TÉCNICO LISBOA

# Instituto Superior Técnico

## Computação Paralela e Distribuída / Parallel and Distributed Computing

## MEEC

---

**Project Assignment: Traveling Salesperson - Branch and Bound - OpenMP, 2022/2023**

---

*Students:*
Tomás Marques Videira Fonseca | 66325
Afonso Brito Caiado Alemão | 96135
Rui Pedro Canário Daniel | 96317

*Group* **4**

*Teachers:*
José Carlos Alves Pereira Monteiro
Manuel Maria Gil Mata Ribeiro
Nicolas Guidotti
Nuno Roma

March 24, 2023

# 0 Introduction

This class project is a hands-on experience in parallel programming on both shared-memory and distributed-memory systems, using OpenMP and MPI, respectively. For this assignment, we write a sequential and a parallel implementation (OpenMP) of the program that solves the traveling salesperson problem, by computing the shortest route to visit a given set of cities. In the next assignment, we'll write another parallel implementation (MPI) of this program. The sequential program serves as the base for comparisons for both parallel versions. This optimization problem falls in the NP-Hard complexity class, meaning that all known solutions to solve it exactly imply an exhaustive search over all possible sequences.

# 1 Serial Implementation

## 1.1 Project architecture

The sequential code is divided into several files. `main.c` contains the main program; `tsp.c`, `tsp.h` is the serial implementation of TSP Branch and Bound; `structs.c`, `structs.h` manages the auxiliary structures of information; `queue.c`, `queue.h` contains the functions that operate on priority queues; `auxiliar.c`, `auxiliar.h` manages the auxiliar methods of the program; and it also contains a `Makefile`.

The graphical representation of the dependencies between files is given by the tree in Fig. 3 in the attachments.

## 1.2 Data Structures

The attention now turns to the data structures that were utilized to support this project.

**Inputs**: Used to represent the input parameters values. Thus, it contains an adjacency matrix to represent the graph, `adj_matrix`, where entry (`i`, `j`) corresponds to the distance (or cost) between cities `i` and `j`; two integer type variables, `n_cities` and `n_edges`, representing the number of cities and the total number of roads, respectively; the maximum cost value that can be accepted for the solution in a double type variable `max_value`; and the adjacent edges with the lowest and second lowest cost for each node, being represented, respectively, in two double arrays `min1` and `min2`. This said, `min1(i)` is seen as the adjacent road with the lowest cost for city `i` while `min2(i)` is the adjacent road with the second lowest cost for `i`.

**Solution**: Composed of an array of integers called `BestTour` that contains the shortest route to visit a specified set of cities, starting from city 0, and by a double type variable `BestTourCost` that indicates the cost associated with the route stored in `BestTour`. The values of all elements in the `BestTour` array are initialized to $-1$, which implies that the problem has no solution unless this structure is updated during the program's execution.

**Path**: This structure represents distinct branches of the search tree that the program generates while exploring every possible sequence of cities. Each instance features an integer array `Tour` that holds the sequence of cities in a particular branch of the tree. The `Tour` is initialized with all elements set to $-1$ except the element at index 0, which is always 0 as the path's starting point is city 0. It also includes a double variable `cost`, an integer `length`, an integer `node` with the last city added to the path, a double `bound` with the bound of the path.

**Priority Queue**: The queue comprises elements that are instances of the Path structure. It was already implemented and has `max_size` and `size` variables of type `size_t` that, respectively, define the upper limit of the number of elements that the queue can hold and the current number of elements. It also has a `buffer` which is an abstract type array utilized to represent the elements. In order to establish which of the elements have higher priority, it uses a compare function `cmpfn` that we implement. It is used so that the most promising paths can be examined first, and it gives priority to nodes with lower bound. When a tie occurs, compare the elements based on their index. Each pop returns the element with higher priority.

## 1.3 Algorithm

The algorithm implemented consists on the process of building a search tree that covers all possible sequences of cities. However, using estimated lower bounds, branches of this tree, i.e, paths in the queue, will be discarded when it is guaranteed that no solution better than the current one can be found down that branch.

The serial program starts by creating a new `queue` and a new path with only one city that is pushed to the queue. We initialize an integer array `isInTour` that will mark the elements in the current path (`isInTour(i)` equal to 0 if city `i` is in it, otherwise is equal to 1). Additionally, we use a variable of type Solution, `sol`, to keep the current best solution.

