AnthillSched: A Scheduling Strategy for Irregular and Iterative I/O-Intensive Parallel Jobs

Luís Fabrício Góes, Pedro Guerra, Bruno Coutinho, Leonardo Rocha,
Wagner Meira, Renato Ferreira, Dorgival Guedes
Universidade Federal de Minas Gerais, Belo Horizonte, MG,Brasil
{lfwgoes,pcalais,coutinho,lcrocha,meira,renato,dorgival}@dcc.ufmg.br

Walfredo Cirne
Universidade Federal de Campina Grande, Campina Grande, PB
walfredo@dsc.ufcg.edu.br

Abstract

Irregular and iterative I/O-intensive jobs need a different approach from parallel job schedulers. The focus is not only the processing requirements anymore. Memory, network and storage capacity must be considered in a job scheduling decision. Jobs execution are irregular and data dependent, alternating between CPU-bound and I/O-bound phases. So, in this paper, we propose and implement a parallel job scheduling strategy, called AnthillSched, based on a simple heuristic. To test and verify AnthillSched scheduling strategy, we used data mining jobs and logs obtained from a real system. Our ***main contributions*** *are: the implementation of a parallel job scheduling strategy called AnthillSched in a real system; a performance analysis of AnthillSched, which discarded some other alternative solutions.*

1. Introduction

Increasing processing power, network bandwidth, main memory, and disk capacity has been enabling efficient and scalable parallelizations of a wide class of applications that include data mining [10, 21], scientific visualization [5, 17], and simulation [18]. These applications are not only demanding in terms of system resources, but also a parallelization challenge, since they are usually irregular, I/O-intensive, and iterative. We refer to them as I^3 applications or jobs. As irregular jobs, their execution time is not really predictable, and pure analytical cost models are not accurate. The fact that they are I/O intensive make them even less predictables, since their performance is significantly affected by the system components and by the amount of overlap between computation and communication that is achieved during the job execution. Further, I^3-jobs perform computations spanning through several domains, not only consuming data from those domains, but also generating data and increasing the volume of data to be handled in real time. Finally, iterativeness raises two issues that affect the parallelization: locality of reference and degree of parallelism. The locality of reference is affected because of the access pattern of iterations across time. The degree of parallelism is a function of the dependencies among data across iterations and thus among iterations. As a consequence of these characteristics, scheduling of I^3-jobs is quite a challenge, and determining optimal scheduling for I^3-jobs seems to be very complex, since it should consider locality, input size, data dependences, and parallelization opportunities.

Generally, parallel job schedulers are designed to deal with CPU-intensive jobs [4, 6, 7, 8, 13, 14, 15, 16, 20, 23, 25]. Some researches have proposed strategies to deal with I/O-intensive and irregular jobs [2, 3, 17, 18, 19, 22, 24], but not with I^3-jobs.

In this paper we investigate the scheduling of I^3-jobs, in particular filter labeled stream programs [5]. These programs structure are composed of filters that communicate using labeled streams, which guarantee consistent addressing among filters. These programs are fundamentally asynchronous and implemented using a event-based paradigm. In the scope of this work, a job is the execution of a program on a specific input data and execution parameters using a number of filter instances for each filter. The main issue is that each filter demands a different amount of CPU and I/O, and, in order to be efficient, they require a continuous and balanced dataflow between filters. Our premise is that the balance among the dataflow between filters may be achieved by scheduling the proper number of filter copies or instances. In this paper we propose, implement, and evalu-

ate a parallel job scheduling strategy called AnthillSched, which determines and schedules the number of filter instances according to each filter's CPU and I/O demand. We evaluate AnthillSched using logs based on real workloads submitted to the Tamanduá system, which is a data mining service platform that executes data mining I^3-jobs.

This paper is organized as follows. We present the related work in Section 2. The following section introduces the programming framework Anthill, and Section 4 describes our proposed scheduling strategy. We then present the workload, metrics, experimental design, results and the performance analysis of AnthillSched in the following sections. Finally, we present our conclusions and future works in Section 6.

2. Related Work

This section presents other research related to the scheduling of parallel I/O-intensive jobs. Wiseman et.al [22] presented a Paired-Gang Scheduling, in which I/O-intensive and CPU-intensive jobs share the same time slot. Thus, when an I/O-intensive job waits for an I/O request, the CPU-intensive job uses the CPU, increasing utilization. This approach points that processor sharing is a good mechanism to increase performance.

Another work [24] shows three versions of an I/O-Aware Gang Scheduling (IOGS) strategy. The first, for each job, looks for the row in the Ousterhout Matrix (time-space matrix) with the least number of free slots where jobs file nodes are available, considering a serverless file system. This approach is not efficient for workloads with lower I/O intensity. The second version, called Adaptive-IOGS, uses the IOGS, otherwise it tries the traditional gang scheduling approach. It fails to deal with high I/O-intensive workloads. The last version, called Migration-Adaptive IOGS, includes the migration of jobs to their file nodes during execution. This strategy outperformed all the other ones.

