remote sensing

MDPI

Article

SparkCloud: A Cloud-Based Elastic Bushfire Simulation Service

Saurabh Garg [1,*,†], Nicholas Forbes-Smith [1], James Hilton [2] and Mahesh Prakash [2]

[1] School of Technology, Environments and Design (TED), University of Tasmania, Sandy Bay, TAS 7005, Australia;

[2] Data61, Eveleigh NSW 2015, Australia; James.Hilton@data61.csiro.au (J.H.); Mahesh.Prakash@data61.csiro.au (M.P.)

* Correspondence: Saurabh.Garg@utas.edu.au

† Current address: Discipline of ICT, School of TED, UTAS, Sandy Bay, Tas 7005, Australia

Received: 31 October 2017; Accepted: 20 December 2017; Published: 7 January 2018

Abstract: The accurate modeling of bushfires is not only complex and contextual but also a computationally intensive task. Ensemble predictions, involving several thousands to millions of simulations, can be required to capture and quantify the uncertain nature of bushfires. Moreover, users' requirement and configuration may change in different situations requiring either more computational resources or modeling to be completed with a stricter time constraint. For example, during emergency situations, the user may need to make time-critical decisions that require the execution of bushfire-spread models within a deadline. Currently, most operational tools are not flexible and scalable enough to consider different users' time requirements. In this paper, we propose the SparkCloud service, which integrates features of user-defined customizable configuration for bushfire simulations and scalability/elasticity features of the cloud to handle computation requirements. The proposed cloud service utilizes Data61's Spark, which is a significantly flexible and scalable software system for bushfire-spread prediction and has been used in practical scenarios. The effectiveness of the SparkCloud service is demonstrated using real cases of bushfires and on real cloud computing infrastructure.

Keywords: bushfires; ensemble predictions; cloud computing; deadline-based resource allocation

1. Introduction

Bushfires are known for their incredibly destructive potential. With the current increase in urbanization and changing climate conditions, these fires are increasingly becoming a major threat. To minimize the impact of these fires, it is important to understand the dynamics of bushfires and to be able to predict their propagation. A common way to study them is to use fire prediction models and discrete event simulators [1]. There are several operationally used simulators, such as Phoenix [2], Australis [3], FarSite [4] and Spark [5]. Despite there being so many simulation modeling tools available, it is still a computationally challenging task, as bushfires are a very complex phenomenon—involving interactions with several factors, including vegetation, climatic condition and altitude at very broad temporal and spatial scales. These include considerations that range from chemical reactions at the molecular scale, through the micro-scale physics of pyrolysis, to the macro-scale turbulent interactions with the surrounding atmosphere [5,6]. Ensemble predictions, involving several thousands to millions of simulations, can be required to capture and quantify the uncertain nature of bushfires. In other words, in real-world scenarios, these simulations can take several hours or days before one can predict bushfire spread precisely. Current bushfire simulators are designed to either work on desktop machines or Graphics Processing Units (GPUs) [7], which cannot be changed, reconfigured or scaled in a timely manner according to the requirements of an end user. However, during emergencies, time is critical.

The early detection and spread path of bushfires cannot only save many lives but can also reduce the loss of natural and urban resources.

Because of these reasons, several researchers have begun to see cloud computing [8–11] technology as a solution to process large remotesensing data, using frameworks such as Hadoop in Cloud [12] in a scalable and cost-effective manner. Cloud computing provides elastic and on-demand access to an almost infinite storage, network and computational resources. Thus, cloud computing infrastructure can provide on-demand resources to manage simulation time constraints. However, cloud computing does not automatically improve the efficiency of the execution of bushfire simulation or provide support to meet specified time constraints. There is no resource allocation system that can adapt the underline cloud resources according to the computational requirements of the simulation and volume of data that needs to be processed. There has been very little work done in this context [7,13]. Kalabokidis et al. [13] introduced a fire simulation model to the cloud computing infrastructure. Garg et al. [7] provided a conceptual model for providing a scalable fire prediction service using cloud computing resources. However, these works do not integrate any real-world operational simulator or adapt the cloud resources on the basis of given time constraints (or deadlines).

Therefore, to fill this gap, we propose the SparkCloud service—a Web-based cloud-platform system. This service utilizes Data61's Spark wildfire simulator [5] for executing the user's fire-spread simulations. Spark is the most flexible software available from a scalability and ensemble modeling perspective and thus is very well suited to cloud computing. It also allows for the integration of different fuel and propagation models. An independent study conducted by the Bureau of Meteorology [14] analyzed different mature simulators, such as Phoenix [2] and Australis [3], and concluded that Spark is the most versatile from the perspective of scalability for ensemble simulation runs, the ability to incorporate new fuel models and computational transparency. The proposed SparkCloud services provide a Web interface for easy configuration of bushfire simulation parameters and users to specify time constraints. The users can easily upload configuration and other input files through a drag-and-drop interface. The SparkCloud enables automatic initiation and configuration of appropriate virtual machines (VMs) that can run the required simulation with the specified deadline. When the number of simultaneous requests increases, the SparkCloud auto-scales the number of workers needed to serve these requests. The evaluation of the proposed system, using two real bushfire case studies, proves the system's effectiveness in operational scenarios.

In the next section, we give a brief overview of the Spark simulation framework, and in the subsequent sections, we describe the design and implementation of SparkCloud. In Section 5, we evaluate the performance of SparkCloud using real case studies and on a real cloud infrastructure. In the final section, we present conclusions and future directions.

2. Spark: Introduction

Spark [5] is a computational framework for modeling the spread, behavior and risk of wildfires. The core of Spark is a robust two-dimensional propagation algorithm that models the progression of a fire perimeter over time. The progression of the perimeter is entirely user-defined, allowing for different rates-of-spread for the fire in various fuel types and fire conditions. The propagation algorithm is based on the level set method, allowing the simulation of any number of distinct fire perimeters, spot fires and coalescence between different parts of the fire as the fire evolves.

