SOME WORKLOAD SCHEDULING ALTERNATIVES IN A HIGH PERFORMANCE COMPUTING ENVIRONMENT

Tyler A. Simon, University of Maryland, Baltimore County
James McGalliard, FEDSIM

Abstract

Clusters of commodity microprocessors have overtaken custom-designed systems as the high performance computing (HPC) platform of choice. The design and optimization of workload scheduling systems for clusters has been an active research area. This paper surveys some examples of workload scheduling methods used in large-scale applications such as Google, Yahoo, and Amazon that use a MapReduce parallel processing framework. It examines a specific MapReduce framework, Hadoop, in some detail. It describes a novel dynamic prioritization, self-tuning workload scheduler, and provides simulation results that suggest the approach will improve performance compared to standard Hadoop scheduling.

1. INTRODUCTION

This paper describes an approach to workload scheduling on commodity cluster HPCs along with evidence that the approach can provide improved performance compared to the default scheduler in a particular environment. The paper begins by discussing common workload scheduling methods for commodity clusters, including Backfill (Section 1). It goes on to describe the MapReduce framework now used for some very large applications that run on cluster HPCs and a specific implementation of this framework, Hadoop (Section 2). Papers describing diverse alternative workload scheduling approaches for MapReduce and Hadoop have proliferated in recent years, and Section 3 surveys representative examples of these approaches. Section 4 describes a novel approach along with evidence that this approach can improve performance.

In their original incarnations, supercomputers used custom processor designs optimized for executing arithmetic instructions – addition, subtraction, and multiplication – very fast on data gathered into special purpose vector registers akin to a column of cells in a spreadsheet. Typical supercomputer application workloads included science or engineering problems where some physical system was represented by a multi-dimensional matrix of numbers. Each cell in the matrix represented some element of the physical system and the behavior of the system was simulated by performing arithmetic and logical functions on data stored in those cells.

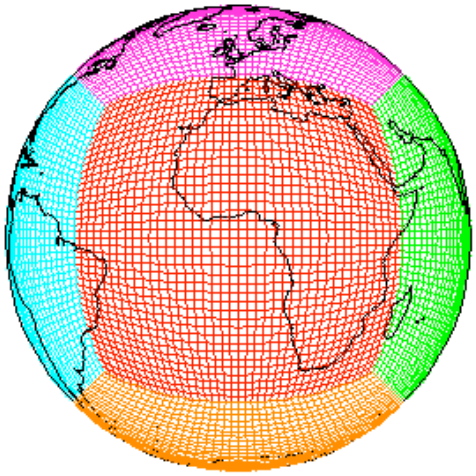

Figure 1. Cubed Sphere Mapping

Figure 1. Is an example of such a physical system – the Earth's atmosphere – mapped into a matrix of cells, illustrated by the grid superimposed on an image of the globe. This Cubed Sphere mapping is used in weather and climate forecasting [Glassbrook].

In time, the high cost of custom supercomputer CPUs, coupled with their low production numbers – perhaps a few dozen annually – was overcome by the low cost of commodity microprocessors sold in the millions of copies. The price/performance of commodity cluster systems drove custom-designed systems out of the market. Current generation supercomputers may have 1,000 or 10,000 commodity microprocessor cores[1] harnessed for application to problems like simulation of the Earth's climate by mapping the atmosphere into a matrix of thousands of cells.

The workload scheduler and application software assign groups of adjacent cells in this matrix to the many cores in the system. Each core solves the simulation for some subset of the system represented – say, a chunk of the atmosphere over some land and ocean on the west coast of Africa – and the system gathers all the individual results into an integrated picture of the Earth's climate as a whole. In this way, cluster supercomputers solve large problems by dividing them into small pieces.

1.1 HPC WORKLOADS

High performance computers (HPCs) include the vector supercomputers of the 60s and 70s; exotic systems such as the [Butterfly], [Multiflow] and [Connection Machine]s of the 80s and 90s; and the commodity cluster systems of the last 15 years. Common design goals of HPC systems are:

- Fast execution of integer and floating point arithmetic operations
- Parallelism

HPC workload examples include

- Mathematical systems – systems of equations such as linear equations, partial differential equations, and Fourier analysis
- Physical systems – in domains such as particle physics, molecular physics, weather, fluid dynamics, and structure design
- Logical systems – including, more recently, patterns of web activity, as will be discussed further in this paper.

As stated above, in many such applications, the system under study is represented as a multi-dimensional grid and the behavior of that system is simulated by mathematical operations executed independently on each grid point and summarized into the behavior of the system as a whole.

Using the same code to simulate the behavior at thousands of grid points makes application software development tractable – without parallelism, writing discrete software for each grid point would be too complex and the code would never get written. Although small jobs (few processors, short duration) are the most common in HPC data centers, the design goals of HPCs are to run large jobs using many processors (e.g., a thousand) with long run times (e.g., days).

1.2 WORKLOAD DISPATCHING IN CLUSTER HPCS

Because it is a scarce and expensive resource, system designers want to optimize the use of central processing units and not leave them idle. An important CPU optimization method in a traditional mainframe is to swap a new workload into the processor when the current workload no longer requires it – say, at the start of an input/output operation. While the processor waits for roughly 10 milliseconds for an I/O operation to complete, it could be busy executing 50 million or so instructions, so waiting is expensive.

The tradeoff is different in a cluster supercomputer, where a given workload may be using 1,000 CPU cores simultaneously. Swapping a single core to another workload can leave the other 999 cores idle, not an efficient use of that hardware. For this reason, cluster supercomputer workloads typically run to completion and without interruption (but more on that below).

