DECENTRALISED WORKLOAD SCHEDULER FOR RESOURCE ALLOCATION IN COMPUTATIONAL CLUSTERS

Leszek Sliwko and Vladimir Getov
Distributed and Intelligent Systems Research Group
University of Westminster, 115 New Cavendish St., London W1W 6UW, U.K.

Leszek.Sliwko@my.westminster.ac.uk, V.S.Getov@westminster.ac.uk

Technical Report, June 2018

Abstract: This paper presents a detailed design of a decentralised agent-based scheduler, which can be used to manage workloads within the computing cells of a Cloud system. This scheme in based on the concept of service allocation negotiation, whereby all system nodes communicate between themselves and scheduling logic is decentralised. The architecture presented has been implemented, with multiple simulations run using real-word workload traces from the Google Cluster Data project. The results were then compared to the scheduling patterns of Google's Borg system.

Keywords: Cloud computing, workload scheduling, resource allocation, platform as a service

1. INTRODUCTION

Cloud computing has become a very widespread phenomenon in daily lives, especially if we consider how accepted smart phones are. A typical phone application is heavily depended on infrastructure and remote processing capabilities Clouds deliver. Also, in desktop computer's programs area, a significant number of applications have been moved into Software-as-a-Service model, where desktop serves only as a client, but actual processing is done on a remote Cloud, e.g.: business subscriptions for Microsoft Office 365 offer access to cloud-hosted versions of Office programs. Therefore, the current world is more and more reliant on Cloud computing.

A variety of vendors, services and business models have created an extremely complex environment. The latest developments in Cloud technologies gravitate towards federated, inter-Cloud cooperative models and, therefore, already very complex solutions are destined to become even more complex. It's is also a very competitive market. A recent rise in Big Data systems fuelled a growth in demand of cheap computing power; however, a number of vendors joined and market offering computing services has greatly expanded. It also has driven prices down; a 4GB machine with Intel Xeon (Haswell) cores can be rented out for as little as two cents per hour.

The main difference between a Cluster and a Cloud is business model and access. Clusters were traditionally available only to a very limited number of institutions and corporations, who pooled their resources in order to get advantage of scale of computing. Public generally did not have access to computing capabilities of Clusters.

Because access to Clusters was very restricted and incoming workload was planned well in advance, it was practical to centralise task scheduling and system management functions.

Therefore, a centralised architecture was generally used a base for Cluster management software, such as SLURM (Yoo et al., 2003), Maui (Jackson et al., 2001), Univa Grid Engine (Gentzsch, 2001), etc.

One of the most developed and published Cluster managers is Google's Borg system. The current concept behind Borg is to deploy several schedulers working in parallel. The scheduler instances are using a share state of available resources; however, the resource offers are not locked during scheduling decisions (optimistic concurrency control). In case of conflict, when two or more schedulers allocated jobs to the same resources, all involved jobs are returned to the jobs queue and scheduling is re-tried (Schwarzkopf et al., 2013). When allocating a task, Borg's scheduler is scoring a set of available nodes (best-fit algorithm) and selects the most feasible machine for this task. The central module of Borg architecture is BorgMaster, which maintains an in-memory copy of most of the state of the cell. Each machine in a cell is running BorgLet, an agent process responsible for starting and stopping tasks and also restarting them if they fail. A single BorgMaster controller is able to manage a cell of more than 12k machines (highest value is not specified). Google's engineers achieved this impressive result by a number of optimisations and, so far, they've managed to eliminate or work around virtually every limitation they have approached (Verma et al., 2015). Nevertheless, a centralised architecture is a limitation within a current design of a Cloud.

One might also ask a fundamental question – do Cloud systems really require more inter-connected nodes in one cell? Computing power of 12k machines working together is already a very considerable force and, barring few exceptions, it's highly unlikely that an application would require such a processing power. However, in recent years, software development is trending towards Big Data systems. Big Data systems are characterised by a high degree of parallelism. A typical Big Data system design is based on a distributed file system, where nodes have dual function of storing data as well as processing it. One process of such system might need to crunch thousands of TBs of data split across thousands of nodes. Even with ideal allocation of Big Data tasks, where every task is processing data only available locally (i.e. locality optimisation), a single node would still need to process GBs of data locally. Therefore, the answer to the above question is yes – it's highly likely that we will require larger computing cells in the near future.

Therefore, since centralised architectures have their ultimate scalability limits, we shall consider alternative approaches. In this paper we present a working prototype of decentralised Cloud manager – Multi-Agent System Balancer (MASB), which relies on a network of software agents to organically distribute and manage good system load. MASB prototype has been built on top of AGOCS framework (Sliwko and Getov, 2016) and such all research and development has been continuously tested on real-world workload traces from Google Cloud.

The remaining of this paper is organised as follows. Section two provides introduction to agent systems and a brief of research history how they have been used to schedule tasks. Section three describes MASB's design principles, defines scope of project and lists all main used technologies. Section four has details about MASB architecture and its objects. Section five present allocation score functions, which are modelling distribution of tasks and their allocations across nodes through tasks' lifecycles. Section six describes in detail the allocation negotiation protocol that is used between system components to allocate new task or re-allocate existing task on a node. Section seven specifies experiments, which were performed as a part of his research and provides a discussion on achieved results. Section eight describes competitive solutions. Section nine provides a summary of important optimisations and lessons learnt along with the conclusions.

2. Load Balancing with Agents

Agent technologies can be dated back to 1992 (Sargent, 1992), at which point it was predicted that intelligent agent would become the next mainstream computing paradigm. Agents were described as the most important step in software engineering, as well as a new revolution in software (Guilfoyle and Warner, 1994). Since its inception, the field of multi-agent systems has experienced an impressive evolution, and today it is an established and vibrant field in computer studies. By its nature, software agents research field spans across many related disciplines, including mathematics, logic, game theory, cognitive psychology, sociology, organization science, economics, philosophy, and so on (Weiss, 2013). Agents are considered to be a viable solution for large-scale systems, for example through spam-filtering and traffic light control (Brenner et al., 2012), or by managing an electricity gird (Brazier et al., 2002).

Software agents are found across many computer science disciplines, including AI, decentralised systems, self-organising systems, load balancing and expert systems (Guilfoyle and Warner, 1994; Milano and Roli, 2004; Cabri et al. 2006). Previous research has also shown that by deploying agents it is possible to achieve good global system performance (Nguyen et al., 2006), improve system stability and reduce downtime (Corsava and Getov, 2003), attain dynamic adaptation capability (Kim et al., 2004) and to realise robustness and fault-tolerance (Xu and Wims, 2000). Agents were also found to be useful for the performance monitoring of distributed systems (Brooks et al., 1997). Several additional benefits may also be obtained, including more cost-effective resource planning (Buyya, 1999), a reduction of network traffic (Montresor et al., 2002), the autonomous activities of the agents (Goodwin, 1995), and decentralised network management (Yang et al., 2005).

It is difficult to argue for any precise definition of an agent, although it is generally defined as a component of software and/or hardware capable of acting independently in order to accomplish tasks on behalf of its user (Nwana, 1996). An agent can be described as a being which is supposed to act intelligently according to environmental changes and the user's input (Goodwin, 1995).

The research literature seems to suggest there are four key properties of an Agent (Castelfranchi, 1994; Gensereth and Ketchpel, 1994; Wooldridge and Jennings, 1995):

- Autonomy when allowing agents to operate without direct human intervention;
- Social ability when agents communicate and interact with other agents;
- Reactivity when agents actively perceive their environment (physical or digital) and act on its changes;
- Pro-activeness when agents not only respond to changes in environments but are also able to take initiative and exhibit goal-oriented behaviour.

Agent-based systems often rely on decentralised architecture (Jones and Brickell, 1997; Shi et al., 2005), considering it to be more reliable. However, resources are handled through much more complex algorithms, with negotiation schemas often being required for distributed architecture to attain a good level of performance (Bigham and Du., 2003; Yang, 2005).

The idea of job scheduling with agents is not new; a single-machine multi-agent scheduling problem was introduced in 2003 (Baker and Smith, 2003) and (Agnetis et al., 2004). The problem since then has been extended, and at present exists in several variations, such as deteriorating jobs (Liu and Tang, 2008), the introduction of weighted importance (Nong et al., 2011), scheduling with partial information (Long et al., 2011), global objective functions (Tuong et al., 2012), and adding jobs' release times and deadlines (Yin et al., 2013).

All this previous research has contributed immensely to the state of knowledge, with the papers below in particular having influenced the design of the MASB.

- (Schaerf et al., 1995) presents a study about a multi-agent system in which all decision making is performed by a learning AI. The likeness of selection of a particular node for the processing of a given task depends on the past capacity of this node. The Agent’s AI uses only locally-accessible knowledge, meaning that it does not rely on information shared by other agents.

- (Chavez et al., 1997) introduces Challenger, a multi-agent system, in which agents communicate with each other to share their available resources, in an attempt to utilise them more fully. In Challenger, agents act as buyers and sellers in a resources marketplace, always trying to maximise their own utility. MASB follows like pattern in of nodes trying to maximise its allocation (utilisation) score. The cooperative model in which agents negotiate between themselves is the base of distributed scheduling presented in this research.

- (Bigham and Du, 2003) shows that cooperative negotiation between agents representing base stations in a mobile cellular network can lead to a near global optimal coverage agreement within the context of the whole cellular network. Instead of using a negotiation model of alternating offers, a number of possible local hypotheses are created and parallel negotiations are initiated based on them. The system commits to the best agreement found within a defined timeline.

- (Kim et al., 2004) proposes a load-balancing scheme in which a mobile agent pre-reserves resources on a target machine prior to the occurrence of the actual migration. The system also prevents excessive centralisation through the implementation of a mechanism whereby when the workload processed on a particular machine exceeds a certain threshold, this machine will attempt to offload its agents to neighbour machines.

- (Cao et al., 2005) describes a solution in which agents representing a local grid resource uses past application performance data and iterative heuristic algorithms to predict the application’s resource usage. In order to achieve a globally-balanced workload, agents cooperate with each other using a P2P service advertisement and discovery mechanism. Agents are organised into a hierarchy consisting of agents, coordinators and brokers, who are at the top of the entire agent hierarchy. The authors conclude that for local grid load balancing, the iterative meta-heuristic algorithm is more efficient than simple algorithms such as FCFS.

- (Ilie and Bădică, 2013) details a solution based on the ant colony algorithm, a solution which takes its inspiration from the metaphor of real ants searching for food. ‘Ants’ are software objects that can move between nodes managed by agents. A move between nodes which is managed by the same agent is less costly. As ‘ants’ explore paths between nodes, they mark them with different pheromone strength. Whenever an ‘ant’ visits a node, the agent managing it saves the recorded tour, and updates its own database. Ants who subsequently visit this node read its current knowledge, meaning they have the potential to exchange information in this environment.

- (Klusáček et al., 2013) – authors have implemented an extension to TORQUE scheduler which uses TS algorithm to optimise jobs queue. This routine is cyclically run for a limited period (10 seconds) and the queue is re-arranged based on the best solution found within this cycle. MASB implements similar approach when selecting a set of candidate services when a node is overloaded (see Section 6.1).

- (Eddy et al., 2015) presents a prototype in which agents operate electricity market. Agents exchange ‘offers’ and ‘bids’ for those offers via a custom-designed communication protocol based on TCP/IP. Among other specialised agents, system implements a short-lived coordinating agent to facilitate those exchanges and ensure the supply of electricity is managed. A comparable schema is implemented in MASB, in which the Broker Agent (BA) initially advises of candidate target nodes where an overloading service can be re-allocated.

- (Lewis and Oppenheimer, 2017) – the scoring functions implemented in Kubernetes evaluate the balance of utilised resources (CPU and memory) on a scored node (BalancedResourceAllocation routine). Kubernetes scheduler tries to preserve proportional allocation of all resources – this approach has been further expanded into two types of scoring functions implemented in MASB (discussed in Section 5).

3. MASB Design Principles

Many other Cluster managing systems, such as Google’s Borg (Verma et al., 2015), Microsoft’s Apollo (Boutin et al., 2014) and Alibaba’s Fuxi (Zhang et al., 2014b), were built around the concept of the immovability and unstoppability of a task’s execution. This means that once a task is started it cannot be reallocated: it can only be stopped/killed and restarted on an alternative node. This design is particularly well suited for when there is a high task churn, as observed in Apollo or Fuxi where tasks are generally short-lived, meaning that the system’s scheduling decisions do not have a lasting impact. However, in order to support a mixed workload which features both short-lived tasks and long-running services, alternative solutions needed to be developed, such as the resource recycling routines present in Borg in which resources allocated to production tasks but not currently employed are used to run non-production applications (Verma et al., 2015).

MASB takes advantage of virtualisation technology features, namely VM-LM, to dynamically reallocate overloading tasks. VM-LM allows programs which are running to be moved to an alternative machine without stopping their execution. As a result, a new type of scheduling strategy can be created which allows for the continuous re-balancing of the cluster’s load. This feature is especially useful for long-term services which initially might not be fitted to the most suitable node, or where their required resources or constraints change. Nevertheless, this design creates a very dynamic environment in which it is not sufficient to schedule a task only once. Instead, a running task has to be continuously monitored and reallocated if the task’s current node cannot support its execution any longer. As experiments in (Sliwko and Getov, 2016) with a centralised load balancer based on meta-heuristic algorithms demonstrated, a scheduling strategy implemented on a single machine is highly unlikely to efficiently manage a large number of tasks due to the high overheads of these algorithms.

Therefore, MASB has been built around the concept of a decentralised load balancing architecture which could scale well beyond the limits of a centralised scheduler. This model is backed by P2P communication in which software agents negotiate the task allocations on behalf of their nodes. NAs actively monitor the used resources and are themselves responsible for keeping their nodes stable. When an NA detects that a node is overloaded, it will attempt to find an alternative node for overloading tasks with the help of an inter-node SAN protocol. The first step of SAN communication is to retrieve a set of suitable candidate nodes from a subnetwork of BAs which cache the state of the computing cell. However, because the information found in BAs is assumed to be outdated, more steps are required to successfully offload a task.

The MASB project has been developed over several years, during which time it has undergone many changes in terms of both the technology used and the design of the architecture, such as migration from Java to Scala, change from thread pools to an Akka Actors framework, introduction and use of the concurrency packages and non-locking object structures, etc. However, the main design principles have not been altered and are presented below:

- To provide a stable and robust (i.e. no single point of failure) load balancer and scheduler for a Cloud-class system;
- To efficiently reduce the cost of scaling a Cloud-class system so that it is able to perform in an acceptable manner on smaller clusters (where there are tens of nodes) as well on huge installations (where there are thousands of nodes);
- To provide an easy way of tuning the behaviours of a load balancer where the distribution of tasks across system nodes can be controlled.

The following challenges remained outside the scope of this research:

- MASB does not address fault-tolerance issues in decentralised scheduling. In centralised architecture, fault-tolerance can be achieved through the implementation of a centralised database of tasks and their statuses. When task execution fails, the system just reschedules unsuccessful tasks to alternative nodes. This approach, however, will not work in a decentralised system since there is no central database of tasks.
- The majority of centralised schedulers implement some kind of usage quota, e.g. per user, user group or other entity. Due to the decentralised nature of MASB, it is difficult to design a reliable usage monitoring and limiting system that is not reliant on central storage.

The main technologies and specific features of MASB use are listed below:

- VM-LM allows the transfer of a running application within the VM instance to an alternative node without stopping its execution. At present, the vendors' strategy is to implement mixed production and low-priority jobs on a single machine. While production jobs are idler, low-priority jobs consume the nodes' resources. However, when production's jobs resources need to be increased, the low-priority jobs are killed. Google Cluster (Verma et al., 2015) and Amazon EC2 (Yi et al., 2012) use such an approach. MASB takes advantage of VM-LM to offload services without stopping their execution, collecting information about services in order to estimate the cost of VM-LM of such a service.
- Decentralised agents network – software agents create a network of independent AI entities that can negotiate between each other and allocate Cloud workload between them. In MASB, each NA is responsible for its own node and manages its own workload. Because of the decentralised nature of MASB there is no complete up-to-date system state, meaning that each BA has data about a subset of nodes, and that it periodically exchanges these data with other BAs. The BA provides an interface whereby the NA can request a set of candidate nodes which a particular service or task can be migrated to. Once the NA receives a set of candidate alternative nodes for a particular service, it communicates with their NAs so as to migrate this service.
- Meta-heuristic selection algorithms – while the majority processing of load balancing logic is done via negotiation between NAs, there are a few system processes that are better handled locally by the node itself. One such example is that when NA discovers its node is overloaded, it will select a subset of its services which it will attempt to migrate out. This selection is performed by Tabu-Search. For previous research on meta-heuristic algorithms that focus on scheduling, see (Sliwko and Getov, 2016).
- Functional Programming language Scala and Akka Actors/Stream framework – due to the decentralised design and loose coupling between the system's components, the implementation language is of secondary importance. However, load balancing algorithms require a significant amount of tuning, especially if the Cloud is designed to have a high utilisation

of available resources. This would mean that resource waste is low, and therefore the cost-per-job execution is also low. Due to the complexity of inner-system relations and dependencies, a high-fidelity simulation environment is necessary to evaluate the expected performance of a given configuration and implemented changes before is deployed to a production system, e.g. the FauxMaster simulator used by Google Engineers (Verma et al., 2015). To develop such a highly concurrent program, an Akka actors' framework has been used since it has a very low overhead per instance, roughly 300 bytes. Additionally, Scala allowed access to a wide range of mature Java libraries.

4. MASB ARCHITECTURE

MASB consists of four main components:

- Networked servers/nodes and other hardware layer infrastructure;
- Event objects that are passed around system components and which carry system information. Event objects are immutable and have a timestamp which ensures the information carried is never modified;
- NA representing nodes and BA responsible for gathering system statistics data and providing a resource quoting mechanism;
- Services or tasks – applications and batch jobs executed on Cloud nodes.

4.1. EVENT TYPES

In order to avoid costly broadcasts, since broadcast packages need to be rerouted through a whole network infrastructure consuming the available bandwidth, both NA and BA always communicate point-to-point. In the system there are several types of requests and responses between agents as shown in Table 1.

4.2. NODE AGENT

In decentralised load balancer design, NA represents every system node. NA the migration of services to an alternative node whenever the node's resources are over-utilised. In addition, NA is responsible for periodically reporting the node status and the state of resources to BAs.

4.2.1. NODE RESOURCES MONITORING AND REPORTING

NA continuously monitors the levels of defined resources and periodically reports the state of its node and levels of utilised resources to BA. Should any of the monitored resources be over-allocated, NA will initialise a service allocation negotiation process.

4.2.2. ACCEPT/DENY SERVICE MIGRATION REQUESTS

NA also listens to service migration requests and accepts or denies them. This routine is simple, with NA projecting its resource availability with that service as follows: projected allocation of resources = current allocation of resources (existing services which also includes services being migrated out from this node) + all services being migrated to this node + requested service (from request). If the projected resources do not overflow the node, the service is accepted, and service migration is started. The source NA does not relinquish ownership of the migrated service until the service is successfully migrated to the target node. It should be noted that during service migration, its required resources are allocated twice, to both the source node and the target node.

