Masivo: Parallel Simulation Model Based on OpenCL for Massive Public Transportation Systems’ Routes

Juan Ruiz-Rosero 1,*, Gustavo Ramirez-Gonzalez 1 and Rahul Khanna 2

1 Departamento de Telemática Universidad del Cauca, Calle 5, No. 4-70, Popayan, Cauca 190002, Colombia; gramirez@unicauca.edu.co
2 Intel Corporation, 2111 NE 25th Ave., Hillsboro, OR 97124, USA; rahul.khanna@intel.com
* Correspondence: jpabloruiz@unicauca.edu.co; Tel.: +57-(2)-820-9900

Received: 19 November 2019; Accepted: 3 December 2019; Published: 8 December 2019

Abstract: There is a large number of tools for the simulation of traffic and routes in public transport systems. These use different simulation models (macroscopic, microscopic, and mesoscopic). Unfortunately, these simulation tools are limited when simulating a complete public transport system, which includes all its buses and routes (up to 270 for the London Underground). The processing times for these type of simulations increase in an unmanageable way since all the relevant variables that are required to simulate consistently and reliably the system behavior must be included. In this paper, we present a new simulation model for public transport routes’ simulation called Masivo. It runs the public transport stops’ operations in OpenCL work items concurrently, using a multi-core high performance platform. The performance results of Masivo show a speed-up factor of 10.2 compared with the simulator model running with one compute unit and a speed-up factor of 278 times faster than the validation simulator. The real-time factor achieved was 3050 times faster than the 10 h simulated duration, for a public transport system of 300 stops, 2400 buses, and 456,997 passengers.

Keywords: simulation; parallel; multi-core; public transport; mass transit; OpenCL

1. Introduction

Millions of more people are expected to migrate to urban centers in the following years. The United Nations predicts that for 2050, 66% of the population will live in urban areas [1]. Hence, transportation planners are embracing new ways of tackling the old problem of congestion. Public Transport Systems (PTS) move a large number of people while consuming fewer resources (lower energy requirements, lower accident cost, and air quality improvements). Currently, one of the top applications of the Internet of Things (IoT) is the development of smart cities [2]. Smart cities target the increasing capacity to collect a large quantity of data from a PTS and process them in real-time (big data processing). This capacity has become the basis for the development of more effective transit modeling and simulation [3]. An efficient transit simulation powers the optimization agents for public transport systems. These optimization agents improve the route path, route frequency, number of transfers, the waiting time, bus or train sizes, ticket price, and others. With these optimizations, different studies reduced the wait time by 75% [4], the travel time by 14% [5], and operational cost by 11% [5].

The basis of a PTS simulation is system modeling. This modeling is divided into three different approaches: macroscopic, mesoscopic, and microscopic [6]. The macroscopic models aggregate traffic streams without analyzing the internal elements. The mesoscopic ones model homogeneous groups of actors and the interactions with little detail. Finally, the microscopic ones model in detail the dynamics of each actor (vehicles or passengers) and their interactions with others [6]. The public transport system components have been simulated inside these three different models. The macroscopic modeling has been used for simulating the vehicle accumulation of public transport vehicles along with private...
cars [7]; furthermore, for the modeling of the public transport vehicles’ speed [8] and for the road safety impacts of public transport [9]. Similarly, the mesoscopic models have been used for simulating the crowding levels on-board the public transport vehicles [10], emergency evacuation methodologies [11], traffic safety policy evaluation [12], and a multi-modal mobility simulator for large scale scenarios [13]. The microscopic models are the most used for public transport simulation. These models are included in the simulation of the operations at public transport stops [14,15], the public transport system’s reliability [16,17], bus priority methods [18,19], exclusive bus lanes [20,21], bus routes [22], transfer organization of public transportation terminals [23], multimodal systems [24], and traffic networks [25].

Similarly, we performed a literature review supported by the scientometric tool ScientoPy [26] to find the most simulated and modeled components in the PTS. For this, we downloaded a dataset from the Clarivate Web of Science (WoS) and Scopus, with the search string (“public trans*” OR “mass transit”) AND (“simulation” OR “model*”) applied to the documents’ title. Figure 1 shows the top seven most simulated or modeled components for PTS. In-door air quality had the most documents, with papers that included diseases transmission [27] and air quality prediction [28–33]. Second, accessibility included modeling the potential effect of shared bicycles on public transport travel times [34] and modeling access to PTS [35–37]. Line planning simulation and modeling covered models and methods for line planing [38], passenger routing decision [39,40], and PTS connection simulation [41].

Waiting time was the topic with the highest growth in PTS simulation and modeling, with 60% of the documents published in the last two years. It includes impacts on passengers’ waiting time uncertainty [42], timetable optimization models for minimizing passenger waiting [43], and a stochastic model for the passenger arrival process [44]. Finally, other simulation and modeling components are passenger flow [45–48], transfers [49–52], and path choice [53,54].

These kinds of implementations suffer from two critical issues. First is scalability, because most of them only offer single machine solutions that are not adequate to process large scale data. Second is granularity, due them not considering microscopic traffic situations such as individual passengers/bus/train modeling [55]. For these reasons, novel works have developed new models and simulators that employ parallel architectures to get the simulation results more efficiently. Currently, we find mesoscopic [13,56–60] and microscopic [61] traffic simulators based in GPUs (Graphics Processing Units), HPC (High Performance Computing) [62–64], and microscopic simulation based on

![Figure 1. Number of documents related to Public Transport Systems’ (PTS) components simulated or modeled reported by the scientific literature in the WoS and Scopus databases.](image-url)
Apache Spark [55,65]. Nevertheless, these modeling and simulation approaches do not consider each passenger as a unique entity in the simulation model.

In this paper, we present Masivo. Masivo is a new parallel simulation model based in OpenCL using a multi-core high performance platform for massive public transportation systems. Masivo works with predefined public transport system conditions, which include the stops’ total number, the stops’ capacity, and the Origin-Destination matrix (OD). This OD matrix and routes’ information are updated to this model via CSV files. Masivo gets the simulation results for total alighted passengers and average commute time. Similarly, it shows the performance indicators.

We organized this paper as follows. First, “Section 2. Public Transport Systems’ Basis” describes the public transport system, the simulation variables, and output indicators. Then, in “Section 3. Masivo Public Transport Routes’ Simulation”, we present the components, the input parameters, the output results, and the algorithms for the Masivo simulator. In “Section 4. Results”, we summarize the validation and performance results of Masivo. Later, in “Section 5. Discussion”, we discuss Masivo’s performance against other parallel traffic and public transport simulators. Finally in “Section 6. Conclusions”, we summarize all the concluding remarks and future work.

2. Public Transport Systems’ Basis

In a public transport system, the buses, trains, or other forms of transport are used by people to travel from one place to another [66]. Figure 2 shows a small public transport system example, which consists of three stops (S$_i$) separated 2 km and two routes (R$_1$ and R$_2$). In this system, we have, for instance, passengers from the origin stop S$_1$ that travel to the destination stops S$_2$ and S$_3$. The origin-destination matrix (OD) denoted as D represents the number of passengers that travel from each stop to another. For example, if we have 10 passengers that travel from S$_1$ to S$_2$, the value D$_{1,2}$ is 10.

\[
D = \begin{bmatrix}
0 & 10 & 20 \\
10 & 0 & 10 \\
20 & 10 & 0
\end{bmatrix}
\]

The routes are designed to transport passengers from one stop to another. Figure 2 shows two routes (R$_1$ and R$_2$). R$_1$ has the direction west to east, and R$_2$ has the direction east to west. The RT$_r$ value defines the total trip time (in seconds) for route r, that is the time needed by the bus to complete the full route. The value $f_r$ defines the frequency (in seconds$^{-1}$) of the route r, and it represents how often the bus is dispatched for each route. Therefore, the number of buses needed for each route is defined by Equation (2) as $BR_r$, where the symbols “$\lceil$” and “$\rceil$” represent a ceiling function that returns the nearest integer that is greater.

\[
BR_r = \lceil RT_r f_r \rceil
\]

For instance, we need an R$_1$ frequency of 5 min ($f_r = 1/300$ s$^{-1}$), and the total trip is 570 s, defined by Equation (3), where N is the number of stops:
\[ RT_1 = \frac{\text{Total distance}}{\text{Bus avg. speed}} + (\text{Stop time}) \times N \]

\[ RT_1 = \frac{4 \text{ km}}{30 \text{ km/h}} + (30 \text{ s}) \times 3 \]

\[ RT_1 = 570 \text{ s} \]

As a result, the number of buses needed for \( R_1 \) for this case is defined by Equation (4) as:

\[ BR_r = \left\lceil \frac{570 \text{ s}}{300 \text{ s}} \right\rceil \]

\[ BR_r = 2 \]  

As we can see in Equation (2), if we increase the frequency of a route, we will need to have more buses to meet the demand. Nevertheless, the number of buses is limited by the total fleet size or even by the roads’ capacity. Then, to find the route’s frequency, we have an optimization problem described by Equation (5), where the objective is to minimize the total travel time, restricted by the total fleet size [67].

\[
\text{minimize} \sum_{i \in N} \sum_{j \in N} D_{ij}T_{ij}(f) \\
\text{subject to:} \sum_{r \in R} [RT_rf_r] \leq \text{total fleet size}
\]

where:

- \( N \) = total stops;
- \( R \) = total routes;
- \( D_{ij} \) = OD (Origin-Destination) demand;
- \( T_{ij} \) = passenger travel time;
- \( RT_r \) = round-trip time on route \( r \);
- \( f_r \) = frequency on route \( r \);

The passenger travel time \( T_{ij} \) includes the expected waiting time, as a function of the routes’ frequencies \( f \). Unfortunately, this travel time cannot be calculated directly, because it depends on other variables such as the buses’ capacity, the stops’ capacity, and the routes’ stops’ table. Furthermore, the buses’ free seats depend on the buses’ capacity and the number of passengers that have boarded the buses at the previous stops. This correlation between the buses’ free seats with other variables of the systems converts this into an NP-hard problem, where the approach to this problem relies on heuristic techniques for systems of a reasonable size [67].

