Sapienza
Università di Roma

Faculty of Information Engineering, Informatics and Statistics

Master's degree thesis in

ENGINEERING IN COMPUTER SCIENCE

ORCHESTRA
An Asynchronous Wait-Free Distributed GVT Algorithm

Tommaso Tocci

Academic Year

2016/2017

Supervisor

Prof. Ciciani Bruno

Advisor

Ing. Pellegrini Alessandro

Co-advisor

Prof. Salza Silvio

Contents

Abstract

Taking advantage of computing capabilities offered by modern parallel and distributed architectures is fundamental to run large-scale simulation models based on the Parallel Discrete Event Simulation (PDES) paradigm. By relying on this computing organization, it is possible to effectively overcome both the power and the memory wall, which are core limiting aspects to deliver high-performance simulations. This is even more the case when relying on the speculative Time Warp synchronization protocol, which could be particularly memory greedy. At the same time, some form of coordination, such as the computation of the Global Virtual Time (GVT), is required by Time Warp Systems. These coordination points could easily become the bottleneck of large-scale simulations, hindering an efficient exploitation of the computing power offered by large supercomputing facilities. In this dissertation is presented ORCHESTRA, a coordination algorithm which is both wait-free and asynchronous. The nature of this algorithm allows any computing node to carry on simulation activities while the global agreement is reached, thus offering an effective building block to achieve scalable PDES. The general organization of ORCHESTRA could be adopted by different high-performance computing applications, thus paving the way to a more effective usage of modern computing infrastructures.

Part I

Optimistic synchronization strategy

Chapter 1

Commitment horizon

1.1 Fossil collection

One of the main issues with speculative/simulation software turns to be memory management [5], in fact these platforms are usually characterized by a very high memory demand. During the actual simulation, a lot of data is produced and tons of intermediate states are reached. Given that the memory capacity offered by the underlying hardware machine is limited, it is crucial to keep only the essential data that are useful to support the speculative part of the simulation and discard the committed states, which are no longer necessary. In most cases a certain data will be of interest just for a definite period of time. This is the time in which this specific piece of information needs to be accessed in order to produce other future simulation states. Now, it should be clear, that the memory reclaim of obsolete objects residing in main memory will play an important role in the simulation and will have a big impact on the overall performance of the platform. The production rate of intermediate data should be balanced by the deletion rate of the obsolete data, in fact, if the fossil collection is too slow compared to the generation of new data, we'll have a constant increment of the residing memory set, that on long running simulations, will lead to an out of memory condition. On the other hand if the fossil collection procedure is invoked too frequently with respect to the advancement of the LP, most of the computing power will be spent in searching fossil objects instead of producing useful data for the simulation, producing the well known thrashing phenomena [6].

Memory management get even more challenging when dealing with simulation platform that follow the Time Warp approach. Indeed, the fact that the simulation trajectory is divided into a speculative part (which can be subject to rollbacks) and a committed part (**??**) make it difficult to define which of the events and states have became obsolete, i.e. they cannot be involved in a rollback operation. Every LP, in reaction to an out-of-order event, will perform a roll-back to an earlier state in order to reprocess the last events in the correct time order. This demands for the availability of previously-executed events and intermediate simulation states (in case of rollback operations supported by checkpointing). The challenging question related to the memory management of Time Warp systems is: *How many already processed events do I need to keep in memory?*. From a practical point of view, this question can be reformulated and is equivalent to: *How far an LP can roll-back?*.

1.2 Global Virtual Time

It is always possible to define a commitment horizon, that is a virtual time instant in the simulation's trajectory, for which we are sure that none of the LPs will rollback prior to this specific time. Assuming this, all the states and event that are associated with a time stamp less than the commitment horizon won't ever been accessed again, and can be considered fossilized. This commitment horizon has been named by Jefferson as the Global Virtual Time (GVT) [11].

Definition 1. At a specific real time t, the GVT can be defined as the minimum between:

- all virtual times in all virtual clocks at time t.
- timestamps of all sent but not yet processed events at time t.

In figure 1.1 is depicted a 3-LPs simulation snapshot at a specific time t. The time-lines of the three LPs are populated with the events, represented by rectangles. The grayed events residing on the left part of the picture have been already executed, the black ones on the right are still pending, while the orange ones are the last executed event of each LP. The 3^{rd} logical process (LP$_3$) is the one with the smallest virtual clock among all, thus, following the Jefferson definition is the one marking the current Global Virtual Time (purple line). In order to give an

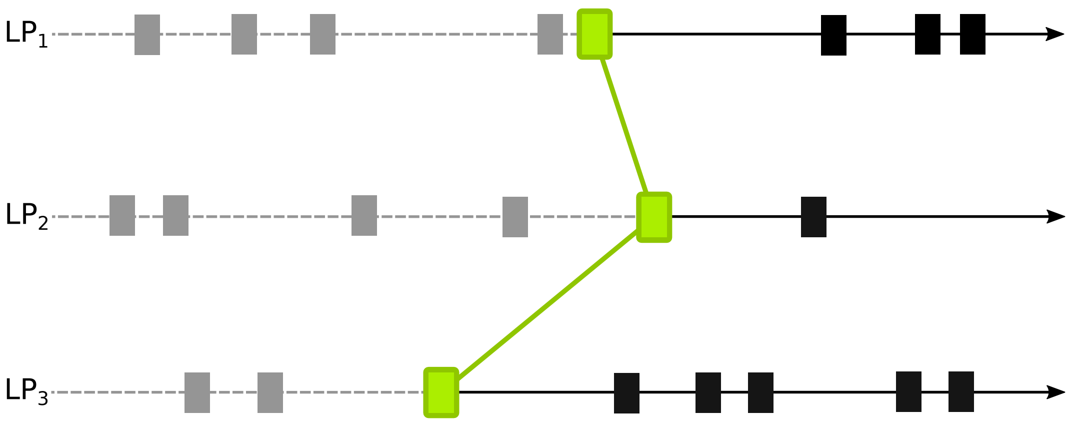


Figure 1.1: Global Virtual Time definition

intuition about why an LP cannot ever rollback prior to the GVT, without loose of generality, we will focus just on the first clause of the definition; The second clause is intended to cover a special case known as the *transient message* that will be analyzed in section 1.3.1. Let's consider the worst case scenario depicted in fig-

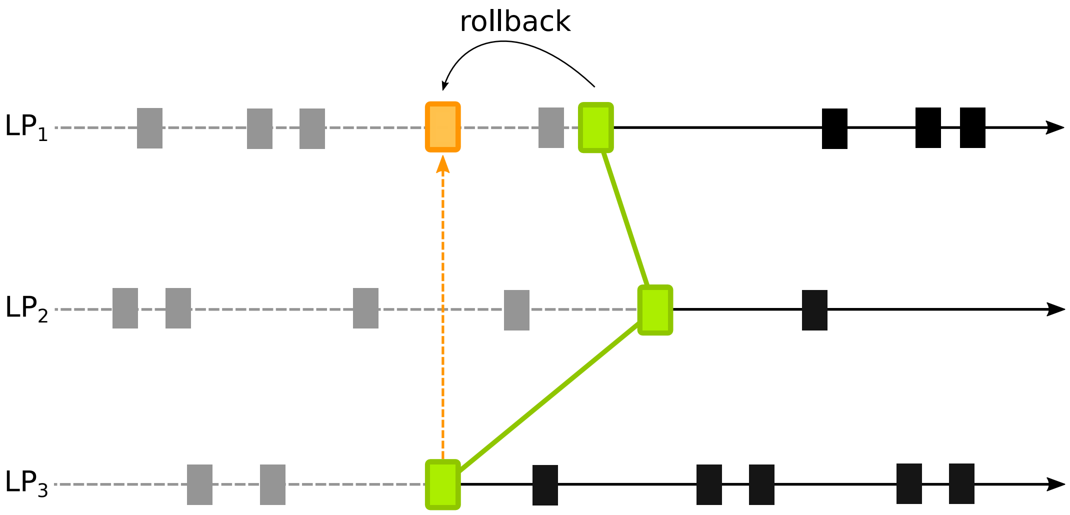


Figure 1.2: GVT definition: Rollback worst case scenario

ure 1.2; The first logical process LP_1 receives an event from the furthest one LP_3, with the smallest possible time stamp. Since an LP can only schedule events with a time stamp that is greater or equal to its own virtual time, the time stamp for the orange event coincides with the current virtual time of LP_3 that is also the current GVT. Thus the orange event is the event with the smallest possible time stamp that can be generate at the moment. When the logical process LP_1 will receive the orange event, it will notice that is out of order and will perform a rollback to the Global Virtual Time and will re-execute all the events starting from the orange one. Thus in the worst case an LP can rollback at most to the current Global Virtual Time.

Now it should be clear that none of the objects residing further back then the GVT would never be accessed and they can be deleted on the next fossil collection schedule. Actually, as explained in section **??**, due to performance issue, in the worst case the LP will rollback to the last saved checkpoint before to the current GVT and will re-process all the subsequent events. Thus, being more precise, we can assume that:

Statment 1. None of the events with a time stamp less or equal to the last saved checkpoint before the current GVT will be ever accessed again.

Regarding the GVT definition, we report now an alternative formulation proposed later by Fujimoto [8]:

Definition 2. Global Virtual Time at time r is defined as the minimum time stamp of any unprocessed events in the system at real time r.

Since the virtual times of the LPs are directly related to their last processed events, Fujimoto decided to just focus on events' time stamp, thus ending up with a compact definition that includes both the two clauses of Jefferson (Definition 1).

In the last sections it has been explained the importance of the GVT, that is one of the fundamental pieces of the Time Warp synchronization protocol, because it allows to contrast and overcome the huge memory requirements of this optimistic approach. In general the Global Virtual Time can be see as a global information that tracks the evolution of the simulation, it is the commitment horizon that marks the boundary between the unsteady work that is temporary and can be potentially rolled back and the certain work that has been definitely committed and cannot be modified anymore. Thus the GVT can be taken as reference point for a wide range of tasks, for instance, non-reversible operations such as the interaction with external output devices [1] (like log, display and so on) should be perform only on the basis of definitive data. Moreover the termination detection routine aimed at verifying that specific termination condition have been reach is another operation that can be carried on only on committed data.

1.3 GVT computation algorithms

Calculating the GVT is all but an easy task. Especially on distributed environment it often requires to put in place cooperation and consensus algorithms. In fact this operation is the only global one required by the Time Warp synchronization protocol. All others, such as rollbacks, state saving, and sending and handling of anti-messages, can be carried out locally. Therefore, GVT computation is known to be the least scalable component of Time Warp and it is no surprise that the accuracy and overhead of the GVT computation may dominate the overall performance of Time Warp-based simulations.

Following its definition, in order to compute the exact Global Virtual Time, it would be necessary to collect the virtual times of all the LPs and the time stamps of all transient messages at a specific real time r, this would require to freeze the entire simulation during the GVT calculation. It is obvious that this kind of approach would make the Time Warp optimistic methodology useless from a performance point of view.

Fortunately it is not mandatory to compute the exact Global Virtual Time, in fact a lower bound of it, would be good enough to carry on fossil collection and all the others GVT related routine. During fossil collection all the events residing between the estimated GVT and the exact one, would not be discarded, but this indeed won't have a significant impact on the memory usage, as long as the lower bound is tight enough.

A lot of algorithms aimed at calculating the Global Virtual Time in Time Warp based simulator have been proposed by the scientific community along the past years; while reviewing some of the most interesting solutions among those, we will analyze two very common problem related to the GVT calculation, that are known under the name of *Transient message* and *Simultaneous reporting* [18]. Anyway, for a brief review and a well done comparison of GVT algorithms the reader is suggested to read the 2^{nd} section of [4].

1.3.1 The transient message problem

Starting from the Jefferson's GVT formulation (Definition 1), it is possible to build a very simple and inefficient GVT algorithm that will help us to understand the problem of the transient messages and the importance of the second clause. Re-

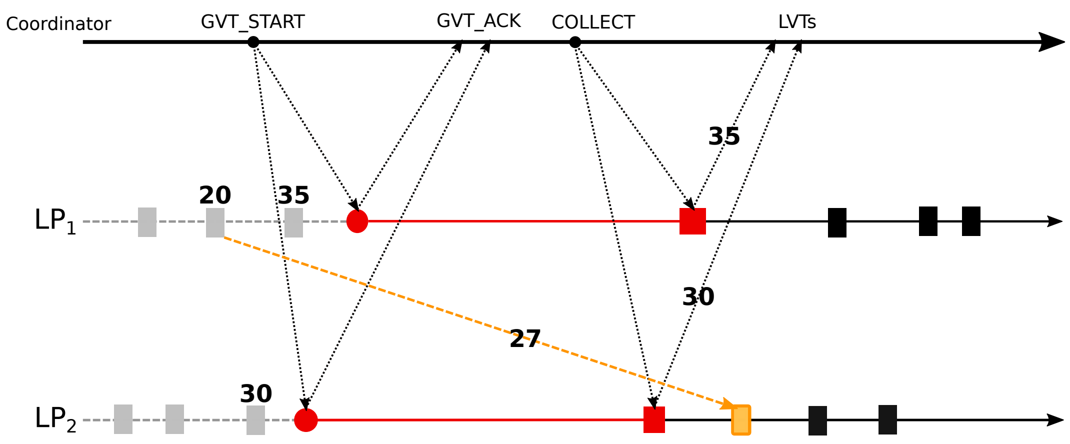


Figure 1.3: Example of transient message problem that justify the second clause of the Jefferson's definition

capping, we would like to stop all the LPs from processing events, reports their current virtual time, calculate the minimum among those values and restart the simulation activity. Let's suppose that an external coordinator is in charge of orchestrating the GVT routine and the collective procedure by communicating with all the LPs. The algorithm at each invocation will follow the next steps:

- The coordinator starts the GVT routine by broadcasting a `START_GVT` message.
- An LP that receive the `START_GVT` message, stops processing new events, enters into the *GVT* phase and send a `GVT_ACK` to the coordinator.
- As soon as the coordinator collects all the acknowledgments from the LPs it broadcast a `COLLECT` message.
- The LP receiving a `COLLECT` message, sends to the coordinator its *local virtual time* (*LVT*), will exit from the *GVT* phase and will restart processing events.
- The coordinator computes the new *GVT* value by calculating the minimum among all *LVT*s received and broadcasts the new global value to all the LPs.

As we can see in the example run depicted in Figure 1.3, the two LPs, after they get notified by the *coordinator*, and they both suspend their execution, they send their current *LVT*, that is the virtual time of their last correctly executed events.

The two logical processes LP_1 and LP_2, send respectively, 35 and 30, thus making the coordinator choosing the lower one as the new GVT. The problem here is that LP_1 during the execution of the past event with virtual time 20, has scheduled an event to the second LP at time 27; due to the latency of the underlying communication network this *transient message* will arrive at LP_2 only after the GVT round has been terminated, thus making impossible to account it in the calculation of the Global Virtual Time.

The second clause of the Jefferson's *GVT* definition (1), refers exactly to this scenario, in fact it remarks the importance of including the virtual time of ongoing messages, while calculating the lower bound of the new *commitment horizon.* The *transient message* problem is related to concept of *message non-observable Time Warp systems* [8], that are those systems in which messages could possibly "disappear" while they are in transit. Indeed, after a message is created at the sender and it is passed to the underlying communication network for the delivery, it enters into a "black window" in which it became non-observable, until it will be actually delivered at the destination LP. The key solution to this problem is not to loose metadata of messages while they are passing through this black window; two options arise, either the sender or the receiver will be in charge of keeping track of time stamps of in-transit messages, in such a way that they can be accounted while reporting the local minimum (*LVT*).

Since the message is created at the sender, a very simple schema based on message acknowledgments can be used to overcome the *transient message* problem [3, 7, 18]. Each message or anti-message is accounted by the sender until the matching acknowledgment will arrive from the receiver. The problem of this approach is that introducing acknowledgments, has the huge drawback of duplicating the number of overall exchanged messages between the LPs, with the possibility of overloading the communication channels. Even if the underlying network protocol actually make use of message acknowledgments (e.g. TCP), they are usually invisible from the application level. Some optimizations have been proposed in order to reduce the network overhead, such as piggy-backing [2] the acknoledgments or the implementation of sequence numbers [14]. Another drawback of the message acknowledgment scheme is quite subtle. It is not a trivial task to find out the earliest time stamp among unacknowledged messages. Such an operation is not constant time and may require the use of a priority queue.

1.3.2 Simultaneous reporting problem

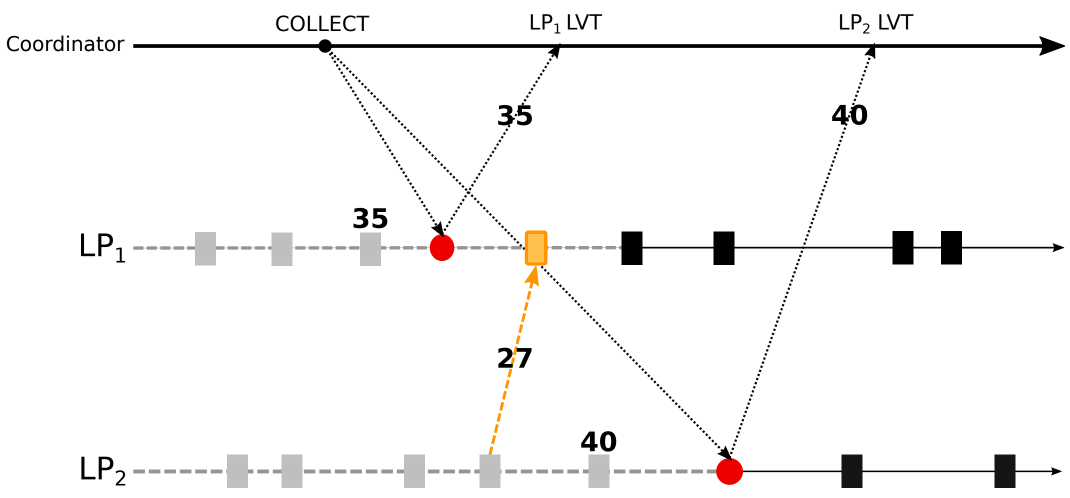


Figure 1.4: Simultaneous reporting problem

The algorithm presented in the previous section has the main drawback of requiring logical processing to stop processing events while they are participating to the GVT computation (red line section of figure 1.3). The length of this strike is usually correlated with both the total number of LPs participating into the simulation and the performance of the communication channel between them and the coordinator. In fact, a lot of LPs on a slow communication network, will delay the arrival of all the `GVT_ACK` messages, with the consequence of shifting forward as well the start of the $LVTs$ collection step. It should be clear that this algorithm on distributed platforms that cannot relay on fast communication channel such as shared memory, will have a disastrous performance impact.

