CPU Specifications

The experiments were conducted on a system equipped with an **Intel Core Ultra 9 185H** processor. This high-performance mobile CPU has the following key specifications:

- **Cores and Threads**: The CPU features **16 cores** (8 Performance cores and 8 Efficiency cores) and supports up to **24 threads** due to hyper-threading on the Performance cores.
- **Base Frequencies**: The **P-core base frequency** is **2.6 GHz**, while the **E-core base frequency** is **2.0 GHz**.
- **Maximum Turbo Frequency**: The processor can boost up to **5.2 GHz** for demanding, single-threaded tasks.

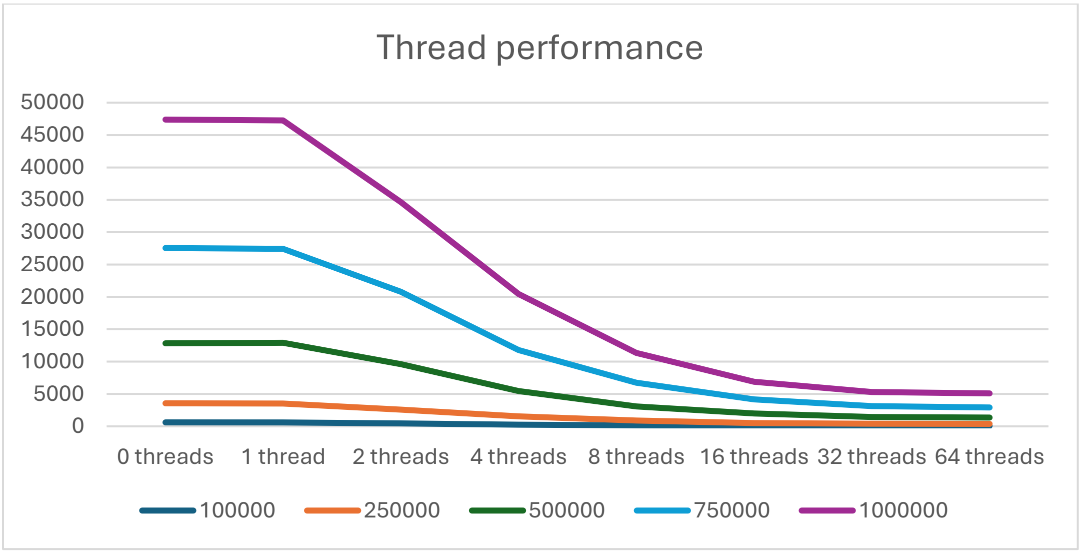
This multi-core architecture makes the processor well-suited for parallel computing tasks, where the P-cores handle high-performance workloads and the E-cores optimize efficiency.

Laboratory 1 experiment Results and Discussion

The goal of the experiment was to measure the execution time for finding prime numbers using various thread counts (0, 1, 2, 4, 8, 16, 32, and 64 threads) and analyze how the performance scaled with increasing threads. This chart based on the table with values illustrates the relationship between thread count and execution time across different input sizes (100,000 to 1,000,000):

Execution Time (clock cycles)

	0 threads	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads	64 threads
100000	**611**	**608**	**446**	**253**	**141**	**98**	**75**	**71**
250000	**3556**	**3541**	**2575**	**1547**	**877**	**526**	**421**	**381**
500000	**12804**	**12910**	**9594**	**5448**	**3084**	**2004**	**1434**	**1355**
750000	**27543**	**27432**	**20813**	**11780**	**6731**	**4185**	**3137**	**2916**
1000000	**47388**	**47252**	**34666**	**20424**	**11354**	**6890**	**5308**	**5086**



No Threads vs. 1 Thread

The execution time for **no threads** (single-threaded) and **1 thread** was almost identical across all input sizes. This indicates that the overhead introduced by creating a single thread was minimal. In theory, a single-threaded process should slightly outperform using one thread, but the difference is negligible in practice, as reflected in the results.

Performance Scaling with Multiple Threads

As the number of threads increased, there was a consistent decrease in execution time, but the speedup was **not perfectly linear**. Here's a summary of the scaling behavior for 1,000,000 inputs:

- **2 threads** provided ~1.36x speedup (compared to 1 thread),
- **4 threads** provided ~2.31x speedup,
- **8 threads** provided ~4.16x speedup,
- **16 threads** provided ~6.85x speedup,
- **32 threads** provided ~8.91x speedup,
- **64 threads** provided ~9.29x speedup.