Fire simulations require a number of input datasets to operate. These include maps of the land classification or fuel type, topography (as fires spread faster uphill), fuel information and weather data. Some of these datasets can be highly specialized, for example, grass dryness levels, and may only apply to regional areas. Furthermore, different fire-spread models are used in different areas. Spark implements the GDAL (Geospatial Data Abstraction Library) library, allowing various raster layers representing these inputs to be read. These layers can then be used and combined in any manner using scripts to define the rate-of-spread of the fire in different fuel types. The system handles all temporal

and spatial interpolation of these input layers, allowing the user to write these scripts in a clear and straightforward manner.

The framework also supports a number of plug-in packages including models for the generation and topographic correction of wind fields, firebrand transport and ignition models, road and transmission line crossing models and fire-line interaction models. All calculations within Spark are parallelized on GPU architecture, enabling simulations to run much faster than in real time. This faster-than-real-time capability is crucial for the operational prediction of fire spread, in which multiple fire-spread scenarios using the latest weather predictions can inform fire suppression and evacuation operations. Alternatively, Spark can be used in an offline capacity to calculate the risk from fire by carrying out massive ensembles of simulations and reducing the results to a given risk metric. The deployment of Spark to calculate risk in this way can be used to inform stakeholders of potential fire risk to sensitive assets or to guide fire reduction strategies.

3. SparkCloud Architecture

As previously discussed, different users may run fire-spread simulations with different input parameters, such as input data or spread models, and also with different time constraints. These different situations will require the selection of appropriate computational resources. The SparkCloud service has been designed to satisfy these requirements using scalable resources of the cloud. Figure 1 shows the different architectural components of SparkCloud, which have been developed using the master-slave model and REST Application Programming Interfaces (APIs) [15] to handle user interaction. At the high level, the user will submit their deadline requirements, with configuration and input files, to the SparkCloud user interface (SparkJobUI). SparkCloud's different components will interact with each other to run Spark with the given configuration—using the best possible execution plan for the Spark application to satisfy the user's requirements. After the completion of the simulation, output files can be downloaded by the user using the REST APIs (Application Program Interface) (or the SparkJobUI).

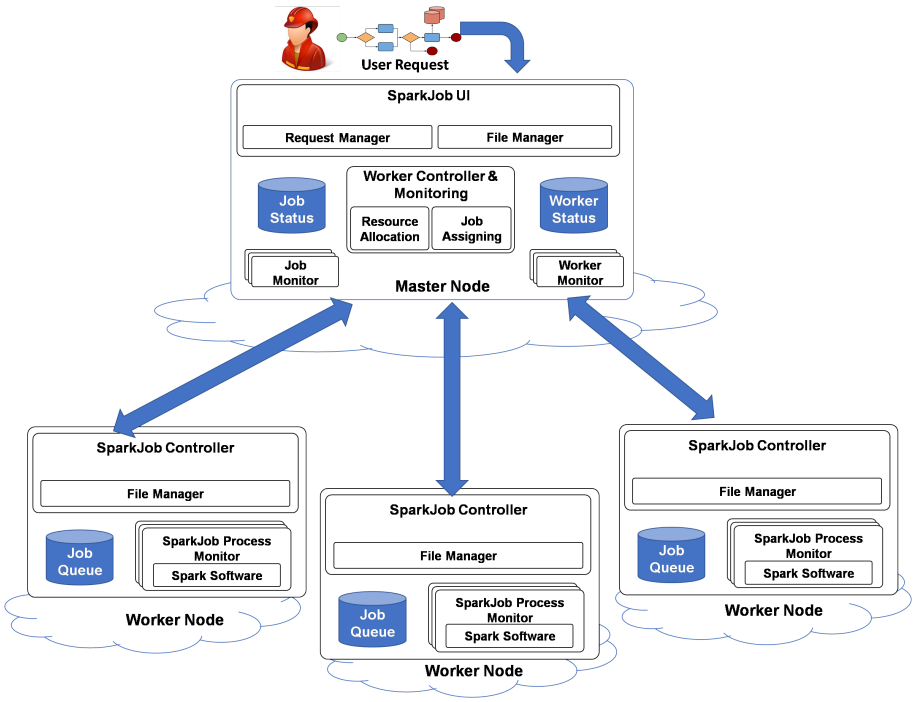


Figure 1. SparkCloud architectural components for master node and worker nodes. Master node manages user request allocations to different workers, which manage the execution of the simulation model on the basis of the request.

3.1. Spark Master Components

The master node plays the key role in interacting with users to provide regular updates on their requests and to manage the different workers to complete the requests while adhering to their requirements. The basic functionality of the different components is outlined below:

- **Request Manager:** The request manager defines job configurations on the basis of the user's request and input parameters. It also defines the job requirements on the basis of the user's deadline and the computational requirement of the bushfire-spread model (on the basis of input parameters). Requests are served on a first-come first-served basis. The jobs are then are passed to the Worker Controller for resource allocation and job submission.
- **File Manager:** The file manager handles job files such as input/output files and configuration files. All file uploads/downloads are done using the REST API.
- **Job Monitor:** Each job has its own monitor service that contains all state-related information of the job. The job monitor ensures that the job completes each stage of execution successfully and that it meets its deadline (see Figure 2).
- **Worker Controller and Monitoring:** This component manages worker nodes and coordinates resource allocation and the submission of jobs. Depending on the requirement of a job, either a new worker (VM) will be initiated or an already-running worker will be selected for submission. In either case, the job will be assigned to a worker that can execute the job within the user-specified deadline. The Worker Controller and Monitoring component is also responsible for handling load balancing. When a worker becomes free, jobs with the tightest deadline (and in a waiting state) will be re-assigned.
- **Worker Monitors:** "Worker Monitors" are used to monitor each worker's status and the cloud instance (VM) state and to handle communication with the worker. The only direct master-to-worker communication required is for job submission—all other communication is performed using the master's REST API (which is all worker-to-master).

