SAPIENZA
UNIVERSITÀ DI ROMA

Asymmetric Runtime Environments for Increased-Performance Speculative PDES

Dipartimento di Ingegneria Informatica
Automatica e Gestionale "Antonio Ruberti"

Master of Science in Engineering in Computer Science

Candidate

Lorenzo Altamura
ID number 1538468

Thesis Advisor

Prof. Alessandro Pellegrini

Co-Advisor

Prof. Bruno Ciciani

Academic Year 2018/2019

Thesis defended on 20 January 2020
in front of a Board of Examiners composed by:

Prof. Riccardo Rosati (chairman)
Prof. Aris Anagnostopoulos
Prof. Riccardo Lazzeretti
Prof. Alessandro Pellegrini
Prof. Leonardo Querzoni

Asymmetric Runtime Environments for Increased-Performance Speculative PDES
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Version: January 15, 2020

Author's email: altamura.1538468@studenti.uniroma1.it

Acknowledgements

I would like to thank Prof. Alessandro Pellegrini for providing me with all the support I needed throughout all these months in which I've learned so much.
I would also like to thank Prof. Bruno Ciciani for the precious advice and Dr. Stefano Conoci for the help offered during the initial phases of this work.

My deepest gratitude goes to everyone who supported me along this journey, to my parents, sister and grandmother and to my dearest friends Jonattan and Matteo.
A special recognition is addressed to my colleagues at DIAG who shared the path with me, especially to Roberto Adduci, Roberto Avagliano, Paolo Mastrobuono Battisti, Domenico Ciampa, Luca Deodati, Andrea Fantoli, Moreno Labbate, Chiara Mele, Marzio Monticelli, Riccardo Ostani and Riccardo Vecchi.

Abstract

Future exascale systems will require runtime environments able to manage the complexity of the underlying heterogeneus hardware. This thesis discusses about asymmetric features in existing high performance applications to obtain consistent increase in terms of performance by properly exploiting the asymmetry shown by current pre-exascale systems. In particular, the focus is on parallel discrete events simulation (PDES) and possible solutions to best exploit asymmetry in threads by limiting the drawbacks in terms of the overhead brought by the time warp optimistic synchronization protocol.

Experimental data show how relevant the gain in terms of performance is when self-adjusting algorithms autonomically manage the balance between asymmetric thread incarnations.

Contents

Chapter 1

Introduction and Context

In applied sciences, "simulation" plays a very important role as it is used as a mathematical, logical or scientific representation of any form of real phenomenon, may it be a system, an entity or a process. In cases where it is impossible or not convenient to reproduce in laboratory the real conditions implied in the study, simulation builds upon mathematical rules a reproduction of the case study, with dynamics and mechanics modeled as close as possible to the real system: the closer to reality the simulation is, the more reliable the analysis on behaviors will be. Comprehension of case studies is easier this way, as it implies less costs and a virtually unlimited amount of data to analyze. Anyway, this approach incurs some unavoidable approximations and assumptions that sometimes affect the validity of the outcomes.
In recent times, the use of computers widely spread into these practices, bringing an outstanding amount of benefits: for example, mathematical models were once used to solve the problem by finding analytical solutions predicting the behavior of the system by manipulating variables into extremely complex equations; computers helped to realize models with a much wider scope and reliable results.
Physical and interactive simulations involve the creation of a physical object, usually smaller and/or cheaper than the original, that takes the place of the actual object. For example, aircraft pilots train on flight interactive simulators which are safe and considerably less expensive than an actual plane, especially in case of modern warplanes. **Continuous simulations** refer to a computer model of a physical system built around differential equations in which variables change continuously in function of time and system state is updated continuously. Since it's impossible to represent continuous time flow, time is increased in small steps to simulate the

correct behavior. **Monte Carlo** methods are a class of algorithms that rely on continuous application of random sampling to obtain numerical results. Those methods use randomness to solve problems that originally were deterministic. They usually apply to physical and mathematical problems and are most useful when it is difficult or impossible to use different approaches.
Simulation has been classified into many subcategories that describe the methodologies applied to model the case studies (Fig. 1.1).

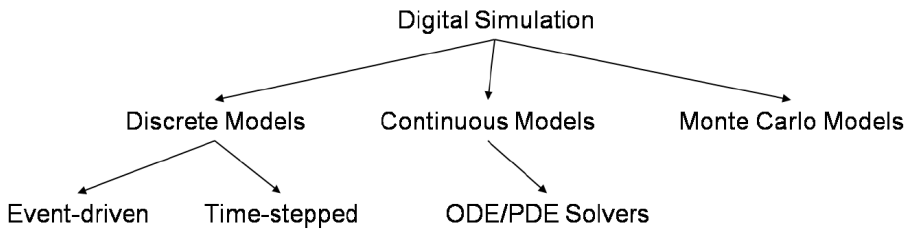


Figure 1.1. Simulation Taxonomy

Simulations can be roughly split into the following two categories:

- **Stochastic simulations** depend on variables drawn from some probability distribution. Any round of the simulation produces different results and, as outputs are produced, final outcomes converge into a distribution of such outputs that is the result in which the simulation with a pre-determined set of values may fall in.
- **Deterministic simulations** are basically the opposite of stochastic ones: values for the variables are fixed and the outcomes are always supposed to be the same.

Actual implementations of the various methods may follow different paradigms.
A **distributed simulation** is about distributing a single simulation run across multiple processors.
As described by Fujimoto in [1], there are many advantages in a distributed simulation: in a paradigm known as "parallel execution", the run is performed over a tightly coupled computer system (e.g. a supercomputer or a shared memory multiprocessor) and the main reason for the distribution is to reduce the execution

time of the simulation. The great advantage in sharing the execution between N processors stands in the possibility to achieve a theoretical maximum speed increase of (Amdahl's law applied to parallelism)

$$\frac{1}{(1-F)+\frac{F}{N}}$$

compared to a single processor run, where F is the portion of the code that can be parallelized and *1-F* the part that remains sequential.

One more advantage resides in the possibility to allow a particularly big execution to run: if limited to a single computer, memory may not be enough to support the execution of that particular run, while distributed execution allows the exploitation of a hugely bigger pool of memory.

A second motivation is the necessity to build a big cooperative simulation environment starting from several different simulators: this necessity can be found for example in military training simulators, flight simulators and various other models where personnel need to train for hypothetical scenarios and situations.

Recently, this kind of simulations has spread into different scopes in modern society. In both military and infrastructure simulations, it is far more convenient to create a bond between existing simulations than to create a brand new one within the context of a single software. Simulation distribution allows personnel to operate remotely through internet and cuts lot of the costs; at the same time, high performance simulations have now the possibility to execute on multiprocessor computers situated in a single cabinet or in a room, as proximity is needed to remove the inefficiencies caused by the delay that occurs during remote communication between processors.

The terminology may be confusing: the term "parallel simulation" was once used to qualify simulations executed on a tightly coupled parallel computer, while "distributed simulation" was about physically distributed simulations; with new computer paradigms (e.g. clusters of workstations or grid computing) the distinction became less clear. From now on, "distributed simulation" may refer to both.

One more paradigm is **sequential simulations** and implies a single-threaded execution.

1.1 Discrete-event simulation

Discrete-event simulations (DES) manages the execution of the model as a sequence of impulsive events, where every event occurs at a certain time instant and nothing occurs between two of them.
Traditionally, discrete-event simulations employ an inherently sequential algorithm, but large simulation models are limited by this sequentiality because of the limited amount of computing resources which are exploited. An interesting overview by S.Robinson [2] describes DES as one of the most popular modeling techniques that has spread in the '50s and kept evolving until today, where it is applied in visual interactive modeling, simulation optimization, virtual reality, distributed simulation and the worldwide web. DES features the following characteristics:

- **Time** in DES advances following two possible approaches: fixed-increment time progression, in which time is divided in a series of fixed intervals and the state is updated basing on activities happening between these time slices, and next-event time progressing, the most used as typically runs much faster, in which the simulation time jumps between the timestamps of the processed events.

- A system **state** is needed to keep track of the main properties of the model by keeping a set of explanatory variables. One node can change the current state by **processing events**, and generate new events destined for other nodes in the system as a consequence. Fan-out is a messaging characterization to manage the quantity of events injected inside the system.

- In DES, a **clock** is used to keep track of the current simulation time by taking into account the instantaneous "hops" caused by events processing.
  Every event is timestamped so that the system evolves by processing the events in their timestamp order and it's forbidden for DES models to generate events in the virtual past.
  Time is measured by whatever measure unit is chosen by the simulation designer. At least one **pending event set** is required to store and keep track of all those events that are waiting for their turn to be executed. As already stated, every event has his own timestamp and must be executed according to a timestamp-based chronological order over all events.

Usually, these sets take the form of priority queues sorted by event timestamp: the events are inserted in the queues in the order they arrive, but are always picked in strict chronological order.

- The simulation begins with an **initial state** and some default starting events as input; processing these initial events triggers the generation of more events to be injected inside the system. When new events arrive into the input channel of a node, the node itself updates its output and generates new events to be sent to the nodes in its fan-out.

- **Ending conditions** are determined by the designer, since the simulation may run forever as it could be a model of a physical system. Typically, parameters that determine when simulation should stop are (i) time (i.e. "at time t simulation must stop"), (ii) number of processed events and, generally speaking, (iii) when a specific event occurs (e.g. measure x is greater than a value v or whatever condition over the state, even more complex than this).

- The **main loop** of the execution is the core of the simulation and repeats the following steps until one of the ending conditions is satisfied.
  During the **first step** the "next" event to be processed is picked.
  Second step sets the clock to the next slice or to the next-event time, basing on the "time" pattern chosen.
  Third step executes the next event and removes it from the event list.
  Last step carries out the message delivery procedure that takes the messages to their receivers.

1.2 Parallel discrete-event simulation

Parallel computing is a computation technique where processes calculations are carried out simultaneously on different processing units, for example by splitting large problems into smaller ones. Parallelism has been intensively used in high-performance computing in the past, but now is one possible answer to a variety of physical constraints, such as power consumption, that have become a concern for computers in recent years. Power wall and memory wall [3] are two problems that affect evolution in computer infrastructures and that need to be faced. Power wall refers to the theoretical peak of power that a system can sustain: miniaturization of transistors is leading to an extremely high density of them and, consequently, even if modern productive processes achieved a very low electricity consumption over single units, the huge number of simultaneous switching of these transistors causes a tremendous increase of the temperature: more and more expensive cooling systems are asked to keep the hardware at a proper working temperature, a task that is getting very hard to accomplish. In extreme cases, CPU throttling is an automatic system that reduces operating frequencies to avoid damages to the hardware caused by high temperatures.
Memory wall, on the other way, was initially theorized in 1994 by Wulf and McKee. They predicted that sometime in the future, RAM won't be fast enough to keep up the pace of CPUs frequencies; that means that program execution time will depend almost entirely on the speed at which RAM can send data to the CPU. Linked to the memory speed problem, the increase of data size also requires bigger and bigger size of memory over time, and this is actually a different limit to be faced. With these two problems to deal with, distributed execution in general (and PDES in particular), can lighten the weight a single processor must stand by distributing the load of a single execution into many processing units, may they be cores, sets of CPUs or cluster of machines. At the same time, the use of many distributed devices allows systems to access a much bigger pool of memory.

