Simulation Based Job Scheduling Optimization for Batch Workloads

Dheeraj Chahal
TCS Research
Mumbai, India
d.chahal@tcs.com

Benny Mathew
TCS Research
Mumbai, India
benny1.m@tcs.com

Manoj Nambiar
TCS Research
Mumbai, India
m.nambiar@tcs.com

ABSTRACT

We present a simulation based approach for scheduling jobs that are part of a batch workflow. Our objective is to minimize the makespan, defined as completion time of the last job to leave the system in a batch workflow with dependencies. The existing job schedulers make scheduling decisions based on available cores, memory size, priority or execution time of jobs. This does not guarantee minimum makespan since contention for resources among concurrently running jobs are ignored.

In our approach, prior to scheduling batch jobs on physical servers, we simulate the execution of jobs using a discrete event simulator. The simulator considers available cores and available memory bandwidth on distributed systems to accurately simulate the execution of jobs using resource contention models in a concurrent run. We also propose simulation based job scheduling algorithms that use underlying contention models and minimize the makespan by optimally mapping jobs onto the available nodes. Our approach ensures that job dependencies are adhered to during the simulation.

We assess the efficacy of our job scheduling algorithms and contention models by performing experiments on a real cluster. Our experimental results show that simulation based approach improves the makespan by 15% to 35% depending on the nature of workload.

CCS CONCEPTS

• **Computing methodologies → Modeling and simulation**;

KEYWORDS

Batch jobs, makespan, job scheduling algorithms, service demand

ACM Reference Format:
Dheeraj Chahal, Benny Mathew, and Manoj Nambiar. 2019. Simulation Based Job Scheduling Optimization for Batch Workloads. In *Tenth ACM/SPEC International Conference on Performance Engineering (ICPE '19), April 7–11, 2019, Mumbai, India.* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3297663.3310312

ICPE '19, April 7–11, 2019, Mumbai, India

ACM ISBN 978-1-4503-6239-9/19/04…$15.00
https://doi.org/10.1145/3297663.3310312

1 INTRODUCTION

Workflows representing complex applications are important in several business and research domains such as banking, medical, industrial automation and telecom, bioinformatics, astronomy and earth sciences [23]. Jobs in these workflows are coupled through control or data dependencies such that execution of a job may start only after the execution of one or more other jobs. Moreover, these jobs are compute or memory intensive and require high performance computing servers. Distributed environment comprising of high-performance servers with large number of cores, high memory bandwidth and high speed inter-connects are used for execution of such workflows. A job scheduler is used to manage the resources available in such distributed environment and schedule jobs on these resources. However, on shared-memory multiprocessor platforms, interconnects between main memory and processors become bottlenecks. Moreover, different jobs have different resource requirements and availability of these resources keeps changing as jobs that are part of the workflow arrive and exit.

In this context, the problem of job scheduling refers to allocating the available resources such as cores, memory, network, I/O to the jobs and establish an optimal order of their execution by mapping jobs onto the most suitable server. In such a scenario, multiple resource aware job scheduling can result in improved performance.

Scheduling jobs in a distributed environment is a NP-hard problem even in case of two identical servers [13]. Use of meta-heuristic approach is the most suitable option to solve job scheduling problems. However, meta-heuristic methods impose numerous challenges in real-world situations where manual tuning of parameters is difficult or where the execution time is small. In such situations, pure heuristics provide better solutions.

Scheduling of jobs in distributed environments have been studied extensively. However, with the advent of new architectures having large number of cores and high memory bandwidth, there is a need to study the simulation based methods to address the challenges posed by such servers and the complex workloads that run on them.

Most job schedulers are either load aware or consider contention on only one of the resources such as CPU, IO or network. These job schedulers are not optimal for clusters where resource requirement vary from job to job. For example, co-scheduling multiple CPU and/or memory bandwidth intensive jobs can result in contention for these resources and need to be scheduled using multiple resource aware scheduler.

In our previous work, we developed models for predicting the execution time of CPU and memory bandwidth intensive batch jobs when these jobs run concurrently [6, 7]. In this work, we propose job scheduling algorithms using CPU and memory bandwidth contention models. The proposed algorithms use simulation based

approach to generate a job onto the server mapping resulting in an improved makespan.

Our approach can be implemented successfully in scenarios where:

(1) Workflow can be presented as a direct acyclic graph and there are dependencies among jobs.
(2) There is more than one server available for scheduling jobs.
(3) Multiple jobs may run concurrently on the server and there is contention for CPU and memory bandwidth.

The contributions of our work are as follows:

(1) We propose two simulation based job scheduling algorithms to minimize the makespan of an application workflow.
(2) We propose a method to use CPU and memory bandwidth contention models for optimal scheduling of batch workflows.
(3) An experimental evaluation of the scheduling algorithms using makespan as a metric.

In our model, we consider the homogeneous servers for scheduling jobs. However, the model can be applied for heterogeneous servers as well with job characterization efforts on each unique server. The scope of this work is limited to modeling systems with CPU and memory bandwidth intensive workloads. The idle time is used to represent wait on IO or RPC for remote resources. Detailed modeling of external resources is out of scope of this paper. However our model can be extended to include these resources.

Our simulation approach and proposed algorithms can be integrated with the existing resource management tools to improve their job scheduling capabilities.

Rest of the paper is organized as follows. Related work is discussed in section 2. Problem statement and the workflow is explained in the section 3. Resource contention models and job scheduling algorithms are discussed in section 4. The experimental details and results are discussed in section 5. Conclusion and future scope is presented in section 6.