Optimization of the code within a single job is primarily the responsibility of the application programmer. Optimization of the system as a whole is the responsibility of the system administrator. ([Heger], [Herodotou], and [Rao] are examples of this type of optimization in the MapReduce and Hadoop environments.) When jobs use multiple processors and run uninterrupted to completion, sequencing the assignment of jobs to processors so as to increase throughput and utilization without unduly harming response time is a key system optimization consideration. This paper focuses on performance optimization of the job scheduling subsystem, one of the provinces of the sys admin and the system software developer.

Figure 2 depicts the most common, default HPC scheduling algorithm, first-come, first served (FCFS). The figure can be interpreted as follows. There are

[1] For the purposes of this paper: *Cores* = *Processors* = central processing units, including the logic needed to execute the instruction set, registers & local cache. *Node* = a board with one or more processors and local memory, with disk and the network attached. *Task* = application software that can run on at most one core. *Job* = one or more tasks, constituting a complete program.

multiple processors available on the system, represented by the vertical axis. The upper horizontal line represents the total number of processors on the system, which is the maximum number that any job can use. Individual jobs are represented by the rectangular boxes – J1 is Job 1, the first to arrive and the first to be dispatched. J2 is Job 2, the second to arrive, and so on. The horizontal axis represents time, which flows from left to right. J1 requires less than half of the total processors available on the system and runs for a relatively short time. J2 requires about twice as many processors as J2 and runs for about twice as long. J3 is smaller than J1 in both dimensions; J4 uses the same number of CPUs as J1 but runs longer than J1.

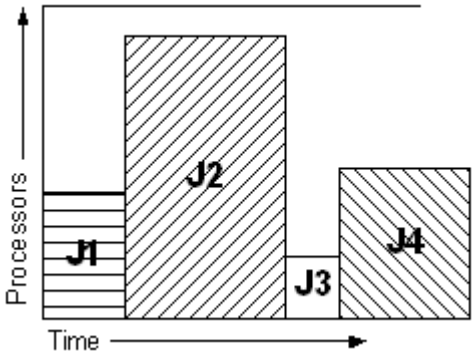

Figure 2. First Come, First Served Scheduling

The blank space on the upper side of the diagram represents wasted, idle processors. FCFS is fair and easy to understand, but is an inefficient user of scarce CPU resources. To improve efficiency, most HPC data centers use some form of what is called *Backfill*, which is illustrated in Figure 3.

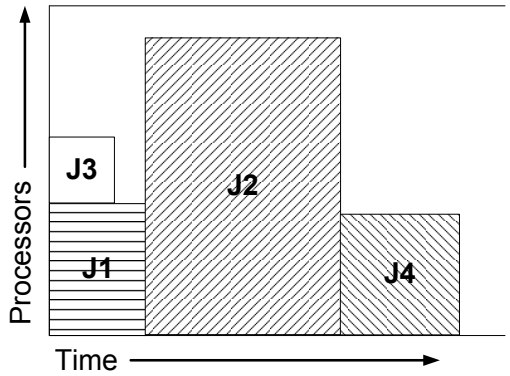

Figure 3. Backfill

Figure 3 is formatted the same as Figure 2 – the same number of CPUs, the same flow of time from left to right, and the same 4 jobs arriving in the same sequence. The difference is that in Figure 3 the workload scheduler looks ahead in the queue and finds a way to use idle CPU time productively. Although it isn't possible to run J1 and J2 simultaneously, because taken together they need more CPUs than are available on the system, it is possible to schedule J3 to run alongside J1. Taken together, J1 and J3 will not need more CPUs than are available. Importantly, J3 will complete before J1 if they are both dispatched at the same time, so large job J2 will not be delayed. J2's owner can't complain that he has been treated unfairly, as he will run as soon as he would have under pure FCFS.

Inspecting the unused CPU time in the upper left corner of Figure 3, it is evident that even with Backfill, the processor is not running at maximum efficiency. In this situation, further improvements in processor utilization are possible, although at a cost in terms of fairness.

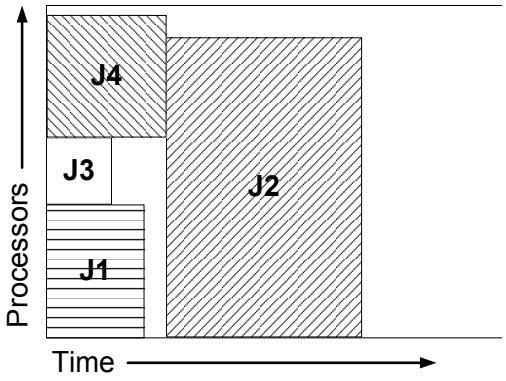

Figure 4. Relaxed Backfill

Figure 4 illustrates a policy called Relaxed Backfill. It is possible to dispatch three jobs simultaneously – J1, J3 and J4. Taken together, all three jobs will use almost but not quite all of the available CPUs on this HPC system. The problem is that, unlike J3, J4 will take longer to run than first-arriving J1. Because J4 runs longer than J1, the start of second-arriving job J2 will be delayed, a violation of fairness. The narrow open space between J1 and J2 in Figure 4 is that delay.

Some form of Backfill is used in many HPC data centers. Backfill requires some method of assigning priorities to jobs, of which wait time is only one example. Section 3.1, below, surveys some factors that can be used to set job priorities. Section 3.2 describes some alternatives to Backfill once the priorities have been set. Section 3 is focused on alternatives specific to the MapReduce framework, which the following section briefly describes.

