Acta Cybernetica **25** (2021) 469–484.

A Parallel Event System for Large-Scale Cloud Simulations in DISSECT-CF*

Dilshad Hassan Sallo[ab] and Gabor Kecskemeti[acd]

Abstract

Discrete Event Simulation (DES) frameworks gained significant popularity to support and evaluate cloud computing environments. They support decision-making for complex scenarios, saving time and effort. The majority of these frameworks lack parallel execution. In spite being a sequential framework, DISSECT-CF introduced significant performance improvements when simulating Infrastructure as a Service (IaaS) clouds. Even with these improvements over the state of the art sequential simulators, there are several scenarios (e.g., large scale Internet of Things or serverless computing systems), which DISSECT-CF would not simulate in a timely fashion. To remedy such scenarios this paper introduces parallel execution to its most abstract subsystem: the event system. The new event subsystem detects when multiple events occur at a specific time instance of the simulation and decides to execute them either on a parallel or a sequential fashion. This decision is mainly based on the number of independent events and the expected workload of a particular event. In our evaluation, we focused exclusively on time management scenarios. While we did so, we ensured that the behaviour of the events should be equivalent to realistic, larger-scale simulation scenarios. This allowed us to understand the effects of parallelism on the whole framework, while we also shown the gains of the new system compared to the old sequential one. With regards to scaling, we observed it to be proportional to the number of cores in the utilised SMP host.

Keywords: cloud computing, parallel simulation, DISSECT-CF

1 Introduction

There are several obstacles that stop increasing the performance of DES frameworks. First of all, most are designed to execute sequentially. The need of simulat-

*This research was supported by the Hungarian Scientific Research Fund under the grant number OTKA FK 131793.

[a]Institute of Information Technology, University of Miskolc, Miskolc, Hungary

[b]E-mail: sallo@iit.uni-miskolc.hu, ORCID: 0000-0003-3672-4601

[c]Department of Computer Science, Liverpool John Moores University, Liverpool, UK

[d]E-mail: kecskemeti@iit.uni-miskolc.hu, E-mail: g.kecskemeti@ljmu.ac.uk, ORCID: 0000-0001-5716-8857

DOI: 10.14232/actacyb.287937

ing multiple events in parallel, is now essential for several scenarios. For example, simulating Internet of Things (IoT) involving millions or more devices worldwide; or simulating billions of service invocations and their interactions in serverless computing situations. Introducing a parallel approach to the core event handling in DESs aimed at distributed systems simulations would be the first step towards the support of the aforementioned scenarios.

Always applying parallel execution to the simultaneously occurring events in the simulation does not necessarily lead to a well scaling DES though. When only a few events occur simultaneously, sequential execution is often times beneficial as we can avoid the overheads of parallel constructs; otherwise, the parallel execution can lead to better performance. The necessity of determining at a specific simulated time instance, whether the events will execute sequentially or parallel manner is crucial to increase the performance and to avoid unnecessary overhead.

Despite several DESs support simulating parallel and distributed computing, the majority lack of parallel execution. For instance, Cloudsim [3] and GroundSim [14] execute computing tasks sequentially. This raises challenges when trying to simulate novel technologies (e.g., serverless) that require large scale simulation to be used as a support tool. DISSECT-CF [10] is one of the frameworks capable to simulate internal components and processes of distributed systems (ranging from cloud and fog infrastructures to even IoT systems). Although the execution time of DISSECT-CF is significantly faster than the most prominent simulator in the field CloudSim [13], this performance advantage is still not sufficient for the most demanding current research use cases (e.g., simulating millions of IoT devices and their continuum with clouds). Its sequential execution is a significant bottleneck, thus parallelisation is needed for scaling its performance efficiently to meet the newest challenges in the field.

This paper introduces a parallel execution mode for the event subsystem of DISSECT-CF. Our approach automatically switches between this new mode and the old one based on the number of simultaneous events that occur at a given time instance of the simulation. To avoid the overhead of applied parallel constructs under low workloads, our approach keeps using the original sequential mode for situations when only a few simultaneous events are detected. Otherwise, a parallel event executor will be selected. This executor divides and distributes the simultaneous events equally over the available processors and balances the load across the system. These two operations avoid idle CPUs (or cores) behind the simulator. To avoid issues with the initial event distribution, our parallel approach also uses work stealing to further reduce the contention among threads.

