MASTER'S THESIS

LARGE-SCALE DISTRIBUTED CROWD SIMULATION IN REAL-TIME

February 14, 2019

Karim Machlab
ICA-5906083

Supervised by: Dr. Roland J. Geraerts
Co-Supervised by: Dr. Jan Martijn van der Werf

Utrecht University
Game and Media Technology

Universiteit Utrecht

Abstract

The demand of large-scale simulations running in real-time has only been increasing. Whether we are simulating large crowds to test the effect of a newly constructed mall on the traffic around it, or to test how large crowds act in evacuation scenarios, or to create massive crowds in an animation or video game, the ability to do so and the information that we gain out of it is very beneficial. Current state-of-the-art simulations can only achieve a few thousand agents in real-time; these typically run on a single machine with shared memory. The logical step beyond this, if we want to scale-up to massive crowds and still simulate in real-time, is to distribute the simulation onto multiple machines that are in constant communication with each other.

In this work we present a distributed crowd simulation architecture and implementation. This architecture consists of multiple nodes, where each node handles a part of the simulation, and a simulation manager that manages all operations. With a decent partitioning structure, efficient communication between nodes, and a balanced workload, our system is able to simulate 12 thousand characters in real-time on a machine with 24 cores. Our results show that our system is able to scale well, and with the proper hardware, we can simulate hundreds of thousands of characters in real-time in big environments.

Contents

1 INTRODUCTION

Crowd simulation in virtual environments has become an essential tool in many different applications nowadays. One could simulate a crowd in a fictional 3D world with the intention of creating a realistic-looking map for a video game or animation, or one could simulate plausible scenarios that are based on real-life events. For example, if we wanted to study the effects a newly constructed mall would have on its surrounding area, we could use such a simulation software that shows us how a crowd would act in that environment; this would give us insights on congestion areas, the traffic status, and more.

Most crowd simulations in virtual environments manage to run in real-time on a single computer for a few thousand characters, but what if the purpose is simulating a hundred thousand characters at once, or even a million characters? An example is simulating a large crowd at the Hajj in Mecca, which can reach up to 3 million people [1]. Alternatively, our purpose could also be to simulate in super real-time to obtain hours of simulation results in just a few minutes. Simulating that many characters in real-time on a single computer is impossible (at least for the time being). In order to achieve such a thing, a distributed system of multiple computers is needed, where each machine simulates a specific part or region of the virtual environment. In this way, work is distributed across multiple machines, which makes it computationally possible to reach real-time performance for a large number of agents. All the relevant data updates from a simulation would be sent back to a single client computer, where the visualization of the simulation takes place. The term 'agent' could refer to any individual entity, such as a character, NPC, pedestrian, robot, or unit (which are all terms used in different research fields). Many questions arise when talking about distributed systems in the application scope of crowd simulation:

- How do we distribute the work evenly to different machines or nodes?
- How do we partition the virtual environment to achieve the best performance?
- What is the ideal number of nodes to have in our system?
- How many agents do we assign each node?
- Which machine has access to which data, and to which other machines can it communicate to?

Many more questions could be asked, and these all have to be answered to design the perfect distributed crowd simulation architecture. One possible way of setting up a distributed system is using a cloud. Cloud computing has been around for some time, and many platforms exist that use it in coordination with the 'Software as a Service' (SaaS) model, such as Google Cloud and Amazon Web Service. Distributing work onto a network of nodes leads to a vast speed-up in the application at hand, which in this case is crowd simulation. In this thesis we explore an efficient approach to distributing crowd simulation onto a cloud or network of computers, and the various architecture choices that are made along the way.

1.1 Project Goals

The main goal of this project is to achieve real-time, interactive crowd simulation on a large scale (as large as a million agents). Distributing the work of simulating agents onto multiple nodes is an effective way to achieve this. The biggest challenge is keeping the system running in real-time, meaning we have to study and experiment with different techniques, algorithms and parameters for optimal performance. We also want our system to be interactive, which means that we should be able to interact with the simulation while it's running; for instance adding agents or obstacles.

In this thesis we will explore a new distributed system for crowd simulation. This system will consist of multiple working nodes communicating with each other. We will study different approaches and methods to partition a 3D multi-layered environment, such that each node will handle and simulate the agents inside the region it was assigned to. We will analyze and compare several techniques and parameters of the system, such as the number of nodes, the number of agents per node and the optimal inter-communication model between nodes.

1.2 Research Questions

The main research question that we want to answer in this thesis is the following:

How can we distribute the simulation of a real-time, interactive crowd onto different nodes of a cloud to yield maximum speed-up?

This raises a few sub-questions concerning what 'maximum speed-up' actually means:

- What is the ideal way to partition a virtual environment?
- What is the optimal number of nodes and agents per node?
- How can we achieve minimum communication between nodes?
- How can we achieve a balanced workload across all nodes (minimum average idol time)?

1.3 Contribution

In this thesis we propose a new method for distributed crowd simulation in multi-layered virtual environments, one that ensures real-time performance and addresses scalability. A multi-layered environment signifies a set of two-dimensional layers and a set of connections that form between them. An example is a building with multiple floors, with stairs acting as the connections between them.

While methods for distributed crowd simulation in real time do exist, they are limited by poor scalability. There are a number of reasons for that; current methods have resorted to poor partitioning of the virtual environment and suffer from communication overhead. We will present an efficient method to partition the virtual environment such that each partition is handled by a separate node (server), as well as an efficient architecture and method that allows for minimum intercommunication. We will also present a method for balancing the work load between nodes.

1.4 Outline

The rest of this thesis is structured as follows. Section 2 gives an overview of the related work in distributed crowd simulation techniques. Section 3 defines the preliminaries and the current framework used for implementation. In Section 4 we present the proposed architecture and substeps of our distributed crowd simulation. In Section 5 we present different methods for partitioning a virtual environment. In Section 6 we present methods for communication between nodes. Section 7 discusses workload balancing between nodes. Section 8 discusses multi-layered environments and how they are incorporated in our system. Section 9 features groups and how they could be implemented in a distributed crowd simulation system. In Section 10 we present our implementation of the current distributed ECM framework. Section 11 illustrates our experimentation, results and analysis. In Section 12 we attempt to estimate and predict how our system scales with different parameters. In Section 13 we conclude that our methods are effective for real-time, scalable, distributed crowd-simulation applications. Lastly, in Section 14 we leave room for discussion about limitations and future work.

2 Related Work

2.1 Software Model

It is important to mention the model of our software and how it can be used by users. Azevedo et al. [2] discuss the topic of SimSaaS (Simulation Software as a Service) and HLA (High Level Architecture). Nowadays, many simulations are software used as a service. HLA is a standard for interoperability between simulators developed by IEEE. HLA and SOA (Service-Oriented Architecture) can be integrated together: HLA has good interoperability, synchronization and effective communication between federates (communicating components), and SOA benefits from scalability and reusing components. Together, HLA and SOA could extend integrated simulations [2].

For this project, we are interested in a Software as a Service (SaaS) model, more specifically, a Modeling and Simulation as a Service (MSaaS) model. MSaaS enables users to access modeling and simulation applications and capabilities (typically web applications) [3]. One important perspective concerning MSaaS is MSaaS as a cloud-service model; this deals with how an M&S application is deployed and accessed by a user. Cloud computing is a model that enables on-demand network access to a shared pool of computing resources (networks, servers, applications) [3]. Our distributed crowd-simulation software will follow a similar MSaaS model.

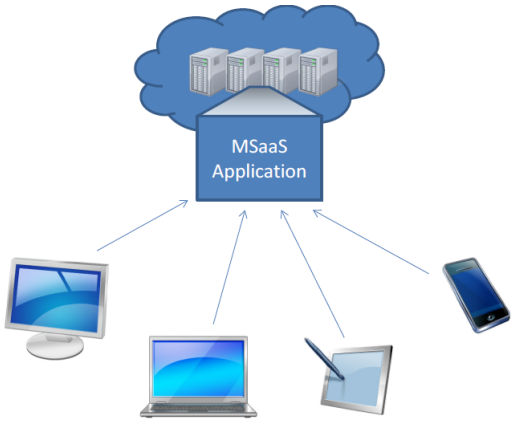


Figure 2.1: MSaas as a Cloud Service Model [3].

2.2 Architecture

When dealing with distributed systems, it is essential to think about what kind of architecture we need. Different applications require different architectures; one could think of a client-server architecture, where every client has to communicate directly with the same server. This, of course, is a bad architecture if we are interested in maximum speed-up and minimum latency. The server is the bottleneck in this situation because all clients have to constantly wait for a response from the server before performing their next task. Another type of architecture is peer-to-peer, where every client is also a server. Even though this might seem a good solution for fast and distributed computation, there is the awareness problem; each node has to be aware of the state and data in the other nodes in order to synchronize processes. Creating awareness in a peer-to-peer architecture is difficult to do, and quite costly. Then there is the networked-server architecture, where we have multiple servers that can communicate with each other, and each server is connected to a specific number of clients. This is the best solution when it comes to distributed crowd simulation, since it eliminates having a bottleneck of a single server, as well as the awareness problem. Nodes still need to synchronize with each other and be aware of each other, but this can be done with minimal communication [4]. We will implement a networked-server architecture in our system since the main goal is to reach real-time simulations with up to a million agents at once.

Besides the architecture type, it is important to establish the architecture itself. Many authors proposed an architecture consisting of a master component and several smaller slave components. Lozano et al. [4, 5] proposed a networked-server architecture where agents are uniformly distributed onto different servers that can inter-communicate with each other. There is a single and unique Action Server (AS) that acts as a world manager; it controls and modifies all the information a crowd can perceive. For scalability, the AS should be placed on the computer with highest computational power. Apart from the AS, there are multiple Client Processes (CP) that handle a certain number of agents. The AS consists of a centralized Semantic Data Base (SDB), which represents the global knowledge, and an Action Execution Module (AEM) that processes different actions. Instead of using a SDB that saves object and agent maps, the authors also propose using a grid. Each CP has a copy of the SDB; SDB updates are sent from the AS when available. After a CP receives the updated SDB, it can think and process information locally for the agents it handles, and then asynchronously send its actions to the AS. Many others have gone with a similar architecture, and it seems to be efficient, as long as communication is minimal and workload is balanced, but is there an architecture that can be even more efficient?

Lees et al. propose a High Level Architecture (HLA) that allows different simulations to be combined into one large simulation.[15] In other words, the simulation of multiple agents (and other environmental objects) is distributed over multiple 'federates', and the entire global simulation is called a 'federation'. A federate can have many purposes and simulate different things; federates can also be written in different languages and run on different machines, making *interoperation* possible. Each federate has a Federation Object Model (FOM) that saves objects attributes and event parameters, essentially the state of the simulation. Simulation is done in 3 phases: sensing, internal processing and action execution. Logically, agents might need data found in other federates, such as an agent attribute. In order to do that, a Runtime Infrastructure (RTI) is used as a bus for all communication between federates. Each federate also includes an RTI Ambassador and a Federate Ambassador. An RTI Ambassador handles all outgoing information from the simulation to the RTI, resulting in a corresponding callback to other federates. A Federate Ambassador handles these callbacks and invokes appropriate actions. Federates can *publish* certain attributes that it updates in a simulation and *subscribe* to attributes that it would like to receive updates for. In this way all needed information is passed on from one federate to another so that a simulation can proceed. Attributes are saved locally in each federation, even for an agent in another federation.

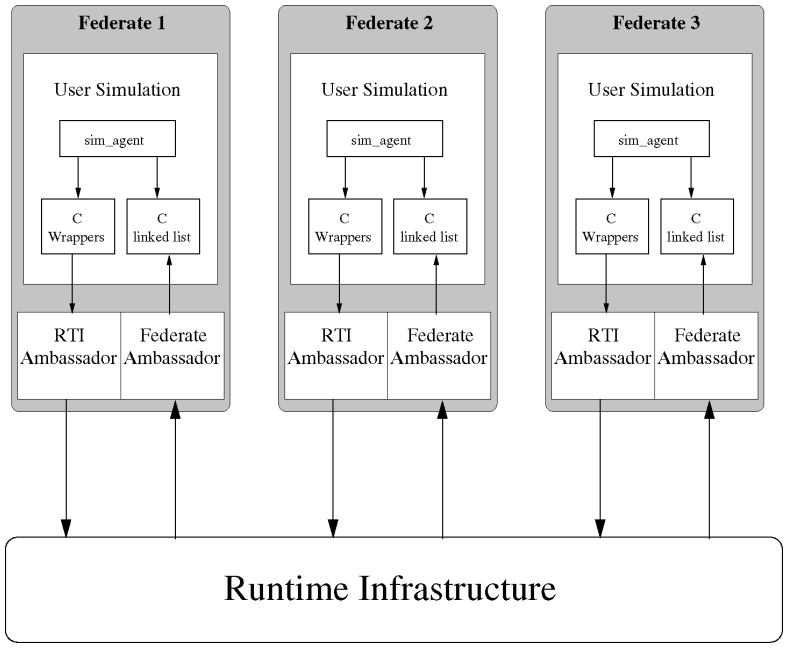


Figure 2.2: HLA AGENT architecture [15].

It is important to establish which component has access to what data, and what data should be communicated to which component. In order to simulate a large crowd in real time (up to a million agents) and be able to scale up, we need to decide on the optimal architecture. Spatial partitioning of the virtual environment, or better yet the navigation mesh of the environment, is also important to establish since it can be the key to an efficient distribution. This brings us to our next point.

2.3 Partitioning the Virtual Environment

In order to partition the environment into regions that nodes can work on, a certain model is needed. So far, previous work has referred to a grid-based partitioning [6, 7]. This is obviously not the best partitioning model, since it does not take into account the distribution of the agents in the environment. We might thus end up with several empty grid cells, or grid cells that have way too many agents. Using a grid, however, is a good first step and allows easy and fast communication between cells. Steffen et al. [6] decompose or spatially partition their environment into weakly interacting parts by using a grid. The domain partition is static, but it is also done in a way that takes the environment into consideration (rooms as separate partitions, if possible). One reason that a static decomposition is a good idea is that agents' movements are predictable, up to a certain extent. For example, congestion or gathering of pedestrians will happen at exits of the environment. Similarly, Wijerathne et al. [7] use a simple high-resolution grid partitioning, except they did not take into account environmental factors or the geometry of the environment. A smarter decomposition would be a k-d tree or any other type of balanced tree. Vermeulen et al. [8] show that a k-d tree offers the best performance in terms of construction and query time, specifically the nanoflann implementation. An even smarter decomposition would be one that takes into account the environment, such as a BVH (Bounding Volume Hierarchy). So far the current state-of-the-art technology is lacking in this regard; a smarter partitioning algorithm is needed: one that takes the environment and crowd densities into consideration.

We can also present another way of looking at the partitioning problem. Instead of thinking about it as a spatial partitioning problem, we can think about it as an agent-to-server assignment problem. What is the best and optimal way to assign a set of agents to a set of nodes or servers? Previous work has illustrated different approaches to dynamic assignment of agents, typically dealing with minimizing a cost function. Since these approaches 'partition'

agents (assign agents to servers, as opposed to spatially partition the environment) based on trying to balance the workload, they will be discussed in the next section (Section 2.4).

2.4 Balancing Workload

Another key element for fast and efficient execution is having a balanced workload. Many approaches exist to partition the DVE (distributed virtual environment) by solving the problem of assigning each agent to the optimal server. A typical cost function (which is used as the deciding factor) is the following:

$$C_P = W_1 C_P^W + W_2 C_P^L, \tag{1}$$

where C_P^W denotes the computing workload of all servers (which is minimized by sharing the workload proportionally onto all servers), and C_P^L denotes inter-communication cost (which is minimized by assigning agents that share the same AOI to the same server). W_1 and W_2 are weights such that $W_1 + W_2 = 1$ [9].

Other approaches include genetic algorithms (GA) which use biological concepts (like natural selection) to provide a learning method that maximizes efficiency and balances workload [10]. The GA method starts with a population of elements called 'chromosomes'. Each chromosome is a vector of N values from 1 to S, where S is the number of servers available and N is the total number of border agents. The number of chromosomes is the number of partial solutions for the GA. Therefore, each chromosome represents an assignment vector of each border agent to a server. After initialization (different initialization algorithms exist), mutation can take place, which involves changing the assignment of one of the elements at random [10]. The goal is to reach a mutation that has a low cost, based on a the cost function illustrated in equation 1. This is an iterative process which takes several iterations to learn an efficient partition. While this seems like an interesting choice, it might take a while for the system to learn and stabilize, making it an unsuitable option for real-time applications.

Another similar approach is an implementation of the Ant Colony System (ACS), which is a heuristic-search method based on evolutionary computation [11]. This method tries to assign border agents to the best possible candidate server based on their pheromone level. Each candidate server is tested by using a cost function (equation 1), and is assigned a pheromone level based on that. The server with the highest pheromone level is chosen. Again, this has the same problem as genetic algorithms, since it takes a while for the system to

learn. This method also has unreasonable computation overhead since each scenario (with a candidate server per border agent) is tried and tested.

Many authors discussed repartitioning the environment periodically to balance workload, since agents are constantly moving and changing positions. The decomposition is done in a way to have equal work load at each node, therefore a load balancer is needed. When there is significant load imbalance (due to significant agent movements), repartitioning takes place. Aguilar et al. [12] tried to create a repartitioning system to balance workload between different nodes. This paper extends the work of the previous one: HPC (high performance computing) enhancements of a large-scale evacuation simulator. A key feature that this model has is efficient load balancing. Each agent keeps track of its execution time, and instead of balancing the number of agents per node, the average execution time per node is balanced. This minimizes the idle time per node, reducing the cumulative waiting times down to 3%.

Instead of partitioning the environment spatially, one can also partition it by objects, namely agents. Vigueras et al. [13, 14] focused on load balancing a distributed crowd simulation system by partitioning the agents into different servers. The authors used a simple architecture with multiple action servers (AS) that can communicate with each other; each AS handles a client, and each client is responsible for a certain number of agents. The authors propose the QHull partitioning method, which partitions agents based on a distance function. The convex hull of all agents in a partition creates the special decomposition. The authors used a fitness function for their model that is a weighted sum of the number of agents intersecting two or more regions (border agents) and a balancing factor of the partition. The QHull partition is designed to have the least number of agents on the border of partitions, so that we have the least amount of communication between nodes. The QHull method repartitions periodically; it tries to minimize a function based on the distance from an agent to the centroid of a region and on the number of agents in that region. By testing all regions for each agent, we select the region that gives the minimum score in that function. Using this function, workload balancing is achieved.

While these methods seem effective in balancing workload, they can be quite computationally costly. We do not want to sacrifice performance to calculate the best workload balance. However, knowing the best balance will increase performance, so in a way it's a double-edged sword. Current methods do not state what the best period of repartitioning is

(how often we repartition), and the workload balancing problem is still an ongoing problem that needs to be addressed to achieve large-scale crowd simulations in real time.

2.5 Inter-communication

The next step in building an efficient distributed model is having minimum communication between slave nodes and between them and the master. Steffen et al. [6] designed model for real-time large-scale pedestrian evacuations that uses MPI (Message Passing Interface). The simulation on each node is run in a multi-threaded environment using OpenMP. Wijerathne et al. [7] also use MPI for intercommunication. They also try to minimize the communication overhead: Exchanging dynamic data needs at least 2 messages and packing/unpacking a large amount of data. The authors try to hide communication by sending large chunks of data together. Another optimization they did was minimizing data exchange in repartitioning; the repartitioning algorithm should detect whether a partition is assigned to the same CPU and exchange only the newly assigned agents. In our case MPI seems like a valid option since the biggest overhead will be communication between nodes; the more we minimize intercommunication, the more agents we can fit in an environment while still maintaining real-time performance.

2.6 Synchronization

Synchronization between nodes can be the biggest bottleneck in a distributed simulation. Xu et al. propose two heuristics to minimize the frequency of synchronization between nodes in a distributed parallel agent-based road traffic simulation [16]. The authors use a similar *subscription* system between nodes, or logical processes (LP) as they call them: An agent in one LP can subscribe to attributes of an agent in another LP. To minimize the number of times two neighboring LPs need to synchronize, they propose a Mutual Appointment (MA) protocol, such that two neighboring LPs only communicate on certain mutually agreed times (appointments). At each appointment, migrating agents are exchanged and proxy agents are updated (local copies of agents in neighboring LPs that are in the buffer region), after which a new appointment is set. The new appointment is the minimum between the lookaheads of both LPs. The lookahead is generally the minimum time that any agent in a LP needs to travel to the buffer region connected to the neighboring LP, where the buffer region is that region at the borders of two LPs. An agent in a buffer region affects agents in the neighboring LP, and so agents in the neighboring LP need to subscribe to its attributes.

Other than reducing synchronization frequency by using lookaheads and Mutual Appointments (MA), the authors used two heuristics. The first one is to increase the lookahead and therefore skip some synchronization operations (relaxing a synchronization). This means that some agents might not be migrated in time and some discrepancies might occur because LPs might need an updated version of their proxies from neighboring LPs. To deal with that, a Dead-reckoning function is used to estimate the current state of the proxy. Discrepancies might still occur between the proxy and its original agent. In order to choose when to increase the lookaheads, the stability of traffic is considered (based on the density and flow of car agents). Different classes of stability are introduced, such as 'stable', 'linearly unstable' and 'metastable'. The more stable the traffic is, the safer is it to increase lookahead without having many discrepancies. The second relaxation heuristic reduces the temporal resolution of boundary agents; this means that the update interval is increased for agents that are inside the buffer region between two LPs (reduced temporal resolution). Once these agents exit the buffer region, their update intervals are reduced again to normal. The relaxed update interval is calculated based on traffic densities, stability and states. Two examples can be seen in Figures 2.3 and 2.4 below.

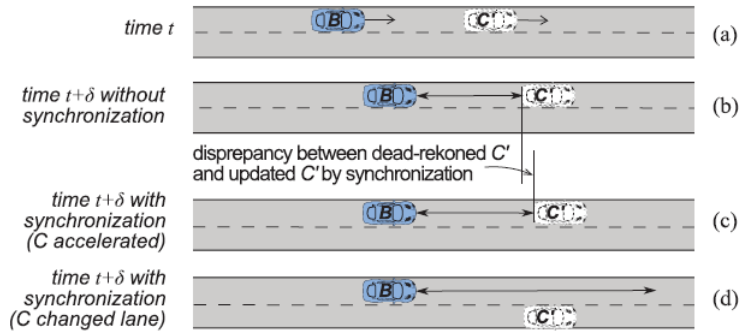


Figure 2.3: Synchronization skipping and dead-reckoning [16].

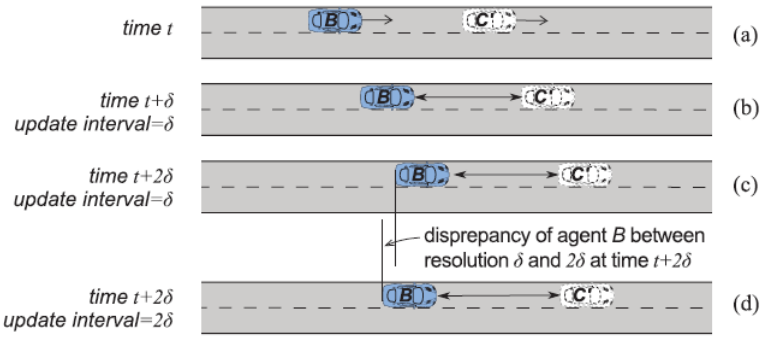


Figure 2.4: Reducing temporal resolution [16].

2.7 Number of Nodes

A question that frequently arises when dealing with distributed systems is how many nodes do we use? In other words, how much do we distribute the work? Cutting the work into smaller and smaller regions will surely reduce the amount of local work in each region, but will increase the communication and synchronization between nodes. At some point the communication overhead will take over and will be the bottleneck of the simulation. In order to find the 'sweet spot', experiments have to be run, testing a different number of nodes each time.

In the distributed simulation of agent-based systems with HLA by Lees et al. [15], the authors experimented with different numbers of federates to see the effect on the execution time of each federate simulation, as well as the communication time through the RTI. They tested this on two scenarios: reactive agent simulations with minimal CPU requirements, and deliberative agent simulations with heavy CPU loads that dominate the communication overhead. For experimentation, they used a tiled environment that is 50 units by 50 units in size. The results they obtained were as follows: Initially, as more nodes (federates) were used for the distributed simulation, the elapsed times decreased. However, after about 4 to 8 nodes, elapsed times started increasing again. This is because the overhead of the RTI communication becomes prominent. Even for reactive simulations (CPU light), distributing the work onto multiple nodes does help in performance, but only up to 4 nodes. Elapsed time includes both the simulation time and RTI (communication) time; as simulation time decreases, RTI time increases. To have a properly balanced system one must find the sweet spot; in this case it's around 4 to 8 federates. Of course, this is highly dependent on the type and size of the environment; we would expect to see a different optimal number of federates for a bigger environment.

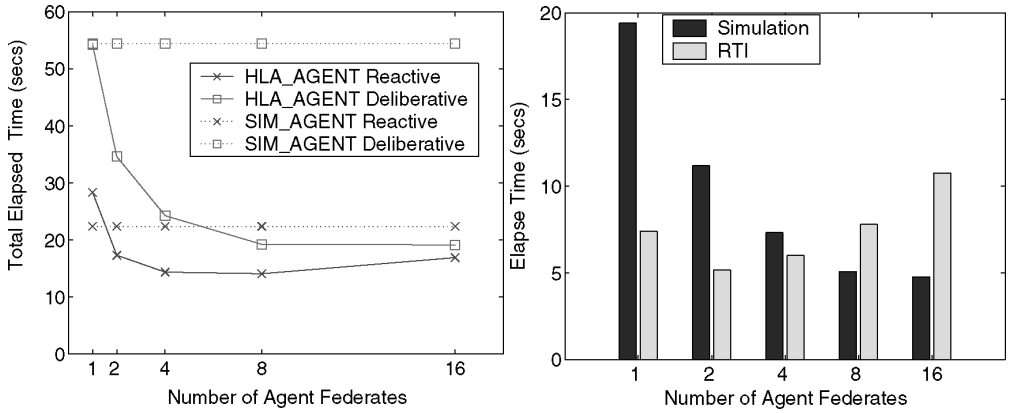


Figure 2.5: The effect of the number of federates on the simulation, RTI and total times [15].

In the case of Xu et al. [16], a different number of LPs was tested, with 4 different modes: **barrier** (normal barrier synchronizations), **MA** (mutual appointments), **MA-DR** (MA with synchronization skipping and dead-reckoning) and **MA-LR** (MA with lower temporal resolutions). The running time, speedup and synchronization messages were recorded for different numbers of LPs. The ideal number of LPs, or nodes, turned out to be 16, specifically for MA-DR, with lowest running times and highest speedup.

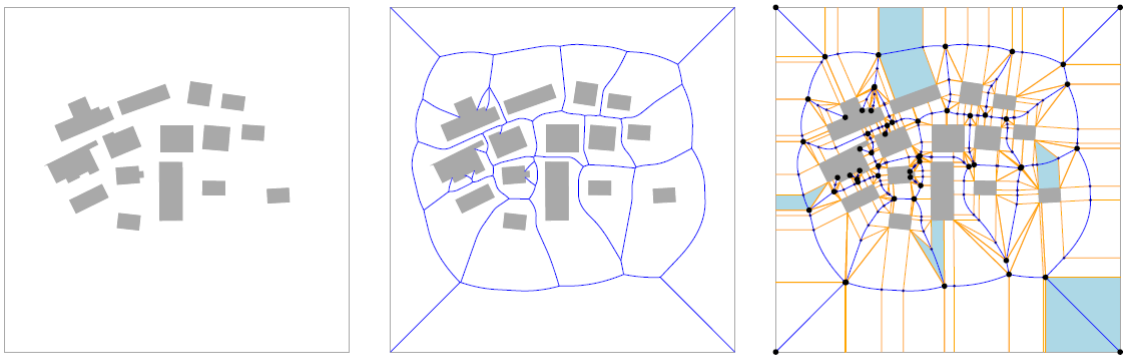
2.8 Conclusions

We can draw significant conclusions from the related work, which will establish some of the decisions we make for our distributed system. For our system, we are interested in a Modeling and Simulation as a Service (MSaaS) model and a Networked-server architecture. We will follow a similar master-slave architecture as Lozano et al. [4, 5], since this architecture proves to be effective and efficient. Having a manager also abstracts and limits the information needed for each slave. As for partitioning the virtual environment, a k-d tree performs well [8] and ensures a balanced workload. Ideally, we could even improve this by taking the environment geometry into consideration. A lot of the discussed methods for workload balancing, such as genetic algorithms and the Ant Colony System (ACS), seem to be effective, but computationally expensive. A k-d tree already balances the workload since it ensures an equal amount of agents on each node; this minimizes the idle time per node. Repartitioning is also effective in updating the k-d tree and re-balancing the workload. For inter-communication, Steffen et al. [6] and Wijerathne et al. [7] both use MPI in their implementations, which seems like the right choice for our system as well. We can also look at the previous work of Xu et al. [16] to reduce synchronization points in our system, this includes lookaheads, Mutual Appointments (MA), and dead-reckoning. Finally, the optimal number of nodes highly depends on the virtual environments and architecture of our system. Similar to the work of Lees et al. [15], we have to test and experiment with different numbers of nodes to conclude what the optimal number of nodes for our system is.

3 PRELIMINARIES

3.1 ECM framework

The framework that we will be working with is the Explicit Corridor Map (ECM) framework. The ECM is a navigation mesh that allows fast, efficient and flexible path planning and crowd simulation. The ECM can be described as a medial axis, which is the set of all points that have more than 1 distinct closest obstacle point, essentially making it a graph structure that runs exactly through the middle of the walkable space. The ECM goes even further and annotates closest obstacle information with the use of retractions. Note that in this framework, the term 'character' is used for the agent being simulated; we will use these two terms interchangeably.



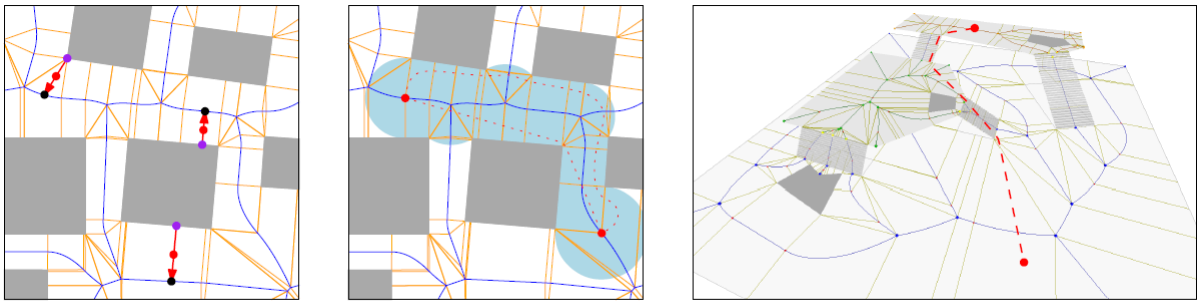
(a) Environment (b) Medial axis (c) Explicit Corridor Map

Figure 3.1: Generating an ECM [17].