2. MAPREDUCE AND HADOOP

2.1 MAPREDUCE

Popular global internet enterprises like Google, Facebook and Yahoo need applications that can process the vast amount of data that these enterprises produce and harvest. This is called data

intensive scalable computing (DISC). Unlike the science and engineering HPC applications of the first generation, these applications need text processing capabilities as well as logic and mathematical operations. The commodity microprocessor cores found in cluster HPCs aren't designed to boost mathematics operations as much as their vector-processing ancestors and were designed to handle text to suit the needs of mass market PC users, a capability that DISC uses to its advantage. (Note: on the other hand, current hybrid HPC systems with commodity and graphics co-processors *are* optimized for math.)

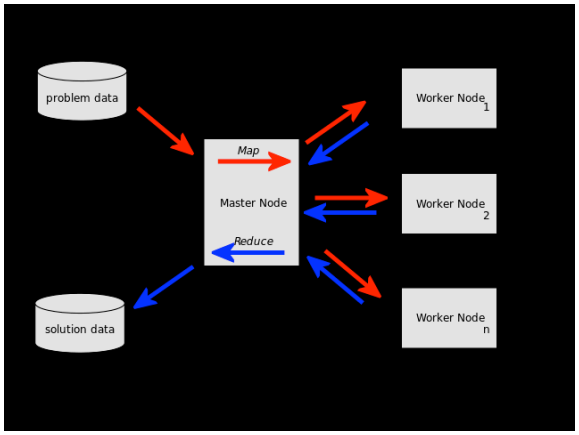

Figure 5. MapReduce Schematic[2]

A decades-old problem in HPC is that parallel code writing tools lag far behind the parallel hardware that runs this code. In the current generation of cluster HPCs, the MapReduce programming framework is one approach to the parallel programming problem (see Figure 5).

In a MapReduce framework, the system software provides callable routines that take a large problem and its input data sets ("problem data" in Figure 5) and divides it into many parallel chunks. This is the map() function. Chunks of data are copied onto disks that are distributed across the cluster's racks and nodes. At each node, that subset of the problem data is processed into an intermediate result. After all nodes have completed this map phase, the reduce phase begins.

In the reduce phase, data collected from the map nodes based on key values are reduced from the intermediate into a final output ("solution data" in Figure 5). Like weather forecasting on the Cubed Sphere, large problems can be broken into small, independent pieces, mapped onto and then processed on many cluster nodes, and finally assembled into a coherent overall solution. The map() and reduce() functions assist the application programmer with the daunting parallelization problem.

MapReduce frameworks have the useful property that jobs can be dispatched with fewer cores than the user requests. Each map() task runs independently, and once a core finishes its assigned task, it can be scheduled to run another. If the scheduler allocates fewer CPUs than the user requested, a CPU that finished its first task can be assigned to run another one, compensating for the initial fractional dispatching. With fractional dispatching, it's easier to fill all available CPU slots with useful work, and backfill can achieve 100% CPU utilization more frequently. However, fractional dispatching can also result in longer run times.

2.2 HADOOP

Hadoop is an example of a MapReduce framework. It is written in Java and was originally created for use at Yahoo. It is named after a toy elephant owned by the creator's son. It is open source and is used by Facebook, Yahoo, LinkedIn and other large web enterprises. IBM's Watson HPC, which competed successfully on the Jeopardy quiz show against two of the world's best human contestants, uses Hadoop along with other fancy technology.

In addition to implementing map() and reduce(), Hadoop has a native file system, HDFS, that handles copying, sorting, and consolidating the massive data sets that MapReduce applications require. See Figure 6.

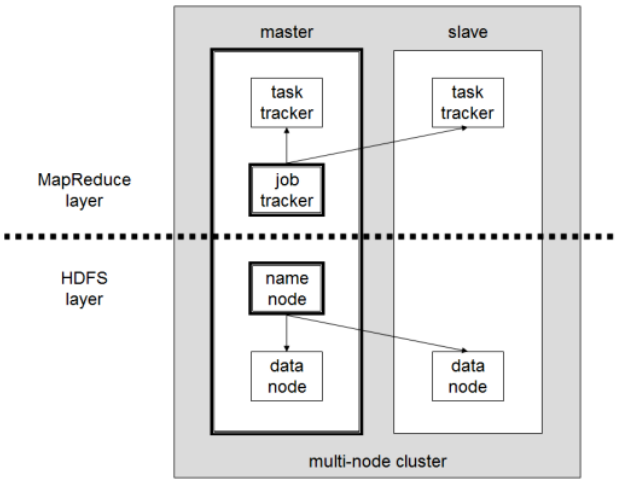

Figure 6. Hadoop Schematic[3]

The default Hadoop scheduler is First Come First Served (FCFS). As discussed in Section1.2, FCFS is

[2] Image licensed by GNU Free Software Foundation

[3] Source: Wikipedia, "Hadoop"

fair but inefficient. Frequent small jobs like development, test, and ad hoc queries get stalled behind infrequent and long running production jobs that use full-size input data sets. To improve performance, Facebook developed the Fair Share Scheduler alternative to FCFS. In Fair Share, incoming work is divided into pools, with typically one user per pool. Each pool is guaranteed a minimum share of the processor cores. As users submit their jobs, the scheduler assigns at least the minimum number of cores in their share or the number of cores the job requests, whichever is less. If unassigned processors survive this distribution, jobs requesting more than their guaranteed minima may receive the surplus. If a user submits more than one job, the competition for resources is handled within that user's pool, not disturbing the other pools' share of the system resources.

Some features of Hadoop that impact workload scheduling include the following.

