Assignment I: Running on Dardel & Performance Modeling & Benchmarking

DD2356 VT24 Methods in High Performance Computing

Long Ma, Yining Hou

April 7, 2024

1 Compiling and Running on Dardel

1.1 Connect to Dardel

1.1.1 Configure SSH

Before connecting to **Dardel**, the local environment must be configured. To configure **Kerberos** on our local machine, we did the following steps:

1) Create *.ssh* folder if it does not already exist in home folder.

2) Create a file called *krb5.conf* in *.ssh*.

3) In *.bash_profile* or *.profile*, add two lines. The first one points **Kerberos** to the right configuration and the second creates a **Kerberos** cache in a fixed location instead of */tmp*.

4) For the Windows WSL system, we create a file in *.ssh* called *config*. For the Mac system, we create a file in *.ssh* called *config* without **GSSAPI** related configurations.

5) Set the correct permission on the file.

1.1.2 Obtain a Kerberos ticket to log into Dardel

We created a **Kerberos** ticket using *kinit*. This is the time we need to have to provide the password provided by **PDC**:

```
$ kinit --forwardable YourUsername@NADA.KTH.SE
```

There is another way to create a **Kerberos** ticket, we can use *ktutil* to create a file which can be used as a password.

```
$ ktutil
$ add_entry -password -p YourUsername@NADA.KTH.SE -k 3 -e aes256-cts-hmac-sha1
   -96
$ write_kt YourUsername.keytab
$ kinit -kt YourUsername.keytab YourUsername@NADA.KTH.SE
```

1.1.3 Use ssh from the command line to access Dardel

Use your **Kerberos** ticket and SSH to connect to **Dardel**:

```
$ ssh YourUsername@dardel.pdc.kth.se
```

1.2 Compile the code

```
cc mpi_hello_world.c
```

1.3 Execute the code

Since each compute node has a total of 256 logical/virtual cores, to run on 128 cores of Dardel, we assign 128 tasks per node and 2 cpus per task.

1.3.1 Interactive mode

```
longm@uan01:~/Private> export OMP_NUM_THREADS=1
longm@uan01:~/Private> OMP_PLACES=cores
longm@uan01:~/Private> salloc --nodes=1 -t 01:00:00 -A edu24.dd2356 -p main --
   ntasks-per-node=128 --cpus-per-task=2
salloc: Pending job allocation 3454935
salloc: job 3454935 queued and waiting for resources
salloc: job 3454935 has been allocated resources
salloc: Granted job allocation 3454935
salloc: Waiting for resource configuration
salloc: Nodes nid001030 are ready for job
```

-n defines the number of running tasks. We assign 128 to *-n*.

```
longm@uan01:~/Private> srun -n 128 ./a.out
Hello world from processor nid001030, rank 26 out of 128 processors
Hello world from processor nid001030, rank 30 out of 128 processors
Hello world from processor nid001030, rank 96 out of 128 processors
...
Hello world from processor nid001030, rank 125 out of 128 processors
Hello world from processor nid001030, rank 127 out of 128 processors
Hello world from processor nid001030, rank 41 out of 128 processors
```

After finishing the job, we can exit using the following line

```
longm@uan01:~/Private> exit
exit
salloc: Relinquishing job allocation 3455073
salloc: Job allocation 3455073 has been revoked.
```

1.3.2 Batch mode

Similarly, we create a file named *mpi_hello_world_job.sh* with following content:

```
#!/bin/bash -l
# The -l above is required to get the full environment with modules
# The name of the script is mpi_hello_world_job
#SBATCH -J mpi_hello_world_job
# Only 1 hour wall-clock time will be given to this job
#SBATCH -t 1:00:00
#SBATCH -A edu24.DD2356
# Number of nodes
#SBATCH -p main
#SBATCH --ntasks-per-node=128
#SBATCH --cpus-per-task=2
#SBATCH --nodes=1
#SBATCH -e error_file.e
# Run the executable file
# and write the output into my_output_file
srun -n 128 ./a.out > mpi_hello_world_output
```

We can now submit the job.

```
sbatch ./mpi_hello_world_job.sh
```

By using squeue, we can monitor the job in the queue:

```
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
---------------------------------------------------------------------
3454956      main mpi_hell yiningho  R       0:01      1 nid001020
```

After the job has finished, we will find a file named *mpi_hello_world_output* in the same directory as we submitted the job that contains execution results.

