HPC | HW1 | Alireza Rafiei

1. Memory hierarchy, latency, and bandwidth (10 pts)

1.1 Define spatial locality and temporal locality.

Spatial locality is a concept where if a memory location is accessed, the locations adjacent to it are likely to be accessed soon. This is based on the observation that data is often stored in contiguous blocks, and program instructions typically execute in sequence.

Temporal locality implies that if a specific memory location is accessed, it's likely to be accessed again in the near future. This behavior is observed in operations like loops, where the same set of memory locations is accessed repeatedly. Utilizing temporal locality can optimize cache memory performance in computer systems.

1.2 What is the peak flops of a quad core processor running at 2 Ghz and has ILP of 6 operations per cycle?

Peak FLOPS = Clock Rate × ILP × Number of cores

Peak FLOPS = $2 \times 10^9 \times 6 \times 4 =$ **48×10^9 = 48 GFLOPS**

1.3 Consider a memory system with a level 1 cache of 32 KB and DRAM of 512 MB with the processor operating at 2 GHz. The latency to L1 cache is one cycle and the latency to DRAM is 100 cycles. In each memory cycle, the processor fetches four words (cache line size is four words). What is the peak achievable performance of a dot product of two vectors of 4K? Note: Where necessary, assume an optimal cache placement policy.

```
/* dot product loop */
for (i = 0; i < dim; i++)
dot_prod += a[i] * b[i];
```

Cache line size = 4 words
Vector dimension = 4K = 4096 bytes
Latency to L1 cache = 1 cycle
Latency to DRAM = 100 cycles
Number of ops/cycle = 6

The processor operates at 2 GHz, meaning it processes 2 billion cycles in a second, so 50ns latency for every cycle. Now, considering that both vectors a and b are not present in the cache initially, we need to account for cache misses when accessing them. For the first access of each vector, there'll be a cache miss, leading to a penalty of 100 cycles to fetch data from DRAM. Given the time complexity of a dot product is O(n) -> 2n (* and +), the total number of cycles for 4000 iterations is 8000. Thus:

Two cache misses (one for a and one for b) and 8 cache hits (4 for each), which is 2 × 104 cycles = 208 cycles × 0.5 ns = 104 ns × 4k/4 iterations ≈ 104 us.

Given that our dot product operation takes 8000 ops, the time taken is:

8000/6 ≈ 1333 cycles × 0.5 ns ≈ 0.67 us.
Total time of read and ops = 104 + 0.67 = 104.67 us

Based on the class discussion, we should consider the write latency approximately two times more than the read; Thus:

Writing latency = 104 × 2 = 208 us
Total taken time = 104.67 + 208 = 312.67

Now, the peak performance is determined by the number of operations over the time taken:

Performance = 8k/312.67us ≈ 25.59 × 10^6 operations/second or **25.59 MFLOPS**.

1.4 Now consider the problem of multiplying a dense matrix with a vector using a two-loop dot-product formulation. The matrix is of dimension 4K x 4K. (Each row of the matrix takes 16 KB of storage.) What is the peak achievable performance of this technique using a two-loop dot-product based matrix-vector product?

```
/* matrix-vector product loop */
for (i = 0; i < dim; i++)
for (j = 0; i < dim; j++)
c[i] += a[i][j] * b[j];
```

We assume that this calculation is independent and we already don't have either a or b in the cache. So:

For the matrix a, after each miss, 4 elements are loaded. So the cache miss + 4 cache hits is:

4K/4 interations × 4K ×104 cycles = 416M cycles × 0.5 ns ≈ 208 ms.

For the vector b, every first access will be a cache miss (one word per each iteration) and a cache hit. Thus:

4K × 4K × 101 cycles = 1616M cycles × 0.5 ns ≈ 808 ms.

Given the time complexity of a two-loop dot product is O(n^2) -> $2n^2$, the total number of cycles for 4000 iterations is 2 × 4000^2 = 32M ops. Thus:

32M/6 ≈ 5,333,333 cycles × 0.5 ns ≈ 2,666,667 ns ≈ 2.67 ms.

Total time = 208 + 808 + 2.67 = 1018.67 ms

Based on the class discussion, we should consider the write latency approximately two times more than the read; Thus:

Writing latency = 1016 × 2 ≈ 2032us

Total taken time = 1018.67 + 2032 = 3050.67

Now, the peak performance is determined by the number of operations over the time taken:

Performance = 32M /3050.67 ms ≈ 10.49×10^6 operations/second or **10.49 MFLOPS**.

