Set of benchmark programs to evaluate the performance of a multicore processor

Student: Buleu Mihai-Andrei

Technical University of Cluj-Napoca

1 Introduction

1.1 Context

The aim of this project is to provide a set of tools that measure and analyze the behavior of **multithreaded execution** on modern processors. The focus is on designing and implementing **multithreaded programs** that perform various computational tasks, while measuring and comparing their **execution times** as the number of threads varies.

Multithreading enables multiple sequences of instructions to run concurrently, allowing better utilization of multicore processors and improving the overall throughput of the system.

1.2 Objectives

The benchmark programs will be designed in C/C++ and **Windows x86 compatible source code** will be provided. The goal is to measure both **global** and **per-thread** execution times and to analyze performance when varying the number of threads.

The final benchmark set will include multiple algorithm types and measurement options.

The types of algorithms and features implemented

Algorithms:

- **Calculating prime numbers**
- **Sorting**
- **Matrix multiplication**

Additional features:

- **Thread selection:** manual or automatic use of all available threads.
- **Timing evaluation:** measurement in seconds and CPU cycles.
- **Timer choice:** clock() or RDTSC.
- **I/O analysis mode.**

2 Bibliographic Research

2.1 What is multithreading?

Multithreading is a method of parallel execution in which multiple sequences of instructions, called *threads*, run concurrently within a single process. Each thread shares the same address space and resources but executes independently, allowing better utilization of multicore processors.
In modern systems, multithreading is widely used to improve performance, responsiveness, and resource efficiency. This project focuses on measuring and analyzing the execution time of multithreaded programs under different configurations.

2.2 Measuring execution time

Performance evaluation in multithreaded programs is typically based on measuring execution time. Several timing methods can be used:

- **clock()** – a standard C function that returns processor time consumed by the program. It offers portability but limited precision.
- **RDTSC (Read Time-Stamp Counter)** – a low-level CPU instruction returning the number of clock cycles since reset, offering high precision on x86 systems.

2.3 Analyzing thread performance

The performance of multithreaded applications can be evaluated by observing how execution time changes as the number of threads varies. Ideally, doubling the number of threads should halve the execution time, but in practice, the improvement is limited by synchronization, communication, and memory access conflicts.
The project uses several algorithms—**pi calculation**, **sorting**, and **matrix multiplication** to represent different types of computational workloads. Each algorithm stresses a specific aspect of parallel execution, allowing comparative analysis.
An additional I/O analysis mode is also provided to examine performance in mixed computation and file-access workloads.
The collected data supports the evaluation of scalability, efficiency, and the effect of timing methods on multithreaded performance measurement.

3.Analysis

This chapter details the specific features and algorithms proposed for the benchmark set. The following sections will analyze the specific algorithms chosen—Calculating prime numbers, Sorting, and Matrix multiplication—along with the different measurement options , detailing how each will be used to evaluate multithreaded performance.

3.1 Algorithms

The core of the project is to evaluate performance by running the same task with a varying number of threads and comparing the execution times. The chosen algorithms—calculating prime numbers, sorting, and matrix multiplication—were specifically selected because they represent a diverse range of computational workloads.

3.1.1 Calculating Prime Numbers

This workload is representative of parallel problems. The total range of numbers to be checked is based on an array populated with random numbers. The total range of numbers to be checked can be divided into many independent sub-tasks, with each thread handling its own sub-range.

3.1.2 Sorting (Parallel Merge Sort)

This implementation utilizes the power of multithreading to accelerate the sorting process. The array is efficiently divided into sub-arrays, and the task of sorting these sub-arrays is distributed across threads. Each thread works independently to complete its assigned portion and significantly speeding up the initial computational phase of the algorithm. Once all sub-arrays are sorted, a single thread is responsible for the final merge operation.

3.1.3 Matrix Multiplication

This algorithm is a example of a computationally intensive task. The problem is efficiently parallelized by assigning each thread the task of computing a single line of the resulting matrix. Specifically, if the first input matrix has N lines, N threads are launched, with each thread responsible for completing one full line of the output matrix.

3.2 Timer

In order to evaluate the performance of the algorithms above, a measurement has to be performed. In this project there will be used types of measurements to determine the cycles and the seconds of the provided algorithms each with it's own benefits.

3.2.1 RDTSC

The Time-Stamp Counter (TSC) is a 64-bit Model Specific Register (MSR) present on all x86 processors, designed for high-resolution timing. It continuously counts the number of CPU cycles that have occurred since the last processor reset. This cycle count is retrieved by executing the RDTSC (read time-stamp counter) instruction, which returns the current value of the TSC, thereby providing a precise measure of elapsed CPU cycles.

3.2.1.1 Out-of-order execution

To ensure accurate cycle counting, the RDTSC instruction must be preceded by a serializing instruction like CPUID. This prevents the CPU's out-of-order execution from running RDTSC prematurely. For an exact measurement, the execution time (overhead) of CPUID must be determined and subtracted from the total cycle count. It is recommended to measure the overhead of the third call to CPUID, as its initial two calls can execute slower.

3.2.1.2 Context Switching

Context switches can bias RDTSC measurements because the cycles consumed by interrupting processes and the context switch time itself are included in the final count.

For small code sequences, this interruption probability is low. However, for longer code sequences, the risk increases.

3.2.1.3 TSC in multi-core processors

In multi-core processors, the Time-Stamp Counters (TSCs) on different cores may not be synchronized unless the processor supports invariant TSC. When using the RDTSC instruction for timing, thread migration across cores can lead to inconsistent measurements. To mitigate this, each thread's affinity is set to a unique core using SetThreadAffinityMask, ensuring that each thread runs on a specific core (e.g., core 0, 1, 2, etc.) with a fallback to core 0 if the requested core is invalid or unavailable.