2 Sustainability and Supercomputers

We used the following command to complete the search for memory and CPU models for one node

```
longm@uan01:~> free --gibi
               total        used        free      shared  buff/cache   available
Mem:             503         266         115          58         121         165
Swap:              9           9           0
longm@uan01:~> lscpu
...
Core(s) per socket:              64
Socket(s):                       2
...
Model name:                      AMD EPYC 7742 64-Core Processor
...
```

Based on these, we know that there are two CPUs in each node, and each CPU has 64 cores. So 10 nodes should have $2 * 64 * 10 = 1280$ cores.

We can also know that the total memory of each core is $503 + 9 = 512$ GB. The total memory of 10 cores is $512 * 10 = 5120$ GB.

By searching on the official AMD website, we found the CPU default TDP of AMD EPYC 7742 64-Core is 225W. Since it has 64 cores, we took $225/64 = 3.515625$ as the TDP of each core.

Fill in the above values into the corresponding positions in `https://calculator.green-algorithms.org`, it can help us evaluate the power usage and carbon footprint of simulating 10 **Dardel** computing nodes.

Long Ma, Yining Hou
longm@kth.se, yiningho@kth.se

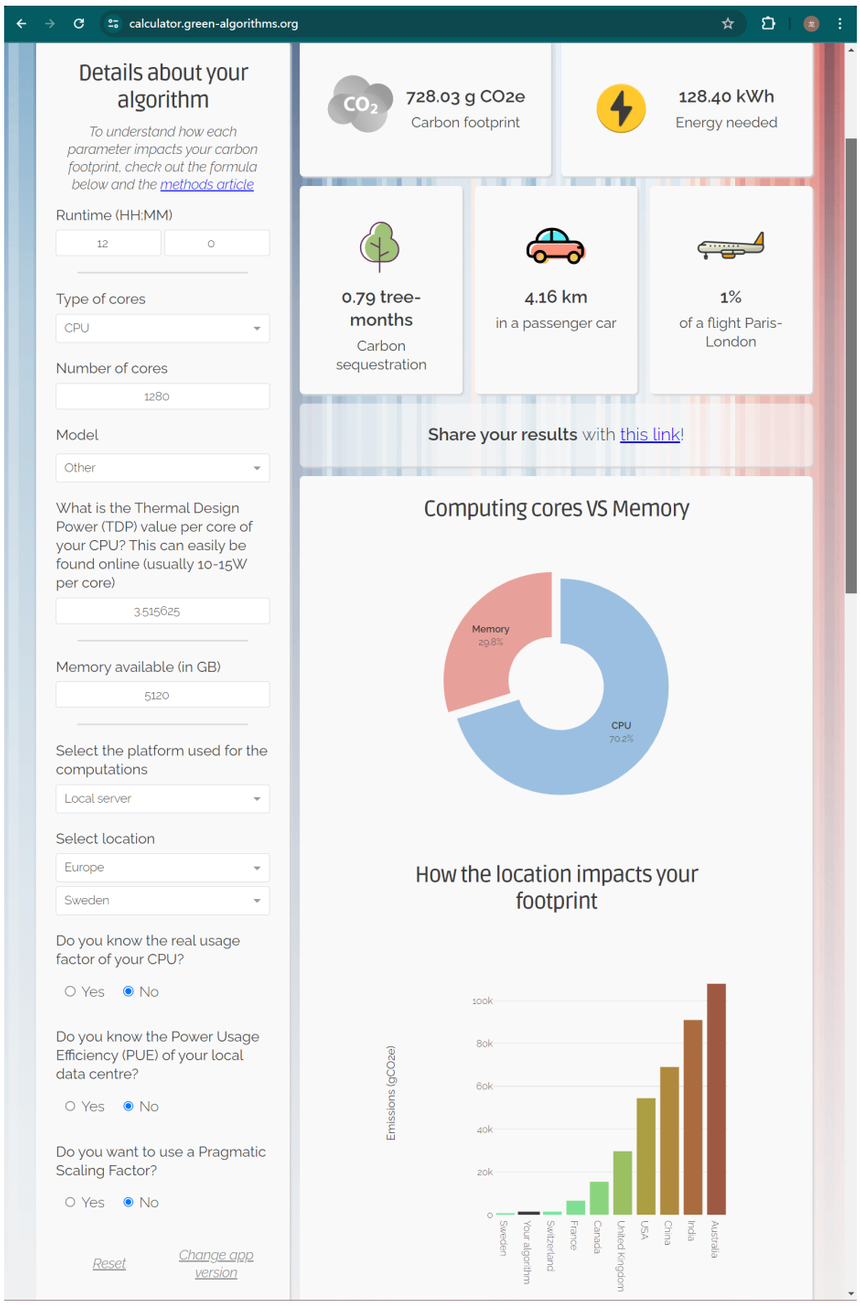

Figure 1: Evaluate result of 10 Dardel computing nodes

3 Modeling Sparse Matrix-Vector Multiply

3.1 Question 1

Use $Time = nnz \times 2c$ and calculate c from the given clock speed of the processor in use.

$$c = \frac{1}{\text{clock speed in Hz}} = \frac{1}{2.25 \times 10^9} \text{ seconds} \tag{1}$$

Calculated Performance			
n	nrows	nnz	Time(s)