2.1. Passenger Demand in Public Transport Systems

The Origin-Destination matrices (OD) specifies the passengers’ travel demands between the origin and destination nodes in the network. Geographic Information Systems (GIS) have been used effectively for OD data analysis, supporting geocoding OD survey data to point locations [68]. The OD matrix is dynamic because it is time varying. Inside a city’s public transport system, the OD matrix varies on weekdays and weekends and even between different hours of the day. A high quality OD matrix is a fundamental prerequisite for any serious transport system analysis. Surveys have been done to define the OD matrices [68,69]; nevertheless, more sophisticated methods are used today for this purpose, such as passive smart card fare collection system data [70,71] or mobile network data [72].

2.2. Public Transport Vehicle Routes

In public transport systems, the vehicles (trains or buses) follow a route to meet the passengers’ demand. The route is defined by a set of stops, where the bus picks up or alights passengers.
Furthermore, the route’s frequency is defined as how often the bus is dispatched for each route. That also means how frequent a bus route passes one stop. Due to the dynamic nature of the OD, one of the essential components in determining the public transport service is the selection of the most suitable routes’ frequency by time-of-day and day-of-week. The routes’ design also includes the development of timetables, scheduling buses, and scheduling drivers. The route layout design depends on the passenger flow because routes are configured to provide a direct or indirect connection between origin-destination demand for passengers.

2.3. Traffic Simulation Software

There is much literature that deals with topics about the simulation of traffic networks especially focused on traffic simulation of private vehicles [73–81]. This literature presents simulation systems based on the different models already explained (macroscopic, microscopic, and mesoscopic). With the help of these models, different computational tools have been designed to help to simulate traffic environments. For macroscopic models, there are for example tools such as: Strada [82], Metacor [83], Visum [84], and Emme [85], among others. On the other hand, SUMO [86], PTV Vissim [87], and Aimsun [88] are some of the tools available that allow traffic simulation with microscopic models. For mixed or mesoscopic models, some of the existing tools are: OmniTRANS [89], TransModeler [90], and the Aimsun [88].

For public transport systems, in Bogotá, Colombia, the Bus Rapid Transit (BRT) system Transmilenio uses two of the most recognized tools for strategic route planning. These tools are PTV Vissim for microsimulation (traffic simulation at intersections, stop crowds simulation, etc.) and Emme for route planning (macrosimulation). With PTV Vissim, it is possible to carry out the specific traffic congestion simulation at an intersection; furthermore, traffic light crossing or lane sections, without these unit simulations interacting with each other to predict a global behavior. On the contrary, Emme allows planning routes based on predictions of passenger demand.

Optimizing the routes’ frequencies has been widely studied because this type of optimization can reduce the operating costs and travel times of passengers [5,91,92]. Unfortunately, frequency programming optimization for buses and public transport trains is known to be in the class of NP-hard [93] problems. These kinds of problems cannot be solved in polynomial time [94], which indicates that an algorithm may take an indeterminate time to find the solution to this problem.

2.4. Traffic Simulation Parameters’ Specification

For this research work, we define the public transport system parameters’ specification with the following conditions and variables. First, we have the system conditions that are defined in Table 1. These conditions define the total number of stops, a full OD matrix of size $N \times N$, the max passenger capacity of each stop, and the stop position.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of stops</td>
<td>$N$</td>
</tr>
<tr>
<td>Origin-destination matrix</td>
<td>OD</td>
</tr>
<tr>
<td>OD value for stop $i$ to $j$</td>
<td>$D_{ij}$</td>
</tr>
<tr>
<td>Stops max capacity</td>
<td>$SC_n$</td>
</tr>
<tr>
<td>Stops X position</td>
<td>$SPX_n$</td>
</tr>
<tr>
<td>Stops Y position</td>
<td>$SPY_n$</td>
</tr>
</tbody>
</table>

On the other hand, Table 2 defines the variables related to the routes and buses. Here, we find the total number of routes, the maximum number of buses available per route, the frequency of each route, the maximum passenger bus capacity, and the route stops’ table array. The stops’ table array represents the stations (or transit stops) where the bus route stops.
Table 2. Routes variables’ definitions and symbols.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of routes</td>
<td>$R$</td>
</tr>
<tr>
<td>Max buses per routes</td>
<td>$BR_r$</td>
</tr>
<tr>
<td>Routes frequency</td>
<td>$f_r$</td>
</tr>
<tr>
<td>Max bus capacity</td>
<td>$BC$</td>
</tr>
<tr>
<td>Route stop table array</td>
<td>$stops_table_r$</td>
</tr>
</tbody>
</table>

Next, Table 3 presents the simulation output variables, which indicate the performance of the public transport system with the actual routes’ configuration. These output variables consist of the total departed passengers per stop, the alighted passengers at the destination stop, the total average passengers’ commute time, and the average passengers’ commute time for the alight stop.

Table 3. Simulation output variables’ definitions and symbols.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Departed passengers from stop</td>
<td>$DPS_n$</td>
</tr>
<tr>
<td>Alighted passengers at stop</td>
<td>$APS_n$</td>
</tr>
<tr>
<td>Average passengers commute time</td>
<td>$ACT$</td>
</tr>
<tr>
<td>$ACT$ at alight stop $n$</td>
<td>$ACTS_n$</td>
</tr>
</tbody>
</table>

2.5. Simulation Performance Indicators

Table 4 shows the simulation performance variables. These variables indicate the simulation execution time and how fast the parallel simulation is relative to the real system and the sequential execution.

Table 4. Simulation performance variables’ definitions and symbols.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total simulation time</td>
<td>$tst$</td>
</tr>
<tr>
<td>Total parallel simulation execution time</td>
<td>$pst$</td>
</tr>
<tr>
<td>Total sequential simulation execution time</td>
<td>$sst$</td>
</tr>
<tr>
<td>Real-time factor</td>
<td>$rtf$</td>
</tr>
<tr>
<td>Speed-up factor</td>
<td>$suf$</td>
</tr>
<tr>
<td>Efficiency</td>
<td>$ef$</td>
</tr>
<tr>
<td>Number of processing threads</td>
<td>$p$</td>
</tr>
</tbody>
</table>

Simulation performance can be measured using two measures of effectiveness, namely the real-time factor and the speed-up factor. The real-time factor is a measure of the relative speed of simulation concerning real time [95]. In this work, we measure the real time factor of the parallel implementation by using Equation (6). For example, if we are simulating 1 h (3600 s) of the public transport systems’ routes in 3 s of execution time by the parallel model, the real-time factor will be 1200.

$$rtf = \frac{tst}{pst} \quad (6)$$

On the other hand, the speed-up factor ($suf$) is a measure of parallel simulation performance, which reveals how much faster the simulation is executed in parallel versus the sequential one [95]. Equation (7) describes how to calculate it. For instance, if the simulation execution time in parallel is 3 s and in the sequential one 30 s, the speed-up factor is 10.

$$suf = \frac{sst}{pst} \quad (7)$$
The efficiency \( (ef) \) describes where the improvements would be useful [96]. Equation (8) describes the efficiency as a function of the number of processing threads \( (p) \). It also represents the relationship between the minimum theoretical execution time for a parallel implementation \( (sst/p) \) and the measured processing time for the parallel run \( pst(p) \).

\[
ef(p) = \frac{sst/p}{pst(p)}
\]  

\( (8) \)

3. Masivo Public Transport Routes’ Simulation

In this section, we describe the components and the operational description of the developed public transport routes’ simulation called Masivo. Masivo is a public transport simulation software built in this research project to evaluate the new parallel simulation model proposed. This software is divided into three main components, as described in Figure 3.

![Figure 3. Masivo main components.](image)

First, we have the Parameters Loading Module (PLM), which is a Python sequential module that loads two CSV files with the information related to the OD matrix, stops, and routes. Then, we have the Parallel Simulation Core (PSC), which runs in OpenCL. The PSC runs the public transport route simulation model, by running each stop’s operation in an independent work-item. Finally, we have the Results Statistics Module (RSM), which is a Python module that extracts the statistical information related to passengers served, commute times, and simulation performance. The following subsections describe in-depth these three components. The full Masivo source code is available on GitHub (masivo codebase: https://github.com/jpruiz84/masivo), including installation instructions.

