Optimizing Routing and Backlogs for Job Flows in a Distributed Computing Environment

David Montana and John Zinky

BBN Technologies
10 Moulton Street, Cambridge, MA 02138
dmontana@bbn.com, jzinky@bbn.com

Summary. We address the problem of optimizing the flow of compute jobs through a distributed system of compute servers. The goal is to determine the best policy for how to route jobs to different compute custers as well as to decide which jobs to backlog until a future time. We use an approach that is a hybrid of dynamic programming and a genetic algorithm. Dynamic programming determines the routing and backlog decisions about individual flows of homogeneous jobs, while a genetic algorithm optimizes the order in which the different flows are fed to the dynamic programming algorithm. We demonstrate the effectiveness of this approach on sample problems, some designed to yield a known correct answer and others designed to test the scaling.

1 Introduction

Distributed computing is the ability to share computational load among multiple computers across a network, and it is a powerful way to increase compute capacity. Effective usage of this joint compute power depends on making good decisions about how to assign the compute tasks to the compute resources. The question of how best to distribute the tasks to the resources can often be formulated as an optimization problem. However, this optimization problem can vary widely depending on the assumptions about the nature of the compute tasks, resources, connectivity, and criteria for what is a good set of assignments.

1.1 Overview of the Problem

We have defined an optimization problem based on the needs of a customer who controls a large distributed network of computing devices, also referred to as an *enterprise grid*. It is formulated in general enough terms that it is applicable to other enterprise grids, and not just that of the customer. Some distinguishing properties of this problem are:

- The jobs are aggregated into **job flows**, where the jobs in a flow are homogeneous in their properties and follow certain arrival statistics. The optimization considers

only flows and not individual jobs. The assumption is that there are enough jobs in a flow that they can be modeled accurately and more efficiently as an aggregate flow.

- The resources are aggregated into **clusters**, each with its own local scheduler that assigns individual jobs to individual resources. We assume that the local scheduling at a cluster is handled separately and focus on the high-level problem of routing the jobs flows between clusters. The resource configuration is an enterprise grid, and the problem is to design a metascheduler.
- Each job consists of a sequences of steps, also referred to as **tasks**. In general, the tasks for a job require different compute resources, and hence fully scheduling a job requires finding a sequence of different clusters for the job to visit, i.e. a route through the clusters. When considering a job flow rather than an individual job, this route is potentially multipath, as different jobs in the flow can follow different routes. Note that these routes are not routes in the networking sense, i.e. a set of intermediate points leading to a destination, but closer to routes in the vehicle routing sense, i.e. a sequence of destinations.
- The jobs can have varying **utilities** and **deadlines**, although the utilities and average deadlines need to be homogeneous among jobs within each job flow. This reflects the reality that some jobs, such as providing an interactive response to a human, require fast turnaround to be useful, while others, such as overnight batch jobs, do not. Similarly, some jobs are more important to complete than others based on the mission of the enterprise. Balancing the tradeoff between jobs with tighter deadlines and those with higher utility is an important functionality of the metascheduler.
- Jobs can be stored in **backlog** until a future time as a means of ceding resources to other jobs. This is generally used to prevent jobs with longer deadlines from blocking the execution of jobs with shorter deadlines when the latter jobs have lesser or equal utility than the former. One assumption is that the local schedulers simply schedule the higher-utility jobs first without considering the deadlines, leaving consideration of deadlines to the metascheduler (which makes sense because the metascheduler has the global view needed to make tradeoffs between utility and deadlines). A second assumption is that the metascheduler is provided predictions about what the future job flow loads and resource availabilities will be to provide a basis for these tradeoffs.

The details of the problem definition are given in Section 2.

1.2 Previous Work

There is a long history of work in distributed computing, and we do not attempt to summarize it all. Past research has addressed various aspects of distributed computing, including both how to write algorithms that execute on distributed infrastructure and how to create the infrastructure. Just on the infrastructure side, which is our focus, many issues have been studied, such as how hosts connect and communicate, how hosts coordinate to share tasks, and security. An example of an application that

addresses a wide range of these issues is Condor [18]. We limit our attention to job scheduling, i.e. how best to share tasks among the assorted compute resources.

Many of the techniques for assigning compute tasks to resources are referred to as *load balancing*, since a key objective is to minimize the amount of time that resources are idle. *Adaptive load balancing* is when the assignment algorithm reacts online to the current situation. Many of the schemes for load balancing are application-specific and need to be revised for each usage, but others are more general [13]. *Scheduling* is potentially more general than load balancing, since scheduling can have more complex objectives than just the immediate correction of imbalances in processing loads.

One important issue to address is scaling. If there are a very large number of compute resources and tasks, it is challenging to efficiently use all the resources. A common approach to scaling is a hierarchical decomposition of the load balancing or scheduling responsibilities. The resources are divided into *clusters*. Each cluster has its own *local scheduler* that assigns tasks to resources, while a high-level scheduler dispatches tasks to clusters (essentially treating clusters as resources). Such a high-level scheduler is now often referred to as a *grid metascheduler*, with the distributed collection of compute resources under its control called a *grid* [11, 19]. There are different development environments for the creation of grid metaschedulers including Community Scheduler Framework, Gridway, and Condor-G [18]. Grimme [8] distinguishes three types of grids: a global grid is a loose confederation across a wide geography with different owners; a high-perfomance-computing grid is a tight clustering of resources; and an enterprise grid (which is the focus of our paper) is a loose cluster like the global grid but with all machines under the ownership of a single organization [16]. There are different scheduling requirements for each type of grid.

