

Benchmarking Parallel Discrete Event Simulations



Utrecht University

ICA-3539733

Supervisors
Dr. ir. J.M. van den Akker
Dr. S.W.B. Prasetya

August 2017

Vincent Bonnet
3539733

Abstract

The single core performance of hardware processors have seen only modest increase over the past decade. Yet simulation needs are growing in both simulation size, and complexity. The Parallel Discrete Event Simulation (PDES) field, is concerned with executing a single simulation run using multiple threads. Executing such a simulation in parallel is non-trivial and requires synchronization algorithms to ensure correctness. To compare performance between synchronization algorithm, the PHOLD benchmark is commonly used. In the PDES field, threads (or 'Logical Processes') can only communicate via messages. The allowed sending pattern can be formulated as a directed graph, where tail nodes are allowed to send messages to the head node. We will present the EPHOLD benchmark algorithm, a more generalized version of the PHOLD benchmark, and a mathematical model to predict the amount of parallelism that a given EPHOLD benchmark will attain when a specific PDES synchronization algorithm (i.e. YAWNS) is used. We will experimentally evaluate the predictive capabilities of this mathematical model, and show bottlenecks that emerge when using EPHOLD yet remain hidden with PHOLD. The EPHOLD and PHOLD benchmarks will be experimentally examined for Scale Free and Complete graphs using three synchronization algorithms commonly used in the PDES field.

Contents

1	Introduction	1
1.1	Simulation	1
1.2	Discrete Event Simulation	2
1.3	Parallel Discrete Event Simulation	2
1.3.1	Application Parallelism	2
1.3.2	Backend Parallelism	3
1.3.3	Model Parallelism	3
1.3.4	Parallel Discrete Event Simulation Graph	4
1.3.5	Causality	4
1.4	The need for Parallel Discrete Event Simulation	5
1.5	Thesis organization	6
2	A literary survey	7
2.1	Conservative Algorithms	7
2.1.1	Deadlock-avoidance	9
2.1.2	Deadlock detection and recovery	12
2.1.3	Conservative time window	12
2.2	Optimistic Algorithms	13
2.2.1	Time Warp	13
2.2.2	Alternative local control mechanisms	16
2.2.3	Alternative global control mechanisms	16
2.3	Benchmarking	17
2.3.1	PHOLD	18
2.4	Scale Free networks	20
2.5	Analytical modeling of potential parallelism	21
3	Research	23
3.1	Questions	23
3.1.1	PHOLD	24
3.1.2	Analytical modeling for a given PDES graph	25
3.2	Relevance for science, technology and society	26

3.3	Methodology	27
3.3.1	PHOLD	27
3.3.2	Analytical modeling for given PDES graphs	28
4	Extending PHOLD	29
4.1	A more formal definition of PHOLD	29
4.2	Adding more details	32
4.2.1	The movement function	33
4.2.2	Initial configuration	33
4.2.3	Timestamp increment	34
4.2.4	Computation grain	34
4.2.5	Extended PHOLD	34
4.3	Examples	34
4.3.1	Complete graph, different weights	35
4.3.2	Simulating PHOLD	36
5	Definitions of parallelism	39
5.1	YAWNS	39
5.2	CMB	43
5.3	Time Warp	44
6	A parallelism approximation algorithm for EPHOLD using YAWNS	45
6.1	The variables	46
6.2	Some probability theory	46
6.3	Movement of events	47
6.4	Defining $S_{q,j}$	47
6.5	Scheduling of events	52
6.6	The algorithm	53
6.6.1	Initialization	53
6.6.2	Iteration	53
6.6.3	Runtime Analysis	55
6.6.4	Design considerations	56
6.7	Numerical Examples	56
6.7.1	PHOLD execution	56
6.7.2	Different weights	62
7	Implementation	67
7.1	Approximation algorithm	67
7.2	The experimentation framework	69
7.2.1	Running experiments	69

7.2.2	Code dependencies	70
7.2.3	Input	70
7.2.4	Output	71
7.2.5	Details	72
7.3	Analysis	73
8	Experiments	75
8.1	Setup	75
8.1.1	Scenarios for complete graphs	77
8.1.2	Scenarios for scale free graphs	79
8.1.3	Generating the graphs	81
8.2	YAWNS	82
8.2.1	Parallelism in complete graphs	88
8.2.2	Conclusion	97
8.2.3	Parallelism in scale free graphs	98
8.2.4	Conclusion	105
8.3	Chandy-Misra-Bryant	106
8.3.1	Complete graphs	106
8.3.2	Scale free graphs	109
8.3.3	Conclusion	111
8.4	Time Warp	112
8.4.1	Complete graphs	113
8.4.2	Scale free graphs	120
8.4.3	Conclusion	127
8.5	Conclusions	128
9	In conclusion	129
	Appendices	131
10	Scale free topology analysis	167
10.1	A Probability Density Function derivation	167
10.2	Hubs and leafs	168
10.3	Degree estimation	168
	Index	171
	Bibliography	173

Chapter 1

Introduction

In this chapter we will introduce the main concepts associated with (Parallel) Discrete Event Simulations (PDES). After introducing the concept of simulation in Section 1.1, Section 1.2 will give a brief introduction into the jargon typically used in the Discrete Event Simulation (DES) field. It also discusses several key elements of a Discrete Event Simulation. Section 1.3 touches on some of the difficulties that come into play when trying to parallelize Discrete Event Simulations, and introduces the three main types of parallelization that are typically considered.

1.1 Simulation

We define a *system* as a collection of entities that (inter)act together to accomplish some logical end. (This definition was coined originally in [52]). "a system" can mean very different things and is highly reliant of the objectives of the study you wish to perform. E.g. suppose we intend to study the throughput (items per hour) of a car-manufacturing factory. In this case we are interested in modeling only a subset of the actual system, because not all entities in the actual system are relevant to the study of throughput (e.g. the configuration of the canteen, etc.).

The *state* of a system is defined to be the collection of all variables that are relevant to the simulation objective and describe the system at a particular (simulation) time. Simulations are typically divided into two types. A *Continuous simulation* changes the state of a system continuously throughout time. *Discrete Event Simulations* (DES) change the state instantaneously at a separate(discrete) points in time. Because the main focus of this thesis is on Parallelizing discrete event simulations, we will not consider continuous simulations.

1.2 Discrete Event Simulation

In discrete event simulations we consider, a simulation's state, ordered container of so called events (e.g. in a list, binary tree, splay tree [54], etc.) and a simulation clock. An *event* 'occurs' at a discrete instant in simulation-time (a timestamp) and is the only way the simulation's state is altered. Since no other mechanism is assumed to alter the simulation's state, the simulation can 'jump' in simulation-time from one event to the next (provided the next event has a higher or equal timestamp than the previous). The simulation clock attains the value of the timestamp of the last executed event, and is thus non-decreasing. The *termination time* of a simulation is the simulation time at which the simulation model ends, and thus the simulation terminates. Any events scheduled after the termination time will thus not be executed. Readers who are interested in entering the Discrete Event Simulation field are recommended to read [34]. In the context of Parallel Discrete Event Simulations, Discrete Event Simulation is sometimes also referred to as *Sequential Discrete Event Simulation*.

1.3 Parallel Discrete Event Simulation

The type of Parallel Discrete Event Simulation we will be discussing in this thesis is named 'Model Parallelism' (Section 1.3.3). For completeness we will briefly introduce the two other types of Parallel Discrete Event Simulation, *Application Parallelism* (Section 1.3.1) and *Backend Parallelism* (Section 1.3.2). Throughout this thesis, when we mention Parallel Discrete Event Simulation, we refer to 'Model Parallelism'. Thus for clarity, a 'PDES algorithm' is an algorithm in the Model Parallelism area.

1.3.1 Application Parallelism

Application parallelism is the type of parallelism concerned with the parallel execution of multiple instances of (the same) sequential discrete event simulations. Because we are executing *sequential* discrete event simulations in parallel, we can only execute one simulation per core (more are possible, but not recommended, because it would slow down performance). For example, suppose we have a quad-core processor at our disposal. We can run four instances of the same sequential discrete event simulation (usually with different seed values). This type of parallelism is usually used to quickly get enough simulation results for statistical soundness and, due to its simplicity, is also referred to as "embarrassing parallelism". As mentioned earlier,

we will not address more of this topic in this thesis. One of the downsides of this approach is its memory consumption. Executing four instances simultaneously, requires four times the amount of memory needed for a single simulation. This puts constraints on the size of the simulated model.

1.3.2 Backend Parallelism

Backend parallelism is concerned with parallelizing the backend of Sequential Discrete Event Simulations and thus increase the performance of the simulation. In this case the term 'backend' is used to address all components with which the simulation interfaces. Such systems are (parallel) access to databases, I/O, User Interface, etc. This approach only provides limited speedups.

1.3.3 Model Parallelism

In this thesis (and most of the literature on this subject), the term 'PDES' is used when referring to Model Parallelism. A Parallel Discrete Event Simulation (PDES) distributes a *single* discrete event simulation over multiple processors (e.g. message-based cluster computers) [20]. This allows us to create large models while still exploiting parallelism.

A PDES simulation consists of a set of Logical Processes (Section 1.3.3), usually one per processor, that communicate via 'messages', and work together to correctly simulate the model. One needs to ensure that causality ('cause and effect') constraints (Section 1.3.5) are not violated while exploiting as much parallelism as possible.

Logical Processes

A parallel discrete event simulation consists of a set of Logical Processes (LPs) that are usually a model of an abstracted Physical Process (PP) in the simulation. LPs do not share any state variables, thus we have a distributed state.

LPs operate independent from each other where each LP has its own state and simulation clock. Each LP also maintains a collection of unprocessed events, referred to as its *Future Event List* (FEL). Each LP is responsible for processing the events in its FEL in timestamped order, as not to violate the *Local causality constraint* [22]. This constraint ensures that the entire simulation follows a valid processing order of events, and thus that the simulation model as a whole is executed correctly.

Each LP can be viewed as a separate 'thread' and only communicate via messages. Typically there are two kind of messages. A *Simulation event* message is a timestamped event message. An event message with timestamp t_t sent from LP_i to LP_j indicates that LP_i schedules a new event at timestamp t_t in LP_j . Usually the message also contains a the timestamp at which the message is sent from LP_i , this has no effect in the simulation model but is usually used by PDES synchronization algorithms (Sections 2.1 and 2.2). An event message can only be sent form LP_i to LP_j if, and only if, there is a 'relation' between Physical Process i (PP_i) and Physical Process j (PP_j) in the real world and thus, the simulation model. The other type of messages are used in order for the PDES synchronization algorithms to correctly function. The amount and content of such messages are dictated by the algorithm used. An Logical Process can thus be viewed as a 'local' Sequential Discrete Event Simulation, it has its own state, simulation clock and set of unprocessed events (or Futer Event List). This FEL can contain events received from other LPs and local events. Both of which can generate new events that are either scheduled locally, or scheduled on a neighboring LP.

1.3.4 Parallel Discrete Event Simulation Graph

LPs can send messages to each other if and only if their respective Physical Processes (PPs) can interact with each other. These kind of inter-LP relations can be modeled into a graph, named the *PDES Graph* throughout this thesis. Constructing the PDES Graph is relatively straightforward. Consider a directed graph $G(V, E)$ constructed in the following manner. Each LP_i is represented by a vertex $v_i \in V$ and we include directed edge $e_{i,j} = (v_i, v_j) \in E$ if and only if LP_i is allowed to schedule events on LP_j (equivalently: iff LP_i is allowed to send event messages to LP_j). The resulting graph G thus models the inter LP dependencies, where each LP_i can only receive event messages from its 'incoming neighbors' in G (i.e. The LP at the tail of a directed edge). Edges are sometimes undirected and can be interpreted as two directed edges, indicating that both LP_i and LP_j can send event messages to each other. The *degree distribution* of a (PDES) graph is the (probability) distribution of the degrees of the graph. The degree distribution of a graph with n vertices is given by $P(k) = \frac{n_k}{n}$, with n_k the amount of vertices with degree k .

1.3.5 Causality

Each LP (and Discrete Event Simulator) needs to process all of its events in timestamped order. Suppose we consider an LP that has a number of events

scheduled to be processed. Then it must always execute E_{min} , the scheduled event with smallest timestamp. Suppose it executes event $E_x \neq E_{min}$ first, then it is possible that E_x alters a state variable that is used by E_{min} . Thus creating a situation where the future influences the past. Such an error is referred to as a *causality error*, E_x is then called an *out-of-order event*. Demanding that no causality errors occur is referred to as the *Local causality constraint*, in the context of PDES. It guarantees the valid execution of the simulation model in Parallel Discrete Event Simulation. Formally[22]:

"Local Causality constraint: A discrete event simulation, consisting of logical processes(LPs) that interact exclusively by exchanging timestamped messages, obeys the local causality constraint if and only if each LP processes events in nondecreasing timestamp order."

To illustrate the constraint more clearly we proceed with an example. Suppose LP_1 has processed all events up to event E_1 , and suppose LP_2 similarly up to E_2 . LP_1 executes E_1 . As a result it sends an event-message to LP_2 scheduling $E_3 < E_2$ in LP_2 .



(a) LPs 1 and 2 have processed all events up to E_1 and E_2 respectively (b) E_1 schedules an event before E_2

Figure 1.1: A causality error

But, since LP_2 has already moved past the planned execution time of E_3 , E_3 is scheduled to be executed 'in the past' at LP_2 .

PDES synchronization algorithms ensure that this constraint is not violated. Some of these algorithms are evaluated in Chapter 2.

1.4 The need for Parallel Discrete Event Simulation

PDES simulations are used in a variety of fields, examples are: traffic [62, 47], aviation [59], networking [50] (where the authors use Distributed DES in

combination with PDES) and the global epidemiology of avian influenza (or: "bird flu") [49].

To illustrate the need for PDES research, we consider PDES performance studies conducted over the last 15 years or so. Summarized in [24], studies over the past 15 years have achieved a performance of 138K [25](2003) 32K [45](2007) 187K [10](2009) and 256K [9](2013) events per second *per core*. These number are to be taken with a bit of a grain of salt, as the authors did not execute the same algorithm, implementation or used the same hardware. However, these results do indicate a trend, namely that over a period of 10 years, an increase of 'only' a factor two in events per second per core was achieved. Processor clock rates of single cores in processors are known to be limited by problems with heat dissipation since around the year 2005. Thus more effort is put into increasing the number of cores in (super)computer architectures [61]. Yet, modern problems create larger and larger simulation demands. Since clock rates per core is only seeing mild increase, increases in simulation execution speed will largely have to come from increased parallelism.

1.5 Thesis organization

This thesis is organized as follows, we will first present a literary study (Chapter 2) to introduce the reader to concepts generally used in the PDES field. It aims to place the research we will conduct in this thesis (Chapter 3) in the context of its field. In this thesis we contribute to the PDES field by introducing a new PDES-synchronization benchmarking algorithm (Chapter 4), and a fast new parallelism prediction algorithm (Chapter 6). Next, we will discuss some of the implementation details of the experiment setup (Chapter 7) that we will use to evaluate both contributions (Chapter 8), before concluding this thesis with our findings (Chapter 9).

Chapter 2

A literary survey

Several surveys have been published earlier (e.g. [3], [21], [44] and [30]). Here we present a modest literary survey to give the reader an introduction into the field of Parallel Discrete Event Simulations (PDES).

In general, there are two types of PDES synchronization algorithms that enforce the Local Causality constraint. *Conservative* algorithms (Section 2.1) avoid causality errors from occurring, whereas *Optimistic* algorithms (Section 2.2) detect such errors and recover from them. In Sections 2.1.1 through 2.1.3 we present several variants of conservative algorithms. Whereas Sections 2.2.2 and 2.2.3 do this similarly for optimistic algorithms. We describe the Parallel HOLD benchmarking algorithm PHOLD in Section 2.3.1. The PHOLD algorithm is frequently used to evaluate the performance of PDES synchronization algorithms and will play an central role in the rest of the thesis.

2.1 Conservative Algorithms

In conservative algorithms, an event is blocked from execution until it is certain that executing the event will not violate the local causality constraint. The performance of conservative algorithms are usually largely determined by the the size of the so-called *lookahead* value. An LP is said to have *lookahead* value L , if this LP is able to predict that it will produce no messages in the simulation time interval $[Clock, Clock + L]$, where $Clock$ is the current simulation time (See Figure 2.1 for an example). Thus (roughly speaking) for every LP_i it is safe to process events up to the minimum over the simulation time of all incoming neighbors summed with their respective lookahead value. Thus: $\min_{\text{neighbor } k} (Clock_k + L_k)$, with $Clock_k$ and L_k the simulation clock and

lookahead value of neighbor LP_k of LP_i .

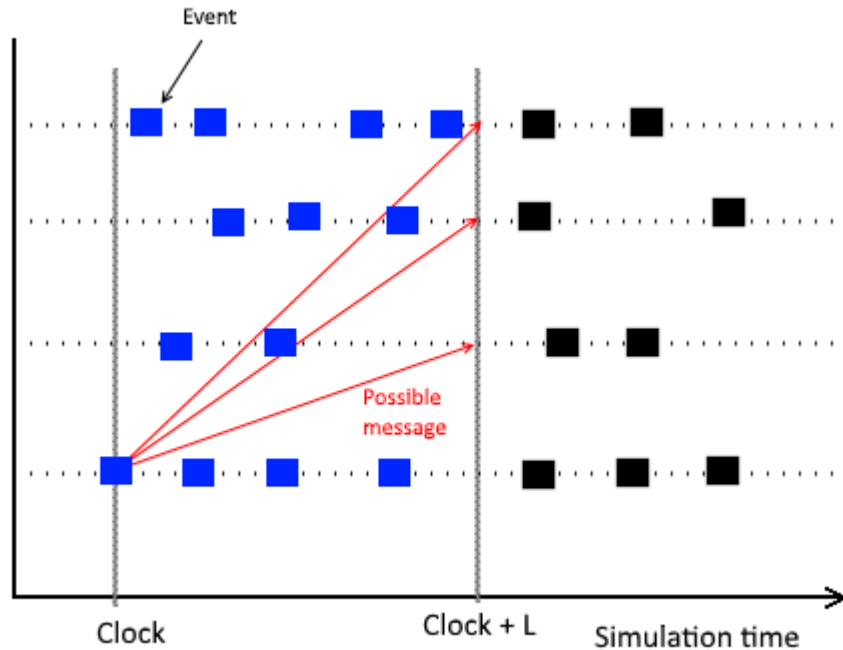


Figure 2.1: Indicating the importance of lookahead a value, here, blue events can be processed safely in parallel, assuming all LPs have lookahead L

Suppose we don't explicitly consider lookahead, then each event, can possibly schedule an event on an other LP with timestamp $Clock + \epsilon$, with $\epsilon > 0$ the smallest timestamp increase (we assume non-zero timestamp increases per event). This implies we have a constant lookahead of the smallest timestamp increase allowed. Having such a small lookahead value L forces LPs to execute a very small amount of events in parallel. The result is that the entire simulation is executed (largely) sequentially. The execution time of such a simulation might actually be greater than the execution time of the sequential DES simulation because we still have the overhead of sending messages between LPs.

In the following sections we consider several types of conservative algorithms. Deadlock-avoidance algorithms (Section 2.1.1), prevent deadlocks from happening, whereas Deadlock detection and recovery algorithms (Section 2.1.2) allow deadlocks to occur. Finally we discuss the Conservative time window type of algorithms in Section 2.1.3, where a set of 'safe' events

is determined and allowed to execute using a global control mechanism. *asynchronous* synchronization algorithms do not have such a global control mechanism.

2.1.1 Deadlock-avoidance

In the late 1970's Chandy & Misra [14], and Bryant[11] independently published the first asynchronous Conservative synchronization algorithm. This algorithm is referred to as the CMB (Chandy-Misra-Bryant) algorithm. The algorithm follows the following main idea. Every time an LP_i sends an event message to a neighboring LP_k , it also sends a *Null-message* to all its other neighboring LPs. This Null message informs the neighboring LPs of a *lower bound on the timestamp* (i.e. its Simulation time plus its lookahead value) of any event message LP_i might send in the future. Neighboring LPs can thus determine a lower bound on the timestamp (LBTS) of all messages it might receive in the future (from all *its* neighbors). Using the LBTS, an LP can determine which events from its FEL are safe to process without risking causality errors in the future. In short, the propagation of Null messages prevent deadlocks. For a formal proof we refer the reader to e.g. [14]

The following example demonstrates a deadlock situation. Suppose our PDES graph is as depicted in Figure 2.2. We assume that 'Source' has no incoming neighbors. All messages in the CMB algorithm are represented as a tuple $(t + L, m)$. Here t is the timestamp of when the message is sent (Thus the originators simulation clock) plus a lookahead value L , and m is the message's content. Here m can take the value of a simulation specific event, or NULL.

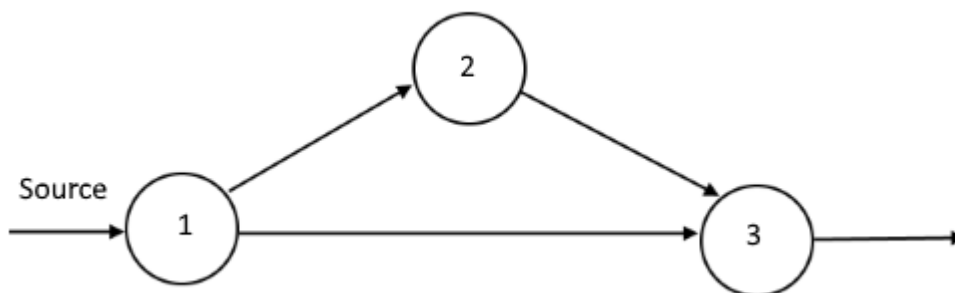


Figure 2.2: An example PDES topology.

Suppose we consider a job shop simulation with termination time Z . LP_1 , LP_2 , and LP_3 are all machines that can process a job in exactly 5 seconds. The Source generates jobs J_1 , J_2 to be scheduled on LP_1 at simulation times 50 and 110 before the simulation time ends at $Z \geq 110$.

In the tuple representation mentioned earlier, the tuple $(t + L, (T, J_i))$ represents that the event-message is sent, from the originating LP at timestamp t , has a lookahead value of L and contains a job (event) J_i to be scheduled, at the receiving LP , at timestamp T . The following sequence of steps will eventually result in a deadlock if no NULL messages are used.

1. Source outputs $(50 + 60, (50, J_1))$ to LP_1 . Here Lookahead L is 60, because the Source does not have any incoming neighbors, and is thus able to ensure that the next event message to be sent to LP_1 is one of its own, and must thus be J_2 , which has a scheduled timestamp of 110.
2. LP_1 executes J_1 , as its event timestamp is less than the received LBTS from the Source.
3. LP_1 outputs $(110 + 5, (55, J_1))$ to LP_2 .
 LP_1 can determine such a high lookahead because it has no events of its own to process, so any events to be executed on LP_1 must come from

the Source. The Source already guaranteed that it will not schedule any events earlier than 110. Since LP_1 also knows that processing a job takes 5 seconds, it can add it to the lookahead that it outputs to LP_2 .

4. LP_2 outputs $(115 + 5, (60, J_1))$ to LP_3 with a similar rationale as mentioned in step 3.
5. LP_3 is unable to determine whether it can safely execute $(60, J_1)$ because it might receive an event-message from LP_1 scheduling an event at a lower timestamp. Thus LP_3 does not execute $(60, J_1)$.
6. Source outputs $(Z, (110, J_2))$ to LP_1 and terminates, as there are no more events to process.
7. LP_1 outputs $(Z, (115, J_2))$ to LP_2 and terminates.
8. LP_2 outputs $(Z, (120, J_2))$ to LP_3 and terminates.

LP_3 has now two pending events, being: $(60, J_1)$ and $(120, J_2)$ which it can not execute as it has no way of knowing if LP_1 will schedule any events on LP_3 and thus the system deadlocks. The CMB algorithm prevents this from happening by sending messages $(115, \text{NULL})$ and (Z, NULL) from LP_1 to LP_3 at steps 3 and 7.

Under the assumption that each event progresses the simulation clock, Null messages also guarantee that the simulation will always progress. When this assumption is relaxed, the system can end up in a *Live lock* caused e.g. by a cycle in the PDES graph where all LPs have a lookahead value of 0. Although there are algorithms that tackle this issue (e.g. [57]), we will assume that all events progress the simulation clock throughout this thesis. In [56] a couple of variants of the CMB algorithms are considered. Among these is the *Demand driven* Null message approach[39]. In contrast to the original CMB algorithm where Null messages are broad-casted, in the demand driven approach, an LP requests a null message from its neighbors when it must wait. This significantly reduces the amount of Null messages sent thus reducing the overhead. One of the downsides of this algorithm is that the request messages grow in size, depending on the amount of nodes in the PDES graph. The Bain and Scott algorithm alleviates this problem by keeping track of pending requests at every LP. Note: The original algorithm is presented in [7], but we were unable to obtain the paper, it is however discussed and explained in [37] and [36].

2.1.2 Deadlock detection and recovery

Chandy and Misra also introduced a conservative algorithm that does not use Null-messages [15]. This deadlock detection and recovery algorithm allows the simulation to end up in a deadlock and has a way of detecting it and recover from it. These algorithms proved to be limited in success, most notably because the simulation is largely processed sequentially leading up to a deadlock.

2.1.3 Conservative time window

Lubachevsky was the first to introduce the idea of using a Moving Time Window in [35]. The idea of a Time Window is to determine the set of safe events that can be executed in parallel at a global level. A window is defined by a lower bound timestamp, and the 'size' of the window. The lower bound B_l is defined as the minimum timestamp of all unprocessed events (over all LPs). The size s of the window is defined such that any event with its timestamp in this window (i.e. $[B_l, B_l + s]$), is safe to be executed in parallel. An iteration (i.e. determining the window and execution of all events in it) is referred to as an *epoch*.

The *YAWNS* (Yet Another Wireless Network Simulator) algorithm, introduced in [41], is an example of an algorithm that uses a time window. This algorithm is relatively simple, and analyzed in [46] to study the potential parallelism of this algorithm on scale-free PDES graphs. Determines a window as mentioned above, by stalling all LPs, determine the lower bound B_l globally, and execute all events in the window $[B_l, B_l + L]$ in parallel.

2.2 Optimistic Algorithms

In contrast to conservative algorithms, optimistic algorithms don't avoid causality errors. Instead, the LP that receives an *out-of-order* event (i.e. an event to be scheduled in the past), performs a 'rollback' of its state and simulation clock, processes the received event, and carries on. An event that causes such a rollback is called a *straggler event*. A straggler event may cause other LPs to also perform a rollback due to the undoing of sent messages. This is referred to as a *cascaded rollback*.

Optimistic algorithms typically consists of a local and a global control mechanism. In short, the local control mechanism governs execution at the LP level (including its local simulation clock), whereas the global control mechanism facilitates in the deletion of redundant memory and progression of the *Global Virtual Time* (GVT), (intuitively) defined as the minimum timestamp over all simulation clocks. In the context of optimistic algorithms, an LP's simulation clock is usually referred to as its *Local Virtual Time* (LVT). The most popular optimistic algorithm is without a doubt, Jeffersons Time Warp algorithm (Section 2.2.1). Since its introduction in 1985, several variants were published, most of which are strictly local control mechanism alternatives (Section 2.2.2) or global control mechanism alternatives (Section 2.2.3).

2.2.1 Time Warp

Jefferson's Time Warp algorithm [31], although published in 1985, is still very relevant today. It is the first optimistic algorithm published and most state-of-the art optimistic algorithms are based on this.

In 2013, the PHOLD benchmarking algorithm (Section 2.3.1) was evaluated on a super computer containing approximately 1.97 million cores. This benchmark ran the Time Warp algorithm with 'reverse computation' (section 2.2.2), producing a record-breaking 504 billion events per second [9].

Below we will introduce the Local and Global control mechanisms for the Time Warp algorithm.

Local control mechanism

In the Time Warp algorithms, LPs are allowed to process events regardless of their neighboring LPs. This puts them at risk for receiving out-of-order events. If such an event is received, a rollback is performed. This rollback has two tasks: restoring the state and undoing the execution of all events that were 'wrongfully processed'.

For example, suppose we consider an LP that is at simulation time 100, and receives a message with a timestamp of (right before) 50. Then the rollback mechanism ensures that the state's values are reverted to timestamp 50, and all events that have been processed that have a timestamp greater or equal to 50 are undone. In order to be able perform the rollback, events must thus only perform at most two undoable actions. i.e.

- 1) Modify state variables
- 2) Send messages to other LPs

Note that we can consider a strictly local event (i.e. events that remain inside the LP and does not send a message to other LPs) as a modification of state variables in the context of a rollback.

In order to be able to revert the state to an earlier point in simulation time, each LP makes and stores a snapshot (i.e. timestamped copy) of its state, prior to executing each event. This method is called *copy state saving*. This method is expensive in both time and memory. Several variations have been proposed to alleviate these costs (Section 2.2.2). When a rollback is performed, an LP restore its state from one of its stored snapshots, and deletes all snapshots that have a timestamp greater than the new simulation time.

A state rollback only affects the current LP locally, whereas a message sent to an other LP might have caused the other LP to also send messages to other LPs and so on. Thus one message sent might affect all LPs in the entire system (!). To undo a sent message, the LP only has to send its *anti-message* to the same destination as the original message. An anti-message is identical to the original (positive) message, but with an extra flag to mark it as an anti-message.

When an anti-message is received by an LP we have two cases.

1. The positive message is not processed yet.
In this case the anti-message 'annihilates' the positive message from the future event list of the LP and both messages are erased from memory.
2. The positive message has already been processed.
In this case the receiving LP is rolled back to the point just before the positive message was processed, and the anti-message annihilates the positive message.

The second case may cause the additional sending of anti-messages to other LPs, which may result in additional rollbacks. The case where a rollback causes rollbacks for other LPs is called a *Cascaded Rollback*.

Global control mechanism

Two major problems arise when trying to perform a rollback

1. I/O operations can not be rolled back.
E.g. writing intermediate output to a database or sending data over a network as a result of the execution of an event.
2. The system can quickly run out of memory if every state snapshot is kept in memory throughout the duration of the simulation.

Both problems can be tackled by calculating a lower bound on the timestamp of any rollback that might occur. This lower bound is referred to as the *Global Virtual Time (GVT)*. State snapshots that are older than the GVT can be discarded safely. Similarly, any pending timestamped I/O operations older than the GVT can safely be committed. (except for the one state snapshot and pending I/O operation right before the GVT, in case of a rollback to GVT). The deletion of obsolete state snapshots is referred to as *Fossil Collection*. Because a rollback is caused by an out-of-order event message, we can define the GVT as the smallest timestamp of the LVT and all messages and anti-messages that have not been processed yet. This includes messages in transit (i.e. a message that has been sent, but not yet received). The original algorithm Computes this value by having a separate process, broadcasting a 'lower bound' request to all LPs at set intervals in *wall-clock* time(real, as in not simulation relation, time). Each LP then calculates its own lower bound by taking the minimum of

1. Local Virtual Time
2. Send times of event/anti-messages that have been sent but not yet acknowledged. I.e. The Local Virtual Times at which the event/anti-message was sent
3. All send times of event-messages that have been received but not yet scheduled. I.e. The Local Virtual Times at which the event/anti-message was sent by the originating LP.

When all LPs have reported their lower bound, the slightly-out-of-date lower bound GVT is computed by taking the minimum over all reported values. A number of alternatives have been proposed since then (Section 2.2.3).

2.2.2 Alternative local control mechanisms

Having to save the current state prior to the execution of each event can quickly put a strain on the available memory if the GVT computation is not accurate enough, or if some LPs are far ahead enough of the GVT.

One approach is called *Incremental state saving*. This method does not save the entire state every time, but only saves the changed variables. Since less information is stored, incremental state saving can reduce the memory footprint of state saves.

A far more complex approach named *reverse computation* [13] ideally does not require any state saves at all. In the case of a rollback, the state is restored by computing the inverse operations of each event that needs to be rolled back. This approach puts additional strain on the simulation programmer as for each operation executed during an event, code has to be inserted to define its inverse. This is undesirable especially for scientists outside the field of Computer Science that typically have less of a programming background. Although this problem can be alleviated (somewhat) by automatically generating the necessary code, not all operations can be inverted (e.g. some bit-wise operations). In these cases incremental state saving is usually used.

2.2.3 Alternative global control mechanisms

Samedi defined two problems that need to be overcome when computing the GVT [51]. A *transient message* is a message that has been sent, but has not yet been received. LPs can be scheduled on different physical CPU cores, even across multiple machines and networks. Thus we can not assume all messages are sent-and-received instantaneously.

The 'transient message' problem is to calculate the GVT while taking transient messages into account. Samedi proposed to use message acknowledgments to address this problem. The other problem arises because different LPs will report their minimum timestamp at different points in wall-clock time and is referred to as the 'simultaneous reporting' problem.

Mattern proposed an alternative algorithm that does not use message acknowledgments [38]. Instead it uses the concept of *cut points*. A cut point is placed on every LP asynchronously, dividing it in a 'past' part, and a 'future' part. The set of all cut points on all LPs thus form a 'cut' of the entire simulation, also dividing it in a past and future part. A *cut message* is a message that is sent from the past part of the cut, and received in the future part, thus crossing the cut. A valid GVT value can then be computed by determining the minimum timestamp of any unprocessed message or anti-

message in the snapshot of the LP at its cut point and of any cut messages. To address the transient message problem, each LP maintains a count of the amount of messages it has sent, and the amount of messages it has received. when both sums over all LPs are equal, there are no transient messages in the system.

A more blunt synchronized '*flushing method*' works as follows:

1. Stop the execution of events.
2. Flush the network of all messages
i.e. Receive all messages
3. Handle all messages
i.e. Schedule or annihilate events or perform rollbacks
4. Goto: 2 if there are still messages to be processed.
5. Determine new GVT, which is the minimum over all unprocessed simulation events.

Because this method is executed synchronously for each LP, and no simulation events are executed, this process will eventually terminate. The result is that no more messages are in transit, and thus the transient message problem is no longer applicable. [42]

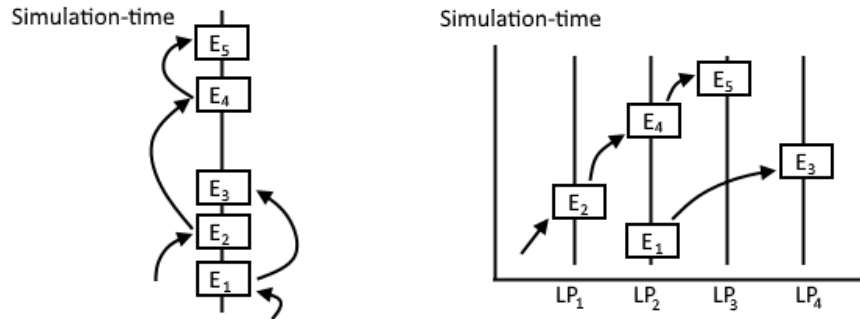
2.3 Benchmarking

After having introduced two main types of synchronization algorithms and several specific algorithms, a natural question one asks is "What is the fastest algorithm?". The answer to this question is "It depends". There are several ways to compare such algorithms. We could perform an actual simulation, and experiment with different kinds of algorithms. This has the disadvantage that results don't necessarily hold for simulation models in other fields.

Having a speedup in execution speed due to parallelism is obviously nice, but it generally is not the case that the speedup scales linearly with the amount of threads. E.g. having 4 threads might speed up the execution of the simulation by a factor of (only) 2. To analyze these kinds of speedups (or slow-downs...) obtained by a synchronization algorithm, it is common to use the PHOLD benchmark (Section 2.3.1). Alternatively, one can attempt to predict the potential amount of parallelism that can be obtain by looking at the PDES topology from a mathematical point of view (Section 2.5). This can be useful in cases where one attempts to map LPs onto the simulation model such that it utilizes parallelism as efficient as possible (Section 3.1.2)

2.3.1 PHOLD

The PHOLD benchmark [23] abstracts away any domain specific modeling aspects, and benchmarks the PDES synchronization algorithms themselves. The PHOLD benchmark is the parallel version of HOLD [58], a benchmarking algorithm for sequential discrete event simulation algorithms (primarily focused on the performance of the Event Queue). In the HOLD algorithm, a set of events is scheduled, each scheduling a new event with an inter-event-timestamp (i.e. the simulation-time between two events) drawn from an exponential distribution. The PHOLD algorithm takes the PDES-graph and schedules events uniformly random across all LPs. Executing an event results in a new event being generated and scheduled, or sent to another LP. This 'other' LP is chosen also uniformly random from the LP's neighbors. Finally, each event is scheduled at the LP's current simulation time, increased by a value drawn from an exponential distribution, usually summed with a constant 'lookahead' (especially when evaluating conservative algorithms). Although the original PHOLD paper evaluated the PHOLD performance of multiple distributions, it is common to only use the exponential distribution (see e.g. [28], [9], [46], [29]) This process ensures that the same amount of messages remain present in the system and all LPs are treated equally. Figure 2.3 shows an example of a possible event scheduling progression for both the HOLD benchmark and the PHOLD benchmark.



(a) A possible HOLD execution

(b) A possible PHOLD execution

Figure 2.3: Event scheduling of a possible HOLD and PHOLD execution

Defining more LPs than there are processors is called *saturated* , whereas having more processors than LPs is called *unsaturated*.

PHOLD has several important deficiencies that can lead to misleading performance results. As summarized in [61], these are.

1. Most uses of PHOLD use highly regular topologies such as e.g. a fully connected network.

2. PHOLD does not consider events with different computational requirements.
3. Each event schedules a single new event on a neighboring LP where each LP is equally likely to be selected.

One of the goals of this thesis is to extend the PHOLD benchmark (Chapter 4) in such a way that its results represent the performance of PDES simulations more accurately.

2.4 Scale Free networks

The *degree distribution* of a graph $G(V, E)$ can be obtained by taking $degree(v) \forall v \in V$. A *scale free network* [8] is a network, for which the node degree distribution follows a *power law distribution*.

$$\Pr(k) \sim k^{-\lambda} [16] \quad (2.1)$$

With node degree k , and parameter λ (sometimes referred to as its *degree exponent*) that usually lies between 1 and 3. Thus $\Pr(k)$ approximately expresses the probability of selecting a node with exactly degree k . Intuitively this means that there are few nodes with high degree (so called *hubs*) and many with low degree (or *leafs*). Figure 2.4 depicts an example graph. A more mathematically thorough analysis of the Scale Free topology with regards to PDES graphs can be found in Chapter 10 in the Appendix.

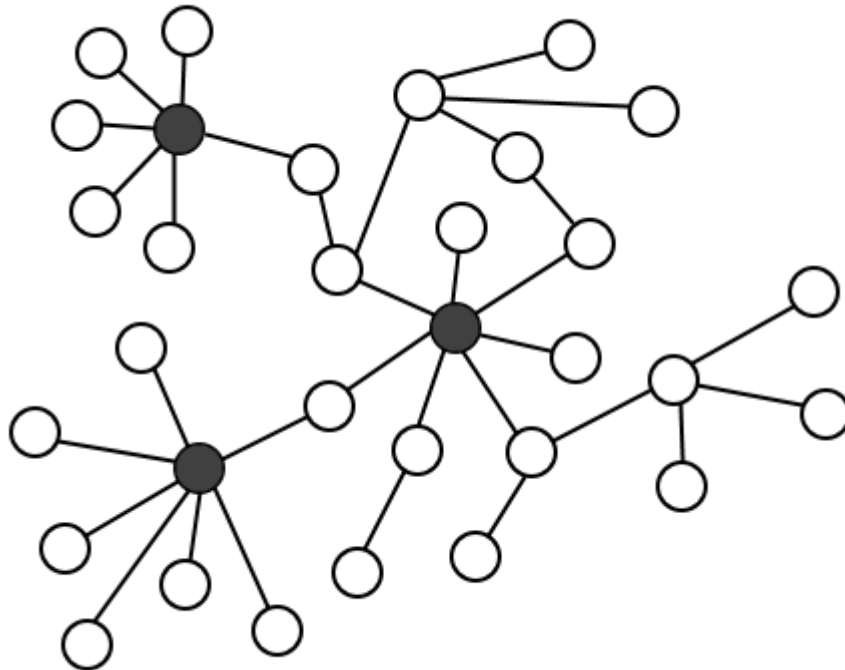


Figure 2.4: An example of a scale-free network, the hubs are highlighted in gray

Many real-life problems exhibit scale-free properties. Some examples are protein-protein interaction networks [32], some financial networks, such as the

interbank payment network [55]. The autonomous system (or 'inter domain') level of the Internet is also believed to be scale-free [53]. Constructing a PDES-graph from such models naturally also results in a Scale-free topology.

The performance increase for scale-free PDES topologies is highly dependent on the value of λ [46]. It turns out that PDES algorithms provide only a modest speed increases for cases where the PDES graph is a scale-free network with small values of λ .

Pareto distribution

A power law is a reformulation of the *Pareto distribution* [6]. Its probability distribution function(PDF) and Cumulative distribution function (CDF) are given by:

$$f(x) = \begin{cases} \frac{\alpha x_m^\alpha}{x^{\alpha+1}} & x \geq x_m \\ 0 & x < x_m \end{cases} \quad (2.2) \quad F(x) = \begin{cases} 1 - (\frac{x_m}{x})^\alpha & x \geq x_m \\ 0 & x < x_m \end{cases} \quad (2.3)$$

Where x_m is the positive minimum value of x , and parameter α .

2.5 Analytical modeling of potential parallelism

Mathematically modeling the execution of PDES simulations is a relatively new direction that the PDES field is taking. Of special interest is the PDES topology, and its effect on the potential amount of parallelism (i.e. 'how many events can be executed in parallel').

Due to its simplicity, the YAWNS algorithm is commonly analyzed, arguing that the results transfer to other synchronization (conservative) algorithms.

Andelfinger and Hartenstein performed 'white-box' analyses on three different network models in [4]. They propose an analytical model that estimates potential parallelism based on specific simulation model knowledge and statistics obtained from sequential simulation runs. This limits its application for (very) large scale models as it might not be possible to run these on sequential simulations due to memory and/or time constraints.

Pienta and Fujimoto introduced an iterative analytical model in [46] to approximate the potential parallelism in cases where the PDES topology forms a scale-free network. They explicitly make a distinction between hub nodes

and leaf nodes and consider them differently. The authors derive the model by mathematically approximating a PHOLD execution for scale-free networks with $2 < \lambda < 3$ (This limit is imposed by some of the integrals they use).

Chapter 3

Research

In Section 3.1 we present our research questions and discuss their relevance in Section 3.2. Given these questions we plan to employ the methodologies presented in section 3.3 to answer them.

3.1 Questions

We distinguish two main topics in this thesis, all having to do with benchmarking PDES synchronization algorithms. These topics are PHOLD (Section 3.1.1) and analytical modeling (Section 3.1.2)

3.1.1 PHOLD

PHOLD [23] is widely used to measure the performance of PDES synchronization algorithms. While useful to the PDES field in its early years, it fails to capture important aspects of real PDES simulations. Most uses of PHOLD use regular topologies such as a toroid or a fully connected network with the message sent to a neighboring LP with any neighbor equally likely to be selected. The problem with this is that a toroid, or fully connected graph leads to a highly symmetric network with a well balanced computation workload per LP. Most performance studies (such as most of the ones mentioned in Section 1.4) use highly regular networks. Such topologies are very different from real-world networks that are typically irregular, with skewed degree distributions. There are cases where a simulation modeler can reasonably predict if and where (at what LP) a larger bulk of the events will be processed. (e.g. Near exits in case of an evacuation of a building (crowd simulation). On, near or toward highway exits at start/end of business day (road traffic simulations), etc. Currently, the PHOLD benchmark does not facilitate in using this information.

PDES simulations are also run on (expensive) super computers, in this case, having to 'guess' which synchronization algorithm will perform best is less than ideal. Specifically for Time Warp, an interesting performance measure, is to consider the amount of rollbacks. This is an interesting measure as rollbacks are the main cause of performance degradation in the Time Warp algorithms. We plan to extend the PHOLD algorithm in such a way that we can answer the following research questions.

1. What happens to the amount of parallelism of a PHOLD execution if events are (unevenly) distributed during the benchmark.
2. What is the relation (if there is one) between rollbacks and the value of λ in scale-free PDES topologies?

3.1.2 Analytical modeling for a given PDES graph

An analytical model, in this context, expresses the potential amount parallelism that can be obtained for a given PDES graph. Of special interest are Scale Free networks. Many problems exhibit scale-free like structures and parallelizing these proves challenging. An analytical model, that is able to predict the amount of parallelism in a PDES graph might provide insights that lead to the development of new, better PDES synchronization algorithms.

Why do we bother with such analytical modeling? Well, it concerns a topic we will not touch in this thesis. This is the problem of constructing a PDES graph from the original simulation graph (where each node is e.g. a source, sink, server, etc. and each edge represents a possible 'communication' between such entities) by grouping nodes into LPs. Two LPs have an edge between them if, and only if one or more of their internal simulation nodes also have an edge between them. Constructing an PDES graph of k LPs (vertices in the PDES graph) from a simulation graph, while minimizing the amount of edges between LPs is NP-Complete. This k -way graph partitioning problem is formally defined as follows. Given a graph $G(V, E)$ and integer $k > 0$, find k disjoint subsets of V of (almost) equal size, such that the amount of edges between the subsets is minimized. This problem is shown NP-Complete in [5].

