RIGHT-SIZING RESOURCE ALLOCATIONS FOR SCIENTIFIC APPLICATIONS IN CLUSTERS, GRIDS, AND CLOUDS

A Dissertation

Submitted to the Graduate School

of the University of Notre Dame

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

by

Li Yu

Dr. Douglas Thain, Director

Graduate Program in Computer Science and Engineering

Notre Dame, Indiana

July 2013

RIGHT-SIZING RESOURCE ALLOCATIONS FOR SCIENTIFIC APPLICATIONS IN CLUSTERS, GRIDS, AND CLOUDS

Abstract

by

Li Yu

Clouds have joined clusters and grids as powerful environments for large scale scientific computing. While these platforms provide virtually unlimited computing resources, using more resources for an application does not always result in superior performance. The extra amount that does not contribute to any performance increase is a waste. This dissertation seeks to answer the question of how many computing resources should be allocated for a given workload. Two categories of workloads – static and dynamic, are identified where viable solutions are found for this problem. For static workloads, we show that distributed abstractions allow for accurate performance modeling on distributed, multicore, and distributed multicore systems and thus can automatically make better resource allocation decisions. For dynamic workloads, we present dynamic capacity management as a solution to avoid resource waste without compromising on the application performance. We evaluate the effectiveness of this technique on a selection of workload patterns, ranging from highly homogeneous to completely random, and observe that the system is able to significantly reduce wasted resources with minimal impact on performance. Finally, we show that both solutions have been successfully applied in real world scientific applications in areas such as bioinformatics, economics, and molecular modeling.

CONTENTS

FIGURES

TABLES

CHAPTER 1

INTRODUCTION

Computer systems have become an integral part of the modern economy. More and more research and operations in science, engineering, and business are now being powered by computers 24/7 around the globe. As the complexities of computer-provided services and the difficulty of cutting-edge science questions keep increasing, the demand for greater computing power has never ceased to grow. Continuous advancements in the chip fabrication process have been bringing greater power to individual computers. However, according to the Moore's law, the single chip performance can only be doubled approximately every two years. Even in the early days, such pace of performance increase is not sufficient to accommodate the increase in computer program complexities.

To further improve computer execution performance, various techniques have been created to exploit the parallelism in computer programs. Symmetric multiprocessing[43], which emerged in the 1960s, is a technique that allows two or more processors to share the same main memory and peripherals. The multiple processors can then simultaneously execute computer programs on the same operating system which reduces the total programs execution time. In the 1970s and 1980s, pipeline[82] and superscalar[51] architectures were built into some commercial processors to exploit the instruction-level parallelism. Concurrent executions of multiple computer instructions became possible with these architectures and the thus the overall CPU throughputs were improved. As the chip temperature became intractable when the CPU frequency is pushed beyond 4 GHz, multi-core processors started to dominated the processor market in the late 2000s. In the same period, GPU computing[76] technology emerged and further accelerated certain scientific and engineering applications with thousands of small but efficient cores on the GPU chip. All the above techniques aim to improve the performance of an individual computing node and have been successfully applied in the construction of supercomputers.

The advent of computer networks lead to the idea of connecting multiple computers to execute a single computer program or application. The computational work that needs to be performed in an application is referred to as a workload. To leverage multiple computers, a workload is divided into multiple tasks and these tasks are dispatched to those computers for simultaneous executions. Computer clusters[12], which is a group of computers connected by a local area network, entered the world of parallel computing in the 1970s. They usually rely on centralized management where a head computing node manages the computing nodes and dispatches tasks to them. With the widespread of the Internet, CPU scavenging computing and volunteer computing[8][94] systems started to gain popularity in the 1990s. These systems allow idle or donated computing resources around the world to join the computation of a single problem. In the 2000s, Grid computing[35] emerged with the goal of allowing computing resources from multiple administrative domains to be combined to solve a single computational problem as needed. In the late 2000s, advances in virtualization technologies[73] lead to the birth of cloud computing[11]. According to Mell and Grance [64]: "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction".

While modern distributed systems, such as clusters, clouds, and grids, have been allowing users to orchestrate tasks on thousands of computing nodes and achieve superior performance, it is challenging to use these computing resources properly and efficiently. (Hereafter, we refer to all of these systems as *clusters*.) A user that wishes to execute a large workload with some inherent parallelism is confronted with a dizzying array of choices. How should the workload be broken up into smaller jobs? How should the data be distributed to each computing node? How many nodes or CPU cores should be used? Should the computing nodes be allocated all at once or according to a set schedule? What actions should be taken when some computing resources fail during tasks execution? Will the network present a bottleneck? Last but not least, would leveraging remote computing nodes even result in improved performance given that the transfer of jobs over the network adds overhead? Often, the answers to these questions depend heavily on the properties of the system and workload in use. Changing one parameter, such as the size of a file or the runtime of a job, may require a completely different strategy.

This dissertation focuses on the question of **how many computing resources should be allocated for a given application in a certain computing environment**? Modern clusters provide effectively unlimited computing power at marginal cost with minimal barriers to entry for the end user. With little effort, a single user can submit jobs to a campus cluster, schedule time on national computing infrastructure, or purchase computing time from a commercial cloud provider – or perhaps all three at once! However, the user of an application is presented with a problem – exactly how many resources should be allocated for the application? Selecting too few will result in less than maximum performance. Too many can result in slowdowns, but can also result in contention, crashes, and excess charges as critical resources like the network are overloaded. Further, the ideal selection will change over time as the infrastructure evolves, or the user selects new input data or parameters. In our experience, end users are much too optimistic about the scalability of their applications, and generally err on the side of too many resources.

Finding the optimal resource allocation for a given workload could be difficult.

If the workload's computational structure (the graph formed by its task dependencies) and task properties (input/output data sizes and computational complexity) are unknown, the problem is essentially intractable. This is analogous to the halting problem[17] in the general case. Without knowing the tasks properties, it is impossible to evaluate the performance of any resource allocation. Without knowing the workload structure, it is impossible to estimate the global impact of a resource allocation directly because a suitable resource allocation for the known part of the workload might be inappropriate for the incoming (unknown) part of the workload.

Knowing the structure and tasks properties alone does not necessarily make the problem tractable. For example, for a workload of independent tasks and known task execution times, calculating the optimal number of computing nodes that maximizes the workload execution performance could be NP-hard[65]. In order to compare the superiority of different amount of resource allocations, one must first determine the optimal task scheduling strategy that maximizes the workload performance for a given number of computing nodes. Classic multiprocess scheduling problem[38] and job shop scheduling[39] problem are examples of such scheduling problems have been proven as NP-hard. Because a sub-step in the resource allocation process is already NP-hard, the entire problem is at least NP-hard.

However, if there are regularities in a workload's task properties and/or structure, the complexity of the problem could be reduced to a more tractable level. For example, in the multiprocess scheduling problem, if we know that all the tasks are of the same size (task regularity), the optimal scheduling strategy becomes obvious, which is to dispatch tasks among the processors evenly. Also in this example, the fact that all tasks are independent can be considered as a workload structure regularity. In general, the regularities in a workload allow workload population characteristics to be estimated with sample characteristics, which could greatly reduce the solution space, and thus simplifies the problem. The more regularity is known in advance

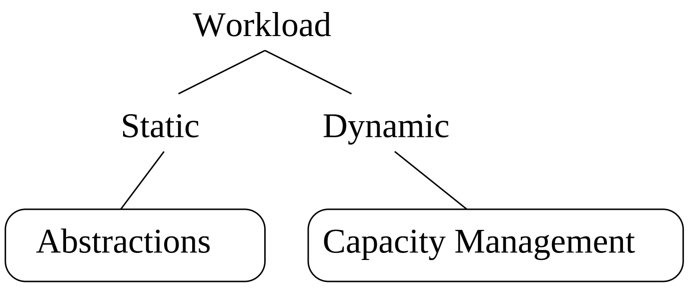


Figure 1.1: Two Categories of Workloads
For static workloads, if the computational structures and tasks properties are regular, abstractions can be used to help make resource allocation decisions. For dynamic workloads, the capacity management method can effectively avoid resource waste without compromising on the application performance.

about the workload, the more tractable the resource allocation problem is.

In this dissertation, we identify two categories of workloads with different degrees of regularities and present the resource allocation strategies for each of them respectively. We define a workload as a *static workload* when its structure is set and known prior to execution. If the structure can not be determined in advance, the workload is referred to as a *dynamic workload*. For example, when tasks in a workload are submitted to a batch system when ready, the batch system would consider the workload as dynamic because it never knows what tasks may be submitted next after as ones currently in the queue are finishing. If the resource allocator is not the batch system, but a program that has the specification of the workload's entire structure, then the workload is static to the resource allocator. For static workloads whose computational structures and task properties are regular, we show that abstractions can help make better resource allocation decisions. For dynamic workloads, we present dynamic capacity management as a solution to avoid resource waste without compromising on the application performance. The workload categorization is shown in Figure 1.1.

For regular static workloads, abstractions can effectively model their performances for a given set of resources and thus are able to assist making resource allocation decisions. An *abstraction* is a declarative structure that joins simple data structures and small sequential programs into parallel graphs that can be scaled to very large sizes. Each abstraction is used to execute workloads that follow the same execution pattern. Because static workloads have known structures, they could be executed with abstractions their structure is recognized by existing abstractions. We argue that abstractions are an effective way of enabling non-expert users to harness clusters, multicore computers, and clusters of multicore computers. Because an abstraction is specialized to a restricted class of workloads, it is possible to create an efficient, robust, scalable, and fault tolerant implementation.

In this dissertation, we focus on three specific abstractions: All-Pairs, Wavefront, and Makeflow abstractions. The All-Pairs and Wavefront abstractions are intended for two type of regularly structured workloads – each abstraction assumes a specific execution pattern. The regularity in the workload structure is a prior knowledge in both abstractions. As we will show later in the dissertation, this workload structure regularity, combined with the task regularity, allows for performance modeling on multicore systems, and clusters of multicore systems. The All-Pairs performance modeling has been previously studied in distributed environments with single-core computing nodes. I extended the research in environments with multi-core computing nodes. And the modeling result is sufficiently accurate to assist making useful resource allocation decisions such as whether to run the workload with 8 local CPU cores or with 4 remote dual-core computing nodes. The Makeflow abstraction is designed for arbitrary DAG (Directed Acyclic Graph) structured workloads. It is useful for workloads whose execution pattern is not capture by any existing abstractions (which are usually designed for regularly structured workloads). Although the workloads that can be run with the All-Pairs or the Wavefront abstractions can be also described by the Makeflow abstraction, the performance with the former could be much greater as the known regularity in the workload structure allows for optimized task scheduling.

All-Pairs and Wavefront abstractions have both been used to accelerate real world applications. For example, in biometrics, the evaluation of identification algorithm fits the All-Pairs pattern. An identification algorithm compares two human characteristics (e.g. iris image) and outputs their similarity. The evaluation application takes two known sets of characteristics and compares every characteristic in one set with every element in the other using the target algorithm. The resulting similarity matrix will then be used to determine the effectiveness of the identification algorithm. The evaluation of classifiers in data mining also follows the similar workload. Another example is the first step in genome assembly – comparing each measured gene sequence to every other sequences (sequenced from the same DNA). Wavefront represents a number of simulation problems in economics and game theory, where the initial states represent ending states of a game, and the recurrence is used to work backwards in order to discover the effect of decisions at each state. Wavefront also represents the problem of sequence alignment via dynamic programming in genomics.

For dynamic workloads, it is impossible to make global performance modeling in advance because the workload structure is unknown. In a research context, one might run the exact same workload at multiple scales in order to generate a parallel speedup curve, and then choose the best value. However, in a production computing context, the end user gains no value from running the same workload more than once. Instead, an *acceptable* decision must be made on the first attempt. For static workloads like All-Pairs and Wavefront abstractions, this can be achieved by working with declared knowledge of the entire workload. For dynamic workloads, information must be gained incrementally. We argue the problem should be addressed through two combined techniques. First, a workload must have some degree of introspection into its own performance to understand and report critical properties such as parallel efficiency and network capacity. Second, an external resource allocator should use this information to allocate and manage the resources consumed at runtime. The resource allocator should be external to the workloads to isolate computer program bugs (in the workloads) from the credit card bill, particularly if the end user is not the bill-payer.

The solution for dynamic workloads is referred to as *capacity management.* It relies on three basic assumptions: (a) workloads are composed of tasks that have a relatively stable computation-to-data ratio (this is task regularity), (b) task properties are not known in advance, and (c) there are more tasks than workers available, so that there is an opportunity to measure and adjust over time. The system will adapt to changes in computation-to-data ratio, as long as the task mix is stable for a sufficient period for the system to adjust. For example, 1000 tasks of type A followed by 1000 tasks of type B can be handled, with some period of adjustment as the B tasks begin. If the tasks have no commonality and nothing is known of their properties in advance, then the resource allocation problem is essentially intractable. However, we will show that even with random workloads, our solution does an adequate job of preventing resource waste.

There are many real world applications that can satisfy the above assumptions required by the capacity management method. As an example, we work with a bioinformatics group at Notre Dame that runs production workloads consisting of thousands of tasks through this system. For example, a BLAST[7] workload contains tasks that compare query DNA sequences with reference DNA sequences. For a single BLAST workload, the reference sequences are the same across all the tasks. The query sequences are different in each task but are of similar sizes (we split a large query into smaller equal-size pieces). Although there could be variances in the BLAST job execution times even if the query sequence sizes are the same, for the workloads our users run, such variances exist but only on the few tasks that contain the actual matches to the database, so that over 90% of the tasks have similar execution times. Similar application properties apply to the workloads that use other gene sequence comparison applications, such as BWA[55], SSAHA[74], and SHRIMP[90].

We have developed a range of resource allocation policies in the capacity management with increasing refinements, taking into account the computation and network loads of the master process and the tasks in the workload. To evaluate these policies, we test them against a range of five synthetic applications, ranging from a single burst of identical tasks to continuous random bursts of random tasks. We demonstrate that these techniques significantly reduce wasted resources with minimal impact upon performance. Additionally, although these techniques assume some degree of homogeneity in the individual tasks, they are still reasonably effective on random workloads. And this is why we did not distinguish irregular and regular workloads under the dynamic workload category in Figure 1.1.

We have implemented the concepts of abstractions and capacity management in CCTools – a distributed computing tools software package developed by the Cooperative Computing Lab at Notre Dame. The All-Pairs, Wavefront, and Makeflow are standalone applications. The dynamic capacity management is implemented within the context of the Work Queue application framework, but the concepts are easily applied to other similar distributed computing frameworks.

CHAPTER 2

RELATED WORK

2.1 Batch Systems

A batch system manages the execution of computer jobs on a set of computing resources. It manages the resource allocation at the task level, which is different from the workload level that this dissertation focuses on. However, the basic ideas in resource management is applicable in both levels and the two levels of resource management can complement each other. A job that gets submitted to a batch system usually consists of a computer program and some input data that needs to be processed by the program. Multiple users can submit multiple jobs to the system simultaneously. A batch system usually puts submitted jobs into several queues such as queues for serial or parallel jobs, and queues for long or short jobs. It constantly monitors the status of its managed resources and dispatches queued jobs to them when desired resources (the ones that match the job's requirements on CPU, memory, disk space, software licenses, and etc.) become available. Because batch systems have centralized control over the shared computing resources, it is possible to optimize the job scheduling to improve the overall resource utilization and system throughput. Also, the jobs can be treated accordingly with respect to their priorities. The following is an overview of some popular batch systems.

SGE [41] (Sun Grid Engine), LSF [108] (Load Sharing Facility), and PBS [46] (Portable Batch System) are popular established batch systems maintained by different companies. They are all used to manage job executions on dedicated computing resources. A typical computer cluster that deploys such batch system would consist of a master host and multiple execution hosts (the terminologies may differ in different systems). The batch system software provides a standard interface for users to submit, delete, and monitor their jobs from the master host. The jobs are immediately queued when submitted to the master host and will be dispatched to execution hosts for actual execution when appropriate. In general, the job scheduling is based on the following criteria: the cluster's current load, the job's importance, the execution host's performance, and job's resource requirements. These systems support a range of customizable job scheduling policies, such as FCFS (First Come First Serve), fairshare scheduling[33], backfilling[69], deadline scheduling[23], exclusive scheduling, preemptive scheduling[106], and SLA (Service Level Agreement) driven scheduling[59]. These policies allow the allocated resources to effectively match the needs of the job submitters.