10	10^2	460	0.409×10^{-6}
100	10^4	49600	44.09×10^{-6}
1000	10^6	4996000	4.44×10^{-3}
10000	10^8	499960000	0.444

3.2 Question 2

We use $nnz \times 2$ and the total execution time to calculate the floating-point operations per second in SpMV.

$$\text{floating point operations} = \frac{nnz \times 2}{\text{total execution time}} \tag{2}$$

Measured Performance				
n	nrows	nnz	Time(s)	floating-point operations per second
10	10^2	460	10^{-6}	9.2×10^8
100	10^4	49600	48×10^{-6}	2.07×10^9
1000	10^6	4996000	5.062×10^{-3}	1.97×10^9
10000	10^8	499960000	0.444	2.25×10^9

Comparing the results from the performance model and experimental results, we find that the execution time is longer than calculated.

3.3 Question 3

Since the simple performance model of sparse matrix-vector multiply is:

$$Time = nnz(2c + 2.5r) + nrows(0.5r + w) \tag{3}$$

We used a simplified model which omits the time for stores and loads. Therefore, the main reason for the observed difference between the modeled value and the measured value can be the data fetching and storing time.

3.4 Question 4

The total bytes read can be calculated as $nnz(sizeof(double) + sizeof(int)) + nrows(sizeof(double) + sizeof(int))$ and we use the total execution time to calculate the bandwidth of SpMV:

$$\text{read bandwidth} = \frac{\text{Total Bytes Read}}{\text{total execution time}} = \frac{nnz \times 12 + nrows \times 12}{\text{total execution time}} \tag{4}$$

Read Bandwidth Values				
n	nrows	nnz	Time(s)	read bandwidth values(GB/s)
10	10^2	460	10^{-6}	6.72
100	10^4	49600	48×10^{-6}	14.9
1000	10^6	4996000	5.062×10^{-3}	14.21
10000	10^8	499960000	0.444	16.22