The ECM has some interesting properties and features:

- **Automatic construction:** The ECM can be automatically and efficiently constructed, given a geometric 3D environment. The walkable space and set of obstacles is automatically generated as a first step.
- **Closest-point queries:** For a query point q in free space, the closest obstacle point can be easily found by finding the cell in which q is located.
- **Retractions:** Each point q in free space has a unique mapping onto the medial axis. A retraction defines that point on the medial axis which is the intersection between the medial axis edge and the line defined by q and its closest obstacle point. Retractions are used to send a character along its global indicative route (see below).

- **Clearance information:** The 'free width' of a point on the ECM, denoting how much free space there is around an agent (distance from q to the closest obstacle). This is useful for path planning: Am agent with radius r can't walk along an edge with a clearance below r.
- **Corridors and Indicative routes:** A character's path planning on the medial axis doesn't just consist of the path itself, but also a description of the free space around it, i.e. the sequence of corresponding ECM cells with extra disk segments around the vertices. This is called a corridor, hence the name 'Explicit Corridor Map'. Within a corridor, a character can plan an indicative route, one that stays on 1 side of the corridor, or the shortest path with a certain clearance. See Figure 3.2-b.
- **Multi-layered environments:** The ECM framework allows for multi-layered environments, i.e. multiple 2D layers connected by line segments. This is useful in simulating a crowd in multi-layered environments such as buildings.
- **Dynamic updates:** The ECM framework allows for dynamically updating the ECM locally after inserting or deleting an obstacle.



(a) Closest points / Retractions (b) Corridor / Routes (c) Multi-layered environment

Figure 3.2: Properties of the ECM [17].

With all these properties of the ECM framework, combined with collision avoidance, it allows simulating visually convincing crowds with tens of thousands of characters in real-time on a single machine. We will be using this framework to create a distributed version, with the goal of greatly scaling up and reaching up to a million characters in real-time.

3.2 Current Simulation Substeps

The current ECM framework has 13 substeps per simulation step [17]. A simulation step is typically executed every one tenth of a second. The simulation substeps are:

1. **Compute Retractions:** Computes and sets the new retraction point for a Character within the ECM.
2. **Update Densities:** Updates the density map by updating the single density values per character.
3. **Handle Dynamic Updates:** Handles dynamic updates to the ECM such as insertions/deletions of obstacles. Recomputes the ECM and updates all characters in the simulation per dynamic update.
4. **Update CQC:** Updates the character query structure (CQC).
5. **Replanning:** Replans character paths when necessary.
6. **Update Group Info:** Updates group-related simulation information.
7. **Compute Preferred Velocities:** Computes a preferred velocity for each character in the simulation.
8. **Update Preferred Velocities Streams:** Updates the preferred velocity of characters when necessary, based on the streams method.
9. **Compute Actual Velocities:** Computes the actual velocity for each character in the simulation.
10. **Compute Group Forces:** Computes the group forces being applied to characters.
11. **Apply Forces:** Applies forces to characters.
12. **Update Positions:** Updates the position of all characters in the simulation.
13. **Update Activity Areas:** Updates activity areas in the ECM. These are areas that trigger events for agents that enter or exit them.

3.3 Shared Memory Multiprocessing

In order to speed things up on a single machine running a crowd simulation in a virtual environment, it's smart to consider parallelizing some processes using shared memory multiprocessing. This is different than distributing the work over multiple machines since different machines do not share memory. For instance, the ECM framework uses OpenMP to distribute some of its substeps mentioned in the previous section onto different threads running on the same machine. Substeps are parallelized separately, meaning that all threads need to finish performing a substep and synchronize before moving onto the next substep. Some of the substeps in the ECM framework iterate over all characters in the simulation, where each character updates its own information. If the information being updated is independent of other characters' information, then this substep can be distributed easily over multiple threads. However, sometimes characters need to access (read) and write into shared objects, such as the density map when updating densities; in that case *atomic* statements are used.

3.4 Area-Of-Interest Data Sharing

In this work we will strictly deal with axis-aligned partition structures for ease of spatial definition/ownership and node-neighbor calculations. We use a structure that minimizes communication between neighboring nodes (sending and receiving agents). Steffen et al. [6] mention the use of ghost layers or ghost areas, which hold the data shared between two neighbor nodes. So, at each node, it will hold copies of agents that are close enough but in the neighboring node. This reduces the amount of communication between nodes when processing nearest-neighbor data. The ghost layer was mentioned in other papers as well. Wijerathne et al. [7] make use of it in their model for large-scale evacuation simulations based on Multi Agent Systems (MAS). At each time step, ghost layers are maintained and updated. For each domain or cell, a k-d tree partitioning is used as a character query structure, so that it is easy to detect the nearest neighbors of each agent within the cell. Aguilar et al. [12] define the ghost layer in more detail: Each partition has an "inner" layer, that does not need any neighbor information from other nodes. In other words, all the neighbors of the agents inside this layer are also inside this layer. The next layer, which is the one at the boundary of the partition, is the "to_send" layer. This region holds the information of all the agents that are close enough to the boundary and that need information of agents in neighboring nodes. The last layer is the ghost layer, or the "to_receive" layer. This layer holds copies of the agents in the neighboring node that are needed for the agents in the "to_send" layer.

Updating these agents per time step gives each node all the information it needs, without the need of communicating with neighboring nodes each time it needs nearest-neighbor information. This structure has only been used with a grid partition in previous work, but it can be extended to work with any axis-aligned partition, such as k-d trees.

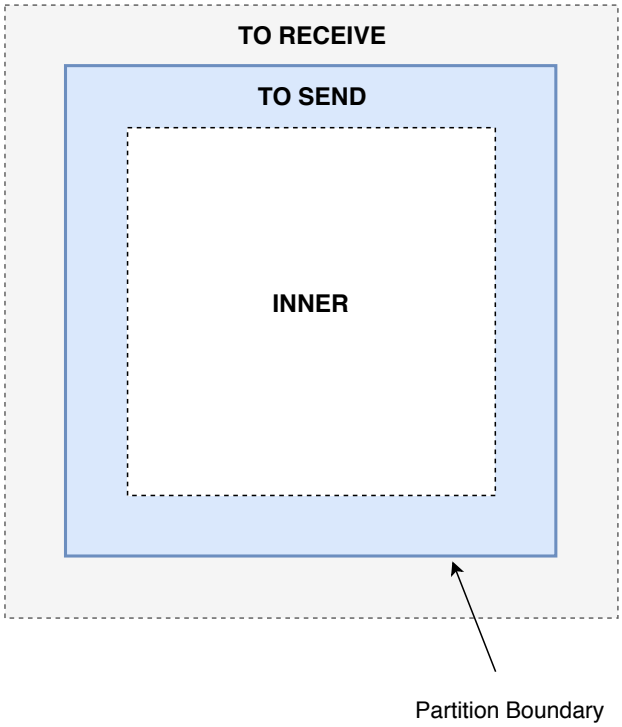


Figure 3.3: Ghost layer model.

4 Distributed Simulation Architecture and Substeps

4.1 Architecture and Layout

The distributed crowd simulation system will follow a networked-server architecture. A *Simulation Manager* will handle all API calls and command nodes by sending them actions. The simulation manager will also handle load balancing by periodic re-partitioning the environment to ensure that all nodes have around the same number of characters. A *Simulation Node* will be the working unit of the system that handles a single partition/region of the environment. The number of nodes will be specified by the user as an input argument. The simulation node performs a single step of the simulation, and distributes needed information across other nodes in between certain substeps. MPI will be used for inter-communication and communication between the nodes and the manager. It is important to note that the implementation of this work is built on top of the ECM framework, but the architecture and logical process can be generalized to any working framework for crowd simulation.

Below we can see an illustration of the architecture and work flow:

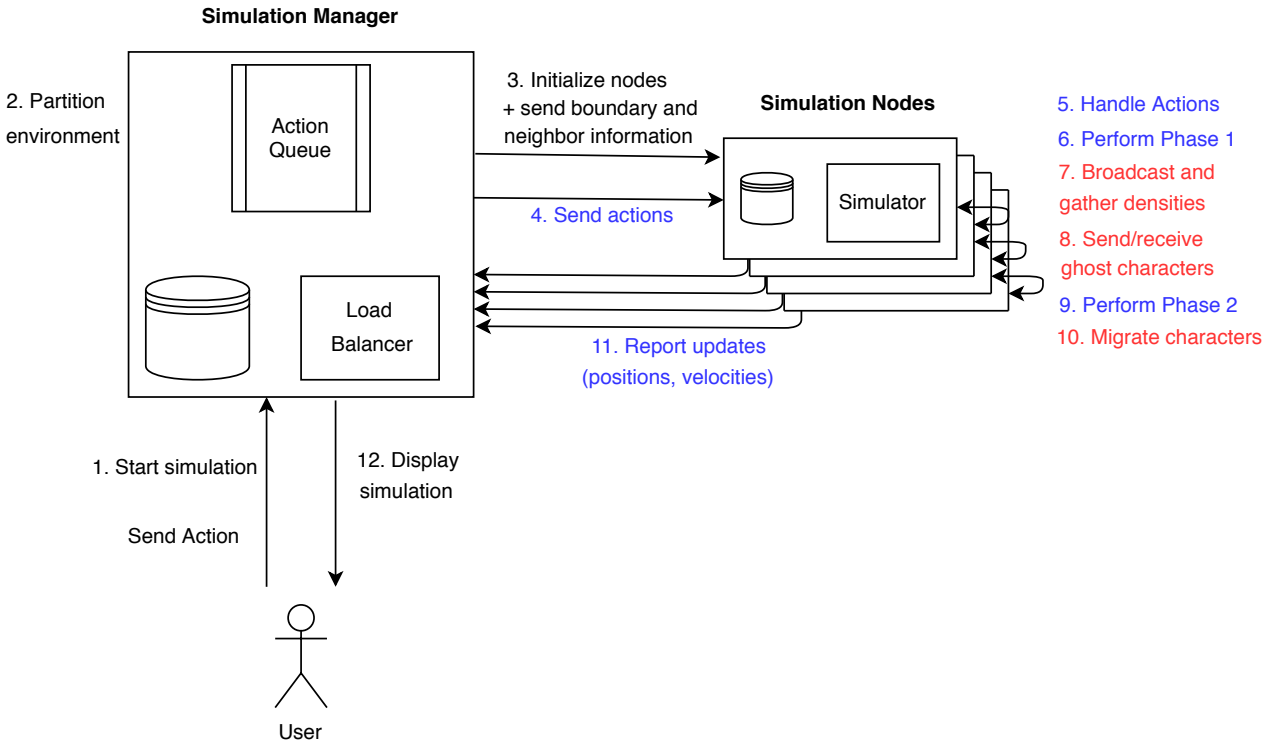


Figure 4.1: Distributed crowd simulation system architecture.

In the illustration above, the colored text (blue and red) refers to the substeps in a simulation step. Red text refers to the substeps dealing with intercommunication between nodes. The simulation manager can receive the command to start a simulation from a user; in that case it will partition the environment and set up the available nodes, such that each node handles a single partition. After partitioning the environment, the manager knows the AABB (axis-aligned bounding box) of each partition, as well as the neighboring partitions with their AABBs (and IDs) for each partition; the manager sends this information to each corresponding node so that all nodes are aware of their spatial ownership and the neighboring nodes that surround them. Once the simulation has started, a user can interact with the simulation via actions. An action can be anything from adding/removing characters, to changing character settings, to dynamically inserting/removing obstacles, and more. The user can send an action to the manager at any time, which will be added to the queue, and sent to the corresponding node(s) at the beginning of a simulation step.

At this point the actual simulation begins: At each simulation step, the set of nodes will handle and execute all actions sent to them, perform all substeps and communicate required information between each other (such as densities and ghost characters). The substeps listed in Section 3.2 are divided into two phases: Phase 1 consists of substeps 1 to 3, and phase 2 consists of substeps 4 to 13. Density information is needed for a character to plan its path. This event does not occur often, as it only happens when there is at least 1 character in any node that needs to re-plan its path. The manager knows beforehand if densities need to be shared (for example if an obstacle was inserted or deleted in the environment, that would also trigger characters to need to re-plan), and thus instructs the nodes with an action if there is a need. Otherwise this intercommunication between the nodes is omitted. Ghost characters are minimal copies of characters that are sent to neighboring nodes. Ghost characters are required for calculations such as collision avoidance. A character can also cross to another region and will therefore be transferred to another node at the end of a simulation step (migration). Each node is aware of the entire environment and keeps an updated copy of it; it can therefore handle global path planning decisions for the characters that it holds. After the completion of a simulation step, all nodes will send the updated character position and velocity information back to the manager. The manager awaits these updates and processes them, after which it will begin another simulation step.

4.2 Distributed Simulation Substeps

The simulation manager and simulation nodes perform different substeps. The manager dictates when the nodes can begin performing its substeps with an action.

4.2.1 Manager Substeps

1. **Broadcast 'Perform Step' action:** The manager broadcasts an action to all nodes, instructing them to begin performing a step.
2. **Send all actions:** The manager sends all its queued actions to the corresponding nodes.
3. **Repartition:** The manager repartitions the environment periodically, so this substep only happens periodically (fixed) and not at every step. This helps to balance the work load between nodes, ensuring that all nodes have around the same number of characters. This is obviously made for balanced partitioning structures like a k-d tree, and not a grid.
4. **Receive node updates:** The manager receives updates from all nodes. There updates include updated character positions and velocities. The manager also checks if any character migrates and updates its mapping from character to node.

4.2.2 Node Substeps

1. **Await an order to start:** A node awaits a broadcasted message from the manager to start a simulation step.
2. **Receive and process all actions:** A node receives all queued actions from the manager (with the needed additional information) and executes them.
3. **Repartition:** If a node receives an action to repartition, it means that the manager has repartitioned the environment in the previous step, and now the node needs to receive its updated AABB and node neighbor information, as well as migrate the characters that don't belong in it anymore.
4. **Perform phase 1:** A node performs the simulator substeps of phase 1, which are computing retractions, updating densities and handling dynamic updates.
5. **Synchronize.**

6. **Updating the density map:** The density map is needed in its entirety for global path planning purposes; a character needs to know all densities to compute a path. Therefore, after each node has updated its own densities, it has to broadcast them to all nodes so that each node can build a complete density map. The intercommunication between the nodes in this step results in all nodes sharing their densities and having the same global updates density map. This step only occurs when needed, i.e. when at least one character in any node needs to re-plan its path.

7. **Exchange ghost characters:** Ghost characters need to be exchanged between neighboring nodes. Each node needs to first send the characters that affect a certain neighboring region to the corresponding node, then receive all ghost characters from neighboring nodes. A ghost character is a local copy of a character in another node, but with limited data which includes the ID, layer, radius, position and velocity of the character. The layer denotes the ID of the two-dimensional layer in a multi-layered environment that the character resides in. Ghost characters are crucial for the substeps in phase 2.

8. **Perform phase 2:** A node performs the simulator substeps of phase 2, which are: updating the Character Query Structure, replanning, updating group info, computing preferred velocities, updating preferred velocity streams, computing actual velocities, computing group forces, applying forces, updating positions and updating activity areas.

9. **Remove ghost characters:** Each node will remove the ghost characters it has.

10. **Migrate characters:** Each node checks if any of its characters stepped out of its region; if any did, it packages them and sends them to the corresponding node (migration). The node will then check to receive migrated nodes for itself; if it receives any, it will add them to its simulation. To minimize communication, a node will only check to receive migrated characters from a node if it received ghost characters from it in the same simulation step.

11. **Report updates:** At the end of a simulation step, each node will report its updated character information back to the manager (mainly positions and velocities).

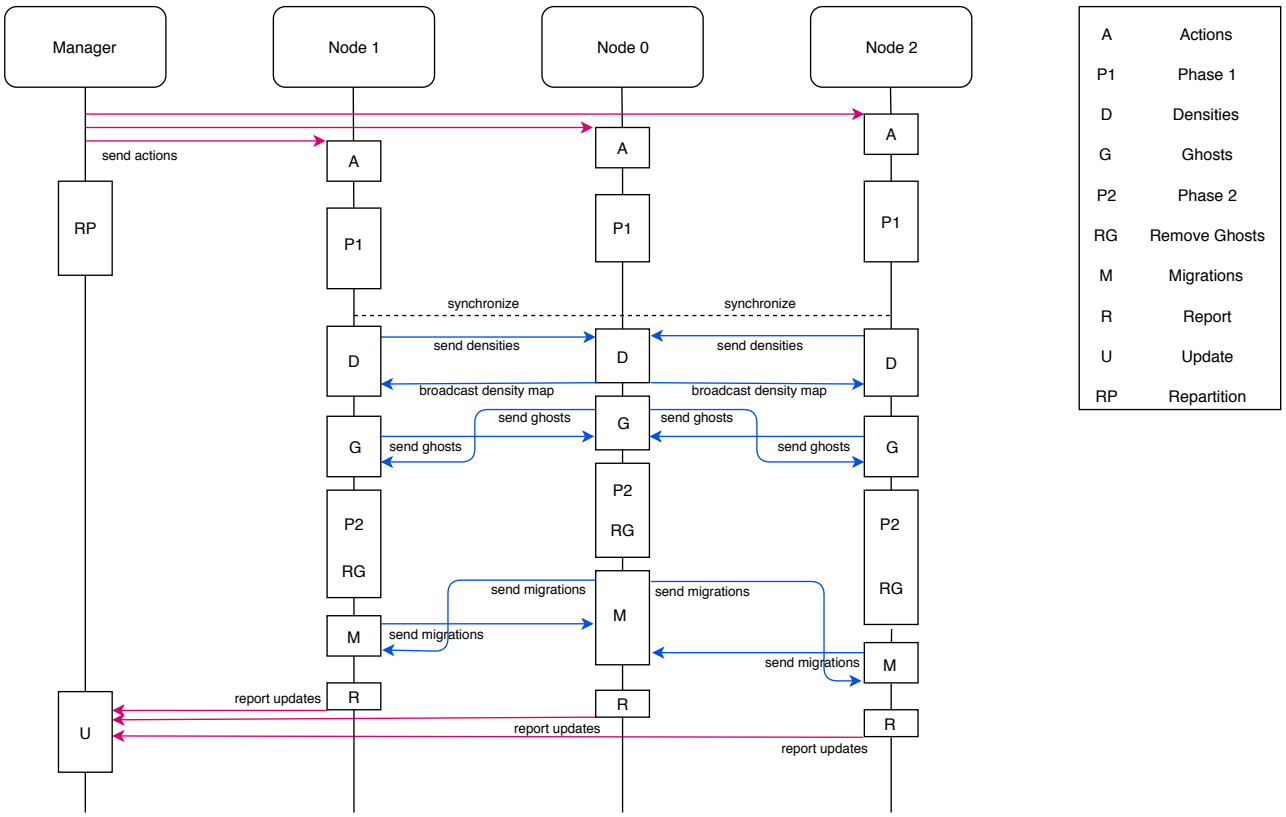


Figure 4.2: Sequence diagram for a single simulation step.

In Figure 4.2 we can see a sequence diagram representing a single simulation step. All substeps are denoted in blocks (see legend). These blocks represent processing activity, and communication during these substeps is illustrated with directional arrows. Some substeps are comprised of both processing and communication parts; for example migrations entail checking which characters need to migrate, packing these characters into a single buffer, sending them to the corresponding neighboring node, waiting for and receiving migrations from neighboring nodes, unpacking the received characters and adding them to the simulation. Intercommunication between nodes is denoted in blue arrows, while manager-node communication is denoted in red arrows.

All nodes synchronize once after phase 1. Density communication is synchronous, while ghosts and migrations are asynchronous: Ghosts and migrating characters are first sent (asynchronously), then received. After receiving ghosts/migrations, all sent signals are waited for to be completed before moving to the next substep; the sent signals only return when the target has received them. Sending actions and updates are done synchronously, but the manager receives updates in any order (from any node), and starts processing and

updating its information upon receiving each update. The manager might also repartition the environment while waiting for the nodes to finish working. This is done periodically and not in every simulation step.

5 Partitioning a Virtual Environment

5.1 Grid

A logical initial way to partition an environment is to use a grid. The grid can split the space into squares or rectangles of equal size, called cells. One way to do this is to assign a cell size and build a grid based on that, after which a certain number of cells are assigned to a specific node. Each node would handle a certain number of cells. It would make sense for each node to handle an equal number of cells. A simple linear mapping from grid cells to a grid of nodes should do the trick. Another way, which is the implementation we use, is to build a grid based on the number of available nodes. Therefore, each cell would correspond to one node. The number of nodes is set and fixed in the beginning, so the grid is built based on that number to make use of all nodes. The grid always favors and prioritizes a square, but if that's not possible with the given number of nodes, it tries to go for the closest rectangle to a square. For example if we have 25 nodes, that gives an easy 5x5 grid. If we have 12 nodes, the closest rectangle to a square would be 4x3, so that's the grid that would be built. If we have 7 nodes (prime), then the only way to partition the environment is into a 7x1 grid.

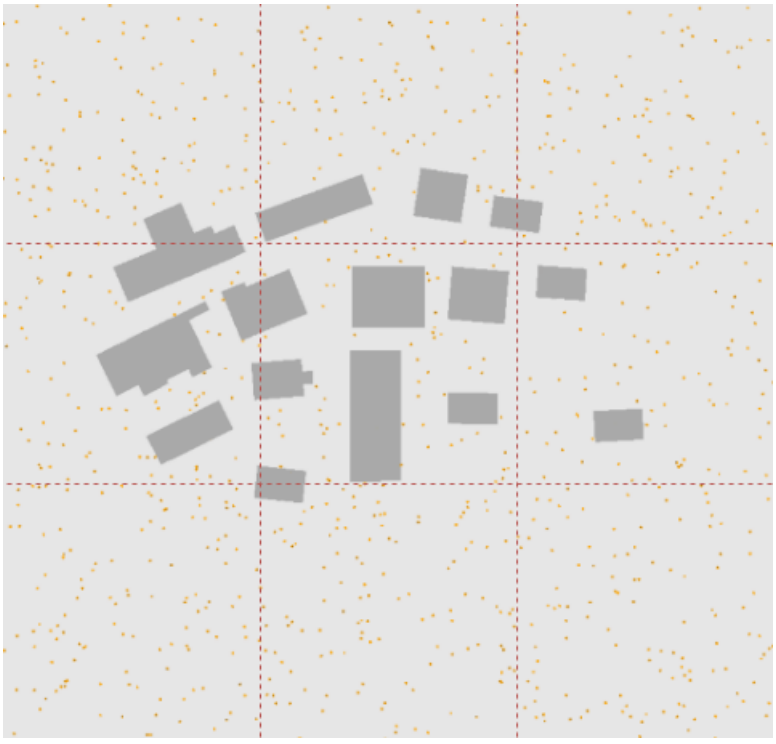


Figure 5.1: Grid partitioning.

The downside of a grid is obvious: it's not balanced. Different cells could have a different number of characters, and some could even be empty, resulting in different execution times for different nodes. This is a problem since some nodes would have to wait for other to finish because there are multiple synchronization steps. A grid is only a good solution if we have a balanced population over the environment. In order to balance the load between nodes, a smarter and more balanced partition should be considered.

5.2 K-d Tree

Another partition structure that is well worth using is the K-d tree [8]. The advantage of using a k-d tree is to have a balanced partition. Unlike a grid, a k-d tree uses character positions as input when being constructed, so that the left and right partitions of every cut are balanced and have an equal number of characters. The k-d tree is axis-aligned, where each cut is either in the x or the y direction. Cuts in the x and y direction alternate. Note that we will be using the term 'tree node' to describe a node in the k-d tree, which represents a spatial partition with a parent and children under it, not to be confused with a 'node', which is the term we are using for a single working unit in the simulation (such as a computer in a cloud).

The k-d tree that we will be using is a slight variation of the normal k-d tree. First of all, at each tree node, which represents an axis-aligned partition, we can cut it into any number of sub-partitions (but all in the same direction, either x or y). Secondly, the number of cuts at each tree node are predetermined by the prime factorization of the number of nodes available. Using the prime factorization as a 'blueprint' for partitioning makes sense since it ensures that we have the same number of partitions under each tree node. Each tree node will hold the same number of tree nodes, and so will their children, and so on. For example if we have a number of nodes that is a power of 2, then this would result in a normal k-d tree with a single cut in the middle (median of character positions) per partition. If we have 12 nodes, then the root tree node (the entire environment) would be first cut into 2 sub-partitions, each of which would be again cut into 2 sub-partitions, each of which would be cut into 3 sub-partitions, making a total of 12 partitions and 12 leaf tree nodes. So unlike a normal k-d tree, the base/termination condition is reaching the last number in the list of cuts (determined by the prime factorization of the number of nodes available).

Instead of cutting a partition at the median, we cut it at the k-median, where k represents the number of children or sub-partitions at this step of construction. So if we have 2 cuts, resulting in 3 children, then we would want to cut at the a third of all positions, as well as at two-thirds of all positions. This would require us to sort the character positions by either their x- or y-values beforehand.

Nodes need to know who their neighbors are and where they are (their spacial ownership) so that they can send/receive ghosts and migrations properly. Therefore, neighbors are also calculated and stored during construction for each tree node. The ghost layer structure proposed in Section 3.4 is extended to work with the k-d tree: as long as each node knows its own AABB and its neighbors with their AABBs, this structure can be used. The way neighboring tree nodes are calculated is as follows:

1. The root tree node has 0 neighbors.
2. Each new tree node will check if an of its parent's neighbors are also neighbors of its own.
3. Each new tree node will check if any of its sibling tree nodes are neighbors.

To check if 2 AABBs are neighbors, we check the shortest distance between them and compare it to the ghost layer distance, which also corresponds to the maximum visibility distance for a character (how far a character can see). This is also the radius that is used to find the k-nearest neighbors for collision-checking purposes. The maximum visibility distance or ghost layer distance in our system is set to 10 meters.

In the case of where there are no or insufficient characters in a partition that needs to be cut (i.e. the number of characters is less than the number of cuts), then we refer to another method for further partitioning. Instead of using the k-median method, we resort to partitioning the AABB independent of the characters that reside in it: We perform spatial partitioning such that the children (sub-partitions) have equal-sized AABBs. If we start a simulation with no characters, then this method will be used in the entirety of the tree's construction. Upon adding more characters, we can reconstruct to create a character-balanced tree.

6 COMMUNICATION METHODS

Communication is an integral part of a distributed system. In a distributed simulation, all simulation nodes need to communicate with each other (intercommunication) and with a manager, which will give commands (actions) and collect updates from the nodes. All communication will be done through MPI (Message Passing Interface) which offers a wide variety of advantages:

- **Optimized communication:** MPI is very optimized when it comes to parallel computing; it provides efficient point-to-point and collective communication based on the given topology.
- **Standardization:** MPI is the only message passing library that is considered a standard.
- **Portability:** There's no need to change the source code when porting to a different platform.
- **Network Code Abstraction:** With MPI there is no need to write network code, it does this automatically. There is no need to rewrite the code for a network or cluster, since the same code that works on a single machine (using threads) works for a network of computers as well.

6.1 Manager-Node Communication

All communication between the manager and the nodes is done via **Actions**. The manager holds a queue of actions that are sent to the corresponding nodes at the start of each simulation step. The first action sent to all nodes is to start the simulation. Other actions include, but are not limited to:

- Ending the simulation.
- Toggle pause.
- Add new character.
- Remove character.
- Remove all characters.
- Set character position.
- Compute path for character.
- Add character goal.
- Add group goal.
- Add character to group.
- Remove character from group.
- Add activity area.
- Remove activity area.
- Insert obstacle.
- Delete obstacle.
- Update densities.
- Set parameter values.

The simulation manager keeps track of where each character is (which node it is being handled by), and so certain actions are only sent to those corresponding nodes, like for example adding/removing a character. Other actions have a global effect and are thus sent to all nodes, such as inserting/deleting an obstacle since all nodes need to update their copy of the environment. Certain actions require the master to send additional information, such as adding a character requires sending the new character's ID, position, layer and character settings. The manager keeps track of this, mapping the additional data to their corresponding nodes. At the end of each simulation step, the nodes report updates back to the manager, which includes sending all character positions and velocities. This is done without an action, as the manager expects this update at each simulation step.

6.2 Node Intercommunication

Nodes have 3 intercommunication points within a simulation step, and 1 synchronization point right before density and ghost communication. These intercommunication points can be seen in Subsection 4.2.2.

- **Sharing Density Information:** The first communication between nodes is gathering and sharing densities in order to update the local density map. A single node assumes the job of gathering all densities from all nodes, typically the root node with id 0. Since each node has a different number of densities to pack and send, densities of all nodes are packed into a size that is agreed upon by the manager. The manager sends this size in the beginning of a step; this size is the maximum number of characters in any node. If this size changes at the end of a step, the manager becomes aware of the change and sends the updated size as an action to all nodes in the next step. After the root node receives all densities, it builds the complete density map and broadcasts it to all other nodes. It is important to note that this communication event rarely happens, as nodes only need updated densities when one (or more) of its characters needs to re-plan its path. An illustration of this step can be seen in Figures 6.1 and 6.2 below.

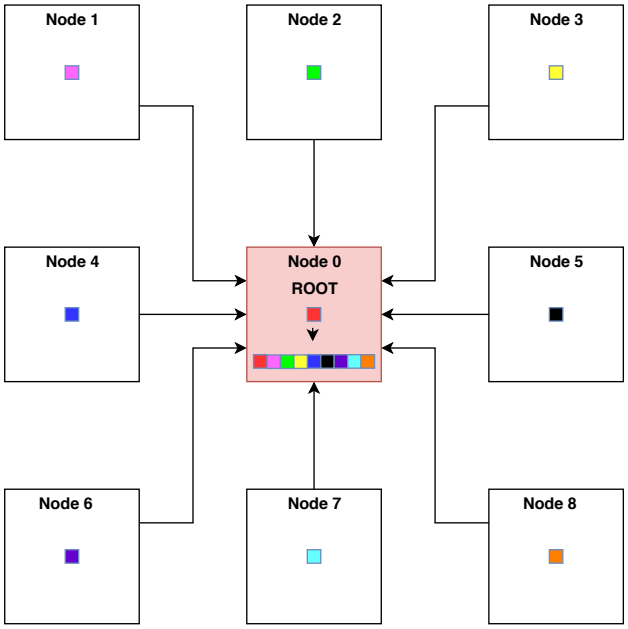


Figure 6.1: Gathering densities.

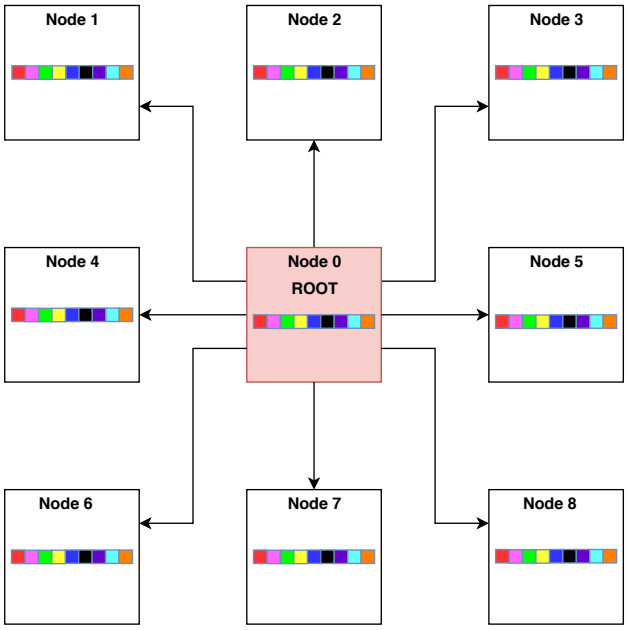


Figure 6.2: Broadcasting densities.