A job scheduling strategy for data mining applications to a cluster/grid is proposed in [19]. It groups independent tasks that use the same data to form a big job and schedules it to the same group of processors. Thus, the amount of transferred data is reduced and the jobs performance is increased.

Storagge Affinitty is a job scheduling strategy that exploits temporal and spatial data locality for bag-of-tasks jobs [18]. It schedules jobs near to their data according to the storage affinity metric (distance from data) and also uses task replication when necessary. It has presented better performance than XSufferage (apriori informed) and WQR (non-informed).

Finally, a very close work is LPSched, a job scheduling strategy that deals with asynchronous dataflow I/O-intensive jobs using linear programming [17]. It assumes that jobs information is available a priori and run-time monitors of cluster/grid resources. It maximizes the dataflow between tasks and minimizes the number of processors used per job. AnthillSched differs from LPSched in many points: it supports labeled streams, iterative dataflow communication; it uses a simple heuristic and does not use run-time monitors.

3. The Programming Environment

DataCutter is a middleware that enables efficient application execution on distributed heterogeneous environments. It was designed based on the Filter-Stream programming model [1], in which the applications are partitioned into a set of Filters that communicate using Streams [5]. Datacutter allows the instantiation of several copies of each filter (transparent copies) at runtime so that the application can balance the different computation demands of different filters as well as achieve high performance. The stream abstraction maintains the illusion of point-to-point communication across filters, and when a particular copy outputs data to the stream, the middleware takes care of delivering the data to one of the transparent copies on the other end. Broadcast is possible in Datacutter, but selecting a particular copy is tricky. Datacutter implements automatic selection mechanisms based on round-robin, or demand driven models.

We extend such middleware in Anthill by providing a mechanism called labelled stream which allows the selection of a particular copy based on data related to the messages (the labels). Such extension allows for a richer programming environment, making it easier for transparent copies to partition their global state. Besides that, Anthill provides a task framework, in which the application execution can be accomplished in a succession of intermediate tasks that need to be performed, and that can spawn multiple filters in an iterative environment. It explores parallelism in two levels: time, space, as well as it makes it easy to exploit asynchrony.

As we see in Figure 1, a job in Anthill explores time parallelism like a pipeline, because it is composed of N filters (processing phases or stages) connected with streams (communication channels). This job model explicitly forces the programmer to divide the job in well defined phases (filters), in which an input data is transformed, by a filter, to another data domain that is required to the next filter.

Anthill programming model also explores space parallelism as each filter can have multiple copies or instances. Each communication channel between filters can be labeled to direct a dataflow to a specific filter copy (point-to-point, ex: filter 1 to filter L) or to all filter copies of the next filter level (broadcast, ex: filter L to filter N). A consequence of space parallelism is data parallelism, because a database

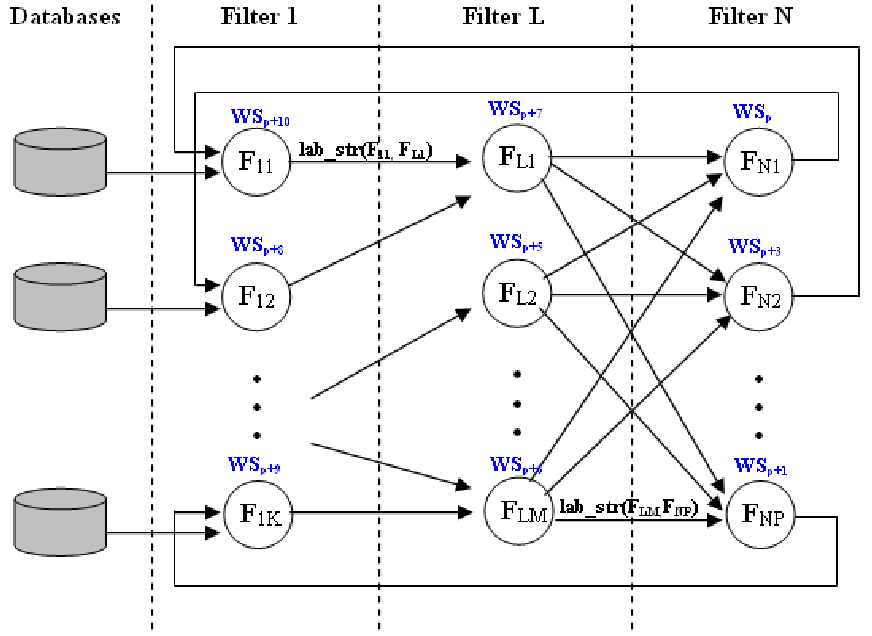


Figure 1. Anthill programming model.

is automatically partitioned among filter copies. Together with streams, data parallelism provides an efficient mechanism to divide I/O demand among filters.