Diminishing Returns with More Threads

As the thread count increased beyond **16-32 threads**, the performance gains started to diminish. This can be attributed to several factors:

- **Thread management overhead**: As more threads are created, there is an increase in the overhead for managing these threads (context switching, synchronization, etc.).
- **CPU core limitations**: The Intel Core Ultra 9 185H has **24 threads** available, so adding more than 24 threads results in some threads sharing CPU resources, leading to reduced efficiency.
- **Parallelism limitations**: Beyond a certain point, the workload may not be efficiently parallelizable, resulting in less performance gain from additional threads.

Conclusion

The experiment demonstrated that multi-threading provided significant performance improvements, especially when using up to **16 threads**. However, as the number of threads increased beyond the number of physical cores (16 cores) and logical threads (24 threads), the gains diminished due to the overhead of managing more threads and limitations in parallelism. The CPU's architecture played a crucial role in these results, highlighting that the number of threads should be carefully chosen to balance performance gains and overhead.

Laboratory 2 experiment 1 results

Average Overhead of CPUID Instruction

The **average CPUID execution time** of around **896 clock cycles** is reasonable for a CPUID instruction. The CPUID instruction causes all the instructions preceding it to be executed, before its execution. If CPUID instruction is placed before, the RDTSC instruction will be executed in-order. To make an exact measurement, the overhead of CPUID can be measured and subtracted from the cycle count obtained for the measured code.

Measurement number	CPUID execution time (clock cycles)
1	816
2	812
3	924
4	934
5	924
6	970
7	924
8	926
9	828
10	908
Average time	896

ADD(reg)	ADD(var)	MUL	FDIV	FSUB
35	35	139	427	389
21	29	151	435	383
25	27	95	401	385
31	23	87	395	309
37	29	101	431	385
29.8	28.6	114.6	417.8	370.2

Instruction Execution Times

Here are the average execution times for the individual instructions based on the results:

- **ADD (reg)**: 29.8 clock cycles
- **ADD (var)**: 28.6 clock cycles
- **MUL**: 114.6 clock cycles
- **FDIV**: 417.8 clock cycles
- **FSUB**: 370.2 clock cycles

Explanation of Results

1. **ADD (register vs. variable)**:
   - Both the register-based and variable-based ADD instructions have low execution times, around **30 clock cycles**. This is expected because integer addition is a basic operation and modern processors are highly optimized for such tasks, especially when the values are in registers. The slightly lower time for the variable version could be due to cache optimizations if the variable is already cached in a nearby register.

2. **MUL**:
   - Multiplication takes **114.6 clock cycles**, which is noticeably higher than addition. Multiplication is inherently a more complex operation compared to addition(in fact it requires multiple additions), requiring more hardware resources. The time is still relatively low, showing that the CPU is efficiently handling multiplication, but the difference with ADD reflects the varying complexity of these operations.
3. **FDIV and FSUB**:
   - Floating-point division (**FDIV**) and subtraction (**FSUB**) have much higher execution times. FDIV is the slowest of all, with an average of **417.8 clock cycles**, while FSUB takes **370.2 clock cycles**. Floating-point operations generally require more computation than integer operations due to precision handling and the complexity of the floating-point unit (FPU).
   - Division is especially expensive in terms of CPU time compared to addition or multiplication. It involves more complicated algorithms (like Newton-Raphson iterations) in hardware, which explains why it takes significantly longer than multiplication.

Conclusion and Logical Coherence

- The measurements make sense based on the CPU architecture and instruction complexity. Simpler operations like ADD (both register and variable) are handled very efficiently, while operations involving more complex calculations, such as floating-point division, take more time. This is a typical characteristic of modern CPUs, which are optimized for integer arithmetic but take longer for floating-point operations due to their computational complexity.
- The results are in line with the expectations of how modern processors handle arithmetic and floating-point operations. The slight variations in timing for the CPUID and ADD operations are natural and likely due to CPU load and other minor factors.