We have designed several experiments to evaluate the scalability and performance of the new approach. These experiments focus on core functionality and time management mechanisms of the event subsystem in DISSECT-CF. The evaluation had independent control on the following four properties: (*i*) event independence (no influence on future events); (*ii*) pattern of events throughout a simulation (i.e., how many events do we have in total and when should they happen); (*iii*) number of simultaneous events (degree of parallelism) happening at an average time-simulated instance; (*iv*) the single event workload (i.e., how compute heavy is a particular

event). We instrumented and measured the behaviour of realistic simulations in terms of these properties. Then, we implemented simple synthetic event patterns (that are only exercising the event subsystem of DISSECT-CF) for the simulator which we calibrated to imitate the properties of the previously measured realistic simulations. To ensure the quality of our experiments, we collected the synthetic event pattern's properties with the same measurement approach that we applied for the realistic setting to compare and analyse them. We also evaluated with random event patterns to test the behaviour of parallel version under unforeseen conditions.

We have executed our experiments on 12 core SMP hosts. Our experiments have been conducted with different degrees of parallelism and single event workload size. With respect to the number of cores, evaluation results show that two factors have affected the performance of the parallel version. First, if we have at least two simultaneous events for more than 50% of the simulated time instances, then the parallel version already runs two times faster than the sequential. Second, increasing the single event workload leads to 2.4 times faster simulation execution than sequential.

The remainder of this paper is structured as follows. In Section 2, we discuss work related to our approach. Section 3 discusses our methodology of employing parallelism in DISSECT-CF. Section 4 covers the evaluation of the parallel version. Finally, Section 5 concludes the paper and identifies future work.

2 Related work

Over the last decade, several DES frameworks have been designed to offer researchers an opportunity to evaluate and predict the behaviour of cloud computing applications. Each framework was designed with a specific purpose and having unique features that able solve some challenges.

CloudSim [3] is mostly used as general purpose cloud simulation environment. Due to the extensible nature of CloudSim, several extensions have been developed to integrate new features to it. DISSECT-CF [10] is a simulation framework that improved the modelling of resource-, network utilisation, power consumption and data centre configurations, by providing the capability of simulating IaaS internal behaviour. GroudSim [14] is a platform mainly focused on scientific application modelling (e.g., workflows) in cloud and grid computing. GreenCloud [11] is a simulator specifically built for estimating the energy consumption of cloud data centres. In addition to the above, the authors [1, 2] have conducted the detailed survey of over 33 simulators. Each of these is built for a specific purpose and is having unique features around cloud simulation. Although these simulators offer several features for cloud computing, the majority were built in a sequential fashion. Thus, they are all struggling to address recent challenges such as simulating millions of IoT devices.

Parallel discrete event simulation (PDES) approach has been applied in various fields such as simulation of networks, with the primary goal of performance. For example ROSS [4] and GWT [6] are parallel discrete event simulators that execute on

shared-memory multiprocessor systems. They mostly used in large-scale networking simulation models and telecommunication networks. DaSSF [12] is also parallel simulator targeting network simulation and it achieves high performance through parallel processing. Unfortunately, these frameworks have limited applicability in the research areas surrounding cloud computing. Parallelising existing systems remains a challenge [7]. Moreover, the prominent language for cloud simulators [1, 2] is Java, while current PDESs are not easily adoptable to this language. However, one of the frameworks called Cloud2Sim [9] supports concurrent and distributed simulations of clouds, based on the following libraries: Hazelcast, Infinispan and Hibernate.

In [7] the author raises many challenges that researchers could face in a PDES. One of these challenges is the complexity of using a parallel implementation correctly and simplifying code to understand it easily. Agreeing with this, our approach aims to keep the original sequential APIs while making a parallel solution in the background. In [8] the authors suggested the initial steps towards cloud supporting PDESs, unfortunately these steps were not yet adopted by current simulators. Introducing parallel execution to simulators needs easy simulation control as well as repeatable tests. In [5] the authors explained that sequential execution can be insufficient for modelling real complex systems, and parallel execution could manage resources efficiently. Sequential approaches are unable to fulfil many requirements, and lead to trade-off between the cost and performance. However, there is opportunity for applying parallelism to DISSECT-CF simulator to gain better performance. Introducing parallel executions of its event system can benefit all subsystems built on top of it. Finally, in [15] the authors showed a possibility to execute simulations over multiple virtual machines.

The previous works show the minimal advances towards parallel execution in cloud computing frameworks, which are needed to address the present challenges that accompanied modern technologies in this field. Therefore, our proposed solution provides parallel execution to the event system of DISSECT-CF simulator to speed up the simulation to foster simulating larger scale systems and technologies (e.g., IoT). Although the techniques proposed here are likely to be applicable to other frameworks as well, this paper is solely focused on DISSECT-CF to show our approach's applicability.