Moreover, Anthill takes advantage of asynchrony and provides iteractiveness. Each job has a set of work slices (WS) to be executed. According to data sets read from databases, the WS are created. They can be executed independently, respecting only a data dependency graph, where a WS_p is the former WS executed. A WS also can generate other WS, this fact justifies the need of an iterative process.

4. Anthill Scheduler

It should be noted that Anthill's programming model deals with only qualitative aspects of I^3-jobs. As we presented, Anthill allows asynchrony, space parallelism, iterativeness, but it does not deal with quantitative aspects such as the number of filter copies, number of transmitted bytes during an iteration etc. Thus, to deal with quantitative aspects, we need a job scheduling strategy that can be translated in the calculation of the number of filter copies, considering filter execution time, filter I/O demand, data complexity etc. It is important to notice that the overall application performance is very dependent on the schedule of the filters.

We propose AnthillSched, a parallel job scheduling strategy, implemented as the job scheduler of Anthill. It focuses on the proper scheduling of a I^3-job among processors, that is, the number of copies for each filter, according to job input parameters (minimum node size, support, attribute etc). There are two possible scheduling alternatives: analytical modeling that is very complex and can be infeasible to our problem or a less complex approach as an experimental heuristic.

Our approach is based on a simple experimental heuristic to solve a very complex problem in a efficient way. We based this on the fact that the applications have complex interactions, for the processing is iterative and the applications themselves are interactive. Going for a full analytical model may be very complicated. With this simpler scheduling strategy however, we still expect to eliminate possible bottlenecks and provide a continuous dataflow with high asynchrony to a I^3-job.

AnthillSched requires q controlled executions, one for each permutation of the input parameters. For example, if we have input parameters A and B, and each parameter can assume 10 different values, we have 100 possible permutations. A controlled execution is the execution of a job with one copy per filter (sequential pipeline) with certain input parameters. For each copy, we collect the number of input bytes B_{ij} and the execution time E_{ij}, where i is the permutation of input parameters for a job and j is the filter identifier.

In Anthill, each job is executed according to a FCFS strategy with exclusive access to processors. When a new

job arrives, Anthill calls AnthillSched with the permutation of input parameters i for the job and the number of available processors p, and outputs the number of filter copies C_i, after m iterations. First, for each iteration, the number of copies for each filter C_{ij} is calculated according to Fig. 2, where n is the number of filters. In this step, we normalize the number of input bytes B_{ij} and the execution time E_{ij} dividing by the sum of B_{ik} and E_{ik}. Then, it sums the normalized values and divide by two, in order to obtain the percentage of each filter, which represents the amount of resources that a filter consumes comparing to the other ones. If we have a job with 3 filters, we can find that filter1, filter2 and filter3, respectively, utilize 0.6, 0.2 and 0.2 of the total of resources to execute the job. Finally, according to the number of available processors p, it calculates the number of filter copies C_{ij} proportionally to the percentage of each filter.

The second step handles broadcast operations, since when a broadcast occurs between two filters, the number of input bytes of the destination filter will increase according to it's number of copies. So, when a broadcast occurs, the number of input bytes received by the destination filter $S_{i((j+1)mod(n))}$ is multiplied by its number of copies $C_{i((j+1)mod(n))}$. Thus, AnthillSched must recalculate the number of input bytes B_{ij} and consequently all the process to find the number of copies for each filter, until a certain number of iterations m.

If we have a large number of permutations, its infeasible to run all controlled executions and store them. So, a solution would be consider only a sampling of the possible permutations. When a new, or not considered, permutation of input parameters of a job arrives, an interpolation between the two nearest permutations can approximate the number of copies for each filter in the new job.

For each new submitted job, Anthill calls AnthillSched with the job's permutation of input parameters. The scheduling process overhead is negligible, because AnthillSched's scheduling heuristic is very simple and can be solved in polynomial time as we see in Figure 2.

During to preliminary tests, we verified that controlled executions that spend less than 5 seconds do not need to be parallelized. This threshold can vary according to the jobs and input data, but as a general rule, short sequential jobs do not need to be parallelized to improve performance. Thus, we created an optimized version of AnthillSched that determines if a certain job must execute in parallel (more than one copy per filter). Otherwise, it executes a sequential version of the job. We called this version as Optimized AnthillSched (OAHS).

function AnthillSched (i, p : integer) : array of integer
  for 1 **to** m **do**
    for j = 1 **to** n **do**

$$C_{ij} = p \times \frac{\left(\frac{B_{ij}}{\sum_{k=1}^{n} B_{ik}} + \frac{E_{ij}}{\sum_{k=1}^{n} E_{ik}} \right)}{2}$$

    endfor;

    for j = 1 **to** n **do**
      if$(broadcast(S_{ij}, S_{i((j+1)mod(n))}))$
        $B_{i((j+1)mod(n))} = B_{i((j+1)mod(n))} \times C_{i((j+1)mod(n))}$
    endfor;
  endfor;
  return C_i
end.