3.2.2 Clock()

The standard C library function clock() is a straightforward method for timing code because it's easy to implement and highly portable across different operating systems. However, its resolution is tied to the operating system's internal timer tick rate, which is typically in the millisecond range. Consequently, clock() is inaccurate for measuring short code segments (those that execute in a few hundred thousand cycles or less) because the timing overhead and the low resolution can easily exceed the actual execution time of the code itself. While ideal for timing large, long-running processes, it cannot provide the necessary cycle-level precision required for

3.3 Input/Output

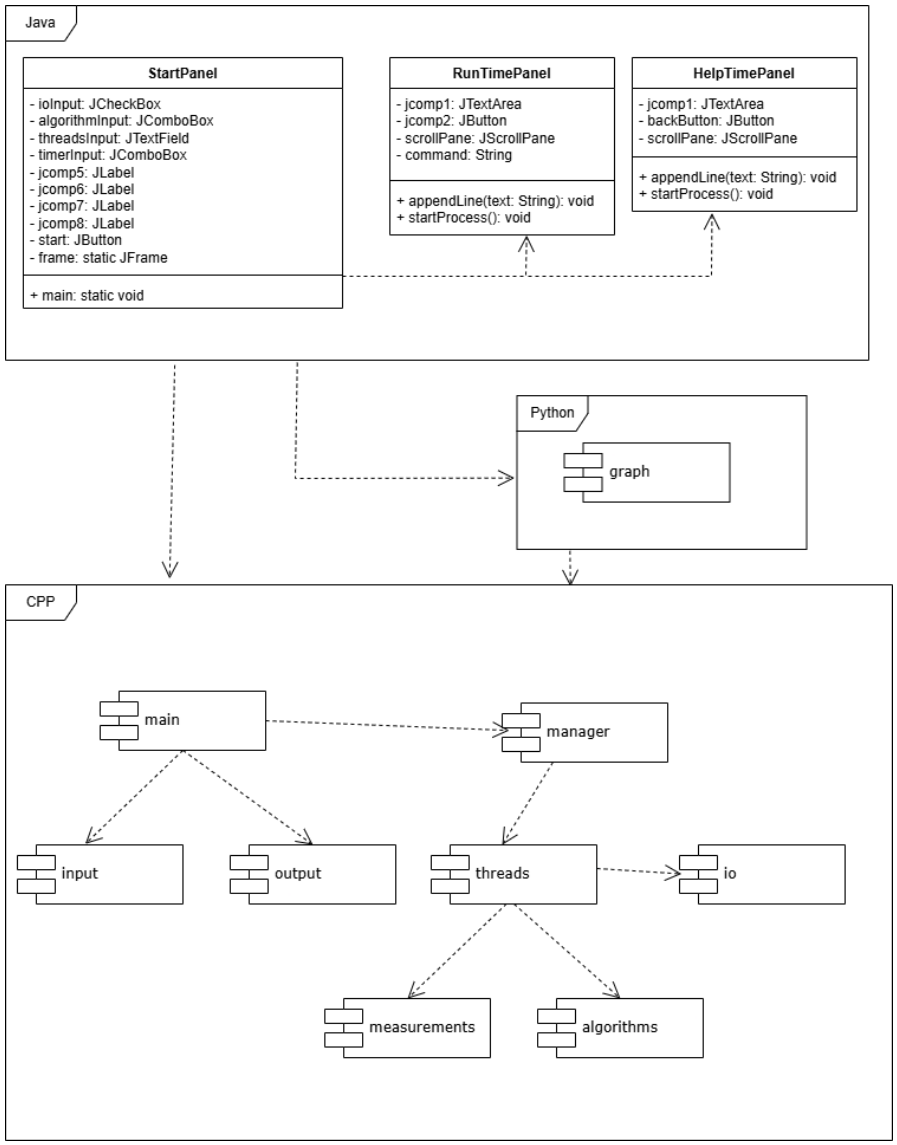
Analyzing I/O in a multithreading benchmark primarily involves measuring how efficiently the CPU can switch work when threads become blocked waiting for slow I/O operations (like reading from disk). The program will have multiple files from which the theads will read data byte by byte. The number of files will be distributed as evenly as possible within the threads.

4 Design

This chapter outlines the high-level architecture of the benchmark program, focusing on the software structure, data flow, and control mechanisms that manage multithreaded execution and timing. The design ensures that the core objectives—testing different

algorithms, varying thread counts, and supporting precise timing methods—are met efficiently. The primary control flow is managed by a user menu that allows for dynamic selection of algorithms and measurement modes.

4.1 Component Diagram



The provided Component Diagram offers a high-level, structural view of the software architecture, identifying the major, replaceable, and well-defined units of the system (the Components) and illustrating how these units interact and depend on one another.

The diagram shows three main sections:

Java Components: This section outlines the Java-based classes. These represent graphical user interface elements.

Python Components: This section details the Python-based component, specifically highlighting a graph component used for displaying data.

CPP Components: This section illustrates the C++-based components, which form the core processing logic. Key components include main, manager, input, output, threads, io, measurements, and algorithms.

The dashed lines indicate dependencies and interactions between these components, showing how the different language-specific parts of the system are integrated.

4.1.1 Design of “measurements” component

The measurements component contains 2 structs used for information once the measurement is done and for differentiating measurements from each other. There is a funcion provided to calculate cpuid, that will be used only once, and a function that handles tsc problems made by multicore programs. There are start/end funcions for the rdtsc and clock timers.

measurements
MeasuredInfo MeasurementState

void measurement_state_init(MeasurementState *state);
void compute_cpuid_overhead()
void rdtsc_start(MeasurementState *state);
MeasuredInfo rdtsc_end(MeasurementState *state);
void clock_start(MeasurementState *state);
MeasuredInfo clock_end(MeasurementState *state);

4.1.2 Design of “algorithms” component

algorithms

void generate_random_array(unsigned int a[], int n, int max_val = MAX);
void compute_matrix_row(unsigned int *matrix_a, unsigned int *matrix_b, unsigned int *matrix_c, int row_index, int n, int p);
void mergesort(unsigned int a[], int left, int right);
void print_matrix(unsigned int *matrix, int m, int p);
void merge_sorted_subarrays(unsigned int a[], int subarray_starts[], int subarray_ends[], int num_parts);
void shuffle_array(unsigned int arr[], int n);
bool is_prime(unsigned int n);

The Algorithms Component encapsulates the core computational logic of the application. It has funcions used for checking if a number is prime, for creating or shuffling a random array, mergesort and the matrix computation function.