3 Methodology

DISSECT-CF simulator introduced substantial features to foster the rapid evaluation of IaaS clouds and its extensibility lead to support for other concepts such as IoT and fog computing. Although, DISSECT-CF reduces the execution time of equal quality/detail simulations done compared to several other frameworks in the field, it still does so in a sequential fashion. In the past, DISSECT-CF was shown simulating hundreds of thousands of computing entities within a few hours. But it has little chance to sequentially simulate recent systems within an acceptable time frame. With this research, we aim to set the foundations to support simula-

tions where the number of simulated computing components can easily reach over a billion of devices (like the IoT cloud continuum, or serverless computing).

3.1 Overview of DISSECT-CF simulator

DISSECT-CF is a simulation framework that offers insight into advanced cloud concepts supporting modern technologies. DISSECT-CF provides an amalgamation of several features that hardly exist in any previous simulators such as capturing low-level resource sharing behaviour and introducing an adequate energy consumption model. This aims to support previously problematic IaaS simulation scenarios that require all these advanced features to be available in the same framework.

The extensible core of the DISSECT-CF simulator consists of five major subsystems that mostly implement different concepts around clouds and distributed systems in a layered fashion [10]. Generally, each layer attempts to provide a comprehensive implementation for a particular concept without being dependent on the rest of the framework. The lowest subsystem, **event system** provides an appropriate mechanism to manage the behaviour of regular and irregular events as well as controlling the basic state of the simulation in a given time instance (so called tick in DISSECT-CF terminology). This subsystem is the foundation of all layers and introducing substantial features here such as parallelism has the highest potential impact on higher level subsystems. Next, **Unified resource sharing** subsystem introduces a holistic approach to establish a central resource provider able to share behaviour among low-level computing concepts. Then, the **Energy modelling** subsystem provides a unique approach that allows monitoring and analysing energy usage of all simulation resources by decoupling energy modelling from resource simulation (i.e., allows performance gains by only offering selective energy monitoring). On a layer above, the **Infrastructure simulation** subsystem deals with modelling the behaviour of typical distributed system components like virtual machines, physical machines, storage and networking. Finally, the highest layer of abstraction is provided in the **Infrastructure management** subsystem which contains major IaaS components such VM scheduler and PM scheduler that simulate the management of users requests and fosters the creation of custom internal IaaS behaviours. It also provides components such as Repository and the IaaS service to interact with users of the simulator.

Although the subsystems of DISSECT-CF have originally been written to execute sequentially, most of them can be executed in a parallel fashion as well. As all subsystems depend on the event subsystem, it comes as a natural point to adopt parallelism. As most of the operations in the higher level components are driven by events delivered from the event subsystem, these operations will be automatically parallelised with the parallelisation of the event subsystem itself.

3.2 Prominence of recurrent events

The lowest (event) subsystem of DISSECT-CF has two main classes: (*i*) the `Timed` class, used for recurring events; and (*ii*) `DeferredEvent` class used for irregular

events. Recurring events are events that the simulator invoked them regularly based on a specified frequency. Thus, recurring events can subscribe notifications, when subscribing, an event frequency must be specified to determine how many ticks must pass to get repeated notifications. As the other subsystems are built on the top of mostly recurring events, our target is enhancing the performance of this subsystem, which will reflect the outcome over the rest of the framework and its extensions. The event sub-system had a sequential execution design. Based on the existing API of DISSECT-CF, parallelisation could happen for executing of simultaneously happening events (i.e., events that should happen in the same time instance or tick of a simulation).

To understand such simultaneous events, we have provided a simple example scenario with three event objects with various frequencies (this demonstrates events derived from the `Timed` class of DISSECT-CF which allows defining events that can happen repeatedly). Table 1 shows the basic details of our simple example scenarios. The first three rows show the event objects and their behaviour. The last line shows the time instances in our simple simulation. In the table, we can see for every time instance when the events will be processed. E.g., the second event ($e2$) is processed in time instances 3,6 etc. Figure 1 shows how the event queue will look like at any particular time instance in case we execute the events defined in the previous table.

The degree of parallelism denotes the number of events that happens at a specific time instance (tick). Which mainly depends on the frequencies of subscribed objects

Table 1: Three events with different frequencies.

Events	Freq	Next events of e1, e2 and e3 based on their frequencies(Freq)														
e1	1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
e2	3	3	3	3	6	6	6	9	9	9	12	12	12	15	15	15
e3	5	5	5	5	5	5	10	10	10	10	10	15	15	15	15	15
Time		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Degree(%)		33	33	66	33	66	66	33	33	66	66	33	66	33	33	100

