Scheduling Data Intensive Particle Physics Analysis Jobs on Clusters of PCs

S. Ponce
European Laboratory for Particle Physics (CERN)
Information Technology Department
CH-1211 Geneva 23, Switzerland
sebastien.ponce@cern.ch,

R.D. Hersch
Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
rd.hersch@epfl.ch

Abstract:
Scheduling policies are proposed for parallelizing data intensive particle physics analysis applications on computer clusters. Particle physics analysis jobs require the analysis of tens of thousands of particle collision events, each event requiring typically 200ms processing time and 600KB of data. Many jobs are launched concurrently by a large number of physicists. At a first view, particle physics jobs seem to be easy to parallelize, since particle collision events can be processed independently one from another. However, since large amounts of data need to be accessed, the real challenge resides in making an efficient use of the underlying computing resources. We propose several job parallelization and scheduling policies aiming at reducing job processing times and at increasing the sustainable load of a cluster server. The complexity of each policy is analysed as a measure of the scalability of the system.

Since particle collision events are usually reused by several jobs, cache based job splitting strategies considerably increase cluster utilisation and reduce job processing times. Compared with straightforward job scheduling on a processing farm, cache based first in first out job splitting speeds up average response times by an order of magnitude and reduces job waiting times in the system's queues from hours to minutes.

By scheduling the jobs out of order, according to the availability of their collision events in the node disk caches, response times are further reduced, especially at high loads. In the delayed scheduling policy, job requests are accumulated during a time period, divided into subjob requests according to a parameterizable subjob size, and scheduled at the beginning of the next time period according to the availability of their data segments within the disk node caches. Delayed scheduling sustains a load close to the maximal theoretically sustainable load of a cluster, but at the cost of longer average response times. We also propose an adaptive delay scheduling approach, where the scheduling delay is adapted to the current load. This last scheduling approach sustains very high loads and offers low response times at normal loads.

We analyse the benefits of pipelining computation and accesses to tertiary storage and to the local disk caches. Pipelining tends to increase the throughput of jobs and allows the system to sustain higher loads.

Finally we analyse the complexity of the different scheduling algorithms both in terms of space and time. The system is highly scalable and supports a cluster of up to several tens of thousands of nodes.

Keywords: High-energy physics; particle collision analysis; cluster computing; data intensive computing; data distribution; cache-based job splitting; delayed scheduling; adaptive delay scheduling; pipelining; scheduling complexity.

1 Introduction

We are interested in parallelizing concurrent data intensive applications on clusters of PCs. The data intensive application we consider is a particle collision analysis software used in high energy physics experiments. The LHCb experiment at the European Laboratory for Particle Physics (CERN) faces the issue of analysing several petabytes of data distributed across the world.

Data is stored on tapes using Castor [7], a tertiary mass storage system manager developed at CERN that completely hides the tape archives from the user by caching data on large disk arrays. Data retrieval time from tertiary mass storage is about three times slower than the corresponding data processing time.

There are many strategies for scheduling parallel applications on a cluster of PCs [18, 2, 13, 1]. However, these strategies are generally not applicable to data intensive applications which are arbitrarily divisible [5], which access partly overlapping data segments and whose data needs to be loaded from tertiary storage. On the other hand, strategies focusing on data retrieval [15, 4] deal mainly with I/O throughput without considering job distribution and scheduling issues. Kadayif et al. address the problem of data aware task scheduling but only for a unique CPU [12].

We present new paradigms that aim at optimising the parallelization and scheduling of jobs on a cluster of PCs for data-intensive applications. The proposed parallelization and scheduling policies rely on job splitting, disk data caching, data partitioning [16], out of order scheduling as well as on the concept of delayed scheduling. The time and space complexity analysis of the proposed scheduling policies is studied carefully leading to a measure of the scalability of our policies. The gain of I/O and CPU pipelining is analysed for the proposed scheduling policies as well as the consequence on the original physics code.

In Section 2, we present the context of this work, i.e. the LHCb experiment at CERN and its computing and data access requirements. We introduce the simulation tools and the simulation parameters used to evaluate and compare the scheduling policies. In Section 3, we present simple first come first served job parallelization and scheduling policies. In Section 4, we introduce out of order job scheduling and data replication policies. In Section 5, we propose a delayed scheduling policy optimising the resources of the system. Delayed scheduling may have disadvantages for end users but allows to sustain considerably heavier loads. In Section 6, we propose an adaptive delay strategy that allows to minimise user waiting time at low and normal loads and to optimise the utilisation of the computing resources at high loads. In Section 7, we show the influence of pipelining I/O and CPU on the different scheduling policies. Both network I/O and disk I/O are considered. In Section 8, we analyse the time and space complexities of the proposed scheduling policies. In Section 9, we draw the conclusions.

2 Context and Tools

The present work is based on studies made at CERN in the context of the LHCb [8] experiment. Let us first present the particle collision event analysis problem and its associated computing and data access requirements.

2.1 The LHCb experiment

LHCb [8] is the name of one of the future Large Hadron Collider (LHC) experiments. Its primary goal is the study of the so called CP Violation [9]. This physical theory suggests that, in the world of subatomic particles, the image of a particle in a mirror does not behave like the particle itself [6]. One of the fundamental reasons of this effect is the existence of the bottom-quark and its cousin, the top-quark. The LHCb experiment intends to study the bottom-quark, in the form of the B-meson. Particles such as this meson are produced when colliding other high energy particles, accelerated protons in the case of LHC. These collisions produce hundreds of new particles among which the physicists try to detect B-mesons in order to measure their parameters and to deduce their behaviour e.g. the way they decay.

2.2 The computing requirements

Parameters of the particles produced by the LHCb detector are obtained by analysing the huge amount of data coming out of the experiment, approximately of a few terabytes per second, 24 hours a day.

However, most of the acquired experiment data is irrelevant. Nevertheless, about one petabyte (10^{15} bytes) of data per year is detected as being potentially interesting and recorded on tapes. These tapes are analysed off-line by the community of physicists. This analysing process requires substantial CPU power as well as access to several terabytes of data per analysis job. Therefore, it is necessary to parallelize the data analysis programs.

The parallelization is facilitated by the nature of the processing patterns and of the corresponding data segments. Data segments comprise a succession of “small”, fully independent collision events (around 600KB per event). Event processing consists of scanning and analysing the events one by one and of creating corresponding statistics. The event analysis output is very small. It comprises a set of histograms which can be easily merged when carried out on several nodes in parallel. Merging and transferring the results requires therefore a negligible effort. Since events can be analysed independently one from another (no data dependency), event analysis does not induce inter-node communications.

2.3 Simulation Framework

We needed a simulation framework allowing to develop and test different job parallelization and scheduling policies before running them on a real computing cluster. Since no efficient software adapted to our needs is available, we created our own simulation tools. Existing general purpose simulation tools [17, 11] simulate communications between nodes and are therefore too slow to simulate days or weeks of particle collision event processing.

Since particle collision event processing does not induce communications between computing nodes, we only take into account the communication time related to data transfers. The simulation framework (Fig. 1) simulates the behaviour of a cluster of PCs (CPU + memory + disk) connected via a high speed network (typically Gigabit Ethernet) to a shared tertiary storage device (e.g. the CASTOR system at CERN).

Cluster management and job scheduling are carried out on a dedicated node called “master node”. The job parallelization and scheduling software may run both on the simulated and

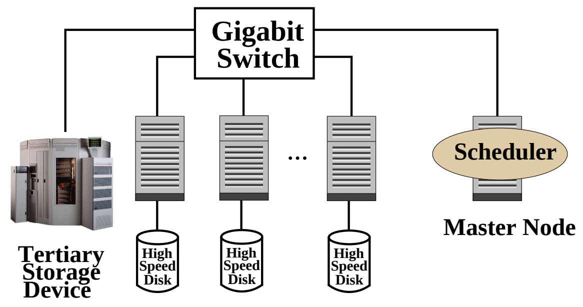


Figure 1: Architecture of the simulated cluster

on the target system (production environment). It implements a plugin model, enabling new scheduling policies to be easily added.