Laboratory 2 experiment 2.1 results

1. **Execution Times**:
   - The table shows the individual measurements and average execution times for both static and dynamic arrays across five runs using RDTSC and clock().

	rdtsc time(clock cycles)		clock() time(clock cycles)	
Measurement number	static array	dynamic array	static array	dynamic array
1	1484782	1747442	3800000	3800000
2	1269115	1211171	5320000	4180000
3	1623458	2046767	4180000	5760000
4	1261877	1320099	4080000	4560000
5	1551105	1675409	4560000	4940000
Average time	1438067.4	1600177.6	4388000	4648000

   - **Static Array RDTSC Time**: Ranges from 1,261,877 to 1,623,458 clock cycles.

- **Dynamic Array RDTSC Time**: Ranges from 1,217,171 to 2,046,767 clock cycles.
- **Static Array clock() Time**: Ranges from 4,180,000 to 5,320,000 clock cycles.
- **Dynamic Array clock() Time**: Ranges from 4,180,000 to 5,760,000 clock cycles.

2. **Average Execution Time**:
   - **RDTSC Average**:
     - **Static Array**: 1,438,067.1 cycles
     - **Dynamic Array**: 1,600,177.6 cycles.
   - **clock() Average**:
     - **Static Array**: 4,388,000 cycles.
     - **Dynamic Array**: 4,648,000 cycles.

Analysis of Results

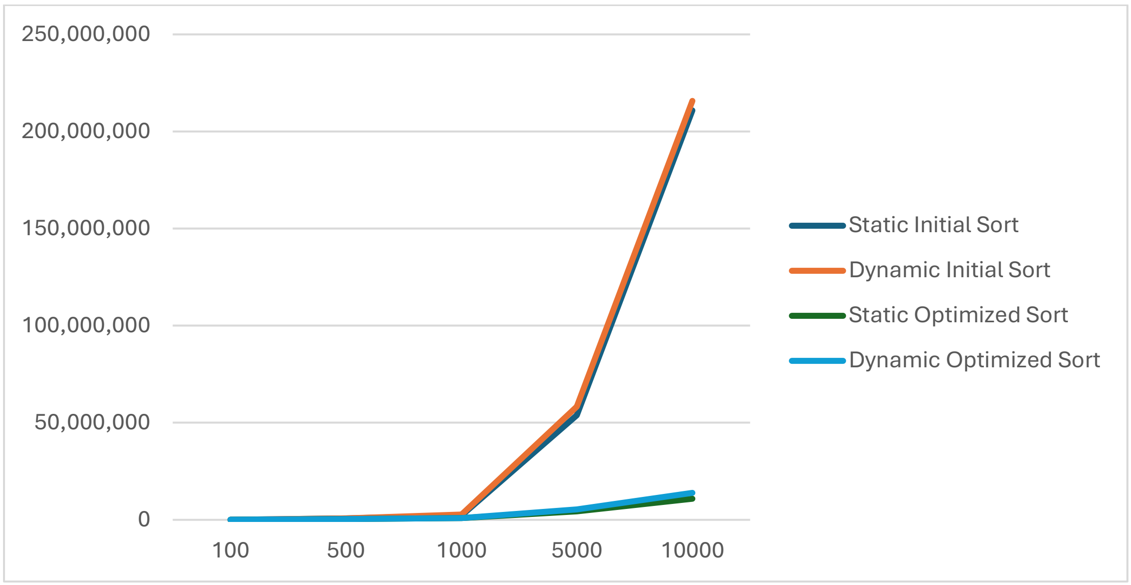
1. **Comparison Between RDTSC and clock()**:
   - The RDTSC instruction generally shows lower average cycle counts for both static and dynamic arrays compared to clock(). This difference is expected because RDTSC directly measures CPU cycles, while clock() measures elapsed time, which can be affected by various factors such as context switching, interrupts, and other system activities.
2. **Static vs. Dynamic Arrays**:
   - For both measurement methods, dynamic arrays consistently have higher average execution times compared to static arrays. This discrepancy can be attributed to:
     - **Memory Allocation Overhead**: Dynamic arrays require memory allocation at runtime, which introduces additional overhead compared to static arrays, which are allocated at compile time.
     - **Cache Efficiency**: Static arrays may benefit from better cache locality, leading to improved access speeds during sorting.
3. **Variation in Measurements**:
   - There is a noticeable variation in the execution times across different runs. This variability could result from factors such as system load, CPU temperature, and other background processes that may interfere with CPU scheduling during sorting.
4. **Practical Implications**:
   - The findings indicate that for performance-critical applications, using static arrays can lead to better execution times compared to dynamic arrays, especially in sorting algorithms.
   - The results underscore the importance of choosing the right data structure based on the specific requirements and constraints of an application.