We now mention some previous research that addresses aspects of the scheduling problem not commonly investigated but highly relevant to our work. Lo et al. [10], in their work on metascheduling, address the issue of the effect of time zones on scheduling, allowing the scheduler to anticipate lower loads on compute resources when it is night in their local time zones. In general, the ability to predict loads in the future can help inform scheduling decisions made in the present, particularly if some tasks can wait until the future to be executed [6, 3]. Bose et al. [3] show that it is possible to use a genetic algorithm as a metascheduler and that it can execute fast enough to be used online under certain circumstances. This genetic algorithm uses a direct encoding with a chromosome that maps each task to its assigned resource. Andresen and McCune [1] define the concept of a *task chain*, a sequence of compute tasks that must be performed to complete a compute jobs, and scheduling a job means routing the job in sequence between resources. Stone [15] uses a network flow algorithm for determining routes of jobs through the resource; this differs from most techniques for scheduling of distributed computing, which consider each task in isolation rather than as part of a flow.

A flow-based view of scheduling leads to a fundamentally different scheduling problem, one that is less reactive and more predictive and one that focuses more on statistical trends rather than individuals jobs and tasks. While uncommon for

scheduling of distributed processing, a flow-based approach is common for network routing, and some techniques for determining routes in networks are actually more similar to our approach than those from distributed computing. For example, Casetti et al. [4] has used hierarchical load balancing in the network routing context with the statically determined routing strategy based on the *offered loads*, which are the average flows of different types of traffic. Oueslati and Roberts [14] have demonstrated the benefits in networks of flow-aware routing, i.e. considering each packet as part of a larger flow, as opposed to flow-oblivious routing, i.e. treating each packet separately. Barolli et al. [2] use a genetic algorithm whose chromosomes directly encode a routing tree to determine optimal routes through a network. Okuhara et al. [12] also use a genetic algorithm whose chromosomes directly encode a route, or multiple routes, for optimizing flow-based routing. They include the concept of optimizing flow control, which is the prevention of certain flows from entering the network in order to prevent congestion and which is very similar to our use of backlog. Key and Massoullie [9] integrate the concept of utility associated with a flow into their optimization criterion used with their fluid model for network routing.

1.3 Overview of Our Approach

A distinguishing property of our solution to this problem is that it is a hybrid approach, i.e. a combination of different techniques for handling different parts of the problem. These different techniques are the following.

- A simple **greedy algorithm** selects the assigned resource cluster for a given task in a given job at a given time. For each cluster that can handle the task, the algorithm temporarily assigns the task to that cluster, propagates the consequences of this assignments, and determines which assignment minimizes the overall increase in the score.
- The question of when and for how long to place a given job in backlog is addressed using **dynamic programming**. For each task/step in a job, it explores different lengths of time for which to backlog the job at this step, creating a new branch in a search tree for each choice and pruning the tree to explore only the best possibilities. This cannot properly be done as a greedy search because the selection of the backlog times at earlier steps constrains the options at later steps, and the effects cannot be determined until handling the later steps.
- The order in which to consider the flows for routing and backlog decisions is determined by a **genetic algorithm**. The dynamic programming and greedy algorithms determine the routing and backlog decisions one flow at a time, and the order in which these flows are handled greatly affects these decisions. Genetic algorithms have proven good at rapidly searching spaces of permutations, and hence we use one to find the ordering of flows that produces the best overall score.

This general approach is a common one for genetic-algorithm-based scheduling and was first described by Whitley et al. [20] and Syswerda [17]. A fast schedule builder incrementally constructs a schedule one job/task at a time with different schedules

resulting from different presentation orders of the jobs/tasks; a genetic algorithm optimizes the presentation order. The details of the optimization algorithm are presented in Section 3. Experiments that demonstrate the effectiveness and good scaling properties of the algorithms are described in Section 4.

2 Problem Definition

We now discuss the various components of the problem definition.

2.1 Jobs, Tasks and Flows

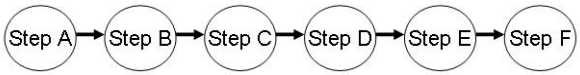

Fig. 1. A sample compute job consisting of six steps/tasks.

Compute jobs consist of a set of steps, or tasks, which are the atomic units of computational work to be performed. Each step must be executed in sequence, so a task cannot begin until its predecessor has completed. Figure 1 illustrates a job with six tasks labeled A-F, which is the standard task breakdown for all jobs in the experiments described below. Each task is assigned to and executed by a single cluster, but the various tasks in a job can be, and generally will be, assigned to different clusters. Therefore, a job will in general visit a sequence of clusters, which means that it will follow a route through the distributed system of computational resources.

Each job is part of a job flow. The jobs in a flow are homogeneous, i.e. they all possess the same properties. These include

- the sequence of tasks to execute
- the mean execution time of each task
- the mean lifespan of the job, i.e. the time between the arrival time and the deadline
- utility, i.e. how important it is to accomplish the job before the deadline
- the routing constraints, which specify for each task which clusters are allowed to be assigned to that task (The constraints can arise from a variety of causes including network connectivity and the inherent capabilities of the clusters.)

The jobs in a flow enter the system with a known mean time between arrivals.

2.2 Resources and Clusters

A cluster is an aggregation of individual compute resources together with a manager to distribute the tasks among the resources and a local scheduler that decides how to assign the tasks to resources. In the work described here, we are not performing the

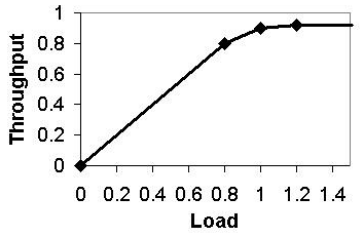

Fig. 2. Example load-to-throughput map for a cluster.

local scheduling but rather just the metascheduling, i.e. the routing of jobs between the clusters. To support the metascheduler, we do need a model of the behavior of a local scheduler.

Not all jobs/tasks that enter a cluster can be completed by the cluster, as each cluster has a finite capacity. We define the *capacity* of a cluster as the maximum number of task-seconds (where a task-second is the amount of computation accomplished on a single task by a "standard" compute engine) that can be completed every second. If the cluster consists of all "standard" resources, then the capacity equals the number of resources. The *load* on a cluster is the number of task-seconds entering the cluster per second, or alternatively, this quantity normalized by dividing by the capacity. The *throughput* of a cluster is the number of task-seconds of processing completed per second, or alternatively, this quantity normalized by dividing by the capacity. The average (normalized) throughput is constrained to be less than 100% and less than the average load.