Let's try to analyze another a very simple algorithm in which the LPs do not need to wait each other in order to report their own local minimum:

- The coordinator starts the GVT routine by broadcasting a `START_GVT` message.
- An LP that receives the `START_GVT` message, calculates and sends to the coordinator its own local minimum.
- As soon as the coordinator collects all the $LVTs$ from the logical processes, it chooses the lower one and it broadcasts it as the new GVT.

The avoidance of global synchronization points introduces another issue known as the *Simultaneous Reporting Problem*, that occurs when a message exchanged between two $LVTs$ report of two different LPs it is not accounted by either of them. In Figure 1.4 an example run characterized by this flaw is depicted. The first process LP_1 execute the event at 35 and as soon as it receive the `COLLECT` message from the coordinator it replies with its local minimum that it happens to be exactly 35. Later on LP_2 will also participate with its own LVT that is equal to 40. The problem here is represented by the orange message that has been sent by LP_2 to LP_1 after the latter has reported its minimum but before the former has sent its own. It should be clear from the Figure 1.4 that the time stamp of the orange message 27, won't be accounted for the calculation of the new GVT even if at the moment (freezing wall-clock time of the picture) results to be the unprocessed event with the lowest time stamp.

1.3.3 Samadi's algorithm

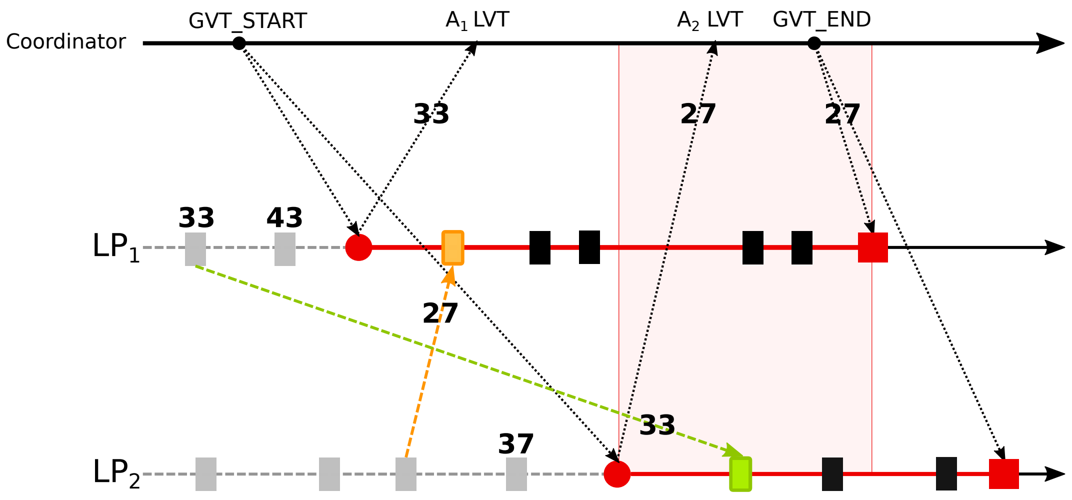


Figure 1.5: Samadi's GVT algorithm

The algorithm proposed by Samadi [18] stands as one of the first GVT calculation approaches that manage to solve both the *Transient message* (1.3.1) and *Simultaneous reporting* (1.3.2) problems, moreover it is the reference for a larger group of algorithms known as ”Overlapping Intervals“ [4]. The algorithm obeys to the following steps:

- The coordinator starts the GVT routine by broadcasting a `START_GVT` message
- When a process receives the `START_GVT` message, it enters into the red phase and calculates the minimum time stamp among:
    - unprocessed events in his incoming queue
    - unacknowledged sent messages
    - sent messages that were red acknowledged after the end of the previous GVT round

  And it immediately reports this value to the coordinator.
- As soon as the controller receives $LVTs$ from all the processes, it computes the minimum of all these values as the new GVT and broadcasts the new GVT to all the processes.
- When a process get informed about the new GVT value, it leaves the red phase.

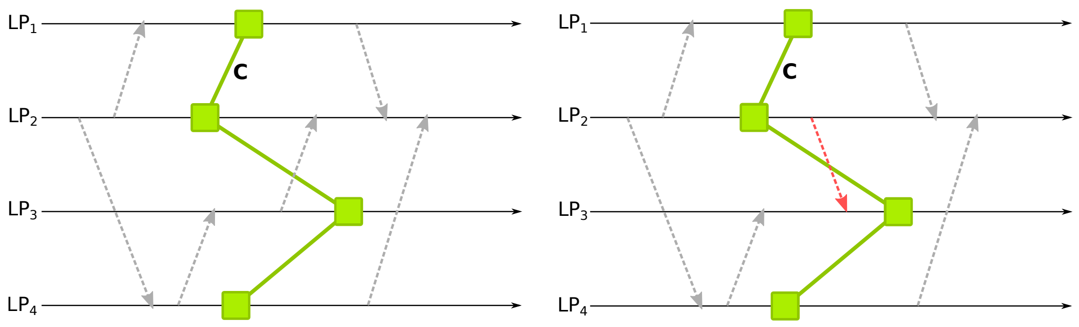
The algorithm encompasses that acknowledgments are sent for all received messages, moreover the acks need to be red marked if the process that is sending them is in red phase, otherwise the acks would be unmarked. By implementing the acknowledgment schema, Samadi managed to solve the *transient message* problem. In Figure 1.5 it is depicted an example run of this algorithm. The green message sent by LP_1 is a transient message because is still in transit while both the processes are calculating and sending their $LVTs$; At the time process LP_1 calculates its local minimum, the green message has not been delivered yet and, of course, neither the acknowledgment for it have been received, thus LP_1 needs to account it, making him electing 33 as its current LVT.

On the other hand, the differentiation of marked and unmarked acknowledgments, make it possible to solve the *simultaneous reporting* problem. Messages sent and acknowledged between two $LVTs$ reporting will be correctly accounted in the calculation of the new GVT. Let's suppose, as depicted in Figure 1.5 that the first process acknowledges the orange message after it has been reported his local minimum to the coordinator, since it is still into red phase, the acknowledgment that it will send will be red marked. Process LP_2 receive the ack for the orange message before to start calculating its minimum; since the ack it received is red marked, it is in charge for accounting it in the current *GVT* computation. In fact even if the message has been acknowledged it would be impossible for the receiver LP$_1$ to include it in its *LVT* report, because it has already sent the report back to the coordinator.

Samadi has demonstrated [18] that his algorithm computes a new GVT value that is no larger than the true GVT value at the instant the controller broadcast the `START_GVT` message.

1.3.4 Mattern's algorithm

Friedemann Mattern in 1993 has revolutionized the Time Warp based simulation ecosystem, by proposing an asynchronous algorithm [15] that does not requires message acknowledgment, it opened a way to a completely new family of algorithms based on the *two cuts* approach. The Orchestra GVT algorithm presented in this dissertation actually belong to this family too.



(a) Consisten *cut* (b) Inconsisten *cut*

Figure 1.6: *cut* definition

The notion of *cut* is the key point of Mattern's idea. Referring at Figure 1.6, a *cut* can be thought as a broken line that separates the time diagram into two parts, the past side at the left and the future side at the right of the line. A cut is said to be *consistent* if no message arrow starts in the future and ends into the past. Figure 1.6a shows a *consistent cut* while an inconsistent one is depicted in Figure 1.6b.

The important property of a *consistent cut* C is that the events around it can be arranged in such a way that it is possible to draw another straight line C' at wall-clock time T that equivalent to C in the way that it separates both the send and arrival events in the exact same past and future sets. This transformation can be performed because it is not relevant the wall-clock time of the events, rather are the relation between them from a virtual time point of view. For a more formal argumentation the reader is invited to read the Mattern's original paper [15]. Since a *consistent cut* C is equivalent to its straight version C' at time T, can be easily use to calculate a global characteristic of the system such as the GVT, without freezing the simulation.

Definition 3. Given a consistent cut C and its straight version C' at wall-clock time T, a lower bound of the real GVT at time T can be calculated as the minimum among:

- virtual times of all the processes at the cut point of C
- time stamp of all messages crossing C from past to future

Starting from this definition, it is possible to construct an algorithm to calculate a GVT approximation in Time Warp based system. The first problem that arises is the necessity for the cut to be *consistent*, as a matter of fact, we cannot ensure that there will be no messages crossing the cut from the future to the past. Fortunately as demonstrated by Mattern [15], since the Global Virtual Time is a monotonic function over the wall-clock time of the simulation, we can simply discard all messages that make C inconsistent. Indeed, these messages are sent from the future (the right side of the cut) to the past (the left side), meaning that some process has sent them after its cut point. Given that, after its cut point a process can only roll-back to a value grater then the GVT calculated over C, it implies that those messages can only have a value grater then the GVT and won't influence the calculation at all, thus they can be discarded. Thus is possible to construct a GVT algorithm based on the definition 3 with the relaxation that the cut needs not to be a *consistent one.* The only implementation obstacle is finding a way of determining the set of transient messages of the cut without using acknowledgments. The idea is based on the construction of two consecutive cuts C_1 and C_2 as depicted in Figure 1.7; after the processes participate to the first cut C_1 they will enter into a red phase (region on the diagram between C_1 and C_2) and they will start keeping track of all sent messages. This set is composed by all those messages that have been

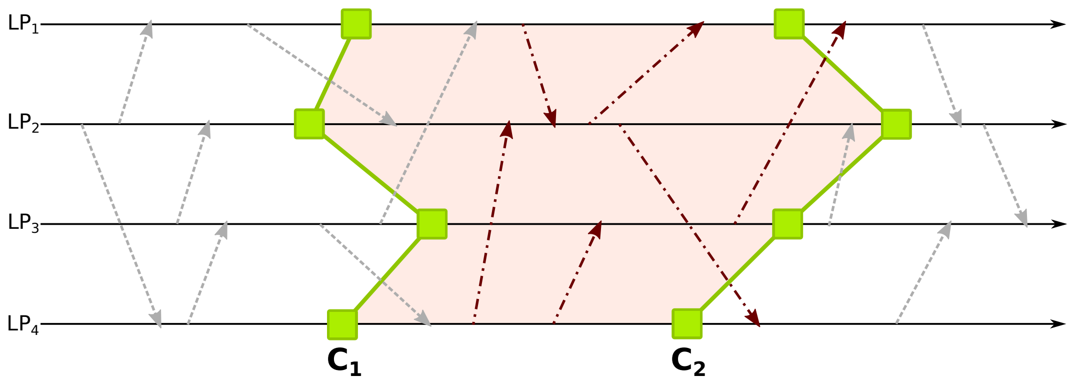


Figure 1.7: GVT algorithm based on the *two cuts* approach

sent across LPs between their two cut points, that in Figure 1.7 are characterized by a red arrow starting from the intermediate red zone. The second cut C_2 will be constructed in such a way that there would not be messages crossing both the cuts, that graphically means that we are trying to avoid that there would be arrows starting before the first cut and ending after the second cut. Thus C_2 can be taken after that all transient messages crossing C_1 have been successfully delivered at the destination process. With the help of the first cut, is thus possible to construct the set of red messages, that is actually a superset of the red messages crossing C_2 from the past to the future. The minimum time stamp among all red messages is a lower bound on the minimum time stamp of all transient messages crossing C2.

Practically every process LP$_i$ needs to keep track of the following variables:

- TU_i: minimum time stamp among all unprocessed events at i^{th} process
- TR_i: minimum time stamp among all red messages sent by LP$_i$
- WS_i: vector holding at n^{th} position the number of white messages sent from LP$_i$ to LP$_n$ before the first cut
- WR_i: number of white messages received by LP$_i$

To construct the first cut C_1, a control message is exchanged among all the processes in a ring fashion; the control message carries a vector V used to accumulate the total number of white messages that have been sent to each process before C_1; when a process receives the control message during the construction of the first cut, it first enters into the red phase, updates the vector V by summing to it its local vector WS_i and then it passes it to next process in the ring. Therefore at the end of the first complete communication across the ring the vector V will contain at n^{th} position the total number of white messages sent by all others to process LP_n before C_1.

As already mentioned, in order to construct the second cut C_2 it is necessary that all the in-transit white messages have been delivered. The second round can actually start immediately, but this time a process LP_n will pass the control message to the next one in the ring, only after it is sure that it has correctly received all white messages destined to it; formally this condition will be reached as soon as $WR_n = V[n]$. Indeed, every LP will move again to the white phase when passing the control message. Moreover, along with the vector V, during the second round the control message will carry also a time stamp value T that the processes will use to perform a minimum reduction over their LVT. Therefore every process LP_n will update T as follows:

$$T = min\{T,\ TU_n,\ TR_n\} \tag{1.1}$$

At the end of the second complete communication across the ring, when also the second cut C_2 is closed, T will store the minimum time stamp among all unprocessed event (TU) and all red messages (TR) in the system, that by definition 3 is exactly the new GVT value over the second cut C_2.

1.3.5 GVT in shared memory systems

Almost twelve years ago, around 2003, the IT world has experienced a very big turn due to the power-wall phenomenon [19]; the processors manufacture industry stopped pursuing a higher driving clock speed and instead they started focusing and developing massive hyperthreading and multicore architectures. Similarly to all IT fields, also the simulation ecosystem have tried to adapt to this change by developing innovative techniques to exploit the new possibilities offered by these new promising architectures. In particular, shared memory capabilities provided by all modern platforms can be used to implement very efficient inter-process/inter-thread synchronization strategies and more generally to drastically reduce the communication cost. In this specific context the actions of each LP are usually carried out by a specific thread bounded to a specific processing unit of the underlying hardware platform.

Fujimoto's algorithm

Fujimoto [8] has been the first one to take advantage of these new features, using them to achieve better performance into Time Warp based simulation frameworks. He demonstrated how, by exploiting shared memory communication primitives, it is possible to build an *observable systems* [8] in which there cannot ever be transient messages. In fact, since the send operation is implemented through a write operation on the memory portion visible to both the sender and the receiver, the in-transit time of the message is actually zero, because, as soon as the sender has written the message into shared memory, it becomes immediately visible to the receiver. We report here the original Fujimoto definition of the GVT computation:

Definition 4. A Time Warp system is said to be message observable if at any instant in time, each unprocessed message in the system can be observed by at least one processor, and the observability of a message by a processor does not change without some explicit action by some processor in the system.

Given that observable shared memory Time Warp systems does not suffer form the *Transient message* problem (Section 1.3.1) by design, Fujimoto was able to design a much easier and optimized GVT algorithm. A global shared counter is used to notify all the threads about the start of the new GVT round; instead of sending a broadcast message, the coordinator simply sets the counter to the total number of threads. As soon as a thread becomes aware that a new GVT round has been started (the counter is greater then zero), it reports his current LVT into a shared array and decrements the counter by one. The last thread that updates the LVTs is in charge of calculating the global minimum and storing it in another shared variable.

Even if the update of the shared counter is visible to all the processes immediately (without any network delay), and they check it at every simulation loop, still every thread would reach the test instruction at different wall-clock time instants from the others, because their execution of the main simulation loop it is not synchronized. Indeed, the *Simultaneous reporting* problem (Section 1.3.2) could potentially occur. Let's imagine that, after the counter is set, a thread process an event and send some messages before to actually check the counter, moreover let's suppose that one of these messages are destined to a thread that have already reported his LVT into the shared array. In this case the sender is the only one that can account for that message. Fujimoto solved the issue by ensuring that the shared counter check is performed at the end of the main loop, after any send operation and that the time stamp of the last executed event its accounted by every thread in its local minimum.

Fujimoto's GVT algorithm [8], still has some scalability issues, due to the usage of a big critical section; the instructions responsible for the update of the shared array and the decrement operation of the shared counter, are executed into a critical section to ensure exclusivity access of those variable residing into the shared memory portion. The mutual exclusion code snippet could really affect negatively the performance of the simulation, by having the threads stop processing events while waiting their turn to contribute to the current GVT round.

Pellegrini's algorithm

Trying to achieve a very high degree of parallelism and reducing as much as possible blocking synchronization code ,Pellegrini and Quaglia, come up with another GVT algorithm [16] targeting tightly coupled shared memory systems.

Their idea was to use *Compare and Swap* (CAS) primitive [10] in order to prevent the critical section of the previous algorithm. In particular, they used a wait-free implementation of the atomic counter used to synchronize the threads on the GVT round, and manage to remove the critical section around the code in charge of updating the shared array holding the LVTs. Indeed, their proposal still targets *message observable* systems, but their implementation of the data structures used to store incoming messages make actually possible to temporary lose visibility of some of those messages. Thus from a software point of view they cannot actually rely on this property to ensure correctness of their algorithm, this is the cost paid to have a non-blocking implementation of the receive operation (Section 2.5.4).