4.2.3. SERVICE MIGRATION

After accepting the service migration request, NA immediately starts listening for incoming VM-LM. In order to perform service migration, NA must have access to the administrative functions of VM and be able to initiate VM-LM to another node. This functionality can be either implemented by the calls of the VM manager API or by executing the command line command. This process may vary considerably per VM vendor.

Request Type	Description
GetCandidateNodesRequest	Requests a number of candidate nodes for the migration of a specified services set. This request is sent from NA to BA.
GetCandidateNodesResponse	Reply with a set of candidate nodes for service migration, together with their resource statistics.
TaskMigrationRequest	Request from source NA to candidate NA as to whether service migration is accepted.
TaskMigrationAcceptanceResponse	Replay from target candidate NA that service migration will be accepted. Note: No resource allocation takes place after this request.
TaskMigrationRejectionResponse	Replay from target candidate NA that service migration will not be accepted.
TaskMigrationProcessRequest	Request to selected candidate NA to start service migration. Note: this request has an optional forced flag, requesting the target NA to skip the currently available resources check. The total node's resources check and constraints check will be still performed.
TaskMigrationProcessConfirmationResponse	Confirmation from the target NA that the service migration process can start. Note: Resources are allocated for the migrated service and the live migration process starts.
TaskMigrationProcessErrorResponse	Denial of service migration process. This reply is generated if the NA configuration – he available resources or constraints – can no longer accommodate the migrated service.

Table 1: Request types

4.3. Broker Agent

In this system, in addition to NAs, BA which is responsible for storing and maintaining information about nodes' online status and their available resources, is also present. NA periodically reports to its BA about the state of its node and available resources. These data are then used to compute a list of suitable nodes. BA is a separate process which can coexist with NA on the same node since its operations are not computing-intensive. BA has two main purposes in the system which are outlined below.

4.3.1. Nodes Resources Utilisation Database

NAs periodically update BA with their currently available and total resource levels. BA stores all this data and can query them on demand. Every node entry is additionally stored with its timestamp, showing how long ago the data were updated. As additional protection against the node silently going offline, e.g. through hardware malfunction or the network becoming unreachable, if this entry is not updated for three minutes, the node is assumed to be offline and entry is removed. This means that it will not be returned as the candidate node.

4.3.2. Computing Service Migration Candidate Nodes

BA listens for GetCandidateNodesRequest and computes a list of candidate nodes for service migration. In order to create a list of candidate nodes, BA retrieves nodal data from the local cache and then scores them using Allocation Scoring Function.

Candidate nodes are scored in a forecasting manner, i.e. as if a certain service was assigned to them. Examples include:

- The service being migrated currently utilises 1 CPU code and 2 GB of memory;
- The scored node total resources constitute 8 CPU cores and 16 GB of memory;
- All other services on the scored node currently use 3 CPU cores and 5 GB of memory.

When scoring this node, the score will be calculated from a forecasted resource usage of 3+1=4 CPU cores and 5+2=7 GB of memory. This enables BA to score the future state of the system, as if service migration were being carried out. After scoring all the cached nodes, BA selects a configured number of candidate nodes with the highest score and sends them back to the asking node. In this research this number was set to fifteen candidate nodes, wherein a higher number failed to yield superior results.

5. Allocation Scoring Functions

Scoring functions are used when a new task is allocated or when a system needs to migrate a task to an alternative node. Scoring functions return a score value which evaluates how well a given task will fit a scored node system-wise. If a node cannot fulfil a task's constraints, the node is deemed non-suitable and the scoring function is undefined.

Google's Borg implements a set of scoring routines (Verma et al., 2015). Unfortunately, being part of Google's proprietary code, the details are unavailable. In this implementation, the scoring function input is constructed from the total node resources, the currently available node resources and the currently required resources for a given task. It should be noted that scoring functions return a score when a task fits the available resources on a node, and also when a node is overloaded by a task. In addition, scoring functions should not be computation-expensive, especially when a new task is allocated.

The node SAS function is a crucial part of the system that greatly impacts global resource usage level, that is, determining how well nodes' resources are utilised. This research concludes that node allocations are failing in six separate areas:

- Idle Node – a completely idle node is a special case of allocation, in which no service has been allocated to this node. Such a node could be completely shut down, resulting in lower power usage for a cluster. In this research, idle nodes are scored most highly when determining a suitable node for initial service allocation.

- Super-Tight Allocation (STA) – where some of node's resources are utilised in the 90%-100% range. STA is regarded as stable allocation; however, due to the dynamic resource usage, this is actually not a desirable scenario. Complete, or almost complete, resource usage can frequently lead to resource over-allocation, whereby one or more services increase their resource utilisation. This experimentation has determined that leaving 10% of any given resource unutilised gives the best results since it reduces service migration (see discussion in Section 5.4).

- Tight Allocation (TA) – where all node resources are utilised in the 70-90% range. This is the most desirable outcome as it promotes the best fitting allocation of services and, therefore, low resource wastage.

- Proportional Allocation (PA) – while tight-fit is the most desirable outcome, the majority of services in this research consumed a small amount of each resource. The majority of scheduled services are short batch jobs which have a very short execution time. In such a scenario, it is desirable to keep proportional resources' usage ratios on all nodes which would, therefore, generally enable nodes to fit more services with ease.

- Disproportional Allocation (DA) – where the node's resources are not proportionally utilised, thereby making it difficult to allocate additional services if required. E.g. a setup where services on a node allocate 75% of CPU, but only 20% of memory, is not desirable.

- Overloaded Node – when allocated resources overload the total available resources on the node. Naturally, this is an unwanted situation, and such a node is given a score of zero.

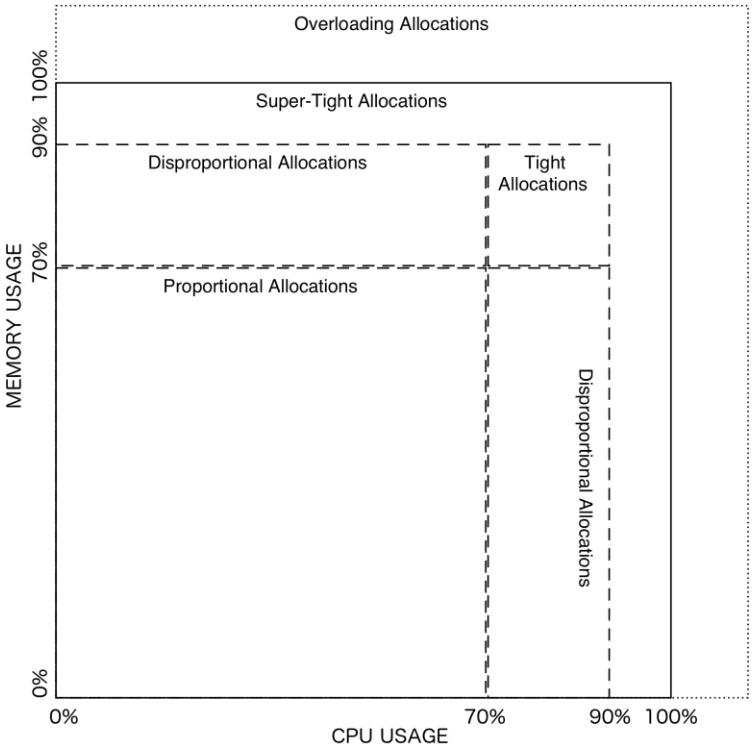


Figure 1: Allocation Score types (two resources)

Figure 1 illustrates all five areas for the two resources (CPU and memory). The scoring function should never allow overloading allocations to take place. Additionally, during research, it was determined that STAs are very prone to over-allocate nodes and are damaging to overall system stability. Therefore, they are also accorded a score of zero. DAs increase global resource wastage and should be avoided; nevertheless, they are acceptable if none of the other types of allocations are possible.

5.1. Service Initial Allocation Score

Tightly fitting services on as few nodes as possible are beneficial for global system performance. However, the initial allocation should rather aim to distribute services across nodes. During this research, two main reasons were identified:

- Initially, a Cloud user specifies the task's required resources. User tend to overestimate the amount of resources required, wasting in some cases close to 98% of the requested resource (Moreno et al., 2013). Therefore, only after the task is executed could realistic resource utilisation values be expected. Allocating new services in a tight-fit way does result in turmoil when the service is actually executed, and the exact resource usages levels are logged.

- MASB uses a network of BAs to provide a set of the best candidate nodes (nodes with the highest AS) – to allocate the service. However, the nodes' statistics are not very accurate and are delayed. Big data systems often send multiples of an identical task in a batch. Those tasks execute the same program and have the same (or very similar) resource requirements. As such, a limited set of nodes will be highly scored. The result of repeatedly allocating multiple services to the same node over a very short period of time results in collisions.

Therefore, when initially allocating existing services, candidate nodes should be scored in the following order: PA, TA and finally DA. Figure 2 is a graphical representation of such a scoring function. It should be noted that the maximum resource usage is 90%, and that values above this level are in an undesired STA's area.

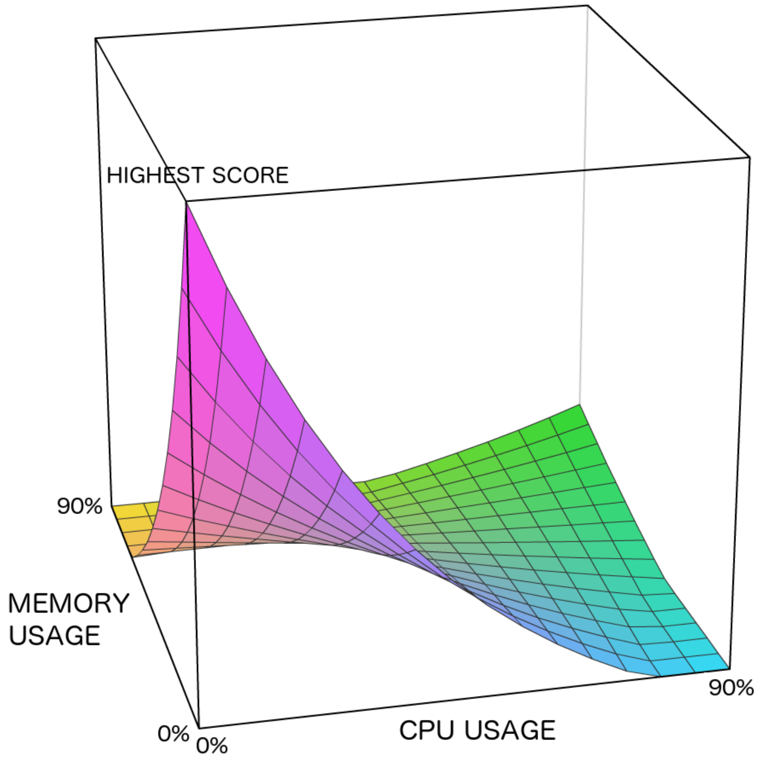


Figure 2: Service Initial Allocation Score (two resources)

Three separate areas can be noticed:

- Lower-left (the highest score) – this promotes PA that will leave resource utilisation on a low level or proportionately used.
- Upper-right corner (the medium score) – this promotes TA, where tasks on this node will closely utilise all of its resources.
- The upper-left and lower-right corners (the lowest score) – these DAs will leave one resource utilised almost fully and the other resource wasted.

As an additional optimisation, only a limited number of candidate node recommendations are calculated, in this specific situation two hundred, before selecting the top recommendations. This is to prevent scoring routine calculations from processing for too long. The limit of two hundred applies only to non-forced recommendations for matching nodes.
It should be noted that the SIAS is calculated exclusively from user-defined resource requirements since the actually-used resource requirements are unknown before the task execution actually starts.

5.2. Service Reallocation Score

This research has found that the best results are achieved when tasks are packed tightly into available nodes, i.e. where global resource utilisation is the highest. Therefore, the best fit scenario, where the task fully utilises 90% of all available resources on a node, is scored the highest. In addition, all scoring functions should minimise situations when allocated tasks fully utilise only a few resources, as this prevents additional tasks from being added to a node, and adds to global resource waste.

The most viable approach identified is to score the best fit scenarios the highest and then score allocations that will leave few resources unutilised, while simultaneously allocating other resources fully as the lowest. However, the majority of scoring queries fall between those two scenarios. In such cases, it was beneficial to promote balanced tasks allocations, that is, allocating tasks to nodes that will utilise available resources in the most proportional manner.

Therefore, when migrating existing services, candidate nodes should be scored in the following order: TA, PA, then DA. Figure 3 is a graphical representation of this scoring function:

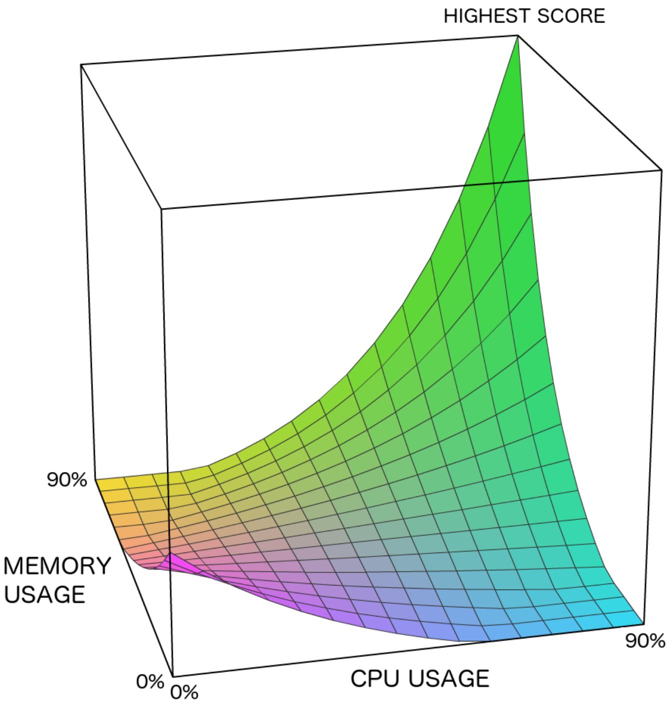


Figure 3: Service Reallocation Score (two resources)

Three separate areas can be noticed:

- Upper-right corner (the highest score) – this promotes TA, where tasks on this node will closely utilise all of its resources.

- Lower-left (the medium score) – this promotes PA that will leave resource utilisation on a low level or proportionately used.
- The upper-left and lower-right corners (the lowest score) – these DAs will leave one resource utilised almost fully and the other resource wasted.

Similar to calculating recommendations for new tasks, as an additional optimisation, only a limited number of candidate node recommendations are calculated before selecting the top recommendations. However, because this routing is invoked much less frequently, two thousand nodes are analysed. The two thousand limit applies only to non-forced recommendations for matching nodes. It should be noted that the SRAS is calculated exclusively from actually-allocated resource requirements. User-defined resource requirements are evaluated as part of the Resource Usage Spikes routine, explained in detail below.

5.3. Service Allocation Lifecycle

As explained in the sections above, the ideal scenario for a service is to be initially allocated on a lowly-utilised node, before it is gradually migrated towards more tightly-fitted allocations with other tasks, i.e. in order to lower global resource waste.

Originally, the MASB framework did not have distinct scoring functions for SIAS and SRAS; a single SAS function, with the same scoring model as SRAS, was used for all allocations. However, when scaling workload for simulations, it was discovered that user-defined resource requirements can differ significantly from actually-allocated resources, and that initially tightly fitting them onto the nodes is counter-productive. More nodes need to be scored which therefore consumes more CPU time when allocating a single task. It should further be noted that the majority of tasks are short lived and generally do not exist long enough in the system to be moved to an alternative node. Therefore, the design was altered to prioritise the allocation of new tasks to relatively unutilised nodes, and prioritise the reallocation of existing and currently running services to nodes where it will tightly fit with other programs running there.

In a researched Cloud system, only about 20-40% of tasks qualify as long-running services, meaning that they run for longer than 12-20 minutes (Schwarzkopf et al., 2013). The remaining scheduled tasks consisted of short-term jobs which generally have much lower resource requirements than long-running services. The majority of tasks are short and will not exist for long at all in the system. Therefore, it is important for an initial allocation not to spend too much time in trying to tightly fit them into available nodes.

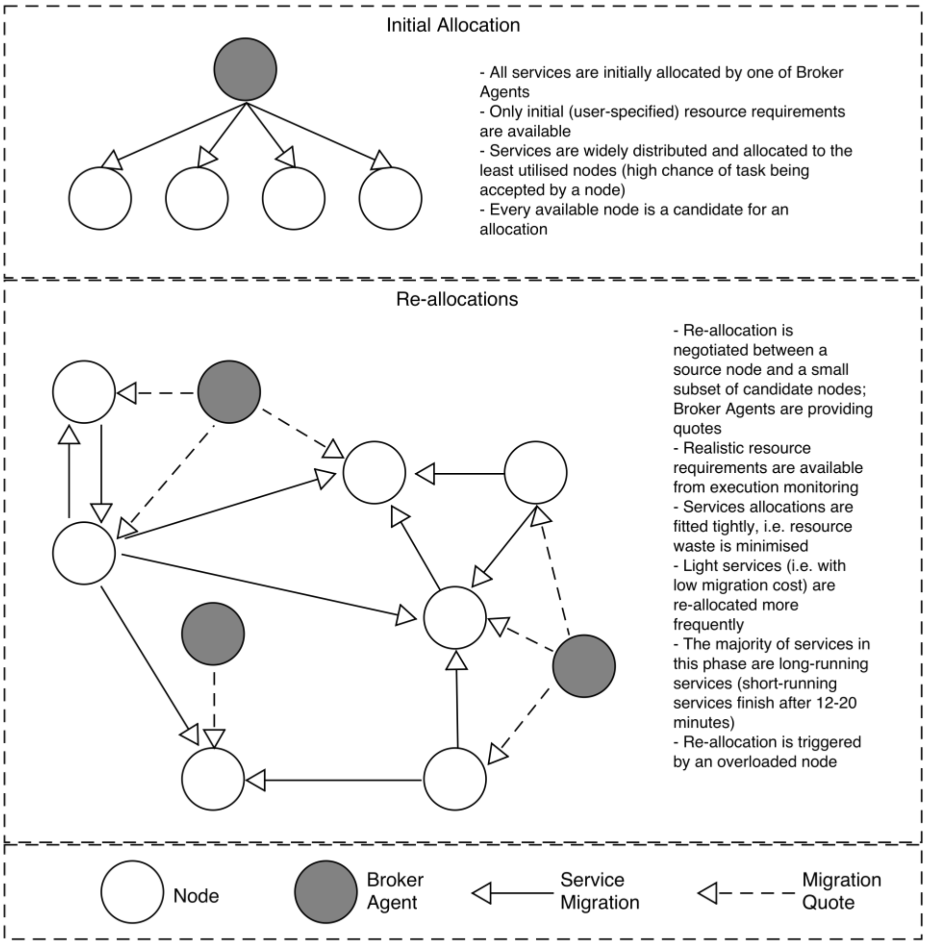
While the majority of tasks are short, and are rarely migrated after the initial allocation, there exists a number of long-running services that have more demanding resource requirements, meaning that the majority of resources (55–80%) are allocated to service jobs (ibid.). Therefore, it is more difficult to fit them into nodes, and these allocations should be much tighter to minimise global system resource waste. Figure 4 represents the desired service allocation lifecycle.