Hadoop is aware of which nodes possess copies of which chunks of the problem data and attempts to schedule tasks to run on the same node as the data assigned to that task. The purpose of this data locality is to optimize network performance. Input/output operations local to a node have higher bandwidth than to a remote node and won't tie up the network backbone. In addition, if the data is not available on the local node's disk, Hadoop will search for a node on the same physical rack. Rack data locality is not as good as node locality, but is superior to I/O to a node on a remote rack. The cost of data locality, though, is the inability to assign the highest priority task to any nodes as soon as they complete their previous assignment, as those nodes may not possess the data local to that highest priority task.

Another Hadoop feature is that each task is dispatched redundantly on three nodes. The primary purpose of this dispatch redundancy is to provide fault tolerance – if one of the nodes running a particular task fails, at least one of the other two nodes will be able to take its place. Because of this fault tolerance, an Hadoop system can tolerate task pre-emption or killing better than other HPC architectures. Hadoop tasks can be interrupted, although interruption can cause delayed job completion. Hadoop systems are different from many cluster HPCs in this ability to tolerate job interruption.

3. IN SEARCH OF A WORKLOAD SCHEDULING TAXONOMY

In a 2002 paper, [Ward] speculated about the need for a taxonomy of the range of possible HPC workload policies. One can argue that the time for this taxonomy is not yet ripe, as new and diverse approaches to the problem are emerging rapidly, with no consensus yet apparent. Classification efforts such as the queueing system in [Feitelson1998] don't cover the entire scheduling space. However, we would suggest that two sides of the taxonomy coin will be (1) alternative methods for prioritizing jobs in the queue and (2) methods for scheduling jobs once prioritized. The next two sections survey these two aspects of HPC job scheduling.

3.1 JOB PRIORITIZATION

As [Ward], [Nguyen], [Feitelson1999] and others have proposed, jobs priorities can be set according to diverse parameters or conditions. In the discussion of Backfill in Section 1.2, we already have these three:

- Wait Time. In the default case, the longer the job has been waiting, the higher its priority becomes. If Wait Time is the sole priority criterion, this yields FCFS.

- Run Time. In many HPC environments, the user submitting the job includes the estimated job run time as a parameter of the submission script. Commonly, longer running jobs receive lower priorities than short jobs, with the administrative goal of lowering average response time across all jobs (short jobs are usually much more frequent than long ones) or lowering the average expansion factor. (The "expansion factor" is the sum of wait and run time divided by the run time – how much longer the job takes to run due to waiting, compared to running on an idle system.) If run time is the sole priority criterion, this can yield either shortest job first (SJF) or longest job first (LJF) schedules.

- Number of Processors. High performance systems are often classified as either "capacity" systems or "capability" systems. A capability system is designed to be capable of running a single job that is as large as possible (while also able to run many small jobs); a capacity system is designed to run the largest workload in terms of the sum of jobs of all sizes. In the discussion of Backfill, the ability to backfill a job was dependent on both the estimated run time and the number of processors required.

Some other prioritization alternatives include these.

- Queue. Many production HPC scheduling systems are structured in functional queues – such as development, debug, test, and production – with resource consumption targets and assigned workload class priorities. In chargeback or virtual budget-based data centers, prices per CPU hour vary by queue – the higher the priority, the higher

the hourly rate. This resembles the pricing structure of the timesharing systems of the 70s and 80s.

• Composite Priorities. [Ward] and [Sandholm2009], among others, describe priorities built up from multiple factors. [Ward] proposes a scheme based on the product of the normalized or average wait time, processors requested, estimated run time, and queue name, with each of these four factors given an exponential weight. The equation at the top of Algorithm 1 depicts this type of product prioritization. Many other factor combinations are also possible.

• Dynamic Priorities. Priorities based on wait times are inherently dynamic, as the wait time increases constantly until the job is dispatched. Priorities can also be dynamic for other reasons.

[Ward] depicts priorities that are recalculated once per minute. Wait times accumulate continuously and queued jobs arrive and depart. The weights (exponents) for each parameter (wait time, run time, number of processors, queue) don't change. These recalculated priorities are used along with Backfill to select the next job(s) to be dispatched.

[Streit] describes a dynamic prioritization mechanism that switches between FCFS, SJF, and LJF algorithms, all with Backfill. Priority recalculation is triggered whenever the number of queued jobs exceeds a limit. If Batch jobs outnumber interactive jobs by a certain margin, the system switches to LJF. If the reverse situation occurs, the system switches to SJF.

In [Sandholm2009], users assign and re-assign priorities using templates, subject to an overall priority budget limitation. The templates specify the priority of the job as a whole; by workload stage (e.g., map() vs. reduce()); and in the event of a workload bottleneck. Users can also monitor and adjust priorities in real time.

The above are illustrative examples of the prioritization schemes that have emerged in the literature over the past several years.

3.2 SCHEDULING PRIORITIZED JOBS

The literature advances many alternatives for handling the workload once priorities have been set (e.g., [Calzolari], [Hovestad]). These include:

• Global scheduling vs. local scheduling within queues or pools. In the Hadoop Fair Share scheduler already described, resource allocation is balanced across workload pools but then prioritized and scheduled within each pool. In the Hadoop Capacity scheduler, resources are similarly balanced across queues, but scheduled according to priority within each queue. And systems can ignore queue assignments and schedule globally. Another approach to workload local scheduling is virtualized MapReduce, where each user is granted their own virtual machine running their own copy of Hadoop or another MapReduce host.