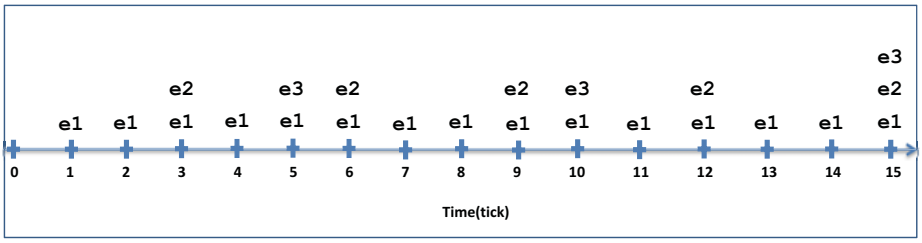


Figure 1: Representing multiple events in Table 1 occur at a specific time

that determine how frequently these events occur. When all subscribed events happen at a specific time, the degree of parallelism is 100%. When half of them occurs at one time, the degree is 50% and so on. Thus, the degree of parallelism varies according to the occurrence of events at each tick. Therefore, the average degree of parallelism in a single simulation run is deduced from all ticks for the whole system. In Figure 1, there are 15 simulated time instances, out of these 7 are having parallel events, making the example's average degree of parallelism 50.66%. If we execute simultaneously occurring events (e.g., $e1$ and $e2$ in time instance 3 in the Figure) in a sequential fashion, then we pay a penalty of using a sequential simulator. This observation will guide the next the sub-section where we discuss how we identify these kinds of events and how we execute them in parallel.

3.3 The parallelisation of simultaneous events

Figure 2 shows the basis of our extension. The diagram shows only the relevant parts of the original `Timed` class, and the new `Parallel` class. The `Parallel` is created as an inner class within `Timed` class, to ensure easy access to the original data structures within the event subsystem's main class. The user of the system is still expected to interface with the existing methods of the `Timed` class (thus all previous extensions to the simulator would benefit from our parallelisation approach). Note that inside the simulator, all higher level subsystems (e.g., those which simulate virtual machines) are considered as users of the `Timed` class. As parallelisation is automatically executed depending on the state of the event queue, the higher level subsystems benefit from the improvement on `Timed`.

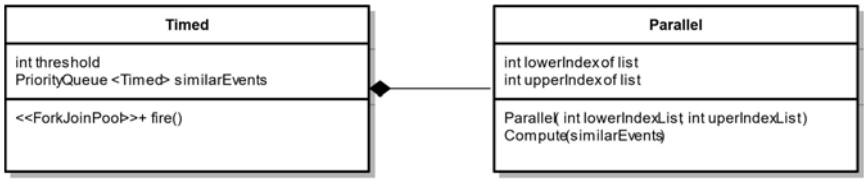


Figure 2: Diagram of Timed class and Parallel class

In DISSECT-CF, time is measured in ticks [10] and users of the simulator are free to interpret ticks the way they want. The events taking place in a particular tick are handled with the `fire()` method (see Figure 2). Our approach changes the behaviour of this method by introducing Algorithm 1. Here we first collect the list of simultaneously occurring events at each particular tick (see line 2) – note that this list was not needed for the sequential sub-system as that would only work with one event at a time. As a result, the collection of this list is an overhead of the new parallel algorithm. The discussed approach below aims at minimising this overhead.

Our new `fire()` method now checks the size of the list to determine if we need to execute in sequential or parallel fashion. The old, sequential execution is shown in the loop of line 4, this is still kept and used if we have too few simultaneous events

Algorithm 1 Determining the need for parallelism

```
1: threshold = specified size
2: list = all simultaneous events
3: if list.size <= threshold then
4:    while list.notEmpty do
5:       event = get single event from list
6:       Execute event
7:    end while
8: else
9:    invoke Parallel(list.lowIndex, list.upperIndex)
10: end if
```

in the queue. The parallel execution utilises our new `Parallel` class to distribute the work over threads, that will be created implicitly according to the number of available cores. This is done by passing the `lowerIndex` and `upperIndex` that specify the indices of first and last elements of the list (see line 9). The `threshold` (minimal size of the list which leads to parallel execution) is configurable by the user of the simulator. To aid the user determining the threshold, an auto-tuning approach is also going to be offered for the threshold which determines its value when suitably long running simulations are executed. The auto-tuning approach bases its decision on the threshold on the typical single event workload. As the auto tuning approach also needs some compute time it is possible to disable it for simulations where the threshold is known to the user.

Algorithm 2 Mechanism of Parallel class

```
1: Procedure Parallel( list.lowIndex, list.upperIndex )
2: lowerIndex = list.lowIndex
3: upperIndex = list.upperIndex
4: Funct compute ()
5: if  upperIndex - lowerIndex <= threshold then
6:    while  list.notEmpty do
7:       Execute events of list
8:    end while
9: else
10:   midIndex = (lowerIndex + upperIndex / 2)
11:   invoke all (Parallel(lowIndex, midIndex), Parallel(midIndex, upperIndex))

12: end if
```

After the decision to parallelise, the actual parallelisation is organised by the `Parallel` class according to Algorithm 2. Instances of this class are executed in their own threads. Thus, they will likely run on another CPU core compared to the original `fire()` method. When a `Parallel` instance is instructed to compute, it again uses our the previously discussed and determined `threshold` value to decide if the workload assigned to the thread is sufficiently small or not.