2 RELATED WORK

Extensive work has been performed in last two decades in the field of job shop scheduling (JSS) [8] [4]. JSS is an optimization problem such that we are given n jobs $J_1, J_2 \ldots, J_n$ with differing service demands that needs to be scheduled on m different machines of varying processing power so their makespan is minimized. Each job consists of a sequence of operations which must be performed in some order and on a specific machine. Flexible job shop scheduling (FJSS), an extension of JSS technique, is an important research problem in the domain of production management and combinatorial optimization [18]. FJSS allows one job to run on more than one machine.

The workflow optimization problem for computer systems differs significantly from the JSS and FJSS problem. There is contention for various resources while one job is running concurrently with other jobs on the same machine. Hence methods used for solving JSS and FJSS problems can not be used directly for addressing the workflow optimization problem that we are solving.

Researchers have developed techniques for cost and deadline constrained scientific workflow scheduling and performance prediction in a cloud environment [15, 19]. Our work is different from the cloud based unpredictable workloads containing workflows. We use the profile of jobs obtained from runs in isolation as input.

Rodriguez *et al.* have developed algorithms for workflows using the meta-heuristic technique, particle swarm optimization (PSO) [22]. Chen *et al.* used well known ant colony optimization technique for workflow optimization and QoS requirements [9]. Yu *et al.* proposed scheduling techniques for workflows using genetic algorithms [27]. These meta-heuristic techniques generate good results but may not be used in many real-world problems where execution time is small and systems are automated such that manual tuning is impossible.

Simulation based techniques to study workloads for distributed systems is a popular approach. Buyya *et al.* developed a popular toolkit called GridSim for modeling and simulation of distributed resource management [5]. Zheng *et al.* used monte-carlo simulations for stochastic DAG scheduling [28]. Practitioners use simulations for studying scheduling methods in large scale systems. Ramaswamy *et al.* used monte-carlo simulations for scalable behavioral emulation of extreme scale systems [21].

Researchers have developed resource aware scheduling tools for advanced architectures. Herbein *et al.* developed an IO aware scheduling approach for HPC clusters [12]. IBM has implemented network-aware scheduling in their production resource manager and scheduler [2]. In a similar work, Xu *et al.* developed policies to mitigate memory bandwidth contention using bandwidth aware scheduling [25]. In another work, prediction accuracy and co-scheduling performance due to cache-coherence and memory bandwidth contention was studied in detail [10, 11]. None of these methods account for fine job characterization of CPU and memory bandwidth requirements and how their concurrent execution interferes with each other in a shared memory server architecture.

There exists research work on predicting the service demand (CPU time in isolation) [14, 20], memory bandwidth requirement [24] of jobs and effects of memory or thread placement on multi-core shared memory parallel system [26] but very little efforts have been made to use service demand and memory bandwidth requirements of a job for predicting the execution time in a concurrent run using simulation methods.

The proposed algorithms using CPU-memory bandwidth contention models for co-scheduling jobs on a distributed system are original contributions that have not been experimented earlier.

3 THE WORKFLOW MODEL AND PROBLEM STATEMENT

Consider an application batch workflow represented by a DAG G = (V, E). Here Vertices V = $\{v_1, v_2, \ldots, v_n\}$ in the graph represents multithreaded batch jobs and E is set of edges representing the precedence among the jobs. A dependency $(v_i, v_j) \in E$ implies that job v_j can start execution only after v_i has finished its execution.

Given that total i machines are available for scheduling represented by $M = \{m_1, m_2, \ldots m_i\}$. Let n be the number of jobs in the workflow DAG represented by $V = \{v_1, v_2, \ldots . v_n\}$. Job v_j executes on machine m_i in time $ET_{i,j}$. If V_j represents the jobs scheduled on machine i, schedule length is denoted as

$$ET_i = \sum_{j \in V_j} \{E_{i,j}\} \quad (1)$$

and the makespan is

$$ET_{max} = max_{i \in M}\{ET_i\} \quad (2)$$

Based on this, problem can be defined as follows. Given a DAG G representing a batch workflow, find job mapping for each job v_j ($for\ 1 \leq j \leq n$) onto the available server m_k ($for\ 1 \leq k \leq i$) for execution such that the makespan ET_{max} of the DAG is minimized.

4 OUR APPROACH

In this section we discuss the various steps that are part of our approach. A high-level structural model we use is as shown in Figure 1. We first characterize all batch jobs in isolation. In this work we are assuming all machines to be identical such that $ET_{i,j}$ of a job $v_j \in V$ is same on all servers $m_i \in M$. Hence we can characterize each job only on one server. The job characterization data includes service demand, resource utilization etc. Execution of jobs represented as DAG is simulated using our simulator called DESiDE [16]. DESiDE is a discrete event simulation tool which implements CPU and memory contention model for concurrent run of jobs.

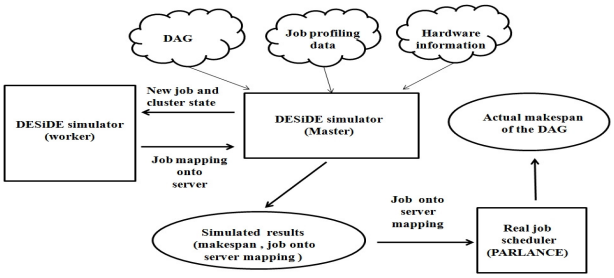

Figure 1: The model

The job characterization data, the number of available servers and DAG is given as input to the simulator. The master instance of simulator starts the execution of the workflow from the root node in the DAG and also keeps track of remaining execution time of each active job. On completion of the job, all successor jobs that are eligible for the execution are derived from the DAG. For each newly arriving job, we generate the service demand from its demand distribution [6, 7] . The execution of each new job is simulated by another instance of simulator called the worker, using contention models, current state of the servers and the status of currently executing jobs on these servers. The worker simulator uses one of our job scheduling algorithms to find most suitable server for mapping each of the new jobs and returns the job mapping information to the master simulator. Master simulator maps the new job to the server and updates the status of the server.