1.2.1 Introducing PDES

Many fields of application of DES include real-world scenarios, such as engineering, computer science and military applications.
Parallel Discrete Event Simulation, or PDES, is the parallel (and/or distributed) execution of DES simulations and responds to the always growing necessity to optimize time requirements of such fields of large-scale simulations, other than try to find a solution to problems like "power wall" and "memory wall".
The introduction of parallel computing in DES implies the use of multiple processes or threads, assigned to different CPUs, to perform a single simulation. Of course, the final goal of this distribution is to reduce the execution time and increase the scalability of the simulation.
The infrastructure in which the simulation runs can be divided into two more categories.

- **Shared memory systems** (Fig. 1.2), composed by a group of CPUs that share a common memory, are the easiest to implement since communication between different processors is performed on the same memory but, at the same time, contention over shared resources may limit the scalability of the application and introduce synchronization problems.

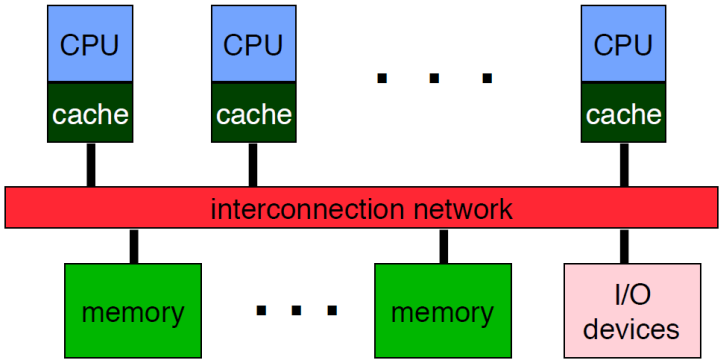


Figure 1.2. Shared Memory Machines

- In a **distributed memory system** (Fig. 1.3), the memory is associated with a set of processors which are only able to access their own memory. Many refer to this kind of architecture as a "multicomputer", since the parts that compose the whole architecture are themselves simple computer systems, complete with processor and memory.
  CPUs can only operate on local data, so it's harder to grant synchronization: a message-exchange protocol must be established to regulate the traffic between nodes, keeping into account that external factors, such network delays, may affect the causal consistency over the delivery of messages.

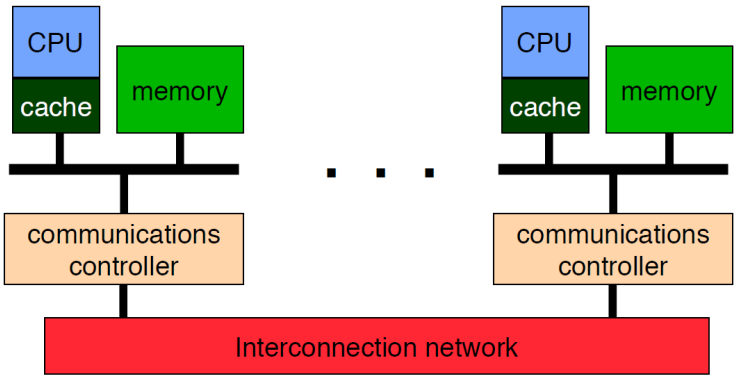


Figure 1.3. Distributed Memory Machines

Distributed Shared Memory (DSM) architectures implement features of both the previous architectures, where physically separated memories appear to the system as a single logically shared address space. Multiple independent nodes have their own memory connected in a shared interconnected network.
The synchronization protocol is invisible for the developers when implemented in an operating system but not when implemented at application level. Actually, in PDES implementations, communication is performed using message passing. The goal of a DSM system is to apply the shared-memory paradigm to a system with physically distributed memory.
DSM systems for PDES have been first studied in [4].

1.2.2 Implementation of synchronization

Thread synchronization is the mechanism that grants two or more threads or processes not to incur into simultaneous execution of some critical section of the code. Control over accesses on those section is performed by synchronization techniques that regulate the flow: basically, when one thread executes the code inside a critical section, the other threads should stop the execution and wait until the previous thread finishes: this is called "mutual exclusion". Lack of control techniques over critical section accesses leads into race conditions where the values for variables may vary in unpredictable ways, basing on the order the threads access to sensible code. Bad synchronization over threads is the origin of many issues such as the followings:

- **Deadlock** is a state of the computation where every member (i.e. thread) stops the execution waiting for some other member to complete a task, such as sending messages or releasing a lock; this happens when the state of a process is unable to change indefinitely because resources it should access to are always under use.
- **Starvation** happens when a process can never obtain access over a shared resource required to continue the computation: other processes may deny access as consequences of errors in scheduling or in the mutual exclusion algorithm.

Among all the alternatives, there are three main ways to implement synchronization over a multi-processes algorithm.

- **Spinlock**: before accessing a critical section, the processor checks a flag that states if a concurrent process already accessed that area and is still in it; if so, it will wait until this process eventually leaves the critical area and the flag is restored, by spinning in a loop and continuously checking if the flag variable changes (busy waiting). **Busy waiting** is the continuous control over a particular condition (or a series of conditions) to determine if a certain value has changed. Of course, accesses to flags need to be atomic (i.e. to grant integrity, contemporary access by concurrent threads to the resource must be avoided).

- A **barrier** is a synchronization method applied to a group of threads or processes where a process (or a thread), once incurs in a barrier mechanism, must stop the execution and wait until every process or thread gets into that same part of code.
  Barriers may be required in cases where particular areas of the code can be executed only after all the threads performed some required action.

- A **semaphore** is a structure that controls access to common resources. Every semaphore has a fixed value that says how many processes can enter the critical area. Every time a process requests the access into a section controlled by a semaphore, it checks the counter value: if it is 0, the process must wait, differently it decreases the value by some amount (wait operation) and enters the critical section. When critical operations are over, the process exits from the semaphore-controlled area and increases the value by some amount (signal operation).

1.2.3 The synchronization problem: stragglers

Due to the parallelism nature and various external factors such as network delays, the messages exchanged between threads risk to arrive at their destination without respecting the causal order in which they are supposed to be processed; that means that actions may be executed in the wrong order.
If the timestamp of the message is in the local past once it arrives at destination, it is called a "straggler message" (Fig.1.4). The system is no longer in a consistent state and needs to be brought back to a previous "safe" state undoing the effects caused by the execution of events made out of the causality order. The overall procedure is called "rollback". On the other hand, conservative synchronization approach aims to completely avoid violations on local causality constraints.

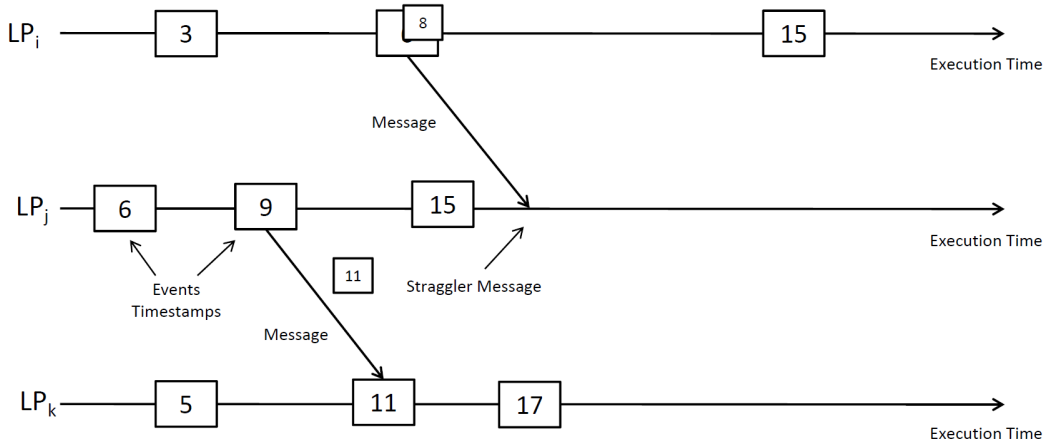


Figure 1.4. The Synchronization Problem

1.2.4 The local causality constraint

As concurrent processing may lead into different errors, PDES requires strict controls over the synchronization of the entities (LPs) in order to grant the correct results. Synchronization between LPs is violated if a LP receives a straggler (i.e. an event with arrival time smaller than the current LP clock time) and such violations are known as "causality errors". There's a single necessary and sufficient condition that needs to be satisfied in order to ensure synchronization: the "local causality constraint". This condition is respected by a discrete event simulation consisting in

LPs that communicate only by exchanging timestamped messages, if and only if each LP processes events in non-decreasing timestamp order.
To satisfy the local causality constraint, many different synchronization techniques have been designed, which generally can be split into two main categories: conservative, which are designed to avoid any kind of causality issue; and optimistic, which don't prevent causality conflicts from happening, but implement a recovery pattern to restore the system from inconsistent states. In [5] there's a review of the classical and recent efforts in the field of parallel and distributed synchronization mechanisms. Smallest time-stamp first(STF) is a scheduling strategy that could be applied to PDES that processes events in increasing order of timestamp: picking messages in the correct order is crucial in terms of local synchronization, as messages coming from a "virtual past" can cause conflicts and causality errors into the model if those singularities are not correctly managed.

1.2.5 Conservative and optimistic synchronization

The nature of PDES itself makes it impossible to find a general solution that puts an order to sequence of instructions and determines which computations must be executed first respect to the others.
Two possible synchronization schemes have been introduced to make distributed simulation respect the local causality constraint:

- **Conservative synchronization protocols** are about avoiding any kind of causality error. Nothing is processed as far as it is safe to continue the execution by checking the processing status of all the events that could affect the execution of the current event. In brief, causally linked events are never processed in the wrong order and this is achieved by using a combination of lookahead and barrier-synchronization strategies. Lookahead could be informally defined as the skill to predict the future, but Fujimoto in [6] defined it more precisely: a LP is said to contain a lookahead of L at time T if it can schedule events with timestamp at least $T + L$.

- **Optimistic synchronization protocols** do not care about possible causality conflicts the execution may encounter. Once errors are a-posteriori detected, a series of procedures bring back the system into a consistent state (i.e. rollback).