- **Exchanging Ghosts:** Neighboring nodes need to send and receive ghosts between one another. Ghosts are copies of characters that hold limited and necessary information used for character streams and neighbor calculations. Each node first checks which of its characters it has to send to which neighboring node as ghosts. After that, each node sends its ghosts to the corresponding neighbors; this is done asynchronously. After sending ghosts, each node then receives ghosts from its neighboring nodes. Finally all sent signals are waited for to complete and return, which only happens after the targeted neighbor node has received the signals on the other end. A basic illustration of this communication can be see in Figure 6.3 below.

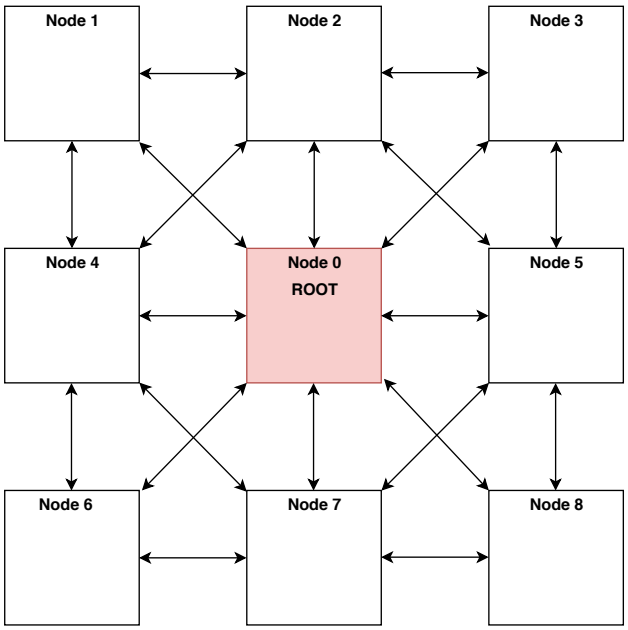


Figure 6.3: Transferring ghost agents between neighboring nodes.

- **Migrations:** Sometimes characters cross the borders of the bounding box defining the spacial ownership of a node. When that happens, these characters need to migrate to the neighboring node. This communication event is essentially similar to exchanging ghosts, but entire characters (with all their information) are sent and received. Nodes first check which characters need to be migrated (and to which nodes), then send all these characters to the corresponding neighboring nodes (asynchronously). After that each node receives migrations from its neighbors. Similarly to the ghost substep, all sent signals are waited for to return before moving on to the next substep. This step is done at the very end, after completing all simulation substeps.

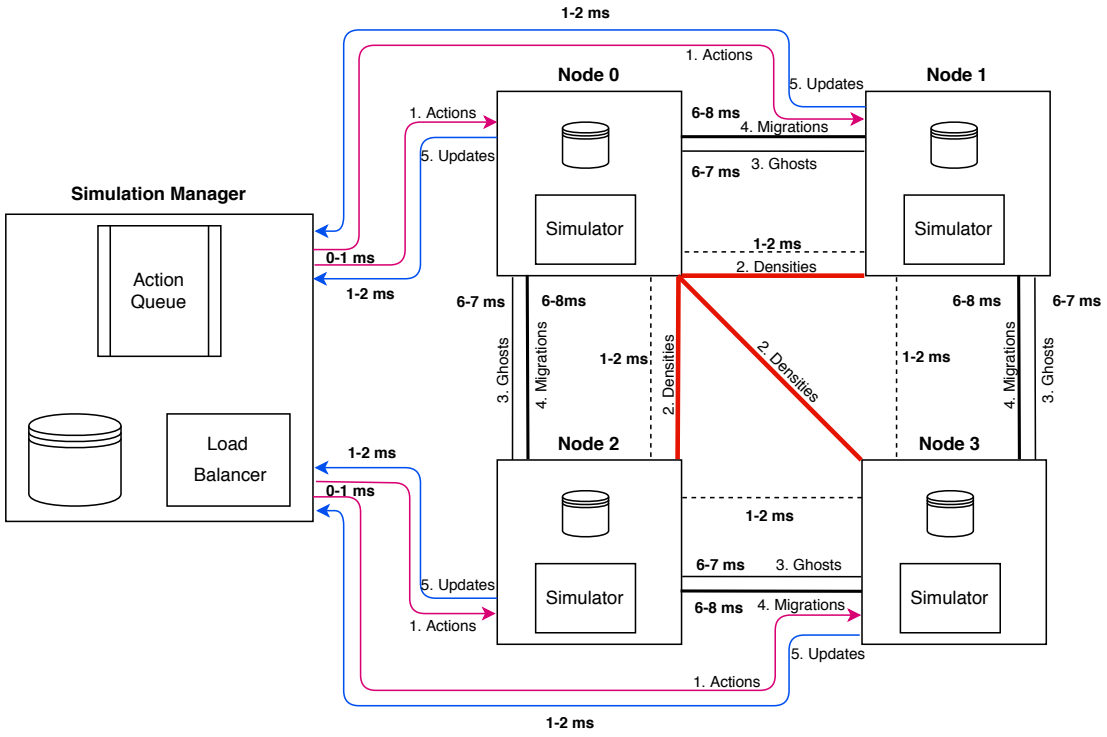


Figure 6.4: Communication between 4 nodes and a manager (with real times).

In Figure 6.4, we can see an illustration of the communication happening between 4 nodes and a manager. In this example we have 4 nodes with the following neighbor relations:

- Node 0 is neighbors with nodes 1 and 2.
- Node 1 is neighbors with nodes 0 and 3.
- Node 2 is neighbors with nodes 0 and 3.
- Node 3 is neighbors with nodes 1 and 2.

In this specific example, nodes 0 and 3 are not neighbors, and neither are nodes 1 and 2 (but they could be, if the shortest distance between them is less than the ghost-layer distance). Node 0 acts as the root node that collects all density information from other nodes, and sends back the complete density map. Real numbers are denoted in the figure, representing the range of execution time for the communication substeps. The results were obtained by simulating 10K agents on 4 nodes (and 1 manager), with 11 threads for OpenMP use per node, on a 5000x5000 environment ("city_minimal_10x10") and a realistic scenario of a crowd crossing the city from left to right. The details for the experimental setup and simulation results are illustrated in the experimentation section (See Section 11). To put

things into perspective, the total time for a single simulation step on average for this case was around 100 ms (i.e. real-time).

7 WORKLOAD BALANCING

The way workload is balanced across all nodes through periodic repartitioning. Obviously a grid partition would result in the exact same grid each time we build it, and would therefore not help in balancing the work. The grid by itself is not balanced and does not take character positions into consideration. When using a k-d tree, however, repartitioning periodically every certain number of steps ensures that we have a balanced system again. Every time we repartition, the k-d tree is built from scratch. The AABB and neighbor information for each node changes, and so that is sent to the nodes as an action in the next step. Each node updates its own AABB and neighbor information (a list of neighbors with their IDs and AABBs), after-which it check if any characters don't belong in it anymore (since its spatial ownership has changed). Since nodes don't have access to the spacial ownership of every other node (only their direct neighbors), then characters that don't belong anymore can't always be migrated to the correct node, if the correct node is not a neighbor. Therefore, each character belonging to a different node is migrated to the closest neighbor that minimizes the distance between the character's position and the AABB of the neighbor. This is done repeatedly until all characters propagate to their correct nodes. The maximum number of propagations or jumps in the worst case scenario is equal to the total number of nodes.

8 MULTILAYERED ENVIRONMENTS

It's possible to have multi-layered virtual environments, so that character positions can be annotated by an (x,y)-point and a layer. These different layers are connected to one other by ramps, stairs or other means. This allows us to create more realistic environments and scenarios, such as a hospital building with multiple floors or even an entire city. The ECM framework already supports multilayered environments, and it would be interesting to see how this translates to a distributed version.

When partitioning a virtual environment (using a grid, k-d tree or another axis-aligned partitioning method), each partition handles a 2D region of the environment: this region can be multi-layered, and a single node would handle all the layers found in that single region. We can think of it as cutting a cake with multiple layers; we cut it from the top, and each slice is a partition with multiple layers. Another thing to consider is the ghost agent structure: Each node requires ghost agents from its neighbors, and so all characters in the sending layer (in all layers of the environment) would be sent to the neighboring node. It might be the case that ghost agents that are received exist in a different layer than the ones in the current node, which would make that communication useless since the ghost characters would not be needed and would not affect the characters in this node. In any case, all needed ghost characters are always sent and received, and so multi-layered environments are supported with the current architecture.

9 GROUPS

One of the features of the ECM framework is allowing multiple characters or agents to move as a group. The group has a leader and other members following it. The positions and velocities of the members affect each other so that they stay close to each other. The groups are meant to be of small size, between 1 and 4 members. The dependency between the group members makes it difficult to distribute them across multiple nodes. Splitting group members across the borders of nodes would need every node that holds at least one of the members to also hold a copy of the group and all the group's needed information, making communication quite costly at each step. The extra overhead of communicating group and member information across neighboring nodes would not scale well, and thus we have to think of another way to incorporate groups into a distributed crowd simulation system.

Since the groups are meant to be of small size, we can simply always assign all members to the same node, even if some of the members don't belong in the spatial vicinity of the node's ownership. One way to do this is to let the leader of the group decide where the entire group will reside. Once the group leader migrates, all the other members will migrate with it (as well as the group itself). Nearest neighbor information would still be provided as ghost agents, given the ghost layer thickness is sufficient.

10 IMPLEMENTATION

10.1 Distributed ECM Framework

Building upon the ECM framework, we have added 3 main new classes: **SimulationManager**, **SimulationNode** and **MPIBase**. These classes handle the distribution of simulating a crowd through the ECM framework.

The **MPIBase** class contains many MPI methods for sending, receiving, broadcasting, packing and unpacking different types of data. It also contains simplified character and action information structures, as well as custom MPI types.

The **SimulationManager** class represents the simulation manager. The manager includes several public methods that the user can call. These methods could potentially trigger adding actions to queues, and are quite similar to the methods in the **Simulator** class, such as starting a simulation, performing a simulation step, adding a character, adding a group, adding a goal, adding an activity area, or changing a certain parameter. The simulation manager includes an instance of the **Simulator** class for the sole purpose of using its methods and holding character information; it does not use the simulator class to simulate on it or perform any simulation steps. The manager has a map that maps node IDs to action queues, which represents the actions that node has to receive.

The **SimulationNode** class represents a simulation node, which handles a single partition in the environment. Each node is aware of the entire ECM (i.e. navigation mesh data structure), and is therefore able to perform global path planning for its characters. Each node is also aware of its ownership in the environment, meaning that if one of its characters steps out of the region the node governs, it has to migrate it to the neighboring node. The SimulationNode class includes an instance of the **Simulator** class, which is used to perform all substeps. In a way, the SimulationNode is a wrapper to the Simulator, but with communication and action-handling methods.

10.2 Partitioning the Virtual Environment

Partitioning the virtual environment is written in a generalized way to incorporate any partition that is axis-aligned. The **PartitionStructure** class handles partitioning an environment. The two sub-classes **PartitionStructure_Grid** and **PartitionStructure_kdTree** inherit from **PartitionStructure** and represent a grid and k-d tree partition respectively. We can add more partition structures that inherit from it as long as they are axis-aligned. A single partition is defined by an AABB and a vector of neighbors; a neighbor is defined by an ID that represents the partition it belongs to, and an AABB. The four main and generalized methods that each partition structure needs to implement are the following:

- **build:** Builds the partition structure. For a grid, it creates a grid of cells based on the number of nodes we have. The grid favors a square or the closest rectangle to a square. Setting the neighbors for each node is simple: Each cell can potentially have 8 neighbors around it. For a k-d tree, building a tree has been explained in Section 5.2. The tree is built in a breadth-first form, where each tree node stores the following:
  - ID: The tree node's ID.
  - depth: The tree node's depth level in the tree.
  - leaf: whether this tree node is a leaf or not.
  - AABB: The bounding box defining this node.
  - characters: The characters spatially residing in this node's AABB.
  - children: The children tree nodes.
  - neighbors: The neighboring tree nodes.

  Neighbors are set during the tree's construction: For each child, we check if any of the parent's neighbors are also neighbors, and then we check if any siblings are neighbors. This would result in the leaf nodes having neighbors that aren't leaves.
- **queryPoint:** Returns the ID of the node where the query node belongs. For a grid, we can easily query the position since we knows the grid's boundaries and the size of the cells in the x- and y-direction. For a tree, a depth-first search is performed on the tree from the root to the correct leaf node.
- **getNodeAABB:** Returns the AABB defining the node's spatial region.
- **getNodeNeighbors:** Returns the neighbors of a node. For a grid it's straight forward. For a k-d tree, we only return the neighbors that are leaves.

10.3 MPI

The MPI implementation that we use is OpenMPI. The libraries MPICH2 and Microsoft MPI also work and were used in early development.

10.3.1 Communicators

Each process in MPI must belong to a communicator, which is an object in MPI that describes a group of processes. All processes belong to the *MPI_COMM_WORLD*. It is beneficial to separate the manager from the nodes and have them in separate communicators; the reason for this is to be able to use communicator-specific methods for communication, such as for broadcasting and gathering information across all processes of a communicator. Therefore the base communicator the *MPI_COMM_WORLD* can be split into two, depending on the rank (ID) of the process, via **MPI_Comm_split**. The root process with rank 0 will be assigned to one communicator, while all the rest with a rank greater than 0 will be assigned to another communicator (thus separating the simulation manager and nodes). One problem lies ahead, which is making processes in different communicators to communicate with each other (manager-node communication). To solve this issue, we create intercommunicators, which are used for communication between 2 disjoint groups, done via **MPI_Intercomm_create**. The simulation nodes receive a manager intercomm to communicate with the manager, and the manager receives a nodes intercomm to communicate with the different node processes.

10.3.2 Point-to-Point Communication

For the most part, **MPI_Send** and **MPI_Recv** were used to handle point-to-point communication, such as for communicating actions from the manager to the simulation nodes, reporting node updates to the manager, and sending AABB and node neighbor information upon each repartition. For more deadlock-sensitive communication, such as transferring ghosts and migrating agents, **MPI_Isend** and **MPI_Irecv** are used for non-blocking point-to-point communication.

10.3.3 Collective Communication

For sharing density information, collective communication is used, namely **MPI_Gather** and **MPI_Bcast**. **MPI_Bcast** is also used in other collective communication, such as broadcasting an action to all nodes to start a simulation step at the beginning of each step.

10.3.4 Dynamic Receiving

When receiving a message, its size is unknown. To work around that, **MPI_Probe** is used to peek at the received data before actually receiving it, creating an MPI_Status object. The status is then used an in input to the **MPI_Get_count** method, which gives us the size of the message being received. After we know the size, we can create a buffer of exactly that size and receive the message into it.

10.3.5 Packing

All data that is sent is first packed into a single char buffer using **MPI_Pack**, which results the data being of typed MPI_Packed. If we have a pointer object, we first store whether or not it is a null pointer; if it's not, we then pack and store the data it holds. When receiving data, it is unpacked and copied into a char buffer using **MPI_Unpack**. After that the buffer is unpacked (in the same order that it was packed), and the objects are recreated from the unpacked data. Packing a vector or map requires to also pack the its size so that we know how much of the buffer is reserved for it when unpacking. This applies to any array or list that we pack, so that unpacking is done correctly.

10.3.6 Custom MPI Types

Custom MPI types can be created to be frequently used in packing/unpacking data. We have created MPI types for several objects such as characters, points, AABBs, density information, neighbor information, and more. Creating an MPI type does not store all members of a class, such as non-primitive members, booleans and vectors. These members will not be encompassed in the type and must be packed and unpacked separately.

10.3.7 Synchronization

Synchronization is needed before the first intercommunication step to ensure that all needed information is updated and ready to be shared between nodes (densities and ghosts). There is no need to synchronize after that because things are done asynchronously. **MPI_Barrier** is used for synchronization and synchronizes all processes in a given communicator.

10.4 Communication Methods

Several methods exist for communicating different data between nodes. This includes:

- Sending/receiving vectors and maps of different types.
- Sending/receiving ghosts.
- Sending/receiving character and group migrations.
- Sending/receiving node update information.
- Sending/receiving dynamic updates (action).
- Sending/receiving activity area parameters (action).
- Sending/receiving character position parameters (action).
- Sending/receiving group goals (action).
- Sending/receiving new character parameters (action).
- Sending/receiving AABBs and node neighbors.
- Gathering/Broadcasting density information.

10.5 MPI vs. OpenMP

As previously mentioned and discussed, there are two ways to distribute work in a crowd simulation. One way is shared memory multiprocessing, which is distributing work onto multiple threads that share the same memory; this is done on a single machine via OpenMP. The other way, which is the topic and core of this thesis, is distributing the work onto multiple machines (that do not share memory); this is done using MPI. Given the nature of MPI, we can also use it to distribute the work on a single machine, in which case it uses different threads (or processes). The ideal scenario is using MPI to run the distributed crowd simulation on a cloud or cluster of machines, and having each machine use OpenMP as well to distribute its own work further onto multiple threads. This makes our implementation a hybrid of MPI and OpenMP. The problem arises when we try to do this on a single machine: If we use MPI and OpenMP on a single machine, they will both use different threads and fight over the limited available threads on that single machine. Therefore, it is important to only use MPI when testing its effect in distributing a crowd simulation. This can be done by

setting the number of threads to be used my OpenMP to 1. Both MPI and OpenMP can be used (even on a single machine) as long as the total number of threads used does not exceed the number of threads available on all cores combined. For example if we have 24 cores with 2 threads per core, we can run 4 nodes with each using 12 threads for OpenMP, bringing us to a total of 48 threads being used.

10.6 OpenMP Optimizations

Other than the OpenMP optimizations done in the original ECM framework, OpenMP was used in the manager class when receiving node updates: Upon receiving an update from a node, character information in the manager is updated in parallel using OpenMP. Each character updates its own information independently (mainly positions and velocities), making it easy to use OpenMP to distribute this work.

10.7 Groups

Creating groups is done in two steps: First we add a character to the simulation, then we can add that character to a group. If that character is the first to be added to a group with a certain ID, then that group is first created before adding the character to it. The tricky part in a distributed crowd simulation system is adding a character that's in one node to a group that's in another node. This would require migrating the character first and then adding it to the group. The way this is done is that the manager sends actions to both nodes, telling one to migrate a character and another to expect a migration to be able to receive it. After that, the character can be safely added to the group. If too many of these instances occur in the same step, where characters have to first migrate to be added to groups, then rare deadlocks could appear. A few hundred of these instances per step seem to be non-threatening, but more than that can be problematic. Also if the number of nodes is very large, this would also result in a deadlock. Typically that doesn't happen that often in practice, and, furthermore, we typically want to create characters and add them to groups at the same time; we rarely want to add preexisting characters to groups. In that case, we can add group characters to the correct node (the node that holds the leader) beforehand, avoiding this whole mess in the first place.

In terms of correctness, groups in a distributed simulation might lead to slightly different results than ones in a single-machine version. If group members spread out too much, then the correctness of the lagging members would change a bit. If the ghost layer is insufficiently thick, then not all needed neighbors are transferred as ghosts, especially for the lagging members. If the area is not too crowded, groups members tend to stick close together, resulting in correct position and velocity updates.

When the need to migrate a group arises, groups are sent together with single-character migrations (in the same buffer). A vector of groups that need to be migrated is packed and sent, which minimizes communication. If there are no groups to migrate, then an empty vector is sent.

10.8 Testing

Functional testing was used to test for bugs and deadlocks in the system, but also to test for correctness. The latter is done by running a clone simulation on the manager, that has the exact same environment, characters and/or groups. At each time step, the manager sends its actions to the nodes then proceeds to simulate its own single-machine version of the simulation. When it receives the updates from the nodes, it checks per time step if all the character positions and velocities match the ones it its own simulation. If everything matches, then the distributed simulation is producing *correct* results.

11 EXPERIMENTATION

11.1 Setup

There are many factors/parameters to consider when experimenting with a distributed crowd simulation in a virtual environment. We will perform each experiment with a set of values for these parameters:

- **Virtual environment:** It's important to select representative environments that create different scenarios: We will choose a small congested environment, a large environment, and a multilayered environment (specified in the next subsection).
- **Number of nodes:** The number of nodes obviously affects execution time; the more nodes we have, the more distributed our system is.
- **Number of characters:** For each experiment, we will populate the environment with a different number of characters. For the most part, we will use a population of 1K and 10K characters. Since running simulations for 100K characters is very time consuming (can take up to 90 minutes for a single run), it's unfeasible to experiment with 100K characters for all experimental conditions; testing for different conditions could take days and even weeks of running the simulation. Therefore, we only simulate 100K characters for the conditions that prove optimal for 10K characters.
- **Number of threads for OpenMP:** For testing the effect of OpenMP vs. MPI, we will test with multiple numbers of threads, ranging from 1 to 24. This is the number of threads that each node is allowed to use for OpenMP distribution. Obviously we will not use more threads than available, so the total for MPI and OpenMP should always be less than the maximum available threads on our machine. For example if we have 48 threads available, we can run 4 nodes with 12 threads for OpenMP each.
- **Number of steps:** Certain experiments require running more simulation steps to see a proper effect, such as testing the effect of periodically repartitioning an environment. Some scenarios also require a stabilization period before starting to record, and so we will start counting and recording steps after that phase. We will test with 500 (active) steps for standard experiments and 2000 steps for repartitioning experiments. Scenarios that require a stabilization period will run for 1500 steps (1000 steps for stabilization and 500 active steps). Note that 1 step typically translates to 0.1 seconds of real time, but in our experimentation process, we eliminate any waiting time (in

case a step takes less than 0.1 seconds), and we simulate as many steps as we can in 0.1 seconds. Our goal is to see how long the average step takes, and to compare it to 0.1 seconds to determine if the simulation lies in the real-time range or not.

- **Number of runs:** Each experiment will be run 5 times with the same parameters/variables so we can get an idea about the standard deviation.
- **Character scenarios:** For each experiment, we will test two different scenarios:
  - **Random:** A scenario that populates the environment with a uniform random sample. The character and goal positions are chosen at random (on any layer) and a path is calculated between each pair of positions. If no path could be calculated, we generate new uniformly random pairs of character/goal positions, until we have a successful path. All characters are added instantaneously at the beginning of the simulation, and so recording simulation times also starts at simulation step 0. See Figure 11.1 for a visualization of this scenario.

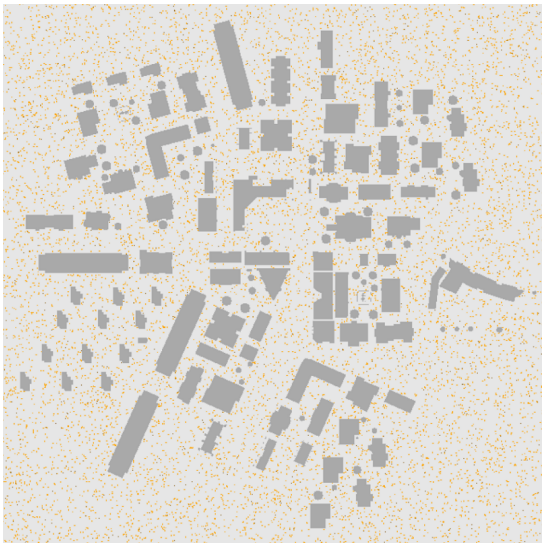


Figure 11.1: Random scenario.

– **Crossing:** A realistic scenario that periodically populates the environment. Characters spawn on the most-left side of the environment and need to cross to the most-right side; both the start and goal areas are vertical rectangles with a width of 50 meters. In this scenario, 1% of the population spawns every 10 simulation steps; this means that after 1000 steps, all characters will have been spawned. This scenario is meant for single-layered environments, as it simulated a crowd passing through an environment like a city. Unlike the random scenario, recording time in experiments starts at simulation step 1000, when all characters have been added. The period between step 0 and 1000 is a stabilization step, after-which the simulation has been stabilized and the crowd is already well dispersed and scattered. See Figure 11.2 below (next page) for a visualizations of the crossing scenario at different simulation steps.

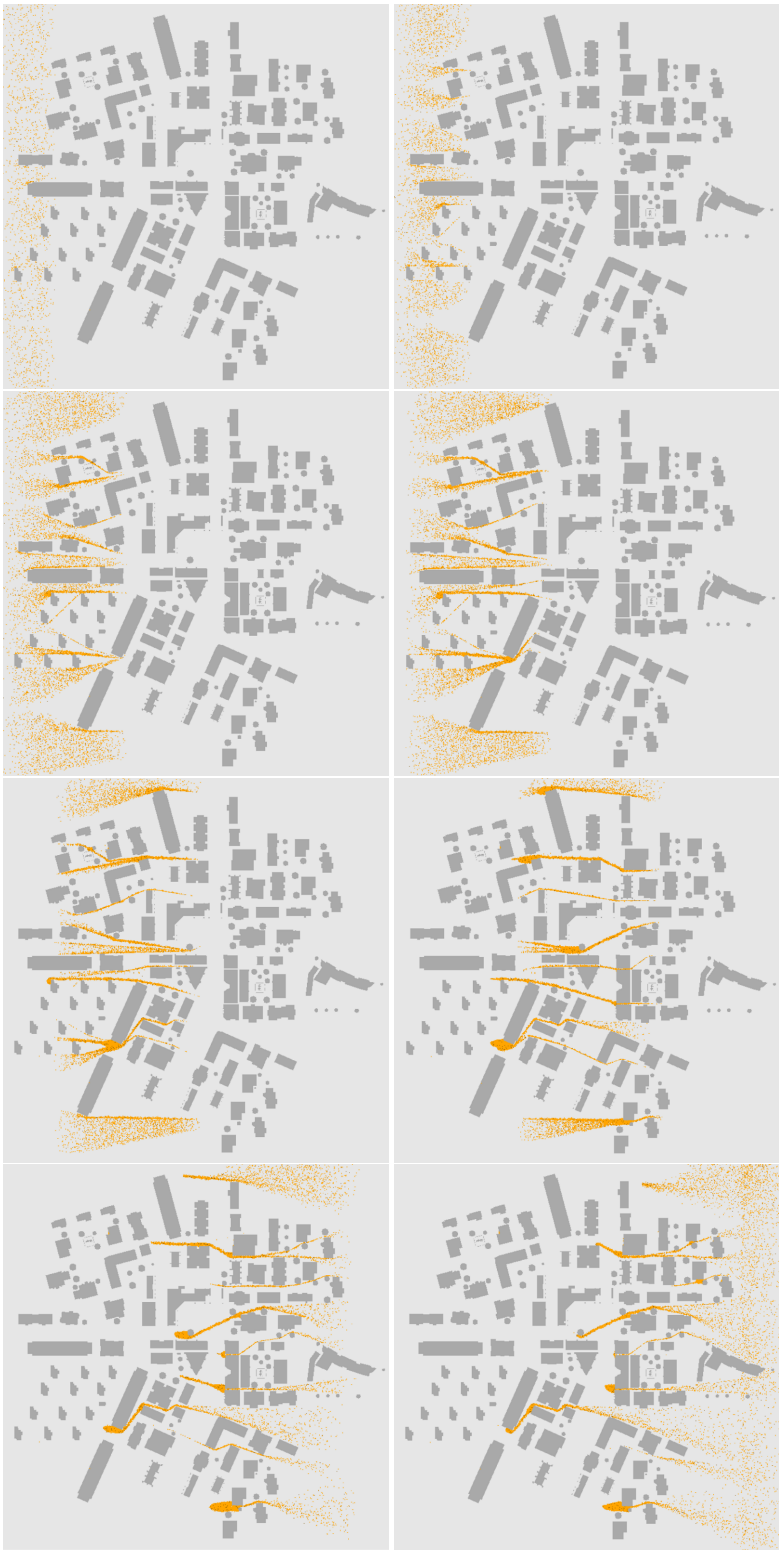


Figure 11.2: Crossing scenario time lapse.

- **Virtual environment partition structure:** This can be either a grid of cells or a k-d tree. We will use a k-d tree partition for all experiments except for the partition structure experiment.

We will first run the distributed ECM crowd simulation on a single machine, where MPI uses different threads (or processes) for distribution. As a last step, we will run it on a cloud (cluster of nodes) to observe the performance in practice; this would also allow us to see the effect of real latency between different machines. The results should be easy to interpret; all simulation times are recorded in milliseconds (ms). The results for simulation step times and substep times are the average over all nodes.

The single-machine experiments will be performed on a computer called **'Borg'**. This machine is a 64-bit Linux (Ubuntu) operated PC, with 2 sockets, 24 cores (12 per socket), and an Intel Xeon CPU E5-2690 v3 @ 2.60 GHz. This machine uses hyper-threading (2 threads per core), giving us a total of 48 threads to work with. We will use OpenMPI as the MPI implementation for this machine.

The cloud experiments will be performed on a cluster of 6 virtual private servers (VPS). Each of these machines is a 64-bit Linux (Ubuntu) operated PC, with 1 socket, 8 cores and a Westmere (Nehalem-C) E56XX CPU @ 2.40 GHz. We will use OpenMPI as the MPI implementation for this machine. The VPSs are hosted on www.transip.nl in Amsterdam.

11.2 Environments

We will experiment on three different virtual environments:

1. City: Represents a city with buildings and consists of a single layer. Size: 500 x 500 meters, with 550 polygons.

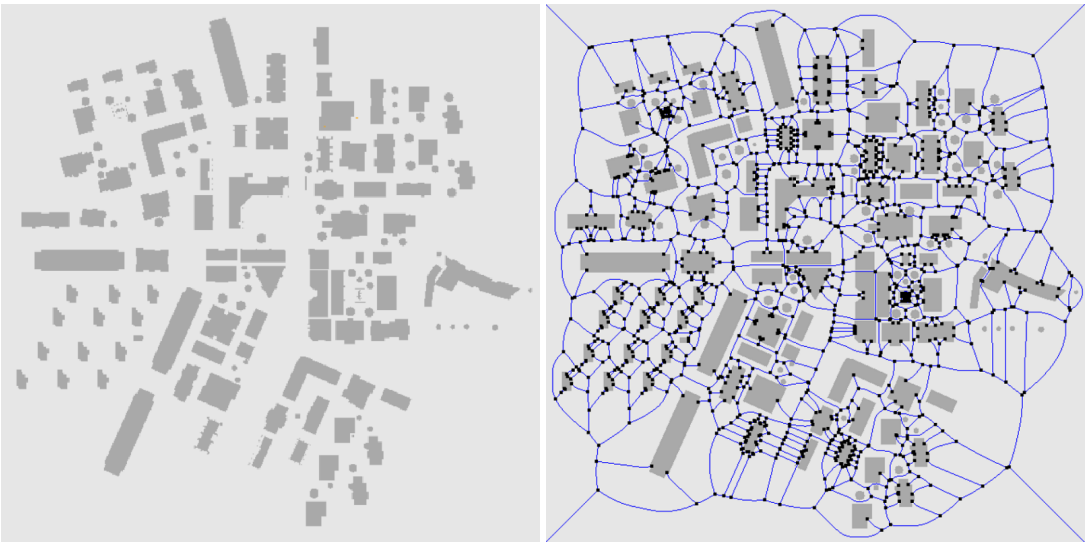


Figure 11.3: City environment (left) with generated ECM (right).

