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 in this case is not only the processing requirements anymore: memory, network and storage capacity must all be considered in making a scheduling decision. Job executions are irregular and data dependent, alternating between CPU-bound and I/O-bound phases. In this paper, we propose and implement a parallel job scheduling strategy for such jobs, called AnthillSched, based on a simple heuristic: we map the behavior of an parallel application with minimal resources as we vary its input parameters. From that mapping we infer the best scheduling for a certain set of input parameters given the available resources. To test and verify AnthillSched we used logs obtained from a real system executing data mining jobs. Our main contributions are the implementation of a parallel job scheduling strategy, called AnthillSched in a real system, and a performance analysis of AnthillSched, which allowed us to discard some other scheduling alternatives considered previously.

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 usually not accurate. The fact that they are I/O intensive make them even less predictable, 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 several data domains, not only consuming data from those domains, but also generating intermediary them, increasing the volume of information 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 important because of the access patterns vary over time with each iteration. The degree of parallelism is a function of the data dependencies among iterations. As a consequence of these characteristics, scheduling of I^3 jobs is quite a challenge, and determining optimal scheduling for them is a very complex task, since it must consider locality, input size, data dependences, and parallelization opportunities.

Generally speaking, parallel job schedulers have been 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]. Those programs are structured as filters that communicate using streams, where parallelism is achieved by the instantiation of multiple copies of any given filter. Consistent addressing among filter instances is guaranteed by the use of labels associated with any data that traverses a stream. These programs are fundamentally asynchronous and implemented using an event-based paradigm. In the scope of this work, a job is the execution of a program with specific input parameters on specific data 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, there must be a continuous and balanced data flow between filters. Our premise is that the balance of the data flow between filters may be achieved by scheduling the proper number of filter copies or instances. In this paper we propose, implement, and evaluate a parallel job scheduling strategy called AnthillSched, which determines the number of filter instances according to each filter's CPU and I/O demands and schedules them. We evaluate AnthillSched using logs derived from actual workloads submitted to the Tamanduá[1] system, which is a data mining service that executes data mining I^3 jobs on Anthill, our filter-stream run time system.

This paper is organized as follows. We present the related work in Section 2. The following section introduces the Anthill programming environment, 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 remaining sections. Finally, we present our conclusions and future work.

2. Related Work

While we are not aware of works on scheduling I^3 jobs, other researchers have addressed the issue of scheduling parallel I/O-intensive jobs. Wiseman *et al.* [22] presented Paired-Gang Scheduling, in which I/O-intensive and CPU-intensive jobs share the same time slots. Thus, when an I/O-intensive job waits for an I/O request, the CPU-intensive job uses the CPU, increasing utilization. This approach indicates that processor sharing is a good mechanism to increase performance in mixed loads.

Another work [24] shows three versions of an I/O-Aware Gang Scheduling (IOGS) strategy. The first one, for each job, looks for the row in the Ousterhout Matrix (time-space matrix) with the least number of free slots where job 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 IOGS, but also 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 associated file nodes during execution. This strategy outperformed all the other ones.

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

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

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

3. The Programming Environment

A previous implementation of the Filter-Stream programming model [1] is DataCutter, a middleware that enables efficient application execution on distributed heterogeneous environments [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 between filters, and when a given 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, but selecting a particular copy to receive the data is tricky, since DataCutter implements automatic destination selection mechanisms based on round-robin or demand driven models.

We extend that programming model in the Anthill environment by providing a mechanism named labeled stream which allows the selection of a particular copy as destination based on some information related to the data (the labels). Such extension provides a richer programming environment, making it easier for transparent copies to partition global state. Besides that, Anthill provides a task-oriented framework, in which the application execution can be modeled as a collection of tasks and that represent iterations over the input data that may or may not be dependent on one another. In that way, Anthill explores parallelism in time and 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, since it is composed of N filters (processing phases or stages) connected by streams (communication channels). This job model explicitly forces the

