A Reactive Scheduler for Closed Loop Control Systems Using Historical Run-time Statistics

Sudhir Shetiya and Siddhartha SenGupta
TCS Research and Innovation
TATA Consultancy Services
Olympus, Hiranandani Estate
Thane West 400607, Maharashtra, India
sudhir.shetiya@tcs.com, siddhartha.sengupta@tcs.com

Supratim Biswas
Department of Computer Science and Engineering
IIT Bombay
Powai, Mumbai 400076, India
sb@cse.iitb.ac.in

Abstract

This paper presents a method of scheduling tasks of a closed loop control system on multi-core systems using historical statistics - mean run-time and standard-deviation in run-times - of the tasks. The run-time statistics is used to first predictively and then to reactively schedule the tasks in order to minimize the makespan. The predicted schedule is a list of tasks ordered by a function of mean and standard deviation, to be executed on each core. Reactive scheduling involves periodically probing the progress and reacting to imbalances in progress across the cores by altering the schedule. It is shown that, when the tasks have high variance in run-times, this method results in smaller makespan compared to a non-reactive scheduler using plain heuristics. This method also performs better than the default Linux OS task scheduler for such tasks. 9% improvement in makespan over OS task scheduler is noticed for a workload comprising of 1100+ CPU intensive tasks with high standard deviations in run-times on a 20-core system.

Keywords

Scheduling using historical runtime statistics, independent periodic non pre-emptive tasks, makespan, reactive scheduling, homogeneous multi-processor.

1. Introduction

Large closed loop control systems have many components that run as tasks in groups in a cyclic manner. Such control systems manage environments often with hundreds of components in a group where the tasks for these components arrive concurrently. Tasks in a group are independent – a computer can execute them concurrently or in any order. They complete their computation and release the resources held, if any, before the tasks in the next group run (Figure 1).

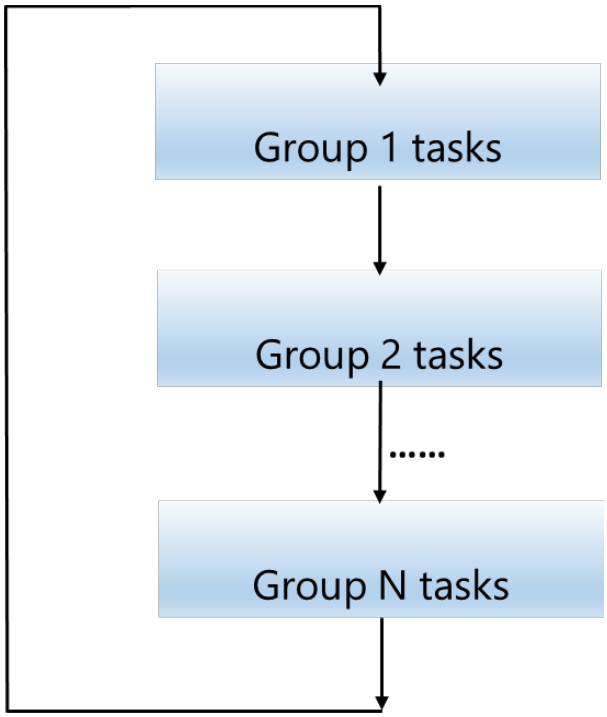

Figure 1: Closed loop control system flow

An example of such a closed loop control system is a Computer Aided Dispatcher for scheduling trains on a large area network. Large railway networks can be partitioned into a number of smaller subnetworks connected at boundary stations (called interchange points / junction stations) so that the scheduling of trains in each sub-network can be done in parallel [SenGupta et al (2015), Sinha et al (2016)]. The planning of the entire network is done iteratively by three group of tasks. Each subnetwork is assigned to a Group 1 planner task that is responsible for generating a schedule of the trains running on the assigned subnetwork independent of all other Group 1 tasks. Group 2 and Group 3 are helper tasks with similarities with Group 1. Tasks in a group execute only after all the tasks in the previous group complete their execution. Group 1 tasks execute after the Group 3 tasks have completed their execution in the previous cycle.

All the tasks are CPU intensive. Tasks in a group are independent and can execute in parallel. The execution time of Group 1 tasks may vary depending on situations (for example, delays in arrival or departure of trains, disruptions) especially on congested sub-networks. In presence of delays, the Group 1 tasks perform significant amount of computations to generate a feasible train schedule in the next cycle.

As tasks in such a control system run repeatedly in a cyclic manner, the control system can easily collect and maintain a history of run-times of the tasks (average processing time μ and standard deviation σ) with very little overhead. This paper investigates the use of this historical run-time statistics, collected in previous cycles, for scheduling the tasks in an efficient manner in the following cycle. An abstracted real-world Computer Aided Dispatcher for scheduling trains on a network comprising of seven railway lines was used for experimentation. The experimental results, on a 20-core system when scheduling 1100+ CPU intensive tasks with high standard deviations in run-times, show 9% improvement in makespan compared to default Linux OS task scheduler.

The requirement of CPU intensive repetitive tasks is not as severe a restriction as it may first appear to be. Such tasks are common in industrial applications. For example, a Computer Aided Dispatcher for scheduling trains on a large area network described above. In the near future, control of plants, large machinery, etc. will require repetitively processing of vast amounts of data from large number of sensors in Internet Of Things (IOT) in near real-time. Data processing in such applications will necessitate deployment of such run-time statistics driven specialized schedulers.

The rest of this paper is organized as follows. Section 2 discusses related work on task scheduling that make use of μ and/or monitors the progress in execution of tasks. Section 3 defines the notations and formulates the problem. Section 4 describes the scheduling methodology. Section 5 describes the computational experiments. Section 6 presents the analysis of experimental results. Finally, section 7 concludes the paper with a summary of the work and contributions.