2. Stadium: Represents a stadium and consists of 18 layers. Size: 194.58 x 176.57 meters.

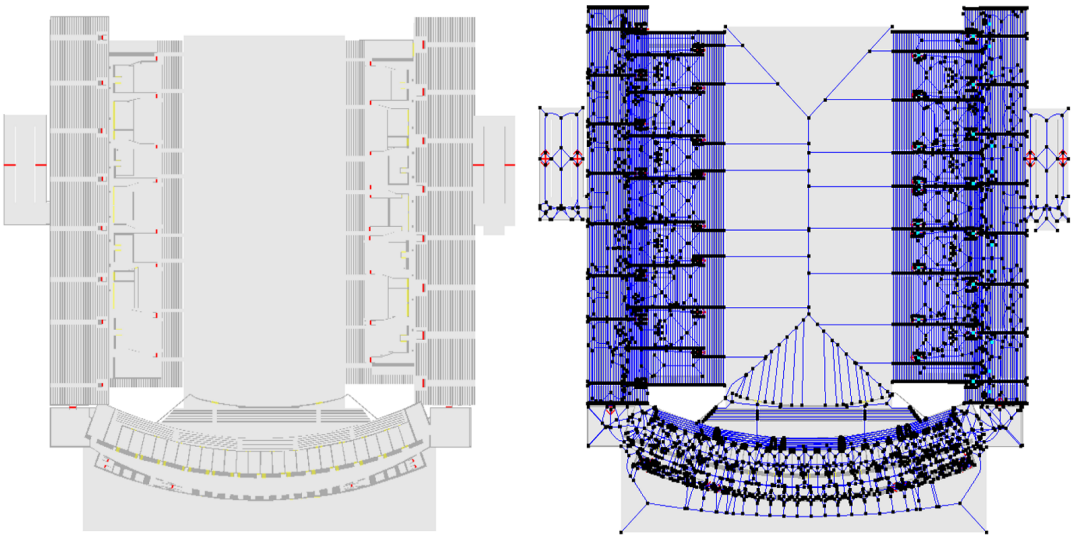


Figure 11.4: Stadium environment (left) with generated ECM (right). All layers are visualized (top-down view).

The Stadium environment file is defined by the walkable space as opposed to the obstacle polygons, therefore it's hard to give a number for the polygons. The number of obstacle vertices, however, is given below (See Table 1).

3. City_Minimal_10x10: In order to truly test how much a simulation can scale, we need a massive environment that can fit a massive number of characters. This environment is a 10x10 grid of copies of the city environment, but the city is much simpler (minimal) and a lot of buildings and other obstacles are removed. Duplicating the normal city environment was too heavy and the ECM generator could not handle its size. This environment also consists of 1 layer. Size: 5000 x 5000 meters, with 13400 polygons.

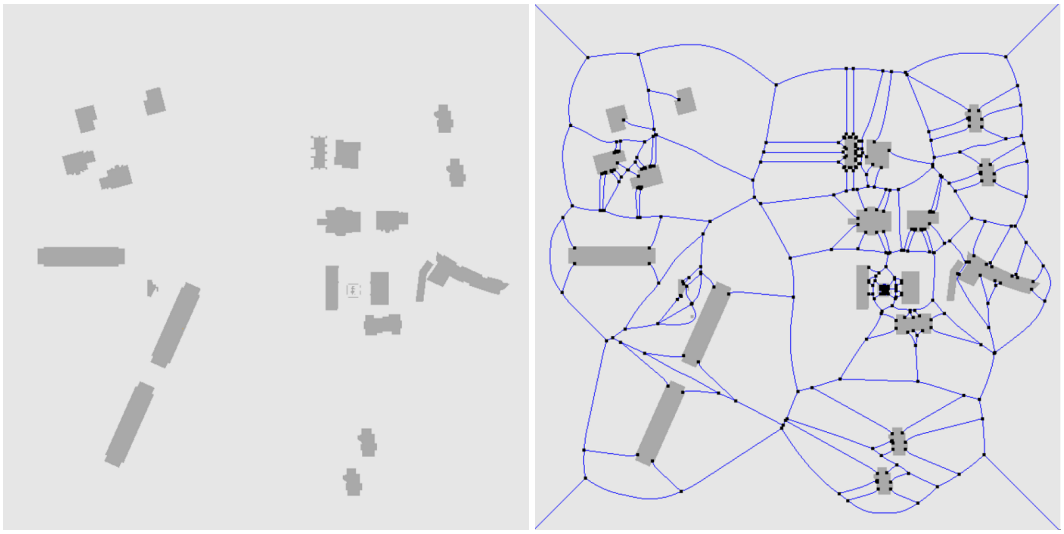


Figure 11.5: City_Minimal environment (left) with generated ECM (right).

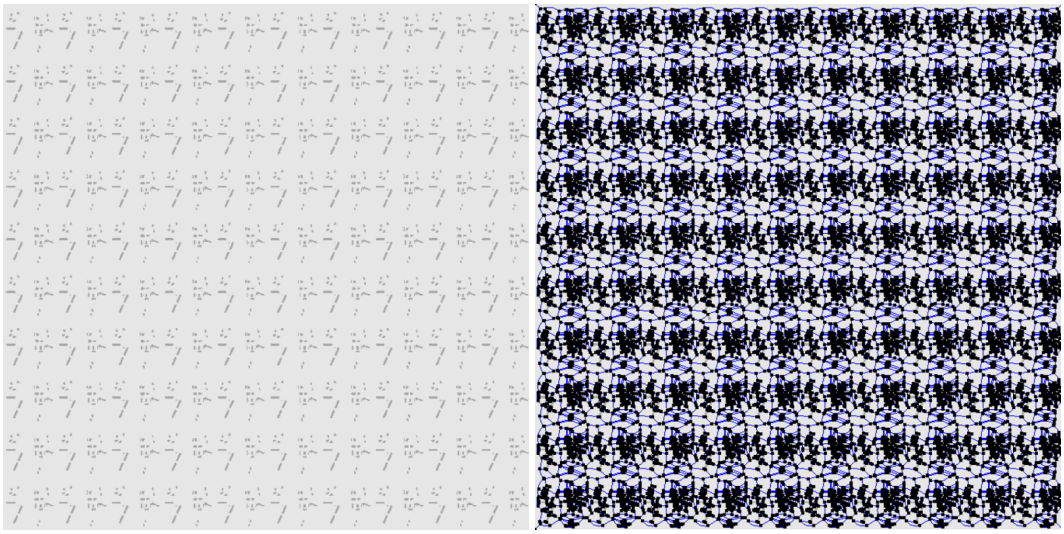


Figure 11.6: City_Minimal_10x10 environment (left) with generated ECM (right).

Environment	Size (m)	#Obstacle vertices	#Layers	#Connections
City	500x500	2102	1	0
City_Minimal_10x10	5000x5000	46700	1	0
Stadium	194.58x176.57	12915	18	82

Table 1: Details of the environments used in our experiments. The number of connections denotes the number of physical connections between different layers in a multi-layered environment. [18]

11.3 Notations

There are a few notations that we will be using for our experiments. Let **N** denote the number of nodes, and let **T** denote the number of threads that are being used for OpenMP.

We have 5 different environment-scenario pairs that are used for our experiments. For future reference, let's denote each pair as such:

Environment-Scenario	Notation
City-Random	C-R
City-Crossing	C-C
City_Minimal_10x10-Random	CM10-R
City_Minimal_10x10-Crossing	CM10-C
Stadium-Random	S-R

Table 2: Environment-scenario notations.

11.4 Experiments

We will perform 9 experiments in total. Most experiments have multiple independent variables, meaning that they follow a factorial experiment design. Some experiments might share the same variables and results, but have different goals. Experiments can also have multiple goals. For all experiments (except for exp. 6) we use a repartitioning period of 50 steps; the reasoning for why it's the optimal choice is explained in experiment 6 (repartitioning experiment).

1. **Simulation Step Performance Experiment:**

    - Goal:
        - Studying the effect of different parameters (listed in the independent variables) on the average simulation step time.
        - Discovering which experimental condition is optimal for performance.
        - Finding out which environment and scenario our system works best for.
    - Hypotheses:
        - Increasing the number of characters will increase simulation time.
        - Increasing the number of nodes and/or the number of threads for OpenMP will decrease simulation time (as long as the total does not exceed the maximum threads out machine has).
        - The optimal experimental condition will be the city_minimal_10x10 environment, with the random scenario, with any combination of number of nodes and threads for OpenMP that results in the maximum total threads of 48 (or close to 48).
        - Our system will work best for single-layered environments and large environments. Multilayered environments will result in slower and more inefficient results. Our system should work well with both scenarios.

- Independent variables:
  - The number of nodes and number threads to use with OpenMP. We have 18 total pairs of nodes and OpenMP threads:
    (T=1,N=2), (T=1,N=4), (T=1,N=8), (T=1,N=16), (T=1,N=24), (T=1,N=32), (T=1,N=46), (T=2,N=2), (T=2,N=4), (T=2,N=8), (T=2,N=16), (T=2,N=21), (T=6,N=2), (T=6,N=4), (T=6,N=6), (T=11,N=2), (T=11,N=4), (T=23,N=2).
    Note that the numbers are chosen such that $T.N < 48$. Remember that the manager takes the slot of 1 thread, so we only have 47 left for nodes, which is why we have N=46 instead of N=48, T=11 instead of T=12 and T=23 instead of T=24. Also some number of nodes don't partition well (if it's a prime number or if its prime factorization is weak) and so an appropriate number of nodes was chosen accordingly.
  - The number of characters: 1K and 10K. In addition, 100K only for the condition(s) that proved optimal for 10K.
  - The virtual environment: city, city_minimal_10x10, stadium.
  - The scenario: random and crossing.

  The stadium environment is only tested with the random scenario, as the crossing scenario is meant for single layered environments. Notice that these add up to a total of 185 distinct experimental conditions ((18x2x5) + 5).
- Controlled variables:
  - The step number: 500 steps.
  - The repartitioning period: 50 steps.
  - The partitioning structure used: k-d tree.

2. **Simulation Substep Performance Experiment:**

   - Goal:
     - Studying the effect of different parameters (listed in the independent variables) on the average node substep times.
     - Discovering the bottlenecks of our system.
     - Observing how the substep times scale with different numbers of characters.
     - Discovering how the substep times vary across different environments and scenarios.
   - Hypotheses:
     - Phase 2 will be the biggest bottleneck in our system, since it's basically all the work (the actual simulation). The biggest communication bottleneck will be ghosts and migrations.
     - All substeps should scale in a similar way; substeps that depend on the number of characters will scale the worst (such as reporting updates and ghosts).
     - Different environments and scenarios will have different ranges for substep times.
   - Independent variables:
     - The number of nodes and number threads to use with OpenMP. We have 18 total pairs of nodes and OpenMP threads:
       (T=1,N=2), (T=1,N=4), (T=1,N=8), (T=1,N=16), (T=1,N=24), (T=1,N=32), (T=1,N=46), (T=2,N=2), (T=2,N=4), (T=2,N=8), (T=2,N=16), (T=2,N=21), (T=6,N=2), (T=6,N=4), (T=6,N=6), (T=11,N=2), (T=11,N=4), (T=23,N=2).
     - The number of characters: 1K and 10K. In addition, 100K only for the condition(s) that proved optimal for 10K.
     - The virtual environment: city, city_minimal_10x10, stadium.
     - The scenario: random and crossing.
   - Controlled variables:
     - The step number: 500 steps.
     - The repartitioning period: 50 steps.
     - The partitioning structure used: k-d tree.

3. **Distributed vs. Non-Distributed Comparison Experiment:**

   - Goal: Comparing the average simulation times of the distributed version to those of the original non-distributed version to see if there is any improvement in performance.
   - Hypothesis: Results will be close and comparable; there will be a small communication overhead.
   - Independent variables:
     - The number of nodes and number threads to use with OpenMP:
       For the distributed version: (T=1,N=46), (T=2,N=21), (T=6,N=6), (T=11,N=4), (T=23,N=2).
       For the non-distributed version: T=1,2,6,12,24,48.
     - The number of characters: 1K, 10K and 100K only for the condition(s) that proved optimal for 10K.
     - The virtual environment: city, city_minimal_10x10, stadium.
     - The scenario: random and crossing.
   - Controlled variables:
     - The step number: 500 steps.
     - The repartitioning period: 50 steps.
     - The partitioning structure used: k-d tree.

4. **MPI vs. OpenMP Experiment:**

   - Goal: Comparing the scaling effect of MPI vs. OpenMP and determining which method distributes better.
   - Hypothesis: Both should scale in a similar way and be comparable. MPI has the obvious communication overhead that OpenMP does not have, since OpenMP shares memory. Scaling with only OpenMP should therefore better, but hardware limitations prohibit scaling with purely OpenMP (it's hard to have so many cores on a single machine), and so a hybrid of MPI and OpenMP allows us to scale very well on multiple machines with today's hardware.
   - Independent variables:
     - The number of nodes and number threads to use with OpenMP. We have 18 total pairs of nodes and OpenMP threads:
       (T=1,N=2), (T=1,N=4), (T=1,N=8), (T=1,N=16), (T=1,N=24), (T=1,N=32), (T=1,N=46), (T=2,N=2), (T=2,N=4), (T=2,N=8), (T=2,N=16), (T=2,N=21), (T=6,N=2), (T=6,N=4), (T=6,N=6), (T=11,N=2), (T=11,N=4), (T=23,N=2).
     - The number of characters: 1K and 10K.
   - Controlled variables:
     - The virtual environment: city_minimal_10x10.
     - The scenario: random.
     - The step number: 500 steps.
     - The repartitioning period: 50 steps.
     - The partitioning structure used: k-d tree.

5. **Partition Structure Experiment:**

    - Goal: Studying the effect that the partitioning structure has on the average simulation time: Grid vs. k-d tree.
    - Hypothesis: K-d tree partitioning will result in better results for unbalanced crowds.
    - Independent variables:
        - The partition structure: Grid, k-d tree.
        - The scenario: random and crossing.
        - The number of characters: 1K, 5K, 10K, 50K, 100K.
    - Controlled variables:
        - The number of nodes and number threads to use with OpenMP: (T=1,N=46).
        - The virtual environment: city_minimal_10x10.
        - The step number: 500 steps.
        - The repartitioning period: 50 steps.

6. **Repartitioning Experiment:**

   - Goal:
     - Studying the effect of periodic partitioning on performance (average simulation time).
     - Determining the optimal partitioning period.
   - Hypothesis: The smaller the repartitioning period, the better the performance, since nodes will be more balanced in time. Having the repartitioning period be too small will be redundant and costly.
   - Independent variables:
     - The repartitioning period: 10, 25, 50. 100, 250, 500 steps.
     - The number of characters: 1K, 5K, 10K.
     - The scenario: random and crossing.
   - Controlled variables:
     - The step number: 2000 steps.
     - The partitioning structure used: k-d tree.
     - The number of nodes and number threads to use with OpenMP: (T=1,N=46).
     - The virtual environment: city_minimal_10x10.

7. **Density Experiment:**

    - Goal:
        - Studying the performance of our distributed simulation with density sharing enabled.
        - Discovering how much the density sharing substep occupies from the total time and how it scales.
    - Hypothesis:
        - Performance will drastically decrease with density sharing enabled.
        - Sharing densities will be the biggest bottleneck of our system and will scale very badly.
    - Independent variables:
        - The number of characters: 1K, 5K, 10K.
    - Controlled variables:
        - The virtual environment: city_minimal_10x10.
        - The scenario: random.
        - The number of nodes and number threads to use with OpenMP: (T=1,N=46)
        - The step number: 500 steps.
        - The repartitioning period: 50 steps.
        - The partitioning structure used: k-d tree.

8. **Group Experiment:**

   - Goal:
     - Studying the performance of our distributed simulation on group crowds (groups of 1 to 4 members).
     - Comparing performance of groups in the distributed version to those in the non-distributed version.
     - Comparing performance of groups vs. single agents in the distributed version.
   - Hypothesis:
     - Performance of group crowds in the distributed and non-distributed simulations should be comparable.
     - Group crowds should perform relatively the same as single-agent crowds.
   - Independent variables:
     - The number of characters: 1K, 5K, 10K.
   - Controlled variables:
     - The virtual environment: city_minimal_10x10.
     - The scenario: random.
     - The number of nodes and number threads to use with OpenMP: (T=1,N=46)
     - The step number: 500 steps.
     - The repartitioning period: 50 steps.
     - The partitioning structure used: k-d tree.

   Groups are created in a similar way as single agents in a random scenario, but the members of the same group spawn close to each other, within a certain group radius (on the same layer). The group has its own goal (randomly assigned, can be on a different layer), which is followed by its individual members. The group size can range from 1 to 4 members.

9. **Cluster Experiment:**

    - Goal:
        - Studying the scalability of our distributed simulation on a cluster of servers.
        - Comparing results of those obtained on a single machine vs. those obtained on a cluster of servers to determine if latency has a significant impact on performance.
    - Hypothesis:
        - Our system should scale well on a cluster of nodes.
        - Latency should affect and result in higher communication times.
    - Independent variables:
        - The number of characters: 1K and 10K.
        - The virtual environment: city, city_minimal_10x10, stadium.
        - The scenario: random and crossing.
    - Controlled variables:
        - The number of nodes and number threads to use with OpenMP: (T=8,N=4)
        - The step number: 500 steps.
        - The repartitioning period: 50 steps.
        - The partitioning structure used: k-d tree.

11.5 Results

11.5.1 Experiment 1: Simulation Step Performance

For this experiment, we ran distributed simulations on different environment-scenario combinations, for 1K, 10K and 100K characters and for different combinations of (T,N). All running times are in milliseconds (ms). Simulations are run 5 times for each conditional setup; the average across all nodes is calculated for a single run, and the average of those 5 values at different runs is again averaged and denoted in the tables as AVG. The standard deviation (SD) is illustrated in the tables as well.

The best performance was found in CM10-R, peaking at **9.68** ms for 1K characters, **83.09** ms (real-time) for 10K characters, and **900.03** ms for 100K characters (See Table 3). The optimal condition for 1K, 10K and 100K characters is (T,N) = (1,46). We can notice from Table 3 that our standard deviations are quite small. This is a good sign and tells us that our system is consistent.

(T,N)	Number of Characters					
	1K		10K		100K	
	AVG	SD	AVG	SD	AVG	SD
(1,2)	73.86	0.98	743.85	4.81		
(1,4)	39.49	0.63	399.00	2.39		
(1,8)	21.83	0.28	215.10	0.58		
(1,16)	12.24	0.16	117.11	0.33		
(1,24)	13.56	0.24	121.17	2.36		
(1,32)	11.71	0.16	102.32	0.67		
(1,46)	**9.68**	0.21	**83.09**	0.97	**900.03**	4.09
(2,2)	45.37	0.71	458.90	5.56		
(2,4)	24.85	0.30	252.51	2.29		
(2,8)	13.45	0.20	147.95	8.05		
(2,16)	11.48	0.17	99.91	0.74		
(2,21)	11.46	0.14	86.71	0.34		
(6,2)	18.58	0.22	216.76	28.98		
(6,4)	11.54	0.14	133.84	6.33		
(6,6)	12.66	0.17	94.53	0.98		
(11,2)	11.45	0.072	136.75	26.16		
(11,4)	15.12	0.37	87.94	0.78		
(23,2)	46.10	0.25	87.37	4.08		

Table 3: Average step time (ms) results for CM10-R.

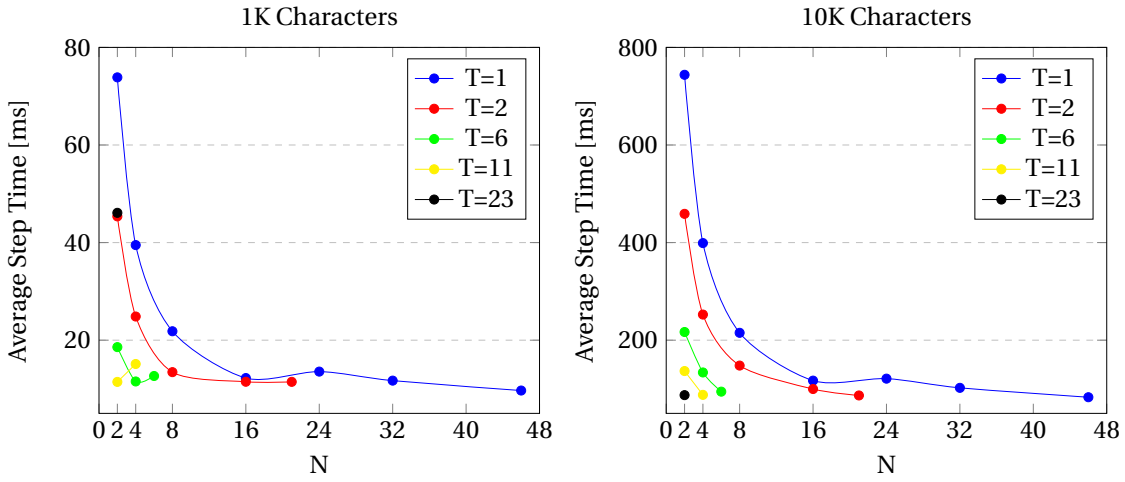


Figure 11. 7: Average step times for 1K (left) and 10K (right) characters in CM10-R.

For the other environment-scenario combinations, similar behavior was observed. Only the CM10 environment was able to achieve real-time simulations for 10K characters. This is probably due to the fact that characters are more spread out in this environment, and areas are less congested. This also decreases the amount of ghost characters to transfer along borders. For CM10-R, all 3 optimal results were achieved with the same (T,N) condition, but that's not always the case for other environment-scenario combinations (See Appendix A for more results). Obviously, as we increase the number of characters, the simulation time increases. As we increase the number of nodes, we would expect the simulation time to decrease as well, but that's not always the case. For example in the cases of T=6 and T=11, for 1K characters, increasing the number of nodes slightly slowed the simulation; this could be due to more syncing and communication time between the nodes and the manager (which is actually the case, shown in the next experiment).

The two plots in Figure 11.7 summarize the results for CM10-R. Unlike for 1K characters, average step times strictly decrease for 10K characters as N increases, except in the case of jumping from 16 to 24 nodes with T=1. For simulations with 1K characters and 2 nodes, increasing T did not always help, as can be seen at the jump from T=11 to T=23; the average step time became significantly slower when more threads were added. This shows that, for 1K characters, we shouldn't go overboard with parallelization on a single node (but that's not always the case). Notice that doubling the number of nodes has a much greater effect when the number of nodes is very small. As the number of nodes increases, doubling or increasing the number of nodes adds a less significant speed-up. The other four environment-scenario conditions produce similar results and behavior (See Appendix A).

Env.-Scenario	Number of Characters								
	1K			10K			100K		
	(T,N)	AVG	SD	(T,N)	AVG	SD	(T,N)	AVG	SD
C-R	(6,4)	15.26	0.26	(11,4)	126.81	0.72	(11,4)	1379.80	6.71
C-C	(6,4)	16.80	0.69	(11,4)	130.55	2.15	(11,4)	1422.86	5.34
CM10-R	(1,46)	9.68	0.21	(1,46)	**83.09**	0.97	(1,46)	**900.03**	4.09
CM10-C	(1,46)	**9.45**	0.28	(1,46)	91.77	0.54	(1,46)	1018.69	2.75
S-R	(11,4)	51.32	2.30	(23,2)	412.98	13.24	(11,4)	4102.35	68.10

Table 4: Best average step time (ms) results for all environments-scenarios.

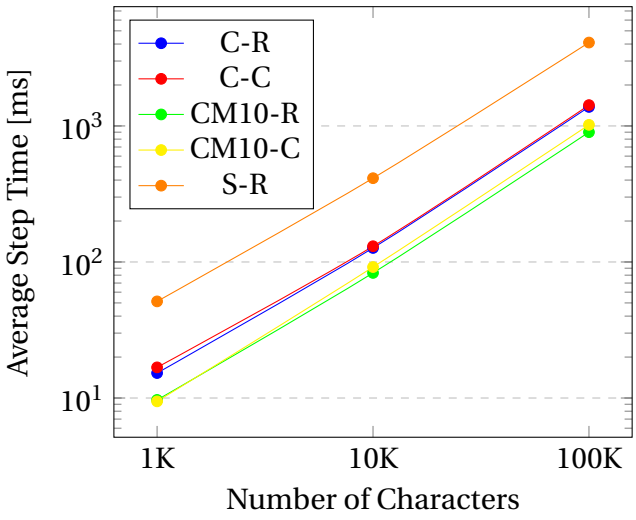


Figure 11. 8: Best average step times for all environment-scenario combinations.

Table 4 summarizes the best results for all 5 environment-scenario combinations. We can conclude that the Stadium environment is by far the worst in terms of performance. This is because our architecture is not optimized for multi-layered environments. Furthermore, the Stadium environment is more complex and has many narrow paths. The random and crossing scenarios gave similar results (for both City and City_Minimal_10x10), with the random scenario being slightly faster. This is because characters are more spread out in the random scenario, while in the crossing scenario characters tend to follow a lot of the same paths and bump into one another; some paths become congested. CM10-R provided the best results. This makes sense because it's the largest environment (it can fit a lot of characters) and characters are spread out (random scenario). In addition, all 5 curves scale linearly.

11.5.2 Experiment 2: Simulation Substep Performance

In this experiment we will look at the node substeps of our distributed simulation. We have 9 main node substeps: Actions (**A**), Phase 1 (**P1**), Synchronization (**S**), Densities (**D**), Ghosts (**G**), Phase 2 (**P2**), Remove Ghosts (**RG**), Migrations (**M**), Reporting (**R**). Naturally, the sum of these substep times will not equal the 'Step Total', which is measured as the time from one step to the next (on the manager's side). 'Step Total' includes the processing time that the manager does, such as updating character information. If we subtract the sum of node substep times from the total step time, we get the time for which nodes are idol and waiting for the manager's command to start the next step.

The results for all 5 environment-scenario pairs are summarized in Appendix A in separate tables for 1K, 10K and 100K characters; refer to them for more insight about how separate substep times change with different (T,N) in different environments-scenarios (See Appendix A2). In all the performed experiments, there were no times where densities needed to be shared; no character ever needed to replan. Therefore, the average density substep times in all tables is almost 0. Actions are very cheap as well, on average, since all characters are added before we start recording.

The obvious bottleneck of our system is Phase 2, which is the actual simulation of characters. This is to be expected, and little can be done about this, unless we change the original framework. The second biggest bottleneck, which is a communication substep, is sending/receiving migrations. The reason this substep can take so much time is because it includes a significant amount of waiting time; nodes have to wait to receive migrations from neighbor nodes since sending is done asynchronously but receiving is done synchronously. In early development, we had an extra synchronization step before migrations, which resulted in migrations taking exactly as long as ghosts do. Therefore, the recordings for the migrations substep is actually migrations + synchronization. The rest of the substeps are pretty cheap and perform well, such as phase 1, synchronization, sending/receiving ghosts and removing ghosts. Reporting updates could take a significant time for 100K characters, since it strictly depends on the number of characters. In general, communication substeps don't take up much time.

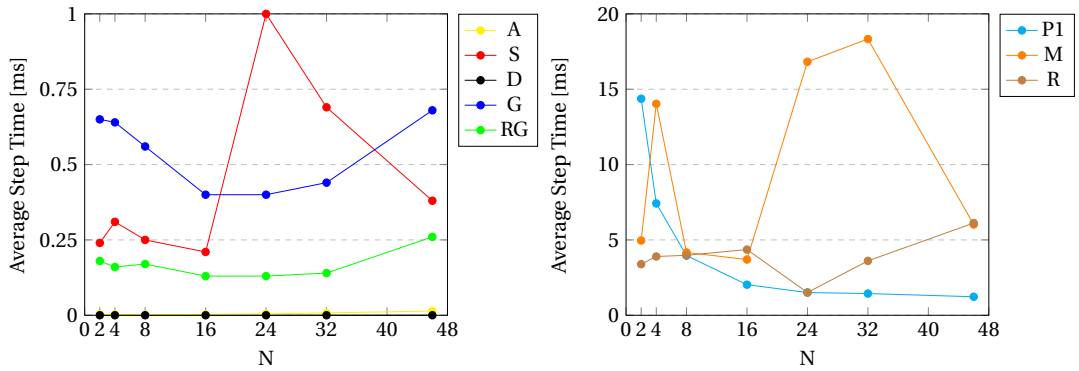


Figure 11. 9: Class 1 (left) and class 2 (right) substep average times for CM10-R, 10K Characters, T=1.

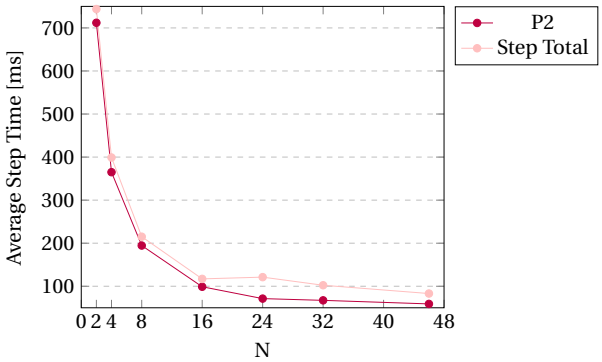


Figure 11. 10: Class 3 substep and 'Step Total' average times for CM10-R, 10K Characters, T=1.

We have 3 classes of substeps: The first one contains all substeps that take less than 1 ms; these include actions, density sharing, exchanging ghosts and removing ghosts. Naturally, actions and density sharing times are negligible and almost 0 ms, since no densities are being shared and very few actions are being handled. The synchronization substep fluctuates almost unpredictably, since this depends on the waiting time between nodes until all are ready (which varies quite easily). Generally, we have consistent synchronization time below 16 nodes. Exchanging ghosts has an almost inverted bell curve: When the number of nodes is low, we have a lot of ghosts being exchanged across the borders of neighbor nodes, since each nodes owns a large territory of the environment, and thus many ghosts (remember this is the random scenario). As we increase the number of nodes, each node's territory becomes smaller in size, resulting in less ghosts to transfer. As the number of nodes increases a lot, exchanging ghosts becomes slower again, since a lot of nodes have more neighbors now, and the k-d tree for partitioning is more complex. Removing ghosts has a similar curve and strictly depends on the number of ghosts each node has.

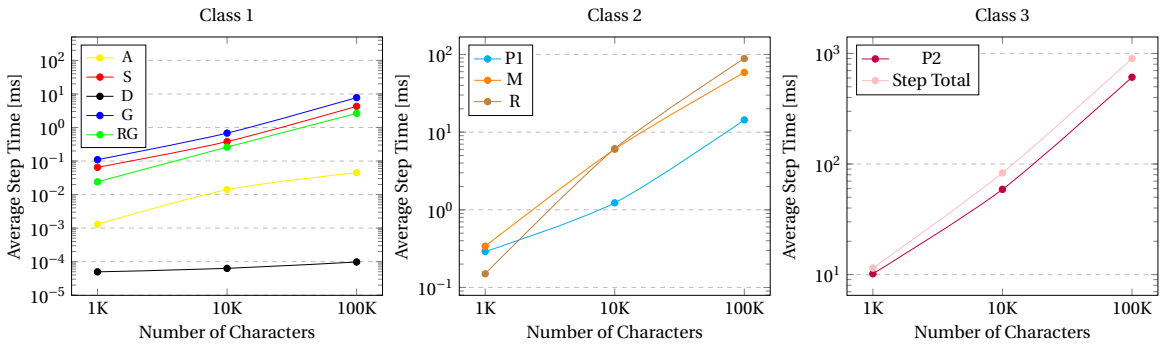


Figure 11. 11: Average substep times (for all classes) for CM10-R at optimal (T,N).