Due to the available CPU power limitations, the simulations are carried out at a reduced scale. The total data space is reduced from 2 PB to 2 TB (unless specified) and the number of nodes in the cluster from 10000 to 10 or 20 nodes, in addition to the master node (except for the complexity studies).

2.4 Parameters of the simulation

In real computing clusters, important parameters are CPU speed, node memory, disk space, disk throughput, etc. Within our simulation framework, we made the following assumptions :

- The processing node memory is considered to be infinite since we only run a single job or subjob per processor at any given time. We therefore expect the node memory to be always large enough.
- Processing nodes are single CPU computers and all nodes are identical.
- Processing power requirements are expressed in terms of single CPU seconds.
- Disk throughput is 10 MB/s.
- The node disk cache size is either 50 GB, 100GB or 200GB.
- The tertiary storage system is accessed through Castor [7]. This system caches tape data on disk arrays. We therefore do not take the tertiary storage system data access latency into account.
- Throughput from tertiary storage to each node is 1 MB/s.
- The total data space accessible by the high energy physics analysis jobs is 2 TB.
- The default number of processing nodes in the cluster is 10. Simulations were also carried out for 5 and 20 nodes and lead to similar results. Thus, we only present the case of 10 nodes.
- Jobs are typical high energy physics analysis jobs. They consist of a large collection of events, 40000 events on average. The number of events follows an Erlang probability distribution with parameter equal to 4. Each event requires 200 ms CPU processing time and access to 600 KB of data.
- The events accessed by a given job are contiguous. The data segment they form starts at a random position within the dataspace. The distribution of the job start points within the dataspace is homogeneous except for two regions, representing together 10% of the total data space but incorporating 50% of the start points. Such a distribution mimics the fact that the fraction of the data associated with some very interesting events is accessed far more frequently than the remaining data. This start point distribution yields the data utilisation curve shown in Figure 2. The probability of having two jobs with overlapping data is around 6.5%. Note that this probability would be two times smaller with a uniform distribution of start points. When jobs have overlapping segments, the average size of the overlapping region is half the average size of the data segment associated to a job, i.e. 12 GB.
- Job requests arrive randomly, with an exponential distribution of intervals between arrival times. The mean number of arrival jobs per time (cadence) depends on the considered load.

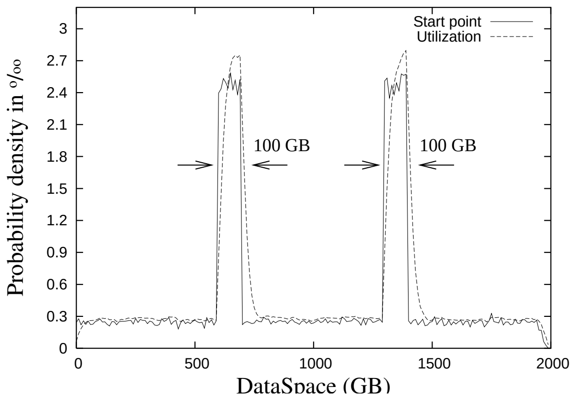


Figure 2: Start point and data utilisation distribution

3 First come first served scheduling policies

The proposed scheduling policies rely on the following basic principles :

- Once started, a job never hangs. Therefore, at least one dedicated node is allocated to it.
- Jobs are started in a first come first served order in order to ensure a fair treatment of user requests.

Keeping these principles in mind, we analyse the speedup that can be obtained with load balancing strategies such as simple job splitting and cache oriented job splitting.

3.1 Processing farm oriented job scheduling

The simplest scheduling policy relies on the processing farm paradigm. This is the policy in use at CERN for scheduling jobs on a computing cluster comprising hundreds of nodes. Jobs are queued in front of the cluster and are transmitted to the first available node. This node remains dedicated to that job until its end. No disk caching is performed. All data segments are always transferred from tertiary storage when needed.

This simple scheduling algorithm is well studied and understood. A mathematical model can be established which describes the cluster behaviour as a special case of a M/Er/m queueing system [14]. We simulate the processing farm oriented job scheduling policy as a reference for judging the performance of the proposed more advanced parallelization and scheduling policies.

3.2 Job splitting

Since jobs contain tens of thousands independent events, they may be split into subjobs running in parallel on different nodes of the cluster. Job splitting occurs when new nodes become available and would remain idle, i.e. no new jobs are in the queue. The job splitting scheduling policy ensures that the maximum possible number of nodes is used at any time. Data segments are only requested from tertiary storage when they are needed i.e. when the corresponding event analysis code is being executed. Job splitting induces therefore no data replication. The job splitting scheduling policy is described in Table 1.

The job splitting policy performs always better than the simple processing farm oriented job scheduling policy. The job processing time is always lower compared with the processing farm approach since there are always as many jobs running as in the processing farm approach and since job splitting reduces the job processing time. In both approaches, all data segments are transferred from tertiary storage.

Table 1: The job splitting scheduling policy

Upon *job arrival*

- If some nodes are idle, the new job is split into subjobs of equal sizes, one per idle node. All subjobs are launched in parallel. To avoid too small jobs, we do not split beyond a minimal job size (10 events).
- If no node is idle but some job(s) is/are running in parallel on several nodes, one node is released by the job having the largest number of nodes per event to process. The corresponding subjob is suspended and the new job is launched on the released node.
- If there are as many jobs running as nodes, the new job is queued.

Upon *subjob end* (but not job end)

- If there are suspended subjobs within the same job, one of them is activated and runs on the node becoming free.
- Otherwise the node is allocated to an already running job. The largest subjob running on the cluster is split into two equal parts, one of them being launched on the free node. Again, jobs below a minimal size are not split.

Upon *job end*

- If there are queued jobs, the first queued job is run.
- Otherwise the free node is allocated to an already running job, as in the case of *subjob end*.

3.3 Cache oriented job splitting

The job splitting scheduling policy remains very close to the processing farm approach since the disks of the processing nodes are not used for data caching. By always caching data arriving from tertiary storage on node disks, we improve the effectiveness of job splitting. We try to schedule jobs on those nodes which store, at least partly, their corresponding data segments. This strategy leads to the new cache oriented job splitting policy detailed in Table 2.

Cache oriented job splitting offers on average more performance than simple job splitting since it takes advantage of disk

Table 2: Details of the cache oriented job splitting policy

Upon *job arrival*

- The new job is split into subjobs depending on the content of node disk caches : data processed by a given subjob should always either be fully cached on a node or not cached at all. Again, there is a lower limit on the smallest job size.
- If some nodes are idle, they are given the most suitable subjob (a fully cached subjob if such a subjob exists). If there are not enough subjobs for all nodes, they are further subdivided. If there are too many subjobs, the ones that cannot immediately be scheduled are suspended.
- If no node is idle but some job(s) is/are running in parallel on several nodes, one selected node is released by a selected job. Node and job selection are performed so as to maximise cached data access. We try to replace a subjob working with non cached data by the new job or one of its subjobs, working on cached data.
- If there are as many jobs running as nodes, the new job is queued.

Upon *subjob end* (but not job end)

- If there are suspended subjobs within the same job, one of them is activated on the node becoming free. The chosen subjob is the one having the largest amount of data cached on that node.
- Otherwise the node is allocated to an already running job. The subjob that is split is the one for which the caching benefit is the largest.

Upon *job end*

- If some jobs are in the queue, the first one is taken and run.
- Else if there are suspended subjobs, the most suitable one is activated.
- Otherwise an already running subjob is split, as carried out in case of *subjob end*.

The scheduler maintains the job and subjob queues as well as the state of all disk caches in the cluster. When needing new disk cache space, it deallocates the least recently used cached segments.

caches. Not every single job terminates earlier since the ordering may change but, in the general case, the average time a job spends in the system is reduced. Furthermore, the cluster is better utilised and therefore capable of sustaining slightly higher loads.

3.4 Simulation results

Two variables define the performance of a scheduling strategy, the *average waiting time* and the *average speedup*, both a function of the cluster *load*.