3.1. Parameters Loading Module

The Parameters Loading Modules (PLM) is a Python module that executes sequential operations to load and prepare the passenger arrival queue for the PSC. The PLM opens two CSV files, one with the information needed for the stop operations and others with the information needed for the bus operations, as described in Figure 4. The content of the two input files is described below.
3.1.1. ODM CSV File

The ODM (OD Matrix) CSV file contains the information related to the operation of the stops. This information includes the names of the stops, stops’ position (X and Y), stops’ max capacity, and the OD matrix. Table 5 shows an example of an ODM file for a three stop system. The first column (stop_number) has the following list of numbers starting at zero to identify each stop. The second column (stop_name) presents a text string with the stop name due in real transport systems. Here, a name, instead of a number, is used to identify a stop or station. Then, the third and fourth columns indicate the X and Y position in meters. The fifth column shows the maximum passenger capacity at each stop. Finally, the last three columns indicate the OD matrix, where the first row of these columns represents the names of the destination stops.

Table 5. Origin-Destination Matrix (ODM) CSV example file with 3 stops’ information.

<table>
<thead>
<tr>
<th>Stop_Number</th>
<th>Stop_Name</th>
<th>x_Pos</th>
<th>y_Pos</th>
<th>Max_Capacity</th>
<th>Stop_00</th>
<th>Stop_01</th>
<th>Stop_02</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Stop_00</td>
<td>400</td>
<td>1000</td>
<td>200</td>
<td>0</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>Stop_01</td>
<td>800</td>
<td>1000</td>
<td>200</td>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>Stop_02</td>
<td>1200</td>
<td>1000</td>
<td>200</td>
<td>20</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

3.1.2. Routes’ CSV File

The routes’ CSV file contains the information related to the route’s operation. This information includes the routes’ set, names, routes’ frequencies, routes’ time offset, routes’ direction, routes’ notes, and stops’ table. Table 6 shows an example of a routes’ file with four routes’ information. The first column (number) has the following list of numbers starting at zero to identify each route. The second column (name) presents the text string with the route name, as in a real transport system, names composed of letters and numbers are used to identify a route. The third column (freq.) indicates the route frequency in seconds, which represents how often a bus is dispatched for each route. The fourth column (offset) represents the route time offset in seconds, and it is the time when the first bus of each route is sent. The fifth column (buses) indicates the maximum number of buses available for each route. The sixth column (dir.) indicates the direction of this route, where W-E is west to east and E-W is east to west. The seventh column (notes) is used only for information proposes (not used by the PSC) and indicates, in this case, the type of route. Finally, the eighth column (stops_table) contains the stops’ table, which indicates the stop stations where the bus has to stop to pick up or alight passengers.

Table 6. Routes CSV example file with 4 routes’ information. Freq., frequency; Dir., direction.

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Freq.</th>
<th>Offset</th>
<th>Buses</th>
<th>Dir.</th>
<th>Notes</th>
<th>Stops_Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Route_00</td>
<td>300</td>
<td>100</td>
<td>50</td>
<td>W-E</td>
<td>all stops</td>
<td>Stop_00, Stop_01, Stop_02</td>
</tr>
<tr>
<td>1</td>
<td>Route_01</td>
<td>300</td>
<td>100</td>
<td>50</td>
<td>E-W</td>
<td>all stops</td>
<td>Stop_00, Stop_01, Stop_02</td>
</tr>
<tr>
<td>2</td>
<td>Route_02</td>
<td>900</td>
<td>100</td>
<td>50</td>
<td>W-E</td>
<td>express</td>
<td>Stop_00, Stop_02</td>
</tr>
<tr>
<td>3</td>
<td>Route_03</td>
<td>900</td>
<td>100</td>
<td>50</td>
<td>E-W</td>
<td>express</td>
<td>Stop_00, Stop_02</td>
</tr>
</tbody>
</table>
### 3.1.3. Passenger Arrival Queue Generation

After loading the ODM and routes’ CSV files, the PLM generates the passenger arrival queue \((paq)\) for each stop from the OD matrix defined as \(D\). The \(paq_i\) is an array that contains the passenger that will arrive at the stop during the simulation. The passengers are sorted by the arrival time, starting with the first passenger that arrives at the stop. Algorithm 1 describes the full process to generate the \(paq\). First, in Line 2, we define the \(PASS\_TOTAL\_ARRIVAL\_TIME\) as 3600 s. This means that the passengers will arrive at the stops from Time 0 to 3600 s during the simulation. Then, in Line 3, we declare a for loop that will run in each of the stops \(i\) and then will run the rows of the OD matrix \(D\). The passengers’ queues are statics list, with a fixed size of \(STOP\_MAX\_PASS\) (by default, 10,000). Then, we initialize these lists, with a status value in each field that indicates the end of the list \((PASS\_STATUS\_END\_LIST)\) and an arrival time that is an unsigned short \((UINT16)\) variable to the maximum possible value. This initialization is done by the operation performed in the loop from Lines 4 to 8.

**Algorithm 1** Passenger arrival queue generation algorithm.

```plaintext
1: procedure GENERATE_PAQ
2:  PASS_TOTAL_ARRIVAL_TIME ← 3600;
3:  for \(i \leftarrow 0\) to total_stops do
4:    for \(j = 0\) to STOP_MAX_PASS do
5:      \(paq_i[j].status = PASS\_STATUS\_END\_LIST\);
6:      \(paq_i[j].status = PASS\_STATUS\_END\_LIST\);
7:      \(paq_i[j].arrival_time = UINT16\_MAX\);
8:    end for
9:  end for
10: for \(j = 0\) to total_stops do
11:    for \(k = 0\) to \(D_{ij}\) do
12:      \(paq_i[j].orig_stop = i;\)
13:      \(paq_i[j].dest_stop = j;\)
14:      \(paq_i[j].arrival_time = PASS\_TOTAL\_ARRIVAL\_TIME \times k/D_{ij};\)
15:    end for
16:  end for
17:  Sort \(paq_i\) by \(arrival\_time;\)
18: end procedure
```

Later, in Line 10, we define a for loop to cover all destination stops. This will run the columns of the OD matrix \(D\). In Line 11, we set another for loop that will run for each passenger that will travel from stop index \(i\) to stop index \(j\). In Lines 12 to 14, we set the passenger’s origin stop \(paq_i[j].orig\_stop\), the destination stop \(paq_i[j].dest\_stop\), and the arrival time \(paq_i[j].arrival\_time\). Finally, after the full list of passengers is generated for the stop \(i\), we sort the passengers of \(paq_i\) by the arrival time. With this, the PSC can process \(paq\) efficiently, as we will discuss later.

### 3.2. Parallel Simulation Core

The Parallel Simulation Core (PSC) runs in an OpenCL kernel the passenger operations and the buses’ position update. The simulation tick is 1 s (this means that all simulation components are updated every 1 s). Each passenger has a data structure that contains the data needed for his operations. Figure 5 shows the passengers’ operations and the passengers data information. Each passenger has an ID that identifies him/her in the system; furthermore, the origin and destination stop numbers that indicate at which stop the passenger has arrived and at which stop he/she will alight. The arrival time is the time in the simulation that indicates when the passenger has arrived, and the alight time
indicates when the passenger has alighted. The status points out the actual status of the passenger inside the simulation (to arrive, arrived, on the bus, alighted, or empty). The empty status means that this space is free for another passenger, such as a seat on the bus. This passenger structure uses 13 bytes of data per passenger.

The stops are defined in the simulator by an array of structures. As described in Figure 5, each element inside the stops’ array has a stop number. Furthermore, they include the stop position in the map, total passengers, the last empty index, which indicates which was the last space empty in the waiting queue, and the passengers’ queues. Second, we include the arrival queue for passengers that will arrive at the stop. Third is the waiting queue for passengers that have arrived at the stop and are waiting for the bus. Finally, there is the alight queue for passengers that have finished the trip and have alighted at the stop. Each stop structure uses 130 KB of data.

Next, we have an array of structures that describes the buses in the simulator. As shown in Figure 5, each element inside the buses’ array has a bus number. Besides, we include the current stop index, which indicates at which stop the bus is, and the last stop index, which indicates which was the last stop where the bus stopped. Similarly, we incorporate here the stops’ arrays that have the stops’ index where the bus will stop to pick up or alight passengers. Finally, we include the total number of passengers on the bus and the passengers’ array that has the data of the passengers that are on the bus.

![Figure 5. Masivo simulation passenger operations.](image)

Moreover, in Figure 5, we see the three passengers’ operations (1, arriving; 2, boarding; 3, alighting). During the arriving, the passenger passes from the arrival queue to the waiting queue at the origin stop. During the boarding, the passenger moves from the origin stop waiting queue to the bus passenger array. Finally, during the alighting, the passengers move to form the bus’s passenger array to the alight queue at the destination stop. These passengers’ move operations are optimized to run efficiently in OpenCL and are described in the following subsections.