• Resource-aware scheduling. A design goal of Hadoop is to schedule using node- and rack-awareness. The job's input data is spread out redundantly across the allocated nodes of the system, and when a job is dispatched, the scheduler attempts to send Map tasks to run on a node where that part of the input data is located, thereby optimizing network bandwidth. If those nodes are unavailable, the scheduler can attempt to run the job on the same rack as the data, which will minimize global network contention using the backbone local to that rack.

Resource-aware scheduling includes other considerations besides data locality. Some software such as an optimizing compiler may only be licensed for certain nodes. [Zaharia2009] has pointed out that Reduce tasks typically require more memory than Map tasks, for example, and can be steered to run on cores that have a larger local memory. Resource awareness can also include resource utilization, such that nodes that have instantaneously or recently experienced, say, I/O hotspots, can be avoided when dispatching new work [Sandholm2010].

• Phase-Based. MapReduce jobs run in phases – Map, where the workload is distributed among multiple nodes and processed independently – followed by shuffle and sort operations – and then Reduce, where the results of the Map phase are tabulated to yield the overall result of the problem. Workload scheduling is constrained such that no Reduce job will run until all Map jobs have completed [Sandholm2009]. The sharing mechanism (as in Fair Share) can allocate a different number of cores for map() (usually more) than for reduce(). Other phased-based scheduling methods are also possible.

• Delay Scheduling. If the highest priority task does not currently possess its input data on the node that has come available for a new task, its dispatch is delayed in the expectation that the next available node may possess a copy of that data; and another, lower priority task is dispatched in its place while the highest priority task waits.

The maximum duration of that delay is set by the sys admin, and if the maximum delay expires, the task is dispatched even if its input data is only available on a remote node. There are two time delay limits – one before the task can go to a different node on the same rack (slower than running on the same node, but still more local than running on an off-rack node) and the second limit before the task can go to a node on a another rack (see Resource-Aware Scheduling, above). Delay scheduling is more appropriate for I/O bound jobs that will make good use of that data locality than for CPU-bound jobs. MapReduce applications tend to be I/O bound.

• Copy-Compute Splitting. [Zaharia2009]. In MapReduce, a job may obtain processors for the reduce phase but not use or release them until the map phase is 100% complete on all CPUs (otherwise, a "reducer" CPU might not have all of its input data available). No Reduce task can complete until all Map tasks complete, because all Map tasks can contribute input to each Reduce task. If at least one map job takes a long time to complete, the Reduce slots may be held idle waiting for it and competing jobs can be starved of node/CPU resources. [Zaharia2009] calls this "slot hoarding." Because all of the Reduce tasks are waiting for the same event – the completion of the last Map task – they tend to run nearly synchronously with one another. In response to this potential bottleneck, Zaharia proposes that the Reduce phase be split into copy and compute components that run independently. In the copy sub-phase, intermediate map() output data is copied to targeted Reduce nodes. The compute sub-phase generates the solution data sets. Zaharia proposes a mechanism for finer-grained control of the Reduce function to alleviate slot hoarding and wasted idle Reduce slots.

• Preemption and Interruption. For the reasons stated in Section 1, above, most HPC data centers configure their schedulers to allow all jobs to run to completion and without interruption. However, some HPC system designers have advanced strategies that contemplate task pre-emption, interruption, or killing.

Such strategies are easier to conceive in a MapReduce environment, compared to more monolithic cluster HPC environments. In the first place, cluster systems of hundreds or thousands of networked servers are more unreliable than, say, a mainframe. To ensure against the possible (and likely) failure of a node, Hadoop dispatches multiple copies of each Map or Reduce task. If one fails, another will run to completion; at the same time, the job control subsystem will detect the failure and clone another copy of the failed task as insurance against a second failure. Hadoop can respond to and tolerate preemption the same as it would to a node failure, by dispatching an alternate copy of the tasks.

In the second place, programs or production application systems implemented using MapReduce are by their nature more independently parallel. When tasks are independently parallel, preempting one task will not cause a halt to the other tasks, and fewer resources will be left idle.

For these and other reasons, there is less impact from pre-empting or killing a single MapReduce task. If copy 1 of the task is pre-empted, there are two backup copies of the task that can still run to completion. And the level of independence that each task possesses means that the slow-down is local and not global to all tasks in the job.

[Zaharia2010] proposes that if delaying a task until a node with local data becomes available prevents the task from being dispatched past a certain time limit, the scheduler can kill off a job on a node that has the data required and insert the waiting task. One is willing to contemplate killing the job, with its attendant waste, to gain the benefit of data locality.

• Social Scheduling. Social scheduling is a manual process where the user community reaches an agreement on resource allocation. Requests for, say, special dedicated use of the system due to an external deadline are honored by the users' coworkers. Data center management may also intervene in scheduling conflicts and determine who gets to use the system and when.

• Variable Budget Scheduling formalizes social scheduling. At the start of the fiscal year, users and project teams submit proposals for the type of work they wish to use the system for and request so many processors hours along with other resources, such as an Hadoop virtual machine, time slots such as prime time or weekends, or week long blocks related to some external event, milestone or deadline.

Sections 3 provided illustrative examples, but by no means a comprehensive list, of prioritization (3.1) and scheduling (3.2) alternatives.

4.0 OPTIMIZATION BY EXHAUSTIVE SIMULATION WITH AN OBJECTIVE COST FUNCTION

The literature is full of alternative approaches to both prioritization and scheduling. A possible approach to addressing this diversity is to build a system that continuously searches a range of alternatives and selects the best one. Dynamic prioritization of various kinds have been suggested by several authors, as mentioned in Section 3.1, above. This section describes an extension to the idea of dynamic prioritization.