3.5 Question 5

The bandwidth obtained by running the STREAM benchmark:

```
Array size = 10000000 (elements), Offset = 0 (elements)
Memory per array = 76.3 MiB (= 0.1 GiB).
Total memory required = 228.9 MiB (= 0.2 GiB).
Each kernel will be executed 10 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 5005 microseconds.
   (= 5005 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           31664.1     0.005078     0.005053     0.005088
Scale:          18574.3     0.008639     0.008614     0.008668
Add:            22957.9     0.010500     0.010454     0.010553
Triad:          21798.5     0.011040     0.011010     0.011064
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
```

The bandwidth measured by the STREAM benchmark is significantly higher than that measured for the SpMV. The STREAM benchmark operates on large arrays where consecutive elements are accessed sequentially. This access pattern is very cache-friendly. On the other hand, SpMV operations on sparse matrices often access memory in an irregular pattern due to the sparsity of the data, which can result in more cache misses and less efficient use of the memory bandwidth.

4 The Memory Mountain

4.1 Question 1

We can use *lscpu* to check the name of the processor and the size of the L1, L2, and L3 of the processor.

```
longm@uan01:~/Private/DD2356/Assignment-I/memory-mountain-example> lscpu
...
Model name:                      AMD EPYC 7742 64-Core Processor
...
L1d cache:                       4 MiB
L1i cache:                       4 MiB
L2 cache:                        64 MiB
L3 cache:                        512 MiB
```