2. Background and Related Work

The simplest approach to task scheduling on multi-processor environments is not to use any custom scheduler but to allow the underlying Operating System (OS) task scheduler to schedule the tasks. OS task schedulers, such as the Linux Completely Fair Scheduler CFS [Wong (2008), and Kobus and Szklarski (2009)], typically use the priority of runnable tasks and try to be fair to them by allowing each of them to run for small intervals of time (time-slice). The time-slice depends on the cpu-time consumed by a task and its priority. Such systems do not use task history for any purpose. Minimizing makespan is also not the primary or sole objective.

While OS task schedulers do not use any history of run times of tasks for scheduling purpose, several algorithms use expected run-time of tasks/jobs (μ) for only predictive scheduling. None of these approaches can or use the standard deviation σ for reactive control of the execution of the tasks to minimize the makespan. Panwalkar and Iskander (1977) surveys over 100 rules for scheduling including priority-based rules and heuristic scheduling rules. Some of the rules described in this paper use processing time of jobs (for example, the SR rule selects job with the shortest processing time for execution). For repetitive jobs, the processing time can be average of the historical processing times and no use is made of standard deviation σ. Gupta and Ruiz-Torres (2001) describe three known bin-packing based heuristics and a new heuristic, called LISTFIT, for identical parallel-machine scheduling problem to minimize the makespan. Their LISTFIT heuristic is limited to predictively scheduling jobs. Laha and Behera (2017) provide a comprehensive review of the heuristics described in Gupta and Ruiz-Torres (2001) and describe experiments that shows LISTFIT heuristic to outperform the other heuristics for only predictive scheduling. Lee et al. (2006) propose using the LPT (Longest Processing Time) heuristics for generating a schedule and then using simulated annealing to improve on it in order to generate a near-optimal predictive solution. Cossari et al. (2012) describe an algorithm also for predictively scheduling independent jobs on identical parallel machines using the expected run-times of the jobs to minimize the normalized sum of square for workload deviations (NSSWD). Zhang et al. (2012) present a method called Load Balance Scheduling Algorithm (LBSA) for scheduling of real time periodic independent tasks with deadlines. LBSA assigns tasks to processors so that the loads are statically (predictively) balanced where load is the ratio of estimated run-time of a task (μ) to the period after which the task arrives again. Jain and Jain (2015) improves on LBSA but also to predictively ensure that the load on any processor does not differ from the average load on all the processors by a specified factor. Stavrinides and Kartaza (2018) present another predictive method of scheduling parallel independent jobs that can arrive at arbitrary times on a cluster of distributed processors and study the impact of routing and scheduling policies. Jobs are first routed to the processors using a random routing policy. Then the jobs assigned to a processor are scheduled using a scheduling policy. Two of the scheduling policies studied – First Come First Served, Approximate Shortest Cumulative Time First (ASCTF) and Approximate Periodic SCTF-x (APSCTF-x). - use estimated task service time (μ) for scheduling. The ASCTF policy orders tasks in processor queues by decreasing order of estimated service time (μ). APSCTF-x policy also remains static as the processor queues are not rearranged whenever a new task arrives, but periodically at every x interval. The rearrangement is done using ASCTF policy. Kápolnai et al. (2013) present a framework to schedule independent jobs, but whose execution time is not even known a priori, on identical parallel machines. As jobs are executed iteratively, their framework learns the execution time from history, creates and executes a new schedule and updates a historical database with the execution times. After each execution, the method improves the upper bound on the makespan but without using standard deviation σ. Thus, this method will work when the execution times of the jobs do not change significantly across multiple runs.

Unlike the OS task schedulers, all the above listed methods make use of only the expected run-time (μ) of the tasks to create a predictive un-ordered list of tasks for execution. All the above methods work well when the expected run-times of tasks are known accurately (i.e. σ is small) and the actual run-times are very close to the expected run-times. In actual practice, as in the closed-loop control application mentioned above, there are bound to be differences between the actual run-times and estimated run-times leading to imbalance in load across the parallel machines. None of the above studies attempt to monitor and dynamically improve on the schedule generated by the

heuristics to dynamically correct the imbalance. When the deviations are large, the actual makespan generated by the above methods will be significantly larger than expectation due to imbalances over the parallel machines.

There do exist several systems that implement dynamic or reactive load balancing. On multi-core architectures, the default OS task scheduler does dynamic load balancing to ensure that available cores are effectively used. However, OS task scheduling heuristics use only information such as cache affinity while balancing the load. They cannot use the run-time statistics of the tasks for resource allocation and load balancing purpose as no such history is captured, the tasks rarely being repetitive.