If the sublist of simultaneous events is short enough (see line 5), the sublist is executed in the current thread entirely. This sublist execution is done just like the sequential one was discussed before (see Algorithm 1's line 4). But instead of going through the entire list of simultaneous events, now we have a shorter list to process which was assigned only to the thread of this `Parallel` object in the parallel invocations of Algorithms 1 and 2.

In contrast, when there are more simultaneous events than a single thread should handle, we sub-divide the list of events based on its size in equal parts and pass them on to further threads (see line 11). We repeat this process until the list of events divided into sublists (sublists size become less than or equal threshold) and all threads have sufficiently short lists, then the threads are scheduled according to a fork-join model. This list division method ensures that we execute on all available processors in the current machine and also offers an initial load balance. The fork-join model uses work-stealing algorithm by allowing the thread to steal workloads from others. Although each thread has an almost equal number of sublists, work-stealing approach ensures that the threads workloads are almost equal to avoid wasting time.

4 Evaluation

A private cloud at LJMU was used for the evaluation of our parallel DISSECT-CF. For our experiments, we used a VM with the following specifications: Intel (R) Core (TM) i7-8700 CPU @ 3.2GHz (6 cores + 6 hyper threaded cores), 64GB memory, 1T SSD, 1T HDD, Debian Linux Buster 10.4, OpenJDK 11.0.6. We have designed several scenarios to test the performance of the parallel version by focusing on time management while ensuring complete control over event occurrence. We also made sure the evaluation was validating the parallel version: we used the complete API of the `Timed` class to verify if the parallel version produces results matching output from the unmodified sequential code.

4.1 Validation

To ensure that the behaviour of our evaluation is following real life simulation patterns, we have instrumented the `JobDispatchingDemo` class of the dissect-cf-examples project. This class was already validated before to produce realistic simulations e.g., comparable to CloudSim (see [10]). Our instrumentation focused on how the realistic simulation utilises the lowest abstraction layer of DISSECT-CF. We measured, the degree of parallelism, the typical event behaviour, the number

of events in total and the average execution time of a single tick method call in nanoseconds (i.e., the single event workload). To enable the comparison, we have also instrumented our parallel `Timed` class in the same way allowing us to acquire the typical workload of our synthetic tick methods.

We have set up our realistic simulation with `JobDispatchingDemo` as follows: (*i*) maximum number of jobs that exist in parallel was set to 2; (*ii*) the amount of seconds the job startup times was set to 10; (*iii*) minimum execution time of a single job was set to 10s; (*iv*) maximum execution time of a single job was set to 90s; (*v*) minimum and maximum gaps between the last and the first job submission of two consecutive parallel batches were set to 200s; (*vi*) minimum number of processors for a single job was set to 1; (*vii*) maximum number of processors for a single job was set to 2; (*viii*) total number of processors usable by all parallel jobs was set to 4; (*ix*) total number of jobs was 100000; (*x*) the number of nodes was 5000.

To allow our evaluation to focus at the lowest abstraction layer (and our parallelism evaluations not to be distracted by upper layer behaviour), we set out to capture the event workload behaviour of the above complex simulation, but with a synthetic workload. Our synthetic tick method (implemented in the class `TimeRandomGenerator`), does a busy waiting loop by calculating the following formula:

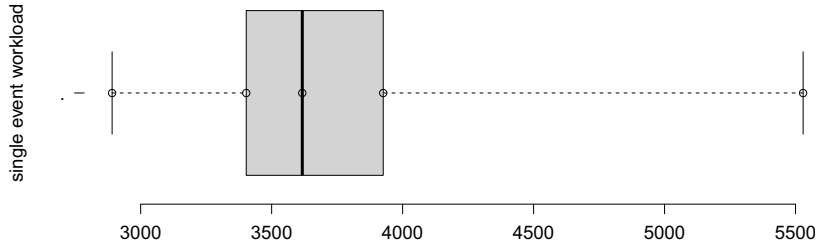
$$SyntheticEventWorkload(size) := \sum_{i=0}^{size} \left(2e^i\sqrt{i}\right) \mod \left|\left\lfloor \frac{i+5}{i+1} \right\rfloor\right|, \qquad (1)$$

where *size* can control the single event workload, while the denoted operations ensure that the distribution of the single event execution time is closely matching the above mentioned more realistic simulation.