We characterize the aggregate behavior of a cluster and its local scheduler using a piecewise-linear load-to-throughput mapping, which specifies the expected throughput for a given load. Figure 2 shows an example of such a mapping, which is the one that we used for all the clusters of the experiments described below. To determine which tasks are the ones that are completed, we order the tasks according to the utility of their jobs. The highest-utility tasks are completed at a rate equal to the throughput-to-load ratio of just these tasks, the next-highest-utility tasks at a rate which is the ratio of the additional throughput to the additional load, and so on.

Each cluster has an *input queue* and an *output queue*. Jobs wait in the input queue until a resource becomes available. The output queue is for backlog, where jobs can be saved until a future time when they are released for the next step in their processing sequence. Any job whose deadline passes while waiting in a queue is removed from the queue and discarded.

2.3 Routing/Backlog Policies and Routing Constraints

A *routing/backlog policy* determines the decisions for where (i.e., at which cluster) and when the steps of each job are executed. A policy for a given job flow is represented as a set of probabilities. For each step of a job in the flow, the policy specifies for each cluster the probability that the step will be executed at that cluster, as well

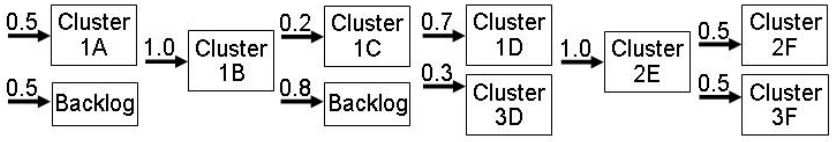

Fig. 3. Graphical depiction of a routing/backlog policy for a flow.

as the probability that the job will instead be held in backlog until a future time. Jobs held in backlog are released back into the system and re-evaluated when the routing policy is updated. Figure 3 illustrates a sample routing policy for a job flow whose jobs have six steps.

The routing/backlog policy is the one aspect of the system under external control and is what we can vary to optimize the performance of the system. Section 3 discusses how to determine an optimal routing/backlog policy.

Routing constraints limit the possibilities of which clusters can have non-zero probabilities. For each step of each job flow, there is a list of legal clusters that are allowed to handle this step. These constraints can reflect underlying constraints of the system, such as limitations on network connectivity, or can serve to aid the optimization process (either automated or manual) by limiting the choices available and hence reducing the size of the search space.

2.4 Epochs and Time Dependence

An epoch is a time interval over which we can assume that all aspects of the system remain constant. This includes the job flows and all their properties, and the clusters and their capacities. We say that the routing policies will remain the same throughout an epoch, as there is no reason for them to change, although the policies will in general change at epoch boundaries. This assumption of piecewise constant behavior for the system allows us to apply a model based on mean-value analysis, as discussed in Section 2.5.

The changes in the properties of the job flows and clusters across epochs reflect predictions about how the loads and capacities will vary with time. For example, in Section 4 we will consider job flows whose arrival rates follow a 24-hour cyclical pattern, with arrival rates higher during the local daytime and lower during the local nighttime. Similarly, knowledge of future scheduled service times for a resource can be reflected by changing the capacity of its cluster in future epochs. Predictions of future conditions are important for determining not just future policy but also current policy, since backlogs and deadlines can extend across epoch boundaries.

2.5 Evaluation Function

To evaluate the effectiveness of a particular policy, we simulate the system with this policy in place. The simulation does not consider individual jobs but rather examines

the aggregate flows using a mean-value analysis. The results of the simulation can be scored using an optimization criterion.

The simulator propagates each flow in each epoch over a multipath route through the network of clusters. It multiplies the system arrival rate of a flow by the probability of the first step being assigned to a cluster to obtain the rate of the flow entering this cluster at this step. Similarly, multiplying the system arrival rate by the probability of backlog for the first step yields the rate at which the flow is backlogged before its first step is executed. If more than one of the probabilities is non-zero, the flow will split along multiple paths, which is why we refer to the route as multipath. The simulator uses the load-to-throughput map for a cluster to determine the output rate of the flow following this step. This process is continued for the subsequent steps, with the cumulative throughput for step N multiplied by the probabilities for step N+1 to give the input rates at the various clusters for step N+1.

While the simulator is relatively simple, there are a few details that complicate it. One is the need sometimes to retract throughput that has been allocated to a flow. If a cluster that handles a flow is then assigned a new flow of utility greater than or equal to that of the original flow, it may be that the original flow loses some of its throughput to the new flow. When this happens, there is in general a chain reaction, since downstream steps of the original job now have lower rates, which in turn can allow other flows to grab some of the forfeited throughput, and so on. The simulator propages these perturbations until they die out.

A second detail is what happens to jobs that are backlogged or waiting in queues across epoch boundaries. Such jobs are added to the flows at the appropriate step in the process for the new epoch, with flow rates that cumulatively across the epoch would integrate to the right number of jobs.

The goal is to minimize the number of dropped jobs, i.e. jobs not completed within their deadline, with an emphasis on not dropping higher-utility jobs. Therefore, the primary component of the optimization criterion is the sum of the utilities of all the dropped jobs. A secondary component of the optimization criterion is a penalty for delaying the execution of jobs into future epochs. The rationale is that the strategy of backlogging jobs for the future depends on the future occurring as predicted, which it often will not in a dynamic environment, so there is benefit to finishing a job earlier rather than later. Optionally, we can add other penalties, such as one for jobs traveling between clusters in order to minimize network traffic, but we do not consider these other types of penalties in this paper.