Special task schedulers, especially for heterogeneous computing environments, have been studied by several groups that primarily select the best processor for task execution. Gregg et al. (2011) and Jiménez et al. (2009) present method for dynamically scheduling applications but to exploit heterogeneous platforms (multi-core CPU and many-core GPU systems) in order to maximize overall throughput. Their method estimates when an application would finish execution on a given resource based on μ and assigns tasks to the best fit resource. However, they do not use standard deviation σ. Jooya et al. (2011) describe a history aware resource based dynamic (HARD) scheduler for scheduling tasks on heterogeneous CMPs (chip level multiprocessor) comprising of both high-complexity and low-complexity cores. HARD scheduler dynamically monitors resource utilization during execution of tasks and adaptively assigns demanding tasks to more complex and powerful cores and less demanding tasks to smaller and simpler cores in order to save power and improve performance. Bernardin and Lee (2006) describes a method of scheduling tasks in jobs on distributed computing system comprising of heterogeneous processors where all tasks in a job are assumed to have nearly the same amount of processing time. Their method keeps track of the run-time statistics of finished tasks in job. If a subsequent task in the same job takes longer to execute, a redundant instance of the task is launched for execution on a better processor. Chen et al. (2013) describe a History based Auto-Tuning (HAT) MapReduce scheduler for scheduling map and reduce tasks on heterogeneous systems. It uses history to detect slow tasks and launch backup of such tasks on other nodes. The goal of Bernardin and Lee (2006), and Chen et al. (2013) is to provide reliable scheduling on heterogeneous systems and not to minimize makespan using historical run-time statistics. Nanduri et al. (2011) use resource usage pattern of a job before its tasks are scheduled on a cloud. They avoid overloading any node in a cluster and utilize maximum resources on a particular node thereby decreasing race condition for resources. Chen et al. (2012) describe a workload-aware task scheduler for scheduling tasks on Asymmetric Muti-core Architectures (AMC) that uses history to allocate tasks to heterogeneous cores. Their scheduler uses preference-based task stealing as opposed to random task stealing to steal a “preferred” task from another core in order to balance workloads among different groups of heterogeneous cores. The task stealing mechanism does not use standard deviation σ and cannot minimize makespan when scheduling tasks that have significant differences from expected run-times.

While scheduling heterogeneous processors may not be relevant to closed loop control systems, the methods applied also do not take advantage of the more detailed statistics that are available in such environments.

No task scheduler was found that uses run-time statistics of tasks μ and σ for task allocation and then, if the actual runtimes deviate from μ leading to load imbalances across computing resources, dynamically improve the schedule using the statistics to minimize makespan.

3. Problem Description

The goal is to minimize makespan for the entire group of tasks in a closed loop control system (Figure 1) using their run-time statistics μ and σ. Such a system is specified as a set of *'n'* groups, $G = \{G_1, G_2, \dots, G_n\}$, where a group G_i is a set of N_i tasks, $\{T_1, T_2, \dots, T_{N_i}\}$. The groups have precedence, group G_i can be executed only after group G_{i-1} has completed its execution, $2 \leq i \leq n$. Let R be a set of $'m'$ identical parallel machines (computing resources) on which the n groups are to be scheduled. All tasks in a group arrive at the same time and are independent of each other and $N_i \gg m$. The group G of tasks, $G = \{G_i \,|1 \leq i \leq n\}$, are endlessly executed in a loop.

Let A_i^k denote the actual (i.e. the realized) makespan for k^{th}cycle, $k \geq 1$, of the group G_i. Then the makespan for the k^{th} cycle of G is $A^k = \sum_{i=1}^{n} A_i^k$. A strategy that minimizes A_i^k, for an arbitrary group G_i, will also minimize A^k.

The formulation reduces the problem of minimizing the makespan of a closed loop control system to minimizing the makespan of any one group for any arbitrary cycle. Hence, minimizing makespan of each group will minimize the total makespan, as all the tasks in a group should complete their execution before the tasks in the next group can run.

Let us consider a set of *'N'* tasks (jobs) in a single group $T = \{t_1, t_2, \dots, t_N\}$ with estimated processing times $\{\mu_1, \mu_2, \dots, \mu_N\}$ and standard deviations $\{\sigma_1, \sigma_2, \dots, \sigma_N\}$. Let ***P*** denote the predicted makespan of T on R using a heuristic that utilizes only $\mu's$ and not the $\sigma's$. Let ***A*** denote the actual (i.e. the realized) makespan while executing the plan generated by the heuristic. Typically, $A > P$ if $\sigma's$ are high. The objective is to determine a schedule for execution of the *N* tasks on *R* and dynamically improve on the schedule in order to minimize ***A***. It may be noted that traditional bin packing heuristics are used to predictively minimize the predicted makespan ***P*** [see, for example, Gupta and Ruiz-Torres (2001); Laha and Behera (2017); Lee et al. (2006)]. None of the earlier works attempt to reduce ***A***.

4. Dynamic Scheduling Approach

This section describes the novel customized scheduler (referred to as controller in the rest of this paper) for scheduling repetitive CPU intensive tasks using their historical run-time statistics on multiple homogeneous computing resources. The goal of the controller is to minimize ***A***. It uses both the estimated processing time(μ) and the standard deviation(σ) and is implemented as an ordinary user level task and not inside the OS Kernel. It controls the time and the resource on which a task will be released for execution to the OS and then probes the progress and, if necessary, adjusts the schedule as described below.

Using the Longest Processing Time (LPT) heuristics, the controller first assigns tasks to the resources in order to predictively balance the resources and thereby minimize ***P*** as described in Gupta and Ruiz-Torres (2001); Laha and Behera (2017). This assignment is based on the estimated run-times of the tasks(μ). The output at the end of this is an assignment of tasks for each resource. For resource *x*, let $\boldsymbol{Load}_x = \sum(\mu \; of \; tasks \; assigned \; to \; x)$. The predicted makespan is $\boldsymbol{P} = \max_{1 \le x \le m} (\boldsymbol{Load}_x)$.