However, in order to allocate a service, an additional constraint has to be considered - Resource Usage Spikes which is outlined in section below.

5.4. Resource Usage Spikes

At times, a task might instantly increase its resource usage; as such times, a node should have the capacity to immediately accommodate this request, without needing to migrate the task to an alternative node. In such a situation, other VMs running on this machine can be paused or killed to let the VM instance executing this task instantly allocate more resources. In this research, Resource Usage Spikes (RUS) were defined as being when there was a greater than 10% increase in the overall node resource utilisations levels in any of the monitored resources. A value of 10%

was selected because the previously described scoring functions aim to allocate node resources at 70-90% of total capacity, meaning that an increase of more than 10% could destabilise the node.



- All services are initially allocated by one of Broker Agents
- Only initial (user-specified) resource requirements are available
- Services are widely distributed and allocated to the least utilised nodes (high chance of task being accepted by a node)
- Every available node is a candidate for an allocation

- Re-allocation is negotiated between a source node and a small subset of candidate nodes; Broker Agents are providing quotes
- Realistic resource requirements are available from execution monitoring
- Services allocations are fitted tightly, i.e. resource waste is minimised
- Light services (i.e. with low migration cost) are re-allocated more frequently
- The majority of services in this phase are long-running services (short-running services finish after 12-20 minutes)
- Re-allocation is triggered by an overloaded node

Figure 4: Service Allocation Lifecycle

As such, an additional feature was implemented in MASB to handle RUS instances. Aside from checking the actually-used resources for tasks and ensuring that the node has the capacity to support it, the system also calculates the maximum possible resource usage of all production tasks based on user-defined resource requirements as well as making sure that the node has the capacity to support all production tasks at their full resource utilisation. This constraint is limited only to production jobs since VMs running non-production jobs can be suspended without disturbing business operations. The continuously fulfilment of this constraint is referred to as goal (IV).

The introduction of RUS constraint adds another dimension to the tasks allocations logic. Figure 5 visualises how user-defined resource requirements for production tasks and actually-used resources for all tasks are integrated. In this sixty-node sample, approximately half the nodes have a very high CPU user-defined allocation for production tasks, while the real usage is much lower. It should be noted that while memory usage stays proportionally high thorough the GCD workload, the gaps between the requested and actually used memory are much smaller. This is a relatively common pattern for GCD workload. Additionally, the chart marks the allocation type (STA, TA, DA, PA) for each node in this sample on the horizontal axis.

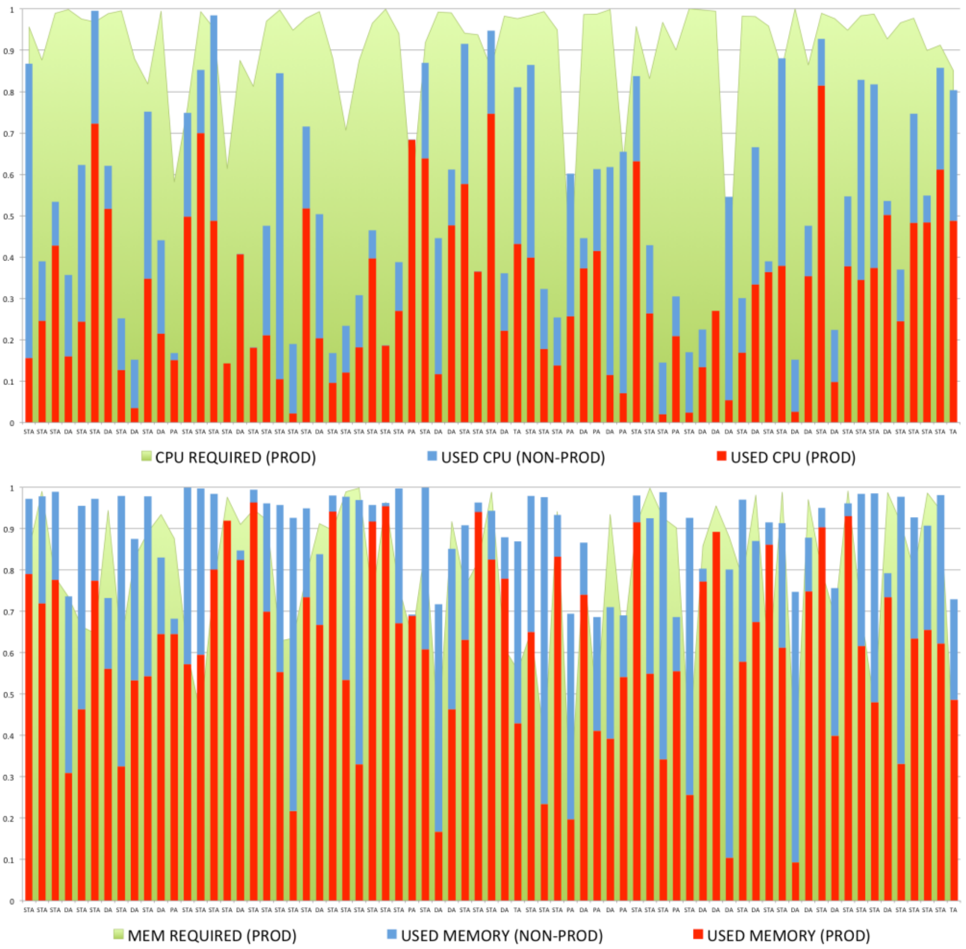


Figure 5: Production vs. non-production allocated resources (60-nodes sample)

Whilst RUS do not occur frequently, they do have the significant potential to destabilise an affected node. In examined GCD workload traces with ca.12.5k nodes and ca.140k tasks being continuously executed by them, RUS occurred with an estimated frequency of 212 per minute. Table 2 represents the average frequency of RUS with different resource increase thresholds, based on experiments.

RUS threshold	Average RUS (count per minute)	Peak RUS (count per minute)
5%	659	7538
10%	212	4362
15%	66	2390
20%	47	1925
25%	26	1135

Table 2: Resource Usage Spike frequencies

RUS are a significant design consideration. To cite an alternative solution to handle RUS, Google's engineers implemented a custom resource reservation strategy using a variant of step moving average, as detailed by John Wilkes in a presentation during the GOTO 2016 conference in Berlin (Wilkes, 2016).

6. Service Allocation Negotiation Protocol

When NA detects its node is overloaded, it will select a service (or a set of services) and attempt to migrate them to an alternative node or nodes. Since Service Allocation Negotiation (SAN) is an asynchronous process, this means a single NA can run several SANs in parallel. In the current implementation, NA selects a number of services in the first step (Select Candidate Node) and processes their allocation in parallel. However, for simplicity, the chart below presents the allocation negotiation of one service only.

SAN is a five-stage process, involving a single source node (Node Agent S), one of the system BAs and several of other nodes in the system (Node Agent A, Node Agent B, Node Agent C) as shown in Figure 6.

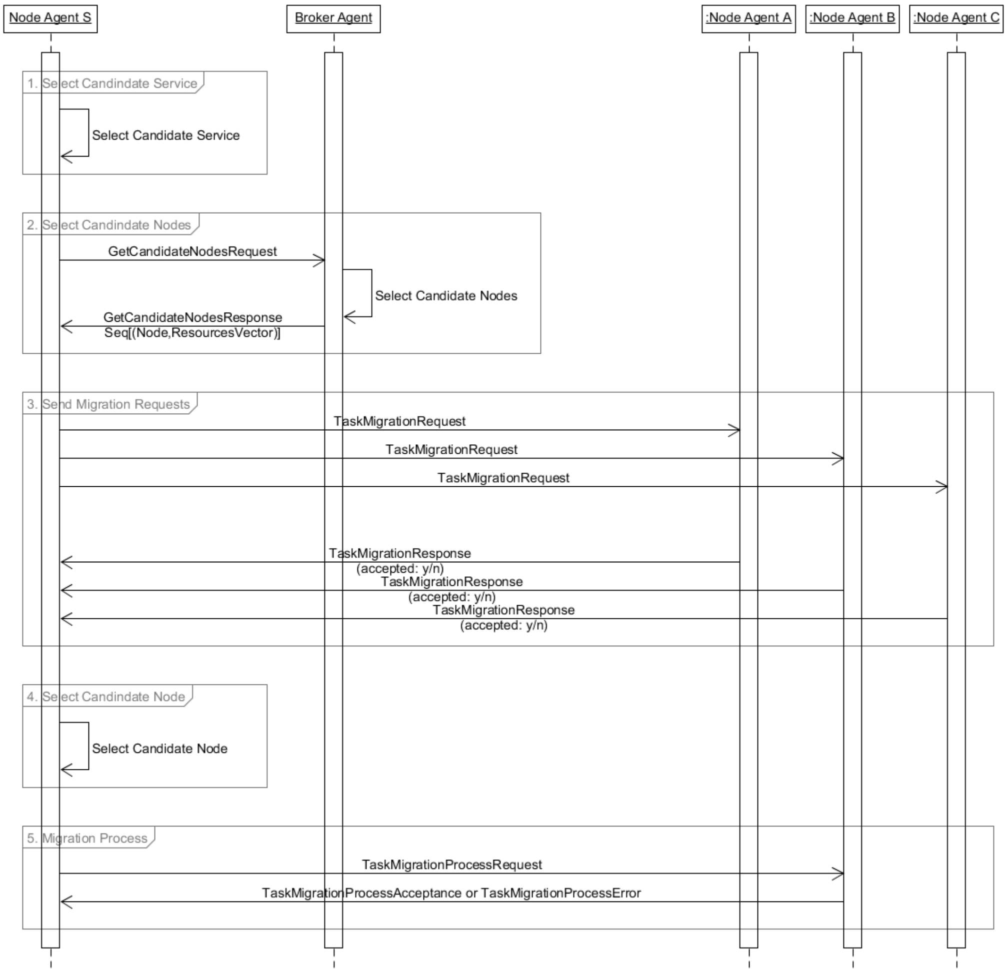


Figure 6: Service Allocation Negotiation

When migrating-out a given service, NAS at first sends a GetCandidateNodesRequest to BA to get with a set of candidate nodes where the task cab potentially be migrated to. BA scores all of its cached nodes and sends back the top fifteen to NAS. Additionally, in order to help to avoid collisions, BA does not directly select only top candidate nodes, but instead selects them randomly from a node pool, where candidate node score is a weight, wherein higher scored nodes are selected more frequently. This design helps to avoid a situation where an identical subset of candidate nodes is repeatedly selected for a number of tasks with the same resource requirements.

Upon receiving this list, NA sends service migration requests to all of those candidate nodes, and waits for a given time (in this case for 20 seconds) for all replies. After this time, NA evaluates all accepted service migration responses and orders them in relevance order (nodes with the highest score first) and then attempts to migrate a service to a top candidate node. If candidate node returns an error, the source NA will pick the next candidate node and attempt to migrate a service there.

At each of those stages, the candidate NA might reject task migration or return an error, e.g. when task migration is no longer possible because the current node's resource utilisation levels have increased or because the node attributes no longer match the service's constraints. Depending on a system utilisation level, such collisions might be more or less frequent. However, they are resolved at node-to-node communication level and do not impact the system performance as a whole.

In a situation where there are insufficient candidate nodes available due to the lack of free resource levels, the BA will return candidate nodes with the 'forced migration' flag set to true. The forced migrations feature is detailed in Section 6.6 below.

6.1. STEP 1: SELECT CANDIDATE SERVICES

Select Candidate Services (SCS) routine is executed when the NA detects that the currently existing services are overloading its node. This step is processed on the node wholly locally. The purpose of this routine is to select the service (or set of services) that the N will attempt to migrate out and become stable (i.e. non-overloaded) during that process. All services currently running on this node are evaluated, taking into consideration various aspects, namely:

- The cost of running a service on this particular node. NA will aim to have the highest node score for its own node. If removing this particular service will cause its score (calculated by Score Node Utilisation Function) to be higher, then this service is more likely to be selected.
- The cost of migration of a service – Virtual Machine migrations cause disruptions on the Cloud system. In this research, cost is measured as the additional network traffic required to migrate the running VM instance to an alternative node. This cost has been experimentally estimated to be generally exponentially correlated to allocated memory for a given service (Sliwko and Getov, 2017). It should be also noted that different types of applications have a more-or-less step increase in total network traffic required to migrate VM.
- The likeness to find an alternative node - the majority of services do not have major constraints and can be executed on a wide range of nodes. However, there are a small number of services with very restrictive constraints that significantly limit the number of nodes that the service can be executed on. If such a service can only be executed locally (i.e. the node has enough total resources capacity and service constraints are matched), then NA will never migrate out those services.
- Any service which cannot be executed on a local node is compulsory selected as a candidate service. This scenario could occur if the service constraints or node attributes were updated.

NA first computes a list of compulsory candidate services, i.e. services that can no longer be executed on this node. Following this, if the remaining services are still overloading the node, it will select a subset of services to be migrated out.

The candidate services selection algorithm tries to minimise the total migration cost of selected services, and also to achieve the highest AS for a node, under the assumption that the selected subset of candidate services is successfully migrated to the alternative node. In order to achieve this, the algorithm defines the Fitness Function as coded inside SCS:

$$Fitness\ Function = \frac{Node\ allocation\ score}{Total\ migration\ cost} \qquad (1)$$

For the above in a NP-Hard problem with a substantial search space, e.g. 20 services on a node, the search space size is over one million combinations. Therefore, the use of meta-heuristic algorithms is justified. In previous research (Sliwko and Getov, 2016), a variant of TS with a limited number of steps and repeated runs, has been successfully applied to solve a similar class of problems. It was possible to copy this previous implementation with only a small alteration. The TS algorithm has the following properties:

- It has a small memory imprint since only the list of visited solutions is maintained thorough execution;
- It can be easily parallelised as a variant which is restarted multiple times;
- It is very controllable through setting up a limited number of steps and number of runs;
- It is stoppable, and the best found result can be retrieved immediately;
- It generally returns good results.

```
20:39:13.342 NodeAgentActor (node=5782509) INFO
SAMPLE:
 Selected overloading tasks for node [5782509]
 Node total resources = [0.5000000000,0.2493000000]
 Node used resources (all tasks) = [0.1686352000,0.1673750000]
 Node used prod resources (all tasks) = [0.5837480000,0.3572020000]
 All tasks (* Selected):
  Task [6250257820-2156] Priority=2  Required resources=[0.0625000000,0.1592000000] Used
resources=[0.0062500000,0.0159200000] Migration cost = 2708.33 [MB]
  Task [2902878580-2584] (PROD) Priority=11  Required resources=[0.0062480000,0.0014570000] Used
resources=[0.0093380000,0.0269500000] Migration cost = 541.55 [MB]
  Task [331416465-22] (PROD) Priority=9  Required resources=[0.1875000000,0.0894800000] Used
resources=[0.0550500000,0.0206900000] Migration cost = 438.98 [MB]
  Task [4857080139-282] (PROD) Priority=2  Required resources=[0.0625000000,0.0477300000] Used
resources=[0.0002184000,0.0310400000] Migration cost = 608.56 [MB]
 *Task [4857081234-132] (PROD) Priority=2  Required resources=[0.0625000000,0.0477300000] Used
resources=[0.0002184000,0.0135700000] Migration cost = 322.33 [MB]
 *Task [4857082814-371] (PROD) Priority=2  Required resources=[0.0625000000,0.0477300000] Used
resources=[0.0002174000,0.0135700000] Migration cost = 322.33 [MB]
  Task [5351216030-28] Priority=1  Required resources=[0.0312500000,0.0159000000] Used
resources=[0.0005217000,0.0019320000] Migration cost = 131.65 [MB]
  Task [5852047245-0] (PROD) Priority=9  Required resources=[0.0343600000,0.0104200000] Used
resources=[0.0025140000,0.0019300000] Migration cost = 131.62 [MB]
  Task [6114773114-1295] Priority=0  Required resources=[0.0125000000,0.0077670000] Used
resources=[0.0002022000,0.0019190000] Migration cost = 131.44 [MB]
  Task [6217659576-9] (PROD) Priority=0  Required resources=[0.0125000000,0.0039750000] Used
resources=[0.0201100000,0.0057370000] Migration cost = 194.00 [MB]
  Task [6221861800-10453] Priority=0  Required resources=[0.0125000000,0.0159000000] Used
resources=[0.0002031000,0.0019280000] Migration cost = 131.59 [MB]
  Task [6221861800-3053] Priority=0  Required resources=[0.0125000000,0.0159000000] Used
resources=[0.0002069000,0.0019300000] Migration cost = 131.62 [MB]
  Task [6221861800-4253] Priority=0  Required resources=[0.1562000000,0.1399000000] Used
resources=[0.0002041000,0.0019300000] Migration cost = 131.62 [MB]
  Task [6221861800-9435] Priority=0  Required resources=[0.0125000000,0.0159000000] Used
resources=[0.0002031000,0.0019320000] Migration cost = 131.65 [MB]
 *Task [6225099547-1718] (PROD) Priority=2  Required resources=[0.0306400000,0.0131700000] Used
resources=[0.0002079000,0.0038800000] Migration cost = 163.57 [MB]
  Task [6238340468-1853] (PROD) Priority=0  Required resources=[0.0625000000,0.0396100000] Used
resources=[0.0175800000,0.0116000000] Migration cost = 290.05 [MB]
  Task [6238842108-115] (PROD) Priority=9  Required resources=[0.0312500000,0.0279500000] Used
resources=[0.0319800000,0.0057370000] Migration cost = 194.00 [MB]
  Task [6238844615-173] (PROD) Priority=9  Required resources=[0.0312500000,0.0279500000] Used
resources=[0.0234100000,0.0051800000] Migration cost = 184.87 [MB]
 Node used resources (remaining tasks) = [0.1679915000,0.1363550000]
 Node used prod resources (remaining tasks) = [0.4281080000,0.2485720000]
 Total migration cost (selected tasks) = 808.2316800065191 [MB]
```

Slightly better results can be achieved with SGA-TS, albeit at the cost of higher memory imprint, more complex implementation and decreased controllability. It was found that multiple restarts (herein a twenty-five-rerun limit) with a shallow limit of steps (herein five) yields very good results, with only about 2-7% of solutions in the whole search space being examined in each invocation. Additionally, instead of restarting the algorithm an arbitrary number of times, a stop condition for this algorithm has been implemented when the best-found solution has not been improved in a certain number of the last steps (herein six). A sample log entry is shown above, wherein the subset of candidate services is being computed.

The selected services are then added to any compulsory candidate services, and NA will attempt to migrate out this set in the next step. The potential reduction of used resources is an effect of removing a subset of tasks (marked with *) from this node. It can also be seen that CPU reserved for production tasks is potentially reduced from 0.583748 to 0.428108 which is ca.86% utilisation of total 0.5 CPU available on this node, and memory reserved for production tasks is potentially reduced from 0.357202 to 0.248572 which is ca.99% utilisation of total 0.2493 memory available on this node. The total migration cost for this set of migrations is ca.808 MB.

