A Scalable Master-Worker Architecture for PaaS Clouds

Vibhor Aggarwal, Shubhashis Sengupta, Vibhu Saujanya Sharma, and Aravindan Santharam
Accenture Technology Labs, Bangalore, India
{vibhor.aggarwal, shubhashis.sengupta, vibhu.sharma, aravindan.santharam}@accenture.com

Abstract—**Clouds provide an attractive infrastructural option to deploy highly-scalable distributed applications. Platform as a Service (PaaS) clouds offer basic software stack and services along with the execution containers to simplify the hosting of user applications. However, traditional many task computing architectures cannot be hosted as-is on current PaaS platforms due to certain limitations. This paper describes a novel modified architecture for master-worker, a well-known many task computing paradigm, to take advantage of the fast scalability provided by PaaS. The architecture is transformed into a multi-agent system where the distributed agents use a message broker for communication and to store the computation progress. The agents are capable of dynamically shifting between a master and a worker role based on the information available with a durable message broker. This state-less feature of the agents makes them amenable for a PaaS platform and adds fault-tolerance to the system. The experiments illustrate the promising potential of the architecture to efficiently scale computationally intensive tasks on PaaS.**

I. Introduction

Cloud computing provides an emerging and cost-effective infrastructure which can be used on-demand in a "pay-as-you-go" mode. One of the key advantages of cloud is its ability to scale-up infinitely to match the application needs. For a perfectly parallel application, theoretically, this implies that cloud can handle any size of the workload in almost no time. This is a very attractive proposition as compared to hosting the application in-house, if the application load varies significantly. In-house infrastructure is usually difficult to scale, due to difficulty in procurement and system management related issues, than the options available on cloud.

Cloud computing can be broadly categorized into three types [1]: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). IaaS vendors provide infrastructure such as virtual machines, storage and network and the users maintain their own software stack and applications. SaaS providers offer user-customizable business applications in the cloud. PaaS platforms offer a basic software stack for example operating system, run-time, databases etc. and services such as message broker, logging etc. This allows the users to deploy their own applications (similar to IaaS) without the responsibility of maintaining the basic software stack (similar to SaaS). In PaaS, the user applications are typically hosted in containers which can be added or deleted nearly instantaneously (horizontal elastic scaling) to adjust to fluctuating workloads. The key strengths of PaaS platforms are ease of use and flexibility as compared to the other two types.

However, few issues will have to be addressed in order to take the full advantage of a PaaS cloud. Typically, a PaaS container has ephemeral storage. Therefore, any persistent data needs to be moved outside the container using one or more of the available services. Also, the PaaS middleware handling these containers can shift them around for better load management or recycle them after a given time period. This volatility of the containers can be overcome by hosting state-less applications. Applications which need to maintain state have to be carefully re-engineered to checkpoint their state outside the container. The dynamic nature of the containers also prevents mechanisms for direct communication between them; thus seamless synchronization, message passing and clustering of the instances running in the containers become problematic.

Master-worker architecture is a distributed application paradigm, frequently employed for Many Task Computing (MTC) [2], to complete large number of loosely-coupled tasks in parallel. The master acts as the central authority to drive the computation forward and is in charge of delegating relevant tasks to the workers, who perform them independently in parallel. The global state of affairs is generally available at the master, making it a crucial entity in the system. The workers typically do not communicate with each other or use the master to route messages. Many applications such as batch systems, parallel work-flows, map-reduce, parameter sweep applications etc. can be implemented using the master-worker design pattern. Message Passing Interface (MPI) benchmarking of the cloud by Zhai et al. [3] suggested that its performance is similar to low-cost clusters but the communication can be a challenge. However, for a master-worker system computation to communication ratio is high therefore its workload is expected to be more amenable to cloud.

There are two main concerns with master-worker architecture. First, the master is a single point of failure in the system. It is necessary to ensure that the master node is fault-tolerant for the computation to progress. If a worker dies, the loss of computation is minimal as compared to if the master dies. Second, the master can become a bottleneck in the whole process if it is not able to delegate tasks at the same rate at which the workers are finishing them. This can happen if too many workers are connected to a single master and can limit the scalability of the system.

This paper presents a novel scalable agent-based alternative to the traditional master-worker system for PaaS cloud. The key intention of the work is to exploit the immense scalability of the cloud to execute parallel tasks, and in order to do so, effectively tacking the limitations of volatility of the containers and absence of effective message passing scheme. This is achieved using multiple state-less agents running in the PaaS containers, each capable of performing both master and worker tasks depending on the current state of the overall execution. This global state is stored outside the containers with a reliable and scalable message broker and each agent can take actions based on its perception of the global state. This adds fault-tolerance to the architecture. The scalability of the system relies on the scalability of the containers and that of the message broker. The application also needs to be efficiently parallelized for good scalability because scalability is a property dependent on the algorithm and machine combination rather than being exclusive to the either of them [4].