$\boldsymbol{Load}_x$ and, therefore, ***P*** are independent of the order of tasks in the queue of resource *x*. The tasks assigned to each resource are sorted in decreasing order of a function f(μ, σ) (e.g. $f = \sigma, f = \frac{\sigma}{\mu}$, etc.). f is chosen such that tasks that have been historically problematic (e.g. tasks that have had large variations or large proportionate variations in run-times in the past) are placed near the head of the queue and get dispatched ahead of the more predictable ones so that a few reactive corrections (described below) are more effective. If necessary, the function f(μ, σ) may be written to incorporate additional domain specific knowledge for ordering of the tasks. The assigned tasks are then released one per resource optionally with an instruction to the OS to bind the released task to its assigned resource. This binding of task to resource is useful if other tasks (such as performance monitoring task, etc.) run on the same system on dedicated resources. Binding prevents the OS from migrating the tasks to another resource where such other tasks may be running. Binding of such other tasks will also prevent them from migrating to resources managed by the controller. Thus, binding operation prevents potential interference of execution of controller assigned tasks by such other tasks. When a task finishes its execution, the controller releases the next task on that resource's queue. During execution time, the controller checks the progress of tasks on each resource at predefined configurable times depending on the predicted makespan. Large variations in progress across resources will result in load imbalance where load of a resource, as defined previously, is the expected time when the currently running task and all the remaining tasks in the resource's queue will complete their execution. If the variation in progress across the resources is large, balancing the load across resources will help reduce the actual makespan ***A***.

Two approaches can be used for performing this dynamic load balancing –

(1) Remove all the tasks that are yet to run from their respective queues and then reassign them to the computing resources using the LPT heuristics. Consider the expected remaining time of currently running tasks while doing this re-assignment.
(2) Move tasks from the queue of resource whose progress is the slowest to the queue of the resource whose progress is the fastest to balance the load on the two resources. Repeat this step of task movement for the next slowest and the next fastest pair of queues and so on until all such pairs of queues is balanced.

Option# 2 was used in the controller due to its low cost. If n is the number of tasks that are yet to run, the cost of complete reassignment of tasks using option# 1 is $O(m * n)$ whereas the cost of migration of tasks in option# 2 is $\sim O(n)$. The reactive load balancing is effective because of the ordering of tasks in decreasing variability.
During this workload re-balancing operation, tasks that are already released for execution are not pre-empted and moved to another resource; only the tasks that are yet to run can be migrated. The probing and re-balancing by the controller has some overheads. To minimize this overhead, the controller does not probe too frequently. Instead, it probes at user-defined intervals.

5. Data Collection and Computational Experiments

The efficiency of the controller was empirically evaluated by measuring the makespan for 4 data-sets with varying workloads. This section describes the data corpus, the environment and the experimental setup.

5.1. Data corpus

The performance evaluation of the controller was done using data-sets generated from measurements of a real-life control system. This section describes how the data corpus was generated.

The CPU times consumed for dispatching trains on a small sub-network of Indian Railways by a simulator software developed at TCS Research & Innovation Labs were taken as a base for generating realistic data-sets for performance evaluation of the controller. The Indian Railway sub-network selected comprised of seven railway line sections. The railway line sections chosen were heterogeneous in nature with the smallest line spanning a distance of 52 km and the largest line spanning a distance of about 1350 km. All sections were double line sections with the exception of one, which was a mix of single line and double line. The traffic on these lines were also heterogeneous in nature – some have relatively low traffic and others high. The station infrastructure and the information of trains running on these lines were all taken from publicly available source.

Table 1: Data sets used for performance evaluation

Data-Set 1 (500 tasks)		**Data-Set 2 (500 tasks)**		**Data-Set 3 (500 tasks)**		**Data-Set 4 (1135 tasks)**	
#Tasks	**μ (milli-seconds)**	**#Tasks**	**μ (milli-seconds)**	**#Tasks**	**μ (milli-seconds)**	**#Tasks**	**μ (milli-seconds)**
182	U(10, 20)	182	U(10, 20)	182	U(10, 20)	415	U(10, 20)
100	U(20, 40)	100	U(20, 50)	100	U(20, 60)	228	U(20, 60)
102	U(40, 100)	102	U(50, 150)	102	U(60, 180)	233	U(60, 180)
86	U(100, 250)	86	U(150, 450)	86	U(180, 550)	193	U(180, 550)
30	U(250, 750)	30	U(450, 1350)	30	U(550, 1700)	66	U(550, 1700)

Table 1 lists four input data-sets of Group 1 tasks that were used for performance evaluation of the controller. The steps followed in arriving at Data-Set 1 in this table is described below.

(1) μ_{min}, the minimum run-time when there are no disturbances (i.e. no delays in any train), was measured for each of the seven line-schedulers on the test system. μ_{min} was co-related with the number of trains and stations on those lines. Similarly, μ_{max}, the run-time when a train was delayed near a junction, was measured for each of the seven line-schedulers.

(2) 30 railway line sections of different sizes and traffic mix on Indian Railways network were studied in detail. The number of trains and stations on those lines were determined from publicly available information. μ_{min} and μ_{max} were estimated for those 30 lines using the co-relation determined in step 1.

(3) Government of India (2015) groups the line sections in the Indian Railway network into 5 broad categories (ignoring the OTOS - one train only system category) based on the capacity utilization data. The 30-line sections studied were also divided in to five groups. μ_{min} of each subgroup was assumed to be the minimum of μ_{min} of all the lines of the subgroup. Similarly, μ_{max} of each subgroup was assumed to be maximum of μ_{max} of all the lines included in that subgroup.

(4) The number of lines on the entire Indian Railways is around 500. Therefore, to plan train movements in the entire Indian Railway network, 500 Line Scheduler tasks are needed in Group 1. The 500 lines were distributed into five subgroups in the same proportion as the number of sections in the five categories published in the

white paper by Government of India (2015). μ_{min} and μ_{max} for each subgroup was kept at the respective minimum and maximum estimated run-times identified for the sub-groups in step 3. Then, for each sub-group, the estimated run-time μ for individual tasks in the subgroup were arrived at by using uniform-distribution $U(\mu_{min}, \mu_{max})$.