The simulation process starts from the root job in the DAG and continues till all the jobs and their successors in the DAG are simulated. At the completion of DAG simulation, an execution schedule with job onto the server mapping is generated by the master simulator. The job onto the server mapping generated by the simulator can now be given to a real job scheduler for execution on physical servers. This approach is explained in more detail in the following sub sections.

4.1 Job Characterization

First we characterize each DAG job in isolation. We collect per thread service demand (CPU time), number of threads, per thread memory bandwidth requirement BW_{req}, CPU utilization U_t measured at small intervals of execution of each job in the batch. Our model characterizes service demand (SD) of each thread of a particular job individually. Since we take service demand distributions as input, any inaccuracy or missing information during job characterization can be defined from this distribution or appropriate assumptions. Each thread of a job may have unique demand characteristics due to the fact that work may not be equally divided between threads. When CPUs are oversubscribed, the execution time of a thread will affect the execution time of peer threads that are part of the same job or across the jobs running concurrently.

In case the CPU utilization of a job changes during its execution, we record the utilization at small intervals of time. The CPU utilization data can be fit in a regression function (linear, non-linear) or represented as some distribution. While simulating the concurrent execution of the jobs, instantaneous value of CPU utilization is used instead of average value.

We also observe average memory bandwidth requirement of individual thread when jobs are profiled in isolation. The memory bandwidth requirement of a thread is observed for both local and remote memory access. The available memory bandwidth for local and remote memory are different. Latencies due to remote and local memory access is replicated while simulating the job execution.

4.2 Job Scheduling Algorithms

A job upon completion in a DAG may spawn one or more jobs that needs to be mapped onto servers available for execution. In this section we discuss two heuristic algorithms called LET and MOM that we have developed for mapping jobs onto most appropriate servers in the distributed environment. We also describe the average load based algorithm called SLF that we have implemented in our simulator.

4.2.1 Least Execution Time (LET). In this approach we use least execution time of the job on all available servers as a criterion for mapping onto a server. On completion of a job in the DAG, new jobs may be spawned and need to be mapped onto servers for execution. Current state of each server (number of cores available) and remaining execution time of each job running on them is extracted. New job v_j is concurrently executed with already running jobs on all servers $M = \{m_1, m_2, ...m_n\}$ and its simulated execution time $ET(v_j, m_i)$ for $1 \leq i \leq n$, is observed. A server m_k is selected from the set of servers such that

$$m_k = min\{ET(v_j, m_i)\} for 1 \leq i \leq n \quad (3)$$

New job is mapped onto m_k for execution and state of the server is updated. The pseudocode for the algorithm is as shown in Algorithm 1.

4.2.2 Min of Max (MOM). In MOM algorithm we use minimum impact of incoming job on currently running jobs as a criterion for mapping. We simulate the execution of new job on all servers $M = \{m_1, m_2, ...m_n\}$ by running concurrently with the already executing set of jobs A_{ni} on machine m_l. Execution time of the longest running job on each server i.e. $T_l = max\{ET_l(A_i, m_i)\}$ for

Algorithm 1: LET algorithm

Data: Input DAG $G = (V, E)$ representing a workflow
Result: Job mapping onto the server and execution time of the workflow
Schedule ROOT job in the DAG on a server
On completion of ROOT, find successor jobs $V_j = \{v_1, v_2, ..v_m\}$
while $V_j \neq EMPTY$ **do**
  for *each job* v_i *in* V_j **do**
    for *each server* $m_l, l = 1, 2, 3....n$ **do**
      Simulate concurrent execution of the job with active jobs and find execution time ET_l of v_i $\{ET_l(v_i, m_l)\}$
    end
    find server m_k executing v_i in least time $\min\{ET_l\} for\ 1 \leq l \leq n$
    Allocate job to m_k and update server state
  end
  On completion of v_i find its successors jobs and update V_j
end

Algorithm 2: MOM algorithm

Data: Input DAG $G = (V, E)$ representing a workflow
Result: Job mapping onto the server and execution time of the workflow
Schedule ROOT job in the DAG on a server
On completion of ROOT, find successor jobs $V_j = \{v_1, v_2.., v_m\}$
while $V_j \neq EMPTY$ **do**
  for *each job* v_i *in* V_j **do**
    for *each server* $m_l, l = 1, 2, 3....n$ **do**
      Simulate concurrent execution of the job v_i with active jobs $A_{ni} \subset V$ on machine m_l
      Find execution time ET_l of longest running job T_l on machine machine m_l
      $T_l = max\{ET_l(A_{ni}, m_l)\}$
    end
    Find server m_k where longest running job is fastest among all servers $min(T_l)$ for $1 \leq l \leq n$ Allocate job to m_k and update server state
  end
  On completion of v_i find its successors jobs and update V_j
end

$1 \leq i \leq n$ is observed. New job is mapped onto a server m_k where longest running job on any server is fastest among all the servers i.e. $min(T_l)$ for $1 \leq l \leq n$. This process is repeated till all jobs are finished in the DAG. The workload and state of the server is updated after job allocation. The pseudo code of the algorithm is shown in Algorithm 2.