II. Related Work

Master-worker architecture is a generic parallel computation paradigm which has been widely studied and applied in many domains where the problem can be broken into multiple independent parallel tasks. It has been used for various flavors of distributed computing - cluster computing [5], grid computing [6], cloud computing [7] and volunteer computing [8]. Scalability of master-worker systems is an active area of research and many decentralized or hierarchical approaches have been presented. Banino et al. [9] discuss a cost model for deploying of multiple masters to optimize system throughput without budget violations. Scalability analysis of applications on hierarchical platforms is presented in [10]. Bendjoudi et al. [11] have recently published an adaptive hierarchical framework where each distributed process can switch between master and worker at run-time depending on the resource availability. This enables the computations to start faster and performs more efficiently than a static hierarchical approach. However, traditional master-worker architecture is not well suited for PaaS platforms as explained in Section III.

Traditional parallel infrastructures have been expensive and the prospect of cheaper and elastic resources is driving researchers to explore the potential offered by cloud computing to solve their challenges. Wang et al. [12] proposed a dynamic service provisioning model for cloud vendors to offer MTC run-time environments and also showed that the MTC service providers can leverage cloud platforms economically. The viability of deploying a compute cluster for MTC, using resources from multiple cloud vendors, has been studied in [13]. It presents a favorable scalability, performance and cost considerations for such clusters with small number of resources and extrapolates the results for larger infrastructures.

Iosup et al. [14] studied the performance of IaaS platforms for MTC scientific applications. They found the IaaS performance to be an order of magnitude worse than traditional parallel infrastructures. But IaaS was cheaper and feasible alternative for jobs with short deadlines due to low and steady wait times. However, ramping-up a computation on IaaS can be time consuming as compared to a PaaS platform where basic software stack is pre-installed. Prodan et al. [15] evaluated Google App Engine [16], a PaaS platform, for computationally intensive tasks and observed that it can be cheaper for jobs shorter than one hour as compared to Amazon EC2 [17], an IaaS. However, they were limited to small problem sizes due to a 30 second execution time limit enforced by the platform earlier.

III. System Architecture

In traditional distributed computing systems, direct communication methods such as Message Passing Interface (MPI), Parallel Virtual Machine (PVM) or even Transmission Control Protocol (TCP) are employed for communication between master and workers. The node running the master is usually hosted on reliable infrastructure; therefore, master replication may not be required. If the workers run on unreliable hosts then redundancy and result voting mechanisms can be employed for obtaining reliability and consistency [8].

Migrating applications to PaaS requires their architectural patterns to be modified to gain full advantages as mentioned earlier in Section I. The same applies to master-worker based applications as well. The container which hosts the master needs to store the global information outside so that the application progress is unaffected in case the container is restarted/migrated to another node by the PaaS middleware during task execution. Furthermore, as direct communication methods such as the ones employed in traditional distributed computing are not supported, asynchronous message queue based communication (present in most PaaS platforms) between the instances needs to be devised.

The idea behind master-worker system is that it is much easier to nominate one machine to keep records to maintain the global state (results, finished and unfinished tasks) in a distributed system, than have the state in all machines, as update and synchronization operations become expensive. Therefore, workers do not have the information on what needs to be done for progression and they query the master to find that out. However, a master on PaaS needs to store the information about global state outside which the workers can query directly rather than querying the master. Furthermore, any worker can perform the master job as it has access to information needed to drive the system forward. The system can thus be transformed into a multi-agent system where each agent is capable of behaving as a master or a worker depending on the state of the system.

Each agent is modeled as a simple reflex agent [18]. It uses a library of condition-action rules to choose an appropriate action to be performed based on the current condition. The agent relies only on current perception rather than any stored past information, i.e. it is state-less. The master-worker system can be implemented in a fault-tolerant manner using such agents. If an agent dies, another agent can take up its predecessor's responsibilities efficiently as no information related to the progress of complete workload is lost. Simple reflex agents can only work if the environment is fully observable and a message broker can serve this purpose on a PaaS infrastructure.