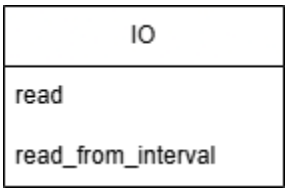
4.1.3 Design of “threads” component

The threads component handles the tread’s struct and funcion depending on the required workload. Each algorithm and io mode has a specific struct and a funcion that executes when the thread launches. The funcions make use of other funcions defined in different components like the previously mentioned “algorithms” component.

threads
MatrixRowData SortArrayData PrimeData IOData
MatrixThreadFunction PrimeThreadFunction SortThreadFunction IOThreadFunction

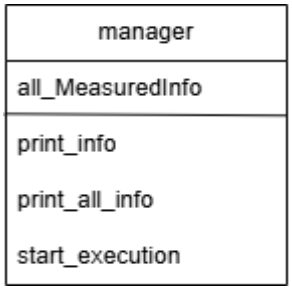
4.1.4 Design of "IO" component

The IO component is used in measuring io-bound capabilities by reading byte by byte data from a handfull of files already present in the project. It handles the application's file access, specifically reading from the numbered binary files in the resources directory.



4.1.5 Design of "input" and "output" components

The input and output components, not to be confused with IO component are 2 different components that handle the input data recived from user and the output data that the user can read. The output component isn't the only one that can output messages on the screen, other components can also do that when an error or a warning appears.



4.1.3 Design of "main" and "manager" components

These two components are the ones that manage the other components. Main is used for high level interactions and the manager component serves as the central coordination and benchmarking engine for the application, handling the complex logistics of running parallel tasks across multiple threads. It contains specialized functions (io_execution, prime_execution, etc.) that are responsible for preparing input data, calculating how to divide the workload into chunks, initiating threads, and then waiting for all threads to finisy\h. Crucially, the manager interfaces directly with the measurements component to start and stop global and per-thread timers.

4.1.6 Design of "StartPanel" and "RunTimePanel" java classes

These two components form the graphical front-end of the application. They are responsible for collecting benchmark parameters from the user, initiating the C++ backend process, and displaying the live output and results (as if the program was lauched from the comand line). The RunTimePanel component in the Java code uses the Python component for visualization.

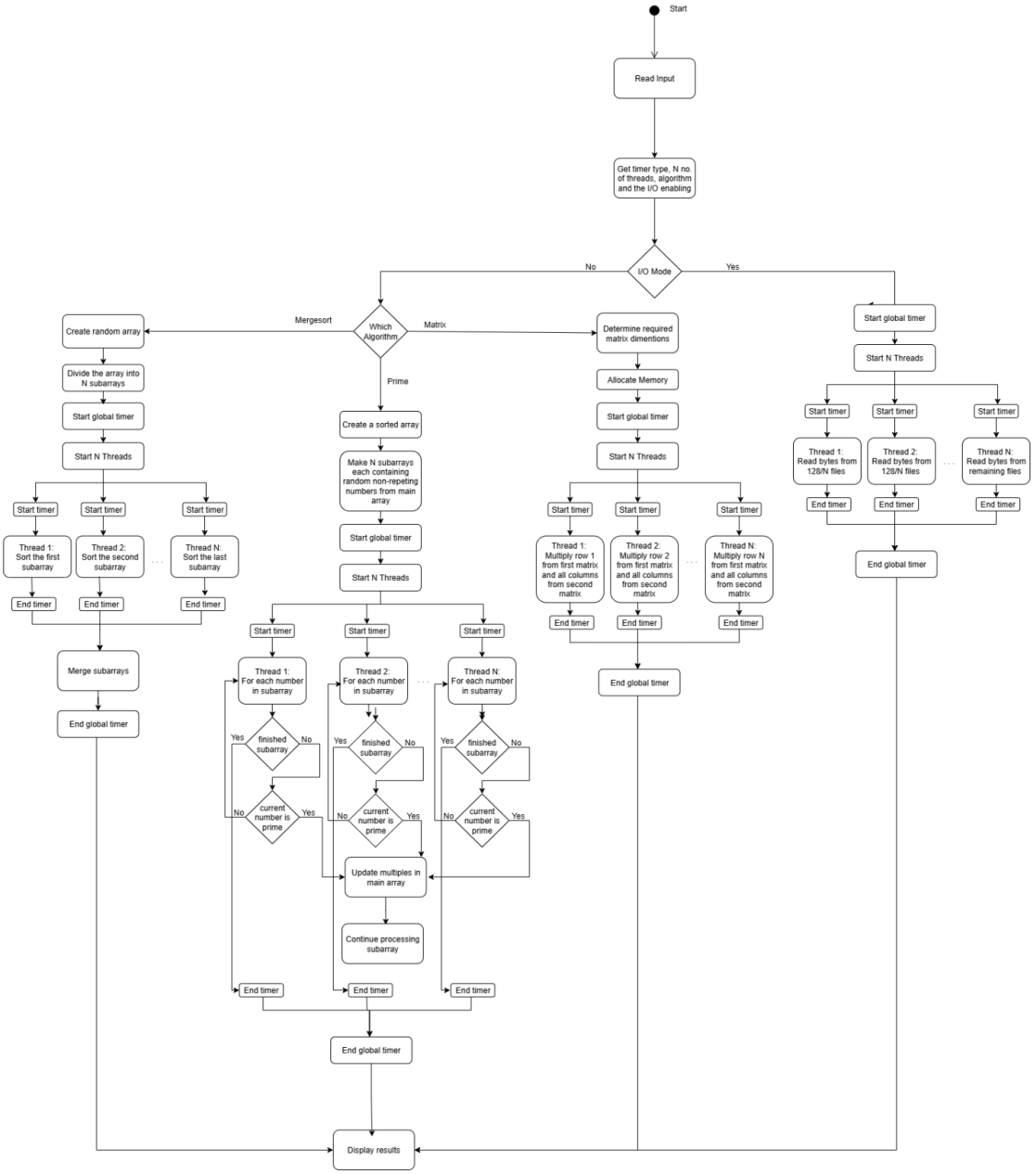
4.1.7 Design of "graph" component

The graph component is a Python script responsible for taking the raw performance data generated by the C++ execution and transforming it into a clear, visual representation

using bar charts. This script is executed by the Java RunTimePanel after the C++ benchmark completes.

4.2 Flowchart

The flowchart serves as a graphical representation of the program's main execution loop. It details the initial setup, the decision process based on user input, the execution of the chosen benchmark algorithm, and the final timing calculation.