Figure 2. AnthillSched's algorithm.

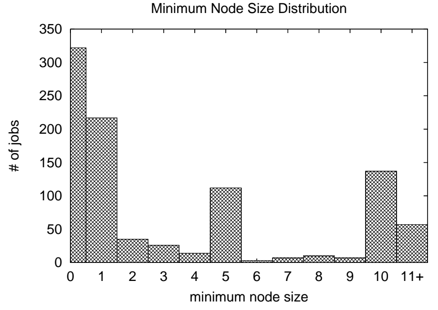
5. Results

In this section we evaluate our scheduling strategy by applying it to a data mining application: the ID3 algorithm for building decision trees. More specifically, we want to investigate whether the number of filter copies C_i for a I^3 job depends equally to the number of input bytes B_{ij} and execution time E_{ij} of each filter j. Thus, if the number of each filter copies C_i is proportionally distributed according to B_{ij} and E_{ij}, we eliminate possible bottlenecks and provide a continuous dataflow with high asynchrony to a job.

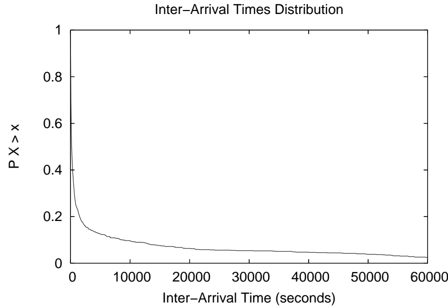
To test and analyze our hypothesis, we compared two versions of AnthillSched (non-optimized and optimized) to other two job scheduling strategies: *Balanced Strategy* (BS) and *All Strategy*(AS). The proposed strategies use the maximum number of processors available. The BS tries to balance the number of filter copies among processors, considering that each filter has an equal percentage. For example, if we have a job with 3 filters and a cluster of 15 processors, each filter will have 5 copies. In AS, all filters have a copy on each processor executing concurrently.

5.1. Experimental Setup

As the workload, we used data mining jobs and real logs obtained from Tamanduá platform. It is a scalable and service-oriented dataming platform that uses efficient algorithms for clusters with large databases. Nowadays, Tamanduá is used for mining government databases. Today, there are various data mining techniques implemented in Anthill as Apriori, K-Means etc. In our experiments, we are concerned about ID3 (Decision Tree) technique [10]. The main input parameter that influences in ID3 is the minimum node size, which determines the minimum number of homogeneous points to create a node.



(a) Minimum node size histogram.



(b) Inter-arrival time inverse cumulative distribution.

Figure 3. Workload characterization from Tamanduá.

Based on real logs from Tamanduá, we characterized the inter-arrival time between jobs in Fig. 3(a) and the minimum node size pattern all jobs in Fig. 3(b). As we see in Fig. 3(a), the majority of minimum node size values are concentrated between 0 and 10. As the minimum node size is inversely proportional to the execution time, it signifies that long-running jobs are predominant over short jobs. Based on the characterization of Tamanduá logs, we created a workload model that uses the inter-arrival time pattern between jobs and the minimum node size pattern. As we see in Fig. 3(b), the inter-arrival time fits in a exponential distribution with parameter $\lambda = 0.00015352$, with chi-square test value equal to 0. The minimum node size can fit in a Pareto distribution with parameters θ = 0.61815 and a = 0.00075019, where θ is the continuous shape parameter and a is the continuous scale parameter (Fig. 3(a)). Using a workload generator, we varied the random seed and generated 10 workloads. Each workload is composed of 1000 jobs, all jobs are implementations of ID3 technique. Each job is characterized by a submission time and a minimum node size value.

To test the scheduling strategies under different conditions, we varied the load (job arrival rate) in low, medium and maximum. The low load consider the inter-arrival time between jobs for all points shown in 3(b), so it has long periods of inactivity and a few peak periods, in which the inter-arrival time between jobs is small. To create a medium load workload, we used only the inter-arrival time a subset composed of the peak periods found in 3(b). Finally, the maximum load assumes that all jobs of a workload arrived at same time. So, in this case, we do not consider the inter-arrival time between jobs.

$$WorkloadExecTime = \sum_{i=1}^{n} JobExecTime_i \quad (1)$$

$$WorkloadIdleTime = TotalTime - \sum_{i=1}^{n} JobExecTime_i \quad (2)$$

$$MeanJobsWaitTime = \sum_{i=1}^{n} \frac{JobWaitTime_i}{NumberOfJobs} \quad (3)$$

$$MeanJobsRespTime = \sum_{i=1}^{n} \frac{JobWaitTime_i + JobExecTime_i}{NumberOfJobs} \quad (4)$$

$$MeanJobsSlowdown = \sum_{i=1}^{n} \frac{\frac{JobRespTime_i}{JobExecTime_i}}{NumberOfJobs} \quad (5)$$

To evaluate our proposal, we used 5 performance metrics: workload execution time (Eq. 1), workload idle time (Eq. 2), mean jobs response time (Eq. 4), mean jobs wait time (Eq. 3) and mean jobs slowdown (Eq. 5). As the parallel computer, we used a Linux-cluster composed of 16 nodes with 3.0GHz Pentium 4 processors, 1 GB main memories and 120 GB secondary memories, interconnected by a Fast Ethernet Switch.