The *waiting time* is the time spent between job submission and beginning of job processing. It is interesting to compare it with the average time needed to run a simple isolated job on a single processing node without cache. In our context, the average single job single node processing time without disk cache is 32000 seconds, i.e. almost 9 hours.

The *speedup* of a job is the single job single node processing time in processing farm oriented job scheduling divided by the job processing time in the parallel system when scheduled according to the current policy. The *processing time* is defined as the time between the effective start of job processing, i.e. start of processing of the first part of the job, and the end of job processing, i.e. end of processing of the last part of the job. Thus, the processing time may include periods where the job or part of its subjobs are suspended.

The speedup is larger than one if there is a gain in terms of processing time. Two main factors contribute to the speedup : parallelization of jobs by job splitting and data caching. The parallelization is maximised when each job is subdivided into as many subjobs as available nodes. The performance improvement due to parallelization is thus less or equal to the number of nodes, i.e. 10 in our cluster configuration. Data caching is maximised when data is always read from disk caches instead of tertiary storage. In our context, the performance improvement due to maximal data caching is slightly larger than 3. Therefore the maximal overall speedup that can be reached is 30.

The *load* of the cluster is measured in terms of mean number of job arrivals per hour. The maximal load of a cluster corresponds to the load sustainable when all processors run at 100% cpu utilization and data is accessed from disk caches only. In our context (0.2 s CPU and 0.06 s disk I/O per event, 40000 events per job and 10 nodes), the maximal load is 3.46 jobs per hour.

Hereinafter, all measures make the assumption that the cluster runs in steady state. We do not take into account the startup period of the cluster, when empty disk caches are filled.

Figure 3 gives the average speedup and waiting time for different loads and for each of the scheduling policies described so far, including different cache sizes for the cache oriented job splitting. The curves on the graph are cut at high loads when the system leaves the steady state and becomes overloaded. When overloaded, the notion of average waiting time does not make sense anymore since jobs are accumulating and the waiting time grows to infinity.

The results show that the job splitting policy improves the performance, especially when the load is not too high. The cache size in the cache oriented job splitting policy appears to be

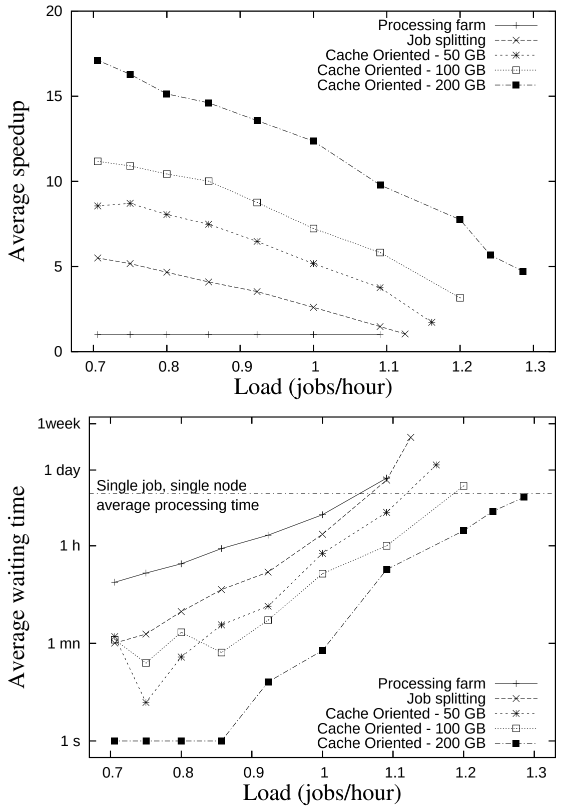


Figure 3: Average speedup and waiting time for different scheduling policies, different cache sizes and different load levels for a system comprising 10 processing nodes

decisive. The simulation shows that the maximal data caching speedup factor (i.e. 3) is reached for a disk cache size of 200GB (in Fig. 3, compare the job splitting with the cache oriented policy at low loads). For smaller caches, the gain in performance compared with the non cache-oriented job splitting is approximately proportional to the size of the disk cache.

Besides being vertically shifted, the waiting time curves are similar one to another. As foreseen, the policies providing a higher speedup induce a lower waiting time. Increasing the cache size decreases the job waiting time from days to hours.

4 Data distribution and data replication

In Section 3, we introduced a cache based job splitting policy ensuring a high degree of fairness by running the jobs in a first in first out order.

Let us study what can be gained by relaxing this constraint and analyse to what extend a certain degree of fairness can still be reached. Our hypothesis is that the usage of cache may be improved if we let jobs that find useful data in the cache execute before jobs that have to load their data from tertiary storage.

4.1 Out of order job scheduling

We define a new scheduling policy relying on out of order job scheduling that aims at making a maximal usage of node disk caches. Table 3 describes the out of order job scheduling policy.

Table 3: Details of the out of order job scheduling policy

Each node maintains a queue of subjobs. These subjobs only need data that is cached on their node. An extra queue contains subjobs with no cached data.

Upon job arrival

- When a job enters the cluster, it is split into subjobs so as to ensure that each subjob is either fully cached on a node or not cached at all. Jobs are not split beyond a minimal size.
- Subjobs with cached data are immediately run if the node is idle or if it is running a subjob without cached data. In that case, the former subjob is suspended and placed back at the first position of the queue where it came from (queue of subjobs with no cached data or a specific node queue).
- The remaining subjobs with cached data are queued on the nodes where their data is cached.
- If some nodes are still idle, they are fed with the subjobs having no cached data. These subjobs may be split in order to feed all nodes.
- Remaining subjobs with no data cached (if any) are put in the "no cached data" subjob queue.

Whenever one or several node(s) become(s) available

- If the node has subjob(s) waiting in its queue, the first subjob is run.
- Otherwise a subjob waiting in the "no cached data" subjob queue is run. In case several nodes are available and there are not enough subjobs in the queue, subjobs may be further split. The lower limit on the size of subjobs also applies here.
- If some nodes are still available (no subjobs at all in the special queue or too small ones to be split), they will take work from the most loaded nodes. By doing this, some subjobs that had cached data will be run on nodes that don't have the data. Thus, the subjobs are split so as to ensure that the two subjobs terminate around the same time. The new subjob incorporates a flag specifying that a subjob with cached data may take precedence over it in the future (see second point above).

This scheduling policy does not guarantee fairness i.e., according to the availability of cached data segments, the job execution order is modified. One may imagine a succession of jobs where a given job having no data in cache would never be executed. In order to ensure a minimal degree of fairness, we add an extra feature. Whenever the waiting time of a given job in the queue of jobs with no data cached exceeds a given maximum (2 days in our context), the job is run with a higher priority. The first available node executes this job before running any other job or subjob. When the cluster is not overloaded, such an event occurs very seldom since there will always be a time point at which the queues become empty and jobs with non cached data may be launched. It typically occurs for less than 0.5 ‰ of the jobs.

This out of order job scheduling policy clearly optimises the global job throughput by maximising accesses to data from disk caches. However, isolated jobs may have an exceptionally long waiting time.

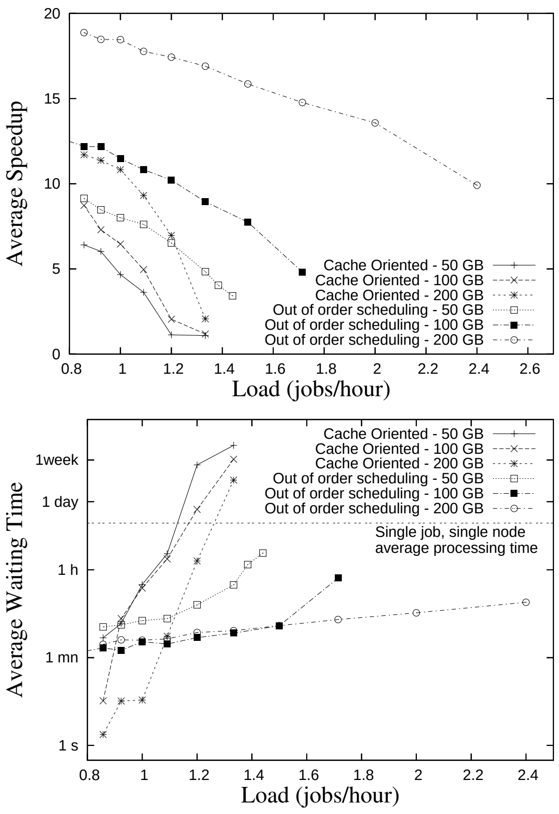


Figure 4: Average speedup and waiting time for cache-oriented job splitting and out of order scheduling policies

Figure 4 gives the average speedup and waiting time for the out of order job scheduling policy. As in Figure 3, the curves are cut at high loads when the cluster becomes overloaded, i.e. when queues start growing indefinitely.