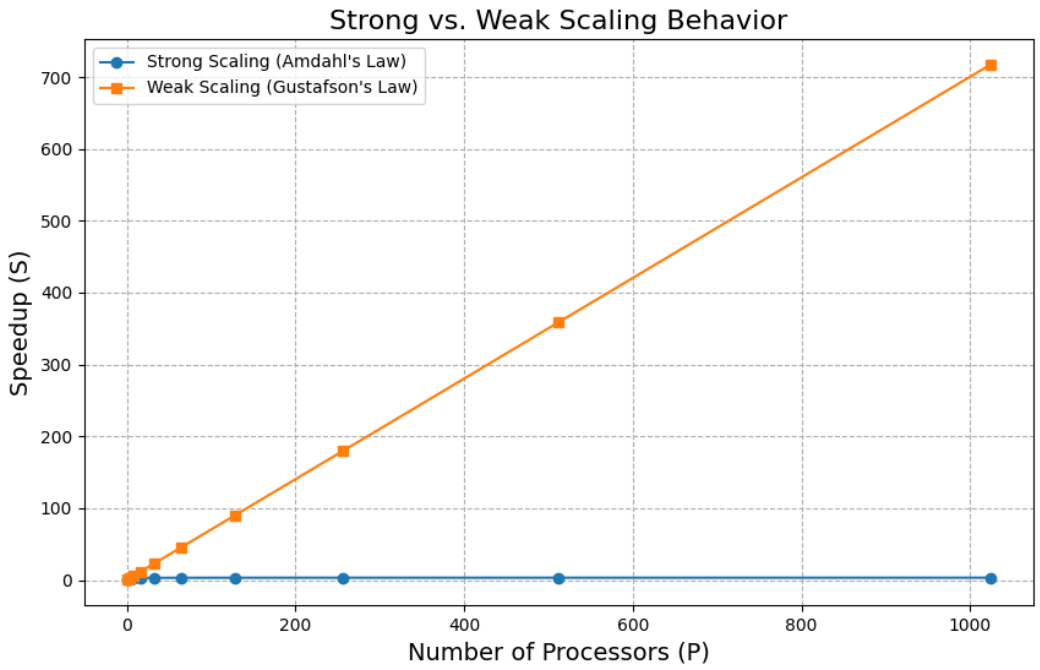
2. Amdalh's law and Gustafson's law (10 pts)

2.1 Suppose 30% of a program cannot be parallelized, and the rest are perfectly parallelizable. Plot the strong and weak scaling behavior for core counts P 1 to 1,000 (use nearest powers of 2).

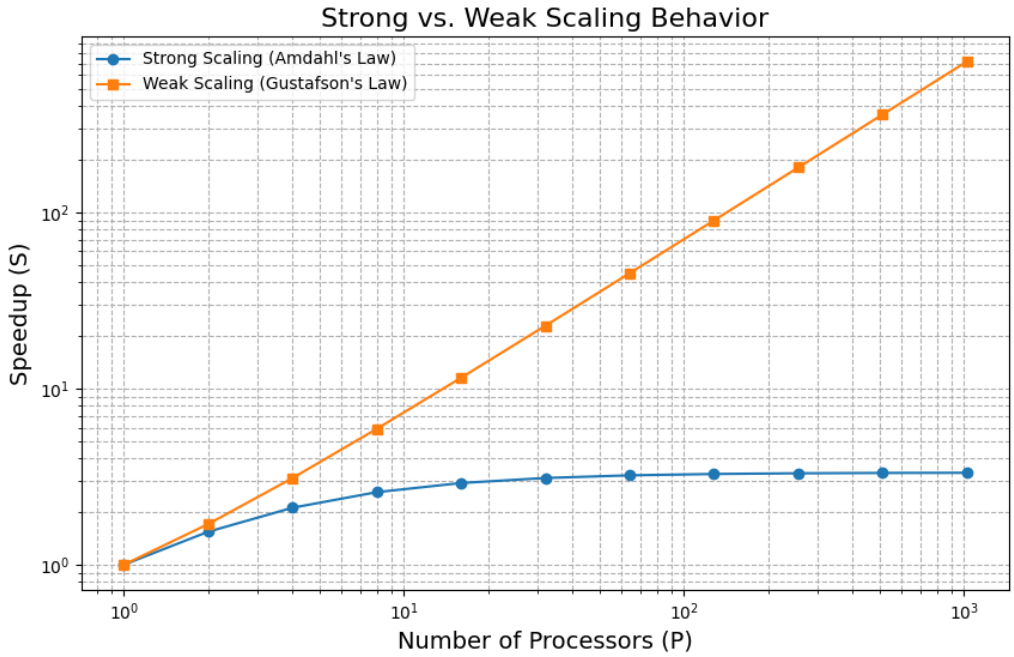
Amdahl's Law (Strong Scaling): Given a fraction f of the program that cannot be parallelized, the speedup $S(P)$ from using P processors is: $S(P) = \frac{1}{f+\frac{1-f}{P}}$