5.2. Experimental Results

In this section, we present some experimental results in order to evaluate the effectiveness of the scheduling strategies discussed. More specifically, we evaluate how well the scheduling strategies work when the system is submitted to varying workload and number of processors.

In order to evaluate the impact of the variability of the workload on the effectiveness of the strategies, we increased the load on each experiment to test which scheduling strategy presents a better performance to each situation and which strategies are impossible to use in practice. In the first three experiments (low, medium and maximum load), we used a cluster configuration composed of only 8 processors. With a maximum load, we saturate the system to test the alternatives. In our final experiment (scalability under a maximum load), we compare the two better strategies with the

Strategy	Average	Min	Max	Std. Dev	c_1	c_2
AS	2065488.41	2029272.33	2155312.68	36921.23	2042604.83	2088371.99
BS	2065463.77	2029279.84	2155222.41	36903.64	2042591.09	2088336.45
NOAHS	2065464.68	2029245.12	2155317.20	36930.32	2042575.47	2088353.90
OAHS	2065410.48	2029242.96	2155136.99	36896.76	2042542.06	2088278.89

(a) Execution time for each strategy under a low load.

Strategy	Average	Min	Max	Std. Dev	c_1	c_2
AS	1189.08	61.86	3980.32	1309.36	377.54	2000.61
BS	1214.08	57.10	4070.59	1321.27	395.17	2033.00
NOAHS	1209.70	90.26	3975.80	1304.82	400.98	2018.43
OAHS	1263.55	90.19	4156.01	1328.39	440.22	2086.88

(b) Idle time for each strategy under a low load.

Strategy	Average	Min	Max	Std. Dev	c_1	c_2
AS	9.61	7.07	11.82	1.49	8.69	10.53
BS	7.87	5.77	9.32	1.15	7.16	8.59
NOAHS	8.13	6.01	10.24	1.33	7.30	8.95
OAHS	4.84	3.64	5.89	067	4.43	5.25

(c) Mean jobs wait time for each strategy under a low load.

Strategy	Average	Min	Max	Std. Dev	c_1	c_2
AS	189.28	186.76	191.46	1.50	188.35	190.21
BS	162.16	158.03	164.50	2.04	160.89	163.42
NOAHS	168.28	166.28	170.67	1.36	167.44	169.44
OAHS	107.82	101.68	110.65	2.63	106.45	109.19

(d) Mean jobs response time for each strategy under a low load.

Table 1. Scheduling strategies performance for different workloads under a low load.

same optimizations and analyze the scalability of the strategies for different cluster configurations (8, 12 and 16 processors). We used a 0.95 confidence level and approximate visual tests to compare all alternatives. The confidence intervals are represented by c_1 (lower bound) and c_2 (upper bound).

This first experiment tests the scheduling strategies under a low load for a cluster with 8 processors. As we see in Table 1(a), the mean execution time for all workloads and strategies was too close. A low load implies in large inter-arrival times between jobs, in this case, they were larger than the time necessary to execute a job. Thus, if a scheduling strategy spends more time than another, for a low load, it does not matter. In despite of it, we observe in Table 1(b) that system using Optimized AnthillSched (OAHS) spent more time idle than the other ones. This is a first indication that jobs executed with OAHS strategy have a lower response time, as we confirm in Table 1(d). When a job spends less time executing, as the inter-arrival time is long, the system stays idle for more time, waiting for a new job submission, than a system in which a job spends more time executing. According to an approximate visual test of the confidence intervals, table 1 (c) shows that the mean jobs wait time, and consequently the mean jobs response time for OAHS, is really lower than the other strategies. As a job starts its execution, it cannot be preempted, so the difference between two executions of the same job is only on its wait time.

As our first conclusions, this experiment shows that for a low load, independent of scheduling strategy, the inter-arrival time between jobs prevails over the workload execution time, because jobs are shorter than it. As we expected, a scheduling strategy that reduces the mean jobs wait time and consequently the response time, increases the idle time of the system.

Under a medium load, the obtained results showed in Table 2 are clearer. A medium load avoids that the inter-arrival time prevails over response time (be larger than). In Table 2(a), AS presented the worst execution time for all workloads. While, BS and Non Optimized AnthillSched

Strategy	Average	Min	Max	Std. Dev	c_1	c_2
AS	179681.76	179401.58	179807.61	106.49	179615.75	179747.77
BS	154321.87	150995.24	155923.21	1394.01	153457.87	155185.87
NOAHS	160181.04	159284.49	161230.22	548.70	159840.95	160521.12
OAHS	103100.88	97447.90	106026.05	2413.90	101604.76	104597.01

(a) Execution time for each strategy under a medium load.