Condor [99] is an open-source high-throughput batch system for compute-intensive jobs developed at the University of Wisconsin-Madison. Although Condor has been renamed to HTCondor in 2012, we will refer to it as Condor in the rest of the dissertation. Like with other batch system, Condor provides job queueing, scheduling, monitoring, and resource management functionalities. But in addition to managing clusters of dedicated computing resources, Condor is capable of integrating the power of heterogeneous, non-dedicate workstations (a concept referred to as cycle scavenging). Jobs can be farmed out to desktop workstations when they are idle. Condor uses a ClassAd[83] mechanism to match jobs with qualified resources (e.g.: Operating System = Linux, Memory>1 GB). Condor job queues implement priority queueing. The amount of resources that a job can get is based on the user's dynamic priority. The fairshare algorithm ensures that each user gets the same amount of resources over a specified period of time and lower-priority user's jobs will not be starved. Condor is also one of the supported scheduler in GRAM[25] (Grid Resource Allocation Manager, a component of the Globus Toolkit) and has been the resource management backend for many Grid applications.

2.2 Workflow Management Systems

A workflow is a collection of jobs with dependencies among them. A job in a workflow can be dispatched for execution only after all the jobs that it is dependent on have been completed successfully. To execute a workflow on a cluster managed by an batch system, the jobs must be submitted to the cluster incrementally as the job dependencies are resolved. Of course, the users do not want to, and most of the times can not afford to, monitor the job status constantly and manually submit more jobs when the submitted ones are finished. Thus, workflow systems are created to manage the automatic executions of whole workflows. The workflow systems usually accept a description of the entire workflow, including job descriptions and dependencies, and submit individual jobs to the batch systems when prerequisite jobs are done. The following is an overview of some popular production workflow management systems.

DAGMan [98] is a scheduler for DAG (Directed Acyclic Graph) structured workflows built for Condor. A DAG structured workflow would form a DAG if we draw each job as a vertex and each dependency (between two jobs) as an edge connecting the two corresponding vertices. Because Condor does not schedule jobs based on dependencies (jobs are considered independent to each other once submitted to Condor), DAGMan is a necessary layer of software for managing the automatic execution of DAG workflows. If a job fails, DAGMan can be configured to retry failed jobs for a certain mount of times without interrupting the execution of the rest of the workflow. If an entire workflow fails, only unfinished jobs will be dispatched upon recovery. DAGMan also supports advanced features such as allowing jobs within a workflow to have different priorities and limiting the total amount of resources can be acquired by the workflow at any given time.

Pegasus [29] is a workflow management system for mapping and executing workflows on various computing environments including Condor, Grid infrastructures such as Open Science Grid and TeraGrid, and cloud computing platforms such as Amazon EC2[1] and Nimbus[3], and many campus clusters. The same workflow can run in any of these systems or leverage resource across multiple platforms. Given an abstract, high-level description of the workflow, Pegasus is able to automatically locate the necessary software, required data, and qualified computing resources for execution. The tasks in a workflow may be automatically reordered, grouped, or re-prioritized to optimize the performace of the entire workflow execution. The processed, or restructured workflow description will be passed to DAGMan, its execution engine, for actual execution. Pegasus has been accelerating many real world applications in different domains such as bioinformatics, chemistry, neuroscience, and climate simulation.

Taverna [48] is an open-source suite of tools to build and execute scientific workflows at a higher level of workflow abstraction. Instead of asking users to construct workflows from files and computer programs, Taverna exposes data and operations to the users in an integrated workflow design environment. The operation in a Taverna workflow can be any WSDL-style web service, which allows workflows to be constructed from commonly accessible web services. Users can search for services from service catalogs (e.g. BioCatalogue) to include in their workflows without knowing how to invoke them. Once data flows have been defined across identified services, the workflow is ready to be executed. Taverna workflows can be executed on grids and clouds infrastructures as well. EGEE[54], caGrid[91], KnowARC[70], and NGS[40] are examples of running Taverna workflows on grids. Next Generation Sequencing, SCAPE, and e-Science Central are examples for the clouds.

Kepler [6] is a workflow management system that faciliates the creating, executing, and sharing of scientific workflows across multiple disciplines. Like with Taverna, Kepler provides direct access to commonly used data archives and allows constructing workflows from web services. Unlike many other workflow systems, Kepler allows common models of computation to be applied to the construction of a workflow. Examples of such common models are Synchronous Data Flow, Continuous Time, Process Network, and Dynamic Data Flow. Thus, complex workflows can be build with simpler components. Kepler workflows can be exported and shared via web services and thus users can conveniently search and integrate others' analysis workflows into their own workflows. In addition to invoking web services directly, Kepler supports job execution on grids and grid-based data access.

Galaxy [6] is a scientific workflow management system that allows computational biology scientists to leverage distributed computing power without prior computer programming experience. It is originally developed for genomics application, but now supports a wide range of bioinformatics applications. Galaxy provides a web-based graphic user interface for domain scientists to construct analysis workflows and manage scientific data. The input data and computational tasks can be selected from dynamically generated graphic menus and the results are displayed in intuitive plots. Galaxy's high level of accessibility greatly benefits the end users who are not trained for computer programming. And more importantly, as a full-fledged workflow management system, it records all the tools, parameters, and data that have been used in the workflows and thus ensures that any result obtained from the system can be reproduced and reviewed later. Galaxy's is now a leading platform in computational analysis of DNA sequence data.

As summarized in [105], some of these systems use static scheduling [50] and some of them use dynamic scheduling [79] or both. The scheduling strategies used in these systems are all performance driven. Our work does not seek a better scheduling scheme to improve performance. Instead, we focus on identification of potential resource waste and make the system be able to avoid such waste automatically at runtime.

2.3 Distributed Computing Abstractions

A distributed computing abstraction is a common and abstract computational model extracted from a range of applications. As a software tool, it allows users to construct parallel applications that have the same computational model with minimal efforts. Usually the users only need to supply the input data and the serial programs. The users do not even need to master the techniques for using distributed computing resources. The distributed computing abstractions would plug the user inputs into the computational models and conduct the parallel computations in a suitable and efficient way in the actual distributed computing environment.

Bag-of-Tasks[24] is a simple but powerful abstraction of many applications across a broad range of disciplines. It represents the computational model of independent tasks. Many real world applications fall into this model, such as parameter sweeps, massive search, image processing, and computational biology. Because the model is so general, many parallel/distributed computing research are based on this model. The aforementioned classic multiprocess scheduling and job shop scheduling are based on this model. Many scheduling algorithms[18], such as FCFS (First Come First Serve), Min-Min, Max-Min, and Sufferage, have been studied for BoT applications based on how much prior knowledge is known about the tasks and the goal of the computation, such as to minimize cost or to maximize performance. Because there are no inter-task communications, BoT applications are especially suitable for Grid computing platforms (Condor, BOINC[8], and OurGrid[10]) where network bandwidth is precious. Projects such as SETI@Home[9] and Folding@Home[53] have successfully demonstrated that BoT applications can achieve great scalability over widely distributed grids.

The All-Pairs abstraction [66] computes the Cartesian product of two sets, generating a matrix where each cell M[i,j] contains the output of the function F on objects A[i] and B[j]. This computational model is found in many different fields. In bioinformatics, one might compute All-Pairs on a set of gene sequences as the first step of building a phylogenetic tree. In biometrics, one might compute All-Pairs to determine the accuracy of a matching algorithm on a collection of faces. In data mining applications, one might compute All-Pairs on a set of documents to generate a graph of relationships. The All-Pairs abstraction allows large-scale pair-wise comparisons to be created by specifying only a few command-line arguments. And more importantly, the abstraction automatically optimizes the parallel execution of the tasks on the target computing environment and tends to be more efficient than a naive implementation of the same computational pattern.

Map-Reduce [28] is a distributed computing abstraction popularized by Google for using it to index the enormous amount of contents on the Internet. Map-Reduce allows users to invoke large-scale parallel processing by defining two simple functions – the mapper and the reducer. The mapper defines how data should be processed, or more specifically, be transformed to intermediate name-value pairs. The reducer defines how the intermediate name-value pairs should be combined into the final result. When executing a Map-Reduce application, the original input data is usually divided into pieces and the pieces are spread (and often replicated) across multiple computing nodes. The mapper and reducer functions are then moved to the computing nodes, which have the data pieces, for actual execution. Because the computation is moved to where the data is, Map-Reduce is well-suited for data-intensive applications. Popular implementations of Map-Reduce include Hadoop [16], Sphere [44], and Twister [32].

2.4 Auto-Scaling Techniques

Batch systems such as Condor, SGE, and PBS Pro provide interfaces for multiple users to request abstract, dynamic, and managed computing resources. These systems handle scheduling, load-balancing, and fault-tolerance at the task level. Our

Work Queue application framework, as an implementation of the Pilot-Job concept, adds the possibility of workload-level scheduling and data caching. Pilot-Job is essentially an abstraction that allows computing resources to be acquired by a workload such that the workload's tasks can be scheduled to the resources directly instead of going through a job scheduler (e.g. waiting in a batch system's job queue). Condor-Glidein [36] is one of the ealiest implementations of the Pilot-Job idea. A glidein is a computing node that can automatically join a specified Condor pool and become dedicated resource to that pool. The worker in the Work Queue framework is analogous to a glidein, which is the pilot job.

There are many successful Pilot-Job frameworks with different specialties. SAGA BigJob [58] interfaces with various computing infrastructures (Grids and Clouds) and natively supports MPI[95] applications. In order for an application to utilize this framework, the application needs to specify the number of resources (i.e. pilot jobs) needed prior to submitting individual tasks. Falkon [80] is optimized for efficiently dispatching many small tasks. Its resource provisioner dynamically matches the number of executors (the pilot job) to the number of queue tasks without surpassing a user-defined upper-bound. Coaster [45] uses a centralized process – called Coaster Service, to queue user jobs and submit pilot jobs based on the characteristics of the queued user jobs. The Coaster Service determines the amount of pilot jobs by matching the total acquired resource time to the total job time, which necessitates prior knowledge of job execution times. GlideinWMS [93] is a Glidein based workload management system. Its front-end polls the user's local Condor pool to see if the number of glideins (worker nodes) is greater than the number of user jobs. If not, it submits requests to the glidein factory to create more glideins to join the user's local Condor pool. DIRAC [103] and PanDA [60] are another two examples of Pilot-Job based workload management systems which submit additional pilot jobs to match each newly queued job. Most of these systems equate the number of user jobs with the number of pilot jobs needed. However, this might lead to significant resource waste due to limited network bandwidth and job characteristics as will be shown in section 4.2. The capacity management technique described in this dissertation can be used to reduce such resource wastes. It is implemented in our Work Queue framework, but the idea can be applied to any of the aforementioned Pilot-Job frameworks.

As clouds provides the illusion of unlimited costly resources, techniques that dynamically scale resources according to application needs have gain great popularity in both industry and academia. AWS[2][15], RightScale[4], and Elastack[14] allow users to scale resources based on system metrics such as CPU utilization and time schedule. Elastic VM[27] scales resources vertically (single-CPU instance to multi-CPU instance) based on CPU utilization history and has been shown to reduce response time in both the web-tier and the database-tier. Our approach scales resources horizontally based on the application resource needs, in contrast to hardware metrics. Another way of approaching the dynamic resource scaling problem is to first predict the workload and then use certain function to determine the appropriate resource amount for the incoming workload. Caron et al.[21] introduces a string matching based algorithm to predict future workload based on where the recent workload pattern stands in the historical workload pattern. Instead of trying to obtain more accurate estimate of future workload, Lin et al.[56] describes how trend analysis – predicting the direction of workload change can help making better auto-scaling decisions. Roy et al.[89] uses a second order autoregressive moving average method to predict the incoming workload and minimizes the scaling cost based on how far the estimated response time is from the SLA bounds, cost of leasing additional resources, and the cost of re-configuration. Qiu et al.[78] uses a non-linear autoregressive neural network method to predict future workload and uses a function that minimizes resource provision without violating SLA (e.g. percentage of unsatisfied user requests) to determine the amount of resources.

Nephele [104] uses a profiling subsystem to monitor the time resources spent on user code and waiting for data. The feedback data is not currently used to dynamically adjust the amount of resources but it is in their future work. Marshall et al.[63] proposes a elastic site architecture that extends local cluster capacity with cloud resources and examines three dynamic resource allocation policies. Similar to the capacity estimation assumption, Nagavaram et al.[71] assumes the workloads include tasks of equal sizes and predicts the number of cores needed based on the execution times of the first N tasks (N is equal to the number of cores in the cluster). Mao and Humphrey[62] introduces methods to scale cloud resources based on deadline and budget constraints. And their recent work in [61] presents methods to minimize cloud computing resource cost given user assigned soft deadlines on jobs (each job can have multiple tasks). Their solution takes advantage of the different cost-efficiencies in on-demand VM instances and uses heuristics to obtain an optimized task-resource mapping. Our work differs in that we use dynamic configuration where we rely on the continuously changing properties reported by the application, instead of requiring information in advance.

CHAPTER 3

STATIC WORKLOADS

A static workload is a workload whose computational structure is known prior to execution. A user that wishes to execute a static workload in a distributed environment is confronted with a dizzying array of choices. How should the workload be broken up into jobs? How should the data be distributed to each node? How many nodes should be used? Will the network be a bottleneck? Often, the answers to these questions depend heavily on the properties of the system and workload in use. Changing one parameter, such as the size of a file or the runtime of a job, may require a completely different strategy.

Multicore systems present many of the same challenges. The orders of magnitude change, but the questions are similar. How should work be divided among threads? Should we use message passing or shared memory? How many CPUs should be used? Will memory access present a bottleneck? When we consider clusters of multicore computers, then the problems become more complex.

We argue that *abstractions* are an effective way of enabling non-expert users to harness clusters, multicore computers, and clusters of multicore computers. An abstraction is a declarative structure that joins simple data structures and small sequential programs into parallel graphs that can be scaled to very large sizes. Because an abstraction is specialized to a restricted class of workloads, it is possible to create an efficient, robust, scalable, and fault tolerant implementation.

In previous work, we introduced the All-Pairs [66] and Classify [67] abstractions, and described how they can be used to solve data intensive problems in the fields

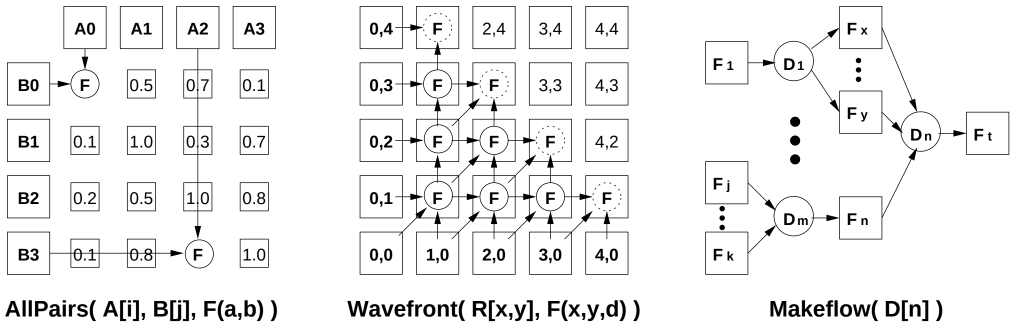


Figure 3.1: Three Examples of Abstractions
All-Pairs, Wavefront and Makeflow are examples of abstractions. All-Pairs computes the Cartesian product of two sets A and B using a custom function F. Wavefront computes a two-dimensional recurrence relation using boundary conditions and a custom function F as an input. Makeflow takes an array of dependencies, which could be visualized as a directed acyclic graph structured workload, computes according to the workflow and produces a target file. Using different techniques, each can be executed efficiently on multicore clusters.

of biometrics, bioinformatics, and data mining. Our implementations allow non-experts to harness hundreds of processors on problems that run for hours or days using the Condor [100] distributed batch system. In this chapter, we extend the concept of abstractions to multicore computers and clusters of multicore computers. We demonstrate that it is feasible to accurately model the performance of large scale abstractions across a wide range of configurations, allowing for the rational selection of appropriate resources.