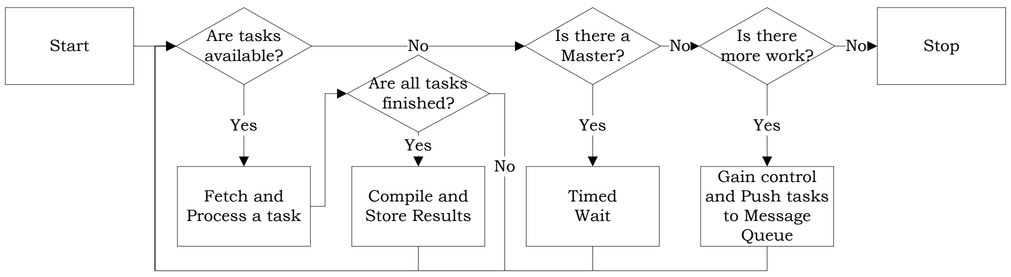


Fig. 1: Flowchart for the distributed agent-based MTC system

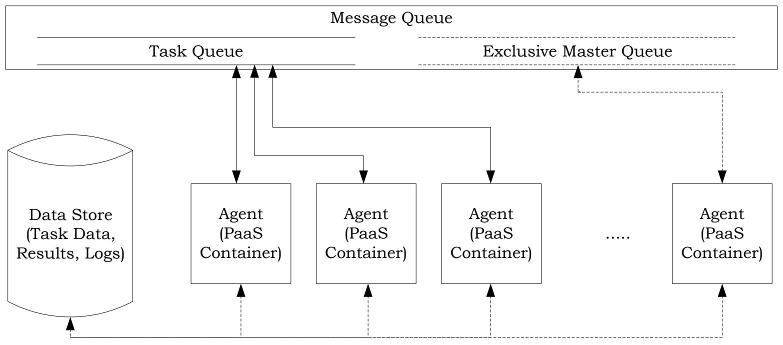
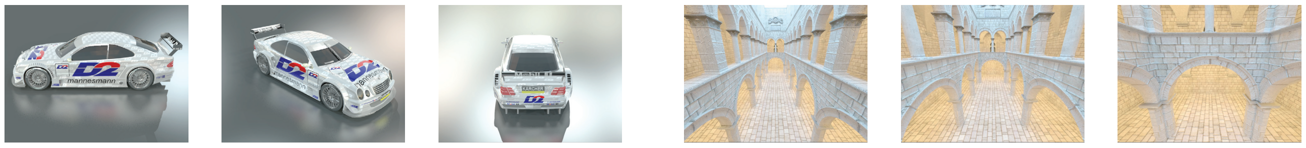


Fig. 2: System architecture for the agent-based MTC system on a typical PaaS platform



(a) Race Car Animation (W1) (b) Sponza Animation (W2)

Fig. 3: Sample Animation Frames

An emerging open standard messaging protocol, Advance Message Queuing Protocol (AMQP) [19] provides scalable, durable, and inter-operable brokering mechanism for guaranteed asynchronous messaging. In one of the first projects, it was deployed for 2000 users to process 300 million messages in a day [20]. AMQP has also been successfully used earlier for session management of web-based software systems on PaaS clouds [21].

The proposed architecture employs AMQP message broker to achieve scalability and decoupling of the architecture components. The agents can asynchronously store messages in queues to maintain the state as it would be light-weight in comparison to database updates. If the input data needed for computation is structured as files, then they can be stored in a data store accessible by all agents and the messages can hold their locations. The information that an agent is performing as a master has to be shared with other agents, hence it must be able to exclusively publish and gain control. AMQP provides a mechanism to create an exclusive queue which can be accessed only with a single connection and is deleted when this connection closes. Such an exclusive queue can be used to gain control to act as a master. If the agent which holds the exclusive queue dies due to a fault, the connection is closed thereby relinquishing the control to another agent which ensures system progress.

The flowchart of the system is depicted in Fig. 1. An agent starts and checks if there are tasks placed in a message queue marked for execution. If such tasks exist, the agent fetches one and processes. The results are then stored outside the container in a message queue and data store, and then the agent checks if all the tasks of a type have been finished. If so, it compiles the partial results and the process repeats. If there is no task available for execution, the agent checks for presence of a master in the distributed system by querying for an exclusive master queue. If no other master is present, it checks if any more work description is present in a data store. If so, it creates an exclusive queue indicating to the other agents that it has taken control and they should wait until the task queue is populated. Once it places the tasks in the queue it deletes the exclusive queue marking the end of its role as a master and allowing any other agent to takeover as a master when the need arises. In case no work description is present, the agent stops execution. The architecture diagram depicting the components of the system is presented in Fig. 2.

In the case where a single master becomes a bottleneck, multiple agents can be used to parallelize the master's task for example parallel compilation of results to decide further set of tasks for the next phase of computation. In such a case, multiple exclusive queues would have to be created, one for each master. However, still each agent can still choose to be a master or a worker dynamically based on the computation progress and the global state.

IV. Implementation Detail

A. Application details

High-fidelity rendering was chosen as a suitable MTC application for deployment in the PaaS cloud. High-fidelity rendering is the process of generating realistic images from a three-dimensional description of an environment using physically-based material properties of the objects and light source details. The computation is carried out by solving the rendering equation [22] which is estimated using Monte Carlo integration. For each pixel of the image, the light reaching the camera through it, is estimated multiple times to find its final color value. This makes the process computationally expensive; an image can take multiple hours to render on a single machine depending on the resolution and quality.

Parallelization is frequently used to render realistic images in reasonable times [23]. An image can be subdivided into set of tiles which can be rendered in parallel and then the results can be combined to form the final image. As a set of such images is required to create an animation, parallelization is highly essential for generating them. Traditionally dedicated clusters known as render farms are employed for such renderings, however cloud computing provides an interesting alternative by offering scalable on demand resources. High-fidelity rendering of animations rendering fits the MTC paradigm due to the large number of heterogeneous tasks with variable execution times, each producing an output file for compilation into one of the frames of the animation. It can be easily parallelized using the master-worker architecture and thus it was used to conduct scalability studies to evaluate the proposed architecture.

