuma.sh : used to build and run the code on 0-18 cores of first ccNUMA domain and generate the performance graph with roof line indicated.
- script_4ccnuma.sh : used to build and run the code on 0-72 cores of all 4 ccNUMA domain and generate the performance graphs.

- script_codebalance_1ccnuma.sh is used to run our code on 18 cores of first ccNUMA domain and get the code balance for each kernel and create a file to store the data.
- script_allrun.sh : used to schedule runs of script_1ccnuma.sh, script_4ccnuma.sh and script_codebalance_1ccnuma.sh.
- script_cleaner.sh is used to remove all the generated and auxiliary files generated while compiling and running our codes.

all the scripts mentioned above are placed in the project folder along with the source code.

Chapter 2

Parallelize Code

After reviewing timing summary there are multiple kernels that are good candidates for parallelization, in order of time taken in descending order.

1. PCG
2. CG
3. GS_PRE_CON
4. AXPBY
5. APPLY_STENCIL
6. DOT_PRODUCT

But it can be observed that PCG and CG are formed by the using AXPBY, DOT_PRODUCT, APPLY_STENCIL and GS_PRE_CON Kernels.

2.1 AXPBY

AXPBY function has only spatial locality with no temporal locality. it can be easily parallelized by performing the parallelization along the rows. we need to call "LIKWID_MARKER_START" and "LIKWID_MARKER_STOP" from all the threads so adding it inside a omp parallel block will suffice. using a nowait clause will result in elimination of implicit barriers at the end of for loop.

```
#pragma omp parallel
  {
#ifdef LIKWID_PERFMON
    LIKWID_MARKER_START("AXPBY");
#endif

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

#ifdef LIKWID_PERFMON
  LIKWID_MARKER_STOP("AXPBY");
#endif
  }
```

2.2 DOT_PRODUCT

Similarly, DOT_PRODUCT also has only spatial locality. and can be trivially parallelized. As dot_res is going to be written from multiple threads care has to be taken to prevent data race. One of the possible options is to use atomic add but this is not scaleable as all threads write to same memory location. So we preferred using a omp reduction clause for better scalability.

```
#pragma omp parallel
  {
#ifdef LIKWID_PERFMON
    LIKWID_MARKER_START("DOT_PRODUCT");
#endif
#pragma omp for reduction(+ : dot_res) nowait
    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);
      }
    }

#ifdef LIKWID_PERFMON
  LIKWID_MARKER_STOP("DOT_PRODUCT");
#endif
  }
```

Chapter 3

Possible Optimization

3.1 APPLY_STENCIL

The APPLY_STENCIL has temporal locality, that can be exploited using spatial blocking to get the best performance. The spatial blocking has been performed on L2 cache and is parallelized along the rows of the blocks of columns.

```
int collimit = (1.25 * 1000 * 1000) / 48;
int colend = 0;
collimit = std::min(collimit, xSize - 1);

#pragma omp parallel private(colend)
  {
#ifdef LIKWID_PERFMON
    LIKWID_MARKER_START("APPLY_STENCIL");
#endif
    for (int colstart = 1; colstart < xSize - 1; colstart += collimit)
    {
      colend = std::min(colstart + collimit, xSize) - 1;
#pragma omp for nowait
      for (int j = 1; j < ySize - 1; ++j)
      {
        for (int i = colstart; i < colend; ++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));
        }
      }
    }

#ifdef LIKWID_PERFMON
  LIKWID_MARKER_STOP("APPLY_STENCIL");
#endif
  }
```

3.2 GS_PRE_CON

GS_PRE_CON cannot be trivially parallelized. The standard Gauss-Seidel algorithm updates each variable in a sequence, with each update depending on the previously computed variables. This sequential nature makes it hard to parallelized directly because each update depends on the completion of previous updates in the same iteration. Wavefront parallelization can be used here. It can be thought of as working diagonally or in a wave-like pattern, where certain diagonals of the matrix can be updated independently of others.

```
int num_th, th_id, jj, j, i;
#pragma omp parallel private(num_th, th_id, jj, i, j)
  {
#ifdef LIKWID_PERFMON
    LIKWID_MARKER_START("GS_PRE_CON");
#endif
    num_th = omp_get_num_threads();
    th_id = omp_get_thread_num();

    int interval = (xSize - 2) / num_th;
    int interval_s = interval * th_id + 1;
    int interval_e = (th_id == (num_th - 1)) ? (xSize - 2) : (interval_s + interval - 1);

    // forward substitution
    for (j = 1; j < ySize - 1 + num_th - 1; ++j)
    {
      jj = j - th_id;
      if (jj >= 1 && jj < ySize - 1)
      {
        for (i = interval_s; i <= interval_e; ++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 + num_th - 1; j > 0; --j)
    {
      jj = j - th_id;
      if (jj < ySize - 1 && jj >= 1)
      {
        for (i = interval_e; i >= interval_s; --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
  }
```

Chapter 4

Roofline prediction

we have 3 different grid sizes to consider for roofline modelling.