The first approach ever introduced is the conservative one: as already stated, the main concern is about determining when an event is free to be processed, and efficiency mainly depends on how good the lookahead strategy is.
Briefly, an event E_1 with timestamp T_1, can be processed only if it's sure that not a single event E_x with timestamp smaller than T_1 will appear anytime in the future.
Execution of processes with events that are not "safe" must block until the safety condition is respected: too many blocks can lead into a deadlock, if not correctly managed with deadlock avoidance mechanisms, and this is a serious concern for conservative approaches.
Due to the necessity to respect the causal order of the messages' timestamps, many conservative algorithms are prone to deadlocks. Null messages were created with the purpose to avoid deadlocks by exchanging dummy messages between LPs. Dummy messages are created only for deadlock avoidance and do not perform any simulation activity as they may be considered as a promise made by the sending LP to the receiver that every message sent after that null message will never have a timestamp bigger than the null message's one. The receiving process may use this information to decide which already received pending messages may be processed. Processes can communicate with each other to determine a global lower-bound over the timestamp of the next outgoing message on each output link.
Null messages appear to the LP like ordinary non-null messages, except no activity is simulated by their processing. Null messages are introduced by Fujimoto in [7] and [8].
We will not go deeper with the conservative approach since the higher grade of parallelism exploitation makes the optimistic approach the only possible choice for the purposes of this work: asymmetry in PDES makes sense only with an optimistic synchronization scheme.

1.3 Optimistic Synchronization

Runtime environments based on optimistic synchronization do not care of any possible causality error as they simply keep processing events until a conflict happens. It may seem to be a risky approach, but it allows the simulator to run in situations in which a causality error is theoretically very likely but, in the end, nothing bad happens. Anyway, the simulator incurs into inconsistent states, that's why designing a good recovery algorithm is the main deal of an optimistic mechanism. Optimistic synchronization borrows several concepts from the broader field of speculative execution.

Speculative execution is an optimization technique in which, in order to improve the overall performance, some action is performed well before all the inputs required to make a decision are available. A classic example is branch prediction in CPUs, where execution paths are chosen basing on which ones are likely to be required soon; data flow analysis might optimize the order in which instruction must be processed for an optimal execution. Predictions are often correct and the gain in terms of performance is real.

Anticipating instructions execution may allow CPUs to prevent delays, reduce total execution time and improve overall performance. The results may be discarded if eventually the guess was wrong: all the pipeline is flushed, the state of the program is rolled back to a certain previous state and the execution on the correct path is re-started. Speculation is employed in various fields, including branch prediction in pipelined processors, prefetching memory and files, and optimistic concurrency control in database systems. Another methodology based on speculation is that of "transactional memory". Transactional memory is an approach that promotes transactions rather than locks to synchronize parallel processes with shared memory. A transactional memory system can be implemented both in hardware and software and guarantees that, even if transactions in areas that usually would be managed by a lock are executed in parallel by different threads, the final result would be equivalent to a series of sequential operations for the same area.

Operations are performed by continuously monitoring the concurrent access to transaction variables: if a conflict between two transactions accessing the same variables is detected, it will cause one of the transactions to abort and rollback to start the procedure anew.

One of the most important goals is the increase of parallelism when a bunch of operations access some data structure: a lot of them might read to it, actually just few of them modify the data. Lock based synchronizations typically require to assure that the risk to incur in any sort of conflict is totally avoided, allowing only sequential runs to assure data integrity. As far as some process doesn't write to the data structure, transactional memory allows most of all the operations to run in parallel.

1.3.1 Time warp

Time warp is probably the most used optimistic protocol for PDES and it relies on the concept of Virtual Time.
In case of straggler messages (Fig.1.4), a "rollback" procedure manages to bring the system back into a consistent state by undoing any effect caused by the out-of-order execution of events. As mentioned, there are two possible effects that the processing of an event may cause: modification of the state of the logical process and/or sending of event messages to other processes.
Rollbacks work by bringing back the LP clock to a virtual time that comes before the straggler's timestamp and by restoring the LP state into a previous one that is safe. The appropriate previous LP state is restored by picking it from a list of checkpointed states; safe messages that come before the new bound and that need to be reprocessed are immediately reprocessed, while the others are erased by sending the matching antimessages.
Negative messages, also known as anti-messages, are introduced into the rollback mechanics to manage messages that need to be discarded. Every negative message has a corresponding positive message, where the positive one is always an event that arrived to the LP inbound channel with timestamp bigger than the straggler message's one. The work in [9] suggests the use of reverse computation to reduce the required overhead for state saving in PDES, defining an approach based on reverse event codes and demonstrating performance advantages of this approach over traditional state saving for fine-grained applications.
The key property that reverse computation exploits is that most of the operations that modify the state variables are easy reversible and such operations requires no history.

Only the most current values of the variables are required to undo the operation. At the end of the rollback procedure, when the LP's state is restored to a consistent one, a negative message is received by the LP and a match is found between the stored message that was previously received: both negative and positive messages are canceled. This kind of time warp optimization is known as "**lazy cancellation**": before an antimessage is sent, the process resumes execution from the new LVT and waits to see if the re-execution of the events regenerates the same message; if so, there's no need to cancel the message.

Even if it may look like there's the possibility for the process as a whole to stop advancing, it can be shown that the execution always goes on: all the LPs that take part in the computation are asked to periodically find an agreement upon a common time barrier (i.e. the global virtual time - 1.3.1). All the processing made before the agreed time barrier is considered to be safe. The STF scheduling mechanism always picks the LP having the lowest "next" event's timestamp on a single node, yet none of those timestamps ever come before the global virtual time, that is a value that can only advance along the computation. Anyway, an overwhelming amount of rollbacks may drastically slow down the progresses. In Fig.1.5 is depicted a scheme of the entire recovery procedure.

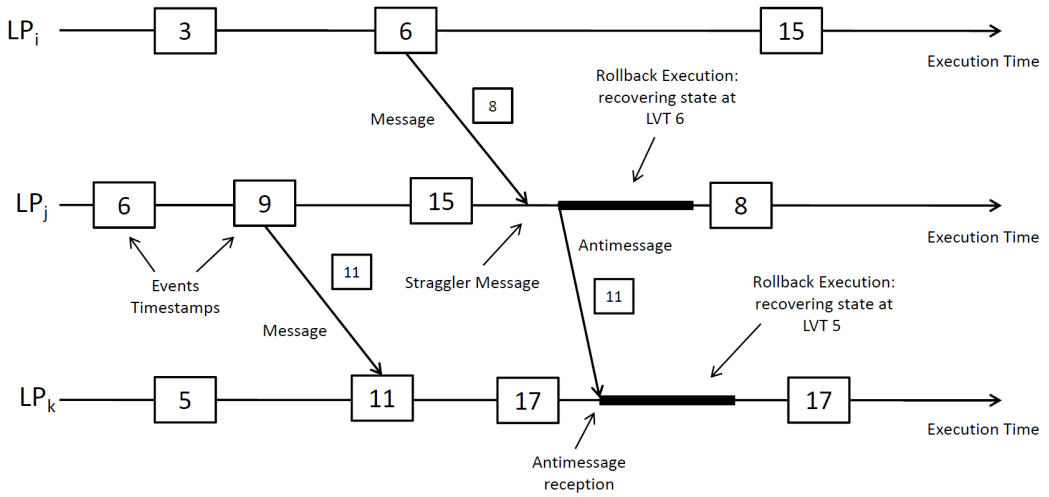


Figure 1.5. Time warp: State Recoverability

Global Virtual Time

A local time (called Local Virtual Time or LVT) works as a clock and states how far the computation is gone for a particular entity. Generally speaking, the event with the smallest timestamp among all the unprocessed or partially processed (in transit) events in the simulation is considered to be safe.
In time warp this timestamp is better known as "Global Virtual Time" (GVT) and works like a barrier: it is impossible for a LP to receive a message with a timestamp smaller than the GVT and not a single event that comes before the GVT will ever be rollbacked. For this reason, operations that are impossible to be rollbacked (e.g. outputs) are required to wait that the GVT overcomes the respective events' timestamp.

Fossil collection

As GVT keeps advancing in time, more and more messages fall behind that virtual line that separates messages that come before the GVT from those whose timestamp is greater. To save space and avoid lack of memory, a "fossil collection" mechanism reclaims memory by deleting messages that are no longer needed to be stored and commits irrevocable operations.

1.3.2 Performance of optimistic synchronization

Time warp succeeded in speeding-up various real-world simulation problems.
The areas that have benefited the most are many: battlefield simulations of military purposes, communication networks, simulations of digital hardware, biological systems and many other physical phenomena simulations.
Richard M. Fujimoto measured the performance of time warp by using a workload model called "parallel hold", or phold [10]. The results showed that time warp increases performance in all cases, but improvements are proportional to the grade of parallelism that the simulation can offer.
Tests focused on two different settings: the first one involves the use of a bigger grade of parallelism than the number of available processors, in the second one the grade of parallelism is lower than processors. In none of those cases there's the need for additional information about the state of the execution such as a wider

use of lookahead. Anyway, the overhead required by state-saving may lead the simulation into a loss of performance: this is a problem that could be addressed by implementing hardware support for state saving [11], or by ensuring that the granularity for the event processing pays off the state management overhead.

1.3.3 Asymmetry in time warp

A typical parallel implementation of the time warp paradigm implies a common control flow for all the threads involved in the execution. This means that every thread is supposed to take care of both housekeeping tasks and event processing.
In [12] is proposed an alternative asymmetric implementation of time warp: threads are split in two different incarnations: a portion of the pool runs less critical tasks while more critical ones (such as "housekeeping operations") are assigned to the remaining threads. Housekeeping operations include rollback (with state reconstruction), communication management, GVT computation, fundamental to avoid waste of speculative computation, and fossil collection, used to remove from memory messages that are no longer needed. This discrimination between the roles assigned to threads, opens to a wide range of possibilities that space from power-capping, as discussed in [12], to load balancing techniques over the assignation of a proper incarnation to every single thread. This paper puts the basis of the work presented on this master thesis.
One of the most important aspects to face in optimistic systems is the evaluation of execution time spent for housekeeping procedures.
Overhead over the critical path is also a concern, since synchronization between threads requires lots of resources to properly manage communication. If the application provides only a limited amount of parallelism in relation to the number of available processors, it is very likely that the execution will incur a considerable number of rollbacks. Researchers showed that such behavior is not relevant and can be bypassed by applying some basic corrections: the cost of rollback needs to be maintained sufficiently low.
A way more serious problem stays in the need to save periodically the state of each LP. These problems affect the performance at many levels: finding a particular state in a complex data structure used for dynamic allocation may take more time than

expected, and this can increase significantly the overall time passed performing rollback procedures.
Jefferson [13] has shown that, if needed, space can be reclaimed back by using “induced” rollbacks, named "cancelbacks", that fit the space required by the simulation execution basing on the available amount of memory.
An interesting protocol designed by S.R.Das and R.M.Fujimoto [14] showed a possible way to exploit cancelbacks to solve lack-of-free-memory problems for low ends machines in order to maximize the performance.
In low level languages, where manipulation of pointers is explicit, wrong memory accesses may lead to some unwanted overwriting: time warp must deal with this risk, preventing illegal pointer usages that may lead into runtimes errors, and wrong computations from overwriting crucial memory areas.