4.2.3 Smallest Load First (SLF). Average load based scheduling is a popular approach used by most of the job schedulers for load balancing. We have implemented SLF in our simulation tool and compared it with the performance of the above mentioned algorithms with it. Unlike the LET and MOM algorithms, worker simulation instance is not required since load information of each server is already available with master. In this approach, we calculate the load L_l on each server in $M = \{m_1, m_2, ...m_n\}$ instantaneously

$$L_l = \frac{No.\ of\ threads\ on\ the\ server}{No.\ of\ cores\ on\ the\ server} \quad (4)$$

New job is scheduled on a server m_k with minimum load among all servers

$$m_k = min\{L_l, m_l\}\ for\ 1 \leq l \leq\ n \quad (5)$$

4.3 CPU and Memory Contention Model

Once an appropriate server has been identified for the new job, it is added to the current pool of jobs running on that server in the simulator. In our previous works we have developed CPU and memory contention models [6, 7] for predicting the execution time of a job in a batch when multiple jobs start at the same time. We improved and extended these models in the current work while simulating the concurrent execution of workflow jobs. These models are discussed here.

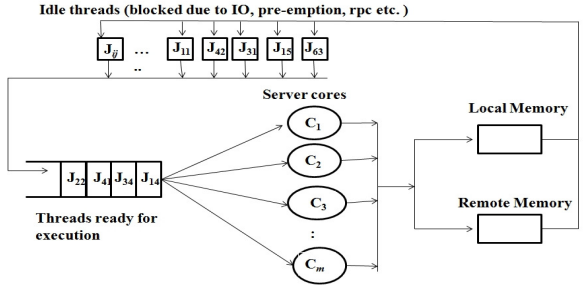

Figure 2: CPU and memory contention model

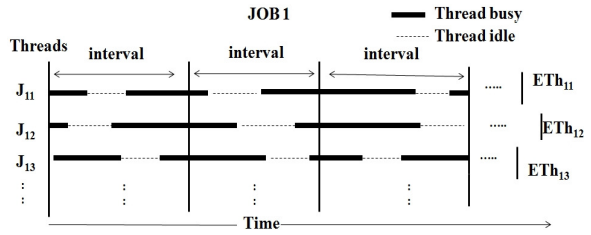

Figure 3: Execution of threads in small intervals

4.3.1 CPU contention model. During its execution, a thread migrates between between multiple states. In our model we assume that when a thread is ready for execution, it is mapped onto a core for execution (Figure 2). A thread acquires core to execute for some time and then migrates to an idle state by releasing the core to perform some work e.g. IO, rpc etc. The completion time of a thread is sum total of time spent by thread in different states. In our CPU contention model we divide the total execution time ETh_{ij} of each thread J_{ij} of a job i with j threads in n small intervals of size time T such that $ETh_{ij} = n * T$ as shown in Figure 3. In each interval, thread is executing in the CPU for some duration and is idle for the remaining time of the interval. The time for which the thread acquires the CPU is derived from the CPU utilization observed for the thread in isolation. The busy time t_{busy} and the idle time t_{idle} of a thread in an interval is derived from the CPU utilization U_t at time t

$$t_{busy} = U_t * T \quad (6)$$

$$t_{idle} = (1 - U_t) * T \quad (7)$$

We replicate the behavior of real operating system by including the fluctuations in the idle time and busy time. Also, we do not want all threads to start or go to idle state at the same time. Hence we choose average t_{busy} and t_{idle} randomly from the distribution as follows

$$t_{busy} = [(1 - \sigma) * t_{busy}, (1 + \sigma) * t_{busy}] \quad (8)$$

$$t_{idle} = [(1 - \sigma) * t_{idle}, (1 + \sigma) * t_{idle}] \quad (9)$$

Where σ represents the standard deviation in CPU utilization as observed in small intervals during job characterization in isolation. In our experiments time interval used was .1s. We did not see any improvement in the results by reducing interval length further.

4.3.2 Memory contention model. While a thread is executing using a core, it exchanges the data by reading and writing to the memory. If the total bandwidth requirement of a thread BW_{req} exceeds the available bandwidth BW_{avail} per thread in a core, the busy time t_{busy} of the thread in an interval gets extended by a factor σ_{bw} (Figure 4)

$$\sigma_{bw} = (BW_{req} - BW_{avail})/BW_{avail} \quad (10)$$

The available bandwidth BW_{avail} is measured from the maximum bandwidth BW_{max} available on the server and the number of threads Th_i running.

$$BW_{avail} = BW_{max}/Th_i \quad (11)$$

In case there is thread and memory binding and data is retrieved from local mode, a threadś busy time in an interval gets extended due to memory bandwidth contention as

$$t_{busy} = U_t * T(1 + \sigma_{bw}) \quad (12)$$

However, in the absence of thread and memory binding, threads in a job may access data from a remote memory as well. Based on the research work done by Bardhan *et al.* [3], our simulator also assumes that in the absence of any thread and memory binding, 25% of all threads in a job access data from a remote memory node. Similar behavior was observed by us in our previous work [7]. The additional time due to data retrieval from remote memory δ_r as compared to retrieval time from local memory δ_l results in an overhead δrl

$$\delta_{rl} = \delta_r/\delta_l \quad (13)$$

The busy time of a thread in such a state is given as

$$t_{busy} = U_t * T(1 + \sigma_{bw} * \delta_{rl}) \quad (14)$$

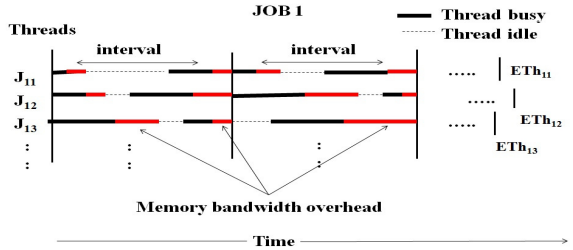

Figure 4: Execution of threads with memory contention in small intervals

4.4 Simulation Model

We enhanced our discrete event simulation tool called DESiDE to implement job scheduling algorithms, CPU and memory contention models as discussed in the sections 4.3 and 4.2 respectively. We have extended DESiDE to accept workflows as a DAG. The profile of each job including per thread bandwidth requirement, CPU utilization etc. as discussed in the section 4.1 is also given as input to the simulation tool. We use processor sharing with time slicing in DESiDE for simulating execution of multi-threaded concurrent jobs. Simulation tool is initialized with the sever information including number of cores, maximum memory bandwidth in the server.