(a) 2000 x 20000 (case 1) = 20000 x 2000 (case 2) = 40 MLUP

(b) 1000 x 400000 (case 3) = 400 MLUP

4.1 dotProduct(v, p)

The Dot product can use AVX512 FMA instruction to improve the performance. To get the best performance compiler performs MVE to remove data dependency. the final instruction mix would be

- 2 AVX512 LOAD
- 1 AVX512 FMA

"Ice Lake" Execution Units / Ports (Max 5 instruction per cycle)				
		Cache line : 512 bit		AVX512ADD
			AVX512MULT	AVX512MULT
AVX512LD	AVX512LD	AVX512ST	AVX512FMA	AVX512FMA
AVX512LD	AVX512LD		AVX512FMA	

here the bottleneck is AVX512 LOAD unit

$$P_{max}^{dot(v,p)} = \frac{8*2}{1} = 16\frac{flops}{cy}$$

it can be observed that it takes 2 flops per each LUP

$$P_{max}^{dot(v,p)} = \frac{16\frac{flops}{cy}}{2\frac{flops}{LUP}} = 8\frac{LUP}{cy}$$

at the clock frequency(f) of 2 GHz

$$P_{max}^{dot(v,p)} = 8\frac{LUP}{cy} * 18\ cores * 2\ GHz = 288\frac{GLUP}{s}$$

for each LUP we need to load 2 double elements and as we only have spatial locality in this kernel we need to load 16 bytes per each lattice site.

$$B_c = 16\frac{bytes}{LUP}$$

Now the roofline performance for dotProduct(v, p) is

$$P_{roof}^{dot(v,p)} = min(P_{max}^{dot(v,p)}, \frac{b_s}{B_c}) = min(288\ \frac{GLUP}{s}, \frac{82\frac{GB}{s}}{16\frac{bytes}{LUP}}) = 5.125\frac{GLUP}{s}$$

time taken can be calculated using

$$T_{1,2}^{dot(v,p)} = \frac{LUP}{P_{roof}^{dot(v,p)}} = \frac{40\ MLUP}{5.125 * 10^3 \frac{MLUP}{s}} = 7.804\ ms$$

$$T_{3}^{dot(v,p)} = \frac{LUP}{P_{roof}^{dot(v,p)}} = \frac{400\ MLUP}{5.125 * 10^3 \frac{MLUP}{s}} = 78.04\ ms$$

4.2 dotProduct(r, r)

The Dot product can use AVX512 instruction and FMA units to improve the performance. To get the best performance compiler performs MVE to remove data dependency. the final instruction mix would be

- 1 AVX512 LOAD
- 1 AVX512 FMA

but we need to perform 2 AVX512 iteration for best performance.

"Ice Lake" Execution Units / Ports (Max 5 instruction per cycle)				
		Cache line : 512 bit		AVX512ADD
			AVX512MULT	AVX512MULT
AVX512LD	AVX512LD	AVX512ST	AVX512FMA	AVX512FMA
AVX512LD	AVX512LD		AVX512FMA	AVX512FMA

here the bottleneck is AVX512 LOAD/FMA unit

$$P_{max}^{dot(r,r)} = \frac{8 * 4}{1} = 32\frac{flops}{cy}$$

it can be observed that it takes 2 flops per each LUP

$$P_{max}^{dot(r,r)} = \frac{32\frac{flops}{cy}}{2\frac{flops}{LUP}} = 16\frac{LUP}{cy}$$

at the clock frequency(f) of 2 GHz

$$P_{max}^{dot(r,r)} = 16\frac{LUP}{cy} * 18\ cores * 2\ GHz = 576\frac{GLUP}{s}$$

for each LUP we need to load 2 double elements and as we only have spatial locality in this kernel we need to load 8 bytes per each lattice site.