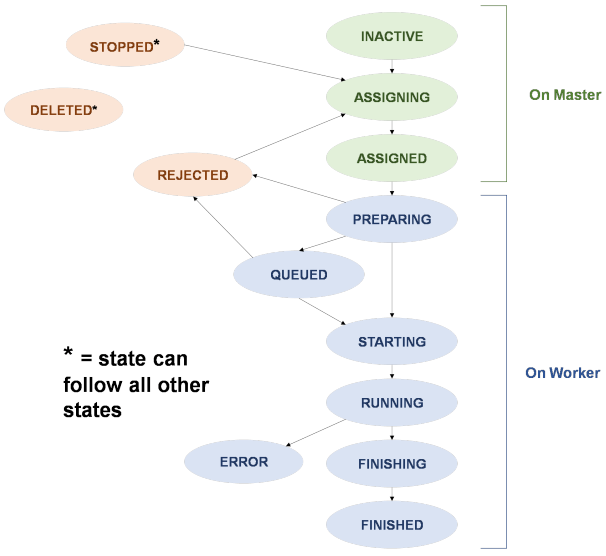


Figure 2. State diagram.

3.2. Spark Worker Components

The worker components are responsible for coordinating job execution, after the jobs have been submitted to a worker node.

- **SparkJobController:** The SparkJobController handles the following: receiving job submissions, managing job files (input/configuration and output files)—through the File Manager—queueing

jobs, coordinating job execution, and notifying the "Worker Monitor" (on the master) of changes in the workers' status.

- **File Manager:** The File Manager ensures that job config files and input files are successfully downloaded from the master before job execution and that job output files are successfully uploaded to the master after a job has finished.
- **SparkJob Process Monitor:** A "Process Monitor" is used to monitor a job's execution. It listens to Spark's standard output and error ports to capture log information and to watch for errors, and it will also monitor the process status to confirm that it finishes without error. The component also notifies the "Job Monitor" (on the master) of changes in the job status (see Figure 2).

4. SparkCloud Design and Implementation

In this section, we provide finer details about the design of the SparkCloud service. Figure 3 shows the relationship between its different fundamental classes, each of which correspond to the different architectural components of SparkCloud. Firstly, we describe the design considerations and how they have been implemented in the SparkCloud system through sequence diagrams, which show the interaction between different classes of the SparkCloud service.

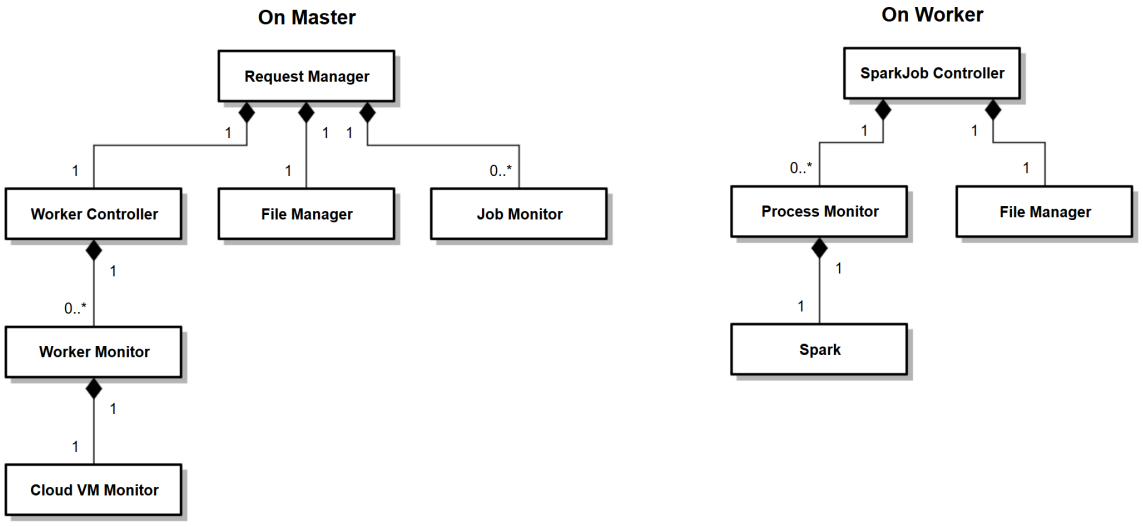


Figure 3. SparkCloud class diagram.

4.1. Design Considerations

The SparkCloud service was implemented with the following considerations in mind:

- **Deadline-Based Execution:** As previously discussed, the end user may have a time constraint on the requested execution of a bushfire simulation model. Thus, the system needs to estimate the execution time of the requested simulation job and completion time of previous simulation jobs. On the basis of these, and the requested deadline, SparkCloud will choose an appropriate size of VM or use an existing VM, provided it can complete the job within the given deadline.
- **Load Balancing:** Although the Request Manager tries to make best possible estimation of the completion time of a job, as a result of the multitenancy model of the cloud infrastructure, there is a possibility of some job finishing early and some taking slightly more time than expected. This situation can adversely affect the utilization of worker nodes and also can lead to missing deadlines. To avoid this and improve resource utilization, SparkCloud tries to balance the load across different worker nodes. Moreover, if any worker node becomes free, jobs with a tight deadline (which are waiting for execution to start) can be reassigned to this idle worker node.

- **On-Demand Execution:** SparkCloud automates the process of the configuration and installation of Spark software on the basis of the requirement of the user; or, in other words, users can do their bushfire simulation whenever they choose.
- **Fault Tolerance:** SparkCloud can handle failures of worker nodes. In the case of failure, the requests are reassigned to other workers by the WorkerController. These reassignment requests will be given higher priority than new requests.
- **Scalability:** Most SparkCloud components work independently, which allows for a scalable implementation, as each component can be distributed as a separate service accessing information through a shared database. Moreover, Spark software is scalable in itself, which is shown through benchmarking results in the evaluation section.

4.2. Sequence Diagrams: Interaction between SparkCloud Components

Each user request results in the creation of a Spark job, which goes through different states during its lifetime. These states are described in Figure 2. Figures 4 and 5 highlight the sequence of operations performed for each request received by SparkCloud. In the following section, we describe these diagrams in detail.