The impossibility to count on message visibility properties in conjunction to the necessity of avoiding the *simultaneous reporting* problem (Section 1.3.2), led them to construct an algorithm based on multiple phases, in a very similar way to the one proposed by Mattern (Section 1.3.4).

As depicted in Figure 1.8, all the threads participating to the GVT computation pass trough three different phases (`phase A` , `phase Send` , `phase B`). The start of the round, at wall-clock time t_1, is made instantaneously visible to all the threads by atomically setting a special `GVT_flag` to `true` . On the other hand, the con-

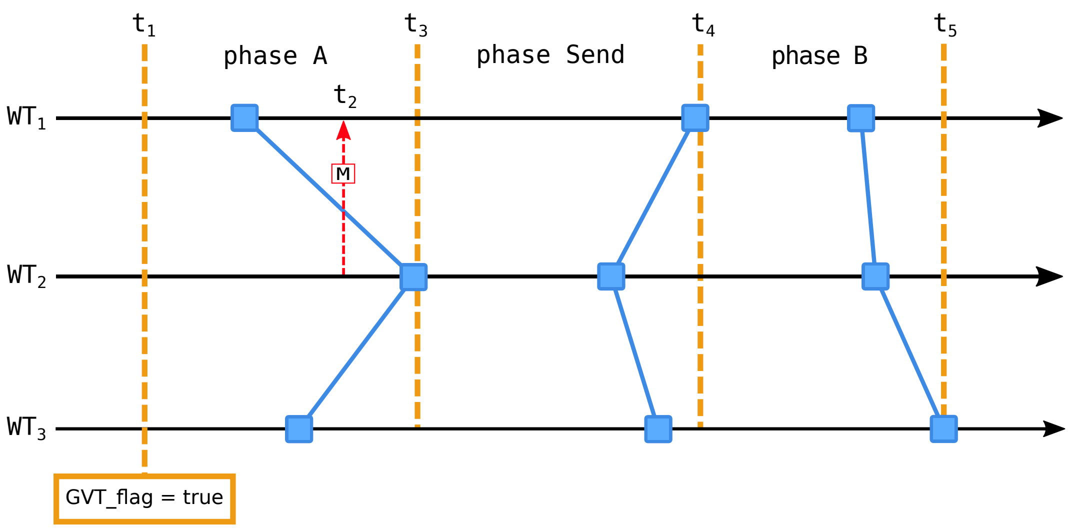


Figure 1.8: Wait-free GVT algorithm on shared memory systems

clusion of the different phases on different threads can occur at different instants of wall-clock time as explicitly shown in Figure 1.8 (blue lines). No successive phase can be entered by any thread unless all the other worker-threads have left the previous phase, thus the actual system- wide end of a phase is defined by the time at which the last thread leaves from that phase, depicted in the picture as dotted orange vertical lines.

Each worker-thread WT_i computes its local minimum two times, hence determining two values min_i^A and min_i^B, respectively on `phase A` and `phase B`. During the intermediate `phase Send`, every thread is requested to process at least one pending event, if any, and to send newly scheduled events produced during such processing phase towards the destination worker threads. `phase Send` starts right after all the worker threads ended their tasks related to `phase A`, for the example in Figure 1.8 this occurs at wall-clock-time t_3. Given the intrinsic sequential nature of the activities carried by each thread and system observability, at the end of `phase Send` (at time t_4) it is guaranteed that every message sent during `phase A` has been already incorporated into the destination data structure (namely the incoming message queue of the destination worker-thread). Thus the second local minimum calculated on `phase B` has the goal of accounting for all those messages sent before `phase A` that have not accounted by during the calculation of the first local minimum, hence, min_j^B represents the lower bound on the logical time value that can be affected by WT_j when also considering incoming information after `phase Send` is over.

For example consider again Figure 1.8, in which the message m is sent at wall-clock time t_2 from WT_2 to WT_1 after WT_1 already computed min_1^A, but before WT_2 computes min_2^A. The timestamp of this message would therefore be missing in the global reduction. However this timestamp will be accounted by having the worker-threads computing second minimum min_*^B when it is guaranteed that any message sent by some worker thread up to the end of `phase A` have become observable.

Therefore, $min(min_i^A, min_i^B)$ is the absolute lower bound (LVT_i) on the logical time value that can be affected by the generic worker thread WT_i after `phase Send` is over. The algorithm terminates by having each worker-thread writing its LVT_i into a shared array and then computing the absolute minimum across all these values to determine the new GVT.

Part II

The ORCHESTRA algorithm

Chapter 2

Simulation platform

The ORCHESTRA distributed GVT algorithm has been implemented and tested into the **RO**me **O**p**T**imistic **Sim**ulator (ROOT-Sim), a general purpose platform oriented to Discrete Event Simulation DES) and based on optimistic synchronization. ROOT-Sim's development started as a research project late back in 1987, and is currently run by the *High Performance and Dependable Computing Systems* group at the *Dipartimento di Ingegneria Informatica, Automatica e Gestionale*, Sapienza, University of Rome. The platform is completely Open Source [1] and it is developed using C/POSIX technology.

The simulator comes as a static library which can be linked to executables implementing simulation models using the ANSI-C programming standard [13], in particular the compilation process produces a binary, namely `rootsim-cc`, that is a wrapper of the GNU Compiler Collection (GCC) [2] that can be used by the model developer to compile the model source code, link it to the simulation library and produce the final executable model binary.

2.1 Simulation entities hierarchy

Computing clusters have become bigger and bigger, often composed by heterogeneous architecture, they can reach thousand of teraflop per second through the cooperation of millions of processing unit. Exploiting in the most fruitful way

[1]ROOT-Sim source code @ Github - `https://github.com/HPDCS/ROOT-Sim`

[2]GNU Compiler Collection (GCC) - `https://gcc.gnu.org/`

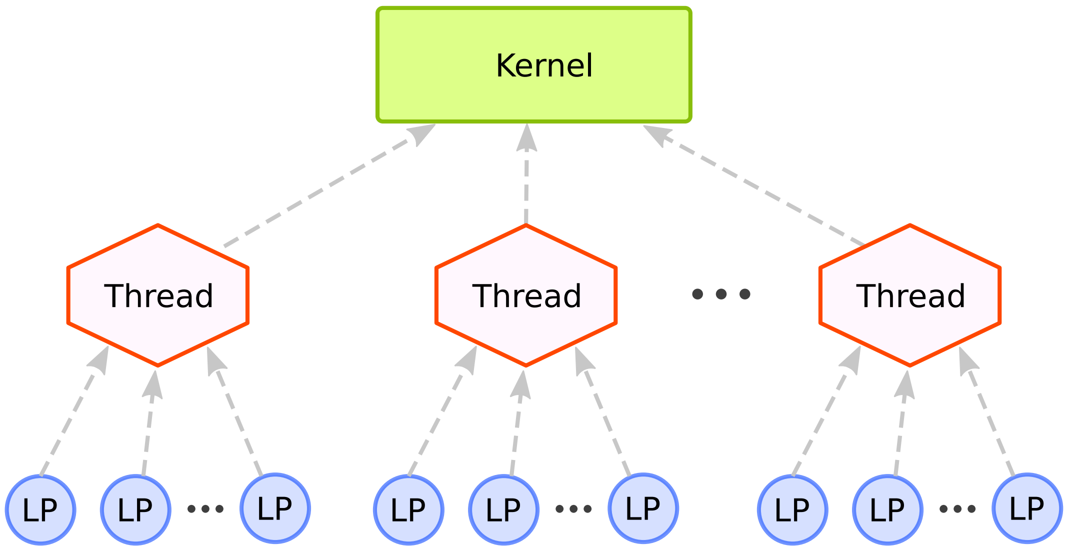


Figure 2.1: Hierarchical tree structure organization of the simulation entities

the computation capacity of the underlying hardware platform is one of the main goals that has been pursued during the development of ROOT-Sim. In order to support modern supercomputers' architecture, the simulation platform has been logically divided into simulation entities organized according to a hierarchical tree structure composed by 3 different levels.

Logical process As depicted in Figure 2.1, at the bottom of the tree there are the logical processes. These are the direct representation of the entities described by application model, the actual ones to be simulated. Each of them have its own event queue and it is characterized by a **L**ogical process **ID**entifier (LID). The events can be sent from one LP to another and always they will be executed on behalf of a logical process.

Thread LPs are grouped together under a common simulation thread. From the point of view of the operating system, a simulation thread is actually a process running on a specific physical core unit. It acts as a scheduler for the LPs it is managing and it is in charge of dividing the computing power of the physical core among them. At each instant of wall-clock time one simulation thread will execute on one core the simulation activities specific to one LPs in its group. Every thread is characterized by a **T**hread **IDentifier** (TID) and will manage a group of LPs denoted as $LPSet_t$ where t is the thread ID.

Kernel Multiple threads can be grouped under the same simulation kernel that represents one physical machine. The threads residing on the same kernel are denoted by $TSet_k$ where k is the **K**ernel **ID**entifier (KID). A kernel is in charge of managing common data structure of its threads and logically act as an intermediate orchestrator of all the inter-kernel communications. The threads that belongs to the same $TSet$ are physically running on the same machine and thus they can interact by means of share-memory communication.

The flexibility of the structure according to which the simulation entities have been organized allows the simulation framework to adapt to the underlying hardware and gives a wide range of deploying possibility by supporting very different clusters' architectures. It is in fact possible to scale horizontally each individual level:

Scaling LPs: The total number of LPs represent the total number of entities that we want to simulate, thus modifying this number allows to change the overall size of the simulation.

Scaling threads: Since each thread is bounded to a specific core units, changing the number of threads permits to choose how many physical cores we are going to use on a specific machine.

Scaling kernels: The number of kernels reflects the total number of machine that will participate to the simulation.

As already pointed out, the LPs are the lowest simulation objects of the hierarchy, however, it is common practice that the model group different agents under the same LP. For instance, let's suppose that we want to simulate migration of a population over the surface of a planet, it will came natural to make every LP representing a single individual of the population. A more optimized approach would be to divide the planet into several geographic regions and to assign each of them to one LP; every LP will then be in charge of simulating the individuals that currently are in its geographical region.

2.2 Application level

The application level is logically the higher software layer of the platform, it is the most abstract one in the way it hides all the complex implementation aspects of the underlying simulation engine. The models' code resides in this layer and that usually implements a mathematical model, defining the rules according to which the simulation will evolve. Thanks to this abstraction layer, the application developer needs only to write simple sequential code, that the simulation platform, in a completely transparent way, will distribute and execute in parallel, exploiting all the processing units available on the running machines.

In this context, the developer has the full control over the events, that became the fundamental tool to control the simulation; they can be used to both move informations between the different simulation entities, thus acting as a communication channel, as well as to schedule new simulation activities. The application model can be implement by means of a very simple Application Public Interface (API) exposed by the simulation engine.

-
  ```
  void ProcessEvent(int me, time_type now, int event_type, void
   ↪  * event_content, void *state)
  ```

  This is the main callback used by the kernel to give control to the application layer. Every time a new event need to be processed the engine call this function passing all the proper parameters that will allow the application code accessing the events' data and metadata. The first parameter `me` is the the ID of the LP to which the event belongs. The second one, `now` is the current value for the local clock that indeed coincides with the timestamp of the current event being scheduled. The type of the event is stored in the `event_type` parameter, and can be used by the model developer to differentiate the different events among them, making possible to implement a switch-case approach to split this main execution entry point into several specialized ones. During the startup procedure, the platform generates a special event for each LP, with the reserved type `INIT` and timestamp equal to zero. These special events can be capture by means of the `ProcessEvent` callback in order to perform all the necessary startup actions at each LP, such as the setup of application-level data structure that will be use during the simulation.

The payload of the event containing the actual data exposed through the `event_content` pointer. The platform does not impose any constraints on the structure of the event payload, giving complete freedom to the developer among the kind of data that can be passed by means of event exchange. However the developer needs to keep in mind that does not make any sense to pass memory references inside the event payload, in fact events can be potentially sent by LP residing on different kernels and thus on different physical machines with complete separate memory layout, making impossible for the receiving LP to perform a proper indirection over those pointers.

During the event execution, it is allowed to apply changes to the state of the LP to which the event belongs. The platform does not impose any particular schema neither on the state of the LPs that are presented to the model developer as a plain memory area in which wherever data structure can be stored. The access to this area is exposed at each event execution through the last functions' parameter `state`, that is actually a general memory pointer.

As already discussed the simulation engine has been developed to efficiently support models based on the discrete event approach, in which the time needed for processing a single event is considered relatively small. Thus, it is developer's responsibility to distribute the computations of the model fairly among the different events of the simulation in order to keep the average execution time of the single event small enough.

As already mention, the execution of `ProcessEvent()` code is completely speculative, in fact the events that are executed might be potentially undone. Nevertheless the model developer is completely unaware of this aspect and he is allowed to implement state transition within this callback. Transparently, the underlying ROOT-Sim engine will take care of performing the roll-back procedures in the case a inconsistency has been detected and will commit the definitive events according to the advancement of the commitment horizon

Complete implementation freedom has been given to the model developer allowing to implement this callback using standard ANSI-C code, however, given the uncertainty of the event committing introduced by the speculative approach, the developer is advised to not perform any non-rollbackable actions in this context. For instance if the programmer perform some text printing on the output stream using standard functions during the execution of an event that will eventually rolled-back, the simulation engine will not be able to revert such output.

-
  ```
  void ScheduleNewEvent(int receiver, time_type timestamp, int
  ↪  event_type, void *event_content, int event_size)
  ```

  This function can be used during the execution of a generic event, to program new simulation activities. In fact it allows to schedule events for any LP that is participating to the simulation. The new scheduled event will be inserted into the pending queue of the targeting LP. In the case the event is scheduled from an LP to different one, meaning that the destination LP does not coincide with the LP scheduling the event, the underlying engine will deliver the event in the form of a message. Indeed the destination LP could reside on a completely different kernel, requiring the message to navigate from one machine to the other through the supporting communication network. Given that the events are actually delivered as messages, and that they embed a payload, they can be actually used to make the LPs communicate between each others.

  The ID of the destination LP need to be specified through the parameter `receiver` while the scheduled time of the new event through `timestamp`. The developer has also the possibility to label the event with a specific type through the `event_type` parameter.

  The payload need to be passed by means of a local memory pointer `event_content`, specifying the actual size of the data through the `event_type` parameter.

  More then one events can be scheduled during the execution of one single events but they will buffered and delivered asynchronously as soon as the execution of the current event will terminates. This allows to perform various optimization such as packing end delivering together events destined to the same LP.

  Given the speculative nature of the platform, the execution order consistency it will be guaranteed, even in the case that some events have been scheduled out of order.

- `bool OnGVT(void *snapshot, int gid)`

  Through this function the plaftform communicate to the application model that a committed state has been successfully reached among all the LPs belonging to the current kernel, i.e. that could not ever be reverted. This callback has a very important role, because gives the possibility to the programmer to check if the simulation has reached a final configuration. In fact various termination condition can be checked in the context of this function implementation.

  More generally, all sort of global predicates over the simulation state can be evaluate in the scope of this callback.

  The developer can access the last simulation state snapshot of the current kernel (`gid`), by means of the memory pointer `snapshot`.

2.3 Startup

In this section we are going to give an overview of the main steps that characterize a simulation run. For the sake of generality we assume that the simulation has been configured to use more then one kernels, thus involving several interconnected machines.

Once the model has been correctly linked and compiled Section 2.2, it is possible to run the simulation in a distributed fashion through the `mpirun` command utility. As already mentioned, the three runtime configuration parameters (number of kernels, threads and LPs) can be adjust to scale the simulation according to the available underlying hardware.

The control machine is the one that will handle the very first phases of simulation startup, this is the machine on which the simulation start command has been issued. According to the number of kernels specified, the control machine will instantiate a number of main processes by placing each of them in a different computing node of the cluster. Once the kernels have been spawn, all of them will start to setup and prepare their local environment. In the very first place, the configuration parameters need to be parsed and stored in the relative data structure, then the setup of each single platform subsystems will take place. In general every module needs to allocate some memory and setup the data structures that will be used during the simulation; for instance the communication submodule will start allocating the event queues. The MPI subsystem will perform the handshakes among all kernels, in order to provide inter-kernel communication support. Moreover, during this initial phase, the LPs will be distributed among the kernels in a deterministic way.

Once all the submodules have been correctly initialized, every kernel will start spawning the threads that will perform the actual simulation for the LPs in their *LPSet* (Section 2.2). Every thread, possibly bound to a specific processing unit, will then enters into the main simulation loop reported in Listing 1.

2.4 Main simulation loop

The first bunch of actions taken by every thread after they've entered the main simulation loop are enclosed into the `initialize_worker_thread()` function, in which the every simulation thread binds to itself the LP that will manage during the simulation, and will generate the special `INIT` event (Section 2.2) for each of them, by pushing it into their event queue.

In the case the simulation is using MPI to support multi kernel execution, all the threads participate to a synchronization barrier that guarantees that the actual simulation will start only after all the threads on all the kernels have been correctly spawned and setup. After the barrier, all the threads enter into the *while loop* (Listing 1 - line 10) that will be executed repeatedly until the termination condition will be reached.

As will be explained with details in Section 2.5, communications between LPs residing on the same *LPSet* are faster with respect to the inter-thread ones; for this reason the kernel try to periodically reshuffle all the *LPSet* in order to group together the LPs that are having higher interaction between them.

At the beginning of every cycle of the while loop, the threads will perform the local LP rebinding, in order to update their *LPSet* according to the aforementioned reshuffle.