Two more data-sets Data-Set 2 and Data-Set 3 were created by gradually increasing μ_{min} and/or μ_{max}. Another Data-Set 4 was created by significantly increasing the number of tasks in order to test the controller under stress. For each of the four data-sets listed in Table 1, 50 sets of input data were created by randomizing σ and the actual run-time (cpu-time). Actual run-time of a task is the time for which the task runs before terminating. σ of each task was kept at 1.5 * μ, randomized between -50% to $+50\%$. Gamma-distribution was used to generate the actual run-times of the tasks in order to get a skewed distribution with non-negative run-times. The actual run-times were shifted by $+\mu_{min}$.

To summarize, three pieces of information were generated for each task in the four data-sets – estimated run-time μ, standard deviation in run-time σ and the actual time for which the task will run. As described in the previous sections, the controller uses μ and σ for resource allocation and ordering of the tasks. The difference between actual run-time and μ creates load imbalances that triggers the controller to rebalance the load.

5.2. Experimental Setup

The single-threaded Controller was evaluated on a dedicated Intel® Xeon® server using the data corpus described above. The server configuration was as follows:

CPU E5-2650 v3 @ 2.30 GHz server
2 processors having 10 cores each
25 MB L3 Cache
64 GB RAM
OS: 3.10.0 (CentOS 7 distribution)
Compiler: GCC version 4.8.3

5.3. Environment

Performance evaluation of the controller was carried out by controlling the following two variables:
(1) The number of cores *m*. Five values of *m* were used in the experiments – 4, 8, 16, 19 and 20.
(2) The number of probes *'p'* to check the progress of execution. Five values of *p* were used in the experiments – 0, 2, 4, 6 and 8. *p* = 0 implies no probing and load balancing.

This set up gave us 5000 test cases for evaluating the controller:

*50 x 4 **(data-sets)** x 5 **(different number of cores)** x 5 **(different number probes)***

Tasks in resource queues were sorted in decreasing order of $f(\mu, \sigma) = \sigma$.

The following parameters were measured:
(1) Realized makespan with the scheduling left to the default OS scheduler. Here, all tasks were released simultaneously to the OS for execution. All tasks ran at the same priority and they were not bound to run on a specific core. This arrangement allows the OS to choose the core, the order of execution and time-slice for each task.
(2) Actual makespan A with the controller running in a reactive manner. Here the controller maintains a separate queue of tasks for each core.
(3) Actual makespan A with the controller running in a non-reactive manner in two modes:
   a. Controller maintaining a separate queue of tasks for each core like default Linux OS. This is essentially scheduling using LPT heuristics as described in Gupta and Ruiz-Torres (2001).
   b. Controller maintaining a single queue of tasks. Here the task at the head of the queue is released on the first core that becomes free.
(4) Cost of probes and dynamic re-balancing.

Parameters 1 and 2 enable us to evaluate the performance of the reactive controller with that of the default OS task scheduler that uses its own load balancing technique based on the busyness of cores. Parameters 2 and 3a enable us

to compare the performance of controller versus scheduling using plain LPT heuristics with no dynamic load balancing. Parameters 3b and 4 helps us to evaluate the benefits of dynamic load balancing.

The tasks used for testing were CPU intensive programs that consumed CPU cycles for a specified amount of time and then die. In all the experiments, the controller task was bound to run on the last core.

6. Results and Discussion

Table 2 gives the average value of actual makespan noticed for the four data- sets. In this table, *p* is the number of probes done during execution. During each probe, the controller checks the progress of tasks on all the cores and, if necessary, moves tasks across pairs of cores as described in section 4 to balance the load.

Table 2: Actual makespan for large σ

Data Set [A]	#Cores [B]	OS Scheduling [C]	Controller scheduling using a single Q [D]	Controller scheduling with probe done 'p' times [E]				
				p = 0	p = 2	p = 4	p = 6	p = 8
1	4	18.47	18.23	20.04	18.55	18.25	18.19	18.00
	8	10.51	10.55	11.78	10.13	9.85	9.87	9.77
	16	6.82	7.01	7.53	6.23	5.93	5.82	5.83
	19	6.33	6.47	6.84	5.74	5.54	5.44	5.41
	20	6.17	6.50	6.83	5.62	5.40	5.3	5.31
2	4	28.36	28.14	31.12	28.84	28.18	28.03	27.79
	8	16.32	16.45	18.18	15.84	15.45	15.15	15.08
	16	10.67	10.94	12.05	10.20	9.79	9.52	9.48
	19	9.90	10.12	10.82	9.18	8.87	8.74	8.67
	20	9.66	10.08	10.57	9.08	8.70	8.54	8.50
3	4	33.27	33.27	36.85	33.53	33.03	32.91	32.77
	8	19.50	19.37	22.10	18.98	18.40	18.25	18.17
	16	12.74	13.04	14.02	12.00	11.58	11.35	11.20
	19	11.89	12.11	13.09	10.96	10.47	10.35	10.22
	20	11.63	12.05	12.81	10.79	10.36	10.21	10.26
4	4	74.50	73.62	80.87	75.87	74.71	74.41	74.23
	8	40.49	39.94	45.81	40.78	39.97	39.31	39.21
	16	24.04	24.06	27.81	23.11	22.19	22.03	22.04
	19	21.63	21.64	25.45	21.16	20.43	20.29	20.08
	20	20.94	21.44	24.05	20.34	19.40	19.07	19.01

Figure 2 shows the percentage improvement in makespan when scheduled by the controller versus when the scheduling was done by the OS. In this figure, "No Rebal" indicates no reactive change in schedule was done by the controller, "Rebal X" indicates dynamic load balancing was done with number of probes = X, "Single Q" indicates the controller maintains a single queue of tasks for all the cores, as different from a queue for each core, ordered by decreasing μ with no reactive change in schedule.