3.1 Abstractions

An *abstraction* is a declarative framework that joins together sequential processes and data structures into a regularly structured parallel graph. An abstraction *engine* is a particular implementation that materializes that abstraction on a system, whether it be a sequential computer, a multicore computer, or a distributed system. Figure 3.1 shows three examples of abstractions: All-Pairs, Wavefront and Makeflow.

All-Pairs(A[i], B[j], F(x,y))

returns matrix M

where M[i,j] = F(A[i],B[j])

The All-Pairs abstraction computes the Cartesian product of two sets, generating a matrix where each cell M[i,j] contains the output of the function F on objects A[i] and B[j]. This sort of problem is found in many different fields. In bioinformatics, one might compute All-Pairs on a set of gene sequences as the first step of building a phylogenetic tree. In biometrics, one might compute All-Pairs to determine the accuracy of a matching algorithm on a collection of faces. In data mining applications, one might compute All-Pairs on a set of documents to generate a graph of relationships.

Wavefront(R[i,j], F(x,y,d))

returns matrix R

where R[i,j] = F(R[i-1,j], R[i,j-1], R[i-1,j-1])

The Wavefront abstraction computes a recurrence relationship in two dimensions. Each cell in the output matrix is generated by a function F where the arguments are the values in the cells immediately to the left, below, and diagonally left and below. Once a value has been computed at position (1,1), then values at positions (2,1) and (1,2) may be computed, and so forth, until the entire matrix is complete. The problem can be generalized to an arbitrary number of dimensions. Wavefront represents a number of simulation problems in economics and game theory, where the initial states represent ending states of a game, and the recurrence is used to work backwards in order to discover the effect of decisions at each state. Wavefront also represents the problem of sequence alignment via dynamic programming in genomics.

Makeflow(R[n])

where each rule R[i] is (input files, output files, command)

returns output files from all R[i]

The Makeflow abstraction expresses any arbitrary directed acyclic graph (DAG). Whereas All-Pairs and Wavefront are problems that can be decomposed into thousands or millions of instances of the same function to be run with near-identical requirements, a DAG workload may be structurally heterogeneous and consist of programs and files of highly variable runtime and size. Many such problems are found in bioinformatics, where users chain together multiple independent tools to solve a larger problem. Below, we will show Makeflow applied to a genomics problem.

On very small problems, these abstractions are easy to implement. For example, a small All-Pairs can be achieved by just iterating over the output matrix. However, many users have *very large* examples of these problems, which are not easy to implement. For example, a common All-Pairs problem in biometrics compares 4000 images of 1MB to each other using a function that runs for one second, requiring 185 CPU-days of sequential computation. A sample Wavefront problem in economics requires evaluating a 500 by 500 matrix, where each function requires 7 seconds of computation, requiring 22 CPU-days of sequential computation. To solve these problems in reasonable time, we must harness hundreds of CPUs. However, scaling up to hundreds of CPUs forces us to confront these challenges:

- **Data Bottlenecks.** Often, I/O patterns that can be overlooked on one processor may be disastrous in a scalable system. One process loading one gigabyte from a local disk will be measured in seconds. But, hundreds of processes loading a gigabyte from a single disk over a shared network will encounter several different kinds of contention that do not scale linearly. An abstraction must take appropriate steps to carefully manage data transfer within the workload.
- **Latency vs Concurrency.** Dispatching sub-problems to a remote CPU can have a significant cost in a large distributed system. To overcome this cost, the system may increase the granularity of the sub-problems, but this decreases the available concurrency. To tune the system appropriately, the implementation must acquire knowledge of all the relevant factors.
- **Fault Tolerance.** The larger a system becomes, the higher the probability the user will encounter hardware failures, network partitions, adverse policy decisions, or unexpected slowdowns. To run robustly on hundreds of CPUs, our model must accept failures as a normal operating condition.

- **Ease of Use.** Most importantly, each of these problems must be addressed without placing additional burden on the end user. The system must operate robustly on problems ranging across several orders of magnitude by exploring, measuring, and adapting without assistance from the end user.

Examples of abstractions beyond the three mentioned above include Bag-of-Tasks [13, 26], Bulk Synchronous Parallel [22], and Map-Reduce [28]. None of these models is a universal programming language, but each is capable of representing a certain class of computations very efficiently. In that sense, programming abstractions are similar to the idea of systolic arrays [52], which are machines specialized for very specific, highly parallel tasks. Abstractions like All-Pairs and Wavefront are obviously less expensive than general purpose workflow languages such as DAGMan [100], Pegasus [29], Swift [107], and Dryad [49]. But, precisely because abstractions are regularly structured and less expressive, it is more tractable to provide robust and predictable implementations of large workloads. Once experience has been gained with specific abstractions, future work may evaluate whether more general languages can apply the same techniques.

3.2 Architecture

Figure 3.2 shows a general strategy for implementing abstractions on distributed multicore systems. The user invokes the abstraction by passing the input data and function to a *distributed master*. This process examines the size of the input data, the runtime of the function, consults a *resource catalog* to determine the available machines, and models the expected runtime of the workload in various configurations. After choosing a parallelization strategy, the distributed master submits sub-problems to the local *batch system*, which dispatches them to available CPUs. Each job consists of a *multicore master* which examines the executing machine, chooses a parallelization strategy, executes the sub-problem, and returns a partial result to the distributed master. As results are returned, the distributed master may dispatch more jobs and

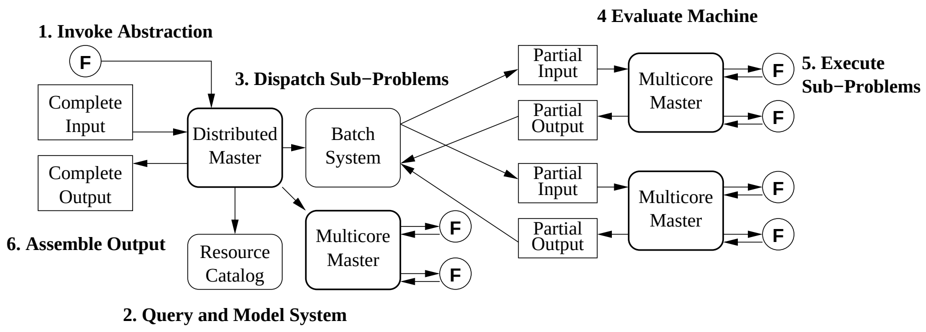


Figure 3.2: Distributed Multicore Implementation
All-Pairs, Wavefront, and other abstractions can be executed on multicore clusters with a hierarchical technique. The user first invokes the abstraction, stating the input data sets and the desired function. The distributed master process measures the inputs, models the system, and submits sub-jobs to the distributed system. Each sub-job is executed by a multicore master, which dispatches functions, and returns results to the distributed master, which collects them in final form for the user.

assembles the output into a compact final form.

For ease of use and implementation, both the distributed and multicore masters are contained in a single executable and invoked in the same way. Both All-Pairs and Wavefront are invoked by stating directories containing the input data and the name of the executable that implements the function:

```
allpairs  function.exe Adir Bdir
wavefront function.exe Rdir
```

Without arguments, the distributed master will automatically choose how to partition the problem. When dispatching a sub-problem to a CPU, the distributed master simply invokes the same executable with options to select multicore mode on a given sub-problem, for example:

```
wavefront -M -X 15 -Y 20 -W 5 -H 5 function.exe Rdir
```

Of course, this assumes that the necessary files are available on the executing machine. The distributed master is responsible for setting this up via direct file transfer, or specification through the batch system. Note that this architecture allows for more than two levels of hierarchy – a global master could invoke distributed masters on multiple clusters – but we have not explored this idea yet.

The user may specify the *function* in several different ways. The function is usually a single executable program, in which case the input data is passed through files named on the command line, and the output is written to the standard output. This allows the end user to choose whatever programming language and environment they are most comfortable with, or even use an existing commercial binary. For example, the All-Pairs and Wavefront functions are invoked like this:

```
allpairs_func.exe  Aitem Bitem       > Output
wavefront_func.exe Xitem Yitem Ditem > Output
```

Invoking an external program might have unacceptable overhead if the execution time is relatively short. To overcome this, the user may also compile the function into a threaded shared library with interfaces like this:

```
void * allpairs_function(
        const void *a, int alength,
        const void *b, int blength );

void * wavefront_function(
        const void *x, int xlength,
        const void *y, int ylength,
        const void *d, int dlength );
```

Regardless of how the code is provided, we use the term *function* in the logical sense: a discrete, self-contained piece of code with no side effects. This property is critical to achieving a robust, usable system. The distributed master relies on its knowledge of the function inputs to provide the necessary data to each node. If the function were to read or write unexpected data, the system would not function.

As the results are returned from each multicore master, the distributed master assembles them into a suitable external form. In the case of Wavefront, it is not realistic to leave each output in a separate file (although the batch system may deposit them that way), because the result would be millions of small files. Instead, the distributed master stores the results in an external sparse matrix. This provides efficient storage as well as checkpointing: after a crash, the master reads the matrix and continues where it left off.

The distributed master does not depend on the features of any particular batch system, apart from the ability to submit, track, and remove jobs. Our current implementation interfaces with both Condor [100] and Sun Grid Engine (SGE) [41], and expanding to other systems is straightforward. The distributed master also interfaces with a custom distributed system called Work Queue, which we will motivate and describe later.

To use Makeflow, a user needs to create a Makeflow script that describes the workflow of his workload. This language is very similar to traditional Make [34]: each rule states a program to run, along with the input files needed and the output files produced. Here is a very simple example:

```
part1 part2: input.data split.py
        ./split.py input.data


out1: part1 mysim.exe
        ./mysim.exe part1 >out1


out2: part2 mysim.exe
        ./mysim.exe part2 >out2
```

Like All-Pairs and Wavefront, Makeflow can run an entire workload on a local multicore machine, or submit jobs to Condor, SGE, or Work Queue. However, it does not have a hierarchical implementation: only single jobs are dispatched to remote machines. This is because graph partitioning is algorithmically complex, and impractical for heterogeneous workloads where runtime prediction is unreliable. Put simply, Makeflow has greater generality, but this comes at the cost of implementation efficiency, as we will emphasize below.

3.3 Building Blocks

Our overall argument is that highly restricted abstractions are an effective way of constructing very large problems that are easily composed, robustly executed, and highly scalable. To evaluate this argument, we will begin by examining several questions about each abstraction at the level of microbenchmarks, then evaluate the system has a whole.

3.3.1 Threads and Processes

It is often assumed that multicore machines should be programmed via multi-threaded libraries or compilers. Our technique instead employs *processes*, because they are more easily adapted to distributed systems. How does this decision affect performance at the level of a single machine?

As a starting point, we constructed simple benchmarks to measure the time to dispatch a null task using various techniques. Each measurement is repeated one thousand times, and the average is shown. (Unless otherwise noted, the benchmark machine is a 1GHz dual core AMD Opteron model 1210 with 2GB RAM running Linux 2.6.9.) Table 3.1 shows the results. `pthread` creates and joins a standard POSIX thread on an empty function, `fork` creates and works for a process which simply calls `exit`, `exec` forks and executes an external program, and `popen` and `system` create new sub-processes invoked through the shell.

TABLE 3.1

TIME TO DISPATCH A TASK

Method	**Time**	
pthread	6.3	μs
fork	253	μs
exec	830	μs
popen	2500+	μs
system	2500+	μs

It is no surprise that creating a thread is several orders of magnitude faster than creating a process. However, it is not so obvious that `popen` and `system` are considerably more expensive than `exec`, and often vary in cost from user to user. This is because these methods invoke the user's shell along with their complex startup scripts, which can have unbounded execution time and create troubleshooting problems. If we are careful to avoid these methods, then executing an external program can be made reasonably fast. Moreover, it is only necessary for the execution time to dominate the invocation time: a task in an abstraction running for a second or more is sufficient.

3.3.2 Concurrency and Data in All-Pairs

Of course, within a real program, we must weigh invocation time against more complex issues such as synchronization, caching, and access to data. To explore the boundaries of these issues, we studied the All-Pairs multicore master running in sequential mode on a single machine, comparing 1MB randomly generated files. A simple comparison function counts the number of bytes different in each object. From a systems perspective, this is similar to a biometrics problem, and provides a

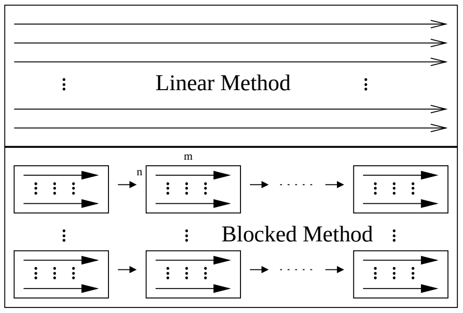


Figure 3.3: Linear Method vs Blocked Method.
The linear method evaluates cells in the matrix line by line. The blocked method evaluates cells block by block with a width chosen to fit in the file system buffer cache.

high ratio of data to computation. Any realistic comparison function would be more CPU intensive, so these tests explore the worst case.

In this scenario, we vary several factors. First, we vary the invocation method of the function: create a thread to run an internal function (*thread*) or create a process to execute an external program (*process*). The author of a function is free to choose their own I/O technique, so we also compare buffered I/O byte-by-byte (*fgetc*), block-by-block (*fread*), and memory-mapped I/O (*mmap*). A naive implementation would simply iterate over the output matrix in order, causing cache misses at all levels on every access. A more effective method, as shown in Figure 3.3, is to choose a smaller block of cells and iterate over those completely before proceeding to the next block. The width of the block is referred to as the *block size*. (This technique is sufficient for our purposes, but see Frigo et al [37] for more clever methods.)

Figure 3.4 shows the relative weight of all these issues. Each curve shows the runtime of a 1000×10 comparison over various block sizes. The two slowest curves are *thread* and *process*, both using *fgetc*. The two middle curves are *process* using *fread* and *mmap*, and the fastest is *thread* with *mmap*. All curves show significant slowdown when the block size exceeds physical memory.

Clearly, threads with *mmap* execute twice as fast as the next best configuration.

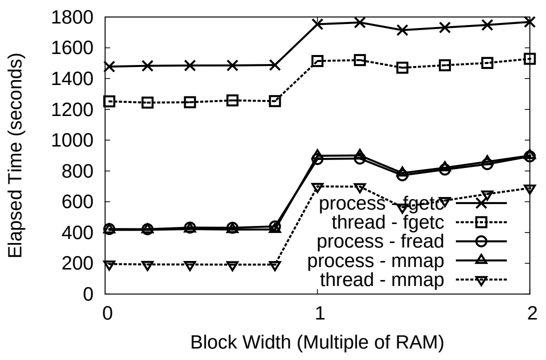


Figure 3.4: Threads, Processes, and I/O Techniques.
The performance of a data intensive 1000×10 All-Pairs in sequential mode using threads and processes with various I/O techniques. While threads provide the best performance, processes are a reasonable method even on this worst case.

If the user is willing to write a thread-safe function for use with the abstraction, they should do so. However, the use of processes is only twice as slow *in this artificial worst case* and will not fare as poorly with a more CPU-intensive function. Moreover, the appropriate use of virtual memory by the abstraction and the I/O technique chosen by the function are *much more significant factors* than the difference between threads and processes. We conclude that using processes to exploit parallelism is a reasonable tradeoff if it improves the usability of the system.

We re-emphasize that each abstraction can accept either an external program or a threaded internal function. So far, none of our users has chosen to use threads.