Before to schedule the LPs' event execution, every thread will check if there are pending incoming messages destined to some LP residing on its kernel, and if there are any, it will receive them and put into the correct destination data structure (further details will be analyzed in Section 2.5). Additionally the thread will try to

Listing 1 Main loop routine of ROOT-Sim simulation platform

```c
static void *main_simulation_loop(void *arg) {

   // Do the initial (local) LP binding, then execute INIT at all (local) LPs
   initialize_worker_thread();

   #ifdef HAS_MPI
   syncronize_all();
   #endif

   while (!end_computing()) {

      // Recompute the LPs-thread binding
      rebind_LPs();

      #ifdef HAS_MPI
      // Check whether we have new ingoing messages sent by remote instances
      receive_remote_msgs();
      prune_outgoing_queues();
      #endif
      // Forward the messages from the incoming message queue to the destination
       ↪  LPs
      process_bottom_halves();

      // Activate one LP and process one event. Send messages produced during
       ↪  the events' execution
      schedule();

      gvt_operations();

      #ifdef HAS_MPI
      collect_termination();
      #endif
   }

   // If we're exiting due to an error, we neatly shut down the simulation
   if(simulation_error()) {
      simulation_shutdown(EXIT_FAILURE);
   }
   simulation_shutdown(EXIT_SUCCESS);
}
```

release memory by discarding obsolete data related to sent event that have been already received at destination (Listing 1 - line 18).

Now is the time [3] for the thread, to withdraw the messages that were temporary stored into the incoming queue of each LP and move them to the event queue of the same LP (Section 2.5.4).

Once also the last messages has been incorporated, the thread will select the furthest behind LP, the one with the minimum virtual clock, and will invoke the `ProcessEvent()` callback (Section 2.2) upon the next event on the event queue of the selected LP.

Before to return from the `schedule` procedure (Listing 1 - line 24), the simulation thread will perform the asynchronous delivery of the new events produced during the execution of the `ProcessEvent` callback, if any.

The thread can now gives its contribution on the GVT synchronization algorithm by executing the `gvt_operations()]` function.

The action on line 29 need to be performed only in the case the current simulation run involves multiple kernels, in fact this action aim at collecting the termination declaration coming from other kernels that have already reached the final simulation state.

Now, if the local termination condition has not been reached yet, the thread will start another round of the *while loop*, otherwise will simply die by exiting from the `main_simulation_loop()` function.

2.5 Journey of an event

In this section we are going to review the main issues involving the delivery of an event, we will analyze in details all the steps that it will pass through, from its creation till its final processing.

2.5.1 Event creation

As already discussed in Section 2.2, the journey of our message starts inside the model application code, in particular during the execution of the `ProcessEvent()` callback. In this scope the LP$_j$ being simulated can create a new event, by simply

[3]Cit. Izner M. *shish* the launch time idea - http://scientificpubs.eu/718293

allocating memory, filling it with the message payload and pass the pointer of this memory to the simulation engine through the invocation of the `ScheduleNewEvent()` function. The other parameter of the function, among other things, allow to specify also the timestamp of the new scheduled event and the identifier of the target LP_k to which the event is destined.

As soon as the engine receives all the metadata and the payload, it packs the event into a message structure composed by a payload and an header and attaches to it additional metadata such as the sending time (t) and the identifier of the sending LP (LP_j). The message is then buffered into the temporary *outgoing queue* of the sender. Once the `ProcessEvent` routine in which the message was generated has been terminated the simulation thread will empty temporary *outgoing queue* by pulling the buffered events one by one.

The header of the message (that is relatively small) will be stored in the definitive output queue. This queue is used to keep an history of sent messages and it will be used in the case of a future rollback of the LP in order to construct the relative anti-messages.

2.5.2 Message forwarding

After the metadata of the message (the header) has been recorded into the local LP message history, it can be safely forwarded to the actual delivery phase by passing it to the `Send()` procedure. Exactly from this time, the message will be considered in transit and thus it will became non-observable (Section 1.3.1); during this transitional phase, the message will be handled by the underlying communication subsystem making impossible for the simulation software framework to track it until it will actually be delivered to the destination LP.

During this non-observability period the message could take two different route as depicted in Figure 2.2, in particular, if the sender LP resides on a kernel different from the one to which belong the destination LP, the message need to traverse the communication network to travel from one kernel's machine to the other. This first case of inter-kernel communication is represented in Figure 2.2 by the purple route taken by the second message. In fact, M_2 is sent from LP_2 to LP_3 that belong respectively to $Thread_2$ and $Thread_3$. The two threads resides on different kernels and thus on different physical machine.

On the other hand, there is the intra-kernel communication, represented in

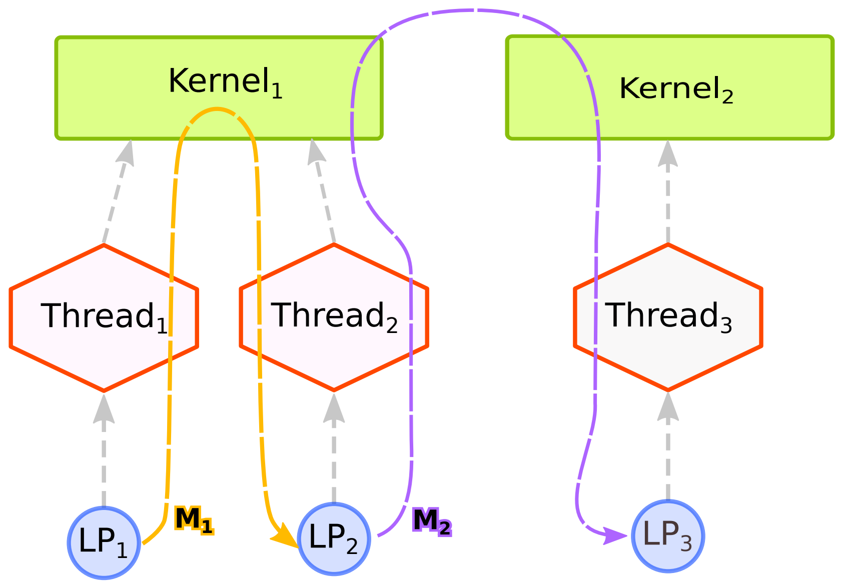


Figure 2.2: Example of the communication routes of two events, while M_1 follows an intra-kernel communication path exploiting shared-memory of the local machine, M_2 needs to travel from one kernel to the other through the interconnection network

Figure 2.2 by the orange route taken by the first message M_1. In this case the message will not ever leave the local machine, at the most it will travel from one thread to another still residing on the same computing node. Thanks to the kernel locality property of the route, this communication can be implement by means of the shared memory of local machine. In fact if the destination data structure are placed in this portion of memory they can be accessed by both the sending and the receiving thread. Thus in order to deliver the message the thread of the sending LP will need just to insert the message into the temporary incoming queue of the destination LP, to which it has write access. A theoretical discussion of the intra-kernel communication pattern has been reviewed already in Section 1.3.5.

Listing 2 reports the `Send()` routine in which one of the two aforementioned routes will be chosen according to the message destination LP. In the very first place we need to understand if the destination LP resides on the same kernel that is executing the code, that is the kernel on which is currently bound the LP that have been generated the event. This can be achieved by means of the `GidToKernel` macro that given the identifier of an LP it returns the ID of the kernel on which it is currently bound. Thus on Line 3 we check if the kernel of

Listing 2 Code snippet of the send message procedure

```c
void Send(msg_t *msg) {
    // Check whether the message recepient kernel is remote
    if(GidToKernel(msg->receiver) != kid){
        send_remote_msg(msg);
        return;
    }
    insert_bottom_half(msg);
}
```

the receiver LP is different from the local kernel executing *kid*, if turns out to be true means that the message will be forwarded to the communication networks by means of the `send_remote_message()` function. On the other hand, if the destination LP is local to the kernel, the shared-memory will be used through the `insert_bottom_half()` procedure that will be analyzed later on.

Listing 3 Code snippet of the send remote message procedure

```c
void send_remote_msg(const msg_t* msg){
    outgoing_msg* out_msg = allocate_outgoing_msg();
    // message copy
    out_msg->msg = *msg;
    out_msg->msg.colour = threads_phase_colour[local_tid];
    unsigned int dest = GidToKernel(msg->receiver);

    register_outgoing_msg(&(out_msg->msg));

    MPI_Isend(&(out_msg->msg), 1, msg_mpi_t, dest, MSG_EVENT,
     ↪  MPI_COMM_WORLD, &(out_msg->req));

    // Keep the message in the outgoing queue until it will be received
    store_outgoing_msg(out_msg, dest);
}
```

Let's now focus on the first case, analyzing in details the steps involving the delivery of the message that needs to travel across communication network. The actions related to this phase and executed at the sender side, are coded into the `send_remote_msg()` function (Listing 3). The core instruction is reported on Line 11 in which the message is sent through the `MPI_Isend` non-blocking routine, that allows the executing thread to deliver the message to the underlying MPI implementation that will take care of all the communication issues and will transport the message to the destination kernel.

Once the caller thread will exit from the `send_outgoing_msgs()` it will consider the message as delivered and it will free the message memory referenced by the `msg` pointer. On the other hand the `MPI_Isend` function is non-blocking, meaning that the actual read of the message buffer and the relative delivery could happen even after the routine has already gave back the execution to the caller. For this reason the very first actions (Line 2-5) performed into the `send_remote_msg()` aim at creating a copy of the message `msg`. The new copy, namely `out_msg`, will be temporary stored into an outgoing queue in which it will remain until the underlying MPI implementation will have delivered the message (Line 15 - `store_outgoing_msg()`). This data structure is shared among all the local threads that will periodically check for the delivery status of these messages. As soon as a message will result to be already received by the receiver LP the relative memory will be freed.

The `register_outgoing_msg()` function (Listing 3 - Line 8) marks the entrance of the message into the "*non-observable time window*" (Section 1.3.1) by registering the message as in-transit. This information is required by the ORCHESTRA algorithm and is discussed with details on Section 3.2

The message is now in the hands of MPI that will eventually deliver it.

2.5.3 Message delivery

From the point of view of the simulation framework every event is issued by an LP and is destined to another LP. As depicted in Figure 2.2, in order to travel from its source to its destination, the event will go up through several software layers on the sender side and then will descend at receiver side until it will be incorporated into the destination data structure. At the higher software layer, there is MPI that manages communication between different computing nodes. At this abstraction level there is not concept of logical processes, messages could only travel from one machine to another, from the source kernel to the destination one. Coming back to our `send_outgoing_msgs()` we could see that the actual destination of the message, specified as parameter of the `MPI_Isend` call, is in fact the identifier of the kernel in which resides the destination LP.

Thus even if an event is destined to some specific logical process, while it is crossing the communication network, it will be considered as a message destined to some specific kernel. Before to unpack the message at the receiving kernel it is impossible to know which logical process the event has been sent too. For this reason the following receiving approach has been developed.

Listing 4 Code snippet of receive messages procedure

```
1  void receive_remote_msgs(void){
2      if(!spin_trylock(&msgs_lock)) return;
3
4      int res;
5      msg_t msg;
6
7      while(pending_msgs(MSG_EVENT)){
8          res = MPI_Recv(&msg, 1, msg_mpi_t, MPI_ANY_SOURCE, MSG_EVENT,
            ↪  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
9          if(res != 0){
10             rootsim_error(true, "MPI_Recv did not complete correctly");
11             return;
12         }
13         insert_bottom_halve(&msg);
14     }
15
16     spin_unlock(&msgs_lock);
17 }
```

At each cycle of the main simulation loop (Section 2.4) every thread checks if there are pending incoming messages through the `receive_remote_msgs()` routine reported on Listing 4. In a nutshell, the thread loops while there are any pending messages, and at every cycle receives one of them and stores it into the destination datas structure.

The `pending_msgs()` function present into the while condition (Line 7) is actually a wrapper to an `MPI_Iprobe` call that allows to check for pending messages in a non-blocking fashion. Every message is actually received through the `MPI_Recv` blocking call, to which the thread passes the kind of message that it is ready to receive and the pointer to the newly allocated message buffer to where it will be stored.

The whole function code is executed in a `try_lock` scope, in this way only one thread at time on the same kernel can receive messages. This execution exclusivity allows to avoid the following pattern:

- Both Thread$_1$ and Thread$_2$ enters the function and since there are message pending they evaluates as true the while condition (Line 7).
- Thread$_1$ get preempted by the system on Line 8.
- Thread$_2$ proceeds and cycles through the loop until all pending messages are received and exits from the function.
- The system schedules back the execution of Thread$_1$ that try to perform the `MPI_Recv()` on Line 8. Since there are no longer pending messages and the call is blocking the thread will freeze until some new messages will arrive.

By using the `try_lock` construct, race conditions on incoming messages are avoided. A thread will try to lock to acquire the section lock, in the case another thread is already receiving messages, it just gives up and continues with the execution of its main simulation loop. Moreover the critical section allows to stock the messages in the same order of their arrival.

Once the message has been successfully received and copied into the local buffer, it will be passed to the `insert_bottom_halve()` that will insert it into the destination data structure.

2.5.4 Bottom halves

During the startup of the simulation, one data structure for each logical process is allocated into the shared portion of the main memory. This structures are called *bottom halves* and are used as a temporary buffer for the incoming events of the logical processes. Since they resides into the shared-memory, every process can enqueue a new event to another local LP by simply writing the event into its bottom halves. The access pattern to one of those data structure is known as *multiple writer - single reader*, in fact all the threads on the local machine could possibly need to enqueue a new event into a specific bottom halve. On the other hand only the owner of the bottom halve have the right to dequeue them from the structure since it is the only recipient of these events.

In order to support both the write and read action in a efficient way, the data structure has been implemented with two buffers an upper one and a bottom one. All the threads enqueue new events on the bottom one, events are enqueued one after the other and a lock synchronization approach is used to avoid race condition while writing. The recipient of the events can only read the messages from the upper buffer, one at time by dequeuing them. When the upper buffer get empty, all the writer get blocked and the two buffer are exchanged. The bottom one that potentially contain new messages becomes the upper one and the empty one becomes the one dedicated to the insertion. After the change the writers are unlocked and can continue to enqueue message on the new empty buffer.

Since the exchange procedure involves the swap of two pointers the writers will not be blocked for much time. Moreover since the reader access only the upper buffer it does not need to acquire any synchronization lock except when it needs to swap the two buffers.

2.5.5 Incoming queue

During the main loop every thread will execute the `process_bottom_halves()` function (Section 2.4), in which it empties all the bottom halves of the LPs bound to it. The messages pulled out form these data structure are then enqueued into the definitive incoming queue of the corresponding LP ordered by their schedule timestamp.

Chapter 3

Asynchronous wait-free GVT algorithm

3.1 Baseline description

The overall sequence of messages and events which compose the ORCHESTRA algorithm is shown in Figure 3.1[1]. ORCHESTRA belongs to the family of two-cut distributed GVT algorithms (Section 1.3.4), and relies on the notion of phases to let the global computation of the GVT value advance, without any form of explicit synchronization. A kernel instance can decide independently from any other to start a GVT computation (i.e., moving from the *idle* phase to the *start* phase), thus avoiding any initial form of communication. Eventually, all distributed instances will move to the *start* phase, collaborating to determine the new GVT value.

At the level of the single kernel instance k, ORCHESTRA relies on phases which are governed by relying on a set of atomic counters. In particular, during the computation of the GVT, every worker thread in the $TSet_k$ set (Section 2.1) carries on reduction actions on the LPs bound to it, in a way similar to the Pellegrini's algorithm presented in Section 1.3.5. Once a portion of the computation is carried out, each worker thread notifies its completion by atomically decrementing a counter in shared memory, and enters the next phase after all the threads have concluded the current one—this can be done by simply checking the value of a shared counter.

[1]This illustration has given the "ORCHESTRA" name to the algorithm, due to it resembling a music score.