The second class of substeps is the one containing substeps that take between 1 and 20 ms. Comparing that to the average total step time (80-700 ms), these substep times are still insignificant. Phase 1 acts as expected, decreasing as the number of nodes increases. Migrations act unpredictably, since the migration substep includes waiting/synchronization time (as discussed previously). Reporting updates tends to generally increase with the number of nodes, with an exception at 24 nodes. The third and final class is for largely significant substeps, the only one being phase 2. Comparing it to the 'Step Total' curve (Figure 11.10), we can see that they are very similar (in time and shape). Since we care about scaling our simulation to accommodate massive crowds, we can observe how these substeps scale from 1K to 10K to 100K characters. For each number of characters, we only consider the optimum (T,N) condition.

We can see in Figure 11.11 that all substeps tend to scale in a similar manner (linear). Ghost, migrations, reporting and phase 2 increase at a greater slope, which makes sense since they all depend on the number of characters. Reporting updates scales the worst. We can visualize the portion each substep takes from the total in Figure 11.12, which is for CM10-R at optimal conditions (T=1,N=46) with 10K characters.

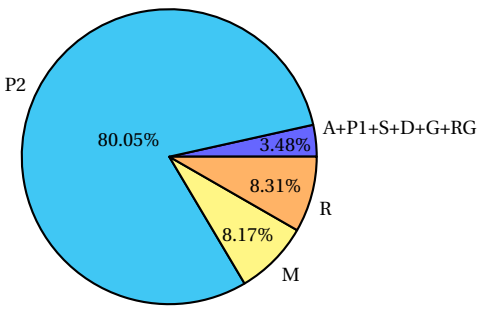


Figure 11. 12: Occupation of substep times for CM10-R, 10K characters.

Considering the optimal T and N for each environment-scenario pair, we can compare the average substep times across all 5 environment-scenario conditions:

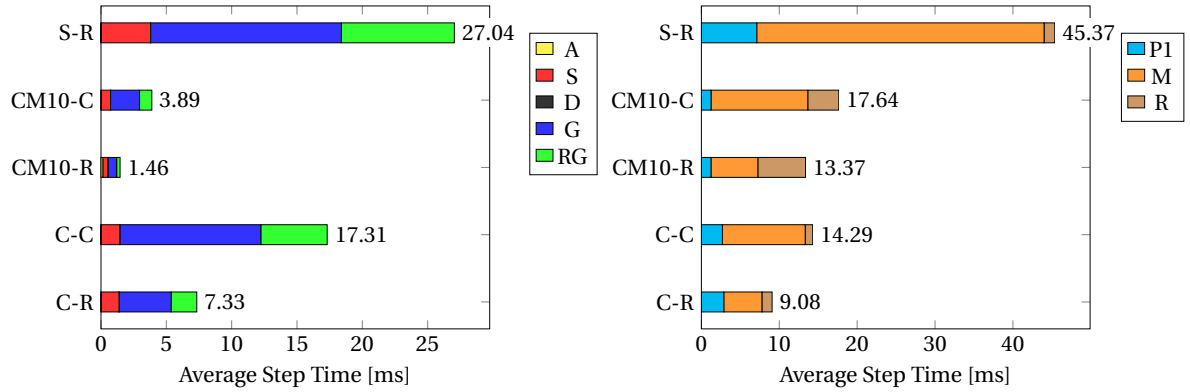


Figure 11. 13: Class 1 (left) and class 2 (right) average substep times for 10K characters.

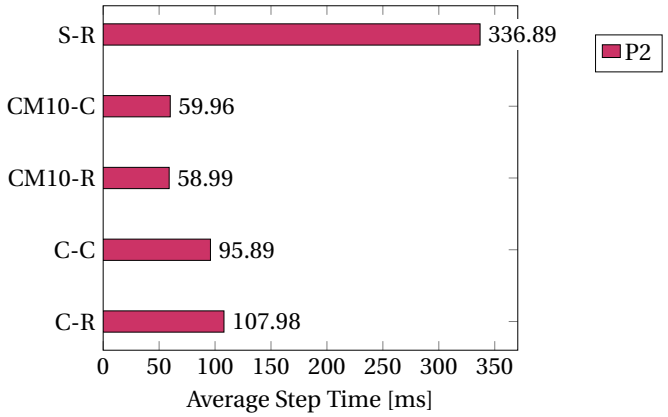


Figure 11. 14: Class 3 average substep times for 10K characters.

The number on the right of each stacked bar represents the sum of all substeps in that bar. We can see that migrations take longer in the stadium environment and crossing scenarios. The City environment has much lower reporting times than the CM10 environment. Exchanging and removing ghosts takes longer in the crossing scenarios, which makes sense since the crossing scenario is very congested, and a lot of ghosts are being exchanged.

11.5.3 Experiment 3: Distributed vs. Non-Distributed Comparison

In this experiment, we compare the distributed version to the original non-distributed version. In Table 5, the top half represents results for the original non-distributed version, and the bottom half represents the results for the distributed version.

T or (T,N)	Number of Characters					
	1K		10K		100K	
	AVG	SD	AVG	SD	AVG	SD
1	141.66	0.80	1475.75	5.24		
2	96.57	3.00	971.56	4.81		
6	39.52	0.67	382.93	5.71		
12	21.96	0.56	217.96	2.88		
24	13.01	0.28	121.00	0.79		
48	19.53	3.77	95.18	0.74	1004.90	5.34
(1,46)	9.68	0.21	83.09	0.97	900.03	4.09
(2,21)	11.46	0.14	86.71	0.34		
(6,6)	12.66	0.17	94.53	0.98		
(11,4)	15.12	0.37	87.94	0.78		
(23,2)	46.10	0.25	87.37	4.08		

Table 5: Distributed vs. non-distributed average step time (ms) comparison for CM10-R.

We can see that for the optimal conditions, the non-distributed and distributed versions are pretty comparable. Tables displaying the results of the remaining four environment-scenario combinations can been seen in Appendix A3.

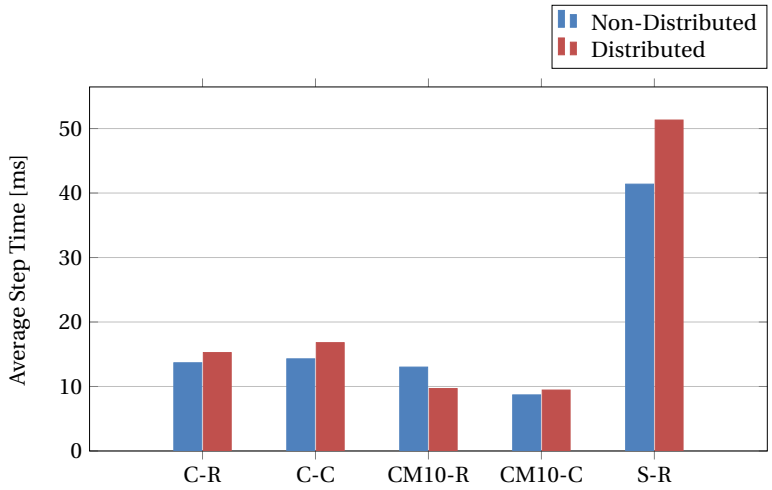


Figure 11. 15: Distributed and non-distributed simulation comparisons for 1K characters.

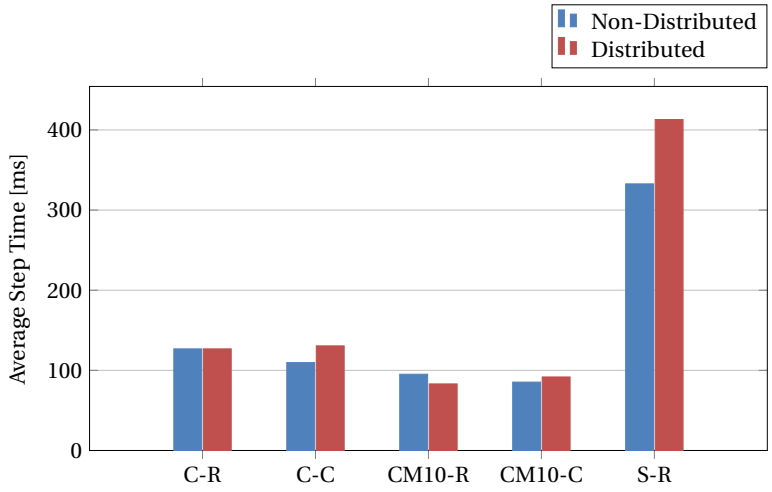


Figure 11. 16: Distributed and non-distributed simulation comparisons for 10K characters.

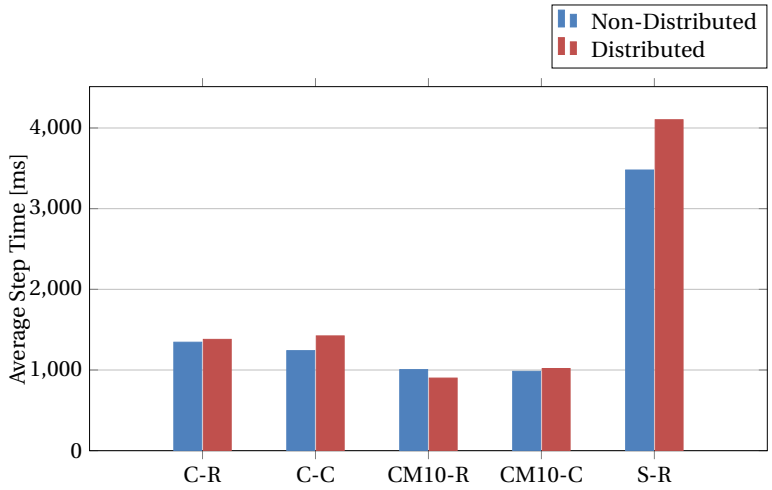


Figure 11. 17: Distributed and non-distributed simulation comparisons for 100K characters.

We can see from Figures 11.15, 11.16 and 11.17 that simulation results are very close and comparable for both versions, which means that there is very little overhead (the overhead is in the communication/synchronization substeps). Both versions also scale in a similar way. Typically, the non-distributed version performs better, except for CM10-R. For CM10-R, the optimum (T,N) condition was (1,46), meaning that this is a distribution using purely MPI. Meanwhile, for the non-distributed version, it's a purely OpenMP distribution. The distributed version performed better in this case, making MPI a better choice for distributing. MPI vs. OpenMP results and analysis are shown in the next experiment.

11.5.4 Experiment 4: MPI vs. OpenMP

In this experiment, we study the effect of MPI vs. OpenMP, and how our system scales with each (and combinations of both). We've already seen in experiment 1 how the average step time varies with the number of nodes for different values of T (threads for OpenMP); for all 5 environment-scenario combinations. We can also observe how the average step time changes as a function of T, with different values for N:

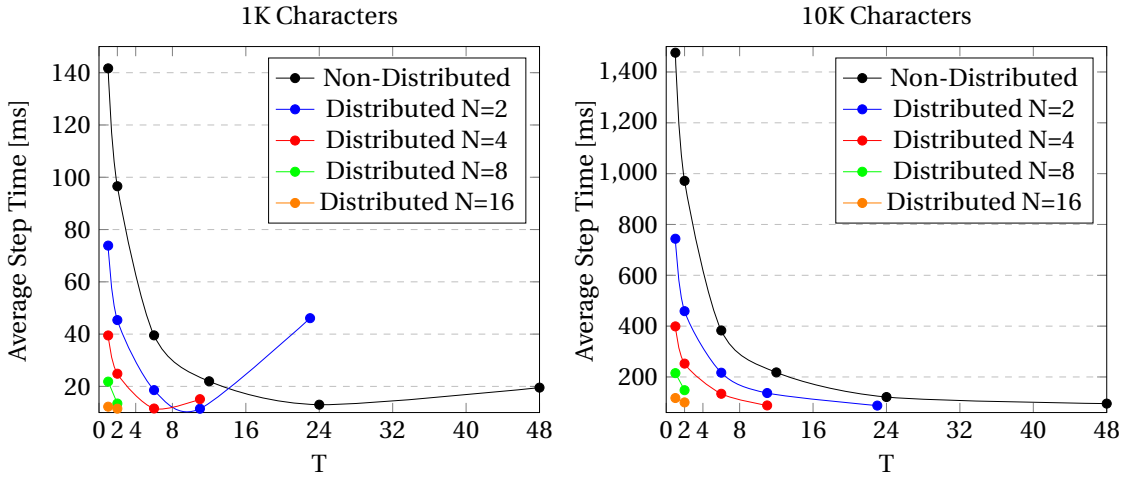


Figure 11. 18: Average step times vs. T for 1K (left) and 10K (right) characters in CM10-R.

There are some inconsistencies with 1K characters, where average step time increases again at the end of the curve, for N=2 and N=4. For 10K characters, all curves strictly decrease. There is no real need to plot these graphs for all other environments ans scenarios, since they all display a similar pattern.

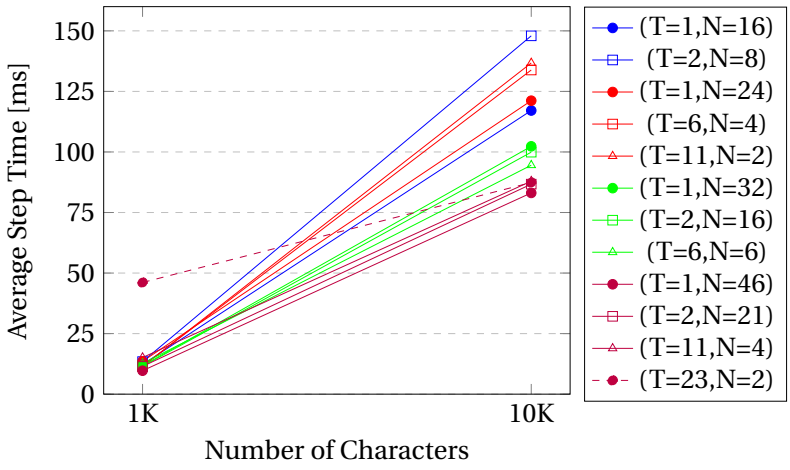


Figure 11. 19: Scaling of different (T,N) combinations in CM10-R.

Figure 11.19 illustrates how simulation times scale with different combinations of (T,N). There are some notable observations that we can distract: First, we have 4 classes of colors; each color class represents combinations of (T,N) that have an equal or comparable product of T.N. All lines within the same color class scale very similarly, with a few exceptions. For the blue class, both lines use the same number of total threads (on a single machine), using both MPI and OpenMP, yet (T=1,N=16) scales much better than (T=2,N=8). For the red class, we also see a similar effect with (T=1,N=24) scaling a bit better than the other lines in its class. From these two examples, it looks like scaling only with MPI results in better performance. This is not always consistent, however, since (T=6,N=6) gave the best result in the green class. For the purple class, (T=23,N=2) starts very poorly for 1K characters, but scales to the same point for 10K characters as the other lines in the same class.

For CM10-R, the best result was obtained when only using MPI, setting T=1 for OpenMP. This even gave better performance than the non-distributed version which uses 48 threads for OpenMP. Distributing via MPI introduces obvious communication overhead, but with OpenMP threads have to wait for all to finish before proceeding to the next substep. Both scale in a comparable way, but sometimes it's uncertain which is best. Performance also greatly depends on an efficient partition of the environment; that's how we can make the most out of MPI. Ideally, if we are running our simulations on a cloud, we would use a hybrid of both MPI and OpenMP (See Experiment 9).

11.5.5 Experiment 5: Partition Structure

In this experiment, we study the effect of the partition structure used on the average simulation time. The two partition structures we have are a grid and a k-d tree. This experiment is only performed on the CM10 environment, with both the random and crossing scenarios.

Partition	Scenario	Number of Characters									
		1K		5K		10K		50K		100K	
		AVG	SD	AVG	SD	AVG	SD	AVG	SD	AVG	SD
Grid	R	11.26	0.35	42.61	0.71	83.58	1.76	433.55	3.52	897.93	7.77
Grid	C	114.30	5.60	613.80	12.96	1270.54	31.87	6798.68	59.10	14194.87	123.54
K-d Tree	R	9.68	0.21	41.57	0.37	83.09	0.97	432.63	1.93	900.03	4.09
K-d Tree	C	9.45	0.28	43.10	0.35	91.77	0.54	493.73	1.47	1018.69	2.75

Table 6: Average step times (ms) for CM10-R and CM10-C with grid/k-d tree as the partition structure.

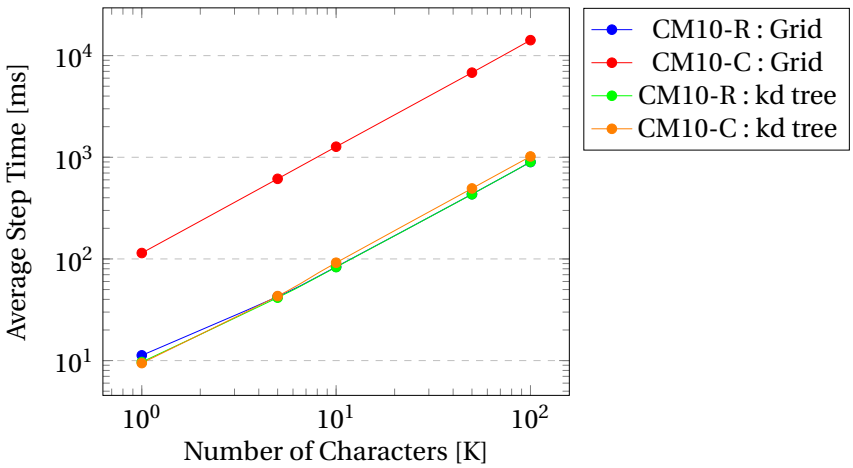


Figure 11. 20: Environment partition structure comparison: Grid vs. k-d Tree.

The logarithmic plot above (Figure 11.20) displays similar curves for all 4 cases. When using a k-d tree, the random and crossing scenarios are very similar in performance, since the crossing scenario is unbalanced over the entire environment but the k-d tree partition makes the spacial ownership of the nodes balanced; each node owns almost the same amount of characters. When using a grid, we get almost the same results as those with a k-d tree, but only for the random scenario; results in the random scenario for a grid are even slightly better sometimes. There's no reason for using a k-d tree on an already balanced environment (such as with the random scenario), unless characters might scatter out in an unbalanced way over the long run. When simulating a crossing scenario with a grid, performance becomes absolutely terrible, which is expected since in the crossing scenario, all characters start from the left, leaving the right-half side of the environment empty for the majority of the simulation. Remember that we are using 46 nodes, which is a lot, so most nodes would be completely empty and would be idol the whole time. The busiest nodes would be on the left side and in the spots where there is a lot of congestion (around the middle, in tight spaces). Essentially, one or a couple of nodes would be doing all the work, while the rest stay idol. Therefore, it's safe to say that a k-d tree is usually always the better choice when partitioning a virtual environment for a distributed simulation.

11.5.6 Experiment 6: Repartitioning

In this experiment we study the effect of the repartitioning period on the average simulation time for both CM10-R and CM10-C.

Repartition Period (steps)	Number of Characters					
	1K		5K		10K	
	AVG	SD	AVG	SD	AVG	SD
10	9.65	0.11	41.67	0.08	83.44	0.30
25	9.44	0.15	42.07	0.19	84.16	0.51
50	9.58	0.22	41.55	0.31	83.23	0.47
100	9.53	0.17	41.16	0.44	83.56	0.28
250	9.78	0.13	41.34	0.30	83.90	0.54
500	9.58	0.22	41.62	0.46	83.65	0.08

Table 7: Average step times (ms) for CM10-R with different repartitioning periods.

As we can see in Table 7, changing the repartitioning period for the random scenario had little to no effect; we essentially obtain the same performance for all different periods. This is expected, since the random scenario is already well scattered and balanced.

Repartition Period (steps)	Number of Characters					
	1K		5K		10K	
	AVG	SD	AVG	SD	AVG	SD
10	11.24	0.09	53.01	0.11	107.35	0.25
25	11.55	0.16	53.90	1.52	107.61	2.16
50	11.52	0.24	53.38	1.15	107.88	2.14
100	12.10	0.08	54.26	0.33	110.41	1.16
250	12.96	0.44	60.08	0.31	122.81	1.60
500	15.82	14.97	71.47	0.57	144.90	2.84

Table 8: Average step times (ms) for CM10-C with different repartitioning periods.

Table 8 shows a declination in performance as the repartitioning period increases. This makes sense, since the crossing scenario is unbalanced and becomes more unbalanced over time after repartitioning once. For 10, 25 and 50 steps we achieve fairly similar results. After that, performance decreases in a more significant and obvious way (for 100, 250 and 500 steps). We can visualize these results better in the plots below.

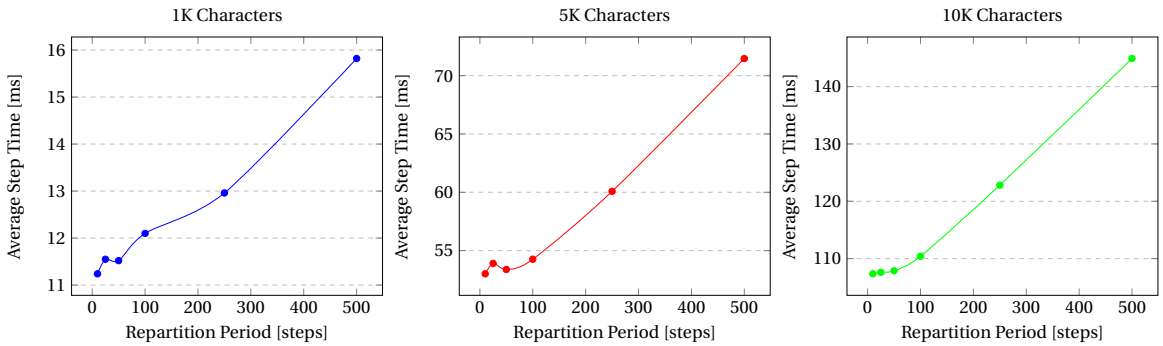


Figure 11. 21: Average step times vs. repartitioning period for CM10-C, with 1K (left), 5K (middle) and 10K (right) characters.

In all three plots, we can see the same trend. After 50 steps, the increase in step time becomes more significant. For this reason, we have chosen a repartitioning period of 50 steps as the standard for all experiments. Choosing 10 or 25 would also be fine, but it seems a bit too frequent; in addition, 50 steps is already giving the same results. Characters can't move that far in under 50 steps, which is why repartitioning more frequently than that will not change performance. After 50 steps, characters can scatter out more, and this is where our k-d tree will need to be updated by repartitioning and rebalancing our simulation nodes.

11.5.7 Experiment 7: Density

In this experiment, we enable density sharing in every simulation step. We predict that this substep will be the biggest bottleneck and will scale badly, since all nodes communicate in this step, not just neighbors. We can see the average substep times in Table 9 below, for 1K, 5K and 10K characters.

Number of Characters		A	P1	S	D	G	P2	RG	M	R	Step Total
1K	AVG	0.020	0.13	0.23	**26.75**	0.74	5.84	0.035	1.56	0.012	37.65
	SD	0.0036	0.0011	0.019	0.068	0.12	0.052	0.00087	0.047	0.00039	0.59
5K	AVG	0.018	0.62	0.32	**132.36**	4.32	29.29	0.13	4.48	1.77	180.97
	SD	0.0026	0.0035	0.026	0.26	0.17	0.19	0.0036	0.29	0.28	1.95
10K	AVG	0.014	1.23	0.41	**263.43**	7.28	58.90	0.26	7.16	4.35	354.10
	SD	0.00090	0.0030	0.030	1.09	0.32	0.20	0.0057	0.58	0.81	1.71

Table 9: Average substep and step times for CM10-R, with density sharing enabled.

We can see that the density substep is occupying most of the time per step, around 77% of the substep times (See Figure 11.22 for visualization). Phase 2, which was once the bottleneck of our system, is now the secondary bottleneck, occupying only 17%.

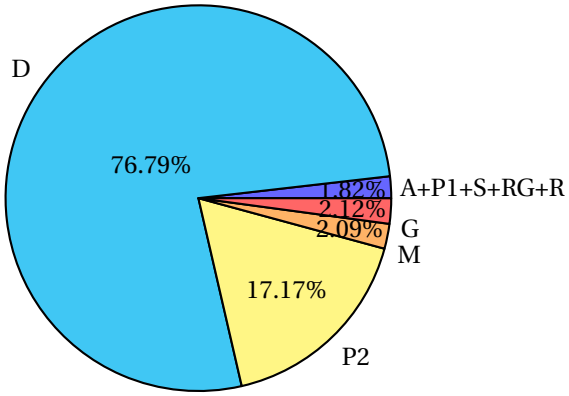


Figure 11. 22: Occupation of substep times for CM10-R, 10K characters (with density sharing).

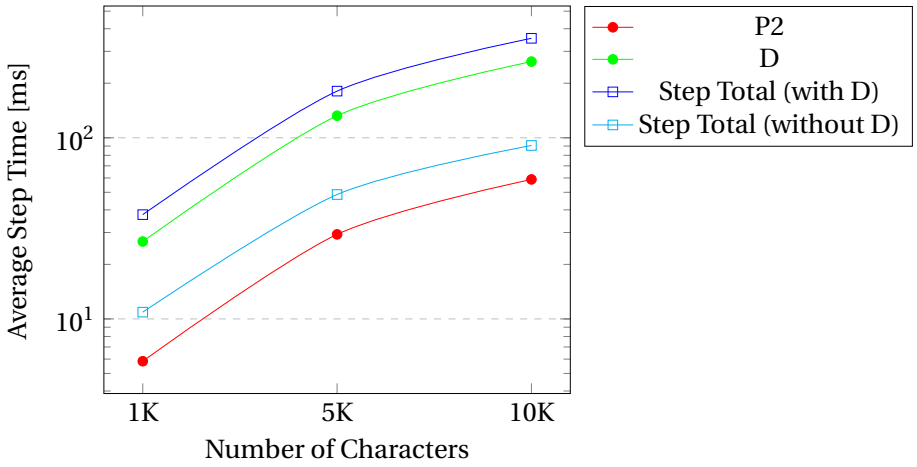


Figure 11. 23: Scaling of Density, Phase 2 and Step Total.

Figure 11.23 illustrates how density sharing scales from 1K to 5K to 10K characters. We can see a clear difference between the old 'Step Total' (that didn't include D) and the new 'Step Total' (that includes D). We have enabled density sharing at every substep, but what if we only share densities when characters need to replan their paths and have them replan periodically? In that case, we could have a simulation in real-time but that stutters/peaks each time densities are shared between nodes. As long as that doesn't happen often, it's not a big problem. In order to avoid the stutter completely and have our simulation be in real-time at all times, we would have to think of a different way to keep a copy of the updated density map on each node.

11.5.8 Experiment 8: Groups

In this experiment, we take a look at groups. Groups are added randomly the same way single characters are added in the random scenario. By specifying a number of characters, groups of random sizes (1 to 4 characters) will form, until we reach the specified number of characters. So we don't specify the number of groups, but the number of characters. Below we can see a comparison between the distributed and non-distributed simulations with group crowds.

Simulation Type	Number of Characters					
	1K		5K		10K	
	AVG	SD	AVG	SD	AVG	SD
Distributed	20.19	0.37	60.90	4.59	106.35	1.46
Non-Distributed	33.35	4.42	57.31	3.09	104.33	4.84

Table 10: Average step times (ms) for CM10-R for groups.

The results are quite similar for both types of simulations. Group performance for the distributed simulation was even better with 1K characters. We can also compare these two to the results of a distributed/non-distributed single-agent simulations. We can see from Figure 11.24 that single-agent simulations perform better than group simulations. Groups perform similarly in both distributed and non-distributed simulations (a bit worse when distributed).

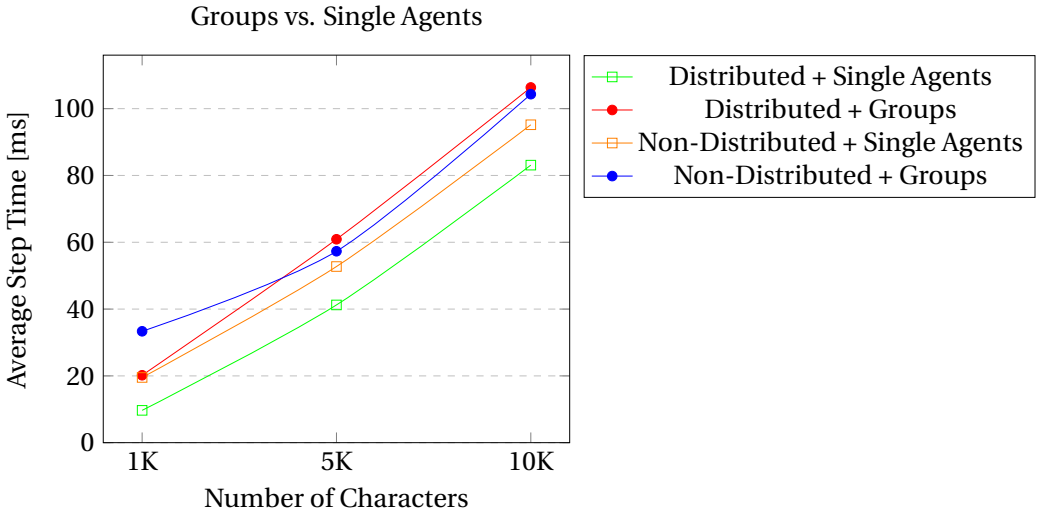


Figure 11. 24: A comparison of groups and single agents in both distributed and non-distributed simulations.

11.5.9 Experiment 9: Cluster

In this experiment, we test our distributed simulation on an actual cluster of nodes (cloud), where each node is a separate machine. The specifications of these machines are described in the setup of this section. We have 5 VPSs (virtual private servers): one runs the simulation manager while the other four run simulation nodes. For this experiment, we are using (T=8,N=4), since we have 4 nodes each with 8 cores.

The results of average substep times are summarized in Appendix A4. The substeps typically display similar trends and results as we have seen in Experiment 2. We would have expected to see some more latency in the communication substeps, but they performed well even on a cluster. This might be because the VPSs could be on the same physical machine, or the connection between then is super fast.

The average step time for all 5 environment-scenario combinations can be seen in Table 11. We can also see results for non-distributed simulations, which are performed on a single VPS (this is why these results are slow). Finally, refer to the logarithmic graph in Figure 11.25 for a visualization of the distributed simulation results.