Figure 4 shows that the out of order scheduling policy performs on average much better than the cache oriented job splitting policy both from a cluster utilisation and a user point of view. For the same amount of cache and under the same load, we obtain a much higher speedup and an average waiting time which is an order of magnitude lower. The server also sustains far larger loads, especially in the case of large caches. The degradation of the speedup at the proximity of the maximal load is excellent.

Regarding the possibly longer waiting time for individual jobs, Figure 5 shows the typical waiting time distribution for the out of order scheduling policy near the maximal sustainable load. The worst-case waiting time is one to two days, depending on the cache size. This is acceptable since the single job single node average processing time is 9 hours. The waiting time distribution curves characterise the out of order scheduling policy. Arriving jobs can be classified into two categories : either they have cached data and can overpass the other jobs (left part of the curves) or they have no cached data and are overpassed (right part of the curves).

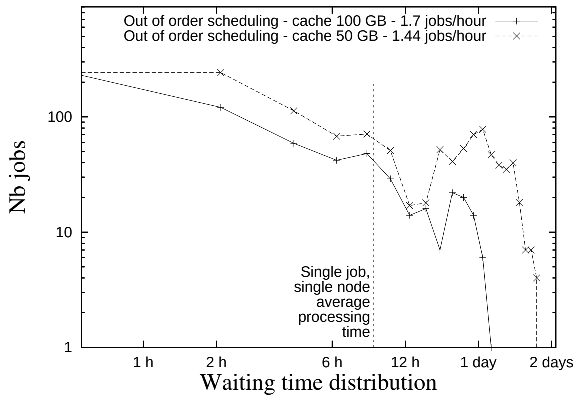


Figure 5: Waiting time distribution for the out of order scheduling policy near the maximal sustainable load.

4.2 Data replication

One may think that for high loads the previous scheduling algorithm may be further improved by performing data replication between the nodes of the cluster.

Whenever a node is overloaded and other nodes take work from it without having the corresponding data in their cache, it is pretty inefficient to grab the data again from the tertiary storage system. It is better to read the data directly from the disk of the overloaded node and copy it onto the local node disk.

However, such a replication strategy has drawbacks. The replication of the data segment onto the new node requires another data segment to be removed from the disk cache. The removed data segment may have been useful for future jobs. Thus, replicating a data segment when using the replicated data segment only once is not worthwhile. It is therefore preferable to directly read the data segment from the other node and use it without replication.

Replication should take place when the cost of not replicating is larger than the cost of replication [3, 10]. Applying this principle, we adopted the following strategy. Whenever a node works on a data segment that is cached on another node, the data segment is remotely read from the other node. By default, a data segment read from a remote node is not put in the cache of the new node. A data segment is replicated only when the cost of not having replicated it from the beginning exceeds the cost of the replication. This information is obtained by keeping in each node the number of remote accesses to its data segments. In our context, data replication is carried out only on data items that are accessed for the third time.

Out of order job scheduling with and without data replication are compared in Figure 6. Basically, there is no difference : data replication does not improve our scheduling algorithm.

The reason for the poor behaviour of data replication appears clearly if one tries to analyse the consequences of the out of order scheduling policy in respect to the distribution of data across the different nodes. The scheduling algorithm, taking advantage of job or subjob independence, always tries to use all available nodes whenever a job arrives. This leads to a situation where jobs are always split onto many nodes. As an example, the first

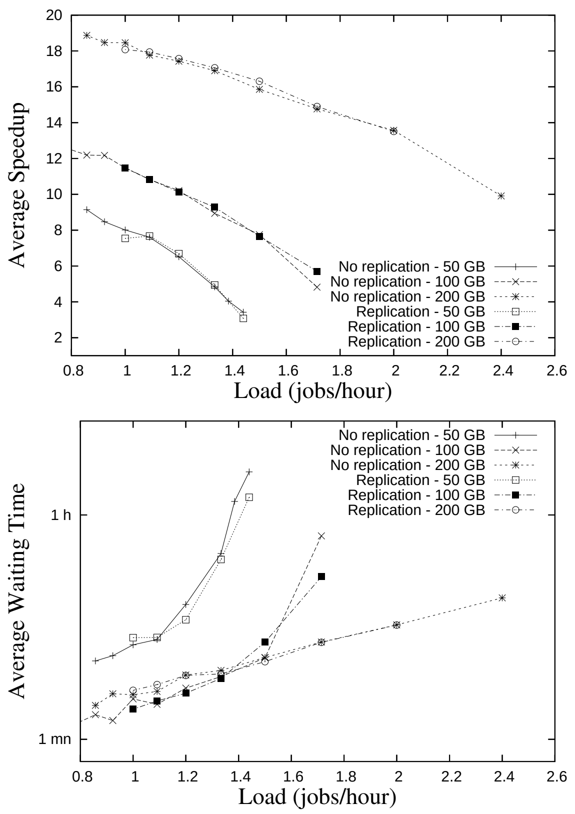


Figure 6: Gain in speedup and waiting time by considering replication versus no replication of data items for the job out of order scheduling algorithm

job of each busy period will be split onto all nodes. Even jobs starting on a single node will, after some time, take advantage of the nodes released by other terminated jobs.

Therefore, there is never a large continuous segment of data residing within a single node disk cache. A large data segment is always split onto several nodes. When a new job requiring a large data segment arrives, it will be immediately split onto several nodes, ensuring a high degree of load balancing.

The case where a node is overloaded compared with other nodes, i.e. where data segment replication would improve the performance, occurs very seldom. The detailed analysis of the simulation reveals that data replication is used in less than 1 ‰ of the job arrivals.

5 Toward cluster-optimal scheduling

We consider a policy to be optimal from a cluster utilisation point of view if it is able to sustain a larger load than any other policy. In our context, the load is defined as the mean number of job arrivals per time interval, all other conditions being identical.

With this definition of optimality, one can enumerate the properties of an optimal scheduling policy. The maximal load occurs when all data is cached and all nodes are fully utilised. This however can never happen if the total disk cache in all nodes is less than the total data size. Thus an optimum can be achieved if data is loaded at most once from tertiary storage. An optimal behaviour supposes that the scheduler has a complete knowledge of all future jobs in order to schedule jobs before their data segments are removed from the cache. In other words, an optimal policy is an off-line policy.

5.1 Delayed scheduling

The previous statement does not help much in an on-line context. But it shows a tendency : the more we know about future jobs, the better we can schedule them.

This leads to the definition of a *delayed scheduling policy* where several jobs are scheduled at fixed time intervals. Time is divided into periods of equal size during which jobs are accumulated without being scheduled. They are then scheduled at once at the end of the period and processed during the next period. A data segment *stripe size* is also defined as being the largest acceptable size of a data segment associated to a subjob. We use different values for the stripe size, ranging from 200 to 25000 events. Table 4 describes the delayed scheduling policy.