We now provide a mathematical definition of the optimization criterion. From the viewpoint of individual jobs (as opposed to job flows), the criterion is

$$\sum_{j \in J_d} u(j) + \sum_{j \notin J_d} u(j)(1 - P^{t(j)}) \tag{1}$$

where J_d is the set of all dropped jobs (i.e. jobs that did not complete before their deadline), $u(j)$ is the utility of job j, $t(j)$ is the time in the future at which j is completed, and $P < 1$ is a constant whose role is to penalize the deferral of jobs to the future. This translates into the following formula at the flow level

$$\sum_{e \in E} P^{t(e)} (\sum_{f \in F(e)} u(f)[d(f) + b(f)(1 - P^{\tau(e)})]) \quad (2)$$

where E is the set of all epochs, $F(e)$ is the set of all job flows during epoch e, $t(e)$ is the start time of e, $u(f)$ is the utility of flow f, $d(f)$ is the rate at which jobs in flow f are dropped, $b(f)$ is the rate at which jobs in f are backlogged, and $\tau(e)$ is the duration of epoch e.

3 Scheduling Algorithm

The policy optimization algorithm has three levels, with each of the lower two levels feeding results to the next higher level. We now present these.

3.1 Level 1: Single-flow, single-epoch optimization

This component determines a routing/backlog policy for the jobs from a single flow entering the system during a single epoch. If a flow is small enough, then a single set of decisions is used for all the jobs in the flow, i.e. for each step the policy has a single non-zero probability and hence all the jobs follow the same route. The test for whether the flow is small enough is whether the flow cannot load any cluster more than x% of its capacity, where we have used x=20%.

Alternatively, if the flow is large, i.e. can produce a load of more than x% on a cluster, it is instead split into N identical smaller subflows, where N is just large enough to reduce the maximum load on a cluster below the threshold. A single-path route is determined for each of these subflows independently and in succession. The results are then aggregated into a single policy, or equivalently a multipath route, using probabilities to specify what fraction of the flow follows each path. Splitting large flows allows the routing to distribute the load across multiple clusters, which may be necessary for efficiently handling the flow.

We now discuss how to determine the single-path route for one of these indivisible subflows. For each step/task in the process, there are two decisions to make: (i) to which cluster to assign the tasks and (ii) in which epoch to execute the tasks (i.e., how long, if at all, to backlog the tasks). The former is done using a purely greedy approach; in the epoch of choice, select the cluster for which the overall penalty (i.e., the increase in the value of the optimization criterion) is minimized by the assignment. Note that the assignment of a flow to a cluster can result in another flow losing throughput at this cluster, and this effect is accounted for in the optimization criterion and hence the greedy selection.

The selection of the epoch in which to execute each step of a flow's processing chain (i.e., decisions about backlog policy) is done using dynamic programming. The rationale is that the choice to postpone the processing of one step can have large consequences for downstream steps that cannot be foreseen when deciding about the current step. So, instead of a greedy selection of the epoch for each step one at a time, we perform a more computationally intensive optimization over all combinations of

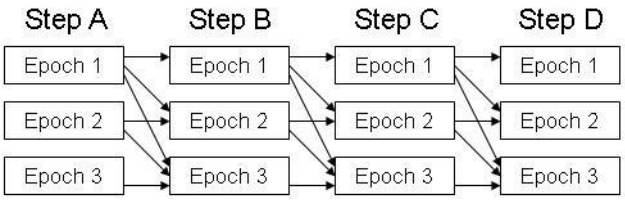

Fig. 4. Assignment of a single flow uses dynamic programming to select an epoch for each step. A step can be assigned to any epoch not earlier than that for the previous step.

legal selections of epochs for each step. Note that an epoch is legal for a step if it is not earlier than the epoch of the previous step and not later than the deadline of the flow.

The combinatorics of considering all possible combinations of epochs per step means that it is important to find an efficient optimization technique. Dynamic programming is such a technique because it (i) eliminates entire branches of the search tree early in the process and (ii) pursues the most promising branches first. The first branch point in the search tree is based on the selection of the epoch for the first step, with subsequent branch points under each of these branches based on the selection of the epoch for the second step, etc. The different branches correspond to the different paths through the graph shown in Figure 4. For each epoch E and step S, the search procedure eliminates all but the single best path leading up to the selection of epoch E at step S, which quickly prunes many branches. Furthermore, since the penalty (i.e., change in the optimization criterion) is non-decreasing with each step, we can restrict the search to pursuing only the path with the lowest penalty so far, declaring the search finished when a path that has completed all the steps has a score less than or equal to the score of any partial path.

3.2 Level 2: Multi-flow, single-epoch optimization

Using the single-flow route optimizer, we can define what we call the *rapid route builder*, which creates an entire set of routes, i.e. a full set of routing/backlog policies, for the jobs flows in an epoch. Given an ordering of the flows, the rapid route builder uses the single-flow optimizer to create the routes for each flow in succession in the order given. Due to interactions between the flows, the policies produced are potentially very different depending on the order in which the flows are routed. Therefore, finding the best ordering of jobs to feed the rapid route builder is an optimization problem we need to solve.

To perform this optimization, we use an order-based genetic algorithm. The development of order-based genetic algorithms [5, 7] was inspired by the recognition that for problems like the traveling salesman problem, the goal is to find the best ordering of N objects. Its chromosome is a direct representation of a permutation of N objects, labeled 1 through N, and its operators are designed to manipulate chromosomes of this type. Order-based genetic algorithms have been demonstrated to be

very effective and efficient at searching the space of permutations, which is why we have chosen this technique.

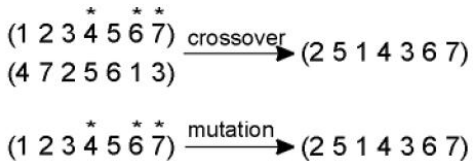

Fig. 5. The crossover and mutation operators. The *'s indicate the randomly selected positions that remain fixed in the (first) parent.

The crossover operator used by the genetic algorithm is position-based crossover [17], and its operation is illustrated in Figure 5. It works as follows. A set of positions is randomly selected (which in the example of Figure 5 are positions 4, 6 and 7). The elements at these selected positions in the first parent (which in the example are the integers 4, 6 and 7) are maintained at these positions in the child. The remaining elements (which in the example are the integers 1, 2, 3 and 5) are used to fill in the remaining slots in the child, but will in general be at different positions in the child than in the first parent. The order of these elements in the child will be the same as their order in the second parent (which in the example means that 2 is placed in the first empty position, followed in order by 5, 1 and 3).