1.3.4 The choice of asymmetry

Asymmetry for time warp implementation in PDES has been a design choice for this work because it opens to a wide variety of new possibilities.
Though complexity raises, especially due to new synchronization requirements, if threads are split into different classes with different workflows and computational weights, the bias of the execution can be changed at will, both "manually" or from auto-adjusting mechanics. As already discussed, a part of the threads pool is assigned to maintenance (or housekeeping) tasks related to the rollback procedures that time warp needs to manage; as their complexity is prominent if compared to simple forward processing, it is convenient to separate forward execution from all the rest and arrange the sets in two different threads classes in order to gain a better control over computational resources assignation.

Chapter 2

Related Work

Alfred Park and Richard M. Fujimoto in [15] introduce a client/server approach for parallel simulation, where clients repeatedly download state vector of LPs and associate messages from a remote server (master); there are numerous advantages over conventional PDES systems, such as support for execution on heterogeneous distributed computing platforms, load balancing and simple fault tolerance.
There is a prototypical implementation called "Aurora Parallel and Distributed Simulation System" that aims to fill the gap between general purpose distributed computing project that operate on the principle of individuals donating spare processor cycles toward a common goal and PDES, making use of web services. Because of the "online" nature of Aurora, node failures require the implementation of a checkpointing system.
Aurora applies the master/worker paradigm exploiting the advantages of web services: logic processes that compose the parallel simulation program communicate by exchanging timestamped messages and are organized in clusters in "work units". A single work unit is instantiated by the server-side application containing simulation variables, an event list and a I/O message buffer associated with that work unit. The buffer is a table of messages destined for LPs contained in the work unit that have been received from other LPs, and each message is wrapped in a data structure providing information such as the message timestamp, destination and size of the packed message.
Once a work unit is successfully executed, the Aurora LP manager updates the state vector of the LPs contained in the work unit stored in the server, and the messages packed in the output buffer are sent to the right input buffer of the receiver work

units; memory is freed during this process.
Worker threads (or processes) perform the necessary computation on these work units and return results to the master, while master thread controls the global available work pool and manages the overhead associated of each worker. The workers are implemented by Aurora clients that (i) pull the necessary information from the server through web service requests, (ii) execute the implemented simulation model, (iii) upload the state vector and message buffers back to the master server after execution has been completed (according to some criterion).
The work presented in the thesis has some similarities with Aurora, yet there are a few considerable differences. The asymmetric simulation implementation does not rely on a web-service infrastructure as only local simulation instances have been tested at the current state; in later releases, tightly coupled parallel computer communication based on MPI may be implemented. As it will be later discussed in details, the master/worker paradigm has been implemented differently as the pool of threads is dynamically split into different classes: processing threads and controller threads. Controller threads manage the maintenance work (rollback, logging, fossil collection) and only one of them is considered as the "master thread", namely the thread "0" out of n-1. Master thread has more tasks to perform, but most of the assignments are shared with the other "non-master" controller threads. Of course, both the simulations rely on optimistic time management, so message exchange may incur in causal conflicts that need the be recovered. One of the greatest disadvantages of developing a PDES framework under web services is the inherent low performance: the work presented in this thesis is basically performance-oriented exploiting the asymmetric nature of parallel threads, even if limited to a single local machine.
The asymmetric master/worker paradigm implementation of PDES has been further deepened in [16] and various proposals for performance improvement, such as work unit caching, pipelined state updating, expedited message delivery and PDES-based scheduling policies have been proposed.
Differently from a non-parallel infrastructure, optimization techniques are required for an efficient M/W implementation for PDES. The implementation presented in this work will make use of its own optimization techniques that best fit to the software architecture.

The work in [17] proposes an interesting runtime system that detects and tolerates asymmetric races, a problem in which asymmetric runs very often incur. Data exchange between master/worker threads is crucial: shared variables are locally copied anytime a thread enters into a critical section and, in case some sort of conflict change in shared data is detected after the completion of a critical section, modifications over those variables are later notified to all the participants.

The work in [18] addresses the same problem and proposes a synchronization mechanism that allows non-dominant processes to push to a dominant process the access to the shared values in the critical section via a set of message-passing mechanisms instead of waiting to be granted access to the shared resource.
The term "asymmetry" in this case is mainly referred to the privileges the "dominant" thread has comparing to the others, as all the critical operations are performed only by it.

This last two works introduce different approaches for synchronization management over critical area access. Further studies over the implementation presented in this work may be made in this directions, possibly achieving consistent improvements on time wasted on locks. Currently, the asymmetric implementation provides a mixture of synchronization techniques as spinlocks and barriers.

Chapter 3

Asymmetric Time Warp

As anticipated in the chapter 1, the asymmetric approach for PDES has been chosen to achieve better results in terms of balance over different classes of tasks.
Optimistic mechanisms require a huge load of work to ensure consistency on the simulation, and asymmetry over threads might allow a considerable improvement of adaptability over different tasks and load balance. In particular, separation between tasks allows the removal of the housekeeping operations from the critical path of the computation. In this chapter we initially focus on architectural differences between symmetric and asymmetric PDES architectures, comparing improvements and drawbacks that each one can offer. After that, a possible implementation that takes into account all the problems that need to be addressed for such architecture is introduced. The asymmetric implementation analyzed in this work is based on the observations made in [12]. The same paper theorizes an architecture for asymmetry and proposes a preliminary version of the implementation; while the core of this work puts its roots in it, many changes have been made in both the theoretical model and in the code implementation.

3.1 Asymmetric time warp architecture

Implementation of asymmetry in time warp is introduced by discriminating all the tasks performed during the execution into two different classes. Those two classes of operations are assigned to distinct threads that are indeed "asymmetric", as they perform different operations in terms of computational weight and number/typology of the operations. Each class corresponds to a different thread incarnation.

- **Class-1**: Forward mode processing of simulation events;
- **Class-2**: All the rest: GVT (Global Virtual Time) computation, fossil collection, state saving, rollback, scheduling events to be processed in forward mode, message exchange, and so on

Class-1 tasks are assigned to so-called "**Processing Threads**" (PTs) and Class-2 tasks are assigned to "**Controller Threads**" (CTs).
Processing threads do all the housekeeping work and their execution is crucial to the progress of the asymmetric synchronization dynamics: if they don't act reactively, there's a high risk to waste speculative computation.
We refer to N_{cores} as the number of available CPU-cores in the system in which the simulation runs; N_{CT} and N_{PT} represent the number of used CTs and PTs respectively. Of course, the relation $N_{cores} = N_{CT} + N_{PT}$ is always valid.
One more fundamental rule holds in the design: $N_{CT} \leq N_{PT}$, and suggests that usually there's a smaller number of threads running the most critical housekeeping tasks.
Every CT is asked to manage a subset of the pool of simulation objects, namely the LPs.
Every LP, according to a partitioning scheme (Fig. 3.1), is bound to one and only one CT and is also associated with one of the PTs that are bound with that CT. In particular, CT_j associates its LPs only with its bound PTs, so a partition p_i of all the LPs managed by CT_j is bound to a single PT_i, meaning that a certain LP can be scheduled only by a single PT: this arrangement assures that no two different PTs can ever schedule the same LP, preventing any conflict on its state. Both PTs and CTs will need to alter the data structures used to manage their associated LPs and even if the tasks belong to disjoint classes by design, still they may need work on the same LP memory image.

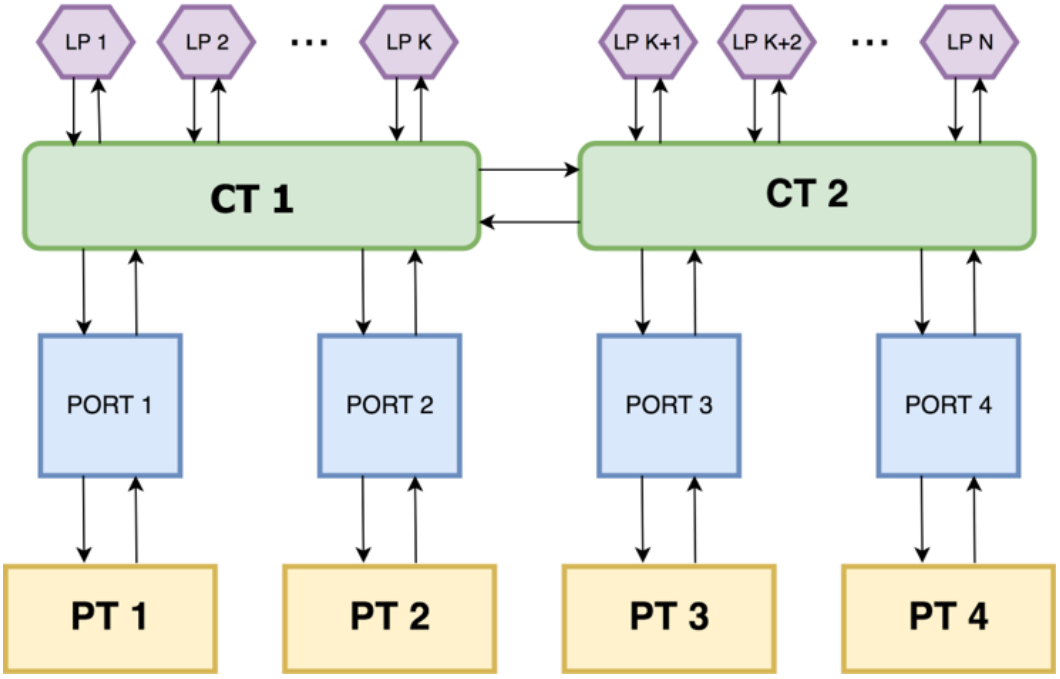


Figure 3.1. The LP partitioning scheme

A CT and its PTs live in the same machine and both may require memory write operations on a particular LP: PT is in charge of manipulating the state during forward execution of an event, CT can require the access to the memory image during a rollback procedure in case the state must be restored. The use of shared-memory multi-core machines allows CTs and their bound PTs to have always access to the shared data related to the simulation execution.

The following architectural details take their base organization from [19].

Every LP has its inbound events queue, where events picked from the logic process' bottom half (i.e. a FIFO queue with the inbound events) are sorted by their timestamps. CTs should manage scheduling and the insertion of those events into the proper input port and, in case, discard those events that received a matching anti-message after a rollback procedure. Anti-messages are a special kind of events (i.e. "negative events") used by the rollback mechanism to tell which messages should be canceled from the target LP's input queue. Management of the queue that keeps the LP's checkpointed states is one more task left to the CT: snapshots of states are taken and logged into this queue when required.