For each path, we compute its bound (detailed in the problem statement [2]). Then, there is a loop where at each iteration the first entry of the queue (the most promising path) is removed and analyzed. If its bound is higher than the best solution, the algorithm finishes, as all other entries in the queue have higher bounds than the current one and thus cannot be better than the solution we already have. In that case, we remove all of the remaining elements from the queue and free them. Otherwise, we check if this entry completes a tour, and if so, in case this tour cost is better than the previous best solution, we update `sol`. Otherwise, we discard this path.

If none of the previous conditions are verified, for each city not yet on the path tour, if its new bound cost is higher than the best solution cost, we discard this new path. Otherwise, we create a new path and insert it in the queue. Finally, following the end of the loop and the uncovering of a solution, all of the remaining elements from the queue are freed.

A pseudocode of the previously explained algorithm can be found in the problem statement [2].

## 1.4 Results

In this section, we measure program performance (time) for the serial implementation in several test files. Our execution was performed in the lab machines Lab2P4 and Lab3P10. The Lab2P4 has 16 GB of RAM, an Intel® Core™ i5-11500 @ 2.70GHz CPU, with 2 threads per core and 6 cores per socket, and supports simultaneous multithreading. The Lab3P10 has 16 GB of RAM, an Intel® Core™ i5-10500 @ 3.10GHz CPU, with 2 threads per core and 6 cores per socket, and supports simultaneous multithreading. The remaining specifications in Fig. 4 and in Fig. 5, respectively, can be found in the attachments.

We measure the execution time of `tsp()` with the routine `omp_get_wtime()` from OpenMP, which measures real-time (also known as "wall-clock time"). The experimental results obtained are reported in Table 1 in the attachments.

In order to obtain summary statistics of performance metrics we use the VTune profiler. By analyzing them, we identify the performance bottlenecks and hotspots, which are the methods involved in the manipulation of the queue (pops and pushes).

In the next section, we parallelize our program in order to actuate in the identified bottleneck hotspots, by using task decomposition, where independent `work()` is performed in different queues, parallelizing the manipulation of the queue.

# 2 OpenMP Implementation

In this section, we describe the OpenMP implementation of the algorithm. The project architecture is similar to the one in 1.1, by replacing the keyword `tsp` in the file names with `tsp-omp`.

Our goal is to make the parallelization as effective and as scalable as possible taking into account synchronization and load balancing. To test for scalability, we run this program assigning different values to the shell variable `OMP_NUM_THREADS`.

To ensure we still use a priority queue, we did not use the priority feature of tasks in OpenMP.

OpenMP is an open specification for multi-threaded, shared memory parallelism. It is a Standard Application Programming Interface (API), with a simple programming model that allows separating a program into serial and parallel regions, rather than concurrently executing threads. So we start by detecting dependencies to determine what can be parallelizable, prevent data races, and identify what accesses need to be synchronized.

In the OpenMP memory model, the concurrent programs access two types of data, shared data which is visible to all threads, and private data, which is visible to a single thread (often stack-allocated). Global variables are shared and local variables are private to each thread.

## 2.1 Approach for Parallelization

In this section, we describe our OpenMP implementation of TSP Branch and Bound.

Using the directive `#pragma omp parallel shared(BestTourCost, dealer)`, we create $N$ parallel threads and all of them execute the subsequent block and wait for each other at the end of it (implicit barrier synchronization). We define `BestTourCost` and `dealer` as shared variables that exist in a single location and all threads can read and write it.

Each thread gets its thread ID. Our main approach for parallelization is to create a queue for each thread. This is secured by a shared variable that is an array of pointers to queues. Each thread will be working in the queue associated with its thread ID. This decision was made in order to allow easy communication and resource manipulation between queues when needed (load balancing and synchronization).

First, using the directive `#pragma omp master` the first path element goes to the master thread queue. Then, we start with a short warm-up procedure where we run the algorithm with the master thread. So we process the elements in the master thread by popping them and execute `work()` on them. `work()` is the method that processes a popped path element from the queue and executes the problem algorithm, creating new possible paths and pushing these new elements to the queue.