5 Implementation

The project is implemented as a system of three distinct modules: a C++ backend for high-performance computation and measurement, a Java Swing GUI for user interaction and control, and a Python script for data visualization. The Java frontend acts as the main entry point, launching and coordinating the other two components.

5.1 UI Panels

The interface is composed of two main panels:

StartPanel.java: This is the initial view. It contains JComboBox and JTextField components to allow the user to select the algorithm, number of threads, timer, and I/O mode. Upon clicking the "Start" button, it constructs the command-line argument string for the C++ backend.

RunTimePanel.java: This panel is displayed after the benchmark starts. It features a JTextArea that shows the live standard output from the C++ backend, providing real-time feedback.

HelpPanel.java: This panel shows the output of the program if it was run with the –help flag.

5.1.1 Process Management

The Java application uses the ProcessBuilder class to manage the external processes.

C++ Backend Execution: When the benchmark is started, RunTimePanel launches the compiled C++ executable (program.exe). It captures the process's InputStream and pipes the output line-by-line to the JTextArea.

Python Script Execution: When the user clicks the "View Results" button, a second ProcessBuilder instance is created to run the Python visualization script. The implementation requires a python virtual enviorment to be created beforehand at py_src, named myenv and to have all the libraries required to run the application inside the virtual environment.

5.2 Data Visualization

The graph.py component is a Python script, which is responsible for parsing the JSON output and creating a visual representation of the results.

5.2.1 Data Parsing and Processing

The script uses the json library to load the raw data from build/output.json. This data is then loaded into a pandas DataFrame, which provides a powerful and convenient structure for data manipulation and analysis.

5.2.2 Plot Generation

The matplotlib library is used to generate the final plots. The script creates a bar chart that visually compares the elapsed_sec and elapsed_cycles for each thread, including the "global" result. This allows for easy identification of performance characteristics and thread-level variations.

5.3 C++ Computational Backend

The backend is a native command-line application (program.exe) responsible for executing the actual benchmark. It is designed for performance and provides detailed timing results.

5.3.1 Argument Parsing

The application's behavior is controlled via command-line arguments, which are parsed in input.cpp. This module defines the ProgramOptions structure and uses a simple loop through argv to identify and validate flags such as --threads, --algorithm, --timer, and --io. If arguments are missing or invalid, it falls back to a set of default options. There is a –help flag that prints more detailed information on the usage of the other flags:

```
Usage: program.exe [OPTIONS]

Options:

  -h, --help             Display this help message and exit.

  --threads <number|auto>  Set the number of threads.
                         'auto' uses hardware concurrency.
                         (Default: hardware concurrency or 4 if unavailable)

  --algorithm <type>     Set the algorithm to run.
                         Available types: sorting, matrix, prime
                         (Default: matrix)

  --timer <type>         Set the timer type.
                         (Default: clock)

  --io                   Enable I/O mode.
                         (Default: disabled)
```

5.3.2 I/O Workload Component

The I/O workload is implemented in io.cpp and is designed to test the performance of file system read operations. This module is invoked when the user selects the "I/O" mode from the GUI or uses the –io flag. Its core responsibility is to read a predefined set of binary data files from the disk.

A feature of this component is its ability to locate the data files relative to the application's executable path. This avoids hardcoded absolute paths and allows the application to be run from different locations, provided the directory structure is maintained (resources folder with it's files has to be in the same root parent folder as the executable) The read function first calls the Windows API function GetModuleFileNameA to retrieve the full path to the running program.exe. The expected structure of the files is: .../build/resources/file_N.bin, where N is a pozitive integer from 1 to 128.

The primary logic is within the read(int file_number) function. This function is responsible for reading a single, specified binary file.

1. The target file is opened in binary read mode ("rb").

```
FILE *f = fopen(final_path, "rb");
```

2. The file is read byte-by-byte using fgetc() inside a while loop that continues until the End-Of-File (EOF) marker is reached.

3. The bytes_sum variable is placed there to ensure the read operation is not optimized away by the compiler.

```
while ((buff = fgetc(f)) != EOF) {
    bytes_read_count++;
    bytes_sum +=
        buff; // need so the compiler doesn't optimize the read and the loop
}
```

The read_from_interval function serves as a wrapper to facilitate concurrent execution. If the number of files per thread its definded equally this funcion can be used to measure the distribute file reading equally.

```
void read_from_interval(int start, int end) {
    for (int i = start; i <= end; ++i)
        read(i);
}
```

5.3.3 Algorithm Component

The algorithm workload is implemented in algorithms.cpp with its public functions declared in algorithms.h. The algorithms are matrix multiplication, parallel mergesort, and a prime number computation. Helper functions like generate_random_array and shuffle_array are used to prepare randomized data inputs for these workloads.

1. Matrix Multiplication: This workload computes the product of two matrices filled with random data generated by generate_random_array. The parallel implementation is centered around the compute_matrix_row function.

```
void compute_matrix_row(unsigned int *matrix_a, unsigned int *matrix_b,
                        unsigned int *matrix_c, int row_index, int n, int p) {
  for (int j = 0; j < p; j++) {
    unsigned int sum = 0;
    for (int k = 0; k < n; k++) {
      sum += matrix_a[row_index * n + k] * matrix_b[k * p + j];
    }
    matrix_c[row_index * p + j] = sum;
  }
}
```

This is the core function distributed to worker threads. It calculates the values for a single row of the output matrix (matrix_c), therefore in an multi-threaded environment the complete result matrix will be computed by having each thread work on a different line of matrix_a.

2. Parallel Mergesort: This algorithm sorts a array of random integers. The process is split into two main phases: a parallel sort and a final sequential merge. The mergesort funcion is executed by each thread on a distinct, contiguous partition of the main array

```
void mergesort(unsigned int a[], int left, int right) {
  if (left < right) {
    int mid = left + (right - left) / 2;
    mergesort(a, left, mid);
    mergesort(a, mid + 1, right);
    merge(a, left, mid, right);
  }
}
```

After all threads have completed sorting their partitions via mergesort, the merge funcion this function performs the final merging of the sorted subarrays.