Strategy	Average	Min	Max	Std. Dev	c_1	c_2
AS	11.40	0.00	86.56	27.22	-5.48	28.27
BS	10.42	0.00	84.74	26.52	-6.02	26.86
NOAHS	21.01	0.00	120.53	39.66	-3.57	45.60
OAHS	32.59	0.00	119.77	48.37	2.61	62.57

(b) Idle time for each strategy under a medium load.

Strategy	Average	Min	Max	Std. Dev	c_1	c_2
AS	56660.14	53618.32	58737.13	1442.26	55766.23	57554.05
BS	44165.69	41342.79	46325.72	1612.33	43166.37	45165.00
NOAHS	46849.72	43929.01	49249.10	1449.23	45951.49	47747.95
OAHS	18721.51	15228.08	21785.22	2032.29	17461.90	19981.11

(c) Mean jobs wait time for each strategy under a medium load.

Strategy	Average	Min	Max	Std. Dev	c_1	c_2
AS	56839.81	53798.01	58916.78	1442.28	55945.88	57733.73
BS	44319.97	41497.70	46481.64	1613.01	43320.23	45319.71
NOAHS	47009.87	44089.37	49409.53	1449.21	46111.66	47908.09
OAHS	18824.49	15325.52	21891.15	2033.92	17563.88	20085.10

(d) Mean jobs response time for each strategy under a medium load.

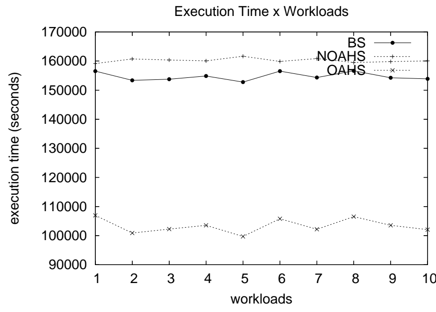
Table 2. Scheduling strategies performance for different workloads under a medium load.

(NOAHS) presented similar performance, with a little advantage for BS. Table 2(b) shows that on average, OAHS achieved the higher idle time. As we confirm in Table 2(c,d), the mean jobs wait and response time are lower when the OAHS is used, so the system have more idle time waiting for another job arrival.

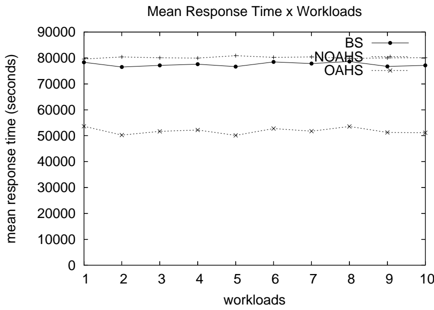
With this experiment, we observe that All Strategy (AS) is inviable according to all evaluated metrics. In our context, we cannot assume that all filters are complementary (CPU and I/O), as AS does. But, like presented in [22], for other type of jobs or maybe a subgroup of filters, resource sharing can be a good alternative. Moreover, Balanced Strategy (BS) and Non-Optimized AnthillSched (NOAHS) presented similar performance. So, we cannot discard both alternatives.

According to our previous experiment, we will not consider AS alternative anymore. In this third experiment, we submit the scheduling strategies under a maximum load. So, does not make sense the analysis of the system idle time. In Fig. 4, for all metrics, OAHS was the best scheduling strategy, and NOAHS the worst strategy. NOAHS parallelizes short jobs, creating unnecessary overhead and decreasing performance. In our data ming jobs, the first filter executes much more work than the others. NOAHS assigns more (useless) processors to the first filter and few processors to the other ones. Thus, the other filters become bottlenecks. This generates more overhead than a balanced distribution of filter copies or instances among processors.

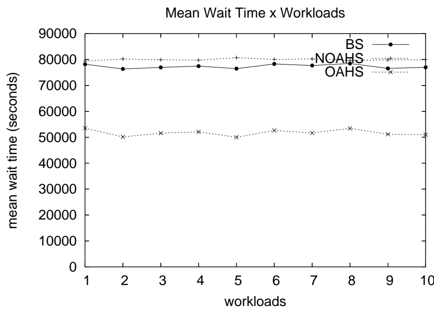
Based on the confidence intervals, our results show that NOAHS is not a viable alternative, because for short jobs, the parallelization of jobs leads to a high response time, as we see in Fig. 4(b). Moreover, BS presented a lower performance than OAHS. Despite of the low performance achieved with BS, we are not convinced yet that OAHS is really better than BS. Because of the considerable amount of short jobs in the workloads, the optimization in AnthillSched takes advantage over BS. To solve this problem, we included the same optimiza-



(a) Execution time per workload.