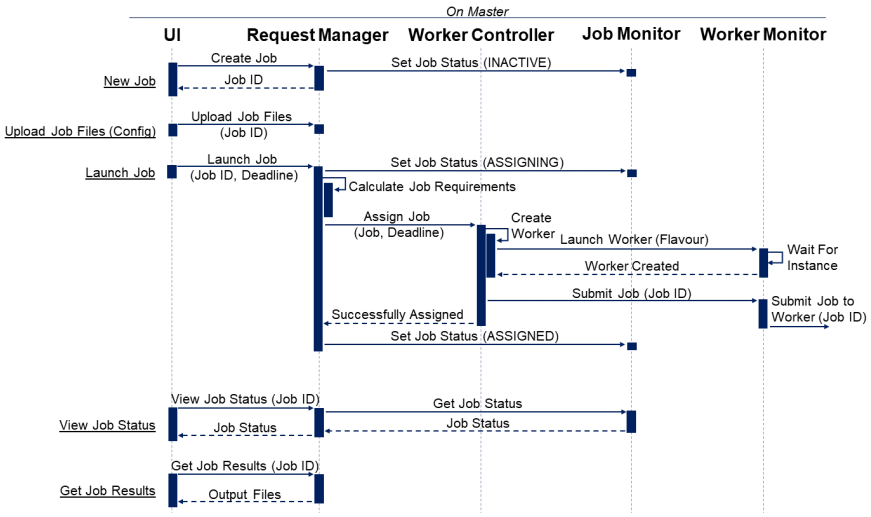


Figure 4. Master-node sequence diagram.

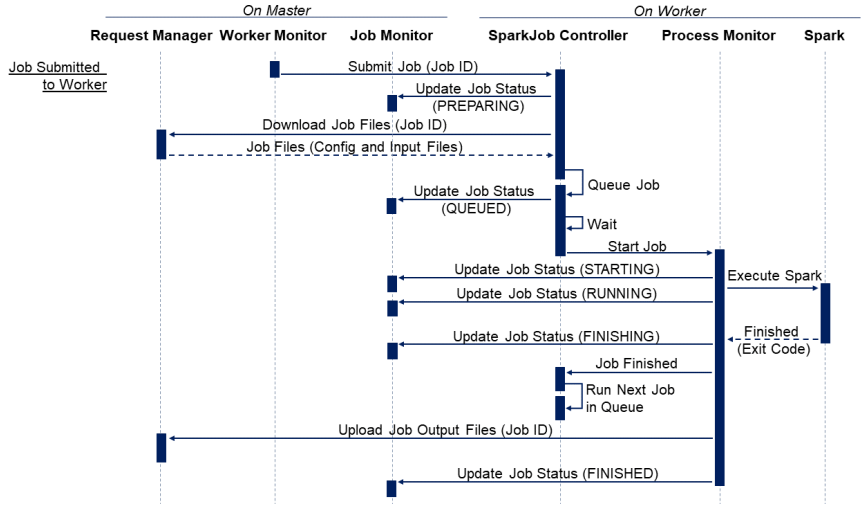


Figure 5. Worker-node sequence diagram.

4.2.1. Master Sequence Diagram

Figure 4 illustrates the basic interactions between the client and the master. When the user makes a request, it is forwarded to the Request Manager, which coordinates all the core services on the master (namely, the Worker Controller and Job Monitors). For example, when a user requests to launch a job, the UI will send the job's ID string with the user's desired deadline to the Request Manager. To start the process, the Request Manager will update the job's status to ASSIGNING through the Job Monitor. The Job Monitor contains all state-related information for a job and ensures that the job successfully completes each stage of execution (see Figure 2). The Request Manager will then calculate the job requirements—which includes predicting the execution time for each worker flavour (i.e. a different size of the cloud VM instance) and identifying any other hardware/software requirements. The job, with its requirements, is then passed to the WorkerController for assigning.

To reduce possible concurrency complications, the WorkerController only assigns a single job at any given time (using a queue—on a FCFS basis). As the WorkerController is trying to assign a job, it will iterate through available workers (using Worker Monitors) and calculate an estimated queue completion time. This is then compared to the job's estimated execution time (according to that worker's flavour) to see whether it is possible for the given worker to complete the job within the deadline. If there is more than one worker that can successfully complete the job within the deadline, the WorkerController will assign the job to the worker, which can complete the job in the quickest estimated completion time.

If a new worker is required, the WorkerController will create a worker with the smallest possible flavour (VM size) that can complete the job within its deadline. The process of creating, launching and initiating a new worker is allocated to a new WorkerMonitor. The Worker Monitor will use the cloud VM Monitor class to create and launch a new cloud VM instance (with the flavour specified by the WorkerController). The cloud VM Monitor will supervize the launching process and notify the WorkerMonitor when the instance has been successfully launched. The WorkerMonitor will then attempt to connect to the worker and submit the job (sending only the job's ID string), at which point the job status is set to ASSIGNED (Figure 2).

To reduce the amount of time required to assign a job on the WorkerController, a job is considered ASSIGNED when the cloud VM Monitor has successfully started the launching process (it typically takes over 2 min to successfully launch and connect to a new instance). If an error occurs during launching or connecting to the new worker, the job is reassigned with high priority.

At this point, the master is effectively finished with the job, all further processes are handled by the worker node. The master is only responsible for overseeing the job state (via its Job Monitor), to ensure successful completion of the job within its given deadline.

4.2.2. Worker Sequence Diagram

Figure 5 highlights the sequence of actions performed on a worker node as it receives a new job submission (as a job ID string) from the master (from its respective Worker Monitor on the master). When a job ID is received, the worker (SparkJob Controller) will confirm that it has accepted the job by sending a request to the master (to its respective Job Monitor) to update the job's status to PREPARING (Figure 2). If the worker wishes to reject the job, it can set its status to REJECTED, which will trigger the master to reassign the job (with high priority).