3. Prime Number Sieve: This algorithm identifies prime numbers within a range. The workload is parallelized by dividing the number range into segments and assigning each segment to a thread. The funcion check_primes is executed by each thread. It iterates through its assigned segment of the array. The check_primes function has no optimizations made, thus having an O(n) execution time.

```
void check_primes(unsigned int array[], int array_start, int array_end) {
    for (int k = array_start; k < array_end; k++) {
        if (is_prime(array[k]) == true) {
            for (int j = 2 * array[k] - 2; j < array_end;
                 j += array[k]) { // array_end e doar la subarray
                array[j] = 0;
            }
        } else {
            array[k] = 0;
        }
    }
}
```

5.3.4 Measurement

The measurements component implements two primary timing methods: RDTSC and clock(). Uses the struc MeasurementState to hold cycle counts, other information necessary for the RDTSC instruction to perform corectly and start/end times. The final results are saved in the MeasuredInfo struct.

```
typedef struct MeasuredInfo {
    long long elapsed_cycles;
    double elapsed_sec;
    int thread_index;
} MeasuredInfo;

typedef struct {
    unsigned cycles_high1, cycles_low1;
    unsigned cycles_high2, cycles_low2;
    unsigned __int64 temp_cycles1, temp_cycles2;
    __int64 total_cycles;
    clock_t start, end;
} MeasurementState;
```

The measurement_state_init function ensures measurement integrity by resetting all cycle counter variables within the MeasurementState structure to zero. The SetThreadAffinityForCore(int threadIndex) function uses the Windows API to "pin" the current thread to a specific CPU core. The cpuid is computed 3 times in order to accuaretly get it's value and it needs to be computed only once in the program.

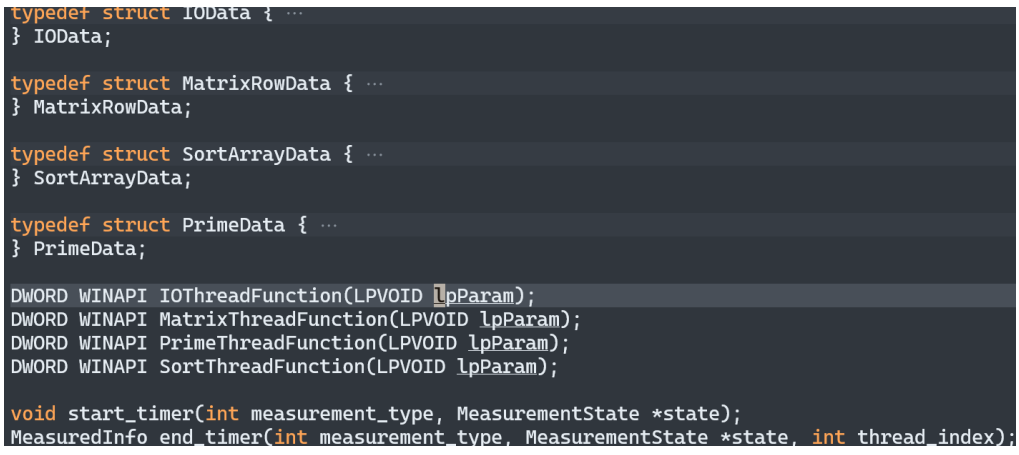
```
void measurement_state_init(MeasurementState *state);
void compute_cpuid_overhead();
void SetThreadAffinityForCore(int threadIndex);
```

The rdtsc_start/rdtsc_end and clock_start/clock_end functions handle the timing of code execution, utilizing the MeasurementState struct to store and retrieve the timestamps and MeasuredInfo to return the total time and cycles.

5.3.5 Thread management funcions

Each workload (I/O, Matrix, Prime, Sort) has a dedicated thread function. These functions follow a standardized execution pattern:

1. Thread Initialization:
    - Cast the generic LPVOID lpParam to the specific data structure for the workload (e.g., IOData *data).
    - Call SetThreadAffinityForCore() using the thread's unique index to pin it to a CPU core, ensuring consistent measurements.
    - Initialize a local MeasurementState struct.
2. Timing Start:
    - Call start_timer() to begin the global or per-thread timing using either clock() or RDTSC.
3. Workload Execution:
    - Call the specific function from the algorithms or io components to perform the assigned task (e.g., read_from_interval, compute_matrix_row). The prime implements the critical synchronization logic within the function, which is necessary for the parallel algorithm to safely modify the shared array by marking multiples of a prime number.
4. Timing End:
    - Call end_timer() to stop the timer, calculate the result, and store the resulting MeasuredInfo in the thread's data structure (data->info).



5.3.6 Core management of the benchmark

The manager component defines a high-level start_execution function that orchestrates the benchmark, and four specialized functions (io_execution, prime_execution, matrix_execution, mergesort_execution) for each specific workload.

1. start_execution()

This is the main entry point for running the benchmark and handling multiple iterations.

- Initialization: Allocates memory for the all_MeasuredInfo structure (total_sum) which will hold the accumulated results across all iterations. It initializes the cycle and second counts to zero.
- Iteration Loop: A for loop executes the selected benchmark (type) for the specified num_iterations (10)
- Workload Dispatch: A switch statement calls the appropriate specialized execution function (e.g., prime_execution) based on the type parameter.

```
switch (type) {
case PRIME:
  result = prime_execution(num_threads, measurement_type);
  break;
case IO:
  result = io_execution(num_threads, measurement_type);
  break;
case MATRIX:
  result = matrix_execution(num_threads, measurement_type);
  break;
case SORT:
  result = mergesort_execution(num_threads, measurement_type);
  break;
default:
  printf("Execution type not implemented\n");
  total_sum.array_size = 0;
  free(total_sum_array);
  return total_sum;
}
```

- Accumulation: After each iteration, the results are added to the running total using accumulate_measured_info(). `accumulate_measured_info(&total_sum, result);`
- Averaging: After the loop completes, it divides all accumulated cycle and second counts (global and per-thread) by num_iterations to provide an averaged result.

2. **Execution funcions:**

All four execution functions (io, prime, matrix, mergesort) follow a consistent pattern for multithreaded execution:

1. Data Setup:
   Prepares input data (e.g., matrices, arrays, file indices) and calculates how to divide the workload into chunk_size for each thread.

2. Global Timer:
   Starts the benchmark's global timer.

3. Thread Setup:
   Allocates memory for an array of thread handles (HANDLE*) and an array of thread data structures (...Data**).

4. Thread Launch:
   A for loop populates the thread data structures (assigning thread_index, start/end ranges, shared data pointers) and launches the threads.