Now, we detail the main steps of `work()`. First, check if all remaining nodes in the queue are worse than the current `BestTourCost` and, if they are, `work()` finishes its execution. Then, check if the path is a complete tour and if it is better than the better current solution. In that case, update the best solution tour and cost. The update of this shared structure requires a critical region to secure that a thread waits at the beginning of that block until no other thread is executing that section. Finally, if the path element is not a complete tour, generate all the possible paths from it, pushing them to the queue.

That way, `work()` can be performed by multiple threads at the same time in their associated queues and we use it to perform task decomposition (detailed in 2.2).

This warm-up procedure is executed until either the algorithm finishes or the queue size is large enough to evenly distribute elements (1 or 2) among the queues of the other threads. This queue size threshold and the number of distributed elements were optimized through a calculation that uses the density of the graph in question, defining it as half the quotient between the number of edges and the number of nodes. To check if the algorithm has finished we can verify if there are only irrelevant elements in the master queue (`flag = 1`) or if the master queue was emptied (`queue[tid]->size = 0`).

Then, the master thread verifies if the algorithm has finished. Otherwise, after synchronizing all threads, we evenly distribute one or two elements among the queues of the other threads, in order to assign the task decomposition and obtain performance gain through parallelization. The number of distributed elements depends on the master thread queue size and the distribution of the elements is made taking into consideration a balance relative to the priority of each element: current empty threads receive the elements with the highest priorities.

Then comes the most important part of the program: each thread processes its queue according to the algorithm until the queue is empty or there are only irrelevant elements in it, for the queues of **all** threads. We need to account for load balancing, by distributing work to threads that have already finished processing their queue. The description of this process is in sections 2.3 and 2.4.

Finally, we free the irrelevant elements in all queues and the auxiliary structures and check if a valid solution was found.

## 2.2 Used Decomposition

Decomposition is a key concept in OpenMP programming, as it allows developers to exploit the full power of multicore and multiprocessor systems to achieve significant performance gains. It is very important due to Amdahl's Law which states that the dependencies limit maximum speedup due to parallelism.

Task decomposition involves breaking down a large computation into smaller tasks that can be executed concurrently by different threads. It is required that these tasks can be carried out in parallel: task decomposition is performed implicitly and not explicitly since we do not use any task parallelization directive. We need to create at least enough tasks to keep all execution units busy. The key challenge of decomposition is to identify dependencies, decomposing a program into independent tasks.

In the last section, we define `work()` as the processing of a path and the execution of the problem algorithm for that element, creating new possible paths and pushing these new elements to the queue. This `work()` is always independent from path to path because each possible path is unique and leads to unique path pushes.

Furthermore, when a thread executes a `work()` it will operate in a queue that is associated with its thread ID. So we do not need to use critical sections in these queue pushes and pops.

The only non-independent section that needs a critical section is the update of the best solution found so far. This way, `work()` is considered a task. As such, these tasks can be carried out in parallel, keeping all execution units busy. Additionally, when a thread exhausts its tasks it will receive `work()` from other threads, as we will describe in the following sections.

## 2.3 Synchronization

Thread synchronization is crucial in multi-threaded programming to ensure that concurrent threads coordinate and share resources in a controlled and orderly manner, preventing conflicts and guaranteeing the correctness of the program's behavior. Without proper synchronization, threads can interfere with each other, leading to race conditions, deadlocks and other concurrency bugs that can cause the program to crash or produce incorrect results. Therefore, proper synchronization of threads plays a vital role in ensuring the reliability, safety and performance of concurrent programs.

This said, our program contains several instances where thread synchronization is necessary. These synchronization points have been carefully designed and implemented to prevent the previously enumerated problems. The first time we use it, it appears in the form of a `#pragma omp barrier` directive, after the warm-up section that was earlier discussed, where all threads wait for each other at the end of the executing block. This is mandatory since all threads need to know, before continuing executing the program, if the master thread has finished the algorithm (`exit_global = 1`) or if the master thread queue size is large enough to evenly distribute elements (1 or 2) among the queues of the other threads. If this distribution happens, we also need to synchronize after that.

Moreover, synchronization is also used in the code section dedicated to threads that are giving away elements to a whistling thread whose queue is empty (described in more detail in section 2.4). This happens in a `#pragma omp critical` region where it is executed a `queue_pop()` of the most priority element from the donator thread queue, and that element is pushed into the empty thread queue.