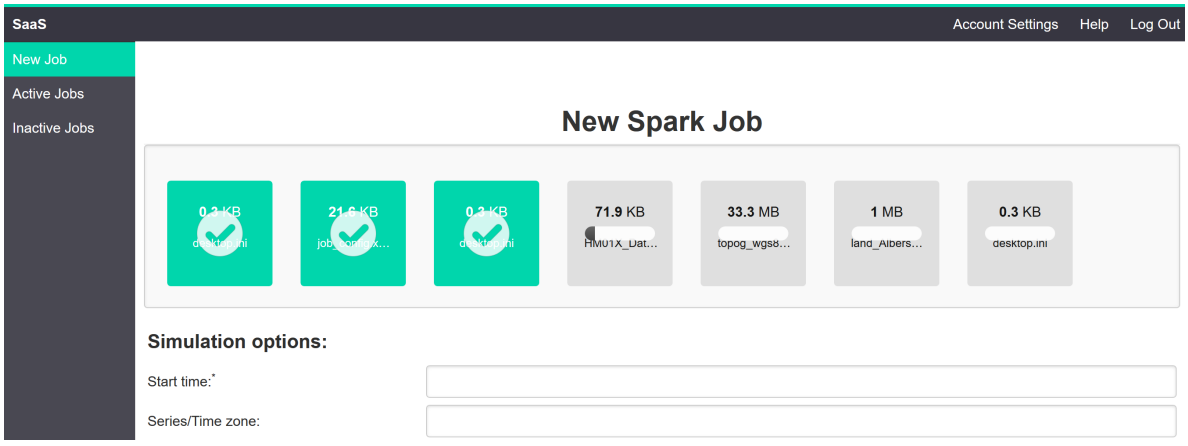
After a job has been accepted, the worker (SparkJob Controller) will request to download the job configuration and input files through the master's REST API. If the files are successfully downloaded, moved into their correct locations and processed further if required, the job is then added to the back of the worker's job queue. If the queue is not empty, the job's status will be updated to QUEUED (on the Job Monitor), and it will remain in the queue until all previous jobs have finished executing. If the queue was initially empty, the job is started immediately, and its status is updated to STARTING (Figure 2).

At this point, a new Process Monitor is created to coordinate the execution and monitoring of the Spark software. If the Spark process starts successfully, the job status is updated to RUNNING (Figure 2); if the Spark process then exits without error, the status is updated to FINISHING (Figure 2), and the SparkJob Controller is notified that the process has finished executing, which will trigger the next job in the queue to start.

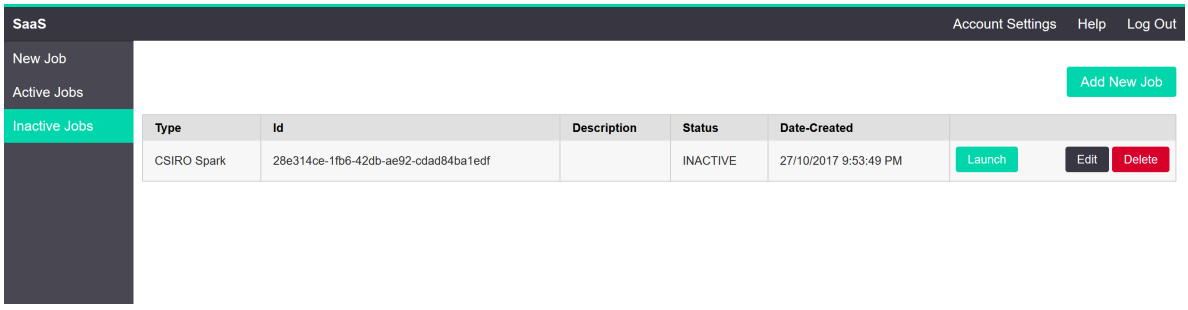
After the Spark process has finished executing, the Process Monitor will then upload the output files to the master (via the REST API). If all files are uploaded successfully, the job status is set to FINISHED (Figure 2). The user can then retrieve the job's output files with the client (or REST API).

4.3. Implementation Technologies

The prototype implementation of SparkCloud was created using Java and was deployed on an OpenStack-based cloud infrastructure [16]. To initiate VMs, and configure them as a worker node, we utilize the JClouds [17] library which provide Java-based wrapper APIs for OpenStack. The UI of SparkCloud is Web-based and implemented using VueJS (https://vuejs.org/), which allows access to the service from different devices. To implement the Web server, we utilize Jetty and Gretty, which allows for the easy deployment of zero-dependency Java Web apps. For uploading and downloading the files through the Web UI, we utilized DropezoneJS (www.dropzonejs.com), which provides an easy drag-and-drop interface to the end user. Some of the screenshots of the proposed system are given in Figure 6. Figure 6a shows the request creation and file uploads view, and Figure 6b shows different jobs submitted to the system with their status.



(a)



(b)

Figure 6. Spark benchmarking using real bushfire case studies. (**a**) Starting page view; (**b**) job status.

5. Performance Evaluation

In this section, we present the performance evaluation of SparkCloud's prototype implementation. The master and worker nodes are hosted on Nectar Cloud (https://www.nectar.org.au/research-cloud)

infrastructure in the Tasmania datacenter. The VMs that are utilized for the experiments are presented in Table 1. As SparkCloud's resource allocation mechanism needs to estimate the execution time of each user request, we conducted some benchmarking of the Spark software using real case studies. These results are discussed in the following section.

Table 1. Nectar Cloud virtual machine flavours.

