Quincy: Fair Scheduling for Distributed Computing Clusters

Michael Isard, Vijayan Prabhakaran, Jon Currey,
Udi Wieder, Kunal Talwar and Andrew Goldberg
Microsoft Research, Silicon Valley — Mountain View, CA, USA
{misard, vijayanp, jcurrey, uwieder, kunal, goldberg}@microsoft.com

ABSTRACT

This paper addresses the problem of scheduling concurrent jobs on clusters where application data is stored on the computing nodes. This setting, in which scheduling computations close to their data is crucial for performance, is increasingly common and arises in systems such as MapReduce, Hadoop, and Dryad as well as many grid-computing environments. We argue that data intensive computation benefits from a fine-grain resource sharing model that differs from the coarser semi-static resource allocations implemented by most existing cluster computing architectures. The problem of scheduling with locality and fairness constraints has not previously been extensively studied under this model of resource-sharing.

We introduce a powerful and flexible new framework for scheduling concurrent distributed jobs with fine-grain resource sharing. The scheduling problem is mapped to a graph datastructure, where edge weights and capacities encode the competing demands of data locality, fairness, and starvation-freedom, and a standard solver computes the optimal online schedule according to a global cost model. We evaluate our implementation of this framework, which we call Quincy, on a cluster of a few hundred computers using a varied workload of data- and CPU-intensive jobs. We evaluate Quincy against an existing queue-based algorithm and implement several policies for each scheduler, with and without fairness constraints. Quincy gets better fairness when fairness is requested, while substantially improving data locality. The volume of data transferred across the cluster is reduced by up to a factor of 3.9 in our experiments, leading to a throughput increase of up to 40%.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Scheduling*

General Terms

Algorithms, Design, Performance

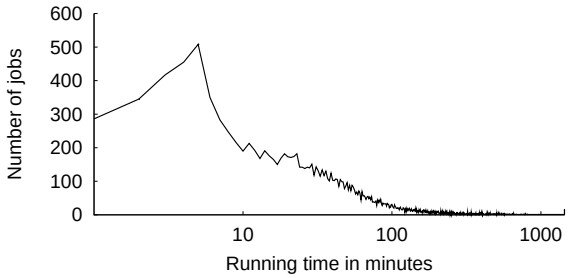

Figure 1: **Distribution of job running times from a production cluster used inside Microsoft's search division.** The horizontal axis shows the running time in minutes on a log scale, and the vertical axis shows the number of jobs with the corresponding running time.

Run time(m)	5	10	15	30	60	300
% Jobs	18.9	28.0	34.7	51.31	72.0	95.7

Table 1: **Job running time.** The table shows the same data as Figure 1 but here presented as the percentage of jobs under a particular running time in minutes.

1. INTRODUCTION

Data-intensive cluster computing is increasingly important for a large number of applications including web-scale data mining, machine learning, and network traffic analysis. There has been renewed interest in the subject since the publication of the MapReduce paper describing a large-scale computing platform used at Google [7].

One might imagine data-intensive clusters to be used mainly for long running jobs processing hundreds of terabytes of data, but in practice they are frequently used for short jobs as well. For example, the average completion time of a MapReduce job at Google was 395 seconds during September 2007 [8]. Figure 1 and Table 1 show data taken from a production cluster used by Microsoft's search division logging the duration of every job submitted over a period of 39 days from September 2008 to November 2008. While there are large jobs that take more than a day to complete, more than 50% of the jobs take less than 30 minutes. Our users generally strongly desire some notion of fair sharing of the cluster

resources. The most common request is that one user's large job should not monopolize the whole cluster, delaying the completion of everyone else's (small) jobs. Of course, it is also important that ensuring low latency for short jobs does not come at the expense of the overall throughput of the system. Section 5.2 sets out the exact definition of unfairness that we use to evaluate our system. Informally, however, our goal is that a job which takes t seconds to complete given exclusive access to a cluster should require no more than Jt seconds of execution time when the cluster is shared among J concurrent jobs. Following Amdahl's law, most jobs cannot continuously make use of the resources of the entire cluster, so we can hope that many jobs will complete in fewer than Jt seconds.

Many of the problems associated with data-intensive computing have been studied for years in the grid and parallel database communities. However, a distinguishing feature of the data-intensive clusters we are interested in is that the computers in the cluster have large disks directly attached to them, allowing application data to be stored on the same computers on which it will be processed. Maintaining high bandwidth between arbitrary pairs of computers becomes increasingly expensive as the size of a cluster grows, particularly since hierarchical networks are the norm for current distributed computing clusters [7, 17]. If computations are not placed close to their input data, the network can therefore become a bottleneck. Additionally, reducing network traffic simplifies capacity planning. If a job is always allocated a certain fraction of the cluster's computing resources then ideally its running time should remain approximately the same. When concurrent jobs have high cross-cluster network traffic, however, they compete for bandwidth, and modeling this dynamic network congestion greatly complicates performance prediction. For these reasons, optimizing the placement of computation to minimize network traffic is a primary goal of a data-intensive computing platform.

The challenge for a scheduling algorithm in our setting is that the requirements of fairness and locality often conflict. Intuitively this is because a strategy that achieves optimal data locality will typically delay a job until its ideal resources are available, while fairness benefits from allocating the best available resources to a job as soon as possible after they are requested, even if they are not the resources closest to the computation's data.

This paper describes our experience comparing a set of schedulers and scheduling policies that we operate on clusters containing hundreds of computers. The clusters are shared between tens of users and execute multiple jobs concurrently. Our clusters run the Dryad distributed execution engine [17], which has a similar low-level computational model to those of MapReduce[7], Hadoop [3] and Condor [29]. We believe that the Dryad computational model is well adapted to a fine-grain resource sharing strategy where every computer in the cluster is in general multiplexed between all of the running jobs. This is in contrast to traditional grid and high-performance computing models in which sets of cluster computers are typically assigned for exclusive use by a particular job, and these assignments change only rarely. We examine this argument in detail in Section 2.1.

The main contribution of this paper is a new, graph-based framework for cluster scheduling under a fine grain cluster resource-sharing model with locality constraints. We discuss related work in Section 7, however the problem of scheduling under this resource-sharing model has not been extensively studied before, particularly when fairness is also a goal. We show for the first time a mapping between the fair-scheduling problem for cluster computing and the classical problem of min-cost flow in a directed graph. As we demonstrate in Section 6, the global solutions found by the min-cost flow algorithm substantially outperform greedy scheduling approaches. The graph-based formulation is also attractive from a software-engineering standpoint, since scheduling policies and tuning parameters are specified simply by adjusting weights and capacities on a graph datastructure, analogous to a declarative specification. In contrast, most previous queue-based approaches encode policies using heuristics programmed as imperative subroutines and this can lead to increased code complexity as the heuristics become more sophisticated.

The structure of this paper is as follows. Section 2 sets out the details of our computing clusters and the type of jobs that they are designed to execute. Section 3 describes several traditional queue-based scheduling algorithms that we use as a baseline for our experiments. Section 4 introduces the mapping between scheduling and min-cost flow, and explains the design of the Quincy scheduler in detail. In Sections 5 and 6 we outline our experimental design and evaluate the performance of the system on a medium-sized cluster made up of several hundred computers. After reviewing related work in Section 7, we conclude with a discussion of lessons learned, limitations of the approach, and opportunities for future work.

2. THE PROBLEM SETTING

This section describes our computing environment, the type of jobs we execute, the definition of fairness we adopt, and the broad class of scheduling algorithms we consider. Not all distributed jobs or scheduling policies fit into our framework, but we postpone a discussion of how our system might be generalized until Section 8. We assume a homogeneous computing cluster under a single administrative domain, with many users compet-

ing for the cluster resources. There is a hierarchical network in which each rack contains a local switch and the rack switches are interconnected via a single core switch, so communication between computers in the same rack is "cheaper" than communication between racks. Our methods extend trivially to networks with no hierarchy (where the cost of communication is approximately equal between any pair of computers) and those with a deeper hierarchy.

2.1 Computational model

Our clusters are used to run jobs built on the Dryad distributed execution platform [17, 31]. Each job is managed by a "root task" which is a process, running on one of the cluster computers, that contains a state machine managing the workflow of that job. The computation for the job is executed by "worker tasks" which are individual processes that may run on any computer. A worker may be executed multiple times, for example to recreate data lost as a result of a failed computer, and will always generate the same result. This computational model is very similar to that adopted by MapReduce [7], Hadoop [3] and Condor [29] and the ideas in this paper should be readily applicable to those systems.

A job's workflow is represented by a directed acyclic graph of workers where edges represent dependencies, and the root process monitors which tasks have completed and which are ready for execution. While running, tasks are independent of each other so killing one task will not impact another. This independence between tasks is in contrast to multi-processor approaches such as coscheduling [24] and programming models like MPI [2] in which tasks execute concurrently and communicate during their execution. In fact, the data-intensive computational model leads to a somewhat different approach to resource sharing compared to traditional high performance computing clusters.

When a cluster is used to execute MPI jobs, it makes sense to devote a specific set of computers to a particular job, and change this allocation only infrequently while the job is running, for example when a computer fails or a higher priority job enters the system. This is because MPI jobs are made up of sets of stateful processes communicating across the network, and killing or moving a single process typically requires the restart of all of the other processes in the set. In addition, high-performance computing clusters traditionally do not have a large quantity of direct-attached storage so while it may be advantageous to ensure that the processes in a job are scheduled close to each other in network topology, the exact set of computers they run on does not have a major effect on performance. It is therefore not usually worth moving a job's processes once they have been allocated a fixed set of resources.

In contrast, workloads for systems such as Dryad and MapReduce tend to be dominated by jobs that process very large datasets stored on the cluster computers themselves. It generally makes sense to stripe each large dataset across all the computers in the cluster, both to prevent hot spots when multiple jobs are concurrently reading the same data and to give maximum throughput when a single job is executing. In addition, as noted above, tasks run independently so they can start and complete in a staggered fashion as resources become available, and killing a task only forfeits the work that task has done without affecting other running processes. Since tasks are staggered rather than executing in lockstep, most jobs fluctuate between periods in which they have many more ready tasks than there are available cluster resources, and periods in which a few straggler tasks are completing. The number of computers it is worth assigning to a job is therefore continuously changing.

Consequently, a natural way to share a cluster between multiple Dryad jobs is to multiplex access to all the cluster computers across all running jobs, so a computer is in general assigned to a different job once its current task completes, rather than giving any job a long-term private allocation. This is the primary reason that new approaches such as the scheduling framework described in this paper are needed, and standard techniques such as gang scheduling [10] are not applicable to our setting.

In this paper we consider two broad techniques for giving a job its fair share of the cluster's resources: running the job's worker tasks in sub-optimal locations instead of waiting for the ideal computer to be available, which reduces latency but increases network traffic; and killing running tasks of one job to free resources for another job's tasks, which can improve the latter job's latency without sacrificing its data locality but wastes the work of the preempted task. Since both of these strategies can harm overall throughput, there is potentially a penalty to be paid for improving fairness. As we show in Section 6, scheduling algorithms that optimize a global cost function do a better job of managing this tradeoff than those which rely on greedy heuristics.