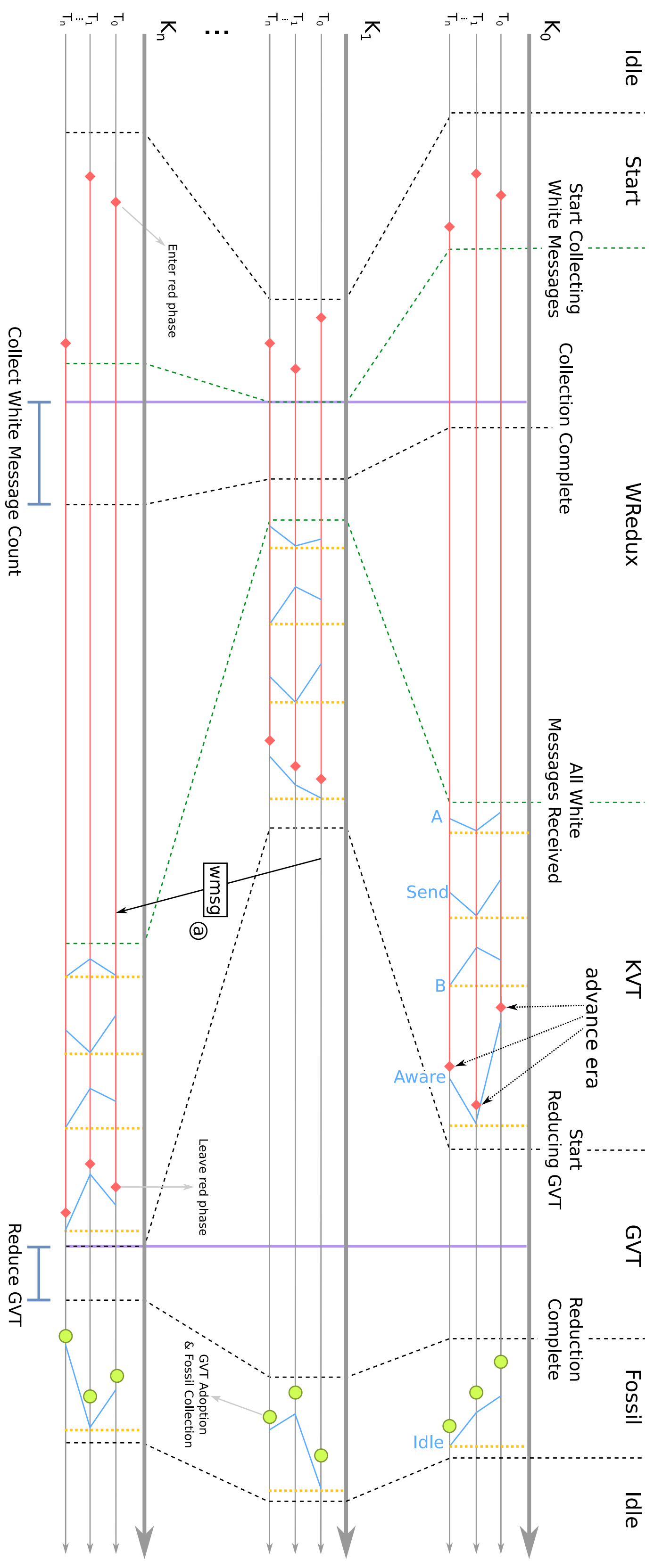


Figure 3.1: Illustration of the ORCHESTRA algorithm

Phase changes at the level of kernel instances are triggered by some global (i.e., distributed) event. In particular, each simulation kernel instance in $KernelSet$ transits through a number of *kernel phases*. The succession of these phases is governed by two different factors. On the one hand, each kernel instance maintains a set of counters to determine when some global condition is met. On the other hand, the completion of some asynchronous collective communication primitive determines the advancement to a different phase.

As for the conditions checked by relying on counters, ORCHESTRA inherits from the proposal in [15] the notion of *colored messages*. In particular, worker threads in ORCHESTRA continuously alternate their execution in a red and a white color. Messages sent while running in the red phase are colored red, and messages sent running in the white phase are colored white. Each worker thread switches from the white to the red phase independently of each other. This means that, at the same wall-clock time instant, two different worker threads (possibly in two different kernel instances) can live in a completely different phase, as depicted in Figure 3.1. There are only two phase-changing points which are not allowed to be inverted. These are marked by a purple vertical bar in Figure 3.1. Anyhow, we emphasize that this behavior is not supported by any form of explicit synchronization. In fact, these bars are associated with the completion of asynchronous collective communication primitives. We note that by relying on this scheme, every worker thread on any kernel instance is allowed to carry on simulation work while the GVT value is being computed, thus allowing for an efficient usage of the available computing resources.

As a last note, each kernel instance maintains a counter to identify the current *era value*. This value is used to discriminate between messages sent across two consecutive white phases. In this way, we are able to reduce the amount of metadata exchanged across different kernel instances.

Throughout the description of the various procedures of ORCHESTRA, we rely extensively on the OnlyOnce pseudo-code statement. This statement represents a block of code that should be executed only once by any of the worker threads which concur in the activities of a simulation kernel instance. In practice, we propose to implement such a statement by relying on the Compare and Swap (`CAS`) construct. In particular, for each OnlyOnce statement, an integer *token variable* should be declared. All token variables should be initialized to zero, and every worker thread will try to execute a `CAS(&token,1,0)`, meaning that all threads passing through the OnlyOnce statement will try to update the value of `token` to 1. By the semantics of the `CAS` construct, only one thread will be able to successfully perform the update. Therefore, by checking if the `CAS` has succeeded, only one thread will actually perform the actions associated with this statement.

In the next section, we present the details of ORCHESTRA, explaining the meaning of each phase (both at kernel and thread level), and discussing which are the conditions which allow to switch to a next phase.

3.2 The Algorithm

Algorithm 1 Register Outgoing Event

```
1: procedure RegisterOutgoing(event $e$)
2:     if $e.recipient$ is locally hosted then
3:         return
4:     end if
5:     if $e.colour = red$ then
6:         $min_t^{RED} \leftarrow \min\{e.timestamp, min_t^{RED}\}$        ▹ R1
7:     else
8:         AtomicInc($w\_counter\_sent[e.kern\_dest]$)
9:     end if
10: end procedure
```

Algorithm 2 Register Incoming Event

```
1: procedure RegisterIncoming(event $e$)
2:     if $e.recipient$ is locally hosted then
3:         return
4:     end if
5:     if $e.colour = white$ then
6:         AtomicInc($w\_counter\_recv[e.era \% 2]$)        ▹ R2
7:     end if
8: end procedure
```

In a non-observable Time Warp system, there is a time window along which a message associated with a send operation at some LP is considered as *in-transit*, and therefore it has not yet been incorporated into the recipient LP's message queue. To cope with in-transit messages, every time a new event is injected into the system targeting a remote LP, the sender simulation kernel tracks the boundaries of the non-observability window of the message carrying the event by relying on the RegisterOutgoing() and RegisterIncoming() procedures, executed at the source and destination simulation kernels, respectively. The pseudo-code of these procedures is depicted in Algorithms 1 and 2. As we have discussed, our asynchronous algorithm relies on colored messages to differentiate among the different execution phases of the simulation run in which there is the need to explicitly account for in-transit messages. This is reflected by the fact that the source kernel tracks the minimum timestamp of events scheduled by any LP in the LP sets of its worker threads.

Periodically, a kernel instance determines (independently of the others) that a global agreement on the value of the GVT is to be made [2], starting the flow across the different phases, which are associated with different states of threads and kernels.

3.2.1 Kernel virtual time

We start by illustrating the procedure which is used to compute a local estimation of the GVT value at a certain kernel, which we refer to as the *Kernel Virtual Time* (KVT). The KVT is computed by following the same definition of the GVT, yet by considering only correctly-executed events and in-transit messages which are related to the workers of the specific simulation kernel instance. The KVT computation is carried out in a way similar to what has been presented in [16], and the pseudo-code of the KVT() procedure is presented in Algorithm 3. In particular, when the simulation kernel has entered a GVT-computation phase, it repeatedly (i.e., at every main loop iteration) invokes the `KVT()` procedure. The goal of this procedure is to pass through all the required phases (associated with all the worker threads) which ensure a correct estimation of the KVT at the local kernel, returning `true` only when the correct KVT value has been finally computed (**K7**). Once the local KVT computation is started, all the worker threads have been already set into the A phase. In this phase, all threads execute the following actions:

- messages being received from remote kernels are extracted from the underlying communication channel.

[2]The algorithm proposed in this paper is independent of the actual condition which triggers the GVT computation, which can therefore depend on some elapsed wall-clock time or memory shortage. We note that the reception of a red message from a remote kernel could be an additional good trigger to start the computation, helping to reduce as well the total duration of the GVT computation.

Algorithm 3 Kernel Virtual Time—KVT

```
1: procedure KVT( ): returns boolean
2:     if $th\_phase$ = A then
3:         Receive messages from remote kernels
4:         Incorporate messages into event queues
5:         $min_t^A \leftarrow \min_{i \in LPSet_t} LVT_i$                          ▷ K1
6:         $th\_phase \leftarrow$ Send
7:         AtomicDec($C_A$)                                                         ▷ K2
8:         return false
9:     end if
10:    if $th\_phase$ = Send ∧ $C_A = 0$ then                                      ▷ K3
11:        Receive messages from remote kernels
12:        Incorporate messages into event queues
13:        Execute the next event
14:        Send output messages/anti-messages
15:        $th\_phase \leftarrow$ B
16:        AtomicDec($C_{Send}$)
17:        return false
18:    end if
19:    if $th\_phase$ = B ∧ $C_{Send} = 0$ then
20:        Receive messages from remote kernels
21:        Incorporate messages into event queues
22:        $min_t^B \leftarrow \min_{i \in LPSet_t} LVT_i$
23:        $th\_current\_era \leftarrow th\_current\_era + 1$                      ▷ K4
24:        $th\_colour \leftarrow white$
25:        $min_t \leftarrow \min\{min_t^A, min_t^B, min_t^{RED}\}$               ▷ K5
26:        $th\_phase \leftarrow$ Aware
27:        AtomicDec($C_B$)
28:        if $C_B = 0$ then
29:            OnlyOnce:
30:                $min_k \leftarrow \min_{t \in TSet_k} min_t$                    ▷ K6
31:                return true                                                  ▷ K7
32:        end if
33:    end if
34:    return false
35: end procedure
```

- the events associated with these messages are incorporated into the event queues.
- the minimum LVT across all LPs bound to each thread is found, and stored into a per-thread variable (**K1**).

Once the minimum has been computed, each thread moves to the Send phase, and notifies that it has completed its A phase via the atomic counter C_A (**K2**). This counter ensures that no worker thread will be ever executing the actions associated with the Send phase, until all threads have completed the execution of the A phase (**K3**). We note that the different worker threads are fully allowed to complete the execution of phase A at different wall-clock time instants. This is illustrated by the skewed blue lines in Figure 3.1. The vertical dotted lines illustrate that the next phase is never started until all threads have completed the previous phase. This algorithmic organization ensures wait-freedom of execution across the different threads.

The steps associated with the Send phase entail the following actions:

- messages are received from the communication channel and incorporated into the event queues.
- the next event (in a Lowest-Timestamp First fashion) is executed by the worker thread.
- possibly-generated messages/antimessages are sent.

These actions ensure that if some LP bound to a worker thread has executed a rollback operation due to a straggler message received during phase A, the LVT of that LP is realigned to that of the straggler message. For a thorough discussion on the correctness of this approach, we refer the reader to [16]. Similarly to the previous phase, all threads switch to the B phase, but they do not start the actions associated with it until all threads have actually completed the Send phase.

At this point, the KVT() procedure shows some differences from the algorithm in [16]. In particular, after having received and incorporated the messages/events, a worker thread can consider the red phase as concluded, switching to the next era as well (**K4**), and computes again its minimum LVT taking into account as well the minimum timestamp of the red messages (**K5**). We note, that switching to the next era does not require any synchronization among the worker threads (as depicted as well in Figure 3.1), as the correctness of the algorithm is ensured by the consecutive flow of white/red phases. At the same time, once a thread enters the white phase associated with the next era, this white phase can be regarded as a completely-different phase. With respect to Figure 3.1, let us consider the white message ⓐ sent from T_0 at K_1 to T_0 at K_n. It crosses the “equivalent” second cut of the algorithm in [15]. Since our algorithm does not require FIFO network ordering, this would break the correctness of the algorithm in [15]. Differently, in our algorithm, the correctness is ensured by the fact that the next era virtually bounds the white message ⓐ to the next round of red messages, as depicted in Algorithm 2 (**R2**).

All threads then enter the Aware phase, telling that they know that their contribution to the KVT computation is over. Once all the threads are in the Aware phase, the global minimum among all the worker threads at the given kernel, namely the KVT value, can be computed as the minimum among the minima of each worker thread (**K6**).

We note that after the execution of every phase, the execution of the KVT() procedure is explicitly interrupted via a **return** statement. This prevents any thread from possibly executing two different phases consecutively. While this would not hamper the correctness of the algorithm, we note that this gives higher priority to the execution of simulation events, which ensures higher performance by reducing the overall rollback probability.

3.2.2 Global Virtual Time

The algorithm to compute the KVT value is used in an asynchronously-coordinated manner to compute the GVT value. As mentioned previously, once a certain condition is met, all the simulation kernel instances start executing at every main loop iteration the procedure illustrated in Algorithm 4.

A simulation kernel starts its portion of the GVT computation in the Start phase. Similarly, all its threads are in the Idle phase. In this case (**G1**), we set to ∞ the local min_t and min_t^{RED} values, which will be used in the KVT algorithm to compute the minimum accounting as well for in-transit messages. We then color the running thread red, so that all the messages it will be sending from now on will be colored red as well. Finally, during the initialization phase, we move the

Algorithm 4 Global Virtual Time—GVT

```
 1: procedure GVT( )
 2:     if $ker\_phase$ = Start $\wedge$ $th\_phase$ = Idle then                      ▷ G1
 3:         $min_t \leftarrow \infty$
 4:         $min_t^{RED} \leftarrow \infty$
 5:         $th\_colour \leftarrow red$
 6:         $th\_phase \leftarrow$ A
 7:         AtomicDec($C_{Init}$)                                                   ▷ G2
 8:         if $C_{Init} = 0$ then
 9:             OnlyOnce:
10:                 WMsgsRedux( )                                                   ▷ G3
11:                 $ker\_phase \leftarrow$ WRedux
12:         end if
13:         return
14:     end if
15:     if $ker\_phase$ = WRedux then
16:         if WMsgsReduxCompl( ) $\wedge$ WMsgsRecv( ) then                        ▷ G4
17:             OnlyOnce:
18:                 $\forall i \in KSet$: AtomicSet($w\_counter\_sent[i]$, 0)
19:                 $ker\_phase \leftarrow$ KVT
20:         end if
21:         return
22:     end if
23:     if $ker\_phase$ = KVT $\wedge$ $th\_phase \neq$ Aware then
24:         if KVT( ) then
25:             GVTRedux( )                                                         ▷ G5
26:             $ker\_phase \leftarrow$ GVT
27:         end if
28:         return
29:     end if
30:     if $ker\_phase$ = GVT $\wedge$ GVTReduxCompl( ) then                        ▷ G6
31:         OnlyOnce:
32:             $new\_gvt \leftarrow last\_reduced\_gvt$
33:             $ker\_phase \leftarrow$ Fossil
34:         return
35:     end if
36:     if $ker\_phase$ = Fossil $\wedge$ $th\_phase$ = Aware then
37:         FossilCollection( )
38:         $th\_phase \leftarrow$ Idle
39:         AtomicDec($C_{End}$)                                                    ▷ G7
40:         if $C_{End} = 0$ then
41:             OnlyOnce:
42:                 $ker\_phase \leftarrow$ Idle
43:         end if
44:     end if
45: end procedure
```

thread currently running this procedure into the A phase—since these are symmetric kernel instances, all worker threads run the same code, and will eventually all enter the A phase. As previously illustrated in Algorithm 3, this phase is associated with the computation of the local (per thread) minimum. Nevertheless, this is only a preparation towards that phase, as it will not affect the execution flow until the KVT() procedure is called, which will happen in a future phase. Similarly to what we have done in the Algorithm 3 (**K2**), we rely on atomic counters (**G2**) to determine when all threads have finished executing the tasks associated with this phase. At the end, by still relying on the OnlyOnce construct, we start collecting all white messages (**G3**). As mentioned before, this is an asynchronous task: each kernel receives the total number of white messages sent by the other kernels, and starts counting all white messages which are received.

The completion of this asynchronous task takes place in the WRedux phase (**G4**): when the total number of white messages is received—this is checked via the WMsgReduxCompl() call—and the counter of in-transit white messages is zero—this is checked via the WMsgRecv() call—the global GVT computation can advance to the next phase. Again, only one thread will force the advancement to the next phase, after having set the white counter to zero for all threads in the local kernel instance. We note that after this phase all in-transit messages have been incorporated into the message queues (possibly causing rollbacks), and therefore all threads can observe the relevant information for the GVT computation locally.

Therefore, the KVT computation as depicted in Algorithm 3 can be repeatedly invoked at every simulation loop. As shown, the KVT() procedure returns true after that all the threads on a simulation kernel instance have flown through all the phases, and the local minimum for one simulation kernel instance has been computed. At this point, another asynchronous global computation can take place, namely the global GVT reduction (**G5**). This phase is semantically equivalent to computing $new\_gvt \leftarrow \min_{i \in KSet} min_i$. Again, this is done by relying on asynchronous calls, and therefore once all kernels agree on the global minimum (**G6**), one single thread will make the kernel advance to the next phase, namely the Fossil phase.

The Fossil phase allows all the worker threads to execute the fossil collection phase on the LPs currently bound to them. While in theory the GVT computation might be considered as already completed, this phase ensures that, independently of the condition which triggers the activation of the GVT computation, a new asynchronous wait-free computation of the GVT value will never be started before the fossil collection is completed. This has the benefit of significantly simplifying the overall structure of the algorithm, making it suitable for most simulation engines without any need to check for critical races on data structures. This is again done by relying on one atomic counter (**G7**). Once this atomic counter is set to zero, one single thread will set the kernel phase to Idle, thus allowing the next GVT computation to take place, whenever the condition is met. At this point all threads are already in the Idle phase, thus the initial conditions of the algorithm have been restored.

3.2.3 Initialization

Algorithm 5 Initialization

```
1: procedure StartGVT( )
2:     if $ker\_phase$ = Idle then
3:         OnlyOnce:
4:             $w\_counter\_recv[(th\_current\_era - 1)\%2] \leftarrow 0$        ▷ S1
5:             AtomicSet($C_{Init}$, $|TSet|$)
6:             AtomicSet($C_A$, $|TSet|$)
7:             AtomicSet($C_{Send}$, $|TSet|$)
8:             AtomicSet($C_B$, $|TSet|$)
9:             AtomicSet($C_{End}$, $|TSet|$)
10:            $ker\_phase \leftarrow$ Start
11:    end if
12: end procedure
```

To conclude the description of ORCHESTRA, we present in Algorithm 5 the steps executed to initiate a new GVT computation, The StartGVT() procedure is activated, as mentioned before, whenever some condition is met by the simulation engine. Since the initiation of the algorithm should be performed only once per GVT round, we explicitly check in the procedure whether the kernel is in the Idle phase just for the sake of logical correctness. The goal of this procedure is to restore all the atomic counters used in the GVT() and KVT() procedures to the number of threads locally hosted by the kernel, thus allowing the synchronization on zero to take place. Additionally, this procedure must set to zero the counter of white messages received in the previous era. Since we logically consider only two eras, we refer to the previous one using -1, for the sake of the description of the algorithm. In practice, +1 can be used, allowing to start from 0 as the value of the $th\_current\_era$ variable. Finally, to actually notify all threads in the kernel that they have to participate in the GVT calculation, the kernel is moved to phase Start, causing the procedure GVT() to actually perform the computation (**G1** in Algorithm 4). We note that this initialization procedure should reset as well all token variables used to implement the OnlyOnce statement. For the sake of clarity, this is not reported in the pseudocode of Algorithm 5.