(b) Mean jobs response time per workload.



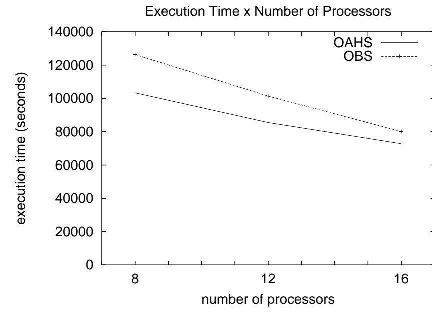
(c) Mean jobs wait time per workload.

Figure 4. Scheduling strategies performance for 10 different workloads.

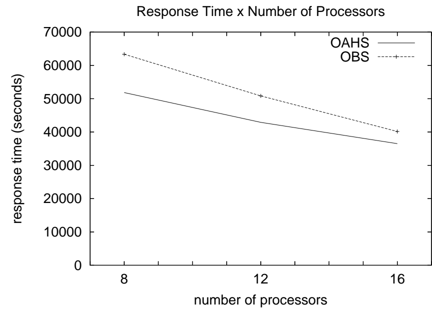
tion in BS and made another experiment.

Our final experiment verifies whether OAHS has a better performance than OBS (Optimized Balanced Scheduling) and if it scale up form 8 to 16 processors. We used four performance metrics (mean value for all workloads) and varied the number of processors from 8, 12 to 16. Moreover, we included the same optimization in BS (OBS), as mentioned before.

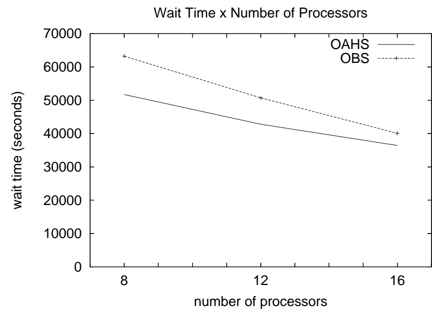
According to Fig. 5 and a visual test of the confidence intervals, all metrics showed that even with the same optimization, OAHS has a higher performance. In Fig. 5(d), we see a large slowdown due to the short jobs execution time,



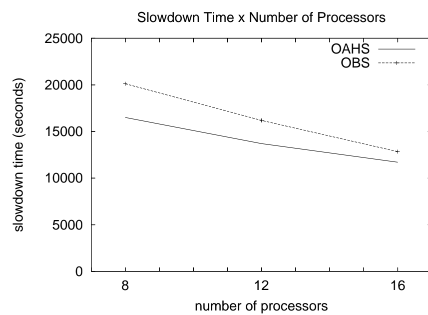
(a) Execution time per workload.



(b) Mean jobs response time per workload.



(c) Mean jobs wait time per workload.



(d) Mean jobs slowdown per workload.

Figure 5. OBS and OAHS performance for all workloads in a cluster with 8, 12 and 16 processors.

but high jobs wait time (Fig.5(c)) under a maximum load.

Finally, this last experiment showed that OAHS is more efficient than OBS. Moreover, the OAHS scaled up from 8 to 16 processors. As a limit of computing resources, we did not varied the number of processors to up than 16 processors. Neither a regression model can be used to estimate reliable metric values for a larger number of processors due to the limited number of points (8, 12 and 16). Thus, our main hypothesis that the number of filter copies C_i for an I^3-jobs depends equally to the number of input bytes B_i and execution time E_i of each filter was verified. In preliminary tests, not showed in this paper, we observed that the considering of different weights for CPU and I/O requirements did not seem to be a good alternative as the execution time of a controlled execution increases. But, as a future work, these experiments can be more explored to definitely discard this alternative.

6. Conclusion

In this paper, we proposed, implemented (in a real system) and analyzed the performance of AnthillSched. Irregular and Iterative I/O-intensive jobs have some features that are not take in account by parallel job schedulers. To deal with those features, we proposed some scheduling strategies based on simple heuristics.

Our experiments showed that resource sharing among all filters is not a viable alternative. They also showed that a balanced distribution of filter copies among processors is not the best alternative. Finally, we concluded that the use of a scheduling strategy, which considers jobs input parameters and distributes the filter copies according to each job's CPU and I/O requirements is a good alternative. We called this scheduling strategy AnthillSched. It creates a continuous dataflow among filters, avoiding bottlenecks and taking in account iterativeness. Our final experiment showed that AnthillSched is also a scalable alternative.

Our **main contributions** are: the implementation of a parallel job scheduling strategy called AnthillSched in a real system; a performance analysis of AnthillSched, which discarded some other alternative solutions.

As future works, we remark: creation and validation of a mathematical model to evaluate the performance of parallel I^3-jobs; exploration of different weights for CPU and I/O requirements in AnthillSched; use of other application types etc.