Table 4: Details of the delayed scheduling policy

Each node maintains a queue of subjobs. These subjobs only need data that is cached on their node. An extra queue contains meta-subjobs with no cached data. A meta-subjob is an aggregation of subjobs requiring overlapping data segments.
At the end of a period, all waiting jobs need to be scheduled. Jobs are scheduled as follow :

- Each job is split into subjobs so as to ensure that each subjob data segment is either fully cached on a node or not cached at all. There is a lower limit on the subjob size.
- Subjobs with cached data are queued on the corresponding nodes.
- The other non-cached subjobs are further split as follows :
  - A list defining data segment start and end points of subjobs is built
  - Points creating stripes below half the “stripe size” are removed. Points are also added so as to ensure that no stripe is above the “stripe size”
  - The final list of points is used to split the subjobs into subjobs having a number of events equal to or lower than the stripe size.
- Non cached subjobs working on the overlapping data segments are gathered into a single meta-subjob. Note that the size of the meta-subjob data segments is defined by the stripe size.
- Meta-subjobs are queued according to their arrival time so as to introduce fairness in their order of execution. The arrival time of a meta-subjob is defined as the earliest of its subjobs arrival times.

Once the queues are filled, a new period starts during which the subjobs are processed as follow :

- Nodes run in priority the subjobs located in their private queue.
- If a node’s queue is empty and the node becomes idle, it pops the first meta-subjob from the queue of non cached meta-subjobs and places every subjob contained in it into its queue. By construction, these subjobs are all requiring a partially common contiguous data segment which is not present on the node and which will be loaded from tertiary storage.

The goal of the delayed scheduling policy consists in loading the data from tertiary storage only once during a given period. In addition, the “stripe size” parameter controls the average size of a subjob and thus on how many nodes a job may be distributed.

5.2 Results and comparison with the out of order scheduling policy

The comparison between the out of order and the delayed scheduling policies is not obvious since they don’t try to achieve the same goal. The out of order scheduling policy, despite its name, is still pretty fair concerning the job execution order. On the contrary, the delayed scheduling policy only focuses on cluster utilisation optimisation. The average speedup will thus be lower since many jobs requiring non cached data may stay idle during long periods while other jobs are running on cached data (no fairness). For the same reason, their waiting time will also be worse. In addition the waiting time becomes longer due to the fact that jobs have to wait until the end of a period before being scheduled. This extra “period” delay is not included in the waiting time shown in the figures.

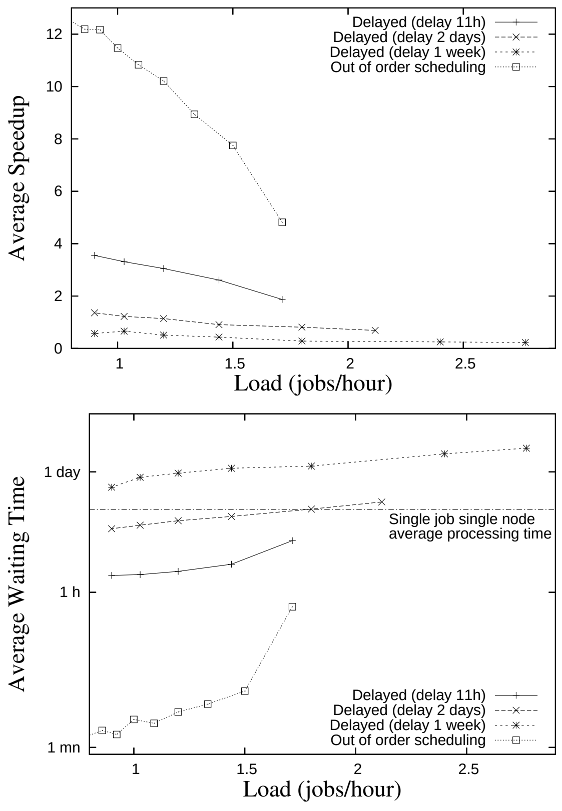


Figure 7: Speedup and waiting time (period delay excluded) of the delayed scheduling policy for different delays (cache size 100 GB, stripe size 5000 events)

Figure 7 shows the results of the simulation for delayed scheduling with different "period" delays. As in Figure 3, the curves in Figure 7 are cut at high loads when the cluster becomes overloaded i.e. when queues start growing indefinitely. For comparison purposes, the out of order scheduling policy is also shown. Delayed scheduling behaves poorly both in terms of average speedup and average waiting time. On the other hand, it allows to sustain very high loads, especially if the "period" delay is large (up to 1 week for 9 h jobs). However the total waiting time becomes really high if one takes into account the "period" delay.

The influence of the stripe size on the delayed scheduling algorithm performance is presented in Figure 8. It shows a very clear improvement in term of speedup for small striping values and no influence at all on the average waiting time. This reinforces the idea that the parallelization potential is better exploited with smaller stripe sizes. A larger average speedup allows to sustain higher loads.

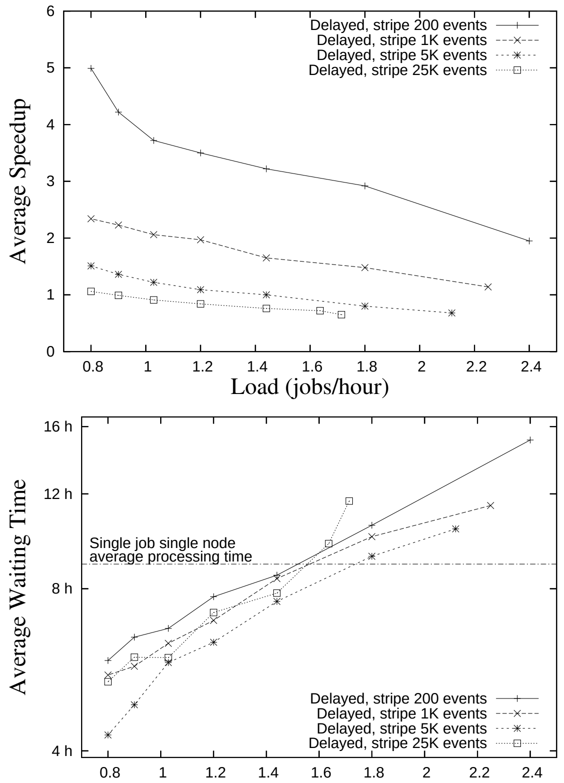


Figure 8: Average speedup and waiting time (delay excluded) of delayed scheduling for different stripe sizes (cache size 100 GB, period delay 2 days)

A summary of the maximal sustainable loads under different period delays, cache sizes and stripe sizes is given in Figure 9. We can observe an almost linear dependency of the maximal load with respect to both the delay and the stripe size. The more we wait and the finer we split the jobs, the larger the maximal load we can sustain. The simulation experiments show that a maximal load of 3 jobs per hour can be reached by using 200 gigabytes of disk cache, 1 week of delay and a stripe size of 200 events. This maximal load can be compared with the maximal theoretical load of 3.46 jobs per hour. It is close to 3 times the load of 1.1 jobs per hour sustained by processing farm scheduling without disk caching (see sections 3.1 and 3.4).

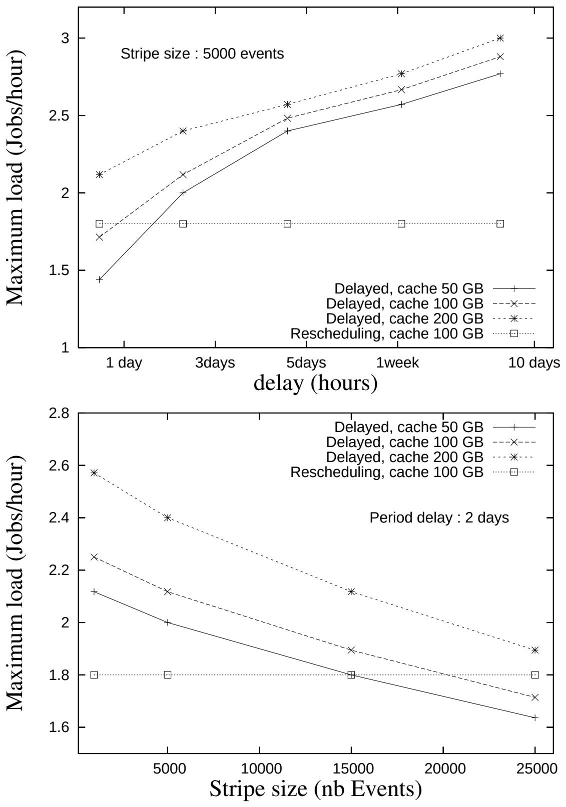


Figure 9: Maximal sustainable load of the delayed scheduling policy with different cache sizes in function of the period delay and the stripe size (stripe size on the left : 5000 events, period delay on the right : 2 days)