2.2 Cluster architecture

Figure 2 shows an outline of our cluster architecture. There is a single centralized scheduling service running in the cluster maintaining a batch queue of jobs. At any given time there may be several concurrent jobs sharing the resources of the cluster and others queued waiting for admission. When a job is started the scheduler allocates a computer for its root task. If that computer fails the job will be re-executed from the start. Each running job's root task submits its list of ready workers, and their input data summaries as defined below in Section 2.3, to the scheduler. The scheduler then matches tasks to comput-

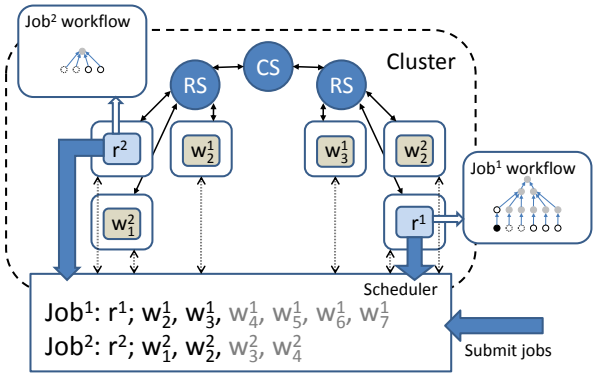

Figure 2: **The cluster scheduling architecture.** The figure shows a small cluster with 6 computers organized into two racks. Each rack contains a switch RS that communicates with a core switch CS. Each running job j first starts a root task r^j that submits worker tasks to the scheduler according to a job-specific workflow. The scheduler determines which tasks should be active at any given time, matches them to available computers, and sets them running. The worker tasks shown in gray are ready to run but have not yet been assigned to a computer.

ers and instructs the appropriate root task to set them running. When a worker completes, its root task is informed and this may trigger a new set of ready tasks to be sent to the scheduler. The root task also monitors the execution time of worker tasks, and may for example submit a duplicate of a task that is taking longer than expected to complete [7, 17]. When a worker task fails because of unreliable cluster resources, its root task is responsible for back-tracking through the dependency graph and resubmitting tasks as necessary to regenerate any intermediate data that has been lost. The scheduler may decide to kill a worker task before it completes in order to allow other jobs to have access to its resources or to move the worker to a more suitable location. In this case the scheduler will automatically instruct the worker's root so the task can be restarted on a different computer at a later time.

The scheduler is not given any estimates of how long workers will take, what resources they will consume, or given insight into the upcoming workers that will be submitted by jobs once running tasks complete. Section 8 discusses ways in which our system could be enhanced if such information were available.

2.3 Data locality

A worker task may read data from storage attached to computers in the cluster. These data are either inputs to the job stored as (possibly replicated) partitioned files in a distributed file system such as GFS [12] or intermediate output files generated by upstream worker tasks. A worker is not submitted to the scheduler until all of its input files have been written to the cluster, at which time their sizes and locations are known. Consequently the scheduler can be made aware of detailed information about the data transfer costs that would result from executing a worker on any given computer. This information is summarized as follows.

When the nth worker in job j, denoted w_n^j, is ready, its root task r^j computes, for each computer m, the amount of data that w_n^j would have to read across the network if it were executed on m. The root r^j then constructs two lists for task w_n^j: one list of preferred computers and one of preferred racks. Any computer that stores more than a fraction δ^c of w_n^j's total input data is added to the first list, and any rack whose computers in sum store more than a fraction δ^r of w_n^j's total input data is added to the second. In practice we find that a value of 0.1 is effective for both δ^c and δ^r. These settings ensure that each list has at most ten entries and avoids marking every computer in the cluster as "preferred" when a task reads a large number of approximately equal-sized inputs, for example when a dataset is being re-partitioned. Some schedulers, including our flow-based algorithms, can make use of fine-grain information about the number of bytes that would be transferred across the network as a result of scheduling w_n^j on computer m. The precise information we send to the Quincy scheduler is described in the Appendix.

2.4 Fairness in a shared cluster

As stated in the introduction, our goal for fairness is that a job which runs for t seconds given exclusive access to a cluster should take no more than Jt seconds when there are J jobs concurrently executing on that cluster. In practice we only aim for this guarantee when there is not heavy contention for the cluster, and we implement admission control to ensure that at most K jobs are executing at any time. When the limit of K jobs is reached subsequent jobs are queued and started, in order of submission time, as running jobs complete. During periods of contention there may therefore be a long delay between a job's submission and its completion, even if its execution time is short. Setting K too large may make it hard to achieve fairness while preserving locality, since it increases the likelihood that several jobs will be competing for access to data stored on the same computer. However with too small a value of K some cluster computers may be left idle if the jobs do not submit enough tasks. Section 8 briefly discusses possible mechanisms for setting K adaptively.

A job is instantaneously allocated a fraction α of the cluster by allowing it to run tasks on αM of the cluster's M computers. The allocated fraction α can change over the running time of the job as other jobs enter and leave the system, as can the particular computers that the job's tasks are scheduled on. Of course this is a simplified model of sharing. We do not currently attempt to give jobs a fair share of network resources, or account for the resources consumed when a job's task reads remote data

from a computer that is assigned to another job. Also, we currently restrict each computer to run only one task at a time. This is a conservative policy that is partly determined by the fact that jobs do not supply any predictions about the resource consumption of their workers. Section 8 discusses ways of relaxing some of these restrictions.

Our implementation aims to give each running job an equal share of the cluster. However, the methods we describe can straightforwardly be extended to devote any desired fraction of the cluster to a given job, and a system could implement priorities and user quotas by choosing these fractions appropriately. In the case that controlling end-to-end job latency is important even when the cluster is suffering contention, one could implement a more sophisticated queueing policy, for example to let jobs that are predicted to complete quickly "jump the queue" and begin execution ahead of jobs that are expected to be long-running.

3. QUEUE-BASED SCHEDULING

This section describes a number of variants of a queue-based scheduling approach that we will use as a baseline for comparisons with our new flow-based framework. It seems natural to use a set of queues as the fundamental datastructure for a locality-based scheduler. The Hadoop scheduler adopts this model [1] and, while there are no published details of the MapReduce scheduler [7], it appears that queue-based approaches are the standard design for the public state of the art. (As far as we know, the Condor project [29] does not include any schedulers that make use of fine-grain locality information.)

We include fairly sophisticated variants of the queue-based approach that, we believe, extend previously published work. This exhaustive treatment is an attempt to provide a fair comparison between queue-based scheduling and flow-based scheduling as organizing frameworks, rather than concentrating on a particular implementation. We are familiar with queue-based schedulers since they have been used extensively by Dryad in the past, and this section surveys our good-faith attempt to implement effective fair scheduling in this model.

3.1 Outline of a queue-based architecture

The schedulers in this section maintain a number of queues as illustrated in Figure 3: one for each computer m in the cluster (denoted C_m), one for each rack l (denoted R_l), and one cluster-wide queue denoted X. When a worker task is submitted to the scheduler it is added to the tail of multiple queues: C_m for each computer m on its preferred computer list, R_l for each rack l on its preferred rack list, and X. When a task is matched to a computer using one of the algorithms below it is removed from all the queues it had been placed in. This family of

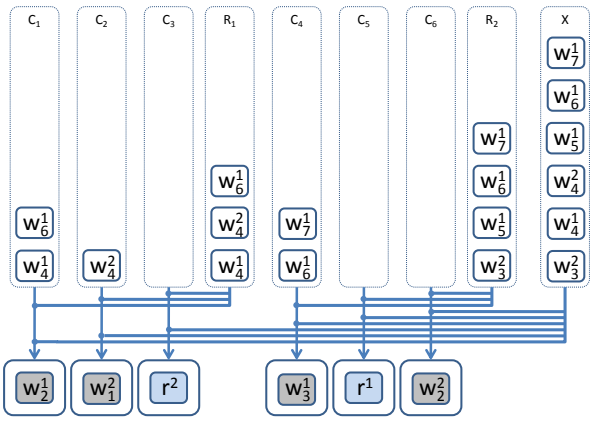

Figure 3: **A queue-based greedy scheduler.** The figure shows a scheduler for a small cluster with three computers organized into two racks. The scheduler maintains queues for each computer and rack, and a cluster-wide queue denoted X. Two jobs are running with root tasks on computers 5 and 3 respectively. Worker tasks are submitted to the scheduler by the root tasks, annotated with a list of preferred computers and a list of preferred racks. On arrival a worker task is added to the tail of the queue for each of its preferred computers and racks, and also to the tail of the cluster-wide queue. For example, job 1 submitted task 6 with preferred computers 1 and 4 and racks 1 and 2. When a computer becomes free the scheduler chooses among the available tasks to run on it using one of the policies described in Section 3. When a task is matched to a computer it is removed from all the queues it had been added to.

algorithms ignores exact data transfer sizes and treats all preferred computers as equal.

When a new job is started its root task is assigned a computer at random from among the computers that are not currently executing root tasks, and any worker task currently running on that computer is killed and re-entered into the scheduler queues as though it had just been submitted. K must be small enough that there are at least $K + 1$ working computers in the cluster in order that at least one computer is available to execute worker tasks.

3.2 Baseline algorithm without fairness

The simplest algorithm we consider schedules worker tasks independent of which job they come from, so there is no attempt to achieve fairness. Whenever a computer m becomes free, the first ready task on C_m, if any, is dispatched to m. If C_m does not have any ready tasks, then the first ready task on R_l is dispatched to m, where l is m's rack. If neither C_m nor R_l contains a ready task then the first ready task, if any, on X is dispatched to m. We refer to this algorithm as "G" for "greedy."

This algorithm aims to keep all computers busy at all times as long as there are worker tasks available to run. It will always try to match tasks to their preferred computers or racks where possible, in order of task submission. This can have two adverse effects on latency: if a job submits a large number of tasks on every computer's queue

then other jobs will not execute any workers until the first job's tasks have been run; and in a loaded cluster a task that has no preferred computers or racks may wait for a long time before being executed anywhere since there will always be at least one preferred task ready for every computer or rack.

3.3 Simple greedy fairness

A simple concept of fairness can be used to reduce the chance that a job that starts up and submits a large number of workers to all the scheduler's queues will starve subsequent jobs from receiving resources on which to run their tasks. We implement the same fairness scheme as the Hadoop Fair Scheduler [1]. In our implementation it is achieved by "blocking" a job, which is defined as follows. When a job is blocked its waiting tasks will not be matched to any computers, thus allowing unblocked jobs to take precedence when starting new tasks. The matching procedure in Section 3.2 above pulls the first "ready" task from a queue. In the simple greedy algorithm every task in any queue is always considered to be ready. In order to implement simple fairness we instead define a task to be ready only if its job is not blocked. For now we do not consider killing running tasks, so any workers that have already been started when a job becomes blocked continue to run to completion.

Now, whenever a computer becomes free and we want to match a new task to it, we must first determine which jobs are blocked. At this time each job is given an instantaneous allocation corresponding to the number of computers it is currently allowed to use. Job j gets a baseline allocation of $A_j^* = \min(\lfloor M/K \rfloor, N_j)$ where the cluster contains M computers, there are K running jobs, and N_j is the number of workers that j currently has either running or sitting in queues at the scheduler. If $\sum_j A_j^* < M$ the remaining slots are divided equally among jobs that have additional ready workers to give final allocations A_j where $\sum_j A_j = \min(M, \sum_j N_j)$. As jobs start and end, and new workers are submitted or stop running, the allocation for a given job will fluctuate. This fair-sharing allocation procedure could easily be replaced by a different algorithm, for example one that takes into account job priorities or group quotas and allocates different numbers of tasks to different running jobs.