Also illustrated in Figure 5 is the mutation operator. It works the same as the crossover operator except without a second parent to provide the ordering for the subset of elements that are reordered in the child. Instead, the new order of the shuffled elements is randomly selected.

Each member of the initial population is generated by selecting a random ordering. The flow of operations of the genetic algorithm is shown in Figure 6.

The genetic algorithm is steady-state, which means that it generates and replaces one individual at a time rather than an entire population. The advantage of a steady-state replacement strategy is that the search generally proceeds faster, since the genetic algorithm can use good individuals as soon as they are created rather than waiting for generational boundaries. Since there are no generations, the amount of work done by the search algorithm is measured by the number of individuals evaluated.

Two key parameters that control performance are the population size and the number of evaluations. Increasing them increases the expected quality of the solution found, at the expense of increasing the search time. Hence, the selection of these parameters controls the inherent tradeoff between solution quality and search time. We have found empirically for this problem that it is generally good to have the number of evaluations five times the population size, since on average this provides enough time for the search to converge without spending too much time at the end of the run stuck without making progress. So, for each run, we specify the number of evaluations and automatically set the population size to be one-fifth of that quantity. In general, the number of evaluations (and population size) needs to be larger when there are more flows, since the search space is larger. However, we can choose

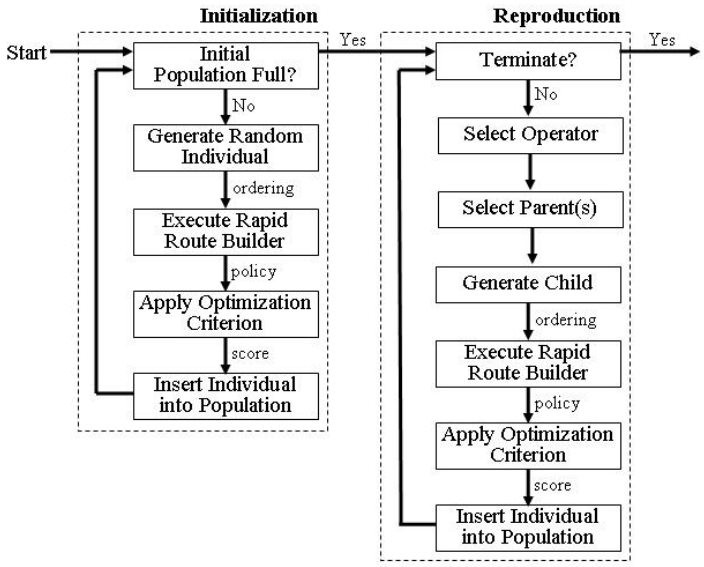

Fig. 6. The operation of the genetic algorithm.

a smaller number of evaluations and quicker search time in exchange for a worse expected solution. This ability to shorten the search is important, since the policy optimizer is potentially used adaptively to update the routing policy in real time (in response to an unexpected change in operating conditions such as a surge in load or disabled resources). Note that taking advantage of the inherent parallelism of genetic algorithms by using multiple processors can also improve the execution speed, but without sacrificing solution quality.

3.3 Level 3: Multi-epoch optimization

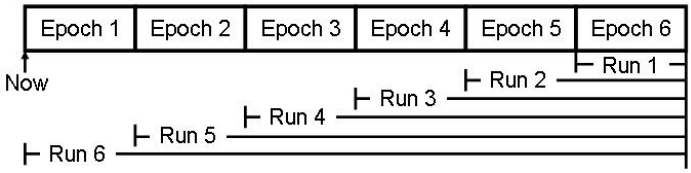

Fig. 7. The optimization algorithm starts by optimizing the routing policies for the last epoch and working backwards.

This component of the optimization algorithm steps through the epochs one at a time and executes the Level 2 optimization for all the flows in the current epoch. It starts with the final epoch and works backwards in time, as shown in Figure 7. The rationale for working backwards in time is that flows from a particular epoch can be

postponed to the future, hence requiring knowledge of the future loads on the clusters to make good decisions about whether to backlog the flows or not. Furthermore, the earlier epochs are the more important ones to do correctly, since they will be the ones executed first without the opportunity for revision.

The result of the entire process is a set of routing/backlog policies, one for each epoch. While these generally will be good policies, usually optimal or near optimal, there are three places in the process which can lead to suboptimality:

- The best policy for an epoch is not guaranteed to be generated by any job ordering fed to the rapid route builder.
- The genetic algorithm is not guaranteed to find an optimal ordering, since it is a heuristic search technique.
- Optimizing each epoch in succession rather than all in single large optimization is potentially suboptimal.

What our approach does provide is a good tradeoff between finding a good solution and keeping the search time relatively small, so that using this procedure is actually practical even for large distributed systems. In the next section, we demonstrate experimentally both the ability to find good policies and the relatively rapid execution times even as the problem size grows.

4 Experiments

We start with a set of experiments that show that the approach just described finds the right solution on a set of problems for which we can determine a good solution by analysis. The next experiments examine the scaling properties of the approach, i.e. how the performance, and in particular the execution speed, of the algorithm increases as the problem size increases.

4.1 Sample Scenario and Perturbations

This set of experiments involves a relatively small (though not trivially small) problem containing 24 job flows and 18 clusters and lasting for 6 epochs. Because of its symmetries, this particular problem lends itself to analysis by a human, so we can determine whether our approach finds a good solution. Perturbing the problem causes the optimal strategy to change. We introduce perturbations that include losses of resources in the present, anticipated losses of resources in the future, and surges in the loads, and we verify that the algorithm makes the proper adjustments to the policy.

Baseline Problem - We now describe the initial problem on which we test our approach. There are 18 clusters in total. Each cluster is specialized to handle one of the six steps of the jobs, whose sequence of steps is shown in Figure 1, with three clusters per step. Each cluster has a capacity of 13, with the underlying assumption that there are 13 identical compute resources aggregated at each cluster. The load-to-throughput map for each cluster is that shown in Figure 2.