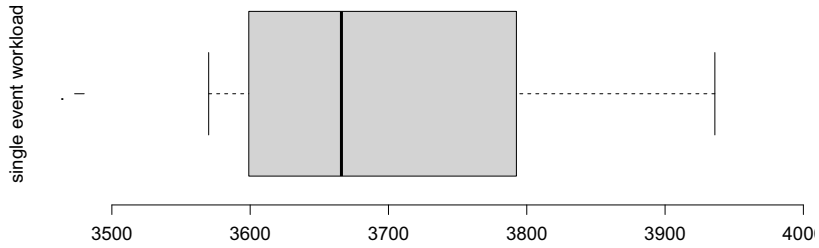
To ensure that the workload produced by this busy waiting loop is equivalent to the realistic simulation, we have executed the same number of events we have recorded in the realistic simulation and repeated the measurement 100 times. The repetition allowed us to collect several statistical properties of the single event workload in both the synthetic and the realistic simulations. We present our findings for the realistic simulation in the box plot of Fig 3a. Our best approximation of this realistic workload was captured by our synthetic workload parametrised with $size = 49$.

Fig 3b shows the behaviour of our best approximate synthetic workload. Our median duration is within 3% of the realistic. The distribution of our workload is a bit narrower and more even, but the upper and lower whiskers of our synthetic experiment are within the typical range of the realistic simulation's values. As a result, from this point onwards, we will refer to synthetic workloads set up with this particular parameter as the *original* single event workload.

Note, that later we have evaluated the system with other single workloads. For example, changing the *size* to 147, leads to a threefold increase in single event workload compared to the realistic setting. In contrast, changing it to 16, leads to a three fold reduction in single event workload again compared to the realistic setting. These two values will be the extremes used in Figure 5.



(a) The execution time(ns) of single event workload using JobDispatchingDemo class



(b) The execution time(ns) of single event workload using our experiments classes

Figure 3: Boxplot diagrams for JobDispatchingDemo class and our classes

4.2 Performance

Our evaluation scenarios create 35,000 recurrent event objects. The object count was set so the minimum execution time of the sequential version is at least 5 minutes, allowing sufficient time for the parallelisation to take effect. The recurring events subscribe with different frequencies so we have control over the degree of parallelism. We provided controls to these scenarios, so we can easily adjust the degree of parallelism (through event subscription changes) and the single event workload (through changing the *size* in Equation 1). The evaluation scenarios are publicly available in the ParallelTimed package released in the dissect-cf-examples project on GitHub[1] using the GPL 3.0 license.

The invocation of `Parallel` class depends on the threshold value (see Algorithm 1 for details) to determine the maximum length of the event list processed by a single thread. To determine the ideal setting for the threshold, we evaluated our solution with four different values: 8, 16, 32 and 64. We have also generated recurring events with four different degrees of parallelism as shown in table 2. Based on our analysis of the execution times in the table. Even though the differences are not big, it is recommended to use a threshold equal or exceed 32 to enhance the performance.

With the respect to the number of cores, there are two factors that influence the performance of the parallel version. First, the degree of parallelism plays a

[1]https://github.com/dilshadsallo/dissect-cf-examples

Table 2: The execution time(s) of parallel version using four different sizes of list

Degree of Parallelism				
Threshold	**25%**	**50%**	**75%**	**100%**
8	235	412	549	657
16	234	411	548	657
32	231	408	545	654
64	229	406	541	652

significant role and it is shown in Fig 4 that the parallel version can significantly improve performance. We evaluated both the parallel and sequential versions of the simulator with four different degrees of parallelism (25%, 50%, 75%, 100%). Even though the evaluation of this scenario has been done with the same number of aforementioned objects, the number of events that occur, and the number of events that occur at the same time significantly increase. This is because we simulated for the same amount of simulation time, but with increasing subscription frequency each object receives more event notifications. E.g., to increase the degree of parallelism on the scenario in table 1, we can change the subscription frequency of event 2 to 1. In this example, the degree of parallelism increases to 73%, but we see more event notifications delivered as we will have 15 notifications for event 2 as well.

With regards to Fig 4, in 25% of parallelism, the parallel version runs 1.72 faster than the sequential. When the degree reaches 50%, the ratio increased to 1.74. The parallel version executes simulations 1.84 faster than the sequential version when 75% of all subscribed events occur recurrently during a simulation time. Finally, the parallel version reaches 2 times faster than the sequential version when the degree of parallelism is 100%. Even with a high degree of parallelism and using multi-core, we cannot use all cores because there is still a performance cost such as coordinating threads that introduced by multi-thread compared to a single-threaded approach.