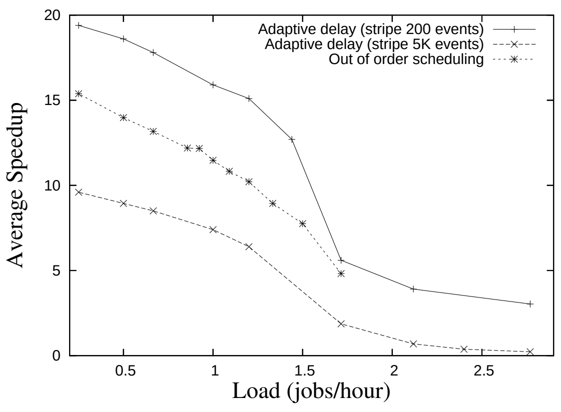
6 Adaptive delay scheduling

As shown in the previous section, large period delays allow to sustain much higher loads at the cost of excessive waiting times for the end-users.

We define here a new adaptive delay policy that aims at minimising the waiting time, while sustaining the current load. This policy makes use of the performance parameters shown in Figures 7, 8 and 9 in order to choose the minimal "period" delay that allows to sustain the current load.

Figure 10 shows the performance of adaptive delay scheduling compared to the out of order scheduling policy. As in previous figures, the curves are cut at high loads when the cluster becomes overloaded. As expected, the use of delayed scheduling allows the adaptive delay policy to sustain loads that the out of order scheduling policy cannot sustain.

At low loads and for small stripe sizes, the adaptive delay policy is performing in terms of speedup as well or slightly better than the out of order scheduling policy. At these low loads,



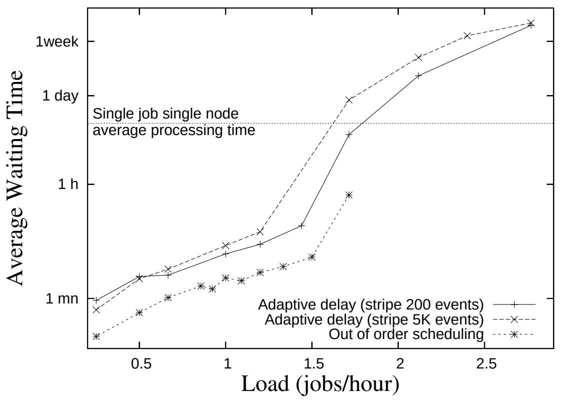


Figure 10: Speedup and waiting time (delay included) of the adaptive delay policy for different stripe sizes (cache 100 GB) compared with the out of order scheduling policy

the "period" delay is actually reduced to zero. The counterpart is a little overhead (up to 1h) in the average waiting time. However, this overhead is not really significant when compared to the single job single node average processing time (9h).

One may wonder why there is still a difference between the out of order scheduling policy and the delayed policy at low loads (where the "period" delay is zero). This is a consequence of the different data distribution strategies. In the out of order scheduling policy, the data distribution is a side-effect of the parallelization that occurs when computing nodes are idle. However priority is given to starting new jobs over parallelizing (splitting) running jobs. In the delayed scheduling policy, in most of the cases, the data distribution is triggered by a predefined stripe size. Thus, for small stripe sizes, the level of parallelization of delayed scheduling is higher, leading to larger speedups. On the other hand, since jobs are in most cases split into subjobs running in parallel, and since only one subjob runs on one node at a given time, the number of concurrently running jobs becomes smaller, leading to longer waiting times.

7 Pipelining processing and I/O operations

In the previous sections, we considered that I/O operations are performed serially, i.e. processing operations can operate on data after the disk accesses have been carried out. However modern computing systems allow to asynchronously perform I/O and processing operations and to hide the shortest of the two activities. We consider two types of pipelining :

- pipelining of tertiary storage access and computation : the access to tertiary storage and the processing will be performed asynchronously
- pipelining of local disk I/O, tertiary storage access and computation : reading data segments from the disk caches is pipelined with the processing activities.

7.1 Pipelining accesses from tertiary storage and event processing

The computation of a given event can be carried out in parallel with the loading of the next event from the tertiary storage (if the next event is not cached). With this strategy, we expect to be able to hide completely the computation activity for non cached events. The theoretical speedup for an event read from tertiary storage is $\frac{T_{CPU}+T_{I/O}}{T_{IO}} = \frac{0.2+0.6}{0.6} = 1.33$, i.e. a performance improvement of 33%.

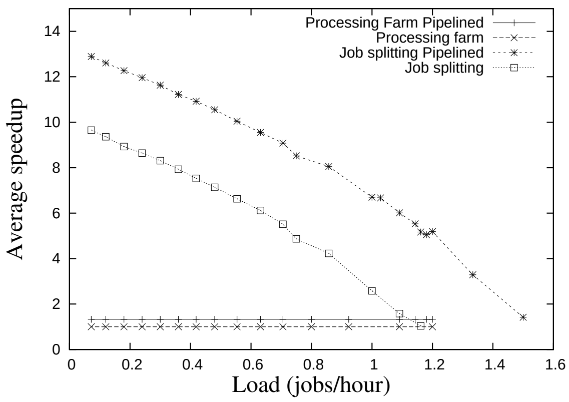


Figure 11: Average speedup for different scheduling policies with and without pipelining of tertiary storage accesses, deduced from simulations.

Figure 11 shows the improvement in speedup due to pipelining according to simulations for both the processing farm and the job splitting policy. The theoretical value of 33% is confirmed for the processing farm policy : the speedup is stable at 1.33 with pipelining.

For the job splitting policy, the theoretical speedup improvement of 33% is confirmed at low loads. We obtain speedup of 13 with pipelining and of 9.75 without pipelining. At higher loads, the speedup improvement is higher than the theoretical value. This is mainly due to a virtuous cycle starting with the reduction of the average time spent in the cluster by the jobs. Whenever a job spends less time in the cluster, it leaves its nodes earlier. The other running jobs can take advantage of these nodes to increase their parallelization degree and run faster. They will in turn stop earlier, leaving even more nodes for the next jobs.

At higher loads, the speedup of the job splitting strategy decreases linearly with the load along the same slope as in the case with no pipeline. Figure 11 shows that the speedup improvement of 33% at low loads becomes 50% for 0.5 jobs per hour, 100% for 0.9 jobs per hour and 300% for 1 jobs per hour. The maximum sustainable load of the cluster using the job splitting policy increases from 1.15 to 1.5 jobs per hour.

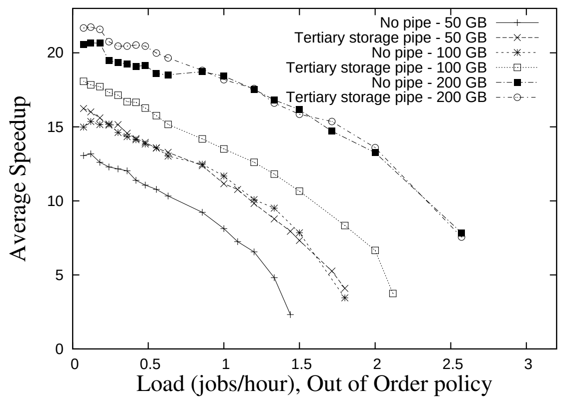


Figure 12: Average speedup for the out of order scheduling policy with and without pipelining of tertiary storage accesses for different disk cache sizes and different load levels, deduced from simulations

Figure 12 shows the gain of pipelining for the out of order scheduling policy according to simulations. The relative speedup gain is proportional to the amount of data that is loaded from tertiary storage. This leads to large improvements in case of a small disk cache and no improvement at all when all data is located within the disk caches. The maximum load supported by the cluster is also improved in the case of small disk caches. From a performance point of view, pipelining has the same effect on the out of order scheduling policy as increasing the disk cache size.