As discussed earlier in the contention model description, total execution time of threads is divided in small intervals. At the beginning of each interval, total bandwidth available and the remaining execution of threads is calculated. If available memory bandwidth is less than the required bandwidth of a thread, its execution time is dilated according to equations 12 and 14.

We run two instances of DESiDE called master and worker. Each job entering the simulator invokes arrival function in the master instance of DESiDE. Master saves the current state of simulation on all servers and invokes the worker instance. The worker instance executes job scheduling algorithms to find the most suitable server for execution. The master instance schedules the job on the server and updates the state of the server. The simulation process continues till all jobs in the DAG are processed. At the end of simulation a job onto server mapping sequence is generated with the estimated time of execution of each individual job and the makespan of the DAG. The job mapping generated using simulation is executed with a real job scheduler.

5 EXPERIMENTS

Firstly, we performed simulations for workflows provided as DAGs using our algorithms and compared the results of our simulation experiments. Then, to verify the applicability of these algorithms in the production environment, we used the same DAG with server mapping generated during simulation and conducted experiments on physical servers using a job scheduler. Results of our experiments are discussed below.

5.1 Experimental Setup

In all our experiments we used Intel Xeon servers running centOS 6.0 Linux system. The simulator DESiDE ran on a single server and simulated execution of DAG on clusters of different sizes. We also tested job mapping generated by simulator using our algorithms on clusters of 2, 3 and 4 physical servers. Each server had 56 logical cores and 80GB/s local memory bandwidth available.

We used an in-house job scheduler called PARLANCE to run the experiments on physical servers. PARLANCE is a proprietary tool that implements average load based scheduling by default but can be programmed to follow user defined job onto the server mappings. Its performance is comparable to other popular job scheduling algorithms that use average load based scheduling strategy such as LSF job scheduler [1]. It has in-built capabilities that allow users to define job constraints and dependencies among jobs.

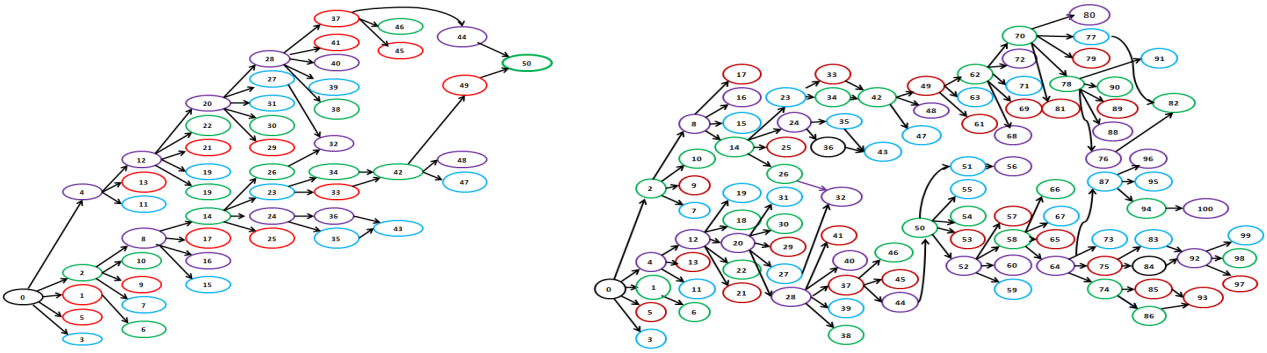

Figure 5: DAG of 50 jobs (DAG-1) and 100 jobs (DAG-2) used in experiments. Colors denote average job bandwidth requirement (red: 7.9-10.5 GB/s, green: 5.4-7.9 GB/s , blue: 3.6-5.4 GB/s , purple: 0-3.6GB/s)

No. of Servers	Simulation time(s)	
	DAG-1	DAG-2
2	40	89
3	56	100
4	73	156
5	92	190

Table 1: No. of servers vs simulation time for DAG-1 and DAG-2 using LET and MOM

STREAM benchmark [17] was used to create various batch workflows for our experiments. The STREAM benchmark workload performs four different vector operations namely copy, add, scalar, triad. These operations are CPU, memory bandwidth intensive and most suitable for testing our contention models and scheduling algorithms. We generate 36 unique STREAM benchmark jobs by varying the number of iterations performed and number of threads in each job. Number of threads per job varies from 4 to 28 and bandwidth requirement per thread varies from 3.6 GB/s to 10.5 GB/s. We constructed DAG of 50 jobs (DAG-1) and 100 jobs (DAG-2) using these 36 unique STREAM jobs. DAG-1 and DAG-2 used for our experiments are shown in Figure 5.

We profiled each of these STREAM jobs in isolation. Service demand, CPU utilization and memory bandwidth of each thread in each job was recorded at fixed intervals.

5.2 Simulations

We evaluated and compared our scheduling algorithms by performing simulations using our tool DESiDE . We instantiate the simulator with direct acyclic graphs containing 50 and 100 jobs i.e. DAG-1 and DAG-2. Apart from the DAG, DESiDE's input is the job characterization data. This data includes service demand of each thread, number of threads, memory bandwidth requirement of each thread and CPU utilization. Simulation starts with the root job in the DAG. On completion of a job, DESiDE refers to the DAG to find new jobs to be spawned. Using various algorithms discussed in the section 4.2, DESiDE maps the job onto the server.