Another critical section can be encountered in the region of code where a thread, after verifying that its queue is empty and that the algorithm has not yet finished, decides to become the next whistling thread by updating the shared variable `whistle` with its thread ID.

When updating the best solution tour and cost, both stored in `sol` structure since this is a shared structure, a critical region is indispensable in order to secure that a thread waits at the beginning of that block until no other thread is executing that section.

Finally, there is a barrier before the free of a thread queue, in order to secure that the still working threads still can access the size of the thread queues that already finished their execution. Otherwise, this could lead to memory access problems.

## 2.4 Load Balancing

Load balancing is a technique used in multi-threaded programs to distribute the workload evenly among threads, thereby optimizing the performance of the system. In this approach, the total workload is divided into smaller chunks and assigned to different threads, to evenly distribute the workload among the threads. This way, we establish implicit tasks.

In the program, when a thread finishes the execution of its tasks it asks for elements from the other threads in order to process them. By doing this, it becomes the whistling thread. This is achieved by defining a shared variable `whistle` that represents the thread ID of the current whistling thread, i.e, the thread asking for elements. So whenever a thread wants to ask for elements, it changes the value of the `whistle` to its thread ID.

Then the other threads that are still running the algorithm and have elements in their queue will check if there is a whistling thread (`whistle != -1`). If this condition is verified, one of the threads, whose ID is equal to `global_tid` (shared variable), will give their most priority element to the whistling thread. The variable `global_tid` starts with an initial value of 0 because it is the master's ID. Following the course of the program, `global_tid` will change when necessary, for example, when the thread whose ID is the `global_tid` can't give more elements (because its queue size is 0) or when the whistling thread ID is equal to the `global_tid`. After receiving the element from another thread, the whistling thread resets the value of `whistle` to -1, to prevent receiving more elements from other threads.

This process requires synchronization which was described in section 2.3.

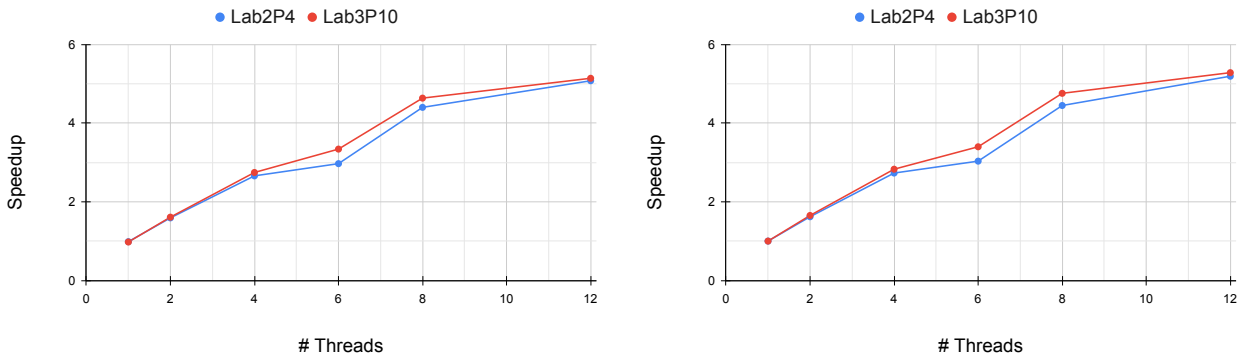
## 2.5 Results

In this section, we measure program performance (time) for the parallel OpenMP implementation in several test files. Our execution was performed in the lab machines Lab2P4 and Lab3P10. The specifications for the workstations Lab2P4 and Lab3P10 are reported in Fig. 4 and Fig. 5 in the attachments. We measure the execution time of `tsp-omp()` with the routine `omp_get_wtime()` from OpenMP, which measures real-time (also known as "wall-clock time").

In Tables 2 and 3 in the attachments it is reported the average time taken to run each test on Lab2P4 and Lab3P10 workstations, respectively, for a different number of threads. Then, in Tables 4 and 5 in the attachments it is reported the speedups against the serial version on Lab2P4 and Lab3P10 workstations, respectively. The speedups against a single thread OMP version on Lab2P4 and Lab3P10 workstations are also reported in Tables 6 and 7 in the attachments. We do not report speedups for the test files `ex1-20.in`, `ex2-40.in`, `gen10-20.in` and `gen15-25.in` because the performance times are too small, and as we are rounding to one decimal digit, that can cause division by 0.