$$B_c = 8\frac{bytes}{LUP}$$

Now the roofline performance for dotProduct(r, r) is

$$P_{roof}^{dot(r,r)} = min(P_{max}^{dot(r,r)}, \frac{b_s}{B_c}) = min(576\ \frac{GLUP}{s}, \frac{82\frac{GB}{s}}{8\frac{bytes}{LUP}}) = 10.25\frac{GLUP}{s}$$

time taken can be calculated using

$$T_{1,2}^{dot(r,r)} = \frac{LUP}{P_{roof}^{dot(r,r)}} = \frac{40\ MLUP}{10.25 * 10^3 \frac{MLUP}{s}} = 3.902\ ms$$

$$T_{3}^{dot(r,r)} = \frac{LUP}{P_{roof}^{dot(r,r)}} = \frac{400\ MLUP}{10.25 * 10^3 \frac{MLUP}{s}} = 39.02\ ms$$

4.3 axpby(x, 1.0, x, lambda, p)

Perform 2 AVX iteration at a time for better utilization The final instruction mix would be

- 2 AVX512 LOAD
- 1 AVX512 ST
- 1 AVX512 ADD
- 2 AVX512 MULT

"Ice Lake" Execution Units / Ports (Max 5 instruction per cycle)				
		Cache line : 512 bit		AVX512ADD
			AVX512MULT	AVX512MULT
AVX512LD	AVX512LD	AVX512ST	AVX512FMA	AVX512FMA
AVX512LD	AVX512LD	AVX512ST	AVX512MULT	AVX512ADD
AVX512LD	AVX512LD	AVX512ST	AVX512MULT	AVX512MULT
			AVX512MULT	AVX512ADD

here the bottleneck is AVX512 MULT/ADD unit

$$P_{max}^{axpby} = \frac{8 * 6}{3} = 16 \frac{flops}{cy}$$

it can be observed that it takes 3 flops per each LUP

$$P_{max}^{axpby} = \frac{16 \frac{flops}{cy}}{3 \frac{flops}{LUP}} = 5.33 \frac{LUP}{cy}$$

at the clock frequency(f) of 2 GHz

$$P_{max}^{axpby} = 5.33 \frac{LUP}{cy} * 18\ cores * 2\ GHz = 192 \frac{GLUP}{s}$$

for each LUP we need to load 3 double elements and as we only have spatial locality in this kernel we need to load 24 bytes per each lattice site.

$$B_c = 24 \frac{bytes}{LUP}$$

Now the roofline performance for axpby(x, 1.0, x, lambda, p) is

$$P_{roof}^{axpby} = min(P_{max}^{axpby}, \frac{b_s}{B_c}) = min(192\ \frac{GLUP}{s}, \frac{82 \frac{GB}{s}}{24 \frac{bytes}{LUP}}) = 3.42 \frac{GLUP}{s}$$

time taken can be calculated using

$$T_{1,2}^{axpby} = \frac{LUP}{P_{roof}^{axpby}} = \frac{40\ MLUP}{3.42 * 10^3 \frac{MLUP}{s}} = 11.6959\ ms$$

$$T_{3}^{axpby} = \frac{LUP}{P_{roof}^{axpby}} = \frac{400\ MLUP}{3.42 * 10^3 \frac{MLUP}{s}} = 116.959\ ms$$

4.4 applyStencil

It has been observed using likwid-perfctr only scalar instructions are used. for The final instruction mix for 1 iteration would be

- 5 LOAD
- 1 ST
- 4 ADD
- 3 MULT

we need to perform 2 iteration at a time to get the best performance.

"Ice Lake" Execution Units / Ports (Max 5 instruction per cycle)					
		Cache line : 512 bit		ADD	ADD
		2 stores if hit the same cache line		MULT	MULT
LOAD	LOAD	STORE	STORE	FMA	FMA
LOAD	LOAD	STORE		MULT	ADD
LOAD	LOAD	STORE		MULT	ADD
LOAD	LOAD			MULT	ADD
LOAD	LOAD			ADD	ADD
LOAD	LOAD			MULT	ADD
				MULT	ADD
				MULT	ADD

here the bottleneck is AVX512 ADD/MULT unit

$$P_{max}^{applyStencil} = \frac{14}{7} = 2\frac{flops}{cy}$$

it can be observed that it takes 7 flops per each LUP

$$P_{max}^{applyStencil} = \frac{2\frac{flops}{cy}}{7\frac{flops}{LUP}} = 0.29\frac{LUP}{cy}$$

at the clock frequency(f) of 2 GHz

$$P_{max}^{applyStencil} = 0.29\frac{LUP}{cy} * 18\ cores * 2\ GHz = 10.44\frac{GLUP}{s}$$

for each LUP we need to load 1 double element and as we have implemented spatial blocking in this kernel we need to load 8 bytes per each lattice site and store 8 bytes.