One can consider a system in which the job scheduler uses simulation to scan the space of workload parameters (such as the exponential weights described under Composite Priorities, above) and locate the choice of parameters that yields the best performance, where 'best' can be defined by an objective cost function. Examples of such a cost function include lowest overall workload execution time, lowest average job turnaround time/response time, or highest processor utilization, but any arbitrary and calculable objective function could be used. The scheduler continuously re-evaluates the expected performance of the enqueued workload as it changes over time and adjusts the scheduling parameters to give the 'best' performance for workload conditions at that specific time.

$$Pj = (Tasks/avg.Tasks)^{\alpha} * (waittime)^{\beta} * (runtime/avg.runtime)^{\gamma}$$

Data: job file, system size
Result: Schedule performance

```
Read job input file;
for α,β,γ = -1 → 1,+ = 0.1 do
  while jobs either running or queued do
    calculate job priorities;
    for every job do
      if job is running and has time remaining then
        update_running_job();
      else
        for all waiting jobs do
          pull jobs from priority queue and start based
              on best _t;
          if job cannot be started then
            increment waittime
          end
        end
      end
    end
    Print results;
  end
end
```

Algorithm 1. Algorithm for Dynamic Scheduler

The algorithms depicted in the two figures show how this could be done. Algorithm 1 shows how to prioritize each job using stated exponential weights and then dispatching jobs in descending priority sequence. Algorithm 2 loops through a range of values for the α, β and γ exponential weights, selecting the set of values that yield the lowest cost as determined by the objective function.

Algorithm 1can be interpreted as follows:

- Calculate the priority of all enqueued jobs using stated weights α, β and γ. The priority for a job is the product of the number of cores requested, the current wait time, and the estimated runtime (all multipliers with exponential weights).
- Sort these jobs in descending priority order.
- Starting with the highest priority job, dispatch jobs until the available capacity is consumed.
- If the remaining capacity is too small for the number of processors that the next highest priority task requests, dispatch a fraction of the requested processors equal to the number remaining available, such that exactly all available capacity is allocated to some job.

Require: *C*, capacity of the knapsack, *n*, the number of tasks, *a*, array of tasks of size *n*
Ensure: A cost vector containing E_f for each job class Cost (*A, B, C, D, WRT*).

```
1:  i = 0
2:  for α = -2 ≤ 2; α+=0.1 do
3:    for β = -2 ≤ 2; β+=0.1 do
4:      for γ = -2 ≤ 2; γ+=0.1 do
5:        Cost[i++] = schedule(α,β,γ)
          {For Optimization}
6:        if cost[i] < bestSoFar then
7:          bestSoFar = cost[i]
8:        end if
9:      end for
10:   end for
11: end for
```

Algorithm 2. CostEval (*a, n, C*)

Algorithm 2 can be interpreted as follows:

- Scan all values of the 3 parameters α, β and γ between -2 and +2 in increments of 1/10
- For each triplet of α, β and γ, simulate the workload currently enqueued using Algorithm 1.
- For each simulation, calculate the cost, which may be a weighted function of average task

run time, total workload run time, or some other objective function.
- Select the simulated values of α, β and γ yielding the lowest cost and use those values to prioritize and dispatch jobs.
- Rerun this simulation each time a job completes and processors become available for new work.

Prior tests [Simon] using natural workload logs have shown that the dynamic scheduler can improve overall workload completion time compared to the default FCFS scheduler. Because the dynamic scheduler includes FCFS among the parameter options tested, we argue that it will never provide worse results than the default.

The results shown below depict alternative fixed scheduling policies against the dynamic scheduler in terms of three objectives – overall wait time (minimize the time to complete all jobs); system utilization (maximize processor utilization); and total wait time (minimize the sum of the wait times of all jobs). Figures 9 and 10 will show that a single scheduler cost function yielded better performance for two objectives (overall wait and system utilization) than common fixed policies and was better than all except one fixed policy for the third objective (total wait). These results are based on a synthetic workload log.

The next graphs show system behavior using various fixed scheduling policies - choices for the three parameters α, β and γ.

One can read Figure 7 as follows:

- The horizontal axis is time, moving from left to right, in increments of seconds.
- The left-hand vertical axis is for the solid red upper line and depicts the current overall system system utilization, from 0 up to 1=100%.
- The right-hand vertical axis is for the lower (dotted green) line and depicts the number of jobs running.
- The fixed parameters are α = 0, β = 1, and γ = 0. All weight is placed on the wait time; job size and duration are ignored. This translates to first come first served (FCFS), the default fixed policy. We show fixed policies first, as a basis for comparison with the dynamic scheduler in Figure 9.
- About 10 jobs run at first but this number increases to a maximum of 30 jobs after about 20 seconds. The workload drops below 10 jobs after about a minute, then stays below 5 jobs for more than half the time total time before entire workload completes after 156 seconds.
- System utilization is at or above 90 percent for the first half of the run time, and falls below 50 percent for about the last 20 seconds.

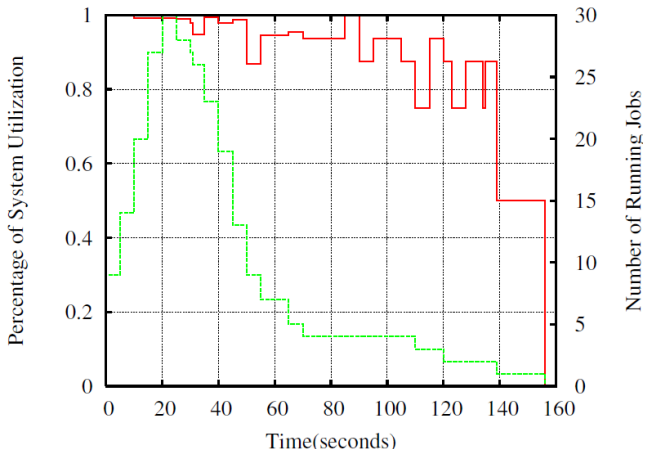

Figure 7.