4.2 Question 2

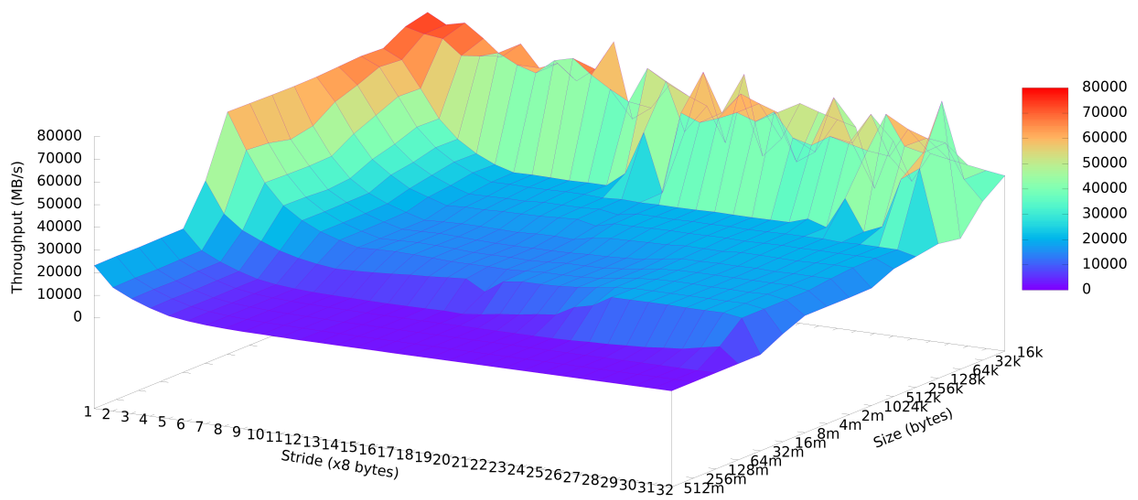

Figure 2: Memory mountain in shared partition

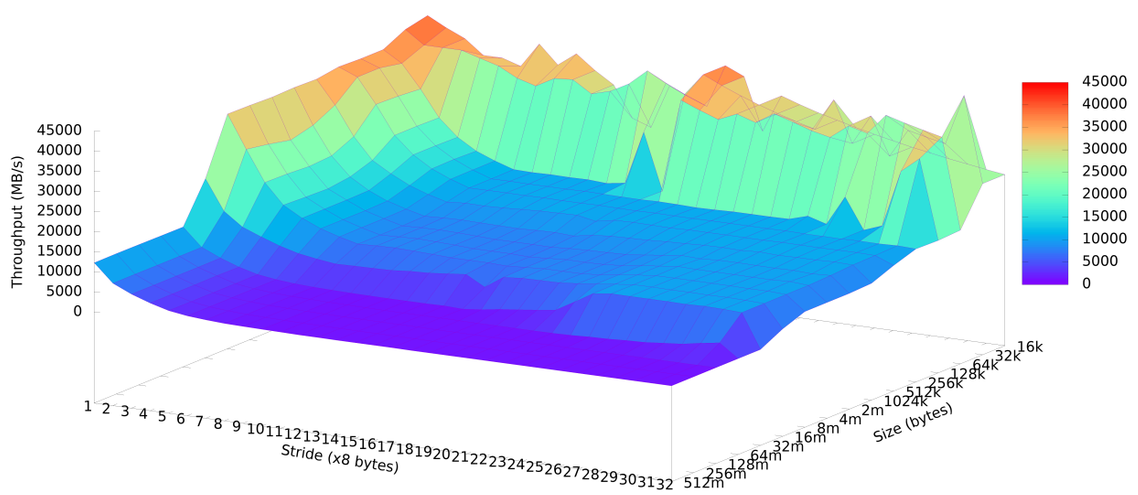

Figure 3: Memory mountain in main partition

4.3 Question 3

The most consistently **high** performance was obtained with an array size of **1** and a stride of **16k**. The reported read bandwidth in a shared partition is **74877**, and in a main partition is **39972**.

4.4 Question 4

The most consistently **low** performance was obtained with an array size of **32** and a stride of **512m**. The reported read bandwidth in a shared partition is **2330**, and in a main partition is **1247**.

4.5 Question 5

Hardware data prefetching is a technique that can improve overall processor performance by reducing the average memory access latency of processor cores by preloading the data they need into caches.

When the stride is 1, the program sequentially accesses each element in the array. Even if the size of the array exceeds the capacity of the L3 cache, this predictable access pattern enables hardware to efficiently prefetch data. The processor can predict which elements will be accessed next and preload them into the cache in advance. Because the data may already be in the cache, subsequent accesses are faster.

On the other hand, when the stride is 32, the program skips many elements in the array. This access pattern is more difficult to predict for prefetching. The processor may not accurately predict which elements will be accessed next, leading to cache misses and the need to load data from the main memory, which reduces performance.

4.6 Question 6

"Temporal locality" means that when a program is running, a recently referenced memory location is likely to be referenced again. If a program accesses the same data multiple times within a short time frame, it exhibits temporal locality.

"Spatial locality" means that recently referenced memory locations and their surrounding memory locations are easy to use again. If a program frequently accesses adjacent memory locations within a short time frame, it exhibits spatial locality.

4.7 Question 7

Adjusting the total array size impacts temporal locality because larger array sizes may lead to a more slack distribution of data in memory, reducing the probability of accessing the same memory locations repeatedly. As the array size increases, the data becomes more slack, decreasing the possibilities of repeated access to the same data within a short time. Consequently, temporal locality may weaken with increasing array size. **Therefore, increasing the array size may decrease temporal locality.**

4.8 Question 8

Adjusting the read stride impacts spatial locality because a larger read stride means the program skips more elements when accessing the array, resulting in more slack memory accesses. A larger read stride increases the distance between memory locations accessed by the program within a shorter time, reducing the frequency of accessing adjacent memory locations and hence decreasing spatial locality. **Therefore, increasing the read stride may decrease spatial locality.**

5 Measure the Performance of Matrix-Matrix Multiply and Transpose with perf

5.1 Task 5.1

Because we want *srun* to allocate resources and run the job within the Slurm system, and then *perf* to analyze the performance of that job, we use *perf* after *srun*.

```
yiningho@uan01:~/Private/HPC/Assignment1/Exercise5> srun -n 1 perf stat -e
    cycles,instructions,L1-dcache-load-misses,L1-dcache-loads ./matrix_multiply.
    out

1. Initializing Matrices
2. Matrix Multiply
3. Sum = 249.968991
4. time = 0.221695

 Performance counter stats for './matrix_multiply.out':

     8,315,058,096      cycles:u
    19,416,611,350      instructions:u            #    2.34  insn per cycle
     3,148,468,159      L1-dcache-load-misses:u   #   56.39% of all L1-dcache
        hits
     5,583,859,814      L1-dcache-loads:u
       2.459406713 seconds time elapsed

       2.454859000 seconds user
       0.004004000 seconds sys
```