$$B_c = 16\frac{bytes}{LUP}$$

Now the roofline performance for applyStencil is

$$P_{roof}^{applyStencil} = min(P_{max}^{applyStencil}, \frac{b_s}{B_c}) = min(10.44\ \frac{GLUP}{s}, \frac{82\frac{GB}{s}}{16\frac{bytes}{LUP}}) = 5.125\frac{GLUP}{s}$$

time taken can be calculated using

$$T_{1,2}^{applyStencil} = \frac{LUP}{P_{roof}^{applyStencil}} = \frac{40\ MLUP}{5.125 * 10^3 \frac{MLUP}{s}} = 7.804\ ms$$

$$T_{3}^{applyStencil} = \frac{LUP}{P_{roof}^{applyStencil}} = \frac{400\ MLUP}{5.125 * 10^3 \frac{MLUP}{s}} = 78.04\ ms$$

4.5 GSPreCon(r, z)[Forward/Backward]

It has been observed using likwid-perfctr only scalar instructions are used this is due to data dependency in the gsprecon forward and backward pass. The final instruction mix would be for 1 loop iteration be

- 3 LOAD
- 1 ST
- 2 ADD
- 3 MULT

we need to do 2 iterations at a time for best performance.

"Ice Lake" Execution Units / Ports (Max 5 instruction per cycle)					
		Cache line : 512 bit		ADD	ADD
		2 stores if hit the same cache line		MULT	MULT
LOAD	LOAD	STORE	STORE	FMA	FMA
LOAD	LOAD	STORE		MULT	ADD
LOAD	LOAD	STORE		MULT	ADD
LOAD	LOAD			MULT	ADD
				MULT	ADD
				MULT	MULT

here the bottleneck is scalar ADD/MULT units

$$P_{max}^{GSPreCon} = \frac{10}{5} = 2\frac{flops}{cy}$$

it can be observed that it takes 5 flops per each LUP

$$P_{max}^{GSPreCon} = \frac{2\frac{flops}{cy}}{5\frac{flops}{LUP}} = 0.4\frac{LUP}{cy}$$

at the clock frequency(f) of 2 GHz

$$P_{max}^{GSPreCon} = 0.4\frac{LUP}{cy} * 18\ cores * 2\ GHz = 14.4\frac{GLUP}{s}$$

for each LUP we need to load 2 double elements (as x(j,i-1) in forward pass and x(j, i+1) in backward pass will be in the L1 cache) we need to load 16 bytes per each lattice site and store 1 element of 8 bytes.

$$B_c = 24\frac{bytes}{LUP}$$

Now the roofline performance for GSPreCon is

$$P_{roof}^{GSPreCon} = min(P_{max}^{GSPreCon}, \frac{b_s}{B_c}) = min(14.4\ \frac{GLUP}{s}, \frac{82\frac{GB}{s}}{24\frac{bytes}{LUP}}) = 3.416\frac{GLUP}{s}$$

time taken can be calculated using

$$T_{1,2}^{GSPreCon} = \frac{LUP}{P_{roof}^{GSPreCon}} = \frac{40\ MLUP}{3.416 * 10^3 \frac{MLUP}{s}} = 11.709\ ms$$

$$T_{3}^{GSPreCon} = \frac{LUP}{P_{roof}^{GSPreCon}} = \frac{400\ MLUP}{3.416 * 10^3 \frac{MLUP}{s}} = 117.09\ ms$$

4.6 CG roofline performance

the CG solver is composed of multiple kernels listed below

- pde− >applyStencil
- dotProduct(v, p)
- axpby(x, 1.0, x, lambda, p)
- axpby(r, 1.0, r, -lambda, v)
- dotProduct(r, r)
- axpby(p, 1.0, r, alpha_1 / alpha_0, p)

stitching all the time taken for each kernels we get the expected roofline performance for the CG solver.

$$P_{roofline(1,2)}^{CG} = \frac{2000 * 20000}{T_{1,2}^{dot(v,p)} + T_{1,2}^{dot(r,r)} + 3 * T_{1,2}^{axpby} + T_{1,2}^{applystencil}} = 732.14\frac{MLUP}{s}$$

$$P_{roofline(3)}^{CG} = \frac{1000 * 400000}{T_{3}^{dot(v,p)} + T_{3}^{dot(r,r)} + 3 * T_{3}^{axpby} + T_{3}^{applystencil}} = 732.14\frac{MLUP}{s}$$