Concurrent execution of jobs on multiple servers is simulated by the tool as described in section 4.4. On completion of the DAG processing, expected completion time of individual job, makespan

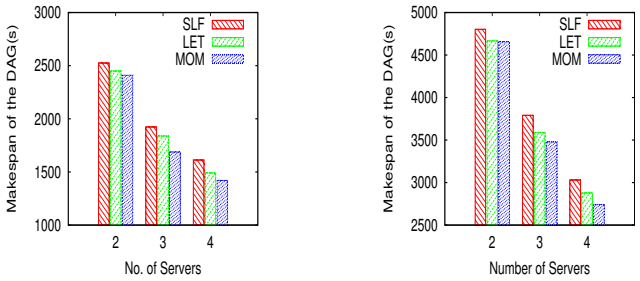

Figure 6: Comparison of job scheduling algorithms using simulator with different cluster sizes (a) 50 job DAG (b) 100 job DAG

of the whole DAG and job onto the server mapping is generated as an output. Simulations were conducted for cluster size of 2, 3 and 4 servers. Execution of jobs on each server was simulated with 56 logical cores and 80GB/s memory bandwidth. The simulations were repeated with three different seeds and the average makespan was calculated. In case job onto the server mapping changes with different seeds, we repeat the simulations till the dominant mapping is identified. Simulation time for DAG-1 and DAG-2 using LET and MOM algorithms on cluster size of 2,3,4 and 5 servers is as shown in Table 1. We observe that with increase in the number of servers, the simulation time also increases. Since we are generating a static schedule, this simulation time is acceptable. In case of dynamic scheduling, optimizations like parallel worker simulators, thread pool of simulators, IPC mechanisms can be used.

Figure 6a shows the comparison of makespan observed using our algorithms with DAG-1. MOM algorithm shows an improvement of approximately 5%, 13% and 12% over SLF using 2, 3 and 4 server clusters respectively. LET shows an improvement of 3%, 5% and 8% over SLF using 2, 3 and 4 servers respectively. Similar results were observed with DAG-2 containing 100 jobs as shown in Figure 6b. MOM algorithm shows an improvement of approximately 3%, 9% and 9% over SLF for 2, 3, 4 servers respectively. LET also performs better than SLF with an improvement of approximately 3%, 5%, 5% in the makespan for 2, 3, 4 servers respectively. Based on these results we can conclude that among three algorithms, MOM performs

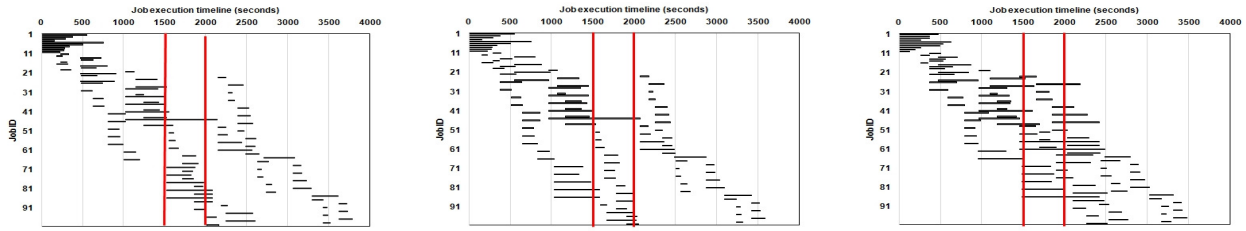

Figure 8: Comparison of expected execution time of individual jobs of DAG-2 on a cluster size of 3 servers using (a) SLF (b) LET (c) MOM

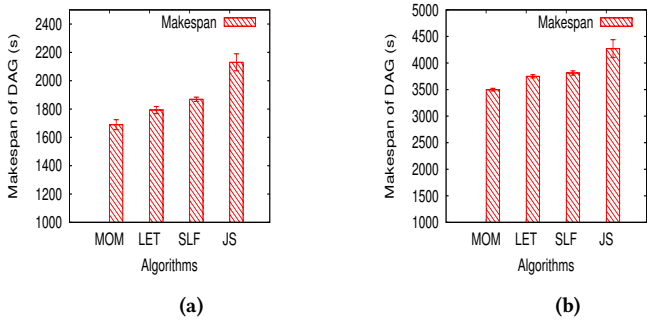

Figure 7: Makespan observed using job mapping generated by the simulator for MOM, LET and SLF and executed with real job scheduler PARLANCE on a cluster of three servers. Makespan generated by PARLANCE in default mode (JS) is used as a baseline

better than LET and SLF. We do not see large improvement when resource availability is very low, which is the case when there are only 2 servers. This is due to the fact that all resources are used to full capacity and there is very little scope for performance improvement. Also makespan does not improve further when resources are under-subscribed as in case of 4 servers.

5.3 Experimental Evaluation

Our objective is to generate an optimal job onto server mapping sequence using simulations that can be executed using any job scheduler. We tested the accuracy of our contention model and simulation algorithms by scheduling the same DAG with simulation generated job server mapping using our in-house job scheduler PARLANCE.

We generated job onto the server mapping with MOM, LET and SLF algorithms using DESiDE. The job mapping sequence generated with simulator for these algorithms is then executed using PARLANCE. Each experiment was repeated 5 times. We also ran PARLANCE in the default mode, henceforth called JS, without user specified mapping and compared with our algorithms.