### 3.2.1. Passenger Arriving

In the passenger arriving operation, the passengers arrive at the stop at the arrival time. Algorithm 2 describes this process. First, we have an infinite while loop that breaks when there are no more passengers that will arrive at the stop in the given simulated time. In Line 2, we check if the total number of passengers remaining in \(paq\) is zero. If yes, we break the while loop. In Line 6, we extract the working index \(w\_\text{index}\) of the passengers’ arrival queue \(paq\) in \(w\). This working index represents the last passenger’s index that has been removed from \(paq\). Then, in Line 9, if the arrival time of a passenger that we are processing \(paq.p[w].arrival\_time\) is greater than the actual simulation time, we do not process this passenger yet. Hence, we break the while loop. Due to the passengers in \(paq\) being sorted by \(arrival\_time\), we do not need to process the next passengers, because we already know that \(arrival\_time\) will be greater than \(sim\_time\).
Algorithm 2 Passengers’ arriving operation at stop \( n \), where passenger arrival queue \( paq \) and passenger waiting queue \( pwq \) are specific for stop \( n \).

1: \textbf{while} TRUE \textbf{do}
2: \hspace{1em} if \( paq.p[l].total = 0 \) then
3: \hspace{2em} break;
4: \hspace{1em} end if
5: \hspace{1em} \( w \leftarrow paq.w\_index \);
6: \hspace{1em} if \( paq.p[w].arrival\_time > sim\_time \) then
7: \hspace{2em} break;
8: \hspace{1em} end if
9: \hspace{1em} \( l \leftarrow pwq.last\_empty \);
10: \hspace{1em} \( pwq.p[l] \leftarrow paq.p[w] \);
11: \hspace{1em} \( pwq.p[l].status \leftarrow ARRIVED \);
12: \hspace{1em} \( pwq.last\_empty \leftarrow pwq.last\_empty + 1 \);
13: \hspace{1em} \( pwq.last\_total \leftarrow pwq.total + 1 \);
14: \hspace{1em} \( paq.p[l].status \leftarrow PASS\_STATUS\_EMPTY \);
15: \hspace{1em} \( paq.p[l].w\_index \leftarrow paq.p[l].w\_index + 1 \);
16: \hspace{1em} \( paq.p[l].total \leftarrow paq.p[l].total - 1 \);
17: \hspace{1em} \( paq.p[l].status \leftarrow PASS\_STATUS\_EMPTY \);
18: \hspace{1em} \( paq.p[l].w\_index \leftarrow paq.p[l].w\_index + 1 \);
19: \hspace{1em} \( paq.p[l].total \leftarrow paq.p[l].total - 1 \);
20: \textbf{end while}

Thus, we move the passenger from the passenger arrival queue \( (paq) \) to the passenger waiting queue \( (pwq) \) only if the conditions of Lines 2 and 7 are not fulfilled. If we have a valid passenger with \( arrival\_time \) equal to or less than the simulation time, we move it. For this, in Line 11, we extract the \( pwq \) last empty index, which indicates the index of the last empty space in \( pwq \). Then, we assign the actual processing passenger in the arrival queue \( paq.p[w] \) to the last empty space in the waiting queue \( pwq.p[l] \). Latter, in Lines 13–15, we update the passenger status, and we increment the last empty index and the total counter for \( pwq \). In Lines 17–19, we update the \( paq \) passenger’s status to PASS\_STATUS\_EMPTY, we update the \( w\_index \) of \( paq \), to process the next passenger in the next iteration, and we decrement the total counter of passengers in \( paq \).

3.2.2. Passenger Boarding

The passenger boarding operations run concurrently at each stop. Algorithm 3 describes the boarding operations for the stop \( n \). For this, in Line 1, we start with a main for loop, then run for all the buses. Then, in Line 2, we have an if that only lets us process the bus passenger array \( (bpa) \) that is at the current stop. In Line 3, if the total number passengers in the waiting queue is zero \( (pwq.total) \), we brake, and we do not process more buses, because we do not have passengers at this stop waiting for boarding. In Line 6, we check if the total number of passengers of bus \( bpa[j] \) has reached the maximum BUS\_MAX\_PASS (the bus is full). If yes, we continue to process the next bus.

At this point, in Line 9, we know that passengers are waiting at the stop, and the bus is not full. Therefore, we check for each passenger at the stop to check which of them needs to board the actual bus. For this, in Line 9, we set a temporal variable \( last\_empty\_seat\_in\_bus \) to zero. This variable indicates which is the last seat on the bus that is empty. In this way, if there is more than one passenger in the stop to board the bus, we will not start from the beginning of the seats to check which is empty for each passenger.

To check if a passenger is at the stop, in Line 10, we have a for loop, then it runs over passenger waiting queue until STOP\_MAX\_PASS, that is the last position of the queue (in this case, 10,000). The queue is a static list with a size of 10,000. We know that there will be no more than 10,000 passengers...
in this queue; the total number of passengers in this queue is always less than this size. To know if we have passed the last passenger that has arrived in the queue, we check that the passenger’s status is PASS_STATUS_END_LIST. If yes, we break, and we finish checking for passengers to board at this stop. In Line 15, we check if the status of the passenger \(pwq[k]\) is PASS_STATUS_ARRIVED, which indicates that the passenger has arrived at the stop and is waiting for the bus. Then, in Line 16, we check if the destination stop of this passenger is in the stops’ table of the bus that we are processing. If yes, in Line 17, we have a for loop that checks the next empty seat on the bus, with the if condition in Line 18. If the position is empty on the bus, we move the passenger from the waiting list to the bus empty seat in Line 19. In Lines 20 and 21, we update the passenger’s status to PASS_STATUS_IN_BUS, and we increment the total number of passengers on the bus. In Lines 22 to 23, we update the status of the empty space in the \(pwq\), and we decrement the total number of passengers at the stop. Finally, in Line 24, we update the temporal variable \(last\_empty\_seat\_in\_bus\) to \(l\), so that when we process the next passenger at this stop in this simulation time iteration, we do not start from the beginning of the bus seats, which we already know are not empty.

**Algorithm 3** Passengers’ boarding operation at stop \(n\), where \(pwq\) is the passenger waiting queue specific for the stop \(n\).

1: for \(j \leftarrow 0\) to \(total\_buses\) do
2: \hspace{1em} if \(stop\_num = bpa[j].curr\_stop\) then
3: \hspace{2em} if \(pwq.total = 0\) then
4: \hspace{3em} break;
5: \hspace{1em} end if
6: \hspace{2em} if \(bpa[j].total \geq BUS\_MAX\_PASS\) then
7: \hspace{3em} continue;
8: \hspace{1em} end if
9: \hspace{2em} last\_empty\_seat\_in\_bus \leftarrow 0;
10: \hspace{2em} for \(k \leftarrow 0\) to \(STOP\_MAX\_PASS\) do
11: \hspace{3em} if \(pwq[k].status = PASS\_STATUS\_END\_LIST\) then
12: \hspace{4em} break;
13: \hspace{3em} end if
14: \hspace{2em} end for
15: \hspace{2em} end if
16: \hspace{1em} end if
17: \hspace{2em} if \(pwq[k].status = PASS\_STATUS\_ARRIVED\) then
18: \hspace{3em} if \(pwq[k].dest\_stop \in bpa[j].stops\_table\) then
19: \hspace{4em} for \(l \leftarrow last\_empty\_seat\_in\_bus\) to \(BUS\_MAX\_PASS\) do
20: \hspace{5em} if \(bpa[j].bpl[l].status = PASS\_STATUS\_EMPTY\) then
21: \hspace{6em} \(bpa[j].bpl[l] \leftarrow pwq[k]\)
22: \hspace{6em} \(bpa[j].bpl[l].status \leftarrow PASS\_STATUS\_IN\_BUS\)
23: \hspace{6em} \(bpa[j].total \leftarrow bpa[j].total + 1\)
24: \hspace{6em} \(pwq[k].status \leftarrow PASS\_STATUS\_EMPTY\)
25: \hspace{6em} \(pwq.total \leftarrow pwq.total - 1\)
26: \hspace{6em} \(last\_empty\_seat\_in\_bus \leftarrow l\)
27: \hspace{5em} break;
28: \hspace{4em} end if
29: \hspace{3em} end for
30: \hspace{3em} end if
31: \hspace{2em} end for
32: \hspace{1em} end if
33: end for
3.2.3. Passenger Alighting

The passenger alighting operations also run concurrently at each stop. Algorithm 4 describes the boarding operations for the stop \( n \). For this, in Line 1, we start with a main for loop that runs for all the buses. Then, in Line 2, we have an if that only lets us process the bus passenger array \( \text{bpa} \) that is in the current stop. In Line 3, we have an if that only lets us process buses that have more than zero passengers. In this way, at this point, we know that the bus is at the stop, and we have at least one passenger there. Then, in Line 5, we have another for loop that runs for all the bus’s seats. Later, the if condition in Line 6 checks if there is a passenger in the actual processing seat. Next, the if condition in Line 7 check if the passenger’s destination stop is the same as the actual stop. If yes, we know that the passenger must alight from the bus.

To alight the passenger from the bus to the stop, first in Line 8, we get the last empty index of the passenger alight queue \( \text{plq} \) for the temporal variable \( l \). Next, in Line 9, we move the passenger from the bus \( \text{bpa}[j].\text{bpl}[k] \) to the alighting stop \( \text{plq}[l] \). Later, in Lines 10 to 11, we update the passenger status to \text{PASS\_STATUS\_ALIGHTED}, we set the alight time, and we increment the total number of passengers in the alight queue. Finally, in Lines 14 and 15, we set the status of the empty seat on the bus to \text{PASS\_STATUS\_EMPTY}, and we decrement the passengers’ total number for this bus.