Often, some form of Local search algorithm is used with a fitness function to tackle such a problem. In this case we would use a function of the amount of edges as a fitness function. We hypothesize that an (general) analytical model that expresses the potential parallelism, taking into account the resulting topology, will be a better fitness function compared to only minimizing the amount of edges.

Most synchronization PDES algorithms are developed for general PDES topologies (one exception is: [33] which was developed for feed-forward networks). Knowing more about the relation between the scale free topology and parallelism, might aid in the development for synchronization algorithms that work well for scale free networks.

In Section 3.1.1, we proposed to extend the PHOLD benchmark. Here we propose to develop an analytical model (or algorithm) that makes predictions for this extended version of the PHOLD benchmark. A good prediction model of the extended PHOLD benchmark, should be able to answer most of the same questions asked for the extended PHOLD benchmark. Natural

questions are thus:

1. How well can our analytical model approximate the amount of parallelism for a given PDES graph?
2. What happens to the amount of parallelism of a PHOLD execution if events are (unevenly) distributed during the benchmark. I.e. Some LPs have to process significantly more events than others.

Note: When we started working on this thesis we first wanted to extend the analytical model presented in [46]. This model was specifically designed to predict the amount of parallelism present in the topology of general case scale free networks. The authors mathematically derived their model using primarily the mathematical properties of scale free networks. Resulting in a model that attempts to predict the amount of parallelism for different values of λ .

When we started working on the implementation of this model in Mathematica [60] we realized that we could use some of its aspects to develop a prediction algorithm for *any* given PDES graphs. Using this new prediction algorithm, we can predict the amount of parallelism for complete graphs and instances of scale free networks. We decided that this was a more interesting undertaking than the original, because of its wider applicability.

3.2 Relevance for science, technology and society

Many interesting fields of research exhibit scale free properties (See Section 2.4 for some examples). Yet simulating very large model that exhibit such properties proves troublesome using the PDES synchronization algorithms known to date. PHOLD attempts to benchmark PDES synchronization algorithms but lack several important aspects, limiting its applicability. Enhancing PHOLD in such a way that it better approximates a PDES simulation, increases the validity of comparing different PDES synchronization algorithms. Having this, hopefully encourages the development of PDES synchronization algorithms that are (also) able to utilize large amounts of parallelism in settings where the PDES graph exhibits scale free properties. Hopefully, both the extension of the PHOLD benchmark and the approximation model, provide more insights into the relation between the PDES graph topology and parallelism. Ultimately we hope this aids in the development of synchronization algorithms that (also) work well for e.g. scale free networks. Having good synchronization algorithms for such networks might

enable researches to develop complex global-scale models. E.g. To research the global effects of new packet level protocols for the Internet.

3.3 Methodology

In this Section we describe the methodologies we plan to employ to answer our research questions. In Section 3.3.1 we will discuss the approach for answering the questions relating PHOLD and Section 3.3.2 will do this similarly for analytical modeling questions.

3.3.1 PHOLD

We will implement the standard PHOLD benchmark and run an implementation of the CMB algorithm, the YAWNS algorithm, and the Time Warp algorithm (with incremental state saving, and the network flushing method of calculating the GVT).

These three algorithms are chosen because they are (still) commonly used in the PDES field.

The PHOLD benchmark will run on a number of complete, and scale free PDES graphs with varying values of λ . These graphs will be generated using the statistical software application **R** [48] using the **igraph** package [17] that uses a generalized version of the Barabasi-Albert algorithm [8]. This method allows us to generate scale free networks where we can define the values of λ and amount of nodes.

We will give a definition of the *average parallelism*. For YAWNS this is the fraction of the amount of events the 'bottleneck LP' has to process, over the total amount of events that are processed in the whole window. For the CMB algorithm this is the number of Null messages vs. number of event messages. For the Time Warp algorithm, we will use the amount of rollback events.

We will setup experiments where we schedule events un-evenly across LPs during the simulation. This will answer all questions from Section 3.1.1. Most PDES simulators available either implement some form of the Time Warp algorithm (e.g. [12], [2]), or an conservative algorithm (e.g. [1] that implements an algorithm as described in [43], and the NS-3 simulator [29]). To our knowledge there is no simulator available that is able to switch between these two types of algorithms. We could use separate simulators, but we would have to assume that the simulators are equally well written, and one simulator does not introduce more overhead than the other. This assumption does not hold in general for different simulators. Ideally we'd have

a simulator that is able to utilize both types of algorithms while sharing as much code as possible to increase the 'fairness' of the comparison.

3.3.2 Analytical modeling for given PDES graphs

Developing the prediction algorithm primarily requires a pen-and-paper approach and will be based on the YAWNS algorithm. Answering the questions proposed in Sections 3.1.2, requires running experiments and comparing the algorithm's predictions to experimental results.

Chapter 4

Extending PHOLD

The original PHOLD benchmarking algorithm lacks several aspects that limit its applicability. In this chapter we will define the PHOLD algorithm (Section 4.1). This allows us to accentuate the extensions we propose (Section 4.2) to form a more generalized version of the PHOLD algorithm i.e. The 'Extended PHOLD' (EPHOLD) benchmarking algorithm. This algorithm will allow us to differentiate between LPs within the algorithm. Finally, we will see that we can simulate PHOLD using this new algorithm, and more (Section 4.3).

4.1 A more formal definition of PHOLD

In PHOLD, each event executed schedules 1 new event and roughly consists of 2 phases, the *initialization* phase, and *operational* phase. The initialization phase constructs the LPs and schedules the initial set of events across LPs. The operational phase executes events per LP.

The initial PHOLD algorithm [23] defined 6 parameters:

1. *Number of LPs*
2. *Message population*: A positive integer, indicating the amount of events in the entire system.
3. *Timestamp increment function*: A function that determines the timestamp increments between events. I.e. the difference between the timestamp of the event currently being executed, and its newly generated event.
4. *Movement function*: Determining where a message is sent to

5. *Computation grain*: The computational intensity required for event execution. Note that this affects wall-clock time, not simulation time.
6. *Initial configuration*: Defining the initial distribution of events across the LPs.

We will discuss these parameters one by one:

The *Number of LPs* N is an positive integer value, indicating the amount of LPs used in the PHOLD benchmark.

The *Message population* M is the summed amount of events present in the entire system. I.e. At not point during the benchmark will there be any more, or less events present in the system than M . Throughout the PHOLD benchmark, no LPs will be deleted, and no new LPs will be introduced.

The *Timestamp increment function* f_T is a function that generates a timestamp increment value (usually taken from some kind of distribution e.g. $Exp(1)$). This value, summed with a (fixed) lookahead value L , and the current simulation time, defines the timestamp of the newly generated event. As such, we demand that $f_T \geq 0$ holds.

The *Movement function* f_M , in PHOLD determines where new events are sent to. In PHOLD, when LP_i generates a new event to be scheduled, in general, two components need to be computed i.e. The timestamp of the new event, and the neighboring LP_j the event will be scheduled on.

The timestamp of this new event is generated using f_T as mentioned above. The event will be scheduled at neighbor LP_j . This destination LP_j is computed by f_M which, in PHOLD, selects a neighboring LP j from LP_i 's neighbors, discrete uniformly random.

The *computation grain* f_C determines the amount of 'work' that needs to be done to execute an event in this instance of PHOLD. With this, the computational intensity of all events can be configured. Note that the computation grain affects *wall-clock* time, not simulation time. In almost all PHOLD implementations this function is not used.

Finally the *Initial distribution* f_I determines how many events are scheduled at which LPs.

The initialization phase uses parameters 1, 2, 3 and 6. The operational phase uses parameters 3, 4 and 5. At first glance, these parameters seem to be all that we need to approximate the performance of PDES simulations.

Note however, that these parameters are defined over the entire system, and are thus not customizable per LP. As far as we know, there are no PHOLD implementations, or PHOLD experimental results conducted with LP specific characteristics.

Before extending the PHOLD benchmark, we will first introduce the notion of a *Space-time diagram*. A Space time diagram can be seen as a 2 dimensional grid. The vertical axis represents the progression in simulation time, whereas the horizontal axis indicates the N LPs of the whole PDES simulation (i.e. the 'space'). Figure 4.1 is an example of a space-time diagram. Note that we already showed this Space-time diagram without mentioning its concept in Figure 2.3b in Section 2.3.1. Also note that Figure 2.3a depicts a 1 Dimensional space-time diagram.

Simulation-time

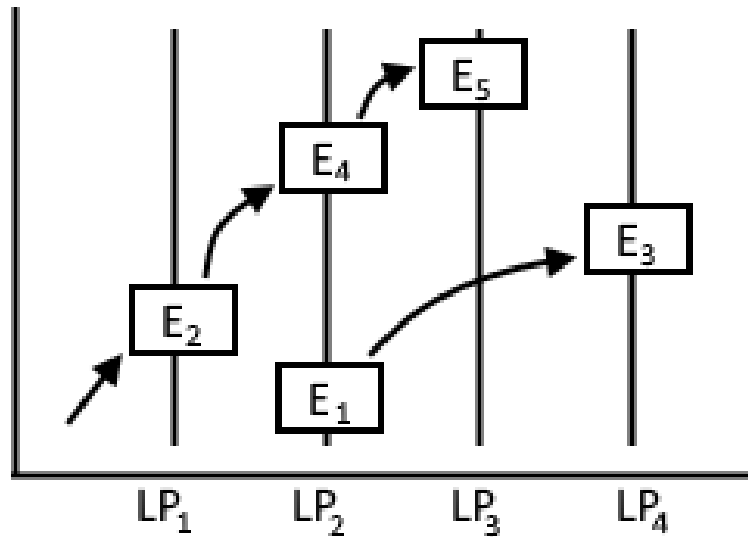


Figure 4.1: A possible PHOLD execution, depicted in a 2D space-time diagram

We assume (throughout this thesis) that the amount of LPs is finite and fixed (as a parameter in PHOLD), thus no LPs are dynamically created or deleted during the PHOLD execution. Assuming there are N LPs, LP_1, \dots, LP_N , LP_i is mapped to spatial coordinate i (Assuming we start at 1). An arc extending from space-time coordinate (s, T_s) to (r, T_r) represents that the execution of an event at simulation-time T_s on LP_s sends an event-message to (i.e. 'schedules an event on') LP_r at time T_r , where $T_r > T_s$. In the paper in

which PHOLD is introduced[23], Fujimoto divided up the spatial coordinates into an x and y value. The point (x, y, T) essentially gives an LP a coordinate in a 2D plane. The Movement function then consisted of selecting a point in the neighborhood (i.e. the LPs within a certain (euclidean) distance) of the LP in the 2D plane. This made sense in the early days of the PDES field, nowadays we have the concept of a PDES-graph, for which finding appropriate movement functions in this manner can get 'messy' relatively quick. Instead we will define PHOLD in terms of its PDES-graph $G(V, E)$.

Vertex $v_i \in V$ represents LP_i and we have that $(v_i, v_j) \in E \iff LP_i$ can send event messages to LP_j (thus Physical Proces i (PP_i) can 'interact' with PP_j). First we define the set of all *outgoing neighbors* for a given vertex v_i as $N_i = \{v_j \in V \setminus \{v_i\} : (v_i, v_j) \in E\}$.

The Number of LPs N is given by the cardinality of V , $|V|$. The message population M remains the same. The Movement function f_M called for LP_i , now returns some vertex $v_j \in V$ where $(v_i, v_j) \in E$, or equivalently, we demand that: $f_M(v_i, N_i) \in N_i$ holds. The computation grain function f_C remains the same. Finally, the initial configuration f_I , indicating the amount of events to be scheduled per LP, is usually defined by $\frac{N}{|V|}$ rounded to some integer, for the total amount of events to be initially scheduled N . Note that e.g. the Timestamp increment, movement, and computation grain functions function usually use an initialized Pseudo Random Number Generator and thus returns different results (with high probability) every time they are called.

Overview 4.1 summarizes all previously mentioned parameters of PHOLD.

$$\begin{array}{l}
 \text{Number of LPs} \\
 \text{Message population} \\
 \text{Timestamp increment} \\
 \text{Movement} \\
 \text{Computation grain} \\
 \text{Initial configuration}
 \end{array}
 \left(\begin{array}{c}
 N \\
 M \\
 f_T \geq 0 \\
 f_M(v_i, N_i) \in N_i \\
 f_C \geq 0 \\
 f_I(M)
 \end{array} \right) \quad (4.1)$$

PHOLD does not allow us to adjust e.g. the timestamp increments depending on what LP is executing an event. (e.g. in Scale-free networks, if the function is called from a hub node, or a leaf node).

4.2 Adding more details

In this section we will give a more generalized definition of the PHOLD algorithm, by adapting some of its functions. This will provide us with more

fine-grained control over the output of these functions. In Chapter 8 we will use this more generalized model and evaluate it empirically. Instead of distributing all events evenly across all LPs, we argue that a simulation designer can sometimes already predict, to some extent, which LPs will process relatively more events, and which will not. (e.g. end of business day at 17:00 vs. 13:00 in road simulations). To incorporate this kind of information, we re-evaluate the PHOLD movement function (Section 4.2.1), Initial Configuration (Section 4.2.2), Timestamp increment function (Section 4.2.3) and computation grain function (Section 4.2.4). Finally, in Section 4.2.5 we present the Extended PHOLD model.

4.2.1 The movement function

The movement function governs the movement of events through the LPs, following the PDES graph. In PHOLD, when an event is executed at LP_i , it schedules a new event on a neighboring LP_j , where j is chosen uniformly random from LP_i 's outgoing neighborhood in the PDES-graph. We argue that this is not always realistic to do, because it causes an even and symmetric workload. (e.g. especially in complete PDES-graphs). To alleviate this issue, we propose to assign weights to edges, and incorporate these weights in the movement function f_M .

We define $w_{i,j}$ as the *weight* of directed edge $(v_i, v_j) \in E$. We assume $0 < w_{i,j} : \forall (v_i, v_j) \in E$, i.e. all edges have non-negative weights. Furthermore, we define the set relating these weights to neighbors, per LP_i as: $\mathcal{W}_i = \{(w_{i,j}, v_j) : \forall v_j \in N_i\}$. Thus an element of \mathcal{W}_i is a tuple containing an outgoing neighbor v_j of v_i , and the weight $w_{i,j}$ of the edge between them. In other words, its the set of 'weighted outgoing neighbors' for v_i . Now we pass \mathcal{W}_i as argument to our movement function f_M . I.e. we have $f_M(v_i, \mathcal{W}_i) \in N_i$. This way, if LP_i executes an event, it can select a neighboring LP to send a message to with a probability that is proportional to the weights of its neighbors.

4.2.2 Initial configuration

We define the initial configuration's type as a function $f_I(G, M)$ that produces an array of $|V|$ tuples (v_i, m_i) , indicating the amount of events m_i to be scheduled, on v_i (i.e. LP_i) during the initialization phase. It still takes the total amount of events to be scheduled as a natural number, but now also takes the PDES graph as an argument. This allows us to e.g. differentiate between nodes with large neighborhood, and those with small neighborhood.

4.2.3 Timestamp increment

We would like that the timestamp increment function f_T is able to differentiate based on what LP is doing the computation. To this end we now pass the LP i that is calling the function: $f_T(v_i)$. We assume that the function is first initialized with $|V|$ (different) increment functions, one per LP . Thus by passing v_i to the function, it is able to select the appropriate increment function. This way, we can e.g. have $f_T(v_1) \sim Exp(1)$, and $f_T(v_2) = Exp(2)$, if we wanted to.

4.2.4 Computation grain

Like we did with the other functions, we want to have the ability to differentiate between LP s. To this end we also pass the calling LP as a parameter i.e. $f_C(v_i)$. Again, we assume that the function is first initialized with $|V|$ (different) computation grain functions. The parameter v_i can then be used to select the appropriate function.

4.2.5 Extended PHOLD

In earlier sections we proposed several changes to several PHOLD parameter functions. Overview 4.2 gives this in one clear overview and is a reformulation of expression 4.1.

$$\begin{array}{l}
 \text{Number of LPs} \\
 \text{Message population} \\
 \text{Timestamp increment} \\
 \text{Movement} \\
 \text{Computation grain} \\
 \text{Initial configuration}
 \end{array}
 \left(\begin{array}{c}
 N \\
 M \\
 f_T(v_i) \geq 0 \\
 f_M(v_i, W_i) \in N_i \\
 f_C(v_i) \geq 0 \\
 f_I(G, M)
 \end{array} \right) \quad (4.2)$$

4.3 Examples

In this section we will provide two different examples. The first example (Section 4.3.1) will go through all parameters of the EPHOLD algorithm. This example will later on be analyzed for its parallelism and experimentally verified (Sections 6.7.2 and 8.2.1 resp.) The second example (Section 4.3.2) will illustrate that we can also use EPHOLD to simulate the PHOLD algorithm. Similar to the first example, this one will also be analyzed and experimented with (Sections 6.7.1 and 8.2.1 resp.)

4.3.1 Complete graph, different weights

For this example, let our PDES graph $G(V, E)$ be a complete graph of 4 nodes, we construct 4 LPs and our message population is 40 events. For our Timestamp increment function, we use an $Exp(1)$ distribution for all events executed at even-numbered LPs, and $Exp(2)$, for all events executed at odd-numbered LPs. Our movement is governed by the following function: Let x be drawn from a $U(0, 1)$ distribution. Let the weights be j for any edge going to LP_j , normalized proportional to the rest of the outgoing neighbors, s.t. all outgoing weights sum to 1. Send the event, from LP_i , over outgoing edge $w_{i,k}$, when the following holds:

$$w_{i,k-1} < x \leq w_{i,k} \quad (4.3)$$

For all normalized weights of outgoing edges. Our computation grain is a constant function, and our Initial configuration is 5, 15, 7, 13 events scheduled at LP_1, LP_2, LP_3, LP_4 respectively. This example is shown in figure 4.2.

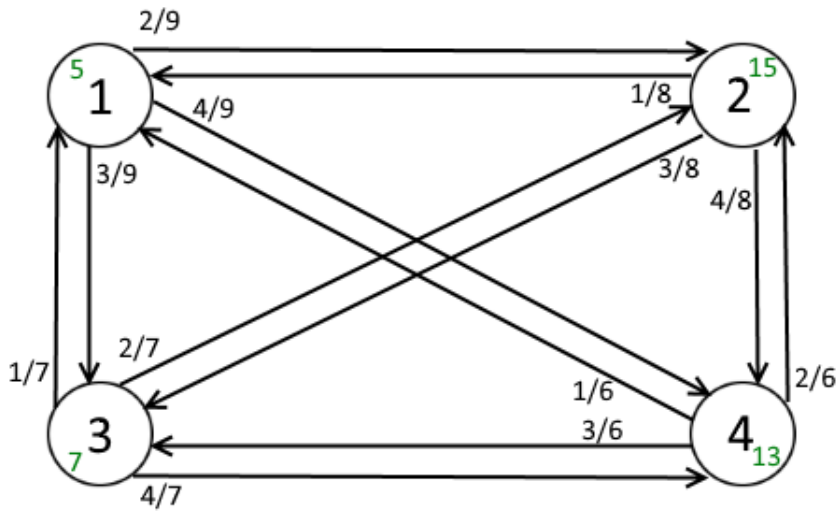


Figure 4.2: Snapshot of the execution of the EPHOLD algorithm at timestamp 0, right after the simulation is initialized. The green numbers indicate the amount of events scheduled per LP. The probability of sending an event over a particular edge (i.e. weight) is drawn at the *tail* of the edges

We define timestamp increment vector R as:

$$R = \begin{pmatrix} Exp(1) \\ Exp(2) \\ Exp(3) \\ Exp(4) \end{pmatrix}$$

Where the i -th element r_i , of R is the timestamp increment function for LP_i . Given a fixed lookahead L , a possible execution can be as follows:

1. LP_1 executes an event
2. LP_1 draws $x \in U(0, 1)$, which results e.g. in selecting outgoing edge (LP_1, LP_4) (which has probability $\frac{4}{9}$ of being selected)
3. LP_1 schedules an event at LP_4 , with timestamp increment of $L + X$, with X drawn from e.g. $Exp(1)$
4. LP_2 executes an event
5. LP_2 draws $x \in U(0, 1)$, which results e.g. in selecting outgoing edge (LP_2, LP_3) (which has probability $\frac{3}{8}$ of being selected)
6. LP_2 schedules an event at LP_3 , with timestamp increment of $L + X$, with X drawn from e.g. $Exp(2)$

Notable differences with the PHOLD algorithm are the fact that we draw timestamp increments from both $Exp(1)$ and $Exp(2)$ distributions, and the way we select edges to send events over. The next example (Section 4.3.2) shows us how we can simulate PHOLD using the same algorithm.

4.3.2 Simulating PHOLD

In this section, we will show that PHOLD executions form a subset of EPHOLD executions. Consider the example we gave in the previous Section. To simulate PHOLD, all we need to do is set all weights to $\frac{1}{3}$, distribute the initial amount of events uniformly across all LPs (e.g. 10, 10, 10, 10). And draw our timestamp increments (in practice usually) from and $Exp(1)$ distribution at each LP. Figure 4.3 shows this instance.

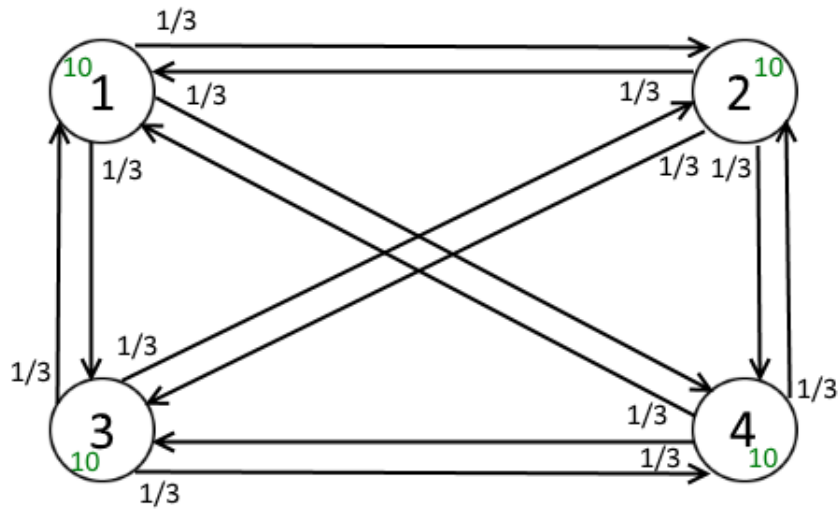


Figure 4.3: An initialized example of the EPHOLD algorithm where the green numbers indicate the amount of events scheduled per LP

An example execution follows the same logic as the EPHOLD algorithm. Given a fixed lookahead L :

1. LP_2 executes an event
2. LP_2 draws $x \in U(0, 1)$, which results e.g. in selecting outgoing edge (LP_2, LP_4) (which has probability $\frac{4}{9}$ of being selected)
3. LP_2 schedules an event at LP_4 , with a timestamp increment of $L + X$, with X drawn from $Exp(2)$
4. LP_1 executes an event
5. LP_1 draws $x \in U(0, 1)$, which results e.g. in selecting outgoing edge (LP_1, LP_3) (which has probability $\frac{3}{8}$ of being selected)
6. LP_1 schedules an event at LP_3 , with a timestamp increment of $L + X$, with X drawn from $Exp(1)$

Chapter 5

Definitions of parallelism

Earlier (Section 3.1.1) we posed questions regarding the amount of parallelism, and the influence of topology, and message flow on it. A straight forward way of measuring parallelism is to consider the amount of (committed, in the case of Time Warp) events per time unit (e.g. per second). The obvious advantage of this approach is being able to compare the performance of all different synchronization algorithm. We argue that this approach is highly influenced by the hardware used to perform the experiments on. E.g. since the Time Warp algorithm typically accesses the computer's memory more often than e.g. YAWNS or CMB, one can 'increase' Time Warp's parallelism by using quicker memory. Thus this makes comparing experimentation results from different researchers less straight forward.

Instead we intend to define parallelism in terms of the 'overhead' introduced by each synchronization algorithm. This does not allow us to compare the amount parallelism between e.g. Time Warp and YAWNS, as they are too different. But this does allow us to reliable compare e.g. different variations of Time Warp

In this chapter we will give definitions of parallelism for the YAWNS (Section 5.1), CMB (Section 5.2) and Time Warp (Section 5.3) synchronization algorithms.

5.1 YAWNS

In conservative window based PDES synchronization algorithms, the LP with the longest running time will become the bottleneck. (This can be because it has the most events to process, or the most computationally demanding, or both). Other LPs will thus have more idle time than the bottleneck LP. The

more the amount of events, per window varies between LPs, the more idle time we thus have. Throughout this section, we will assume that the execution of an event at LP_k takes $p_k \in \mathbb{R}$ units of wall-clock time. The value of p_k is determined by the *Computation grain* function in EPHOLD. Although we can have different types of events, having different computational needs, we argue that we can still approximate an average case computational need per LP. Let the number of events, per window, for LP_k , be $n_k \in \mathbb{N}$, the *bottleneck LP* per window is the LP that has to do most of the work during the window. We can compute the bottleneck LP of any window as follows:

$$\arg \max \{n_k \cdot p_k\} \quad (5.1)$$

I.e. the expression iterates over all LPs and returns the index k of the bottleneck LP. Let e_{max} be the amount of wall-clock time it takes to process all the events in a window for the bottleneck LP. A logical definition is thus:

$$e_{max} = \max \{n_k \cdot p_k\} \quad (5.2)$$

The total amount of *work* W per window (i.e. all events in the window) is then defined by:

$$W = \sum_k n_k \cdot p_k \quad (5.3)$$

The amount of parallelism P obtained in a window is given by dividing the amount of work that needs to be done, by the amount of time it actually takes. Thus $0 < P \leq |V|$, P also expresses the speedup that can be obtained versus a sequential discrete event simulation. A logical definition is thus:

$$P = \frac{W}{e_{max}} \quad (5.4)$$

We will illustrate the use of these variables by a small example. Suppose we consider a complete PDES graph of four nodes, thus $V = \{LP_1, LP_2, LP_3, LP_4\}$ and $E = V \times V \setminus \{(v_j, v_j) \mid v_j \in V\}$ the Cartesian product of V with itself, excluding self loops. Furthermore, suppose we have 2, 3, 4 and 3 events scheduled to be executed with running time $p_k = 1$, in the current window, at LP_1, LP_2, LP_3, LP_4 respectively.

The bottleneck LP is LP_3 , with $e_{max} = 4$, $W = 2+3+4+3 = 12$, resulting in parallelism $P = \frac{12}{4} = 3$. Thus, a speedup of factor 3 can theoretically be obtained when compared to a sequential DES (for this set of events), where 4 is the maximum in this case. Figures 5.1 and 5.2 show the interpretation of these values.

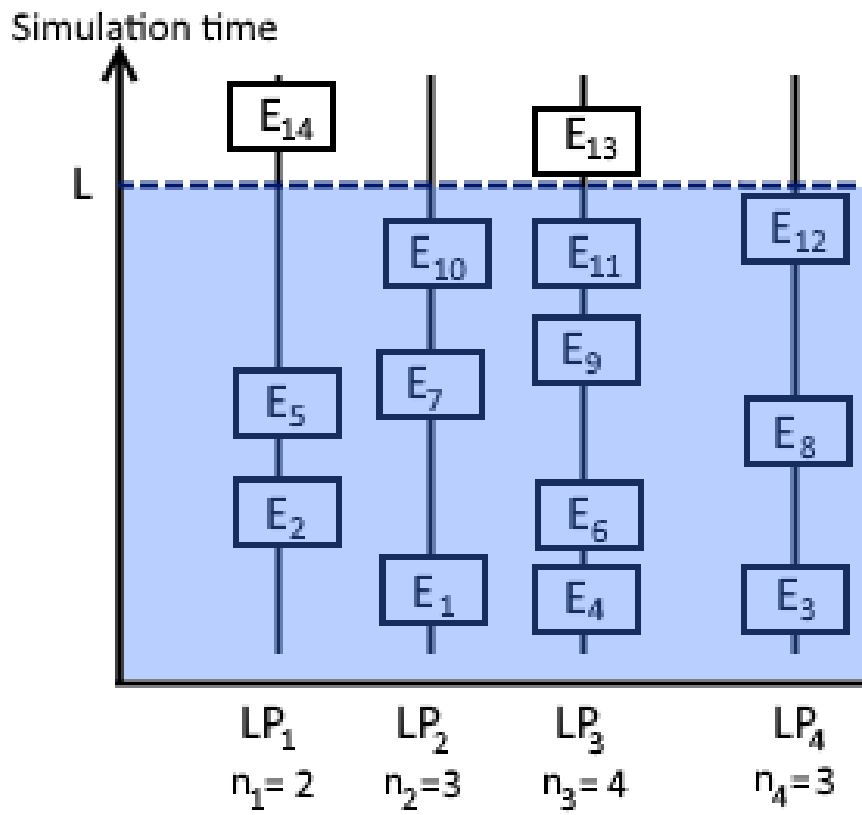


Figure 5.1: An example usage of some of the variables, a window of size L , containing $\sum_k n_k = 12$ events in total.

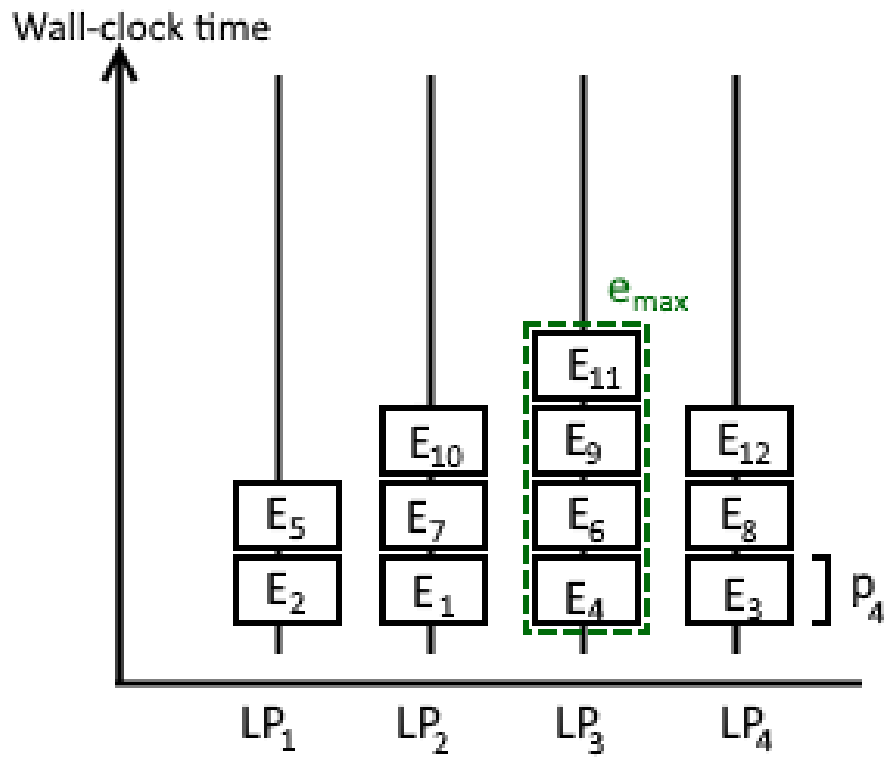


Figure 5.2: In wall-clock time all LPs can process their events in parallel, the processing time in wall-clock time in this figure is equal for all events (p_4 units of wall-clock time)

5.2 CMB

In the previous section we've defined parallelism for the YAWNS synchronization algorithm. In this section we will similarly do this for the Chandy-Misra-Bryant (CMB) algorithm. Since CMB is not a windows based algorithm, like YAWNS, we can not use the same definition of parallelism. Instead, we should consider the amount of event messages sent, and the amount of Null messages sent. These Null messages are non-simulation messages and are thus overhead introduced by the synchronization algorithm. Thus it makes sense to define a definition of parallelism that punishes the amount of Null messages. The amount of parallelism can then be defined as[46]:

$$P = \frac{e_{Event}}{e_{Event} + e_{Null}} \quad (5.5)$$

For the total amount of event messages sent e_{Event} and Null messages sent e_{Null} . Note that we can not compare the value of P for CMB with the value of P for YAWNS.

5.3 Time Warp

We have not been able to find a definition of parallelism for Optimistic algorithms. Most authors report amounts of MPI messages sent per second, or something similar (e.g. committed events per second). We argue that these kind of metrics don't capture the essence of parallelism in Time Warp and are highly influenced by the hardware used. These statistics do not account for the effect that anti-messages, and consequently, rollbacks have. We argue that a rollback can be considered as 'overhead' introduced by the TimeWarp algorithm. An increase in the amount of rollbacks, thus increases the amount of non-simulation work that needs to be done. Any definition of parallelism for Optimistic algorithms, thus has to 'punish' the amount, and/or the 'severity' of the rollback (i.e. the amount of work required to recover from an out-of-order event).

We choose to distinguish two types of rollbacks i.e.

1. Idle rollbacks, i.e. When an LP was done processing events (and thus was idling), and has to roll back from simulation time ∞ .
2. Busy rollbacks, i.e. When an LP was still processing events, and has to roll back.

A high amount of Idle rollbacks indicates that many LPs are idling often, whereas relatively little other LPs are processing more events. This thus indicates a non-uniform workload. Conversely, a high amount of busy rollbacks indicates a more evenly distribute workload.

Thus we introduce our definition of parallelism as:

$$P = \frac{Busy}{Idle} \tag{5.6}$$

For the total amount of *Busy* and *Idle* rollbacks. Note that, again, we can not compare the value of P for Time Warp to the values of P for YAWNS and CMB as these are defined differently.

Chapter 6

A parallelism approximation algorithm for EPHOLD using YAWNS

In this chapter we will develop an algorithm to estimate the amount of parallelism for a given PDES graph and weight configuration for the EPHOLD algorithm (Chapter 4) for the YAWNS synchronization algorithm (Section 2.1.3). To this end we will analyze the steady-state amount of events scheduled to be executed per window, per LP.

In the work of Dickens [18], a YAWNS window is also referred to as a *generation* of messages across a known period of simulation time. A generation i constitutes a time-window $[t_i, t_i + L)$ (For some lookahead value L , and simulation time t). All events with timestamp $e_t \in [t_i, t_i + L)$ are said to belong to generation i . Any events scheduled in higher generations due to the execution of messages in generation i are not a part of generation i . For clarity, we will use the term 'window' for the remainder of this chapter, instead of 'generation' as they are (almost) identical. A definition of parallelism in YAWNS has already been given in Section 5.1. In Section 6.1 we introduce several variables that we will be using throughout the rest of the algorithm. After analyzing the way newly generated events move between LPs (Section 6.3), we will use several aspects of probability theory (Section 6.2) to estimate in what future window, newly generated events will be scheduled (Section 6.5 and 6.4). These computations are performed iteratively until a steady state of the amount of events scheduled per window, per LP, is reached. In section 6.6 we construct the final algorithm, and evaluate several examples (Section 6.7). This is an iterative algorithm. We argue that the EPHOLD benchmark will always result in a steady state with regard to the amount of events scheduled per YAWNS window. This is because the timestamp increment

function does not change over time, and will thus cause a constant portion of events to be scheduled in consecutive windows on average. The Movement function also stays the same over the duration of the benchmark, and thus will schedule a constant portion of its events on separate neighboring LPs. We do not have to account of queue sizes, or waiting times, as the EPHOLD benchmark does not use these elements. Finally, the total amount of events in the EPHOLD benchmark remains the same throughout the benchmark. These four arguments do not form a formal proof, but we argue that they are strong enough to assume that the average amount of events scheduled per window, will eventually reach a steady state, in (E)PHOLD.

6.1 The variables

Let A be the weighted adjacency matrix of PDES graph $G(V, E)$. I.e. we have: $a_{ij} > 0 \iff (v_i, v_j) \in E$, and the sum of every row equals 1. (i.e. $\forall i : \sum_j a_{ij} = 1$).

Our next variable is our 'simulation state' C . As mentioned earlier, the algorithm iteratively simulates the execution and scheduling of events per window. We define C as a $M_c \times |V| + 1$ matrix. Here M_c is a user defined constant that serves as an upper bound to the maximum amount of windows our algorithm will process at. Element $C_{i,k}$ represents the amount of events (that were) scheduled at LP_k at window i . $C_{i, (|V|+1)}$ on the other hand equals i if window i has been processed by the algorithm, and equals 0 otherwise. Furthermore, let \vec{R} be a vector of Probability Density Functions (PDFs). Where the element at index k , (r_k) is the PDF of the probability distribution used to compute the timestamp increment at LP_k . E.g. in the standard PHOLD algorithm, all elements of \vec{R} hold the PDF of the Exponential distribution ($Exp(\lambda)$). The elements of \vec{R} are initialized such that all constants of the PDFs are known, and there is only one variable left free (typically denoted as x). Thus every element r_k of \vec{R} is a function $r_k(x)$, taking a value $0 \leq x \leq 1$ and returning the probability density at x .

6.2 Some probability theory

By the properties of a PDF, we have

$$\int_0^{\infty} r_k(x) dx = 1 \tag{6.1}$$

For each PDF $r_k \in \vec{R}$. The probability that a variable X_k , drawn from a distribution with PDF r_k , lies between values a and b is given by. For all elements r_k of \vec{R} .

$$\Pr(a \leq X_k \leq b) = \int_a^b r_k(x) dx \quad (6.2)$$

The final probability expression we introduce is the Expected value $E[X_k]$ (also referred to as the probability's mean).

$$E[X_k] = \int_0^\infty x r_k(x) dx \quad (6.3)$$

These expressions will come into play later in this chapter.

6.3 Movement of events

Every event schedules one new event on a neighboring LP. Suppose we have a complete graph of 4 nodes as PDES graph and suppose 10 events are scheduled in window i at LP_1, LP_2, LP_3 and LP_4 . I.e. 40 events in total, scheduled to be executed in window i . Let all weights be $\frac{1}{3}$ (i.e. standard PHOLD), then we can estimate the amount of events each LP will receive. E.g. On average LP_4 will receive $10 \cdot \frac{1}{3}$ events from each of its neighboring LPs, totaling $3 \cdot \frac{10}{3} = 10$ on average (for 3 neighbors). In the general case, let $C_{i,k}$ be the amount of events scheduled to be executed at LP_k for window i . The amount of events, produced by window i , that LP_k will receive from its incoming neighbors j , equals

$$I_{i,k} = \sum_{j=1}^n C_{i,j} \cdot a_{j,k}$$

With $I_{i,k}$, the amount of events produced by window i , that arrive at LP_k . Note, for now, we disregard in which subsequent window these events are scheduled, we are only concerned with the amount of events that LP_k receives from its incoming neighbors as a result of executing the events in window i

6.4 Defining $S_{q,j}$

Let $S_{q,j}$ indicate the probability that LP_j will schedule an event in the q -th subsequent window. E.g. Suppose LP_1 executes an event during the current window. Then the probability $S_{0,1}$ expresses the probability that the newly generated event will be scheduled in the next window, $S_{1,1}$ the probability of

scheduling it in the following window, etc.

There are two important observations:

1. A window's lower bound is defined by the unprocessed event with smallest timestamp.
2. The distribution of event timestamps in the current window influences the way that newly generated events will be distributed across subsequent windows.

First we will introduce some variables. Let t_c be the timestamp of the lower bound of the current window, let t be the offset between an event's scheduled timestamp and t_c . I.e. the timestamp of an event e in the current window is of the form $t_c + t$ for some defined t . As dictated by (E)PHOLD, let L be a fixed lookahead. Next we define t_{min0} as the smallest timestamp of any events scheduled outside of the current window. Thus, we have $t_{min0} \geq t_c + L$, or $t_{min0} = t_c + L + \Delta_0$ for some unknown value $\Delta_0 \geq 0$. Note, that no event is scheduled in We know that the size of any window is going to be exactly L . The relation between these variables are shown pictorially in figure 6.1.

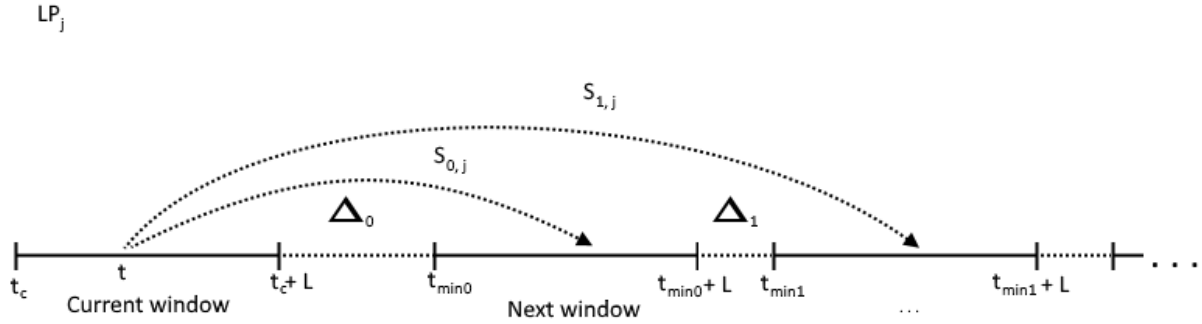


Figure 6.1: A pictorial representation of the idea behind $S_{q,j}$ for a given LP_j that is about to execute an event at timestamp $t_c + t$, note the window bounds. The 'current' window has bounds t_c and $t_c + L$, the second t_{min0} and $t_{min0} + L$, etc. Note that the width of Δ_0 , and Δ_1 are exaggerated.

The bounds of the windows, are not fixed in advance. We thus have to somehow predict at what timestamp the next window will start. All we know is that the timestamp of the upper bound of the window will be L higher than the timestamp of the lower bound of the window. I.e. the current window has bounds t_c and $t_c + L$, the next has bounds t_{min0} and $t_{min0} + L$ etc.

Also note that the timestamp intervals $[t_c+L, t_c+L+\Delta_0)$, $[t_{mink}+L, t_{min(k+1)}+\Delta_{k+1}) \forall k = 0, 1, 2, \dots$, contain no events. To illustrate this, suppose that all events in the current window are executed and all schedule new events outside the current window (this has to be the case by definition of the window). Then, the next window starts at the event with smallest timestamp (i.e. t_{min0}) outside of the current window. Here we have $t_c + L \leq t_{min0}$, and $\Delta_0 = t_{min0} - (t_c + L) \geq 0$. Thus the 'space' occupied by Δ_0 contains no events to be executed.

We can not give strict bounds on the sizes of Δ_q , as they will rely on the definition of the timestamp increment function(s). We do know however that there will never be any events scheduled in the portions occupied by $\Delta_k \forall k = 0, 1, 2, \dots$ in the average case, because it lies outside of any window. Suppose event e , scheduled at timestamp $t_c \leq t_c + t \leq t_c + L$ is executed, and generates a new event to be scheduled at timestamp t_{New} . We now attempt to derive the probability that t_{New} will be scheduled as such that it will be executed during the next window (i.e. $q = 0$), or subsequent windows (i.e. $q = 1, 2, 3, \dots$). First we will derive the probability that any event scheduled in the current window will schedule its new window falls into the next window (i.e. $q = 0$). When this event is executed at LP_j , we know that t_{New} is of the following form: $t_{New} = t_c + t + L + X_j$, for random variable X_j drawn from r_j (i.e. $X_j \sim r_j$). For brevity, we will write X instead of X_j , as this analysis is LP independent. Since we already know that the portion indicated by Δ_0 in figure 6.1 is void of any scheduled events, we can equally write: $\Pr(t_c + L + \Delta_0 \leq t_{New} \leq t_c + L + \Delta_0 + L) = \Pr(t_c + L \leq t_{New} \leq t_c + L + L)$. In fact, the probability that event e will schedule a new event in the portions occupied by Δ_k will always be 0. Thus, for the purpose of determining a probability, we can redraw Figure 6.1 as Figure 6.2.

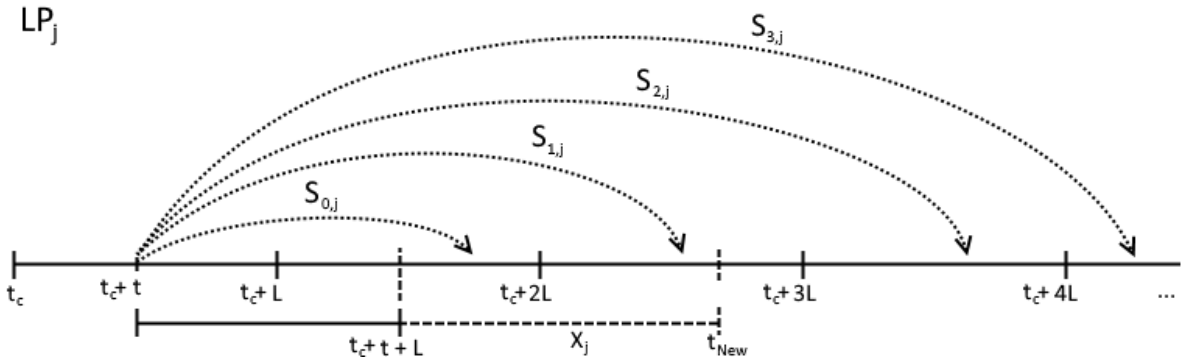


Figure 6.2

For completeness we will include the definitions of Δ_k in the derivations below. Finally, we note that X will always produce positive values i.e. $0 \leq X < \infty$, thus $\Pr(-a \leq X \leq b) = \Pr(0 \leq X \leq b)$ for any positive values a and b .