Communication between CTs and PTs is managed via the notion of **ports**, a bidirec-

tional communication structure working on a shared memory support that multiplexes data flows. Communication that flows from the CT to the PT is multiplexed in two channels with different priorities (i.e. low and high) and is referred as **input port or channel**. A single output channel manages communication flowing from the PT to the CT and low priority input channel receives events to be processed in forward mode.
CT implements a scheduling mechanism that cyclically picks the event to be processed with the lowest timestamp from the LPs' event queue (Smaller Timestamp First -STF- policy) and sends it to the low priority input channel of a bound PT. STF grants that events are picked from a LP according to their timestamp order, excluding a few particular cases: (i) the arrival of a straggler message reveals causal inconsistencies, or (ii) the cancellation of some event in the queue, or (iii) the LP produces a new event for itself with timestamp lower than ones already in the queue. If the execution of an event produces events, those new events (if any) are posted by the PT to its output channel to be later extracted by the bound CT; if the receiver LP belongs to the current CT domain, events are put in the event queue of that LP, otherwise are sent to the appropriate CT.
There is no blocking synchronization between the two threads for accessing the I/O ports, meaning that the CT is not supposed to wait the production of outputs from the PT he just sent some messages to, instead is free to manage other PTs that are bound to it.

3.1.1 Ensuring consistency

The double priority structure in the PTs' input ports has been designed to manage cases in with causality is compromised and inconsistencies are detected in already scheduled events (i.e. that were previously inserted in the low priority input port). The detection of a straggler message starts a series of procedures involving state restoration, out-of-order messages removal from PT input port and possible rollback if some unwanted processing has already been made. High priority port is designed to receive messages that are created by the CT with the only purpose of warning the PT in case the simulation ran into one of the previous scenarios.
When the scheduling routine notices that one of the bound LP needs to be rollbacked, a special procedure sends to the LP's assigned PT a couple control messages: this

kind of messages created inside the simulation have just maintenance functions and are not involved in forward execution.
A control message, better known as "notice", is sent to the high priority input port; immediately after one more control message, the "bubble", is sent to the low priority input port. The two control messages are initialized with the same mark (i.e. a non-decreasing object identifier) and with the timestamp of the new "last correctly processed" event of the LP to be rollbacked.
The PT's processing routine in charge of picking messages from the input ports always check first if any high priority message (i.e. a "notice") is waiting to be extracted. When a "notice" high priority message is picked, all the messages in the low priority port with same receiver as the "notice" and higher timestamp need to be discarded because are no longer consistent; others are free to be processed. The procedure keeps extracting low priority messages until at some point the "bubble" message appears.

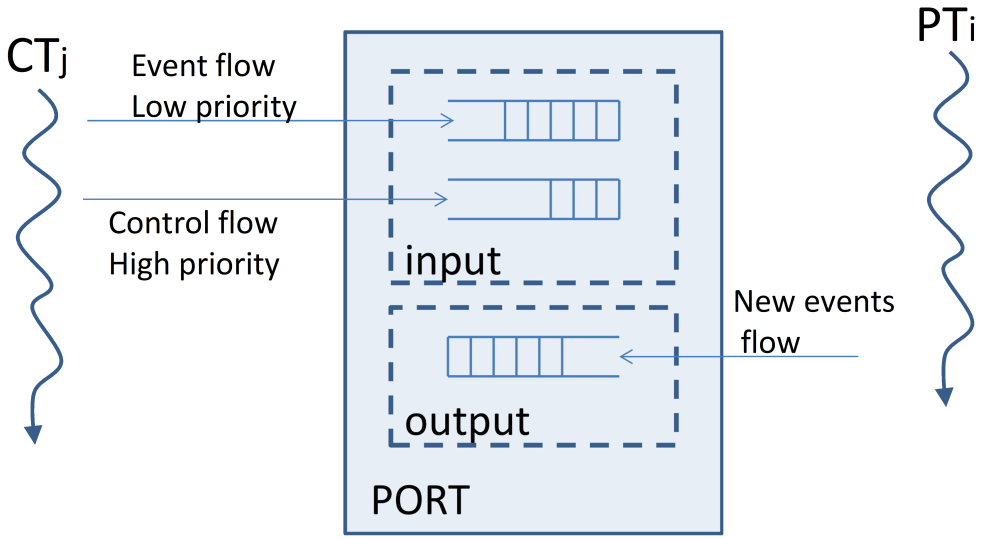


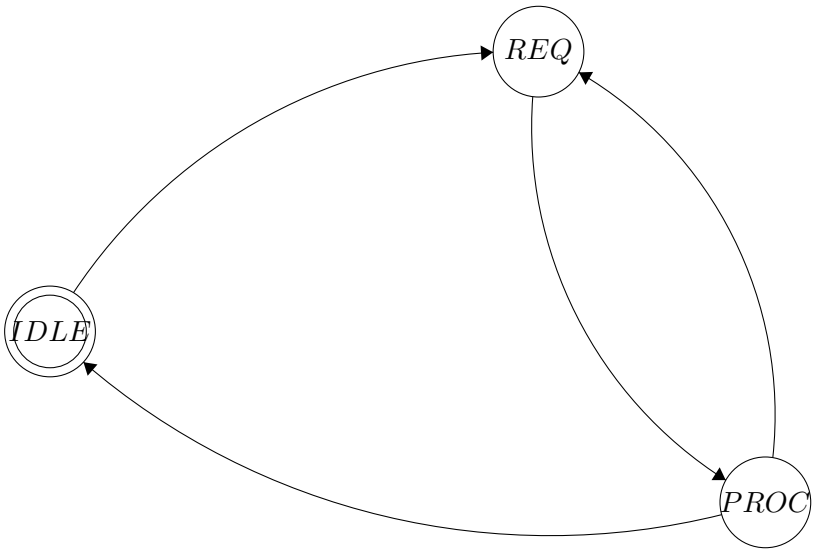
Figure 3.2. The ports

Bubble and notice messages match if and only if they have the same mark. When the routine eventually incurs into the matching "bubble", it means that the "cleaning" procedure is over, the port is free from out-of-orders messages and the CT can proceed with the next phases of the rollback procedure for the LP.
The PT notices its CT that the "bubble" and the "notice" matched by sending back

one more control message, known as "ack", through the output port; when the processing thread receives such type of message, it continues the restoration procedure for the LP.
Fig.3.2 depicts how events flow into the PT's input and output ports and the pseudocode of the main routine performed by the processing thread is shown in the algorithm 1. Messages that follow the bubble are safe by design, since the port is organized as a FIFO channel and messages are always extracted by the LP and scheduled in (virtual) time order. It is possible that the only out-of-order messages are the ones standing in the CT's input port: this means that the last processed event is still compliant with the casual order and state restoration is not required.
Asymmetry on simulation threads may cause overlaps between distinct rollback procedures for the same LP: the risk is that an "ack" message may be mistakenly confused as the response to a more recent causality conflict. Measures have been taken to solve this issue by splitting the rollback procedure into various phases organized in a finite state machine, this should grant that conflicts between two or more different stragglers are avoided.

Algorithm 1 Asymmetric Process

while *there's a rollback* NOTICE *in the hi priority channel* **do**
  hp_msg ← *pick a message from the hi priority channel*
  while *true* **do**
    lp_msg ← *pick a message from the lo priority channel*
    if *lp_msg is a rollback* BUBBLE **then**
      send rollback ACK
    else if *lp_msg's timestamp* < NOTICE*'s timestamp* || *different message receivers*
    then
      process lp_msg
    else
      discard lp_msg
    end if
  end while
end while
lp_msg ← *pick a message from the lo priority channel*
if *lp_msg is a rollback* BUBBLE **then**
  wait for the matching NOTICE (*external algorithm*)
end if
process lp_msg



The finite state machine depicted above shows the states of a logic process during the rollback signaling phase in which the processing thread is asked to empty both the high and the low priority channels. This is a control system required to avoid unwanted rollback instances overlaps.

- IDLE STATE: is the starting state, where a logic process should stay when no rollback procedure is required.
- REQUESTED STATE: a straggler message has been picked up during the LP's bottom half processing phase and LP needs to be rollbacked. Signaling procedure will actually begin when the LP is scheduled by its controller process.
- PROCESSING STATE: the to-be-rollbacked LP has been scheduled and the couple of NOTICE/BUBBLE messages have been immediately sent to the PT. It may happen that one more straggler message is received by the same LP before the "PT channel cleaning" procedure is completed and the ACK message, if received too late, may be mistakenly confused to be referring to this last request of rollback. To avoid cases like this, the state goes back to "REQUESTED". The ACK event has a "rollback mark" that identifies what instance of rollback procedure the LP is running: once the CT receives the ACK, it first checks the state of LP, then verifies that the rollback marks match. If the CT finds the LP in the REQUESTED STATE upon the arrival of an ACK event, it immediately discards that event since there must be a new ACK arriving. Discarding non-

completed rollback procedures is possible since new procedures involve at least the same amount of messages that an older procedure do.

3.1.2 Checkpointing

When talking about task separation, checkpointing is one important aspect and is crucial for state restoring procedures.
In this time warp architecture, checkpointing is a Class-2 task, so is carried by the CT: the system periodically takes a snapshot of the logic processes by creating a new log instance. This instance contains all the crucial information required to safely restore a previous state of the LP. All those logs are inserted into a queue, ready to be picked once requested.

3.1.3 Dynamic resources assignation

Degree of speculation is also managed by the CT through a short-term self-tuning mechanism that decides the proper number of events to be injected into each of the input ports at every iteration of the scheduling routine. The amount of events to be inserted in a port at each turn totally depends on the utilization rate of the same channel during the previous scheduling iterations.
This way, performance is maximized by finding the best trade-off between speculation and efficiency. The equation that regulates the number of events to be added into a PT input channel at every CT round is the following:

$$N_{events} = current\_PT\_batch\_size - current\_PT\_port\_size$$

Where the first term represents a dynamic boundary used to put a limit on the number of messages sent toward a particular PT; variations over this boundary depend on the utilization rate of the PT's low priority input channel.
Second term is the number of events that currently are in the low priority input channel for the same PT.

3.1.4 A comparison with classic symmetric multithread architecture

In a classic multithreaded implementation with no discrimination between roles assigned to threads, there's a considerable smaller amount of elements to be taken into account. All the threads manage both the scheduling and the processing routines,

rollback procedures are free from collisions, inter-threads communication is less complex and there are definitely less race conditions to deal with, so synchronization is much easier to achieve. Of course, compared to an asymmetric implementation, a symmetric approach allows a smaller grade of granularity into the simulation.
Asymmetry of threads is the starting point for the purposes of this work, as many of the features introduced will be exploited to drastically improve performance.