B. PaaS Platform

The master-worker architecture has been implemented and deployed on Heroku, a PaaS cloud [24]. Heroku was chosen as at the time of experimentation because some other open-stack based PaaS platforms were still in a beta preview stage while it provides full production support. Also, it provides a worker role which can run long asynchronous jobs while most other PaaS providers only offer a web role where the container is tied to short web requests. The agents are hosted inside Heroku's containers, known as dynos, which can be horizontally scaled easily using a toolkit provided by Heroku.

The parameters required for describing the computation are stored in a MongoDB [25], a NoSQL database addon service provided for Heroku. The agent which spawns first, creates an exclusive queue in RabbitMQ [26], an AMQP-based message broker, to inform any other agents that it has taken over. Then it pushes the tasks for the workloads into the queue using the parameters stored in MongoDB. The exclusive queue is deleted and the agents read the task descriptions from RabbitMQ to carry them out in parallel. The agents also fetch the model description files required for the rendering from MongoDB at the start. Each computed tile is then stored into MongoDB for later composition as Heroku provides ephemeral local file storage. Time stamp data is also saved in MongoDB for gathering timing statistics.

The RabbitMQ is configured such that each agent can dequeue only a single task from the task queue and fetch another one after an acknowledgement of the task completion is sent back to RabbitMQ. In case the acknowledgement is not received, RabbitMQ enqueues the task back. This is done to provide fault-tolerance in the system so that in case an agent dies the workload computation would be unaffected. A message is also stored into the RabbitMQ after a tile is rendered so that once all the tiles of an animation frame are computed, it can be compiled by an agent. This work is usually done at the master, but since any agent can check the status of completion of a frame from the messages in RabbitMQ, it can carry out the compilation by fetching the individual tiles from MongoDB. The compiled frame is also pushed to MongoDB.

C. Workloads

The master-worker architecture was assessed using three workloads on Heroku. The first two workloads (W1, W2) consist of rendering two animations with 120 and 90 frames respectively, see Fig. 3. They are rendered at a resolution of 800×600 using a Java-based renderer employing path tracing algorithm [22]. Each frame of the animations is subdivided into 200 tiles which are independently queued-up as tasks. These two workloads are random in nature i.e. non-deterministic number of calculations is performed to compute the results based on a randomized algorithm. A third and more deterministic workload (W3) is also chosen, consisting of fixed number of calculations to determine the N^{th} prime number. Five type of tasks to calculate the value for different N are queued-up as tasks. A total of 3600 task instances for each task type are queued in W3. Table I presents statistics related to the execution times of the three types of workloads in Heroku's dynos. It takes approximately 240 dyno hours to compute a five second long animation, indicating the computational complexity involved in high-fidelity rendering of larger animations at higher resolutions. The computational capability of a single dyno is not published by the vendor, however, its benchmarking indicates the performance to be similar to one Amazon EC2 compute unit [27] (Heroku is hosted on Amazon EC2).

V. Results

The scalability of the proposed architecture was studied by executing the three workloads on different number of dynos ranging from 4 to 512.

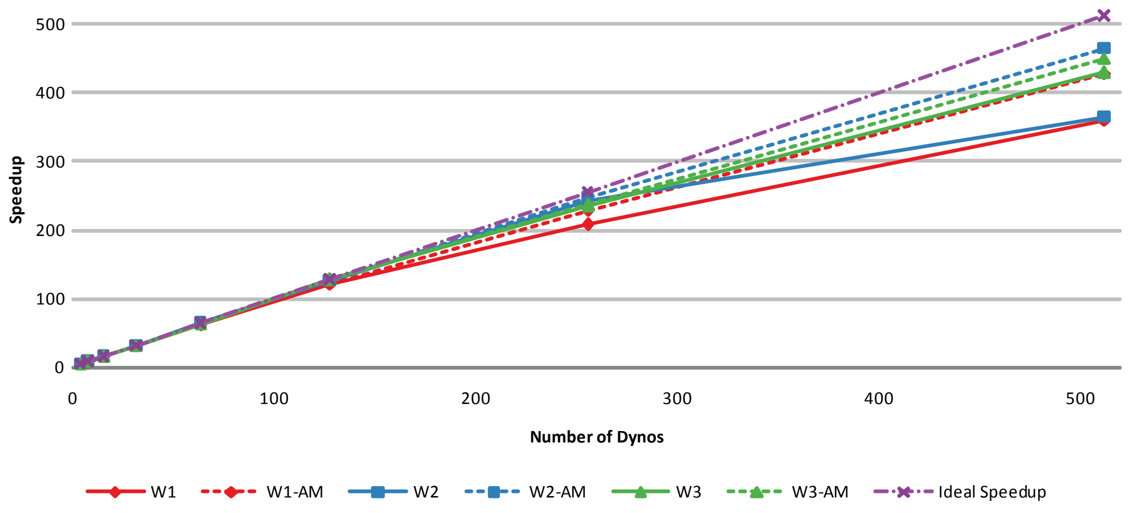


Fig. 4: Speedup Graph with respect to time taken using 4 dynos

TABLE I: Workload Statistics

	Min Task Time (in s)	Max Task Time (in s)	Total Time (in Hr)
W1	1	460	240
W2	20	220	400
W3	20	60	190