For $q = 0$, we derive:

$$\begin{aligned}
\Pr(t_{min0} \leq t_{New} \leq t_{min0} + L) &= \Pr(t_c + L + \cancel{\Delta_0} \leq t_{New} \leq t_c + L + \cancel{\Delta_0} + L) \\
&= \Pr(\cancel{t_c} + \cancel{L} \leq \cancel{t_c} + t + \cancel{L} + X \leq \cancel{t_c} + \cancel{L} + L) \\
&= \Pr(0 \leq t + X \leq L) \\
&= \Pr(-t \leq X \leq L - t) \\
&= \Pr(0 \leq X \leq L - t)
\end{aligned} \tag{6.4}$$

Note that we can only perform the first step in derivation 6.4 in *this* context by the definition of Δ_0 , we can *not* perform such simplifications in general probability theory. Our earlier observation for Δ_0 , holds equally for Δ_1 . We also observe that we can define $t_{min1} = t_{min0} + L + \Delta_1$. The probability that t_{New} will fall in the following window (i.e. $q = 1$) is then derived as follows:

$$\begin{aligned}
\Pr(t_{min1} \leq t_{New} \leq t_{min1} + L) &= \Pr(t_{min0} + L + \cancel{\Delta_1} \leq t_{New} \leq t_{min0} + 2 \cdot L + \cancel{\Delta_1}) \\
&= \Pr(t_{min0} + L \leq t_{New} \leq t_{min0} + 2 \cdot L) \\
&= \Pr(t_c + L + \cancel{\Delta_0} + L \leq t_{New} \leq t_c + L + \cancel{\Delta_0} + 2 \cdot L) \\
&= \Pr(\cancel{t_c} + \cancel{L} + L \leq \cancel{t_c} + t + \cancel{L} + X \leq \cancel{t_c} + \cancel{L} + 2 \cdot L) \\
&= \Pr(L \leq t + X \leq 2 \cdot L) \\
&= \Pr(L - t \leq X \leq 2 \cdot L - t)
\end{aligned}$$

In fact, we can define $\forall q = 0, 1, 2, \dots : t_{minq} = t_{min(q-1)} + L + \Delta_q$ with $t_{min(-1)} = t_c$. This recurrence solves to:

$$t_{minq} = t_c + (q + 1) \cdot L + \sum_{k=0}^q \Delta_k \tag{6.5}$$

With these observations we can derive the probability of whether t_{New} falls

in the q -th subsequent window as follows:

$$\begin{aligned}
& \Pr(t_{minq} \leq t_{New} \leq t_{minq} + L) \\
&= \Pr(t_c + (q+1) \cdot L + \sum_{k=0}^q \Delta_k \leq t_{New} \leq t_c + (q+1) \cdot L + (\sum_{k=0}^q \Delta_k) + L) \\
&= \Pr(t_c + (q+1) \cdot L \leq t_{New} \leq t_c + (q+2) \cdot L) \\
&= \Pr(t_c + q \cdot L + \mathcal{L} \leq t_c + t + \mathcal{L} + X \leq t_c + (q+1) \cdot L + \mathcal{L}) \\
&= \Pr(q \cdot L \leq t + X \leq (q+1) \cdot L) \\
&= \Pr(q \cdot L - t \leq X \leq (q+1) \cdot L - t)
\end{aligned}$$

Note, that we still need to estimate the value of t , we will do this later. For now it is sufficient to bound t . We know that $0 \leq t \leq L$. With this assumption on t , we will show that $S_{q,j}$ forms a valid probability distribution. For this, we need to show that all possible values sum to 1, and all values are in the interval $[0, 1]$ for all values of q . Since all we've done is impose bounds on the value of t_{New} for all values of q , we must show that the fractions over all intervals sums to 1. Thus we need to show:

$$\forall j : \left(\sum_{q=0}^{\infty} S_{q,j} = 1 \right) \quad (6.6)$$

Observing that for all the mentioned intervals earlier, we have:

$$\begin{aligned}
\bigcup_{q=0}^{\infty} [\max[0, q \cdot L - t], (q+1)L - t) &= [0, L - t) \quad \bigcup \quad \left(\bigcup_{q=1}^{\infty} [q \cdot L - t, q \cdot L + L - t) \right) \\
&= [0, \infty)
\end{aligned} \quad (6.7)$$

Which shows that all intervals are continuous. The intersection for all mentioned intervals is the empty set i.e.

$$\begin{aligned}
\bigcap_{q=0}^{\infty} [\max[0, q \cdot L - t], (q+1)L - t) &= [0, L - t) \quad \bigcap \quad \left(\bigcap_{q=1}^{\infty} [q \cdot L - t, q \cdot L + L - t) \right) \\
&= \emptyset
\end{aligned} \quad (6.8)$$

showing that the intervals don't overlap for $0 \leq t \leq L$. For $S_{q,j}$ we have:

$$S_{q,j} = \int_{\max[0, q \cdot L - t]}^{(q+1) \cdot L - t} r_j(x) dx \quad (6.9)$$

We can rewrite equation 6.6, using equations 6.7, 6.8 and 6.9, resulting in that we need to show that the following proposition holds:

$$\forall j : \left(\int_0^\infty r_j(x) dx = 1 \right) \quad (6.10)$$

By the properties of PDF r_j , this is the case, and $S_{q,j}$ is a valid probability distribution over q for any given LP_j .

What remains is to estimate the value of t . Any PDF that has a converging mean $E[X]$ will attain this mean when we consider the stabilized 'average case'. I.e. The mean of the observed timestamp increment values will converge to $E[X]$. Thus, in the average case we can assume, that event timestamps are evenly spread apart in any given window (with spacing $E[X]$). I.e. in the average case, we can assume that event timestamps are Uniformly distributed between bounds 0 and L (denoted $U(0, L)$) within a window. Thus, to approximate t , we can set $t = E[U(0, L)] = \frac{L}{2}$.

Note that we are considering the *average* case here, where, in the long run, the average of the observed values will attain these possibly after a relatively long time.

Finally, tying everything together, we derived:

$$S_{q,j} = \int_{\max[0, q \cdot L - \frac{L}{2}]}^{(q+1) \cdot L - \frac{L}{2}} r_j(x) dx \quad (6.11)$$

6.5 Scheduling of events

An important part of the YAWNS algorithm is the concept of a time window. To accurately approximate the execution of this algorithm, we thus must approximate the amount of events that are to be scheduled and executed per window. Suppose the algorithm has just executed all events in window i , then LP_k receives $a_{j,k} \cdot C_{i,j}$ events, on average, from incoming neighbor j . We also know that the timestamp increments of these events are drawn from r_j .

Let $S_{q,j}$ (Section 6.4) express the probability that an event, originating from incoming neighbor LP_j , will be scheduled in the $q - th$ next window.

Thus we expect that $S_{0,j} \cdot a_{j,k} \cdot C_{i,j}$ events will be scheduled in the next window (i.e. $i + 1$), $S_{1,j} \cdot a_{j,k} \cdot C_{i,j}$, in window $i + 2$, etc.

In the general case, we can thus state:

$$C_{i+q+1,k} = \sum_{j=1}^n S_{q,j} \cdot a_{j,k} \cdot C_{i,j} \quad (6.12)$$

Note, that by the properties of probability distributions, we have that $\forall j : \sum_q S_{q,j} = 1$, and thus, we are not losing, or introducing events in our calculations above.

6.6 The algorithm

In this Section, we will utilize all concepts introduced in the previous Sections of this chapter, to construct the final approximation algorithm. The algorithm has two main parts of interest, i.e. the initialization (Section 6.6.1), the main loop and its guards (Section 6.6.2) We will present pseudo code for the algorithm, and derive its worst-case run time complexity (Section 6.6.3) before mentioning some design decisions (Section 6.6.4).

6.6.1 Initialization

Now that we have all our building blocks in place, we will construct the final algorithm. We initialize state matrix C such that all elements are 0, except the first row. The $|V|$ elements in the first row of C hold the amount of events initially scheduled at their respective LPs (i.e. $C_{1,k} = ie_k$ for initial amount of events ie_k for LP_k). \vec{R} is initialized with PDF functions, and A is initialized as the Graph's weighted adjacency matrix. Once these initializations have taken place we start the main loop of the algorithm (Section 6.6.2).

6.6.2 Iteration

The main loop simulates the execution and scheduling of events per window, per LP. This is simulated by the following expression:

$$C_{i+q+1,k} \leftarrow C_{i+q+1,k} + \sum_{j=1}^{|V|} a_{j,k} S_{q,k} C_{i,j}$$

For Current window i , and LP_k . This computation is done for increasing values of q up to a certain constant M_q . After these computations, we also include the remainder:

$$C_{i+M_q+1,k} \leftarrow C_{i+M_q+1,k} + \sum_{j=1}^n a_{j,k} C_{i,j} \left(1 - \sum_{q=0}^{M_q} S_{q,k}\right)$$

Such that we don't lose any events. We repeat this for all LPs k and finally mark window i as completed, in C :

$$C_{i,|V|+1} \leftarrow i$$

Now we increment i and repeat until $\forall k : C_{i,k} = C_{i-1,k}$. I.e. The amount of events scheduled in window i and $i - 1$ are equal for all LPs, and thus the solution is stable. In pseudo code the algorithm can be expressed as follows: First, we define the convenience function **IsStableGeneration**. Its task is to determine if the algorithm has reached a stable state. I.e. that two consecutive iterations resulted in the same state.

Algorithm IsStableGeneration

Input: State matrix C and current window index i

Output: A boolean indication if window i is stable

1: **return** $\bigwedge_{k=1}^{|V|} (C_{i,k} = C_{i-1,k})$

Note that, especially when dealing with floating point numbers, we can define equality here as 'close enough'. In our implementation, two numbers a and b are considered equal iff $|a - b| \leq \frac{1}{100000}$. More implementation details are discussed in Section 7.1. The 'work' of the algorithm is done in the **CalcGenerationForLP** function. It computes the amount of events a specified LP will receive, in which window they are scheduled and updates the state matrix accordingly.

Algorithm CalcGenerationForLP

Input: State matrix C , current window index i and LP index k .

Output: All events to be scheduled at LP_k are updated in C .

1: $p \leftarrow i + 1$
2: **for all** $q \in \{0, 1, \dots, M_q - 1\}$ **do**
3: $C_{p+q,k} \leftarrow C_{p+q,k} + \sum_{j=1}^{|V|} a_{j,k} \cdot S_{q,j} \cdot C_{i,j}$
4: $C_{p+M_q,k} \leftarrow C_{p+M_q,k} + \sum_{j=1}^{|V|} a_{j,k} \cdot C_{i,j} \cdot (1 - \sum_{q=0}^{M_q} S_{q,j})$
5: **return** C

The main algorithm then becomes:

Algorithm ApproximationAlgorithm

Input: Adjacency matrix A , PDF vector \vec{R} , list ℓ that hold the initial amount of events scheduled per LP.

Output: A list of steady state amount of events per LP.

- 1: Initialize C as an $M_c \times (|V| + 1)$ matrix of 0s, where row 1 is initialized with the values of ℓ
 - 2: $g \leftarrow 1$
 - 3: **while** $g \leq M_c \wedge (g \leq 1 \vee \neg \text{IsStableGeneration}(C, g))$ **do**
 - 4: **for all** LP_k **do**
 - 5: $C \leftarrow \text{CalcGenerationForLP}(C, g, k)$ // Let all LPs execute their events.
 - 6: $C_{g, |V|+1} \leftarrow g$ // Mark this window as 'executed'
 - 7: $g \leftarrow g + 1$
 - 8: **return** The first $|V|$ elements of C of row g
-

Note: for all algorithms, we assume that M_c and M_q are known constants. Secondly, we note that the value of $|V|$ can be deduced from the adjacency matrix A , and is thus not a required parameter.

6.6.3 Runtime Analysis

The runtime analysis of the algorithm derived early is relatively straightforward. We will first analyze the **IsStableGeneration** function.

Comparing two element of a matrix takes $O(1)$ time, assuming the matrix is represented as an array. The comparison is executed at most n times, with $n = |V|$. Thus the entire function takes $O(n) \cdot O(1) = O(n)$ time.

Next, we analyze the **CalcGenerationForLP** function. The assignment statement takes a constant amount of time $O(1)$. The for loop is executed M_q times, the body of which, can be bounded above by $O(n)$. The running time of the loop thus becomes $O(M_q \cdot n)$. Step 4 also takes take $O(M_q \cdot n)$ time and all other steps take $O(1)$. Finally, the entire function thus takes: $O(1) + O(M_q \cdot n) + O(M_q \cdot n) + O(1)$, with M_q being a constant, this results in a running time of $O(n)$

Finally, we analyze the **ApproximationAlgorithm** algorithm. The initialization step takes $O(M_c \cdot (n + 1)) + O(1)$ time, for C and g . The while loop is iterated at most a constant M_c times, each time, the body of the for loop is executed exactly n times. The call to the **CalcGenerationForLP** function is the only operation performed in its body, which takes $O(n)$

time, as analyzed earlier. The increment of g takes $O(1)$ time, as does the assignment of $C_{g,|V|+1}$. The total cost of the while loop thus becomes $O(M_c \cdot (n \cdot n + 1)) = O(M_c \cdot n^2 + M_c)$.

Thus, the running time of the entire algorithm: $O(M_c \cdot (n + 1)) + O(1) + O(M_c \cdot n^2) = O(M_c \cdot n) + O(M_c \cdot n^2 + M_c) = O(n^2)$ for constant M_c .

The algorithm will always terminate, due to the constraint M_c on the amount of windows to process. Whether the algorithm terminates with a stable solution is found is less obvious. Similar to EP HOLD, the timestamp increment, and movement functions do not change during the algorithm, and no events are introduced, or lost during the execution of the algorithm. This is not a formal proof, but we believe it is a strong argument to assume that the algorithm will terminate with a stable solution, given a high enough value of M_c .

6.6.4 Design considerations

By design, the algorithm does not calculate the amount of parallelism directly. This can easily be done using its output. As we saw, it outputs the steady state amount of events scheduled to be executed per window, per LP. If we have an estimation for all values of p_k (usually this is assumed to equal 1, for all LPs), we can compute e_{max} (Eq. 5.2), W (Eq. 5.3), and consequently the amount of parallelism P (Eq. 5.4). We explicitly designed the algorithm not to directly compute P , because the algorithm output offers more insight into the bottlenecks in the PDES graph, as we will see when we work out some numerical examples in Section 6.7.

6.7 Numerical Examples

In this section we will show two walk-through examples for given PDES configurations and weights. We will show that we can approximate a normal PHOLD execution (Section 6.7.1), and what happens if we change its weights (Section 6.7.2). We will compare these examples with actual experiments in Section 8.1.1

6.7.1 PHOLD execution

As an example, we will analyze the EP HOLD algorithm that simulates the PHOLD algorithm for the PDES graph depicted in figure 6.3. Note that we will analyze the exact same example presented earlier in Section 4.3.2.

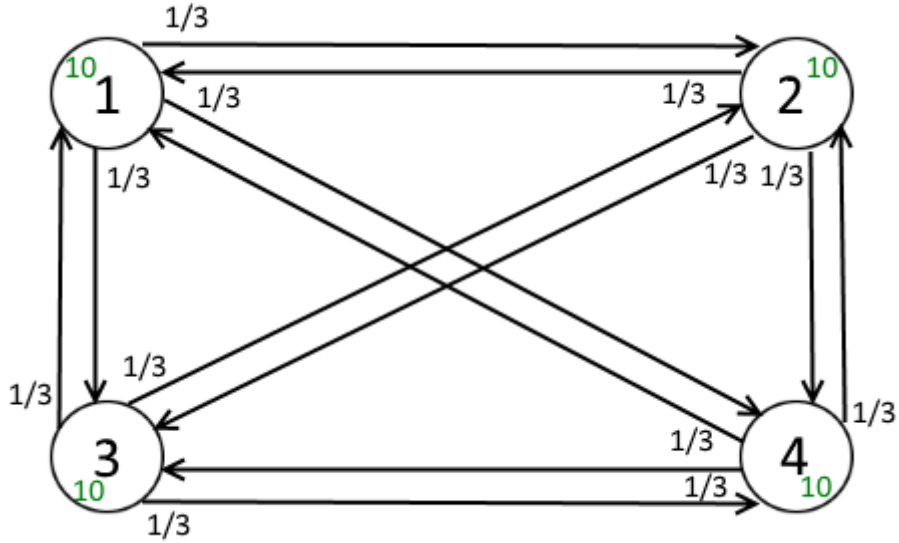


Figure 6.3: The PDES graph for our example, the probability of sending an event over a particular edge is drawn at the *tail* of the edge

I.e. Our PDES graph is a complete graph of 4 nodes, where for all LPs, each LP has an equal probability of receiving an event. The weighted adjacency matrix is:

$$A = \begin{matrix} & v_1 & v_2 & v_3 & v_4 \\ \begin{matrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{matrix} & \begin{pmatrix} 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 \end{pmatrix} \end{matrix}$$

The PHOLD algorithm schedules new events with timestamp increments drawn from an exponential ($Exp(1)$) distribution, Thus we have:

$$\vec{R} = \begin{pmatrix} e^{-x} \\ e^{-x} \\ e^{-x} \\ e^{-x} \end{pmatrix}$$

For this example, we chose constants: $M_q = 3$ and $M_c = 100$. I.e. We limit the amount of windows that can be scheduled ahead M_q to 3, and the maximum amount of windows to compute to 100 (or if a stable solution is found earlier). Now that we have all parameters, we can initialize state

matrix C :

$$C = \begin{pmatrix} 10 & 10 & 10 & 10 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

We initialize $g \leftarrow 1$ and start the while loop. Because we have 4 LPs, the for loop's body is executed 4 times, updating C one LP per iteration. The main part of the algorithm is the **CalcGenerationFOLL** function. We will provide explicit calculations for LP_1 , (I.e. $k = 1$) We have $p \leftarrow 1 + 1 \implies p = 2$, thus, in the first iteration of the for loop we have:

$$\begin{aligned} C_{2,1} &\leftarrow C_{2,1} + a_{1,1} \cdot S_{0,1} \cdot C_{1,1} \\ &\quad + a_{2,1} \cdot S_{0,2} \cdot C_{1,2} \\ &\quad + a_{3,1} \cdot S_{0,3} \cdot C_{1,3} \\ &\quad + a_{4,1} \cdot S_{0,4} \cdot C_{1,4} \end{aligned}$$

Resulting in:

$$\begin{aligned} C_{2,1} &\leftarrow 0 + 0 \cdot 0.393 \cdot 10 \\ &\quad + \frac{1}{3} \cdot 0.393 \cdot 10 \\ &\quad + \frac{1}{3} \cdot 0.393 \cdot 10 \\ &\quad + \frac{1}{3} \cdot 0.393 \cdot 10 \end{aligned}$$

Similarly, for $q = 1$ and $q = 2$

$$\begin{aligned} C_{3,1} &\leftarrow C_{3,1} + a_{1,1} \cdot S_{1,1} \cdot C_{1,1} \\ &\quad + a_{2,1} \cdot S_{1,2} \cdot C_{1,2} \\ &\quad + a_{3,1} \cdot S_{1,3} \cdot C_{1,3} \\ &\quad + a_{4,1} \cdot S_{1,4} \cdot C_{1,4} \\ C_{4,1} &\leftarrow C_{4,1} + a_{1,1} \cdot S_{2,1} \cdot C_{1,1} \\ &\quad + a_{2,1} \cdot S_{2,2} \cdot C_{1,2} \\ &\quad + a_{3,1} \cdot S_{2,3} \cdot C_{1,3} \\ &\quad + a_{4,1} \cdot S_{2,4} \cdot C_{1,4} \end{aligned}$$

Equating:

$$\begin{aligned}C_{3,1} &\leftarrow 0 + 0 \cdot 0.383 \cdot 10 \\ &\quad + \frac{1}{3} \cdot 0.383 \cdot 10 \\ &\quad + \frac{1}{3} \cdot 0.383 \cdot 10 \\ &\quad + \frac{1}{3} \cdot 0.383 \cdot 10 \\ C_{4,1} &\leftarrow 0 + 0 \cdot 0.141 \cdot 10 \\ &\quad + \frac{1}{3} \cdot 0.141 \cdot 10 \\ &\quad + \frac{1}{3} \cdot 0.141 \cdot 10 \\ &\quad + \frac{1}{3} \cdot 0.141 \cdot 10\end{aligned}$$

After the end of the for loop, with: $1 - (0.393 + 0.383 + 0.141) = 0.082$ we compute:

$$\begin{aligned}C_{5,1} &\leftarrow 0 + 0 \cdot 0.082 \cdot 10 \\ &\quad + \frac{1}{3} \cdot 0.082 \cdot 10 \\ &\quad + \frac{1}{3} \cdot 0.082 \cdot 10 \\ &\quad + \frac{1}{3} \cdot 0.082 \cdot 10\end{aligned}$$

The process is similar for the other LPs, in fact, in this case, all numerical values are equal. After this, we mark the window as executed by setting: $C_{1,5} \leftarrow 1$. After processing one window in the algorithm, the state matrix

looks as follows:

$$\begin{pmatrix} 10 & 10 & 10 & 10 & 1. \\ 3.935 & 3.935 & 3.935 & 3.935 & 0 \\ 3.834 & 3.834 & 3.834 & 3.834 & 0 \\ 1.410 & 1.410 & 1.410 & 1.410 & 0 \\ 0.821 & 0.821 & 0.821 & 0.821 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 \end{pmatrix}$$

One more iteration results in:

$$\begin{pmatrix} 10 & 10 & 10 & 10 & 1 \\ 3.935 & 3.935 & 3.935 & 3.935 & 2 \\ 5.382 & 5.382 & 5.382 & 5.382 & 0 \\ 2.919 & 2.919 & 2.919 & 2.919 & 0 \\ 1.376 & 1.376 & 1.376 & 1.376 & 0 \\ 0.323 & 0.323 & 0.323 & 0.323 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 \end{pmatrix}$$

Until finally, after 25 iterations, we reach a stable solution:

$$\begin{pmatrix} 10 & 10 & 10 & 10 & 1 \\ 3.935 & 3.935 & 3.935 & 3.935 & 2 \\ 5.382 & 5.382 & 5.382 & 5.382 & 3 \\ 5.037 & 5.037 & 5.037 & 5.037 & 4 \\ 5.421 & 5.421 & 5.421 & 5.421 & 5 \\ & & \dots & & \\ 5.231 & 5.231 & 5.231 & 5.231 & 24 \\ 5.231 & 5.231 & 5.231 & 5.231 & 25 \\ 5.231 & 5.231 & 5.231 & 5.231 & 0 \\ 3.173 & 3.173 & 3.173 & 3.173 & 0 \\ 1.167 & 1.167 & 1.167 & 1.167 & 0 \\ 0.429 & 0.429 & 0.429 & 0.429 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ & & \dots & & \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 6.4 shows the progression of the solution in a graph.

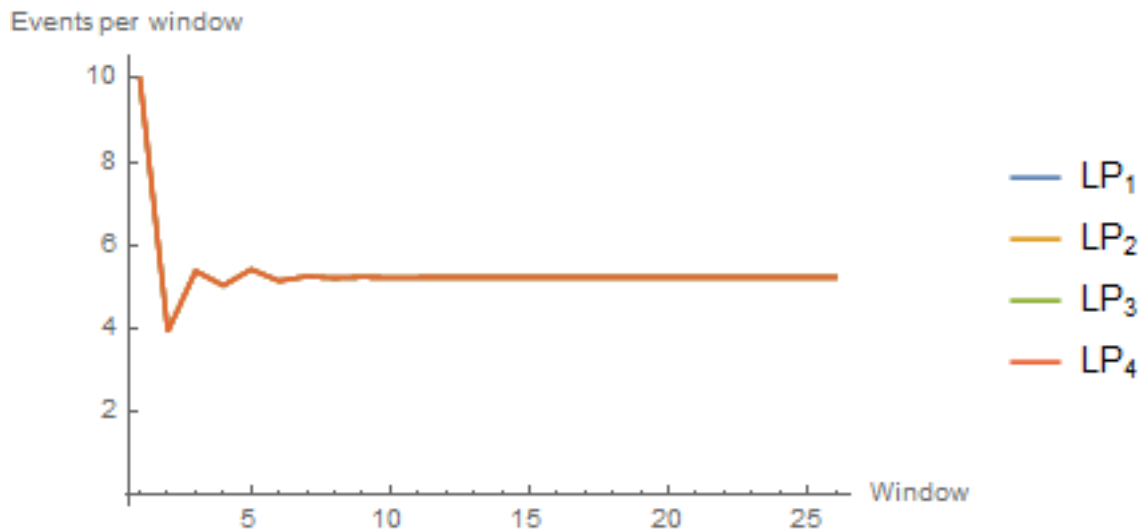


Figure 6.4: Solution progression of the analysis of PHOLD for a complete 4-node PDES graph

We can clearly see that the solution oscillates first, and then becomes

relatively stable after having processed about 8 windows. The long, seemingly 'steady' tail of the graph is explained by the 'close enough' equality enforced in the implementation of the `IsStableGeneration` function. All LPs are predicted to process an equal amount of approximately 5.231 events per window on average. This even distribution of workload is precisely one of the critiques of the PHOLD algorithm. It only studies a most optimal performance of a PDES simulation. In fact, if we redistribute the initial set of events over all LPs, e.g. as 5, 15, 7, 13, for LP_1, \dots, LP_4 , the average amount of events per window still stabilizes at approximately 5.231 as we can see in figure 6.5.

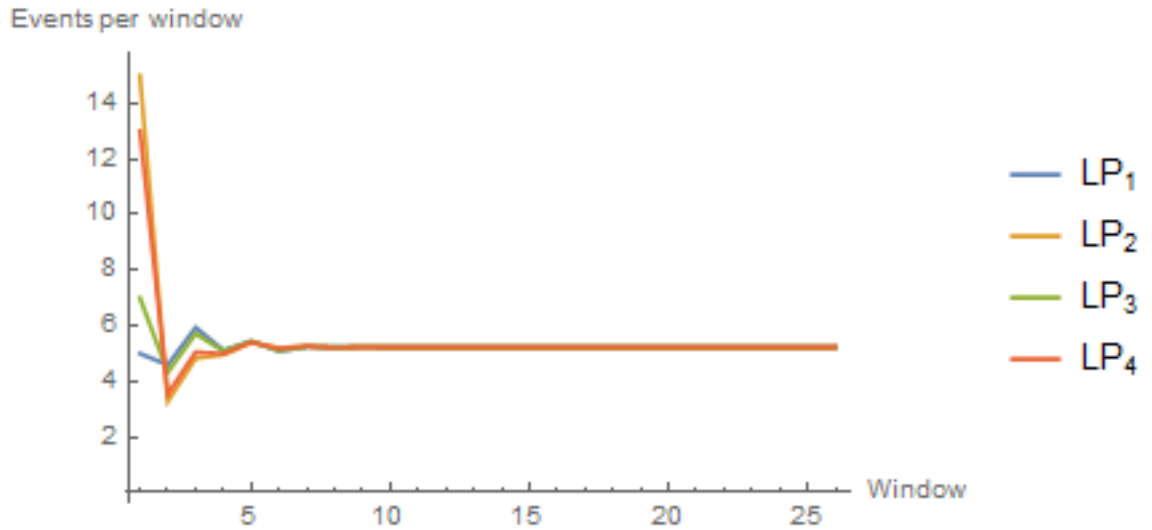


Figure 6.5: Solution progression of the analysis of PHOLD for a complete 4-node PDES graph, with 13, 15, 5 and 7 events initially scheduled at LP_1, \dots, LP_4 respectively

6.7.2 Different weights

Like in our previous example, let our PDES-graph $G(V, E)$ be a complete graph of four nodes indexed 1 through 4. We leave everything the same as in the previous example, except for the weights. Let weight w be defined as the index of the LP at the head of the associated edge, normalized s.t. all weights per LP sum to one. E.g. for LP_1 , we have outgoing neighbors $N_1 = \{LP_2, LP_3, LP_4\}$ we thus have weights: $\frac{2}{9}, \frac{3}{9}, \frac{4}{9}$ for edges $(LP_1, LP_2), (LP_1, LP_3)$ and (LP_1, LP_4) respectively. Figure 4.3 shows this pictorially.

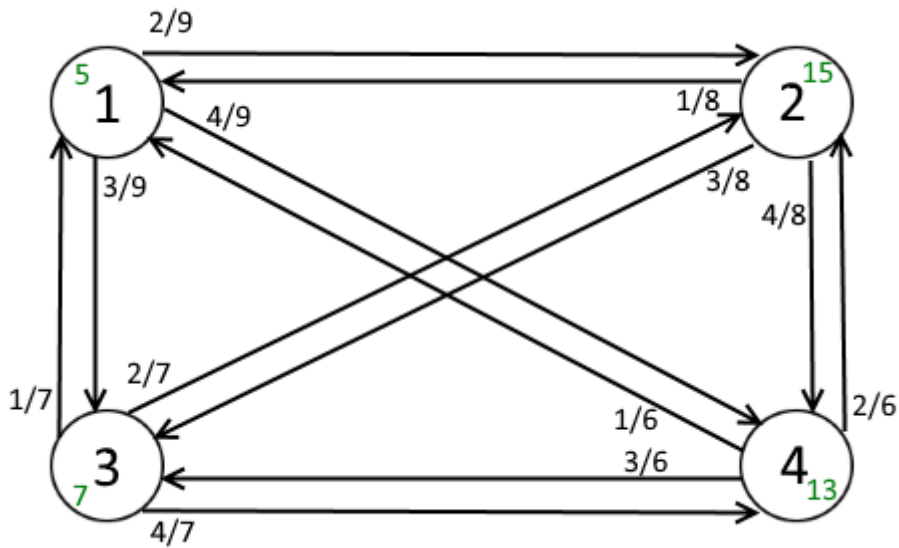


Figure 6.6: A complete 4-node PDES graph, with 5, 15, 7, 13 events initially scheduled at LP_1, \dots, LP_4 respectively, notice the weights are based on the index number of the head of their associated edges

The associated Adjacency matrix for this graph is:

$$\begin{pmatrix} 0 & \frac{2}{9} & \frac{3}{9} & \frac{4}{9} \\ \frac{1}{8} & 0 & \frac{3}{8} & \frac{4}{8} \\ \frac{1}{7} & \frac{2}{7} & 0 & \frac{4}{7} \\ \frac{1}{6} & \frac{3}{6} & \frac{3}{6} & 0 \end{pmatrix}$$

We again run the approximation two times. Once with 10 events scheduled at each LP and once where we schedule 5, 15, 7, 13 events on LP_1, \dots, LP_4 respectively (as in figure 6.6). When we run the approximation algorithm for both initial configurations we immediately see that the average amount of events per LP per window vary between LPs (Figures 6.7 and 6.8).

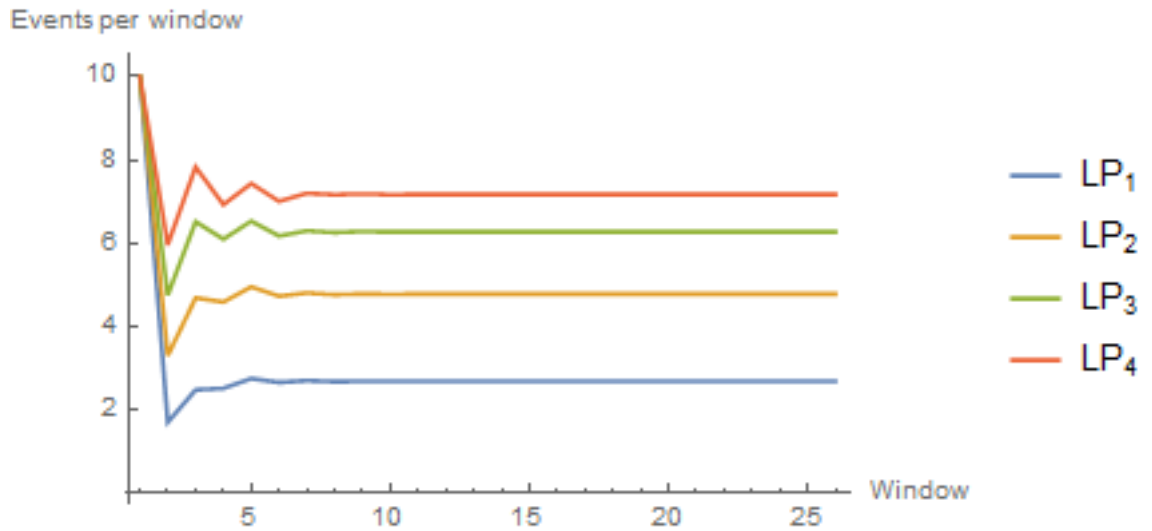


Figure 6.7: Solution progression of the approximation algorithm for our example with 10 events scheduled on each LP as initial configuration

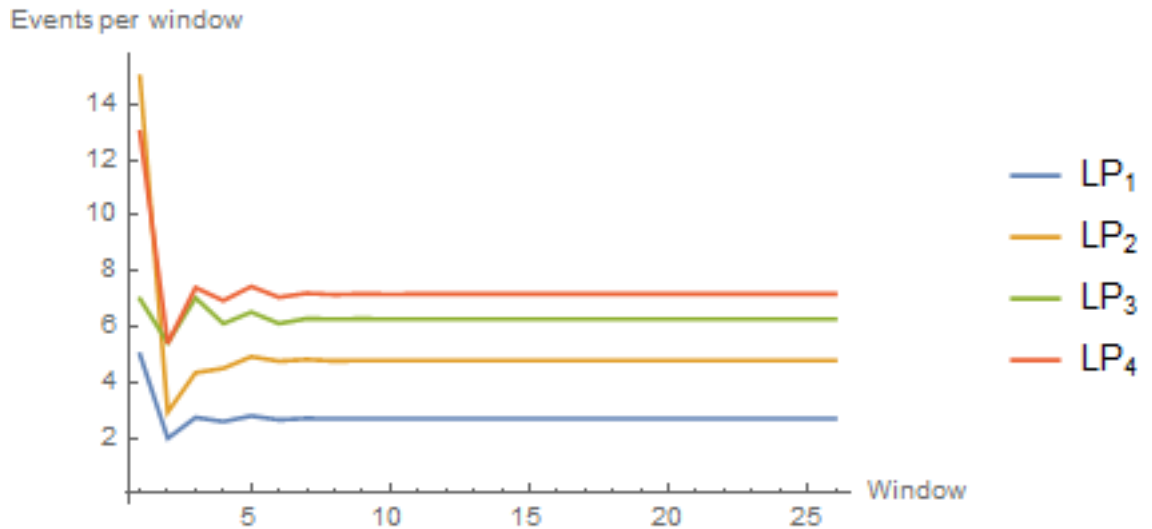


Figure 6.8: Solution progression of the approximation algorithm for our example with 5, 15, 7, 13 events scheduled on LP_1, \dots, LP_4 respectively as initial configuration

A clearly unbalanced workload distribution between LPs, the initial configuration does not seem to matter in the average case. Both initial configurations attain a stable state at window 26, where we have C for our first

initial configuration:

$$\begin{pmatrix} 10 & 10 & 10 & 10 & 1. \\ 1.70972 & 3.31014 & 4.75442 & 5.9645 & 2. \\ 2.48716 & 4.6917 & 6.51883 & 7.83105 & 3. \\ 2.52378 & 4.59273 & 6.10122 & 6.92924 & 4. \\ 2.75659 & 4.95836 & 6.53347 & 7.4362 & 5. \\ & & \dots & & \\ 2.69014 & 4.78246 & 6.27698 & 7.1737 & 24. \\ 2.69014 & 4.78246 & 6.27699 & 7.1737 & 25. \\ 2.69014 & 4.78246 & 6.27699 & 7.1737 & 0 \\ 1.63165 & 2.90071 & 3.80718 & 4.35107 & 0 \\ 0.600251 & 1.06711 & 1.40058 & 1.60067 & 0 \\ 0.22082 & 0.392569 & 0.515246 & 0.588853 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ & & \dots & & \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The state matrix for our second initial configuration arrives at the same solution:

$$\begin{pmatrix} 5 & 15 & 7 & 13 & 1. \\ 1.98374 & 2.92916 & 5.4266 & 5.39927 & 2. \\ 2.73615 & 4.34586 & 7.04234 & 7.4044 & 3. \\ 2.58888 & 4.50557 & 6.11175 & 6.94076 & 4. \\ 2.78919 & 4.9204 & 6.52482 & 7.4502 & 5. \\ & & \dots & & \\ 2.69014 & 4.78246 & 6.27698 & 7.1737 & 24. \\ 2.69014 & 4.78246 & 6.27699 & 7.1737 & 25. \\ 2.69014 & 4.78246 & 6.27699 & 7.1737 & 0 \\ 1.63165 & 2.90071 & 3.80718 & 4.35107 & 0 \\ 0.600251 & 1.06711 & 1.40058 & 1.60067 & 0 \\ 0.22082 & 0.392569 & 0.515246 & 0.588853 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ & & \dots & & \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

For both cases, the algorithm predicts that LP_4 will approximately execute about 2.667 times more events per window than LP_1 . We will experimentally test one of these configurations in Section 8.1.1

Chapter 7

Implementation

To answer our research questions, we've implemented two software tools. We will discuss some of the high-level aspects of our Mathematica implementation of the Approximation algorithm from Chapter 6 (Section 7.1). The 'experimentation framework' (Section 7.2), implemented in C++, is capable of executing EPHOLD using several PDES synchronization algorithms. The approximation algorithm outputs a prediction of the amount of parallelism present in a given PDES configuration (PDES graph, PDFs etc.) for EPHOLD when the YAWNS synchronization algorithm is used.

7.1 Approximation algorithm

The approximation algorithm presented in Chapter 6 is implemented as a Mathematica package. Mathematica evaluates equation symbolically by default, allowing to see that fractions are equal without having to specify a precision. This precision is normal when dealing with floating point (i.e. numerical) equality. For performance reasons we decided to force Mathematica to evaluate most of the algorithm numerically. Because of this, we need to define a precision margin for two floating point numbers to be 'equal enough'. For the algorithm, we accept two numbers a and b as equal iff $|a - b| \leq \frac{1}{100000}$.

All experiment graphs are generated using **R**. For each experiment, the Adjacency matrix is constructed in Mathematica from output generated by our experiments that use the YAWNS algorithm.

To execute the algorithm, we need to provide a couple of input parameters to the algorithm.

1. Adjacency matrix A , I.e. A list of lists, where each list represents a row.
2. PDF vector \vec{R} . The algorithm expects that all elements of \vec{R} use the same free variable symbol (i.e. the variable to integrate over). Because we need to integrate over \vec{R} 's elements, we also need to supply the free variable symbol to the algorithm. Thus, the algorithm expects a list of two elements:
 - A list of PDF functions with one free variable.
 - The free variable symbol.
3. Initial configuration C_{init} . I.e. A list of integers, indicating the amount of events to be scheduled at the LPs in the first ('initial') window.
4. Fixed lookahead value L .
5. Fixed value for estimated timestamp $0 \leq t \leq L$.
6. An upper bound M_q to the amount of YAWNS windows the algorithm 'schedules ahead' for.
7. An upper bound M_c to the amount of iterations allowed, in case no stable solution is found.

The algorithm outputs a list of two variables:

1. The index g that marks the solution row in state matrix C .
2. The state matrix C .

The example in Section 6.7.1 was computed with the following call:

```

ln[47]:= R =  $\left( \begin{array}{l} \mu e^{-x\mu} /. \{\mu \rightarrow 1\} \\ \mu e^{-x\mu} /. \{\mu \rightarrow 1\} \\ \mu e^{-x\mu} /. \{\mu \rightarrow 1\} \\ \mu e^{-x\mu} /. \{\mu \rightarrow 1\} \end{array} \right) // Flatten // N;$ 
```

```

L = 1;
n = 4;
Mq = 3;
Mc = 100;
t =  $\frac{L}{2}$ ;
A = {{{0,  $\frac{1}{3}$ ,  $\frac{1}{3}$ ,  $\frac{1}{3}$ }, { $\frac{1}{3}$ , 0,  $\frac{1}{3}$ ,  $\frac{1}{3}$ }, { $\frac{1}{3}$ ,  $\frac{1}{3}$ , 0,  $\frac{1}{3}$ }, { $\frac{1}{3}$ ,  $\frac{1}{3}$ ,  $\frac{1}{3}$ , 0}}};
{g, c} = ApproximatePDESParallelism[A, {R, x}, {10, 10, 10, 10}, L, t, Mq, Mc]
```

```

Out[46]= {26, {{10, 10, 10, 10, 1.}, {3.93469, 3.93469, 3.93469, 3.93469, 2.}, {5.38219, 5.38219, 5.38219, 5.38219, 3.},
{5.03674, 5.03674, 5.03674, 5.03674, 4.}, {5.42116, 5.42116, 5.42116, 5.42116, 5.},
{5.14626, 5.14626, 5.14626, 5.14626, 6.}, {5.25557, 5.25557, 5.25557, 5.25557, 7.},
{5.21905, 5.21905, 5.21905, 5.21905, 8.}, {5.23938, 5.23938, 5.23938, 5.23938, 9.},
{5.22623, 5.22623, 5.22623, 5.22623, 10.}, {5.23267, 5.23267, 5.23267, 5.23267, 11.},
{5.23003, 5.23003, 5.23003, 5.23003, 12.}, {5.23127, 5.23127, 5.23127, 5.23127, 13.},
{5.23058, 5.23058, 5.23058, 5.23058, 14.}, {5.23094, 5.23094, 5.23094, 5.23094, 15.},
{5.23077, 5.23077, 5.23077, 5.23077, 16.}, {5.23085, 5.23085, 5.23085, 5.23085, 17.},
{5.23081, 5.23081, 5.23081, 5.23081, 18.}, {5.23083, 5.23083, 5.23083, 5.23083, 19.},
{5.23082, 5.23082, 5.23082, 5.23082, 20.}, {5.23082, 5.23082, 5.23082, 5.23082, 21.},
{5.23082, 5.23082, 5.23082, 5.23082, 22.}, {5.23082, 5.23082, 5.23082, 5.23082, 23.},
{5.23082, 5.23082, 5.23082, 5.23082, 24.}, {5.23082, 5.23082, 5.23082, 5.23082, 25.},
{5.23082, 5.23082, 5.23082, 5.23082, 0}, {3.17265, 3.17265, 3.17265, 3.17265, 0},
{1.16715, 1.16715, 1.16715, 1.16715, 0}, {0.429372, 0.429372, 0.429372, 0.429372, 0},
{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0},
{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0},
{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0},
{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0},
{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0},
{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0},
{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0},
{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}}

```

Figure 7.1: The result of the approximation algorithm. Here the stable solution can be found in row 26 (indicated by the first element), in the outputted state matrix c (second element, where each element indicates a row)

7.2 The experimentation framework

The experimentation framework has the ability to run the EPHOLD benchmark using a user-defined PDES synchronization algorithm. In this section we will show how we can use our framework to conduct experiments (Section 7.2.1), the dependencies it has. (Section 7.2.2) and the input it requires (Section 7.2.3). Next, we will discuss the output it generates (7.2.4) and some of the implementation details (Section 7.2.5).

7.2.1 Running experiments

Once compiled, the experimentation framework is operated via the command line. And takes three arguments, being the scenario id, the PDES synchronization algorithm to use, and whether to enable debug logging. Available scenario Ids are listed in Sections 8.1.1 and 8.1.2.

Scenarios differ by the PDES graph, and distributions used. The graphs are loaded from text files formatted in the edge list format (indicated by the ".edg" extension). For the distributions we use the distributions available in the C++11 libraries. The framework works with a concept of so called

Scenarios and Experiments. A scenario dictates the EPHOLD configuration (PDF functions, PDES graph, etc.) and an experiment actually runs the EPHOLD benchmark. each scenario executes 100 experiments using different seeds.

7.2.2 Code dependencies

The experimentation framework is written in the C++ programming language with C++11 features enabled. Messages between LPs are sent using the OpenMPI library [26]. Unfortunately, the current implementation of the Message Passing Interface (MPI) lacks support for Windows Operating Systems. Thus the code was written on an Ubuntu 16.04 Operating System (OS) using the Eclipse Parallel Tools Platform IDE (Eclipse PTP) [19]. The other dependency is used to generate figures of the used PDES graphs. For this, a system call to invoke the GraphViz library[27] is used.

7.2.3 Input

The framework uses several files and parameters as input.

1. Command line arguments i.e. The name of the PDES synchronization algorithm, the scenario id, and '1', or '0' to enable, or disable debug logging respectively.
2. The PDES graph files (in .edg format)
3. (Optional) Seed file

Available scenario Ids are listed in Sections 8.1.1 and 8.1.2.