3.2 Dynamic thread role assignment

Asymmetry in threads introduces a discrepancy between the load the two incarnations must carry: CTs are in charge of governing the lifetime of the application (in terms of scheduling and correctness control), from all perspectives (energy efficiency, performance, self-tuning, etc.) while PTs are much simpler and process units of work, so it is very likely that input ports remain empty due to the fact that the CT could be running housekeeping routines instead of sending to the PT new events to be processed. This is the case in which thread incarnation balancing could be useful: dynamism in thread role assignment is meant for a general purpose simulation, so the auto-adjusting balance system adapts the simulation settings for different workloads.
So far, we have discussed about features, advantages and drawbacks of the asymmetric implementation of PDES applied to an existing parallel/distributed simulation platform. Asymmetry in an actual run implies some sort of role-assignment procedure during the threads initialization routine; of course the constraints $N_{cores} = N_{CT} + N_{PT}$ and $N_{CT} \leq N_{PT}$ are always valid.
The provisional asymmetric implementation required to receive as input both the number of working threads involved in the simulation and the number of threads that would have been assigned the incarnation of "CT"; of course the remaining threads were initialized as PTs. This working mode obviously lacks any sort of assignment criteria but (i) any model that is ran requires settings based on its nature, so blindly deciding the proportion between PTs and CTs, or empiric evaluation over running times, are not viable choices; (ii) different time intervals of the simulation may require a different arrangement between PTs and CTs to improve performance.
Dynamic assignment for thread incarnation was initially not planned: the previously introduced short-term self-tuning mechanism has a much smaller scope and PTs

still risked to waste computational time with no events to process in their ports. An implementation for a long-term self-tuning mechanism for dynamic thread role assignment in an asymmetric environment is therefore introduced.

3.2.1 How to dynamically assign threads

Every thread is identified by a unique ID starting from 0, controller threads are assigned to values from 0 to the number of CTs the simulation is running with (e.g. with 24 threads and 8 controller thread, IDs from 0 to 7 are assigned to CTS, while IDs from 8 to 23 belong to processing threads). Controller thread with ID 0 is the "master thread" and has a special role over various simulation mechanics. In particular, the master thread is in charge of managing all the procedures related to the auto-tuning procedure to dynamically assign thread incarnations.
PT/CT balancing system is built around a "score" system: every iteration of the main loop begins with the evaluation of a "score" that is the composition of various efficiency parameters over the last run of the loop. The score is reset at the end of the evaluation process, ready to be updated by a series of assessments made from data and statistics analysis. Score starts from 0 and can assume both positive and negative values; if score evaluation is made over a positive value, more CTs may be required, conversely, negative score values may suggest that PTs are not enough and need to be increased in number. Of course, constraints on the rates between the number of PTs and CTs are always valid. There is an upper bound and a lower bound over the values the score can get: if value goes over the upper bound, threads' incarnations must be rearranged and, if possible, one of the processing threads changes his nature into "controller thread". Conversely, if score falls behind the lower bound, processing threads are increased by one at the expense of the controller threads pool. This process has some similarities with the "hill climbing" heuristic search.
Once one of the bounds is exceeded, the threads rearrangement is performed basing on the new outcomes: simulation is temporarily paused, all the threads receive a signal and start the "emptying" procedure by processing all the events inside the PT's I/O ports until all the channels are empty. Only after all the I/O ports are emptied, incarnations are reassigned and logic processed are linked to threads following the new system settings. The execution can now be resumed from the point it was

stopped.
The following pseudocode shows the score evaluation procedure.

Algorithm 2 Score evaluation procedure

```
if current thread is master thread then
  if SCORE >= higher threshold AND it's possible to increase controllers
  then
    controller_threads modifier ← +1
  else if SCORE <= lower threshold AND it's possible to decrease controllers
  then
    controller_threads modifier ← −1
  else
    no thread balance required
  end if
end if
```

At the end of the evaluation procedure, the modified value is used to determine the new balance on threads assignment.
The score is the outcome of evaluations made over various criteria: the goal of the score is to summarize in an indicative value the "health" status in terms of balancing between threads incarnations.
Different models have different behaviors, so different necessities in terms of balance between resources. The simulation currently has four criteria to be evaluated, each one has its weight and its methodology for evaluation.

- The first criterion is related to the **overcoming of the upper or lower bound** of the port utilization rate of a single PT. Every time a bound is passed, we can get hints about the necessity to find a new balance between PTs and CTs. If the upper bound is passed, it means that the current size of the input channel must be increased, unless it already reached the size limit: more PTs may be required.
  At the same time, passing some PT's port utilization rate lower bound can suggest that one of the PTs is underused and its input port may be getting empty soon: it can be useful to reduce the number of processing threads

because events injection from CTs probably won't match the pace of the event processing rate of the PTs. Anytime a bound is hit, the score is immediately modified.

- The second criterion is linked to the first but is more accurate: during the scheduling process, but before the injection of an event into the proper PT, the CT checks **if the input ports of that PT are empty** or not. This time, evaluations are made through an exponential moving average on the number of times the ports have been found empty.
  There are definitely not enough controller threads if ports of whatever PT are found empty for too many times; this means that the amount of events sent by the controller threads is not enough to keep all the PTs busy.

- The third criterion is again made through exponential moving average but this time the matter of the calculation is the **bubble turnaround time** (i.e. the interval of time between the instant in which the couple of control messages notice/bubble is sent to the processing thread and the moment in which the CT whose PT must be rollbacked receives the acknowledgment control message). This value in very meaningful since the more time passes between the request sending and the acknowledgment receiving, the more messages had to be discarded (or processed) before notice and bubble messages matched. If average values go over a pre-defined bound, it means that ports are full of events and rollback procedures drastically slow down, this may cause overlaps on various rollback instances.
  More processing threads may help in achieving faster notice/bubble matching, in order to speed up the overall rollback procedures.

- The last criterion is the **arrival rate of straggler messages**: evaluations are made on the number of LP that picked stragglers during the current iteration of main loop and if the pre-defined bound is passed, the score needs to be modified. Evaluation on how many LPs require to be rollbacked out of all of them is an index that worth to be checked: speculation grade needs to be decreased to avoid situations in which too many rollbacks can drastically slow down the simulation, so the better choice is to reduce the number of PTs.

Chapter 4

Experimental Assessment

The implementation of the asymmetric variant and the self-adjusting balancing mechanism has been made over a preexisting simulation platform named ROOT-Sim. In this chapter, the results of a series of runs made with different settings are shown and discussed to asses our proposal. All the following experimental runs have been executed on a 12 cores/24 threads CPU machine (AMD Opteron 6174) with 2.2 Ghz of running frequency, 32GiB of system memory running on Debian GNU/Linux 9, kernel version 4.9.88-1+deb9u1. For every run, the model used is based on the "phold" benchmark [10].

4.1 The ROme OpTimistic Simulator

The ROme OpTimistic Simulator [20] is an x86-64 Open Source, parallel/distributed simulation platform developed using C/POSIX technology, which is based on a simulation kernel layer that ultimately relies on MPI for data exchange across different kernel instances. The platform transparently supports all the mechanisms associated with parallelization (e.g., mapping of simulation objects on different kernel instances) and optimistic synchronization (e.g., state recoverability).
The programming model supported by ROOT-Sim allows the simulation-model developer to use a simple application-callback function named ProcessEvent() as the event handler, whose parameters determine which simulation object is currently taking control for processing its next event, and where the state of this object is located in memory. In ROOT-Sim, a simulation object is a data structure, whose state can be scattered on dynamically allocated memory chunks, hence the memory

address passed to the callback locates a top level data structure implementing the object state-layout.

ROOT-Sim's development started as a research project, and is currently run by the High Performance and Dependable Computing Systems group at Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza, University of Rome.

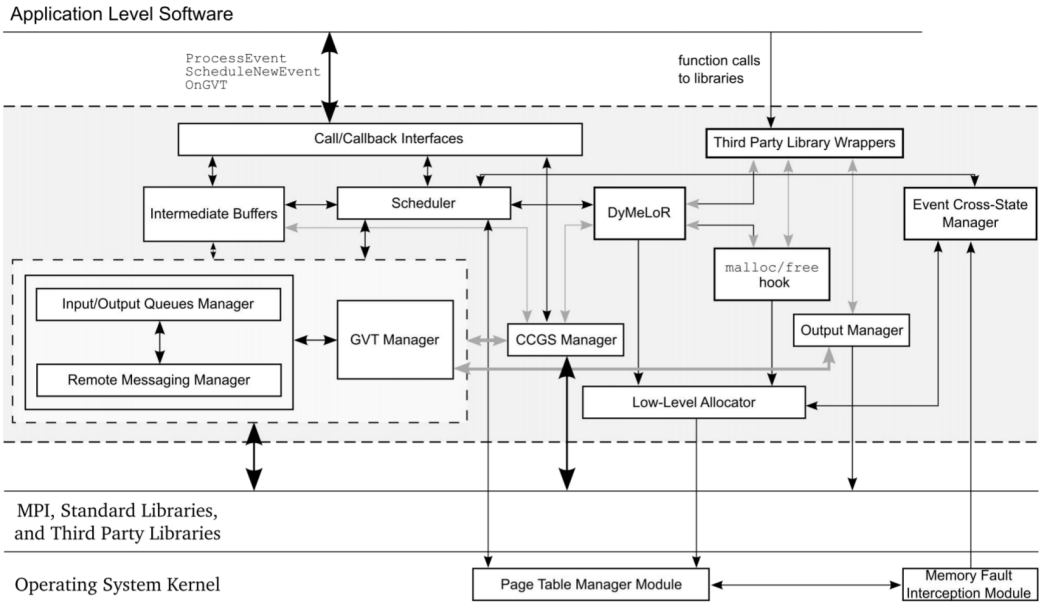


Figure 4.1. ROOT-Sim Internals

4.2 Parameters

The results discussed in this thesis have been calculated with a combination of the following parameters:

- **LP**: the number of logic processes used during the run.
- **Loop Duration**: for how long the model waits upon the arrival of an event, before sending a new event (or events) to the simulation.
- **Fanout**: how many events with different timestamps are sent from the model to the simulation.
- **TAU**: it is the average of an esponential distribution used to describe the interarrival time of events.
- **Event Receiver**: criterion used to decide which LP is the receiver of a new event created by the model. Tests are performed according to the following different running modes: symmetric/asymmetric (auto-adjusting/2/8/12 CTs)

The termination condition is that every logic process must have successfully processed at least 2000 events, and runs are always performed with 24 threads.

4.3 Measurements

4.3.1 First test pool: fixed loop duration, fanout and random LP event receiver

Fig. 4.2 and Fig. 4.3 show the execution time of runs made by keeping constant the loop duration, the fanout and the event receiver. It is immediately clear that the number of LPs involved in the execution is extremely important in terms of execution time. As the number of the LPs increases, the asymmetric structure suffers when is ran with a fixed rate between PTs and CTs: at a first sight, it may seems that best results are obtained by just increasing the number of controller threads but, considering that 12 over a total of 24 threads is, by design, the maximum allowed number of CTs, it is clear that best performance is obtained by dynamically varying the PT/CT rate due to some internal dynamics that may require different settings in different moments of the simulation. TAU variance doesn't seem to be influencing results enough to be considered a crucial factor. Anyway, it can be noticed that the advantages of the auto-adjusting balance system are extremely evident as the number of involved LPs increases.
The difference between Fig. 4.2 and Fig. 4.3 is the loop duration, and the overall simulation seems to benefit from less channel occupation rate. Anyway, the auto-adjusting asymmetric simulation seems to work slightly better with a smaller loop duration.