5. Thread Finish:
   The manager thread waits until all worker threads have completed their tasks.

6. Global Timer End:
   Stops the global timer and stores the result.

7. Result Aggregation:
   Collects the per-thread MeasuredInfo results from the thread data structures (pData[i]->info).

8. Cleanup:
   Closes thread handles and frees all dynamically allocated memory (handles, data structs, arrays).

```
for (int i = 0; i < num_threads; i++) {
  all_info.thread_info_array[i] = pData[i]->info;
  CloseHandle(hThread[i]);
  free(pData[i]);
}
```

3. **Synchronization and Data Handling**

Critical Section Management:

The prime_execution function is the only one that uses a synchronization primitive providing the necessary lock mechanism for the PrimeThreadFunction to safely modify the shared array.

```
InitializeCriticalSection(&critical_section);
DeleteCriticalSection(&critical_section);
```

Data Structures and Utilities:

- all_MeasuredInfo: A struct containing a single global_info and a dynamically allocated array of thread_info_array results, which is the standard return type for all execution functions.

```
typedef struct all_MeasuredInfo {
  MeasuredInfo global_info;
  MeasuredInfo *thread_info_array;
  int array_size;
} all_MeasuredInfo;
```

- print_info / print_all_info: Functions used to format and output the timing results to the console.

```
void print_info(MeasuredInfo info);
void print_all_info(all_MeasuredInfo all_info);
```

- accumulate_measured_info: Adds the cycle and second measurements from one all_MeasuredInfo struct to another. This is the core logic used to compute the total time across multiple benchmark iterations.

```
void accumulate_measured_info(all_MeasuredInfo *accumulator,
                              all_MeasuredInfo new_info) {
```

4. **Specific prime algorithm**

The prime algorithm usese the shuffle_array funcion to randomize the order in which the numbers in the main array are checked for primality. Instead of iterating through the main array sequentially (which could lead to cache conflicts or simple, predictable behavior), threads iterate through their assigned segment of the shuffled index_array. This forces each thread to check elements scattered across the main array, stressing memory access and synchronization more realistically.

5.3.7 Main

The main function is the application driver of the benchmark orchestrating the entire process from configuration to output.

1. I/O Control:
   It disables stdout and stderr buffering (setvbuf) to ensure real-time output is streamed correctly to the Java frontend.

2. Configuration:

   It calls input_management to parse command-line arguments, validates the selected options, and converts the string-based timer type into an internal numerical constant.

3. Execution Dispatch:

   It selects the appropriate benchmark (IO, SORT, MATRIX, or PRIME) based on user options and calls start_execution(), fixing the run duration to 10 iterations for averaging.

4. Result Handling:

   After execution, it prints the average global and per-thread results using print_all_info().

5. Data Visualization Prep:

   It calls write_json_output() from the output component to save the results in a file format required by the external Python visualization script.

   ```
   void write_json_output(all_MeasuredInfo info);
   ```

6. Cleanup:

   It frees the memory allocated

6 Testing & Validation

The Testing and Validation stage involves running the benchmark with different configurations and comparing the different outputs with each other. Because of the nature of this project the benchmark will differ from machine to machine so for the next secion tests will be run on an x64 Windows 11 version 25H2 with and 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz (2.80 GHz) processor and 16GB of RAM.

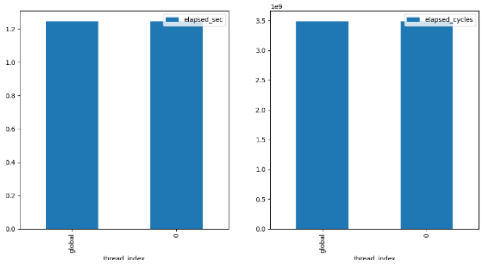
6.1 Testing different configurations

6.1.1 Tesing different number of threads

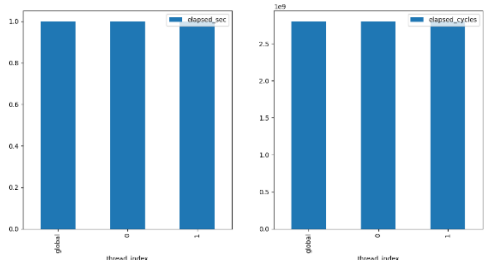
The benchmark should ideally behave by showing a decrease in execution time as the number of threads increases, demonstrating parallel speedup until a point of diminishing returns is reached due to overhead (synchronization, context switching, etc.).

These tests were run with de clock() instrucion, matrix algorithm, no io analysis and by varing the number of threads.