Env.-Scenario	Simulation Type	Number of Characters					
		1K		10K		100K	
		AVG	SD	AVG	SD	AVG	SD
C-R	Distributed	15.88	1.02	161.42	3.69	1797.70	26.00
C-R	Non-Distributed	33.27	1.22	413.41	2.43	4722.78	21.13
C-C	Distributed	18.12	1.16	197.30	7.90	1550.07	11.88
C-C	Non-Distributed	33.97	0.41	391.97	2.02	3664.58	13.91
CM10-R	Distributed	11.61	0.44	103.96	2.50	1062.71	23.74
CM10-R	Non-Distributed	26.22	0.40	275.43	2.58	3049.66	42.09
CM10-C	Distributed	12.01	0.55	124.23	1.89	1402.67	9.47
CM10-C	Non-Distributed	25.48	1.13	305.17	2.78	3584.71	9.92
S-R	Distributed	78.63	3.23	471.16	14.13	3796.74	33.87
S-R	Non-Distributed	107.37	5.92	1106.27	2.27	10674.60	72.85

Table 11: Average step times (ms) on a cluster.

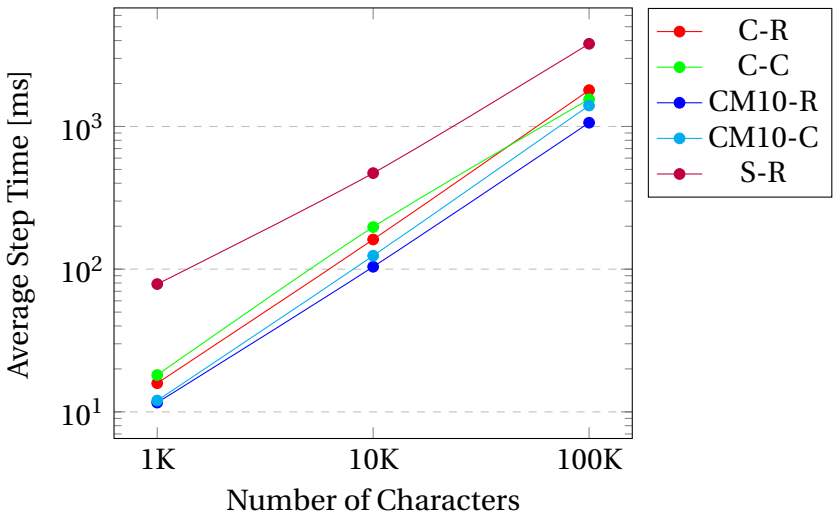


Figure 11. 25: A comparison of how the average step time in a distributed simulation on a cluster scales with different environments and scenarios.

12 Performance Scaling Predictions

Since we're not able to obtain real-time for massive crowds (of hundreds of thousands and even a million characters), it's important to predict how our system scales, using methods like curve fitting. For these predictions, we will focus on CM10-R, using mostly data obtained from the hyperthreaded 24-core machine 'Borg', but also data obtained from the cluster experiment (five 8-core VPSs).

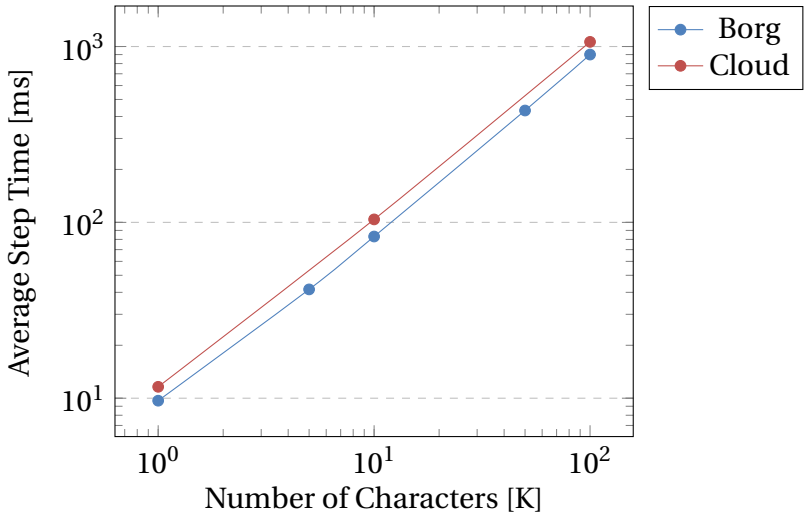


Figure 11. 26: Scaling of CM10-R.

Two curves are displayed in the graph above, one for the data on 'Borg' with (T=1,N=46), and the other for the data on the cloud (cluster). Using curve-fitting, we obtain the following linear functions:

Machine	Best Fit	R^2	Characters in real-time (100 ms)	Time for 1 Million Characters
Borg	$y = 8.99 * C - 5.14$	0.9996	11.69K	8984.86 ms
Cloud	$y = 10.63 * C - 0.61$	1	9.46K	10629.39 ms

Table 12: Curve-Fitting for CM10-R. C denotes the the number of characters in thousands (K), and y denotes the time in milliseconds (ms).

Considering CM10-R on 'Borg', we can fit a function for the average step time that depends on both the number of characters and threads for OpenMP (T). We have done this for N=2 and N=4 only, since step time vs. T curves (displayed in Experiment 4) for other values of N don't have as many recorded samples. Unlike the previous fitted functions, these are not linear, but exponential. Note that these fitted functions are rough approximations; more samples are needed to make better fits and predictions.

N	Function
2	$y = 70 * (C/1000)\,T^{-0.7}$
4	$y = 40 * (C/1000)\,T^{-0.7}$

Table 13: Curve-Fitting for CM10-R, N=2,4.

We can visualize these fits for CM10-R for 1K and 10K characters:

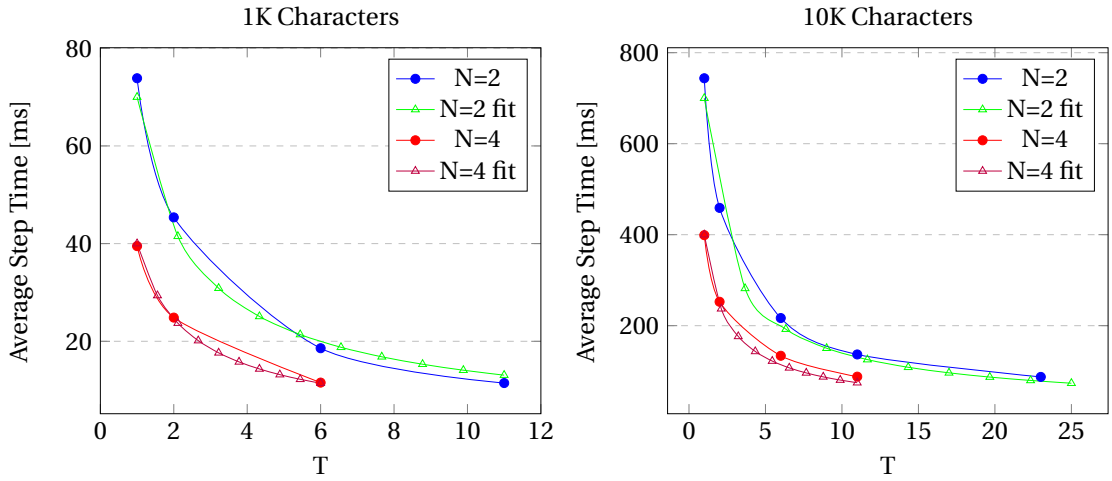


Figure 11. 27: A comparison of the original and fitted curves of the average step time as a function of T, for 1K (left) and 10K (right) characters in CM10-R.

Using these functions, we can estimate how many threads (and thus cores) are needed for real-time performance for 2 and 4 nodes:

N	Number of Characters			
	10K	100K	500K	1M
2	16	432	4310	11600
4	8	195	1938	5215

Table 14: Number of threads (cores) needed for real-time performance.

We can also try to fit a function for the average step time that depends on both the number of characters and nodes (N). We have done this for T=1 only, since it has the most number of recorded samples. These fitted functions are also exponential. Note that these fitted functions are much worse than the previous ones, since it's hard to fit a function for the number of nodes: a lot of other variables affect the average step time, mainly how efficient a k-d tree partition is given a number of nodes, such that neighbors are minimal and spacial ownership for each node is reasonable.

T	Function
1	$y = 120 * (C/1000)T^{-0.68}$

Table 15: Curve-Fitting for CM10-R, T=1.

We can visualize these fits for CM10-R for 1K and 10K characters:

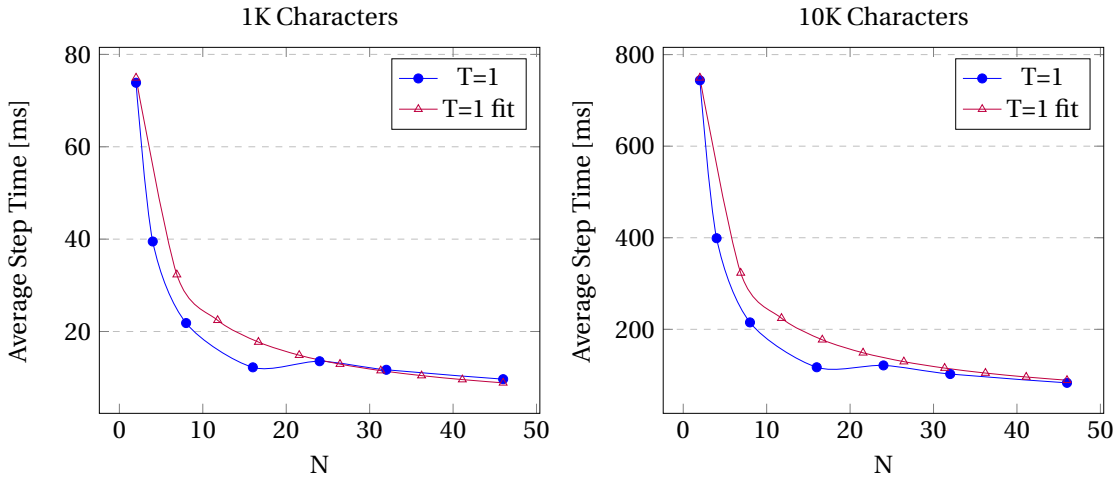


Figure 11. 28: A comparison of the original and fitted curves of the average step time as a function of N, for 1K (left) and 10K (right) characters in CM10-R.

Using these functions, we can estimate how many nodes are needed for real-time performance for T=1:

T	Number of Characters			
	10K	100K	500K	1M
1	39	1142	12176	33745

Table 16: Number of nodes needed for real-time performance

Note that predicting T for real-time is much more accurate than predicting N, since the communication times and latency are not taken into account. Imagine the time needed just for communication if we have a thousand nodes; behavior would be very unpredictable. It's also very difficult to fit a function with multiple variables (T, N and C). If we look at Table 14 again, we can roughly assume that the number of cores is almost halved as N is doubled. Keeping that assumption, we could expect around 12 cores for 64 nodes for real-time performance with 100K characters, and around 325 cores for 64 nodes for real-time performance with 1 million characters.

13 CONCLUSION

In this thesis, we have designed a system for distributed simulation that has a single manager and multiple nodes. The manager communicates with the nodes via Actions, and nodes can only communicate once with the manager at the end of a simulation step, by reporting their updates. Nodes have several communication points, such as exchanging ghosts, migrations, densities and synchronization. The manager partitions the virtual environment using a grid or k-d tree (the latter for better performance), such that each node handles one partition, i.e. an axis-aligned rectangle. In order to keep a balanced workload between nodes, the manager repartitions the environment every 50 steps so that nodes handle the same number of characters. The fact that a lot of the intercommunication is done asynchronously helps speed things up a lot.

We have performed 9 experiments to test the performance and limitations of our system. We showed that step and substep times scale well (linearly), for 5 different environment and scenario combinations. We concluded from the experiments that phase 2 was the biggest bottleneck of our system. Sharing densities is an even bigger bottleneck when enabled at each step, but there is no need to share densities unless at least 1 character needs to replan its path. The largest environment (CM10) was able to achieve the best results, which even surpassed the non-distributed simulation. With smaller environments, crowds will become very congested as we increase their size (such as 100K), and thus simulating them will be slower. We also showed that multi-layered environments perform and scale badly. Different combinations for T and N produced some similar results when the product T.N was the same. Therefore, the ideal system is a hybrid of parallel computing and shared memory multiprocessing. Under the best conditions, we were able to simulate around 12K characters in real-time using our distributed simulation.

Finally, we have added a section for predicting how our system scaled with the number of characters, nodes and threads for OpenMP. These predictions are very rough estimates, since it's difficult to accurately predict what is needed for real-time distributed simulations for large crowds. A lot of variables affect performance, such as the way the environment is partitioned and the latency between nodes when communicating. In any case, the way our average step and substep times scale looks promising, and thus it should be possible, with better hardware and more nodes, to simulate hundreds of thousands of characters in real-time.

14 DISCUSSION

Our goal in this thesis was to reach large scale-ups in simulation performance, but our distributed simulation is only as fast as its bottleneck. Each step needs to be synchronized, which makes our system not *really* distributed (but as distributed as can be). Nodes constantly need updates positions and velocities of characters in neighboring nodes, and this is why things need to be done synchronously. Even though we were able to achieve promising results, with minimum communication overhead, there are still a lot of improvements we can apply to our system. In this section we will be discussing the limitations of our system and improvements that can be done in future work.

14.1 Limitations

We have already discussed how a big limitation is having to synchronize at each step since nodes constantly need updated information, so our system has to be mostly synchronous, with a few asynchronous steps in between (such as sending ghosts and migrations). In terms of environment compatibility, our distributed simulation system works best for large single-layered environments. Multi-layered environments tend to perform badly, because of how our environment partitioning is done. Obviously the virtual environment should be large enough to hold a large crowd so that it's not congested, therefore small environments also tend to perform badly with a large number of characters.

Another limitation is that the way the k-d tree partitioning is done is based on the number of nodes we have. So if we have a prime number of nodes, the partition will be done inefficiently. This isn't a huge problem, but the number of nodes is something to consider when buying and investing in machines.

The way groups are implemented in our original framework made it difficult to distribute them; the workaround we implemented was to have the entire group on the same node (decided by the group leader). Once the leader migrates, so do the rest of the members. Since we are only using groups of small sizes (1 to 4 members), this was fine and our simulation still produced correct results. However, once the group size increases, or once one of the members lags a bit, this can become problematic. This is because our ghost layer is 10 meters, so if a character belonging to a node resides beyond that layer (outside the node) it will not receive its nearest neighbor information (since its neighbors will not be registered as ghosts). A group member could lag and distance itself from other members in the group if

there is a lot of congestion in the area, and so, small congested environments pose problems for group simulations. Increasing the ghost layer could solve these problems, but would make the simulation slower and inefficient.

A limitation in experimentation is the fact that we had limited hardware, and so we couldn't experiment with a wide range of nodes and threads for OpenMP. The product T.N had to always be below the maximum number of threads available (on a single machine). As for the cluster, we also had a limitation of 5 VPSs with 8 cores each. Buying more than that, and with more cores per machine, is very expensive. From the limited samples we had, rough predictions and estimates were made, but are not very accurate due to this limitation. Another limitation was the fact that simulating 100K characters took very long per run, and so we were only able to run 100K simulation only with the best conditions for 10K characters.

14.2 Future Work

Many things could still be improved in our system. These possible improvements will be discussed in this section.

14.2.1 Multi-layered Partitioning

The way the virtual environment is partitioned by the manager does not take into account multi-layering. Crowds that are on top of each other do not need to know each other's information since they don't affect each other. For future work, partitioning could be improved and implemented differently to take multi-layered environments into account. In addition, the geometry of the environment (such as obstacles) could also be taken into consideration, since characters on opposite sides of a wall also don't affect each other's movements. We can go even further and suggest a partitioning scheme that takes crowds into consideration: if there's a crowd of characters always looping in the same circle, it makes sense to group them in the same node, as opposed to cutting between them.

14.2.2 Scheduled Communication

Instead of communicating ghosts and migrations at every step, we could attempt to check if communication is needed in the next step. If not, we can predict the soonest time communication is needed and schedule an appointment between the two neighboring nodes, similar to what Xu et al. have suggested [14]. Other suggestions in that paper could also be applied to our system, such as synchronization skipping and dead-reckoning. This

would make our simulation less synchronized and thus more distributed. With large crowds, it's most likely that communication is needed in each step for all nodes, so scheduling appointments is likely to improve only smaller simulations.

14.2.3 Densities and Global Path Planning

As seen in experiment 7, sharing densities so that each node as an updated density map was the biggest bottleneck by far. Luckily, this substep is only done when needed (which is not that often in practice). In any case, it's still problematic and more work should be invested into this step. Perhaps path planning could be changed so that it's not global anymore (and thus densities won't be needed). This would require changing the original framework that we build our distributed simulation over. If paths are only planned ahead for a limited distance, this would eliminate the need for every node to know all densities in the environment.

14.2.4 Groups

As discussed in the limitations, groups could be reworked such that each member is in its correct node. Doing that poses many synchronization and communication issues, but maybe there is an efficient way to do it.

14.2.5 Extensive Experimentation

If given the time and hardware, more extensive experimentation could be done, with more environments, scenarios, and larger crowds. More combinations of T and N could be tested to find out which combination yields best results. Setting up a cluster with machines that are physically separated could also be interesting; in which case we can observe the effects of latency.

REFERENCES

[1] Statista. (2019). Saudi Arabia: Hajj pilgrims 2018 | Statistic. [online] Available at: https://www.statista.com/statistics/617696/saudi-arabia-total-hajj-pilgrims.

[2] Azevedo, T., Rossetti, R. J., & Barbosa, J. G. (2016). Densifying the sparse cloud SimSaaS: The need of a synergy among agent-directed simulation, SimSaaS and HLA. arXiv preprint arXiv:1601.08116.

[3] NATO STO: Final Report of NATO MSG-131 "Modelling and Simulation as a Service: New Concepts and Service Oriented Architectures". STO Technical Report STO-TR-MSG-131.

[4] Lozano, M., Morillo, P., OrduÃśa, J. M., Cavero, V., & Vigueras, G. (2009). A new system architecture for crowd simulation. Journal of Network and Computer Applications, 32(2), 474-482.

[5] Lozano, M., Morillo, P., Lewis, D., Reiners, D., & Cruz-Neira, C. (2007, July). A distributed framework for scalable large-scale crowd simulation. In International Conference on Virtual Reality (pp. 111-121). Springer, Berlin, Heidelberg.

[6] Steffen, B., Kemloh Wagoum, A. U., Chraibi, M., & Seyfried, A. (2011). Parallel real time computation of large scale pedestrian evacuations. In The Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering (p. 95).

[7] Wijerathne, M. L. L., Melgar, L. A., Hori, M., Ichimura, T., & Tanaka, S. (2013). HPC enhanced large urban area evacuation simulations with vision based autonomously navigating multi agents. Procedia Computer Science, 18, 1515-1524.

[8] Vermeulen, J. L., Hillebrand, A., & Geraerts, R. (2017). A comparative study of kâĂŘnearest neighbour techniques in crowd simulation. Computer Animation and Virtual Worlds, 28(3-4), e1775.

[9] Lui, J. C., Chan, M. F., & Oldfield, K. Y. (1998). Dynamic partitioning for a distributed virtual environment. Department of Computer Science.

[10] Morillo, P., Fernandez, M., & Pelechano, N. (2004). A grid representation for Distributed Virtual Environments. In Grid Computing (pp. 182-189). Springer, Berlin, Heidelberg.

[11] Morillo, P., Fernandez, M., & Orduna, J. M. (2003, April). An ACS-based partitioning method for distributed virtual environment systems. In Parallel and Distributed Processing Symposium, 2003. Proceedings. International (pp. 8-pp). IEEE.

[12] Aguilar, L., Lalith, M., Ichimura, T., & Hori, M. (2017). On the performance and scalability of an HPC enhanced Multi Agent System based evacuation simulator. Procedia Computer Science, 108, 937-947.

[13] Vigueras, G., Lozano, M., & Orduna, J. M. (2011). Workload balancing in distributed crowd simulations: the partitioning method. The Journal of Supercomputing, 58(2), 261-269.

[14] Vigueras, G., Lozano, M., & Orduna, J. M. (2009). Enhancing workload balancing in distributed crowd simulations through the partitioning method. In Proceedings of the 2009 International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE (pp. 1117-1128).

[15] Lees, M., Logan, B., & Theodoropoulos, G. (2007). Distributed simulation of agent-based systems with HLA. ACM Transactions on Modeling and Computer Simulation (TOMACS), 17(3), 11.

[16] Xu, Y., Cai, W., Aydt, H., Lees, M., & Zehe, D. (2017). Relaxing synchronization in parallel agent-based road traffic simulation. ACM Transactions on Modeling and Computer Simulation (TOMACS), 27(2), 14.

[17] van Toll, W. G. (2017). Navigation for Characters and Crowds in Complex Virtual Environments (Doctoral dissertation, University Utrecht).

[18] van Toll, W., Cook, I. V., Atlas, F., van Kreveld, M. J., & Geraerts, R. (2017). The Medial Axis of a Multi-Layered Environment and its Application as a Navigation Mesh. arXiv preprint arXiv:1701.05141.

Appendices

A Data and Results

In this Appendix we will display raw data and additional tables and graphs for simulation results that were not shown in the Experimentation section.

A.1 Experiment 1: Simulation Step Performance

Average step time results for C-R:

(T,N)	Number of Characters					
	1K		10K		100K	
	AVG	SD	AVG	SD	AVG	SD
(1,2)	97.61	0.55	1127.47	2.89		
(1,4)	53.22	0.95	602.17	2.12		
(1,8)	33.29	0.18	377.87	3.44		
(1,16)	20.10	0.56	209.22	1.61		
(1,24)	19.31	1.33	186.48	7.90		
(1,32)	19.25	0.45	183.21	2.55		
(1,46)	16.25	0.44	145.75	4.10		
(2,2)	60.45	1.55	660.95	12.84		
(2,4)	34.29	0.28	379.12	8.85		
(2,8)	20.55	0.28	230.80	4.21		
(2,16)	17.50	0.55	167.58	1.61		
(2,21)	18.56	0.11	159.76	2.35		
(6,2)	25.47	0.54	287.60	14.90		
(6,4)	**15.26**	0.26	197.02	2.86		
(6,6)	16.96	0.15	153.04	3.24		
(11,2)	15.52	0.22	170.13	16.01		
(11,4)	18.75	0.35	**126.81**	0.72	**1379.80**	6.71
(23,2)	48.64	0.28	129.17	6.07	1400.77	5.34

Table A1. 1: Average step time (ms) results for C-R.

Average step time results for C-C:

(T,N)	Number of Characters					
	1K		10K		100K	
	AVG	SD	AVG	SD	AVG	SD
(1,2)	103.63	2.77	1073.35	2.84		
(1,4)	56.82	0.88	583.05	13.57		
(1,8)	34.98	0.58	372.41	4.12		
(1,16)	21.39	0.32	215.67	0.34		
(1,24)	21.87	1.30	200.46	3.28		
(1,32)	20.84	0.65	203.30	1.07		
(1,46)	20.85	0.81	189.33	3.85		
(2,2)	60.85	1.74	610.78	1.07		
(2,4)	36.14	0.71	379.60	6.92		
(2,8)	21.93	0.42	240.22	2.80		
(2,16)	19.22	0.30	174.04	2.291		
(2,21)	20.25	0.13	154.76	1.95		
(6,2)	26.33	0.80	276.27	4.62		
(6,4)	**16.80**	0.69	182.34	2.34		
(6,6)	18.02	0.54	159.53	1.54		
(11,2)	17.19	0.37	184.16	5.50		
(11,4)	20.29	1.21	**130.55**	2.15	**1422.86**	5.34
(23,2)	46.67	0.55	153.48	3.10		

Table A1. 2: Average step time (ms) results for C-C.

Average step time results for CM10-C:

(T,N)	Number of Characters					
	1K		10K		100K	
	AVG	SD	AVG	SD	AVG	SD
(1,2)	69.90	0.79	803.21	3.72		
(1,4)	37.92	0.83	429.63	6.15		
(1,8)	23.36	0.44	263.41	1.67		
(1,16)	12.86	0.23	139.32	0.49		
(1,24)	13.92	0.13	133.48	2.44		
(1,32)	13.66	0.078	132.91	1.07		
(1,46)	**9.45**	0.28	**91.77**	0.54	**1018.69**	2.75
(2,2)	43.44	0.50	496.03	16.93		
(2,4)	24.56	0.64	279.37	5.69		
(2,8)	14.89	0.25	185.74	3.68		
(2,16)	12.18	0.25	116.74	2.76		
(2,21)	11.93	0.20	100.09	1.25		
(6,2)	19.48	0.24	235.43	3.18		
(6,4)	11.36	0.25	137.03	2.67		
(6,6)	12.37	0.10	103.27	1.31		
(11,2)	12.63	0.32	155.51	7.25		
(11,4)	15.97	0.37	95.87	0.92	1099.61	1.35
(23,2)	44.20	1.88	128.40	4.31		

Table A1. 3: Average step time (ms) results for CM10-C.

Average step time results for S-R:

(T,N)	Number of Characters					
	1K		10K		100K	
	AVG	SD	AVG	SD	AVG	SD
(1,2)	355.88	24.57	3725.58	28.90		
(1,4)	204.23	4.43	2278.35	51.67		
(1,8)	151.59	8.74	1429.66	21.09		
(1,16)	103.36	6.15	883.87	28.60		
(1,24)	86.41	5.48	710.58	15.82		
(1,32)	84.74	2.77	786.34	3.32		
(1,46)	77.28	8.79	606.91	9.70		
(2,2)	194.07	5.02	2026.45	37.43		
(2,4)	125.12	5.88	1370.76	25.66		
(2,8)	89.48	12.01	833.48	8.75		
(2,16)	61.66	3.15	572.34	16.34		
(2,21)	63.11	3.94	529.26	21.47		
(6,2)	83.87	1.61	833.70	5.58		
(6,4)	51.37	2.24	512.59	2.51		
(6,6)	53.37	3.92	463.96	7.00		
(11,2)	49.08	1.88	482.42	1.05		
(11,4)	**51.32**	2.30	413.69	4.08	**4102.35**	68.10
(23,2)	66.35	1.27	**412.98**	13.24	4360.14	895.10

Table A1. 4: Average step time (ms) results for S-R.

Experiment 1 plots: Average step time (ms) vs. N:

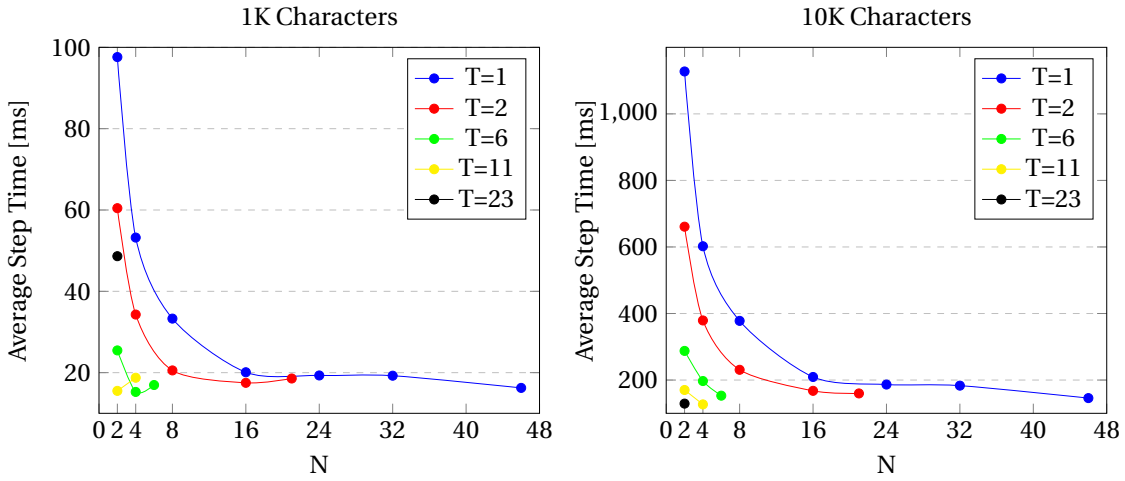


Figure A1. 1: Average step times for 1K (left) and 10K (right) characters in C-R.

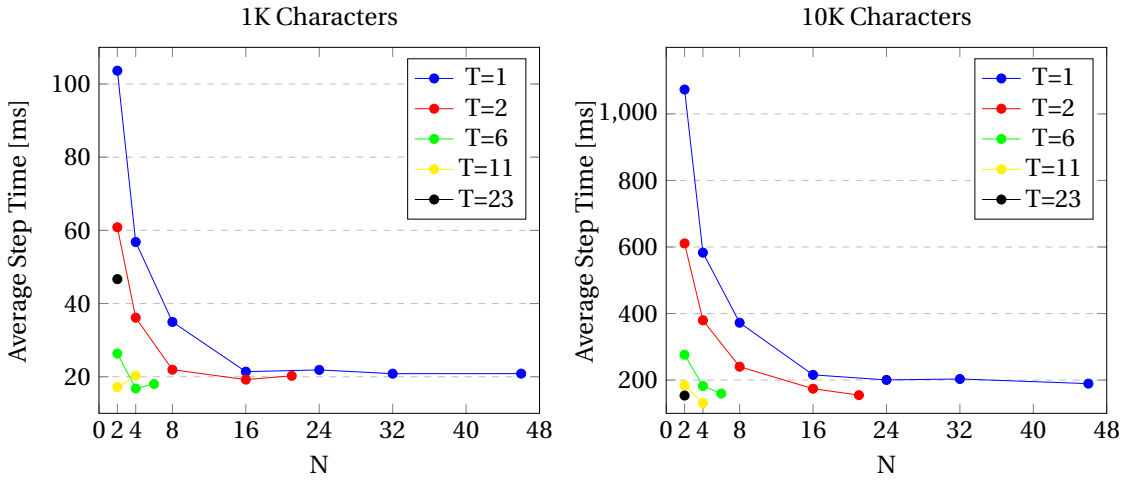


Figure A1. 2: Average step times for 1K (left) and 10K (right) characters in C-C.

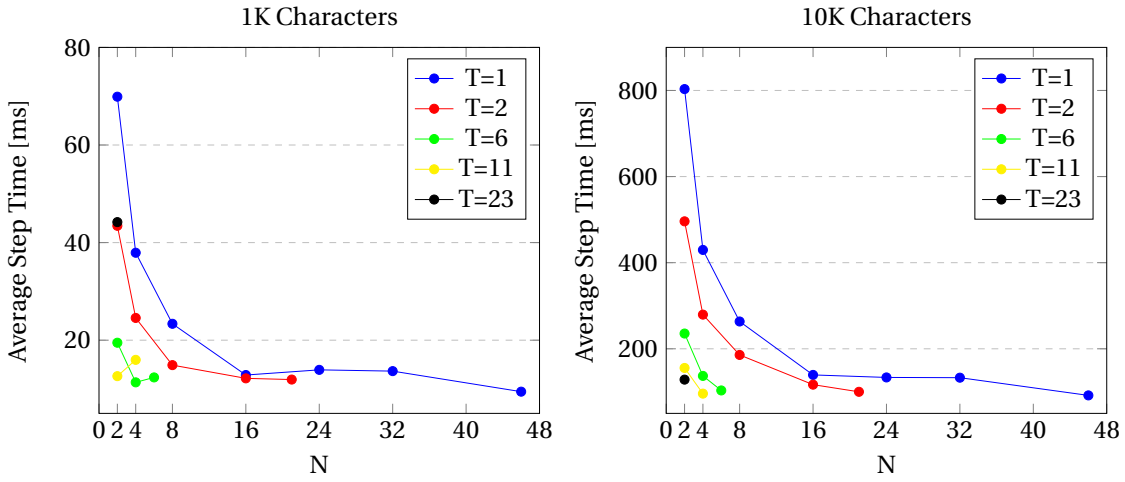


Figure A1. 3: Average step times for 1K (left) and 10K (right) characters in CM10-C.