References

[1] Acharya, A., Uysal, M., Saltz, J.: Active Disks: Programming Model, Algorithms and Evaluation. In Proceedings of the Eight International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII). (1998) 81–91

[2] Andrade, N., Cirne, W., Brasileiro, F., Roisenberg, P.: OurGrid: An Approach to Easily Assemble Gridswith Equitable Resource Sharing. Job Scheduling Strategies for Parallel Processing. (2003)

[3] Batat, A., Feitelson, D.: Gang Scheduling with Memory Considerations. IEEE International Parallel and Distributed Processing Symposium. (2000) 109-114

[4] Beaumont, O., Boudet, V. and Robert, Y.: A Realistic Model and an Efficient Heuristic for Scheduling with Heterogeneous Processors. IEEE Heterogeneous Computing Workshop. (2002)

[5] Beynon, C. M., Ferreira, R., Kurc, T., Sussmany, A. and Saltz, J.:DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage Systems. IEEE Mass Storage Systems. (2000)

[6] Chapin, S.J. et al: Benchmarks and Standards for the Evaluation of Parallel Job Schedulers. Job Scheduling Strategies for Parallel Processing. (1999) 67-90

[7] Feitelson, D. and Nitzberg, B.: Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860.Job Scheduling Strategies for Parallel Processing. (1995) 337-360

[8] Feitelson, D., Rudolph, L.: Evaluation of Design Choices for Gang Scheduling using Distributed Hierarchical Control. Journal of Parallel and Distributed Computing (1996) 18-34

[9] Feitelson, D.G.: A Survey of Scheduling in Multiprogrammed Parallel Systems. Research Report RC 19790 (87657). IBM T. J. Watson Research Center (1997)

[10] Paul E. Utgoff and Carla E. Brodley.: An Incremental Method for Finding Multivariate Splits for Decision Trees. Seventh International Conference on Machine Learning. Morgan Kaufman (1990)

[11] Feitelson, D., Rudolph, L.: Metrics and Benchmarking for Parallel Job Scheduling. Job Scheduling Strategies for Parallel Processing. (1998) 1-24

[12] Feitelson, D.: Metric and Workload Effects on Computer Systems Evaluation. IEEE Computer. (2003) 18-25

[13] Franke, H., Jann, J, Moreira, J., Pattnaik, P., Jette, M.: An Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific. ACM/IEEE Conference on Supercomputing. (1999)

[14] Frachtenberg, E., Feitelson, D.G., Petrini, F. and Fernandez, J.: Flexible CoScheduling: Mitigating Load Imbalance and Improving Utilization of Heterogeneous Resources. 17th International Parallel and Distributed Processing Symposium. (2003)

[15] Góes, L. F. W., Martins, C. A. P. S.: Proposal and Development of a Reconfigurable Parallel Job Scheduling Algorithm. Master's Thesis. Belo Horizonte, Brazil. (2004) (in Portuguese)

[16] Góes, L. F. W., Martins, C. A. P. S.: Reconfigurable Gang Scheduling Algorithm. Job Scheduling Strategies for Parallel Processing. (2004)

[17] Nascimento, L.T., Ferreira, R.: LPSched - Dataflow Applications Scheduling in Grids. Master's Thesis. Belo Horizonte, Brazil. (2004) (in Portuguese)

[18] Neto, E. S., Cirne, W., Brasileiro, F., Lima, A.: Exploiting Replication and Data Reuse to Efficiently Schedule Data-intensive Applications on Grids. Job Scheduling Strategies for Parallel Processing. (2004)

[19] Silva, F. A. B., Carvalho, S., Hruschka, E.R.: A Scheduling Algorithm for Running Bag-of-Tasks Data Mining Applications on the Grid. EuroPar. (2004)

[20] Streit, A.: A Self-Tuning Job Scheduler Family with Dynamic Policy Switching. Job Scheduling Strategies for Parallel Processing. (2002) 1-23

[21] Veloso, A., Meira, W., Ferreira, R., et al.: Asynchronous and Anticipatory Filter-Stream Based Parallel Algorithm for Frequent Itemset Mining. European Conference on Principles of Data Mining and Knowledge Discovery. (2004)

[22] Wiseman, Y., Feitelson, D.: Paired Gang Scheduling. IEEE Transactions Parallel and Distributed Systems. (2003) 581-592

[23] Zhang, Y., H. Franke, Moreira, E.J., Sivasubramaniam, A.: Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques. IEEE International Parallel and Distributed Processing Symposium. (2000)

[24] Zhang, Y., Yang, A., Sivasubramaniam, A., Moreira, J.: Gang Scheduling Extensions for I/O Intensive Workloads. Job Scheduling Strategies for Parallel Processing. (2003)

[25] Zhou, B. B., Brent, R. P.: Gang Scheduling with a Queue for Large Jobs. IEEE International Parallel and Distributed Processing Symposium. (2001)