We define efficiency as the quotient between the speedup and the minimum between the number of cores and the number of threads defined in `OMP_NUM_THREADS`. The obtained efficiency for each thread in these workstations is reported in Tables 4, 5, 6 and 7, by using the average speedup.

We obtain the average speedups by thread, through averaging the relative speedups obtained for each file test by thread and in Fig. 1 we report the average speedup obtained in function of different number of threads. We also check the results for 3, 5, 7, 9, 10 and 11 threads and they align with Fig. 1.




(a): Against the serial version. (b): Against the OMP version with a single thread.

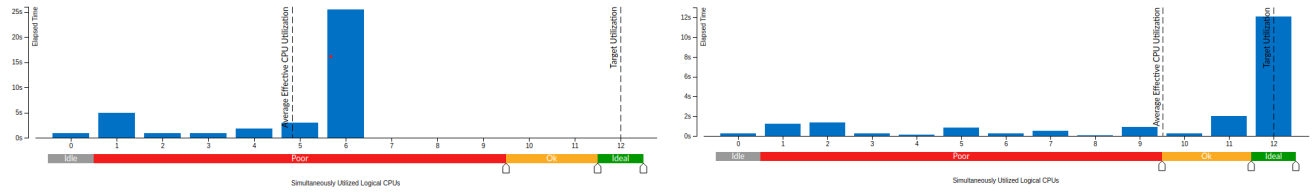
Figure 1: Average speedups obtained in function of different number of threads.

When we run the program using more than 1 thread, we obtain an average of half the maximum possible performance achievable ideally. This is due to Amdahl's Law which states that dependencies limit maximum speedup due to parallelism, i.e, the serial fraction of a program limits the maximum speedup achievable in the parallelization.

In ideal conditions, with an infinitesimal fraction of serial execution for the program, the speedup would be equal to the number of threads (linear). However, the speedup can sometimes reach far beyond the limited linear speedup, known as super-linear speedup. One possible reason for that super-linear speedup in low-level computations is the cache effect resulting from the different memory hierarchies of a modern computer: in parallel computing, there are accumulated caches from different processors.

From 1 to 6 threads, when we increase the number of threads the efficiency decreases due to Amdahl's Law. But for 6 to 12 threads the efficiency increases because each core still has the same number of ALU resources, but multi-threading helps use them more efficiently in the face of high-latency operations like memory access (hide latency). For more than 12 threads the performance is worse as expected because the workstations only have 12 logical cores, and in this case there is overhead to create, resume, manage, suspend, and destroy additional threads that are not useful.

Finally, in order to obtain summary statistics of performance metrics we use the VTune profiler in workstation Lab3P10 for 6 and 12 threads. The report of the effective CPU utilization histograms obtained is in Fig. 2. This histogram displays a percentage of wall time the specific number of CPUs were running simultaneously, Spin and Overhead time adds to the Idle CPU utilization value. For both cases, the program achieves an almost ideal average effective CPU utilization.




(a): 6 threads. (b): 12 threads.

Figure 2: Effective CPU utilization histograms using a different number of threads.

# 3 Conclusion

Initially, we write a sequential implementation for the problem and we analyze the statistics of performance metrics identifying the bottleneck hotspots, that are related to the manipulation of the queue.

Then, we implement a parallel version of the program, using OpenMP and we obtain the results in 2.5. In order to actuate in the identified bottleneck hotspots, we use task decomposition, where independent `work()` is performed in different queues, parallelizing the manipulation of the queues. During the implementation of the program, we consider several factors that impact its performance such as the performance of the algorithm and its implementation, the fraction of the code that runs in parallel, and the amount of synchronization and thread management (overhead to create, resume, manage, suspend, and destroy them). The amount of synchronization covers load balancing and data organization (data locality, data independence, memory conflicts, effective data sharing, and load balancing). The program is also optimized for scalability and performance, minimizing forks/joins, minimizing synchronization, and maximizing private data. We also perform loop fusion in order to reduce synchronization and scheduling overhead.