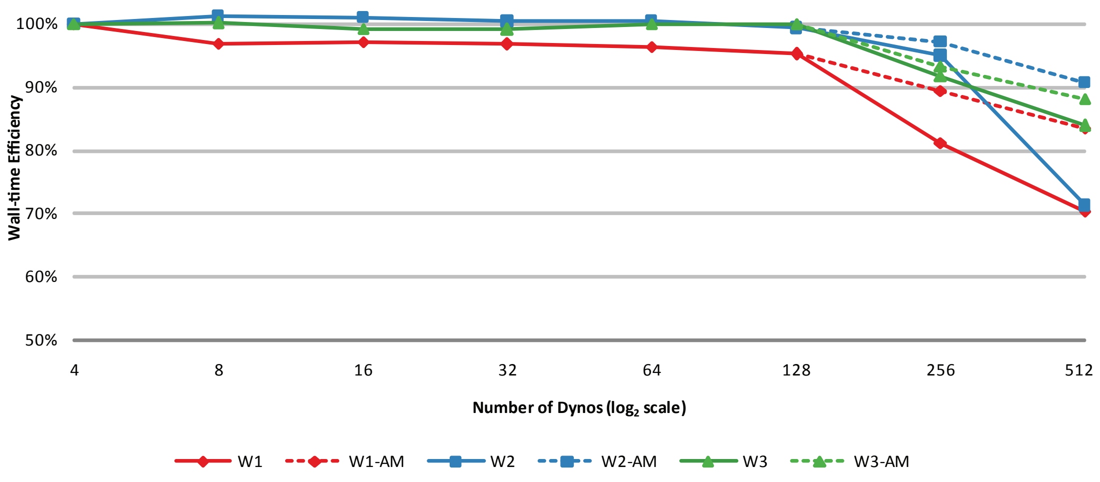
A. Speedup and Efficiency

The speedup of a parallel implementation is defined as the ratio of the wall-time needed to complete the workload on a single resource to the wall-time needed in parallel. As the size of the workloads used prevented them to be completed in reasonable times on a single dyno, perfect speedup was assumed for four dynos to perform the calculations. The speedup graph has been plotted in Fig. 4. It shows that the speedup is almost linear for up to 128 dynos for all the three workloads after which it becomes sub-linear. W1 and W2 are affected much more as compared to W3. This may be attributed to presence of uneven task sizes in W1 and W2 - time needed for the fastest task in W1 is 1 second as compared to the slowest task which requires 460 seconds, see Table I. Ideally all tasks should be of similar size, but for image rendering this can be challenging as it is dependent on the complexity of light paths in a tile which is difficult to ascertain before actually performing the computations.

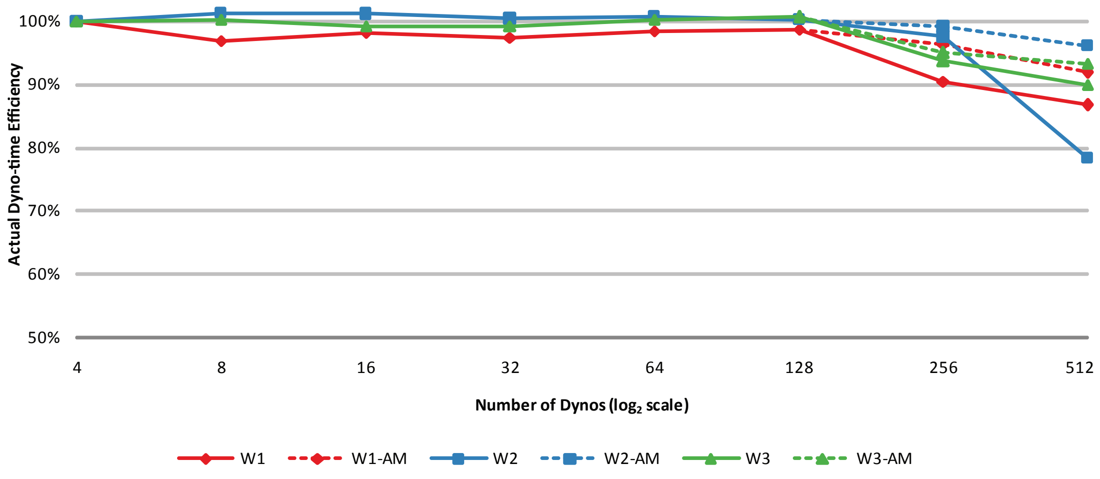
The efficiency of parallelization is shown in Fig. 5. The wall-time efficiency was calculated by measuring the time it took for the first task to start until the completion of the last task. This time was compared with the time taken on four dynos to calculate the efficiency. Fig. 5a shows a plot of the wall-time efficiency with different number of dynos. The efficiency is close to 100% until 128 dynos after which it shows a decrease.