Flavor Name	Virtual CPUs (VCPUs)	Memory	Primary Disk	Ephemeral Disk
Small	1	4 Gb	10 Gb	30 Gb
Medium	2	8 Gb	10 Gb	60 Gb
Large	4	16 Gb	10 Gb	120 Gb

5.1. Benchmarking of Spark on Cloud

As discussed earlier, SparkCloud needs to estimate the execution time for different simulations. For evaluation purposes, we conducted a small-size experiment to obtain the estimated times to run bushfire simulations using Spark on Nectar Cloud. We utilized two real bushfire studies for this purpose: Lithgow's state mine at New South Wales and Forcett bushfire at Dunalley Tasmania. The details of different simulation parameters are given in the Spark evaluation document [18]. The benchmarking results were obtained by running simulations in three different flavours (sizes) of VMs (small, medium, and large) available on Nectar Cloud; properties of each flavour are presented in Table 1. Nectar Cloud has different datacenters across Australia. We conducted the experiments using these different sites. Figure 7 shows how the simulation time varies with the different type of VMs and different locations. The presented results are the average of five repeated experiments. It is clear that Spark scales linearly with the increasing number of virtual cores (i.e., the size of VMs) for both the case studies. There are differences in the execution time of different case studies, which is due to different simulation parameters. It is interesting to observe that different datacenters have a dramatic effect on benchmarking results, which highlights that the underlying physical machines across different locations of Nectar Cloud are quite different. The "NCI (National Computational Infrastructure) Canberra" region has the fastest hardware, and thus the jobs with a tighter deadline can be scheduled to this datacenter.

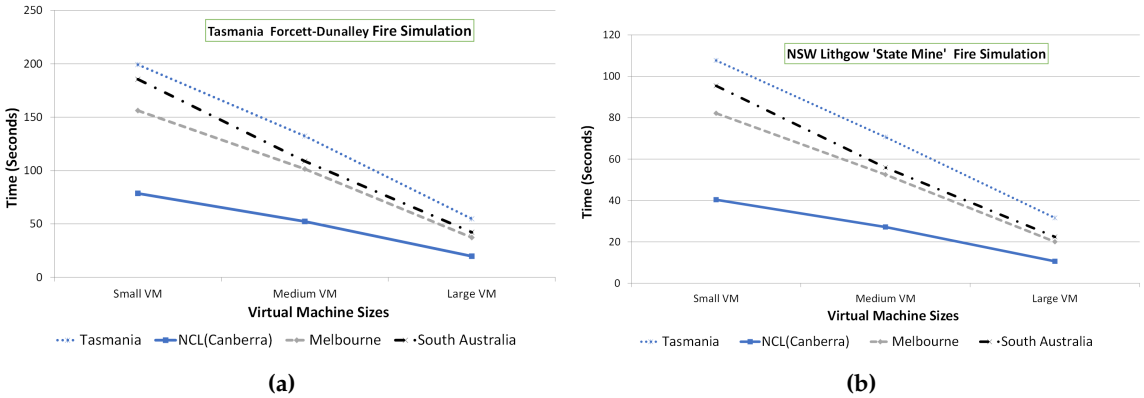


Figure 7. Spark benchmarking using real bushfire case studies. (**a**) Forcett-Dunalley, TAS; (**b**) Lithgow, state mine NSW .

5.2. Evaluation of SparkCloud System

5.2.1. Experimental Setup

Worker Creation Time Estimation

The creation time estimation is computed as an average of the time taken to create each VM flavour, which is updated as more workers are created. It is estimated to take around 2 min for the different flavours.

Job Execution Time Estimation

Estimations are calculated on the basis of the instance region (which region the instance will be launched in), the instance flavour (size of VM instance—number of VCPU cores, memory, etc.) and the size of the job. Currently, the size of the job is determined according to two test-cases (small jobs—Lithgow, state mine NSW; medium jobs—Forcett-Dunalley, TAS), and all instances were launched in the "Tasmania" region to reduce complexity for testing. The instance flavours were limited to m2.small (1 VCPU core, 4 GB ram), m2.medium (2 VCPU, 6 GB) and m2.large (4 VCPU, 12 GB).

The job execution time predictions are only based on the reported simulation time returned by the Spark software (through the benchmarking previously outlined), which does not include the preprocessing of job resources (config and other files)—such as downloading the files from the master server—and it also does not include the presimulation testing done by the Spark software.

Instance Flavour Selection Algorithm

When a new worker is created, its flavour (VM size) is determined by the deadline of the job that triggered the worker creation. If the job has a stricter deadline (or is more resource-heavy), the worker will be launched with a flavour with more resources available—to allow for the deadline to be met. In the future, it would be worth also using the current rate of incoming jobs being submitted to the server as an additional factor when determining the VM flavour, as it better to launch a single larger VM rather than multiple smaller VMs (performance and time-wise).

Experimental Scenarios and Metrics

The performance of SparkCloud is evaluated from the perspective of satisfying user requirements (i.e., deadlines) and load balancing (average CPU time utilized per worker). The evaluation was conducted in two scenarios: (1) variation in number of simultaneous user requests, and (2) variation in deadline requirements of user requests. All experiments were conducted using real case studies (Lithgow state mine, NSW and Forcett-Dunalley, TAS). User requests consisted of 50% of each of these case studies—which were launched in an alternating sequence.

For scenario 1, all jobs had a "medium" deadline—which was 4 times the average estimated execution time (on the default instance flavour). The number of user requests began at 10 job submissions, which was then increased to 20, 30 and then finally 40. Because of the limitation on Nectar Cloud VMs allocated to us, a maximum of 40 simultaneous jobs were created for evaluation.

For scenario 2, only 10 jobs were submitted for all experiments, and the deadline requirements were set to "tight" (which is a deadline multiplier of 2, i.e., 2 times the average estimated execution time of the job), "medium" (multiplier of 4), "relaxed" (multiplier of 8) and finally "very relaxed" (multiplier of 16).

All experiments were repeated five times. It was assumed that there would be no worker machine available at the start of each experiment. Each experiment was conducted using a Java class, which was created to automatically perform the desired user requests with the specified parameters for each scenario test case. The class would then monitor all jobs until completion, at which point the results were saved, all of the completed jobs were deleted and all created workers were terminated. There was a delay of 3 s before the next experiment could begin.

5.2.2. Discussion of Results

Scenario 1: Variation in Number of Simultaneous Requests

Figure 8 shows the results of the first scenario, where the number of simultaneous user requests were varied. As the number of user requests (jobs submitted) increased, the CPU time per worker remained constant—which is ideal for load balancing—and the number of workers created increased linearly—which is also ideal. Figure 8a shows that there was no statistically significant difference between increasing the number of simultaneous user requests and the rate of missed deadlines. (Note: error bars presented on figures represent 95% confidence intervals).