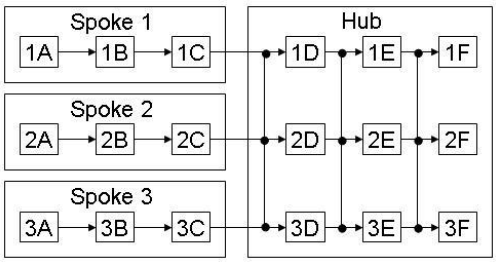

Fig. 8. The topology of the clusters is determined by the routing constraints on the flows. Steps A-C must all be performed in a prespecified one of the three spokes, while steps D-F can be performed in any of three clusters in the hub.

The routing constraints of the job flows induce an inherent connectivity on the clusters, which is the hub-and-spokes configuration shown in Figure 8. The first three steps, i.e. steps A-C of a job flow, are constrained to be executed in one of the three spokes. For example, some of the job flows are constrained to spoke 1, and hence must be assigned to clusters 1A, 1B and 1C for their first three steps. The final three steps, steps D-F, are handled in the hub, and the job flow is free to be assigned to any of the three clusters specializing in that step.

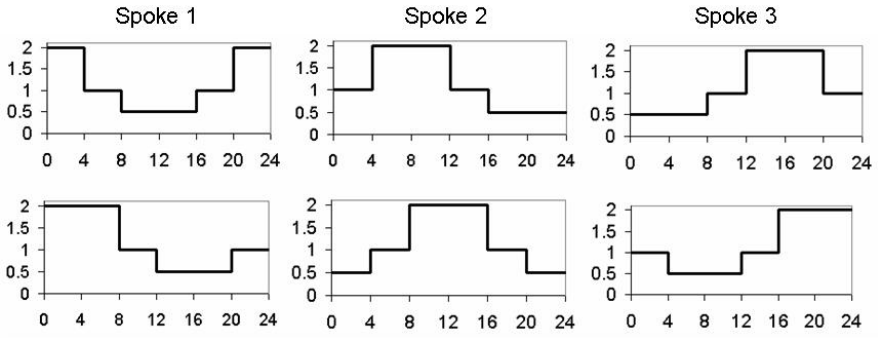

Fig. 9. The six different arrival rate patterns and their associated spikes

There are 24 different job flows. The job flows have all different combinations of the following three properties.

- There are two different utilities, high (numerical value = 2) and low (numerical value = 1).
- There are two different deadlines, short (numerical value = 1 hour) and long (numerical value = 16 hours).
- There are six different arrival rate patterns, i.e. arrival rates as a function of time. These are pictured in Figure 9. The six patterns are all cyclical over 24 hours and all essentially the same pattern with different offsets, so that the peaks and valleys

of each are at different times. (This captures in an idealized form the daily cycles in usage requests, with more requests during the local daytime.) Each pattern is associated with a particular spoke, with two patterns adjacent in their offsets assigned to each spoke.

The six steps in every job flow each require one minute to complete.

There are six epochs each of duration four hours. The entire problem covers a 24-hour period. The result of the optimization will be six routing/backlog policies, one for each epoch.

An analysis of this scenario yields the following. During epoch 1, spoke 1 if overloaded. There are eight flows associated with spoke 1, and each of the flows has arrival rate of 2 jobs/minutes. Therefore, there is an aggregate arrival rate of 16 jobs/minute that are constrained to use the clusters in spoke 1. These clusters have a capacity of 13 jobs/minute, and a maximum throughput of even less. So, not all these jobs can be processed during the first epoch. If all these jobs are allowed to enter the clusters, as opposed to being backlogged until future epochs, the high-utility flows will receive most of the throughput, with the low-utility jobs waiting in the input queues. Most of the low-utility, short-deadline jobs will time out and hence be dropped.

So, a better strategy is to backlog enough long-deadline flows from spoke 1 at the entry to the system to allow the short-deadline flows to all complete in the first epoch. These long-deadline jobs are released from backlog into the clusters of spoke 1 during epochs 3 and 4, when the there is spare capacity compared to the load due purely to arrivals.

Spoke 2 is similarly overloaded in epochs 2-4, with a peak in epoch 3. Hence, the best strategy is to backlog the flows from spoke 2 during epoch 3 and release them during epochs 5 and 6, when the arrival load is lightest.

The arrivals for spoke 3 peak during epoch 5. Because there are no epochs included beyond epoch 6, there is no advantage to backlogging the flows here, and hence the best strategy is to let all the jobs into the system and allow the local schedulers to give first priority to the high-utility jobs. [A lesson here is that it is important to include enough epochs beyond the last epoch whose optimized policy might actually be used so that all policies of interest are not influenced by this type of "boundary effect".]

As partially illustrated in Figure 10, the results from the optimization were as expected from the analysis, so the algorithm found an approximately optimal set of policies.

Perturbation 1: Current Loss of Hub Cluster - This scenario is the same as the baseline problem except with the capacity of cluster 1E set to zero for epochs 1 and 2. In the hub, unlike in the spokes, there is a choice of multiple clusters for each step of the job flows, and tasks that would have been assigned to the missing cluster can instead be sent to the two alternative clusters, 2E and 3E. Because the two clusters cannot quite handle the full load, some of the long-deadline jobs are backlogged until the anticipated return of the disabled cluster. Optimizing the policies produces

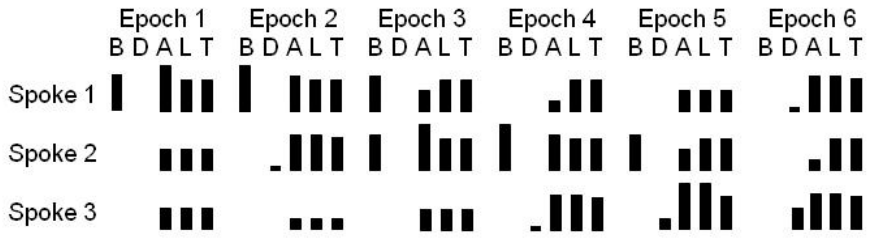

Fig. 10. A graphical depiction of the results for the baseline problem set. For each spoke in each epoch, the figure shows the relative size of the backlog (B), dropped jobs (D), aggregate arrival rate (A), cluster loads (L), and cluster throughputs (T). The units for A, L and T are jobs/minute, while those for B and D are jobs.

the expected behavior; this demonstrates how our approach can be used to modify the routing policy to adapt to changes in the distributed system.