3.2.4. Buses’ Position Update

The buses’ position update runs sequentially per bus in an independent OpenCL work item. This position update consists of moving the bus across the stops and holding the bus at the stops inside the stop table for a fixed time in seconds. Algorithm 5 shows the operations for the bus position update. First, in Line 1, we have a for loop to run for all the buses. Then, in Line 2, we check if we have to start a new bus according to the route frequency and time offset. In Line 4, we check if the bus is already at a stop. If yes, we decrement \text{in\_the\_stop\_counter}, which indicates how many seconds
are remaining for the bus to be held at the stop. In Line 6, we check that \textit{in\_the\_stop\_counter} is zero. If yes, we set \textit{in\_the\_stop} to FALSE, indicating that the bus has finished the holding time at the stop.

\begin{algorithm}
\caption{Bus update position operations.}
\begin{algorithmic}
\For{$i \leftarrow 0$ \textbf{to} total\_buses}
\State \textbf{check\_if\_start\_the\_bus}(bus[$i$]);
\If{$bus[i].in\_the\_stop = TRUE$}
\State $bus[i].in\_the\_stop\_counter \leftarrow bus[i].in\_the\_stop\_counter - 1$;
\If{$bus[i].in\_the\_stop\_counter = 0$}
\State $bus[i].in\_the\_stop = FALSE$;
\EndIf
\EndIf
\If{$bus[i].in\_the\_stop = FALSE$}
\State $bus[i].curr\_pos \leftarrow bus[i].curr\_pos + \text{TRAVEL\_SPEED}$;
\EndIf
\If{$bus[i].curr\_pos = \text{next\_stop\_window}$}
\State $bus[i].curr\_stop \leftarrow \text{next\_stop\_i}$;
\State $bus[i].in\_the\_stop \leftarrow \text{TRUE}$;
\State $bus[i].in\_the\_stop\_counter \leftarrow \text{BUS\_STOPPING\_TIME}$;
\EndIf
\EndFor
\end{algorithmic}
\end{algorithm}

Then, in Line 11, we check if the bus is not at the stop. If it is not held at the stop, we increment the current position (\textit{curr\_stop}) according to TRAVEL\_SPEED. Finally, we check in Line 15 if the bus is in the stop window of the next stop. If yes, we update the \textit{curr\_stop} and \textit{in\_the\_stop} variables accordingly; also, we set \textit{in\_the\_stop\_counter} to BUS\_STOPPING\_TIME, which by default is 20 s.

3.3. Results Statistics Module

The Results Statistics Module (RSM) is a Python module that extracts the statics information results from the executed simulation. It extracts the number of passengers served, the passengers’ commute time, and the simulation performance. Masivo saves this information in the result folder, including CSV files and statistics graphs.

3.3.1. Passengers Served Information

When the simulation is done, each of the stops has a passenger alight queue, which contains the full passenger structure information (passenger ID, origin and destination stops, arrival and alight time, and status), as described in Figure 5. Masivo saves this information and statics about this information in the following output files.

- \texttt{total\_passengers\_results.csv}: contains all the information (passenger ID, origin and destination stops, arrival and alight time, and status) of all passengers inside the simulation.
- \texttt{served\_passengers\_per\_stop.csv}: describes per stop the total number of passengers waiting for a bus, the expected total input passengers, the total alight passengers, the total expected alight passengers, and the alighted passengers’ percentage.
- \texttt{served\_passengers\_per\_stop.eps}: graphs the total alighted passengers contrasted with the remaining passengers expected per stop.
- \texttt{commute\_time\_per\_stop.eps}: graphs the average commute time of the alighted passengers per destination stop and per travel direction. This graph is useful to see how the commute time is affected at a specific destination stop depending on traffic congestion on the road.
3.3.2. Performance Information

To know the simulation performance, Masivo saves useful simulation timings and host computer status information. These data are stored in the following files:

- **performance_timeline.csv**: contains the real-time simulation factor over time, with a sampling frequency of 100 s of the simulation. Furthermore, this file contains information related to CPU usage and CPU operation frequency over time.
- **performance_timeline.eps**: graphs the real-time factor over time, the real-time factor low-pass filtered signal, the CPU usage, and the CPU operation frequency.
- **simulation_brief.csv**: contains all simulation configured parameters, total simulation outputs, and total performance indicators. In particular, as inputs, this file contains the buses’ average travel speed, bus max passengers, bus stopping time, end simulation time, ODM file, passenger total arrival time, routes’ file, and stop to bus windows’ distance. As outputs, we find here total alighted expected passengers, total alighted passengers, total alighted passenger percentage, total average commute time, total simulated buses, total simulated passengers, total routes, and total stops. Finally, for the total performance indicators, we include in this file the average CPU usage, average real-time factor, total simulation execution time, OpenCL device name and compute units used, and the limit of the max OpenCL compute units.

3.4. 3D System Visualization Output

For validation purposes, a 3D system visualization output was integrated into Masivo. This visualization system is based on a Python 3D engine called Panda3D. When enabled, a separate thread runs the 3D visualization, with a running interval of 30 fps. In each frame update, the buses’ position, buses’ occupied seats, stops’ alighted passengers, and stops’ waiting passengers values are updated from the PSC. Figure 6 shows this 3D output interface. Here, the stops are represented with a blue container, which has a horizontal red bar that indicates the percentage of passengers in the waiting queue relative to the maximum stop passenger capacity. The vertical purple cylinder represents the percentage of alighted passengers relative to the total expected alight passengers. The bus has a horizontal red bar, which represents the percentage of occupied seats relative to the maximum bus passenger capacity.

![Figure 6. Masivo 3D game engine simulation output.](image)

3.5. Validation Simulator

A validation simulator was built only using Python (called here Pure Python) with dynamic arrays (to emulate passenger’s queues). This Pure Python simulator helped to get results that ensured the correct operation of the routes inside an origin-destination demand and then compared these
results with the parallel simulator. In this simulator, contrary to the Masivo PSC, the boarding and alighting operations were performed per bus that was held at a stop and not at the stop.

The correct behavior for this simulator was validated with the 3D system visualization output. For this, a three stop small public transport simulation system was simulated. The ODM matrix used here is described by Equation (9), totaling 80 passengers. The passengers arrived at the stops from Time 0 to 3600 s.

The correct behavior for this simulator was validated with the 3D system visualization output.

For this, a three stop small public transport simulation system was simulated. The ODM matrix used here is described by Equation (9), totaling 80 passengers. The passengers arrived at the stops from Time 0 to 3600 s.

\[
\text{ODM matrix} = \begin{bmatrix} 0 & 10 & 20 \\ 10 & 0 & 10 \\ 20 & 10 & 0 \end{bmatrix}
\]  

(9)

Two routes, with a 15 min frequency, were used for this system. The routes’ files are described by Table 7.

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Freq.</th>
<th>Offset</th>
<th>Buses</th>
<th>Dir.</th>
<th>Notes</th>
<th>Stops_Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Route_00</td>
<td>900</td>
<td>950</td>
<td>50</td>
<td>W-E</td>
<td>all stops</td>
<td>Stop_00, Stop_01, Stop_02</td>
</tr>
<tr>
<td>1</td>
<td>Route_01</td>
<td>900</td>
<td>950</td>
<td>50</td>
<td>E-W</td>
<td>all stops</td>
<td>Stop_02, Stop_01, Stop_00</td>
</tr>
</tbody>
</table>

Figure 7 shows the 3D visualization output for three different simulation times. Figure 7a shows the stops at the time 800 s of the simulation. Here, some passengers have arrived at the stops, but the buses have not departed yet. Figure 7a shows the simulation at the time of 1900 s. Here, some passengers have arrived at their destination stops. Furthermore, we see two buses moving passengers. Finally, Figure 7c shows the end of the simulation, where all passengers have arrived at their stops.
Furthermore, Figure 8 shows the total alighted and not alighted passengers at each stop. Here, we note that all passengers (80 of 80 expected) alighted at their destination stops.

![Destination stop passengers (Pure Python)](image)

**Figure 8.** Alighted passengers results for a three stop validation system in Pure Python.

Finally, this validation simulation also calculated the average commute for the passengers per destination stop and travel direction, as shown in Figure 9. Here, we note that the commute time for corner stops (Stop_00 and Stop_02) was greater than the commute time of the center stop (Stop_01). This was because, for instance, Stop_02 received passengers from the closest stop Stop_00 and the furthest stop Stop_01. Meanwhile, Stop_01 received a passenger for the closer stops Stop_00 and Stop_02; hence, the passengers’ average commute time was lower.

![Commute time per dest. stop (Pure Python)](image)

**Figure 9.** Commute time per stop for a three stop validation system in Pure Python. The abbreviation “dest. stop” refers to destination stop, “Pass. direc.” refers to passenger direction, “W-E” refers to West to East direction, and “E-W” refers to East to West direction.
4. Results

In this section, we discuss the simulation model results. First, we present the results for the validation process, where we compare the Masivo PSC outputs with the Pure Python validation simulator. Then, we show the performance results.

4.1. Validation Results

For the validation of the Masivo PSC, four different scenarios were created. These scenarios, with the same conditions, were simulated in the Masivo PSC and the Pure Python validation simulator. Output simulation results related to alighted passengers per stop and the commute time were compared.