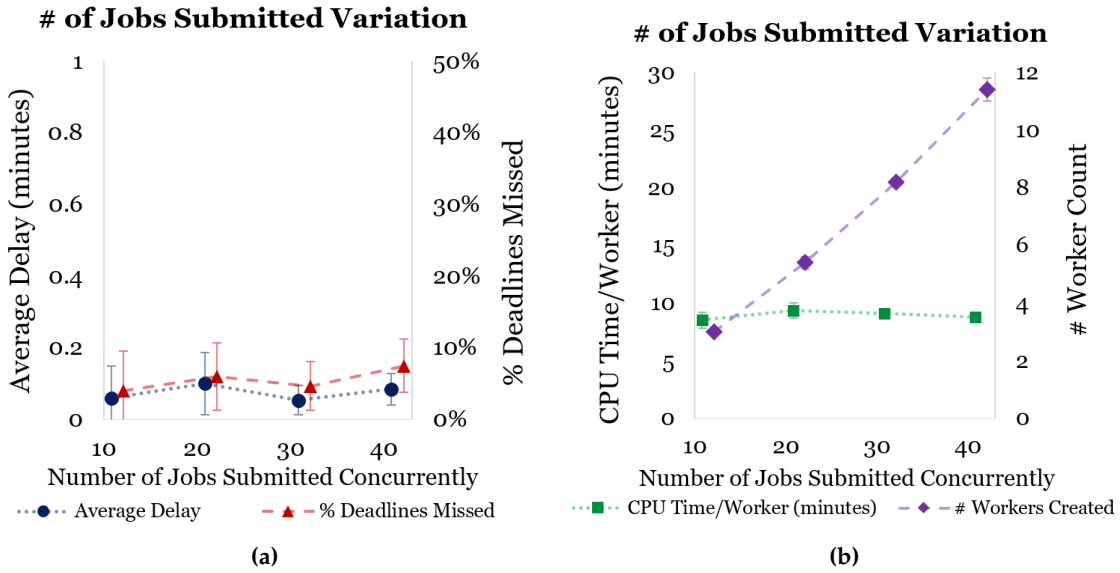


Figure 8. Variation in number of simultaneous jobs. (**a**) Deadline missed; (**b**) resource utilization.

It is important to note that for a job to trigger the creation of a worker, its deadline must cross a threshold of the average creation time for a worker (approximately 2 min). Therefore, it is not uncommon for a job to finish around 2 min after it's deadline. As stated in the Experimental Setup section, there is also some systematic error in the estimated job execution time due to discrepancies in the Spark benchmarking times compared with the true execution time (we estimate that this is at least 1 min of additional time). This error becomes compounded as more jobs are added to workers, which results in higher delays in deadlines.

Scenario 2: Variation in Deadline Requirements

Figure 9 shows how SparkCloud performs when requests with varying deadlines are submitted. As deadlines are made tighter, it is expected that more deadlines will be missed.

The rate of deadlines missed depends on how well the jobs can be divided into workers. This is highlighted by the perfect 0% deadlines missed during the 16 deadline multiplier experiments. The deadlines missed and average delay results for the 4 and 8 deadline multiplier experiments included 0 in their respective confidence intervals; thus more evaluation is required to confirm any relationship between the higher deadline multiplier (4, 8, 16, etc.) and delay/deadlines missed.

It is important to note that as each experiment began with no workers created, it was expected that some jobs would finish a few minutes late of their deadlines. Real-world use of the system will not require this many new workers to be created—as there will be idle (or somewhat free) workers available most of the time. It is also expected that the service will not normally be used to assign batches of jobs at one time, but rather only assign single (time-critical) jobs at one time.

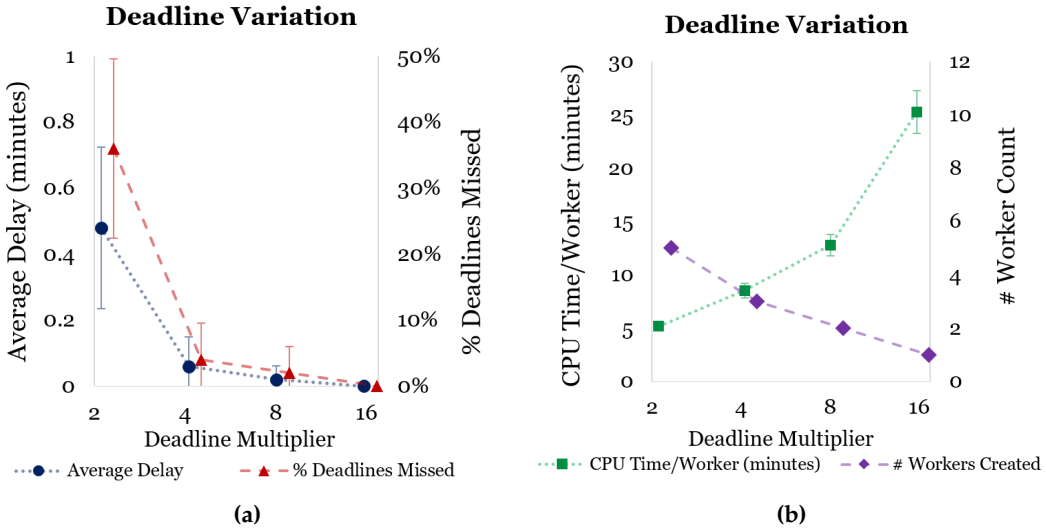


Figure 9. Variation in deadlines. (**a**) Deadline missed; (**b**) resource utilization.

6. Conclusions and Future Directions

There are several works on modeling and simulating bushfires; however, only few simulators are in operational use. Out of the operational simulators, Data61's Spark software is one of the most scalable to our knowledge. Spark can directly utilize elastic features of cloud computing infrastructure to satisfy a user's time constraints. To fill this gap, this paper proposes a real cloud-based bushfire-spread simulation service called SparkCloud. It utilizes Spark fire-spread simulation framework that can be customized for different contexts and can scale well in multi-core machines. We have presented the design features of SparkCloud—which include the load balancing of worker nodes and the satisfaction of a user's deadline. The evaluation results with real case studies show the effectiveness of SparkCloud, which can complete most requests within a specified deadline and for which other requests are only delayed by 2 min, which is caused by the starting of new workers.