Chapter 4

Experimental results

Figure 4.1: Marenostrum III supercomputer located at Torre Girona, Barcelona, Spain. This photo has been granted by BSC-CNS under the Creative Common license (CC BY-NC)

4.1 Cluster architecture

Experiments have been conducted by deploying the ROOT-Sim simulator (Section 2) on MareNostrum III supercomputer [1] located at "Barcelona Supercomputing Center - Centro Nacional de Supercomputación" (BSC-CNS). The cluster, shown in Figure 4.1, was update to the third version in August 2012 thanks to an agreement between the Spanish government and IBM, reaching the 34^{th} position on the TOP500 ranking [2]. The cluster is composed of $3,056$ IBM dx360 M4 compute nodes distributed over 36 IBM iDataPlex Compute racks and can reach a total peak performance of $1,1$ Petaflops. Each compute node is composed of:

- Two 8-core Intel Xeon processors `E5-2670` at 2.6 GHz, 20 MB cache memory, with a peak performance of 332.8 Gflops per node.
- Eight 4 GB DIMM's, 1.5V DDR3 @ 1600 MHz. Having 32 GB per node and 2 GB per core
- Local hard drive: IBM 500 GB 7.2K 6Gbps NL SATA 3.5.
- MPI network card: Mellanox ConnectX-3 Dual Port QDR/FDR10 Mezz Card.
- 2 Gigabit Ethernet network cards (management network and GPFS)

To support the proprietary InfiniBand technology, each rack has 4 Mellanox 36-port Managed FDR10 IB Switches. The operating system driving all the nodes is SuSe Linux 11 SP3 based on linux kernel `3.0.101-0.47.90`

4.2 Communication framework

The message passing interface framework used for the underlying inter-node communication is OpenMPI [9]. To exploit the multi threading capability of the library a self compiled version of the OpenMPI `2.1.0` version has been used. Native threads support has been enabled through the `--enable-mpi-thread-multiple` configuration flag. For a practical analysis of the the multi-threading capabilities of the library the reader is invited to look section **??**.

[1]`https://www.bsc.es/marenostrum/marenostrum/mn3`

[2]TOP500 ranking of November 2013 - `https://www.top500.org/list/2013/11/`

4.3 Benchmark

Regarding the benchmark for assessing the effectiveness of ORCHESTRA, we have used the Personal Communication System (PCS) real-world cellular simulation model, which has already been used as a reference benchmark application in several other studies oriented to optimistic PDES. Each LP models a wireless cell managing $1,000$ wireless channels to provide coverage to mobile devices in a hexagonal region. The model is high-fidelity in terms of how interference across different channels within a same cell and power management upon call setup/handoff is captured and actuated. Particularly, the application handles power management simulation according to the results in [12]. The application is also highly parameterizable by allowing the recalculation of fading coefficients and actual Signal-to-Interference Ratio (SIR) both on the occurrence of specific events (e.g. the startup of a call) and periodically (so as to account for, e.g., changes of conditions in the coverage area). Also, the inter-arrival of calls to mobile devices in the coverage area can be configured, thus leading to different values for the wireless channels' utilization factor. This, in its turn, affects both memory and CPU demand by the simulation. The interaction across the different LPs takes place upon a handoff of a mobile device involved in an ongoing communication, in which case the wireless channel at the source cell is released, and a new one in the destination cell is attempted to be reserved.

The experimentation has been conducted by using 5 minutes value as the average residual residence time in one cell for a mobile device involved in an on-going call, while the average call duration was set to 2 minutes. Both these values have been set to follow exponential distributions. Also, the channel utilization factor has been set to 75%, with balanced workload on all the LPs. This settings produce simulation event's average CPU requirement of about 150 microseconds. ROOT-Sim has been run on a different number of MareNostrum nodes (4.1), namely from 1 node to 32. On every node, all available 16 cores have been always used. Two different families of experiments have been conducted: one using 4096 LPs and running until each cell has completed the simulation of 500,000 calls; one using $16,348$ LPs and running until the completion of $10,000$ calls.

In Figure 4.2 reports experimental results when running PCS configured with 4096 LPs. The y axis is logarithmic. Three different GVT reduction algorithms are

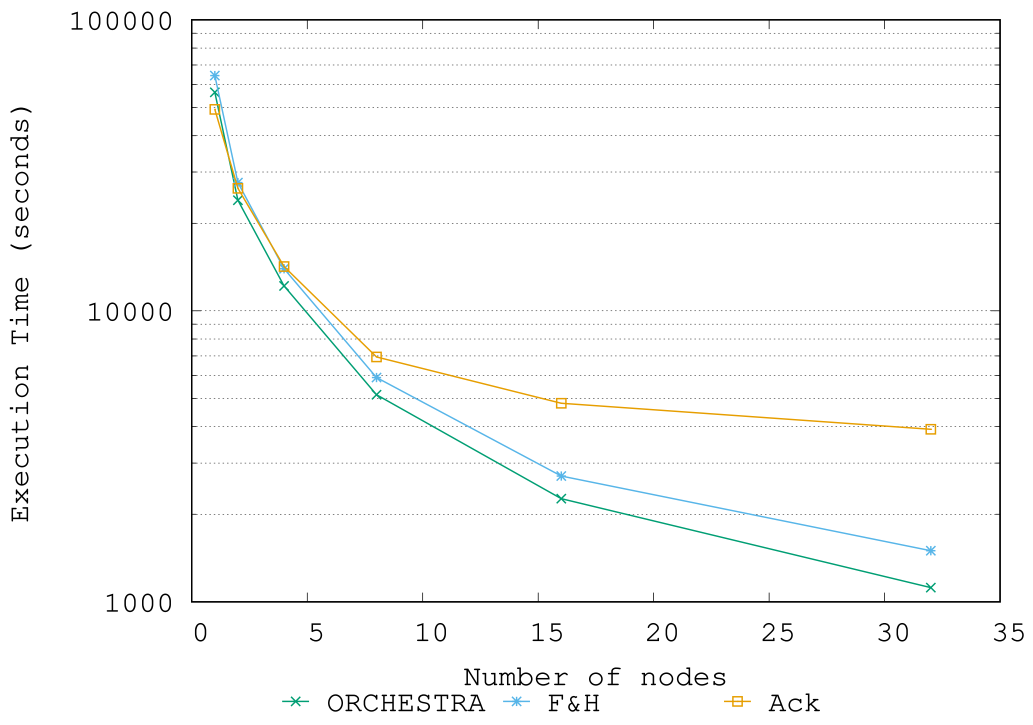


Figure 4.2: Execution time using 4096 LPs

compared: ORCHESTRA, an asynchronous algorithm where the local computation is protected by critical sections, according to the Fujimoto & Hybinette algorithm in [8] (referred to as F&H in the plot), and an acknowledgement-based reduction inspired to the work in [14] (referred to as Ack in the plot).

With respect to F&H, we observe that ORCHESTRA allows to reduce the wall-clock-time required to complete the run up to 25%. This phenomenon is strictly related to the higher overhead paid by F&H on the local (intra-kernel) computation, in terms of CPU-time required to run tasks related to GVT computation. In fact, since on a single node there are 16 concurrent worker threads active, the likelihood of synchronizing on the GVT-reduction critical section increases. This is explicitly avoided by the phase-based wait-free nature of ORCHESTRA.

A slightly different behavior is observed with respect to Ack. In fact, when running with a small number of distributed nodes (namely, 1 or 2 nodes), Ack is able to deliver a performance which is slightly better, on the order of 12%. This is related to the fact that the communication overhead paid to acknowledge in-transit messages is quite reduced, while the steps required to compute the GVT value

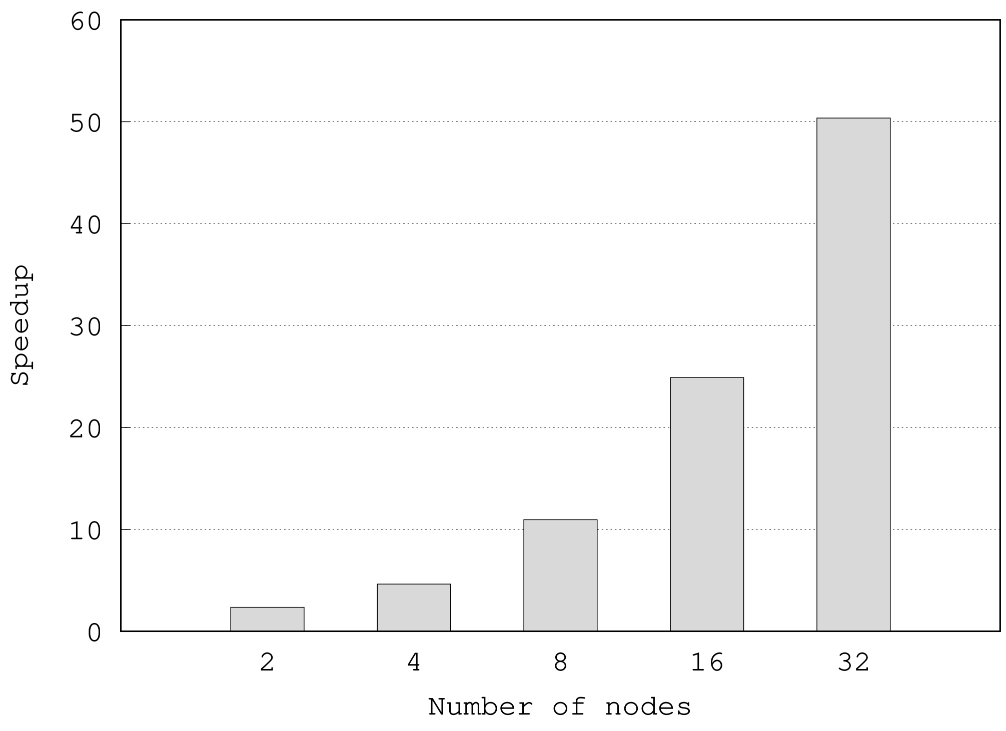


Figure 4.3: Speedup using 4096 LPs

are much simpler than in the case of ORCHESTRA. On the other hand, when the degree of concurrency increases, ORCHESTRA is able to deliver a performance increase up to 70%. This is clearly related to the fact that the communication overhead in ORCHESTRA is significantly reduced.

This phenomenon is confirmed by the results in Figure 4.3, where we report the speedup obtained when increasing the number of distributed computing nodes, with respect to the performance obtained when using one single computing node. As it can be seen, when the number of distributed nodes is higher than 2, we observe a super-linear speedup, thanks to the increased overall size of caches across the distributed nodes. For further details about the reasons behind this behavior, the reader is invited to look at Sec. 4.4.2

Figure 4.4 reports data related to ORCHESTRA's scalability when running with a much more increased workload. In particular, the plot shows the simulation execution time when PCS is configured to run 16,348 LPs. By the results, we observe a maximum speedup of 35, when running with 16 nodes, and a trend which is comparable to the one shown in Figure 4.2, denoting that ORCHESTRA's performance be-

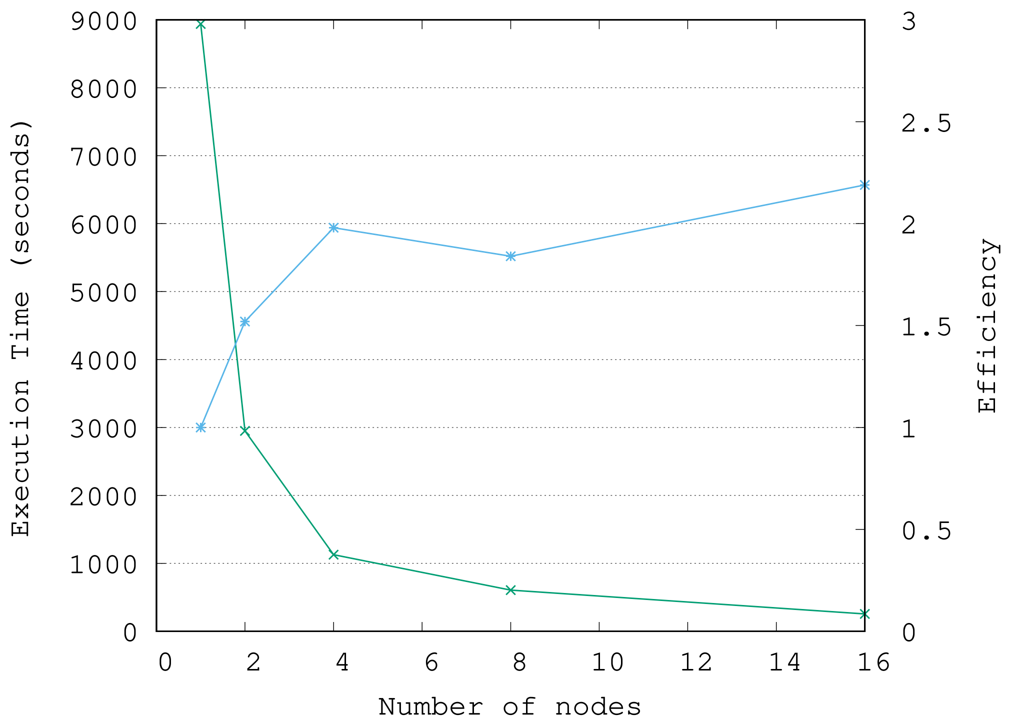


Figure 4.4: Execution time using 16.348 LPs

havior is resilient to a non-minimal scaling up of the workload. The plot shows as well the parallel efficiency of the distributed execution, namely *speedup*/*n*, where *n* is the number of distributed nodes. By the results, we can see that ORCHESTRA has a *strong scaling behaviour*, as the efficiency increases without the need for increasing the size of the simulation model.

To complete the experimental assessment, we present in Figure 4.5 a comparison between the time required by ORCHESTRA, F&H, and Ack to complete the reduction of a GVT value, i.e. the time passed from the initiation of a GVT round and the completion of the same round. By the results, we observe that Ack requires always a smaller time to compute the value of the GVT. This is because, as mentioned earlier, most of the burden to synchronize the computing nodes is placed on frequent communication. Therefore, the actual GVT algorithm is much simpler and can compute the reduction more quickly. Nevertheless, the synchronization cost is distributed over the whole simulation run, thus delivering the smaller performance which has been previously shown in Figure 4.2. Similarly, the GVT round time for F&H is higher than that of ORCHESTRA. This is an additional indication

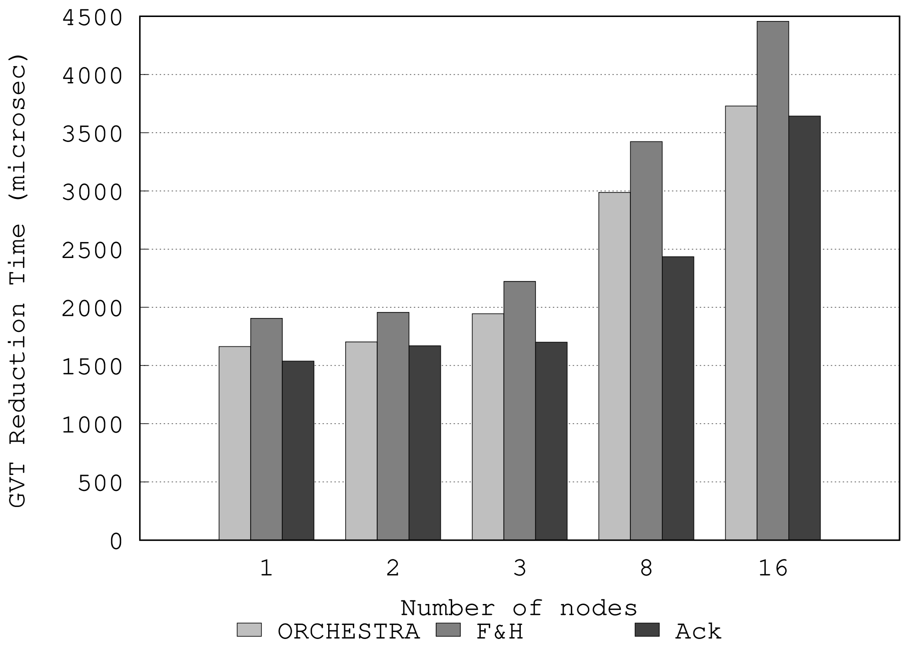


Figure 4.5: GVT round completion time using 16.384 LPs

of the fact that the wait-free nature of ORCHESTRA is able to reduce the cost of computing local GVT candidates.

4.4 In-depth analysis with Paraver

The analysis of the communication pattern among the agents along with the study of the timing of the most important events occurring during the simulation are a very useful practice to discover inefficiencies and/or anomalies of the software. Due to the distributed nature of the simulation and the high rate of exchanged messages, carrying on this kind of analysis can be really challenging. Usually, the biggest obstacle is represented by the impossibility of correlating events occurring on different machines with no-synchronized clock without interfering on the actual simulation operations. A lot of tools, aimed at the analysis of distributed systems, operate by inserting synchronization barriers in strategic point and adding additional overhead upon the communication channels, producing an altered report with a messed up events timeline. On the other hand, one common method to conduct software performance analysis is based on correlating the source code (usually on per function or per line basis) with performance metrics such as the execution time or cache misses in order to highlight the most expensive or most executed code regions. This kind of approach usually provides a very limited and specific view of the behavior of the application that very often leads the analyst to erroneous conclusion about the source of the problem.