4.1.1. Scenario 1, 30 Stops and Two Routes at 54 km/h

The simulation conditions for Scenario 1 are summarized in Table 8. The two routes pick up and alight passengers from all stops, with a bus departing frequency of 1 min. The first route goes from west to east, and the second one goes from east to west. The separation of the stops is 400 m, totaling a line length of 12 km for the 300 stops.

Table 8. Scenario 1 simulation conditions.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus average transit speed (km/h)</td>
<td>54</td>
</tr>
<tr>
<td>Bus maximum passengers</td>
<td>250</td>
</tr>
<tr>
<td>Bus stopping time (s)</td>
<td>20</td>
</tr>
<tr>
<td>End simulation time (s)</td>
<td>7200</td>
</tr>
<tr>
<td>Passenger total arrival time (s)</td>
<td>3600</td>
</tr>
<tr>
<td>Total passengers</td>
<td>38,721</td>
</tr>
<tr>
<td>Total routes</td>
<td>2</td>
</tr>
<tr>
<td>Total stops</td>
<td>30</td>
</tr>
</tbody>
</table>

Figure 10 shows the alighted passenger numbers per stop and the total alighted passenger numbers and percentage. The percentage of alighted passengers for the Masivo PSC was 99.85% and 100.00% for the Pure Python validation simulator, giving an error of 0.15%.

Figure 10 shows the alighted passenger numbers per stop and the total alighted passenger numbers and percentage. The percentage of alighted passengers for the Masivo PSC was 99.85% and 100.00% for the Pure Python validation simulator, giving an error of 0.15%.

Figure 11 shows the average passengers’ commute time per destination and the total average commute time. Masivo PSC showed a total average commute time of 15.77 min. Meanwhile, the Pure
Python validation simulator showed a total average commute time of 16.44 min, giving us an error of 0.52% relative to the total simulation time.

![Commuter times per destination stop](image1.png)

**Figure 11.** Passengers’ commute time per destination stop, 30 stops in Scenario 1. (a) Results from Masivo PSC. (b) Results from the Pure Python validation simulator. The abbreviation “dest. stop” refers to destination stop, “Pass. direc.” refers to passenger direction, “W-E” refers to West to East direction, and “E-W” refers to East to West direction.

### 4.1.2. Scenario 2, 30 Stops and Two Routes at 70 km/h

The simulation conditions for Scenario 2 are summarized in Table 9, where the difference between this and Scenario 1 was the transit bus speed, increased from 54 km/h to 70 km/h.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus average transit speed (km/h)</td>
<td>70</td>
</tr>
<tr>
<td>Bus maximum passengers</td>
<td>250</td>
</tr>
<tr>
<td>Bus stopping time (s)</td>
<td>20</td>
</tr>
<tr>
<td>End simulation time (s)</td>
<td>7200</td>
</tr>
<tr>
<td>Passenger total arrival time (s)</td>
<td>3600</td>
</tr>
<tr>
<td>Total passengers</td>
<td>38,721</td>
</tr>
<tr>
<td>Total routes</td>
<td>2</td>
</tr>
<tr>
<td>Total stops</td>
<td>30</td>
</tr>
</tbody>
</table>

![Total alighted passengers](image2.png)

**Figure 12.** The alighted passenger numbers per stop and the total alighted passenger numbers and percentage. The percentage of alighted passengers for the Masivo PSC was 99.62% and 100.00% for the Pure Python validation simulator, giving an error of 0.38%.
Figure 12. Alighted passengers per stop, 30 stops in Scenario 2. (a) Results from Masivo PSC. (b) Results from the Pure Python validation simulator.

Figure 13 shows the average passengers’ commute time per destination and the total average commute time. Masivo PSC showed a total average commute time of 14.02 min. Meanwhile, the Pure Python validation simulator showed a total average commute time of 14.47 min, giving us an error of 0.37% relative to the total simulation time.

Figure 13. Passengers’ commute time per destination stop, 30 stops in Scenario 2. (a) Results from Masivo PSC. (b) Results from the Pure Python validation simulator. The abbreviation “dest. stop” refers to destination stop, “Pass. direc.” refers to passenger direction, “W-E” refers to West to East direction, and “E-W” refers to East to West direction.

4.1.3. Scenario 3, 30 Stops and Four Routes at 54 km/h

The simulation conditions for Scenario 3 are summarized in Table 10, where the difference between this and Scenario 1 is the number of routes. Here, we have two new routes that do not stop at all stops. These new routes stop at the corners and in the downtown area where the demand is higher than the others. Both new routes have a frequency of 2 min. Precisely, Route 3 stops at Stop_00, Stop_01, Stop_02, Stop_03, Stop_18, Stop_19, Stop_20, Stop_21, and Stop_22. Meanwhile, Route 4 stops at Stop_29, Stop_28, Stop_27, Stop_26, Stop_22, Stop_21, Stop_20, Stop_19, Stop_18.
Table 10. Scenario 3 simulation conditions.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus average transit speed (km/h)</td>
<td>54</td>
</tr>
<tr>
<td>Bus maximum passengers</td>
<td>250</td>
</tr>
<tr>
<td>Bus stopping time (s)</td>
<td>20</td>
</tr>
<tr>
<td>End simulation time (s)</td>
<td>7200</td>
</tr>
<tr>
<td>Passenger total arrival time (s)</td>
<td>3600</td>
</tr>
<tr>
<td>Total passengers</td>
<td>38,721</td>
</tr>
<tr>
<td>Total routes</td>
<td>4</td>
</tr>
<tr>
<td>Total stops</td>
<td>30</td>
</tr>
</tbody>
</table>

Figure 14 shows the alighted passenger numbers per stop and the total alighted passenger numbers and percentage. The percentage of alighted passengers for the Masivo PSC was 99.37% and 100.00% for the Pure Python validation simulator, giving an error of 0.63%.

![Figure 14. Alighted passengers per stop, 30 stops in Scenario 3. (a) Results from Masivo PSC. (b) Results from the Pure Python validation simulator.](image)

Figure 15 shows the average passengers’ commute time per destination and the total average commute time. Masivo PSC showed a total average commute time of 15.71 min. Meanwhile, the Pure Python validation simulator showed a total average commute time of 16.33 min, giving us an error of 0.52% relative to total simulation time.

![Figure 15. Passengers’ commute time per destination stop, 30 stops in Scenario 3. (a) Results from Masivo PSC. (b) Results from the Pure Python validation simulator. The abbreviation “dest. stop” refers to destination stop, “Pass. direc.” refers to passenger direction, “W-E” refers to West to East direction, and “E-W” refers to East to West direction.](image)
4.1.4. Scenario 4, 300 Stops and Two Routes at 54 km/h

The simulation conditions for Scenario 4 are summarized in Table 11. It consists of 300 stops, which have a normal distribution with more demand for longer distances plus a random number of passengers. Furthermore, corner stops have two times more demand than the others. The two routes pick up and alight passengers from all stops, with a bus departing frequency of 30 s. The first route goes from west to east, and the second one goes from east to west. The separation of the stops is 400 m, totaling a line length of 120 km for the 300 stops. Then, the end simulation time is 10 h, so that all passengers have enough time to finish traveling.

Figure 16 shows the alighted passenger numbers per stop and the total alighted passenger numbers and percentage. The percentage of alighted passengers for the Masivo PSC was 99.52% and 100.00% for the Pure Python validation simulator, giving an error of 0.48%.

Table 11. Scenario 1 simulation conditions.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus average transit speed (km/h)</td>
<td>54</td>
</tr>
<tr>
<td>Bus maximum passengers</td>
<td>250</td>
</tr>
<tr>
<td>Bus stopping time (s)</td>
<td>20</td>
</tr>
<tr>
<td>End simulation time (s)</td>
<td>36,000</td>
</tr>
<tr>
<td>Passenger total arrival time (s)</td>
<td>3600</td>
</tr>
<tr>
<td>Total passengers</td>
<td>456,997</td>
</tr>
<tr>
<td>Total routes</td>
<td>2</td>
</tr>
<tr>
<td>Total stops</td>
<td>300</td>
</tr>
</tbody>
</table>

(a) Results from Masivo PSC.

(b) Results from the Pure Python validation simulator.

Figure 17 shows the average passengers’ commute time per destination and the total average commute time. Masivo PSC showed a total average commute time of 297.55 min. Meanwhile, the Pure Python validation simulator showed a total average commute time of 298.37 min, giving us an error of 0.13% relative to the total simulation time.
4.2. Performance

The performance simulation execution was performed in a dedicated server rented from 1&1 IONOS (https://www.ionos.com/servers/dedicated-servers). The specification of the rented server is described in Table 12. Before each performance test, the Linux CPU scaling governors were set in performance mode, to enable the maximum operating frequency.