The simplest fair algorithm we consider is denoted "GF" and blocks job j whenever it is running A_j tasks or more. This prevents j from scheduling any more tasks until its running total falls below its allocation, allowing other tasks to gain their share of the cluster. In the steady state where all jobs have been running for a long time, each job always receives exactly its allocated number of computers. Whenever a new job starts up, however, or a job that had a small number of ready tasks $N_j < \lfloor M/K \rfloor$ submits a new batch of tasks, there will be a period of unfairness while pre-existing tasks from now-blocked jobs are allowed to run to completion. This period can be prolonged in the case that some jobs include very long-lived worker tasks.

3.4 Fairness with preemption

The problem of a job "hogging" the cluster with a large number of long-running tasks can be addressed by a more proactive approach to fairness: when a job j is running more than A_j workers, the scheduler will kill its over-quota tasks, starting with the most-recently scheduled task first to try to minimize wasted work. We refer to this algorithm as "GFP."

As long as a job's allocation never drops to zero, it will continue to make progress even with preemption enabled, since its longest-running task will never be killed and will therefore eventually complete. To achieve this starvation-freedom guarantee the cluster must contain at least $2K$ working computers if there are K concurrent jobs admitted: one for each job's root task and one to allocate to a worker from each job.

3.5 Sticky slots

One drawback of simple fair scheduling is that it is damaging to locality. Consider the steady state in which each job is occupying exactly its allocated quota of computers. Whenever a task from job j completes on computer m, j becomes unblocked but all of the other jobs in the system remain blocked. Consequently m will be assigned to one of j's tasks again: this is referred to as the "sticky slot" problem by Zaharia *et al.* [32] because m sticks to j indefinitely whether or not j has any waiting tasks that have good data locality when run on m.

A straightforward solution is to add some hysteresis, and we implement a variant of the approach proposed by Zaharia *et al.* With this design a job j is not unblocked immediately if its number of running tasks falls below A_j. Instead the scheduler waits to unblock j until either the number of j's running tasks falls below $A_j - M_H$, where M_H is a hysteresis margin, or Δ_H seconds have passed. In many cases this delay is sufficient to allow another job's worker, with better locality, to "steal" computer m. Variants of GF and GFP with hysteresis enabled are denoted "GFH" and "GFPH" respectively.

4. FLOW-BASED SCHEDULING

As queue-based scheduling approaches are extended to encompass fairness and preemption, the questions of *which* of a job's tasks should be set running inside its quota A_j, or which should be killed to make way for another job, become increasingly important if we wish to achieve good locality as well as fairness. In the previous section we adopted heuristics to solve these problems, based around a greedy, imperative approach that consid-

ered a subset of the scheduler's queues at a time, as each new task arrived or left the system.

In this section we introduce a new framework for concurrent job scheduling. The primary datastructure used by this approach is a graph that encodes both the structure of the cluster's network and the set of waiting tasks along with their locality metadata. By assigning appropriate weights and capacities to the edges in this graph, we arrive at a declarative description of our scheduling policy. We can then use a standard solver to convert this declarative policy to an instantaneous set of scheduling assignments that satisfy a global criterion, considering all jobs and tasks at once.

The primary intuition that allows a graph-based declarative description of our problem is that there is a quantifiable cost to every scheduling decision. There is a data transfer cost incurred by running a task on a particular computer; and there is also a cost in wasted time to killing a task that has already started to execute. If we can at least approximately express these costs in the same units (for example if we can make a statement such as "copying 1 GB of data across a rack's local switch costs the same as killing a vertex that has been executing for 10 seconds") then we can seek an algorithm to try to minimize the total cost of our scheduling assignments.

Having chosen a graph-based approach to the scheduling problem, there remains the question of exactly what graph to construct and what solver to use. In this paper we describe a particular form of graph that is amenable to exact matching using a min-cost flow solver. We refer to this combination of graph construction and matching algorithm as the Quincy scheduler, and show in Section 6 that it outperforms our queue-based scheduler for every policy we consider.

4.1 Min-cost flow

We choose to represent the instantaneous scheduling problem using a standard flow network [11]. A flow network is a directed graph each of whose edges e is annotated with a non-negative integer capacity y_e and a cost p_e, and each of whose nodes v is annotated with an integer "supply" ϵ_v where $\sum_v \epsilon_v = 0$. A "feasible flow" assigns a non-negative integer flow $f_e \leq y_e$ to each edge such that, for every node v,

$$\epsilon_v + \sum_{e \in \mathcal{I}_v} f_e = \sum_{e \in \mathcal{O}_v} f_e$$

where $\mathcal{I}_v$ is the set of incoming edges to v and $\mathcal{O}_v$ is the set of outgoing edges from v. In a feasible flow, $\epsilon_v + \sum_{e \in \mathcal{I}_v} f_e$ is referred to as the flow through node v. A min-cost feasible flow is a feasible flow that minimizes $\sum_e f_e p_e$.

It is possible to specify a minimum flow $z_e \leq y_e$ for an edge e from node a to node b without altering the other costs or capacities in the graph. This is achieved by setting a's supply to $\epsilon_a - z_e$, b's supply to $\epsilon_b + z_e$, and e's capacity to $y_e - z_e$, and we make use of this construction to set both upper and lower bounds for fairness as described below.

Known worst-case complexity bounds on the min-cost flow problem for a graph with E edges and V nodes are $O(E \log(V)(E + V \log(V)))$ [23] and $O(VE \log(VP) \log(V^2/E))$ [14], where P is the largest absolute value of an edge cost. For this paper we use an implementation of the latter algorithm, described in [13] and provided free for research and evaluation by IG systems at `www.igsystems.com`.

4.2 Encoding scheduling as a flow network

The scheduling problem described in Section 2 can be encoded as a flow network as shown in Figure 4 and explained in detail in the Appendix. This graph corresponds to an instantaneous snapshot of the system, encoding the set of all worker tasks that are ready to run and their preferred locations, as well as the running locations and current wait times and execution times of all currently-executing workers and root tasks. One benefit, and potential pitfall, of using a flow network formulation is that it is easy to invent complicated weights and graph structures to attempt to optimize for particular behaviors. For elegance and clarity we have attempted to construct the simplest possible graph, with the fewest constants to set, that allows us to adjust the tradeoff between latency and throughput. The only hard design constraint we have adopted is starvation-freedom: we can prove as sketched in the Appendix that under weak assumptions every job will eventually make progress, even though at any moment some of its tasks may be preempted to make way for other workers.

The overall structure of the graph can be interpreted as follows. A flow of one unit along an edge in the graph can be thought of as a "token" that corresponds to the scheduling assignment of one task. Each submitted worker or root task on the left hand side receives one unit of flow as its supply. The sink node S on the right hand side is the only place to "drain off" the flow that enters the graph through the submitted tasks, so for a feasible flow each task must find a path for its flow to reach the sink. All paths from a task in job j to the sink lead either through a computer or through a node U_j that corresponds to leaving the task unscheduled. Each computer's outgoing edge has unit capacity, so if there are more tasks than computers some tasks will end up unscheduled. By controlling the capacities between U_j and the sink we can control fairness by setting the maximum and minimum number of tasks that a job may leave unscheduled, and hence the maximum and minimum number of computers that the algorithm can allocate to it for running tasks.

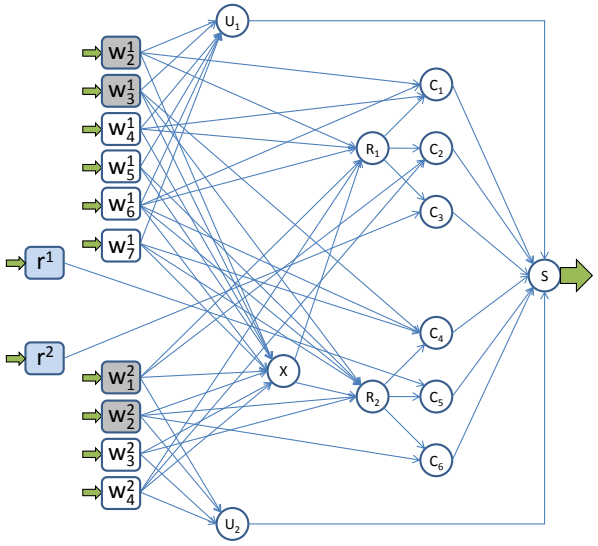

Figure 4: **The Quincy flow-based scheduler graph.** The figure shows a graph corresponding to the same set of jobs and tasks as the queue-based scheduler in Figure 3. There are nodes in the graph for each root and worker task that the scheduler knows about, as well as an "unscheduled node" U_j for each job j. There is also a node C_m for each computer m, a "rack aggregator" node R_l for each rack l, and a "cluster aggregator" node X. S is the sink node through which all flows drain from the graph. Each root task has a single outgoing edge to the computer where it is currently running. Each worker task in job j has an edge to j's unscheduled node U_j, to the cluster-aggregator node X, and to every rack and computer in its preferred lists. Workers that are currently executing (shown shaded) also have an edge to the computer on which they are running. Graph edges have costs and capacities that are not shown in the figure. The Appendix has a detailed explanation of the structure of this graph and the costs and capacities that allow us to map a min-cost feasible flow to a fair scheduling assignment.

Unlike the queue-based algorithms described in Section 3, a flow-based algorithm can easily make use of fine-grain information about the precise number of bytes of input a given worker task w_n^j would read across the rack and core switches if executed on a given computer m. This information is encoded in our graph construction using the costs on each of the edges leading from graph node w_n^j to computer nodes C_m, rack aggregators R_l, and the cluster aggregator X. The cost on the edge from w_n^j to C_m is a function (given in the Appendix) of the amount of data that would be transferred across m's rack switch and the core switch if w_n^j were run on computer m. The cost on the edge from w_n^j to R_l is set to the worst-case cost that would result if the task were run on the least favorable computer in the lth rack. The cost on the edge from w_n^j to X is set to the worst-case cost for running the task on any computer in the cluster.

In principle we could eliminate the cluster and rack aggregator nodes X and R_l from the graph construction and insert an edge between every task and every computer weighted by the exact cost of running the task on that computer. This would cause both the communication bandwidth between the root task and the scheduler and the number of edges in the graph to scale linearly with the number of computers in the cluster, so we adopt the conservative approximation above. Since, as noted in Section 2.3, the preferred lists for each task are of bounded size, the communication bandwidth and the number of edges needed for a given job remain constant as a function of the size of the cluster.

The cost on the edge connecting w_n^j to U_j represents the penalty for leaving w_n^j unscheduled. This increases over time so that a worker that has been queued for a long time is more likely to be set running than one which was recently submitted, and can also be used to encode priorities if some workers within a task should be scheduled in preference to others.

Finally, when a task starts running on computer m we add an additional cost, increasing with time, to all edges from that task to nodes other than C_m, to enforce a penalty for killing or moving it and thereby wasting the work it has consumed. All the edge costs are set out in detail in the Appendix.

The tradeoffs controlling Quincy's behavior are parameterized using just three scaling weights for the costs: ω which sets the cost of waiting in the queue, and ξ and ψ which determine the cost of transferring data across the core switch and a rack switch respectively. As we shall see in the next section, setting higher data costs causes Quincy to optimize more aggressively for data locality, preempting and restarting tasks when more suitable computers become available. Choosing the right parameters to match the network capacity has a substantial performance benefit.