The available synchronization algorithms and their corresponding parameter codes are:

Synchronization algorithm	Parameter
YAWNS	YAWNS
Chandy Misra & Bryant	CMB
Time Warp with incremental state saving	TW

To actually run the framework, we have to invoke a program called **mpirun**. This program, provided by the OpenMPI framework, sets up the MPI capable threads (or 'nodes') and distributes these across the available processor cores. The only arguments it needs from us are the amount of threads (LPs) we

wish to use, a path to our executable, followed by our own parameters. E.g. To execute Scenario 5, with 4 LPs, using the YAWNS algorithm, with debug logging disabled, one would execute:

```
mpirun -n 4 YAWNS 5 0
```

The PDES graph files are located in the Graphs folder, and are automatically loaded based on the provided scenario Id. If there is no seed file available one will be created with automatically generated seeds. These seeds are generating using a Pseudo Random Number Generator(PRNG) initialized with seed: $3539733 \cdot (\text{ScenarioId} + 1)$. The number in the $k \cdot e$ -th row will be used as seed to initialize the pseudo random number generator at LP_k in experiment $1 \leq e \leq 100$. E.g. for a complete PDES graph of four nodes, we have $4 \cdot 100 = 400$ different seeds.

7.2.4 Output

In this section we will discuss the content of the output files produced by the experiments. The **Output** folder holds three other folders, being: **Log**, **Messages** and **Parallelism**. The **Log** folder holds the debug log files that each LP produces during the execution of each Experiment (if enabled). These files are only relevant when debugging issues.

The **Messages** folder holds, for each experiment, two files per LP.

The **LP1_PHOLD_ESP.txt** file contains a record of every event message sent from LP_1 to any other LP. It contains, the LP that the event message is sent from, the timestamp of the event that was executed, the destination LP of the newly generated event, and the scheduled timestamp of the sent event. This file is used to test the validity the synchronization algorithms. Finally, the **LP1_WeightChange.txt** file contains the weights of all its neighbors. This file is later used to determine adjacency matrix A in our approximation algorithm.

The **Parallelism** folder contains, per LP, a log of all parallelism statistics that we are interested in. For YAWNS, these are all windows that occurred per Experiment. For each window, it includes its lower bound, upper bound and amount of events that particular LP had scheduled in the window. For the CMB algorithm, we log the amount of Null messages, and the amount of event messages. Finally, the Time Warp algorithm logs statistics of all Rollbacks that occur.

Finally, Every scenario outputs a **Graph.dot** file, an export of the PDES graph in ".dot" format, and a **Graph.jpg** file, a visual representation of the PDES graph.

7.2.5 Details

In this section we will highlight some of the implementation details. The foremost advantage of this implementation is that all PDES synchronization algorithms are implemented using the same code base. All algorithms share the same implementation for the base LP, Event Scheduling and Output mechanisms.

YAWNS

Although YAWNS algorithm is relatively simple to explain, and is simple to implement, there is an issue that needs to be addressed. Let our PDES graph consist of 2 connected nodes. Consider the following sequence of events:

1. Both LPs start processing events in window $[0, 1)$
2. LP_1 sends an event message e to LP_2 , attempting to schedule an event at 1.15.
3. Both LPs have executed their events and move on to e.g. window $[1, 2)$, and start executing the events they have scheduled in the window.
4. LP_2 receives e during the window.

This is a clear violation of the algorithm's design. However, the MPI specification does not guarantee that a message has been received after the 'send' function call has been completed. Even placing a so called 'Barrier' (a construct that forces all MPI threads/LPs to only continue execution when all LPs have reached that barrier) does not resolve this. Without elaborating too much on the details, the MPI standard only ensures that the order of messages sent is only preserved between two LPs. E.g. LP_1 emits e_1, e_2, e_3 and e_4 to LP_2, LP_3, LP_2 and LP_2 respectively. Then the standard ensures that LP_2 receives e_1, e_3 and e_4 in order, but places no restrictions on the delivery time of e_2 to LP_3 in this context. Using this to our advantage, we resolved this issue as follows. When an LP is finished executing its last event in the window, it sends an additional message to all its outgoing neighbors, indicating to the receiving LPs, that the originator is done sending event messages. When all such a message are received from all incoming neighbors, the receiving LP derives it will not receive any more messages and can thus proceed to a barrier. When all LPs arrive at the barrier, the new window bounds are computed and the algorithm proceeds as intended.

7.3 Analysis

To analyse the output generated by our experiments, we mainly used Mathematica. All charts in Chapter 8 are generated using Mathematica notebooks. We noticed that the Time Warp algorithms produce a lot of data. We developed a C# console application to preprocess this raw data. The preprocessed data is then read into Mathematica to be further processed and generate the charts. Here we present the list of notebooks and applications used to analyze generated data:

1. `TimeWarpRawOutputAnalyzer`, a C# Console application to preprocess raw rollback statistics outputted by TimeWarp experiments.
2. `TimeWarpAnalysis.nb` is used to generate charts based on the preprocessed data
3. `NullvsEventMessagesAnalysis.nb` is used to generate charts for CMB experiments.
4. `MeanParallelismWithModel.nb` generates parallelism statistics (Section 5.1) for the analysis of YAWNS experimental data, and compares it with the approximation model's solution.
5. `MeanDifferenceWithModel.nb` is used to analyze the amount of events scheduled per window from experimentation, and compare it with the approximation model's prediction.

Chapter 8

Experiments

In this chapter we will evaluate the experiments we performed. We set up our way of working (Section 8.1), such that each set of experiments is attached to a single scenario (Sections 8.1.1 and 8.1.2). More specifically, a scenario fixes the PDES graph, and the movement function, for a set of 100, or 30 experiments depending on the scenario. We test all scenarios using the YAWNS, CMB and Time Warp algorithm and make the distinction between experiments performed on complete graphs (Sections 8.2.1, 8.3.1 and 8.4.1, for YAWNS, CMB and TimeWarp resp.) and scale free graphs (Section 8.2.3, 8.3.2 and 8.4.2). Finally we will discuss the overall results, and conclusions we can draw from them (Section 8.5).

8.1 Setup

As mentioned earlier, each scenario varies the PDES graph, and movement function. First, we run our approximation algorithm (Chapter 6) for each scenario. Next we run the YAWNS, CMB and Time Warp algorithm on the same graph, and EPHOLD weight configuration. Each algorithm is run 100, or 30 times per scenario. Each run (i.e. experiment) has a termination time of 3000. (i.e. in simulation time) and schedules 10 events at the start at every LP. Their timestamps are increments of values drawn from the $Exp(1)$ distribution. i.e. the timestamp of an event is the sum of the timestamp of the previous event (or 0, in case of the first event to be scheduled), and a value x , drawn from an $Exp(1)$ distribution. The EPHOLD movement function selects a neighboring LP using a discrete probability distribution, based on the EPHOLD weights of outgoing edges in the PDES graph. Finally, The timestamp increment function is defined as $L + x$, with $L = 1$, and value x drawn from an $Exp(1)$ distribution (i.e. the standard PHOLD definition).

Per experiment, we measure the following:

1. In the YAWNS experiments: The amount of events executed per window, per LP.
2. In the CMB experiments: The total amount of simulation message and total amount of NULL messages sent.
3. In the TimeWarp experiments: The total amount of Idle rollbacks, and Busy rollbacks.

From these statistics, we can calculate the amount of parallelism (Chapter 5). Using these experimental results, together with the scenario's configuration (i.e. PDES graph topology, and weights (Sections 8.1.1 and 8.1.2), we intend to answer all questions relating to PHOLD (Section 3.1.1). We intend to compare the results obtained from the YAWNS experiments to the values generated by the approximation algorithm. This will answer all questions posed in Section 3.1.2. The experiments are run on a MSI Intel i7 quad-core laptop, so most experiments are *saturated*. Each scenario generates $n \cdot 100$ seeds, one for each LP per experiment. A Pseudo Random Number Generator seeded with $(\text{Scenario Id} + 1) \cdot 3539733$ is initialized per scenario. The Pseudo Random Number Generator then draws the $n \cdot 100$ seeds that are used to initialize the Pseudo Random Number Generators for each LP per experiment. For our scenarios, we've generated sets of complete (Section 8.1.1) and scale free graphs (Section 8.1.2) using the **R** code presented in Section 8.1.3. We note that all graphs used here are undirected. Which we model by using two opposing directed edges in the PDES synchronization algorithms, and approximation algorithm.

8.1.1 Scenarios for complete graphs

Table 8.1 associates Scenario Ids with Graph file names. Each file name is formatted as "<n>x<n>.edg", where n is the amount of nodes in the complete graph. In total, we have 46 scenarios using complete graphs.

Scenario Id	Graph File Name
0, 1	3x3.edg
2, 3	4x4.edg
4, 5	5x5.edg
6, 7	6x6.edg
8, 9	7x7.edg
10, 11	8x8.edg
12, 13	9x9.edg
14, 15	10x10.edg
16, 17	11x11.edg
18, 19	12x12.edg
20, 21	13x13.edg
22, 23	14x14.edg
24, 25	15x15.edg
26, 27	16x16.edg
28, 29	17x17.edg
30, 31	18x18.edg
32, 33	19x19.edg
34, 35	20x20.edg
36, 37	21x21.edg
38, 39	22x22.edg
40, 41	23x23.edg
42, 43	24x24.edg
44, 45	25x25.edg

Table 8.1: A list of scenario Ids and the graph files they load. E.g. 9x9.edg loads a complete graph of 9 nodes

Note that for every two scenarios in table 8.1 The difference in even and odd scenario ids is the choice of the weight function per LP. Let $w_{k,j}$ be the weight for edge (v_k, v_j) in the PDES graph. Then we have:

$$w_{k,j} = \begin{cases} \frac{1}{|V|-1} & \text{Scenario Id is even} \\ \frac{j+1}{Q_k} & \text{Scenario Id is odd} \end{cases}$$

With Q_k a normalization constant specific to LP_k . In our implementation, j can be 0, thus we add 1 to j so that we will also send messages to LP_0 .

Note that the even numbered scenarios thus simulate PHOLD. In complete graphs, all nodes have equal in-, and out-degree. Thus the only way to differentiate between nodes is by node index. We hypothesize that even scenarios will result in a similar workload for all LPs, and an uneven workload for odd numbered scenarios.

Finally, we define scenarios 126 through 217 as our '**randomized scenarios**'. These are scenarios where we randomize the EPHOLD weights per LP. We've organized the scenario ids in groups of 4 similarly. E.g. scenarios 126, 127, 128 and 129 are run on the 3 node complete graph, 130 - 133 on the 4 node graph, etc.

8.1.2 Scenarios for scale free graphs

In this section we introduce the scenarios that use Scale Free PDES graphs. The format of the filename indicates several parameters of the scale free network. As we know, the degree distribution of a scale free distribution is determined by the power of its power law distribution (i.e. λ). Each Scale free PDES graph file name has the following format.

ScaleFree_p< λ >_n10_<i>.edg

Where:

1. p< λ > indicates the exponent λ of the power law exponent.
2. n10 indicates the amount of nodes in the graph (i.e. 10, in this case).
3. i indicates that the graph is a result of a i-th graph generated with the same values of λ and n .

Similar to the complete graph, table 8.2 associates scenario Ids to Scale-Free PDES graphs.

The even numbered scenarios again simulate PHOLD using the EP HOLD algorithm. Odd numbered scenarios again have a different weight function. Differentiating between the LP ids, like we did for the complete graphs, does not make a lot of sense here. Instead we differentiate by degree.

We have for weight $w_{k,j}$ for edge (v_k, v_j) in the PDES graph:

$$w_{k,j} = \begin{cases} \frac{1}{|V|-1} & \text{Scenario Id is even} \\ \frac{1+\mathcal{S}_k-\text{degree}(v_j)}{Q_k} & \text{Scenario Id is odd} \end{cases}$$

With Q_k a normalization constant for LP_k , such that all weights for LP_k sum to 1, and \mathcal{S}_k the sum of all degrees of all outgoing neighbors of LP_k . Note that the even numbered scenarios again simulate PHOLD. We hypothesize that the nodes with high degree will become bottlenecks in the even numbered scenarios. We try to alleviate this in the odd numbered scenarios, by sending more event messages to nodes with low degree. We hypothesize that the odd numbered scenarios will have a more even workload across LPs, and thus result in an increase in parallelism.

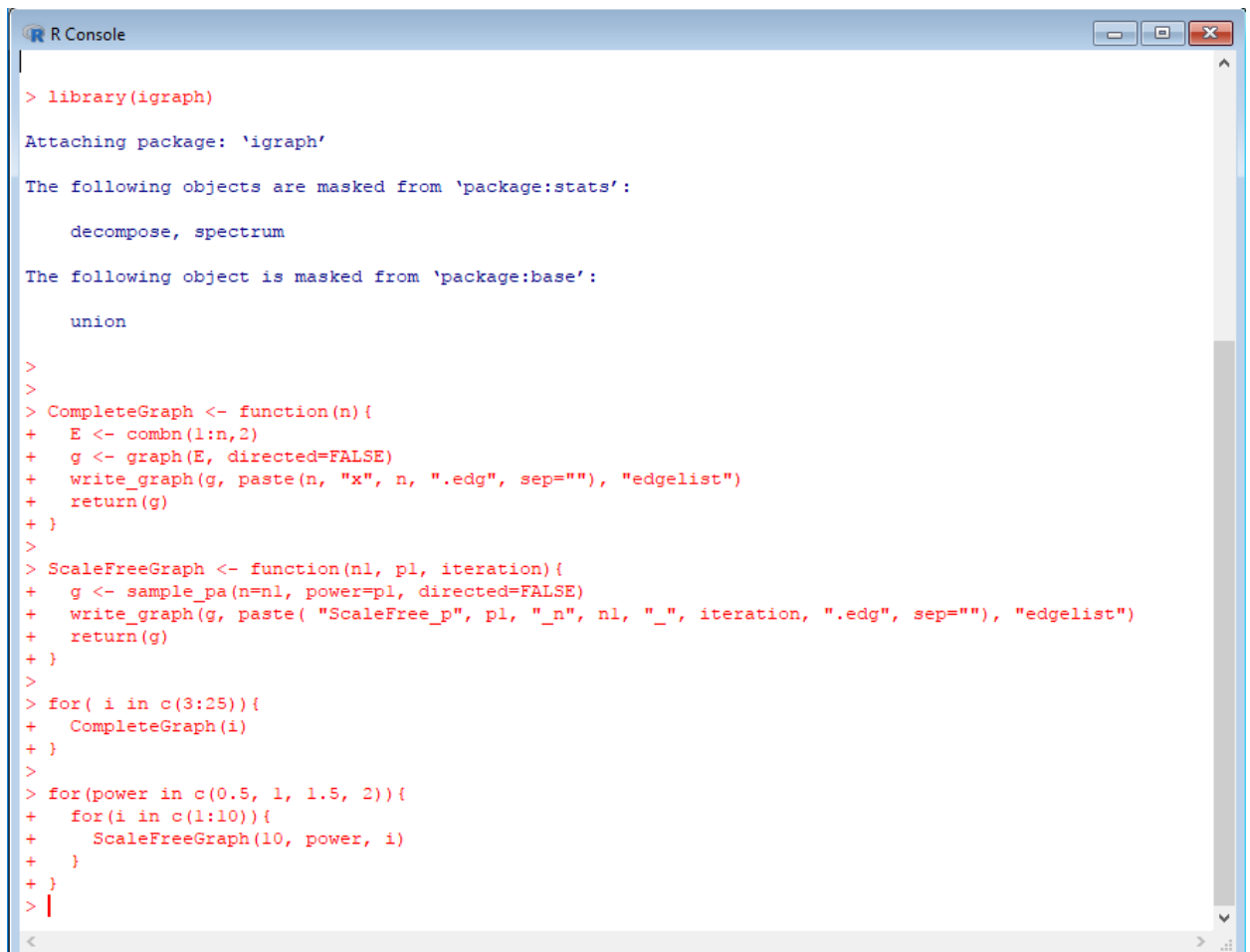
Scenario Id	Graph File Name
46, 47	ScaleFree_p0.5_n10_1.edg
48, 49	ScaleFree_p0.5_n10_2.edg
50, 51	ScaleFree_p0.5_n10_3.edg
52, 53	ScaleFree_p0.5_n10_4.edg
54, 55	ScaleFree_p0.5_n10_5.edg
56, 57	ScaleFree_p0.5_n10_6.edg
58, 59	ScaleFree_p0.5_n10_7.edg
60, 61	ScaleFree_p0.5_n10_8.edg
62, 63	ScaleFree_p0.5_n10_9.edg
64, 65	ScaleFree_p0.5_n10_10.edg
66, 67	ScaleFree_p1_n10_1.edg
68, 69	ScaleFree_p1_n10_2.edg
70, 71	ScaleFree_p1_n10_3.edg
72, 73	ScaleFree_p1_n10_4.edg
74, 75	ScaleFree_p1_n10_5.edg
76, 77	ScaleFree_p1_n10_6.edg
78, 79	ScaleFree_p1_n10_7.edg
80, 81	ScaleFree_p1_n10_8.edg
82, 83	ScaleFree_p1_n10_9.edg
84, 85	ScaleFree_p1_n10_10.edg
86, 87	ScaleFree_p1.5_n10_1.edg
88, 89	ScaleFree_p1.5_n10_2.edg
90, 91	ScaleFree_p1.5_n10_3.edg
92, 93	ScaleFree_p1.5_n10_4.edg
94, 95	ScaleFree_p1.5_n10_5.edg
96, 97	ScaleFree_p1.5_n10_6.edg
98, 99	ScaleFree_p1.5_n10_7.edg
100, 101	ScaleFree_p1.5_n10_8.edg
102, 103	ScaleFree_p1.5_n10_9.edg
104, 105	ScaleFree_p1.5_n10_10.edg
106, 107	ScaleFree_p2_n10_1.edg
108, 109	ScaleFree_p2_n10_2.edg
110, 111	ScaleFree_p2_n10_3.edg
112, 113	ScaleFree_p2_n10_4.edg
114, 115	ScaleFree_p2_n10_5.edg
116, 117	ScaleFree_p2_n10_6.edg
118, 119	ScaleFree_p2_n10_7.edg
120, 121	ScaleFree_p2_n10_8.edg
122, 123	ScaleFree_p2_n10_9.edg
124, 125	ScaleFree_p2_n10_10.edg

Table 8.2: A list of scenario Ids and the graph files they load.

Finally, we define scenarios 218 through 377 as our '**randomized scenarios**'. These are scenarios where we randomize the EPHOLD weights. We've organized the scenario ids in groups of 4, i.e., scenarios 218, 219, 220 and 221 are run on the ScaleFree_p0.5_n10_1.edg graph, 222 - 225 on the ScaleFree_p0.5_n10_2.edg graph, etc.

8.1.3 Generating the graphs

All graphs are generated with **R** using the igraph library. The **R** code to generate all graphs is showcased in figure 8.1



```
R Console
> library(igraph)
Attaching package: 'igraph'
The following objects are masked from 'package:stats':
  decompose, spectrum
The following object is masked from 'package:base':
  union
>
>
> CompleteGraph <- function(n){
+   E <- combn(1:n,2)
+   g <- graph(E, directed=FALSE)
+   write_graph(g, paste(n, "x", n, ".edg", sep=""), "edgelist")
+   return(g)
+ }
>
> ScaleFreeGraph <- function(n1, p1, iteration){
+   g <- sample_pa(n=n1, power=p1, directed=FALSE)
+   write_graph(g, paste("ScaleFree_p", p1, "_n", n1, "_", iteration, ".edg", sep=""), "edgelist")
+   return(g)
+ }
>
> for( i in c(3:25)){
+   CompleteGraph(i)
+ }
>
> for(power in c(0.5, 1, 1.5, 2)){
+   for(i in c(1:10)){
+     ScaleFreeGraph(10, power, i)
+   }
+ }
> |
```

Figure 8.1: The interactive **R** session used to generate the graphs

8.2 YAWNS

In this Section we analyze the average amount of events scheduled per window per LP, and compare this with the solution from the approximation algorithm presented in Chapter 6. For each run, we record the transmitted event messages and compute the amount of parallelism, which is then compared to the output of the approximation algorithm. We ran all scenarios and measured the average amount of events per window, and computed the difference between the results of the approximation algorithm and the measured data. The results are numerically summarized per LP, in Tables 9.1, 9.2, 9.3 and 9.4 in the Appendix.

Even numbered Scenarios

For the even numbered scenarios we present the progression of the measured mean amount of messages per window for scenarios 0, 10, 20, 30 and 40 in Figures 8.2, 8.3, 8.4, 8.5 and 8.6 respectively. The predicted amount of events per window is plotted as a horizontal dashed line. These figures plot the 'running average' of the amount of messages executed per window, per LP (Equation 8.1) based on experimental data.

$$f_i(M_g) = \frac{1}{M_g} \cdot \sum_{k=0}^{M_g} m_{i,k} \quad (8.1)$$

For the first windows M_g , with amount of messages in the k -th window at LP_i , $m_{i,k}$. $f_i(M_g)$ thus expresses the 'running average' of the amount events executed in the first M_g windows for LP_i . We can also plot the amount of parallelism (Section 5.1), which we will do in Section 8.2.1. The advantage of plotting it this way, is that we can evaluate the precision of the predictions with a bit more accuracy, because we can see the expected vs. actual amount of events per window, per LP.



Figure 8.2: Progression of the average amount of events per window, and the approximated solution (in black) for Scenario 0

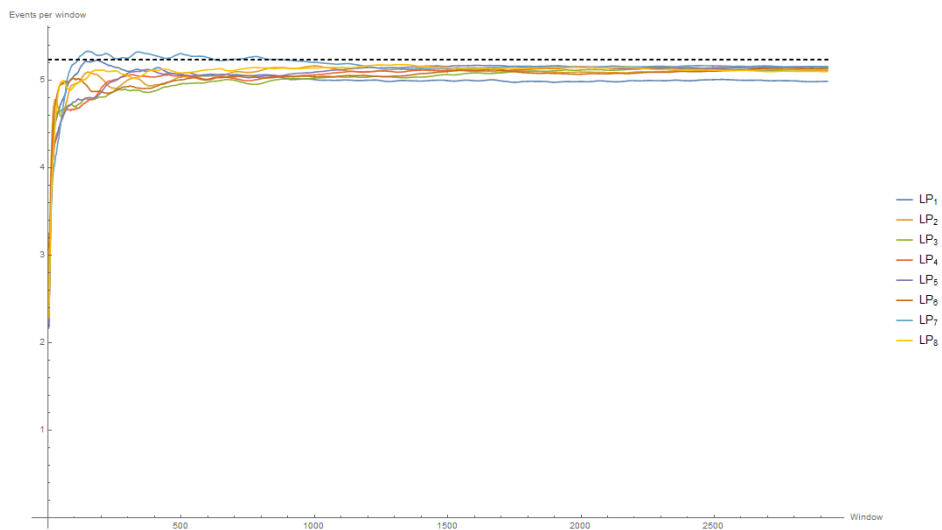


Figure 8.3: Progression of the average amount of events per window, and the approximated solution (in black) for Scenario 10

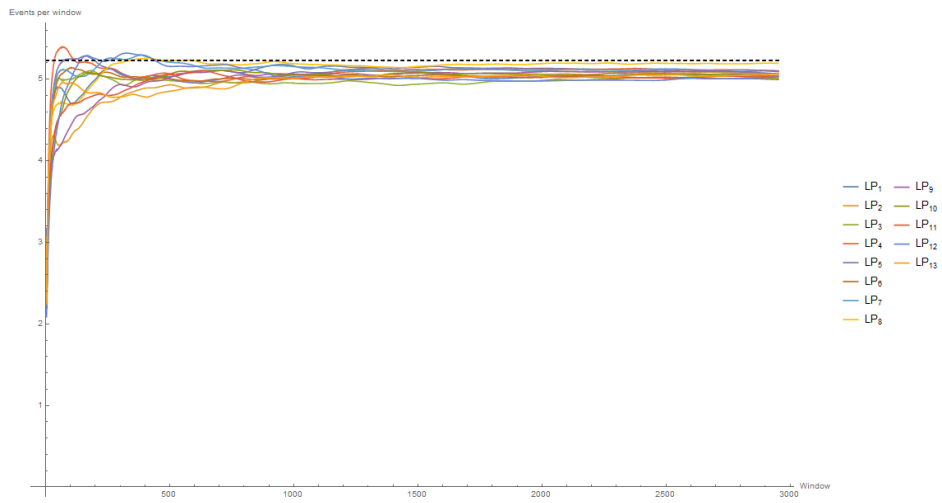


Figure 8.4: Progression of the average amount of events per window, and the approximated solution (in black) for Scenario 20

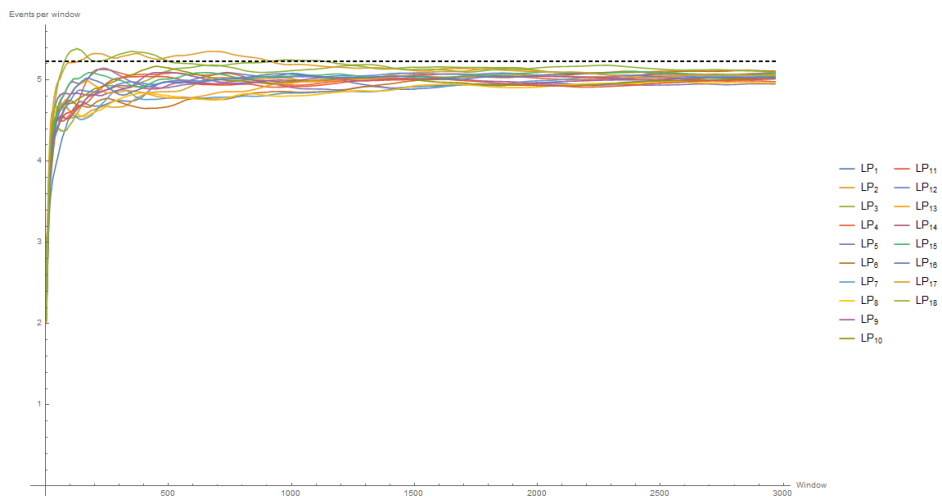


Figure 8.5: Progression of the average amount of events per window, and the approximated solution (in black) for Scenario 30

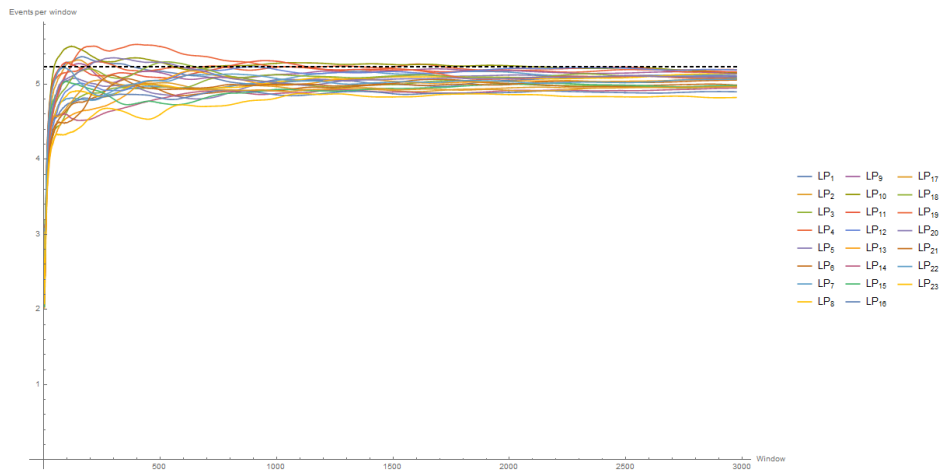


Figure 8.6: Progression of the average amount of events per window, and the approximated solution (in black) for Scenario 40

Interestingly enough, we can clearly see that for small complete graphs, the approximation algorithm performs relatively well for PHOLD. Though when the amount of nodes in the graph increase, so does the deviation from the predicted values. It seems that, for larger PDES graphs, either our assumption: $t = \frac{L}{2}$ was too presumptuous, or we did not reach a sufficiently steady enough state in our experiments. We will further investigate this in section 8.2.1.

Odd numbered Scenarios

For the odd numbered LPs we inspect the differences visually in Figures 8.7, 8.8, 8.9, 8.10 and 8.11 for scenarios 1, 11, 21, 31 and 41 respectively. Again, the predicted amount of events per window is plotted as horizontal dashed lines. This time they are color coded per LP, matching the LP color of the experimental data.

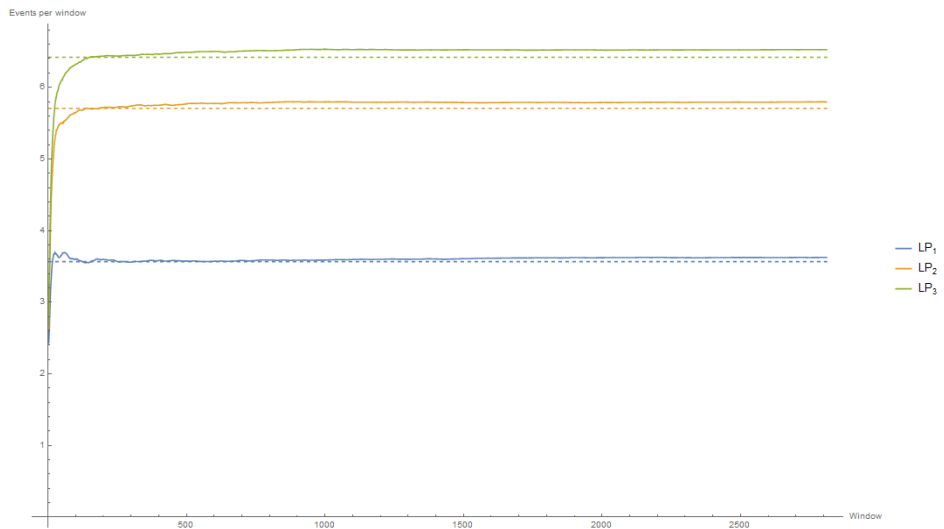


Figure 8.7: Progression of the average amount of events per window, and the approximated solution (dashed) for Scenario 1

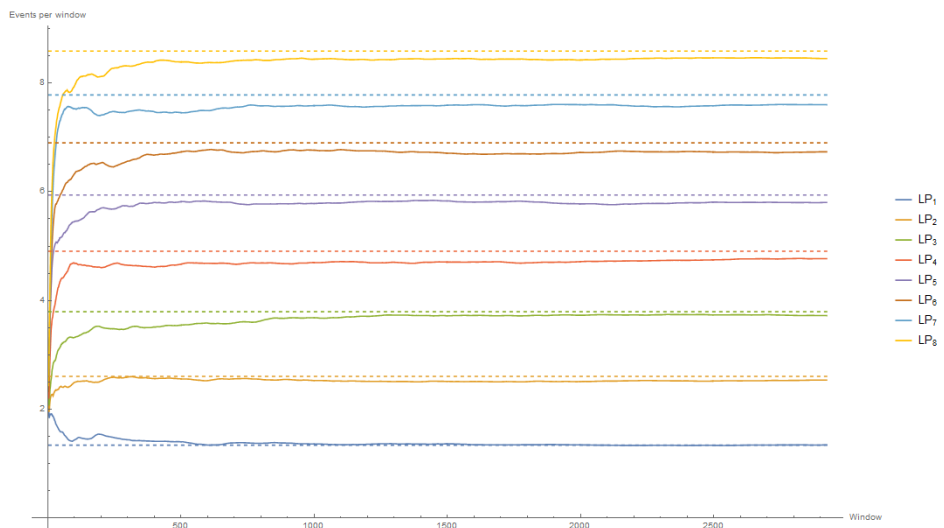


Figure 8.8: Progression of the average amount of events per window, and the approximated solution (dashed) for Scenario 11

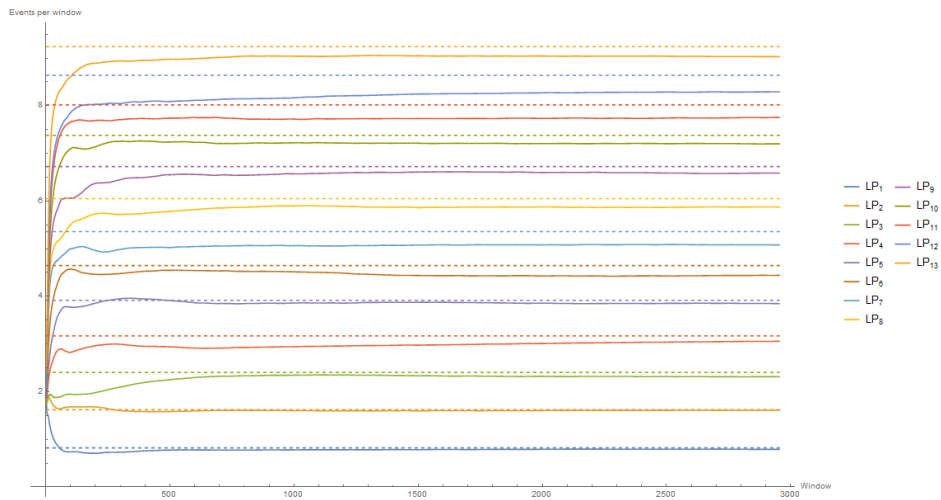


Figure 8.9: Progression of the average amount of events per window, and the approximated solution (dashed) for Scenario 21

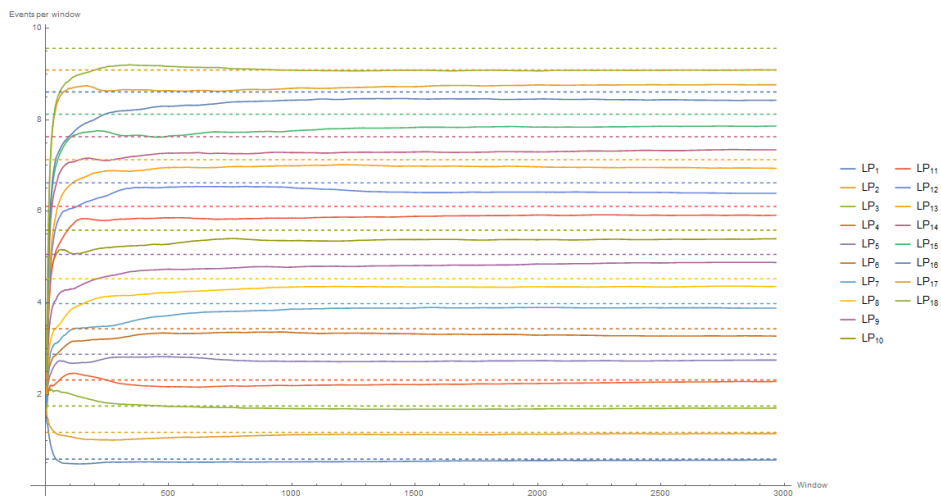


Figure 8.10: Progression of the average amount of events per window, and the approximated solution (dashed) for Scenario 31

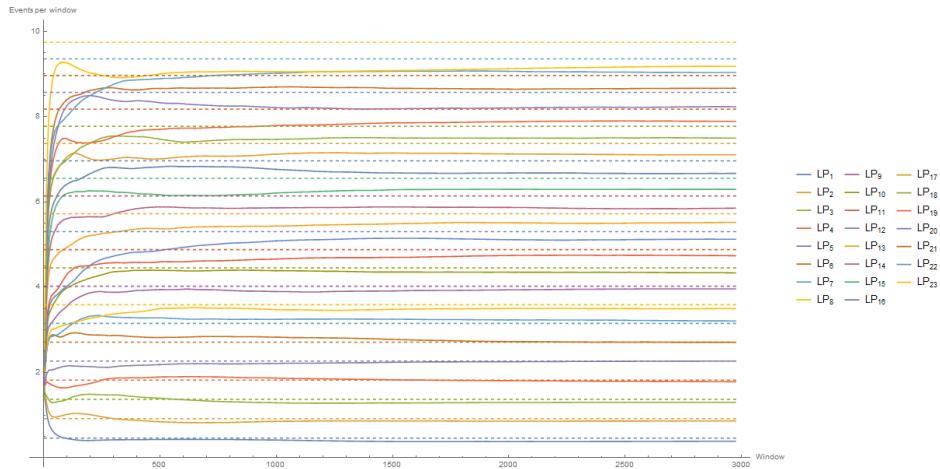


Figure 8.11: Progression of the average amount of events per window, and the approximated solution (dashed) for Scenario 41

We can clearly see that both the approximation algorithm and the experimental results point to a highly non-parallelizable configuration. We see the biggest difference between LPs for scenario 41. The bottleneck LP processes around 9 events per window, whereas the LP with the most idle time only handles 0.5 messages per window on average. Similar to the even numbered scenarios, the approximation seems to lose its predictive capabilities as the PDES graph size increases. We will further investigate this in section 8.2.1.

8.2.1 Parallelism in complete graphs

In this section, we analyze the amount of predicted vs. the amount of observed parallelism. In short, parallelism is defined by the sum of the average amount of events per window, divided by the average amount of events that the bottleneck LP has to process per window. In this analysis we will investigate the *approximation error*, which we define as $100 \cdot (1 - \frac{P_o}{P_p})$, with predicted amount of parallelism P_p , and observed amount P_o .

Even numbered scenarios

When we plot the average predicted and observed amount of parallelism for all even numbered scenarios (Figure 8.12) we observe that:

1. The amount of Parallelism grows linearly proportional to the size of the PDES graph

2. The predicted amount of parallelism seems to be higher, or equal to the observed amount of parallelism
3. The approximation error seems to increase when the PDES graph size increases.

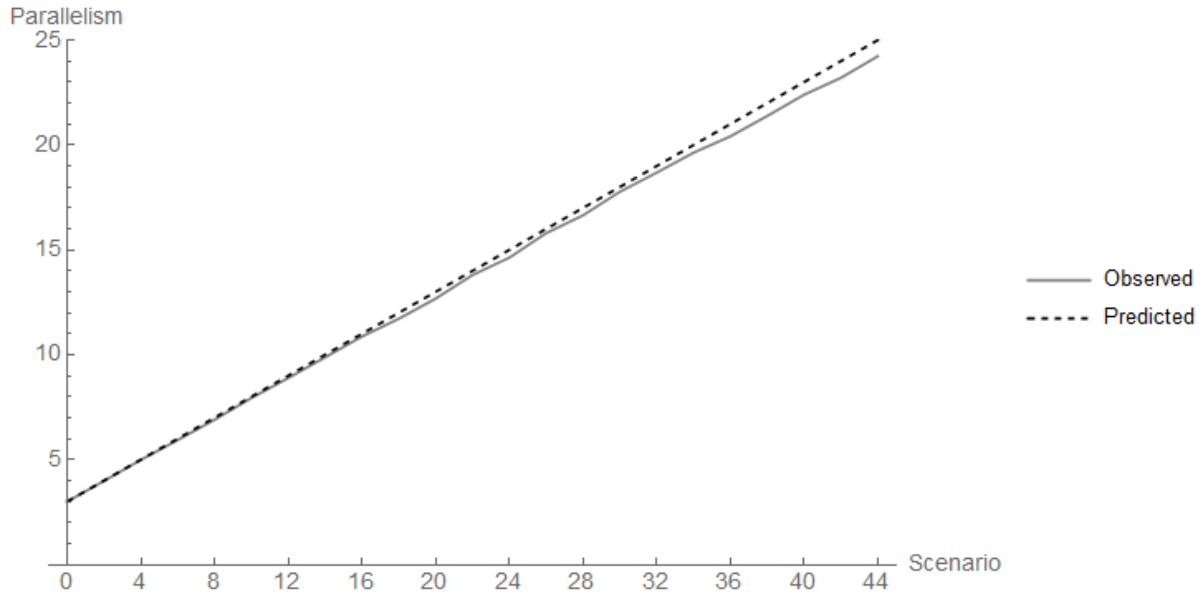


Figure 8.12: Predicted and observed parallelism for even numbered scenarios (i.e. PHOLD scenarios) for complete graphs

The fact that the approximation algorithm overestimates the amount of parallelism for the even numbered scenarios is not that surprising (observation 2). The approximation algorithm consistently predicts the maximum amount of parallelism possible (i.e. n for a PDES graph of n nodes), thus observing a higher amount of parallelism would indicate an error in our experiments. Because these are complete graphs, we have that every node is a neighbor to every other node. When an LP selects a neighboring LP to schedule a new event on, the edge, to send the event message over, is chosen uniformly random (for even scenarios) from the LPs outgoing edges. This creates a highly uniform workload over all LPs, where all LP have to process about the same amount of events per window, maximizing the amount of parallelism.

Observation 3 is similar to what we observed in the previous section, and warrants further investigation. If we look at the numerical approximation errors (8.3), we see that a 'large' error occurs for scenario 38.

Scenario Id	Predicted	Observed	Error %
0	3.	2.995	0.167%
10	8.	7.939	0.762%
20	13.	12.678	2.477%
24	15.	14.63	2.467%
32	19.	18.683	1.668%
34	20.	19.635	1.825%
36	21.	20.415	2.786%
38	22.	21.384	2.8%
40	23.	22.399	2.613%
42	24.	23.204	3.317%
44	25.	24.235	3.06%

Table 8.3: The predicted and observed amount of parallelism, for an even numbered subset of scenarios. The results for all complete even scenarios are shown in table 9.5 in the Appendix

We can also discern that the error increases with the scenario id. This is perhaps better highlighted by Figure 8.13

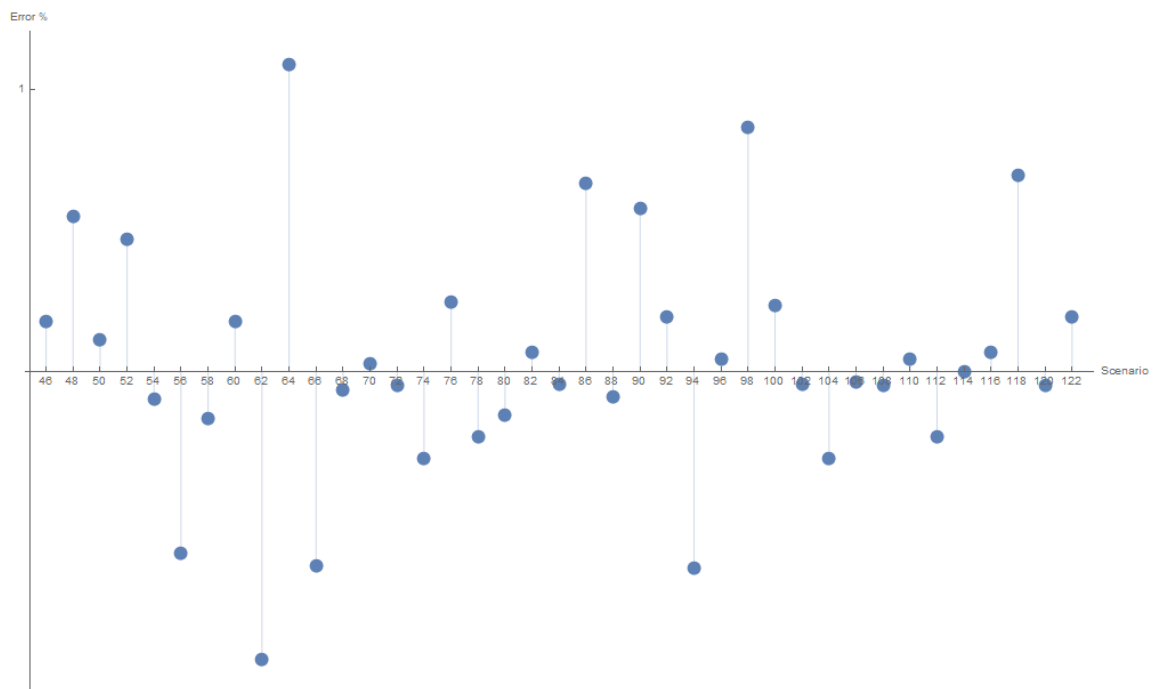


Figure 8.13: Approximation error for the even scenarios

We hypothesize that this is because we increase the amount of LPs per

scenario, but we keep the termination time of a single simulation run constant (i.e. 3.000). This probably causes the simulation to end up in a less 'stable' state, with regards to parallelism, for larger graphs. To test this hypothesis, we re-run a scenario with high error (e.g. Scenario 38), where we increase the termination time by a factor 10, resulting in a termination time of 30.000 per experiment run.

When we re-run Scenario 38 with increased termination time, the Predicted amount of parallelism remains 22, but now we observe a parallelism of 21.804. The approximation error thus becomes 0.891%. A reduction of a factor ≈ 3 . We conclude that the approximation error goes down, the longer the simulation is run. This indicates that the approximation algorithm works reasonably well.