Figure 7 shows the makespan comparison of our algorithms using the real job scheduler on a cluster of three servers. As shown in Figure 7a, makespan generated using MOM results in the minimum makespan followed by LET, SLF and JS. We observed an improvement of 25%, 15% and 13% over JS for DAG-1 using MOM, LET and SLF mapping respectively. Similar results were observed for DAG-2 (Figure 7b). We observed an improvement of 35%, 24% and 21% in comparison to PARLANCE default schedule JS using MOM, LET and SLF respectively.

Although both SLF and PARLACE use average load based scheduling policy but we see a significant difference in the makespan with job mapping predicted by simulation using SLF and actual measurement run using PARLANCE with SLF routing (JS) as shown in Figure 7 . This has been traced to the manner in which load is calculated for SLF scheduling is PARLANCE. PARLANCE used the UNIX load average to determine which server is least loaded. This is the average CPU load over the last minute. In case several dependent jobs are launched at the same time, the load will not change when checked for each of these dependent jobs. So as long as there are sufficient cores available, all dependent jobs are likely to be launched on the same server.

However in DESiDE, to calculate CPU load we take the instantaneous value of CPU utilization. This means that when a job is scheduled to launch on a particular server, the load average is instantaneously updated. Other jobs that are to be launched at the same simulation time will find this server loaded and will be launched on other less loaded servers.

When the job onto server mapping from simulated SLF is given as input to PARLANCE, the difference is reduced to 5%. This shows close match between experimental and simulated SLF results (see SLF data in Figure 6a and 7a for 3 servers) .

Figure 8 shows the comparison of execution time of individual jobs using SLF, LET and MOM algorithm for DAG-2. The LET algorithm maps new job to a server where its own execution time is minimum without considering its effect on execution time of already running jobs. This job placement policy may affect the execution time of critical jobs in the sequence and can result in higher makespan i.e. finish time of the last job in the DAG. However, MOM algorithm mapping ensures minimum effect of newly placed jobs on the execution time of pre-running jobs. As a result critical jobs that have many dependent jobs are relatively unaffected by newly placed jobs. This is clearly visible in highlighted interval in Figure 8c which shows more concurrently running jobs and lower makespan time as compared to SLF and LET in figure 8a and 8b.

6 CONCLUSION

In this work we have proposed simulation based scheduling algorithms to find an optimal job mapping onto servers which results

in an improved overall execution time of a batch workflow represented as a DAG. We have compared simulation results generated using our algorithms with the execution of the workflow using a real job scheduler. The experimental results show that job mapping generated using least execution time and min of max algorithms minimize makespan of a DAG.

The proposed algorithms are resource aware and generate a job mapping by simulating the concurrent execution of jobs. Simulations are performed using a discrete event simulation tool called DESiDE which implements our CPU and memory contention models. Since the two heuristic algorithms are multiple resource aware, they give better results than SLF, which is only CPU aware. The proposed approach can be integrated with real schedulers for real life applications such as bioinformatics, physics, earth science etc. Our approach is based on CPU and memory bandwidth contention models and modeling IO intensive jobs is work in progress.

We have limited our experiments to a cluster of 4 servers only but this approach can be used with large clusters by parallelizing the worker instance to reduce the simulation time.

In future, we intend to embed the simulator in our job scheduler so that a job is immediately placed on the server when appropriate server for its execution is determined by the simulator.

REFERENCES

[1] [n. d.]. IBM LSF. http://ls11-www.cs.tu-dortmund.de/people/hermes/manuals/LSF/users.pdf. Accessed: 2018-10-25.

[2] [n. d.]. IBM Network-aware scheduling. https://www.ibm.com/support/knowledgecenter/en/SSETD4_9.1.3/lsf_admin/pe_network_aware_sched.html. Accessed: 2018-10-25.

[3] S. Bardhan and D. A. MenascÃĺ. 2015. Predicting the Effect of Memory Contention in Multi-Core Computers Using Analytic Performance Models. *IEEE Trans. Comput.* 64, 8 (Aug 2015), 2279–2292. https://doi.org/10.1109/TC.2014.2361511

[4] S. Binato, W. J. Hery, D. M. Loewenstern, and M. G. C. Resende. 2002. *A Grasp for Job Shop Scheduling.* Springer US, Boston, MA, 59–79. https://doi.org/10.1007/978-1-4615-1507-4_3

[5] Rajkumar Buyya and Manzur Murshed. [n. d.]. GridSim: a toolkit for the modeling and simulation of distributed resource management and scheduling for Grid computing. *Concurrency and Computation: Practice and Experience* 14, 13âĂŞ15 ([n. d.]), 1175–1220. https://doi.org/10.1002/cpe.710 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.710

[6] Dheeraj Chahal and Benny Mathew. 2018. PROWL: Towards Predicting the Runtime of Batch Workloads. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018, Berlin, Germany, April 09-13, 2018.* 199–200. https://doi.org/10.1145/3185768.3186407

[7] Dheeraj Chahal, Benny Mathew, and Manoj K. Nambiar. 2018. Predicting the Runtime of Memory Intensive Batch Workloads. In *The 47th International Conference on Parallel Processing, ICCP 2018, Workshop Proceedings, Eugene, OR, USA, August 13-16, 2018.* 45:1–45:8. https://doi.org/10.1145/3229710.3229756

[8] Imran Ali Chaudhry and Abid Ali Khan. [n. d.]. A research survey: review of flexible job shop scheduling techniques. *International Transactions in Operational Research* 23, 3 ([n. d.]), 551–591. https://doi.org/10.1111/itor.12199 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/itor.12199

[9] Wei-neng Chen and Jun Zhang. 2009. An Ant Colony Optimization Approach to a Grid Workflow Scheduling Problem With Various QoS Requirements. 39 (01 2009), 29–43.