Perturbation 2: Future Loss of Spoke Resources - This scenario is the same as the baseline except with the capacity of cluster 1B set to 6 instead of 13 in epochs 3-6. This anticipated future loss of resources changes the current (i.e., epoch 1) optimal backlog policy for the job flows associated with spoke 1. Since there no longer will be excess capacity available in the future, the best current policy is to complete the high-utility jobs and allow some of the low-utility jobs to be dropped. Our algorithm finds this new optimal policy, demonstrating the ability to adapt current policy to anticipated future changes in the distributed system.

Perturbation 3: Surge in Load in a Spoke - This scenario is the same as the baseline except one of the high-utility, long-deadline job flows in spoke 1 has an arrival rate that is increased from 2 jobs/minute to 5 jobs/minute during epochs 1 and 2. This surge means that the optimal backlog policy for spoke 1 now has to focus on completing all the high-utility jobs, letting the low-utility jobs be dropped during the first four epochs. Some of the additional high-utility jobs that are part of the surge are immediately sent to the clusters for processing, while others are backlogged until there is excess capacity in fuure epochs. Note that low-utility jobs continue to be dropped even after the surge has ceased in order to handle the backlog of high-utility jobs accumulated during the surge. Our approach finds this new policy, demonstrating the ability to adjust policy to adapt to changes in the load.

The solutions are generated within roughly 12 seconds on a single 2.8GHz CPU, showing that the approach not only finds a good solution but does so in a reasonably short time.

4.2 Scaling Properties

It is important to understand how our approach performs not just on relatively small problems but also on larger problems. A second set of experiments investigate the scalability of the algorithm, i.e. how increasing the size of the problem affects the algorithm's performance.

The problem size can vary along multiple different dimensions. We have identified what the different dimensions of problem scale are, and we have developed the capability for varying the problem size along one dimension at a time. This allows us to investigate the effects of changing only one, or some subset, of these dimensions, as well as all of them simultaneously. We now enumerate these different dimensions along with our theoretical analysis of how they effect search time:

- **(average) number of legal clusters per task -** For each task, the greedy selection process needs to evaluate the effect on the optimization criterion of assigning that task to each legal cluster. Since each such evaluation is independent of the others, the total time required is proportional to the number of legal clusters to evaluate. Therfore, the overall search time should scale linearly along this dimension.
- **total number of epochs -** For each epoch, the algorithm needs to perform a separate genetic algorithm run. These runs are independent, so the execution time should scale linearly in this dimension.
- **(average) number of epochs before a job's deadline -** For each step of the dynamic programming process, i.e. each task in the job, the algorithm maintains potentially one branch for each epoch before the deadline. For the next step of each branch, it can explore a number of branches that is on average half as many as the number of epochs before the deadline. So, the algorithm potentially scales as a square of this dimension. However, in practice, dynamic programming should eliminate most of these potential branches, and the scaling could be closer to linear.
- **(average) number of steps per job -** The dynamic programming process needs to take one more step in its chain for each step in the job. Since these steps are largely independent, we would predict linear scaling in this dimension.
- **number of job flows -** There are two ways that the number of job flows effects search time. For each individual in the genetic algorithm, the rapid route builder needs to route this many flows. Each flow is mostly independent (although not entirely independent because of competition for throughput at the clusters), so the time should increases linearly. Secondly, increasing the number of job flows increases the number of possible orderings of these flows, and hence the size of the search space for the genetic algorithm. This will increase the number of individuals the genetic algorithm must evaluate to find a near optimal one. Based on past experience with order-based genetic algorithms, we predict that the increase in the required number of evaluations is between linear and quadratic, but this is problem-dependent and can only be determined empirically.

Additionally, there can be assorted costs or savings due to secondary interactions. For example, a decrease in the capacity of each cluster can cause the flows to be split into more subflows during the rapid schedule building process, hence leading to longer execution times.

We have devised methods to increase the scale in one of these dimensions at a time maintaining approximately the same optimization problem.

- To increase the number of legal clusters per task by a factor of N, replace each cluster in the original problem with N clusters, each with 1/N times as much

capacity as the original. For every task for which the original cluster was legal, make all N new clusters be legal.

- To increase the number of epochs, convert each epoch in the original problem into N epochs identical to the original except with duration 1/N as long. Note that this changes both the total number of epochs and the number of epochs before the deadline by a factor of N.
- To increase the number of steps per job, convert each step of each job flow of the original problem into N steps identical to the original, in particular allowing the same legal clusters, except each requiring 1/N the time to complete.
- To increase the number of job flows, convert each job flow into N job flows each identical to the original except with arrival rate 1/N of the original rate.

We have applied these transformations to the baseline data set described in Section 4.1. For each transformed data set, we measure three quantities. One is the amount of time required for the full optimization to execute with the number of evaluations for the genetic algorithm specified to be 100. This measures the change in exeucution time of the rapid route builder. The second quantity is the number of evaluations required of the genetic algorithm to reach a near optimal solution. We determine this value by executing with different numbers of evaluations and finding at what point the result stops improving significantly (no more than 1%). The third quantity is the time required to reach this near optimal solution, which should approximately equal the product of the first two quantities divided by 100. These three quantities are shown in the following table for each of the datasets. Note that all runs are performed on the same single machine with a single 2.8GHz processor.

Dataset Description	**100-Eval Time (secs)**	**Evals for Optimum**	**Time for Optimum**
baseline	12	100	12
10x clusters	166	100	166
10x flows	352	8000	28160
10x epochs	1091	100	1091
10x steps	358	100	358
10x clusters 10x flows	720	4000	28800
2x clusters 2x flows 2x epochs 2x steps	406	250	837

Table 1. Results of the scaling experiments.