Odd numbered scenarios

When we inspect the odd numbered scenarios, a different pattern emerges. We plot the predicted, and observed amount of parallelism per scenario (Figure: 8.14) again. Observing that Figure 8.14 is plotted to the same scale as 8.12, we can clearly see, when comparing these values to their even counterparts that:

1. The amount of parallelism grows less for odd numbered scenarios.
2. The amount of parallelism present in the odd scenarios is substantially less.
3. The Approximation either over, or under estimates the observed amount of parallelism.
4. The approximation error seems to increase with the size of the PDES graph

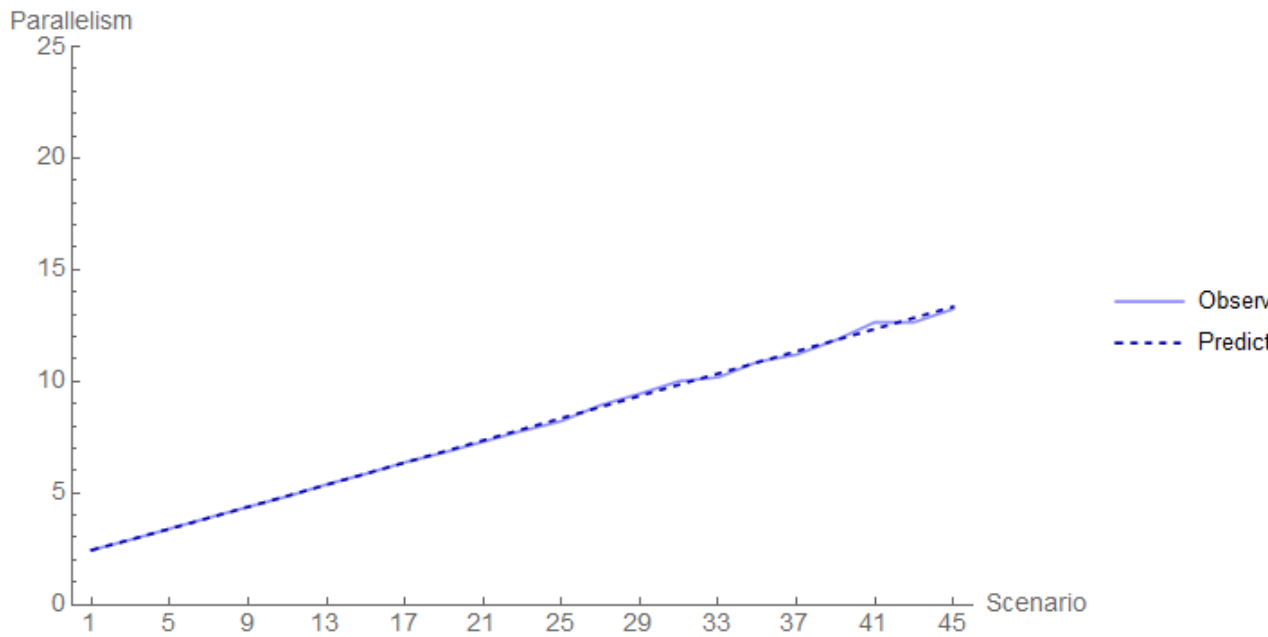


Figure 8.14: Predicted and observed parallelism for odd numbered scenarios (i.e. non-PHOLD scenarios) for complete graphs.

Observations 1 and 2 clearly highlight that general PHOLD executions do not tell the whole story, nor does the topology of the PDES graph. By only varying the weights in the EPHOLD algorithm we see that we can get less parallelism as a result. Just as in the previous section, we see that the approximation error seems to increase with the scenario id (and thus graph size). Figure 8.15 clearly highlights this observation.

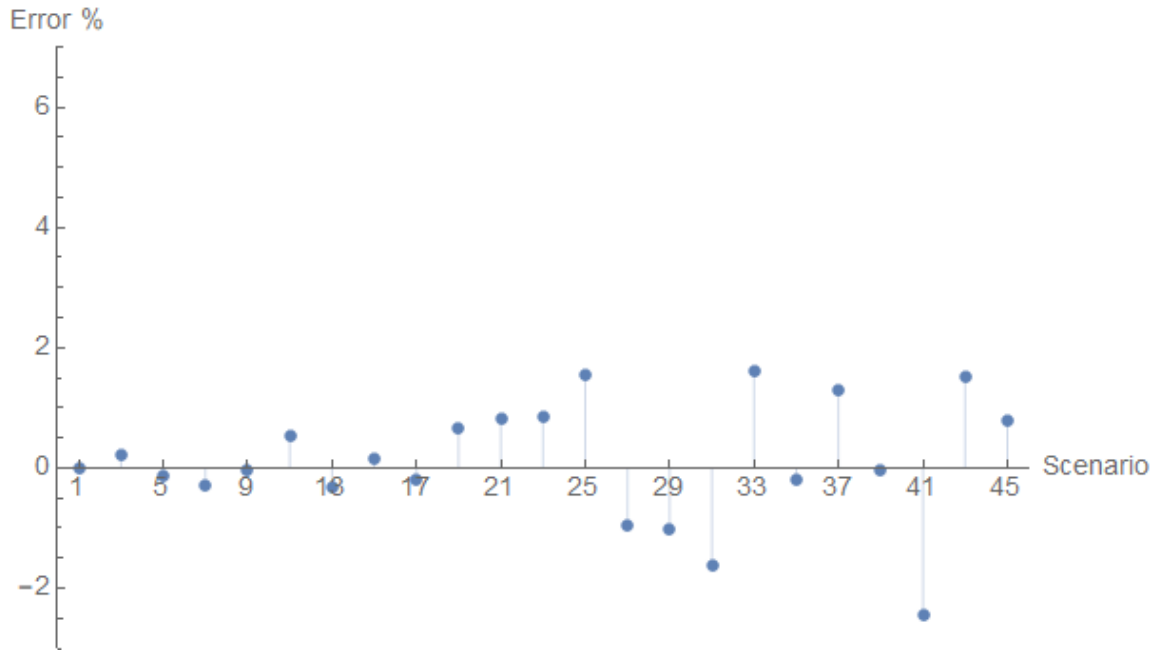


Figure 8.15: Approximation error for the odd scenarios with complete graphs

Table 8.4 shows the predicted, observed amount of parallelism, and the approximation error numerically for a subset of the odd scenarios. The numerical results for all complete odd scenarios are shown in table 9.5 in the Appendix

Scenario Id	Predicted	Observed	Error %
1	2.444	2.444	0%
11	4.875	4.849	0.533%
21	7.359	7.298	0.829%
25	8.356	8.227	1.544%
27	8.854	8.94	-0.971%
29	9.353	9.448	-1.016%
31	9.852	10.011	-1.614%
37	11.349	11.203	1.286%
41	12.348	12.648	-2.43%
43	12.847	12.654	1.502%

Table 8.4: The predicted and observed amount of parallelism, for an odd numbered subset of scenarios.

Scenario 41 has the highest error. Similar to the previous section, we hypothesize that this error is due to an instable state, of the simulation.

Thus we re-run this scenario, where we increase the termination time by a factor of 10 (resulting in a termination time of 30.000). After re-running this scenario the observed amount of parallelism became: 12.351, resulting in an approximation error of -0.024% . A very clear improvement.

Randomized weights

We've repeated the experiments on complete graphs using randomized weights. 4 additional scenarios were run per complete graph. This time we randomized the EPHOLD weights. Again the approximation algorithm was run for each scenario. Each scenario consists of 30 experiments. The results are summarized by figure 8.16, and table 9.7 in the appendix.

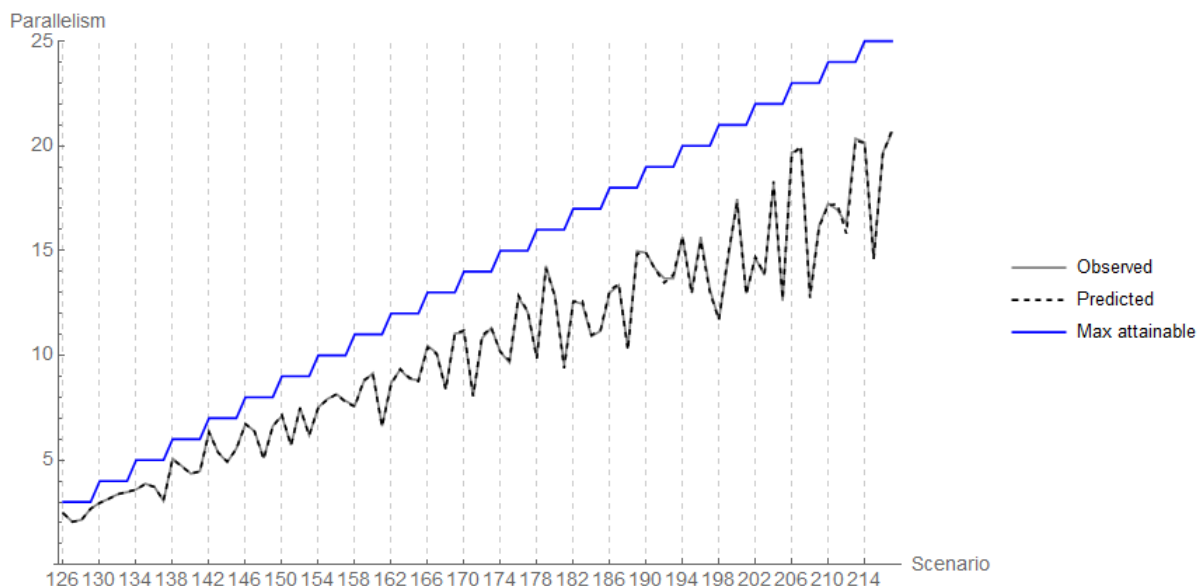


Figure 8.16: Amount of predicted and observed parallelism for complete PDES graphs with randomized weights.

The 'jagged' nature of the amount of parallelism vs. the maximum attainable amount of parallelism observed per scenario is highly indicative of the influence of the randomized weights. It seems that the amount of parallelism is reduced due to the amount of non-uniformity of the weights distribution.

The measured and predicted parallelism, and resulting approximation error can be found in table 9.7 in the appendix . We observe that the largest approximation error is $\approx -3.011\%$, which is observed for scenario 212 (i.e. a

PDES graph of 24 nodes). This is an outlier as all the other measurements are within a 2% range. This is better indicated by figures 8.17 and 8.18

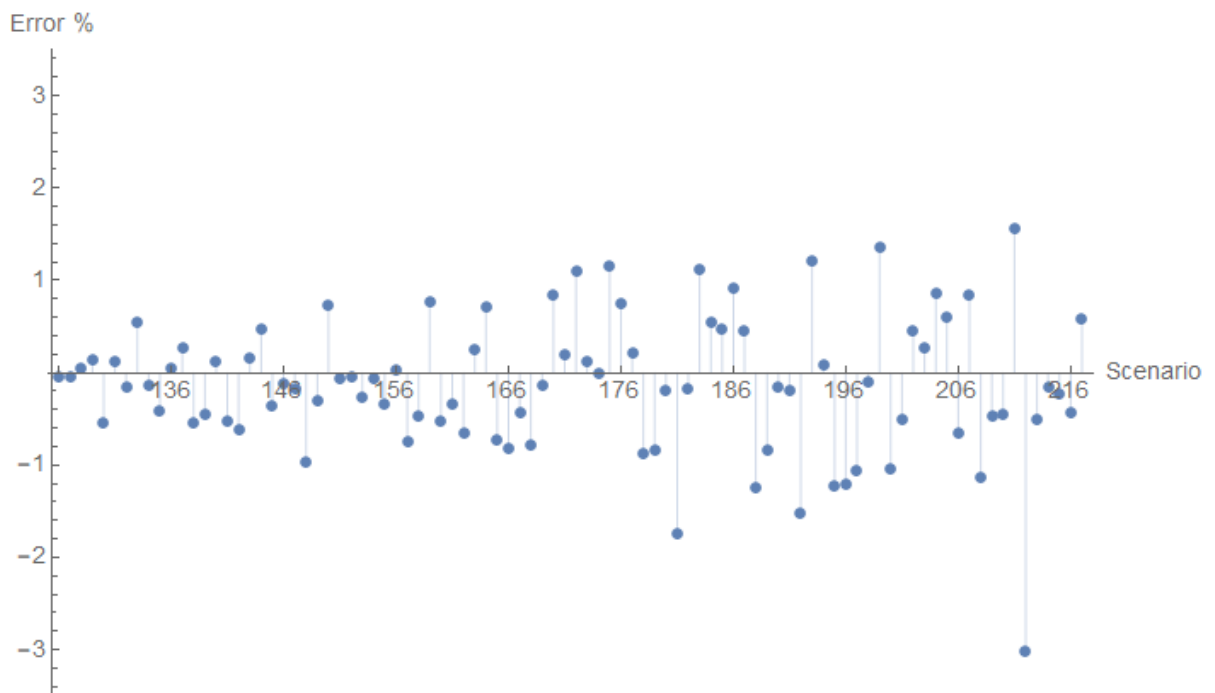


Figure 8.17: Approximation errors for complete scenarios with randomized weights

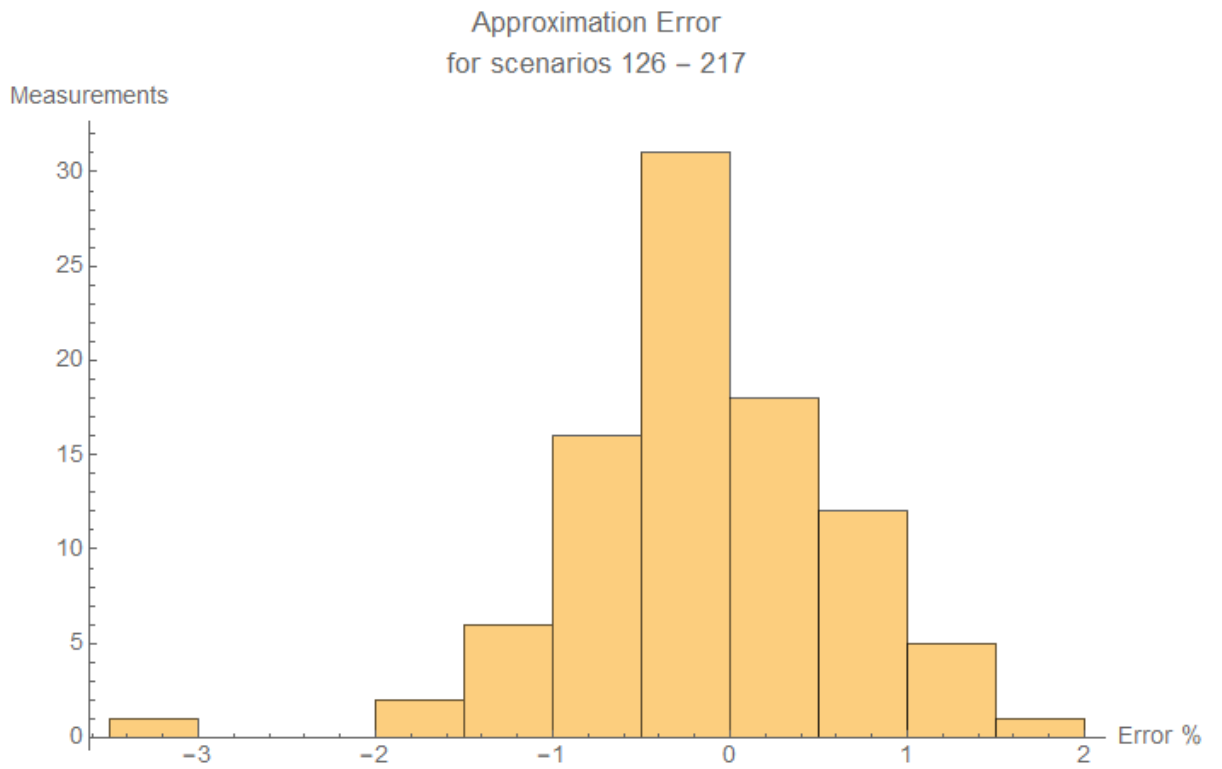


Figure 8.18: A histogram of the observed approximation errors for complete scenarios with randomized weights

8.2.2 Conclusion

Our experiments clearly indicate that the amount of parallelism is highly affected by the weights in the EPHOLD algorithm, causing load imbalances in the entire PDES graph. We've tested the standard PHOLD workload configurations, and compared these to randomized weights, and highly imbalanced weights (odd scenarios). These results are summarized in Figure 8.19.

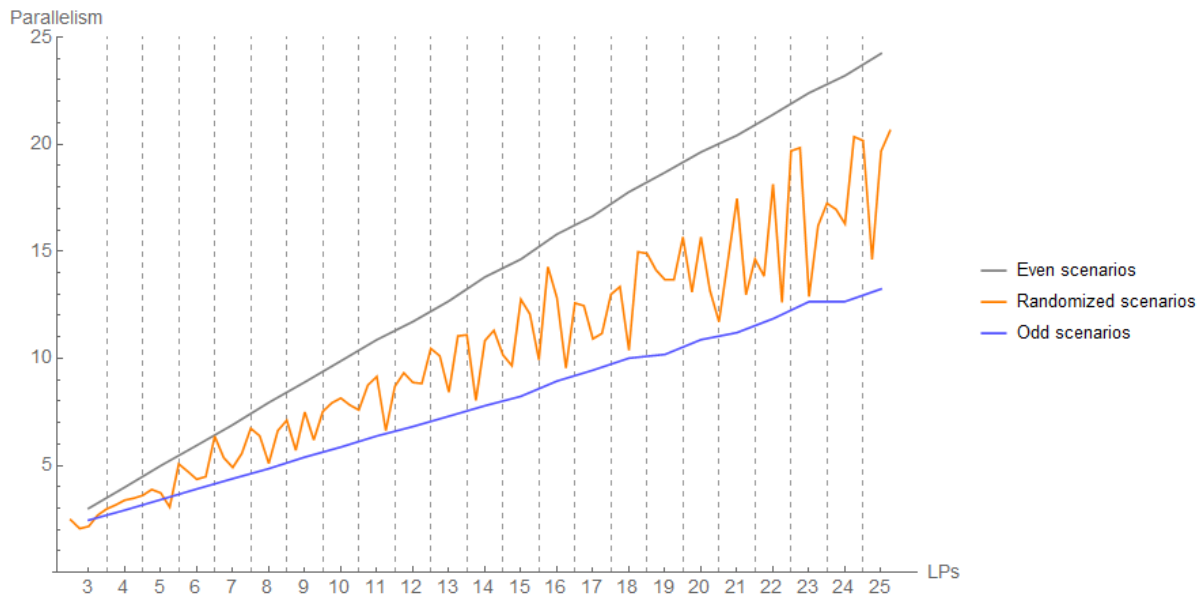


Figure 8.19: Amount of observed parallelism for all types of evaluated scenarios

PHOLD clearly paints the best picture possible for PDES simulations, our weight definitions of the odd scenarios clearly influenced the amount of parallelism more than the randomized scenarios do. Finally, we saw that the approximation algorithm presented in Chapter 6 is able to predict the amount of parallelism within reasonably close margins.

8.2.3 Parallelism in scale free graphs

We proceed to analyze the parallelism in Scale Free graphs similar to how we did for complete graphs. Figure 8.20 shows the average parallelism for even numbered scenarios. We can already observe:

1. The predicted and observed amount of parallelism is relatively low compared to the complete graphs
2. The approximation errors seem to be small.
3. Parallelism varies between different graphs with the same λ value.

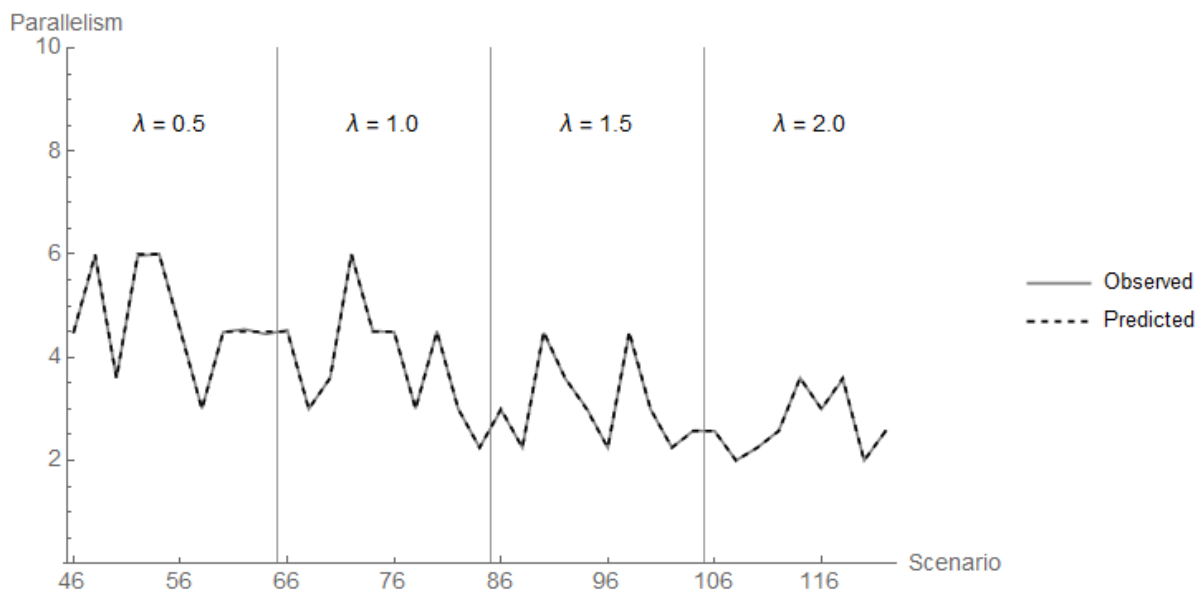


Figure 8.20: Predicted vs. Observed amount of parallelism for even numbered Scale Free scenarios.

Similar observations can be made for the odd scenarios (Figure 8.21). A relatively low amount of parallelism and a seemingly good approximation.

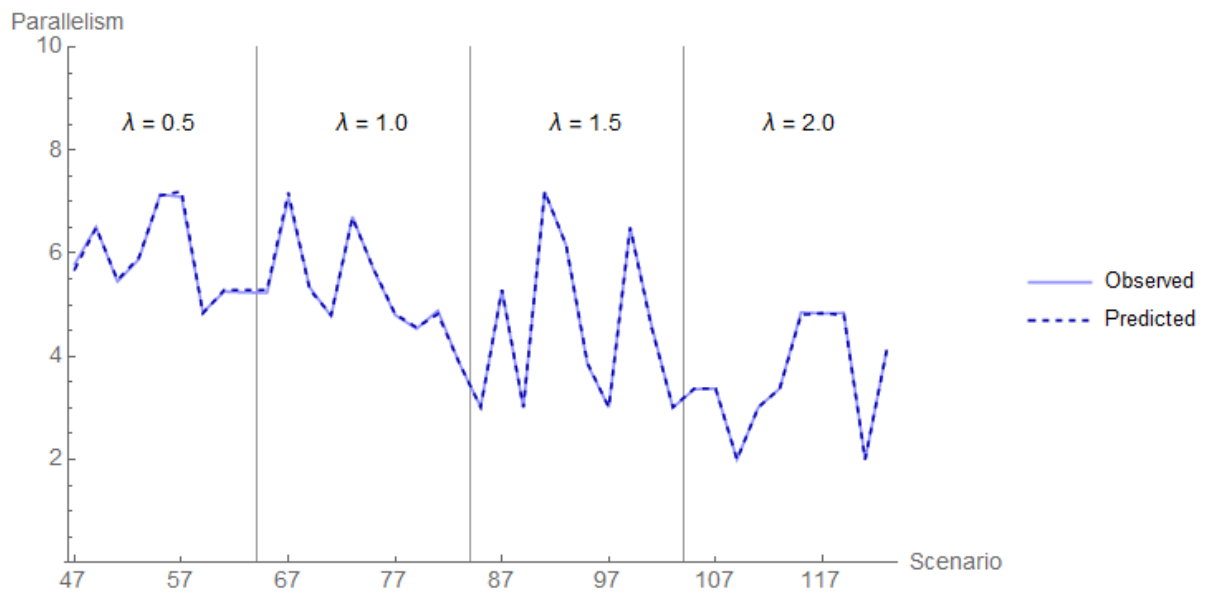


Figure 8.21: Predicted vs. Observed amount of parallelism for odd numbered Scale Free scenarios.

Figure 8.22 plots these parallelism statistics for even and odd numbered scenarios in one single plot. Here we can clearly see that the odd numbered scenarios increase parallelism for the majority of the scenarios compared to their even numbered counterparts. We also clearly see a decline in the amount of parallelism as the value of λ grows.

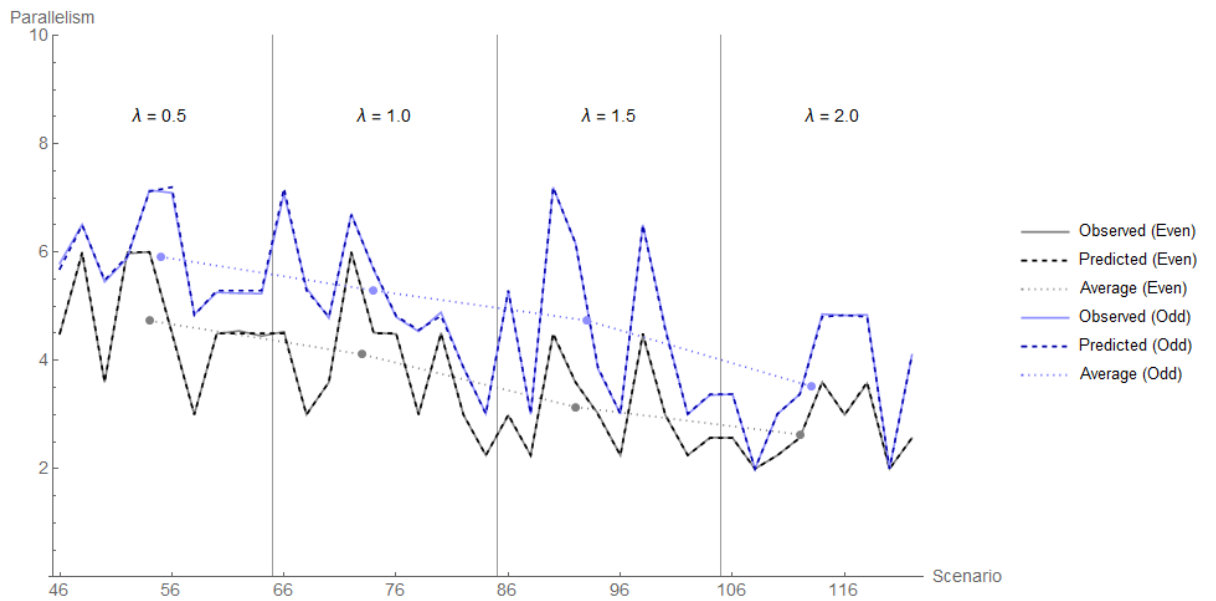


Figure 8.22: Predicted and Observed parallelism, for both Even and Odd scenarios. Note: Any odd scenario id i is plotted at horizontal coordinate $i - 1$. The dotted averages are the average parallelism of all scenarios that share the same value of λ

For completeness, figures 8.23, and 8.15 Showcase the approximation errors for respectively even, and odd scenarios. Tables 9.9 and 9.10, in the Appendix, contains all mentioned statistics in numerical form.

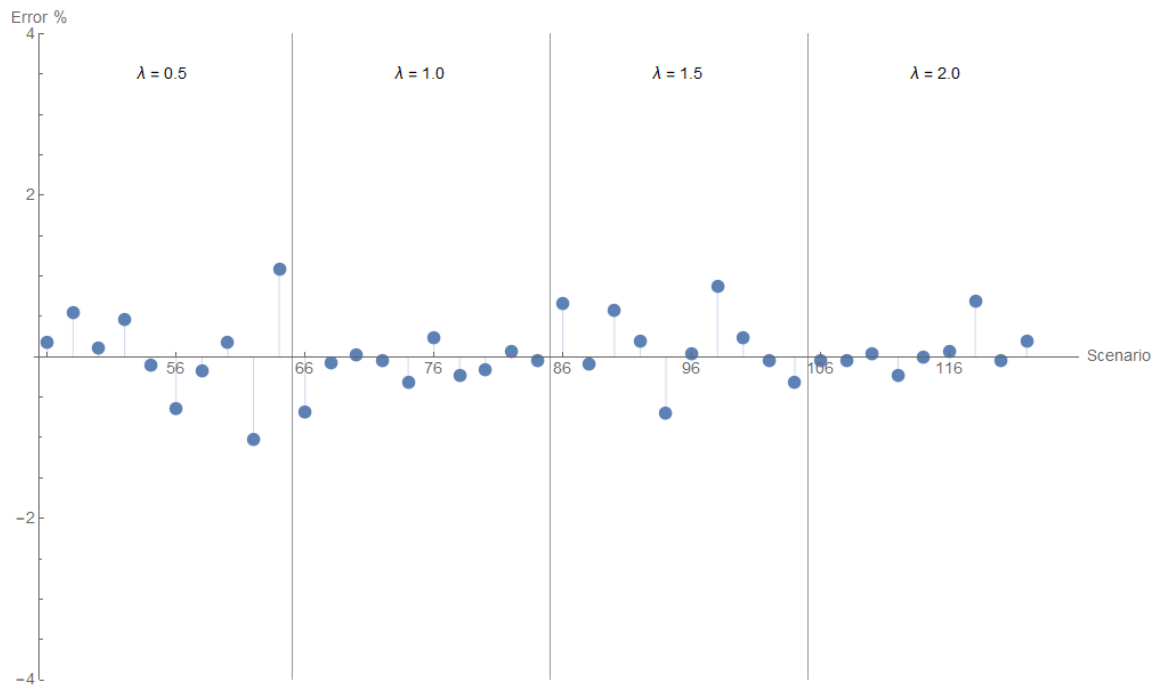


Figure 8.23: Parallelism approximation error for even numbered Scale Free scenarios

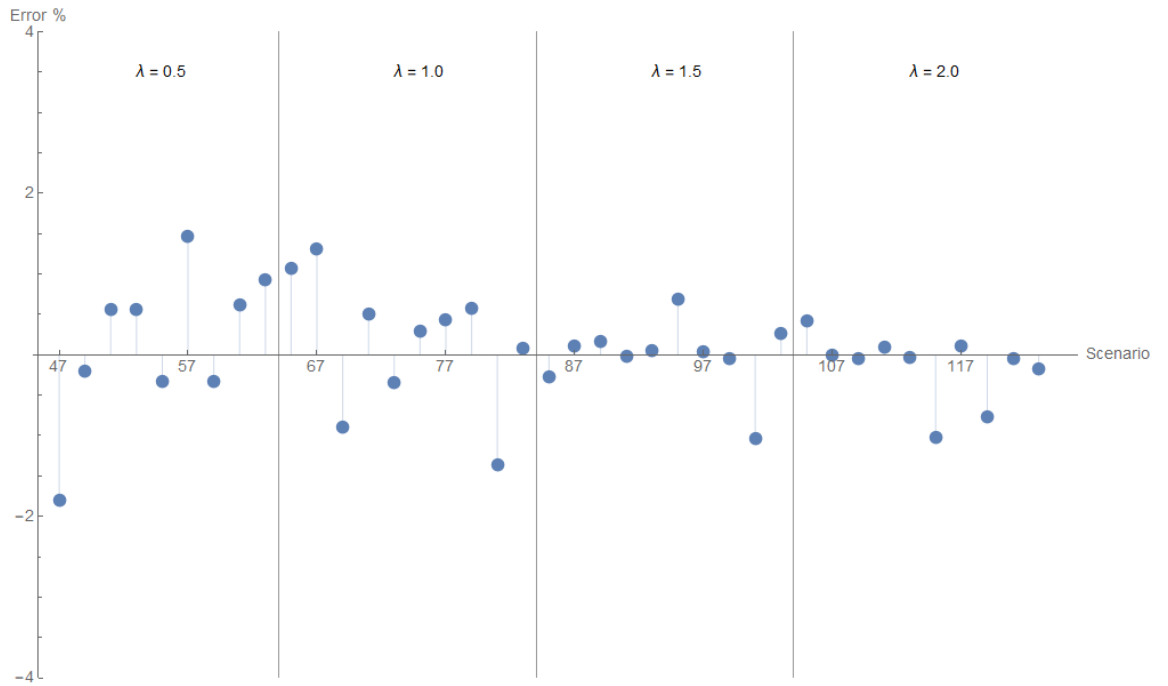


Figure 8.24: Parallelism approximation error for odd numbered Scale Free scenarios

The highest absolute error is obtained for scenario 47, with a value of -1.793%, we feel this is within reasonable bounds, and as such warrants no extra investigation. It is highly likely (due to what we found during our experiments for complete graphs), that increasing the termination time of the simulation will reduce the approximation error even more.

Randomized weights

To further investigate the effects of message flow on the amount of parallelism, we now run 4 scenarios per scale free graph, where all weights are randomized. Each scenario consists of 30 experiments. The results are summarized in table 9.8 in the appendix, and Figure 8.25

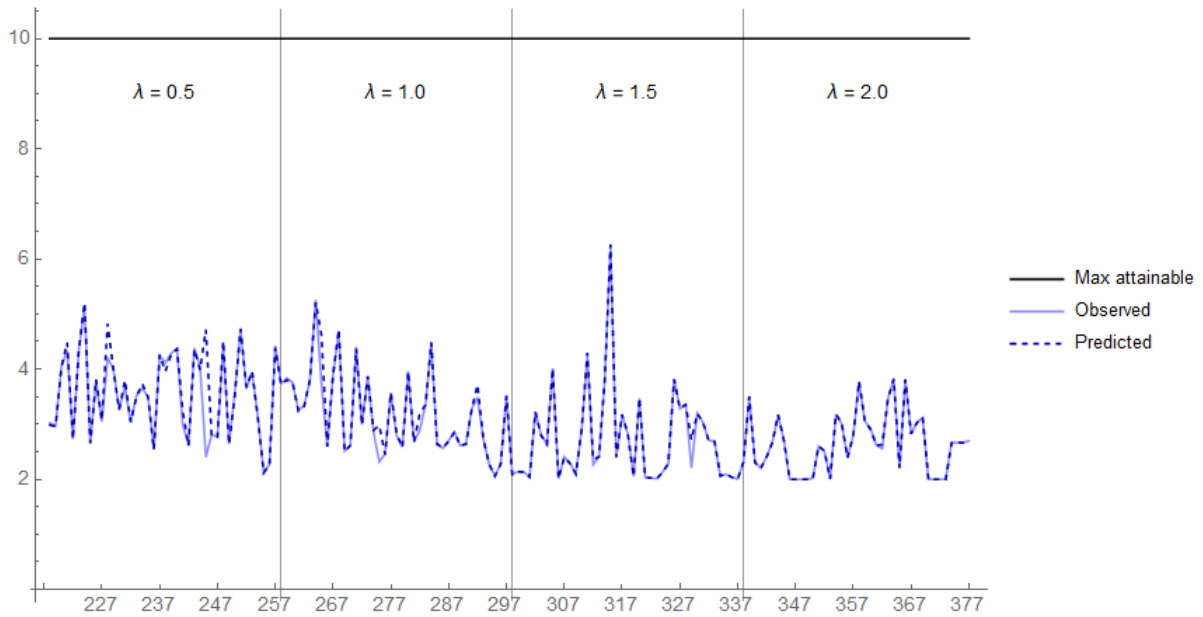


Figure 8.25: Predicted and observed amount of parallelism for Scale Free graphs, where all weights are randomized

Here we see several large approximation errors, the highest of which equals $\approx 49\%$.

Scenario Id	Predicted	Observed	Error %
228	4.826	4.209	12.785%
245	4.717	2.404	49.035%
265	4.611	3.735	18.998%
275	2.979	2.319	22.155%
329	2.716	2.205	18.814%

Table 8.5: Scenarios that have a larger absolute approximation error than 10%

We hypothesize that this is because we haven't run the simulation for a long enough time. We thus increase the termination time tenfold and rerun these particular scenarios (table 8.5). The results are summarized in table 8.6

Scenario Id	Predicted	Observed	Error %
228	4.826	4.186	13.262%
245	4.717	2.063	56.265%
265	4.611	3.633	21.21%
275	2.979	2.24	24.807%
329	2.716	2.151	20.803%

Table 8.6: Predicted vs. Observed amount of parallelism for the scenarios that ran for a longer time (ten times longer)

These results are even worse, and thus don't share the same root cause as in the previous section. It turns out that our approximation algorithm did not exit with a stable solution, but reached the maximum allowed amount of iterations M_c , set at 100. We increased the value of M_c to 100 000, and reran approximation algorithm for the above mentioned scenarios. The results are summarized in table 8.7.

Scenario Id	Predicted	Observed	Error %
228	4.151	4.186	-0.843%
245	2.031	2.063	-1.576%
265	3.659	3.633	0.711%
275	2.228	2.24	-0.539%
329	2.152	2.151	0.046%

Table 8.7: Predicted vs. Observed amount of parallelism for the scenarios that ran for a longer time (ten times longer), and where $M_c = 100000$

We can clearly see a greatly reduced approximation error. The largest deviation now equals 1.576%, instead of $\approx 56\%$. We observe a similar effect for scenario 241, i.e. the only unmentioned scenario with an approximation error of more than 5%, going from an approximation error of $\approx 7\%$, down to -0.268% . We did not rerun the other scenarios because we believe the approximation error to be small enough. (i.e. $\leq 5\%$, where the majority is less than 2%)

8.2.4 Conclusion

For Scale Free graphs, we summarize the average amount of parallelism per value of λ in Figure 8.26.

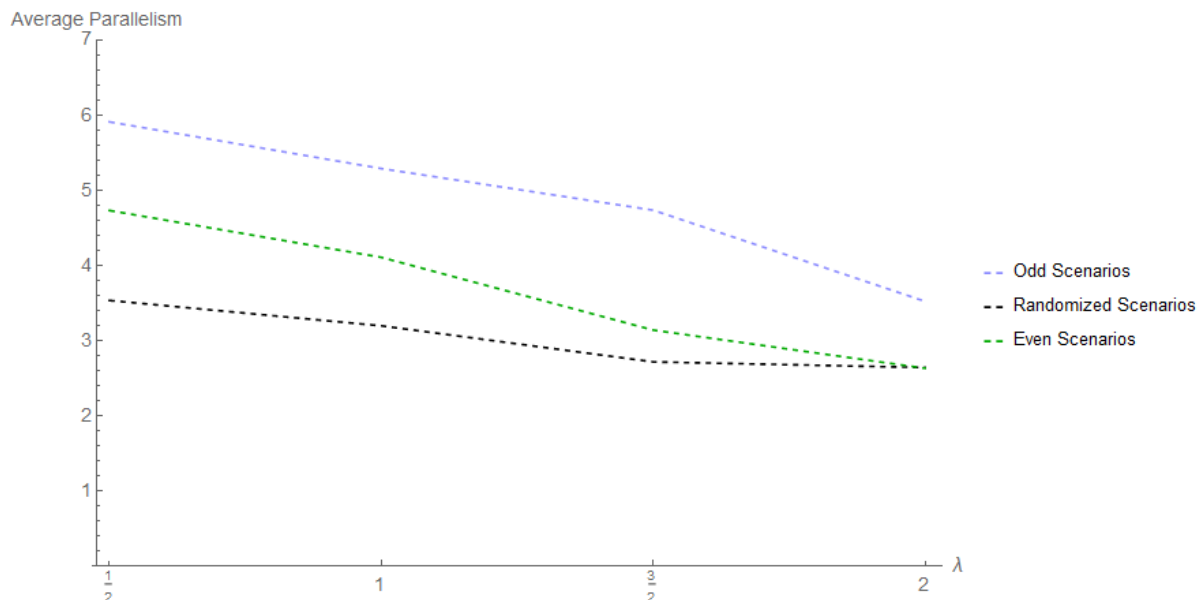


Figure 8.26: Average amount of parallelism for all scenarios per value of λ

We observe a clear decreasing trend in the amount of parallelism as the value of λ increases. This effect seems to be caused solely by the topology. However, the amount of parallelism can be increased by redirecting event messages toward the leaves of the graphs. The randomized scenarios seem to have a negative impact on scale free graphs. Finally, the approximation algorithm presented in Chapter 6 succeeds in predicting the amount of parallelism in complete graphs, and 10 node scale free graphs, within reasonable margins.

8.3 Chandy-Misra-Bryant

We've altered the CMB algorithm such that, Instead of sending a NULL message every time an event is executed, an LP first executes all events it can. It only sends a NULL message once it has processed all events that are safe to process, and has hit its Lower Bound on the Timestamp (LBTS). This alteration reduces the total amount of NULL messages sent.

Similar to the YAWNS experiments, we perform experiments on Complete graphs (Section 8.3.1), and Scale Free Graphs (Section 8.3.2).

8.3.1 Complete graphs

Again, we separate the even from the odd numbered scenarios. We plot the amount of parallelism for the even scenarios together with the odd scenarios in Figure 8.27. Note that the odd scenario is plotted in the same horizontal coordinate as its even numbered counterpart. This way we can easily see the differences in parallelism between the two configurations.

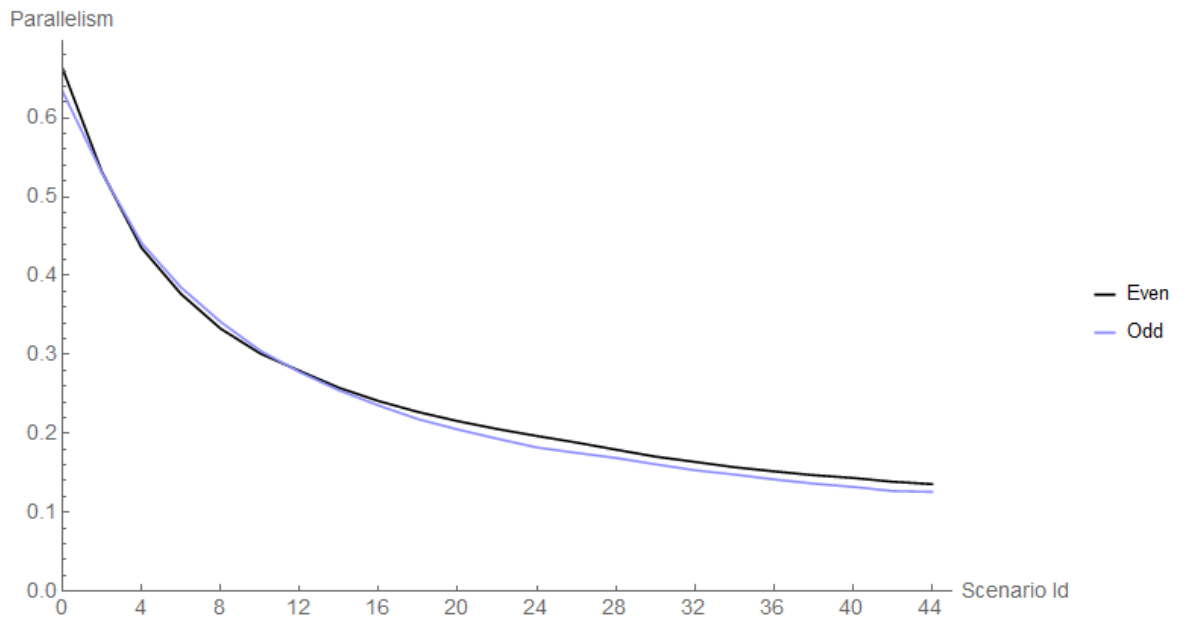


Figure 8.27: Parallelism for complete PDES graphs for the CMB algorithm. The numerical values of this graph can be found in table 9.11 in the Appendix.

We observe that:

1. Parallelism *decreases* when the size of the PDES graph grows (according to definition 5.5).

2. In general, the trend seems to be that the odd numbered scenarios perform worse for larger graphs.

Observation 1 can be explained as follows: Let us first assume that we are dealing with the normal CMB algorithm (The same argument can be made for our version). Once an LP executes an event, it has to inform all its neighboring LPs. For complete graphs, these are all other nodes in the graph. Thus, for a complete PDES graph of n nodes, *every* event execution causes $n - 1$ NULL messages to be sent. Increasing the size of the graph, thus cause more NULL messages to be sent per event execution. The (slightly) different amount of parallelism for the odd numbered scenarios can be explained by the alteration we made to the standard CMB algorithm, the Lookahead value L , the timestamp increment function, and the definition of the EPHOLD weights in the odd numbered scenarios. Each scenario has a fixed Lookahead value of $L = 1$, and timestamp increments drawn from an $Exp(1)$ distribution. Our alteration exploits the fact that all events before the LBTS are safe to executes, and reduces the amount of NULL messages sent, especially when the amount of 'safe' events to be processed is large per LP. However, we have a relatively small lookahead value (compared to the timestamp increment function used for all LPs). Thus the amount of safe events to process is relatively small, reducing the efficacy of our altered algorithm. Note, that Null messages are emitted when the LBTS is increased. This can also happen when an LP has no events to process, thus explaining the increased amount of NULL messages for larger PDES graphs.

Randomized weights

We've repeated the experiments on complete graphs using randomized weights. 4 additional scenarios were run per complete graph. This time we randomized the weights. The results are summarized by figure 8.28, and table 9.12 in the appendix.