# References

[1] Parallel and Distributed Computing Slides - Slides of Theoretical Classes 2022/2023, 1st Semester (MEEC), by José Monteiro;

[2] 1st Semester 2022/2023, Parallel and Distributed Computing (IST, 2022-23), Project Assignment Traveling Salesperson - Branch and Bound;

# Attachments

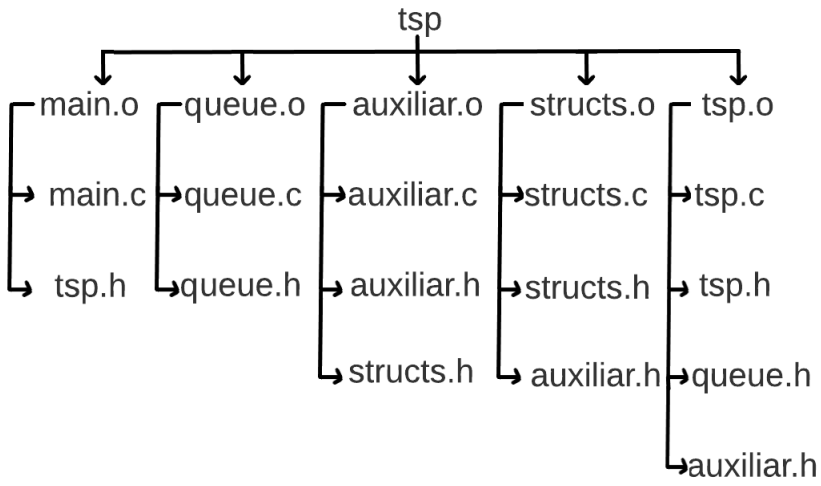



Figure 3: Program dependencies.

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          12
On-line CPU(s) list:             0-11
Thread(s) per core:              2
Core(s) per socket:              6
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           167
Model name:                      11th Gen Intel(R) Core(TM) i5-11500 @ 2.70GHz
Stepping:                        1
CPU MHz:                         2700.000
BogoMIPS:                        5424.00
Virtualization:                  VT-x
L1d cache:                       288 KiB
L1i cache:                       192 KiB
L2 cache:                        3 MiB
L3 cache:                        12 MiB
NUMA node0 CPU(s):               0-11
Vulnerability Itlb multihit:     Not affected
Vulnerability L1tf:              Not affected
Vulnerability Mds:               Not affected
Vulnerability Meltdown:          Not affected
Vulnerability Mmio stale data:   Mitigation; Clear CPU buffers; SMT vulnerable
Vulnerability Retbleed:          Mitigation; Enhanced IBRS
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl and seccomp
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2:        Mitigation; Enhanced IBRS, IBPB conditional, RSB filling, PBRSB-eIBRS SW sequence
Vulnerability Srbds:             Not affected
Vulnerability Tsx async abort:   Not affected
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts
                                 acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art ar
                                 ch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_f
                                 req pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdc
                                 m pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdran
                                 d lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single ssbd ibrs ibpb stibp ibrs
                                 _enhanced tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2
                                  smep bmi2 erms invpcid mpx avx512f avx512dq rdseed adx smap avx512ifma clflushopt i
                                 ntel_pt avx512cd sha_ni avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves dtherm arat
                                  pln pts avx512vbmi umip pku ospke avx512_vbmi2 gfni vaes vpclmulqdq avx512_vnni avx
                                 512_bitalg avx512_vpopcntdq rdpid fsrm md_clear flush_l1d arch_capabilities
```

Figure 4: CPU specifications of the Lab2P4 workstation.

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          12
On-line CPU(s) list:             0-11
Thread(s) per core:              2
Core(s) per socket:              6
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           165
Model name:                      Intel(R) Core(TM) i5-10500 CPU @ 3.10GHz
Stepping:                        3
CPU MHz:                         3100.000
CPU max MHz:                     4500.0000
CPU min MHz:                     800.0000
BogoMIPS:                        6199.99
Virtualization:                  VT-x
L1d cache:                       192 KiB
L1i cache:                       192 KiB
L2 cache:                        1.5 MiB
L3 cache:                        12 MiB
NUMA node0 CPU(s):               0-11
Vulnerability Itlb multihit:     KVM: Mitigation: VMX disabled
Vulnerability L1tf:              Not affected
Vulnerability Mds:               Not affected
Vulnerability Meltdown:          Not affected
Vulnerability Mmio stale data:   Mitigation; Clear CPU buffers; SMT vulnerable
Vulnerability Retbleed:          Mitigation; Enhanced IBRS
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl and seccomp
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2:        Mitigation; Enhanced IBRS, IBPB conditional, RSB filling, PBRSB-eIBRS SW sequence
Vulnerability Srbds:             Mitigation; Microcode
Vulnerability Tsx async abort:   Not affected
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts
                                 acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art ar
                                 ch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulq
                                 dq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 s
                                 se4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3
                                 dnowprefetch cpuid_fault epb invpcid_single ssbd ibrs ibpb stibp ibrs_enhanced tpr_s
                                 hadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms
                                  invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dthe
                                 rm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp md_clear flush_l1d arch_ca
                                 pabilities
```