To conduct the performance study of the new ORCHESTRA algorithm we instrumented the ROOT-Sim platform using Paraver [17], a performance visualization and analysis tool. Instead of focusing on the source code, this application, provides a graphical interface to navigate and visualize performance metrics on a line based graph. Paraver provides a qualitative global perception of the application behavior by visual inspection and allow to focus on the detailed quantitative analysis of the problems. It works by extrapolating data from raw trace files with a custom format that can be generated with the help of Extrae package [3]. Extrae takes advantage of multiple interposition mechanisms to add monitors into the application in order to probe hardware/software counters and generate the final trace file. More specifically, the "*Linker preload*" method has been used to conduct this analysis; the operating systems is exploited to inject a shared library into an application before the application is actually loaded. If the library that is being preloaded provides the same symbols as those contained in shared libraries of the application, such symbols can be wrapped in order to inject code in these calls. In Linux systems this technique can be put in place by using the LD_PRELOAD environment variable. Extrae contains substitution symbols for many parallel runtimes, as OpenMP, pthread, CUDA accelerated applications, and MPI applications.

4.4.1 Communication pattern

The first step taken to tackle the performance study has been the analysis of the communication pattern among the different kernels and threads participating into the simulation. The model that has been used to conduct the target simulation is the Personal Communication System (PCS) real-world cellular simulation model, the same used for the benchmarking (Sec. 4.3).

A very first example of communication pattern is depicted in Figure 4.6, with the purpose of getting the reader used to Paraver's timelines. The graph is built upon a trace of a PCS simulation run among two kernels with one thread each

[3]Extrae tool home page - `https://tools.bsc.es/extrae`

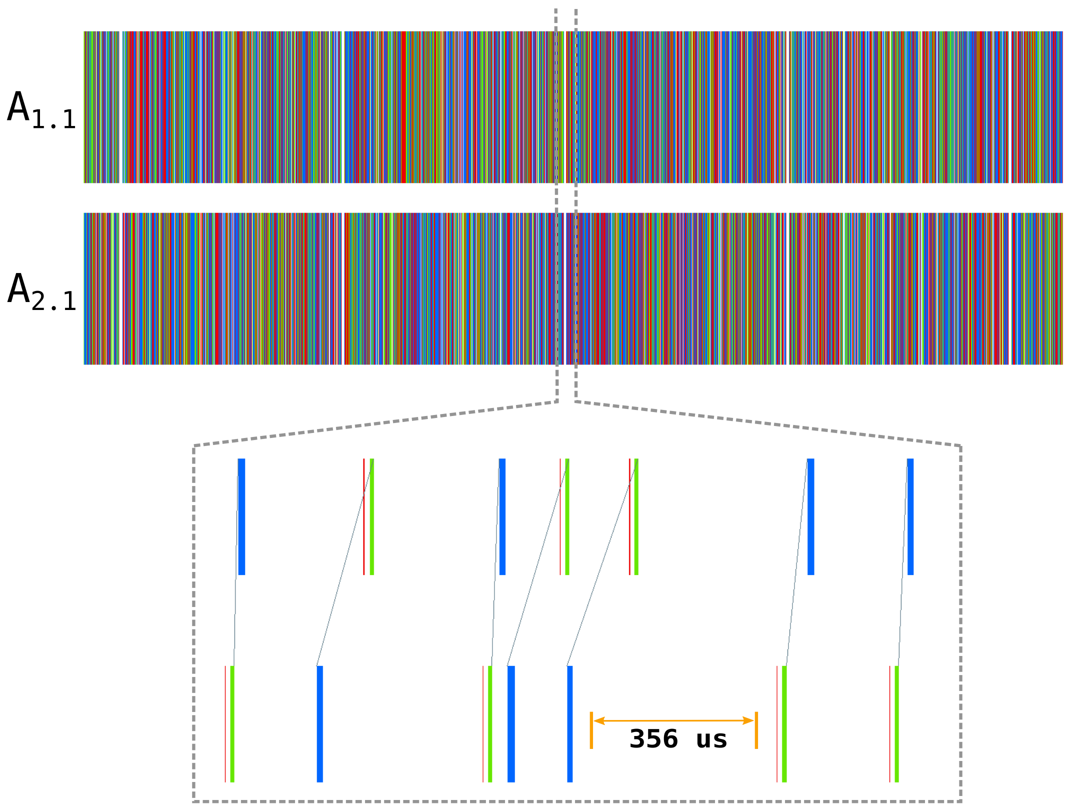


Figure 4.6: Communication pattern between two kernels with one thread each

($A_{1.1}$ and $A_{2.1}$). The timeline is populated with the different events related to communication operations, each of them is represented by a rectangle, characterized by a width that is proportional to the event duration time and a color. These two timelines are actually very difficult to interpret because the events rate is too high with respect to the time scale used, making complex to distinguish the different events. However, looking the graph at this time scale, it is pretty evident that both the threads play a very similar role, at least from a communication point of view, in fact the colors distribution is homogeneous on both. In the same Figure 4.6, a portion of the same graph is reported at a different time scale, in which single events are more easy to distinguish. To give a clue of the message exchange rate between the two threads, the inter-event time distance of $356\ \mu s$ is explicitly reported. In Figure 4.7 is depicted a single message exchange example between the two threads; $A_{1.1}$ performs non-blocking send of the message through the `MPI_Isend` function (blue colored), on the other hand, as soon as $A_{2.1}$ reach the procedure to check for new incoming message it will invoke the non-blocking `MPI_Iprobe` procedure

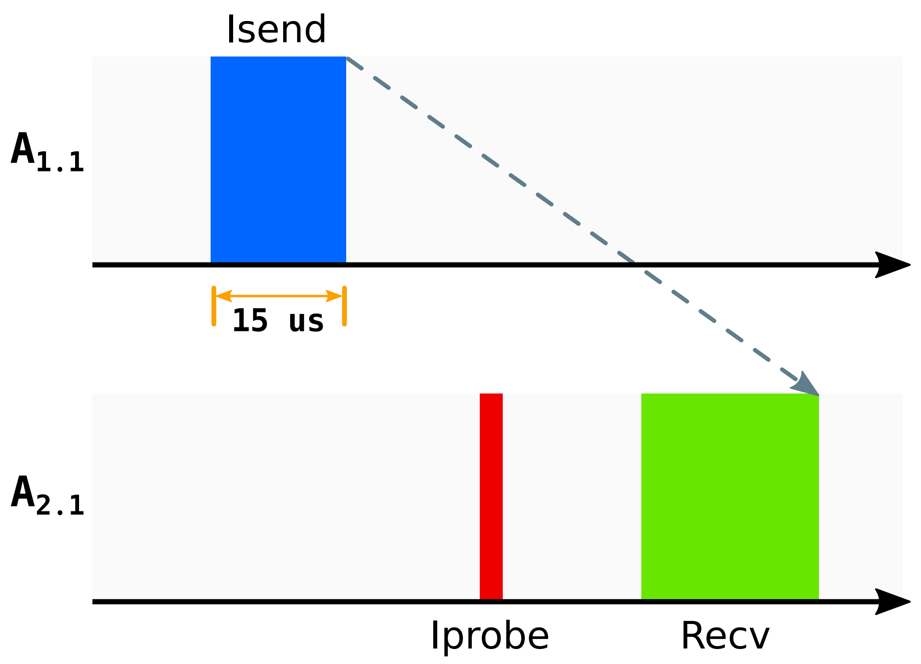


Figure 4.7: MPI calls communication graph example

(red colored). In the case there are incoming message pending at the time the probe is invoked, the receiver will start to actually incorporate them through the `MPI_Recv` synchronous call (green colored).

In Figure 4.8 another communication graph is shown, where 16 kernels with one thread each exchange messages. From these timelines, which represent a time window of about $6637\mu s$, it is possible to notice that almost all communications happen between near neighbors. There is another visualization approach exposed by Paraver that is very helpful to better comprehend the communication pattern, this view, reported in Figure 4.9, is actually a communication matrix in which every cell (X, Y) contains the number of messages sent by thread X to thread Y. All the communications are concentrated around the main diagonal, as it turns out that every thread communicates only with the previous kernel and the subsequent one. This communication pattern is the result of the distribution of the LPs among the threads and the hexagonal subdivision of the terrain assumed by the PCS model. The only exception to this pattern is represented by the thread residing on the first kernel (first row in the matrix) that during this trace performed 4 sends to all the other kernels, in fact, this kernel is in charge of notifying the start of a new GVT round to the others, indeed, during the represented trace, 4 GVT rounds have been

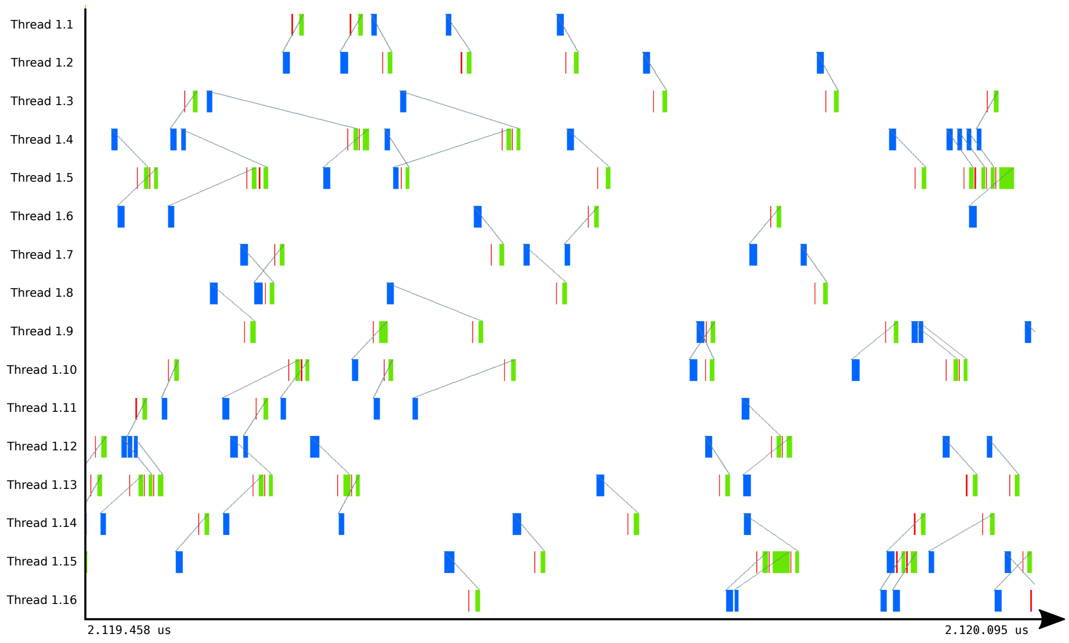


Figure 4.8: Communication graph between 16 kernels with one thread each

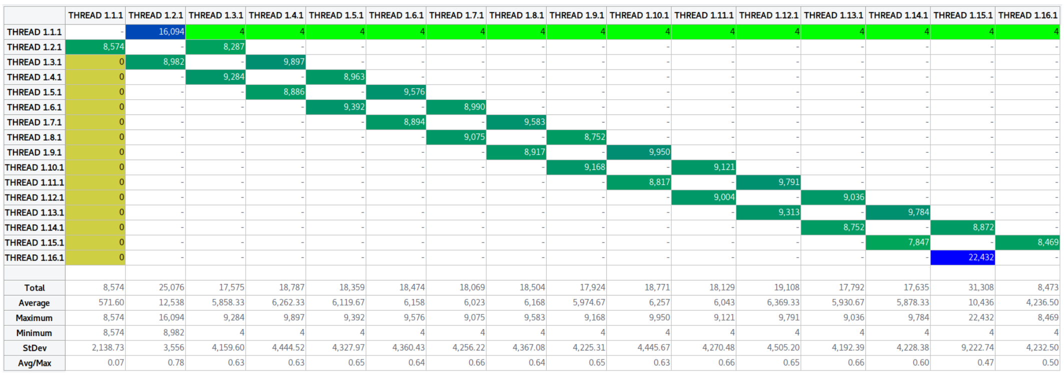
performed.

Multi-thread communications

Paraver has been also exploited to prove that the Open MPI implentation, was actually making use of the new multi-threading capabilities. The timeline of Figure 4.10 represent the trace of a simulation carried on by 3 kernels with 4 threads each, in which the concurrent calls to the MPI library API from threads residing on the same kernel have been highlighted with a purple circle. As we can see from the figure, it happens for instance that while a thread is sending a message through an `MPI_Isend` another thread on the same kernel is checking for new incoming messages through the `MPI_Iprobe` or even receiving the messages through the `MPI_Recv` ; as already discussed the concurrency provided by the underlying communication library allows to reach a very high degree of parallelism inside the kernel and to improve the overall scalability of the platform.

4.4.2 Superlinear speedup investigation

In order to better understand the behavior of the platform related to the strong scaling benchmarking (Sec. 4) and in particular to comprehend the reasons behind



	THREAD 1.1.1	THREAD 1.2.1	THREAD 1.3.1	THREAD 1.4.1	THREAD 1.5.1	THREAD 1.6.1	THREAD 1.7.1	THREAD 1.8.1	THREAD 1.9.1	THREAD 1.10.1	THREAD 1.11.1	THREAD 1.12.1	THREAD 1.13.1	THREAD 1.14.1	THREAD 1.15.1	THREAD 1.16.1
THREAD 1.1.1	-	16,094	4	4	4	4	4	4	4	4	4	4	4	4	4	4
THREAD 1.2.1	8,574	-	8,287	-	-	-	-	-	-	-	-	-	-	-	-	-
THREAD 1.3.1	0	8,982	-	9,897	-	-	-	-	-	-	-	-	-	-	-	-
THREAD 1.4.1	0	-	9,284	-	8,963	-	-	-	-	-	-	-	-	-	-	-
THREAD 1.5.1	0	-	-	8,886	-	9,576	-	-	-	-	-	-	-	-	-	-
THREAD 1.6.1	0	-	-	-	9,392	-	8,990	-	-	-	-	-	-	-	-	-
THREAD 1.7.1	0	-	-	-	-	8,894	-	9,583	-	-	-	-	-	-	-	-
THREAD 1.8.1	0	-	-	-	-	-	9,075	-	8,752	-	-	-	-	-	-	-
THREAD 1.9.1	0	-	-	-	-	-	-	8,917	-	9,950	-	-	-	-	-	-
THREAD 1.10.1	0	-	-	-	-	-	-	-	9,168	-	9,121	-	-	-	-	-
THREAD 1.11.1	0	-	-	-	-	-	-	-	-	8,817	-	9,791	-	-	-	-
THREAD 1.12.1	0	-	-	-	-	-	-	-	-	-	9,004	-	9,036	-	-	-
THREAD 1.13.1	0	-	-	-	-	-	-	-	-	-	-	9,313	-	9,784	-	-
THREAD 1.14.1	0	-	-	-	-	-	-	-	-	-	-	-	8,752	-	8,872	-
THREAD 1.15.1	0	-	-	-	-	-	-	-	-	-	-	-	-	7,847	-	8,469
THREAD 1.16.1	0	-	-	-	-	-	-	-	-	-	-	-	-	-	22,432	-
Total	8,574	25,076	17,575	18,787	18,359	18,474	18,069	18,504	17,924	18,771	18,129	19,108	17,792	17,635	31,308	8,473
Average	571.60	12,538	5,858.33	6,262.33	6,119.67	6,158	6,023	6,168	5,974.67	6,257	6,043	6,369.33	5,930.67	5,878.33	10,436	4,236.50
Maximum	8,574	16,094	9,284	9,897	9,392	9,576	9,075	9,583	9,168	9,950	9,121	9,791	9,036	9,784	22,432	8,469
Minimum	8,574	8,982	4	4	4	4	4	4	4	4	4	4	4	4	4	4
StDev	2,138.73	3,556	4,159.60	4,444.52	4,327.97	4,360.43	4,256.22	4,367.08	4,225.31	4,445.67	4,270.48	4,505.20	4,192.39	4,228.38	9,222.74	4,232.50
Avg/Max	0.07	0.78	0.63	0.63	0.65	0.64	0.66	0.64	0.65	0.63	0.66	0.65	0.66	0.60	0.47	0.50

Figure 4.9: Communication matrix between 16 kernels with one thread each

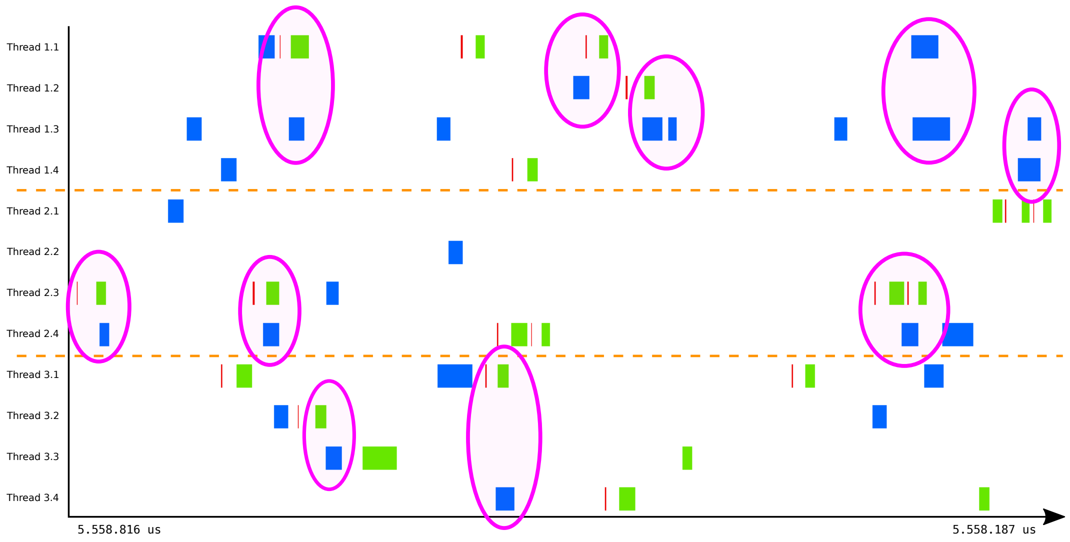


Figure 4.10: Communication graph of 3 kernels with 4 threads each, the concurrent MPI calls inside the same kernel have been highlighted

the superlinear speedup more experiments have been collected and analyzed with the Paraver analysis tool.