Next we consider how to carry out All-Pairs on a multicore machine. Although there are many possible ways, we may consider two basic strategies. One is to generate N contiguous sub-problems, and allow each core to run independently. The other is to write an explicit multicore master that proceeds through the entire problem coherently, dispatching individual functions to each core. Figure 3.5 compares both of these against a simple sequential approach. As can be seen, the sub-problem approach performs far worse, because it does not coordinate access to data, and caches at all levels are quickly overwhelmed. Thus, we have shown it is necessary

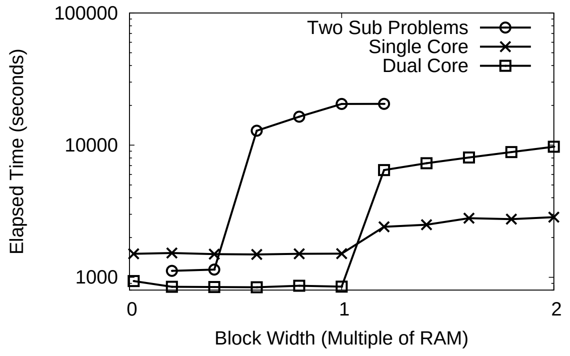


Figure 3.5: Multicore vs Sub-Problems.
The performance of an 1000×10 All-Pairs in sequential mode, in dual-core mode, and as two independent sequential sub-problems, using various block sizes. This demonstrates the importance of an explicit multicore strategy.

to have a deliberate multicore implementation, rather than treating each core as a separate node.

3.3.3 Control Flow in Wavefront

As we have shown, the primary problem in efficient All-Pairs is managing data access. However, in Wavefront the problem is almost entirely control flow. The first task of the problem is sequential. Once completed, two tasks may run in parallel, then three, and so forth. If there is any delay in dispatching or completing a task, this will have a cascading effect on dependent adjacent tasks. We will consider two control flow problems: dispatch latency and run-time variance.

Figure 3.6 models the effect of latency on a Wavefront problem. This simple model assumes a 1000×1000 problem where each task takes one second to complete. On the X axis, *block size* indicates the size of sub problem dispatched to a processor. Each curve shows the runtime achieved for a system with dispatch latency ranging from zero (e.g. a multicore machine) to 30 seconds (e.g. a wide area computing grid).

As block size increases, the sub-problem runtime increases relative to the dispatch latency, but less parallelism is available because the distributed master must wait for

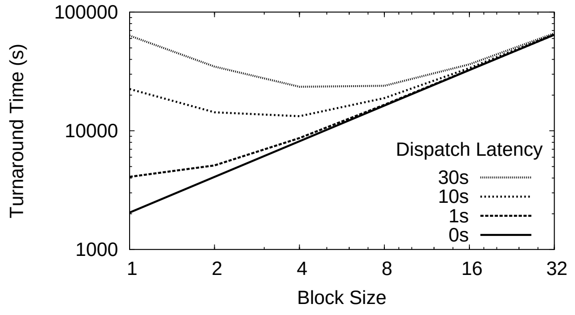


Figure 3.6: The Effect of Latency on Wavefront
The modeled runtime of a 1000×1000 Wavefront where each function takes one second to complete, with varying block size and dispatch latency. As dispatch time increases, the system must increase block size to overcome the idle time.

an entire sub-problem to complete before dispatching its neighbors. The result is that for very high dispatch times, a modest block size improves performance, but cannot compete with a system that has lower dispatch latency. So, the key to the problem is to minimize dispatch latency.

Although Wavefront can submit jobs to Condor and SGE batch systems directly, the dispatch latency of these systems when idle is anywhere from ten to sixty seconds, depending on the local configuration. For short-running functions, this will not result in acceptable performance, even if we choose a large block size. (This is not an implementation error in either system, rather it is a natural result of the need to service many different users within complex policy constraints.)

To address this, we borrowed the idea of a *fast dispatch execution system* as in Falkon [80]. We built a simple framework called Work Queue that uses lightweight worker processes that can be submitted to a batch system. Each contacts the distributed master, and provides the ability to upload and execute files. This allows for task dispatch times measured in milliseconds instead of seconds. Workers may be added or removed from the system at any time, and the master will compensate by assigning new tasks, or reassigning failed tasks.

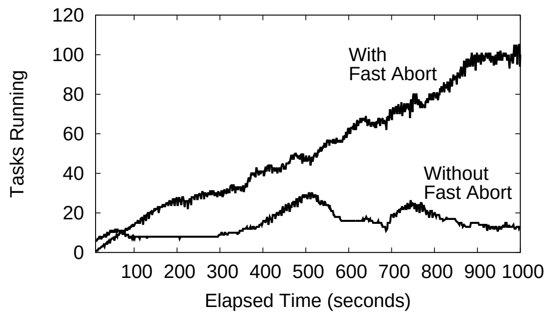


Figure 3.7: The Effect of Fast Abort on Wavefront
The startup behavior of a 500×500 Wavefront with and without Fast Abort. Without Fast Abort, every delayed result impedes the increase in parallelism, which stabilizes around 20. With Fast Abort, delays are avoided and parallelism increases steadily.

However, even if we solve the problem of fixed dispatch latency, we must still deal with the unexpected delays that occur in distributed systems. When Work Queue runs on a Condor pool, a running task may still be arbitrarily delayed in execution. It may be evicted by system policy, stalled due to competition for local resources, or simply caught on a very slow machine. To address these problems, the Work Queue scheduler keeps statistics on the average execution time of successful jobs and the success rate of individual workers. It makes assignments preferentially to machines with the fastest history, and proactively aborts and re-assigns tasks that have run longer than three standard deviations past the average. These techniques are collectively called Fast Abort.

Figure 3.7 shows the impact of Fast Abort on starting up a 1000×1000 Wavefront on 180 CPUs. Without Fast Abort, stuck jobs cause the workload to however around twenty tasks running at once. With Fast Abort, the stragglers are systematically resolved and the concurrency increases linearly until all CPUs are in use. Figure 3.8 shows this behavior from another perspective. The distributed master periodically produces a bitmap showing the progress of the run. Colors indicate the state of each cell: red is incomplete, green is running, and blue is complete. Due to the

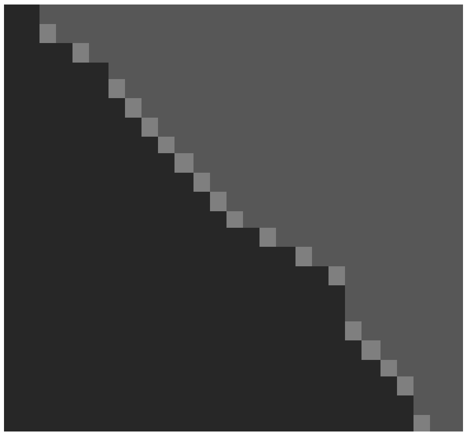


Figure 3.8: Asynchronous Progress in Wavefront
A progress display from a Wavefront problem. Each cell shows the current state of a portion of the computation: the darkest gray in the lower left corner indicates incomplete, the lighter gray in the upper right indicates complete, and the light cells in between are currently running. The irregular progress is due to heterogeneity and asynchrony in the system.

heterogeneity of the underlying machines, the wave proceeds irregularly. Although an N×N problem should use N CPUs at maximum, this perfect diagonal is rarely seen.

3.3.4 Greater Generality with Makeflow

Makeflow provides a different type of building block for large multicore workflows with abstractions. Makeflow combines many functions together (instead of many instances of the same function) to express more complex series of operations.

Makeflow uses a syntax very similar to traditional Make, but it differs in one critical way: each rule of a Makeflow must exactly state *all of the files* consumed or created by the rule. (In traditional Make, one can often omit files, or add dummy rules as needed to affect the control flow.) Makeflow is more strict, but this allows it to accurately generate batch jobs, exploit common patterns of work, and schedule jobs to where their data is located. This allows Makeflow to run correctly on both

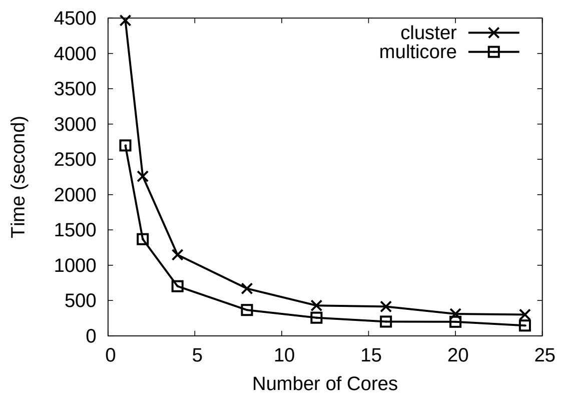


Figure 3.9: Makeflow on Multicore and Cluster
The performance of a genomics application run through Makeflow, using 1-24 cores on both a multicore machine and a cluster using Work Queue.

local multicore machines as well as a distributed system.

The Makeflow abstraction can be configured to use different numbers of cores. Figure 3.9 shows the turnaround times varying the number of cores used with two different options for executing a genomics workload on 1-24 cores. The top curve ("cluster") presents Makeflow using Work Queue, with workers submitted to remote machines as Condor jobs. The bottom curve ("multicore") executes all work as Makeflow-controlled local processes, in which Makeflow automatically takes advantage of multiple cores on the submitting machine. Makeflow jobs running locally outperform jobs tasked to remote workers and scale well up to the number of available cores.

3.4 Performance Modeling

In a well-defined dedicated environment in which the distributed master knows exactly which resources will be used, a model can partition work to the resources in such a way as to optimize the workload [101]. This applies to multicore environments as well – the distributed master could build multicore assumptions into the model to optimize a workload. However, this finely-tuned partitioning does not adapt well to heterogeneous environments or resource unavailability. Previous work derived a more realistic solution for modeling the turnaround time of an All-Pairs workload in a cluster [66]. Is it possible to use the multicore version of the All-Pairs abstraction transparently beneath the cluster abstraction?

If the abstraction is to use the multicore master transparently, then it must continue to exclude considerations of the number of cores per node from the model . If the workload is benchmarked on a single-core system or with a single-threaded executor, then the model will choose appropriate resources to run the workload efficiently assuming single-threaded operation. Adding multicore execution to this workload, then, will only serve to make the batch jobs complete faster on the multicore resources. It does not change the overall workload any more than having benchmarked on a slow node would: the success of the model in avoiding disastrous cases is maintained, the faster resources (in this case multicore nodes) will account for a greater portion of the batch jobs than their "fair share", and any long-tail from slow nodes would extend out at most to the same duration as without any multicore nodes.

So it is possible, but this is little solace if there is a clearly better solution for modeling a distributed All-Pairs workload using multicore resources. Another option is to integrate the multicore master (instead of the original single-threaded executor) into the benchmarking process for the model. If the function runtime is benchmarked using the multicore master, then the function execution time (computed as the average time per function over a small set of executions) will be comparable to the expected execution of batch jobs on the same number of cores. This is a good approach for submitting to homogeneous clusters of resources in which the same number of cores are available for every batch job. In a heterogeneous environment, however, this only serves to exacerbate the model's assumption that the benchmark node reflects the cluster's resources. Whereas the original model conceded that individual resources might be perhaps a generation newer (faster) or older (slower) than the benchmark node, the inclusion of multicore uncertainty into the benchmarking increases the potential range of resource capabilities and thus the potential for long-tail effects in a workload.

Another option would be to include a coefficient of the average number of cores within the model. Because the model includes a component for the time to complete a single batch job, an adjustment for the number of cores could be made by dividing the batch job execution time in the model by this average. This retains the same prerequisite measurements (plus the calculation of the average number of cores), however it has several limitations. First, the pool of resources must be well-defined so that the average number of cores may be determined; but because the model is used to select the appropriate number of resources, the exact set of hosts is not known *a priori*. Thus, the average number of cores available for each host is a pool average rather than one specific to the actual resources used. Further, contention for resources means that not all hosts will be utilized equally or predictably, which presents the same problem in trying to include a factor of the number of cores in the turnaround time model. This is especially problematic as we move beyond workstations with at most a few cores: unavailability of a machine with dozens of cores significantly changes the average number of cores of the available machines.

With that said, can we accurately model the performance of our abstractions? Figure 3.10 shows the modeled performance of All-Pairs workloads of varying sizes running on an 8-core machine and a 64-core cluster. Figure 3.11 shows the modeled performance of Wavefront workloads running on a 32-core machine and a 180-core cluster. In both cases, the multicore model is highly accurate, due to a lack of competing users and other complications of distributed systems. Both models are sufficiently accurate that we may use them to choose the appropriate implementation at runtime based on the properties given to the abstraction. Figure 3.12 shows the

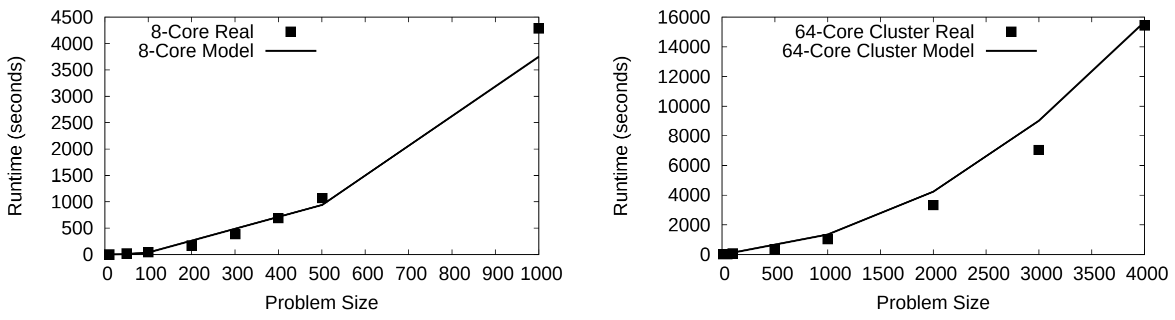


Figure 3.10: Accuracy of the All-Pairs Model on Multicore and Cluster
The real and modeled performance of an All-Pairs benchmark of varying sizes on a 8-core machine (left) and an 64-core cluster (right).

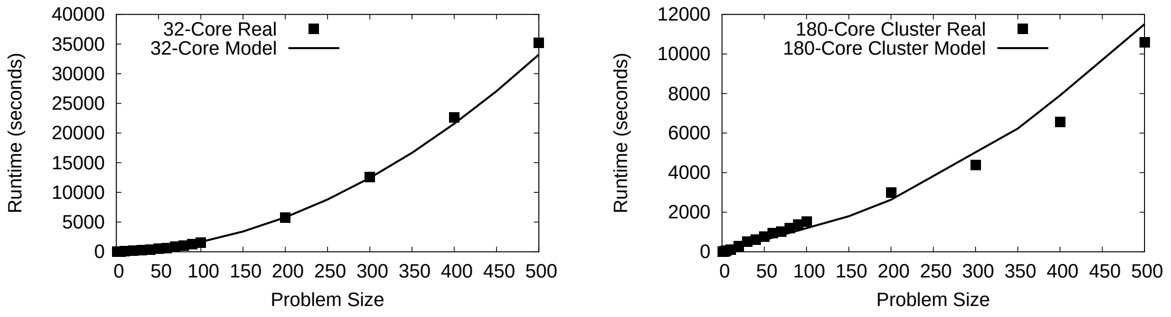


Figure 3.11: Accuracy of the Wavefront Model on Multicore and Cluster
The real and modeled performance of a Wavefront benchmark of varying sizes on a 32-core machine (left) and an 180-core cluster (right).

modeled performance of Makeflow workloads running on a 24-core machine and a 60-core cluster. Figure 3.13 compares the multicore and cluster models for the previous All-Pairs and Wavefront examples, and demonstrates the actual performance achieved when selecting the implementation at runtime.

3.5 Makeflow vs Specific Abstractions

With the Makeflow abstraction for arbitrary DAG workflows, could we choose to use it as a general tool instead of implementations of the specific abstractions mentioned above? In our experience, the answer is that we could, but in doing so

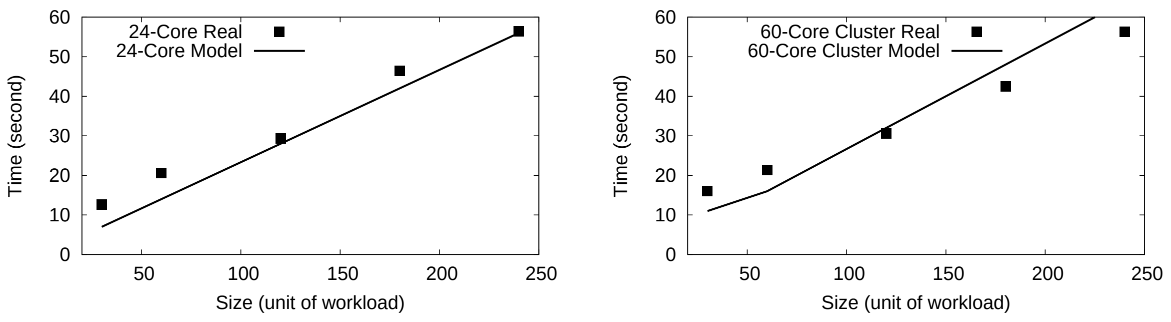


Figure 3.12: Accuracy of the Makeflow Model on Multicore and Cluster
The real and modeled performance of a Makeflow benchmark of varying sizes on a 24-core machine(left) and a 60-core cluster(right).