Table 12. IONOS dedicated server 4XL specifications.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Dedicated Server 4XL</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel Xeon Gold 6210U</td>
</tr>
<tr>
<td># of Cores</td>
<td>20</td>
</tr>
<tr>
<td># of Threads</td>
<td>40</td>
</tr>
<tr>
<td>Processor Base Frequency</td>
<td>2.50 GHz</td>
</tr>
<tr>
<td>Max Turbo Frequency</td>
<td>3.90 GHz</td>
</tr>
<tr>
<td>Cache</td>
<td>27.5 MB</td>
</tr>
<tr>
<td>RAM</td>
<td>192 GB</td>
</tr>
<tr>
<td>HDD</td>
<td>2 × 4000 GB Hardware RAID 1</td>
</tr>
<tr>
<td>OS</td>
<td>Ubuntu 16.04</td>
</tr>
</tbody>
</table>

For the performance tests, the most complex simulation scenario (Scenario 4 previously tested) was used. This is a simulation scenario that has 300 stops and two routes. Table 13 describes the full list of conditions. It consists of 300 stops, which have a normal distribution with more demand for longer distances plus a random number of passengers. Furthermore, corner stops have two times more demand than the others. The two routes pick up and alight passengers from all stops, with a bus departing frequency of 30 s. The first route goes from west to east, and the second one goes from east to west. The separation of the stops is 400 m, totaling a line length of 120 km for the 300 stops. Then, the end simulation time is 10 h, so that all passengers have enough time to finish traveling.
Table 13. Simulation conditions for performance tests, Scenario 4.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus average transit speed (km/h)</td>
<td>54</td>
</tr>
<tr>
<td>Bus maximum passengers</td>
<td>250</td>
</tr>
<tr>
<td>Bus stopping time (s)</td>
<td>20</td>
</tr>
<tr>
<td>End simulation time (s)</td>
<td>36,000</td>
</tr>
<tr>
<td>Passenger total arrival time (s)</td>
<td>3600</td>
</tr>
<tr>
<td>Total passengers</td>
<td>456,997</td>
</tr>
<tr>
<td>Total routes</td>
<td>2</td>
</tr>
<tr>
<td>Total buses</td>
<td>2400</td>
</tr>
<tr>
<td>Total stops</td>
<td>300</td>
</tr>
</tbody>
</table>

4.2.1. Pure Python Validation Simulator Performance

The performance simulation of Scenario 4 was evaluated in the Pure Python validation simulator. Table 14 shows the performance output values. The total simulation execution time was near one hour (3303 s) for 10 h of simulation time. Then, the average real-time factor was 32.76 times faster than in real-time. Furthermore, we noted that due to this simulator being a sequential implementation, only one of the 40 processing threads was used, then only 2.19% of all the processing power was used.


<table>
<thead>
<tr>
<th>Performance Indicator</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total simulation execution time (s)</td>
<td>3303.612</td>
</tr>
<tr>
<td>Average real-time factor</td>
<td>32.76</td>
</tr>
<tr>
<td>Average CPU usage (%)</td>
<td>2.19</td>
</tr>
</tbody>
</table>

Figure 18 shows the real-time factor (RTF) vs. the simulation time. Here, we note that the RTF starts at 780, and then, it has an exponential decline, getting values under 10 times. This decline is because the Pure Python simulator uses dynamic lists to handle passengers and buses. When the system starts, the number of passengers and buses is low. However, then, more passengers arrive at the stops, and more buses are dispatched. Hence, the systems have more elements to process, and the simulation speed decreases.

Figure 18. Real-time factor vs. simulation time for the Pure Python validation simulator with CPU usage percentage and CPU operation frequency.
Furthermore, in Figure 18, we see that the CPU usage was always near 2.5%, due to the fact that the sequential implementation will never use more than one processing thread. Finally, the CPU frequency started at 3.2 GHz. However, because the CPU was not fully used, the OS decreased the CPU frequency during the simulation execution.

4.2.2. Masivo PSC Performance

The Masivo Parallel Simulation Core (PSC) performance was also evaluated using Test Scenario 4. Table 15 shows the performance output values. The total simulation execution time was 11.822 s, 278 times faster than the Pure Python simulator. The average real-time factor was 3050 times faster than real-time, and the average CPU usage was 84.21%.

Table 15. Masivo PSC performance outputs for Scenario 4.

<table>
<thead>
<tr>
<th>Performance Indicator</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total simulation execution time (s)</td>
<td>11.882</td>
</tr>
<tr>
<td>Average real-time factor</td>
<td>3050.84</td>
</tr>
<tr>
<td>Average CPU usage (%)</td>
<td>84.21</td>
</tr>
</tbody>
</table>

Figure 18 shows the real-time factor vs. the simulation time. Here, we note that the RTF starts in 4300, and then, it has an exponential decline, stabilizing near 2500. This decline is because when the system starts, the number of passengers and buses is low, and the internal for loop that runs the statics list does not need to go to the end of the lists. Later in the simulation, more passengers arrive at the stops, and more buses are dispatched. Hence, the systems have more elements to process, and the simulation speed decreases. This decreasing is not as low as in the Pure Python simulator, due to the concurrent processing operations performed per stop. Finally, from the simulation time of 15,000, we see that the RTF is increasing due to the passengers arriving at the final destination, then there are fewer elements to process.

Figure 19 shows that the CPU usage is near 80%, due to all the cores being used most of the time. Finally, the CPU frequency started and sustained at 3.2 GHz, because the OS maintained this due to the high processing demand.

Figure 19. Real-time factor vs. simulation time for Masivo PSC with CPU usage percentage and CPU operation frequency.
As described in Section 2.5, the simulation performance indicators were extracted. Figure 20 shows these indicators for the simulation executed with the compute units limited from one to 40. These results helped us to identify the performance for the different numbers of processing units and the efficiency of the proposed model.

First, Figure 20a shows the simulation execution time vs. the compute units. We see here that the execution time required to run the 10 h simulation was reduced exponentially with more compute units. Nevertheless, we had a low limit, near 10 s, because there were operations in the model that were not parallelized, such as the position update of the buses. Second, Figure 20b shows the real-time factor, which also increased, with the number of cores reaching a final value near to 3000.

Next, Figure 20c has the same shape as the RTF, because it is a function also of the simulation execution time, but in this case, divided by the simulation execution time for one compute unit. We see here that the maximum speed up factor reached was 10.2 times faster than the sequential simulation. Finally, we have the efficiency in Figure 20d, which shows an efficiency of more than 50% for 10 or less compute units.

Figure 20. Simulation performance indicators vs. number of compute units. (a) Total parallel simulation execution time ($pst$). (b) Real-time factor ($rtf$). (c) Speed-up factor ($suf$). (d) Efficiency ($ef$).
5. Discussion

In this section, we discuss the performance of Masivo PCS versus other simulation tools. Table 16 summarizes the documents related to parallel simulation applications for traffic and transit systems and also for Masivo PSC. FPGA applications include urban traffic simulation [97], with one document in which the authors in 2005 developed a microscopic simulation of road traffic. They simulated the Portland road network with 124,000 road segments and 100,000 intersections, achieving a speed-up factor of 4.5 [98]. For GPU’s parallel architecture, we found five implementations, where four were related to road traffic and one to public transit simulation. In 2012, Wang et al. built a GPU based traffic parallel simulation module, which was able to simulate three hours (10,800 s) of road traffic of a 40 × 40 lattice road network, with 5000 vehicle agents in 109 s, achieving a speed-up of 105 and a real-time factor of 102 [99]. Later, in 2014, Xu et al. implemented a mesoscopic road traffic simulation on CPU/GPU. This implementation was tested in a road network of 10,201 nodes, 20,100 unidirectional links, and 100,000 vehicles during 1000 simulation ticks (each tick was 1 s). The simulation time for this network was 4720 ms for CPU and 423 ms for GPU, getting a speed-up of 11.2 and a real-time factor of 2364 [60].

In 2017, Song et al. implemented the mesoscopic traffic simulation on GPU developed in [60] for a real-world scenario. The scenario was the Singapore expressway system, which is made up of 3179 nodes, 3388 links, and 9419 lanes with a demand modeled as trips from 4106 OD pairs. In total, 100,000 vehicles loaded into the peak traffic scenario during 1000 simulation ticks (each tick is 1 s) were simulated, and the execution time of this simulation was 1250 ms for CPU against 527 ms for GPU, getting a speed-up of 2.37 and a real-time factor of 1897 [59].

Next, in 2019, Saprykin et al. developed GEMSim (GPU-Enhanced Mobility Simulator), a GPU accelerated multi-modal mobility simulator for large scale scenarios. This was a mesoscopic multi-modal, queue based mobility simulator for public transit systems. Here, the large scale scenario of Switzerland with 513,770 nodes, 1,127,775 road links, 27,873 stops, and 21,847 routes was simulated. The simulation ran for a full day (86400 s), and the execution time was 290 s, equivalent to a real-time ratio of 298 for a population of 5.2 million and a real-time ratio of 1300 for 100,000 population size [13].

In the same year, Vinh and Tan created a framework for mesoscopic traffic simulation in GPU tested in an updated Singapore expressways network, which has 3186 nodes, 3386 links, and 9437 lanes, with demand modeled as trips from 4103 origin-destination pairs. Up to 300,000 vehicles were simulated, getting a speed-up of five times against the CPU [56].