6.2. Step 2: Select Candidate Nodes

```
21:29:25.454 NodeAgentActor (node=351622677) INFO
SAMPLE:
 Candidate nodes recommendations for migration-out of task: Task [4765556460-299] Required
resources=[0.0558500000,0.1545000000] Migration cost = 2631.33 [MB]
 Source node: Node [351622677] [0.5000000000,0.4995000000]:
  CandidateNodeRecommendation[time=1489526965342,taskId=4765556460-
299,taskRequiredResources=[0.0558500000,0.1545000000],nodeId=1436310537,nodeAvailableResources=[0.1143434
000,0.1666880000],fitnessValue=3.9457163172031864,forceMigration=false]
  CandidateNodeRecommendation[time=1489526965342,taskId=4765556460-
299,taskRequiredResources=[0.0558500000,0.1545000000],nodeId=1301878,nodeAvailableResources=[0.0705801800
,0.2096340000],fitnessValue=3.9058075263646206,forceMigration=false]
  CandidateNodeRecommendation[time=1489526965341,taskId=4765556460-
299,taskRequiredResources=[0.0558500000,0.1545000000],nodeId=1429191803,nodeAvailableResources=[0.1065830
000,0.2709750000],fitnessValue=2.8282483001788625,forceMigration=false]
  CandidateNodeRecommendation[time=1489526965341,taskId=4765556460-
299,taskRequiredResources=[0.0558500000,0.1545000000],nodeId=372626592,nodeAvailableResources=[0.19000626
00,0.1980110000],fitnessValue=2.7914548077392425,forceMigration=false]
  CandidateNodeRecommendation[time=1489526965310,taskId=4765556460-
299,taskRequiredResources=[0.0558500000,0.1545000000],nodeId=5655258253,nodeAvailableResources=[0.1694482
000,0.2432700000],fitnessValue=2.583337693875162,forceMigration=false]
  CandidateNodeRecommendation[time=1489526965311,taskId=4765556460-
299,taskRequiredResources=[0.0558500000,0.1545000000],nodeId=4027200724,nodeAvailableResources=[0.2221300
000,0.2365560000],fitnessValue=2.236828139428944,forceMigration=false]
  CandidateNodeRecommendation[time=1489526965308,taskId=4765556460-
299,taskRequiredResources=[0.0558500000,0.1545000000],nodeId=610755652,nodeAvailableResources=[0.30181782
40,0.1708320000],fitnessValue=2.039819689997894,forceMigration=false]
  CandidateNodeRecommendation[time=1489526965342,taskId=4765556460-
299,taskRequiredResources=[0.0558500000,0.1545000000],nodeId=6565530,nodeAvailableResources=[0.2909429000
,0.2023160000],fitnessValue=1.9539382168986787,forceMigration=false]
  CandidateNodeRecommendation[time=1489526965309,taskId=4765556460-
299,taskRequiredResources=[0.0558500000,0.1545000000],nodeId=4478563006,nodeAvailableResources=[0.2623356
000,0.2849700000],fitnessValue=1.7019616825022668,forceMigration=false]
  CandidateNodeRecommendation[time=1489526965307,taskId=4765556460-
299,taskRequiredResources=[0.0558500000,0.1545000000],nodeId=938210976,nodeAvailableResources=[0.16753535
00,0.3809450000],fitnessValue=1.6134377154605029,forceMigration=false]
  CandidateNodeRecommendation[time=1489526965340,taskId=4765556460-
299,taskRequiredResources=[0.0558500000,0.1545000000],nodeId=8055084,nodeAvailableResources=[0.3391320000
,0.2168510000],fitnessValue=1.5794143938664662,forceMigration=false]
  CandidateNodeRecommendation[time=1489526965343,taskId=4765556460-
299,taskRequiredResources=[0.0558500000,0.1545000000],nodeId=351638108,nodeAvailableResources=[0.34576310
00,0.2926680000],fitnessValue=1.2671427736705632,forceMigration=false]
  CandidateNodeRecommendation[time=1489526965308,taskId=4765556460-
299,taskRequiredResources=[0.0558500000,0.1545000000],nodeId=778700448,nodeAvailableResources=[0.33851370
00,0.3026960000],fitnessValue=1.2639028696871983,forceMigration=false]
  CandidateNodeRecommendation[time=1489526965309,taskId=4765556460-
299,taskRequiredResources=[0.0558500000,0.1545000000],nodeId=906517,nodeAvailableResources=[0.3179763100,
0.3554060000],fitnessValue=1.1607777796685308,forceMigration=true]
  CandidateNodeRecommendation[time=1489526965342,taskId=4765556460-
299,taskRequiredResources=[0.0558500000,0.1545000000],nodeId=2097421225,nodeAvailableResources=[0.3466404
000,0.3375800000],fitnessValue=1.117142943496929,forceMigration=true]
```

After selecting candidate nodes, NA sends a GetCandidateNodes request to BA. A part of this request, service information data, such as currently used resources and constraints, are sent. BA also itself caches a list of all nodes in system with their available resources and attributes. Based on this information, BA prepares a list of alternative candidate nodes for a service in request. The main objective of this process is to find alternative nodes which have the potentially highest node

AS, under the assumption that the service will be migrated to a scored node. The size of this list is limited to an arbitrary value to avoid network congestion when NA will send actual migration requests query in the next step. In this experiment, it is set to fifteen candidate nodes returned in each response.

This step is the most computing intensive of all steps and is a potential bottleneck for negotiating logic processing. BA needs to examine all system nodes, check their availability for a given service and score them accordingly. The request processing is self-contained and highly concurrent, meaning that the node scoring can be run in parallel and the final selection of top candidate nodes is run in sequence. Originally, this code was extensively optimised, and designed BA to be able to run in a multi-instance mode if needed and to handle heavy usage. However, in experiments, the quoting mechanism proved to be very lightweight and demand not that high, meaning that a single BA was sufficient to handle 12.5k nodes in the system. A sample log when such a list is computed and returned to a NA is shown above.

Here, BA returned top candidate nodes for a given service ordered by their suitability score, i.e. fitness value. It should be noted that within the node recommendation there is additional information, such as node available resources and calculated fitness value. It is not necessary to return this extra information, but it was found to be very useful for logging and sampling purposes.

6.3. Step 3: Send Migration Requests

Upon receiving the candidate nodes list, NA will send TaskMigration request to each of the returned nodes with the exception of forced migration candidates. Forced migration candidates will be always added to list of accepted candidate nodes in the next step. Each NA analyses its own node availability for a given service, i.e. both the available resources and the node's attributes, and then responds with TaskMigrationAcceptanceResponse or TaskMigrationRejectionResponse. It should be noted that acceptance only implies the readiness of accepting a service, and NA is not allocating any resources yet. Acceptance response contains this node's current resources usage levels which are used in the next step to rescore this node, since the data from BA are less recent.

6.4. Step 4: Select Candidate Node

NA waits for defined time, or until all candidate nodes have responded by either the acceptance or rejection of a migrated service and computes a list of candidate nodes that accepted this service. NA evaluates each of the accepting candidate nodes using the allocation scoring function, with the assumption that the service will be reallocated to a scored node. From this pool, a single candidate node is then randomly selected. The selection is weighted with candidate node scores which helps to avoid conflicts when many service migrations compete for the same node. It should also be noted that forced migration candidate nodes are added to this list, but will be selected only in the last place, after all other alternative migrations attempts fail. A sample log entry is presented below.

When a single candidate node is selected, NA sends TaskMigrationProcessRequest to initiate a service migration process itself. NA stores received candidate node recommendations in its memory, in case the service migration fails and the next candidate node has to be selected. Once the service is removed from a node, meaning it is reallocated, has finished its execution, is killed or crashes, all its candidate node recommendations are deleted. Additionally, candidate node recommendations expire after an arbitrary defined time, in this case four minutes. This mechanism exists in order to remove recommendations with out-dated node data.

```
20:40:05.002 NodeAgentActor (node=2X318388085) INFO
SAMPLE:
 Accepted recommendations for migration-out of task: Task [2X5544436183-107] (PROD) Priority=9  Required
 resources=[0.0156200000,0.0590200000] Used resources=[0.0015620000,0.0059020000] Migration cost = 1066.98
[MB]
 Source node: Node [2X318388085] [0.5000000000,0.2493000000]
 All non-expired recommendations (* selected):
  CandidateNodeRecommendation[age=1,time=1501616404981,taskId=2X5544436183-107,taskUsedResources=
  [0.001562000,0.0059020000],nodeId=2X1342749,nodeAvailableResources=[0.1891418000,0.1775330000],
  fitnessValue=2.466743799605,forceMigration=false]
  CandidateNodeRecommendation[age=1,time=1501616404981,taskId=2X5544436183-107,taskUsedResources=
  [0.0015620000,0.0059020000],nodeId=7X3820867554,nodeAvailableResources=[0.3076184300,0.1759794000],
  fitnessValue=1.643246105180,forceMigration=false]
  CandidateNodeRecommendation[age=1,time=1501616404981,taskId=2X5544436183-107,taskUsedResources=
  [0.001562000,0.0059020000],nodeId=2X1672921,nodeAvailableResources=[0.3748082000,0.1112931000],
  fitnessValue=1.474454140561,forceMigration=false]
  CandidateNodeRecommendation[age=1,time=1501616404981,taskId=2X5544436183-107,taskUsedResources=
  [0.0015620000,0.0059020000],nodeId=705685,nodeAvailableResources=[0.4126200000,0.1051633000],
  fitnessValue=1.261414394584,forceMigration=false]
  CandidateNodeRecommendation[age=1,time=1501616404981,taskId=2X5544436183-107,taskUsedResources=
  [0.0015620000,0.0059020000],nodeId=5X3625124643,nodeAvailableResources=[0.4090680000,0.1390167000],
  fitnessValue=1.192644653836,forceMigration=false]
  CandidateNodeRecommendation[age=1,time=1501616404982,taskId=2X5544436183-107,taskUsedResources=
  [0.0015620000,0.0059020000],nodeId=3X1436307767,nodeAvailableResources=[0.3632657800,0.2724450000],
  fitnessValue=1.035791353731,forceMigration=false]
  CandidateNodeRecommendation[age=1,time=1501616404982,taskId=2X5544436183-107,taskUsedResources=
  [0.0015620000,0.0059020000],nodeId=6X347828744,nodeAvailableResources=[0.4451295000,0.1572113000],
  fitnessValue=0.973313266907,forceMigration=false]
  CandidateNodeRecommendation[age=1,time=1501616404981,taskId=2X5544436183-107,taskUsedResources=
  [0.001562000,0.0059020000],nodeId=1X1340913,nodeAvailableResources=[0.3966700000,0.2720053000],
  fitnessValue=0.918316167972,forceMigration=false]
  CandidateNodeRecommendation[age=1,time=1501616404981,taskId=2X5544436183-107,taskUsedResources=
  [0.001562000,0.0059020000],nodeId=4X1094553,nodeAvailableResources=[0.4399146100,0.2276939000],
  fitnessValue=0.857971414808,forceMigration=false]
  CandidateNodeRecommendation[age=1,time=1501616404982,taskId=2X5544436183-107,taskUsedResources=
  [0.001562000,0.0059020000],nodeId=5X602982,nodeAvailableResources=[0.4396702700,0.2803267000],
  fitnessValue=0.761758672976,forceMigration=false]
 *CandidateNodeRecommendation[age=1,time=1501616404982,taskId=2X5544436183-107,taskUsedResources=
  [0.0015620000,0.0059020000],nodeId=907792,nodeAvailableResources=[0.4224480000,0.3210886000],
  fitnessValue=0.736151076814,forceMigration=false]
  CandidateNodeRecommendation[age=1,time=1501616404982,taskId=2X5544436183-107,taskUsedResources=
  [0.0015620000,0.0059020000],nodeId=5X766891169,nodeAvailableResources=[0.4173642000,0.3477250000],
  fitnessValue=0.698808489913,forceMigration=false]
  CandidateNodeRecommendation[age=1,time=1501616404981,taskId=2X5544436183-107,taskUsedResources=
  [0.001562000,0.0059020000],nodeId=6X1274843,nodeAvailableResources=[0.4393896000,0.3416696000],
  fitnessValue=0.656398863515,forceMigration=false]
  CandidateNodeRecommendation[age=1,time=1501616359408,taskId=2X5544436183-107,taskUsedResources=
  [0.001562000,0.0059020000],nodeId=6X2850436,nodeAvailableResources=[0.3813088100,0.0624228000],
  fitnessValue=1.596120533603,forceMigration=true]
  CandidateNodeRecommendation[age=1,time=1501616359409,taskId=2X5544436183-107,taskUsedResources=
  [0.0015620000,0.0059020000],nodeId=1X288943706,nodeAvailableResources=[0.4105824800,0.1324943000],
  fitnessValue=1.201407811061,forceMigration=true]
```

6.5. Step 5: Migration Process

When NA receives TaskMigrationProcessRequest, it performs a final suitability check, wherein both node's available resources and service constraints are validated. If the forced-migration flag is set, NA ignores the existing tasks and validates the required resources against total node resources. Occasionally, the target NA can reject service migration process or migration fails. In such a scenario the algorithm returns to Step 2 and selects an alternative candidate node. In practice, this happens for 6-8% of all service migration attempts, the majority being the result of service migration collisions where two services are being migrated to the same node. There have been no observations of an increase in collisions when the larger Cloud system is simulated.

6.6. Forced Migration

In rare circumstances, approximately 10-15 tasks out of 12.5k tasks, arise constraints which restrict the execution of a task to a very limited number of nodes. Since Google Cluster Project data have been obfuscated, the exact details of those constraints are unavailable. As such, there is a scenario in which NA wants to migrate out a given task but is unable to find an alternative node because any suitable nodes have already been allocated with other tasks and had the majority of their resources utilised. In such a scenario, BA returns candidate node recommendations with a forced-migration flag set. In its response, the BA can also mix non-forced migrations and forced migrations. In a worst case scenario, all returned recommendations would be forced, but this approach ensures there is always an acceptable node to run a given task on. This prevents a starvation of the task resources, where the task is never executed.

A forced migration flag signals that a node is capable of executing service but that its current utilisation levels do not allow it to allocate additional services as this will cause the node to be overloaded. Forced migration forces the node to accept the service migration request while skipping the available resources check; however, service constraints are still validated (including the check if node's total resources are sufficient to run the service). This design helps to avoid a situation where a service has very limiting constraints and only a few nodes in the system can execute it. If those nodes have no available resources then it will not be possible to allocate a service to them, and therefore services will not run. As such, the nodes are forced to accept this service which then triggers NA's AI to migrate out some of its existing tasks to alternative nodes.

7. EXPERIMENTAL RESULTS

The previously developed AGOCS framework is used as the base of the experimental simulation. AGOCS simulates workload from a project using month-long workload traces. AGOCS is a very detailed simulator which provides a multiple of parameters and logical constraints for simulated jobs. The scope of the available variables is very broad, including memory page cache hit and instructions per CPU cycle, but in this project the focus and simulation were based on the following:

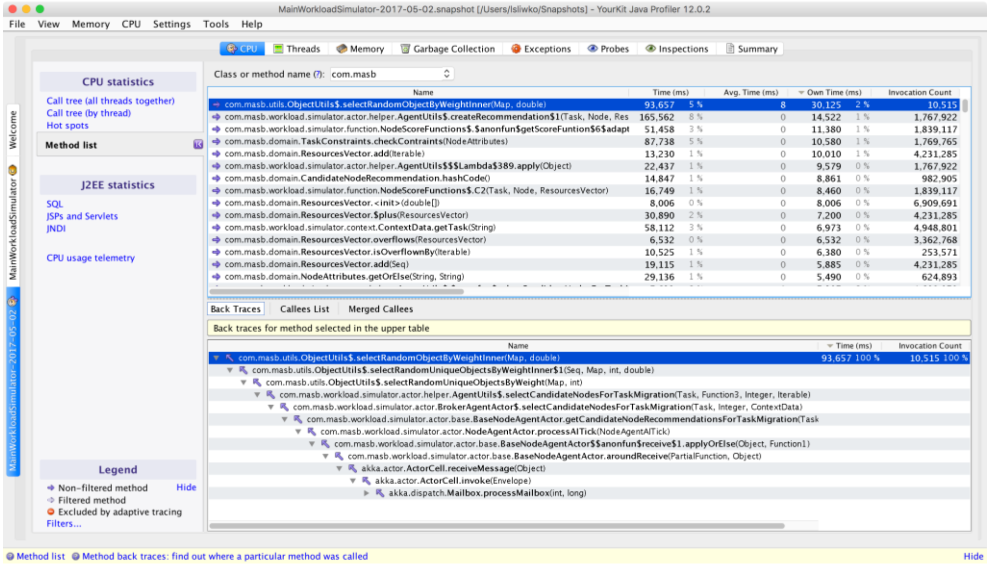
- Requested (by user) and realistic (monitored) resources' utilisation levels for memory and CPU;
- Detailed timing of incoming tasks and any changes in available nodes;
- Nodes attributes and attributes' constraints defined for tasks.

This level of detail comes at the price of extensive computing power requirements. While dry simulation itself can run on a typical desktop machine in ca.9 hours, adding layers of scheduling logic, agents' states and inter-system communication requires a significant increase in processing time. In order to realistically and correctly simulate scheduling processes on a Cloud system, the Westminster University HPC Cluster was used.

7.1. TESTING ENVIRONMENT

The MASB prototype was initially developed on a personal desktop, but as the size and level of detail of the simulations grew, it was necessary to move to a Westminster University HPC Cluster environment where more computing power was available. All the experiments were executed on the Westminster University HPC Cluster.

While this cluster offered a sizable array of GPUs, the simulations did not take advantage of that computing power, and instead all processing took place on CPUs. Although it would have been possible to achieve higher throughput when using GPU with frameworks such as ScalaCL or Rootbeer, JVM does not natively support GPU processing. This meant that it was preferred to have as few external dependencies as possible, since they make maintaining the project more time-consuming. Interestingly, Google's BorgMaster process which manages a single cell in the production environment for one computing cell, uses 10–14 CPU cores and up to 50GB of memory. The statistics presented are valid for an intensely utilised computing cell, for example one which completes more than 10k tasks per minute on average (Verma et al., 2015).