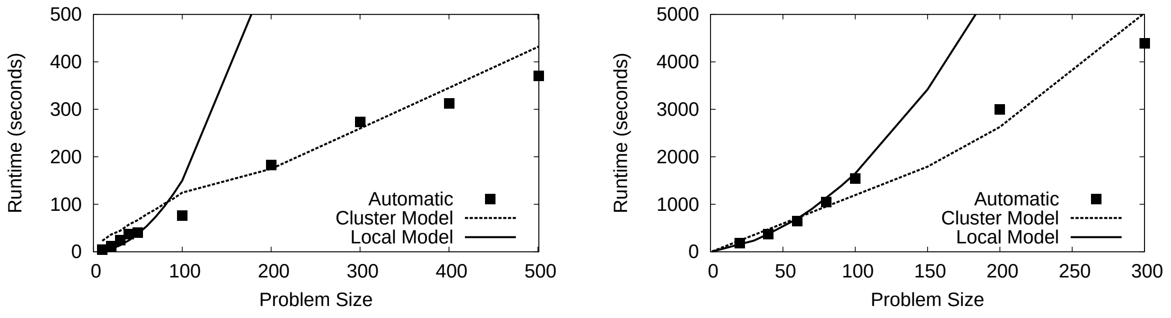


Figure 3.13: Selecting An Implementation Based on the Model
These graphs overlay the modeled multicore and cluster performance on problems of various sizes for All-Pairs (left) and Wavefront (right). The dots indicate actual performance for the selected problem size. As can be seen, the modeled performance is not perfect, but it is sufficient to choose the right implementation.

we lose many of the problem-specific advantages given by the less general abstractions. We carry out All-Pairs on a 24-core machine using both the all-pairs multicore abstraction and the Makeflow abstraction.

We vary the size of the workloads from creating a 10×10 matrix to creating a 1000×1000 matrix. Each matrix cell is computed by comparing two 20KB files. With the Makeflow abstraction, each cell value depends on a comparison and the cell value is stored in a file after it is computed. And we have to write an additional program, which depends on all the cell value files, to extract all cell values from generated

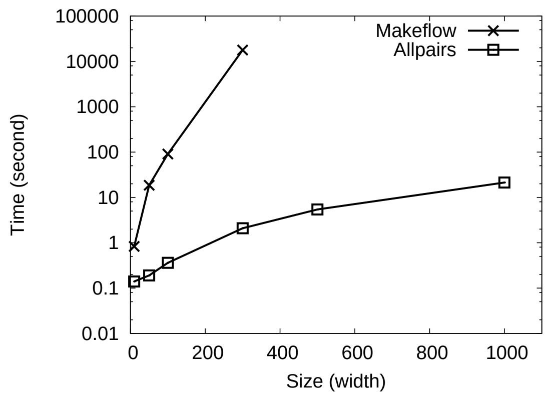


Figure 3.14: Solving All-Pairs with Makeflow and All-Pairs
This figure shows the time to complete an All-Pairs problem of various sizes using the general Makeflow tool and the specific All-Pairs tool. The more general tool is considerably more expensive, because it uses files for output storage, and is unable to dispatch sub-problems to multicore processors.

files and put them into the target matrix. The running time of both abstractions on different workloads are shown in Figure 3.14. It is easy to see that the All-Pairs multicore abstraction scales almost linearly as the workload increases. However, the Makeflow abstraction is several orders of magnitude slower at this problem, because it uses files for output storage, and is unable to manage work in organized blocks.

The increased generality of Makeflow has a significant price, so we conclude that there is a great benefit to retaining specific abstractions such as All-Pairs and Wavefront for specialized problems.

CHAPTER 4

DYNAMIC WORKLOADS

A dynamic workload is a workload whose computational structure is unknown prior to execution. Because the workload structure is unknown, the abstraction modeling method we used for static workloads is impossible to be applied. The resource allocation problem on general dynamic workloads is difficult. The dynamic capacity management method described in this chapter relies on the following three assumptions that reduces the problem complexity to a solvable level: (a) applications are composed of tasks that have a relatively stable computation-to-data ratio, (b) task properties are not known in advance, and (c) there are more tasks than workers available, so that there is an opportunity to measure and adjust over time. The system will adapt to changes in computation-to-data ratio, as long as the task mix is stable for a sufficient period for the system to adjust.

4.1 Architecture

This work is done in the context of the Work Queue [19] elastic application framework as shown in Figure 4.1, but the lessons apply to any system where the provisions of the computing resources must be right-sized to the applications. In this framework, a **master** process represents a particular application, generating tasks with are sent off to multiple **workers**. Any workload that runs as a Work Queue master is automatically elastic as it can adapt to different computing resource availabilities and progress through resource loss. The master advertises its location and other details

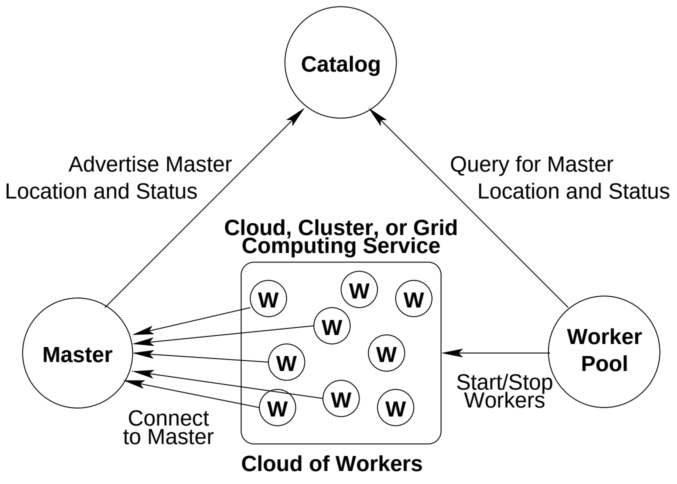

Catalog
Advertise Master Location and Status
Query for Master Location and Status
Cloud, Cluster, or Grid Computing Service
W
Master
Worker Pool
Start/Stop Workers
Connect to Master
Cloud of Workers

Figure 4.1: Work Queue Architecture

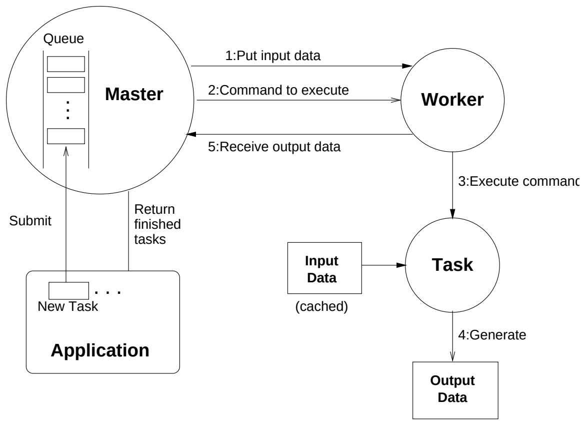


Figure 4.2: Detail of Master-Worker Interaction

to a **catalog** server, which makes it known to the **worker pool**, which is responsible for setting up and maintaining an appropriate number of workers for the master.

A **master** program is constructed by creating a custom application and linking it with the Work Queue master library. The library provides an API in multiple languages, allowing the caller to define a new task, submit it to the queue, and wait for a task to complete. Each task consists of an arbitrary command line to be executed, along with the executables and files necessary to carry it out. In pseudo-code a program written to the Work Queue API looks like this:

```
while( not_done ) {
    for( each_new_task ) {
        task = work_queue_task_create( details );
        // add more details to the task
        work_queue_task_submit( task );
    }
    task = work_queue_task_wait( timeout );
    if( task ) {
        process_result(task);
        work_queue_task_delete( task );
    }
}
```

A standard **worker** program is common to all Work Queue applications. The worker is a lightweight executable that can be deployed to any kind of computational infrastructure and connects back to the desired master program. The master and worker then work together to move the necessary files and executables to the worker, and the run the task's command line. Together the master and worker are robust to a variety of network and system failures, so that when something goes wrong, a task can be re-assigned to another worker.

In the most basic form, a number of workers can be started by hand and given the exact address and port of a known master to work with. However, managing addresses and ports becomes tiresome at large scale, so a **catalog** is used for discovery instead. The master may optionally advertise its name, location, and other details to the catalog server. When this is done, workers may be started by simply indicating the logical name of the master – each worker queries the catalog for the location of the desired master. The catalog is also the point of contact for a number of status tools that can quickly list the running applications, number of workers, progress toward completion, etc.

For clarity, we will confine our discussion to the Work Queue software. However, the principles are general and could easily apply to any system with multiple worker nodes and the dynamic generation of work, including but not limited to Azure [47], SAGA [58], Falkon [80], Swift [107], and Condor-MW [57], to name a few.

4.2 The Problem of Capacity

The framework this far – master, workers, and catalog – has been to construct a variety of large scale scientific applications, including scalable genome assembly [68], genome analysis tools [102], and advanced molecular dynamics ensembles [5]. As mentioned in Chapter 3, Work Queue is also used as the driver underlying higher level computing abstractions, such as All-Pairs, Wavefront, and Makeflow. Typical applications manage thousands to millions of tasks running on hundreds to thousands of workers.

However, a common problem across these applications is that managing the set of workers becomes a burden on the end user as scale increases. Submitting a large number of workers to a batch system or to a public cloud may take significant time and be interrupted by network failures. System outages or other problems may cause workers to be returned to the user, as if complete. Further, the end user must judge the proper number of workers that the application could use, and manage them them manually at runtime. In our experience, the necessary information was completely opaque to the end user, so that often 1000 workers would be running when only 10 were necessary, or vice versa.

To address this problem, we introduce a new component into the framework. The **worker pool** serves to deploy and maintain the proper number of workers needed on a resource for the benefit of the elastic application. The user starts the single process once with basic instructions (e.g. *Don't run more than 500 workers or spend more*

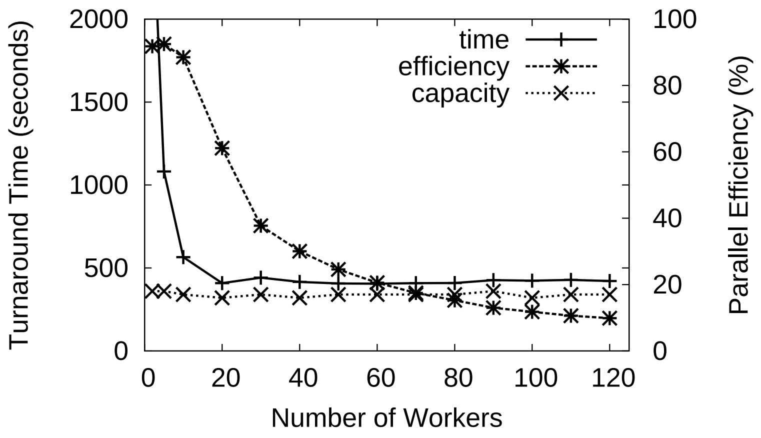


Figure 4.3: Performance vs Parallel Efficiency

than $100.) and then is relieved of further details. The worker pool is built with drivers that allow it to submit and remove jobs from traditional batch systems like Condor and SGE as well as public clouds like Amazon EC2 and Microsoft Azure[84].

This chapter focuses on the key design question of the worker pool, which is simply this:

How many workers should be running at once?

Put simply, if too few workers are run at once, the master will be left underutilized, missing on opportunities to execute tasks. If too many workers are run at once, the master will not have enough bandwidth (or other resources) to keep them busy, and workers will be left idle, wasting those allocated resources and possibly decreasing overall performance.

To demonstrate this problem, we constructed a simple benchmark application, generating 1000 uniform tasks where each task uses 1 MB of input data, takes 5 seconds to execute, and generates 1 MB of output data, and then ran it on a varying number of dedicated workers from 1 to 120. Figure 4.3 shows the turnaround time and parallel efficiency of running the benchmark application with a varying number of workers. Equation 4.1 shows how the estimated parallel efficiency is calculated at runtime. T_i is the execution time of the i_{th} task and N is the number of tasks that are already completed. Equation 4.2 shows how the observed parallel efficiency is computed. In this equation, T_{seq} is the turnaround time of executing the application tasks sequentially. In both equations, $T_{wall-clock}$ is the turnaround time of executing the application tasks in parallel. As can be seen, the application turnaround time first decreases as more workers are provided. The parallel efficiency, however, keeps decreasing because more workers adds more communication overheads. When the number of workers reaches 20, adding more workers no longer improves the performance because the master is unable to keep all the workers always doing useful work.

$$calculated = (\sum_{i=1}^{N} T_i)/(N * T_{wall-clock}) \tag{4.1}$$

$$observed = T_{seq}/(N * T_{wall-clock}) \tag{4.2}$$

For the application in Figure 4.3, 20 is close to the optimal number of workers to provide if the goal is to maximize the performance. When the number of workers are less than 20, the master experiences idle times as it would spend time waiting for tasks to finish, which means that the master has additional resource to take on more workers and reduce the turnaround time through increased parallelism. When the workers number is over 20, while the master's idle times are greatly reduced, the workers start to experience idle periods. During a worker's life cycle, there are several stages in which it could be idle – not doing useful work, such as waiting for the master to accept its connection, waiting for a new task from the master, or waiting for the master to receive its computing results. When a master is too busy to handle all the workers in time, such idle times emerge and lead to resource waste.

This is all easy enough to state on paper, but in a production setting, it is very challenging to determine whether an application is running efficiently. Moreover, the user who wishes to accomplish real work has no interest in running the application many times to determine the optimal configuration – it is much more important to

TABLE 4.1

CAPACITY IN VARIOUS SYSTEMS – EXPERIMENT SETUP

	Master Location	Worker Location
A	Campus Workstation	Campus Condor
B	Campus SGE Head Node	Campus SGE
C	Campus Workstation	FutureGrid
D	FutureGrid Head Node	FutureGrid
E	Campus Workstation	Amazon EC2
F	Amazon EC2	Amazon EC2

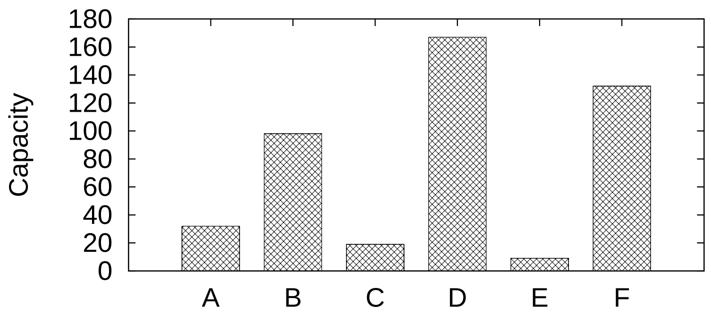


Figure 4.4: Capacity in Various Systems – Results

make a good choice the first time. Further, the scale achievable may vary considerably as the same application is moved between systems.

Figure 4.4 demonstrates this by showing the optimal number of workers for the same benchmark application moved between multiple computing environments as shown in Table 4.1: (A) Master on a workstation, workers on a campus Condor pool. (B) Master and workers on a dedicated campus cluster. (C) Master on a workstation, workers on FutureGrid. (D) Master and workers on FutureGrid. (E) Master on a workstation, workers on Amazon EC2. (F) Master and workers on Amazon EC2. As can be seen, the optimal number of workers varies from 9 to 167, depending on the bandwidth available to the master. Thus, an ideal resource management system must be flexible such that the amount of resources for a workload is determined dynamically according to the runtime environment.