The min-cost flow algorithm performs a global search for the instantaneous optimal scheduling solution, with respect to the costs we have assigned, subject to fairness constraints. The scheduler updates the graph whenever a salient event occurs (a worker task completes, a new task is submitted, etc.), and on a regular timer event since some costs are time-dependent as explained above. Whenever the graph is updated the scheduler computes a new min-cost flow then starts or kills tasks as necessary to reflect the matching induced by the flow.

The particular scheduling problem presented by our cluster computation model maps very elegantly to min-cost flow. We believe there are many other scheduling tasks that could benefit from a similar flow-based solution, however the constraints imposed by the graph encoding mean it is not suitable for all problems. We discuss these issues in more detail in Section 8.

4.3 Controlling fairness policy

As explained in the previous section, the capacities on the outgoing edge of job j's unscheduled node U_j control the number of running tasks that the job will be allocated. This allocation controls the fairness regime, and we con-

sider four regimes. Here, E_j is the minimum number of running tasks job j may receive and F_j is the maximum number of running tasks it may get. Each job j has submitted N_j worker tasks where $\sum_j N_j = N$, and the cluster contains M computers.

• **Fair sharing with preemption**: For this policy, fair shares A_j are computed as described in Section 3.3, and $E_j = F_j = A_j$ so job j is constrained to use exactly its share of the cluster computers. Unlike the greedy algorithm in Section 3, Quincy may kill tasks at any time in pursuit of a lower-cost flow solution, so a running task may be killed even if its job's allocation does not decrease—in which case another task from that job will be started to maintain its allocation. This policy is denoted "QFP" for Quincy with Fairness and Preemption.

• **Fair sharing without preemption**: Fair shares are computed as above but if a job is already running more than its fair share of workers its allocation is increased to that total and the allocations for other jobs are proportionally decreased. In addition the capacities of all edges to all nodes except the current running computer are set to zero for running tasks, so Quincy is prevented from preempting any running worker. This policy is denoted "QF".

• **Unfair sharing with preemption**: Each job j is allowed as many computers as necessary to minimize the overall cost of the scheduling assignments, so $E_j = 1$ and $F_j = N_j$. However, if $N \leq M$ we set $E_j = F_j = N_j$ to reduce the chance of idle computers when there are waiting tasks. A computer may be left idle for example if a waiting task w_n^j has a strong data-locality preference for a computer that is currently occupied but the wait-time cost has not yet overcome the cost of scheduling the task in a suboptimal location. This policy is denoted "QP."

• **Unfair sharing without preemption**: Allocations are set as for QP but capacities are adjusted as for QF to prevent Quincy from preempting any running jobs. This policy is denoted "Q".

Note that selecting between policies is achieved simply by changing a set of edge capacities. While the implementations in Section 3.3 are not conceptually complicated, implementing them efficiently is certainly more complex than modifying a single datastructure.

5. EXPERIMENTAL DESIGN

Our cluster contained 243 computers each with 16 GB of RAM and two 2.6 GHz dual-core AMD Opteron processors, running Windows Server 2003. Each computer had 4 disk drives and the workload data on each computer was striped across all 4 drives. The computers were organized into 8 racks, each containing between 29 and 31 computers, and each computer was connected to a 48-port full-crossbar GBit Ethernet switch in its local rack via GBit Ethernet. Each local switch was in turn connected to a central switch via 6 ports aggregated using 802.3ad link aggregation. This gave each local switch up to 6 GBits per second of full duplex connectivity. Our research cluster had fairly high cross-cluster bandwidth, however hierarchical networks of this type do not scale easily since the central switch rapidly becomes a bottleneck, so many clusters are less well provisioned than ours for communication between computers in different racks. Consequently for some experiments (which we denote "constrained network") in Section 6 we removed 5 of the six uplinks from each local switch reducing its full-duplex connectivity to 1 GBits per second.

5.1 Applications

Our experimental workload consists of a selection of representative applications that are run concurrently. We consider a small set of applications, but we can run each application with a variety of inputs. A particular application reading from a particular set of inputs is called an application instance. A given application instance always uses the same input partitions across all experiments.

Each experiment runs a list of application instances, and experiments that compare between policies run the same instances started in the same order. Each experiment has a concurrency value K and at the beginning of the experiment we start the first K instances in the list. Every time an instance completes the next instance in the list is started. The applications in our experiments are as follows:

• **Sort.** This application sorts a set of 100 byte records that have been distributed into S partitions. We consider three instances of Sort with $S = 10, 40$, and 80, denoted Sort10, Sort40, and Sort80 respectively. Each partition contains around 4 GB of data and the partitions are spread randomly over the cluster computers in order to prevent hot spots. The application starts by reading all the input data, sampling it to compute balanced ranges, then range partitioning it into S partitions. Sort redistributes all of its input data between computers during this repartitioning and is therefore inherently network-intensive. Once the data has been repartitioned each computer sorts its range and writes its output back to the cluster.

• **DatabaseJoin.** This application implements a 3-way join over two input tables of size 11.8 GB and 41.8 GB respectively, each range-partitioned into 40 partitions using the same ranges and arranged on the cluster so that matching ranges are stored on the same computer. The application spends most of its time in two phases, each using 40 tasks. In the first phase each task reads a matching pair of partitions and writes its output into 40 buckets, one for each range. In the second phase each

task reads one partition from the larger of the input tables along with 40 inputs generated by the tasks in the first phase. This second phase is inherently network-intensive but has a strong locality preference for the computer where the input table partition is stored. We run two instances of the DatabaseJoin application called DatabaseJoin40 and DatabaseJoin5. For the first, the inputs are distributed over 40 computers each of which holds a single partition. For the second the inputs are only distributed across 5 computers in the same rack, each of which stores 8 partitions. DatabaseJoin5 represents a classic tradeoff between throughput and data-locality: if all 40 tasks are run concurrently they must transfer most of their inputs across the network. In order to minimize network transfers only 5 tasks can be run at a time.

• **PageRank.** This is a graph-based computation with three iterations. Its input dataset is divided into partitions, each around 1 GB, stored on 240 computers. Each iteration is in two phases. Each task in the first phase reads from an input partition and from the output of the previous iteration, and writes 240 buckets. Each task in the second phase reads one bucket from each of the first-phase outputs, repartitioning the first-phase output. Tasks in the first phase have strong locality preference to execute where their input partition is stored, but second-phase tasks have no locality preference. PageRank is inherently network-intensive.

• **WordCount.** This application computes the occurrence frequency of each word in a large corpus and reports the ten most common words. There are ten instances with inputs divided into 2, 4, 5, 6, 8, 10, 15, 20, 25, and 100 partitions respectively, and each partition is about 50 MB in size. WordCount performs extensive data reduction on its inputs, so if it is scheduled optimally it transfers very little data over the network.

• **Prime.** This is a compute-intensive application which reads a set of integers and checks each for primality using a naive algorithm that uses $O(n)$ computations to check integer n. This algorithm was deliberately selected to have a high ratio of CPU usage to data transfer. There are two types of instance: PrimeLarge which tests 500,000 integers per partition, and PrimeSmall which checks 40,000 integers per partition. We use a single instance of PrimeLarge with 240 partitions and 13 PrimeSmall instances with 2, 4, 5, 6, 8, 10, 15, 20, 25, 500, 1000, 1500, and 2000 partitions respectively. No Prime instance transfers much data regardless of where it is scheduled.

Together these instances comprise 30 jobs with a mix of CPU, disk, and network intensive tasks. We always start PrimeLarge first, and the first thing it does is occupy most computers with worker tasks that take 22 minutes to complete. This simulates the case that one job manages to "hog" the resources, e.g. by starting up overnight while the cluster is unloaded. Schedulers that do not preempt tasks must share out a small number of remaining computers until PrimeLarge's workers start completing. The experiments in Section 6.3 remove PrimeLarge from the workload, and confirm that even without a single large job occupying the cluster we still see a large discrepancy between the fairness of different implementations.

5.2 Metrics

We use the following metrics to evaluate scheduling policies:

• **Makespan.** This is the total time taken by an experiment until the last job completes.

• **System Normalized Performance (SNP).** SNP is the geometric mean of all the ANP values for the jobs in an experiment, where ANP stands for the "Application normalized performance" of a job [34]. ANP_j, the ANP of job j, is the ratio of j's execution time under ideal conditions to j's execution time in the experiment of interest. Larger values of ANP and hence SNP are better, where a value of 1 is ideal.

• **Slowdown norm.** We compute some scaled l_p norms of the slowdown factors of the jobs across each experiment, where the slowdown of job j is $\sigma_j = 1/\text{ANP}_j$. The quantities we report are $l_1 = (\sum_j \sigma_j)/\mathcal{J}$, $l_2 = \sqrt{(\sum_j \sigma_j^2)}/\sqrt{(\mathcal{J})}$ and $l_\infty = \max_j \sigma_j$ where $\mathcal{J}$ is the number of jobs in the experiment.

• **Unfairness.** This is the coefficient of variation of the ANPs for all the jobs in an experiment. If a scheduling policy is fair all applications will be affected equally, so lower Unfairness is generally better (though see below) with an ideal value of 0.

• **Data Transfer (DT).** This is the total amount of data transferred by all tasks during an experiment. We split DT into 3 components: data read from local disk, across a rack switch, and across the central switch.

We report a large number of metrics because no single metric exactly captures our informal goal, which is that a job that takes t seconds when given exclusive access to the cluster should take *no more than* Jt seconds when J jobs are running concurrently. Ideally we would like any job that needs less than $1/J$ of the cluster's capacity to continue to execute in t seconds even when the cluster is shared among J jobs—and a job should only see the worst-case slowdown of a factor of J in the case that it would be able to make full use of the entire cluster if given exclusive access.

While a high value of Unfairness may be indicative that a policy is "unfair" in our informal sense, a zero

value for Unfairness may not be what we want, since it is achieved when there is exactly equal slowdown for all jobs. As noted in the preceding paragraph, however, we expect an ideal scheduler to produce ANPs that vary between 1 and $1/J$, and hence have imperfect Unfairness. The same problem occurs with other fairness metrics such as Jain's index [18].

SNP and the slowdown norms are different ways of averaging the performance of all the jobs in an experiment and we include a variety to show that they all exhibit a similar directional trend across experiments.

Makespan is a useful metric to understand the overall throughput of the cluster, and we use it to verify that improvements in Unfairness and SNP do not come at the expense of being able to run fewer jobs per hour on the shared cluster. DT allows us to visualize the tradeoffs the schedulers are making and predict how performance would vary if network resource were provisioned differently.

6. EVALUATION

In this section we run a set of applications under different network conditions and compare their performance when scheduled by the queue-based and flow-based algorithms under various policies. We aim to highlight several major points. First, we show that, for our experimental workload, adopting fairness with preemption gives the best overall performance regardless of the scheduler implementation that is used. Second we compare queue-based and flow-based schedulers with each policy, and determine that Quincy, our flow-based implementation, generally provides better performance and lower network utilization. Finally, we show that when the network is a bottleneck, drastically reducing cross-cluster communication has a substantial positive effect on overall throughput and demonstrate that Quincy, unlike the queue-based approach, is easy to tune for different behavior under different network provisioning setups.