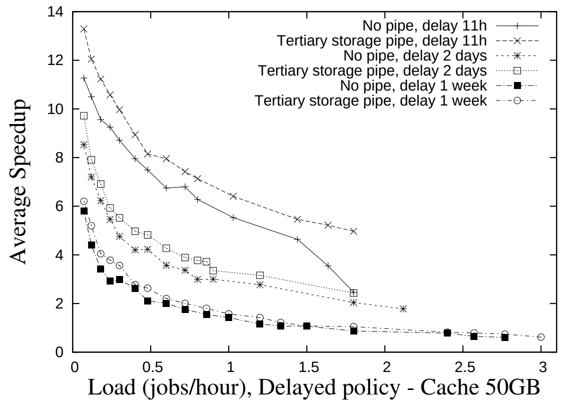


Figure 13: Average speedup for the delayed scheduling policy with and without pipelining of tertiary storage accesses for a small disk cache and different load levels

Figure 13 shows the gain of pipelining the delayed scheduling policy according to simulations. The disk cache size was fixed to 50GB here and different delays were simulated. The improvements are also proportional to the proportion of data read from tertiary storage. Since the delayed scheduling policy makes a heavy use of disk caches, improvements are only present for small delays. Note that the same simulations with a disk cache of 100GB show no improvement at all, even for small delays.

7.2 Pipelining of cache disk I/O

Pipelining of cache disk I/O and event processing allows to reduce the overall processing time of the cached events. Namely, reading cached data from disks can be completely hidden by event processing.

The theoretical speedup for an event read from disk is $\frac{T_{CPU}+T_{I/O}}{T_{CPU}} = \frac{0.2+0.06}{0.2} = 1.3$. Pipelining of cache disk I/O brings no speedup when the event is loaded from tertiary storage. However, in such a case, pipelining from tertiary storage access brings an important speedup. The two pipelining capabilities are therefore complementary.

Note that disk I/O pipelining makes no sense in the case of the computing farm and job splitting policies since, for these policies, data segments are always read from tertiary storage.

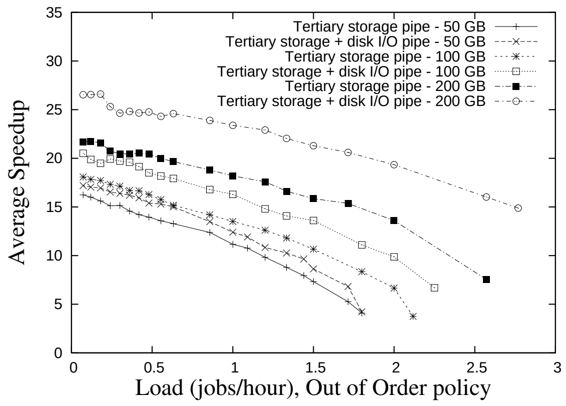


Figure 14: Average speedup for the out of order scheduling policy with tertiary storage pipelining only and with both tertiary storage and disk I/O pipelining for different disk cache sizes and different load levels

Figure 14 shows the gain of pipelining disk I/O and tertiary storage access in the case of the out of order policy compared to the case where only the access to tertiary storage is pipelined. The simulations show that the gain is large for large disk caches where most of the data is read from the disk caches while it is small for small disk caches where data is mainly read from tertiary storage. Since the performance gain of disk I/O pipelining is complementary to the gain obtained by tertiary storage pipelining, the total gain in speedup due to both disk I/O and tertiary storage pipelining is quite constant, around 30% as shown in Figure 15.

8 Complexity and Scalability

The present contribution aims at optimising the scheduling of jobs on large clusters of nodes. Since the scheduling system

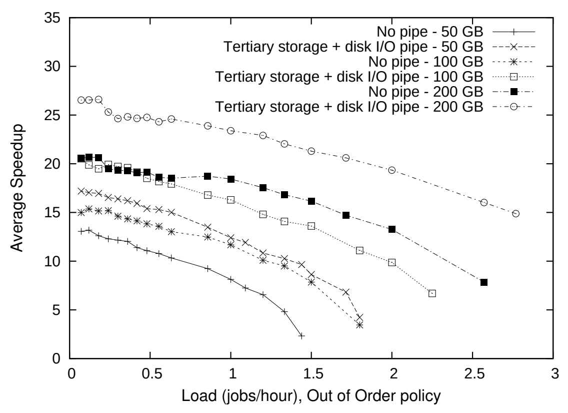


Figure 15: Average speedup for the out of order scheduling policy with no pipelining and with both tertiary storage and disk I/O pipelining for different disk cache sizes and different load levels

runs on a single central node, the scalability of the system depends on the complexity of the scheduling algorithms both in term of scheduling time and memory space.

8.1 Time complexity of the scheduling algorithms

The theoretical time complexity of each of the presented scheduling algorithms can be derived by a careful analysis of their implementation. Table 6 presents the main lines of this analysis leading to the scheduling time complexities per job summarised in Table 5. The complexities are given with respect to the number *n* of computing and storage nodes present in the system.

Policy	Complexity
Processing Farm	$O(n)$
Job Splitting	$O(n^2)$
Out of order	$O(n\sqrt{n})$
Delayed Scheduling	$O(n)$

Table 5: Complexities for scheduling one job according to the different scheduling policies in function of the number *n* of nodes in the cluster

Table 5 gives the complexity of scheduling one job as a function of the number *n* of nodes in the cluster. However, in a real case, the number of jobs processed per unit of time is also dependent on the number of nodes, typically proportional to it. Taking this fact into account, theoretical complexities have to be multiplied by *n* for a comparison with simulation data. Figure 16 shows the average load of the master node as a function of the number of slave nodes under the following conditions :

1. The cluster load is proportional to the number of slave nodes present in the cluster

2. The total data space size is proportional to the number of slave nodes

3. For delayed scheduling, the period delay is inversely proportional to the number of slave nodes. Thus the average number of jobs arriving into the cluster within a period is not dependent on the number of slave nodes. The delay used in Figure 16 is one month for a one node cluster, corresponding to e.g. 3 days for a 10 nodes cluster and 7 hours for a hundred nodes cluster.

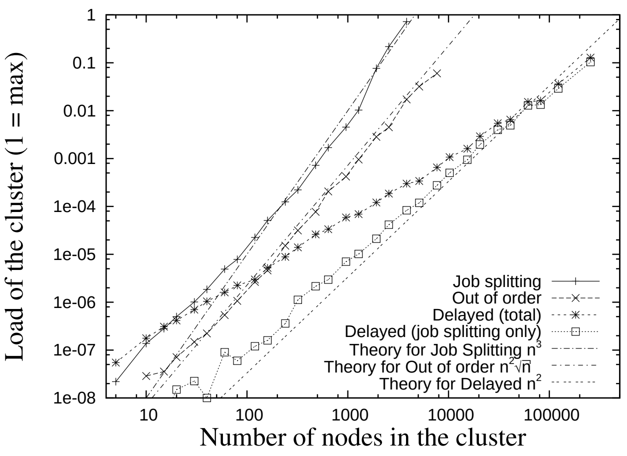


Figure 16: Average load of the master node as a fraction of the maximum load in function of the number of nodes in the cluster. The tests run on a single 2.0GHz Intel Celeron processor, the delay for delayed scheduling is one month divided by the number of nodes

Figure 16 shows both the simulation data (marker lines) and the theoretical models (dotted, without marker lines). The scheduling time complexity gives only the slope of the lines and their position is derived from the simulation data.

Simulations and theory match well for the job splitting and out of order policies. In the case of the delayed policy, the load represents both the subdivision of data into the desired stripe size and job splitting. Thus an extra curve of simulated data shows the time spent only in the job splitting part of the delayed scheduling algorithm. It clearly shows that the complexity of the job splitting part matches the theoretical complexity. However, the simulations show that the job splitting part becomes preponderant only for a large number of nodes, around 50000.