4.3 Measuring Capacity

We define the master's capacity as the maximum number of workers whose average I/O bandwidth needs add up to the available I/O bandwidth at the master. The optimal numbers of workers we show in Figure 4.3 and 4.4 are the master capacities. Since our goal is to avoid wasting computing resources, we want the number of workers provided to a master to be no greater than the master's capacity. As stated earlier, users have no interests in running an application multiple times to determine the capacity for a given setting. Thus, we introduce a method to dynamically estimate master capacity at application runtime.

4.3.1 A Simple Equation

We consider applications that contain independent tasks of similar sizes. That is, for each task, the sizes of its input and output data, as well as the task execution time are similar its peers. Many bioinformatics and biometrics applications mainly contain tasks of similar sizes. Thus, such simplification is of practical value. In this simplified model, the number of sufficient workers to keep a master busy equals the number of tasks that the master can dispatch before the first handled task is complete.

A simple equation to estimate a master's capacity is shown in equation 4.3. In this equation, T_{tran} is the time to transfer a task, which includes the transfer time of both the input and output data. T_{exe} is the time to execute a task on a worker. Equation 4.3 shows that a master's capacity is dominated by the value of T_{exe}/T_{tran}, which will be referred to as the computation/data ratio later in this paper. The intuition behind the capacity estimation is that a masters capacity is approximately the very number of workers that reduces the master idle time to the minimal level. Of course, a master can never be 100% busy due to queueing effects. But when the amount of workers provided is equal to capacity, the master will be just busy enough sending out tasks. Note that the capacity value could be less than 1, in which case there is no advantage to run on remote workers.

$$capacity = T_{exe}/T_{tran} \tag{4.3}$$

Equation 4.3 makes several assumptions that are unlikely to be true in a real distributed environment, namely that the network bandwidths between the master and different workers are the same, and that each worker spends the same amount of time to execute the tasks of the same size. Thus, we can not rely on a single task's data transfer time and execution time to calculate the master's capacity. To incorporate the heterogeneity of distributed systems into the capacity calculation, we replace the items in equation 4.3 with their corresponding average values and get equation 4.4.

$$capacity = T_{avg\_exe}/T_{avg\_tran} \tag{4.4}$$

4.3.2 Sample Selection

An implicit parameter of equation 4.4 is the task samples on which the averages are computed. To make more accurate estimates on the capacity, the selected task samples should reflect the up-to-date capacity of the master. As mentioned earlier, a master's capacity is dominated by the computation/data ratio of the tasks. If the this ratio does not change throughout the execution, choosing the entire set of already finished tasks as the sample is desirable because having the largest possible sample size best offsets the effects of outliers such as a extremely slow worker.

However, there are cases where the computation/data ratio may change over the course of an execution. For example, in the Work Queue framework, the workers are able to cache input files. Whenever a task is dispatched to a worker, any of its input data that is already cached on the worker will not be transferred. To give an example of the computation/data ratio's changing effect, we consider an application with tasks that all share the same input data. At the beginning of the execution, no input data is cached on any workers, so the master has to transfer the shared input file to every worker. If the number of tasks is large enough, eventually all the workers would have the shared input file cached on them. Further tasks would then have less input data to transfer, which changes the computation/data ratio.

Another example would be applications that contain different types of tasks. Consider a slightly more complex application that contains two sequential steps where each step contains tasks of the same type, that is, the computation/data ratios of the tasks are the same. The computation/data ratios in the two steps, however, are different. Assume that the number of tasks in both steps are large enough, the master's capacity at the beginning when all step-one tasks are being executed will be different from that when step-two tasks occupy the workers.

In either case, the entire set of finished tasks might contain tasks that do not reflect the up-to-date mater's capacity. Computing averages on these tasks might lead to greater inaccuracy. Thus, we limit the task samples to the most recently finished N tasks where N is the number of busy workers. Although tasks of different computation/data ratios may be executed at the same time, as long as the number of the same type tasks is large enough, eventually the workers will be filled with the same type tasks and the estimated capacity will move toward the mater's actual capacity.

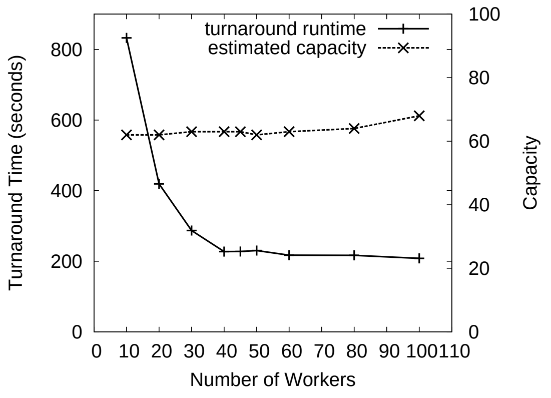


Figure 4.5: Capacity without Think Time

4.3.3 Final Equation

When we first tried using equation 4.4 to estimate the master's capacity, we observed some discrepancies between the estimated master's capacity and the actual master's capacity.

To obtain a close estimate of a master's actual capacity, we execute the same application multiple times with different numbers of workers. When the turnaround time does not improve as the number of workers increases, we record the number of workers at that point as the actual master's capacity for that application. Figure 4.5 shows the results of running a same benchmark application with different amount of workers. The synthetic application contains 800 independent tasks where each task has 5 MB input data, 10 seconds execution time, and generates 5 MB output data. The master estimates its capacity whenever a task finishes. The estimated capacity shown in Figure 4.5 is the dominating capacity value among all the capacity estimates computed during a single execution.

We then discovered that the actual capacity of the application is around 40 as the performance no longer improves after the number of workers goes beyond 40. However, the estimated capacities of the application are around 60 regardless of the number of workers provided. The reason for the deviation in the estimated capacity was the master spending extra time communicating with the upper-level application. When a task finishes, the master returns the task result to the upper-level application so that the upper-level application can take appropriate actions according to the task result. During this communication, the master can not serve any workers which may increase the idle times on the workers. The time that a master spends in communicating with the upper-level application is referred to as think time.

$$capacity = T_{avg\_exe}/(T_{avg\_tran} + T_{avg\_think}) \tag{4.5}$$

After incorporating the think time–T_{avg_think} into the capacity estimation leads to equation 4.5. The new estimated capacities of the same application as we have shown in Figure 4.5 became mostly around 40, which is more accurate compared to the previous estimates of around 60. Further experiments with different numbers of workers and types of tasks have also yielded improved accuracy on capacity estimates.

When more than the capacity amount of workers are provided, the master becomes overloaded and greater resource waste emerge at no performance benefits. In Figure 4.3, the *capacity* curve shows the capacity estimates at the end of each entire run. **No matter how many workers are provided to the master, the estimated capacity (18) is close to the optimal number of workers that achieves the maximum performance.**

4.3.4 Dynamic Behaviors

The capacity estimates we have shown in Figure 4.3 are static estimates obtained at the end of the entire runs. In this subsection, we show two extreme cases of dynamic behaviors in capacity estimation that are worth observing: settle time and cache bounce.

Figure 4.6 shows the settle time behavior in dynamic capacity estimates when

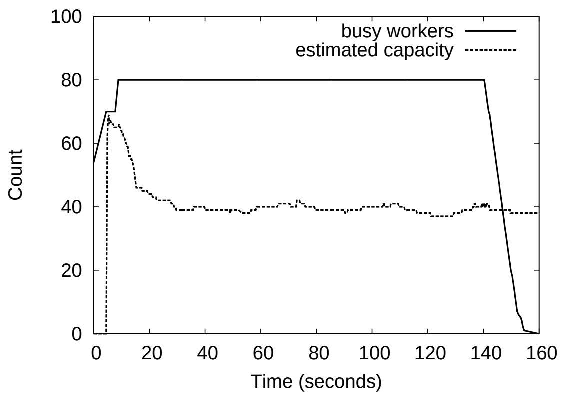


Figure 4.6: Settle Time on an Overloaded Master

80 workers are provided to a benchmark application of 600 independent, uniform tasks. Each task in this workload has unique 5 MB of input, takes 10 seconds to execute, and returns 5MB of output. The estimated capacity peaks (around 70) at the beginning of the execution because the tasks execution statistics are only available from the first few incidental faster workers. As more workers join the master, that is the sample size for calculating the capacity becomes larger, the estimated capacity drops down and become stable at around 40 throughout the rest of the execution. We have repeated the same experiment multiple times only varying the amount of workers, and we have observed that adding more workers beyond 40 does little to none improvement to the application turnaround time. We refer to this initial period when the capacity estimates are unstable and inaccurate as the settle time.

Figure 4.7 shows the cache bounce behavior in a different workload of 500 independent, uniform tasks. Unlike the workload shown in Figure 4.6, all the tasks in this workload share the same 1 GB of input. Each task takes 5 seconds to execute and generates 1 MB of output. Initially, only 1 worker is provided to the master. The first task transfer involves the transfer of the 1 GB input. Because the worker is able to cache input data, later task transfers incurs no input transfer, which changes the C/C ratio of the tasks. Because we configured the low-bound for sample tasks size to

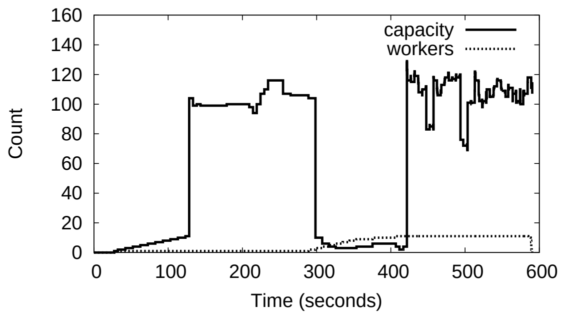


Figure 4.7: Cache Bounce on an Underloaded Master

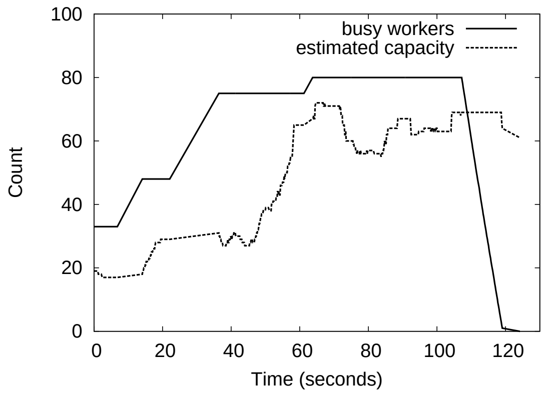


Figure 4.8: Settle Time and Cache Bounce in a Shared-Input Application

be 10, the first task's execution statistics stayed as an outlier in capacity estimation until the 11th task is returned. 10 more workers are added to the master when the estimate capacities become relatively stable. Because the new workers had to repeat the 1 GB input transfer, the capacity declines first and then returns to the previous level as more workers have the shared-input data cached. We refer to the resulted U-shaped capacity curve as the cache bounce.

Figure 4.8 shows the combined effects of settle time and cache bounce in a benchmark application that contains a total of 600 tasks. All the tasks share the same 50 MB input data. Each task takes 10 seconds to execute and generates 5 MB of output. The estimated capacity starts out relatively low (quickly settled at around 20) because every task transfer includes the transfer of the shared 50 MB of input. As the workers begin to have that input data cached on themselves, input data transfer time is shortened and the capacity estimate starts to increase. The cache bounces can be seen at the 40 and 80 second points. After the bounces, the estimated capacity becomes stable at around 60. Because we take task samples only from the most recently finished tasks, the estimated capacity would eventually become in accordance with the true computation/data ratio of the tasks being executed as long as there are enough tasks for the estimation to adjust to.

4.4 Worker Distribution

The capacity estimation allows individual masters to report their introspects on how many workers they need. The worker pool utilize these estimates, as well as other metrics of the master's status, to allocate a proper amount of workers to the masters at runtime.

4.4.1 Master Advertisement

A master, when runs in the catalog mode, advertises its status information to a catalog server periodically. The following list shows some of the items that are sent to the catalog server from a master:

- *hostname* – the hostname of the machine that the master is running on.
- *port* – the port number that the master is listening on for worker connections.
- *project* – the name of the project that this master represents.
- *tasks_waiting* – the number of tasks currently waiting in the master queue.
- *tasks_running* – the number of tasks running on the workers.
- *total_workers* – the number of connected workers.
- *capacity* – the estimated capacity of this master.
- *workers_by_pool* – the number of workers this master gets from each worker pool.

The "project" field allows a worker to identify a master by a name string and the "hostname" and "port" fields tells the end point that a worker should connect to. The "tasks_waiting", "tasks_running", "total_workers", and "capacity" fields help the worker pool to make proper worker distribution decisions and are also served for display purposes (the user can query the status of their masters).

The "workers_by_pool" field is useful when a master is served by multiple worker pools and non-pool-controlled workers. With this field, a worker pool knows exactly how many workers a master has received from itself. This prevents a worker pool from allocating too many workers to a master when a part of the master's worker needs has been satisfied by other worker providers. For example, some of a master's workers may come from the worker submit utilities provided by Work Queue, such as *condor_submit_workers* and *sge_submit_workers*, some other workers may come from individually started worker processes (by running the *worker* program directly), and some other workers may come from other worker pools.

An example of a "workers_by_pool" field is as follows:

```
workers_by_pool: ws1.nd.edu-100:300,ws2.nd.edu-101:500,unmanaged:50
```

4.4.2 Basic Distribution Decision

The goal of the worker distribution decision is to provide each master with as many workers as they need without violating the worker pool policy. The decision specifies the number of workers this worker pool would like to provide to each master. The following line is an example of a worker distribution decision made based on the previous worker pool policy example:

```
distribution: proj1:500,proj2-A:500,proj2-B:1000
```

For the simplicity of discussion, we refer to a master with its project name in the rest of the paper. The above distribution decision line states that the worker pool should provide 500 workers to proj1, 500 workers to proj2-A, and 1000 workers to proj2-B. Note that any of the project name appeared in the decision line can be matched to a project name regular expression listed in the worker pool policy.

The number of workers that a worker pool would decide to provide to a master is derived from two pieces of information: the user defined policy and every matched master's runtime status. The decision making process starts with calculating the maximum number of workers each master would need from it by using equation 4.6. In this equation, W_{max_needed} represents the maximum number of workers that this master would need from this worker pool. W_{still_needed} is the extra number of workers the master needs and $W_{received}$ is the number of workers the master has already received from the worker pool that is making this decision. The value of $W_{received}$ can be directly extracted from the workers_by_pool field in the master's status advertisement. The W_{still_need}, however, involves more complicated calculation on multiple fields in the master's status.

$$W_{max\_needed} = W_{still\_needed} + W_{received} \tag{4.6}$$

When a master's capacity is not reported, the extra number of workers this master still needs (W_{still_needed}) can be calculated with equation 4.7. In this equation, $T_{waiting}$ is the number of waiting tasks on the master and this number represents the maximum possible number of more workers that the master would need because a single task is the minimal task scheduling unit.

$$W_{still\_needed} = T_{waiting} \tag{4.7}$$

When the master capacity is reported, because a master does not need more workers than its own capacity, W_{still_needed} can be computed using equation 4.8. In this equation, *capacity* is the master's reported capacity and $W_{connected}$ is the number of workers that the master has already received, regardless the source. $capacity - W_{connected}$ is the extra number of workers that the master needs to fill its capacity. The MIN function is used because $T_{waiting}$ might be less than the value of $capacity - W_{connected}$ during the execution (e.g. at the end of the master's execution), in which case adding more workers than $T_{waiting}$ would definitely result in resource waste. Also note that W_{still_needed} is never less than zero. This guarantees that the decision on a master will not be reduced solely due to the decline in its capacity, which helps prevent system oscillation as will be discussed in section 4.6.

$$W_{still\_needed} = \min(\max(0, capacity - W_{connected}), T_{waiting}) \tag{4.8}$$

Once the W_{max_needed} is determined for a master, the worker pool compares this value with the master's $W_{default\_max}$ value to determine whether the W_{max_needed} value can be used as the final decision for this master. Before proceeding to the final decision making process, we illustrate how the $W_{default\_max}$ value is computed for each master.