For this specific investigation a modified version of the Personal Communication System (PCS) real-world cellular simulation model has been used. In fact, in pursuance of a more homogeneous workload between the kernels and in order to avoid the ring fashion communication pattern (Sec. 4.4.1) the model has been modified to use a mesh network interconnection between all the hexagonal cells. Four different simulation runs have been conducted and the relative traces have been collected, following the strong scaling principle we keep the number of LPs fixed while increasing the number of kernels (2, 4, 8, 16). In order to reduce the size of the traces and to focus our attention to the inter-kernel interaction we decided to allocate just one thread per kernel

We first started analyzing the *instruction per cycle* performance metrics but we didn't notice any correlation between the variance of this parameter and the superlinear speedup, in fact, it turns out that the IPC it is almost stable around the value of 1.10. Increasing the number of kernels make the IPC to lower a little without a significant variance.

We also analyzed the *useful duration* metric that represents the computation time spent outside communication procedure; this metrics can be used to reveal how the overall time spent into communication varies with respect to the number of kernel participating in the simulation. As expected, we found out that increasing the number of kernels percentage of useful time reduces, making also this metrics not correlated to the superlinear speedup behavior.

Cache analysis

Given the very simple data structures used by the platform, the memory access pattern and the repartition of the LPs among the increasing number of kernels in the strong scaling test, we decided to focus our investigation toward the usage of the memory cache. The memory analysis has been conducted by collecting the *L1 data cache miss ratio* metric, that represents the fraction of accesses to the L1 data cache that produced a miss. The instruction cache has not been taken into account for this specific study because the instructions executed by each kernel on the different strong scaling tests are almost the same.

Figure 4.11 depicts the histograms of *L1 data cache miss ratio* relative to the 4 different runs conducted during the strong scaling test with 2, 4, 8 and 16 kernels. Each histograms is composed of a number of rows equal to the number of kernels participating to that specific simulation. Every cell holds a specific position on the Y axis that expresses a value of the *L1 data cache miss ratio* metric between 1 and 50, the values outside this range are irrelevant for the sake of this analysis and they have been omitted from the histograms. The color of a particular cell at position Y express visually the sum of all the time units of the simulation characterized by the same *L1 data cache miss ratio*, the color gradient goes from light green, that correspond to a very low time value, up to dark blue, that represents long time. The gray cell corresponds to a time sum of zero, meaning that the kernel didn't spend

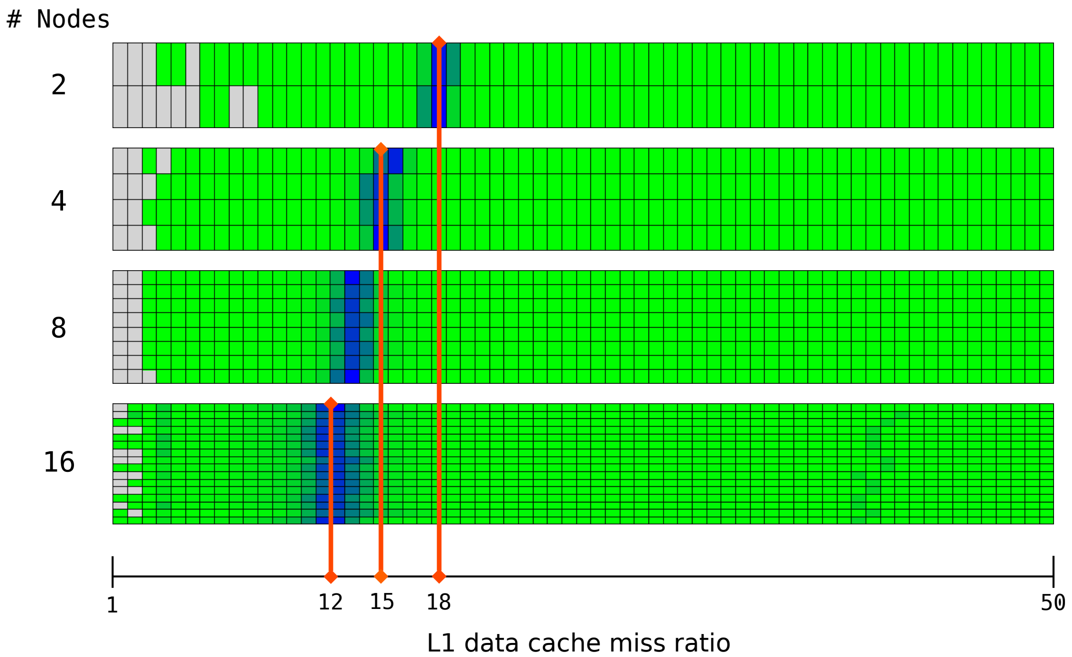


Figure 4.11: Histograms of the *L1 data cache miss ratio* relative to strong scaling tests with 2, 4, 8 and 16 kernels

any time unit with that specific miss ratio. Some of the more interesting miss ratio values (vertical orange lines) are highlighted in Figure 4.11; the histogram of the first run, composed of just two kernels, visually suggests that in the biggest percentage of time (blue cells, almost 12 seconds) both the kernels have experienced a cache miss ratio of 18.The image shows clearly that increasing the number of kernels in the subsequent runs make the average number of misses relative to the L1 data cache diminish. In fact, as we can see from the histogram of the last run, the average cache miss ratio is around 12.

The very same data have been presented also in the form of distributed graph in Figure 4.12, on the x axis, we have the cache-miss ratio over the amount of machine instructions executed in a certain period of time. On the y axis, we have the total execution time in which a certain cache-miss ratio is observed in the simulation. By this plot we can observe that:

- the value of the mean time spent in cache-miss distribution interval decreases
- the cache-miss ratio value for which we observe the highest distribution time similarly decreases

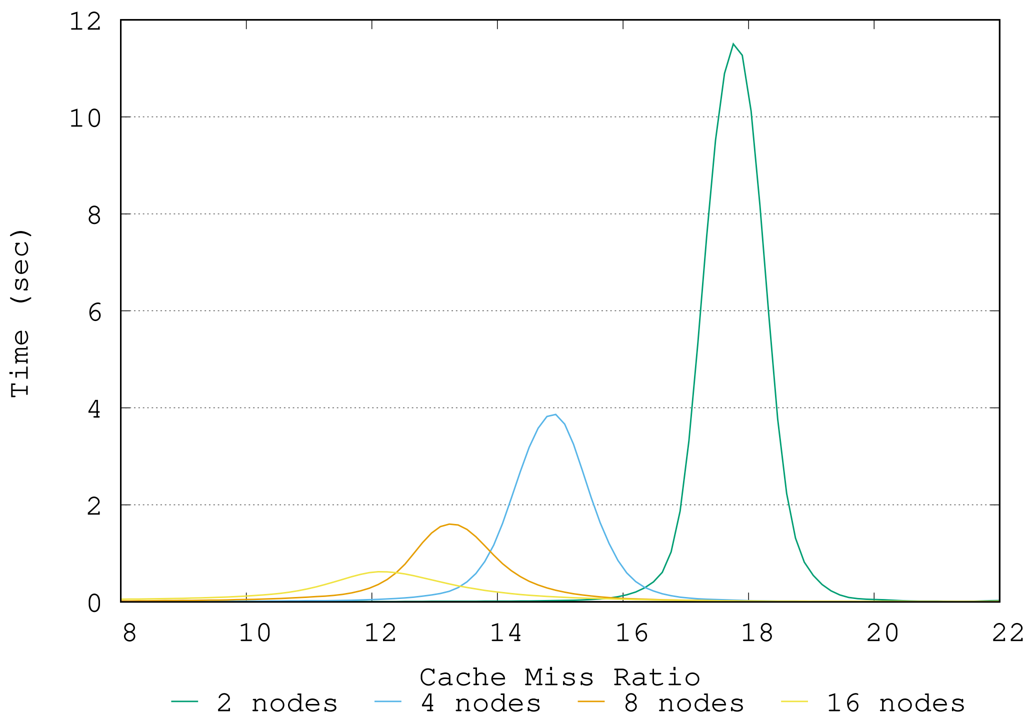


Figure 4.12: Graph of the *L1 data cache miss ratio* relative to strong scaling tests with 2, 4, 8 and 16 kernels

Therefore we can conclude that the total number of L1 data cache misses reduces while increasing the number of kernels participating into the simulation. This means that ORCHESTRA allows to effectively overcome the memory wall, since the overall memory hierarchy is exploited in a more fruitful way. In fact increasing the number of kernels by keeping fixed the number of LPs, makes every kernel having to handle a smaller number of LPs, with the direct consequence of having smaller data structures and thus a smaller residing memory set. This fruitful exploitation, given that the total size of L1 cache across all computing node is larger, is able to deliver the super-linear speedup observed in Figure 4.3.

For the sake of completeness, the *L2 data cache miss ratio* histograms of same runs as before are reported in Figure 4.13. Differently from the L1 cache, the miss rate relative to the accesses of the second level data cache remain pretty stable while increasing the number of kernels on the different simulation runs, in fact, as highlighted in the image by the orange vertical marker, the average L2 cache miss ration stays around 5 for all of the four simulation (2, 4, 8 and 16 kernels).

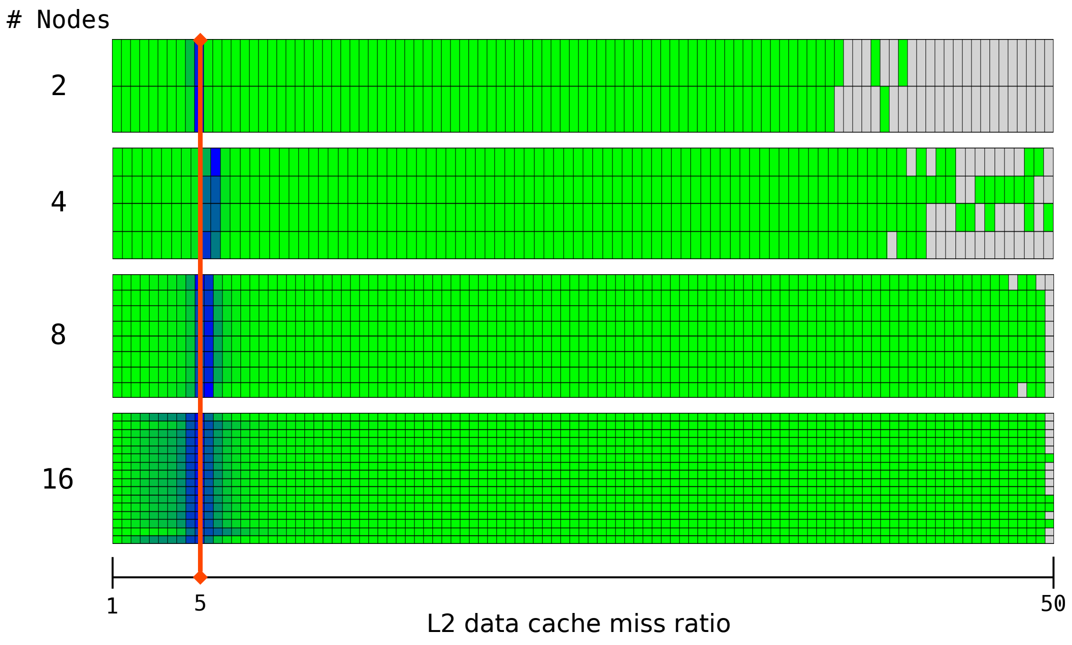


Figure 4.13: Histograms of the *L2 data cache miss ratio* relative to strong scaling tests with 2, 4, 8 and 16 kernels

Chapter 5

conclusions

The ORCHESTRA algorithm signed a step forward in the complex world of speculative simulations. The new GVT algorithm along with the architecture of the simulation platform allows for an orthogonal scaling. On one hand the number of simulation threads can be scaled to exploit all the computing units inside a single machine, while on the other hand the number of simulation kernels can be tuned in order spread the simulation among all the nodes in the clusters. This orthogonal scaling gives to the platforms the elasticity to adapt to very different cluster architectures.

Experimental results obtained on Marenostrum III supercomputer and presented in [20], have proven the scalability of the algorithm. In particular the results show a superlinear speedup on the strong scaling test. Thanks to an in-depth analysis that I conducted using the Paraver analysis tool [17], it was possible to correlate the superlinear behavior to the exploitation of the L1 data cache (a detailed version of this analysis can be found on the dissertation of my thesis).

The general organization of ORCHESTRA could be adopted by different high-performance computing applications. In particular the combined use of atomic counters, collective communications and the "*Compare & Swap*" instruction is the key to construct very efficient coordination algorithms for multi-thread distributed environments. This flexible multi-phase pattern can be implemented to orchestrate a wide variety of distributed asynchronous tasks even outside the world of simulation softwares.

Bibliography

[1] Francesco Antonacci, Alessandro Pellegrini, and Francesco Quaglia. “Consistent and efficient output-stream management in optimistic simulation platforms”. In: *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation.* PADS. ACM, 2013, pp. 315–326.

[2] Reid Baldwin, Moon Jung Chung, and Yunmo Chung. “Overlapping window algorithm for computing gvt in time warp”. In: *Parallel Algorithms and Applications* 6.2-3 (1995), pp. 93–110.

[3] Steven Bellenot. “Global Virtual Time algorithms”. In: *Proceedings of the SCS Multiconference on Distributed Simulation.* 1990, pp. 122–127.

[4] Gilbert G Chen and Boleslaw K Szymanski. “Time Quantum GVT: A Scalable Computation of the Global Virtual Time in Parallel Discrete Event Simulations”. In: 8.4 (2008), pp. 423–435.

[5] Samir R Das and Richard M Fujimoto. “Adaptive memory management and optimism control in time warp”. In: *ACM Transactions on Modeling and Computer Simulation* 7.2 (1997), pp. 239–271.

[6] Samir Das et al. “GTW: A Time Warp System for Shared Memory Multiprocessors”. In: *WSC Proceedings of the 26th Conference on Winter Simulation* (1994), pp. 1332–1339.

[7] S.K. Das and F. Sarkar. “A hypercube algorithm for GVT computation and its application in optimistic parallel simulation”. In: *Proceedings of Simulation Symposium.* IEEE Comput. Soc. Press, 1995, pp. 51–60.

[8] Richard M Fujimoto and Maria Hybinette. “Computing Global Virtual Time in Shared-Memory Multiprocessors”. In: *ACM Transactions on Modeling and Computer Simulation* 7.4 (1997), pp. 425–446.

[9] Edgar Gabriel et al. “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”. In: *Proceedings, 11th European PVM/MPI Users’ Group Meeting*. Budapest, Hungary, Sept. 2004, pp. 97–104.

[10] Tim Harris, Keir Fraser, and Ian a. Pratt. “A practical multi-word compare-and-swap operation”. In: *Distributed Computing* (2002), pp. 265–279.

[11] David R Jefferson. “Virtual Time”. In: *ACM Transactions on Programming Languages and System* 7.3 (1985), pp. 404–425.

[12] Sunil Kandukuri and Stephen Boyd. “Optimal Power Control in Interference-Limited Fading Wireless Channels with Outage-Probability Specifications”. In: *IEEE Transactions on Wireless Communications* 1.1 (2002), pp. 46–55.

[13] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. Vol. 78. 1988, pp. 1–217.

[14] Yi-Bing Lin and Edward D Lazowska. “Determining the global virtual time in a distributed simulation”. In: *Proceedings of the 19th International Conference on Parallel Processing*. Ed. by Benjamin W Wah. ICPP. Pennsylvania State University Press, 1990, pp. 201–209.

[15] Friedemann Mattern. “Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation”. In: *Journal of Parallel and Distributed Computing* 18.4 (Aug. 1993), pp. 423–434.

[16] Alessandro Pellegrini and Francesco Quaglia. “Wait-Free Global Virtual Time Computation in Shared Memory TimeWarp Systems”. In: *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. IEEE, Oct. 2014, pp. 9–16.

[17] Vincent Pillet et al. “Paraver: A tool to visualize and analyze parallel code”. In: *Proceedings of WoTUG-18: transputer and occam developments*. Vol. 44. 1. IOS Press. 1995, pp. 17–31.

[18] Behrokh Samadi. “Distributed Simulation, Algorithms and Performance Analysis (Load Balancing, Distributed Processing)”. PhD thesis. 1985.

[19] H Sutter. “The free lunch is over: A fundamental turn toward concurrency in software”. In: *Dr. Dobb’s Journal* (2005), pp. 1–9.

[20] Tommaso Tocci et al. “ORCHESTRA : An Asynchronous Wait-Free Distributed GVT Algorithm”. In: (2017).