One of the factors affecting the wall-time efficiency was the time required to ramp-up the computation. The amount of time needed to provision the computation on all the dynos rose significantly as the number of dynos was increased, as shown in Fig. 6. For example, it took almost 2 minutes to start the computation on 512 dynos. Although this time for dynamic ramp-up is comparatively much smaller than what can be achieved by adding nodes to an in-house cluster or on an IaaS cloud. The amount of computation that can be carried out on 512 dynos in 2 minutes is worth about 17 hours. For a 240 hour workload, this can lead to an efficiency loss of up to 7%. Therefore, to mitigate this factor another efficiency plot is presented in Fig. 5b by comparing actual dyno-times. This time was calculated by summing up the individual times required for computing all the tasks. The graphs show better dyno-time efficiency as compared to wall-time efficiency, however it was still found to be worse beyond 128 dynos.

The scalability of the proposed architecture is inherently tied to the assumption that the message queue can scale perfectly. The RabbitMQ addon for the Heroku platform however was a beta version with limited scalability where only limited numbers of concurrent connections from agents were supported. This meant that it might not have been hosted on a server powerful enough to handle large number of connections without degradation in quality of service. To test this hypothesis, a private RabbitMQ server was hosted close to Heroku in a large Amazon EC2 instance [17]. The speedup and efficiency plots (W1-AM, W2-AM and W3-AM) for this are shown in Figs. 4 and 5 with dashed lines. These plots show a marked increased in all the data series indicating that the native Heroku RabbitMQ addon was indeed a bottleneck. Therefore, efficient scale up of the proposed architecture can only be realized when the message queue can handle the load seamlessly. For an even better scalability and availability, clustering support provided by RabbitMQ can be leveraged [26].



(a) Wall-time Efficiency



(b) Actual Dyno-time Efficiency

Fig. 5: Efficiency Graphs

B. Discussion

The speedup and efficiency are affected by the resource pooling on a multi-tenant PaaS platform. Multiple containers hosting different applications compete for the same infrastructure (for example CPU, RAM, network etc.) on a PaaS and improper isolation of such containers can result in performance degradation. Although the vendor claims perfect container isolation [28], the reality was found to be different than the claim as shown in Fig. 7. The prime number calculation workload (W3) gave an insight into dyno isolation on Heroku platform. The graph shows a plot of time taken to complete one of the types of the five which constituted W3. Fig. 7a shows a plot of executing each instance the same tasks of type 2, computed 3600 times, on a different number of dynos. It shows that the approximately one-third of the task instances show a noticeable increase in the execution time when 256 and 512 dynos were employed for computation. This indicates that the CPU-bound tasks were throttled when the load on the PaaS platform increased. This can be also noticed for the plot of average execution time shown in Fig. 7b. The minimum time required for the computation of an instance remained constant even with increasing the number of dynos. However, the maximum time required could be as high as up to 300% of the mean time which can be attributed to imperfect isolation. The standard deviation of the execution time also showed a significant increase beyond 128 dynos.

The statistics for execution times for all five types of instances of W3 are presented in Fig. 8. Four dynos were used for computation to study the variation of execution times between different dynos. The data series have been plotted for 8 distinct dynos because the Heroku platform restarts dynos after every 24 hour period, however, only four of these dynos were running at a given point in time. Fig. 8a shows that the mean times for each task type remained fairly constant across the 8 dynos, but, Fig. 8b shows that the standard deviation between the dynos varied by as much as up to two times. This indicates that a dyno competing for resources on a busy host would show a higher variation in the execution times.

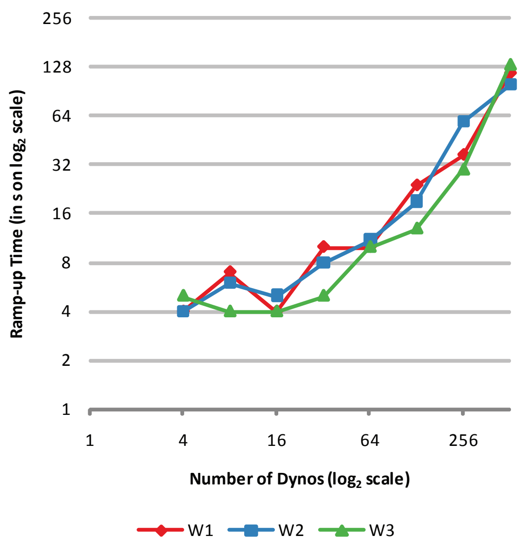


Fig. 6: Plot of Ramp-up times

The experiments presented in this paper have indicated that four things have to be considered while moving to a PaaS infrastructure: container isolation, restart/migration of containers, container calibration and ramp-up time. First, the effects of isolation on the execution time of a task may not be the priority concern when looking at the system performance over a larger time. However in an enterprise context, they would play an important role where service level agreements have to be guaranteed. Second, the PaaS middleware can restart/migrate a container transparently to the application, hence long running computations may need to be check pointed. Third, PaaS vendors typically do not provide information on the underlying hardware on which the containers are hosted, thus it is difficult to ascertain their computational capabilities. This can be a hindrance for time and resource planning for a given workload, and benchmarking and extrapolation may be required to do so. Finally, the computations can be quickly started on a PaaS but the ramp-up time may not be negligible for small workloads.