We use these formulas to calculate results:

$$\text{Instructions per cycle} = \frac{\text{Instructions}}{\text{cycles}} \tag{5}$$

$$\text{L1 cache miss ratio} = \frac{\text{L1-dcache-load-misses}}{\text{L1-dcache-loads}} \tag{6}$$

$$\text{L1 cache miss rate PTI} = \frac{\text{L1-dcache-load-misses}}{\text{instructions}} \times 1000 \tag{7}$$

EVENT NAME	MSIZE = 64 Naive	MSIZE = 64 Optimized	MSIZE = 1000 Naive	MSIZE = 1000 Optimized
Elapsed time (seconds)	0.008419816	0.006441120	2.459406713	1.618644463
Instructions per cycle	1.24	1.42	2.34	2.00
L1 cache miss ratio	17.97%	5.21%	56.39%	16.51%
L1 cache miss rate PTI	100.71	22.66	162.15	145.91

- For same implementation of the matrix multiply operation, when the matrix size gets bigger, the elapsed time gets longer and cache miss ratio gets higher. This is because larger matrices are more likely to exceed the cache size, resulting in increased cache misses, which slows down the computation.

- For same matrix size, the optimized implementation results in a higher performance than the naive one. The way the matrix elements are accessed can significantly affect the cache hit rate. In the naive implementation, there's a higher likelihood of cache misses, as the innermost loop accesses discontinuous memory when reading *matrix_b*. In the optimized version, we take advantage of spatial locality, leading to fewer cache misses by changing the loop nesting order to access the memory in a more continuous manner.

5.2 Task 5.2

EVENT NAME	N = 64	N = 128	N = 2048
Elapsed time (seconds)	0.079671140	0.076118977	28.190939598
Bandwidth/Rate (MB/s)	1.83×10^4	6.94×10^3	1.20×10^2
Instructions per cycle	1.53	1.15	0.02
L1 cache miss ratio	5.89%	18.90%	23.30%
L1 cache miss rate PTI	28.33	148.82a	1984.12

- The size of the matrix impacts the spatial locality of the data. When accessing larger matrices, the cache may not be sufficient to store the required arrays, leading to cache misses and more frequent accesses to main memory, slowing down the transpose operation.

- One way to improve the re-usage of the cache is by changing the order of loops to utilize spatial locality during memory accesses better. For *transposeBase()*, we can iterate over j first and then over i in the loop, as shown in the following code. This allows the elements in one row of matrix a to be accessed in adjacent memory locations during each iteration of the loop, thus enhancing cache locality. This approach can significantly improve locality and, consequently, the performance of the transpose operation, especially when dealing with larger matrices.

```
void transposeOptimized() {
    int i, j;
    for (j = 0; j < N; ++j)
        for (i = 0; i < N; ++i)
            b[i][j] = a[j][i];
}
```

- Another way to improve the re-usage of the cache is by splitting the loop into smaller blocks. For *transpose-Base()*, we can divide the loop into smaller iterations, as shown in the following code. This allows for data reuse in the cache by blocking during each iteration of the inner loop, thus enhancing spatial locality. This blocking technique improves spatial locality by maximizing data reuse within the cache, thereby enhancing the performance of the transpose operation. Adjusting the block size can further optimize cache utilization based on the cache size of the system and the size of the matrices.

```
void transposeOptimized() {
    int i, j, ii, jj;
    for (jj = 0; jj < N; jj+=STRIP)
        for (ii = 0; ii < N; ii+=STRIP)
            for (j = jj; j < jj + STRIP; ++j)
                for (i = ii; i < ii + STRIP; ++i)
                    b[i][j] = a[j][i];
}
```