For all the experiments in this paper we set the wait-time factor ω to 0.5 and the cost ψ of transferring 1 GB across a rack switch to 1. For most experiments ξ, the cost of transferring 1 GB across the core switch, is set to 2. However for some experiments with policy QFP we set $\xi = 20$ and these are denoted QFPX.

6.1 Ideal unconstrained network timings

We first analyze the behaviors of different scheduling algorithms when the network is well-provisioned. In these experiments each rack uses its full 6 GBit link to the central switch.

We start by estimating the ideal running time for each instance, since this is used as the baseline for computing ANP and hence SNP, Unfairness and all the slowdown norms. We ran our 30-instance experiment three times with concurrency $K = 1$ under each of three different policies: G (greedy), Q (Quincy with preemption disabled), and QP (Quincy with preemption), and the mean Makespan and DT for each policy are shown in Table 2.

	Makespan (s)	Data Tranfer (TB)		
		Total	Cross rack	Cross cluster
G	5239	2.49	0.352 (14%)	0.461 (18%)
Q	5671	2.49	0.446 (18%)	0.379 (15%)
QP	5087	2.49	0.177 (7%)	0.131 (5%)

Table 2: **Ideal execution metrics.** The table shows the mean across three experiments of Makespan and DT computed under policies G, Q, and QP when $K = 1$ so only a single job is permitted to execute at a time. DT is broken down to show the amount transferred within and across racks.

	Application	(Partitions, Running Time (s))
CPU	PrimeSmall	(2,14), (4,14), (5,14), (6,14), (8,15), (10,15), (15,15), (20,17), (25,17), (500,29), (1000,44), (1500,57), (2000,71)
	PrimeLarge	(240,1360)
	Word	(2,44), (4,45), (5,47), (6,48), (8,49), (10,47), (15,52), (20,56), (25,54), (100,61)
NW	DatabaseJoin	(40,309), (5,365)
	Sort	(10,365), (40,409), (80,562)
	Pagerank	(240,877)

Table 3: **Job classification based on ideal network traffic.** The table lists the mean running time of our application instances measured under ideal conditions in our unconstrained network. Instances are divided into CPU-bound and network-bound sets labeled CPU and NW respectively. Each instance is listed as a pair giving the number of input partitions and its mean running time in seconds.

QP performs best (and also has the lowest standard deviation, at 1.4% of the mean), so we selected the mean of each job's running time across the three QP experiments as the job's ideal baseline. The times are shown in Table 3. Note that many instances run for only a few seconds: when we run concurrent experiments these very short instances have high variance in SNP across experiments because their running time can be strongly affected by startup costs such as transmitting application binaries across a congested network.

Even when jobs are run in isolation, allowing Quincy to perform preemption results in a dramatic improvement in cross-cluster data transfer and hence a small improvement in running time. This can be explained by examining the behavior of the PageRank instance. The second phase of each iteration has no locality preference, so tasks are run on arbitrary computers. As each task in the second phase of iteration 1 completes, the corresponding task in the first phase of iteration 2 is submitted and a greedy scheduler must assign it to the computer that has just become available, which in general does not store that task's inputs. We observe that Quincy frequently moves such poorly located tasks a few seconds

after starting them when more appropriate computers are released. Also note that it *does not* preempt the second-phase tasks that are about to complete, since they have been running much longer and are expensive to dislodge. The hysteresis described in Section 3.5 would not help here since only one job is executing. A more complex form of hysteresis could be designed for a queue-based scheduler in order to address this issue, however we believe Quincy provides a more elegant solution.

6.2 Unconstrained network experiments

We next focus on the performance of the scheduling policies. We set the concurrency to $K = 10$ which is large enough to keep cluster utilization high for most of the duration of the experiment. We average all results across three experiments for each policy. Figure 5 shows the average running time of each job under each policy, compared to the ideal running time for that job, and Figure 6 shows all the metrics we use to summarize performance. The standard deviations for all policies with fairness and preemption are below 1.5% of the mean, however removing either condition can lead to high variance in performance, up to a standard deviation of 19.5% of the mean for QF's Makespan.

Policies that do not allow preemption see worst-case slowdowns of around a factor of 100 for some small jobs and so are clearly unfair in our informal sense. Adding fairness constraints without enabling preemption actually *increases* the Unfairness of both the queue-based and flow-based schedulers. Figure 5 suggests this increase in Unfairness can be attributed mostly to a *speedup* of some small CPU-bound jobs (increasing the variance of the ANP but actually improving the informal fairness of the cluster as discussed in Section 5.2), and indeed the slowdown norms improve slightly when fairness is added.

The three policies that combine fairness with preemption are clearly the most effective across our range of metrics. All have worst-case slowdown l_∞ well under a factor of 10. GFP (the queue-based scheduler with fairness and preemption) has noticeably worse slowdown for the network-bound jobs than QFP or QFPX and this can be explained by the mean data transfer chart in Figure 6, since GFP is the least effective at enforcing data locality and thus transfers the most data through the core network switch. This also explains the slightly worse slowdown (by a few seconds) that GFP sees for very short jobs compared with QFP and QFPX, since job startup time is increased in a congested network as binaries are copied to the computers where the job will execute. As expected, QFPX reduces data transfer across the core switch compared with QFP. In the unconstrained network this does not lead to a substantial increase in performance and in fact the slowdown norms get slightly worse as more tasks are preempted and moved in order to improve locality.

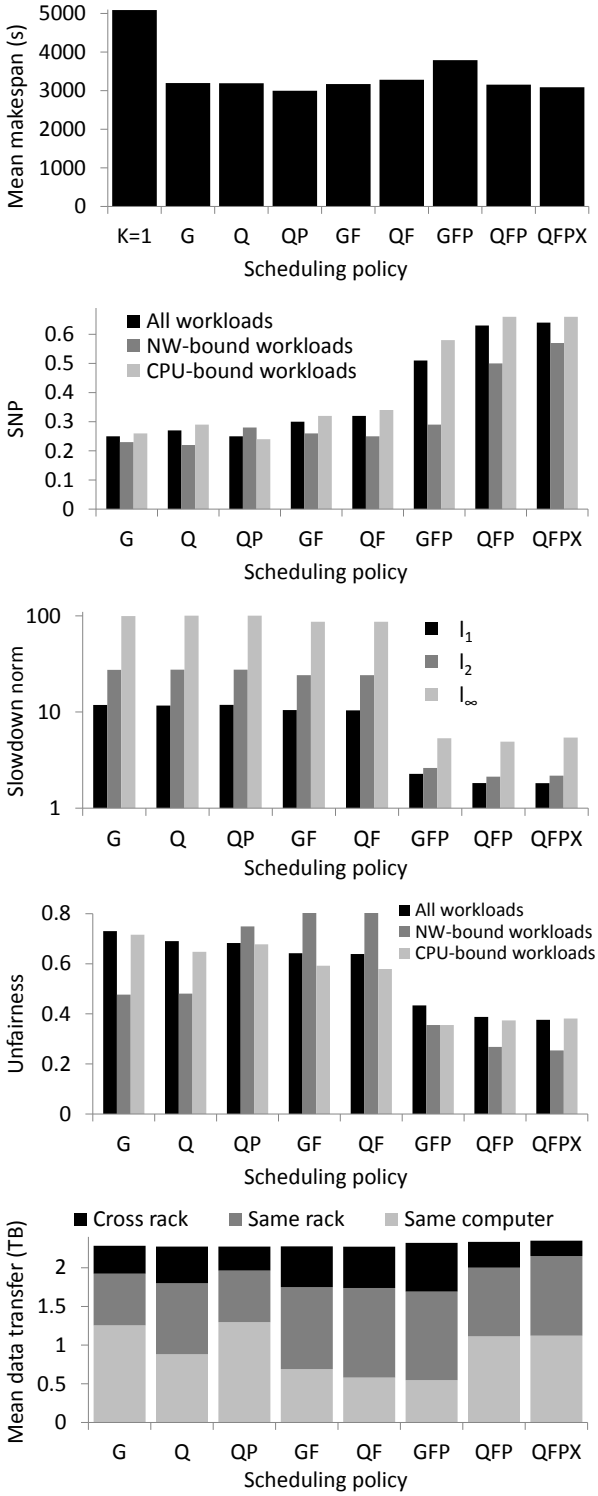

Figure 6: **Performance metrics for experiments using an unconstrained network.**

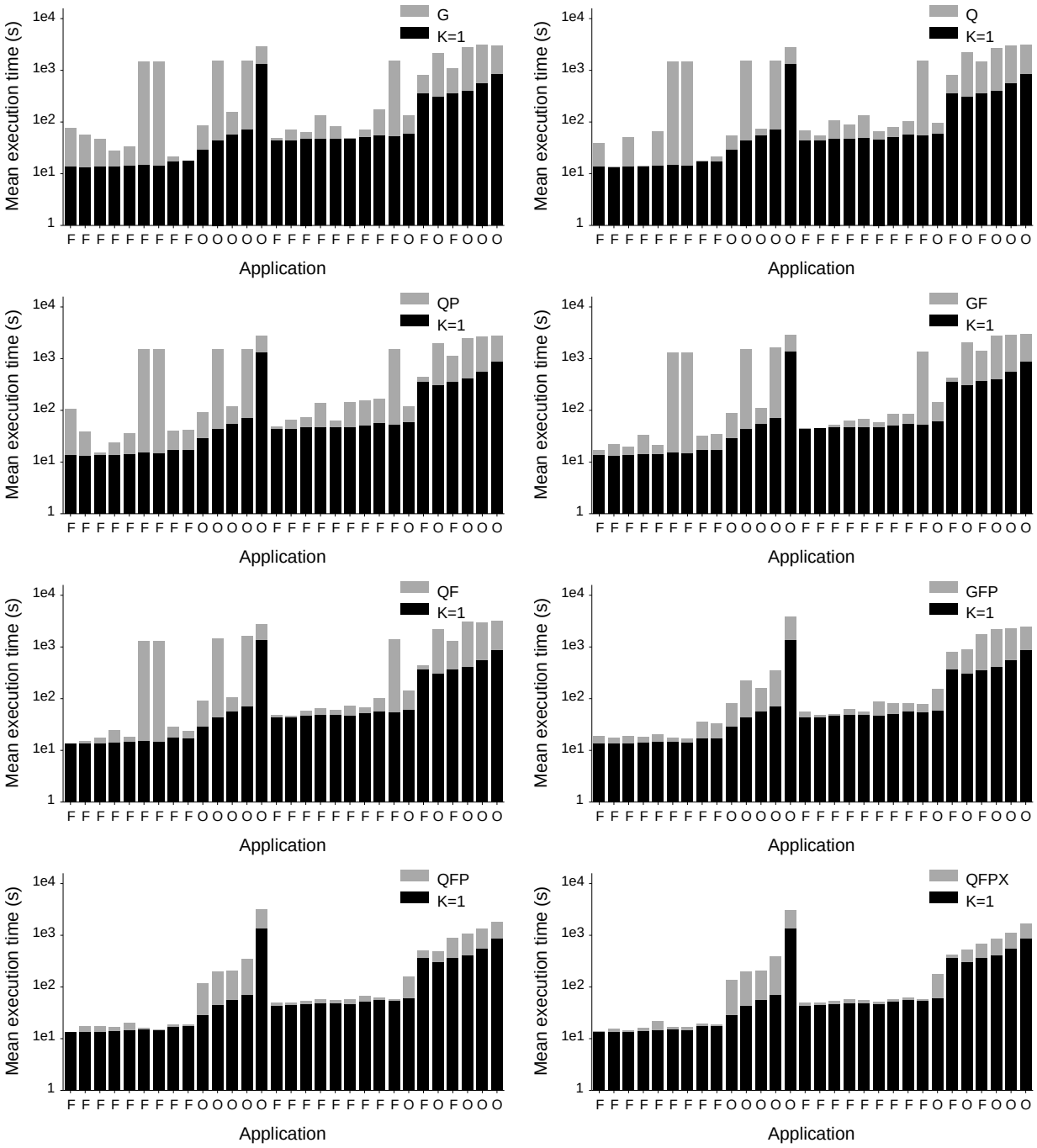

Figure 5: **Average running time of each job for each policy using an unconstrained network with concurrency $K = 10$, compared with the ideal running time for that job with $K = 1$.** Time in seconds is shown on a logarithmic scale on the vertical axis. Jobs are listed in the order given in Table 3. Jobs labeled "F" generally "fit" within their quota of $1/10$th of the cluster's computers and an ideal scheduler should therefore run them with little or no slowdown. Jobs labeled "O" sometimes "overflow" their quota and thus might experience a slowdown from the ideal running time even under perfect scheduling. We classify the rightmost six jobs as network-bound, while the remainder are CPU-bound.