VI. Conclusion and Future Work

This paper presented a novel master-worker architecture for many task computing on PaaS clouds which decoupled distributed processes using a message broker to create a multi-agent system. The progress of the computation was also saved as messages in queues accessible to any agent. This allowed the agents to be state-less and dynamically switch between a master and a worker role. The results indicated efficient scalability for computationally intensive tasks. A 400 hour workload with 18,000 tasks could be completed in mere 52 minutes on 512 dynos in with 90% wall-time efficiency.

The multi-tenant nature of the PaaS cloud coupled with imperfect container isolation led to variable task execution times. This needs further investigation to fully understand the performance variations as it would be necessary for guaranteeing that the quality of service constraints are met. However, the ease of deploying an application, favorable ramp-up times as compared to other infrastructures and scalability provided a glimpse into the potential which PaaS platforms hold for MTC.

The future work would consist of comparing this architecture by employing alternate message brokers and on other platforms such as Windows Azure [29]. The scalability provided by clustered message brokers needs to be studied with a larger number of agents as the scalability of the proposed architecture is dependent on it. Also, the architecture can be extended for map-reduce applications which have recently become popular for data analysis.

Acknowledgements

The authors would like to thank Marko Dabrovic for providing the Sponza model used for rendering and Heroku engineers who provided valuable assistance for migrating the application.

References

[1] P. M. Mell and T. Grance, “SP 800-145. The NIST Definition of Cloud Computing,” Tech. Rep., 2011.

[2] I. Raicu, I. Foster, and Y. Zhao, “Many-task computing for grids and supercomputers,” in *Many-Task Computing on Grids and Supercomputers, Workshop on*, Nov. 2008, pp. 1–11.

[3] Y. Zhai, M. Liu, J. Zhai, X. Ma, and W. Chen, “Cloud versus in-house cluster: evaluating amazon cluster compute instances for running mpi applications,” in *State of the Practice Reports*, ser. SC ’11. ACM, 2011, pp. 1–10.

[4] X.-H. Sun and D. T. Rover, “Scalability of parallel algorithm-machine combinations,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 6, pp. 599–613, 1994.

[5] C. Wu, A. Kalyanaraman, and W. Cannon, “A scalable parallel algorithm for large-scale protein sequence homology detection,” in *Proc. of International Conference on Parallel Processing (ICPP)*, Sep. 2010, pp. 333–342.

[6] C. Moretti, M. Olson, S. Emrich, and D. Thain, “Highly scalable genome assembly on campus grids,” in *Proc. of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*. ACM, 2009, pp. 12:1–12:10.

[7] H. Kim, S. Chaudhari, M. Parashar, and C. Marty, “Online risk analytics on the cloud,” in *Proc. of 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2009, pp. 484–489.

[8] D. P. Anderson, “BOINC: A System for Public-Resource Computing and Storage,” in *Proc. of the 5th IEEE/ACM International Workshop on Grid Computing*. IEEE Computer Society, 2004, pp. 4–10.

[9] C. Banino, “Optimizing locationing of multiple masters for master-worker grid applications,” in *Proc. of the 7th international conference on Applied Parallel Computing: state of the Art in Scientific Computing*. Springer-Verlag, 2006, pp. 1041–1050.

[10] F. A. da Silva and H. Senger, “Scalability limits of Bag-of-Tasks applications running on hierarchical platforms,” *J. of Parallel and Distrib. Comput.*, vol. 71, no. 6, pp. 788–801, 2011.

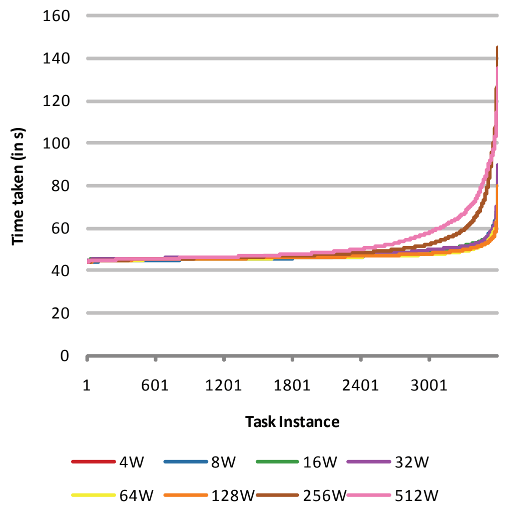
[11] A. Bendjoudi, N. Melab, and E.-G. Talbi, “An adaptive hierarchical masterworker (AHMW) framework for grids-Application to B&B algorithms,” *J. of Parallel and Distrib. Comput.*, vol. 72, no. 2, pp. 120 – 131, 2012.

[12] L. Wang, J. Zhan, W. Shi, and Y. Liang, “In Cloud, Can Scientific Communities Benefit from the Economies of Scale?” *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 2, pp. 296–303, Feb. 2012.