The $W_{default\_max}$ value of a master represents the default maximum number of workers that the worker policy allows to provide to this master. It can be calculated using equation 4.9. $W_{default\_max\_in\_policy}$ is the value on the right side of an assignment whose left side (a regular expression) matches the master's project name. $N_{matched\_masters}$ is the total number of masters that match the assignment's project name regular expression. And R_{scale} can be computed from equation 4.10 where W_{total_max} corresponds to the max_worker field in the worker pool policy. We use the previous worker pool policy as an example to illustrate how the $W_{default\_max}$ is computed for assignment proj2.*=1500. With the given policy, the R_{scale} value is 1, which is result of (500 + 1500) / 2000. If there are three masters – proj2-A, proj2-B, and proj2-C. Then for any of these masters, the corresponding $W_{default\_max}$ value would be 500, which is the result of 1500 divided by 3.

$$W_{default\_max} = \frac{R_{scale} * W_{default\_max\_in\_policy}}{N_{matched_masters}} \tag{4.9}$$

$$R_{scale} = \frac{W_{total\_max}}{\sum W_{default\_max\_in\_policy}} \tag{4.10}$$

As can be inferred, the relation between the $W_{default\_max}$ values for each master and the policy defined max_workers value can be summarized in equation 4.11. Because of this auto-scaling property, the user is free to assign any number to the default maximum worker fields in the policy and does not need to guarantee that the sum of the assigned numbers equates the value of the max_workers field.

$$W_{total\_max} = \sum W_{default\_max} \tag{4.11}$$

Now that the W_{max_needed} and the $W_{default\_max}$ values are obtained for each master, we continue to the final decision making process. For any master, if its W_{max_needed} value is less than its $W_{default\_max}$ value, then the W_{max_needed} becomes the final decision for this master. If not, which means this master can take advantage of more workers than its $W_{default\_max}$, then the worker pool may give more workers than this master's $W_{default\_max}$ if some other masters need less than their own $W_{default\_max}$ values.

The previous process calculates a potential decision (W_{max_needed}) for every matched masters, and some of those decisions can be already determined as final. We refer to those masters whose decisions are not final as hungry masters and the other masters as full masters. The goal of the remaining process is to finalize the decisions for those hungry masters.

The remaining process runs in a loop until the decisions for all the masters are finalized. In each iteration, the maximum number of workers needed by all the hungry workers is computed with equation 4.12 and the number of undecided workers is computed with equation 4.13. In equation 4.13, $W_{decided}$ is the sum of decisions of those full masters. If $W_{total\_max\_needed}$ is less than $W_{undecided}$, which means the worker pool is able to satisfy all the remaining hungry masters' needs, then for each remaining hungry master, set its final decision to its W_{max_needed} and mark it as a full master.

$$W_{total\_max\_needed} = \sum_{hungry} W_{max\_needed} \tag{4.12}$$

$$W_{undecided} = W_{total\_max} - W_{decided} \tag{4.13}$$

If $W_{total\_max\_needed}$ is greater than $W_{undecided}$, then a potential decision for each hungry master with equation 4.14. R_{weight} represents the general proportion of workers that the user wishes to allocate for this master as defined in the policy, which can be computed with equation 4.15. If a master's potential decision is greater than its W_{max_needed} value, then the W_{max_needed} is used as the final decision for this master. If none of the master's potential decision is greater than its corresponding W_{max_needed} in this iteration, then all the remaining hungry masters are marked as full masters and their final decisions are set to their $W_{potential\_decison}$.

$$W_{potential\_decision} = W_{undecide} * R_{weight} \tag{4.14}$$

$$R_{weight} = \frac{W_{default\_max}}{\sum_{hungry} W_{default\_max}} \tag{4.15}$$

Now we examine a concrete example of what distribution decision will be made in a certain situation to consolidate the understanding of the calculations. Let the name of the worker pool be ws1.nd.edu-100. The worker pool policy is the same as in the previous examples, which is:

```
max_workers: 2000

distribution: proj1=500, proj2.*=1500
```

Assume there are three masters: proj1, proj2-A, and proj2-B. The masters' status that is relevant to the worker distribution decision is listed as below:

```
name: proj1

tasks_waiting: 2000

total_workers: 200

capacity: 1100

workers_by_pool: ws1.nd.edu-100:100,unmanaged:100


name: proj2-A

tasks_waiting: 1000

total_workers: 0

capacity: 0

workers_by_pool: N/A


name: proj2-B

tasks_waiting: 300

total_workers: 100

capacity: 700

workers_by_pool: ws2.nd.edu-101:100
```

First, the W_{max_needed} is calculated for each master, and the results are 1000 for proj1, 1000 for proj2-A, and 200 for proj2-B. For proj1, the master capacity is 1100, $W_{connected}$ is 200, W_{still_needed} is 900 (1100 - 200). And because the $W_{received}$ (from worker pool ws1.nd.edu-100) value is 100, the W_{max_needed} is 1000, which is the result of 900 + 100. For proj2-A, since the capacity is not reported and no other workers are connected to the master, the W_{max_needed} value equals its $W_{tasks\_waiting}$ value. For proj2-B, because there are only 300 tasks left and one other worker pool has already provided 100 workers to this master, the W_{max_needed} is the value of 300 less 100.

For $W_{default\_max}$ values, proj1's is 500, proj2-A's is 750, and proj2-B is also 750. Note that the R_{scale} value in this example is 1, which is the result of (500 + 1500)/2000. Thus the $W_{default\_max}$ for the proj2.* masters is derived from 1500 divided by 2 (the number of matched masters). For proj2-B, because its W_{max_needed} is less than 750, 100 becomes the final decision for proj2-B. The decisions for proj1 and proj2-A, however, are not finalized in yet because they can use more workers than their $W_{default\_max}$.

Then the worker pool enters the iterative process to finalize the decisions for the remaining two hungry masters. The $W_{total\_max\_needed}$ is the sum of the two hungry masters' W_{max_needed}, which is 2000. The $W_{undecided}$ is 1800 because only 200 workers are in the final decision and the worker pool can allocate at most 2000 workers. Because $W_{total\_max\_needed}$ is greater than $W_{undecided}$, the worker pool computes a potential decision, using equation 4.14, for each master based on their weights (R_{weight}) as defined in the worker pool policy, and the results are:

proj1: $W_{potential\_decision}$=720, W_{max_needed}=1000
proj2-A: $W_{potential\_decision}$=1080, W_{max_needed}=1000

As can be seen, proj2-A's $W_{potential\_decison}$ is greater than its W_{max_needed}, which means the worker pool could provide 1080 workers to proj2-A based on the user defined policy but the master does not need that many. Thus 1000 becomes the final decision for proj2-A and proj2-A is marked as a full master. This ends the current iteration and the worker pool proceeds to the next iteration. The $W_{undecided}$ now becomes 800 because 200 + 1000 workers has already been decided, and $W_{total\_max\_needed}$ becomes 1000, which is the W_{max_needed} of the only left master – proj1. This time, $W_{total\_max\_needed}$ is still greater than $W_{undecided}$, Thus, a new potential decision computer is for proj1 and the result is 800. Note that R_{weight} value for proj1 has been changed to 1. The new potential decision made in this iteration with the corresponding W_{max_needed} value are listed below:

proj1: $W_{potential\_decision}$=800, W_{max_needed}=1000

Since the W_{max_needed} for proj1 is still 1000, none of the hungry master's potential decision is greater than its corresponding W_{max_needed}. Thus the worker pool mark proj1 as full masters and 800 (the $W_{potential\_decision}$ of proj1) becomes the final decision for proj1. At this point, no hungry master exists and the decision making process ends. A new distribution decision needs to be sent to the catalog server so that the relevant masters and workers can obtain it. The decision update to the catalog server is the name-value pair format. The following fields would be sent to the catalog server as in the :

```
name: ws1.nd.edu-100
decision: proj1:800, proj2-A:1000, proj2-B:200
```

The name field contains a unique name for the worker pool, which consists of the machine's hostname and the worker pool process's process id. The decision field is a string that includes the decision for every matched master. The worker distribution decision affects the behavior of the participating masters and workers, the next section introduce how these components worker together to enforce a certain worker distribution.

4.4.3 Worker Pool Policy

A worker pool manages the worker resources on behalf of the user. By starting a worker pool, the user authorize the worker pool to request resources from the underlying resource management system with his or her credentials and run workers the allocated resources. Because the users own the requested resources, the users can define rules on how the worker pool should manage their resources, such as the maximum amount of workers that the worker pool is allowed to request and the number of workers should be assigned to each of user's applications. The collection of these user defined rules forms the worker pool policy.

The worker pool policy is specified in the form of name-value pairs in a file. The location of the policy file needs to be specified when starting a worker pool. It is also possible for a user to modify the policy file while a worker pool is running. A worker pool can be instructed to adopt an updated worker pool policy file at runtime. The following list shows the basic and required fields in a worker pool policy file:

- *max_workers*: the maximum number of workers that the worker pool can allocate.
- *distribution*: a list of projects and the default maximum amount of workers that can be assigned to each of them.

The $max\_workers$ field limits the maximum amount of resources that the worker pool can request for. At any time, the number of workers that the worker pool is maintaining should be no more the value of $max\_workers$. We illustrate the *distribution* field with the following fragment from a sample worker pool policy file:

```
max_workers: 2000
distribution: proj1=500, proj2.*=1500
```

The *distribution* field contains a series of assignments separated by commas. In each assignment, the left side of the assignment sign is a regular expression for project name matching and the right side is the default maximum number of workers that can be assigned to the matched projects. Assignment $proj1 = 500$ defines that the master of proj1 can get at most 500 workers from the pool by default. Assignment $proj2.* = 1500$ defines that, for all the masters whose project name matches the regular expression $proj2.*$ (e.g. proj2-A, proj2-B), the sum of workers that they can receive from the worker pool should be no greater than 1500 by default.

The default maximum worker limit for a master is not an absolute upper-limit. The actual number of workers received by a master could exceed the predefined default maximum under certain circumstances. However, the default maximum worker limits do imply the different proportions of workers that the user wishes to distribute to different projects. In the above policy excerpt, when 2000 workers are allocated (this happens when all the masters together needs more than 2000 workers), the user would want 25% of them goes to the proj1 master and the remaining 75% goes to the masters whose project names match the regular expression $proj2.*$.

The *distribution* field is the basis for providing fairness (of resource provisioning) among multiple projects when the resources are limited compared to the needs. When the total amount of workers needed by all the masters are less than the value of $max\_workers$, the worker pool can simply satisfy every master's needs. But in the reverse case, the worker pool complies with the user desired proportion for each master based on the *distribution* field. Note that a particular master's worker needs might be less than the user's allowed proportion. In this case, the extra amount of workers not needed by that master can be assigned to other masters that needs more than their allowed proportions.

The following list shows the optional fields that can be defined in the worker pool policy to further constrain the behavior the worker pool.

- *max_change (per minute)*: maximum decision change per minute.
- *default_capacity*: assume a default capacity for a master when its capacity is unknown.
- *billing_cycle*: the time period that each resource unit is billed upon.

After a pool decision is made, the worker pool knows how many workers it should maintain, which is the sum of workers that it has decided to assign to each master. We refer to this sum as the pool decision. When the *max_change* field is specified, that change speed of the pool decision would be no greater than the value of the *max_change* field. When the *default_capacity* field is specified, the worker pool would use the value of *default_capacity* as the capacity for those masters that have not reported their capacities yet.

The *billing_cycle* field is designed to accommodate the billing model of the commercial clouds where computing resources are charged per time period. For example, the Amazon EC2 platform charges their resource usage based on a one-hour boundary. By default, if a worker has not been able to find any masters to serve within a certain period of time, it will time out and terminate itself. But if the *billing_cycle* option is specified, the worker might not terminate itself after the normal timeout period as long as there is time left until the next billing boundary. For example, assuming the default timeout for a worker is 2 minutes and the *billing_cycle* is 20 minutes, if a worker has worked for a master for 5 minutes and has no more work to do, normally, the worker would automatically terminate itself at the 7 minute point. But with the *billing_cycle* specified, the worker would keep finding masters to serve until it is close to the 20 minute point.

4.4.4 Policy Adjusted Distribution Decision

In this subsection, we first show how the pool decision is made with different policies. Then introduce decision enforcement mechanism in the context of our elastic application framework, which involves the coordinations among masters, workers, and worker pools through a catalog server. Finally, we show the evaluations of the worker pool resource management performance with different predefined policies. For each policy, we show the worker pool performance on different patterns of workloads.

The goal of the worker distribution decision is to provide each master with as many workers as they need without violating the worker pool policy. The decision specifies the number of workers this worker pool would like to provide to each master. It is derived from two pieces of information: the worker pool policy and every matched master's runtime status. We refer to the total number of workers that a worker pool decides to maintain as a decision. Now we look at the general criteria that the worker pool use to make a decision and how we can combine them to form distinct policies. The distinct policies are the basis for evaluating the effects of different criteria on the resource allocation performance.

The most basic decision is the sum of the remaining tasks, as shown in equation 4.16. In this equation, $T_{waiting}$ is the total number of tasks that are waiting to be executed and $T_{running}$ is the total number of tasks that are currently being executed. The sum of $T_{waiting}$ and $T_{running}$ represents the total number of tasks that have been submitted but have not run to completion. Because the goal is to complete all the unfinished tasks and each task can only be run on one worker at a time, providing more workers than the sum is obviously unnecessary.

$$D_1 = T_{waiting} + T_{running} \tag{4.16}$$

To avoid allocating too many resources too quickly, we can limit the decision change speed as shown in equation 4.17. $D_{previous}$ is the value of the previously made decision. And Δ is the production of the elapsed time since the previous decision was made and the value of the *max_change* field in the policy. Limiting the decision change speed results in a more conservative worker allocation behavior. This is especially useful when newly started masters have not been able to report their capacities yet (the capacity estimation requires a certain number of tasks being completed). If newly started master's actual capacity is less than its unfinished tasks, a policy without limiting the decision change might make the worker pool allocate more workers to a master than it needed, which would result in resource waste as the master can not use them efficiently.

$$D_2 = \min(D_1, D_{previous} + \Delta) \tag{4.17}$$

To respect the number of workers that the master can consume efficiently, the decision should be no greater than the master's reported capacity, as shown in equation 4.18. Here C stands for the reported master's capacity. If a master has not reported its capacity, ∞ would be used as the value of C. As described in section 4.3, the reported capacity represents the master's estimate of how many workers it can handle efficiently. Adding more workers to a master beyond its capacity would not result in performance gain. By taking the reported capacity into consideration, the worker pool can avoid the allocation of those workers that do not bring in benefits.

$$D_3 = \min(D_1, C), C = \infty \text{ if unknown} \tag{4.18}$$

With decision D_3, the sum of $T_{waiting}$ and $T_{running}$ would dominate the decision when the capacity has not been reported. This could be undesirable because the worker pool would allocate as many workers to match the sum but the sum could be actually much greater than the capacity. One approach to reduce the startup waste is to assign a default capacity value to the masters that have not reported their capacities. As shown in equation 4.19, C_0, which is the value of the $default\_capacity$ field in the policy, would be used as the default capacity for those masters that do not know their capacities yet.

$$D_4 = \min(D_1, C), C = C_0 \text{ if unknown} \tag{4.19}$$

The *max_change* option can help reduce the startup waste as well because it prevents the worker pool from suddenly allocating too many workers when new workloads join with many unfinished tasks. To observe the startup waste reduction effect of the *max_change* option when capacity consideration is turned on, we add the *max_change* option to the D_3 policy and get a new policy setup – D_5, as shown in equation 4.20. With D_5, in addition to the advantage of startup waste reduction, it also protects the worker pool from making drastically different decisions in a short period of time when the estimated capacities are not stable.

$$D_5 = \min(D_3, D_{previous} + \Delta), C = \infty \text{ if unknown} \tag{4.20}$$

Considering master's reported capacity, assuming a default capacity for newly started masters, and limiting the change speed of the decisions all make the resource allocation process more conservative and, in many cases, more reasonable. But each of these options are contributing to the conservativeness from a different perspective. To evaluate the behavior of the resource allocation when all these options are stacked together, we use policy D_6, as shown in equation 4.21.

$$D_6 = \min(D_3, D_{previous} + \Delta), C = C_0 \text{ if unknown} \tag{4.21}$$

The final policy configuration – D_7 in equation 4.22, makes the worker pool behave the same as under D_6 when calculating the decision for each master. However, in addition to D_6, D_7 has the *billing_cycle* option turned on, which enforces the workers to terminate themselves only when their lifetimes are close to the multiples of *billing_cycle*. D_7 makes the workers last for longer time without additional cost on computing resource platforms that are billed upon time periods. This allows a master to connect new workers faster when it appears during some workers' extended lifetime period as the worker pool does not need to request new resources to run those worker on.

$$D_7 = D_6, \text{ enforces worker termination boundary} \tag{4.22}$$

4.5 Decision Enforcement

The decision enforcement mechanism is decentralized. Every participating component, namely the masters, the workers, and the worker pools, communicates with the catalog server independently. No central process is coordinating the actions of the participants. The interactions between the participants and the catalog server are shown in Figure 4.1. The worker pool queries the masters' status from the catalog server periodically and sends a decision back when matched masters are found. The worker pool is allocates new workers when the current decision is greater than the previous one. The master's interactions with the catalog server includes sending its own status and retrieving distribution decision, both periodically. The worker queries the catalog server for distribution decision as well, but only at when it needs to find a new master to worker for. The remaining subsections specify the behaviors of each participant in details.