The Makespan chart in Figure 6 shows that overall throughput is remarkably similar across policies when the cluster is equipped with a fast network, and only the very poor placement decisions from GFP substantially hurt throughput. The data transfer chart shows that data transferred across and within racks increases when the fairness constraint is added because tasks might be placed at suboptimal locations to guarantee fair machine allocation. When preemption is enabled the total data transfered increases slightly due to wasted reads by tasks that are subsequently preempted. With preemption, the data transferred across racks by the greedy algorithm increases, while it decreases under Quincy. This is because the flow-based scheduler can preempt and move tasks to better locations, unlike the greedy scheduler. Finally, when the flow-based scheduler is tuned with higher costs for cross rack transfers, data transfers across racks are reduced to about 9% of total data transferred.

Figure 7 illustrates the cluster utilization for two experiments with policies QF and QFP that differ only in whether or not preemption is enabled. The plots show the effect of starting PrimeLarge first, since when preemption is disabled almost no other jobs make progress until PrimeLarge's workers start to complete. This leads to dramatically lower utilization in the latter part of the epxeriment, and hence increased Makespan, for QF in this run.

Our conclusion is that policies that enforce fairness and include preemption perform best for this workload. They can reduce Unfairness almost by a factor of two and show dramatic improvements in SNP and the slowdown norms, while sending substantially less data through the core switch and causing a minimal degradation of Makespan. Quincy performs better for this policy, under all our metrics, than the queue-based approach. This is explained by the fact that Quincy is able to introduce fairness without harming locality, while the queue-based approach greatly increases cross-cluster network traffic as a side-effect of trying to improve fairness.

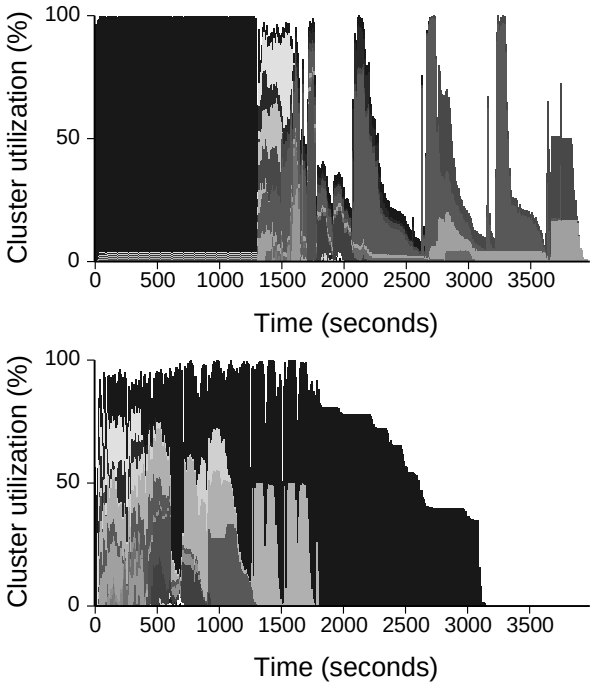

Figure 7: **Cluster utilization without and with preemption.** Shaded regions correspond to application instances, and their area indicates the number of running workers belonging to that instance. The top graph shows QF, with QFP below. The black region is PrimeLarge which occupies most of the cluster at the start and cannot be preempted by QF. QFP shares resources more fairly and completes sooner. Both stages of the final two iterations of PageRank are clearly visible in the lower plot.

6.3 Results under a constrained network

Next, we evaluate our scheduling policies under a constrained network. We reduce the network capacity of the cluster by changing the link bandwidth from each of the 8 racks to the central switch to a 1 GBit duplex link, reducing it to 1/6 of its original capacity.

For these results we remove PrimeLarge from our experiment because it is long-running and CPU-bound, and obscures the Makespan results. Since GFP, QFP, and QFPX give the best performance under the unconstrained network, we compare only these policies for these experiments. We also measure the running time for our instances with concurrency 1 under the constrained network to compute the ideal running time, and we use the results obtained using the QFPX scheduler since it results in the shortest execution times.

Figure 8 shows our summary metrics for the experiments with a constrained network, and Figure 9 shows the average running time of each job in each experiment compared to its ideal running time.

When the network is constrained, the amount of data transferred across the core switch determines the performance of the system as a whole. Table 4 shows that QFP and QFPX transfer less data than GFP both within racks and across the cluster. The power of the flow-based scheduler to optimize the network bandwidth can be seen by the fact that, when told that cross cluster reads are much more expensive than reads within a rack, it reduces the cross cluster transfers to 8% of the total transfer in QFPX trading off with a slight increase within racks and a slight increase in the total due to aggressive preemption.

QFPX performs better than QFP and GFP across all the metrics. The SNP of network-bound instances improves by 32% over QFP and 87% over GFP. While the QFPX settings improve SNP for CPU intensive instances as well, as expected the improvement is moderate when compared with QFP. Along with improving data locality

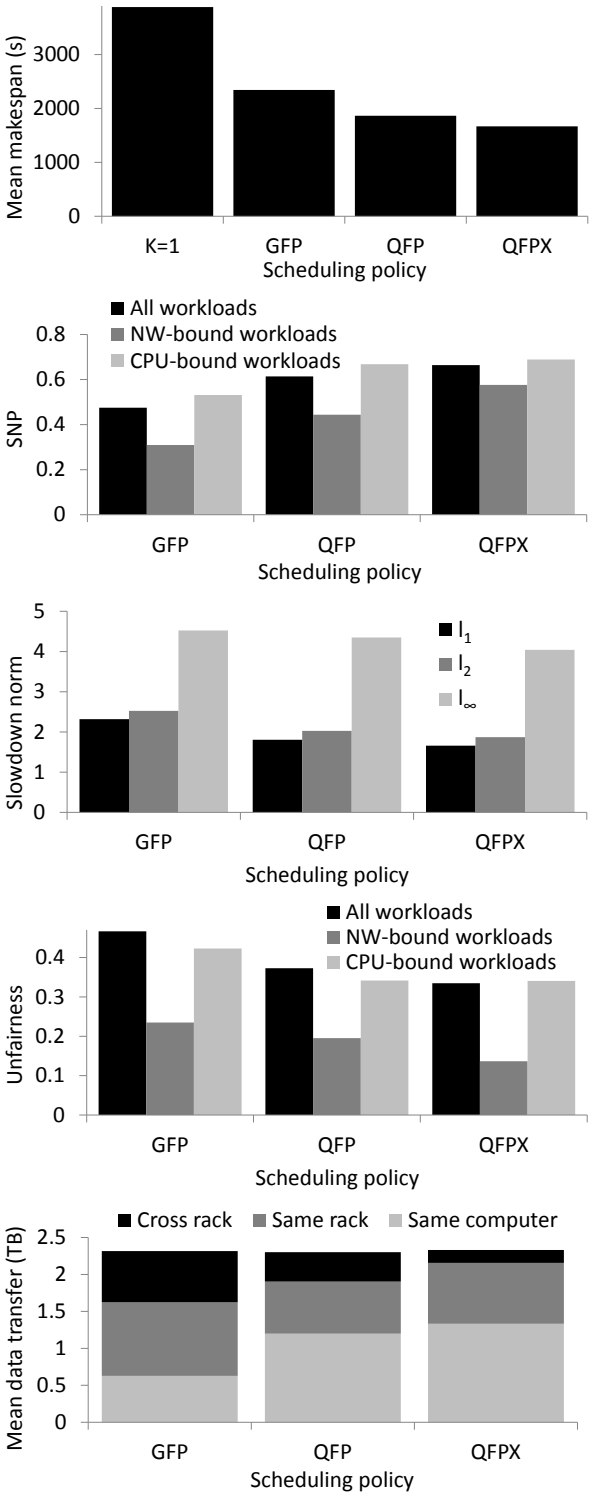

Figure 8: **Performance metrics using a constrained network.**

	DT (TB)		
	Total	Same rack	Cross cluster
ideal	2.47	0.405 (16%)	0.124 (5%)
GFP	2.54	1.095 (43%)	0.760 (30%)
QFP	2.53	0.776 (31%)	0.437 (17%)
QFPX	2.56	0.905 (35%)	0.192 (8%)

Table 4: **DT using a constrained network.** The table presents presents the mean data transferred for different scheduling policies measured across three runs of each experiment using a constrained network.

and performance, QFPX also improves Unfairness compared with the other policies, particularly for network-bound instances, and shows a slight improvement in all the slowdown norms.

6.4 Hysteresis in a queue-based scheduler

We ran experiments using GFH and GFPH, the queue-based policies with hysteresis that are defined in Section 3.5. These policies improve data locality compared to GF or GFP at the expense of Makespan, however they still transferred substantially more data than Quincy.

6.5 Solver overhead

In order to understand the overhead of using min-cost flow, we measured the time Quincy took to compute each network flow solution. Across all experiments, the average cost to solve our scheduling graph was 7.64 ms and the maximum cost was 57.59 ms. This means Quincy is not a bottleneck in our cluster. We simulated graphs that Quincy might have to solve in a much larger cluster of 2500 computers running 100 concurrent jobs, and the computation time increased to a little over a second per solution. It might be acceptable to use this implementation directly, simply waiting a second or two between scheduling assignments, however we believe our implementation can be made to run much faster by computing incremental min-cost solutions as tasks are submitted and complete instead of starting from scratch each time as we do currently.

7. RELATED WORK

The study of scheduling algorithms for parallel computing resources has a long history [20]. The problem of assigning jobs to machines can in general be cast as a job-shop scheduling task [16], however theoretical results for job-shop problems tend to be quite general and not directly applicable to specific implementations.