[10] Andreas De Blanche and Thomas Lundqvist. 2014. A methodology for estimating co-scheduling slowdowns due to memory bus contention on multicore nodes. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks, PDCN 2014 :*. 216–223. https://doi.org/10.2316/P.2014.811-027

[11] Andreas de Blanche and Thomas Lundqvist. 2015. Addressing characterization methods for memory contention aware co-scheduling. *The Journal of Supercomputing* 71, 4 (01 Apr 2015), 1451–1483. https://doi.org/10.1007/s11227-014-1374-8

[12] Stephen Herbein, Dong H. Ahn, Don Lipari, Thomas R.W. Scogland, Marc Stearman, Mark Grondona, Jim Garlick, Becky Springmeyer, and Michela Taufer. 2016. Scalable I/O-Aware Job Scheduling for Burst Buffer Enabled HPC Clusters. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '16).* ACM, New York, NY, USA, 69–80. https://doi.org/10.1145/2907294.2907316

[13] Hesam Izakian, Ajith Abraham, and Vaclav Snasel. 2009. Performance comparison of six efficient pure heuristics for scheduling meta-tasks on heterogeneous distributed environments. *Neural Network World* 19, 6 (2009), 695.

[14] Ajay Kattepur and Manoj Nambiar. 2015. Performance Modeling of Multi-tiered Web Applications with Varying Service Demands. *2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)* 00 (2015), 415–424. https://doi.org/doi.ieeecomputersociety.org/10.1109/IPDPSW.2015.28

[15] Maciej Malawski, Gideon Juve, Ewa Deelman, and Jarek Nabrzyski. 2012. Cost- and Deadline-constrained Provisioning for Scientific Workflow Ensembles in IaaS Clouds. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12).* IEEE Computer Society Press, Los Alamitos, CA, USA, Article 22, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389026

[16] Benny Mathew and Dheeraj Chahal. 2017. DESiDE: Discrete Event Simulation Developers Environment. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17).* ACM, New York, NY, USA, 173–174. https://doi.org/10.1145/3030207.3053672

[17] John D. McCalpin. 1991-2007. *STREAM: Sustainable Memory Bandwidth in High Performance Computers.* Technical Report. University of Virginia, Charlottesville, Virginia. http://www.cs.virginia.edu/stream/ A continually updated technical report. http://www.cs.virginia.edu/stream/.

[18] Maroua Nouiri, Abdelghani Bekrar, Abderezak Jemai, Smail Niar, and Ahmed Chiheb Ammari. 2018. An effective and distributed particle swarm optimization algorithm for flexible job-shop scheduling problem. *Journal of Intelligent Manufacturing* 29, 3 (01 Mar 2018), 603–615. https://doi.org/10.1007/s10845-015-1039-3

[19] I. Pietri, G. Juve, E. Deelman, and R. Sakellariou. 2014. A Performance Model to Estimate Execution Time of Scientific Workflows on the Cloud. In *2014 9th Workshop on Workflows in Support of Large-Scale Science.* 11–19. https://doi.org/10.1109/WORKS.2014.12

[20] J. F. PÃĺrez, G. Casale, and S. Pacheco-Sanchez. 2015. Estimating Computational Requirements in Multi-Threaded Applications. *IEEE Transactions on Software Engineering* 41, 3 (March 2015), 264–278. https://doi.org/10.1109/TSE.2014.2363472

[21] Ajay Ramaswamy, Nalini Kumar, Aravind Neelakantan, Herman Lam, and Greg Stitt. 2018. Scalable Behavioral Emulation of Extreme-Scale Systems Using Structural Simulation Toolkit. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP 2018).* ACM, New York, NY, USA, Article 17, 11 pages. https://doi.org/10.1145/3225058.3225124

[22] Maria Alejandra Rodriguez and Rajkumar Buyya. 2014. Deadline Based Resource Provisioningand Scheduling Algorithm for Scientific Workflows on Clouds. *IEEE Transactions on Cloud Computing* 2 (2014), 222–235.

[23] Jyoti Sahni and Deo Prakash Vidyarthi. 2018. A Cost-Effective Deadline-Constrained Dynamic Scheduling Algorithm for Scientific Workflows in a Cloud Environment. *IEEE Transactions on Cloud Computing* 6 (2018), 2–18.

[24] W. Wang, J. W. Davidson, and M. L. Soffa. 2016. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale NUMA machines. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 419–431. https://doi.org/10.1109/HPCA.2016.7446083

[25] D. Xu, C. Wu, and P. Yew. 2010. On mitigating memory bandwidth contention through bandwidth-aware scheduling. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT).* 237–247.

[26] R. Yang, J. Antony, P. P. Janes, and A. P. Rendell. 2008. Memory and Thread Placement Effects as a Function of Cache Usage: A Study of the Gaussian Chemistry Code on the SunFire X4600 M2. In *2008 International Symposium on Parallel Architectures, Algorithms, and Networks (i-span 2008).* 31–36. https://doi.org/10.1109/I-SPAN.2008.13

[27] Jia Yu and Rajkumar Buyya. 2006. Scheduling Scientific Workflow Applications with Deadline and Budget Constraints Using Genetic Algorithms. *Sci. Program.* 14, 3,4 (Dec. 2006), 217–230. http://dl.acm.org/citation.cfm?id=1376960.1376967

[28] Wei Zheng and Rizos Sakellariou. 2013. Stochastic DAG scheduling using a Monte Carlo approach. *J. Parallel and Distrib. Comput.* 73, 12 (2013), 1673 – 1689. https://doi.org/10.1016/j.jpdc.2013.07.019 Heterogeneity in Parallel and Distributed Computing.