In Figure 8, the significance of the horizontal and vertical axes are as before.

- The parameters are α = 0, β = 0, and γ = -1. This equates to a "shortest job first served" (SJF) discipline. Recall that the third parameter is for estimated run time, and that a negative exponent favors short over long run times.
- Initially, many (~15 to 25) short jobs run, with stair steps down to about 12 jobs after 30 seconds and about 5 jobs about 70 seconds.
- The whole workload completes in 150 seconds, 6 seconds faster with SJF than Figure 7 (FCFS).

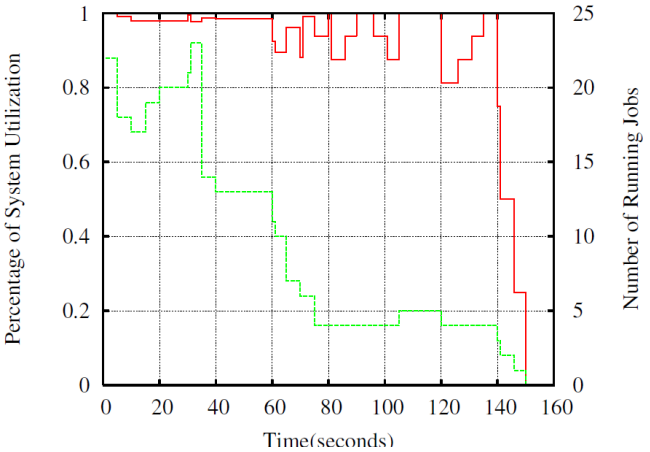

Figure 8.

Figures 7 and 8 were for selected fixed schedule policies. Figure 9 shows a result for the dynamic scheduler.

Running the simulation for this workload for all parameter values shown in Algorithms 1 and 2 yields the optimum parameters in Figure 9, that is, α = -.1, β = -.3, and γ = -.4. These parameters give higher priority to shorter and smaller jobs, although lower priority to jobs waiting longer.

By examining 40 x 40 x 40 = 64,000 alternatives, an option with improved overall workload completion time has been achieved. (Analysis of the cost of running the simulation is beyond the scope of this paper, but possible approaches include varying the frequency of running the simulation and using idle processors to do so when available. And the simulation itself is highly parallel.)

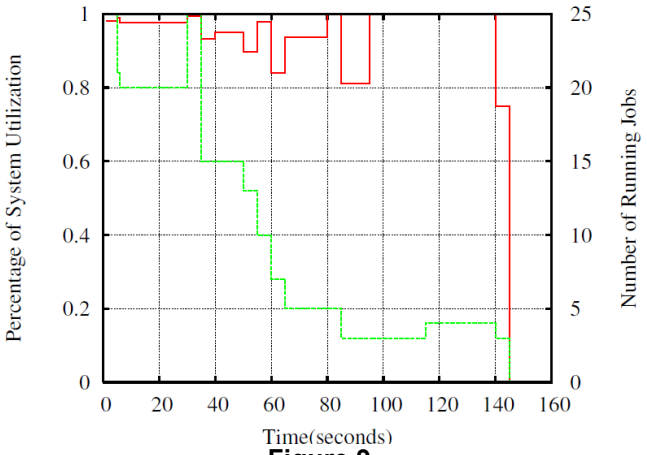

Figure 9.

The following table compares the results graphed (Figure 7 is FCFS; Figure 8 is SJF; Figure 9 is Dynamic) and three other fixed scheduling policies.

Policy	α	β	γ	Wait	Utiliz	TotalWait
LJF	0	0	1	165	84	4820
LargestJF	1	0	0	146	95	5380
SJF	0	0	-1	150	92	3750
SmallestJF	-1	0	0	165	84	4570
FCFS	0	1	0	156	88	4970
Dynamic	-0.1	-0.3	-0.4	145	95	3830

The Dynamic Schedule give the shortest overall workload completion time (145 seconds) compared to the default results (156 seconds). In previous work [Simon], the Dynamic Scheduler completed the work in half the time of the default FCFS for a NASA AIRS Hadoop workload. The Dynamic Scheduler also gave a half percentage point better utilization than the next best policy (Largest Job First). And it did better than all of the other fixed scheduling policies except SJF in terms of TotalWait time. The point we are making in this paper is not that any particular level of improvement is always achievable with dynamic prioritization, but rather that dynamic prioritization will yield performance results at least as good as a fixed priority scheduler using the same factors (e.g., tasks, wait, duration, etc.). The dynamic scheduler performance improvement is dependent on the workload specifics and the fixed priorities to which it is compared.

Note that while dynamic prioritization will yield optimal results for the workload overall, an end user will not have the same performance expectations for a particular job that he or she may be accustomed to – with, say, fixed priority functional queues (like debug, test, production, etc.). If data center management wants to provide the fixed priority functional queues that users are used to, management could use the dynamic schedule simulator to analyze historical workload data and select fixed priority parameters most appropriate to that data center's workload. Another application would be simulation to select fixed parameters for particular time periods (like weekend, overnight, day time peak, etc.).