4.7 PCG roofline performance

the PCG solver is composed of multiple kernels listed below

- pde− >applyStencil
- dotProduct(v, p)
- axpby(x, 1.0, x, lambda, p)
- axpby(r, 1.0, r, -lambda, v)
- dotProduct(r, r)
- pde− >GSPreCon(r, z) formed by both forward and backward substitution.
- dotProduct(r, z)
- axpby(p, 1.0, r, alpha_1 / alpha_0, p)

stitching all the time taken for each kernels we get the expected roofline performance for the CG solver.

$$P_{roofline(1,2)}^{PCG} = \frac{2000 * 20000}{2 * T_{1,2}^{dot(v,p)} + T_{1,2}^{dot(r,r)} + 3 * T_{1,2}^{axpby} + T_{1,2}^{applystencil} + 2 * T_{1,2}^{gsprecon}} = 465.91\frac{MLUP}{s}$$

$$P_{roofline(3)}^{PCG} = \frac{1000 * 400000}{2 * T_{3}^{dot(v,p)} + T_{3}^{dot(r,r)} + 3 * T_{3}^{axpby} + T_{3}^{applystencil} + 2 * T_{3}^{gsprecon}} = 465.91\frac{MLUP}{s}$$

Chapter 5

Experimental Results (1 ccNUMA Domain)

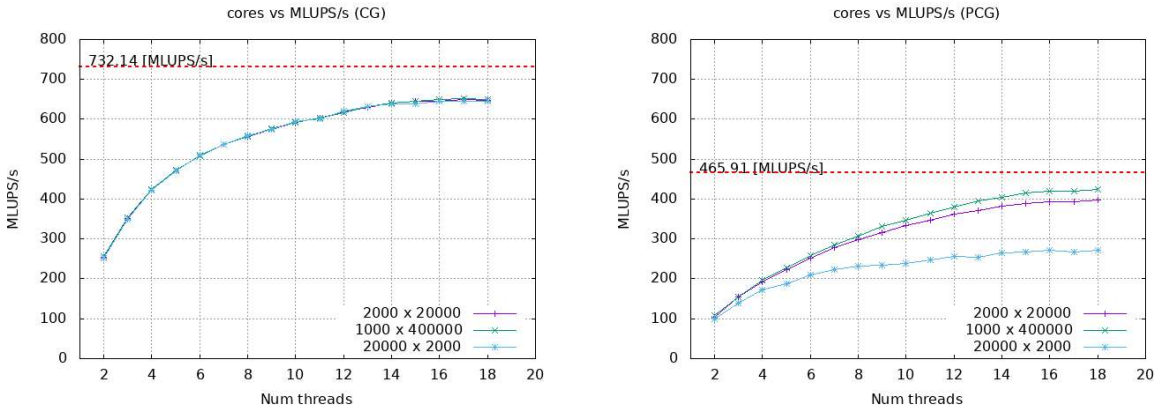

Figure 5.1: cores v/s performance ($\frac{MLUPS}{s}$) of CG/PCG solvers for 1 ccNUMA domains

It can be observed that in both the CG and PCG our measured performance is close to the predicted roof line performance.

- The performance of CG is irrespective of the grid size because the kernels used do not have any data dependency internal to the grid.
- the performance of PCG is dependent on grid dimensions as the GS_PRE_CON kernel has internal dependencies inside grid. this results in over head while synchronization. and wave front parallelization prefer having higher number of lattices in x direction as this would result in each thread would have more number of columns to work on and less barrier overhead.

Chapter 6

Code balance verification

The fully parallelized code with likwid markers was compiled with the command "LIKWID=on CXX=icpx make" to enable LIKWID measurements. and the obtained results are compiled in the following table.