Now let's analyse the effect of the size of the single event workload (as per Equation 1). We tested both of the parallel and sequential versions with various single event workload sizes, commenced with threefold lower than the *original* one to show the behaviour of simulating very low single event workload. Then reaching to threefold higher than the *original* single event workload to demonstrate the advantage of parallel version as shown in Fig 5. When the single event workload is threefold lower than *original* one, the parallel version runs 1.2 faster than the sequential version. This ratio increases to 1.6 when the single event workload is two times lower than the *original* single event workload. The parallel version even runs 2.1 faster than the sequential version using *original* single event workload. When the single event workload size doubled, the parallel version executes the simulation 2.3 times faster than the sequential version. The ratio increases to 2.4 when the single event workload size becomes threefold higher than the *original* one.

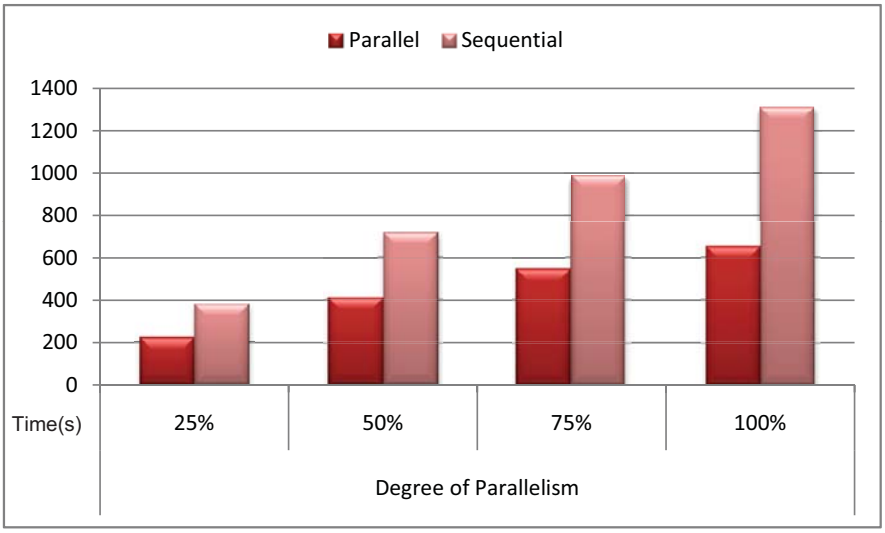


Figure 4: The execution time(s) of parallel and sequential versions in four different degrees of parallelism

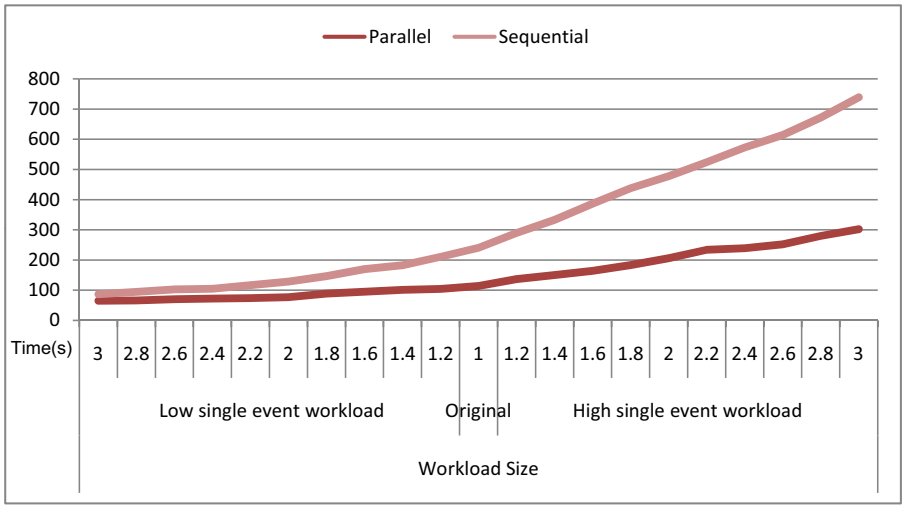


Figure 5: The execution time (s) of different workload sizes simulated by parallel and sequential versions

Thus, the parallel version speeds up the performance of simulation by using the additional cores of the host. The biggest advantages of the parallel version can be exploited when there are larger simultaneously occurring event queues and when the single event workload is larger as well. In the realistic simulations, we have seen that simultaneously occurring event queues are typically larger when advanced features of the simulator are fully utilised (e.g., cloud wide energy metering or virtual machine consolidation).

5 Conclusion and future work

Mostly DES frameworks are used to simulate and evaluate cloud computing environments. The majority executes sequentially. DISSECT-CF is one of the frameworks that brought several features to improve the performance of IaaS simulation. It is built to accompany the latest technology with easy extensibility. In terms of execution, DISSECT-CF was already fast and reliable but still targeted a single core. We devised a parallel version to handle this issue focusing on the use of multi-core when simultaneous events happen in the simulation. The parallel version scales well and leads to significant speed up. The performance of the parallel version is dependent on the number of simultaneous events at a particular time instance in the simulation, as well as on the workload a single event's processing causes. Our introduced parallel execution mode is focused on the event subsystem, as this is the lowest layer in DISSECT-CF all other components benefit from our improvements.