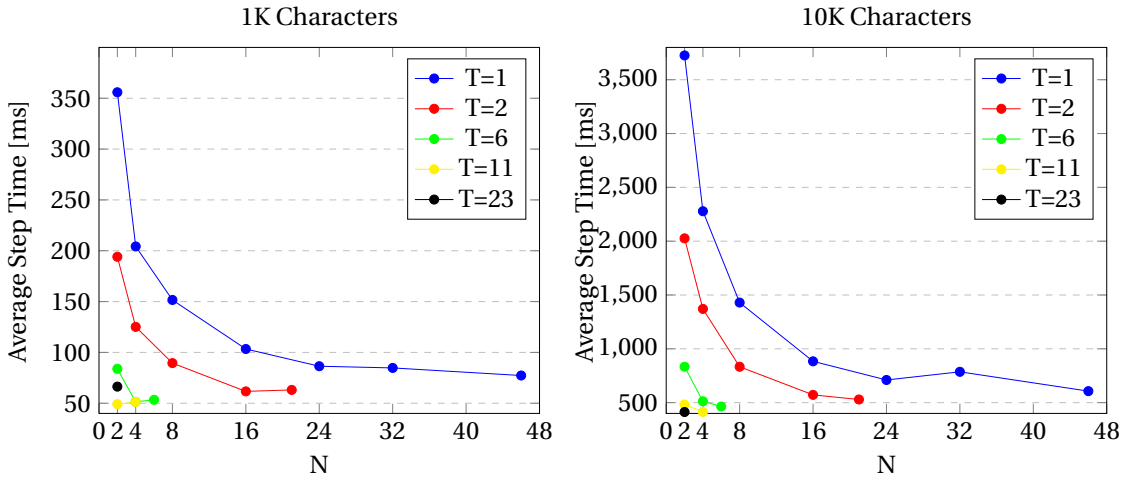


Figure A1. 4: Average step times for 1K (left) and 10K (right) characters in S-R.

A.2 Experiment 2: Simulation Substep Performance

Average step time results for C-R: Results for experiment 2 are displayed in the tables below.

C-R:

(T,N)		A	P1	S	D	G	P2	RG	M	R	Step Total
(1,2)	AVG	0.0035	1.87	0.08	4.62E-05	0.31	92.78	0.14	1.51	0.25	97.61
	SD	0.00029	0.011	0.0094	6.06E-06	0.043	0.52	0.023	0.23	0.0027	0.55
(1,4)	AVG	0.0025	0.97	0.12	4.78E-05	0.33	48.04	0.14	2.66	0.27	53.22
	SD	0.00032	0.013	0.0086	2.95E-06	0.019	0.54	0.010	0.68	0.0034	0.95
(1,8)	AVG	0.0016	0.50	0.11	4.88E-05	0.37	25.73	0.16	4.96	0.17	33.29
	SD	3.37E-05	0.0022	0.0030	8.37E-06	0.022	0.17	0.012	0.14	0.015	0.18
(1,16)	AVG	0.0024	0.26	0.11	4.78E-05	0.29	13.19	0.12	4.13	0.012	20.10
	SD	3.29E-05	0.0013	0.0035	2.17E-06	0.0091	0.16	0.0046	0.28	3.79E-05	0.56
(1,24)	AVG	0.0042	0.20	0.19	4.48E-05	0.29	9.51	0.12	4.80	0.010	19.31
	SD	0.00030	0.0014	0.0013	8.37E-06	0.0038	0.14	0.0017	0.67	3.33E-05	1.33
(1,32)	AVG	0.011	0.19	0.23	5.48E-05	0.35	9.11	0.13	5.27	0.011	19.25
	SD	0.0026	0.0010	0.0015	4.47E-06	0.011	0.093	0.0040	0.30	0.00012	0.45
(1,46)	AVG	0.021	0.16	0.34	6.32E-05	0.69	7.89	0.25	2.86	0.011	16.25
	SD	0.0013	0.00081	0.069	4.47E-06	0.039	0.034	0.0054	0.068	8.00E-05	0.44
(2,2)	AVG	0.0028	1.16	0.67	4.70E-05	0.31	55.55	0.14	1.94	0.35	60.45
	SD	0.00061	0.014	0.0036	3.39E-06	0.036	1.50	0.019	0.17	0.0028	1.55
(2,4)	AVG	0.0022	0.68	0.16	4.91E-05	0.37	29.89	0.16	1.83	0.33	34.29
	SD	0.00034	0.044	0.079	1.00E-06	0.0087	0.21	0.0046	0.27	0.072	0.28
(2,8)	AVG	0.0016	0.40	0.19	4.88E-05	0.36	15.42	0.16	2.89	0.14	20.55
	SD	3.71E-05	0.013	0.014	8.37E-06	0.017	0.11	0.0093	0.22	0.036	0.28
(2,16)	AVG	0.0028	0.35	1.11	5.36E-05	0.32	9.50	0.13	3.71	0.013	17.50
	SD	0.00010	0.010	0.12	2.07E-06	0.0057	0.10	0.0029	0.28	4.03E-05	0.55
(2,21)	AVG	0.0038	0.34	1.54	5.46E-05	0.34	8.85	0.16	4.06	0.011	18.56
	SD	5.25E-05	0.0095	0.076	1.14E-06	0.011	0.10	0.0055	0.072	9.23E-05	0.11
(6,2)	AVG	0.0013	0.53	0.055	4.70E-05	0.33	23.21	0.15	0.56	0.19	25.47
	SD	2.97E-05	0.0036	0.01	1.00E-06	0.058	0.35	0.0031	0.089	0.0036	0.54
(6,4)	AVG	0.0015	0.40	0.21	5.00E-05	0.37	12.37	0.16	1.07	0.21	15.26
	SD	2.14E-05	0.014	0.029	7.07E-07	0.035	0.16	0.018	0.096	0.021	0.26
(6,6)	AVG	0.0018	0.73	1.46	5.34E-05	0.41	10.62	0.19	2.41	0.26	16.96
	SD	3.27E-05	0.021	0.043	5.48E-07	0.021	0.19	0.012	0.079	0.024	0.15
(11,2)	AVG	0.0014	0.38	0.078	5.00E-05	0.36	13.66	0.15	0.42	0.14	15.52
	SD	9.98E-05	0.011	0.080	6.20E-06	0.052	0.21	0.0089	0.014	0.011	0.22
(11,4)	AVG	0.0022	1.49	1.86	5.94E-05	0.40	10.74	0.21	2.30	0.80	18.75
	SD	0.00094	0.043	0.064	2.19E-06	0.040	0.13	0.025	0.058	0.024	0.35
(23,2)	AVG	0.010	3.92	3.36	6.62E-05	0.41	24.31	0.21	6.86	2.93	48.64
	SD	0.0041	0.031	0.029	2.28E-06	0.054	0.29	0.032	0.095	0.018	0.28

Table A2. 1: Average substep and step times for C-R, 1K characters.

(T,N)		A	P1	S	D	G	P2	RG	M	R	Step Total
(1,2)	AVG	0.0043	17.82	0.27	5.08E-05	2.93	1082.28	1.29	11.18	3.37	1127.47
	SD	0.0015	0.029	0.045	1.30E-06	0.22	1.10	0.13	1.99	0.022	2.89
(1,4)	AVG	0.0033	9.17	0.46	5.12E-05	3.66	557.49	1.44	17.93	4.63	602.17
	SD	0.00044	0.042	0.041	1.30E-06	0.40	2.12	0.078	2.64	0.074	2.12
(1,8)	AVG	0.0022	4.93	0.57	5.44E-05	4.37	301.60	1.65	47.55	3.27	377.87
	SD	7.49E-05	0.021	0.019	2.97E-06	0.16	1.48	0.045	1.95	0.045	3.44
(1,16)	AVG	0.0032	2.51	0.46	5.48E-05	2.72	153.75	1.30	35.68	2.92	209.22
	SD	0.00034	0.0047	0.019	1.79E-06	0.070	0.43	0.030	1.59	0.12	1.61
(1,24)	AVG	0.0047	1.81	1.23	4.96E-05	2.64	108.52	1.24	38.25	1.69	186.48
	SD	0.0010	0.011	0.016	5.48E-07	0.043	0.52	0.02	2.88	0.14	7.90
(1,32)	AVG	0.0085	1.73	0.95	5.94E-05	3.22	103.55	1.48	46.42	1.44	183.21
	SD	0.0012	0.0047	0.016	5.48E-07	0.040	0.43	0.021	1.73	0.035	2.55
(1,46)	AVG	0.013	1.47	0.76	6.58E-05	7.21	94.02	2.69	18.44	1.29	145.75
	SD	0.0010	0.0023	0.011	4.47E-07	0.053	0.40	0.023	0.68	0.20	4.10
(2,2)	AVG	0.0037	13.13	2.49	6.00E-05	2.88	609.21	1.38	16.83	4.18	660.95
	SD	0.00056	0.36	0.26	4.85E-06	0.13	12.89	0.13	2.82	0.11	12.84
(2,4)	AVG	0.0041	7.21	1.95	6.5E-05	3.97	341.61	1.61	8.32	5.08	379.12
	SD	0.00024	0.39	0.39	2.91E-06	0.13	7.92	0.033	0.22	0.30	8.85
(2,8)	AVG	0.0022	4.021	1.68	6.42E-05	4.43	190.43	1.69	16.17	3.77	230.80
	SD	8.26E-05	0.074	0.087	1.64E-06	0.20	1.97	0.03	1.49	0.48	4.21
(2,16)	AVG	0.0034	2.79	2.77	6.14E-05	2.84	115.63	1.40	28.95	1.76	167.58
	SD	0.00057	0.028	0.062	2.97E-06	0.06	0.55	0.015	1.045	0.12	1.61
(2,21)	AVG	0.0045	2.05	3.07	5.78E-05	3.29	100.18	1.56	34.32	1.46	159.76
	SD	0.00044	0.022	0.14	1.48E-06	0.066	0.72	0.020	1.25	0.077	2.35
(6,2)	AVG	0.0019	5.11	0.76	6.00E-05	3.08	259.25	1.48	10.41	2.25	287.60
	SD	9.57E-05	0.47	0.31	5.15E-06	0.18	10.15	0.099	3.72	0.23	14.90
(6,4)	AVG	0.0019	3.58	1.24	6.68E-05	3.92	162.35	1.64	17.69	2.29	197.02
	SD	0.00012	0.060	0.024	3.27E-06	0.30	2.23	0.080	0.65	0.11	2.86
(6,6)	AVG	0.0023	3.39	2.33	6.28E-05	4.04	122.42	1.74	13.86	1.66	153.04
	SD	0.00012	0.099	0.098	2.28E-06	0.22	1.70	0.062	1.15	0.13	3.24
(11,2)	AVG	0.0020	3.27	0.66	5.76E-05	3.051	150.03	1.49	6.76	1.55	170.13
	SD	0.00012	0.48	0.36	5.030E-06	0.15	10.77	0.12	4.85	0.020	16.01
(11,4)	AVG	0.0020	2.88	1.36	6.42E-05	3.99	107.98	1.96	4.89	1.31	126.81
	SD	4.45E-05	0.057	0.13	1.48E-06	0.21	0.32	0.10	0.29	0.052	0.72
(23,2)	AVG	0.0023	2.79	0.87	6.28E-05	3.60	112.20	1.99	3.91	1.20	129.17
	SD	2.63E-05	0.17	0.12	1.64E-06	0.24	3.90	0.11	1.65	0.012	6.07

Table A2. 2: Average substep and step times for C-R, 10K characters.

(T,N)		A	P1	S	D	G	P2	RG	M	R	Step Total
(11,4)	AVG	0.0038	31.96	5.23	8.36E-05	45.06	1166.77	21.85	56.88	22.66	1379.80
	SD	0.00036	0.40	0.39	1.67E-06	1.42	4.96	0.57	4.74	0.17	6.71
(23,2)	AVG	0.0047	29.73	1.01	0.00011	31.16	1254.48	17.40	24.23	18.49	1400.77
	SD	8.96E-05	0.20	0.040	3.39E-06	0.16	5.57	0.23	0.57	0.23	5.34

Table A2. 3: Average substep and step times for C-R, 100K characters.

C-C:

(T,N)		A	P1	S	D	G	P2	RG	M	R	Step Total
(1,2)	AVG	0.0045	1.66	0.13	4.62E-05	1.02	92.83	0.49	6.54	0.17	103.63
	SD	0.00053	0.021	0.011	3.11E-06	0.16	1.36	0.025	1.38	0.0077	2.77
(1,4)	AVG	0.0040	0.86	0.19	4.68E-05	0.82	48.09	0.40	5.50	0.26	56.82
	SD	0.00061	0.0065	0.0093	2.397E-06	0.066	0.50	0.036	0.71	0.0094	0.88
(1,8)	AVG	0.0017	0.45	0.15	5.10E-05	0.99	26.18	0.48	4.99	0.14	34.98
	SD	0.00011	0.0011	0.0058	1.48E-06	0.037	0.17	0.017	0.36	0.0096	0.58
(1,16)	AVG	0.0025	0.23	0.14	4.92E-05	0.76	13.53	0.36	4.19	0.012	21.39
	SD	8.84E-05	0.0012	0.0048	1.64E-06	0.013	0.10	0.0072	0.29	6.06E-05	0.32
(1,24)	AVG	0.0044	0.17	0.21	4.70E-05	0.66	9.83	0.30	5.71	0.01	21.87
	SD	0.00023	0.0023	0.015	7.07E-07	0.018	0.13	0.0096	0.77	0.00013	1.30
(1,32)	AVG	0.0095	0.17	0.28	5.70E-05	0.95	9.72	0.41	5.87	0.011	21.84
	SD	0.00093	0.0013	0.015	7.07E-07	0.019	0.097	0.0090	0.42	4.45	0.65
(1,46)	AVG	0.023	0.14	0.36	6.40E-05	1.22	8.55	0.50	4.62	0.011	20.85
	SD	0.0037	0.0011	0.074	8.37E-07	0.050	0.12	0.022	0.53	0.00014	0.81
(2,2)	AVG	0.0051	1.02	0.13	4.76E-05	0.99	53.80	0.50	3.04	0.30	60.85
	SD	0.00042	0.0078	0.018	3.13E-06	0.06	1.60	0.029	0.33	0.018	1.74
(2,4)	AVG	0.0025	0.61	0.15	5.08E-05	0.81	29.60	0.39	3.25	0.35	36.14
	SD	0.00052	0.019	0.0090	2.17E-06	0.062	0.46	0.031	0.43	0.040	0.71
(2,8)	AVG	0.0018	0.36	0.22	5.02E-05	1.02	15.54	0.50	3.02	0.14	21.93
	SD	8.87E-05	0.012	0.016	8.37E-07	0.016	0.15	0.0071	0.31	0.0070	0.42
(2,16)	AVG	0.0027	0.36	1.43	5.50E-05	0.83	9.76	0.40	3.83	0.013	19.22
	SD	7.58E-05	0.012	0.064	1.00E-06	0.036	0.061	0.020	0.18	4.61E-05	0.30
(2,21)	AVG	0.0042	0.34	1.67	5.64E-05	0.76	9.07	0.41	4.43	0.011	20.25
	SD	0.00043	0.015	0.059	8.94E-07	0.026	0.069	0.015	0.096	7.22E-05	0.13
(6,2)	AVG	0.0015	0.50	0.12	4.92E-05	1.03	22.18	0.53	1.29	0.14	26.33
	SD	7.56E-05	0.016	0.0067	4.55E-06	0.035	0.66	0.017	0.21	0.0096	0.80
(6,4)	AVG	0.0016	0.40	0.23	5.02E-05	0.96	12.14	0.48	1.92	0.13	16.80
	SD	6.12E-05	0.0195	0.028	1.30E-06	0.073	0.12	0.040	0.54	0.012	0.69
(6,6)	AVG	0.0019	0.78	1.53	5.56E-05	0.77	10.55	0.39	2.88	0.15	18.02
	SD	9.08E-05	0.033	0.047	2.61E-06	0.05	0.11	0.036	0.53	0.026	0.54
(11,2)	AVG	0.0017	0.38	0.14	4.94E-05	1.12	13.36	0.56	1.06	0.093	17.19
	SD	3.30E-05	0.017	0.015	1.95E-06	0.063	0.25	0.014	0.22	0.0063	0.37
(11,4)	AVG	0.0018	1.50	1.93	5.86E-05	0.99	10.48	0.57	3.12	0.70	20.29
	SD	4.83E-05	0.054	0.11	2.07E-06	0.11	0.33	0.060	0.71	0.061	1.21
(23,2)	AVG	0.0026	3.81	3.29	6.86E-05	1.27	22.65	0.78	6.21	1.93	46.67
	SD	0.00092	0.035	0.053	3.29E-06	0.054	0.28	0.026	0.035	0.11	0.55

Table A2. 4: Average substep and step times for C-C, 1K characters.

(T,N)		A	P1	S	D	G	P2	RG	M	R	Step Total
(1,2)	AVG	0.0031	15.73	1.01	5.04E-05	9.68	987.13	5.39	41.39	2.95	1073.35
	SD	0.00024	0.11	0.011	1.34E-06	0.12	1.42	0.071	1.40	0.013	2.84
(1,4)	AVG	0.0038	8.13	1.21	5.14E-05	9.97	510.78	4.35	42.038	4.29	583.05
	SD	0.00068	0.011	0.091	1.14E-06	0.57	0.68	0.14	0.64	0.078	13.57
(1,8)	AVG	0.0025	4.37	1.11	5.56E-05	13.25	282.21	5.57	47.79	2.84	372.41
	SD	7.58E-05	0.011	0.057	1.67E-06	0.67	0.35	0.10	2.93	0.14	4.12
(1,16)	AVG	0.0036	2.24	0.75	5.62E-05	11.50	146.85	4.36	34.67	2.20	215.67
	SD	0.00011	0.0063	0.027	1.64E-06	0.23	0.094	0.063	0.48	0.13	0.34
(1,24)	AVG	0.0070	1.62	1.45	5.14E-05	9.59	104.31	3.59	44.02	1.17	200.46
	SD	0.0013	0.0053	0.071	5.48E-07	0.21	0.33	0.071	2.44	0.089	3.28
(1,32)	AVG	0.010	1.59	1.46	6.08E-05	14.34	103.74	5.16	47.40	1.17	203.30
	SD	0.0015	0.0067	0.036	8.37E-07	0.14	0.29	0.039	1.54	0.077	1.07
(1,46)	AVG	0.020	1.34	1.03	6.6E-05	16.13	94.32	5.77	39.29	1.33	189.33
	SD	0.0012	0.0018	0.032	0.00	0.23	0.13	0.069	1.75	0.12	3.85
(2,2)	AVG	0.0051	12.24	2.34	6.78E-05	9.99	547.52	5.61	16.49	3.57	610.78
	SD	0.00027	0.13	0.032	1.30E-06	0.19	1.36	0.11	0.13	0.013	1.07
(2,4)	AVG	0.0047	7.51	2.15	6.74E-05	9.70	318.81	4.51	21.62	5.22	379.60
	SD	0.00020	0.49	0.063	1.52E-06	0.80	6.31	0.19	2.15	0.14	6.92
(2,8)	AVG	0.0027	3.59	1.48	6.44E-05	12.76	177.13	5.54	24.27	3.50	240.22
	SD	9.94E-05	0.035	0.14	1.14E-06	0.53	0.75	0.21	2.35	0.048	2.80
(2,16)	AVG	0.0039	2.64	3.12	6.36E-05	11.82	108.29	4.66	29.19	1.53	174.04
	SD	0.00025	0.040	0.15	2.07E-06	0.28	0.78	0.12	1.10	0.082	2.29
(2,21)	AVG	0.0059	1.78	2.62	5.96E-05	11.21	92.01	4.31	28.66	0.98	154.76
	SD	0.00039	0.0071	0.048	1.52E-06	0.13	1.14	0.049	1.25	0.039	1.95
(6,2)	AVG	0.0020	4.83	1.35	6.42E-05	10.03	231.13	5.72	15.20	1.77	276.27
	SD	7.61E-05	0.32	0.15	7.56E-06	0.068	3.65	0.057	0.58	0.14	4.62
(6,4)	AVG	0.0021	3.26	1.68	6.52E-05	10.43	137.41	4.63	18.02	1.83	182.34
	SD	9.36E-05	0.052	0.10	4.15E-06	0.94	0.97	0.39	0.35	0.093	2.34
(6,6)	AVG	0.0026	3.66	3.27	6.68E-05	9.78	113.62	4.14	18.91	1.61	159.53
	SD	0.00010	0.055	0.083	2.594E-06	0.26	0.85	0.090	0.65	0.060	1.54
(11,2)	AVG	0.0021	3.099	1.02	5.58E-05	9.92	144.41	5.63	14.40	1.16	184.16
	SD	0.00020	0.036	0.018	2.05E-06	0.15	4.29	0.085	0.89	0.016	5.50
(11,4)	AVG	0.0023	2.70	1.43	6.40E-05	10.80	95.89	5.079	10.63	0.96	130.55
	SD	0.00014	0.080	0.14	1.22E-06	0.74	1.41	0.010	1.38	0.11	2.15
(23,2)	AVG	0.0025	4.16	2.46	6.98E-05	10.66	113.82	6.26	9.70	1.21	153.48
	SD	0.00012	0.34	0.36	2.28E-06	0.16	1.67	0.094	0.70	0.26	3.10

Table A2. 5: Average substep and step times for C-C, 10K characters.

(T,N)		A	P1	S	D	G	P2	RG	M	R	Step Total
(11,4)	AVG	0.17	14.37	13.31	9.78E-05	178.23	1009.01	58.68	450.26	16.90	2301.23
	SD	0.092	0.043	5.02	3.56E-06	1.25	0.61	0.14	5.61	1.09	22.77

Table A2. 6: Average substep and step times for C-C, 100K characters.

CM10-R:

(T,N)		A	P1	S	D	G	P2	RG	M	R	Step Total
(1,2)	AVG	0.0040	1.51	0.079	4.50E-05	0.077	70.27	0.018	0.97	0.26	73.86
	SD	0.00039	0.019	0.017	3.67E-06	0.010	0.78	0.0049	0.22	0.0021	0.98
(1,4)	AVG	0.0022	0.78	0.086	4.66E-05	0.094	35.75	0.023	1.80	0.24	39.49
	SD	0.00080	0.0026	0.0048	1.67E-06	0.0061	0.33	0.0027	0.44	0.050	0.63
(1,8)	AVG	0.0015	0.40	0.073	4.6E-05	0.090	19.15	0.03	1.15	0.18	21.83
	SD	5.22E-05	0.0048	0.0023	2.00E-06	0.0029	0.21	0.0016	0.070	0.021	0.28
(1,16)	AVG	0.0025	0.21	0.081	4.54E-05	0.075	9.74	0.02	1.03	0.011	12.24
	SD	0.00035	0.0020	0.0036	1.52E-06	0.0030	0.041	0.0014	0.073	5.39E-05	0.16
(1,24)	AVG	0.0050	0.16	0.15	4.36E-05	0.076	7.14	0.020	2.43	0.0090	13.56
	SD	0.0011	0.0043	0.013	1.95E-06	0.0021	0.16	0.00068	0.33	0.00021	0.24
(1,32)	AVG	0.012	0.15	0.18	5.24E-05	0.092	6.73	0.023	2.58	0.0097	11.71
	SD	0.0049	0.00073	0.0093	5.48E-07	0.0037	0.086	0.0015	0.029	6.74E-05	0.16
(1,46)	AVG	0.014	0.13	0.21	6.20E-05	0.15	5.82	0.037	1.49	0.010	9.68
	SD	0.00040	0.00099	0.028	7.07E-07	0.024	0.025	0.0011	0.047	4.13E-05	0.21
(2,2)	AVG	0.0020	0.95	0.058	4.26E-05	0.090	41.87	0.021	1.19	0.33	45.37
	SD	0.0011	0.0056	0.0014	3.65E-06	0.0076	0.63	0.0043	0.12	0.066	0.71
(2,4)	AVG	0.0012	0.52	0.10	4.90E-05	0.099	22.35	0.024	0.81	0.27	24.85
	SD	7.45E-05	0.0077	0.0075	2.55E-06	0.014	0.23	0.0071	0.093	0.052	0.30
(2,8)	AVG	0.0015	0.29	0.13	4.62E-05	0.089	11.39	0.024	0.76	0.17	13.45
	SD	0.00013	0.0070	0.015	8.37E-07	0.0030	0.15	0.0017	0.059	0.040	0.20
(2,16)	AVG	0.0029	0.25	0.62	5.08E-05	0.084	7.22	0.023	1.73	0.012	11.48
	SD	6.25E-05	0.0036	0.026	8.37E-07	0.0029	0.064	0.0016	0.032	6.85E-05	0.17
(2,21)	AVG	0.0044	0.24	1.035	5.48E-05	0.089	6.65	0.027	1.56	0.012	11.46
	SD	0.00024	0.0059	0.033	4.47E-07	0.0038	0.041	0.0019	0.063	0.00011	0.14
(6,2)	AVG	0.0012	0.38	0.033	4.26E-05	0.098	17.09	0.024	0.31	0.21	18.58
	SD	7.20E-05	0.023	0.015	2.07E-06	0.0066	0.19	0.0033	0.030	0.0048	0.22
(6,4)	AVG	0.0014	0.33	0.15	4.70E-05	0.11	9.23	0.028	0.96	0.24	11.54
	SD	3.19E-05	0.0055	0.011	0.00	0.0079	0.088	0.0036	0.044	0.021	0.14
(6,6)	AVG	0.0018	0.62	1.38	5.22E-05	0.11	7.93	0.031	1.32	0.50	12.66
	SD	2.31E-05	0.015	0.020	1.30E-06	0.0039	0.047	0.0017	0.14	0.024	0.17
(11,2)	AVG	0.0013	0.29	0.065	4.96E-05	0.11	10.16	0.024	0.34	0.15	11.45
	SD	6.18E-05	0.010	0.0097	5.32E-06	0.0062	0.045	0.0039	0.050	0.011	0.072
(11,4)	AVG	0.0017	1.40	1.73	5.58E-05	0.12	8.13	0.033	1.93	0.72	15.12
	SD	5.69E-05	0.034	0.057	1.30E-06	0.0048	0.11	0.0020	0.13	0.048	0.37
(23,2)	AVG	0.0057	3.87	3.35	6.28E-05	0.12	21.61	0.032	7.32	2.51	46.10
	SD	0.0036	0.028	0.012	3.11E-06	0.020	0.16	0.011	0.11	0.017	0.25

Table A2. 7: Average substep and step times for CM10-R, 1K characters.

(T,N)		A	P1	S	D	G	P2	RG	M	R	Step Total
(1,2)	AVG	0.0035	14.37	0.24	4.72E-05	0.65	711.94	0.18	4.96	3.39	743.85
	SD	0.00065	0.058	0.035	3.11E-06	0.033	3.97	0.019	1.46	0.023	4.81
(1,4)	AVG	0.0028	7.42	0.31	4.70E-05	0.64	364.95	0.16	14.03	3.90	399.00
	SD	0.00044	0.013	0.013	3.08E-06	0.025	1.02	0.014	1.58	0.81	2.39
(1,8)	AVG	0.0020	3.96	0.25	5.26E-05	0.56	194.53	0.17	4.17	3.98	215.10
	SD	0.00023	0.0090	0.010	1.14E-06	0.023	0.78	0.011	0.54	0.41	0.58
(1,16)	AVG	0.0031	2.03	0.21	5.30E-05	0.40	98.83	0.13	3.70	4.36	117.11
	SD	0.00051	0.0030	0.0055	1.41E-06	0.0080	0.11	0.0034	0.48	0.46	0.33
(1,24)	AVG	0.0044	1.51	1.00	4.80E-05	0.40	71.30	0.13	16.82	1.51	121.17
	SD	0.00031	0.036	0.050	1.73E-06	0.022	1.49	0.0085	2.15	0.25	2.36
(1,32)	AVG	0.0079	1.44	0.69	5.72E-05	0.44	67.24	0.14	18.33	3.61	102.32
	SD	0.0019	0.0098	0.028	8.37E-07	0.011	0.40	0.0044	0.26	0.16	0.67
(1,46)	AVG	0.014	1.23	0.38	6.30E-05	0.68	58.99	0.26	6.02	6.12	83.09
	SD	0.0015	0.0068	0.024	0.00	0.017	0.49	0.0046	0.43	0.98	0.97
(2,2)	AVG	0.0029	10.20	1.21	6.06E-05	0.67	422.68	0.19	9.03	4.15	458.90
	SD	0.00090	0.21	0.18	3.13E-06	0.034	4.84	0.019	3.48	0.11	5.56
(2,4)	AVG	0.0045	4.99	0.67	5.58E-05	0.70	227.34	0.18	5.03	4.74	252.51
	SD	0.00058	0.097	0.095	3.03E-06	0.020	1.34	0.0092	0.63	0.40	2.29
(2,8)	AVG	0.0020	2.90	0.55	5.66E-05	0.58	123.81	0.18	6.80	4.55	147.95
	SD	0.00025	0.22	0.077	1.52E-06	0.021	4.83	0.011	1.82	0.52	8.05
(2,16)	AVG	0.0031	2.16	2.18	5.90E-05	0.44	73.37	0.14	11.60	1.46	99.91
	SD	0.00013	0.029	0.072	7.07E-07	0.0090	0.45	0.0047	0.24	0.094	0.74
(2,21)	AVG	0.0040	1.68	2.52	5.60E-05	0.41	65.29	0.16	8.19	1.78	86.71
	SD	0.00015	0.012	0.053	7.07E-07	0.010	0.16	0.0055	0.21	0.15	0.34
(6,2)	AVG	0.0017	4.31	0.65	5.88E-05	0.71	193.16	0.20	10.16	2.35	216.72
	SD	9.91E-05	0.73	0.38	8.29E-06	0.045	19.91	0.027	7.80	0.20	28.98
(6,4)	AVG	0.0016	2.81	0.91	5.70E-05	0.76	109.55	0.20	13.27	2.06	133.84
	SD	4.69E-05	0.14	0.099	3.08E-06	0.022	4.23	0.012	1.77	0.072	6.33
(6,6)	AVG	0.0019	2.70	2.28	5.92E-05	0.68	76.57	0.21	6.86	1.51	94.53
	SD	0.00011	0.040	0.030	2.17E-06	0.025	0.46	0.0064	0.76	0.18	0.98
(11,2)	AVG	0.0017	2.77	0.51	5.76E-05	0.76	116.56	0.20	10.66	1.67	136.75
	SD	0.00010	0.43	0.18	5.81E-06	0.0054	16.86	0.0069	8.61	0.033	26.16
(11,4)	AVG	0.0018	2.85	2.06	6.00E-05	0.79	72.88	0.25	4.59	1.63	87.94
	SD	0.00011	0.047	0.065	2.74E-06	0.034	0.38	0.023	0.46	0.11	0.78
(23,2)	AVG	0.0020	2.62	1.05	6.10E-05	1.01	75.61	0.30	1.91	1.57	87.37
	SD	2.43E-05	0.28	0.29	3.16E-06	0.033	1.67	0.026	0.89	0.22	4.08

Table A2. 8: Average substep and step times for CM10-R, 10K characters.