MainWorkloadSimulator-2017-05-02.snapshot [/Users/lsliwko/Snapshots] - YourKit Java Profiler 12.0.2
File View Memory CPU Settings Tools Help
CPU Threads Memory Garbage Collection Exceptions Probes Inspections Summary
Welcome
MainWorkloadSimulator
MainWorkloadSimulator-2017-05-02
CPU statistics
Call tree (all threads together)
Call tree (by thread)
Hot spots
Method list
J2EE statistics
SQL
JSPs and Servlets
JNDI
CPU usage telemetry
Legend
Non-filtered method
Hide
Filtered method
Excluded by adaptive tracing
Filters...
Back Traces
Callees List
Merged Callees
Hide

Class or method name (?): com.masb

Name	Time (ms)	Avg. Time (ms)	Own Time (ms)	Invocation Count
com.masb.utils.ObjectUtils$.selectRandomObjectByWeightInner(Map, double)	93,657 5 %	8	30,125 2 %	10,515
com.masb.workload.simulator.actor.helper.AgentUtils$.createRecommendation$1(Task, Node, Res	165,562 8 %	0	14,522 1 %	1,767,922
com.masb.workload.simulator.function.NodeScoreFunctions$.$anonfun$getScoreFuntion$6$adapt	51,458 3 %	0	11,380 1 %	1,839,117
com.masb.domain.TaskConstraints.checkContraints(NodeAttributes)	87,738 5 %	0	10,580 1 %	1,769,765
com.masb.domain.ResourcesVector.add(Iterable)	13,230 1 %	0	10,010 1 %	4,231,285
com.masb.workload.simulator.actor.helper.AgentUtils$$$Lambda$389.apply(Object)	22,437 1 %	0	9,579 0 %	1,767,922
com.masb.domain.CandidateNodeRecommendation.hashCode()	14,847 1 %	0	8,861 0 %	982,905
com.masb.workload.simulator.function.NodeScoreFunctions$.C2(Task, Node, ResourcesVector)	16,749 1 %	0	8,460 0 %	1,839,117
com.masb.domain.ResourcesVector.<init>(double[])	8,006 0 %	0	8,006 0 %	6,909,691
com.masb.domain.ResourcesVector.$plus(ResourcesVector)	30,890 2 %	0	7,200 0 %	4,231,285
com.masb.workload.simulator.context.ContextData.getTask(String)	58,112 3 %	0	6,973 0 %	4,948,801
com.masb.domain.ResourcesVector.overflows(ResourcesVector)	6,532 0 %	0	6,532 0 %	3,362,768
com.masb.domain.ResourcesVector.isOverflownBy(Iterable)	10,525 1 %	0	6,380 0 %	253,571
com.masb.domain.ResourcesVector.add(Seq)	19,115 1 %	0	5,885 0 %	4,231,285
com.masb.domain.NodeAttributes.getOrElse(String, String)	29,136 1 %	0	5,490 0 %	624,893

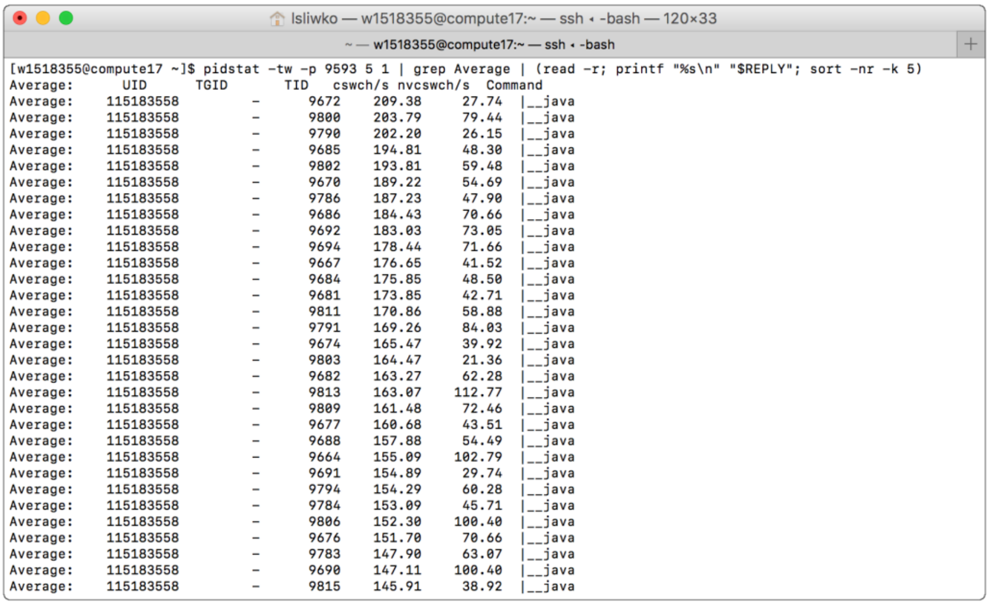
Back traces for method selected in the upper table

```
Name                                                                                          Time (ms)    Invocation Count
com.masb.utils.ObjectUtils$.selectRandomObjectByWeightInner(Map, double)                      93,657 100 %  10,515 100 %
  com.masb.utils.ObjectUtils$.selectRandomUniqueObjectsByWeightInner$1(Seq, Map, int, double)
    com.masb.utils.ObjectUtils$.selectRandomUniqueObjectsByWeight(Map, int)
      com.masb.workload.simulator.actor.helper.AgentUtils$.selectCandidateNodesForTaskMigration(Task, Function3, Integer, Iterable)
        com.masb.workload.simulator.actor.BrokerAgentActor$.selectCandidateNodesForTaskMigration(Task, Integer, ContextData)
          com.masb.workload.simulator.actor.base.BaseNodeAgentActor.getCandidateNodeRecommendationsForTaskMigration(Task
            com.masb.workload.simulator.actor.NodeAgentActor.processAITick(NodeAgentAITick)
              com.masb.workload.simulator.actor.base.BaseNodeAgentActor$$anonfun$receive$1.applyOrElse(Object, Function1)
                com.masb.workload.simulator.actor.base.BaseNodeAgentActor.aroundReceive(PartialFunction, Object)
                  akka.actor.ActorCell.receiveMessage(Object)
                    akka.actor.ActorCell.invoke(Envelope)
                      akka.dispatch.Mailbox.processMailbox(int, long)
```

Method list Method back traces: find out where a particular method was called

Figure 7: YourKit Java Profiler exercise

In experiments, MASB allocated all available 40 CPU cores (20 cores + HT siblings) and used them continuously 60% to 80%. The MASB process allocated ca.7GB of memory. It is difficult to measure exactly how much computing power was spent on supporting activities such as simulating messaging interactions between agents, i.e. enqueuing and dequeuing messages to and from Akka actors. However, after tuning exercises of the default configuration, the Akka Actors framework proved to be quite resilient. It is estimated that the framework's processing did not take up more than 10-15% of the total CPU time, with the relatively lightweight AGOCS simulator framework consuming about 15-25% of all CPU time. All the profiling and the above estimation were completed with help of YourKit Java Profiler 12.0.2 tool. Figure 7 presents a sample screenshot from the code profiling exercises.


lsliwko — w1518355@compute17:~ — ssh ◂ -bash — 120×33
~ — w1518355@compute17:~ — ssh ◂ -bash

```
[w1518355@compute17 ~]$ pidstat -tw -p 9593 5 1 | grep Average | (read -r; printf "%s\n" "$REPLY"; sort -nr -k 5)
Average:      UID      TGID       TID   cswch/s nvcswch/s  Command
Average:    115183558     -      9672    209.38     27.74  |__java
Average:    115183558     -      9800    203.79     79.44  |__java
Average:    115183558     -      9790    202.20     26.15  |__java
Average:    115183558     -      9685    194.81     48.30  |__java
Average:    115183558     -      9802    193.81     59.48  |__java
Average:    115183558     -      9670    189.22     54.69  |__java
Average:    115183558     -      9786    187.23     47.90  |__java
Average:    115183558     -      9686    184.43     70.66  |__java
Average:    115183558     -      9692    183.03     73.05  |__java
Average:    115183558     -      9694    178.44     71.66  |__java
Average:    115183558     -      9667    176.65     41.52  |__java
Average:    115183558     -      9684    175.85     48.50  |__java
Average:    115183558     -      9681    173.85     42.71  |__java
Average:    115183558     -      9811    170.86     58.88  |__java
Average:    115183558     -      9791    169.26     84.03  |__java
Average:    115183558     -      9674    165.47     39.92  |__java
Average:    115183558     -      9803    164.47     21.36  |__java
Average:    115183558     -      9682    163.27     62.28  |__java
Average:    115183558     -      9813    163.07    112.77  |__java
Average:    115183558     -      9809    161.48     72.46  |__java
Average:    115183558     -      9677    160.68     43.51  |__java
Average:    115183558     -      9688    157.88     54.49  |__java
Average:    115183558     -      9664    155.09    102.79  |__java
Average:    115183558     -      9691    154.89     29.74  |__java
Average:    115183558     -      9794    154.29     60.28  |__java
Average:    115183558     -      9784    153.09     45.71  |__java
Average:    115183558     -      9806    152.30    100.40  |__java
Average:    115183558     -      9676    151.70     70.66  |__java
Average:    115183558     -      9783    147.90     63.07  |__java
Average:    115183558     -      9690    147.11    100.40  |__java
Average:    115183558     -      9815    145.91     38.92  |__java
```

Figure 8: Monitoring context switches via pidstat

YourKit Java Profiler was an excellent tool which helped to optimise the code execution time. However, in a truly multi-core environment, a different approach was required which was focused on minimising context switches frequency and average CPU idle time across all available cores. Once the MASB framework was moved into the Cluster environment, the 'pidstat' command tool was used to gather statistics and then refactor and fine-tune framework to achieve better parallelism. Figure 8 presents a sample from ‘pidstat’ output.

7.2. Testability

Building a framework which fully simulates the Google computing cell from GCD traces has been previously recognised as a challenging task, where there are many aspects to consider (Sharma et al., 2011; Abdul-Rahman, et al., 2014; Zhu et al., 2015). GCD traces contain details of nodes, including their resources, attributes and historical changes in their values. Traces also contain corresponding parameters for tasks, such as user-defined and actually-used resources, and attributes’ constraints. This has created a multi-dimensional domain with a range of relations which has resulted in complex error-prone implementation. In order to mitigate the risk of coding errors, especially during rapid iterations, a number of programming practices were used:

- A comprehensive test units suite was developed, along with prototype code. Test units were executed upon every build to catch errors before being deployed to production;
- A number of sanity checks were built into the runtime logic, such as checking whether the task’s constraints could be matched to any node’s attributes within the system and checking whether the total of all scheduled tasks’ resources exceeded the computing cell compatibilities;
- Recoverable logic flow was implemented for both NA and BA. In the case of various errors such as division by zero or null pointer exceptions, the error is logged but the agent continues to run. This feature was based on the supervisor strategy of the Akka actors framework;
- Keeping a separate error log file with the output of all warnings and errors was a considerable help in terms of resolving bugs.

The implementation of the above features gave high confidence in terms of realising a good quality and reasonably bug-free code. Additionally, the very comparable throughput to original GCD traces confirmed the correctness of implementation.

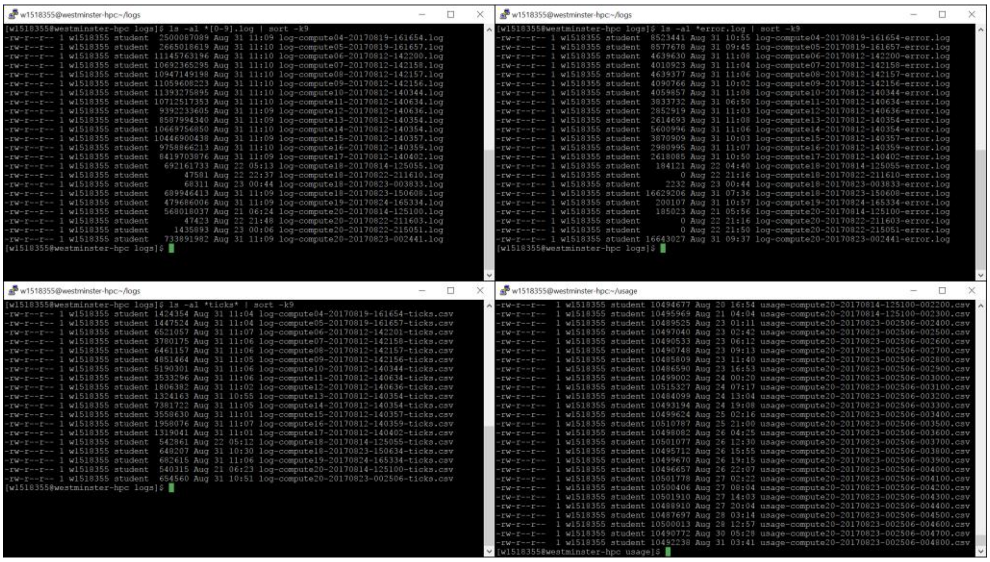
7.3. Platform Outputs

Adding detailed logging features to MASB has proved to be surprisingly difficult. Due to the highly parallel nature of the simulated Cloud environment, an enormous number of log messages were generated upon each simulation, making it difficult to analyse the behaviour of tested algorithms. In addition, writing and flushing log streams caused pauses in simulation. Switching to a Logback framework which was designed with a focus on concurrent writes, solved this problem, although it was necessary to split the data into separate log files in order to improve readability.

7.3.1. Logging

In order to fine-tune MASB, excessive logging routines were implemented. In the final version, all messages, counters and errors are logged to four types of log-files. These are presented in Figure 9.

- /logs/*.log files (top-left corner) – standard log outputs containing all logs messages and also samples;
- /logs/*-error.log (top-right corner) – errors and corrupted data exceptions are written to separate files to help with debugging and tuning;
- /logs/*-ticks.csv (low-left corner) – CSV files with periodically generated overall system stats, such as the number of idle and overloaded nodes, number of migration attempts, number of forced migration attempts, global resources-allocation ratio, and so on;
- /usage/*.csv (low-right corner) - detailed node usage stats and task allocations are written periodically to a file, that is, every 100 minutes of simulation time.


w1518355@westminster-hpc:~/logs
w1518355@westminster-hpc:~/logs
w1518355@westminster-hpc:~/logs
w1518355@westminster-hpc:~/usage

```
[w1518355@westminster-hpc logs]$ ls -al *[0-9].log | sort -k9
-rw-r--r-- 1 w1518355 student  2500087089 Aug 31 11:09 log-compute04-20170819-161654.log
-rw-r--r-- 1 w1518355 student  2665018619 Aug 31 11:10 log-compute05-20170819-161657.log
-rw-r--r-- 1 w1518355 student 11145763196 Aug 31 11:10 log-compute06-20170812-142200.log
-rw-r--r-- 1 w1518355 student 10692365295 Aug 31 11:10 log-compute07-20170812-142158.log
-rw-r--r-- 1 w1518355 student 10947149198 Aug 31 11:10 log-compute08-20170812-142157.log
-rw-r--r-- 1 w1518355 student 11059608223 Aug 31 11:10 log-compute09-20170812-142156.log
-rw-r--r-- 1 w1518355 student 11393275895 Aug 31 11:10 log-compute10-20170812-140344.log
-rw-r--r-- 1 w1518355 student 10712517353 Aug 31 11:10 log-compute11-20170812-140634.log
-rw-r--r-- 1 w1518355 student  9392233605 Aug 31 11:09 log-compute12-20170812-140636.log
-rw-r--r-- 1 w1518355 student  8587994340 Aug 31 11:09 log-compute13-20170812-140354.log
-rw-r--r-- 1 w1518355 student 10669756850 Aug 31 11:10 log-compute14-20170812-140354.log
-rw-r--r-- 1 w1518355 student 10446900438 Aug 31 11:09 log-compute15-20170812-140357.log
-rw-r--r-- 1 w1518355 student  9758866213 Aug 31 11:10 log-compute16-20170812-140359.log
-rw-r--r-- 1 w1518355 student  8419703876 Aug 31 11:09 log-compute17-20170812-140402.log
-rw-r--r-- 1 w1518355 student   692161733 Aug 22 05:13 log-compute18-20170814-125055.log
-rw-r--r-- 1 w1518355 student       47581 Aug 22 22:37 log-compute18-20170822-211610.log
-rw-r--r-- 1 w1518355 student       68311 Aug 23 00:44 log-compute18-20170823-003833.log
-rw-r--r-- 1 w1518355 student   689946413 Aug 31 11:09 log-compute18-20170823-150608.log
-rw-r--r-- 1 w1518355 student   479686006 Aug 31 11:09 log-compute19-20170824-165334.log
-rw-r--r-- 1 w1518355 student   568018037 Aug 21 06:24 log-compute20-20170814-125100.log
-rw-r--r-- 1 w1518355 student       47423 Aug 22 21:48 log-compute20-20170822-211603.log
-rw-r--r-- 1 w1518355 student     1435893 Aug 23 00:06 log-compute20-20170822-215051.log
-rw-r--r-- 1 w1518355 student   733891982 Aug 31 11:09 log-compute20-20170823-002441.log
[w1518355@westminster-hpc logs]$
```

```
[w1518355@westminster-hpc logs]$ ls -al *error.log | sort -k9
-rw-r--r-- 1 w1518355 student  8523441 Aug 31 10:55 log-compute04-20170819-161654-error.log
-rw-r--r-- 1 w1518355 student  8577678 Aug 31 09:45 log-compute05-20170819-161657-error.log
-rw-r--r-- 1 w1518355 student  4639630 Aug 31 11:08 log-compute06-20170812-142200-error.log
-rw-r--r-- 1 w1518355 student  4010923 Aug 31 11:04 log-compute07-20170812-142158-error.log
-rw-r--r-- 1 w1518355 student  4639377 Aug 31 11:06 log-compute08-20170812-142157-error.log
-rw-r--r-- 1 w1518355 student  4090766 Aug 31 10:02 log-compute09-20170812-142156-error.log
-rw-r--r-- 1 w1518355 student  4059857 Aug 31 11:08 log-compute10-20170812-140344-error.log
-rw-r--r-- 1 w1518355 student  3833732 Aug 31 06:50 log-compute11-20170812-140634-error.log
-rw-r--r-- 1 w1518355 student  2852919 Aug 31 11:03 log-compute12-20170812-140636-error.log
-rw-r--r-- 1 w1518355 student  2614693 Aug 31 11:08 log-compute13-20170812-140354-error.log
-rw-r--r-- 1 w1518355 student  5600996 Aug 31 11:06 log-compute14-20170812-140354-error.log
-rw-r--r-- 1 w1518355 student  3870909 Aug 31 10:03 log-compute15-20170812-140357-error.log
-rw-r--r-- 1 w1518355 student  2980995 Aug 31 11:07 log-compute16-20170812-140359-error.log
-rw-r--r-- 1 w1518355 student  2618085 Aug 31 10:50 log-compute17-20170812-140402-error.log
-rw-r--r-- 1 w1518355 student   184121 Aug 22 04:40 log-compute18-20170814-125055-error.log
-rw-r--r-- 1 w1518355 student        0 Aug 22 21:16 log-compute18-20170822-211610-error.log
-rw-r--r-- 1 w1518355 student     2232 Aug 23 00:44 log-compute18-20170823-003833-error.log
-rw-r--r-- 1 w1518355 student 16629206 Aug 31 07:36 log-compute18-20170823-150608-error.log
-rw-r--r-- 1 w1518355 student   200107 Aug 31 10:57 log-compute19-20170824-165334-error.log
-rw-r--r-- 1 w1518355 student   185023 Aug 21 05:56 log-compute20-20170814-125100-error.log
-rw-r--r-- 1 w1518355 student        0 Aug 22 21:16 log-compute20-20170822-211603-error.log
-rw-r--r-- 1 w1518355 student        0 Aug 22 21:50 log-compute20-20170822-215051-error.log
-rw-r--r-- 1 w1518355 student 16643027 Aug 31 09:37 log-compute20-20170823-002441-error.log
[w1518355@westminster-hpc logs]$
```

```
[w1518355@westminster-hpc logs]$ ls -al *ticks* | sort -k9
-rw-r--r-- 1 w1518355 student 1424354 Aug 31 11:04 log-compute04-20170819-161654-ticks.csv
-rw-r--r-- 1 w1518355 student 1447524 Aug 31 11:04 log-compute05-20170819-161657-ticks.csv
-rw-r--r-- 1 w1518355 student 6521057 Aug 31 11:07 log-compute06-20170812-142201-ticks.csv
-rw-r--r-- 1 w1518355 student 3780175 Aug 31 11:06 log-compute07-20170812-142158-ticks.csv
-rw-r--r-- 1 w1518355 student 6461157 Aug 31 11:06 log-compute08-20170812-142157-ticks.csv
-rw-r--r-- 1 w1518355 student 4851464 Aug 31 11:05 log-compute09-20170812-142156-ticks.csv
-rw-r--r-- 1 w1518355 student 5190301 Aug 31 11:06 log-compute10-20170812-140344-ticks.csv
-rw-r--r-- 1 w1518355 student 3533296 Aug 31 11:06 log-compute11-20170812-140634-ticks.csv
-rw-r--r-- 1 w1518355 student 1806382 Aug 31 11:02 log-compute12-20170812-140636-ticks.csv
-rw-r--r-- 1 w1518355 student 1324163 Aug 31 10:55 log-compute13-20170812-140354-ticks.csv
-rw-r--r-- 1 w1518355 student 7381722 Aug 31 11:05 log-compute14-20170812-140354-ticks.csv
-rw-r--r-- 1 w1518355 student 3558630 Aug 31 11:01 log-compute15-20170812-140357-ticks.csv
-rw-r--r-- 1 w1518355 student 1958076 Aug 31 11:07 log-compute16-20170812-140359-ticks.csv
-rw-r--r-- 1 w1518355 student 1319041 Aug 31 11:01 log-compute17-20170812-140402-ticks.csv
-rw-r--r-- 1 w1518355 student  542861 Aug 22 05:12 log-compute18-20170814-125055-ticks.csv
-rw-r--r-- 1 w1518355 student  648207 Aug 31 10:30 log-compute18-20170823-150634-ticks.csv
-rw-r--r-- 1 w1518355 student  682615 Aug 31 11:06 log-compute19-20170824-165334-ticks.csv
-rw-r--r-- 1 w1518355 student  540315 Aug 21 06:23 log-compute20-20170814-125100-ticks.csv
-rw-r--r-- 1 w1518355 student  654560 Aug 31 10:51 log-compute20-20170823-002506-ticks.csv
[w1518355@westminster-hpc logs]$
```

```
-rw-r--r--  1 w1518355 student 10494677 Aug 20 16:54 usage-compute20-20170814-125100-002200.csv
-rw-r--r--  1 w1518355 student 10495969 Aug 21 04:04 usage-compute20-20170814-125100-002300.csv
-rw-r--r--  1 w1518355 student 10489525 Aug 23 01:11 usage-compute20-20170823-002506-002400.csv
-rw-r--r--  1 w1518355 student 10497040 Aug 23 02:42 usage-compute20-20170823-002506-002500.csv
-rw-r--r--  1 w1518355 student 10490533 Aug 23 06:12 usage-compute20-20170823-002506-002600.csv
-rw-r--r--  1 w1518355 student 10490748 Aug 23 09:13 usage-compute20-20170823-002506-002700.csv
-rw-r--r--  1 w1518355 student 10485809 Aug 23 11:40 usage-compute20-20170823-002506-002800.csv
-rw-r--r--  1 w1518355 student 10486590 Aug 23 16:53 usage-compute20-20170823-002506-002900.csv
-rw-r--r--  1 w1518355 student 10499002 Aug 24 00:20 usage-compute20-20170823-002506-003000.csv
-rw-r--r--  1 w1518355 student 10515327 Aug 24 07:17 usage-compute20-20170823-002506-003100.csv
-rw-r--r--  1 w1518355 student 10484099 Aug 24 13:04 usage-compute20-20170823-002506-003200.csv
-rw-r--r--  1 w1518355 student 10493194 Aug 24 19:08 usage-compute20-20170823-002506-003300.csv
-rw-r--r--  1 w1518355 student 10499624 Aug 25 02:16 usage-compute20-20170823-002506-003400.csv
-rw-r--r--  1 w1518355 student 10510787 Aug 25 21:00 usage-compute20-20170823-002506-003500.csv
-rw-r--r--  1 w1518355 student 10498082 Aug 26 04:25 usage-compute20-20170823-002506-003600.csv
-rw-r--r--  1 w1518355 student 10501077 Aug 26 12:30 usage-compute20-20170823-002506-003700.csv
-rw-r--r--  1 w1518355 student 10495712 Aug 26 15:55 usage-compute20-20170823-002506-003800.csv
-rw-r--r--  1 w1518355 student 10499670 Aug 26 19:15 usage-compute20-20170823-002506-003900.csv
-rw-r--r--  1 w1518355 student 10496657 Aug 26 22:07 usage-compute20-20170823-002506-004000.csv
-rw-r--r--  1 w1518355 student 10501778 Aug 27 02:22 usage-compute20-20170823-002506-004100.csv
-rw-r--r--  1 w1518355 student 10500406 Aug 27 08:04 usage-compute20-20170823-002506-004200.csv
-rw-r--r--  1 w1518355 student 10501910 Aug 27 14:03 usage-compute20-20170823-002506-004300.csv
-rw-r--r--  1 w1518355 student 10488910 Aug 27 20:04 usage-compute20-20170823-002506-004400.csv
-rw-r--r--  1 w1518355 student 10487697 Aug 28 03:14 usage-compute20-20170823-002506-004500.csv
-rw-r--r--  1 w1518355 student 10500013 Aug 28 12:57 usage-compute20-20170823-002506-004600.csv
-rw-r--r--  1 w1518355 student 10490772 Aug 30 05:28 usage-compute20-20170823-002506-004700.csv
-rw-r--r--  1 w1518355 student 10492238 Aug 31 03:41 usage-compute20-20170823-002506-004800.csv
[w1518355@westminster-hpc usage]$
```