The results are largely as predicted with a few exceptions, which we now discuss. Perhaps the biggest deviation from predicted behavior is when the number of flows is increased by a factor of ten (10x flows). This leads to an 100-evaluation execution time that is 30 times larger than that for the baseline problem. This execution time was predicted to be linear in the number of flows, and hence we would have instead

expected a factor of 10. A possible explanation is that increasing the flows without increasing the number of clusters resulted in ten times as many flows at each cluster, leading to overhead in accounting, most importantly the propagation of retracted throughput, for all these flows. Note that when the number of flows and the number of clusters are both increased by a factor of ten, the 100-evaluation execution time is only increased by a factor of 60, which is even less than the factor of 100 predicted.

If the problem is such that the time it takes to optimize is longer than the desired time, there are some techniques to reduce the search time. The simplest is just to reduce the number of evaluations of the genetic algorithm, accepting the lesser quality of the solution. A similar method that may sacrifice less of the solution quality is based on the recognition that epochs further in the future are less important to optimize well than epochs closer to the present. Therefore, using less evaluations for the genetic algorithm on these future epochs leads to faster execution with an acceptable decrease in solution quality. An alternative method to decreasing optimization time is to decrease the number of epochs, number of flows, etc. by merging them, blurring some of the finer distinctions of the model (and hence the quality of the solution when applied to the real system), but decreasing the problem size. Again, this technique can be applied more heavily to the epochs further in the future to reduce the effects on the policies that need to be in place soon.

5 Conclusion and Future Work

We have defined a problem involving optimizing the routing and backlog policy of a large distributed computing system. Our approach to this scheduling problem involves a combination of dynamic programming and a genetic algorithm. This approach allows the optimization to proceed rapidly over a large search space while still finding good solutions. One set of experiments has proven the ability of the approach to find an optimal policy, while another set of experiments has demonstrated its scalability.

We have integrated this policy optimization algorithm into a prototype design tool and demonstrated its effectiveness on sample problems; the next steps involve moving this tool into an operational setting. Initially, it would be used offline with data collected from a functioning enterprise grid used to define the optimization problem. The ultimate goal is to integrate this policy optimization algorithm into an online adaptive controller that adjusts routing/backlog policies in real time based on automated data feeds.

References

1. Andresen, D. and T. McCune: 1998, ‘Towards a Hierarchical Scheduling System for Distributed WWW Server Clusters’. *Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing*.

2. Barolli, L., A. Koyama, K. Matsumoto, T. Suganuma, and N. Shiratori: 2002, 'A Genetic Algorithm Based Routing Method Using Two QoS Parameters'. *Proceedings of the 13th International Workshop on Database and Expert Systems Applications*.
3. Bose, A., B. Wickman, and C. Wood: 2004, 'MARS: A Metascheduler for Distributed Resources in Campus Grids'. *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*.
4. Casetti, C., R. Cigno, and M. Mellia: 1999, 'QoS-Aware Routing Schemes Based on Hierarchical LoadBalancing for Integrated Services Packet Networks'. *Proceedings of the IEEE International Communication Conference*.
5. Goldberg, D. and J. R. Lingle: 1985, 'Alleles, Loci, and the Traveling Salesman Problem'. *Proceedings of the First International Conference on Genetic Algorithms*. pp. 154–159.
6. Goswami, K., M. Devarakonda, and R. Iyer: 1993, 'Prediction-Based Dynamic Load-Sharing Heuristics'. *IEEE Transactions on Parallel and Distributed Systems*.
7. Grefenstette, J., R. Gopal, B. Rosmaita, and D. van Gucht: 1985, 'Genetic Algorithms for the Traveling Salesman Problem'. *Proceedings of the First International Conference on Genetic Algorithms*. pp. 160–165.
8. Grimme, C.: 2007, 'Grid Metaschedulers: An Overview and Up-to-date Solutions'. PowerPoint presentation.
9. Key, P. and L. Massoullie: 2006, 'Fluid Models of Integrated Traffic and Multipath Routing'. *Queueing Systems: Theory and Applications* **53**(1-2), 85–98.
10. Lo, V., D. Zhou, D. Zappala, Y. Liu, and S. Zhao: 2004, 'Cluster Computing on the Fly: P2P Scheduling of Idle Cycles in the Internet'. *International Workshop on Peer-to-Peer Systems*.
11. Mausolf, J.: 2005, 'Grid in Action: Managing the Resource Managers'. *IBM developerWorks*.
12. Okuhara, K., T. Tanaka, and H. Ishii: 2003, 'Routing and Flow Control by Genetic Algorithm for a Flow Model'. *Systems and Computers in Japan* **34**(1), 11–20.
13. Othman, O. and D. Schmidt: 2001, 'Issues in the Design of Adaptive Middleware Load Balancing'. *Proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems*. pp. 205–213.
14. Oueslati, S. and J. Roberts: 2006, 'Comparing Flow-Aware and Flow-Oblivious Adaptive Routing'. *40th Annual Conference on Information Sciences and Systems*. pp. 655–660.
15. Stone, H.: 1977, 'Multiprocessor Scheduling with the Aid of Network Flow Algorithms'. *IEEE Transactions on Software Engineering* **SE-3**(1), 85–93.
16. Strong, P.: 2005, 'Enterprise Grid Computing'. *ACM Queue* **3**(6).
17. Syswerda, G.: 1991, 'Schedule Optimization Using Genetic Algorithms'. In: L. Davis (ed.): *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, pp. 332–349.
18. Thain, D., T. Tannenbaum, and M. Livny: 2005, 'Distributed Computing in Practice: The Condor Experience'. *Concurrency and Computation: Practice and Experience* **17**(2-4), 323–356.
19. Vadhiyar, S. and J. Dongarra: 2002, 'A Metascheduler for the Grid'. *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*.
20. Whitley, D., T. Starkweather, and D. Fuquay: 1989, 'Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator'. *Proceedings of the Third International Conference on Genetic Algorithms*. pp. 133–140.