1 Tamanduá means anteater in Portuguese.

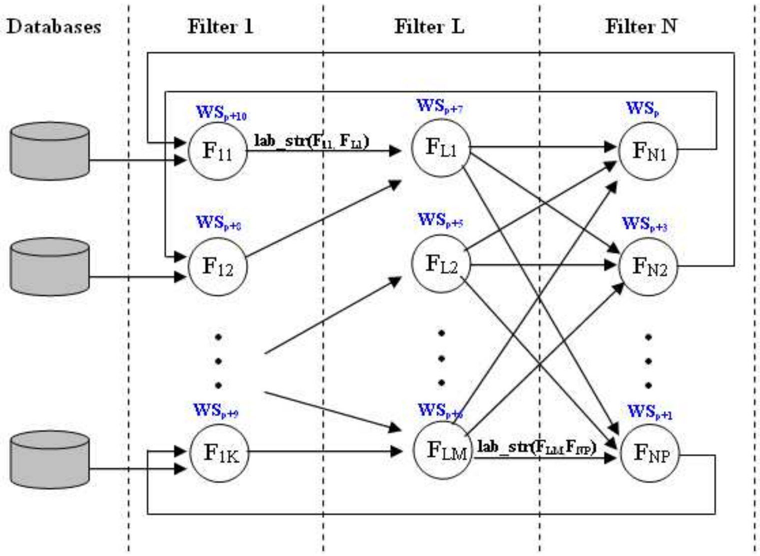


Figure 1. Anthill programming model.

programmer to divide the job in well defined phases (filters), in which input data is transformed by each filter into another data domain that is required by the next filter.

The Anthill programming model also explores spacial parallelism, as each filter can have multiple copies, or instances, executing in different compute nodes. Each communication channel between filters can be defined as point-to-point, to direct each piece of data to a specific filter copy (either round-robin or by defining a labeled stream) or broadcast, where data is copied to all filter copies of the filter in next level. A consequence of spacial parallelism is data parallelism, because a dataset is automatically partitioned among filter copies. Together with streams, data parallelism provides an efficient mechanism to divide I/O demand among filters, while labeling allows data delivery to remain consistent when necessary.

The task-oriented interface is what allows Anthill to efficiently exploit the asynchrony of the application. Each job is seen as a set of work slices (WS) to be executed which may represent each iteration of an algorithm and may be created dynamically, as input data is being processed (that is particularly useful for data-dependent, iterative applications). A work slice WS_i is created, its data dependencies to any previous slice WS_j is explicitly indicated. That gives Anthill information about all synchronization that is really required by the application, allowing it to exploit all asynchrony in slices that already had their dependencies met.

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, iterativeness, spacial and data parallelism, 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 compute the number of filter copies, considering filter execution times, filter I/O demands, data complexity, etc. It is important to notice that the overall application performance is highly dependent on such scheduling decisions.

We propose AnthillSched, a parallel job scheduling strategy, implemented as Anthill's job scheduler. It focuses on the proper scheduling of a I^3 job on a cluster, that is, the decision about the number of copies of each filter, based on the job input parameters. These parameters are specific of the algorithm to be executed in each job; for a clustering algorithm, for example, it might be the number of clusters to be considered, for example. Based on those parameters, AnthillSched must output the number of instances for each filter in the algorithm. There are two possible scheduling alternatives: analytical modeling, which is very complex and can be infeasible to our problem, or a simpler solution, such as an experimental heuristic.

Our approach is based on a simple experimental heuristic to solve a very complex problem in a efficient, although possibly not optimal, way. Our decision to use a heuristic was based on the fact that the I^3 applications in which we are interested have very complex interactions, for the processing is iterative and the applications themselves are iterative (users may run a same algorithm multiple times with different input parameters, trying to get a better result for their particular problems). Going for a full analytical model would most often be a very complex task. Al-

```
function AnthillSched (i, p : integer) : array of integer
    for 1 to m do
        for j = 1 to n do
            C_ij = p × ( B_ij / Σ_{k=1}^{n} B_ik + E_ij / Σ_{k=1}^{n} E_ik ) / 2
        endfor;

        for j = 1 to n do
            q = (j + 1)mod(n)
            if(broadcast(S_ij, S_iq))
                B_iq = B_iq × C_iq
        endfor;
    endfor;
    return C_i
end.
```

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

Figure 2. AnthillSched's algorithm.

though we may not be able to get to an optimal solution, with the simpler scheduling strategy, however, we still expect to eliminate possible bottlenecks and provide a continuous data flow with high asynchrony to I^3 jobs.

Given a program that must be scheduled, the domain of its input parameters must be first identified and clearly mapped. AnthillSched requires q controlled executions, one for each possible 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 of each filter (sequential pipeline) with certain combination of input parameters (say, combination i). For each job execution, we collect the number of input bytes B_{ij} and the execution time E_{ij} for each filter j.