Figure 5: CPU specifications of the Lab3P10 workstation.

Table 1: Serial implementation time (s) results for the Lab2P4 and Lab3P10 workstations.

| Test [.in] \ Workstation | Lab2P4 | Lab3P10 |
|---|---|---|
| ex1-20 | 0.0 | 0.0 |
| ex2-40 | 0.0 | 0.0 |
| gen10-20 | 0.0 | 0.0 |
| gen15-25 | 0.2 | 0.2 |
| gen19-23 | 2.9 | 2.1 |
| gen20-5000 | 131.2 | 105.1 |
| gen22-25000 | 191.6 | 155.5 |
| gen24-50000 | 90.4 | 76.0 |
| gen26-50000 | 92.3 | 71.4 |
| gen30-5000 | 298.7 | 232.0 |

Table 2: Parallel implementation average time (s) taken to run each test on Lab2P4 workstation.

| Test [.in] \ # Threads | 1 | 2 | 4 | 6 | 8 | 12 |
|---|---|---|---|---|---|---|
| ex1-20 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| ex2-40 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| gen10-20 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| gen15-25 | 0.2 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 |
| gen19-23 | 3.0 | 2.1 | 1.0 | 1.0 | 0.9 | 0.6 |
| gen20-5000 | 132.4 | 83.6 | 39.7 | 43.2 | 31.7 | 24.2 |
| gen22-25000 | 195.0 | 125.0 | 79.3 | 57.4 | 41.0 | 46.0 |
| gen24-50000 | 91.5 | 45.1 | 41.7 | 36.6 | 19.4 | 17.0 |
| gen26-50000 | 94.8 | 57.3 | 28.6 | 24.6 | 19.7 | 20.0 |
| gen30-5000 | 302.5 | 172.1 | 136.3 | 142.1 | 110.3 | 52.5 |

Table 3: Parallel implementation average time (s) taken to run each test on Lab3P10 workstation.

| Test [.in] \ # Threads | 1 | 2 | 4 | 6 | 8 | 12 |
|---|---|---|---|---|---|---|
| ex1-20 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| ex2-40 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| gen10-20 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| gen15-25 | 0.2 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 |
| gen19-23 | 2.2 | 1.5 | 0.7 | 0.8 | 0.7 | 0.5 |
| gen20-5000 | 107.4 | 65.9 | 31.2 | 30.9 | 22.4 | 17.7 |
| gen22-25000 | 157.0 | 100.4 | 62.5 | 40.8 | 29.8 | 34.6 |
| gen24-50000 | 77.0 | 35.9 | 31.8 | 23.2 | 12.5 | 12.5 |
| gen26-50000 | 73.6 | 44.0 | 22.4 | 19.4 | 15.6 | 15.6 |
| gen30-5000 | 237.9 | 136.4 | 107.6 | 112.5 | 82.4 | 40.7 |

Table 4: Speedups and efficiencies obtained, against the serial implementation, on Lab2P4 workstation.