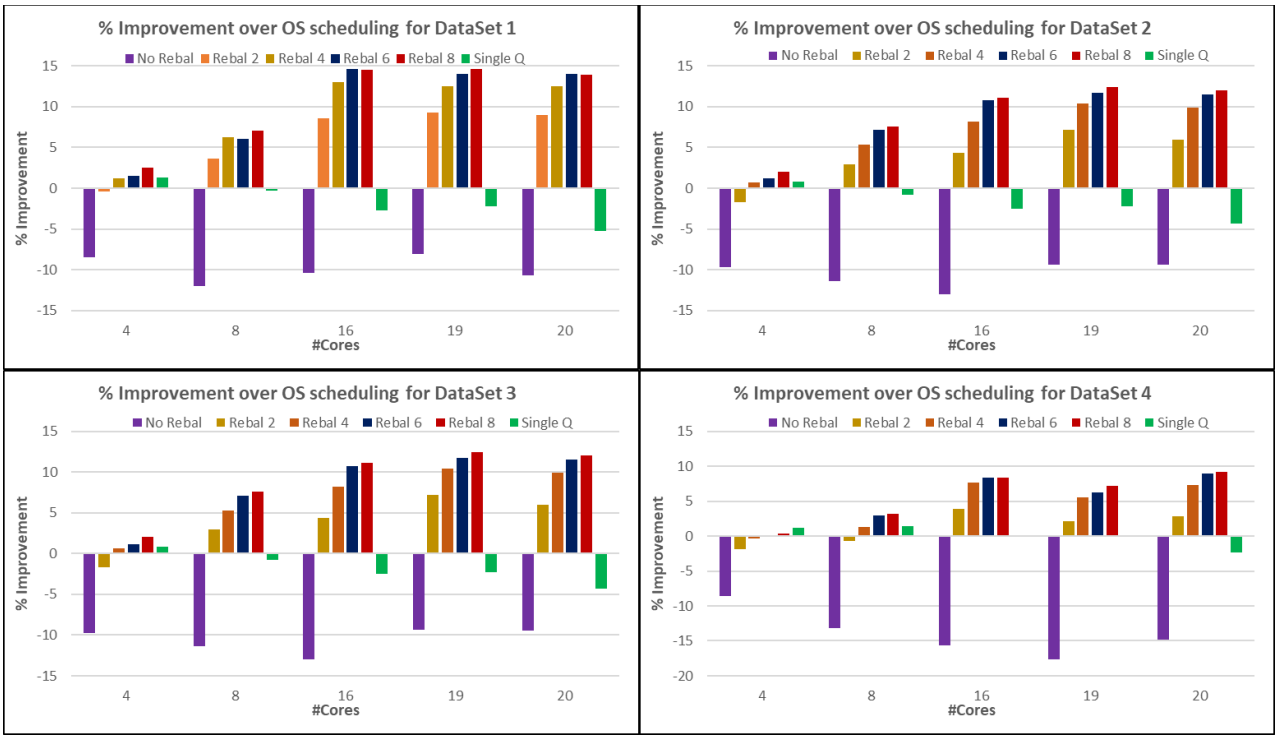

Figure 2: Percentage improvement in makespan

6.1. Effect of Load Balancing

From columns C and E ($p = 0$) in Table 2, scheduling by the OS always result in smaller makespans than the one by the controller operating in non-reactive mode (predictive only mode with no load balancing). This is due to better utilization of resources by the OS compared to that by the Controller using LPT heuristics. Due to high stochasticity, the actual run-times are significantly different from the estimated ones (μ). The plan predicted by the controller, therefore, will not be optimal. The OS does a better job in this case as it does dynamic load-balancing resulting in better resource utilization compared to that by the predictive scheduler.

Also from columns C and E in Table 2, probing and rebalancing by the controller result in better makespans than the default OS scheduler with 4 or more probes.

From columns D and E in Table 2, usage of a single queue of tasks (implying no load balancing) results in smaller makespan compared to that by reactive-scheduler probing less frequently when few resources are available (four cores). The performance of single queue suffers when there are a few of tasks with larger than expected actual-run times at the end of the queue. This creates a scenario where, towards the end of execution, several resources are free but a handful of long running tasks are left leading to unbalanced utilization of resources. With more number of resources, probing and dynamic load balancing gives better results. The results improve as the number of probes is increased.

Thus, probing and dynamic load balancing performs better than scheduling using heuristics and no dynamic corrections. Further, probing and load balancing based on μ and σ is more effective than that done by the OS task scheduler for CPU intensive tasks.

6.2. Overhead of Probing and Load Balancing

Table 3 summarizes the over-heads of probing by the controller, the resulting number of movement of tasks across cores by the controller and also the number of migrations done by the OS when the tasks were scheduled using the default OS scheduling. The average number of task migrations that happened with OS scheduling were determined by running the perf profiler tool.

From Table 3, the number of tasks moved increase with the number of probes. That is, the controller is effectively detecting and balancing the load thereby reducing the makespan. The overhead of probing and re-balancing by the controller is very small, less than 2 milli-seconds for data-sets 1, 2 and 3 and less than 5 milli-seconds for data-set 4. From Figure 2 it is noticed that the percentage improvement in makespan is less for data-set 4 as compared to that for other data-sets. This is because the number of tasks in data-set 4 and their cumulative run-times are significantly larger than those in the other data-sets but the maximum number of probes done by the controller was kept the same for all the data-sets. Since the cost of probes and load balancing is found to be extremely small, probing should be done more frequently for such long running tasks.