In Anthill, each job is executed according to a FCFS strategy with exclusive access to all processors in the cluster. When a new job arrives, Anthill executes AnthillSched with the job's set of input parameters (i) and the number of available processors (p) as input. The scheduler outputs the number of filter copies C_ij for each filter j (represented as a whole as C_i), after m iterations. First, for each iteration, the number of copies of each filter C_{ij} is calculated according to Fig. 2, where n is the number of filters in the pipeline for that application. In this step, we normalize the number of input bytes B_{ij} and the execution time E_{ij} dividing them by the total sum of bytes and execution times, respectively. Then, we sum the normalized values and divide it by two, in order to obtain the relative resource requirements of each filter. For example, if we had a job with 3 filters, we might 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, we calculate the number of copies C_{ij} proportionally to the relative requirements of each filter.

The second step in Fig. 2 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. For every filter j, we must consider its stream to the next filter q ($q = (j+1)mod(n)$); that stream is identified as S_{jq}. If S_{jq} is a broadcast stream, the number of input bytes received by the destination filter B_{iq} must be multiplied by its number of copies C_{iq}. Thus, AnthillSched must recalculate the number of input bytes B_{iq}.

If we have a large number of possible input permutations, it is not feasible to run all controlled executions and store them. A solution in this case is to consider only a sampling of the possible permutations. When a new, or not considered, combination of input parameters of a job is found, an interpolation between the two nearest combinations can approximate the number of copies for each filter for that 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, since it defines a limit for the iterations, m.

During to preliminary tests, we verified that controlled executions that spent 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 named this version Optimized AnthillSched (OAS).

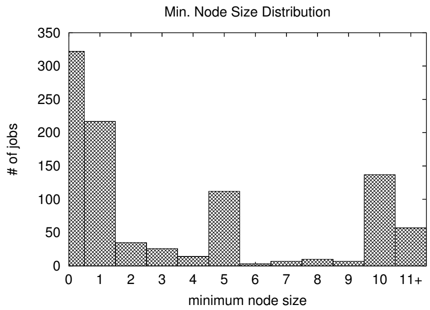
5. Results

In this section we evaluate our scheduling strategy by applying it to a data mining application: the ID3 algorithm for building decision trees. In particular, 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's copies C_ij is uniformly distributed according to B_{ij} and E_{ij}, we eliminate possible bottlenecks and provide a continuous data flow with high asynchrony for a job.

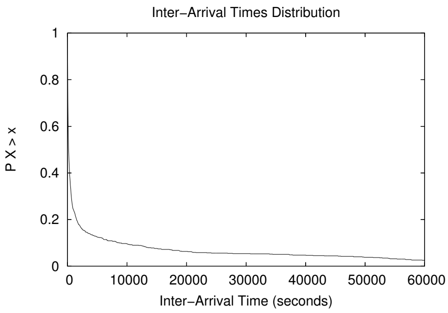
To test and analyze our hypothesis, we compared two versions of AnthillSched (non-optimized and optimized) to other two job scheduling strategies: *Balanced Strategy* (BS) and *All Strategy*(AS). The proposed strategies use the maximum number of processors available. The BS tries to balance the number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

5.1. Experimental Setup

For the workload we used real logs of data mining jobs executed in Tamanduá platform by its users. As previously mentioned, Tamanduá is a scalable, service-oriented data mining platform executing on different clusters and that uses efficient algorithms with large databases. The logs used are from clusters were Tamanduá is being used to mine government databases (one on public expenditures, another on public safety — 911 calls). Today, there are various data mining algorithms implemented in Anthill such as *A priori*, K-Means, etc. In our experiments, we are concerned with ID3 (a decision tree algorithm for classification) [10]. The main input parameter that influences ID3 is the minimum node size, which determines the minimum number of homogeneous points needed to create a node in the tree.



(a) Minimum node size histogram.



(b) Inter-arrival time inverse cumulative distrib.

Figure 3. Workload characterization.

Based on real logs from Tamanduá, we characterized the inter-arrival time between jobs as shown in Fig. 3(b) and the minimum node size pattern all jobs in Fig. 3(a). As we see in Fig. 3(a), the majority of minimum node size values are concentrated between 0 and 10. In ID3, as an approximation, the minimum node size may be considered inversely proportional to the execution time, so it means 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. It was verified that the inter-arrival time (Fig. 3(b)) fits in an 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 generated 10 workloads, each one composed of 1000 jobs executing the ID3 algorithm with minimum node size and submission time derived from the distribution in Figure 3.

To test the scheduling strategies under different conditions, we varied the load (job arrival rate) between low, medium and maximum. The low load considered the inter-arrival time between jobs based on 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 the medium load workload, we used for the inter-arrival time only a subset of Fig 3(b) with the peak periods. Finally, the maximum load assumes that all jobs of the workload arrive at same time; in this case we just ignore the inter-arrival time between jobs.