| Test [.in] \ # Threads | 1 | 2 | 4 | 6 | 8 | 12 |
|---|---|---|---|---|---|---|
| gen19-23 | 0.97 | 1.38 | 2.90 | 2.90 | 3.22 | 4.83 |
| gen20-5000 | 0.99 | 1.57 | 3.30 | 3.04 | 4.14 | 5.42 |
| gen22-25000 | 0.98 | 1.53 | 2.42 | 3.34 | 4.67 | 4.17 |
| gen24-50000 | 0.99 | 2.00 | 2.17 | 2.47 | 4.66 | 5.32 |
| gen26-50000 | 0.97 | 1.61 | 3.23 | 3.75 | 4.69 | 4.62 |
| gen30-5000 | 0.99 | 1.74 | 2.19 | 2.10 | 2.71 | 5.69 |
| Average Speedup | 0.99 | 1.59 | 2.66 | 2.97 | 4.40 | 5.08 |
| Efficiency [%] | 98.5 | 79.5 | 66.5 | 49.5 | 73.3 | 84.6 |

Table 5: Speedups and efficiencies obtained, against the serial implementation, on Lab3P10 workstation.

| Test [.in] \ # Threads | 1 | 2 | 4 | 6 | 8 | 12 |
|---|---|---|---|---|---|---|
| gen19-23 | 0.95 | 1.40 | 3.00 | 2.63 | 3.00 | 4.20 |
| gen20-5000 | 0.98 | 1.59 | 3.37 | 3.40 | 4.69 | 5.94 |
| gen22-25000 | 0.99 | 1.55 | 2.49 | 3.81 | 5.22 | 4.49 |
| gen24-50000 | 0.99 | 2.12 | 2.39 | 3.28 | 6.08 | 6.08 |
| gen26-50000 | 0.97 | 1.62 | 3.19 | 3.68 | 4.58 | 4.58 |
| gen30-5000 | 0.98 | 1.70 | 2.16 | 2.06 | 2.82 | 5.70 |
| Average Speedup | 0.98 | 1.61 | 2.74 | 3.34 | 4.63 | 5.14 |
| Efficiency [%] | 97.7 | 80.4 | 68.6 | 55.6 | 77.2 | 85.6 |

Table 6: Speedups and efficiencies obtained, against a single thread parallel implementation, on Lab2P4 workstation.

| # Threads / Test [.in] | 1 | 2 | 4 | 6 | 8 | 12 |
|---|---|---|---|---|---|---|
| gen19-23 | 1.00 | 1.43 | 3.00 | 3.00 | 3.33 | 5.00 |
| gen20-5000 | 1.00 | 1.58 | 3.34 | 3.06 | 4.18 | 5.47 |
| gen22-25000 | 1.00 | 1.56 | 2.46 | 3.40 | 4.76 | 4.24 |
| gen24-50000 | 1.00 | 2.03 | 2.19 | 2.50 | 4.72 | 5.38 |
| gen26-50000 | 1.00 | 1.65 | 3.31 | 3.85 | 4.81 | 4.74 |
| gen30-5000 | 1.00 | 1.76 | 2.22 | 2.13 | 2.74 | 5.76 |
| Average Speedup | 1.00 | 1.62 | 2.73 | 3.03 | 4.45 | 5.19 |
| Efficiency [%] | 100.0 | 81.0 | 68.2 | 50.5 | 74.1 | 86.5 |

Table 7: Speedups and efficiencies obtained, against a single thread parallel implementation, on Lab3P10 workstation.

| # Threads / Test [.in] | 1 | 2 | 4 | 6 | 8 | 12 |
|---|---|---|---|---|---|---|
| gen19-23 | 1.00 | 1.47 | 3.14 | 2.75 | 3.14 | 4.40 |
| gen20-5000 | 1.00 | 1.63 | 3.44 | 3.48 | 4.79 | 6.07 |
| gen22-25000 | 1.00 | 1.56 | 2.51 | 3.85 | 5.27 | 4.54 |
| gen24-50000 | 1.00 | 2.14 | 2.42 | 3.32 | 6.16 | 6.16 |
| gen26-50000 | 1.00 | 1.67 | 3.29 | 3.79 | 4.72 | 4.72 |
| gen30-5000 | 1.00 | 1.74 | 2.21 | 2.11 | 2.89 | 5.85 |
| Average Speedup | 1.00 | 1.65 | 2.83 | 3.40 | 4.76 | 5.28 |
| Efficiency [%] | 100.0 | 82.6 | 70.7 | 56.6 | 79.3 | 88.0 |