Gustafson's Law (Weak Scaling): As the number of processors increases, the total problem size also increases such that the workload per processor remains constant. The speedup $S(P)$ from using P processors is: $S(P) = P - f \times (P-1)$

Where f is 30%:



On the log scale with a base of 10, we will have:



Refer to 2.1.py script.

2.2 Now suppose the program is 30% sequential globally, 30% that is parallelized by node (sequential within the node, each node containing 4 cores), and the remaining 40% is parallelizable by core. Formulate the performance expressions and plot the strong and weak scaling behavior for core counts P 1 to 1,000 (use nearest powers of 2). Assume data size increases with number of cores P for weak scaling.

Given this scenario, the performance of the program can be described in as:

f_1 = 30%: This fraction of the program is purely sequential.

f_2 = 30%: This fraction is parallelizable across nodes but is sequential within a node (each node contains 4 cores).

f_3 = 40%: This fraction is parallelizable across individual cores.

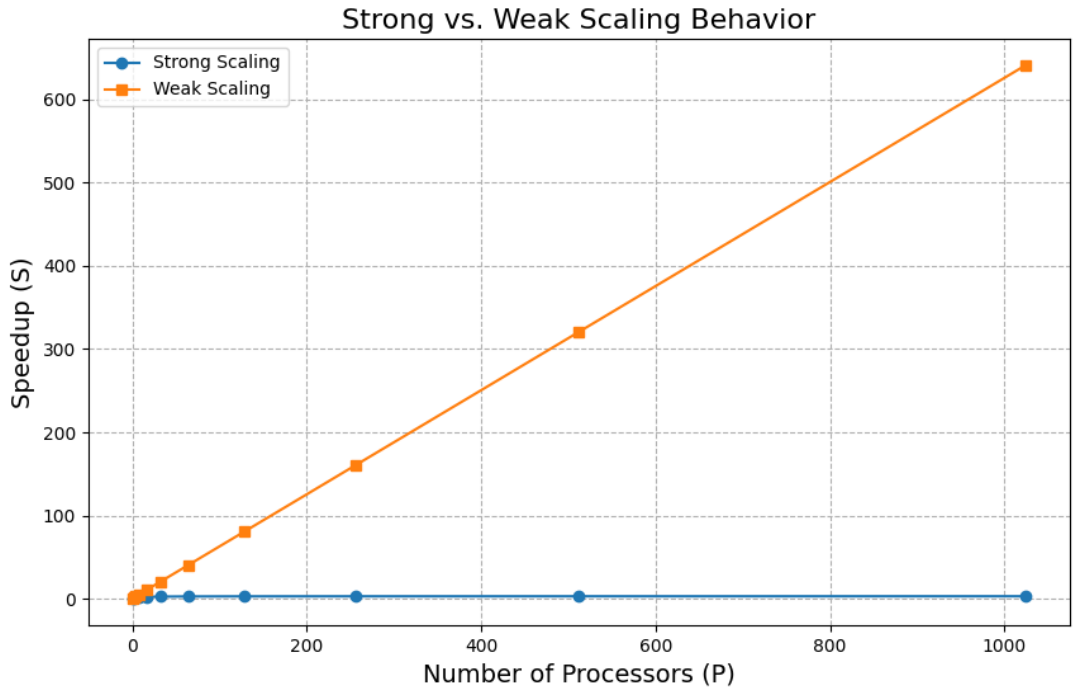
The speedup for the program for strong scaling considering the above constraints can be described as:

$$S(P) = \frac{1}{f_1 + \frac{f_2}{P_{node}} + \frac{f_3}{P}}$$

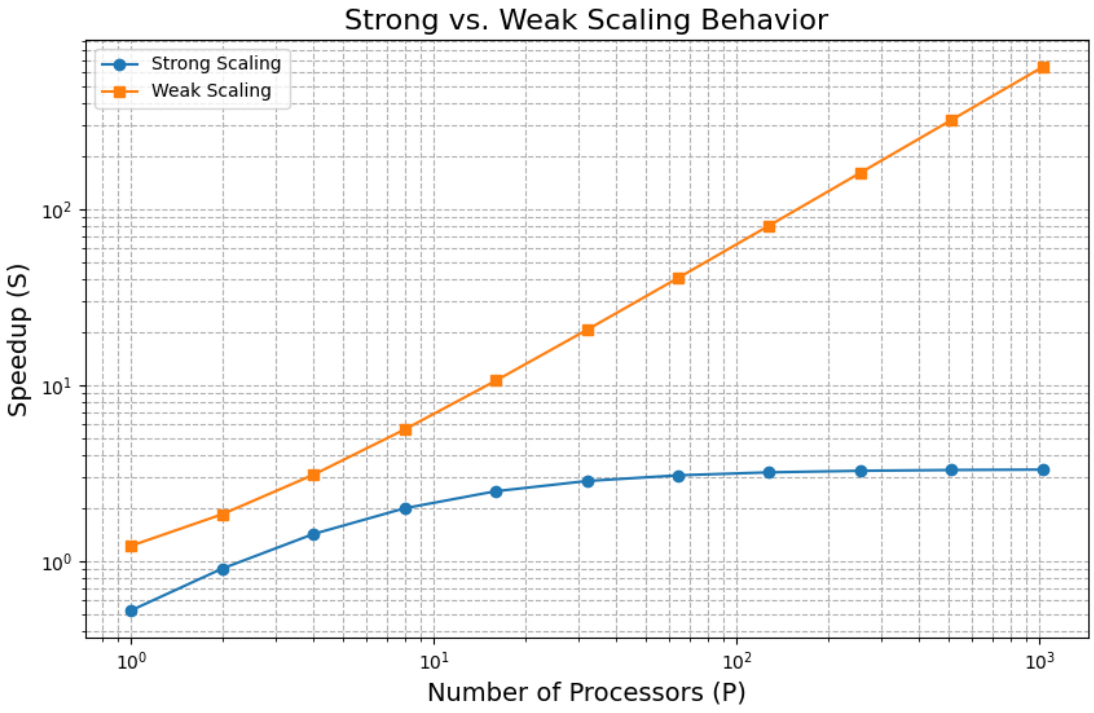
For weak scaling, assuming that the data size increases proportionally with the number of cores P:

The speedup can be directly related to the number of cores used, but considering the parts of the program that are not perfectly parallelizable. This can be expressed as:

$$S(P) = P - f_1 \times (P - 1) - f_2 \times (\frac{P}{4} - 1)$$



On the log scale with a base of 10, we will have:



Refer to 2.2.py script.

3 SIMD Simple matrix multiplication (C[N] = A[N][N] x B[N]) (10points, 5 points for write-up)

3.1 Starting with the simple matrix multiplication code from HW 0, re-implement using SIMD instructions (c/c++). Describe strategies for computing vector dot products using SIMD instructions.

In the updated script, two main strategies have been employed to optimize the matrix multiplication operation. Firstly, SIMD instructions are utilized to parallelize the matrix multiplication at a lower level. This means that for each cycle, instead of processing a single data point, multiple data points are processed concurrently. The function matrix_multiply is modified to exploit this parallelism, where the innermost loop increments in steps. Matrix A's row is directly loaded, whereas matrix B's column is stored temporarily and then loaded, ensuring aligned memory access. The products of these vectors are accumulated into a result vector. After the k-loop, the results stored in this vector are summed up to obtain the final value for the matrix multiplication for that particular i,j position in the result matrix.

Secondly, the random number generation for matrix initialization is optimized by introducing the xorshift128plus random number generator, which is a faster alternative to the conventional rand() function from the C library. The xorshift128plus is an example of xorshift RNGs, known for their speed and decent randomness. Instead of using rand(), values are generated using xorshift128plus, offering a more efficient way to initialize matrix values. This may not significantly boost matrix multiplication itself, but for larger matrices, the time saved during initialization can be noticeable, especially when benchmarking.

Comparison between the time taken for matrix multiplication of the two approaches for different matrix sizes:

matrix size	Naïve approach (s)	SIMD approach (s)	Speed up
2 x 2	0.00	0.00	N/A
4 x 4	0.00	0.00	N/A
8 x 8	0.00	0.00	N/A
16 x16	0.00	0.00	N/A
32 x 32	0.00	0.00	N/A
64 x 64	0.00	0.00	N/A
128 x 128	0.01	0.01	~1
256 x 256	0.17	0.08	2.13
512 x 512	0.98	0.57	1.72
1024 x 1024	12.15	5.83	2.08
2048 x 2048	139.66	52.12	2.68
4096 x 4096	1915.10	854.7	2.24

Refer to 3.c script.