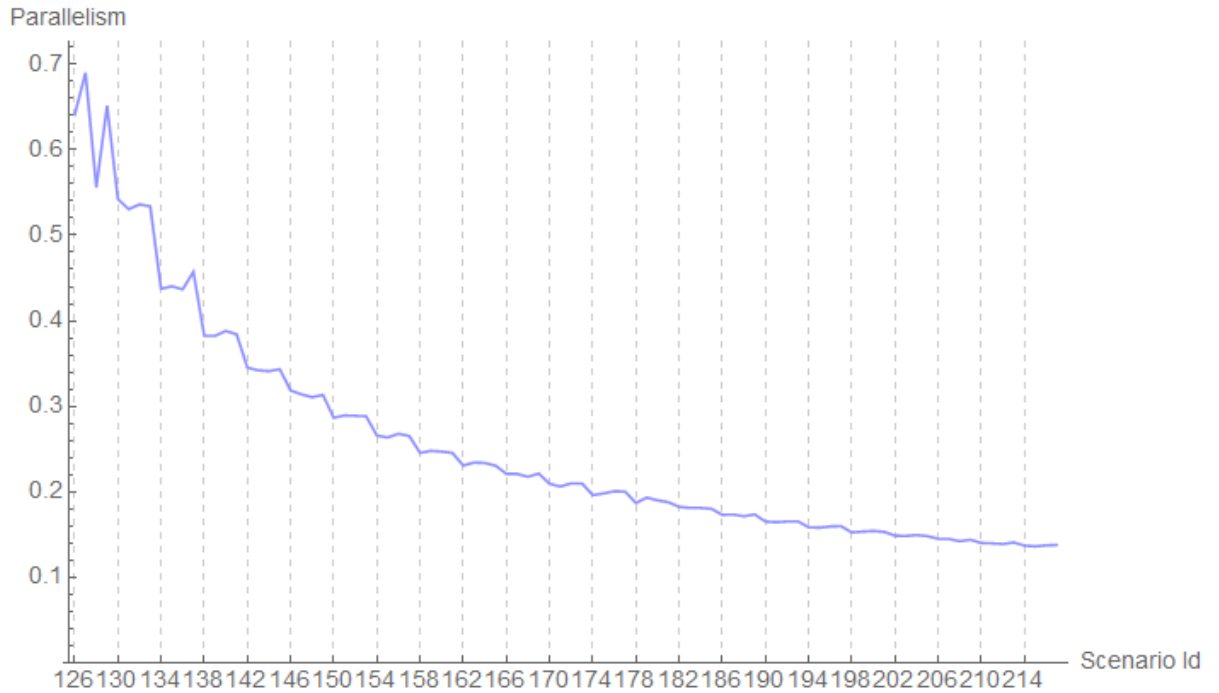


Figure 8.28: Amount of parallelism in CMB for complete graphs and randomized weights

We observe a similar trend as before. The amount of parallelism decreases as the size of the PDES graph grows.

8.3.2 Scale free graphs

We repeat the experiments for scale free graphs using the CMB synchronization algorithm. The results are charted in figure 8.29.

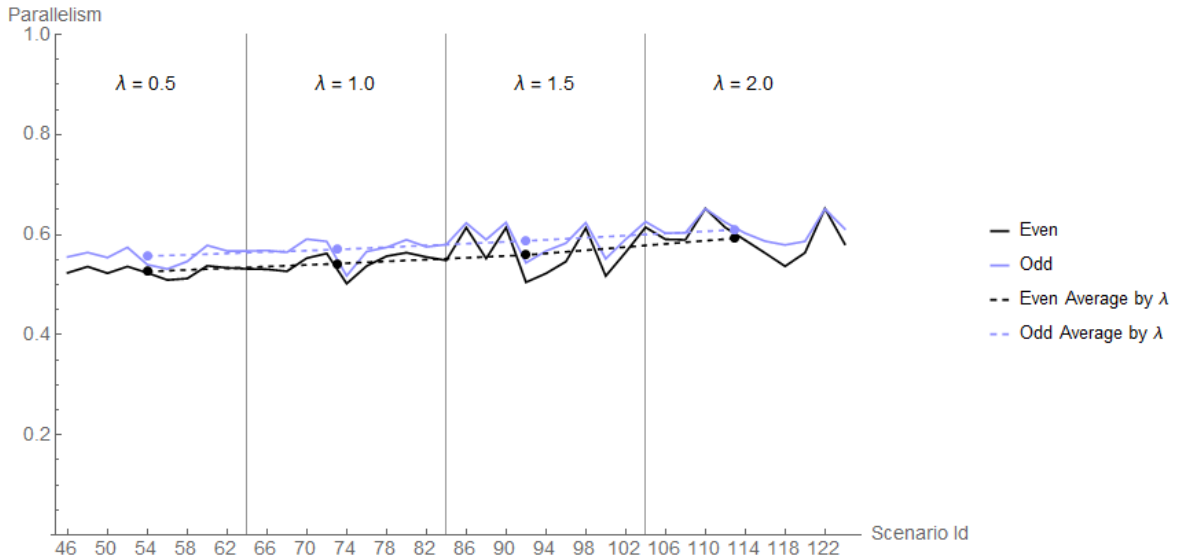


Figure 8.29: The amount of parallelism measured for the CMB algorithm on Scale Free graphs

We do not observe much change in parallelism when the value of λ varies. We can clearly see that the odd numbered scenarios perform at least as well as their even numbered counterparts (i.e. with the same graph). The small change that we measured seems to indicate an increasing levels of parallelism for higher values of λ . We also observe that the amount of parallelism seems to behave more stable for scale free graphs with $\lambda = 0.5$ when compared with e.g. $\lambda = 1.5$. For completeness we've included the numerical results in tables 9.14 and 9.15 in the Appendix. Although we test only a small subset of all possible Scale Free graphs, for the even scenarios, we can see that the increase of parallelism (i.e. the slope of the parallelism) increases from $\approx 3\%$ (from $\lambda = 0.5$ to $\lambda = 1.0$) to $\approx 6\%$ (from $\lambda = 1.5$ to $\lambda = 2.0$), hinting at an (exponential) increase in parallelism as λ grows. However, more research is needed to establish this relation.

Randomized weights

To further investigate the effects of message flow on the amount of parallelism in CMB, we now run 4 new scenarios per scale free graph, where all weights

are randomized. Each scenario consists of 30 experiments. The results are summarized in table 9.13 in the appendix, and Figure 8.30

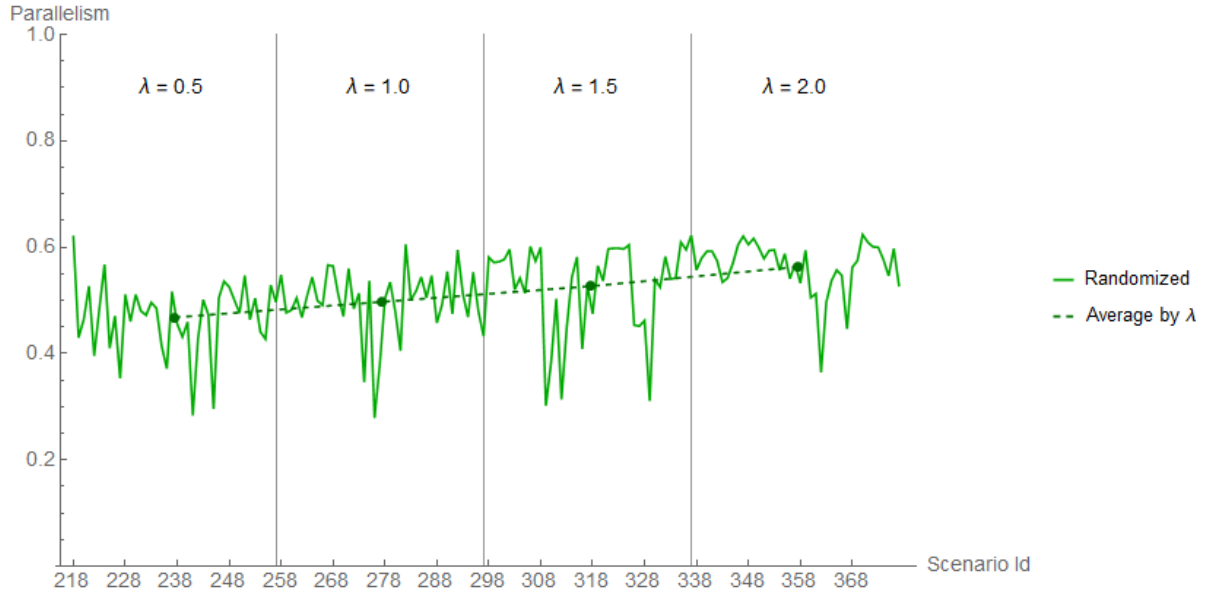


Figure 8.30: The amount of parallelism measured for the CMB algorithm on Scale Free graphs with randomized weights

We can distinguish an increasing trend in the amount of parallelism, as the value of λ increases. These increases are table 8.8

λ	Parallelism	Increase
0.5	0.4680	-
1.	0.4973	6.26 %
1.5	0.5266	5.89 %
2.	0.5630	6.91 %

Table 8.8: CMB Mean parallelism statistics for scenarios with Scale Free graphs and randomized weights, aggregated by values of λ . The increase in parallelism shown is with regard to the previous value of λ

If we compare the results of table 8.8 to the results of table 9.15, we can clearly see that the randomized weights have a negative effect on the amount of parallelism. However, the increase per λ does seem to be higher in the randomized case. These observations are summarized in figure 8.31

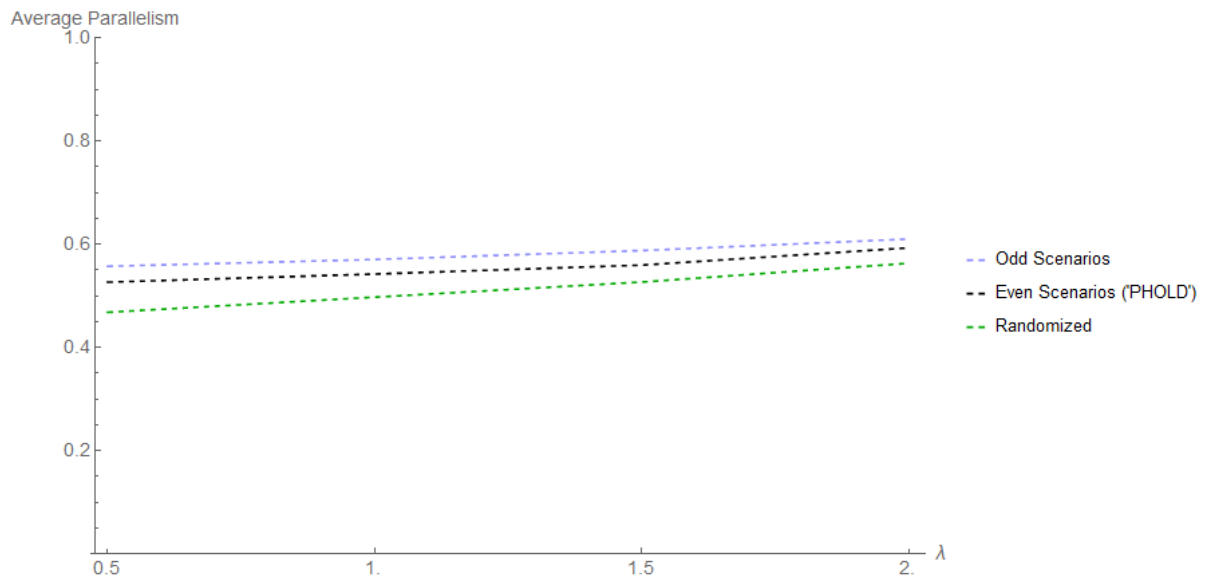


Figure 8.31: The average parallelism of all scenarios, aggregated by λ

8.3.3 Conclusion

We've used a definition of parallelism that has been used in other CMB parallelism research and measured the parallelism of complete graphs and several Scale Free graphs. The parallelism for complete graphs decreases as the size of the PDES graph grows, with a seemingly negative exponential trend. Altering the message flow such that the LPs receive a non-uniform workload in these scenarios, seems to have a negative impact on the overall parallelism. For Scale Free Graphs we see that the average amount of parallelism increases by a modest amount if we send more messages to leaf nodes, as opposed to hub nodes. Finally, we observed hints of a (exponential) growth in parallelism as the value of λ increases. However, more research is needed to establish this relation.

8.4 Time Warp

We re-run all scenarios for Time Warp, so that we can investigate the effect of topology, and message flow, on the amount of parallelism in the Time Warp synchronization algorithm. In Section 5.3 we argued that a rollback can be considered overhead introduced by the Time Warp algorithm. Our implementation tracks rollbacks by assigning an Id to a rollback, that is (among other statistics) included in the output data. This Rollback Id is built up from the LP Id that started the rollback, and a rollback counter. The last two digits of the id reveals the originator of the rollback. E.g. the 3rd rollback originating from LP_{12} would have Id: 312. This way, we can see if there are LPs that cause abnormal amount of rollbacks.

As mentioned in Section 5.3 we make a distinction between two types of rollbacks.

1. Idle rollbacks, i.e. When an LP was done processing events (and thus was idling), and has to roll back from simulation time ∞ .
2. Busy rollbacks, i.e. When an LP was still processing events, and has to roll back.

8.4.1 Complete graphs

Again we re-run the complete scenarios. The execution time per simulation run increased substantially compared to the other algorithms. Because of this, we reduced the amount of experiments performed per scenario from 100 to 30. Although this increases the variance around the presented averages, we argue that this is enough to be able to distinguish trends. Figures 8.32, 8.33 and 8.34 show the average amount of all, busy, and Idle rollbacks per scenario respectively. The numerical results are shown in table 9.16, 9.17 and 9.18 in the appendix.

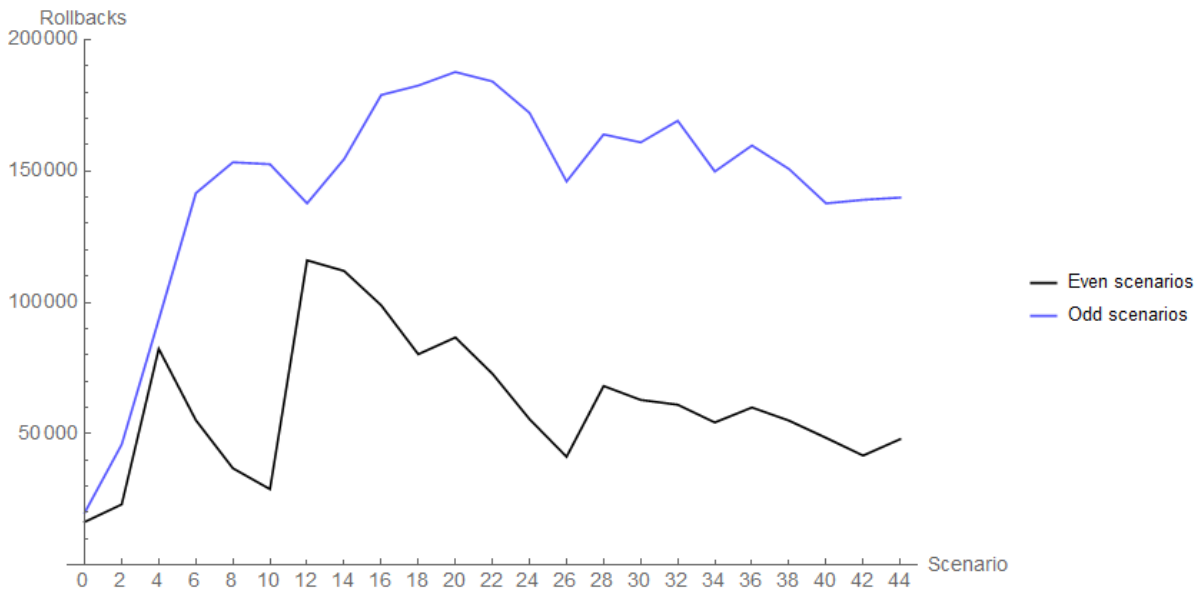


Figure 8.32: The average amount of rollbacks per scenario for complete graphs

After an initial rapid increase, the total amount of rollbacks seem to stabilize somewhat around ≈ 150000 rollbacks for the odd scenarios. Similarly for the even scenarios, after an irregular period, the amount of rollbacks stabilize around ≈ 65000 . Here we can already see that odd scenarios exhibit more rollbacks compared to their even numbered counterparts.

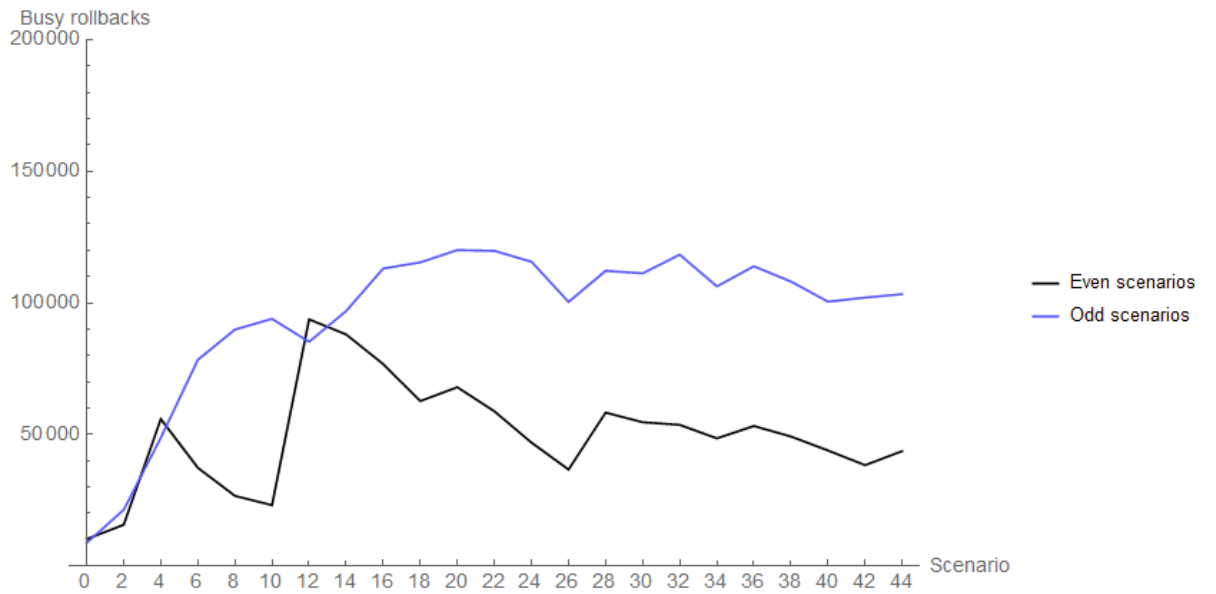


Figure 8.33: Average amount of busy rollbacks per scenario for complete graphs

Figure 8.33 shows an increase in the amount of busy rollbacks for odd scenarios when compared to their even numbered counterparts. We can also see that both amounts seem to stabilize around scenario 18.

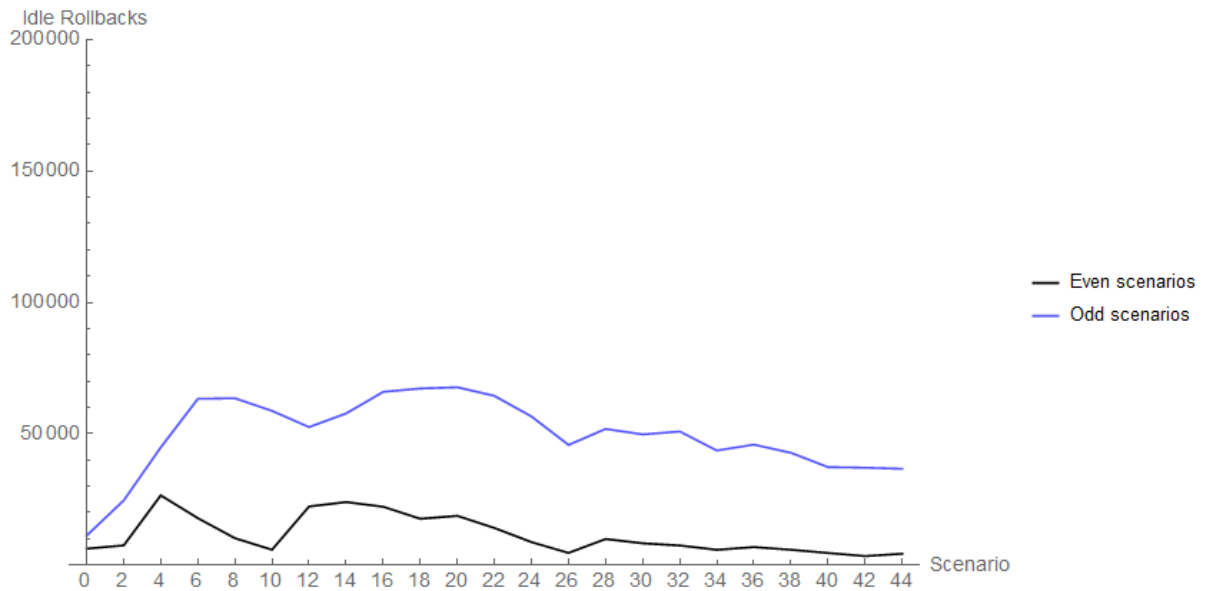


Figure 8.34: Average amount of idle rollbacks per scenario for complete graphs

Clearly, the amount of idle rollbacks is less than the amount of busy rollbacks, indicating that LPs are still processing events when they receive an out-of-order (anti-) message. This is also clearly highlighted by figure 8.35, which shows a proportional increase in busy rollbacks as the graph size increases. We hypothesize that this is mainly caused by the fact that our experimentation pc 'only' contains 4 hardware processing cores. We hypothesize that, as the amount of LPs increase, so does the amount of context switching, and amount of 'fighting' for resources between LPs assigned to the same hardware processing core. This is probably causing them to process events less quick than e.g. in the case of a complete 4-node PDES graph, thus resulting in an increased amount of busy rollbacks.

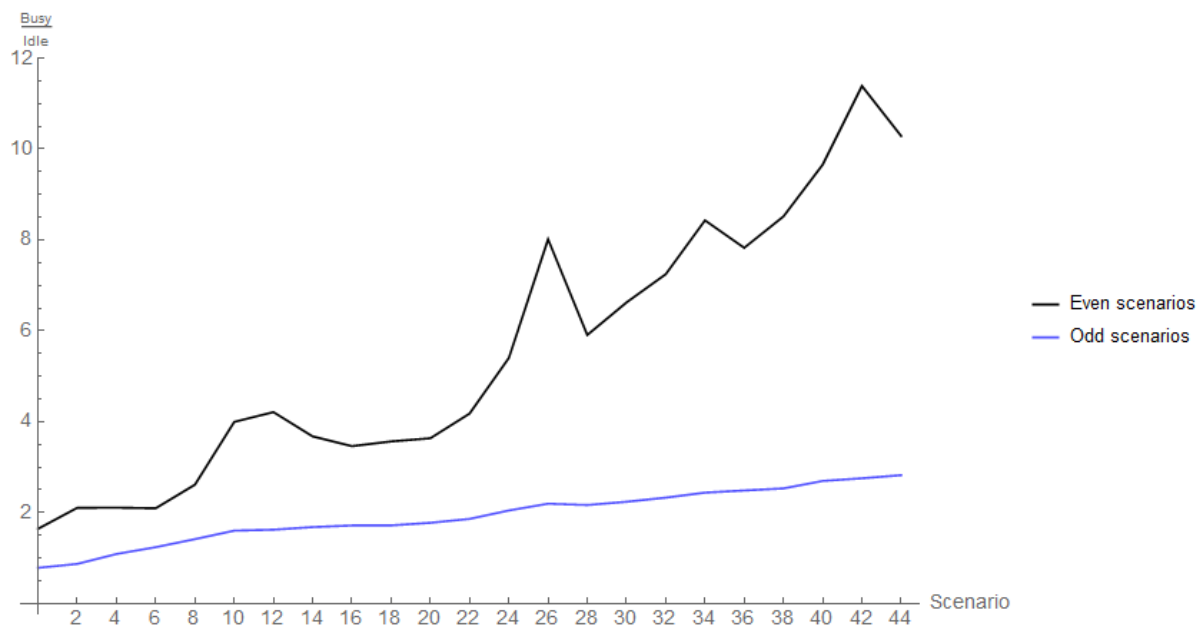


Figure 8.35: Ratio of busy and idle rollbacks for complete graphs

Randomized weights

Again, we've ran additional experiments on the complete graphs. 4 additional scenarios were run per complete graph where we randomized the weights. The results are summarized by tables 9.19 (total amount of rollbacks), 9.21 (total amount of busy rollbacks), 9.20 (total amount of idle rollbacks) and 9.22 (fraction of busy and idle rollbacks) in the appendix.

In the following charts (i.e. figures 8.36, 8.37, 8.38 and 8.39) every section delimited by vertical dashed grid lines indicate results from scenarios executed on the same graph but with different weights. The amount of

nodes per graph is denoted in the result tables mentioned earlier. As an example, scenarios 126 through 130, are run on the same 3 node complete graph, 131 through 134, on the 4 node graph, etc.

First we will investigate the total amount of rollbacks (Figure 8.36 / table 9.19).

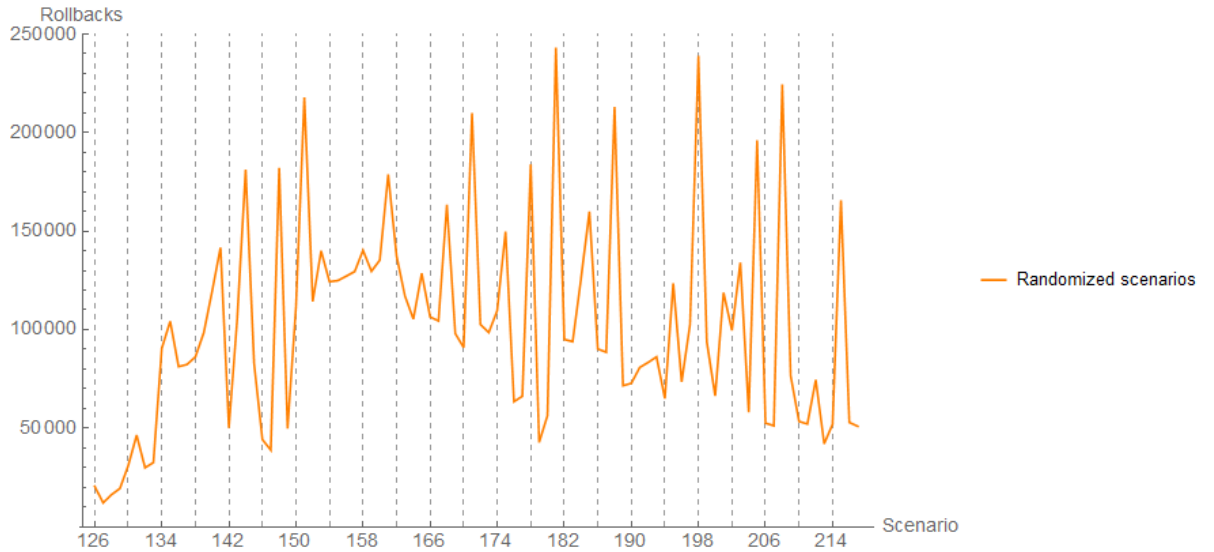


Figure 8.36: Total amount of rollbacks for complete graphs where the weights are randomized

The amount of rollbacks increase from scenario 126 until scenario 138. As of then, the amount of rollbacks behaves highly irregular, with a seemingly stabilizing, or downward trend. It seems that the way weights are distributed across LPs has a very high impact on the amount of rollbacks in Time Warp. These fluctuations seem to become more extreme as the size of the PDES graph increases. This is probably due to the more saturated nature (i.e. there are more LPs than physical processing cores) of the experiments. We hypothesize the downward, or stabilizing amount of rollbacks is due to the following. As the amount of LPs increases, on a saturated system, LPs have a harder time executing events. Event messages that get scheduled on an LP thus have a higher probability of not causing a rollback. Following this hypothesis, we would expect a decreasing amount of Idle rollbacks as the the amount of LPs increases, since LPs will be less likely to have finished executing all their events at any point.

The next aspects of interest are the amount of busy, and idle rollbacks (Figures 8.37 and 8.38 respectively)

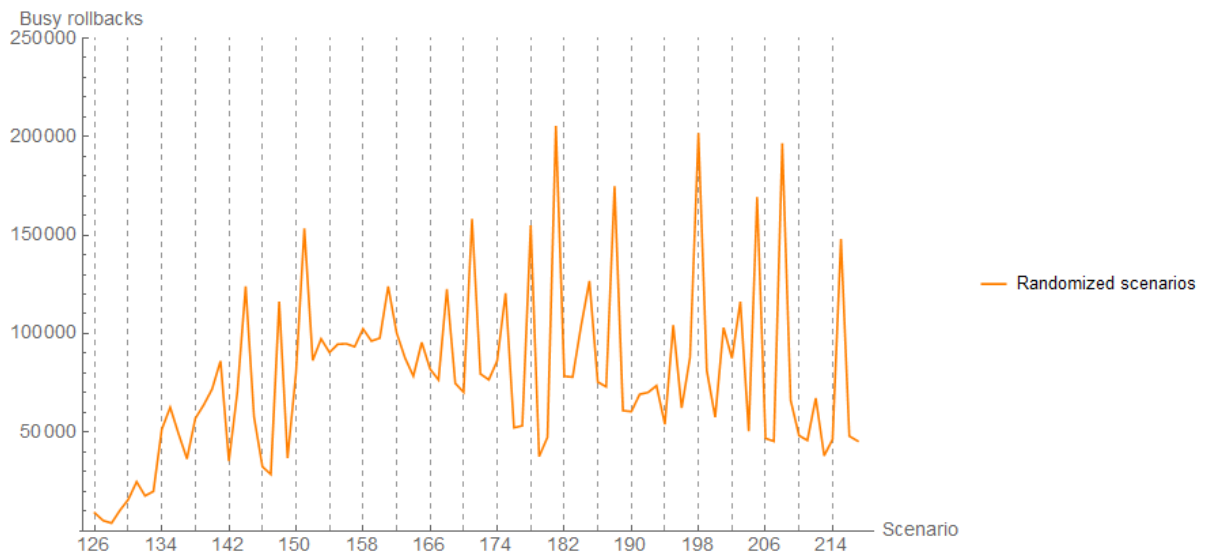


Figure 8.37: Total amount of busy rollbacks for complete graphs where the weights are randomized.

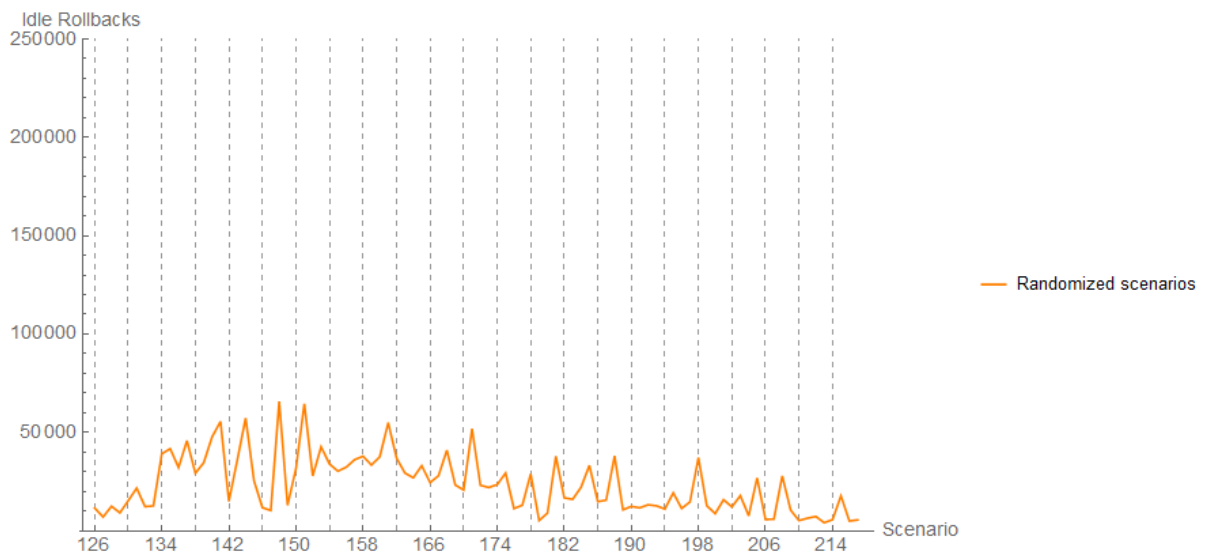


Figure 8.38: Total amount of idle rollbacks for complete graphs where the weights are randomized

Judging by the amount of busy and idle rollbacks, the total amount of rollbacks seem to be largely comprised of busy rollbacks as the size of the PDES graph grows. As we hypothesized, we can see a decreasing amount

of idle rollbacks as the amount of LPs increases, especially compared to the amount of busy rollbacks.

This observation is better illustrated by Figure 8.39.

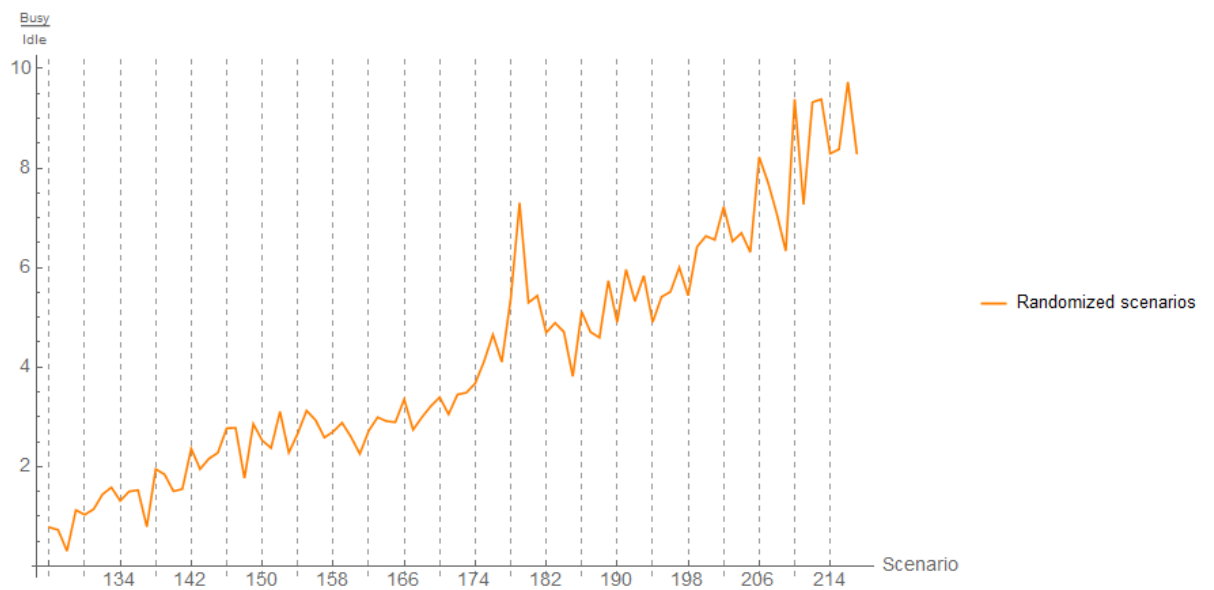


Figure 8.39: Fraction of the total amount of busy and idle rollbacks for complete graphs where the weights are randomized

Figure 8.39 clearly indicates an increasing pattern in the amount of busy rollbacks compared to the amount of idle rollbacks as the size of the PDES graph increases. This indicates that LPs are more often still busy processing events when a rollback occurs. The different weights clearly have an effect on the busy / idle rollback fraction. Finally, in Figure 8.40, we relate these results to the amount of LPs, and compare them to the results from the previous section.

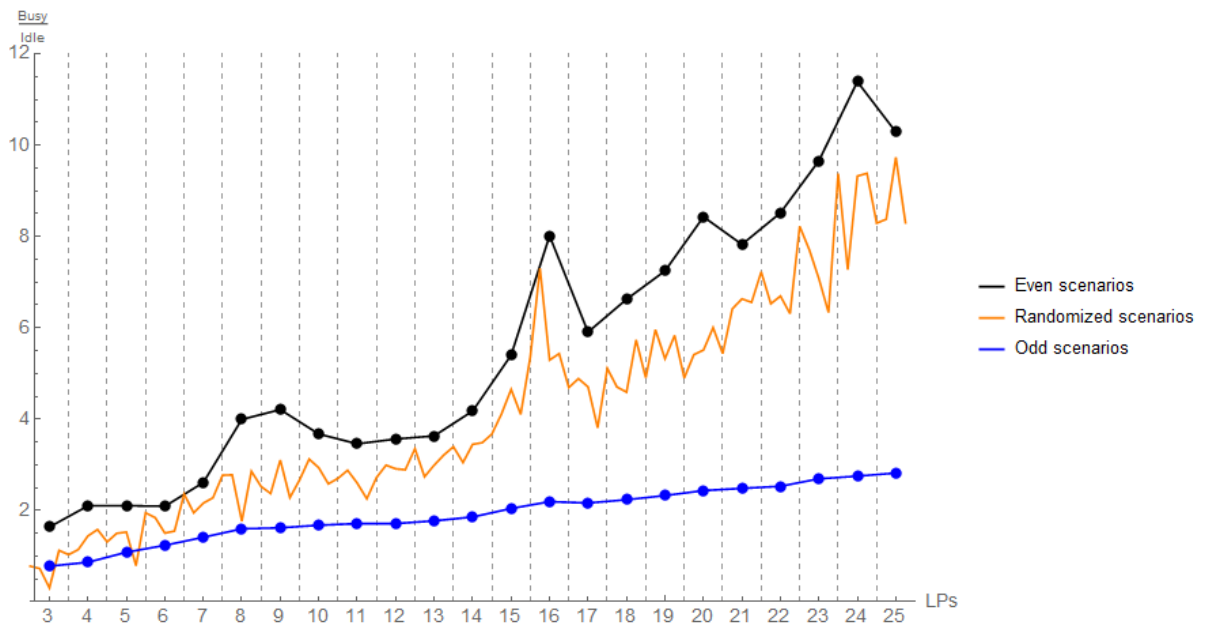


Figure 8.40: Fraction of the total amount of busy and idle rollbacks for complete graphs, by amount of LPs

Here we can summarize that the PHOLD (even) scenarios give the highest amount of busy / idle rollbacks ratio. The odd scenarios perform the worst (with two exceptions), and the randomized weight scenarios are in between. We can also see that whenever the amount of LPs is a multiple of 4, the fraction increases (most of the time) in the even scenarios. This could be caused by the fact that our experimentation laptop has 4 hardware cores.

8.4.2 Scale free graphs

Similar to the scenarios for the complete graphs, the scale free scenarios experiments also increased in execution time. Instead of doing 100 runs per scenario, we decided to do 30 per scenario. This will increase the variance of our results, and thus the representativity of the measured means. However, as a means to indicate a trend (which is what we are interested in), we argue that this is enough. Tables 9.27, 9.28 and 9.29, in the Appendix, show the numerical results for the average amount of all, busy, and idle rollbacks for the scale free scenarios. These numbers are charted in figures 8.41, 8.42 and 8.43.

```
ScaleFreePlot [ { { { 124,  $\frac{19\,364\,066}{305}$  } } }, { { { 124,  $\frac{997\,457}{31}$  } } }, Rollbacks ]
```

Figure 8.41: Total amount of rollbacks per scenario for scale free PDES graphs

We can clearly see that the amount of overhead (i.e. amount of rollbacks) are reduced by the weight definitions in the odd numbered scenarios, in almost all cases. The increases in amount, between different graphs, also seem to be less extreme for odd numbered scenarios. Finally, we observe that there seems to be a slightly decreasing trend in the amount of rollbacks, for both types of scenarios, for increasing values of λ . This hints at a slightly increased amount of parallelism as the amount of overhead, introduced by Time Warp, decreases.

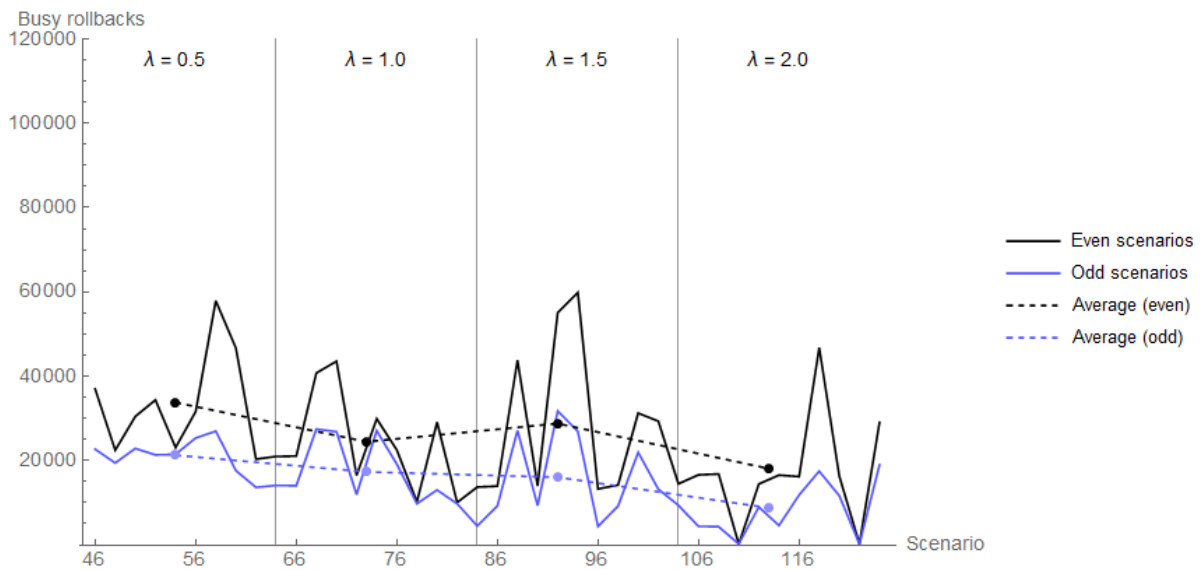


Figure 8.42: Average amount of busy rollbacks per scenario, for scale free PDES graphs

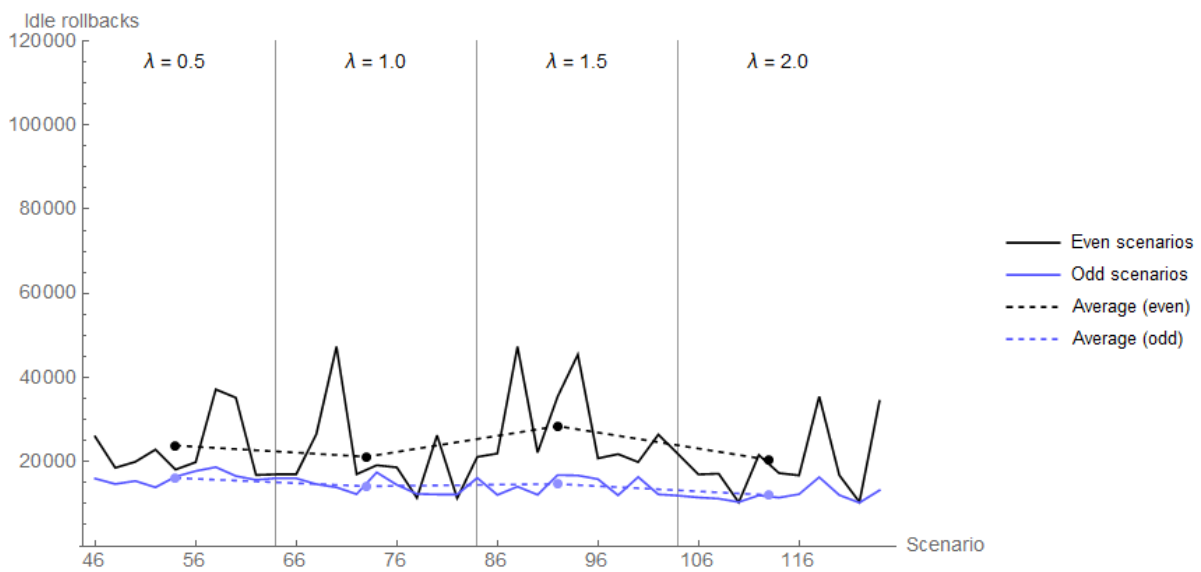


Figure 8.43: Average amount of idle rollbacks per scenario for, scale free PDES graphs

Figure 8.43 shows a relatively low, and steady amount of Idle rollbacks across all scenarios. If we combine this observation with the slightly decreasing amount of busy rollbacks, we expect to see a decreasing trend in the ratio

between the busy vs. idle rollback ratio, which is confirmed by figure 8.44.