Figure 9: Logs and output files structure

7.3.2. SAMPLING

Sampling proved to be one of the most important logging features implemented. While examining every decision process in MASB simulation is virtually impossible, frequent and recurrent analysis of the details and values was useful for fine-tuning the system and the scoring functions. Not all the details of every single decision process were logged, only a small percentage of all invocations. In the current implementation, the following items are sampled:

- The selection of overloading tasks by the NA, ca.1 sample per 50 invocations;
- The scoring and selection of candidate nodes by the BA, ca.1 sample per 5000 invocations;
- The selection of the target node from the candidate node list, ca.1 sample per 5000 invocations.

7.4. SIMULATION RUNS

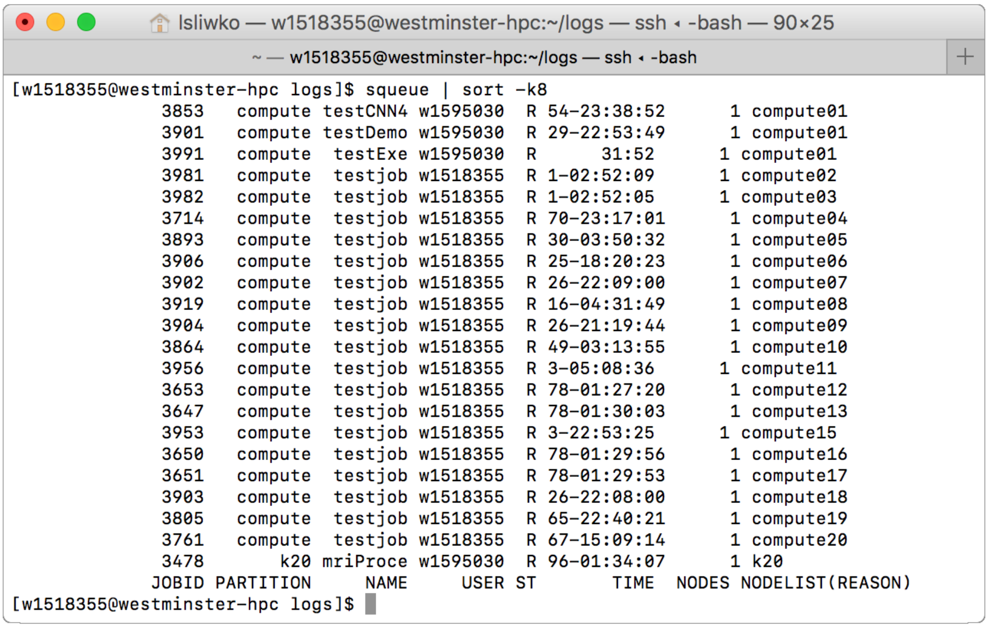
During the later stages of the development of the MASB prototype, several simulations were continuously run. They were frequently paused, tuned and then resumed to see whether a given tweak would improve the results. This methodology allowed to progress the research at a good speed while simultaneously iterating a number of ideas and tweaks. Therefore, the performance testing did not have noticeable stages, but instead the stages blended into each other. This said, it is possible to logically split the performance testing into four main areas:

- Benchmarking – GCD workload traces also contain actual Google scheduler task allocations. In the first simulation, MASB will replay recorded events, mirroring tasks allocations as per the Google scheduler. This simulation was used as a controlling run in order to test the system, and also as a benchmark to compare results with the original allocations.
- Performance testing – secondly, MASB was tested to identify whether it was capable of allocating the same workload as Google's BorgMaster system. The size of the workload was then increased gradually in 2% steps while preserving the configuration of the system nodes. The results were then compared to the original GCD workload.

- Migration Cost – thirdly, a collection of different scoring functions and their variants were tried in order to research their impact on total migration cost while allocating the given workload.
- Scalability – finally, the MASB simulation was run with multiples of a base workload in order to test the scalability limits of the designed solution. Although this step was the least work-intensive, it took the longest time to perform.

As noted in (Zhu et al., 2015), simulating GCD workload is not a trivial task. The main challenge when running such large and complex simulations is the demand for computation power and the continuous processing. During this experiment, the AGOCS framework was modified to also allow the testing of computing cells larger than 12.5k. This was achieved by duplicating randomly selected existing tasks and their events, for example 'Create Task A event from GCD workload trace files' will create events AddTaskWorkloadEvent events for task A and A'. This feature is based on the hashcode of object's id which is a constant value.

The largest experiments simulated a single Cloud computing cell with 100k nodes and required nine months of uninterrupted processing on one of the University of Westminster HPC cluster's nodes. At this juncture, it should be noted that early simulations often fail due to unforeseen circumstances, such as NAS detachment or network failure. One solution to this is to frequently save snapshots of the state of the simulation and to keep a number of previous snapshots in case of write file failure. At the peak of experiments, eighteen out of twenty computing nodes were committed to running MASB simulations, as seen in Figure 10.


lsliwko — w1518355@westminster-hpc:~/logs — ssh ◂ -bash — 90×25
~ — w1518355@westminster-hpc:~/logs — ssh ◂ -bash
+

```
[w1518355@westminster-hpc logs]$ squeue | sort -k8
              3853   compute testCNN4 w1595030  R 54-23:38:52      1 compute01
              3901   compute testDemo w1595030  R 29-22:53:49      1 compute01
              3991   compute  testExe w1595030  R      31:52      1 compute01
              3981   compute  testjob w1518355  R 1-02:52:09      1 compute02
              3982   compute  testjob w1518355  R 1-02:52:05      1 compute03
              3714   compute  testjob w1518355  R 70-23:17:01      1 compute04
              3893   compute  testjob w1518355  R 30-03:50:32      1 compute05
              3906   compute  testjob w1518355  R 25-18:20:23      1 compute06
              3902   compute  testjob w1518355  R 26-22:09:00      1 compute07
              3919   compute  testjob w1518355  R 16-04:31:49      1 compute08
              3904   compute  testjob w1518355  R 26-21:19:44      1 compute09
              3864   compute  testjob w1518355  R 49-03:13:55      1 compute10
              3956   compute  testjob w1518355  R 3-05:08:36      1 compute11
              3653   compute  testjob w1518355  R 78-01:27:20      1 compute12
              3647   compute  testjob w1518355  R 78-01:30:03      1 compute13
              3953   compute  testjob w1518355  R 3-22:53:25      1 compute15
              3650   compute  testjob w1518355  R 78-01:29:56      1 compute16
              3651   compute  testjob w1518355  R 78-01:29:53      1 compute17
              3903   compute  testjob w1518355  R 26-22:08:00      1 compute18
              3805   compute  testjob w1518355  R 65-22:40:21      1 compute19
              3761   compute  testjob w1518355  R 67-15:09:14      1 compute20
              3478       k20 mriProce w1595030  R 96-01:34:07      1 k20
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
[w1518355@westminster-hpc logs]$ 
```

Figure 10: University of Westminster HPC Cluster utilisation

7.5. Allocation Score Ratios

Clearly, when examining the suitability of load balancing, the key parameter is the number of overloaded nodes, which should be kept to minimum. It was found that replaying GCD traces using the Google's original Borg's allocation decisions results in up to 0.5% nodes being overloaded in a simulated one-minute period. It was assumed that this phenomenon was the result of delayed and compacted resource usage statistics which were recorded and averaged over ten-minute periods. Therefore, in further experiments this ratio was used as an acceptable error margin.

The second researched property was how nodes were distributed amongst allocation score types during simulations. Therefore, each experiment recorded a number of nodes with each allocation score type, and averaged them out over the simulation period. The set of normalised values for STA, TA, PA and DA are referred to as Allocation Score Ratios (ASR). Idle Nodes and Overloaded Nodes are discussed separately, and they are excluded from the ASR. The ASR values describe how well the Cluster is balanced, that is, how well nodes are balanced as a whole group. The ASR values are used to describe the experimental results presented in the subsections below to highlight the differences in how various scheduling strategies perform under a GCD workload.

Figure 11 chart visualises the AS distribution during a month-long simulation. The most dominant AS was PA, meaning that each of the node's resources is utilised between 0% and 70%. Ca.68% of all the cluster's nodes are found within these parameters, which is direct result of their initial allocation using SIAS function. The second biggest group, ca.22% of all servers, are nodes allocated disproportionally in which one or more resources are highly used but the other resources are relatively idle. The remainder of the nodes have either an STA or TA allocation score type. The PA to DA ratio of roughly 3:1 is characteristic for a typical workload as recorded in GCD traces and processed by MASB.

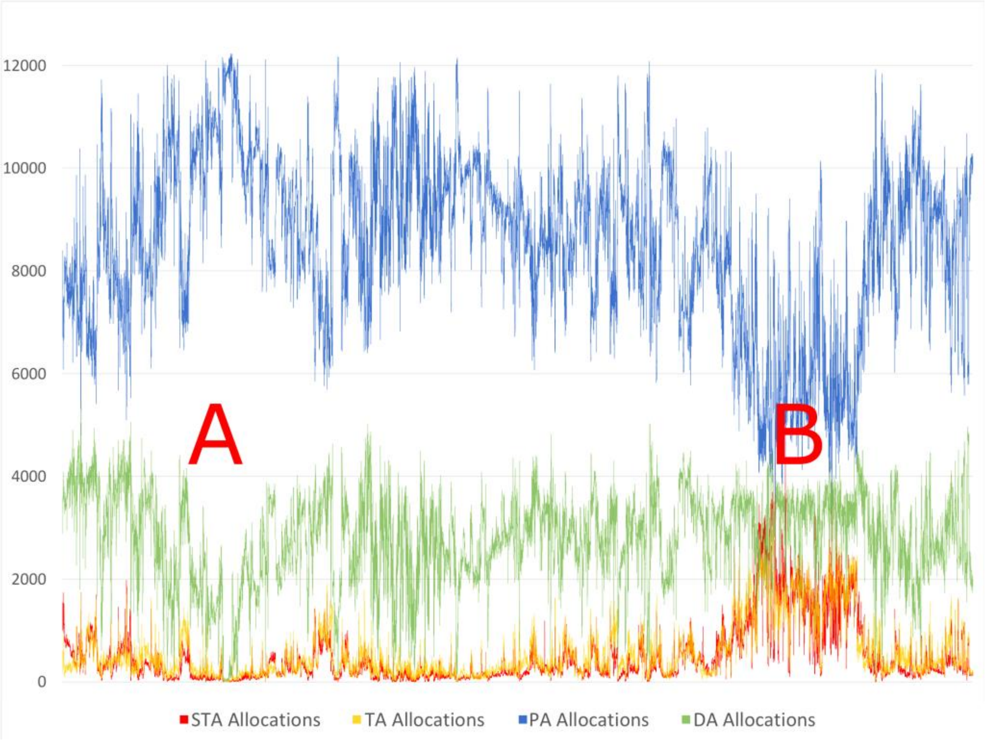


Figure 11: MASB – Allocation Scores distribution (12.5k nodes)

The chart also highlights two periods of low and elevated workload, marked A and B respectively:

- During the low workload period (A), SIAS function can schedule the majority of newly-arriving tasks to relatively unused nodes, thereby successfully preserving their resource usage proportions. As such, the number of PAs increases while the number of DAs decreases. Existing long-running services continue to run uninterrupted on their nodes, and so the ratio of STA to TA remains flat.
- During elevated workload period (B), SIAS function is unable to find relatively unused nodes anymore. It thus selects lower quality allocations, resulting in a decrease in PAs. Due to the

scarcity of resources, services are also reallocated more frequently by SRAS function. This results in tighter fit allocations, which is seen as an increase in STAs and TAs counts.

This cycle is repeated thorough cluster activity, wherein MASB balances the workload. The sections which follow describe a number of implemented optimisations and their rationales, as well as the experimental results and a commentary on them.

7.6. Benchmark

Given that GCD traces, have a complicated structure and contain a vast amount of data, only rarely are they analysed to the full extent of their complexity. MASB design shares similarities with BorgMaster in areas such as constraining tasks, defining memory and CPU cores as resources, using scoring functions for candidate node selection, and handling RUS instances. It also closely follows the lifecycle of tasks as presented in (Sliwko and Getov, 2016). As things stand, there is no publicly available literature which contains descriptions of similar experiments which could be compared with the simulation results of MASB. Therefore, the closest comparable results are the original Borg's allocation decisions that were recorded in GCD traces. For the purposes of this research, it was decided that they be used as a benchmark for the results from MASB's experiments.

Both simulations processed full month-long GCD traces. The average values were used because MASB simulation works in one-minute intervals whilst GCD traces provide usage statistics in ten-minute windows that occasionally overlap. Given this, peak or median values were not accurate. To highlight differences in workings between the MASB and Google Borg algorithms, Figure 12 presents the AS distribution during the period recorded in GCD.