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

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

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

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

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

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

5.2. Experimental Results

Using the workload derived from the previous characterization, 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

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
NOAS	2065464.68	2029245.12	2155317.20	36930.32	2042575.47	2088353.90
OAS	2065410.48	2029242.96	2155136.99	36896.76	2042542.06	2088278.89

(a) Execution time for each strategy under 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
NOAS	1209.70	90.26	3975.80	1304.82	400.98	2018.43
OAS	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
NOAS	8.13	6.01	10.24	1.33	7.30	8.95
OAS	4.84	3.64	5.89	067	4.43	5.25

(c) Mean job 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
NOAS	168.28	166.28	170.67	1.36	167.44	169.44
OAS	107.82	101.68	110.65	2.63	106.45	109.19

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

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

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 the maximum load, we saturated the system to test the alternatives. In our final experiment (scalability under maximum load), we compare the two best strategies with the 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, c_2 (lower, 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 very close. A low load implies large inter-arrival times; in this case, the intervals 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 spite of that it, we observe in Table 1(b) that system using Optimized AnthillSched (OAS) spent more time idle than the other ones. This is a first indication that jobs executed with OAS 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. As can be seen on Table 1(c), the mean job wait time, and consequently the mean job response time for OAS, is really lower than the other strategies.

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 that. As we expected, a scheduling strategy that reduces the mean job wait

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
NOAS	160181.04	159284.49	161230.22	548.70	159840.95	160521.12
OAS	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
NOAS	21.01	0.00	120.53	39.66	-3.57	45.60
OAS	32.59	0.00	119.77	48.37	2.61	62.57

(b) Idle time for each strategy under 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
NOAS	46849.72	43929.01	49249.10	1449.23	45951.49	47747.95
OAS	18721.51	15228.08	21785.22	2032.29	17461.90	19981.11

(c) Mean job wait time for each strategy under 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
NOAS	47009.87	44089.37	49409.53	1449.21	46111.66	47908.09
OAS	18824.49	15325.52	21891.15	2033.92	17563.88	20085.10

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

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

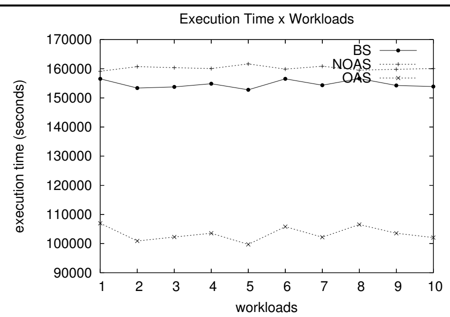
time and consequently the response time, increases the idle time of the system.

Under medium load, the results are shown in Table 2. With medium load, the inter-arrival times are not always larger than response times. In Table 2(a), AS presented the worst execution time for all workloads. After that, BS and Non Optimized AnthillSched (NOAS) presented similar performance, with a little advantage for BS. Table 2(b) shows that on average, OAS achieved the higher idle time. As we confirm in Table 2(c,d), the mean job wait and response time are lower when the OAS is used, so the system have more idle time waiting for another job arrival.

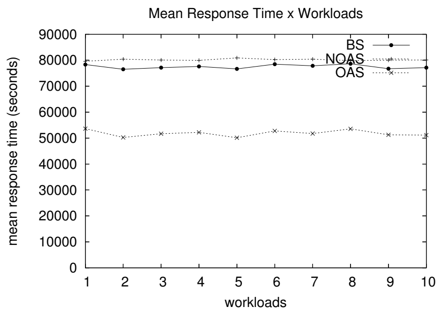
With this experiment, we observe that the All Strategy (AS) is no a viable strategy based to all evaluated metrics. In our context, we cannot assume that all filters are complementary (CPU-bound and I/O-bound), as AS does. However, for other type of jobs or maybe a subgroup of filters, resource sharing can be a good alternative [22]. Moreover, the Balanced Strategy (BS) and Non-Optimized AnthillSched (NOAS) presented similar performance, so we cannot discard both alternatives.

Based on our previous experiment, we do not consider AS an alternative from this point on. In the third experiment, we evaluate the scheduling strategies under maximum load. In this case we do not consider the system idle time, given that all jobs are submitted at the same time. As we can see in Fig. 4, for all metrics, OAS was the best scheduling strategy, and NOAS the worst strategy. NOAS parallelizes short jobs, creating unnecessary overhead and reducing performance. In our data mining jobs, the first filter tends to have much more work than the others. NOAS 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.