In the future, we will improve the execution time prediction model of the system to finish jobs with very tight deadlines. We will also improve the system by adding a more accurate worker creation time prediction model, which according to us will vary between cloud providers. Moreover, the current study has been done using CPU-based VMs. Because Spark gives equally scalable results but faster results with GPUs, we tested SparkCloud with VMs having GPUs. We would also test the system on different commercial clouds, such as Amazon EC2, with different VM configurations.

We also plan to integrate other Web-based storage services to allow the user to submit jobs without needing to upload files through the provided Web UI (or REST API). Currently, Spark's output files are only available for a user to download through the Web UI; we also integrate Web-based visualization tools that will provide an easy way for the user to visualize results through the Web UI, without having to download and process the files locally.

Acknowledgments: We acknowledge Lewie Chan for his assistance in conducting the benchmarking of Spark on Nectar Cloud.

Author Contributions: Dr. Saurabh Garg contributed the main architectural ideas and experiments of the paper. Nicholas Forbes contributed by implementation and conducting the experiments. Dr. James Hilton and Dr. Mahesh Prakash contributed in developing main research ideas of the paper and integrating Spark bushfire simulator with the framework. All contributed to the writing of the final manuscripts.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Papadopoulos, G.D.; Pavlidou, F.N. A comparative review on wildfire simulators. *IEEE Syst. J.* **2011**, *5*, 233–243.
2. Tolhurst, K.; Shields, B.; Chong, D. Phoenix: Development and application of a bushfire risk management tool. *Aust. J. Emerg. Manag.* **2008**, *23*, 47–54.
3. Johnston, P.; Kelso, J.; Milne, G.J. Efficient simulation of wildfire spread on an irregular grid. *Int. J. Wildland Fire* **2008**, *17*, 614–627.
4. Finney, M.A.; Rocky Mountain Research Station—Ogden. *FARSITE, Fire Area Simulator—Model Development and Evaluation*; US Department of Agriculture, Forest Service, Rocky Mountain Research Station: Ogden, UT, USA, 1998; Volume 3.
5. Miller, C.; Hilton, J.; Sullivan, A.; Prakash, M. SPARK—A bushfire spread prediction tool. In Proceedings of the International Symposium on Environmental Software Systems, Melbourne, Australia, 25–27 March 2015; Springer: Berlin, Germany, 2015; pp. 262–271.
6. Wang, Z.; Vo, H.T.; Salehi, M.; Rusu, L.I.; Reeves, C.; Phan, A. A large-scale spatio-temporal data analytics system for wildfire risk management. In Proceedings of the Fourth International ACM Workshop on Managing and Mining Enriched Geo-Spatial Data, Chicago, IL, USA, 14–14 May 2017; ACM: New York, NY, USA, 2017; p. 4.
7. Garg, S.; Aryal, J.; Wang, H.; Shah, T.; Kecskemeti, G.; Ranjan, R. Cloud computing based bushfire prediction for cyber–physical emergency applications. *Future Gener. Comput. Syst.* **2017**, doi:10.1016/j.future.2017.02.009.
8. Bai, F.; Hu, X. Cloud mapreduce for particle filter-based data assimilation for wildfire spread simulation. In Proceedings of the High Performance Computing Symposium, San Diego, CA, USA, 7–10 April 2013, Society for Computer Simulation International: San Diego, CA, USA, 2013; p. 11.
9. Almeer, M.H. Cloud hadoop map reduce for remote sensing image analysis. *J. Emerg. Trends Comput. Inf. Sci.* **2012**, *3*, 637–644.
10. Saleem, Y.; Salim, F.; Rehmani, M.H.; Rehmani, M.; Faheem, Y. Integration of cognitive radio sensor networks and cloud computing: A recent trend. In *Cognitive Radio Sensor Networks: Applications, Architectures, and Challenges: Applications, Architectures, and Challenges*; IGI Global: Hershey, PA, USA, 2014; Volume 288.
11. Saleem, Y.; Salim, F.; Rehmani, M.H. Resource management in mobile sink based wireless sensor networks through cloud computing. In *Resource Management in Mobile Computing Environments*; Springer: Berlin, Germany, 2014; pp. 439–459.
12. Wang, L.; Ma, Y.; Yan, J.; Chang, V.; Zomaya, A.Y. pipsCloud: High performance cloud computing for remote sensing big data management and processing. *Future Gener. Comput. Syst.* **2018**, *78*, 353–368.
13. Kalabokidis, K.; Athanasis, N.; Vasilakos, C.; Palaiologou, P. Porting of a wildfire risk and fire spread application into a cloud computing environment. *Int. J. Geogr. Inf. Sci.* **2014**, *28*, 541–552.
14. Roger Daslandes, H.J. *Final Report: An Evaluation of Fire Spread Simulators Used in Australia*; Bushfire Predictive Services: Victoria, Australia, 2017.
15. Masse, M. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*; O'Reilly Media, Inc.: Newton, MA, USA, 2011.
16. Sefraoui, O.; Aissaoui, M.; Eleuldj, M. OpenStack: Toward an open-source solution for cloud computing. *Int. J. Comput. Appl.* **2012**, *55*.
17. Graham, S.T.; Liu, X. Critical evaluation on jclouds and cloudify abstract APIs against EC2, azure and HP-cloud. In Proceedings of the 2014 IEEE 38th International Computer Software and Applications Conference Workshops (COMPSACW), Vasteras, Sweden, 21–25 July 2014; pp. 510–515.
18. Hilton, J.; Hetherton, L.; Miller, C.; Sullivan, A.; Prakash, M. The Spark Framework; Technical Report, Report Number: EP152898; DATA 61, 2015. Available online: https://research.csiro.au/static/spark/Spark.pdf (accessed on 1 December 2017).