The results shown in Figure 9 use a specific objective cost function – fastest overall workload completion time – and a specific prioritization method – the weighted product of run time, waiting time, and processors requested – plus Backfill. The same strategy – searching the parameter (exponent weights in this case) space for a given prioritization method using simulation and evaluating the space based on an objective cost function – could be applied to many alternative prioritization and scheduling algorithms and perhaps any arbitrary but calculable cost function. The novelty lies in the continuing evaluation and adjustment of the workload scheduling mechanism to search for parameters that fit the enqueued workload as both change over time and constantly tune the scheduler to give the 'best' system performance.

5. CONCLUSIONS

- Where data center management's goals are to maximize some arbitrary but calculable objective cost function, such as lowest overall workload completion time or lowest average turnaround time, it may be possible for an exhaustive search of the workload parameter space to constantly tune the system and provide nearly optimal performance in terms of that objective cost function.

- The turnpike effect[4] suggests that new capabilities will continue to inspire new

[4] *Turnpike Effect: "The availability and unforeseen utility of a resource leads to greater use than was predicted." http://www.virtualsalt.com/crebok3a.htm*

applications. Cluster supercomputer hardware and the Hadoop software framework provide this kind of new capability. Under the circumstances, it's no wonder that so many alternative workload scheduling methods continue to emerge. Perhaps this proliferation of HPC cluster workload scheduling alternatives is the result of (1) the known challenges of application programming for parallel systems, (2) the popularity of open source platforms that are comparatively easy to customize, and (3) the brief lifespan of MapReduce that has not yet had the chance to mature.

BIBLIOGRAPHY

[Butterfly] https://en.wikipedia.org/wiki/BBN_Butterfly

[Calzolari] Calzolari, Federico, and Volpe, Silvia. "A New Job Migration Algorithm to Improve Data Center Efficiency," Proceedings of Science. The International Symposium on Grids and Clouds and the Open Grid Forum, Taipei, March, 2011.

[Connection Machine] https://en.wikipedia.org/wiki/Connection_machine

[Feitelson1998] Feitelson, Dror, and Rudolph, Larry. "Metrics and Benchmarking for Parallel Job Scheduling." Job Scheduling Strategies for Parallel Processing '98, LNCS 1459, Springer-Verlag, Berlin, 1998.

[Feitelson1999] Feitelson, Dror and Naaman, Michael. "Self-Tuning Systems" IEEE Software, March/April 1999.

[Glassbrook] Glassbrook, Richard and McGalliard, James. "Performance Management at an Earth Science Supercomputer Center." CMG 2003.

[Heger] Heger, Dominique. "Hadoop Performance Tuning – A Pragmatic & Iterative Approach" www.cmg.org/measureit/issues/mit97/m_97_3.pdf 1997

[Hennessy] Hennessy, J. and Patterson, D. Computer Architecture: A Quantitative Approach, 2nd Edition. Morgan Kauffmann, San Mateo, California.

[Herodotou] Herodotou, Herodotos and Babu, Shivnath. "Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs," Proceedings of the 37th International Conference on Very Large Data Bases, Vol. 4, No. 11. VLDB Endowment, Seattle, ©2011.

[Hovestadt] Hovestadt, Matthias, and others. "Scheduling in hpc resource management systems: queuing vs. planning." *Job Scheduling Strategies for Parallel Processing*. Springer Berlin Heidelberg, 2003.

[Multiflow] https://en.wikipedia.org/wiki/Multiflow

[Nguyen] Nguyen, Phuong; Simon, Tyler; and others. "A Hybrid Scheduling Algorithm for Data Intensive Workloads in a MapReduce Environment," IEEE/ACM Fifth International Conference on Utility and Cloud Computing. IEEE Computer Society, ©2012.

[Rao] Rao, B. Thirumala, and others. "Performance Issues of Heterogeneous Hadoop Clusters in Cloud Computing" Global Journal of Computer Science and Technology. Volume XI Issue VIII, May 2011.

[Sandholm2009] Sandholm, Thomas, and Lai, Kevin. "MapReduce Optimization Using Regulated Dynamic Prioritization," SIGMETRICS Performance '09. ACM, Seattle, ©2009.

.

[Sandholm2010] Sandholm, Thomas, and Lai, Kevin. "Dynamic Proportional Share Scheduling in Hadoop" Job Scheduling Strategies for Parallel Processing 2010. Springer-Verlag, Berlin, 2010.

[Scavlex] Scavlex. http://compprog.wordpress.com/2007/11/20/the-fractional-knapsack-problem

[Sherwani] Sherwani, Jahanzeb, and others. "Libra: a computational economy-based job scheduling system for clusters." *Software: Practice and Experience* 34.6 2004.

[Simon] Simon, Tyler and others. "Multiple Objective Scheduling of HPC Workloads Through Dynamic Prioritization" HPC 2013, Spring Simulation Conference, The Society for Modeling & Simulation International, 2013.

[Spear] Spear, Carrie and McGalliard, James. "A Queue Simulation Tool for a High Performance Scientific Computing Center." CMG 2007. San Diego, 2007.

[Streit] Streit, Achim. "On Job Scheduling for HPC-Clusters and the dynP Scheduler" Paderborn Center for Parallel Computing, Paderborn, Germany.

[Ward] Ward, William A. and others. "Scheduling Jobs on Parallel Systems Using a Relaxed Backfill Strategy." Revised Papers from the 8th International Workshop on Job Scheduling Strategies for Parallel Processing, JSSP '02, London, Springer-Verlag, 2002.

[Zaharia2009] Zaharia, Matei, and others. “Job Scheduling for Multi-User MapReduce Clusters” Technical Report UCB/EECS-2009-55. University of California at Berkeley, 2009.

[Zaharia2010] Zaharia, Mateir, and others. “Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling.” EuroSys’10.ACM ,Paris, ©2010