~~3.2 What is the speedup? Plot the strong scaling (as a function of number of threads) and the weak scaling (as a function of the number of points). Compare to your sequential code.~~

3.3 Describe the memory access pattern and any spatial or temporal locality.

Memory Access Pattern: Both Naïve and SIMD versions access matrices in a row-major order. This means that the elements of a particular row are accessed sequentially before moving on to the next row. Specifically, during the multiplication process, each row of matrix A is accessed sequentially and combined with the relevant columns of matrix B (8 columns for SIMD version) to produce the resulting matrix C.

Spatial Locality: The row-major order access ensures that sequential elements of matrix rows, which are contiguous in memory, are accessed in order. This takes advantage of cache lines, where fetching one element from memory often brings nearby elements into the cache. The SIMD optimized version further exploits spatial locality. 8 contiguous float values are loaded, processed, and stored at once. This benefits from the contiguous memory layout of matrix rows.

Temporal Locality: In the Naïve approach, elements of matrix A's current row and matrix B's current column are accessed multiple times during their respective inner loop operations. In the SIMD optimized version, while instructions are used to minimize the repeated loading of data, there's still temporal locality as matrix A's broadcasted value is used repeatedly against chunks of matrix B's current column.

4 SIMD Compute π (10 points for code, 5 points for write-up)

4.1 In homework 0, you computed PI using a Monte Carlo approach. Parallelize it using SIMD. Compare the error with the value of $\pi = \cos^{-1}(-1.0)$.

For n = 2000
Estimated value of Pi: 3.142000
Percentage error: 0.012966%

Comparison between the time taken for the estimation PI using the two approaches for different n sizes:

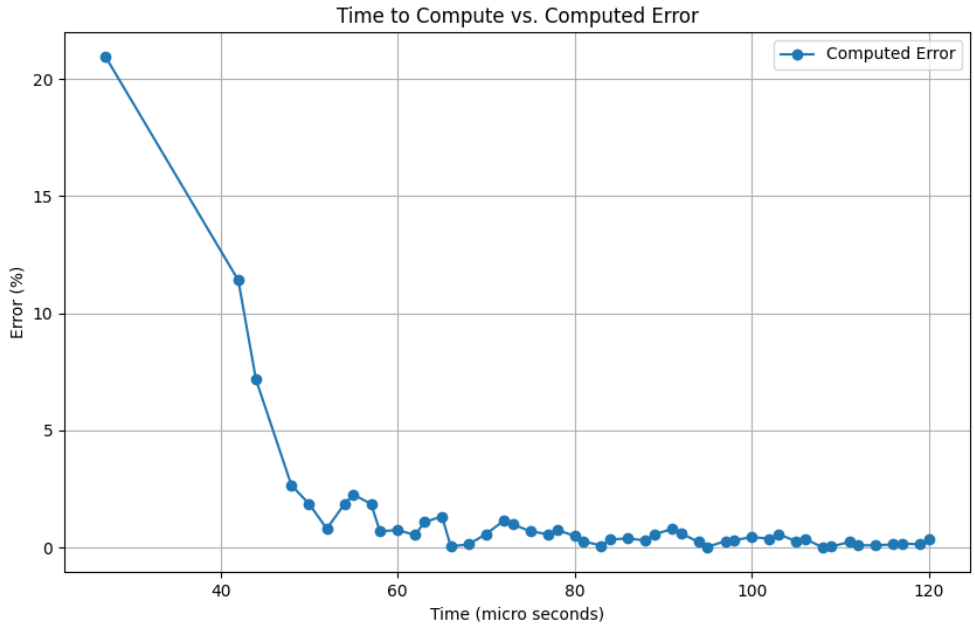
n size	Naïve approach (s)	SIMD approach (s)	Speed up
2000	0.00	0.00	N/A
20000	0.00	0.00	N/A
200000	0.01	0.00	N/A
2000000	0.13	0.04	3.25
20000000	1.29	0.42	3.07
200000000	13.64	3.89	3.51
2000000000	144.76	39.2	3.69

Refer 4.c script.

~~4.2 What is the speedup? Plot the strong scaling (as a function of number of threads) and the weak scaling (as a function of the number of points).~~

4.3 Periodically save the computed error and plot the time to compute vs. computed error.

The computed error periodically saved and the time to compute vs. computed error was plotted for n=2000.



refer to 4.c and 4.3.py scripts.