Future work will focus on the simulator's second most heavily used component: the unified resource sharing subsystem. As this subsystem is having high compute complexity, its parallelisation will enable the rapid estimation of resource sharing on even larger scale distributed systems. Applying these will lead to the seamless transition of DISSECT-CF into simulating more communication intensive systems, or evaluating fog computing & IoT device behaviour.

References

[1] Ahmed, Arif and Sabyasachi, Abadhan Saumya. Cloud computing simulators: A detailed survey and future direction. In *2014 IEEE International Advance Computing Conference (IACC)*, pages 866–872. IEEE, 2014. DOI: 10.1109/IAdCC.2014.6779436.

[2] Byrne, James, Svorobej, Sergej, Giannoutakis, Konstantinos M, Tzovaras, Dimitrios, Byrne, Peter J, Östberg, Per-Olov, Gourinovitch, Anna, and Lynn, Theo. A review of cloud computing simulation platforms and related environments. In *International Conference on Cloud Computing and Services Science*, volume 2, pages 679–691. SCITEPRESS, 2017. DOI: 10.5220/0006373006790691.

[3] Calheiros, Rodrigo N, Ranjan, Rajiv, Beloglazov, Anton, De Rose, César AF, and Buyya, Rajkumar. Cloudsim: a toolkit for modeling and simulation

of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011. DOI: 10.1002/spe.995.

[4] Carothers, Christopher D, Bauer, David, and Pearce, Shawn. Ross: A high-performance, low-memory, modular time warp system. *Journal of Parallel and Distributed Computing*, 62(11):1648–1669, 2002. DOI: 10.1016/S0743-7315(02)00004-7.

[5] D'Angelo, Gabriele and Marzolla, Moreno. New trends in parallel and distributed simulation: From many-cores to cloud computing. *Simulation Modelling Practice and Theory*, 49:320–335, 2014. DOI: 10.1016/j.simpat.2014.06.007.

[6] Das, Samir, Fujimoto, Richard, Panesar, Kiran, Allison, Don, and Hybinette, Maria. Gtw: a time warp system for shared memory multiprocessors. In *Proceedings of Winter Simulation Conference*, pages 1332–1339. IEEE, 1994. DOI: 10.1109/WSC.1994.717527.

[7] Fujimoto, Richard M. Research challenges in parallel and distributed simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 26(4):1–29, 2016. DOI: 10.1145/2866577.

[8] Fujimoto, Richard M, Malik, Asad Waqar, Park, A, et al. Parallel and distributed simulation in the cloud. *SCS M&S Magazine*, 3:1–10, 2010.

[9] Kathiravelu, Pradeeban and Veiga, Luis. Concurrent and distributed cloudsim simulations. In *2014 IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 490–493. IEEE, 2014. DOI: 10.1109/MASCOTS.2014.70.

[10] Kecskemeti, Gabor. Dissect-cf: a simulator to foster energy-aware scheduling in infrastructure clouds. *Simulation Modelling Practice and Theory*, 58:188–218, 2015. DOI: 10.1016/j.simpat.2015.05.009.

[11] Kliazovich, Dzmitry, Bouvry, Pascal, and Khan, Samee Ullah. Greencloud: a packet-level simulator of energy-aware cloud computing data centers. *The Journal of Supercomputing*, 62(3):1263–1283, 2012. DOI: 10.1007/s11227-010-0504-1.

[12] Liu, Jason, Nicol, David, Premore, Brian, and Poplawski, Anna. Performance prediction of a parallel simulator. In *Proceedings Thirteenth Workshop on Parallel and Distributed Simulation. PADS 99.(Cat. No. PR00155)*, pages 156–164. IEEE, 1999. DOI: 10.1109/PADS.1999.766172.

[13] Mann, Zoltán Ádám. Cloud simulators in the implementation and evaluation of virtual machine placement algorithms. *Software: Practice and Experience*, 48(7):1368–1389, 2018. DOI: 10.1002/spe.2579.

[14] Ostermann, Simon, Plankensteiner, Kassian, Prodan, Radu, and Fahringer, Thomas. Groudsim: an event-based simulation framework for computational grids and clouds. In *European Conference on Parallel Processing*, pages 305–313. Springer, 2010. DOI: 10.1007/978-3-642-21878-1_38.

[15] Yoginath, Srikanth B and Perumalla, Kalyan S. Optimized hypervisor scheduler for parallel discrete event simulations on virtual machine platforms. In *SimuTools*, pages 1–9, 2013. DOI: 10.4108/icst.simutools.2013.251736.