(T,N)		A	P1	S	D	G	P2	RG	M	R	Step Total
(1,46)	AVG	0.045	12.37	4.27	9.76E-05	7.80	610.26	2.64	58.75	88.70	900.03
	SD	0.0025	0.013	0.064	7.77E-06	0.029	1.01	0.018	0.41	3.53	4.09

Table A2. 9: Average substep and step times for CM10-R, 100K characters.

CM10-C:

(T,N)		A	P1	S	D	G	P2	RG	M	R	Step Total
(1,2)	AVG	0.0045	1.47	0.23	4.72E-05	1.06	61.07	0.49	4.47	0.18	69.90
	SD	0.00063	0.011	0.020	3.83E-06	0.22	0.16	0.043	0.62	0.012	0.79
(1,4)	AVG	0.0031	0.77	0.17	4.84E-05	0.54	31.64	0.25	3.49	0.22	37.92
	SD	0.00032	0.0088	0.014	1.82E-06	0.011	0.24	0.0051	0.74	0.05	0.83
(1,8)	AVG	0.0017	0.41	0.15	4.86E-05	0.81	17.31	0.39	2.83	0.11	23.36
	SD	6.62E-05	0.0044	0.0092	1.52E-06	0.011	0.17	0.0053	0.25	0.013	0.44
(1,16)	AVG	0.0024	0.21	0.12	4.82E-05	0.46	8.71	0.21	1.73	0.011	12.86
	SD	0.00011	0.00056	0.011	1.10E-06	0.010	0.026	0.0050	0.12	6.98E-05	0.23
(1,24)	AVG	0.0049	0.16	0.20	4.58E-05	0.37	6.44	0.16	2.96	0.0096	13.92
	SD	0.00064	0.0012	0.019	8.37E-07	0.016	0.040	0.0069	0.041	4.29E-05	0.13
(1,32)	AVG	0.011	0.15	0.26	5.50E-05	0.68	6.21	0.29	3.13	0.011	13.66
	SD	0.0025	0.0026	0.012	7.07E-07	0.018	0.065	0.0068	0.057	8.13E-05	0.078
(1,46)	AVG	0.015	0.13	0.23	6.30E-05	0.30	5.16	0.093	1.64	0.011	9.45
	SD	0.0011	0.0011	0.056	1.22E-06	0.034	0.047	0.0039	0.098	0.00012	0.28
(2,2)	AVG	0.0042	0.96	0.20	4.64E-05	1.01	37.33	0.49	2.08	0.28	43.44
	SD	0.00065	0.022	0.016	3.29E-06	0.17	0.31	0.040	0.32	0.014	0.50
(2,4)	AVG	0.0021	0.55	0.17	4.88E-05	0.62	19.83	0.30	2.05	0.27	24.56
	SD	0.00094	0.024	0.014	1.48E-06	0.029	0.26	0.015	0.44	0.046	0.64
(2,8)	AVG	0.0017	0.33	0.23	5.00E-05	0.83	10.39	0.41	1.70	0.086	14.89
	SD	9.67E-05	0.012	0.015	1.58E-06	0.015	0.19	0.0076	0.13	0.015	0.25
(2,16)	AVG	0.0028	0.27	0.85	5.28E-05	0.51	6.33	0.23	2.08	0.012	12.18
	SD	0.00013	0.0059	0.046	4.47E-07	0.0069	0.054	0.0036	0.12	3.55E-05	0.25
(2,21)	AVG	0.0045	0.24	1.15	5.50E-05	0.35	5.82	0.17	1.96	0.012	11.93
	SD	0.00016	0.010	0.079	7.07E-07	0.019	0.021	0.0094	0.051	6.75E-05	0.20
(6,2)	AVG	0.0014	0.46	0.15	4.90E-05	1.09	15.54	0.56	0.95	0.13	19.48
	SD	3.49E-05	0.019	0.015	2.74E-06	0.082	0.17	0.032	0.029	0.0068	0.24
(6,4)	AVG	0.0016	0.32	0.20	4.96E-05	0.64	8.18	0.31	0.96	0.17	11.36
	SD	5.36E-05	0.0075	0.011	1.82E-06	0.039	0.11	0.020	0.091	0.015	0.25
(6,6)	AVG	0.0019	0.64	1.41	5.34E-05	0.49	7.04	0.24	1.32	0.39	12.37
	SD	6.77E-05	0.0075	0.038	5.48E-07	0.019	0.074	0.0095	0.10	0.011	0.10
(11,2)	AVG	0.0015	0.33	0.12	4.88E-05	1.20	9.07	0.57	0.79	0.10	12.63
	SD	4.83E-05	0.012	0.010	1.92E-06	0.13	0.18	0.020	0.11	0.0020	0.32
(11,4)	AVG	0.0018	1.48	1.92	5.68E-05	0.69	7.19	0.38	2.49	0.61	15.97
	SD	3.59E-05	0.027	0.047	1.10E-06	0.028	0.077	0.016	0.18	0.040	0.37
(23,2)	AVG	0.0068	3.80	3.35	6.78E-05	1.21	18.86	0.75	7.34	2.02	44.20
	SD	0.0059	0.12	0.13	1.79E-06	0.17	0.91	0.053	0.52	0.18	1.88

Table A2. 10: Average substep and step times for CM10-C, 1K characters.

(T,N)		A	P1	S	D	G	P2	RG	M	R	Step Total
(1,2)	AVG	0.0031	14.33	2.18	4.40E-05	9.76	708.40	5.46	50.49	2.80	803.21
	SD	0.00043	0.043	0.048	4.18E-06	0.24	2.43	0.12	1.64	0.042	3.72
(1,4)	AVG	0.0032	7.38	1.32	4.60E-05	5.95	363.34	2.82	35.87	3.59	429.63
	SD	0.00015	0.046	0.10	2.35E-06	0.41	2.17	0.058	4.89	0.89	6.15
(1,8)	AVG	0.0024	3.97	1.15	4.96E-05	9.83	199.58	4.32	28.87	2.59	263.41
	SD	5.91E-05	0.016	0.12	8.94E-07	0.27	0.59	0.032	1.12	0.37	1.67
(1,16)	AVG	0.0034	2.03	0.68	5.06E-05	5.66	101.23	2.27	14.65	2.43	139.32
	SD	0.00014	0.0050	0.028	5.48E-07	0.14	0.26	0.037	0.51	0.16	0.49
(1,24)	AVG	0.0060	1.51	1.42	5.90E-05	4.44	73.00	1.79	23.24	1.33	133.48
	SD	0.00059	0.018	0.023	7.07E-07	0.083	0.65	0.034	1.85	0.099	2.44
(1,32)	AVG	0.012	1.47	1.38	6.92E-05	8.80	71.86	3.30	26.58	1.20	132.91
	SD	0.0013	0.0074	0.048	4.47E-07	0.052	0.38	0.015	0.51	0.064	1.07
(1,46)	AVG	0.020	1.24	0.71	7.66E-05	2.22	59.96	0.94	12.42	3.98	91.77
	SD	0.0027	0.0062	0.055	5.48E-07	0.099	0.41	0.028	0.62	0.30	0.54
(2,2)	AVG	0.0037	13.54	3.89	6.64E-05	9.68	420.80	5.49	26.61	3.47	496.03
	SD	0.00040	1.38	0.59	9.79E-06	0.22	12.59	0.13	2.32	0.021	16.93
(2,4)	AVG	0.0045	6.81	2.82	6.26E-05	6.21	231.74	2.95	13.54	4.43	279.37
	SD	0.00067	0.30	0.25	3.21E-06	0.22	5.87	0.077	1.77	0.23	5.69
(2,8)	AVG	0.0025	3.54	2.09	5.96E-05	9.87	131.83	4.45	19.43	3.33	185.74
	SD	4.35E-05	0.038	0.39	2.61E-06	0.23	3.27	0.047	0.42	0.24	3.68
(2,16)	AVG	0.0036	2.32	2.41	5.68E-05	5.81	75.49	2.36	16.10	1.25	116.76
	SD	7.02E-05	0.059	0.15	1.10E-06	0.048	1.35	0.034	1.13	0.12	2.76
(2,21)	AVG	0.0064	1.69	2.64	6.02E-05	3.63	65.80	1.61	14.34	1.06	100.09
	SD	0.0016	0.011	0.03	4.47E-07	0.058	0.40	0.028	0.77	0.090	1.25
(6,2)	AVG	0.0019	4.86	1.71	6.18E-05	10.05	190.38	5.77	14.89	1.60	235.43
	SD	5.08E-05	0.14	0.089	5.17E-06	0.14	2.91	0.076	0.63	0.069	3.18
(6,4)	AVG	0.0019	2.91	1.42	5.70E-05	6.18	103.46	3.08	12.89	1.68	137.03
	SD	4.22E-05	0.059	0.041	2.55E-06	0.29	2.12	0.024	0.87	0.24	2.67
(6,6)	AVG	0.0024	2.80	2.53	6.02E-05	5.03	76.36	2.39	8.49	1.31	103.27
	SD	5.80E-05	0.046	0.098	3.11E-06	0.20	0.63	0.055	0.67	0.095	1.31
(11,2)	AVG	0.0021	3.03	1.34	5.60E-05	9.77	116.18	5.59	13.65	1.07	155.51
	SD	9.92E-05	0.071	0.069	4.74E-06	0.24	5.25	0.13	1.54	0.037	7.25
(11,4)	AVG	0.0021	2.57	1.53	6.12E-05	6.33	71.02	3.33	6.87	1.03	95.87
	SD	7.79E-05	0.027	0.065	1.30E-06	0.37	1.34	0.096	0.38	0.099	0.92
(23,2)	AVG	0.0023	3.85	2.56	6.44E-05	10.54	89.63	6.20	9.57	1.03	128.40
	SD	5.44E-05	0.43	0.43	2.19E-06	0.23	1.97	0.13	0.87	0.32	4.31

Table A2. 11: Average substep and step times for CM10-C, 10K characters.

(T,N)		A	P1	S	D	G	P2	RG	M	R	Step Total
(1,46)	AVG	0.047	12.17	5.32	8.44E-05	31.12	663.33	10.92	100.56	70.46	1018.69
	SD	0.0056	0.053	0.18	2.97E-06	0.030	0.44	0.04	2.11	4.39	2.75
(11,4)	AVG	0.080	28.36	8.67	0.00010	90.72	807.18	46.58	56.48	20.65	1099.61
	SD	0.17	4.58	0.38	3.91E-06	0.60	0.64	0.24	1.25	0.51	1.35

Table A2. 12: Average substep and step times for CM10-C, 100K characters.

S-R:

(T,N)		A	P1	S	D	G	P2	RG	M	R	Step Total
(1,2)	AVG	0.0044	2.55	0.28	4.84E-05	1.44	281.88	0.69	68.11	0.22	355.88
	SD	0.00058	0.049	0.027	3.21E-06	0.091	8.87	0.036	17.05	0.0034	24.57
(1,4)	AVG	0.0038	1.31	0.39	4.82E-05	1.29	139.32	0.64	60.35	0.27	204.23
	SD	0.0013	0.015	0.019	2.28E-06	0.037	2.42	0.016	6.29	0.0032	4.43
(1,8)	AVG	0.0020	0.70	0.51	5.24E-05	1.67	78.17	0.76	37.88	0.086	151.59
	SD	0.00013	0.011	0.041	3.51E-06	0.041	1.36	0.0069	3.54	0.0031	8.74
(1,16)	AVG	0.0032	0.36	0.46	5.02E-05	1.21	39.79	0.58	28.42	0.013	103.36
	SD	0.00048	0.0081	0.014	1.64E-06	0.016	0.65	0.0081	1.53	7.45E-05	6.15
(1,24)	AVG	0.0058	0.27	0.54	4.70E-05	1.18	29.00	0.55	24.42	0.011	86.41
	SD	0.0019	0.0079	0.061	7.07E-07	0.015	0.39	0.0070	2.30	0.00026	5.48
(1,32)	AVG	0.017	0.25	0.65	5.68E-05	1.61	27.67	0.70	26.36	0.013	84.74
	SD	0.0073	0.0014	0.018	8.37E-07	0.021	0.83	0.0097	0.67	0.00011	2.77
(1,46)	AVG	0.037	0.22	0.72	6.52E-05	2.11	24.61	0.85	22.59	0.014	77.28
	SD	0.0097	0.0043	0.056	4.47E-07	0.044	1.05	0.012	2.46	0.00023	8.79
(2,2)	AVG	0.0053	2.07	0.44	5.26E-05	1.56	169.15	0.73	18.95	0.27	194.07
	SD	0.00038	0.25	0.19	3.36E-06	0.040	2.65	0.025	3.07	0.055	5.02
(2,4)	AVG	0.0031	1.16	0.38	5.44E-05	1.43	94.82	0.71	25.41	0.38	125.12
	SD	0.00099	0.087	0.086	2.88E-06	0.022	3.83	0.015	2.44	0.053	5.88
(2,8)	AVG	0.0020	0.71	0.46	5.50E-05	1.69	51.47	0.79	20.28	0.10	89.48
	SD	0.00020	0.037	0.071	2.12E-06	0.060	3.83	0.031	3.42	0.022	12.01
(2,16)	AVG	0.0034	0.48	1.18	5.86E-05	1.32	28.70	0.66	15.62	0.013	61.66
	SD	0.00036	0.022	0.12	1.14E-06	0.037	1.07	0.032	0.94	0.00011	3.15
(2,21)	AVG	0.0047	0.53	2.10	5.86E-05	1.27	26.01	0.70	17.05	0.012	63.11
	SD	0.00036	0.039	0.17	1.34E-06	0.041	0.80	0.013	1.41	0.00012	3.94
(6,2)	AVG	0.0016	1.02	0.17	5.72E-05	1.56	71.21	0.80	8.45	0.16	83.87
	SD	8.59E-05	0.036	0.019	2.17E-06	0.024	1.27	0.012	0.50	0.0087	1.61
(6,4)	AVG	0.0017	0.76	0.46	6.22E-05	1.49	37.83	0.75	9.39	0.12	51.37
	SD	7.64E-05	0.028	0.074	3.56E-06	0.15	1.61	0.059	0.80	0.0028	2.24
(6,6)	AVG	0.0020	1.30	1.93	6.92E-05	1.43	33.52	0.77	12.61	0.14	53.37
	SD	5.71E-05	0.090	0.14	1.48E-06	0.040	1.65	0.026	2.45	0.033	3.92
(11,2)	AVG	0.0017	0.83	0.19	6.30E-05	1.66	38.88	0.83	6.18	0.12	49.08
	SD	9.17E-05	0.030	0.023	5.43E-06	0.14	1.49	0.061	0.45	0.0058	1.88
(11,4)	AVG	0.0020	1.96	2.06	7.36E-05	1.64	33.59	0.94	9.74	0.73	51.32
	SD	0.00014	0.067	0.046	8.94E-07	0.075	1.36	0.034	0.97	0.027	2.30
(23,2)	AVG	0.0066	4.23	3.34	7.30E-05	2.06	42.10	1.11	5.63	1.42	66.35
	SD	0.0044	0.15	0.16	5.34E-06	0.064	0.83	0.036	0.25	0.078	1.27

Table A2. 13: Average substep and step times for S-R, 1K characters.

(T,N)		A	P1	S	D	G	P2	RG	M	R	Step Total
(1,2)	AVG	0.078	21.66	1.94	5.32E-05	13.75	3056.60	7.81	612.68	3.83	3725.58
	SD	0.17	0.11	0.0057	1.30E-06	0.089	19.82	0.035	11.33	0.0035	28.90
(1,4)	AVG	0.0044	11.24	2.30	5.42E-05	17.02	1599.85	6.96	624.07	5.22	2278.35
	SD	1.92E-05	0.069	0.057	8.37E-07	0.31	11.59	0.15	43.57	0.018	51.67
(1,8)	AVG	0.0026	6.083	4.28	5.68E-05	22.28	862.48	8.70	326.19	1.74	1429.66
	SD	2.03E-05	0.028	0.052	8.37E-07	0.59	5.13	0.060	10.34	0.16	21.09
(1,16)	AVG	0.0038	3.13	2.73	5.88E-05	18.66	445.40	7.15	224.62	1.09	883.87
	SD	0.00018	0.032	0.033	1.30E-06	0.25	8.30	0.096	13.83	0.06	28.60
(1,24)	AVG	0.0090	2.27	2.66	5.28E-05	18.74	317.09	6.74	193.59	0.89	710.58
	SD	0.0013	0.014	0.11	8.37E-07	0.43	1.84	0.11	2.73	0.040	15.82
(1,32)	AVG	0.023	2.21	3.38	6.56E-05	24.53	303.80	8.49	228.44	1.13	786.34
	SD	0.0022	0.026	0.051	5.48E-07	0.25	1.92	0.11	4.64	0.054	3.32
(1,46)	AVG	0.017	1.82	2.52	7.16E-05	28.40	275.04	10.14	155.50	0.89	606.91
	SD	0.0063	0.0046	0.047	5.48E-07	0.30	1.70	0.047	4.61	0.055	9.70
(2,2)	AVG	0.0036	18.67	2.41	6.10E-05	13.98	1688.89	7.92	282.86	4.54	2026.45
	SD	0.00020	0.19	0.12	1.00E-06	0.36	25.43	0.21	10.14	0.17	37.43
(2,4)	AVG	0.0034	11.04	3.04	6.72E-05	17.21	985.11	7.39	333.30	6.13	1370.76
	SD	0.00017	0.25	0.21	1.30E-06	0.054	6.26	0.11	18.55	0.024	25.66
(2,8)	AVG	0.0027	5.13	2.54	6.82E-05	22.79	519.28	9.19	182.28	1.82	833.48
	SD	0.00017	0.026	0.16	8.37E-07	0.51	7.17	0.11	4.01	0.24	8.75
(2,16)	AVG	0.0046	3.76	5.32	6.76E-05	19.52	306.80	7.72	129.06	0.80	572.34
	SD	0.00040	0.022	0.052	5.48E-07	0.24	4.87	0.024	4.17	0.032	16.34
(2,21)	AVG	0.020	3.23	5.64	6.78E-05	19.93	273.59	8.11	131.86	0.58	529.26
	SD	0.0040	0.031	0.14	2.28E-06	0.45	4.39	0.11	6.33	0.021	21.47
(6,2)	AVG	0.0023	7.61	1.91	6.96E-05	14.05	702.09	8.05	90.00	2.34	833.70
	SD	0.00011	0.11	0.11	5.50E-06	0.32	6.15	0.19	11.85	0.19	5.58
(6,4)	AVG	0.0022	4.61	1.70	7.32E-05	17.62	397.74	7.52	76.62	2.25	512.59
	SD	5.42E-05	0.033	0.044	1.92E-06	0.19	3.53	0.074	1.05	0.056	2.51
(6,6)	AVG	0.0026	5.57	4.13	7.18E-05	19.77	334.94	7.86	83.66	1.62	463.96
	SD	6.58E-05	0.039	0.12	8.37E-07	0.40	3.29	0.19	4.42	0.027	7.00
(11,2)	AVG	0.0022	4.98	1.54	6.98E-05	14.05	414.10	8.22	35.10	1.63	482.42
	SD	5.76E-05	0.060	0.047	4.21E-06	0.33	4.92	0.22	3.94	0.0092	1.05
(11,4)	AVG	0.0024	5.23	2.32	8.08E-05	19.72	307.70	8.71	65.85	1.37	413.69
	SD	3.93E-05	0.083	0.12	8.37E-07	0.47	2.19	0.11	3.78	0.027	4.08
(23,2)	AVG	0.0026	7.11	3.79	8.36E-05	14.60	336.89	8.65	36.88	1.37	412.98
	SD	0.00012	0.22	0.14	3.21E-06	0.31	5.91	0.16	7.06	0.10	13.24

Table A2. 14: Average substep and step times for S-R, 10K characters.

(T,N)		A	P1	S	D	G	P2	RG	M	R	Step Total
(11,4)	AVG	0.0041	40.95	7.48	9.88E-05	187.70	2965.97	89.91	763.27	14.61	4102.35
	SD	0.00011	0.51	0.64	1.37E-05	1.068	44.71	0.50	36.46	1.54	68.10
(2,23)	AVG	0.0051	40.14	2.16	8.04E-05	178.79	3142.63	95.95	468.04	15.10	4360.14
	SD	9.56E-05	0.51	0.09	4.01E-05	8.11	21.45	1.18	16.47	0.077	895.10

Table A2. 15: Average substep and step times for S-R, 100K characters.

A.3 Experiment 3: Distributed vs. Non-Distributed Comparison

Results for C-R, C-C, CM10-C and S-R in experiment 3 are displayed in the tables below.

T or (T,N)	Number of Characters					
	1K		10K		100K	
	AVG	SD	AVG	SD	AVG	SD
1	186.86	2.43	2211.13	13.47		
2	124.96	3.18	1359.02	18.41		
6	50.51	0.48	517.36	2.07		
12	28.55	0.40	295.32	7.26		
24	16.48	0.46	163.36	1.09		
48	13.68	0.81	126.80	2.04	1344.67	2.35
(1,46)	16.25	0.44	145.75	4.10		
(2,21)	18.56	0.11	159.76	2.35		
(6,6)	16.96	0.15	153.04	3.24		
(11,4)	18.75	0.35	126.81	0.72	1379.80	6.71
(23,2)	48.64	0.28	129.17	6.07	1400.77	5.34

Table A3. 1: Distributed vs. non-distributed average step time (ms) comparison for C-R.

T or (T,N)	Number of Characters					
	1K		10K		100K	
	AVG	SD	AVG	SD	AVG	SD
1	185.71	0.91	1971.22	3.22		
2	115.95	2.58	1210.68	7.62		
6	48.28	1.53	441.25	3.05		
12	27.50	0.88	246.31	5.82		
24	15.54	0.24	137.34	2.70		
48	14.29	0.92	109.65	0.36	1239.99	36.18
(1,46)	20.85	0.81	189.33	3.85		
(2,21)	20.25	0.13	154.76	1.95		
(6,6)	18.02	0.54	159.53	1.54		
(11,4)	20.29	1.21	130.55	2.15	1422.86	5.34
(23,2)	46.67	0.55	153.48	3.10		

Table A3. 2: Distributed vs. non-distributed average step time (ms) comparison for C-C.

T or (T,N)	Number of Characters					
	1K		10K		100K	
	AVG	SD	AVG	SD	AVG	SD
1	123.88	0.68	1430.57	8.42		
2	82.47	1.94	841.57	7.55		
6	33.75	0.92	321.11	3.19		
12	19.47	0.31	178.15	0.63		
24	10.95	0.21	108.54	1.71		
48	8.70	0.07	85.34	0.41	983.85	3.24
(1,46)	9.45	0.28	91.77	0.54	1018.69	2.75
(2,21)	11.93	0.20	100.09	1.25		
(6,6)	12.37	0.10	103.27	1.31		
(11,4)	15.97	0.37	95.87	0.92	1099.61	1.35
(23,2)	44.20	1.88	128.40	4.31		

Table A3. 3: Distributed vs. non-distributed average step time (ms) comparison for CM10-C.

T or (T,N)	Number of Characters					
	1K		10K		100K	
	AVG	SD	AVG	SD	AVG	SD
1	564.59	1.92	5959.28	34.15		
2	376.67	7.53	4102.69	30.37		
6	140.87	4.70	1494.84	11.04		
12	80.40	2.15	792.19	1.44		
24	47.91	0.98	443.75	0.60		
48	41.38	1.08	332.72	2.95	3479.15	3.01
(1,46)	77.28	8.79	606.91	9.70		
(2,21)	63.11	3.94	529.26	21.47		
(6,6)	53.37	3.92	463.96	7.00		
(11,2)	49.08	1.88	482.42	1.05		
(11,4)	51.32	2.30	413.69	4.08	4102.35	68.10
(23,2)	66.35	1.27	412.98	13.24	4360.14	895.10

Table A3. 4: Distributed vs. non-distributed average step time (ms) comparison for S-R.

A.4 Experiment 9: Cluster

Number of Characters		A	P1	S	D	G	P2	RG	M	R	Step Total
1K	AVG	0.0085	0.44	0.58	8.52E-05	0.71	10.05	0.21	2.18	0.093	15.88
	SD	0.0036	0.059	0.20	4.71E-06	0.026	0.24	0.0081	0.41	0.0014	1.02
10K	AVG	0.011	3.87	1.50	0.00013	4.93	113.89	2.04	19.61	6.60	161.42
	SD	0.0033	0.20	0.42	6.62E-06	0.15	0.19	0.10	3.32	0.20	3.69
100K	AVG	0.012	25.95	6.18	0.00019	53.17	1249.97	25.098	307.31	65.23	1797.70
	SD	0.0022	0.57	1.44	1.61E-05	1.32	10.04	0.54	16.67	0.81	26.01

Table A4. 1: Average substep and step times for C-R on a cluster.

Number of Characters		A	P1	S	D	G	P2	RG	M	R	Step Total
1K	AVG	0.0087	0.42	0.57	8.68E-05	1.46	10.60	0.59	2.76	0.087	18.12
	SD	0.0021	0.046	0.13	5.17E-06	0.091	0.40	0.047	0.61	0.0055	1.16
10K	AVG	0.013	3.97	2.90	0.00014	14.88	120.12	7.19	32.83	5.34	197.30
	SD	0.0020	0.23	0.69	1.96E-05	0.53	1.90	0.26	4.69	0.086	7.90
100K	AVG	0.013	25.40	8.94	0.00017	150.84	979.11	73.53	171.39	60.73	1550.07
	SD	0.0017	0.45	1.26	8.79E-06	1.26	6.30	0.060	1.94	0.84	11.88

Table A4. 2: Average substep and step times for C-C on a cluster.

Number of Characters		A	P1	S	D	G	P2	RG	M	R	Step Total
1K	AVG	0.0088	0.34	0.47	8.38E-05	0.36	7.43	0.045	1.28	0.091	11.61
	SD	0.0045	0.025	0.080	1.79E-06	0.024	0.16	0.0051	0.15	0.0029	0.44
10K	AVG	0.010	2.95	1.17	0.00012	1.47	74.16	0.32	8.83	5.96	103.96
	SD	0.0034	0.15	0.31	4.67E-06	0.10	0.72	0.027	1.11	0.48	2.50
100K	AVG	0.014	23.03	4.67	0.00018	13.42	778.80	2.88	102.90	59.88	1062.71
	SD	0.0051	0.24	0.50	1.77E-05	0.28	8.53	0.088	15.03	9.98	23.74

Table A4. 3: Average substep and step times for CM10-R on a cluster.

Number of Characters		A	P1	S	D	G	P2	RG	M	R	Step Total
1K	AVG	0.0076	0.33	0.48	8.36E-05	1.01	6.72	0.36	1.38	0.085	12.01
	SD	0.0012	0.019	0.064	6.35E-06	0.064	0.16	0.023	0.21	0.0047	0.55
10K	AVG	0.018	3.032	2.03	0.00013	8.64	80.66	4.04	9.86	5.20	124.23
	SD	0.012	0.10	0.24	1.01E-05	0.14	0.66	0.068	0.97	0.83	1.89
100K	AVG	0.020	23.70	9.92	0.00019	105.01	923.49	59.33	139.98	49.92	1402.67
	SD	0.00082	0.29	1.52	7.95E-06	1.61	6.23	0.32	1.21	0.25	9.47

Table A4. 4: Average substep and step times for CM10-C on a cluster.

Number of Characters		A	P1	S	D	G	P2	RG	M	R	Step Total
1K	AVG	0.016	0.94	0.86	0.00010	2.29	31.80	0.94	10.57	0.12	78.63
	SD	0.0037	0.061	0.17	6.02E-06	0.082	0.80	0.038	2.03	0.0036	3.23
10K	AVG	0.012	4.88	2.17	0.00014	19.47	299.11	9.28	80.43	21.30	471.16
	SD	0.0011	0.14	0.41	1.62E-05	0.53	3.07	0.23	10.82	0.033	14.13
100K	AVG	0.025	26.38	6.28	0.00019	235.55	2797.87	107.81	482.90	87.73	3796.74
	SD	0.0044	0.42	1.31	2.09E-05	31.55	58.87	0.76	10.29	1.07	33.87

Table A4. 5: Average substep and step times for S-R on a cluster.

B RUNNING OUR DISTRIBUTED SIMULATION

In this Appendix, we discuss the steps and commands to run our distributed simulation. This is specifically for those who have access to the framework and project. To run, we need to use mpiexec or mpirun. For a test simulation, type on the command line:

mpiexec -n N ./Demo_ECMSimulationDistributed -i data/ENV -t TEST -c CHARACTERS -T THREADS -r STEPS

The following table explains what each argument is:

Argument	Meaning	Example
-n N	number of nodes	-n 9
-i data/ENV	environment	-i data/city
-t TEST	test ID	-t 0
-c CHARACTERS	number of characters to add	-c 10000
-T THREADS	threads for OpenMP	-T 2
-r STEPS	number of steps the simulation runs	-r 500

Table B. 1: Command line arguments.

Note that for N, we need to include 1 node for the manager. So '-n 9' translates to 1 manager and 8 working nodes. We can also use scenarios with the '-s' argument. There are a few tests included, each with a different ID; see Appx. B - Table 2. An example of a valid command line would be:

mpiexec -n 9 ./Demo_ECMSimulationDistributed -i data/city -t 0 -c 10000 -T 2 -r 500

Test ID	Test Description
0	Random scenario: Characters are randomly added (random positions, layers and goals).
1	Characters are added to form a circle shape, with goals on opposite ends.
2	Characters are added to form a square shape, with goals on opposite ends.
3	Characters are randomly added as groups (random positions, layers and goals).
4	Characters are added to one of two spawn area, with goals in the other spawn area.
5	Crossing scenario: Characters are periodically added on the left with goals on the right.
100	Same as test ID 0, but with comparison to the non-distributed version.
101	Same as test ID 1, but with comparison to the non-distributed version.
102	Same as test ID 2, but with comparison to the non-distributed version.
103	Same as test ID 3, but with comparison to the non-distributed version.
104	Same as test ID 4, but with comparison to the non-distributed version.
105	Same as test ID 5, but with comparison to the non-distributed version.

Table B. 2: Simulation Tests.

'With comparison' means that the results/attributes of all characters at each step is compared to those produced by the non-distributed version of the simulation. This means that we run two simulations in parallel: The manager sends actions to all nodes then performs a single simulation on its own. Upon receiving node updates, the manager compares the position, layer and velocity values for all characters. If there's a single discrepancy between the two, it will notify us. This is useful to test the correctness our our distributed simulation, since simulation results are deterministic.

Also note that appending a 6 to a test ID will run that test using the original non-distributed simulation, which has its own test ID (shares a lot of common test IDs with the distributed version). This runs the non-distributed simulation without visualization. For example, '-t 60' will run the random test on the non-distributed version. Note that when running with this test, do not use MPI and mpiexec.

On a final note, you can refer to www.mpitutorial.com/tutorials/running-an-mpi-cluster-within-a-lan for setting up an MPI cluster. Setting it up SSH between nodes should be straight forward; this is done so that nodes can access each other via SSH without login passwords. After that, NFS-Server and NFS-Client should be installed, which are used for nodes to mount shared directories.