Grid Size	2000X20000		20000X2000		1000X400000	
	Theoretical	LIKWID	Theoretical	LIKWID	Theoretical	LIKWID
APPLY_STENCIL	16.00	22.07	16.00	21.97	16.00	22.09
GS_PRE_CON	24.00	19.15	24.00	19.28	24.00	19.98

6.1 APPLY_STENCIL

We were instructed to neglect write allocates(WA) during our roofline calculations as ice lake has WA evasion enabled, but as per Dr. Georg Hager blog the WA evasion is only perfect after 29 cores. As all our simulation for code balance are under this mark we don't evade all WA. Resulting in a code balance between $16\frac{bytes}{LUP}$ and $24\frac{bytes}{LUP}$

6.2 GS_PRE_CON

The GS_PRE_CON implementation also has a lot of temporal locality like

- in forward substitution : (*rhs)(j, i) must be loaded from main memory. the elements (*x)(j - 1, i) and (*x)(j, i - 1) can be reused, a store of (*x)(j, i).
- in backward substitution (*x)(j, i) must be loaded from main memory. the elements (*x)(j + 1, i) and (*x)(j, i + 1)) can be reused from cache. store of (*x)(j, i).

using L2 cache(1.25 MB) value for computing layer condition we get that each core can hold upto 2 rows and $\approx$ 39000 columns. which is satisfied for all the given grid sizes when we do the wave front parallelization for more than 10 cores. so our measured code balance is closer to 16 $\frac{bytes}{LUP}$ than our predicted 24 $\frac{byte}{LUP}$

Chapter 7

Scaling to 4 ccNUMA Domains

According to NUMA first touch policy "A memory page gets mapped into the local memory of the processor that touches it first!" here to touch means writing to the array(not just allocate). we need the entire grid being distributed in different NUMA domains. To achieve this we can parallelize the Grid constructors this will result in placing the data in different NUMA domains. if we do not parallelize the arrayPtr initialization all the data that it points to will be placed in the first NUMA domain (given that enough memory is available) resulting the performance be saturated after 18 cores as we would be bound by memory bandwidth of single NUMA domain.

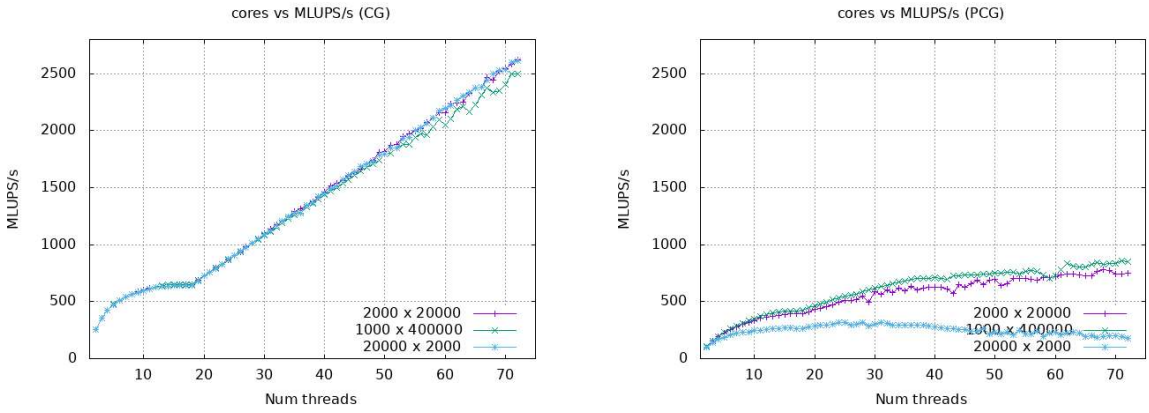

Figure 7.1: cores v/s performance ($\frac{MLUPS}{s}$) of CG/PCG solvers for 4 ccNUMA domains

the following trends are observed in CG performance scaling

- it saturates inside a single NUMA Domain.
- it has a linear scaling beyond a single NUMA domain up to 4 ccNUMA domains due to proper work sharing.
- at higher core counts we say a bit of drop in performance for grid size 1000 x 400000 this may be due to requirement to transfer sizeable amount of data between different domains for kernels like APPLY_STENCIL

the following trends are observed in PCG performance scaling

- it saturates inside a single NUMA Domain but at a lower value than CG.
- it has a improper scaling beyond a single NUMA domain update 4 ccNUMA domains due to improper work sharing.
- for grid size 20000 x 2000 we have a saturation at 36 core (i.e. 1 socket) beyond this we have a drop in performance. this can be explained the synchronization overhead of wave front parallelization as for wave front parallelization is preferred to have more elements in x direction.
- for other 2 grid sizes even after crossing socket boundary we see scaling but not a perfect scaling. this is a direct result of the array's dependency, which causes at least one of the threads to write to it, multiple threads frequently access the same cache line. The Cache Coherence Protocol's guarantee of cache line copy consistency may have contributed to the performance decline. The iteration spaces in the cache line that are allotted to successive threads may cross over.