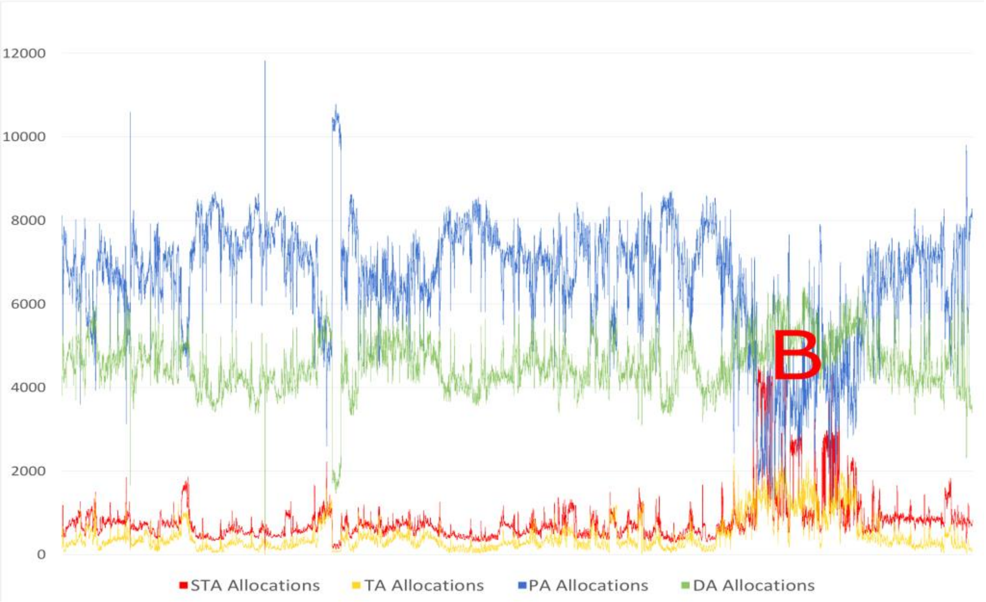


Figure 12: Borg – Allocation Scores distribution (12.5k nodes)

In comparison to experimental data presented in Figure 12, MASB behaves more organically during periods of low and elevated workload. This is especially visible during the period of elevated workload (B) where MASB managed to preserve a better ratio of PA to DA Nodes than Google's Borg. This conduct is the result of allowing a given task to be reallocated during its execution, meaning that MASB can dynamically shape its workload and improve the health of its allocations. This feature also allows greater flexibility in altering the requirements of running services, in which the load balancer attempts to offload an alternative node.

Parameter **(average, one-minute interval)**	**Framework**	
	Borg	**MASB**
Idle Nodes	1.01 (0.01%)	78.10 (0.63%)
STA[1] Nodes	820.49 (6.58%)	487.01 (3.91%)
TA[1] Nodes	459.57 (3.69%)	564.18 (4.53%)
PA[1] Nodes	6597.14 (52.94%)	8508.08 (68.28%)
DA[1] Nodes	4578.49 (36.74%)	2810.69 (22.56%)
Overloaded Nodes	4.04 (0.03%)	12.62 (0.10%)

1. STA, TA, PA and DA as defined in section **Error! Reference source not found.**.

Table 3: Benchmark results – Borg and MASB

Table 3 directly compares ARS parameters of both pre-recorded Google's Borg and MASB simulations. The listed ASR values highlight the differences in Borg and MASB workings:

- Idle Nodes – Borg's design has a definite advantage over MASB because Borg's schedulers can access the shared cluster's state and iterate over the complete set of system nodes. MASB relies on a network of BAs, each of which has only partial information about the cluster's state. Therefore, a subset of idle nodes might never be scored, even if they represent the best allocation for a given service.
- STA and TA Nodes – in both systems, under normal workload conditions, incoming tasks are reasonably well distributed between the nodes. Only ca.10% of all system nodes register higher resource usage scores, when at least one of resource utilisation levels crosses 90%. The exact scoring algorithm of Google's Borg has not been disclosed, but the results suggest a degree of similarity to the SIAS function.
- PA and DA Nodes – the ratio of PAs to DAs is visibly different in Borg and MASB. Borg's original scheduling decisions had a ratio of roughly 3:2, meaning that for every three proportionally allocated nodes in the system, there were two nodes that were disproportionately allocated. MASB managed to achieve a better ratio of 3:1, suggesting that the use of SIAS and SRAS scoring functions together with VM-LM feature can potentially create a more balanced scheduling system.

Given the superior ratio of PA to DA nodes as measured, and the possibility of increased performance (throughput), the next experiment focused on processing increased workload.

7.7. Performance

The MASB framework has been designed as a general solution for balancing workload in a decentralised computing system. It works correctly with default parameters. However, in order to achieve high resources utilisation and low resources waste, extensive fine-tuning was required.

The importance of fine-tuning can be seen in the fact that, for example, MASB was able to initially schedule only ca.85% of GCD tasks without overly overloading the nodes. Several enhancements were implemented, including:

- Splitting the SAS function into SIAS and SRAS and then limiting the number of suitable nodes examined in those functions (200 and 2k respectively);

- Fine-tuning SCS function to maintain the balance between migration cost and the node allocation score as specified in (Sliwko and Getov, 2017), which refers to finding the right combination of steps of the TS algorithm, as well as its termination depth;
- Limiting the number of candidate nodes returned from BA to fifteen and introducing forced migrations.

After numerous iterations, MASB was eventually able to schedule the entire GCD workload, with additional tasks also added. Table 4 presents a comparison of the results with the additional workload.

Parameter (average per minute)	**Workload Size**			
	100% (original)	**102%**	**104%**	**106%**
Nodes Count	12460.39[1]	12460.36[1]	12460.68[1]	12460.35[1]
Tasks Count	132061.15[1]	134738.92[1]	137399.93[1]	142936.05[1]
Global CPU Usage Ratio	43.64%	44.54%	45.42%	46.89%
Global Memory Usage Ratio	62.05%	63.33%	64.58%	66.57%
Idle Nodes	76.41 (0.61%)	73.08 (0.59%)	72.75 (0.58%)	52.18 (0.42%)
STA Nodes	479.91 (3.85%)	480.22 (3.85%)	423.51 (3.40%)	447.73 (3.59%)
TA Nodes	566.20 (4.54%)	545.74 (4.38%)	447.75 (3.59%)	355.88 (2.86%)
PA Nodes	8507.49 (68.28%)	8718.76 (69.97%)	9316.35 (74.77%)	9576.67 (76.86%)
DA Nodes	2818.11 (22.62%)	2610.04 (20.95%)	2084.06 (16.73%)	1718.08 (13.79%)
Overloaded Nodes	12.28 (0.10%)	32.53 (0.26%)	116.25 (0.93%)	309.79 (2.49%)

1. The AGOCS framework itself records minuscule variances in node and task counts that are the result of concurrent update operations, while modifying shared context data objects. In practice, less than 0.04% of update operations were affected. See (Sliwko and Getov, 2016) for details.

Table 4: Performance results (throughput, 100%-106% workload size)

As demonstrated above, MASB was able to schedule, on average, an additional ca.2.6k tasks per minute. Further tuning was unable to improve those results, with workload sizes greater than 102% increasing the number of overloaded nodes above the defined threshold of 0.5%.

To further ensure the correctness of the attained results, another set of experiments was run in parallel. Here, instead of multiplying the original GCD workload, the random machines were removed from the cluster until the workload could no longer be fitted. This method, known as 'cell compaction', is suggested in (Verma et al., 2015). Table 5 details the experimental results:

Parameter (average per minute)	Cluster Size			
	100% (original)	99%	98%	97%
Nodes Count	12460.39	12332.92	12218.61	12081.30
Tasks Count	132061.15	132057.96	132057.54	132055.86
Global CPU Usage Ratio	43.64%	44.09%	44.52%	45.05%
Global Memory Usage Ratio	62.05%	62.72%	63.39%	64.08%
Idle Nodes	76.41 (0.61%)	53.29 (0.43%)	58.96 (0.48%)	75.87 (0.63%)
STA Nodes	479.91 (3.85%)	480.28 (3.89%)	404.13 (3.31%)	448.67 (3.71%)
TA Nodes	566.20 (4.54%)	572.24 (4.64%)	485.55 (3.97%)	500.75 (4.14%)
PA Nodes	8507.49 (68.28%)	8412.71 (68.21%)	8866.13 (72.56%)	8663.88 (71.66%)
DA Nodes	2818.11 (22.62%)	2800.30 (22.71%)	2361.64 (19.33%)	2339.31 (19.35%)
Overloaded Nodes	12.28 (0.10%)	14.11 (0.11%)	42.20 (0.35%)	62.07 (0.51%)

Table 5: Performance results (throughput, 97%-100% cluster size)

Similar to the previously detailed experiments which had augmented workload, even when the cluster size was reduced to ca.98% of its original size (ca.242 nodes being removed on average), the original GCD workload could still be fitted without breaching the 0.5% limit of overloaded nodes.

Although the throughput of the original Google Scheduler could not be significantly improved, the results from both methods of evaluation show the benefits of using VM-LM to fit additional tasks in an already very tightly-fitted cluster.

On average, GCD traces utilise ca.40-50% of the globally available CPUs and ca.60-70% of globally available memory while continuously guaranteeing ca.85% of CPUs and ca.70% of memory to production tasks to handle RUS instances. It should be noted that Borg's scheduling routines have been perfected following decades of work by a team of brilliant Google engineers. The conclusion of this research is that, it is hard to substantially improve this impressive result given those constraints.

7.8. MIGRATION COST

The MASB framework relies on a VM-LM feature to balance workload by moving running tasks across Cloud nodes. While the VM-LM process is reasonably cheap in terms of the computing power, it does incur a non-trivial cost on the Cloud's infrastructure. In order to avoid excessive networks transfers, NAs are carefully deciding which services will be migrated out from a given node. To score candidate services the SCS function is used which takes the service's estimated migration cost into consideration as well as released resources (see 6.1 for more details). Unexpectedly, when searching for ways to lower the total migration cost, although modifications of SCS function seemed to be the most palpable place to start, significantly better results were not obtained. Based on experience from previous experiments, it was discovered that the biggest reduction in service migrations was achieved by improving the quality of the initial task allocation.

Parameter (average per minute)	**Scoring Functions**			
	SIAS **SRAS**	**SIAS** **SRAS|GAIN**	**SIAS|GAIN** **SRAS**	**SIAS|GAIN** **SRAS|GAIN**
Total Migration Cost [GB]	1490.65	7008.30	1252.50	5925.41
Idle Nodes	83.54 (0.67%)	105.80 (0.85%)	76.14 (0.61%)	79.33 (0.64%)
STA Nodes	495.09 (3.97%)	687.77 (5.52%)	490.42 (3.94%)	654.18 (5.25%)
TA Nodes	656.67 (4.54%)	560.65 (4.50%)	558.57 (4.48%)	547.38 (4.39%)
PA Nodes	8515.95 (68.34%)	8492.21 (68.15%)	8511.61 (68.31%)	8451.12 (67.82%)
DA Nodes	2785.23 (22.35%)	2586.43 (20.76%)	2810.95 (22.56%)	2707.73 (21.73%)
Overloaded Nodes	14.88 (0.12%)	27.54 (0.22%)	12.67 (0.10%)	20.61 (0.17%)

Table 6: Results comparison of SAS and SIAS+SRAS (migration cost)

The initial approach focused on prioritising gains in the AS for an individual node. For example, if when migrating a task to node A its AS increased by 20%, it was prioritised over migration to node B which would increase its score by 10% even if node B score had reached its highest allowable utilisation of all resources, i.e. 90% CPU cores and 90% memory. The scoring function variants of SIAS and SRAS are defined here as SIAS|GAIN and SRAS|GAIN respectively:

- $SIAS|GAIN = SIAS(T') - SIAS(T)$ and
- $SRAS|GAIN = SRAS(T') - SRAS(T)$

where T is the current set of allocated tasks, and T' is the candidate set of allocated tasks on a given node. Table 6 presents the results under the variants of the scoring functions.

The combination of SIAS|GAIN and SRAS functions was most efficient (i.e. the total average migration cost was lowest) while ASR remained virtually unchanged. The experiment showed that although focusing on an individual node's allocation score gains is a valid strategy during initial service allocation, service reallocation should be dedicated to maximizing the absolute value of the allocation score of all nodes. As mentioned above, the majority of tasks scheduled on the GCD cluster are short-lived batch jobs which tend not to have high resource requirements (Sliwko and Getov, 2016). As such, there is no need to carefully fit them to a node. As a result of their limited time on the cluster, the chance of reallocation is low. Long-running services, however, should be fitted tightly onto available nodes and continue to run there due to the additional cost of further reallocations because of the typically large amounts of used memory.

7.9. SCALABILITY

Parameter (average per minute)	**Cluster Size**			
	12.5k (original)	**25k**	**50k**	**100k**
Nodes Count	12460.70	24921.49	49842.99	99685.97
Tasks Count	132061.35	264155.80	528336.38	1056645.92
Idle Nodes	71.61 (0.57%)	95.82 (0.38%)	226.42 (0.45%)	413.03 (0.41%)
STA Nodes	492.67 (3.95%)	805.60 (3.23%)	1920.99 (3.85%)	3868.22 (3.88%)
TA Nodes	570.37 (4.58%)	962.14 (3.86%)	2232.10 (4.47%)	4300.70 (4.31%)
PA Nodes	8502.06 (68.23%)	18118.11 (72.70%)	34102.21 (68.33%)	68999.49 (69.13%)
DA Nodes	2812.74 (22.57%)	4914.55 (19.72%)	11324.77 (22.69%)	22031.79 (22.07%)
Overloaded Nodes	11.26 (0.09%)	25.25 (0.10%)	36.49 (0.07%)	71.83 (0.07%)

Table 7: Scalability tests – 12.5k, 25k, 50k and 100k nodes

The final step in the experiments was to examine the scalability of the MASB framework. Due to the simulation's high computational requirements, its one-minute time slices were split into 'rounds' in which every NA could both respond to migration requests as well as send its own requests, although sent requests would be unanswered until the next 'round'. This meant that the simulated scenarios were as realistic as possible whilst also emulating massive Cloud installations. The typical time required to run a single full simulation with 100k nodes cluster, with both nodes and tasks proportionally multiplied, was around nine months. In order to maximise the resource usage of the testing environment, several simulations were run in parallel on the University of Westminster HPC cluster.

Google has never disclosed the size of their largest cluster, but it has been noted in Verma et al. (2015) that Borg computing cells are similarly sized to the clusters managed by Microsoft's Apollo system, which have in excess of 20k nodes (Boutin et al., 2014). A 12.5k node cells in GCD traces have been described as 'average' or 'median', cells with fewer than 5k nodes have been called 'small' or 'test' (Verma et al., 2015). Additionally, (ibid.) gives an example of a larger cell C, which is 150% the size of cell A and therefore also approximately 20k nodes. As such, in this research it is assumed that the computing cell of the large Borg is around 20-25k nodes.

Table 7 demonstrates the results achieved through the multiplication of the original GCD workload; it also highlights the lack of changes in ASR values. MASB was able to orchestrate a cell size of 100k without a noticeable scalability cost and without crossing the limit of 0.5% overloaded nodes. With the current MASB framework implementation, the simulation of this size took around nine months on a single node of the University of Westminster HPC.

One of the questions raised is whether this result could be repeated in a live environment. Although a high-fidelity simulation gives high confidence, this question cannot be answered without actual field tests. Google engineers performed extremely well when building and later fine-tuning Borg's scheduler. The Google Borg system was built with hardware without virtualisation support and offered no mechanism to reallocate the running services to alternative machines, while the concept of MASB relies on VM-LM feature which allows running services to be offloaded. Therefore, the actual field tests would require very substantial monetary investments.

8. Related Work

During this research, several related papers and patents were identified and analysed. In addition to the Omega scheduler described above (Schwarzkopf et al., 2013), several other systems based on a similar concept were identified and are described below.

8.1. ANGEL

The ANGEL system (Zhu et al., 2015) is based on a similar concept wherein a multi-agent system manages its workload in a virtualised Cloud environment. This solution also takes advantage of the VM-LM feature to reallocate running tasks to an alternative node if necessary. Within this system there are three kinds of agents, namely Task Agent, VM agent, and Manager Agent. Task Agent exists for the duration of a task, created upon task arrival and destroyed when the task is complete. VM Agent represents a VM hypervisor running on a physical node and accepting/rejecting tasks. Manager Agent acts as a leader for this computing cell and stores the complete system state in a 'VM Information Board'. Manager Agent also uses stored state information to match Task Agents with VM Agents, a process which is referred to as 'basic matching phase'.

As in MASB, the key task scheduling process is the selection of the environment where the task is to be executed. In ANGEL, newly-arrived Task Agents are bidding for available VMs, using the information stored in Manager Agent. This process is referred to as the 'forward announcement-bidding phase' in the paper. The VM Agent then selects which task is will accept, a process referred to as 'backward announcement-bidding phase'. VM Agents are constantly updating Manager Agent as to changes in their state, such as available resource (CPU and available memory) changes, VM creations and cancellations.

While the basic concept of ANGEL and the MASB system is similar, the design of the architecture differs substantially. In comparison, during the development of MASB, it was found that the sheer number of tasks made it unpractical to create an entity for each task responsible for its allocation; given this, the responsibility was assigned to NAs. In MASB, NAs themselves are responsible for keeping their node stable and offloading overloading tasks to alternative nodes.

The ANGEL stores the global system state in Manager Agent, while in MASB a subsystem of BAs has this responsibility. The ANGEL system assumes that the stored system state is always current, and so bases its allocation decisions on this fact. MASB, however, accepts this information as outdated by design, and so uses it only for building initial candidate nodes list which is then sent to NAs. The actual task allocation is resolved later between NAs themselves.

MASB is focused on a Cluster performance and scalability whereby resource usages gaps are reduced and tasks are fitted into available nodes. The focus of the project was to achieve tightness of task allocations no worse than in the GCD traces while improving scalability. The aim of ANGEL is to guarantee the ratio of tasks guaranteed to meet their deadlines which are also priority-adjusted. It should be noted that the authors of ANGEL also tested their solution on GCD traces. In so doing, they acknowledged the difficulty of conducting experiments on the whole month-long traces because of the enormous count of tasks in the trace logs and performed their experiments exclusively on day 18th of traces which has been recognised as being the most representative time period in GCD traces (Moreno et al., 2013). However, the results presented use different metrics and do not specify further details of the experiments, such as whether authors also matched task constraints and whether tasks were allocated with regards to handing RUS instances.

8.2. US Patent 5,031,089

(Liu and Silvester, 1991) filed a patent which described a set of routines that could be deployed on nodes in order to balance system-wide workload in better way. The first routine periodically examines a number of jobs on the node's queue and computes the 'workload value' which is then provided on request to other nodes by the second routine. The third routine, meanwhile, is triggered periodically when the node is idle, and at the end of each job completion. This routine contains the main bulk of load balancing logic and evaluates whether the node's 'workload value' is below a pre-established value which would indicate that the node is relatively idle. If the node is recognised as being under-utilised and available for more jobs, then the routine will poll all the other nodes for their 'workload value', and transfer jobs from the node with the highest 'workload value' to its own queue.

The feasibility of this invention was validated via a number of simulations, although those results are not shared in the cited patent. The authors list several assumptions made during the performance testing of this study, such as the homogeneity of all the tasks and their resource requirements, as well as the assumption that the job's transfer cost is negligible. Only the job’s queue length was used as 'workload value'.

The main criticism of this solution is that it oversimplifies the Cluster workload's model, and it omits the continuous changes of resources used by jobs. Also, only non-started jobs can be transferred to alternative nodes. The solution relies on polling all nodes in the cluster for their utilisation levels which in a large cluster might be not feasible and may create a bottleneck.