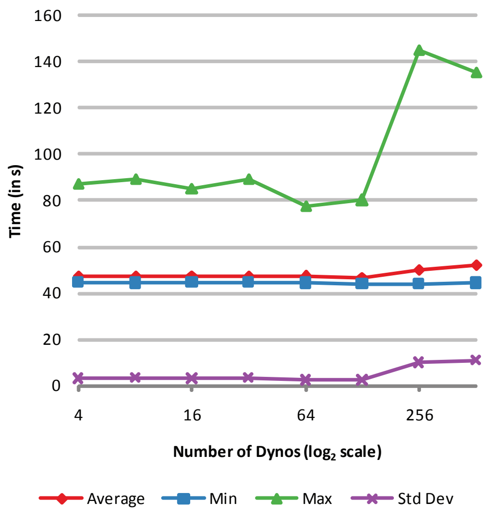
[13] R. Moreno-Vozmediano, R. Montero, and I. Llorente, “Multicloud Deployment of Computing Clusters for Loosely Coupled MTC Applications,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 6, pp. 924 –930, Jun. 2011.

[14] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, “Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 6, pp. 931–945, Jun. 2011.

[15] R. Prodan, M. Sperk, and S. Ostermann, “Evaluating High-Performance Computing on Google App Engine,” *IEEE Software*, vol. 29, no. 2, pp. 52–58, Mar.-Apr. 2012.

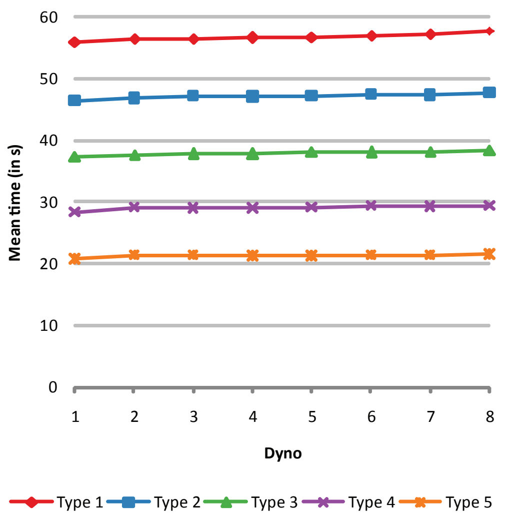


(a) Time taken

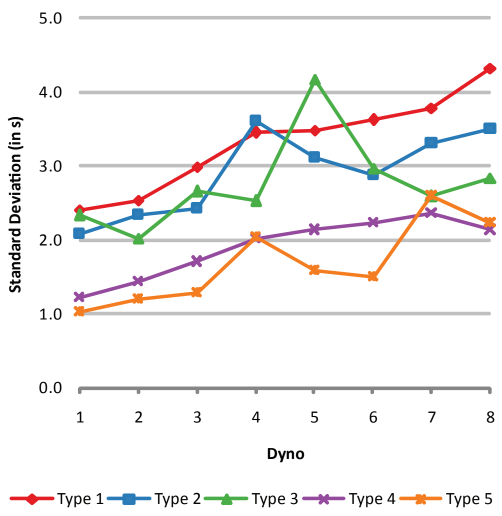


(b) Time taken statistics

Fig. 7: Execution times for Type 2 tasks of W3



(a) Mean Time



(b) Standard Deviation Time

Fig. 8: W3 Task Execution Times

[16] (2012) Google App Engine. [Online]. Available: https://developers.google.com/appengine/

[17] (2012) Amazon Elastic Compute Cloud (Amazon EC2). [Online]. Available: http://aws.amazon.com/ec2/

[18] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice Hall, 2002.

[19] (2012) Advance Message Queuing Protocol. [Online]. Available: http://www.amqp.org/

[20] J. O'Hara, "Toward a Commodity Enterprise Middleware," *Queue*, vol. 5, no. 4, pp. 48–55, May 2007.

[21] V. Sharma, S. Sengupta, and K. Annervaz, "ReLoC: A Resilient Loosely Coupled Application Architecture for State Management in the Cloud," in *IEEE 5th International Conference on Cloud Computing (CLOUD)*, Jun. 2012, pp. 906–913.

[22] J. T. Kajiya, "The rendering equation," *SIGGRAPH Comput. Graph.*, vol. 20, no. 4, pp. 143–150, Aug. 1986.

[23] V. Aggarwal, K. Debattista, T. Bashford-Rogers, P. Dubla, and A. Chalmers, "High-fidelity interactive rendering on desktop grids," *IEEE Comput. Graph. Appl.*, vol. 32, no. 3, pp. 24–36, 2012.

[24] (2012) Heroku, Cloud Application Platform. [Online]. Available: http://www.heroku.com/

[25] (2012) MongoDB, NoSQL Database. [Online]. Available: http://www.mongodb.org/

[26] (2012) RabbitMQ Message Broker. [Online]. Available: http://www.rabbitmq.com/

[27] (2012) Amazon EC2 Compute Unit. [Online]. Available: http://aws.amazon.com/ec2/instance-types/

[28] (2012) Heroku Dyno Isolation. [Online]. Available: https://devcenter.heroku.com/articles/dynos#isolation-and-security

[29] (2012) Windows Azure: Microsoft's Cloud Platform. [Online]. Available: http://www.windowsazure.com/