From the theoretical extrapolations of the simulated data, the expected maximum number of nodes that a single master node can handle is around 5000 for the job splitting policy, around 20000 for the out of order scheduling policy and around 300000 for the delayed scheduling policy. The period delay introduced by the delayed scheduling policy reduces the job scheduling load for very large clusters by an order of magnitude.

8.2 Space complexity

The space complexity is bound by the amount of data needed by the master node to schedule the jobs, namely the list of nodes and the status of their disk caches (when applicable). The required memory space is linear with the number of nodes and remains relatively small :

Table 6: Single job theoretical time complexities of the different scheduling policies in function of the number n of nodes in the cluster.

Processing Farm	**Job Splitting**
Scheduling one job requires traversing the list of nodes to find an idle node. Scheduling complexity is $O(n)$.	Upon job arrival, the worst case requires scanning all nodes to find the subjob to be suspended. Complexity of this case is $O(n)$. Upon job or subjob end, the worst case consists in finding the subjob to split. By looking at all of them (one per node) the complexity is $O(n)$ per split. We assume here that the number of jobs in the system depends only on the load, and is independent of the number of nodes n. Since the jobs are parallelized across all nodes, the number of splits per job is $O(n)$. Since the number of splits per job is $O(n)$ and each split operation is $O(n)$, the overall complexity of single job splitting is $O(n^2)$.
Out Of order	**Delayed Scheduling**
Upon job arrival, the new job has to be split into subjobs. Splitting takes caches into account by reading the content description of each node cache, yielding a complexity of $O(n)$. When a node becomes available, a subjob needs to be split. By looking at all of them (one per node), the complexity is $O(n)$ per split. The global complexity per job is thus of the order of $O(n)$ multiplied with the average number of splits per job, i.e. the average number of subjobs. Let $j(t)$ be the number of jobs running in the cluster and $s(t)$ the average number of subjobs per job at time t. If the cluster is not idle, the job splitting algorithm ensures that all nodes have a running subjob. Thus : $\forall t > 0 \quad j(t)\, s(t) \simeq n$ The simulation shows that $s(t) = O(\sqrt{n})$ and $j(t) = O(\sqrt{n})$. Therefore the scheduling complexity is $O(n\sqrt{n})$.	At each new period, the new jobs are split into subjobs. Job splitting takes caches into account by reading the content description of each node cache, yielding a complexity of $O(n)$. The non cached subjobs are further split. The number of splits per job is given by the stripe size and is independent of the number of nodes yielding a complexity of $O(1)$. The overall complexity per job is thus $O(n)$.

- The list of nodes is not large compared with the representation of the information present in the disk caches
- The size of the representation of the information contained in a disk cache is in the order of the cache size divided by the stripe size. The maximal cache size being around a few terabytes and the stripe size comprising a few events i.e. a few megabytes, the size of the representation of cached information is a few megabytes.

We may therefore store the information contained in two thousands very large disk caches (each disk 500 GB yielding a total size of 1 PB) divided into very small stripes within 1GB of RAM. For a regular stripe size, the scalability limitation due to memory limitations on the master node is of the order of tens of thousands of slave nodes in the cluster.

9 Conclusions

We propose several scheduling policies for parallelizing data intensive particle physics applications on clusters of PCs. Particle physics analysis jobs are composed of independent subjobs which process either non-overlapping or partly overlapping data segments. Jobs comprise tens of thousands of collision events, each one requiring typically 200 ms CPU processing time and access to 600 KB of data.

We show that splitting jobs into subjobs improves the processing farm model by making use of intra-job parallelism. By caching data on the processing farm node disks, cached-based job splitting further improves the performances.

The out of order job scheduling policy we introduce takes advantage of cache resident data segments and still includes a certain degree of fairness. It offers considerable improvements in terms of processing speedup, response time and sustainable loads. For the same level of performance, the typical load sustainable by the out of order job scheduling policy is double the load sustainable by a simple first in first out cache-based job splitting scheduling policy.

We propose the concept of delayed scheduling, where the deliberate inclusion of period delays further improves the disk cache access rate and therefore enables a better utilisation of the cluster. This strategy is very efficient in terms of the maximal sustainable load (50 to 100% increase) but behaves poorly in terms of response time and processing speedup. In order to offer a trade-off between maximal sustainable load and response time, we introduce an adaptive delay scheduling policy with large delays at high loads and zero delays at normal loads. The delay is adapted to the current system load, thus trying to optimise the response time as a function of the current load. This adaptive delay scheduling policy aims at satisfying the end user whenever possible and at the same time allows to sustain high loads.

Pipelining computation and accesses to tertiary storage and to the local disk caches further increases the performances and allows the system to sustain higher loads.

The job scheduling complexity analysis, both in respect to running time and memory space shows that the proposed scheduling policies scale to clusters comprising thousands to tens of thousands of nodes.

The scheduling policies presented here aim at maximising the sustainable load of a processing cluster and at reducing as much as possible individual response times. In the future, we intend to study mixed scheduling strategies combining period delays, immediate processing of job requests and optimal pipelining of data accesses to both tertiary storage and disk caches.

REFERENCES

[1] M. Adler, Y. Gong, and A. L. Rosenberg. Optimal sharing of bags of tasks in heterogeneous clusters. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, Scheduling I, pages 1–10. ACM Press, 2003.

[2] S. V. Anastasiadis and K. C. Sevcik. Parallel application scheduling on networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):109–124, 1997.

[3] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. In *24th Annual ACM STOC*, pages 39–50, 1992.

[4] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–20, Mississippi State, MS, 1994. IEEE Computer Society Press.

[5] V. Bharadwaj, D. Ghose, and T. G. Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing, Special Issue on Divisible Load Scheduling in Cluster Computing*, 6(1):7–17, January 2003. Kluwer Academic Publishers.

[6] CERN web pages. A matter of symmetry. http://lhcb-public.web.cern.ch/lhcb-public/html/symmetry.htm.

[7] CERN web pages. The Castor project. http://castor.cern.ch.

[8] CERN web pages. The LHCb Experiment. http:// lhcb-public.web.cern.ch/lhcb-public.

[9] CERN web pages. What is CP-violation? http://lhcb-public.web.cern.ch/lhcb-public/html/introduction.htm.

[10] R. Fleischer and S. S. Seiden. New results for online page replication. In *Proceedings of the International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX '00)*, pages 144–154. Springer Lecture Notes in Computer Science, 2000. ISBN : 3-540-67996-0.

[11] T. Galla. *Cluster Simulation in Time-Triggered Real-Time Systems*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 1999.

[12] I. Kadayif, M. Kandemir, I. Kolcu, and G. Chen. Locality-conscious process scheduling in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, System design methods: scheduling advances, pages 193–198. ACM Press, 2002.

[13] J. Kaplan and M. L. Nelson. A comparison of queueing, cluster, and distributed computing systems. NASA Technical Memorandum 109025, NASA LaRC, october 1993. http://citeseer.nj.nec.com/kaplan94comparison.html.

[14] Kleinrock. *Queueing Systems*. Wiley-Interscience, 1976.

[15] J. Myllymaki and M. Livny. Efficient Buffering for Concurrent Disk and Tape I/O. *Performance Evaluation*, 27/28(4):453–471, 1996.

[16] P. Triantafillou and C. Faloutsos. Overlay striping and optimal parallel I/O for modern applications. *Parallel Computing*, 24(1):21–43, 1998.

[17] M. Uysal, T. M. Kurc, A. Sussman, and J. H. Saltz. A performance prediction framework for data intensive applications on large scale parallel machines. In *Proc. of 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, volume 1511 of *Lecture Notes in Computer Science*, pages 243–258. Springer-Verlag, 1998.

[18] B. B. Zhou, R. P. Brent, D. Walsh, and K. Suzaki. Job scheduling strategies for networks of workstations. *Lecture Notes in Computer Science*, LNCS 1459:143–157, Springer-Verlag, 1998.