Table 3: Cost of re-balancing

Data Set	**#Cores (m)**	**Average Re-balance computation time in the controller (milli-seconds)**				**Average # task movements by the controller**				**Average # Migrations by the OS**
		p = 2	***p = 4***	***p = 6***	***p = 8***	***p = 2***	***p = 4***	***p = 6***	***p = 8***	
1	4	0.3	0.6	0.8	1.1	158	238	295	375	20
	8	0.3	0.6	0.9	1.3	189	314	402	579	25
	16	0.3	0.7	1.0	1.3	227	495	638	757	91
	19	0.3	0.7	1.0	1.3	223	489	638	841	64
	20	0.2	0.5	0.7	0.9	227	529	669	876	163
2	4	0.4	0.7	1.0	1.3	213	335	453	563	25
	8	0.4	0.7	1.1	1.4	251	415	583	781	44
	16	0.3	0.7	1.0	1.4	249	604	792	1033	123
	19	0.3	0.7	1.1	1.4	258	629	793	1077	89
	20	0.3	0.5	0.7	1.0	320	672	838	1199	193
3	4	0.4	0.7	1.0	1.4	225	386	481	629	27
	8	0.4	0.7	1.1	1.5	254	446	636	869	44
	16	0.3	0.7	1.1	1.4	262	641	852	1079	134
	19	0.4	0.7	1.1	1.5	301	667	864	1192	100
	20	0.2	0.5	0.7	1.0	335	693	918	1283	215
4	4	1.3	2.5	3.6	4.7	426	655	864	1045	52
	8	1.0	2.1	3.0	4.0	514	838	1096	1376	111
	16	0.9	1.7	2.5	3.4	578	1000	1305	1859	308
	19	0.8	1.7	2.5	3.4	591	1013	1542	1980	227
	20	0.6	1.1	1.7	2.2	627	1035	1618	2068	588

7. Conclusion

In this study, a single-threaded reactive scheduler (controller) for scheduling a group of repetitive CPU intensive tasks in closed loop control system was designed, implemented and evaluated on a multi-core server. The controller uses historical run-time statistics μ for allocation of tasks and $f(\mu, \sigma)$ for ordering the tasks to be able to better manage unexpectedly longer runtimes than expected. At pre-defined configurable intervals, the controller checks the progress on each resource and, if necessary, balances the load based on μ and reorders the tasks. None of the papers surveyed use this approach for scheduling repetitive tasks for minimizing the makespan. Performance of the controller was evaluated by measuring the makespan of a group of large number of tasks and comparing it with the elapsed time noticed with the default Linux CFS scheduler. The tasks used for performance evaluation were abstractions of a real-life closed loop control system. The makespan was measured with the controller running in reactive mode (reacting to load imbalances) and in predictive only mode (i.e., no load rebalancing, vanilla LPT heuristics).

When the deviations in run-times are significant, it was noticed that the re-active scheduler outperformed both the OS task scheduler and the predictive only scheduler when more computing resources were available. In the test

instances used by us, with 20 cores, 9% - 14% improvement in makespan over the default Linux task scheduler was noticed. The overhead of probing and adjusting the schedule was found to be low.

This paper improved on the heuristic in other cited work by including σ in ordering of tasks whose run-times are stochastic, dynamically probing the progress of tasks on all computing resources and improving the schedule by doing load-balancing and reordering of tasks based on μ and σ. In the experiments, a fixed number of probes were used to demonstrate the impact of probing and load balancing. However, in a real-life application, the controller need not use a fixed number of probes; it may dynamically auto tune the number of probes depending on load imbalances measured across cycles. This work demonstrates that repeated probing, load-balancing and reordering of tasks bases on historical run-time statistics (μ and σ) results in significant reduction in realized makespan over those achieved by heuristics and traditional OS task scheduler. Deployment of such customized low overhead scheduler in a large control system will allow the system to be run on computers with fewer resources thereby reducing the cost of computing hardware required to run the systems in a redundant manner or can reduce the cycle time.

References

Bernardin, J., and Lee, P., Distributed Computing, *United States Patent,* No. US 2006/0195508 A1, 2006.

Chen, Q., Chen, Y., Huang, Z., and Guo, M., WATS: Workload-aware task scheduling in asymmetric multi-core architectures, *IEEE 26th International Parallel and Distributed Processing Symposium*, May 21 – 25, 2012, pp. 249-260, 2012.

Chen, Q., Guo, M., Deng, Q., Zheng, L., Guo, S., and Shen, Y., HAT: History-based auto-tuning MapReduce in heterogeneous environments, *The Journal of Supercomputing*, vol. *64,* no. 3, pp. 1038-1054, 2013.

Cossari, A., Ho, J. C., Paletta, G., and Ruiz-Torres, A. J., A new heuristic for workload balancing on identical parallel machines and a statistical perspective on the workload balancing criteria, *Computers & Operations Research*, vol. 39, no. 7, pp. 1382-1393, 2012.

Government of India, Ministry of Railways, NEW DELHI. Indian Railways Life of the nation: A White Paper, 2015. Accessed 30 November 2020. http://www.indianrailways.gov.in/railwayboard/uploads/directorate/finance_budget/Budget_2015-16/White_Paper-_English.pdf.

Gregg, C., Boyer, M., Hazelwood, K., and Skadron, K., Dynamic heterogeneous scheduling decisions using historical runtime data, *Workshop on Applications for Multi-and Many-Core Processors* (A4MMC), San Jose, USA, pp. 1-12, June 04, 2011.

Gupta, J. N., and Ruiz-Torres, A. J., A LISTFIT heuristic for minimizing makespan on identical parallel machines, *Production Planning & Control*, vol. 12, no. 1, pp. 28-36, 2001.