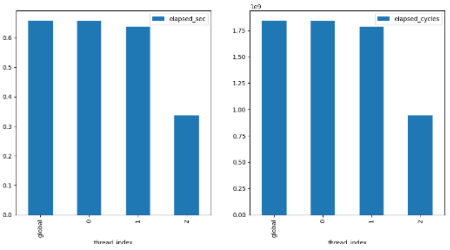
1 Thred



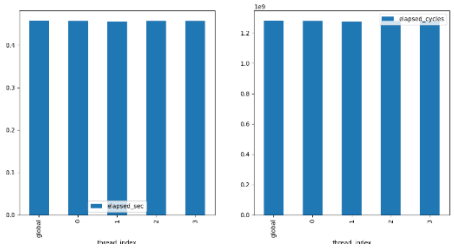
2 Threads



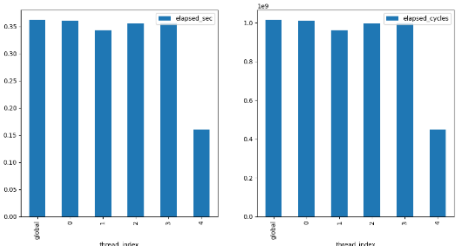
3 Threads



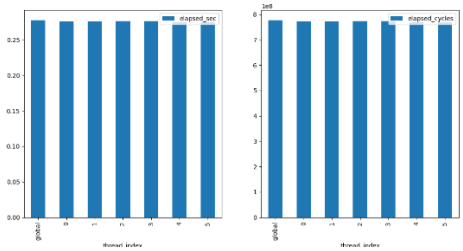
4 Threads



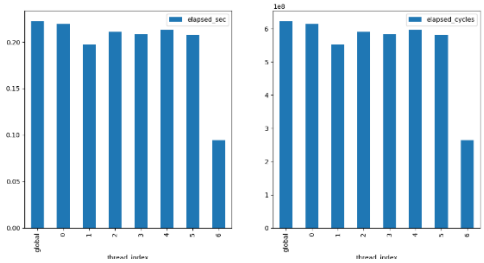
5 Threads



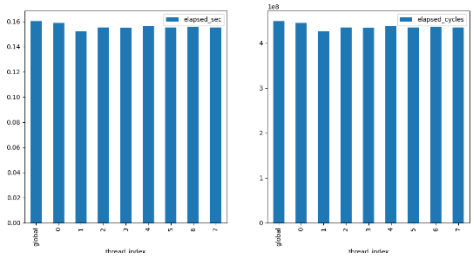
6 Threads



7 Threads



8 Threads



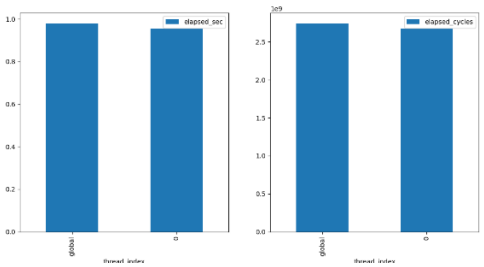
6.1.2 Testing different timing mechanisms

This secion will present a comparison between timing the benchmark with RDTSC and clock(). The overall difference in execution time will be minimal, especially because of the number of itterations made for each measurement and the arithmetic mean. The key difference lies in the precision and resolution of the RDTSC measurement wich can provides cycle-level granularity that is vital for profiling low-level, per-thread workload distribution and consistency, a sensitivity clock() entirely lacks.

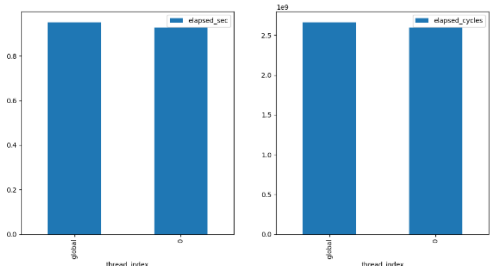
The tests will be run by using the sorting algorithm and by varing the number of threads and the timer.

1 Thread

RDTSC

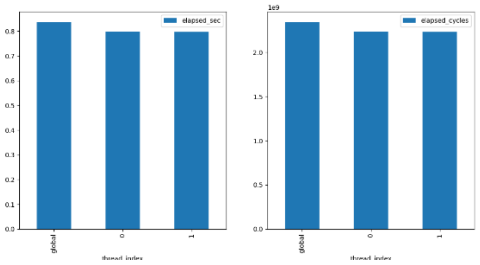


clock()

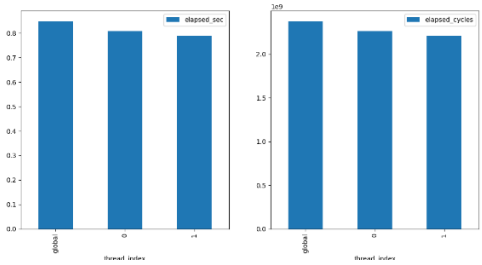


2 Threads

RDTSC

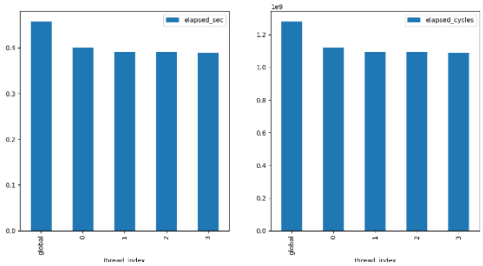


clock()

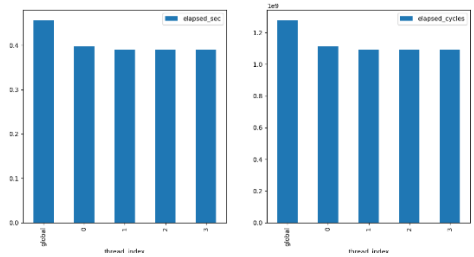


4 Threads

RDTSC

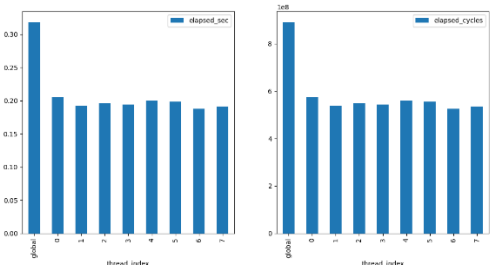


clock()

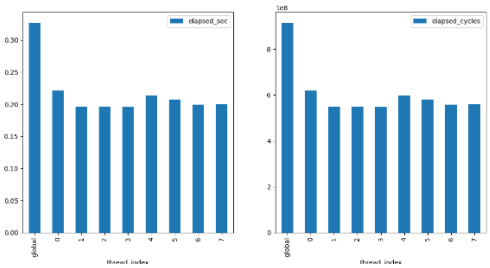


8 Threads

RDTSC



clock()

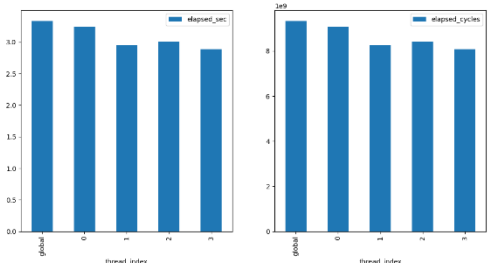


6.1.3 Testing different workloads

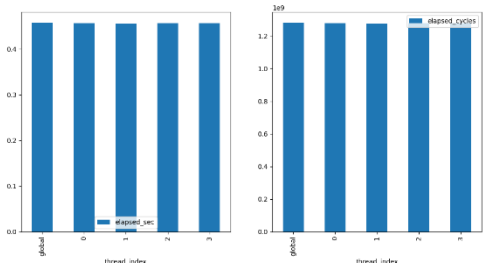
This section will present a comparative analysis of the performance results across the four distinct workload types: Calculating Prime Numbers, Parallel Merge Sort, Matrix Multiplication, and the I/O Workload. The computational algorithms should perform faster than the IO algorithm because of the slow file system. The merge sort should have lower per thread performance and higher global performance because of the final merge. Matrix should have minimal global overhead because the workload is almost purely pararell. Prime should have a slightly higher computation than the matrix and mergesort but smaller than the IO analysis, because of the thread synchronization.