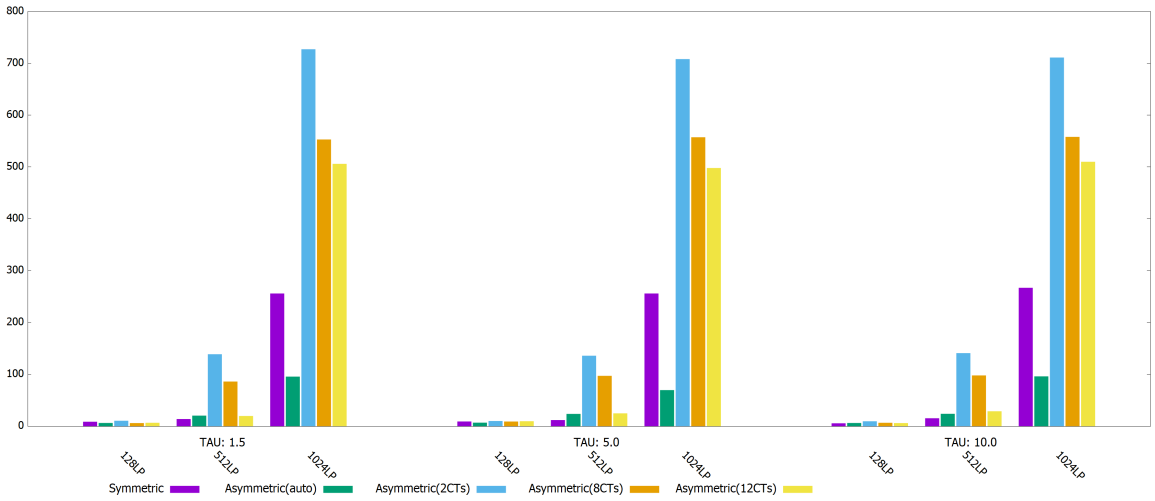


Figure 4.2. Loop duration: 15 musec, Fanout: 1, Random LP event receiver

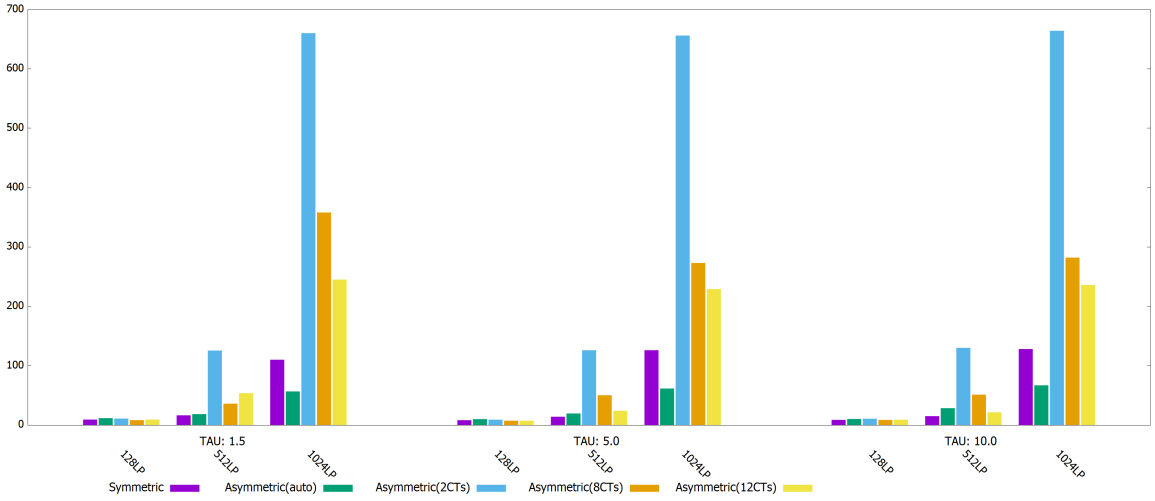


Figure 4.3. Loop duration: 135 musec, Fanout: 1, Random LP event receiver

4.3.2 Second test pool: fixed loop duration, TAU and random LP event receiver

Fig. 4.4, Fig. 4.5 and Fig. 4.6 compare two different values for fanout. With a random fanout that varies from one to three messages sent by the model, the load over channels is considerably higher than sending just one message at a time, and execution is subsequently slowed down. The asymmetric runs with 2 controller threads incur into unusual time peaks even with only 128 LPs: the conclusions are that simulation registers considerably higher execution times even with a slightly higher fanout. Anyway, the trend seems to be confirmed: as execution times increase over all the running modes, the advantages brought by the auto-adjusting mechanism are evident and substantial.

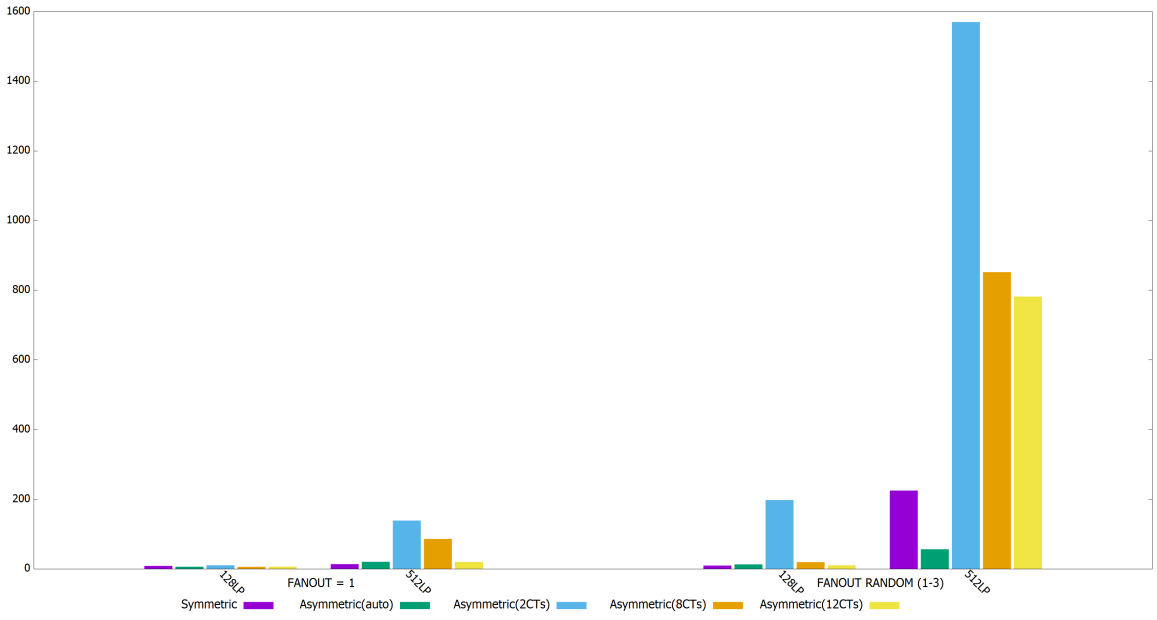


Figure 4.4. Loop duration: 15 musec, TAU: 1.5, Random LP event receiver

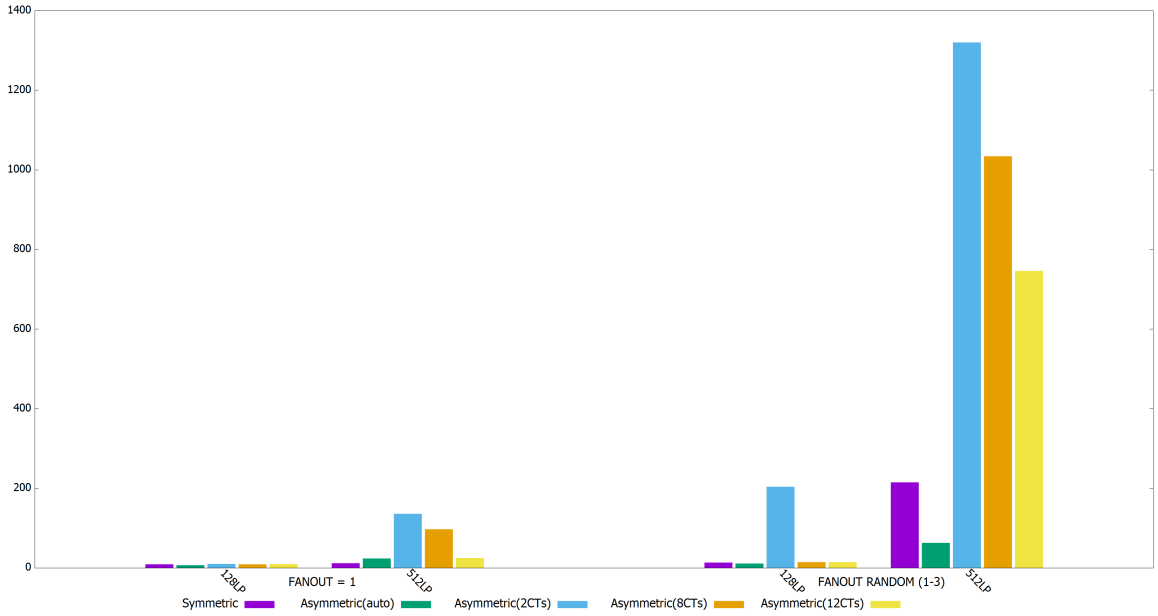


Figure 4.5. Loop duration: 15 musec, TAU: 5.0, Random LP event receiver

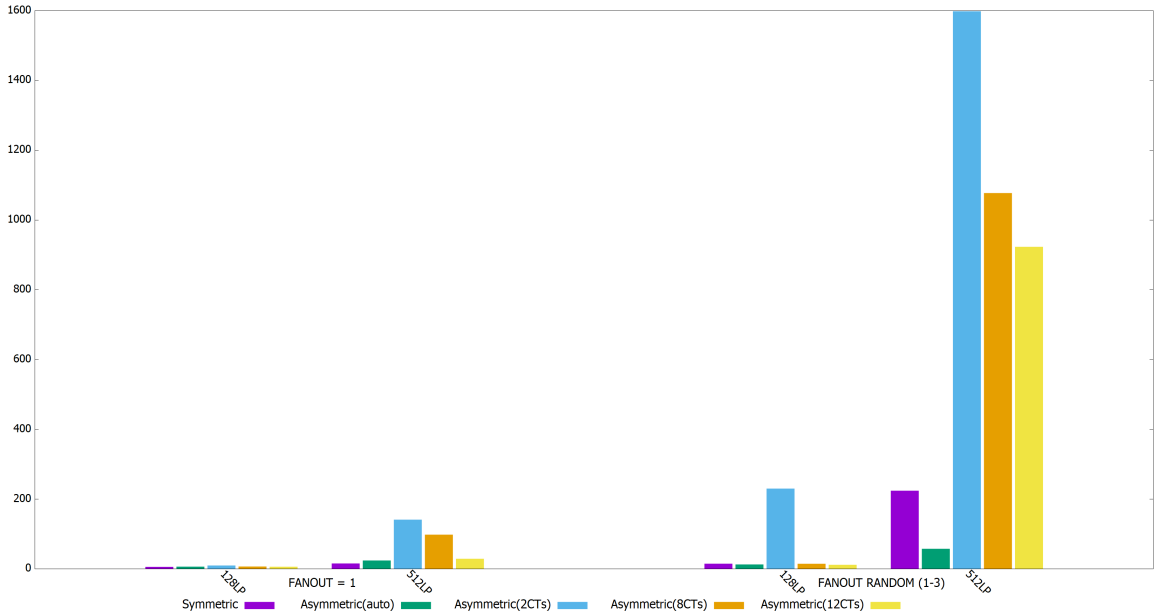


Figure 4.6. Loop duration: 15 musec, TAU: 10.0, Random LP event receiver

4.3.3 Third test pool: fixed loop duration, fanout and close LP event receiver

Fig. 4.7, and Fig. 4.8 are the graphical representation of the last series of tests. They aim to compare execution times with models having different sending strategies with a constant fanout setting of 1-3(random) messages. With "close LP" event receiver, executions with a smaller number of processing threads (2-8) seem to have a benefit, especially with higher values for TAU. Overall, the execution times with a slightly bigger fanout are considerably higher than those in previous tests with fanout limited to just one message. Execution is drastically slowed down when channels are overloaded with messages.