Scheduling Parallel Tasks. Ousterhout introduced the concept of coscheduling for multiprocessors [24], where cooperating processes are scheduled simultaneously to improve inter-process communication. In contrast to co-scheduling, which runs the same task on all processors at once using an externally-controlled context switch, implicit scheduling [9] can be used to schedule communi-

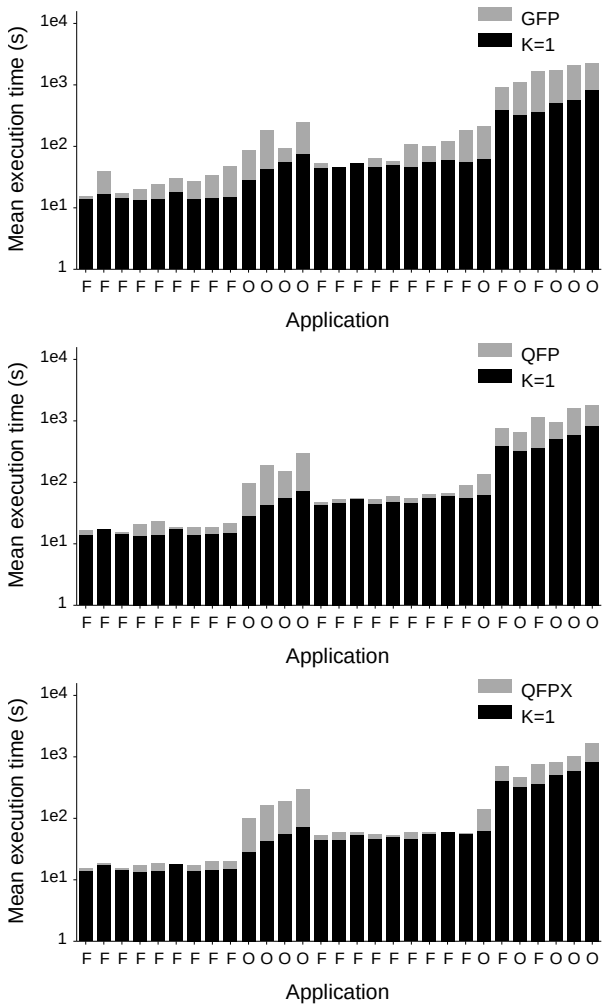

Figure 9: **Average running time of each job for each policy using a constrained network with concurrency $K = 10$, compared to the ideal running time with $K = 1$.** Labels on the horizontal axis are as in Figure 5.

cating processes. Neither technique applies in our setting in which running tasks do not communicate. Naik *et al.* [21] compare three policies for allocating processors to a set of parallel jobs: static, which statically partitions the processors into disjoint sets during system configuration; adaptive, which repartitions the processors when jobs arrive and leave but never changes the allocations during execution; and dynamic, which changes processor allocation as a program is executing. Dynamic partitioning is similar to our approach in that the number of resources allocated to a job can change over time. However Naik *et al.* assume that the data transfer cost across any two processor is the same and this is not true in our cluster setup. While their overriding goal is to improve system performance, we also care about fairness, especially for small jobs. Stone [28] developed a framework based on network flow for partitioning sequential modules across a set of processors, however his graph construction is quite different to ours and does not have efficient solutions when there are more than two processors. There is an excellent survey by Norman and Thanisch [22] of the mapping problem for parallel jobs on multi-processors, however this field of research deals with the offline problem of optimally scheduling a single job and thus does not address fairness.

Scheduling in a distributed computing cluster. Bent *et al.* [6] describe BADFS which addresses many issues relating to storage management and data locality in a cluster setting similar to ours, but does not attempt to achieve fairness. Amiri *et al.* [5] describe a system that dynamically migrates data-intensive computations to optimize a cost function, however the details of the optimization are not given, and there is no mention of fairness. Zaharia *et al.* [33] develop a scheduling algorithm called LATE that attempts to improve the response time of short jobs by executing duplicates of some tasks. Their approach is complementary to our goal of ensuring fair sharing of resources between jobs. Agrawal *et al.* [4] demonstrate that Hadoop's performance can be improved by modifying its scheduling policy to favor shared scans between jobs that read from the same inputs. Their approach relies on correctly anticipating future job arrival rates, which we do not yet incorporate into our scheduling framework, however it may be possible to integrate their approach with Quincy's fair-sharing policies. Condor [29] includes a variety of sophisticated policies for matching resources to jobs [25] including cases where resource requests are not independent [26, 30]. Condor schedulers do not, however, support fine-grain locality-driven resource matching. The closest work to our own in this field is that of Zaharia *et al.* [32] which addresses a similar problem using a queue-based scheduling approach.

Fair-share Scheduling. Proportional share scheduling using priorities [19] is a classical technique but provides only coarse control. Schroeder *et al.* [27] consider the tradeoff between fairness and performance when scheduling parallel jobs, and Gulati *et al.* [15] describe a scheduler called PARDA that implements proportional-share bandwidth allocation for a storage server being accessed by a set of remote clients. However, none of these previous approaches takes account of data locality which is crucial in our setting.

As far as we are aware, ours is the first work to model fair scheduling as a min-cost flow problem.

8. DISCUSSION

Since our scheduling task maps very naturally to min-cost flow, and this mapping leads to extremely effective solutions, it may seem surprising that it has not been previously applied to similar problems in the past. Virtual-machine (VM) placement and migration and many-core

operating system scheduling for example both superficially seem closely related.

We believe there are several reasons for the lack of related work. One is that large shared clusters are a relatively recent development. Batch time-sharing systems did not historically have large numbers of independent resources to allocate. Supercomputers and grid clusters have long been massively parallel, but they are typically scheduled using coarse-grained allocation policies and a fixed set of computers is assigned to a job for a lengthy period. Also, there are restrictions on our cost model that make it easy to express as network flow. In particular:

- We can model the costs of assigning tasks to computers as being independent. For example, we never have a constraint of the form "tasks A and B must execute in the same rack, but I don't care which rack it is." Correlated constraints are very common when a system must schedule processes that communicate with each other while running, and this situation frequently arises in VM placement problems and when executing other distributed computational models such as MPI. It is difficult to see how to encode correlated constraints directly in a network flow model like Quincy's.

- The capacities in our graph construction are naturally one-dimensional. In many scheduling settings, each task may consume less than 100% of a computer's resources. Their resource requirements may be modeled as a multi-dimensional quantity, for example the amount of CPU, disk IO and memory the task demands. Multi-dimensional capacities cannot be easily represented by Quincy's flow network, though as we discuss below we can model fractional capacities.

Nevertheless we are optimistic that there will turn out to be numerous scheduling problems that can be encoded using min-cost flow, and that Quincy's effectiveness will spur further research in the area.

A limitation of Quincy's current design is that fairness is expressed purely in terms of the number of computers allocated to a job, and no explicit attempt is made to share the network or other resources. As we noted in Section 1, Quincy's ability to reduce unnecessary network traffic does improve overall performance predictability and consequently fairness, however when an application does require cross-cluster communication this requirement does not get modeled by our system. We would like to extend Quincy to take account of network congestion in future work: one interesting idea is to monitor network traffic and dynamically adjust Quincy's costs to optimize for reduced data transfer only when it is a bottleneck.

There are several straightforward extensions to Quincy that we are currently exploring:

- We can easily support the allocation of multiple tasks to a single computer by increasing the capacity of the edge from each computer to the sink. It may also be advantageous to temporarily assign more tasks to a computer than its capacity allows, and time-slice the tasks, instead of preempting a task that has been running on the computer for a long time.

- We could get better cluster utilization in some cases by adaptively setting the job concurrency K, for example opportunistically starting "spare" jobs if cluster computers become idle, but reducing their worker task allocation to zero if resource contention returns. In our current design, the choice of K and the algorithm determining appropriate allocations A_j for each job j are completely decoupled from the scheduling algorithm. A future design could unify these procedures, making the admission and fairness policies adapt to the current workload and locality constraints.

- There is a great deal of scope to incorporate predictions about the running time of jobs and tasks, and the known future workflow of jobs, into our cost model. Pursuing this direction may lead us to move all of the jobs' workflow state machines into Quincy and dispense with root tasks altogether.

In conclusion, we present a new approach to scheduling with fairness on shared distributed computing clusters. We constructed a simple mapping from the fair-scheduling problem to min-cost flow, and can thus efficiently compute global matchings that optimize our instantaneous scheduling decisions. We performed an extensive evaluation on a medium-sized cluster, using two network configurations, and confirmed that the global cost-based matching approach greatly outperforms previous methods on a variety of metrics. When compared to greedy algorithms with and without fairness, it can reduce traffic through the core switch of a hierarchical network by a factor of three, while at the same time increasing the throughput of the cluster.

9. ACKNOWLEDGMENTS

We would like to thank Martín Abadi, Mihai Budiu, Rama Kotla, Dahlia Malkhi, Chandu Thekkath, Renato Werneck, Ollie Williams and Fang Yu for many helpful comments and discussions. We would also like to thank the SOSP committee for their detailed reviews, and our shepherd Steve Hand for invaluable advice and suggestions.

APPENDIX

This appendix sets out the detailed costs and capacities used by the Quincy scheduler as outlined in Section 4. Figure 4 shows a typical graph for a small problem setting: a cluster with $L = 2$ racks and $M = 6$ computers on which $J = 2$ jobs are currently executing. (The mapping trivially extends to a network with more than two

levels in its hierarchy.) Rack l contains P_l computers with $\sum_l P_l = M$. Each job j has submitted N_j worker tasks where $\sum_j N_j = N$. Job j has an allocation specified by E_j and F_j where $1 \leq E_j \leq F_j \leq N_j$. E_j is the minimum number of worker tasks that job j must currently be assigned and F_j is the maximum number of running workers that the job is allowed. We discuss in Section 4.3 how these allocation parameters should be set.

As noted in Section 4.2 we use conservative approximations to compute bounds on the data transfer costs when a task is matched to a rack or cluster aggregator node. The following quantities are used to compute these approximations:

• $(\mathcal{R}^X(w_n^j), \mathcal{X}^X(w_n^j))$: these give an upper bound for the number of bytes of input that w_n^j might read across the rack switch and the core switch respectively. They are the values that will result if w_n^j is placed on the worst possible computer in the cluster with respect to its inputs. They are computed exactly by the root task and sent to the scheduler.

• $(\mathcal{R}_l^R(w_n^j), \mathcal{X}_l^R(w_n^j))$: these give an upper bound for the number of bytes of input that w_n^j will read across the rack switch and the core switch respectively if it is executed on a computer in rack l. These numbers are computed exactly and sent to the scheduler for every rack in the task's preferred list as defined in Section 2.3. For the other racks they are approximated using $(\mathcal{X}^X(w_n^j)/P, \mathcal{X}^X(w_n^j))$ where P is the number of racks in the cluster.

• $(\mathcal{R}_m^C(w_n^j), \mathcal{X}_m^C(w_n^j))$: these give the number of bytes of input that w_n^j will read across the rack switch and the core switch respectively if is executed on computer m. If m is in the task's list of preferred computers these numbers are computed exactly and sent to the scheduler, otherwise they are approximated using $(\mathcal{R}_l^R(w_n^j), \mathcal{X}_l^R(w_n^j))$ where l is m's rack.

The nodes in the graph are divided into sets, each represented by a column in Figure 4, and directed edges all point from left to right in the figure. The number of nodes in the graph is $2J + N + L + M + 2$, and the number of edges is $2J + L + 2M + \mathcal{N}$, where $\mathcal{N} = O(N)$. The node sets are as follows:

• **Root task set**: each job j has a node r^j with supply 1 corresponding to its root task. r^j has a single outgoing edge with capacity 1 and cost 0. If the root task has already been assigned a computer m the outgoing edge goes to that computer's node C_m and the scheduler has no choice but to leave the root task running on that computer. Otherwise the outgoing edge is connected to the cluster aggregator node X in which case the task will immediately be set running on some computer chosen according to the global min-cost criterion. There will always be a flow of 1 through r^j.