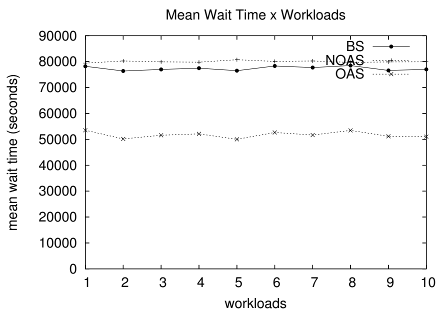
Based on the confidence intervals, our results show that



(a) Execution time per workload.



(b) Mean job response time per workload.

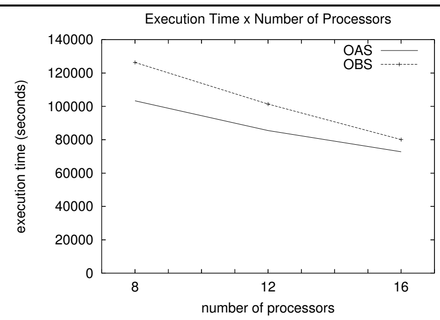


(c) Mean job wait time per workload.

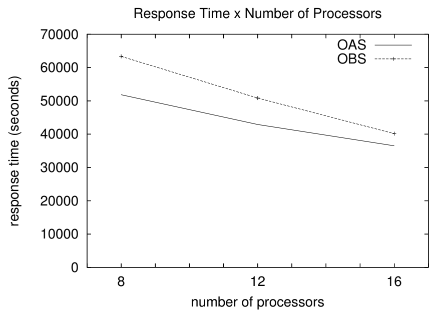
Figure 4. Scheduling strategies performance for 10 different workloads.

NOAS 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 OAS. Despite of the low performance achieved with BS, we are not convinced yet that OAS 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 optimization in BS for the next experiment.

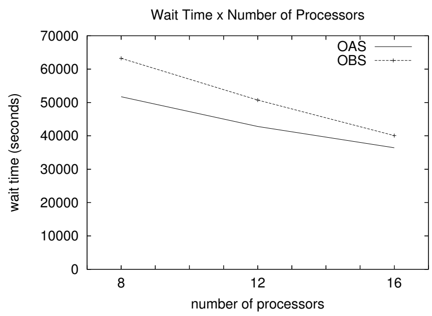
Our final experiment verifies whether OAS has a better performance than OBS (Optimized Balanced Scheduling) and if it scales up from 8 to 16 processors. We used



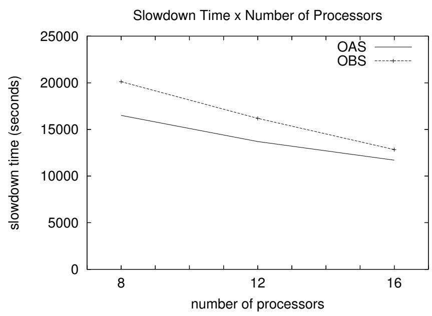
(a) Execution time per workload.



(b) Mean job response time per workload.



(c) Mean job wait time per workload.



(d) Mean job slowdown per workload.

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

the heavy load configuration and varied the number of processors from 8, 12 to 16 while considering four performance metrics (execution, response, wait and slowdown times, mean values for all workloads). 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 show that, even with the same optimization, OAS has better performance. In Fig. 5(d), the large slowdown is due to the short jobs, which have low execution times, but high job wait times (Fig. 5(c)) under the maximum load.

Finally, this last experiment showed that OAS is more efficient than OBS. Moreover, OAS scaled up from 8 to 16 processors. Due to our limitations on computing resources, we could not vary the number of processors beyond 16. From the results, 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 shown in this paper, we observed that the use of different weights for CPU and I/O requirements in the AnthillSched algorithm (Fig. 2) did not seem to be a good alternative as the execution time of a controlled execution increased. However, as future work, these experiments can be more explored to definitely discard this alternative.

6. Conclusion

In this work we have 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 taken into account by parallel job schedulers. To deal with those features, we proposed a scheduling strategy based on simple heuristics.

Our experiments show that resource sharing among all filters is not a viable alternative. They also show that a balanced distribution of filter copies among processors is not the best alternative either. 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 named this scheduling strategy AnthillSched. It creates a continuous data flow among filters, avoiding bottlenecks and taking in account iterativeness. Our experiments show that AnthillSched is also a scalable alternative.

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

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

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 Grids with 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] Utgoff, P., Brodley, C.: 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)