For multi-core parallel architecture, first in 2012, Potuzak developed a distributed parallel road traffic simulator for clusters of multi-core computers, tested in regular square grids of 64,256 and 1024 crossroads. The simulations ran for 1 h (3600 s) with no number of vehicles specified with an execution time for the best cases of 22 s for 64 crossroads (real-time factor: 163), 86 s for 256 crossroads (real-time factor: 41), and 245 s for 1024 crossroads (real-time factor: 14) [100]. Then, in 2013, Bruegmann et al. developed an online, real-time traffic information system called OLSIMv4 (OnLine Traffic SIMulation version 4), which used microscopic traffic simulations exploiting the thread level parallelism on multi-core machines using a coarse-grained parallel microscopic simulation model. This simulation ran in the highway network of North Rhine-Westphalia (NRW), Germany. The simulated network contained more than 3000 loop detectors bundled into 1300 measurement cross-sections with an average statistical distance of 3.5 km. The simulation ran for 60 min with 120,000 vehicles with an execution time 60 s, resulting in a real-time factor of 60 [101].

In the same year, Fernandes et al. developed a parallel microscopic simulator capable of simulating metropolitan scale traffic using OpenMP. For the simulation, they used a real-world road network of the San Francisco Bay Area, which comprises 145,665 roads with a total of 27,439 km. The simulation ran in 60 s intervals with a step time of 0.1 s and generating a vehicle in each road every 2 s following a negative exponential distribution, totaling 4,349,130 vehicles. With these parameters, they achieved an execution time of 70 s (real-time factor 0.86) and a speed-up factor of 16 times, using 24 processors against the single processor simulation [102].
Table 16. Documents related to traffic and transit parallel simulation.

<table>
<thead>
<tr>
<th>Document Title</th>
<th>Year</th>
<th>Parallel Architecture</th>
<th>Model Type</th>
<th>Simulation Core</th>
<th>Speedup Factor</th>
<th>Real-Time Factor</th>
<th>Simulation Complexity</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Urban traffic simulation modeling for reconfigurable hardware</td>
<td>2005</td>
<td>FPGA</td>
<td>Microscopic</td>
<td>Road traffic</td>
<td>4.50</td>
<td>Not specified</td>
<td>100,000 intersections</td>
<td>[98]</td>
</tr>
<tr>
<td>A GPU based traffic parallel simulation module of artificial transportation systems</td>
<td>2012</td>
<td>GPU</td>
<td>Microscopic</td>
<td>Road traffic</td>
<td>105.00</td>
<td>102.00</td>
<td>1600 intersections and 5000 vehicles</td>
<td>[99]</td>
</tr>
<tr>
<td>Mesoscopic traffic simulation on CPU/GPU</td>
<td>2014</td>
<td>GPU</td>
<td>Mesoscopic</td>
<td>Road traffic</td>
<td>11.20</td>
<td>2364.00</td>
<td>10,201 nodes and 100,000 vehicles</td>
<td>[60]</td>
</tr>
<tr>
<td>Supporting real-world network oriented mesoscopic traffic simulation on GPU</td>
<td>2017</td>
<td>GPU</td>
<td>Mesoscopic</td>
<td>Road traffic</td>
<td>2.37</td>
<td>1897.00</td>
<td>4106 OD pairs, 100,000 vehicles</td>
<td>[59]</td>
</tr>
<tr>
<td>GEMSim: A GPU accelerated multi-modal mobility simulator for large scale scenarios</td>
<td>2019</td>
<td>GPU</td>
<td>Mesoscopic</td>
<td>Public transit system</td>
<td>Not specified</td>
<td>1300.00</td>
<td>27,873 stops and 100,000 passengers</td>
<td>[13]</td>
</tr>
<tr>
<td>A framework for mesoscopic traffic simulation in GPU</td>
<td>2019</td>
<td>GPU</td>
<td>Mesoscopic</td>
<td>Road traffic</td>
<td>5.00</td>
<td>Not specified</td>
<td>4103 OD pairs, 300,000 vehicles</td>
<td>[56]</td>
</tr>
<tr>
<td>Distributed-parallel road traffic simulator for clusters of multi-core computers</td>
<td>2012</td>
<td>Multi-core</td>
<td>Microscopic</td>
<td>Road traffic</td>
<td>3.32</td>
<td>163.00</td>
<td>1024 Crossroads</td>
<td>[100]</td>
</tr>
<tr>
<td>Real-time traffic information system using microscopic traffic simulation</td>
<td>2013</td>
<td>Multi-core</td>
<td>Microscopic</td>
<td>Road traffic</td>
<td>Not specified</td>
<td>60.00</td>
<td>1300 cross-sections and 120,000 vehicles</td>
<td>[101]</td>
</tr>
<tr>
<td>Parallel microscopic simulation of metropolitan scale traffic</td>
<td>2013</td>
<td>Multi-core</td>
<td>Microscopic</td>
<td>Road traffic</td>
<td>16.00</td>
<td>0.86</td>
<td>145,665 roads, 4,349,130 vehicles, and 0.1 s step time</td>
<td>[102]</td>
</tr>
<tr>
<td>Masivo PSC</td>
<td>2019</td>
<td>Multi-core</td>
<td>Mesoscopic</td>
<td>Public transit system</td>
<td>10.20/278</td>
<td>3000</td>
<td>300 stops, 2400 buses, and 456,997 passengers,</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Masivo PSC uses a mesoscopic simulation model type, due to the buses’ congestion in the road not being modeled. It is focused on public transit system simulation. It achieved a speed-up factor of 10.2 compared with the simulator model running with one compute unit and a speed-up factor of 278 times faster than the Pure Python validation simulator. Compared with other models, Masivo PSC focused on the buses’ route behavior inside a PTS, letting us define the routes’ stops’ table and frequency; and giving us; as a result, the passengers’ commute time. On the other hand, other implementations for commute or waiting passengers’ time simulation and modeling mentioned in the Introduction used a stochastic collective model for waiting time prediction [44] or a model to predict the waiting time based on the passengers’ learning process and adaptation with respect to waiting time uncertainty and travel information [42].

6. Conclusions

A new simulation model for routes in public transport systems using parallel computing called Masivo was presented. Masivo works with a predefined public transport system conditions, which include the stops’ total number, stops’ capacity, and the OD matrix. This OD matrix can be updated in this model via a CSV file. Masivo was built with three main components. The first one is the Parameters Loading Module (PLM), which is a Python module that loads the stops’ information (OD matrix, stops’ positions, stops’ max capacity) and the routes’ information. The second module is the Parallel Simulation Core (PSC), which was built in OpenCL and contains the new purposed simulation model. The PCS was designed to run in parallel the operations of each stop, including the passengers’ arrival, boarding, and alighting, and to run the buses’ displacement process sequentially over the roads. Finally, the Results Statistics Module (RSM) is a Python tool that ran after the PSC to get and analyze the simulation outputs and the performance results.

Furthermore, for Masivo validation, a Pure Python public transport simulator was built for validation purposes. This simulator performed the arrival operations sequentially for each stop. Then, it ran the boarding and alighting operations of each bus that was currently stopped at a station and, finally, executed the buses’ update position operations. Furthermore, it had a 3D output graphic interface to visualize the movement of the buses through the different stops, the stops, the bus occupancy, and the alighted passengers at each stop. This visualization tool helped to validate the consistent operations of this simulator and the Masivo PSC. With the Pure Python public transport simulator validated, we used it to validate the Masivo PSC in four different scenarios. We ran each scenario with precisely the same condition in both simulators to compare the simulation output results. The simulation output results for validation consisted of the total alighted passengers per stop and for the whole system, as well as the average commute time per stop and for the whole system. We found that the relative error was no higher than 0.7 % in all the tested scenarios.

The performance of Masivo was evaluated with the speed-up and real-time factor indicators. Masivo achieved a speed-up factor of 10.2 compared with the simulator model running with one compute unit and a speed-up factor of 278 times faster than the Pure Python validation simulator. The real-time factor achieved was 3050 times faster than the 10 h simulated duration. The performance test scenario used consisted of a public transport simulation with 300 stops, 2400 buses, and 456,997 passengers. This performance value was compared with the most recent traffic and public transport model that performed the simulation of systems with similar size. We found that, to date, there are no models reported specifically designed for PTS that can achieve the real-time factor that we have achieved in this research.

Future work with this model could integrate the passengers’ interchange between the different public transport modes, such as the train to bus. Furthermore, this model has a fast execution time, making this a high performance solution to use in a public transport optimization resources algorithm.

Funding: This work was funded by the Departamento Administrativo de Ciencia, Tecnología e Innovación (647-2014) and by the Universidad del Cauca (501100005682).

Conflicts of Interest: The authors declare no conflict of interest.

References


65. Fu, Z.; Yu, J.; Sarwat, M. Building a large-scale microscopic road network traffic simulator in apache spark. In Proceedings of the 2019 20th IEEE International Conference on Mobile Data Management (MDM), Hong Kong, China, 10–13 June 2019; Voume 2019, pp. 320–328. [CrossRef]


76. Bu, R. *Simulación: Un Enfoque Práctico*; Ingenieria Industrial; Publisher Editorial Limusa: Mexico City, Mexico, 1994.


84. PTV. VISLIM 11.50 User Manual; PTV: Karlsruhe, Germany, 2011.
97. Ruiz-Rosero, J.; Ramirez-Gonzalez, G.; Khanna, R. Field Programmable Gate Array Applications—A Scientometric Review. Computation 2019, 7, 63. [CrossRef]