• **Worker task set**: each submitted worker task from job j has a corresponding node w_n^j with supply 1. w_n^j has multiple outgoing edges each of capacity 1, corresponding to the potential scheduling decisions for this worker:

  • **Unscheduled**: there is an edge to the job-wide unscheduled node U_j with a cost υ_n^j that is the penalty for leaving w_n^j unscheduled. If this edge gets flow 1 then the worker is not scheduled anywhere.

  • **AnyComputer**: there is an edge to the cluster aggregator node X with a cost α_n^j that is the upper bound of w_n^j's data cost on any computer. If this edge gets flow 1 then the worker will be newly scheduled to a computer that could be anywhere in the cluster.

  • **WithinRack**: there is an edge to the rack node R_l with a cost $\rho_{n,l}^j$ for each rack l on w_n^j's preferred rack list, where $\rho_{n,l}^j$ is the upper bound of w_n^j's data cost on any computer in l. If one of these edges gets flow 1 then the worker will be newly scheduled to a computer that could be anywhere in the corresponding rack.

  • **Computer**: there is an edge to the computer node C_m for each computer m that is on w_n^j's preferred computer list with a cost $\gamma_{n,m}^j$ that is the cost of running w_n^j on m. If the worker is currently running on a computer m' that is not in its preferred list there is also an edge to m'. If one of these edges gets flow 1 then the worker will be scheduled to run on the corresponding computer.

There will always be a flow of 1 through w_n^j. The values assigned to costs υ_n^j, α_n^j, $\rho_{n,l}^j$ and $\gamma_{n,m}^j$ are described below.

• **Job-wide unscheduled set**: each job j has an unscheduled aggregator node U_j with supply $F_j - N_j$ and incoming edges from every worker task in j. It has an outgoing edge to the sink node S with capacity $F_j - E_j$ and cost 0. These settings for supply and capacity ensure that the flow f through U_j will always satisfy $N_j - F_j \leq f \leq N_j - E_j$, and this means that the number s of scheduled worker tasks for j will satisfy $E_j \leq s \leq F_j$ since each of the N_j workers w_n^j must send its flow somewhere. Thus U_j enforces the fairness allocation for job j.

• **Cluster aggregator**: there is a single cluster-wide aggregator node X with supply 0. This has an incoming edge of capacity 1 from every worker task that has been submitted by any job, and every root task that has not yet been scheduled. It has an outgoing edge to each rack node R_l with capacity P_l and cost 0. X will get a flow between 0 and M.

• **Rack aggregator set**: each rack l has an aggregator node R_l with supply 0. R_l has incoming edges from the cluster-wide aggregator X and each worker task with

l on its preferred rack list. R_l has an outgoing edge with capacity 1 and cost 0 to the computer node C_m for every computer m that is in its rack. R_l will get a flow between 0 and P_l.

• **Computer set**: each computer m has a node C_m with supply 0. C_m has incoming edges from its rack aggregator R_l and each worker task with m on its preferred list as well as any worker that is currently running on m. C_m has an outgoing edge with capacity 1 to the sink node S. The flow through C_m is 1 if there is a task running on m and 0 otherwise.

• **Sink**: the graph contains a single sink node S with supply $-\sum_j (F_j + 1)$. This supply balances the supply entering the graph at each worker and root task along with the supply $F_j - N_j$ introduced at each job-wide unscheduled node U_j to enforce the minimum flow constraint through U_j. The flow through S is always equal to $\sum_j (N_j + 1)$.

Once the solver has output a min-cost flow solution, the scheduling assignment can be read from the graph by finding, for each computer m, the incoming edge if any that has unit flow. If this edge leads from a task the assignment is clear. If it leads from a rack aggregation node then one of the tasks that sent flow to that node will be assigned arbitrarily to run on m. Since there is a feasible flow, the number of tasks sending flow to the rack aggregation node will be equal to the number of computers receiving flow from it, so all scheduled tasks will be matched to a computer during this process.

In order to compute min-cost flow we first need to determine the costs υ_n^j, α_n^j, ρ_n^j and γ_n^j on the outgoing edges from each w_n^j.

The cost υ_n^j between worker w_n^j and U_j can be used to set priorities for workers to break ties and determine a preference for scheduling one worker over another when their other costs are approximately equal. We have found it useful to set this cost proportional to the time that the task has been waiting. So $\upsilon_n^j = \omega \nu_n^j$ where ω is a wait-time cost factor and ν_n^j is the total number of seconds that w_n^j has spent unscheduled since it was submitted (this may be summed over several timespans if w_n^j was started and then killed one or more times).

We set α_n^j, the cost of scheduling w_n^j on an arbitrary cluster computer, to $\psi \mathcal{R}^X(w_n^j) + \xi \mathcal{X}^X(w_n^j)$ where ψ is the cost to transfer one GB across a rack switch, ξ is the cost to transfer one GB across the core switch, and $\mathcal{R}^X(w_n^j)$ and $\mathcal{X}^X(w_n^j)$ are data sizes defined above. The cost $\rho_{n,l}^j$ of running w_n^j on an arbitrary computer in rack l is set to $\psi \mathcal{R}_l^R(w_n^j) + \xi \mathcal{X}_l^R(w_n^j)$, and the cost $\gamma_{n,m}^j$ of starting w_n^j on a preferred computer m is set to $\psi \mathcal{R}_m^C(w_n^j) + \xi \mathcal{X}_m^C(w_n^j)$.

If w_n^j is already running on computer m' then its cost has two terms: a data term $d^* = \psi \mathcal{R}_{m'}^C(w_n^j) + \xi \mathcal{X}_{m'}^C(w_n^j)$ and a preemption term p^*. p^* is set to θ_n^j, the total number of seconds that w_n^j has spent running on any computer since it was submitted (this may be summed over several timespans if w_n^j was started and then killed one or more times).

The term p^* represents the amount of work already invested in w_n^j. We set $\gamma_{n,m'}^j$, the cost to keep w_n^j running on computer m', to $d^* - p^*$ and thus leaving w_n^j running becomes increasingly appealing the longer it has been executing. We set θ_n^j to the total running time of w_n^j rather than its current running time in order to guarantee starvation-freedom as follows.

Since Quincy may preempt a worker task at any time, even when a job has not exceeded its allocation of computers, we must be sure that a job's tasks will eventually complete so the job can make progress. In order to achieve this guarantee we set a large maximum value Ω for the data costs and wait-time costs of tasks that are not currently running. Once θ_n^j for a task running on computer m gets large enough, $\gamma_{n,m}^j < -\Omega$ and therefore the cost of evicting w_n^j will be larger than the benefit of moving any other task to run on m. It can easily be shown that as long as job j does not keep submitting new worker tasks, and $E_j \geq 1$ so job j always has at least one running task, eventually $\gamma_{n,m}^j < -\Omega$ for all of job j's running tasks and the tasks will run to completion.

We have no theoretical analysis of the stability of the system, however empirically we have not observed any unexpected reallocations of resources. Also, note that the cost of leaving a task running in its current location monotonically decreases over time, and if $\omega = 0$ or there are no tasks currently unscheduled all other costs in the graph are constant. If a min-cost flow solution is modified strictly by decreasing the cost of edges that are already at full capacity, the flow assignment remains a min-cost solution. Thus when there are no unscheduled tasks the system is guaranteed not to change any scheduling assignment until a task completes or a new task is submitted.

REFERENCES

[1] The hadoop fair scheduler. `https://issues.apache.org/jira/browse/HADOOP-3746`.

[2] Open MPI. `http://www.open-mpi.org/`.

[3] Hadoop wiki. http://wiki.apache.org/hadoop/, April 2008.

[4] P. Agrawal, D. Kifer, and C. Olston. Scheduling Shared Scans of Large Data Files. In *Proc. VLDB*, pages 958–969, 2008.

[5] K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamic function placement for data-intensive cluster computing. In *Usenix Annual Technical Conference*, 2000.

[6] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit Control in a Batch-Aware Distributed File System. In *Proc. NSDI*, March 2004.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. OSDI*, pages 137–150, December 2004.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective distributed scheduling of parallel workloads. *SIGMETRICS Performance Evaluation Review*, 24(1):25–36, 1996.

[10] D. Feitelson and L. Rudolph. Gang scheduling performance benefits for finegrained synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–18, 1992.

[11] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, Princeton, NJ, 1962.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, October 2003.

[13] A. V. Goldberg. An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm. *J. Alg.*, 22:1–29, 1997.

[14] A. V. Goldberg and R. E. Tarjan. Finding Minimum-Cost Circulations by Successive Approximation. *Math. Oper. Res.*, 15:430–466, 1990.

[15] A. Gulati, I. Ahmad, and C. A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST'09)*, pages 85–98, February 2009.

[16] D. S. Hochbaum, editor. *Approximation algorithms for NP-hard problems*. PWS Publishing Co., Boston, MA, USA, 1997.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. Eurosys*, pages 59–72, March 2007.

[18] R. Jain, D. Chiu, and W. Hawe. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. Technical Report TR-301, DEC Research Report, September 1984.

[19] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, 1988.

[20] B. W. Lampson. A scheduling philosophy for multi-processing systems. In *Proceedings of the first ACM symposium on Operating System Principles (SOSP'67)*, pages 8.1–8.24, 1967.

[21] V. K. Naik, S. K. Setia, and M. S. Squillante. Performance analysis of job scheduling policies in parallel supercomputing environments. In *Proceedings of Supercomputing*, pages 824–833, November 1993.

[22] M. G. Norman and P. Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Comput. Surv.*, 25(3):263–302, 1993.

[23] J. B. Orlin. A Faster Strongly Polynomial Minimum Cost Flow Algorithm. *J. Oper. Res.*, 41:338–350, 1993.

[24] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the Third International Conference on Distributed Computing Systems (ICDCS'82)*, pages 22–30, January 1982.

[25] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing (HPDC 7)*, July 1998.

[26] R. Raman, M. Livny, and M. Solomon. Policy driven heterogeneous resource co-allocation with gangmatching. In *Proc. High Performance Distributed Computing*, pages 80–89, 2003.

[27] B. Schroeder and M. Harchol-Balter. Evaluation of task assignment policies for supercomputing servers: The case for load unbalancing and fairness. In *Proceedings of High Performance Distributed Computing (HPDC'00)*, pages 211–219, 2000.

[28] H. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, SE-3(1):85–93, 1977.

[29] D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience*, 17(2):323–356, February 2005.

[30] D. Wright. Cheap cycles from the desktop to the dedicated cluster: combining opportunisitc and dedicated scheduling with Condor. In *Conference on Linux Clusters: The HPC Revolution*, 2001.

[31] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proc. OSDI*, pages 1–14, San Diego, CA, December 2008.

[32] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job Scheduling for Multi-User MapReduce Clusters. Technical Report UCB/EECS-2009-55, University of California at Berkeley, April 2009.

[33] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. OSDI*, pages 29–42, San Diego, CA, December 2008.

[34] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor Hardware Counter Statistics As A First-Class System Resource. In *Proceedings of 11th Workshop on Hot Topics in Operating Systems (HOTOS'07)*, 2007.