Jain, D., and Jain, S., Load balancing real-time periodic task scheduling algorithm for multiprocessor environment. *International Conference on Circuits, Power and Computing Technologies* [ICCPCT-2015], Nagercoil, India, March 19-20, 2015, pp. 1-5, 2015.

Jiménez, V. J., Vilanova, L., Gelado, I., Gil, M., Fursin, G., and Navarro, N., Predictive runtime code scheduling for heterogeneous architectures*, 4th International Conference on High-Performance Embedded Architectures and Compilers*, Paphos, Cyprus, January 25 – 28, 2009, pp. 19 – 33, 2009.

Jooya, A. Z., Baniasadi, A., and Analoui, M., History-aware, resource-based dynamic scheduling for heterogeneous multi-core processors, *IET computers & digital techniques*, vol. 5, no. 4, pp. 254-262, 2011.

Kápolnai, R., Szeberényi, I., and Goldschmidt, B., Approximation of repeated scheduling chains of independent jobs of unknown length based on historical data, *Recent Advances in Computer Science*. WSEAS Press, pp. 41-46, 2013.

Kobus, J., and Szklarski, R., Completely Fair Scheduler and its tuning, *draft on Internet*, 2009. Accessed 30 November 2020. https://fizyka.umk.pl/~jkob/_downloads/cfs-tuning.pdf.

Laha, D., and Behera, D. K., A comprehensive review and evaluation of LPT, MULTIFIT, COMBINE and LISTFIT for scheduling identical parallel machines*, International Journal of Information and Communication Technology*, vol. 11, no. 2, pp. 151-165, 2017.

Lee, W. C., Wu, C. C., and Chen, P., A simulated annealing approach to makespan minimization on identical parallel machines, *The International Journal of Advanced Manufacturing Technology*, vol. 31, pp 328-334, 2006.

Nanduri, R., Maheshwari, N., Reddyraja, A., and Varma, V., Job aware scheduling algorithm for mapreduce framework, *IEEE 3rd International Conference on Cloud Computing Technology and Science*, Athens, Greece, Nov 29 – Dec 01, 2011, pp. 724-729, 2011.

Panwalkar, S. S., and Iskander, W., A survey of scheduling rules, *Operations research*, vol. 25, no. 1, pp. 45-61, 1977.

SenGupta, S., Joshi, S., Salsingikar, S., Sinha, S. K., Dontas, K., and Agrawal, N., System and method for generating vehicle movement plans in a large railway network, *United States Patent*, No. US 2015/9,381,928 B2, 2015.

Sinha, S. K., Salsingikar, S., and SenGupta, S., An iterative bi-level hierarchical approach for train scheduling, *Journal of Rail Transport Planning & Management,* vol. 6, no.3, pp. 183-199, 2016.

Stavrinides, G. L., and Karatza, H. D., Task group scheduling in distributed systems, *IEEE International Conference on Computer, Information and Telecommunication Systems* (CITS), Colmar, France, July 11-13, 2018, pp. 1-5, 2018.

Wong, C. S., Tan, I., Kumari, R. D., and Wey, F., Towards achieving fairness in the Linux scheduler, *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 34-43, 2008.

Zhang, K., Qi, B., Jiang, Q., and Tang, L., Real-time periodic task scheduling considering load-balance in multiprocessor environment, *Proceedings of the 3rd IEEE International Conference on Network Infrastructure and Digital Content*, Beijing, China, September 21-23, 2012, pp. 247-250, 2012.

Biographies

Sudhir Shetiya completed his Masters in Computer Science and Engineering from IIT Bombay in 1992. He is a Researcher and a Solution Architect at TCS Research and Innovation. He has been with TCS since 1995 and has worked on system software and operating systems internals with focus on scheduling and memory management issues in fault-tolerant systems. He is associated with research programs involving planning and rescheduling of trains on large railway networks and profitable last mile delivery problems since 2016.

Siddhartha SenGupta is an Advisor to the Chief Technology Officer of TCS and also an Adjunct Professor in Industrial Engineering and Operations Research, an Interdisciplinary Program at Indian Institute of Technology, Bombay. He had been with Tata Consultancy Services Ltd from 1982. From 1985 he led applied research into AI and Planning, retiring in 2011 as head of the Decision Sciences and Algorithms lab. Dr SenGupta has done his MSc in Physics and Ph. D. in Space Physics from Indian Institute of Technology, Kharagpur. He has published papers and has been awarded patents in automation of large-scale planning & scheduling in Railways and Shipping, Retail Manpower and Pricing, Supply Chain, Enterprise Architecture and Enterprise Engineering, Expert Systems & Knowledge Based Systems, etc. He is a member of ORSI and has been in the Senate or on the Board of several educational institutes in India.

Supratim Biswas is an Emeritus Fellow in the Department of Computer Science and Engineering (CSE) at IIT Bombay (IITB). He completed his Ph. D. in Compiler Optimization from IIT Kharagpur in 1981. He retired as a Professor from IITB in 2018. His primary research interest is in the area of programming language design and implementation and Optimizing Compilers and Parallelizing Compilers in particular. He has several publications in International Journals and conferences and has guided many Ph. D. theses, M. Tech dissertations and B. Tech. projects. He was a visiting faculty at the Center for Supercomputing Research and Development at UIUC during 1992 - 1993. He was awarded the Excellence in Teaching Award at IIT Bombay in 2000. He has held various functionary positions at IITB, such as, Organizing Vice-Chairman of JEE 1997, Head of the CSE (2000 -2003), Dean of Academic Programs (2007 - 2010) and Member Board of Governors (2011-2014).