Conclusion

In summary, the results of the table highlight significant differences in execution time based on the measurement method and the type of array used. The lower execution times recorded with RDTSC demonstrate its precision for low-level timing, while the higher times recorded with clock() emphasize the additional overhead associated with general-purpose timing functions and the dynamic allocation of memory.

Laboratory 2 experiment 2.2 results

	Initial sort time(clock cycles)		Optimized sort time(clock cycles)	
Array length	static array	dynamic array	static array	dynamic array
100	27788	30996	51508	71255
500	779248	792689	413170	438405
1000	2269474	2805558	952902	994371
5000	53811119	58246475	4368984	5439100
10000	210856891	215756111	10860334	13885852



1. Bubble Sort (Initial Sort Time)

Bubble Sort is a simple sorting algorithm with a time complexity of O(n^2). It's inefficient for large datasets as it compares and swaps adjacent elements repeatedly. Let's examine its behavior:

- **Array Size 100 to 10,000**: The sorting time grows significantly as the array length increases. For example:
  - With static arrays: from **27,788 cycles** for 100 elements to **210,856,891 cycles** for 10,000 elements.
  - With dynamic arrays: from **30,996 cycles** for 100 elements to **215,756,111 cycles** for 10,000 elements.

This increase aligns with Bubble Sort's quadratic complexity. As the array size grows, the number of comparisons and swaps increases quadratically.

- **Static vs. Dynamic Arrays**:
  - Dynamic arrays consistently take more clock cycles than static arrays. This is likely due to the overhead involved in managing dynamic memory allocation (e.g., memory allocation and access are slightly more expensive for dynamic arrays).
  - The overhead is especially noticeable for smaller arrays (e.g., 30,996 cycles for dynamic vs. 27,788 cycles for static at array size 100). As the array size grows, the difference becomes less significant, because the overall number of comparisons (due to the algorithm's nature) dominates the time.

2. Quick Sort (Optimized Sort Time)

Randomized Quick Sort is a more efficient algorithm, especially for larger datasets, with an average time complexity of O(nlogn). Here are the key observations:

- **Array Size 100 to 10,000**:
  - The sorting time grows much slower compared to Bubble Sort. For static arrays, the time increases from **51,508 cycles** (100 elements) to **10,860,334 cycles** (10,000 elements). For dynamic arrays, from **71,255 cycles** to **13,885,852 cycles**.
  - This growth is much more moderate compared to Bubble Sort's drastic increase, confirming that Quick Sort is better suited for larger datasets.
- **Performance Trends**:
  - For small array sizes (e.g., 100 elements), Quick Sort already shows better performance than Bubble Sort. However, the difference is more pronounced as the array size increases. For example, at array size 10,000, the Quick Sort time is around **10.8 million** clock cycles (static) vs. Bubble Sort's **210 million** clock cycles, a significant improvement.
- **Static vs. Dynamic Arrays**:
  - Similar to Bubble Sort, dynamic arrays take longer to sort than static arrays. The difference is smaller for large datasets, as the efficiency of Quick Sort helps offset the overhead of dynamic memory management.

Conclusion:

- The results clearly show that **Randomized Quick Sort** significantly outperforms **Bubble Sort**, especially as the array size increases. Bubble Sort's quadratic time complexity leads to a drastic increase in execution time, making it impractical for large datasets. In contrast, Quick Sort's O(nlogn) complexity allows for much more efficient sorting, even for larger arrays.
- Additionally, **static arrays** consistently perform better than **dynamic arrays** due to the lower overhead of memory management. However, this difference becomes less impactful as the array size grows, with the choice of algorithm becoming the dominant factor in execution time.
- In summary, for real-world applications involving large datasets, efficient algorithms like Quick Sort should be preferred, and static arrays may offer slight performance advantages where feasible.