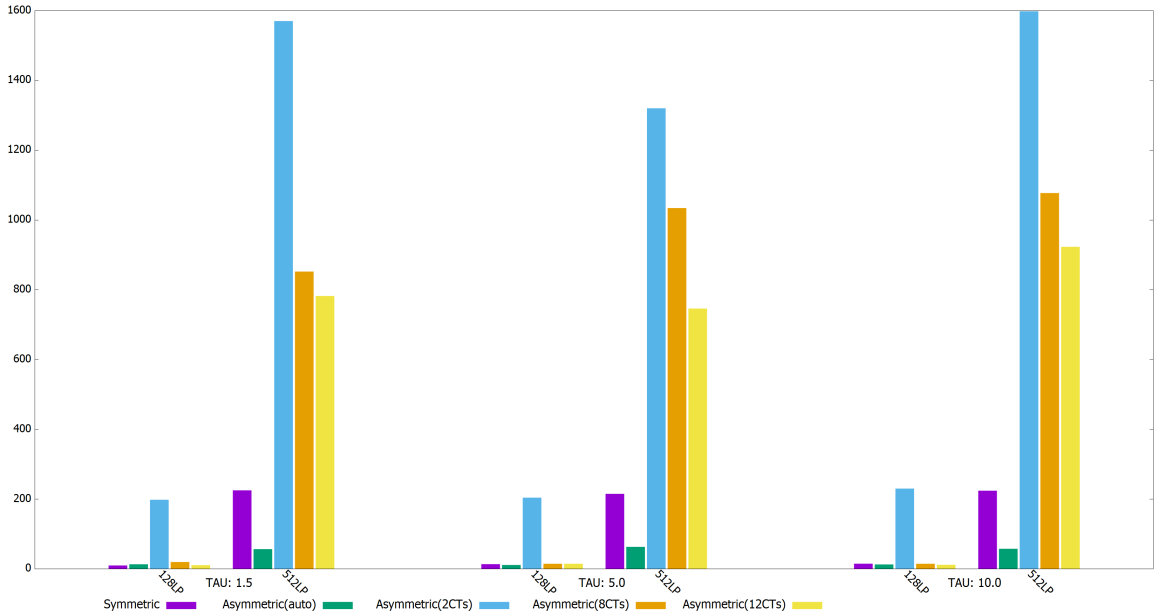


Figure 4.7. Loop duration: 15 musec, Fanout: 1-3(random), Close LP event receiver

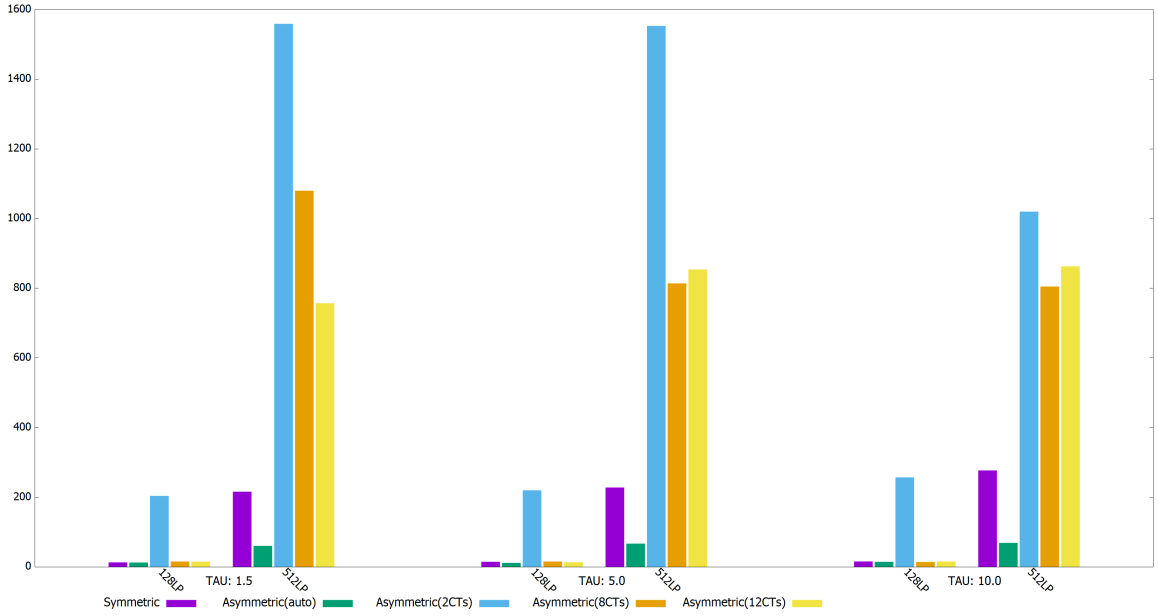


Figure 4.8. Loop duration: 15 musec, Fanout: 1-3(random), Random LP event receiver

Chapter 5

Conclusions

As shown by experimental data, the asymmetric implementation of PDES needs an auto-adjusting algorithm in order to be effective; without it, manual settings on the thread balance do not offer any kind of gain in terms of performance. In fact, simulation completion times are extremely higher with bigger model sizes (i.e. the number of PTs). When threads assignation is managed automatically by an internal algorithm, the asymmetric approach generally grants better results, but performance considerably improves as the model size increases.
Contrarily, runs with fixed balance between treads suffer from the exceeding of events that occupy the channels: the increased amount of rollbacks causes most of those events to be discarded, leading into a considerable waste of speculation.
Possible future studies could be directed towards improvement of the internal management of message channels, trying to design a better scheduling strategy, especially in terms of per-thread number of events to be scheduled at every round of the control thread loop. Furthermore, it may be interesting to deepen the criteria behind the control strategies that converge into the score mechanism that supports the self-balancing system.

Bibliography

[1] R. Fujimoto, "Distributed Simulation Systems", College of Computing Georgia Institute of Technology Atlanta, GA 30332, U.S.A. Proceedings of the 2003 Winter Simulation Conference.

[2] S. Robinson, "Discrete-event simulation: from the pioneers to the present, what next?", Journal of the Operational Research Society.

[3] W. Wulf and S. McKee, "Hitting the Memory Wall: Implications of the Obvious", Department of Computer Science University of Virginia, December 1994.

[4] M. Principe, T. Tocci, P. Di Sanzo, F. Quaglia, and A. Pellegrini, “A Distributed Shared-Memory Middleware for Speculative Parallel Discrete Event Simulation,” ACM Transactions on Modeling and Computer Simulation, Dec. 2020.

[5] S.Jafer, G. A. Wainer and Qi Liu, "Synchronization Methods in Parallel and Distributed Discrete-Event Simulation".

[6] R. Fujimoto, "Performance Measurements of Distributed Simulation Strategies", in Proc. of the Distributed Simulation Conference, 1988, p. 14-20.

[7] R. Fujimoto, “Parallel Discrete Event Simulation”, Communications of the ACM, pp. 30-53, 1990.

[8] R. Fujimoto, "Distribuited Symulation Systems", Proceedings of the 2003 Winter Simulation Conference, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, U.S.A.

[9] C. Carothers, K. Perumalla, R. Fujimoto, "Efficient optimistic parallel simulations using reverse computation", ACM Transactions on Modeling and Computer Simulation.

[10] R. Fujimoto, R. Bagrodia, R. Bryant, K. Chandy, D. Jefferson, J. Misra, D. Nicol, and B. Unger, "Parallel discrete event simulation: the making of a field", In Proceedings of the 2017 Winter Simulation Conference (WSC '17), IEEE Press, Piscataway, NJ, USA, Article 16, 30 pages.

[11] S. Carnà, S. Ferracci, E. De Santis, A. Pellegrini, and F. Quaglia, "Hardware-assisted incremental checkpointing in speculative parallel discrete event simulation".

[12] S. Conoci, D. Cingolani, P. Di Sanzo, B. Ciciani, A. Pellegrini, and F. Quaglia. 2018, "A Power Cap Oriented Time Warp Architecture", In SIGSIM-PADS '18 : SIGSIM-PADS'18: SIGSIM Principles of Advanced Discrete Simulation CD-ROM, May 23–25,2018, Rome, Italy. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3200921.3200930

[13] D. Jefferson, "Virtual Time II: Storage Management in Distributed Simulation", UCLA, May 1990.

[14] S. Das and R. Fujimoto. "An Adaptive Memory Management Protocol for Time Warp Parallel Simulation", Georgia Institute of Technology, Atlanta, Georgia 30332-0280.

[15] A. Park and R. Fujimoto, "Aurora: An Approach to High Throughput Parallel Simulation", Computational Science and Engineering Division College of Computing, Georgia Institute of Technology Atlanta, Georgia, USA 30332-0280

[16] A. Park and R. Fujimoto. "Efficient Master/Worker Parallel Discrete Event Simulation on Metacomputing Systems", IEEE Transactions on Parallel and Distributed Systems, Vol.23, No.5, May 2012.

[17] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal and K. Pattabiraman, "Efficient Runtime Detection and Toleration of Asymmetric Races", IEEE Transactions on computers, vol. 61, no. 4, april 2012.

[18] J. Cleary, O. Callanan, M. Purcell, and D. Gregg, "Fast asymmetric thread synchronization", ACM Trans. Architec. Code Optim. 9, 4, Article 27 (January 2013), 22 pages.

[19] R.Vitali, A.Pellegrini, F.Quaglia. "Towards Symmetric Multi-Threaded Optimitic Simulation Kernels",2012 (PADS)

[20] https://github.com/HPDCS/ROOT-Sim/wiki