The tests will be run by using the clock timer, with 4 threads and by varing the algorithm and activating/deactivating the IO analysis mode respectively.

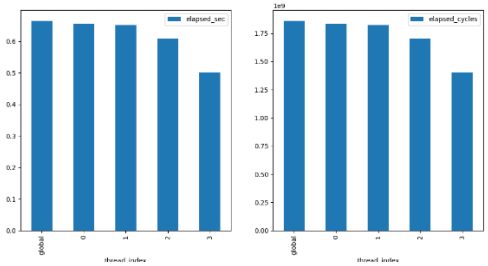
IO



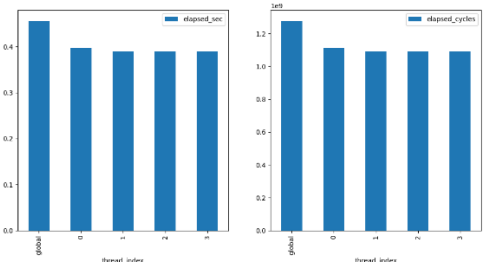
Matrix



Prime



Sorting



6.2 Validation

6.2.1 Input validation

The validation of the input is only necessary when the program is run from a comand line interface because from a graphical user interface , the Java frontend StartPanel handles the initial validation and parsing automatically.

1. The GUI restricts user input by design:

- Type Checking:

  Components like JcomboBox (for selecting algorithms or timers) ensure the input is one of the valid options, eliminating string parsing errors.

- Range/Format Checking:

  Components like JtextField (for the number of threads) can have validation logic built into the Java code before the command-line arguments are constructed and passed to the C++ executable program.exe.

2. Conversely, the C++ CLI must be self-sufficient and robust. The input_management function achieves this through:

- Immediate Defaults:

  If a flag or value is missing, safe defaults are instantly applied

- Non-Numeric Check:

  The form_number function performs explicit character-by-character validation to ensure the thread count is a digit, returning -1otherwise.

- Logical Correction (Warning):

  If a thread count is technically numeric but logically flawed (e.g., zero, negative, or exceeding hardware cores), a Warning is printed, and the thread count is reset to the safe hardware default.

- Conflict Resolution:

  A final check handles the logical conflict between the –io flag and computational settings, issuing a Warning and prioritizing the I/O mode.

6.2.2 Heap Allocation Failure

The benchmark program is designed to enforce system integrity by treating a heap allocation failure as a non-recoverable error. Rather than attempting complex, high-overhead recovery mechanisms, the program employs a defensive strategy that ensures immediate and clean termination.

This behavior is deliberately enforced by placing an explicit safety check immediately after every critical memory allocation (using malloc or calloc). Upon failure, the application performs a two-step clean exit:

- Print Specific Error:

  A specific message identifying the failed allocation point is printed to the console .

- Hard Termination:

  The application immediately executes exit(1).

The inclusion of the specific error message allows the user to instantly identify *which* vital resource allocation failed

```
if (matrix_a == NULL) {
  printf("Error: Memmory alocation of \"matrix_a\" failed\n");
  exit(1);
}
```

6.2.3 The Warning System

The program is designed for resilience using a warning-based error handling system, which ensures the benchmark runs even when users provide non-ideal or incorrect arguments. Instead of crashing, the system issues a warning and self-corrects to a safe, default configuration.

The warnings primarily cover three categories of non-critical input faults handled by input_management:

1. Invalid or Unsafe Thread Count:
   - If the user supplies a non-numeric value, a zero, or a thread count exceeding the detected hardware concurrency, the program prints a warning.
   - Correction: The --threads value is automatically reset to the safe hardware concurrency default, preventing potential instability or resource over-commitment.
2. Invalid Option Value:
   - If arguments for flags like --algorithm or --timer do not match the expected strings (e.g., specifying fast instead of matrix), a warning is printed.

- Correction: The program falls back to the defined default settings (matrix for algorithm and clock for timer type).

3. Logical Conflict:
    - If the user enables the standalone I/O mode (--io) but also specifies an algorithm or timer (which are ignored in I/O mode), a warning is printed to clarify the conflict.
    - Correction: The --io mode is prioritized, and the conflicting flags are effectively ignored, allowing the specialized I/O benchmark to run correctly.

This warning system prevents abrupt failures, making the standalone executable highly robust and user-friendly for direct command-line use.

Conclusion

At first glance, creating a reliable set of multithreaded benchmarks for modern multicore processors may appear to be a straightforward exercise in parallel programming and timing measurement. However, as the analysis, design, and implementation progressed, the number of subtle details quickly accumulated. Achieving accurate and reproducible timing in the presence of out-of-order execution, core migration, context switches and compiler optimizations proved far more challenging than initially expected.

Despite these difficulties, the primary objective has been fully achieved: a complete, open, and extensible benchmark suite capable of quantitatively evaluating the real-world multithreaded performance of x86-64 Windows systems has been designed, implemented, and thoroughly validated.

The resulting tool is immediately useful for students, researchers, and developers who need to understand how algorithms actually behave on contemporary multicore processors, far beyond theoretical speedup formulas.

Future extensions are numerous and straightforward: adding multiple algorithms and timing options, porting the backend to Linux/POSIX threads, integrating support for ARM64 or even implementing global leaderboards. The foundation is solid and the possibilities are effectively unlimited.

Bibliography

[1] SPEC's Benchmarks and Tools , https://www.spec.org/benchmarks.html

[2] CPUID , https://www.cpuid.com/softwares/cpu-z.html

[3] Legacy CPU Benchmarks , https://www.anandtech.com/bench/CPU/2

[4] Intel, Using the RDTSC Instruction for Performance Monitoring , https: //www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf

[5] Peter Kankowski, Performance measurements with rdtsc, https://www. strchr.com/performance_measurements_with_rdtsc

[6] John Lyon-Smith, Getting accurate per thread timing on Windows, https: //web.archive.org/web/20090510073435/http://lyon-smith.org/blogs/code-o-rama/ archive/2007/07/17/timing-code-on-windows-with-the-rdtsc-instruction.aspx