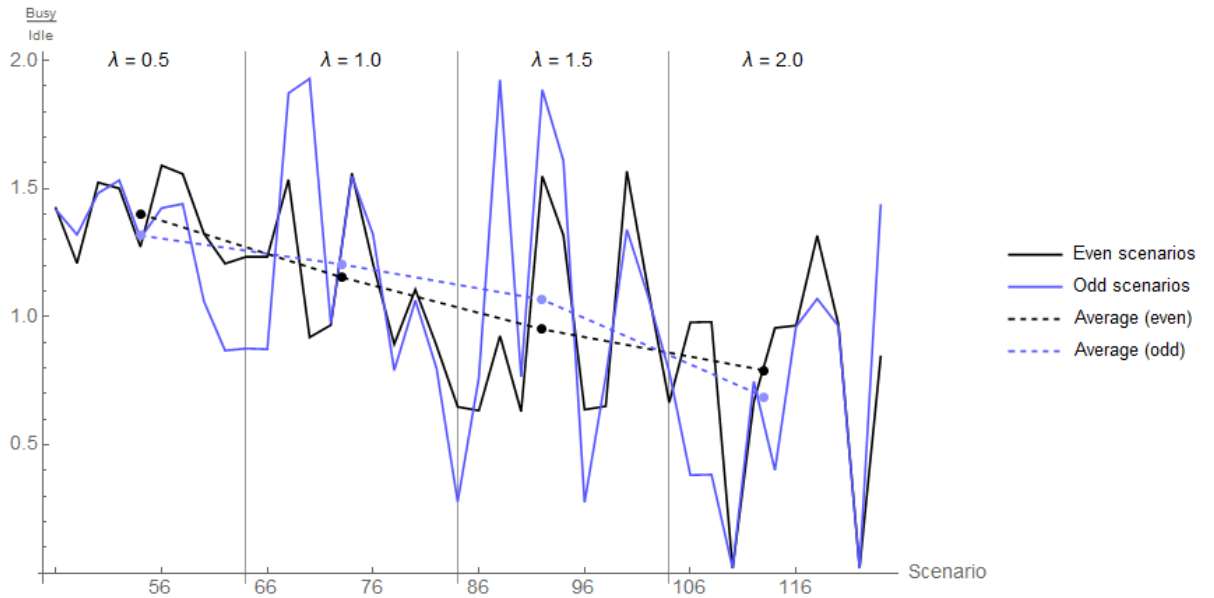


Figure 8.44: Ratio between the average amount of busy and idle rollbacks per scenario, for scale free PDES graphs

This hints at a decline in the busy vs. idle rollback ratio as the value of λ increases. The effect of this ratio on the amount of parallelism, or execution time of a simulation remains to be investigated in future research.

Randomized weights

Again, we've ran additional experiments on the complete graphs. 4 additional scenarios were run per complete graph where we randomized the weights. The results are summarized by tables 9.23 (total amount of rollbacks), 9.24 (total amount of busy rollbacks), 9.25 (total amount of idle rollbacks) and 9.26 (fraction of busy and idle rollbacks) in the appendix. These results are visualized by figures 8.45, 8.46, 8.47 and 8.48 respectively

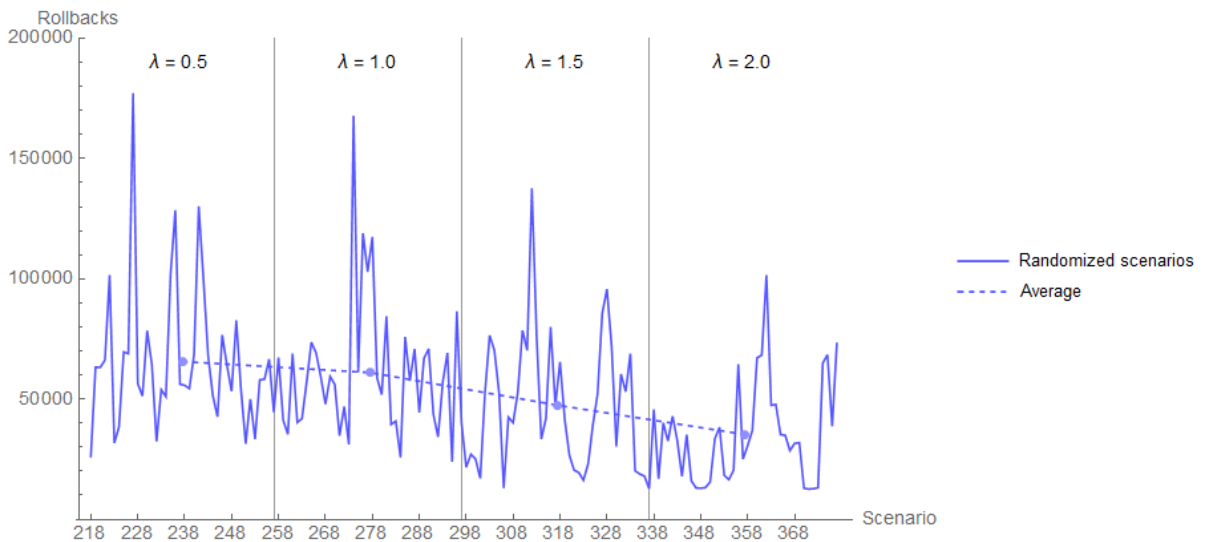


Figure 8.45: Total amount of rollbacks for Scale Free PDES graphs, of 10 nodes, where the weights are randomized

The total amount of rollbacks highly fluctuates between instances of the same graph. Sometimes this causes a difference of more than 100.000 rollbacks between the same graph, but with different weights. The average total amount of rollbacks seems to decrease as the value of λ increases. We also observe that the variance becomes less, as the value of λ increases. Again, we investigate the total amount of busy, and idle rollbacks (Figures 8.46 and 8.47).

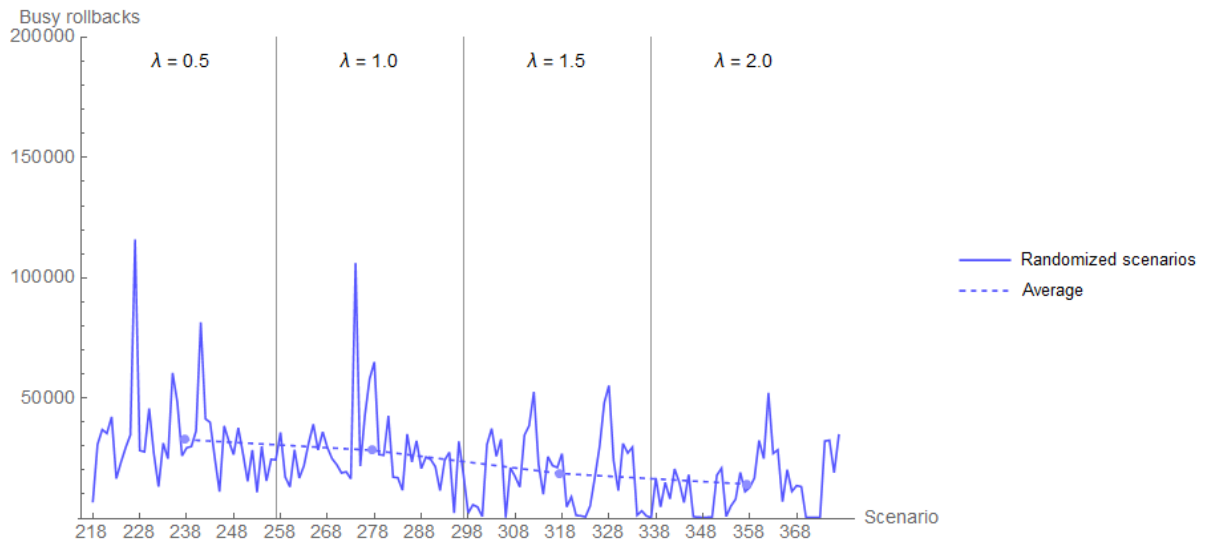


Figure 8.46: Total amount of busy rollbacks for Scale Free PDES graphs, of 10 nodes, where the weights are randomized

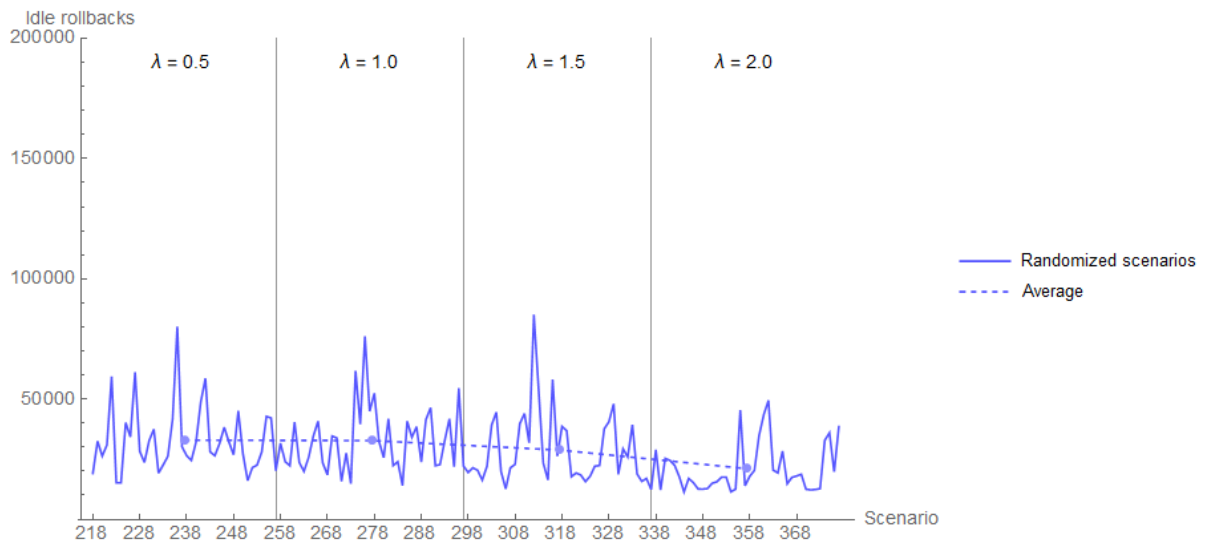


Figure 8.47: Total amount of idle rollbacks for Scale Free PDES graphs, of 10 nodes, where the weights are randomized

Figures 8.46 and 8.47 show a similar pattern concerning the variance as the value of λ increases. No clear other pattern seems to emerge based on these figures. thus we chart the Busy / Idle rollback ratio in Figure 8.48.

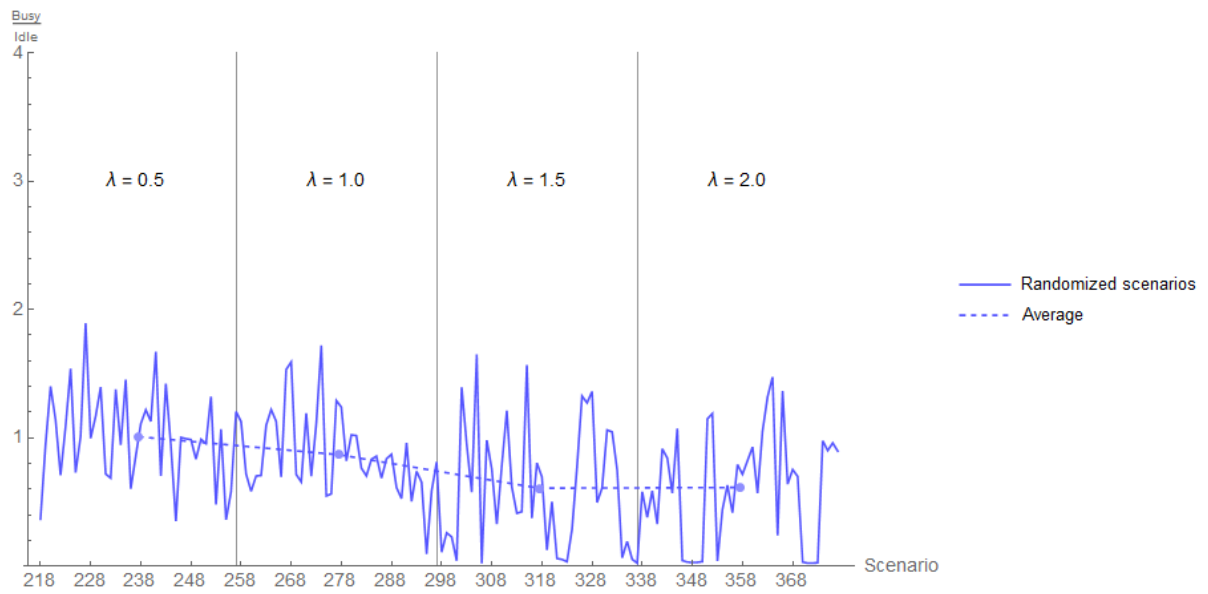


Figure 8.48: Fraction of the total amount of busy and idle rollbacks for Scale Free PDES graphs, of 10 nodes, where the weights are randomized

A slight decreasing trend can be observed in the Busy / Idle rollback fraction, as the value of λ increases. This seems to hint that more idle rollbacks seem to occur, indicating that LPs tend to have idle time more frequently as the value of λ increases. Finally, we summarize all averages per value of λ in figure 8.49.

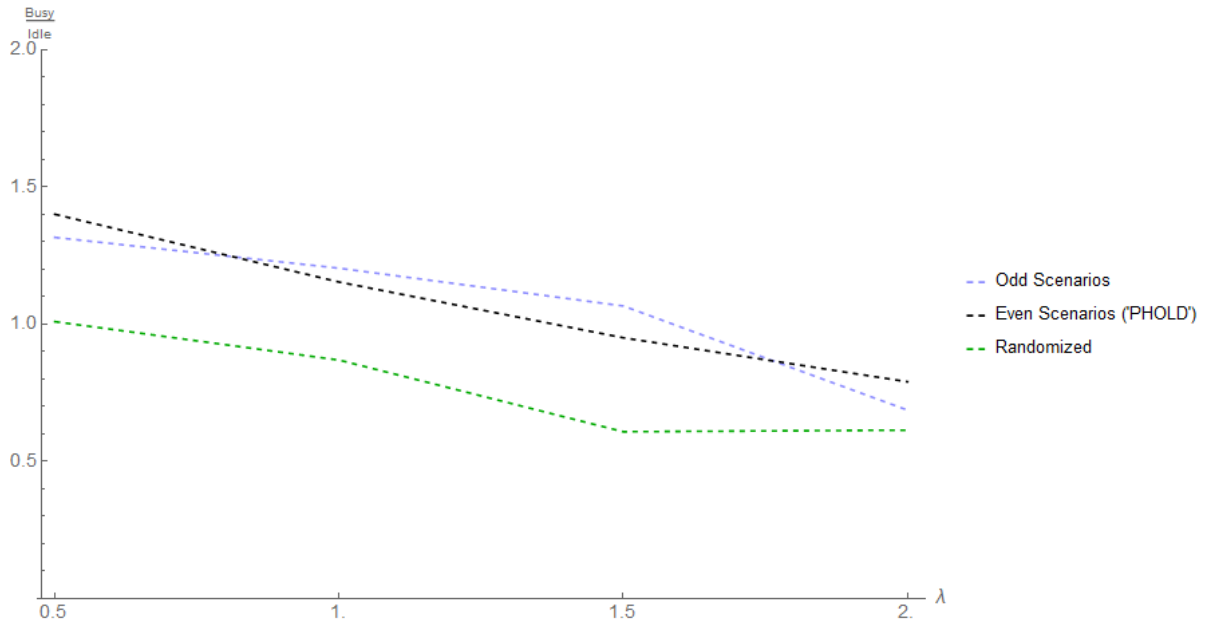


Figure 8.49: Fraction of the total amount of busy and idle rollbacks for Scale Free PDES graphs, of 10 nodes

The average fraction of the randomized scenarios seem to always lie below the odd, and even numbered scenarios from the previous section. However, the variance of these averages is relatively large, thus the averages might not be very reliable. We are able to distinguish a downward trend, for all scenarios, hinting at more frequent idle time, as the value of λ increases.

8.4.3 Conclusion

We've analyzed PHOLD scenarios (Even numbered scenario), scenarios where the event message flow distribution was non-uniform (Odd numbered scenarios), and scenarios where the EPHOLD weights are randomized. We did this for complete graphs, and Scale Free graphs. In complete graphs we observed that the amount of busy rollbacks grew compared to the amount of idle rollbacks when the graph size was increased. This phenomenon was observed for all three cases. The even scenarios relatively produced the most amount of busy rollbacks of all three, indicating a more evenly distributed workload across all LPs. The odd scenarios produced the opposite, those delivered the least amount of busy rollbacks relatively. This indicates that LPs are idling more frequently when the workload distribution is not uniform. The randomized scenarios sit in between, and largely follow the same trend as the even scenarios (with few exceptions).

We've similarly analyzed the execution of Time Warp on various scale free PDES graphs. Here we observed that both the total amount of rollbacks, and the fraction of busy and idle rollbacks, decrease with increasing values of λ . This holds for all three types of scenarios. The decrease in the busy / idle rollback fraction indicates a higher amount of idle rollbacks, indicating that LPs seem to have idle time more frequently as the value of λ increases.

8.5 Conclusions

We experimentally highlighted how the EPHOLD algorithm output differs, and how it is similar to the commonly used PHOLD benchmark. Evaluating EPHOLD revealed bottlenecks that did not show up in PHOLD. We have evaluated the parallelism approximation algorithm for complete and several scale free graphs, for which it worked very well. We have investigated the effects of varying event flows throughout the PDES graphs, for complete and scale free graphs for YAWNS, CMB, and Time Warp. We note that the results obtained for the Scale Free PDES graphs, should not be interpreted as the effect of λ on the amount of parallelism. We have "only" evaluated Scale Free graphs of size 10 due to hardware limitations. Our result hint at a certain trend, but more research is needed to conclusively test this relation. Finally we can firmly conclude that the amount of parallelism in YAWNS, CMB and TimeWarp are not only affected by the PDES topology, but also the flow of the event messages through the PDES graph. Our results suggest that this is not something specific for a single synchronization algorithm, but inherent to the PDES simulation.

Chapter 9

In conclusion

The contributions of this thesis to the PDES field consist of the Extended PHOLD (EPHOLD) benchmark and a Parallelism prediction algorithm for it, for the YAWNS synchronization algorithm. Experiments (Chapter 8) showed the increased relevance of EPHOLD over the PHOLD benchmark that is commonly used. The Parallelism prediction algorithm for YAWNS was shown to be highly accurate for (E)PHOLD executions on complete, and scale free graphs. The accuracy of the prediction algorithm for other types of graphs, and mixed timestamp increment functions remains for future research. We were able to predict, and achieve different amounts of parallelism by altering the event message flow using EPHOLD. This indicates that not only the PDES graph's topology has an effect on the amount of parallelism, but that the message flow is a (highly) influential factor as well.

Appendices

Tables

Id	LP_1	LP_2	LP_3	LP_4	LP_5	LP_6	LP_7	LP_8	LP_9	LP_{10}	LP_{11}	LP_{12}	LP_{13}	LP_{14}	LP_{15}	LP_{16}
0	-0.049	-0.053	-0.056													
2	0.102	0.036	0.036	0.034												
4	0.136	0.042	0.054	0.093	0.079											
6	0.092	0.048	0.062	0.093	0.142	0.129										
8	0.12	0.282	0.279	0.189	0.128	0.132	0.055									
10	0.228	0.164	0.214	0.173	0.152	0.191	0.071	0.13								
12	0.303	0.28	0.205	0.166	0.123	0.087	0.115	0.215	0.179							
14	0.123	0.319	0.201	0.207	0.275	0.251	0.182	0.156	0.147	0.115						
16	0.242	0.094	0.232	0.245	0.199	0.184	0.151	0.13	0.225	0.191	0.209					
18	0.297	0.158	0.172	0.124	0.209	0.253	0.298	0.157	0.095	0.331	0.252	0.076				
20	0.183	0.243	0.257	0.143	0.143	0.215	0.109	0.081	0.22	0.198	0.252	0.201	0.252			
22	0.162	0.3	0.331	0.104	0.255	0.302	0.25	0.126	0.067	0.149	0.125	0.282	0.203	0.244		
24	0.321	0.345	0.123	0.219	0.101	0.057	0.221	0.252	0.153	0.128	0.102	0.123	0.297	0.228	0.291	
26	0.275	0.251	0.232	0.302	0.249	0.195	0.205	0.156	0.129	0.168	0.326	0.37	0.146	0.171	0.291	0.14
28	0.147	0.228	0.189	0.336	0.239	0.161	0.097	0.15	0.246	0.244	0.275	0.322	0.16	0.185	0.231	0.231
30	0.311	0.24	0.174	0.234	0.307	0.315	0.341	0.374	0.259	0.263	0.297	0.231	0.263	0.255	0.199	0.235
32	0.185	0.218	0.264	0.285	0.116	0.147	0.269	0.192	0.181	0.193	0.242	0.35	0.239	0.106	0.239	0.2
34	0.431	0.292	0.472	0.268	0.13	0.294	0.275	0.276	0.311	0.232	0.224	0.275	0.234	0.103	0.123	0.221
36	0.336	-0.013	0.28	0.355	0.412	0.339	0.376	0.331	0.31	0.268	0.232	0.267	0.255	0.232	0.305	0.208
38	0.202	0.298	0.078	0.281	0.427	0.496	0.274	0.25	0.206	0.306	0.282	0.187	0.29	0.223	0.15	0.304
40	0.155	0.273	0.274	0.044	0.203	0.292	0.186	0.209	0.103	-0.01	0.052	0.105	0.248	0.382	0.317	0.374
42	0.243	0.281	0.376	0.081	0.116	0.23	0.223	0.235	0.294	0.193	0.152	0.192	0.244	0.377	0.384	0.176

Table 9.1: Rounded difference between the amount of predicted events per window and obtained, for even numbered scenarios in YAWNS

Id	LP_{17}	LP_{18}	LP_{19}	LP_{20}	LP_{21}	LP_{22}	LP_{23}	LP_{24}	LP_{25}
28	0.145								
30	0.087	0.067							
32	0.242	0.369	0.156						
34	0.134	0.168	0.12	0.315					
36	0.024	0.1	0.085	0.288	0.237				
38	0.351	0.192	0.341	0.309	0.233	-0.069			
40	0.312	0.13	0.038	0.136	0.268	0.202	0.469		
42	0.136	0.087	0.089	0.014	0.226	0.404	0.25	0.364	

Table 9.2: Rounded difference between the amount of predicted events per window and obtained, for even numbered scenarios in YAWNS(continued)

Id	LP_1	LP_2	LP_3	LP_4	LP_5	LP_6	LP_7	LP_8	LP_9	LP_{10}	LP_{11}	LP_{12}	LP_{13}	LP_{14}	LP_{15}	LP_{16}
1	-0.035	-0.056	-0.064													
3	0.018	0.07	0.041	0.042												
5	0.025	0.109	0.056	0.093	0.135											
7	0.017	0.077	0.072	0.141	0.16	0.178										
9	0.091	0.008	0.172	0.19	0.164	0.147	0.227									
11	-0.036	0.082	0.141	0.218	0.18	0.226	0.244	0.216								
13	0.059	0.03	0.171	0.252	0.129	0.173	0.197	0.23	0.276							
15	0.013	0.047	0.117	0.109	0.073	0.112	0.222	0.348	0.341	0.344						
17	0.027	0.008	0.03	0.117	0.088	0.146	0.294	0.366	0.331	0.359	0.27					
19	0.056	0.048	0.045	0.051	0.166	0.092	0.222	0.317	0.349	0.309	0.297	0.363				
21	0.033	0.011	0.126	0.194	0.069	0.194	0.313	0.233	0.201	0.205	0.323	0.478	0.268			
23	0.04	-0.034	0.042	0.078	0.186	0.229	0.264	0.255	0.376	0.364	0.319	0.291	0.316	0.27		
25	0.	0.05	0.145	0.013	0.13	0.225	0.23	0.287	0.176	0.239	0.339	0.403	0.453	0.543	0.184	
27	0.067	0.047	0.022	0.062	0.137	0.144	0.204	0.288	0.298	0.219	0.296	0.378	0.299	0.121	0.411	0.437
29	0.077	0.039	0.02	0.034	0.114	0.181	0.145	0.206	0.249	0.268	0.275	0.355	0.291	0.323	0.405	0.315
31	0.042	0.059	0.026	0.08	0.142	0.146	0.18	0.246	0.295	0.255	0.26	0.222	0.216	0.38	0.367	0.278
33	-0.061	0.058	0.015	0.135	0.159	0.148	0.168	0.208	0.014	0.022	0.222	0.411	0.4	0.322	0.355	0.377
35	-0.01	0.045	0.017	-0.077	-0.014	0.238	0.098	0.206	0.262	0.214	0.171	0.227	0.112	0.171	0.431	0.387
37	-0.014	0.156	0.125	0.048	0.18	0.171	0.215	0.167	0.082	0.321	0.203	0.166	0.282	0.33	0.342	0.257
39	0.019	0.087	0.099	-0.069	0.124	0.104	0.074	0.166	0.14	0.205	0.13	0.297	0.313	0.275	0.176	0.246
41	0.052	0.043	0.048	-0.002	0.046	-0.066	-0.077	0.136	0.124	0.128	0.23	0.318	0.309	0.325	0.328	0.288
43	0.014	0.068	0.002	0.023	0.062	-0.049	0.17	0.138	0.294	0.12	0.28	0.135	0.194	0.2	0.199	0.468

Table 9.3: Rounded difference between the amount of predicted events per window and obtained, for odd numbered scenarios in YAWNS

Id	LP_{17}	LP_{18}	LP_{19}	LP_{20}	LP_{21}	LP_{22}	LP_{23}	LP_{24}	LP_{25}
29	0.463								
31	0.405	0.508							
33	0.441	0.54	0.323						
35	0.355	0.439	0.478	0.446					
37	0.348	0.558	0.575	0.341	0.256				
39	0.423	0.354	0.487	0.483	0.496	0.364			
41	0.299	0.338	0.415	0.372	0.354	0.441	0.678		
43	0.508	0.574	0.589	0.424	0.191	0.365	0.367	0.344	

Table 9.4: Rounded difference between the amount of predicted events per window and obtained, for odd numbered scenarios in YAWNS (continued)

Scenario Id	Predicted	Observed	Error %
0	3.	2.995	0.167%
2	4.	3.981	0.475%
4	5.	4.988	0.24%
6	6.	5.936	1.067%
8	7.	6.899	1.443%
10	8.	7.939	0.762%
12	9.	8.892	1.2%
14	10.	9.876	1.24%
16	11.	10.871	1.173%
18	12.	11.715	2.375%
20	13.	12.678	2.477%
22	14.	13.798	1.443%
24	15.	14.63	2.467%
26	16.	15.799	1.256%
28	17.	16.648	2.071%
30	18.	17.772	1.267%
32	19.	18.683	1.668%
34	20.	19.635	1.825%
36	21.	20.415	2.786%
38	22.	21.384	2.8%
38*	22.	21.804	0.891%
40	23.	22.399	2.613%
42	24.	23.204	3.317%
44	25.	24.235	3.06%

Table 9.5: The predicted and observed amount of parallelism, for even numbered scenarios.

* Scenario has a termination time of 30.000 (i.e. 10 times higher than the other scenarios)

Scenario Id	Predicted	Observed	Error %
1	2.444	2.444	0.0%
3	2.917	2.911	0.206%
5	3.4	3.404	-0.118%
7	3.889	3.9	-0.283%
9	4.381	4.382	-0.023%
11	4.875	4.849	0.533%
13	5.37	5.388	-0.335%
15	5.867	5.858	0.153%
17	6.364	6.376	-0.189%
19	6.861	6.816	0.656%
21	7.359	7.298	0.829%
23	7.857	7.789	0.865%
25	8.356	8.227	1.544%
27	8.854	8.94	-0.971%
29	9.353	9.448	-1.016%
31	9.852	10.011	-1.614%
33	10.351	10.184	1.613%
35	10.85	10.872	-0.203%
37	11.349	11.203	1.286%
39	11.848	11.851	-0.025%
41	12.348	12.648	-2.43%
41*	12.348	12.351	-0.024 %
43	12.847	12.654	1.502%
45	13.347	13.24	0.802%

Table 9.6: YAWNS Parallelism statistics and approximation errors of Complete graphs for the odd numbered scenarios.

* Scenario has a termination time of 30.000 (i.e. 10 times higher than the other scenarios)

Graph size (n)	Scenario Id	Predicted	Observed	Error %
3	126	2.459	2.46	-0.041%
3	127	2.052	2.053	-0.049%
3	128	2.153	2.152	0.046%
3	129	2.682	2.678	0.149%
4	130	2.954	2.97	-0.542%
4	131	3.16	3.156	0.127%
4	132	3.373	3.378	-0.148%
4	133	3.486	3.467	0.545%
5	134	3.596	3.601	-0.139%
5	135	3.859	3.875	-0.415%
5	136	3.721	3.719	0.054%
5	137	3.062	3.054	0.261%
6	138	5.045	5.072	-0.535%
6	139	4.699	4.72	-0.447%
6	140	4.36	4.355	0.115%
6	141	4.454	4.477	-0.516%
7	142	6.326	6.365	-0.617%
7	143	5.368	5.359	0.168%
7	144	4.928	4.905	0.467%
7	145	5.532	5.552	-0.362%
8	146	6.722	6.73	-0.119%
8	147	6.368	6.379	-0.173%
8	148	5.048	5.097	-0.971%
8	149	6.604	6.624	-0.303%
9	150	7.159	7.107	0.726%
9	151	5.708	5.712	-0.07%
9	152	7.492	7.495	-0.04%
9	153	6.178	6.194	-0.259%
10	154	7.512	7.517	-0.067%
10	155	7.893	7.92	-0.342%
10	156	8.148	8.145	0.037%
10	157	7.764	7.822	-0.747%
11	158	7.554	7.59	-0.477%
11	159	8.815	8.748	0.76%
11	160	9.101	9.149	-0.527%
11	161	6.598	6.62	-0.333%
12	162	8.631	8.687	-0.649%
12	163	9.346	9.322	0.257%
12	164	8.94	8.876	0.716%
12	165	8.763	8.826	-0.719%

13	166	10.375	10.46	-0.819%
13	167	10.068	10.112	-0.437%
13	168	8.359	8.424	-0.778%
13	169	11.031	11.046	-0.136%
14	170	11.19	11.096	0.84%
14	171	8.051	8.035	0.199%
14	172	10.949	10.829	1.096%
14	173	11.319	11.306	0.115%
15	174	10.163	10.164	-0.01%
15	175	9.77	9.657	1.157%
15	176	12.85	12.753	0.755%
15	177	12.096	12.07	0.215%
16	178	9.845	9.932	-0.884%
16	179	14.155	14.274	-0.841%
16	180	12.803	12.827	-0.187%
16	181	9.379	9.543	-1.749%
17	182	12.561	12.583	-0.175%
17	183	12.598	12.457	1.119%
17	184	10.973	10.913	0.547%
17	185	11.216	11.164	0.464%
18	186	13.107	12.988	0.908%
18	187	13.407	13.347	0.448%
18	188	10.253	10.38	-1.239%
18	189	14.849	14.973	-0.835%
19	190	14.879	14.901	-0.148%
19	191	14.101	14.129	-0.199%
19	192	13.467	13.672	-1.522%
19	193	13.847	13.679	1.213%
20	194	15.676	15.664	0.077%
20	195	12.938	13.096	-1.221%
20	196	15.473	15.661	-1.215%
20	197	13.031	13.168	-1.051%
21	198	11.696	11.708	-0.103%
21	199	14.774	14.574	1.354%
21	200	17.292	17.471	-1.035%
21	201	12.905	12.971	-0.511%
22	202	14.703	14.637	0.449%
22	203	13.878	13.841	0.267%
22	204	18.301	18.144	0.858%
22	205	12.684	12.608	0.599%
23	206	19.562	19.689	-0.649%

23	207	20.002	19.834	0.84%
23	208	12.742	12.887	-1.138%
23	209	16.121	16.196	-0.465%
24	210	17.166	17.244	-0.454%
24	211	17.223	16.955	1.556%
24	212	15.808	16.284	-3.011%
24	213	20.248	20.349	-0.499%
25	214	20.139	20.17	-0.154%
25	215	14.589	14.623	-0.233%
25	216	19.592	19.677	-0.434%
25	217	20.755	20.634	0.583%

Table 9.7: YAWNS Parallelism statistics and approximation errors of Complete graphs where the EPHOLD weights are randomized.

Scenario Id	Predicted	Observed	Error %
218	2.987	3.028	-1.373%
219	2.964	2.975	-0.371%
220	4.038	4.04	-0.05%
221	4.483	4.336	3.279%
222	2.733	2.732	0.037%
223	4.32	4.325	-0.116%
224	5.202	5.153	0.942%
225	2.657	2.652	0.188%
226	3.813	3.784	0.761%
227	3.076	3.049	0.878%
228	4.826	4.209	12.785%
229	3.997	4.003	-0.15%
230	3.268	3.261	0.214%
231	3.8	3.677	3.237%
232	3.042	3.036	0.197%
233	3.507	3.52	-0.371%
234	3.734	3.656	2.089%
235	3.494	3.491	0.086%
236	2.542	2.548	-0.236%
237	4.262	4.248	0.328%
238	3.96	4.116	-3.939%
239	4.269	4.29	-0.492%
240	4.369	4.368	0.023%
241	3.24	2.996	7.531%
242	2.609	2.619	-0.383%

243	4.359	4.385	-0.596%
244	3.989	4.05	-1.529%
245	4.717	2.404	49.035%
246	2.787	2.813	-0.933%
247	2.775	2.773	0.072%
248	4.512	4.438	1.64%
249	2.64	2.639	0.038%
250	3.517	3.566	-1.393%
251	4.734	4.729	0.106%
252	3.671	3.639	0.872%
253	3.962	3.944	0.454%
254	3.114	3.126	-0.385%
255	2.118	2.131	-0.614%
256	2.265	2.268	-0.132%
257	4.389	4.426	-0.843%
258	3.724	3.734	-0.269%
259	3.805	3.838	-0.867%
260	3.753	3.761	-0.213%
261	3.241	3.257	-0.494%
262	3.328	3.337	-0.27%
263	3.811	3.794	0.446%
264	5.215	5.257	-0.805%
265	4.611	3.735	18.998%
266	2.596	2.603	-0.27%
267	3.897	3.897	0.%
268	4.72	4.694	0.551%
269	2.508	2.516	-0.319%
270	2.618	2.617	0.038%
271	4.371	4.406	-0.801%
272	2.989	2.995	-0.201%
273	3.871	3.869	0.052%
274	2.894	2.863	1.071%
275	2.979	2.319	22.155%
276	2.451	2.46	-0.367%
277	3.576	3.568	0.224%
278	2.83	2.79	1.413%
279	2.582	2.589	-0.271%
280	3.954	3.959	-0.126%
281	2.674	2.684	-0.374%
282	3.198	2.892	9.568%
283	3.353	3.354	-0.03%

284	4.486	4.482	0.089%
285	2.646	2.627	0.718%
286	2.572	2.567	0.194%
287	2.691	2.707	-0.595%
288	2.861	2.862	-0.035%
289	2.613	2.615	-0.077%
290	2.638	2.653	-0.569%
291	3.279	3.284	-0.152%
292	3.718	3.488	6.186%
293	2.734	2.729	0.183%
294	2.279	2.276	0.132%
295	2.057	2.062	-0.243%
296	2.274	2.28	-0.264%
297	3.514	3.526	-0.341%
298	2.089	2.09	-0.048%
299	2.14	2.144	-0.187%
300	2.133	2.134	-0.047%
301	2.043	2.044	-0.049%
302	3.223	3.231	-0.248%
303	2.795	2.81	-0.537%
304	2.639	2.644	-0.189%
305	4.011	4.017	-0.15%
306	2.011	2.013	-0.099%
307	2.4	2.409	-0.375%
308	2.29	2.291	-0.044%
309	2.099	2.104	-0.238%
310	2.894	2.904	-0.346%
311	4.299	4.263	0.837%
312	2.361	2.265	4.066%
313	2.408	2.415	-0.291%
314	3.79	3.801	-0.29%
315	6.265	6.236	0.463%
316	2.401	2.399	0.083%
317	3.179	3.165	0.44%
318	2.819	2.829	-0.355%
319	2.053	2.055	-0.097%
320	3.477	3.467	0.288%
321	2.038	2.033	0.245%
322	2.026	2.024	0.099%
323	2.013	2.014	-0.05%
324	2.126	2.129	-0.141%

325	2.287	2.284	0.131%
326	3.814	3.833	-0.498%
327	3.291	3.281	0.304%
328	3.356	3.358	-0.06%
329	2.716	2.205	18.814%
330	3.196	3.199	-0.094%
331	3.034	3.034	0.%
332	2.7	2.709	-0.333%
333	2.686	2.689	-0.112%
334	2.063	2.064	-0.048%
335	2.094	2.095	-0.048%
336	2.038	2.04	-0.098%
337	2.006	2.007	-0.05%
338	2.314	2.311	0.13%
339	3.505	3.498	0.2%
340	2.307	2.305	0.087%
341	2.203	2.207	-0.182%
342	2.404	2.411	-0.291%
343	2.671	2.677	-0.225%
344	3.174	3.155	0.599%
345	2.707	2.711	-0.148%
346	2.	2.001	-0.05%
347	2.	2.001	-0.05%
348	2.	2.001	-0.05%
349	2.	2.001	-0.05%
350	2.019	2.021	-0.099%
351	2.606	2.602	0.153%
352	2.516	2.516	0.%
353	2.007	2.008	-0.05%
354	3.185	3.183	0.063%
355	3.022	3.006	0.529%
356	2.39	2.396	-0.251%
357	2.798	2.803	-0.179%
358	3.784	3.769	0.396%
359	3.05	3.059	-0.295%
360	2.909	2.918	-0.309%
361	2.616	2.619	-0.115%
362	2.632	2.558	2.812%
363	3.46	3.461	-0.029%
364	3.824	3.823	0.026%
365	2.202	2.203	-0.045%

366	3.813	3.821	-0.21%
367	2.827	2.832	-0.177%
368	3.026	3.031	-0.165%
369	3.123	3.123	0.%
370	2.	2.001	-0.05%
371	2.	2.001	-0.05%
372	2.	2.001	-0.05%
373	2.	2.001	-0.05%
374	2.663	2.664	-0.038%
375	2.667	2.667	0.%
376	2.664	2.671	-0.263%
377	2.68	2.698	-0.672%

Table 9.8: YAWNS Parallelism statistics and approximation errors of Scale Free graphs where the EPHOLD weights are randomized.

Scenario Id	Predicted	Observed	Error %
46	4.5	4.492	0.178%
48	6.	5.967	0.55%
50	3.6	3.596	0.111%
52	6.	5.972	0.467%
54	6.	6.006	-0.1%
56	4.5	4.529	-0.644%
58	3.	3.005	-0.167%
60	4.5	4.492	0.178%
62	4.5	4.546	-1.022%
64	4.5	4.451	1.089%
66	4.5	4.531	-0.689%
68	3.	3.002	-0.067%
70	3.6	3.599	0.028%
72	6.001	6.004	-0.05%
74	4.5	4.514	-0.311%
76	4.5	4.489	0.244%
78	3.	3.007	-0.233%
80	4.5	4.507	-0.156%
82	3.	2.998	0.067%
84	2.25	2.251	-0.044%
86	3.	2.98	0.667%
88	2.25	2.252	-0.089%
90	4.5	4.474	0.578%
92	3.6	3.593	0.194%
94	3.	3.021	-0.7%
96	2.25	2.249	0.044%
98	4.5	4.461	0.867%
100	3.	2.993	0.233%
102	2.25	2.251	-0.044%
104	2.571	2.579	-0.311%
106	2.571	2.572	-0.039%
108	2.	2.001	-0.05%
110	2.25	2.249	0.044%
112	2.571	2.577	-0.233%
114	3.6	3.6	0.%
116	3.	2.998	0.067%
118	3.6	3.575	0.694%
120	2.	2.001	-0.05%
122	2.571	2.566	0.194%

Table 9.9: YAWNS Parallelism statistics and approximation errors of Scale Free graphs for the even numbered scenarios.

Scenario Id	Predicted	Observed	Error %
47	5.689	5.791	-1.793%
49	6.496	6.509	-0.2%
51	5.483	5.452	0.565%
53	5.916	5.883	0.558%
55	7.12	7.143	-0.323%
57	7.2	7.094	1.472%
59	4.837	4.853	-0.331%
61	5.29	5.257	0.624%
63	5.29	5.241	0.926%
65	5.29	5.233	1.078%
67	7.178	7.084	1.31%
69	5.294	5.341	-0.888%
71	4.806	4.782	0.499%
73	6.688	6.711	-0.344%
75	5.689	5.672	0.299%
77	4.823	4.802	0.435%
79	4.562	4.536	0.57%
81	4.823	4.889	-1.368%
83	3.873	3.87	0.077%
85	3.014	3.022	-0.265%
87	5.294	5.288	0.113%
89	3.014	3.009	0.166%
91	7.195	7.196	-0.014%
93	6.161	6.158	0.049%
95	3.873	3.846	0.697%
97	3.014	3.013	0.033%
99	6.501	6.504	-0.046%
101	4.562	4.609	-1.03%
103	3.014	3.006	0.265%
105	3.376	3.362	0.415%
107	3.376	3.376	0.%
109	2.	2.001	-0.05%
111	3.014	3.011	0.1%
113	3.376	3.377	-0.03%
115	4.806	4.855	-1.02%
117	4.837	4.832	0.103%
119	4.806	4.843	-0.77%
121	2.	2.001	-0.05%
123	4.098	4.105	-0.171%

Table 9.10: YAWNS Parallelism statistics and approximation errors of Scale Free graphs for the odd numbered scenarios.

Scenario Id	Parallelism	Scenario Id	Parallelism
0	0.663	1	0.634
2	0.531	3	0.530
4	0.435	5	0.442
6	0.377	7	0.385
8	0.333	9	0.342
10	0.301	11	0.305
12	0.279	13	0.278
14	0.258	15	0.255
16	0.241	17	0.235
18	0.227	19	0.218
20	0.216	21	0.205
22	0.206	23	0.193
24	0.197	25	0.182
26	0.189	27	0.176
28	0.180	29	0.169
30	0.171	31	0.161
32	0.164	33	0.153
34	0.157	35	0.148
36	0.152	37	0.142
38	0.147	39	0.137
40	0.144	41	0.132
42	0.139	43	0.127
44	0.136	45	0.126

Table 9.11: CMB parallelism measurements for scenarios with complete graphs.

Nodes	Scen. Id	P	Scen. Id	P	Scen. Id	P	Scen. Id	P
3	126	0.641	127	0.690	128	0.556	129	0.652
4	130	0.542	131	0.530	132	0.536	133	0.534
5	134	0.437	135	0.440	136	0.437	137	0.457
6	138	0.382	139	0.382	140	0.388	141	0.384
7	142	0.345	143	0.342	144	0.341	145	0.343
8	146	0.319	147	0.314	148	0.311	149	0.313
9	150	0.287	151	0.289	152	0.289	153	0.288
10	154	0.266	155	0.264	156	0.268	157	0.265
11	158	0.246	159	0.248	160	0.247	161	0.245
12	162	0.231	163	0.234	164	0.234	165	0.231
13	166	0.221	167	0.221	168	0.218	169	0.221
14	170	0.209	171	0.206	172	0.210	173	0.210
15	174	0.196	175	0.198	176	0.201	177	0.200
16	178	0.187	179	0.193	180	0.190	181	0.188
17	182	0.182	183	0.181	184	0.181	185	0.180
18	186	0.173	187	0.174	188	0.172	189	0.174
19	190	0.165	191	0.165	192	0.165	193	0.166
20	194	0.159	195	0.158	196	0.160	197	0.160
21	198	0.153	199	0.154	200	0.154	201	0.153
22	202	0.149	203	0.148	204	0.150	205	0.148
23	206	0.145	207	0.145	208	0.142	209	0.144
24	210	0.140	211	0.140	212	0.139	213	0.141
25	214	0.137	215	0.136	216	0.137	217	0.138

Table 9.12: CMB parallelism (P) measurements for scenarios with complete graphs and randomized weights.

Scen. Id	P	Scen. Id	P	Scen. Id	P	Scen. Id	P
218	0.620	219	0.430	220	0.464	221	0.527
222	0.396	223	0.485	224	0.567	225	0.410
226	0.471	227	0.353	228	0.511	229	0.460
230	0.511	231	0.480	232	0.472	233	0.496
234	0.485	235	0.415	236	0.372	237	0.516
238	0.456	239	0.431	240	0.459	241	0.284
242	0.428	243	0.501	244	0.473	245	0.296
246	0.504	247	0.536	248	0.525	249	0.501
250	0.476	251	0.547	252	0.464	253	0.504
254	0.441	255	0.427	256	0.529	257	0.497
258	0.548	259	0.477	260	0.482	261	0.505
262	0.468	263	0.508	264	0.544	265	0.500
266	0.491	267	0.566	268	0.565	269	0.511
270	0.470	271	0.560	272	0.484	273	0.513
274	0.346	275	0.537	276	0.279	277	0.378
278	0.503	279	0.534	280	0.478	281	0.405
282	0.605	283	0.501	284	0.518	285	0.544
286	0.505	287	0.547	288	0.457	289	0.492
290	0.554	291	0.474	292	0.595	293	0.513
294	0.469	295	0.553	296	0.480	297	0.433
298	0.581	299	0.571	300	0.573	301	0.577
302	0.596	303	0.520	304	0.542	305	0.513
306	0.601	307	0.574	308	0.600	309	0.302
310	0.383	311	0.503	312	0.314	313	0.448
314	0.543	315	0.582	316	0.409	317	0.527
318	0.474	319	0.565	320	0.536	321	0.597
322	0.598	323	0.598	324	0.597	325	0.604
326	0.454	327	0.452	328	0.462	329	0.311
330	0.540	331	0.525	332	0.582	333	0.540
334	0.542	335	0.609	336	0.595	337	0.622
338	0.557	339	0.580	340	0.592	341	0.593
342	0.574	343	0.534	344	0.542	345	0.569
346	0.604	347	0.621	348	0.605	349	0.616
350	0.599	351	0.578	352	0.594	353	0.595
354	0.556	355	0.588	356	0.541	357	0.569
358	0.532	359	0.594	360	0.505	361	0.513
362	0.364	363	0.497	364	0.537	365	0.557
366	0.547	367	0.446	368	0.562	369	0.575
370	0.624	371	0.609	372	0.601	373	0.600
374	0.577	375	0.546	376	0.598	377	0.528

Table 9.13: CMB parallelism (P) measurements for scenarios with Scale Free graphs of 10 nodes and randomized weights.