8.3. US Patent 8,645,745

In this publication (Barsness et al., 2014), authors note that there is a problem when a centralised job scheduler needs to pass through a large number of nodes in order to find one which can be used to allocate and run the task. If this scheduler manages the workload of a large computing cell, the scheduling efficiency is reduced. The scheduler may also need to keep track of the currently running task execution states. When the centralised scheduler is tasked to manage a large number of tasks and simultaneously update the global system state, a bottleneck may be created. Additionally, a single point of failure is created, meaning that when a centralised scheduler fails, the entire computing cell cannot function.

To solve the above issues, instead of a centralised job scheduler, a solution is proposed whereby each node is continuously scanning a shared-file to determine which job could be executed on this node. When a job requires multiple nodes, the one on the nodes becomes a primary node which then assigns and monitors the job execution on the multiple nodes.

In comparison to MASB, the main similarity is that there is no centralised manager to assign tasks to nodes which means that nodes are themselves responsible for selecting and then running the accepted tasks. In addition, there are substantial differences in terms of task allocation routines. In this patent, nodes select tasks from shared–files based on task requirements, while in MASB a task allocation is a multi-step process in which each node tries to increase its AS by selecting and offloading tasks. Given that the patent paper provides no results from experiments, it is difficult to directly compare systems' performances.

9. SUMMARY AND CONCLUSIONS

The primary challenge when sequencing a queue of tasks on a cluster is to fit them tightly, and in so doing reduce resource usage gaps. The scheduling algorithm attempts to reduce the situations where a resource on a given node is overly un-utilised at the same time that other resources on that node are mostly allocated. It is extremely important to shrink the gaps in resource utilisation and to allocate them proportionally, especially when initially scheduling new tasks which tend to have balanced resource requirements.

Fitting objects of different volume into a finite number of containers is known as bin-packing problem and belongs to class of NP-Hard problems. The traditional way of solving NP-Hard problems are meta-heuristic algorithms. However, experiments demonstrated that although meta-heuristic algorithms yield good solutions, they do not scale well to the required number of nodes in a Cloud system.

Alternative solutions and a large number of optimisations can be devised, such as caching computed solutions and then retrieving them based on task similarity, multiple concurrent schedulers working on a single data store, and pre-allocating resources for the whole task batches (Verma et al., 2015). However, these solutions and optimisations still incur substantial computational costs, and it is inevitable that any model where the head node processes all scheduling logic by itself will eventually work less effectively when the cluster size grows and the frequency of incoming tasks increases.

The MASB framework offers an alternative approach to task allocations in that all the actual processing of scheduling logic is offloaded to nodes themselves. This framework uses loose coupling at every stage of its scheduling flow, meaning that scheduling decisions are made only on locally-cached knowledge and all communication between nodes is kept to minimum. Each node tries to increase its AS by selecting and offloading tasks, with the assumption being that by bettering individual ASs, the global system performance will be improved. This design also takes advantage of the VM-LM feature, where a running service within a VM instance can be migrated on the fly to an alternative node without stopping a program execution.

Design of this schema created a set of new challenges, such as selecting alternative nodes with limited and non-current knowledge about the state of other nodes, estimating the VM-LM cost of a running service, understanding the classifying and scoring functions of the allocation type of a node, and designing the stateless node-to-node communication protocol, to identify just a few.

In this research, realistic (i.e. pre-recorded) workload traces from GCD were used and were run on the AGOCS framework as a very detailed simulation (Sliwko and Getov, 2016). The costs involved were the substantial computing power required to run experiments as well as time, in that a single simulation run took about a month on a forty-core (twenty physical cores + HT siblings) machine. In order to benchmark the research results, original scheduling decisions made by Google's Borg scheduler are examined which are also part of GCD traces. This generated statistics such as total resource usage, the number of idle nodes and production-allocated resources.

When examining GCD traces, it is important to note that Google's engineers did a phenomenal job in first designing and then iteratively improving the Borg system. Incoming tasks are packed very tightly and, although production jobs always have additional resources available to them within defined requirements' limits, the spare resources are efficiently recycled for low priority jobs. Google Cluster has been built upon hardware without direct support for virtualisation, meaning that its orchestrating software design had to accommodate this limitation. This research should be considered an as-if scenario and assumes the availability of the VM-LM feature to shuffle running tasks within a Cluster.

In this research, there was only limited success in terms of improving the throughput of executed tasks on a simulated computing cell which was mainly due to the constraints arising from handling RUS instances. However, MASB could achieve higher scalability and run multiple sizes of examined computing cell without noticeable scalability costs. Simulations up to 100k nodes from GCD were tested, yielding relatively comparable results when run with smaller instances of simulations.

Although the experimental results prove that it is feasible to deploy the presented decentralised architecture in a live environment, there are a number of possible other improvements, as listed below:

- During experimentations, several nodes remained idle. This effect was a result of iterating only a limited number of nodes while computing a set of candidate nodes for a given task migration. A potential solution to this issue is a separate size-limited list of relatively under-utilized nodes which would be compulsorily scored each time a BA is issued a GetCandidateNodesRequest request. Such a list could be exchanged separately between BAs;

- The SCS routine (step 1 in the SAN protocol) is triggered only when the NA detects that its node is overloaded. However, the system could employ a more proactive approach in which the NA would periodically try to offload its services in order to improve its AS, even if the node is stable. This would create a secondary mechanism to distribute the load, which would potentially reduce resource utilization gaps even further. However, this feature would also place additional pressure on BAs and, as such, needs to be carefully balanced;

- In a real-world system it is expected that a number of nodes will experience failure. NA's AI module could maintain a set of blacklisted nodes which repeatedly did not respond to requests. Such a set could be shared with BAs, similar to the way it is implemented in Fuxi (Zhang et al., 2014), and presented to system administrators.

- The MASB prototype does not address fault-tolerance which is an important aspect of Cloud design. This feature could be realised in multiple ways, such as running cloned instances of services, periodically saving process checkpoints, and ensuring the applications' state is synchronised across all its instances;

- The proposed design does not account for task priorities, i.e. tasks are only split into production and non-production groups. Production tasks have committed resources which, under normal circumstances, are guaranteed to be available. However, during critical system-wide failures, such as a power failure or network infrastructure collapse, the system should degrade gracefully (as opposed to an uncontrolled crash). In scenarios where the current workload cannot be sustained, the system should shut down lower priority tasks first and use remaining available nodes to offload high-priority tasks;

- In this project, it is assumed that NAs and BAs agents are continuously running without breakdowns. Nevertheless, agents are also a piece of software, meaning that they are prone to bugs and errors. As a possible improvement to detect and restore hung agents, a hierarchy model could be introduced in which an agent supervises a number of other agents and restarts them if necessary. This concept is similar to the Akka Actors implementation (Roestenburg et al., 2015) in which a parent actor manages the failures of its children. Additionally, a hierarchy of BAs could be used to propagate the cluster's state knowledge in a more efficient manner;

- The presented experimental results are of good quality; nevertheless, they suggest a number of possible improvements, especially in locating and then scheduling tasks to idle nodes. One possible improvement could be to share idle nodes between all BAs and then prioritising them over utilised nodes;

- Energy efficiency is the next possible area to expand. In its current design, MASB focuses on reducing the cost of service reallocations necessary to keep the Cloud system stable. However, this approach could be shifted to focus on completely offloading idle nodes, at which point the system would be able to power down those nodes in order to save energy;

- Resource usage quotas per user, group or other entity, would make another welcome feature, often present in commercial Cluster schedulers. However, it would also require adding an accountancy module with a decentralised dataset in order to maintain scalability;

- MASB estimates the task migration cost and considers this value it when selecting which tasks to migrate out from a node. However, it does not calculate the fact that neighbouring nodes (e.g. those in the same server rack) might offer much faster transfer rates than more remote nodes. Therefore, adjusting the task migration cost by the nodes' distances could improve the overall cluster performance;

- MASB does not attempt to implement locality optimisation, when the task's part of a distributed file is processed faster if accessed locally. Currently, GCD task descriptions contain only restrictions which disallow nodes that the task could be executed on. However, adding optional meta-data, such as the ID of the distributed file's part, could prioritise a set of nodes and improve overall the cluster performance. This functionality is featured in some of the Big Data frameworks.

The suggested directions of future study and possible expansions as listed above have the potential to improve the results of this research. Nevertheless, the focus of this paper was to research and design a feasible strategy for managing and balancing a workload within a virtualized Cloud system, an objective which has been achieved.

REFERENCES

Abdul-Rahman, Omar Arif, and Kento Aida. "Towards understanding the usage behavior of Google cloud users: the mice and elephants phenomenon." In Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on, pp. 272-277. IEEE, 2014.

Agnetis, Allesandro, Pitu B. Mirchandani, Dario Pacciarelli, and Andrea Pacifici. "Scheduling problems with two competing agents." Operations research 52, no. 2 (2004): 229-242.

Baker, Kenneth R., and J. Cole Smith. "A multiple-criterion model for machine scheduling." Journal of scheduling 6, no. 1 (2003): 7-16.

Barsness, Eric L., David L. Darrington, Ray L. Lucas, and John M. Santosuosso. "Distributed job scheduling in a multi-nodal environment." U.S. Patent 8,645,745, issued February 4, 2014.

Bigham, John, and Lin Du. "Cooperative negotiation in a multi-agent system for real-time load balancing of a mobile cellular network." In Proceedings of the second international joint conference on Autonomous agents and multiagent systems, pp. 568-575. ACM, 2003.

Boutin, Eric, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing." In OSDI, vol. 14, pp. 285-300. 2014.

Brazier, Frances MT, Frank Cornelissen, Rune Gustavsson, Catholijn M. Jonker, Olle Lindeberg, Bianca Polak, and Jan Treur. "A multi-agent system performing one-to-many negotiation for load balancing of electricity use." Electronic Commerce Research and Applications 1, no. 2 (2002): 208-224.

Brenner, Walter, Rüdiger Zarnekow, and Hartmut Wittig. "Intelligent software agents: foundations and applications." Springer Science & Business Media, 2012.

Brooks, Chris, Brian Tierney, and William Johnston. "JAVA agents for distributed system management." LBNL Report (1997).

Buyya, Rajkumar. "High Performance Cluster Computing: Architectures and Systems, Volume I." Prentice Hall, Upper SaddleRiver, NJ, USA 1 (1999): 999.

Cabri, Giacomo, Luca Ferrari, Letizia Leonardi, and Raffaele Quitadamo. "Strong agent mobility for aglets based on the ibm jikesrvm." In Proceedings of the 2006 ACM symposium on Applied computing, pp. 90-95. ACM, 2006.

Cao, Junwei, Daniel P. Spooner, Stephen A. Jarvis, and Graham R. Nudd. "Grid load balancing using intelligent agents." Future generation computer systems 21, no. 1 (2005): 135-149.

Castelfranchi, Cristiano. "Guarantees for autonomy in cognitive agent architecture." In International Workshop on Agent Theories, Architectures, and Languages, pp. 56-70. Springer, Berlin, Heidelberg, 1994.

Chavez, Anthony, Alexandros Moukas, and Pattie Maes. "Challenger: A multi-agent system for distributed resource allocation." In Proceedings of the first international conference on Autonomous agents, pp. 323-331. ACM, 1997.

Corsava, Sophia, and Vladimir Getov. "Intelligent architecture for automatic resource allocation in computer clusters." In Parallel and Distributed Processing Symposium, 2003. Proceedings. International, pp. 8, IEEE, 2003.

Eddy, YS Foo, Hoay Beng Gooi, and Shuai Xun Chen. "Multi-agent system for distributed management of microgrids." IEEE Transactions on power systems 30, no. 1 (2015): 24-34.

Gensereth, Michael R., and Steven P. Ketchpel. "Software agents." Communications of the ACM 37, no. 7 (1994): 48.

Gentzsch, Wolfgang. "Sun grid engine: Towards creating a compute power grid." In Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on, pp. 35-36. IEEE, 2001.

Goodwin, Richard. "Formalizing properties of agents." Journal of Logic and Computation 5, no. 6 (1995): 763-781.

Guilfoyle, Christine, and Ellie Warner. "Intelligent agents: The new revolution in software." Ovum, 1994.

Ilie, Sorin, and Costin Bădică. "Multi-agent approach to distributed ant colony optimization." Science of Computer Programming 78, no. 6 (2013): 762-774.

Jackson, David, Quinn Snell, and Mark Clement. "Core algorithms of the Maui scheduler." In Workshop on Job Scheduling Strategies for Parallel Processing, pp. 87-102. Springer, Berlin, Heidelberg, 2001.

Jones, James Patton, and Cristy Brickell. "Second evaluation of job queuing/scheduling software: Phase 1 report." NAS Technical Report NAS-97-013, NASA Ames Research Center, 1997.

Kim, Gu Su, Kyoung-in Kim, and Young Ik Eom. "Dynamic load balancing scheme based on Resource reservation for migration of agent in the pure P2P network environment." In International Conference on AI, Simulation, and Planning in High Autonomy Systems, pp. 538-546. Springer, Berlin, Heidelberg, 2004.

Klusáček, Dalibor, Václav Chlumský, and Hana Rudová. "Optimizing user oriented job scheduling within TORQUE." In SuperComputing The 25th International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13). 2013.

Lewis, Ian, and David Oppenheimer. "Advanced Scheduling in Kubernetes". Kubernetes.io., Google Inc. March 31, 2017. Available https://kubernetes.io/blog/2017/03/advanced-scheduling-in-kubernetes Retrieved January 4, 2018.

Liu, Howard T., and John A. Silvester. "Dynamic resource allocation scheme for distributed heterogeneous computer systems." U.S. Patent 5,031,089, issued July 9, 1991.

Liu, Peng, and Lixin Tang. "Two-agent scheduling with linear deteriorating jobs on a single machine." In International Computing and Combinatorics Conference, pp. 642-650. Springer Berlin Heidelberg, 2008.

Long, Qingqi, Jie Lin, and Zhixun Sun. "Agent scheduling model for adaptive dynamic load balancing in agent-based distributed simulations." Simulation Modelling Practice and Theory 19, no. 4 (2011): 1021-1034.

Milano, Michela, and Andrea Roli. "MAGMA: a multiagent architecture for metaheuristics." IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) 34, no. 2 (2004): 925-941.

Montresor, Alberto, Hein Meling, and Ozalp Babaoglu. "Messor: Load-balancing through a swarm of autonomous agents." In AP2PC, vol. 2, pp. 125-137. 2002.

Moreno, Ismael Solis, Peter Garraghan, Paul Townend, and Jie Xu. "An approach for characterizing workloads in google cloud to derive realistic resource utilization models." In Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on, pp. 49-60. IEEE, 2013.

Nguyen, Ngoc Thanh, Maria Ganzha, and Marcin Paprzycki. "A consensus-based multi-agent approach for information retrieval in internet." In International Conference on Computational Science, pp. 208-215. Springer, Berlin, Heidelberg, 2006.

Nong, Q. Q., T. C. E. Cheng, and C. T. Ng. "Two-agent scheduling to minimize the total cost." European Journal of Operational Research 215, no. 1 (2011): 39-44.

Nwana, Hyacinth S. "Software agents: An overview." The knowledge engineering review 11, no. 3 (1996): 205-244.

Roestenburg, Raymond, Rob Bakker, and Rob Williams. "Akka in action." Manning Publications Co., 2015.

Sargent, P. "Back to school for a brand new ABC." The Guardian (March 12), no. 3 (1992): 12-28.

Schaerf, Andrea, Yoav Shoham, and Moshe Tennenholtz. "Adaptive load balancing: A study in multi-agent learning." Journal of artificial intelligence research 2 (1995): 475-500.

Schwarzkopf, Malte, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. "Omega: flexible, scalable schedulers for large compute clusters." In Proceedings of the 8th ACM European Conference on Computer Systems, pp. 351-364. ACM, 2013.

Sharma, Bikash, Victor Chudnovsky, Joseph L. Hellerstein, Rasekh Rifaat, and Chita R. Das. "Modeling and synthesizing task placement constraints in Google compute clusters." In Proceedings of the 2nd ACM Symposium on Cloud Computing, p. 3. ACM, 2011.

Shi, Dongcai, Jianwei Yin, Wenyu Zhang, Jinxiang Dong, and Dandan Xiong. "A distributed collaborative design framework for multidisciplinary design optimization." In International Conference on Computer Supported Cooperative Work in Design, pp. 294-303. Springer, Berlin, Heidelberg, 2005.

Sliwko, Leszek, and Vladimir Getov. "AGOCS – Accurate Google Cloud Simulator Framework." In Scalable Computing and Communications Congress, 2016 Intl IEEE Conferences, pp. 550-558. IEEE, 2016.

Sliwko, Leszek, and Vladimir Getov. “Transfer Cost of Virtual Machine Live Migration in Cloud Systems.” DIS-RG-TR, University of Westminster, Nov. 2017.

Tuong, N. Huynh, Ameur Soukhal, and J-C. Billaut. "Single-machine multi-agent scheduling problems with a global objective function." Journal of Scheduling 15, no. 3 (2012): 311-321.

Verma, Abhishek, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. "Large-scale cluster management at Google with Borg." In Proceedings of the Tenth European Conference on Computer Systems, p. 18. ACM, 2015.

Weiss, Gerhard. "Multiagent Systems. 2nd edition." MIT Press. 2013

Wilkes, John. "Cluster Management at Google with Borg." GOTO Berlin 2016. Nov 15, 2016.

Wooldridge, Michael, and Nicholas R. Jennings. "Intelligent agents: Theory and practice." The knowledge engineering review 10, no. 2 (1995): 115-152.

Xu, Cheng-Zhong, and Brian Wims. "A mobile agent based push methodology for global parallel computing." Concurrency - Practice and Experience 12, no. 8 (2000): 705-726.

Yang, Yongjian, Yajun Chen, Xiaodong Cao, and Jiubin Ju. "Load balancing using mobile agent and a novel algorithm for updating load information partially." In Lecture Notes in Computer Science 3619, pp. 1243-1252. 2005.

Yi, Sangho, Artur Andrzejak, and Derrick Kondo. "Monetary cost-aware checkpointing and migration on amazon cloud spot instances." IEEE Transactions on Services Computing 5, no. 4 (2012): 512-524.

Yin, Yunqiang, Shuenn-Ren Cheng, T. C. E. Cheng, Wen-Hung Wu, and Chin-Chia Wu. "Two-agent single-machine scheduling with release times and deadlines." International Journal of Shipping and Transport Logistics 5, no. 1 (2013): 75-94.

Yoo, Andy B., Morris A. Jette, and Mark Grondona. "Slurm: Simple linux utility for resource management." In Workshop on Job Scheduling Strategies for Parallel Processing, pp. 44-60. Springer, Berlin, Heidelberg, 2003.

Zhang, Dongli, Moussa Ehsan, Michael Ferdman, and Radu Sion. "DIMMer: A case for turning off DIMMs in clouds." In Proceedings of the ACM Symposium on Cloud Computing, pp. 1-8. ACM, 2014.

Zhang, Zhuo, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. "Fuxi: a fault-tolerant resource management and job scheduling system at internet scale." Proceedings of the VLDB Endowment 7, no. 13 (2014): 1393-1404.

Zhu, Xiaomin, Chao Chen, Laurence T. Yang, and Yang Xiang. "ANGEL: Agent-based scheduling for real-time tasks in virtualized clouds." IEEE Transactions on Computers 64, no. 12 (2015): 3389-3403.