Scenario Id	Parallelism	Scenario Id	Parallelism
46	0.524	47	0.556
48	0.536	49	0.564
50	0.523	51	0.554
52	0.536	53	0.575
54	0.524	55	0.540
56	0.510	57	0.532
58	0.512	59	0.547
60	0.538	61	0.579
62	0.534	63	0.567
64	0.532	65	0.567
66	0.531	67	0.569
68	0.527	69	0.565
70	0.553	71	0.591
72	0.562	73	0.587
74	0.502	75	0.518
76	0.537	77	0.566
78	0.557	79	0.575
80	0.564	81	0.590
82	0.555	83	0.576
84	0.549	85	0.580
86	0.614	87	0.623
88	0.552	89	0.590
90	0.614	91	0.625
92	0.505	93	0.544
94	0.523	95	0.567
96	0.546	97	0.583
98	0.613	99	0.624
100	0.517	101	0.552
102	0.565	103	0.591
104	0.615	105	0.626
106	0.591	107	0.603
108	0.590	109	0.604
110	0.652	111	0.652
112	0.614	113	0.625
114	0.589	115	0.603
116	0.563	117	0.587
118	0.537	119	0.580
120	0.564	121	0.587
122	0.652	123	0.652
124	0.581	125	0.611

Table 9.14: CMB Parallelism statistics for scenarios with Scale Free graphs.

λ	Parallelism (Even)	Increase (Even)	Parallelism (Odd)	Increase (Odd)
0.5	0.5263	-	0.5571	-
1.0	0.5421	3.01 %	0.5703	2.37 %
1.5	0.5594	3.18 %	0.5876	3.04 %
2.0	0.5928	5.97 %	0.6099	3.80 %

Table 9.15: CMB Mean parallelism statistics for scenarios with Scale Free graphs, aggregated by values of λ . The increase in parallelism shown is with regard to the previous value of λ

Scenario Id	Busy rollbacks	Scenario Id	Busy rollbacks
0	16346.52	1	19877.03
4	82309.11	5	93454.39
8	36798.29	9	153354.08
12	116016.94	13	137699.47
16	98876.80	17	178991.57
20	86642.69	21	187742.42
24	55548.50	25	172180.38
28	68156.60	29	163962.65
32	61020.70	33	169177.35
36	59967.55	37	159744.67
40	48396.61	41	137706.63
44	47888.73	45	139914.41

Table 9.16: TimeWarp Average amount of rollbacks per scenario for complete PDES graphs.

Scenario Id	Busy rollbacks	Scenario Id	Busy rollbacks
0	10157.83	1	8731.85
4	55821.81	5	48686.87
8	26614.96	9	89864.37
12	93752.05	13	85194.17
16	76714.96	17	113071.85
20	67949.57	21	120074.24
24	46871.45	25	115637.83
28	58290.20	29	112173.99
32	53621.30	33	118354.83
36	53173.79	37	113926.42
40	43856.03	41	100449.69
44	43646.16	45	103314.84

Table 9.17: TimeWarp Average amount of busy rollbacks per scenario for complete PDES graphs.

Scenario Id	Idle rollbacks	Scenario Id	Idle rollbacks
0	6188.69	1	11145.18
4	26487.30	5	44767.52
8	10183.33	9	63489.71
12	22264.89	13	52505.30
16	22161.85	17	65919.72
20	18693.12	21	67668.19
24	8677.05	25	56542.55
28	9866.39	29	51788.66
32	7399.40	33	50822.52
36	6793.76	37	45818.26
40	4540.58	41	37256.94
44	4242.57	45	36599.57

Table 9.18: TimeWarp Average amount of idle rollbacks per scenario for complete PDES graphs.

Nodes	Id	Amount	Id	Amount	Id	Amount	Id	Amount
3	126	20461.69	127	12074.20	128	16183.39	129	19342.53
4	130	31001.27	131	46407.58	132	29878.31	133	32512.88
5	134	90538.15	135	104250.45	136	81239.18	137	82305.25
6	138	86150.34	139	98366.40	140	119532.16	141	141657.55
7	142	50138.88	143	105863.08	144	181245.55	145	83105.54
8	146	44184.94	147	38843.24	148	182083.70	149	49736.43
9	150	111491.75	151	217969.20	152	114218.22	153	139955.27
10	154	124353.31	155	124909.44	156	127154.39	157	129490.79
11	158	140259.31	159	129643.19	160	135278.91	161	178807.25
12	162	137405.96	163	117049.72	164	105340.50	165	128560.65
13	166	106449.24	167	104422.11	168	163514.61	169	98034.82
14	170	90928.79	171	210014.91	172	102622.49	173	98538.73
15	174	109604.49	175	149836.72	176	63447.19	177	66183.84
16	178	183812.64	179	42736.88	180	56367.30	181	243189.63
17	182	95039.05	183	93912.17	184	125887.91	185	159927.36
18	186	90179.04	187	88569.42	188	212995.05	189	71549.65
19	190	72748.34	191	80810.42	192	83337.72	193	86194.92
20	194	65041.57	195	123496.40	196	73532.84	197	102535.06
21	198	239006.61	199	93858.75	200	66308.24	201	118807.74
22	202	99574.19	203	134021.21	204	57955.18	205	196197.73
23	206	52532.79	207	51218.75	208	224424.90	209	76580.11
24	210	53391.33	211	52110.33	212	74497.85	213	42054.81
25	214	51970.22	215	165662.84	216	52900.11	217	50933.21

Table 9.19: TimeWarp Average amount of rollbacks per scenario for complete PDES graphs, where all weights are randomized

Nodes	Id	Amount	Id	Amount	Id	Amount	Id	Amount
3	126	11476.17	127	6973.92	128	12383.39	129	9088.98
4	130	15226.20	131	21624.12	132	12227.09	133	12579.66
5	134	39108.83	135	41663.15	136	32119.34	137	45847.15
6	138	29187.12	139	34571.05	140	47673.72	141	55523.17
7	142	14907.76	143	35863.55	144	57301.52	145	25344.09
8	146	11710.74	147	10271.60	148	65788.25	149	12882.74
9	150	31591.76	151	64582.79	152	27813.79	153	42631.65
10	154	33911.47	155	30267.97	156	32281.35	157	36110.99
11	158	37847.89	159	33385.60	160	37567.48	161	54862.23
12	162	36924.38	163	29305.12	164	26906.52	165	33008.73
13	166	24445.87	167	27905.33	168	40930.14	169	23249.82
14	170	20681.59	171	51813.22	172	23058.00	173	21962.44
15	174	23445.15	175	29291.69	176	11223.27	177	12962.89
16	178	28856.81	179	5144.23	180	8952.79	181	37786.99
17	182	16690.98	183	15947.10	184	22056.81	185	33220.78
18	186	14754.40	187	15519.65	188	38078.45	189	10618.78
19	190	12318.35	191	11608.33	192	13178.76	193	12608.60
20	194	11017.96	195	19254.77	196	11286.18	197	14636.24
21	198	37128.26	199	12652.16	200	8685.43	201	15715.00
22	202	12107.32	203	17802.71	204	7530.52	205	26839.09
23	206	5695.26	207	5879.75	208	27790.77	209	10440.19
24	210	5142.77	211	6301.52	212	7216.60	213	4049.91
25	214	5592.37	215	17664.50	216	4928.26	217	5475.35

Table 9.20: TimeWarp Average amount of idle rollbacks per scenario id for complete PDES graphs where all weights are randomized.

Nodes	Id	Amount	Id	Amount	Id	Amount	Id	Amount
3	126	8985.52	127	5100.28	128	3800.00	129	10253.55
4	130	15775.06	131	24783.46	132	17651.23	133	19933.22
5	134	51429.32	135	62587.30	136	49119.85	137	36458.10
6	138	56963.22	139	63795.35	140	71858.44	141	86134.38
7	142	35231.12	143	69999.53	144	123944.03	145	57761.45
8	146	32474.20	147	28571.65	148	116295.45	149	36853.69
9	150	79899.99	151	153386.42	152	86404.43	153	97323.62
10	154	90441.84	155	94641.46	156	94873.05	157	93379.79
11	158	102411.42	159	96257.59	160	97711.42	161	123945.02
12	162	100481.58	163	87744.60	164	78433.97	165	95551.92
13	166	82003.37	167	76516.78	168	122584.48	169	74785.00
14	170	70247.20	171	158201.68	172	79564.49	173	76576.29
15	174	86159.34	175	120545.03	176	52223.92	177	53220.94
16	178	154955.83	179	37592.65	180	47414.51	181	205402.64
17	182	78348.07	183	77965.07	184	103831.11	185	126706.58
18	186	75424.64	187	73049.78	188	174916.60	189	60930.88
19	190	60429.99	191	69202.09	192	70158.96	193	73586.32
20	194	54023.61	195	104241.63	196	62246.67	197	87898.82
21	198	201878.35	199	81206.59	200	57622.81	201	103092.74
22	202	87466.87	203	116218.50	204	50424.66	205	169358.63
23	206	46837.53	207	45339.00	208	196634.13	209	66139.91
24	210	48248.56	211	45808.81	212	67281.26	213	38004.90
25	214	46377.85	215	147998.34	216	47971.85	217	45457.86

Table 9.21: TimeWarp Average amount of busy rollbacks per scenario for complete PDES graphs where all weights are randomized.

Nodes	Scen. Id	<i>B/I</i>	Scen. Id	<i>B/I</i>	Scen. Id	<i>B/I</i>	Scen. Id	<i>B/I</i>
3	126	0.78	127	0.73	128	0.31	129	1.13
4	130	1.04	131	1.15	132	1.44	133	1.58
5	134	1.32	135	1.50	136	1.53	137	0.80
6	138	1.95	139	1.85	140	1.51	141	1.55
7	142	2.36	143	1.95	144	2.16	145	2.28
8	146	2.77	147	2.78	148	1.77	149	2.86
9	150	2.53	151	2.38	152	3.11	153	2.28
10	154	2.67	155	3.13	156	2.94	157	2.59
11	158	2.71	159	2.88	160	2.60	161	2.26
12	162	2.72	163	2.99	164	2.92	165	2.89
13	166	3.35	167	2.74	168	2.99	169	3.22
14	170	3.40	171	3.05	172	3.45	173	3.49
15	174	3.67	175	4.12	176	4.65	177	4.11
16	178	5.37	179	7.31	180	5.30	181	5.44
17	182	4.69	183	4.89	184	4.71	185	3.81
18	186	5.11	187	4.71	188	4.59	189	5.74
19	190	4.91	191	5.96	192	5.32	193	5.84
20	194	4.90	195	5.41	196	5.52	197	6.01
21	198	5.44	199	6.42	200	6.63	201	6.56
22	202	7.22	203	6.53	204	6.70	205	6.31
23	206	8.22	207	7.71	208	7.08	209	6.34
24	210	9.38	211	7.27	212	9.32	213	9.38
25	214	8.29	215	8.38	216	9.73	217	8.30

Table 9.22: TimeWarp, the fraction of Busy (B) and Idle (I) rollbacks per scenario for complete PDES graphs where all weights are randomized.

Scen. Id	Amount	Scen. Id	Amount	Scen. Id	Amount	Scen. Id	Amount
218	26134.93	219	63234.93	220	63156.26	221	66198.20
222	101527.52	223	31693.76	224	38491.70	225	69607.27
226	68893.88	227	177216.39	228	56428.57	229	51294.33
230	78442.82	231	64418.68	232	32384.21	233	53875.85
234	51015.55	235	102032.48	236	128509.26	237	56179.40
238	55727.01	239	54352.23	240	68512.11	241	130217.32
242	99928.22	243	67913.72	244	51447.79	245	42557.85
246	76665.87	247	64154.32	248	53301.39	249	82740.08
250	54112.85	251	31432.88	252	49933.19	253	33352.82
254	57959.98	255	58260.34	256	66615.07	257	44508.55
258	67261.95	259	41167.66	260	35341.45	261	68910.02
262	40301.10	263	41834.77	264	57102.52	265	73642.78
266	69271.59	267	59354.57	268	47838.23	269	59357.91
270	56051.95	271	34680.42	272	46836.94	273	31153.35
274	167851.29	275	61121.68	276	119062.70	277	102932.71
278	117441.15	279	58806.98	280	51823.43	281	84390.90
282	39446.57	283	40805.83	284	25654.12	285	75840.33
286	57519.97	287	70785.32	288	44400.79	289	66931.94
290	70893.18	291	43754.85	292	34249.88	293	56909.61
294	69249.47	295	23880.07	296	86435.70	297	40184.64
298	21597.61	299	27009.68	300	25062.53	301	17005.51
302	52695.12	303	76499.50	304	70297.87	305	52782.95
306	12932.26	307	42453.72	308	40181.72	309	52757.20
310	78616.63	311	70303.63	312	137673.93	313	76756.99
314	33360.43	315	41925.62	316	80042.32	317	47342.29
318	65479.17	319	41565.31	320	26624.27	321	20440.00
322	19382.98	323	16256.47	324	23116.50	325	39196.46
326	52068.16	327	85603.93	328	95763.12	329	71884.75
330	30184.53	331	60390.07	332	53069.52	333	68861.01
334	20122.62	335	18832.72	336	17905.29	337	12716.84
338	45710.18	339	16862.33	340	40164.39	341	32535.25
342	42932.31	343	32583.60	344	17836.25	345	35148.59
346	15940.44	347	13045.22	348	12926.32	349	13177.40
350	15502.32	351	33501.36	352	38288.15	353	18261.68
354	16500.20	355	20448.35	356	64482.18	357	25016.56
358	30661.52	359	36957.50	360	67121.65	361	68300.33
362	101631.57	363	47408.64	364	47735.17	365	35234.28
366	34939.27	367	28583.25	368	31553.45	369	31842.07
370	12919.13	371	12554.15	372	12704.55	373	13102.77
374	64944.31	375	68424.24	376	38656.26	377	73054.52

Table 9.23: TimeWarp Average amount of rollbacks per scenario for Scale Free PDES graphs with 10 nodes, where all weights are randomized

Scen. Id	Amount	Scen. Id	Amount	Scen. Id	Amount	Scen. Id	Amount
218	7032.17	219	30742.58	220	36867.94	221	35308.53
222	42170.68	223	16520.85	224	23330.65	225	29400.20
226	34655.85	227	115974.89	228	28179.97	229	27646.44
230	45722.12	231	26962.00	232	13197.26	233	31231.74
234	24778.15	235	60463.80	236	48383.99	237	25983.15
238	29318.77	239	29861.22	240	36331.42	241	81472.45
242	41332.66	243	39882.31	244	24993.53	245	11079.50
246	38408.74	247	31987.03	248	26481.93	249	37649.10
250	26919.48	251	15370.44	252	28439.54	253	10828.63
254	29933.44	255	15511.73	256	24495.02	257	24331.75
258	35639.95	259	17224.28	260	13040.47	261	28472.01
262	16712.63	263	21892.32	264	31390.87	265	39069.51
266	28464.30	267	35928.79	268	29383.77	269	24755.63
270	22240.89	271	18865.35	272	19325.76	273	16399.74
274	106179.97	275	21658.98	276	42987.19	277	58011.80
278	65005.63	279	26528.66	280	26220.29	281	42574.87
282	17099.63	283	16870.34	284	11664.64	285	35026.95
286	23416.49	287	32278.76	288	20690.78	289	25453.17
290	24497.03	291	21465.77	292	11509.16	293	24218.37
294	27449.80	295	2110.91	296	31953.75	297	18019.82
298	2155.35	299	5597.01	300	4680.62	301	716.20
302	30696.26	303	37292.09	304	25779.07	305	32872.03
306	307.80	307	21050.02	308	17326.24	309	13034.19
310	34609.38	311	38551.47	312	52578.00	313	22470.74
314	9966.82	315	25604.44	316	21878.94	317	21128.97
318	26890.15	319	4726.59	320	8914.86	321	1187.49
322	994.92	323	575.68	324	5174.30	325	17108.01
326	29688.85	327	47947.89	328	55215.07	329	23882.36
330	11452.95	331	31101.26	332	27150.84	333	29536.41
334	1267.24	335	3070.75	336	939.80	337	320.39
338	16820.22	339	4655.56	340	14861.13	341	8050.50
342	20523.43	343	14908.75	344	6481.52	345	18190.85
346	697.75	347	426.27	348	379.11	349	408.33
350	553.45	351	17911.10	352	20806.54	353	738.79
354	5064.73	355	7920.40	356	19023.02	357	11060.12
358	12808.56	359	16678.95	360	32342.34	361	24801.51
362	52158.08	363	26973.81	364	28445.85	365	6855.74
366	20177.16	367	11164.85	368	13561.78	369	13126.61
370	415.21	371	314.32	372	325.08	373	354.07
374	32105.67	375	32451.82	376	18942.13	377	34557.85

Table 9.24: TimeWarp Average amount of busy rollbacks per scenario for Scale Free PDES graphs of 10 nodes where all weights are randomized.

Scen. Id	Amount	Scen. Id	Amount	Scen. Id	Amount	Scen. Id	Amount
218	19102.76	219	32492.35	220	26288.33	221	30889.66
222	59356.84	223	15172.91	224	15161.05	225	40207.07
226	34238.02	227	61241.50	228	28248.60	229	23647.90
230	32720.71	231	37456.68	232	19186.95	233	22644.12
234	26237.39	235	41568.68	236	80125.27	237	30196.25
238	26408.24	239	24491.01	240	32180.69	241	48744.87
242	58595.56	243	28031.41	244	26454.26	245	31478.35
246	38257.14	247	32167.30	248	26819.46	249	45090.98
250	27193.37	251	16062.44	252	21493.65	253	22524.18
254	28026.54	255	42748.61	256	42120.05	257	20176.80
258	31622.00	259	23943.38	260	22300.98	261	40438.01
262	23588.47	263	19942.46	264	25711.64	265	34573.27
266	40807.28	267	23425.78	268	18454.46	269	34602.28
270	33811.06	271	15815.08	272	27511.17	273	14753.61
274	61671.32	275	39462.69	276	76075.52	277	44920.91
278	52435.53	279	32278.32	280	25603.14	281	41816.03
282	22346.95	283	23935.49	284	13989.48	285	40813.38
286	34103.48	287	38506.56	288	23710.01	289	41478.78
290	46396.15	291	22289.07	292	22740.72	293	32691.24
294	41799.67	295	21769.15	296	54481.95	297	22164.83
298	19442.25	299	21412.67	300	20381.91	301	16289.31
302	21998.86	303	39207.41	304	44518.80	305	19910.93
306	12624.46	307	21403.70	308	22855.48	309	39723.01
310	44007.25	311	31752.15	312	85095.93	313	54286.25
314	23393.61	315	16321.18	316	58163.38	317	26213.32
318	38589.02	319	36838.72	320	17709.41	321	19252.51
322	18388.06	323	15680.79	324	17942.20	325	22088.45
326	22379.30	327	37656.04	328	40548.05	329	48002.39
330	18731.58	331	29288.81	332	25918.68	333	39324.61
334	18855.38	335	15761.97	336	16965.50	337	12396.45
338	28889.95	339	12206.77	340	25303.25	341	24484.74
342	22408.88	343	17674.85	344	11354.72	345	16957.74
346	15242.69	347	12618.95	348	12547.20	349	12769.07
350	14948.87	351	15590.26	352	17481.62	353	17522.88
354	11435.47	355	12527.95	356	45459.16	357	13956.45
358	17852.96	359	20278.56	360	34779.32	361	43498.82
362	49473.49	363	20434.83	364	19289.32	365	28378.54
366	14762.11	367	17418.40	368	17991.67	369	18715.46
370	12503.92	371	12239.84	372	12379.47	373	12748.70
374	32838.64	375	35972.42	376	19714.14	377	38496.67

Table 9.25: TimeWarp Average amount of idle rollbacks per scenario for Scale Free PDES graphs of 10 nodes where all weights are randomized.

Scen. Id	B/I	Scen. Id	B/I	Scen. Id	B/I	Scen. Id	B/I
218	0.37	219	0.95	220	1.40	221	1.14
222	0.71	223	1.09	224	1.54	225	0.73
226	1.01	227	1.89	228	1.0	229	1.17
230	1.40	231	0.72	232	0.69	233	1.38
234	0.94	235	1.45	236	0.60	237	0.86
238	1.11	239	1.22	240	1.13	241	1.67
242	0.71	243	1.42	244	0.94	245	0.35
246	1.00	247	0.99	248	0.99	249	0.83
250	0.99	251	0.96	252	1.32	253	0.48
254	1.07	255	0.36	256	0.58	257	1.21
258	1.13	259	0.72	260	0.58	261	0.70
262	0.71	263	1.10	264	1.22	265	1.13
266	0.70	267	1.53	268	1.59	269	0.72
270	0.66	271	1.19	272	0.70	273	1.11
274	1.72	275	0.55	276	0.57	277	1.29
278	1.24	279	0.82	280	1.02	281	1.02
282	0.77	283	0.70	284	0.83	285	0.86
286	0.69	287	0.84	288	0.87	289	0.61
290	0.53	291	0.96	292	0.51	293	0.74
294	0.66	295	0.1	296	0.59	297	0.81
298	0.11	299	0.26	300	0.23	301	0.04
302	1.40	303	0.95	304	0.58	305	1.65
306	0.02	307	0.98	308	0.76	309	0.33
310	0.79	311	1.21	312	0.62	313	0.41
314	0.43	315	1.57	316	0.38	317	0.81
318	0.70	319	0.13	320	0.50	321	0.06
322	0.05	323	0.04	324	0.29	325	0.77
326	1.33	327	1.27	328	1.36	329	0.50
330	0.61	331	1.06	332	1.05	333	0.75
334	0.07	335	0.19	336	0.06	337	0.03
338	0.58	339	0.38	340	0.59	341	0.33
342	0.92	343	0.84	344	0.57	345	1.07
346	0.05	347	0.03	348	0.03	349	0.03
350	0.04	351	1.15	352	1.19	353	0.04
354	0.44	355	0.63	356	0.42	357	0.79
358	0.72	359	0.82	360	0.93	361	0.57
362	1.05	363	1.32	364	1.47	365	0.24
366	1.37	367	0.64	368	0.75	369	0.70
370	0.03	371	0.03	372	0.03	373	0.03
374	0.98	375	0.90	376	0.96	377	0.90

Table 9.26: TimeWarp, the fraction of Busy (B) and Idle (I) rollbacks per scenario for Scale Free PDES graphs of 10 nodes, where all weights are randomized.

Scenario Id	Rollbacks	Scenario Id	Rollbacks
46	62961.72	47	38690.22
50	50467.58	51	38283.95
54	41214.65	55	37925.47
58	95113.65	59	45670.12
62	37168.99	63	29308.98
66	38083.90	67	30072.58
70	90918.97	71	40762.92
74	49071.50	75	44537.43
78	21608.82	79	22104.56
82	21373.88	83	21961.33
86	35867.55	87	21326.72
90	36160.72	91	21438.64
94	105338.32	95	43653.54
98	36000.62	99	21220.31
102	55798.96	103	25462.21
106	33587.10	107	15894.77
110	10527.82	111	10605.93
114	33832.10	115	16057.99
118	82332.20	119	33801.24
122	10631.20	123	10456.49
124	63488.74	125	32176.03

Table 9.27: TimeWarp Average amount of rollbacks per scenario for scale free PDES graphs.

Scenario Id	Idle rollbacks	Scenario Id	Idle rollbacks
46	36989.56	47	22689.86
50	30474.86	51	22861.74
54	23079.19	55	21477.04
58	57920.17	59	26957.67
62	20330.87	63	13619.75
66	21037.37	67	14017.84
70	43556.67	71	26849.11
74	29903.14	75	27061.63
78	10193.89	79	9761.27
82	10044.99	83	9751.98
86	13912.63	87	9218.00
90	13976.01	91	9290.21
94	59879.28	95	26917.46
98	14187.99	99	9202.54
102	29342.16	103	13221.24
106	16606.11	107	4391.38
110	184.46	111	196.03
114	16537.71	115	4595.56
118	46797.30	119	17474.83
122	194.84	123	187.89
124	34435.64	125	13216.94

Table 9.28: TimeWarp Average amount of busy rollbacks per scenario for Scale Free PDES graphs.

Scenario Id	Idle rollbacks	Scenario Id	Idle rollbacks
46	25972.16	47	16000.36
50	19992.73	51	15422.21
54	18135.46	55	16448.43
58	37193.48	59	18712.45
62	16838.12	63	15689.23
66	17046.52	67	16054.74
70	47362.30	71	13913.81
74	19168.37	75	17475.80
78	11414.93	79	12343.29
82	11328.89	83	12209.35
86	21954.92	87	12108.72
90	22184.71	91	12148.42
94	45459.04	95	16736.08
98	21812.63	99	12017.77
102	26456.80	103	12240.96
106	16980.99	107	11503.39
110	10343.36	111	10409.90
114	17294.39	115	11462.43
118	35534.90	119	16326.41
122	10436.37	123	10268.60
124	29053.10	125	18959.09

Table 9.29: TimeWarp Average amount of idle rollbacks per scenario for Scale Free PDES graphs.

Chapter 10

Scale free topology analysis

Scale Free networks are graphs for which the the degree distribution follows a power law. Throughout this analysis, we assume $degree(v) \geq 1 \forall v \in V$, i.e. all nodes have at least one neighbor. This makes sense as an LP that does not have neighbors in the PDES graph does not depend on other LPs in its execution, nor do other LPs depend on it. It thus does not influence the rest of the simulation (and thus its parallelism performance) as it can be considered as a separate entity.

We will employ the following notation. Let $G(V, E)$ be a PDES-graph, with V its set of vertices and E the set of undirected edges.

10.1 A Probability Density Function derivation

For convenience, equation 2.1 is shown again below.

$$\Pr(k) \sim k^{-\lambda} \quad (2.1 \text{ revisited})$$

Here $\Pr(k)$ expresses the probability of selecting a node of exactly degree k , with parameter (degree-exponent) λ . To turn equation 2.1 into a valid probability function we must claim:

$$\sum_{k=0}^{\infty} \Pr(k) = 1 \quad (10.1)$$

i.e. All probabilities must sum to 1. Thus, we introduce a normalization constant c into term 2.1, and get:

$$\Pr(k) = c \cdot k^{-\lambda} \quad (10.2)$$

Where we must estimate c . An obvious choice would be:

$$c = \frac{1}{\sum_{k=1}^{\infty} k^{-\lambda}} = \frac{1}{\zeta(\lambda)} \quad (10.3)$$

Where, ζ is the Riemann Zeta function. here we disregard all nodes with degree 0 as assumed earlier. However, since we will only be considering finite networks, we can bound the maximum degree any node can attain with a constant k_{max} .

$$\frac{1}{\sum_{k=1}^{k_{max}} k^{-\lambda}} \quad (10.4)$$

Where k_{max} is the maximum degree over all nodes in V

10.2 Hubs and leafs

Scale free networks contain a relatively low amount of so called hub nodes, and a relatively large amount of leaf nodes. Let f be the fraction of hub nodes, and $1 - f$ the fraction of leaf nodes. Only considering the average degree over all nodes is not representative of the network's skewed degree distribution. Thus we analyze hub and leaf nodes separately.

10.3 Degree estimation

In this section we will derive an expression to estimate the degrees of the set of hubs, and degrees of the set of leafs. Earlier we defined hubs as the top f fraction of nodes (in terms of degree) and leafs as the rest (i.e. the other $1 - f$ fraction of the network). To calculate this, we are looking for a point x_f in the degree distribution, for which all hub nodes are on one side of x_f in the distribution (the right hand side in the chart) and all leafs on the other. For scale free networks, the degree distribution follows a power law distribution. With x_{min} the minimum degree in the PDES graph, and given normalization constant c , and degree exponent λ , the Probability Density Function (PDF) over all degrees of the network is given by:

$$\int_{x_{min}}^{\infty} c \cdot x^{-\lambda} dx \quad (10.5)$$

To find this point x_f , we equate, for given f :

$$\int_{x_f}^{\infty} c \cdot x^{-\lambda} dx = f \cdot \int_{x_{min}}^{\infty} c \cdot x^{-\lambda} dx \quad (10.6)$$

The integral $\int c \cdot x^{-\lambda} dx$ evaluates to: $\frac{c}{-\lambda+1}x^{-\lambda+1}$ thus, the right-hand side of 10.6 expands to:

$$\begin{aligned} f \cdot \int_{x_{min}}^{\infty} c \cdot x^{-\lambda} dx &= \frac{f \cdot c}{-\lambda + 1} x^{-\lambda+1} \Big|_{x_{min}}^{\infty} \\ &= \lim_{x \rightarrow \infty} \frac{f \cdot c}{-\lambda + 1} x^{-\lambda+1} - \lim_{x \rightarrow x_{min}} \frac{f \cdot c}{-\lambda + 1} x^{-\lambda+1} \\ &= -f \cdot \frac{c}{1 - \lambda} x_{min}^{1-\lambda} \end{aligned}$$

The derivation for the left-hand side is very similar. We can now rewrite equation 10.6 and solve for x_f :

$$\begin{aligned} \int_{x_f}^{\infty} c \cdot x^{-\lambda} dx = f \cdot \int_{x_{min}}^{\infty} c \cdot x^{-\lambda} dx &\implies -\frac{c \cdot x_f^{1-\lambda}}{1 - \lambda} = -f \cdot \frac{c \cdot x_{min}^{1-\lambda}}{1 - \lambda} \\ x_f^{1-\lambda} &= f \cdot x_{min}^{1-\lambda} \\ x_f &= (f \cdot x_{min}^{1-\lambda})^{\frac{1}{1-\lambda}} \\ &= f^{\frac{1}{1-\lambda}} \cdot x_{min} \\ &= f^{-\frac{1}{1-\lambda}} \cdot x_{min} \\ &= \left(\frac{1}{f}\right)^{-\frac{1}{1-\lambda}} \cdot x_{min} \\ x_f &= \left(\frac{1}{f}\right)^{\frac{1}{\lambda-1}} \cdot x_{min} \end{aligned}$$

Now that we know x_f , we can calculate the expected degree of the leafs (i.e. all nodes from x_{min} up to x_f) and the hubs (i.e. all nodes after x_f). The expected value of any continuous distribution can be computed by the following expression:

$$E[X] = \int_{x_{min}}^{x_{max}} x \cdot f(x) dx \quad (10.7)$$

Let V_h and V_l be the expected degree for hub nodes and leaf nodes respectively. By applying equation 10.7 to our normalized power-law distribution, we can compute V_h and V_l in the following manner:

$$\begin{aligned} V_h &= \int_{x_f}^{\infty} x \cdot c \cdot x^{-\lambda} dx = \int_{x_f}^{\infty} c \cdot x^{-\lambda+1} dx \\ &= \frac{c}{-\lambda + 2} x^{-\lambda+2} \Big|_{x_f}^{\infty} \\ &= -\frac{c}{-\lambda + 2} x_f^{-\lambda+2} \end{aligned}$$

Computing V_l produces a bit more terms:

$$\begin{aligned}
 V_l &= \int_{x_{min}}^{x_f} c x^{-\lambda} dx = \int_{x_{min}}^{x_f} c x^{-\lambda+1} dx \\
 &= \frac{c}{-\lambda+2} x^{-\lambda+2} \Big|_{x_{min}}^{x_f} \\
 &= \frac{c}{-\lambda+2} (x_f^{-\lambda+2} - x_{min}^{-\lambda+2})
 \end{aligned}$$

We can use a similar approach to approximate the fraction E of all edges going to hubs. I.e. the fraction E of edges that end in a fraction f of all nodes. Note, that we know have to assume that the degree distribution follows a power law fo directed edges.

$$\begin{aligned}
 E &= \frac{V_h}{V_h + V_l} \\
 &= \frac{-\frac{c}{\lambda+2} x_f^{-\lambda+2}}{-\frac{c}{\lambda+2} x_f^{-\lambda+2} + \frac{c}{\lambda+2} (x_f^{-\lambda+2} - x_{min}^{-\lambda+2})} \\
 &= \frac{-x_f^{-\lambda+2}}{-x_f^{-\lambda+2} + x_f^{-\lambda+2} - x_{min}^{-\lambda+2}} \\
 &= \frac{x_f^{-\lambda+2}}{x_{min}^{-\lambda+2}} \\
 &= \frac{\left(\frac{1}{f}\right)^{\frac{1}{\lambda-1}} \cdot x_{min}}{x_{min}} \\
 &= \left(\frac{1}{f}\right)^{\frac{-\lambda+2}{\lambda-1}} \\
 &= f^{-\frac{-\lambda+2}{\lambda-1}} \\
 E &= f^{\frac{\lambda-2}{\lambda-1}}
 \end{aligned}$$

This expression was derived first in [40], and relates fraction edges to fractions of nodes. i.e. if we select an edge e_r at random, this approximation expresses the probability that it leads to a hub node. From this expression, the overall characteristics of networks with varying values of λ can be gleaned. For example, if we have a scale free network with $\lambda = 2.2$ and decide that the top 40 % are hub nodes, then approximately $0.4^{\frac{2.2-2}{2.2-1}} = 0.4^{\frac{0.2}{1.2}} \approx 0.858 \approx 86\%$ of all edges go to hub nodes, whereas only 14 % go to leaf nodes. Note that this expression is only valid for $\lambda \geq 2$.

Index

- Anti-message, 14
- Approximation Error, 84
- Average parallelism, 27
- Bottleneck LP, 40
- Busy Rollback, 98
- Cascaded rollback, 13, 14
- Causality error, 5
- Continuous simulation, 1
- Copy state saving, 14
- Degree distribution, 4, 20
- Degree exponent, 20
- Discrete Event Simulations, 1
- Embarrassing parallelism, 2
- EPHOLD, 29
- Epoch, 12
- Event, 2
- Event message, 4
- Experiment, 68
- Flushing method (GVT), 17
- Fossil collection, 15
- Future Event List, 3
- Generation, 43
- Global Virtual Time (GVT), 15
- Hubs, 20
- Idle Rollback, 98
- Leafs, 20
- Local Causality constraint, 3
- Local causality constraint, 5
- Local Virtual Time (LVT), 13
- Logical Process, 3
- Lookahead value, 7
- Lower Bound on TimeStamp (LBTS),
9
- Message Passing Interface, 68
- Out-of-order event, 5
- Outgoing Neighbors, 32
- Pareto distribution, 21
- PDES Graph, 4
- PDES graph, 32
- Physical Process, 3
- Power law distribution, 20
- Rollback, 13
- Saturated, 18
- Scale free network, 20
- Scenario, 68
- Sequential Discrete Event Simulation,
2
- Simulation event message, 4
- Snapshot (State), 14
- Space-time diagram, 31
- State, 1
- Straggler event, 13
- System, 1
- Termination time, 2

Transient message, 16

Unsaturated, 18

Wall-clock time, 15

Weight, 33

Work, 40

YAWNS, 12

Bibliography

- [1] ADEVS simulator. <http://web.ornl.gov/~1qn/adevs/>. Accessed: 21-11-2016.
- [2] WARPED2 simulator. <http://eecs.ceas.uc.edu/~paw/research/warped/>. Accessed: 27-10-2016.
- [3] Feature article—parallel discrete event simulation: Will the field survive? *ORSA Journal on Computing*, 5(3):213–230, 1993.
- [4] Philipp Andelfinger and Hannes Hartenstein. Model-based concurrency analysis of network simulations. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '15, pages 223–234, New York, NY, USA, 2015. ACM.
- [5] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 120–124, New York, NY, USA, 2004. ACM.
- [6] Barry C. Arnold. *Pareto and Generalized Pareto Distributions*, pages 119–145. Springer New York, New York, NY, 2008.
- [7] W. Bain and D. Scott. An algorithm for time synchronisation in distributed discrete event simulation. *Distributed Simulation*, pages 30–33, 1988.
- [8] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.
- [9] Peter D. Barnes, Jr., Christopher D. Carothers, David R. Jefferson, and Justin M. LaPre. Warp speed: Executing time warp on 1,966,080 cores. In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '13, pages 327–336, New York, NY, USA, 2013. ACM.

- [10] David W. Bauer Jr., Christopher D. Carothers, and Akintayo Holder. Scalable time warp on blue gene supercomputers. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, PADS '09, pages 35–44, Washington, DC, USA, 2009. IEEE Computer Society.
- [11] R. E. Bryant. Simulation of packet communication architecture computer systems. Technical report, Cambridge, MA, USA, 1977.
- [12] Christopher D. Carothers, David Bauer, and Shawn Pearce. Ross: A high-performance, low memory, modular time warp system. In *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation*, PADS '00, pages 53–60, Washington, DC, USA, 2000. IEEE Computer Society.
- [13] Christopher D. Carothers, Kalyan S. Perumalla, and Richard M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Trans. Model. Comput. Simul.*, 9(3):224–253, July 1999.
- [14] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Softw. Eng.*, 5(5):440–452, September 1979.
- [15] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4):198–206, April 1981.
- [16] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.
- [17] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.
- [18] Phillip M. Dickens, David M. Nicol, Paul F. Reynolds, Jr., and J. M. Duva. Analysis of bounded time warp and comparison with yawns. *ACM Trans. Model. Comput. Simul.*, 6(4):297–320, October 1996.
- [19] The Eclipse Foundation. Eclipse parallel tools platform (ptp), 2017.
- [20] Richard Fujimoto. Parallel and distributed simulation. In *Proceedings of the 2015 Winter Simulation Conference*, WSC '15, pages 45–59, Piscataway, NJ, USA, 2015. IEEE Press.

- [21] Richard Fujimoto. Parallel and distributed simulation. In *Proceedings of the 2015 Winter Simulation Conference, WSC '15*, pages 45–59, Piscataway, NJ, USA, 2015. IEEE Press.
- [22] Richard M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, October 1990.
- [23] Richard M. Fujimoto. Performance of Time Warp under synthetic workloads. In *Proceedings of 22nd SCS Multiconference on Distributed Simulation*, 1990.
- [24] Richard M. Fujimoto. Research challenges in parallel and distributed simulation. *ACM Trans. Model. Comput. Simul.*, 26(4):22:1–22:29, May 2016.
- [25] Richard M. Fujimoto, Kalyan S. Perumalla, Alfred Park, Hao Wu, Mostafa H. Ammar, and George F. Riley. Large-scale network simulation: How big? how fast? In *MASCOTS*, 2003.
- [26] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kam-badur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.
- [27] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.
- [28] Anurag Gupta, Ian Akyildiz, and Richard M. Fujimoto. Performance analysis of time warp with homogeneous processors and exponential task times. *SIGMETRICS Perform. Eval. Rev.*, 19(1):101–110, April 1991.
- [29] Jared S. Ivey, Brian P. Swenson, and George F. Riley. Phold performance of conservative synchronization methods for distributed simulation in ns-3. In *Proceedings of the 2015 Workshop on Ns-3, WNS3 '15*, pages 47–53, New York, NY, USA, 2015. ACM.
- [30] Shafagh Jafer, Qi Liu, and Gabriel Wainer. Synchronization methods in parallel and distributed discrete-event simulation. *Simulation Modelling Practice and Theory*, 30:54 – 73, 2013.

- [31] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, July 1985.
- [32] Hiroaki Kitano. Systems biology: A brief overview. *Science*, 295(5560):1662–1664, 2002.
- [33] Devendra Kumar. Simulating feedforward systems using a network of processors. In *Proceedings of the 19th Annual Symposium on Simulation*, ANSS '86, pages 127–144, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.
- [34] A.M. Law. *Simulation Modeling and Analysis*. McGraw-Hill series in industrial engineering and management science. McGraw-Hill, 2007.
- [35] B. D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Commun. ACM*, 32(1):111–123, January 1989.
- [36] Peter Luksch. A portable and extendible testbed for distributed logic simulation. In *Proceedings of the Conference on European Design Automation*, EURO-DAC '94, pages 368–373, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- [37] Viviana Mascardi. The reduction of the number of nullmessages in conservative lp-simulation engines. 1998.
- [38] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *J. Parallel Distrib. Comput.*, 18(4):423–434, August 1993.
- [39] Jayadev Misra. Distributed discrete-event simulation. *ACM Computing Surveys (CSUR)*, 18(1):39–65, 1986.
- [40] M. E. J. Newman. Power laws, Pareto distributions and Zipf's law. May 2006.
- [41] David M. Nicol. The cost of conservative synchronization in parallel discrete event simulations. *J. ACM*, 40(2):304–333, April 1993.
- [42] David M. Nicol. Global synchronization for optimistic parallel discrete event simulation. In *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation*, PADS '93, pages 27–34, New York, NY, USA, 1993. ACM.
- [43] James J. Nutaro. *Appendix B: Parallel Discrete-Event Simulation*, pages 296–330. John Wiley & Sons, Inc., 2010.

- [44] Kalyan S. Perumalla. Parallel and distributed simulation: Traditional techniques and recent advances. In *Proceedings of the 38th Conference on Winter Simulation*, WSC '06, pages 84–95. Winter Simulation Conference, 2006.
- [45] Kalyan S. Perumalla. Scaling time warp-based discrete event execution to 104 processors on a blue gene supercomputer. In *Proceedings of the 4th International Conference on Computing Frontiers*, CF '07, pages 69–76, New York, NY, USA, 2007. ACM.
- [46] Robert S. Pienta and Richard M. Fujimoto. On the parallel simulation of scale-free networks. In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '13, pages 179–188, New York, NY, USA, 2013. ACM.
- [47] Tomas Potuzak. Distributed-parallel road traffic simulator for clusters of multi-core computers. In *Proceedings of the 2012 IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications*, DS-RT '12, pages 195–201, Washington, DC, USA, 2012. IEEE Computer Society.
- [48] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2016.
- [49] Dhananjai M. Rao and Alexander Chernyakhovsky. Parallel simulation of the global epidemiology of avian influenza. In *Proceedings of the 40th Conference on Winter Simulation*, WSC '08, pages 1583–1591. Winter Simulation Conference, 2008.
- [50] George F. Riley, Mostafa H. Ammar, Richard M. Fujimoto, Alfred Park, Kalyan Perumalla, and Donghua Xu. A federated approach to distributed network simulation. *ACM Trans. Model. Comput. Simul.*, 14(2):116–148, April 2004.
- [51] Behrokh Samadi. *Distributed Simulation, Algorithms and Performance Analysis (Load Balancing, Distributed Processing)*. PhD thesis, 1985. AAI8513157.
- [52] J. William Schmidt and Robert Edward Taylor. *Simulation and analysis of industrial systems*, by J. W. Schmidt and R. E. Taylor. R. D. Irwin Homewood, Ill, 1970.

- [53] Georgos Siganos, Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. Power laws and the as-level internet topology. *IEEE/ACM Trans. Netw.*, 11(4):514–524, August 2003.
- [54] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.
- [55] Kimmo Soramäki, Morten L. Bech, Jeffrey Arnold, Robert J. Glass, and Walter E. Beyeler. The topology of interbank payment flows. *Physica A: Statistical Mechanics and its Applications*, 379(1):317 – 333, 2007.
- [56] Wen Su and Charles L. Seitz. Variants of the chandy-misra-bryant distributed discrete-event simulation algorithm. Technical report, Pasadena, CA, USA, 1988.
- [57] Yong Meng Teo and Seng Chuan Tay. Efficient algorithms for conservative parallel simulation of interconnection networks. In *Parallel Architectures, Algorithms and Networks, 1994.(ISPAN), International Symposium on*, pages 286–293. IEEE, 1994.
- [58] Jean G. Vaucher and Pierre Duval. A comparison of simulation event list algorithms. *Commun. ACM*, 18(4):223–230, April 1975.
- [59] Frederick Wieland. Practical parallel simulation applied to aviation modeling. In *Proceedings of the Fifteenth Workshop on Parallel and Distributed Simulation, PADS '01*, pages 109–116, Washington, DC, USA, 2001. IEEE Computer Society.
- [60] Wolfram Research, Inc. Mathematica 11.0.
- [61] Levent Yilmaz, Simon J. E. Taylor, Richard Fujimoto, and Frederica Darema. Panel: The future of research in modeling & simulation. In *Proceedings of the 2014 Winter Simulation Conference, WSC '14*, pages 2797–2811, Piscataway, NJ, USA, 2014. IEEE Press.
- [62] Srikanth B. Yoginath and Kalyan S. Perumalla. Parallel vehicular traffic simulation using reverse computation-based optimistic execution. In *Proceedings of the 22Nd Workshop on Principles of Advanced and Distributed Simulation, PADS '08*, pages 33–42, Washington, DC, USA, 2008. IEEE Computer Society.