4.5.1 Master

In addition to advertising its own status to the catalog server periodically, the master queries the catalog server periodically to obtain the latest worker distribution decisions. If a worker pool is responsible for supplying workers to a master, that worker pool's decision would specify how many workers it has decided to provide to this master. The master is responsible for guaranteeing the number of workers it receives from that worker pool does not exceed the decision. Because a worker would notify the master which worker pool it is from upon connection, the master is able to keep track of the number of workers it has received from each worker pool. If the number of workers from a worker pool exceeds the decision (this could happen because of the randomized master selection algorithm on the worker side, which will be introduced soon), the master releases the exceeded amount of workers from that pool and rejects future worker connection requests from that worker pool. Thus the worker pool's decision is enforced. Those released workers would immediately start searching for a new master to work for.

4.5.2 Worker

Whenever a worker needs to find a master to work for, it queries the catalog server to obtain a list of running masters as well as its worker pool's worker distribution decision. The worker then knows how many workers are already connected to each master from its worker pool and how many workers its worker pool decides to provide to each master. Based on these information, the worker calculates how many extra workers each master can still receive from its worker pool. The worker then applies a randomized algorithm to select a master to serve.

The randomized algorithm strives to make the probability of selecting a master equal to the proportion of that master's worker needs among all the masters. For example, if a worker pool is serving workers to two masters – proj1 and proj2, and the worker pool has decided that 100 workers should go to proj1 and 300 workers should go to proj2. If 50 workers from that worker pool are already connected to proj1 and 150 to proj2, a worker's probability of choosing proj1 to work for would be approximately 25%, which is the result of (100 - 50)/(300 - 150). Similarly, the probability of selecting proj2 would be approximately 75%. When a worker has finished serving a master, either because the master has completed all its tasks or the master decides to release that worker for decision enforcement purposes, the worker would repeat the above process to find a new master.

The worker's randomized master selection is the system's initial attempt to achieve a decided distribution of workers among the masters. In practice, the randomized algorithm may not deliver an accurate distribution in the first attempt, that is, more workers from a worker pool might choose to connect to a master than the worker pool has decided. In this case, the masters would actively release the extra amount of workers and those workers would each reselect a master to work for based on the latest masters' status. Eventually, the desired worker distribution will be accordant with the worker pool's decision.

4.5.3 Worker Pool

The worker pool has two main responsibilities: making the worker distribution decision and allocating workers when needed. As discussed in the previous sections, a worker distribution decision is made based on the user defined worker pool policy and the runtime status of the masters. While the worker pool is running, it queries the catalog server periodically to obtain the runtime status of the masters. Whenever matched masters are found, the worker pool would examine the masters' status and make a new worker distribution decision. The decision is sent to the catalog server immediately after it is made.

The other responsibility of the worker pool is to allocate and maintain a number of workers as its decision specifies. After a decision is made, the worker pool compares the new decision with the number of workers it is maintaining. If more workers are needed, the worker pool would request the extra amount of workers from the underlying resource management system. The worker pool constantly check with the resource management system to see if any of its workers has terminated (either because of resource failures or normal completion). Whenever the number of maintained workers falls below the decision amount, new workers will be allocated to fill the gap.

If the number of workers a worker pool is maintaining is greater than the decision amount, then when a worker terminates, the worker pool does need to do anything else. This happens when the masters' worker needs are decreasing, for example, when some of the masters are reaching completion or have already completed, or some of the masters are entering a lower capacity stage. When a master's worker needs is decreasing, it will disconnect some or all of its workers, either because the distribution decision requires or there is no more tasks to do. A disconnected worker will time out and terminate itself if it cannot find a master to work for within a certain amount of time. In this case, the worker pool would notice the worker's termination, however, because the amount of maintained workers is still greater than the total needs, the worker pool does not perform any extra actions.

To summarize, the worker pool is capable of scaling up the number of maintained workers when new masters begin to execute or existing masters can take advantage of more workers, as well as scaling down the maintained workers when the total worker needs from the masters are decreasing.

4.6 System Stability

The capacity management architecture can be viewed as a feedback control loop system[31]: the master measures the workers and tells the worker pool how many workers it wants; the worker pool submits the exact number of workers (i.e. controls the number of workers in the system); the submitted workers (once start running) are then measured by the master, which leads to updated worker demand of the master. The master is the plant as in the control theory feedback loop. The worker pool is the controller. And the masters' declared demand of workers is the system input. The stability in this system refers to the ability that the number of workers can converge to a stable amount under changes and does not oscillate around the stable state.

The capacity management system forms a negative feedback control loop by design, which produces stability. If the number of submitted workers is less than the masters' declared demand, the worker pool would add more workers into the system to match the demand. If the number of submitted workers is greater than the demand, however, the worker pool would never remove any worker from the system by itself. Instead, the extra workers would be disconnected by the masters whose new distribution decisions do not allow them to hold that many workers from the worker pool. And these disconnected extra workers would exit the system only if they could not find any suitable master to work for within a certain timeout. As such, the system always applies a negative feedback signal to regulate the number of workers to the declared demand, which forms the negative feedback loop.

Although negative feedback loops tend to reduce fluctuations in general, false negative feedback signals may cause the system to oscillate, if not handled carefully. The cache bounce effect introduced in section 4.3.4 may cause the reported capacity to fluctuate in an opposite direction to the actually trend of the worker demand, which creates a false feedback signal. Recall that in Figure 4.7 and 4.8, the estimated capacity first drops when new workers are added and then returns to the stable level as the new workers have the shared input data cached. Thus, in the system the estimated capacity could decline, though temporarily, while the actual capacity does not, or even increase (e.g. newly added workers are faster).

If the system removes workers immediately in response to the cache bounce capacity decline , then the system might fall into oscillations. The reason for the oscillation is that the removed workers would be added back into the system again when the capacity increases after the drop and the added workers would cause the cache bounce effect again, which triggers a new round of removing workers and adding back workers. As mentioned in section 4.4.2, the decision on a master is never reduced due to a decline in the capacity. Thus, the system will not remove already connected workers (or the master will not disconnect workers) in the case of a capacity decline. So, we can conclude that the false feedback, which signals a capacity decline while it is not, would not cause the system to oscillate.

The capacity management system is also capable of avoiding disastrous resource waste that could be caused by a bad software configuration. When running tasks remotely, the remote execution environment may not be configured as the task would expect. For example, a task may require the Python version to be above 3.0 while the remote machines only have Python 2.7 installed. Or, a user may accidentally forget to include a required file when describing the remote tasks. Without capacity management, a user may allocate hundreds or thousands of computing nodes and none of them could do useful work. And automatically retrying those failed tasks on them, as many of the workflow management systems would do, only exacerbates the waste. Bad configurations usually cause the task to fail very quickly, which leads to a very low capacity (very short execution time). Thus with capacity management, the system would have the opportunity to cut the resources according to the low capacity, which prevents great waste.

4.7 Evaluation

All the experiments described below are conducted on the Condor pool maintained at Notre Dame. To request a worker from the Condor pool, the worker pool submit the *worker* program as a Condor job. Because there is a delay in between when a Condor job is submitted and when a Condor job is executed, the masters will not see worker connections until a certain amount of time has elapsed after the workers has been requested for. We will see these effects in the evaluation results.

4.7.1 Experiment Setup

In order to evaluate the worker pool performance, we identify several distinct workload patterns and worker pool configurations and run experiments on all the combinations of them. The selected patterns of workloads are shown in the following list:

- *P1*: single uniform batch
- *P2*: multiple uniform batches
- *P3*: multiple non-uniform batches

- *P4*: random uniform batches
- *P5*: random non-uniform batches

Workload pattern P1 contains a single batch of 500 uniform tasks. Each task has 2 MB input, takes 15 seconds to execute, and generates 2 MB of output. Pattern P2 contains 5 batches of the same tasks as in P1 and one batch of tasks is submitted after every 400 seconds. Pattern P3 has 5 batches of tasks submitted at the same intervals as in P2, but the tasks between different batches are non-uniform. The input data sizes of the tasks in the 5 batches are 3 MB, 1 MB, 5 MB, 1 MB and 10 MB respectively and the output data sizes are 2 MB, 1 MB, 3 MB, 1 MB, and 2 MB. The execution times on these tasks are all 15 seconds. Note that the tasks within the same batch are still uniform in P3, which should result in more stable capacity estimates. Pattern P4 contains the same uniform tasks as in P1 but they are randomly put into 50 batches. Each batch contains from 1 to 100 tasks and the time between when two adjacent batches are submitted ranges from 1 to 50 seconds. Pattern P5 is similar to P4 except that the tasks are non-uniform across different batches. The P5 tasks, both the input and output data sizes range from 1 MB to 5 MB, and the task execution times range from 5 to 15 seconds.

We create different pool policies by adding or replacing constraints in the policy file in order to observe how the constraints affect the worker pool decisions over time. All the policies as the *max_workers* field set to 200. The following list shows the 7 distinct policy configurations used in our experiments:

- $D1 = T_{running} + T_{waiting}$
- $D2 = D1 + max\_change$
- $D3 = D1 + capacity$
- $D4 = D3 + default\_capacity$
- $D5 = D3 + max\_change$
- $D6 = D3 + default\_capacity + max\_change$

- $D7 = D6 + billing\_cycle$

Configuration D1 determines the worker needs by adding the $T_{running}$ and $T_{waiting}$ reported by that master. Under D1, the worker pool would ignore any master's capacity that is reported. D2 adds more constraints to D1 by limiting the change speed of pool decisions to 60 per minute. D3 forces the worker pool to consider the reported capacities while making decisions. The next four configurations all have the capacity consideration turned on. In addition to D3, D4 requires the worker pool to assume a default capacity for the masters that have not reported their capacities. D5 limits the change speed of pool decisions to 60 per minute in addition to D3. D6 has all the constraints: it has the capacity consideration turned on, the "max_change" set to 60 per minute, and the "default_capacity" set to 20. Finally, we add the "billing_cycle" constraint to D6 and get D7.

4.7.2 Results

Figure 4.9 and Figure 4.10 shows the runtime worker pool decisions and the number of running tasks over time for each experiment. Each column shows the results of the same workload with different pool policies and each row shows the impacts of the same work pool policy under different patterns of workloads. For each individual graph in two figures, the x-axis is time in seconds starting from 0, at which the worker pool and the workload are started. The y-axis is the worker counts. The "pool decision" curve shows the number of workers that the pool decides to maintain over time, which comes from the log of the worker pool. The "tasks running" curve shows the number of tasks that are being executed on the workers and this curves is generated from the log of the workload master.

For the row of pool configuration D1, the estimated capacities of different workloads over time are shown in the individual graphs. For the remaining graphs, the estimated capacity curves are omitted as they would be similar to those shown in the corresponding first-row graphs. For workload P1, we can see that capacity first increases as more tasks are being completed. This is because the tasks all share the same input data. This effect is the same as that in Figure 4.8. Workload P2's tasks also share the same input, and its estimated capacities are quite stable after is has reached its peak at around the 150 second point. For workload P3, we can see that estimated capacities have noticeable differences when different batches of tasks are being executed. This is as expected because the tasks between different batches are non-uniform. For workload P4 and P5, the capacity curves show much more fluctuations as the tasks are non-uniform.

We also summarize the sum of task execution times, the application turnaround times, the sum of the times spent on workers, and the consumed billing cycles in Table 4.2. The sum of task execution time does not include the data transfer time for each task, but it provides a lower bound for the amount of time that needs to be consumed on the resources. The sum of the worker time is the total time that we have spent on the resources for each experiment. The billing cycles are calculated as if the resources on billed upon a unit period of time and the unit period used in our experiments is 20 minutes. That is, even if a resource has been only allocated for 5 minutes, it would add 1 full cycle to the billing cycles.

Now we discuss the results of the homogeneous workload – P1 under different worker pool policies. This workload contains a total of 500 tasks. Under policy D1, the worker pool would allocated as many workers as the number of unfinished tasks without surpassing the policy defined *max_workers*. Thus when the worker pool discovered the new master from the catalog server, it decided to allocated 200 (the minimum value of *max_workers* and the number of unfinished tasks) workers to the master. 200 workers are requested from the underlying resource management system (in our case it is Condor) immediately, but we only start to see the numbers of running tasks after approximately a minute. This is because of the delay between when the resource is requested and when the resources is actually allocated and is ready for use. As can be seen, under D1, the worker pool allocated more workers than the application was able to use during its lifetime, which obviously resulted in a waste.

Under D2, the worker pool requests for workers gradually, as can be seen from the "pool decision" curve in Graph(D1, P1), however, it ended up allocating more than the master needed as the runtime number of unfinished tasks kept staying beyond the number of workers that the master actually needs. For policy D3, although the capacity is taken into consideration, it did not take effect until it has been reported. And before the capacity was reported, the worker pool has already requested for too many workers. Policy P4 adds the $default\_capacity$ into P3. It avoided the great waste at the beginning, but with some sacrifice on the application performance (as can be seen from the turnaround times in Table 4.2) because the $default\_capacity$ in our case is lower than the application's actual capacity. However, if a user has prior knowledge to the capacity of an application, he or she could set the $default\_max$ accordingly to minimize the sacrifice on performance. Policy D5 took away the $default\_capacity$ constraint and adds the limit on decision change to D5. Compared to policy D3, the waste at the beginning is also greatly reduced, although not as much as with D4. But the application performance on D5 is improved over D4 as D5 adds workers more aggressively than D4. Policy D6 uses all the options – the capacity consideration, the maximum decision change limit, and the default capacity. It is the most conservative resource allocation policy of all and shows the minimum resource waste, as can be seem from Graph(D7, P1). Policy D7 is the same as D6 except that it enforces the termination boundary on the workers. The runtime "pool decision" and "tasks running" curves do not show much difference from those in the D6 graph, but the consumed cycles would be reduced with policy D7 when the workloads run over a single billing cycle. The reduced billing cycle effect more noticeable on workload P2 and P3 in Table 4.2 when you compare the billing cycles of policy D6 and D7.

For workload patterns other than P1, we can observe the similar pool policy effects as those seen in the workload P1 results. In general, the more conservative the policy is, the more resources waste is avoided. Conservatively allocating resources does sometimes result in sacrifice of performance. For example, in workload P1, the turnaround time on D4 is much greater than that on D1. This is because the workers that the master needs are not provided in the earliest possible time with a conservative resource allocation policy. However, when the capacity is taken into account, the sacrifice of performance is minimized and even eliminated. As can be seen from the turnaround times on the workloads other than P1, the application performances are actually better on D6 (more conservative) than D1. This is because the overheads that come from the master communicating with too many workers shadows the performance gain from providing enough workers at the earliest times.

Another interesting observation is that, although the capacity estimation is not specifically designed for workloads with non-uniform tasks, policy D6 was able to achieve the similar application performances on workload P4 and P5 as the aggressive allocation policy D1. And the total time consumed on the allocated resources and the billing cycles used are greatly reduced with policy D6 and D7.

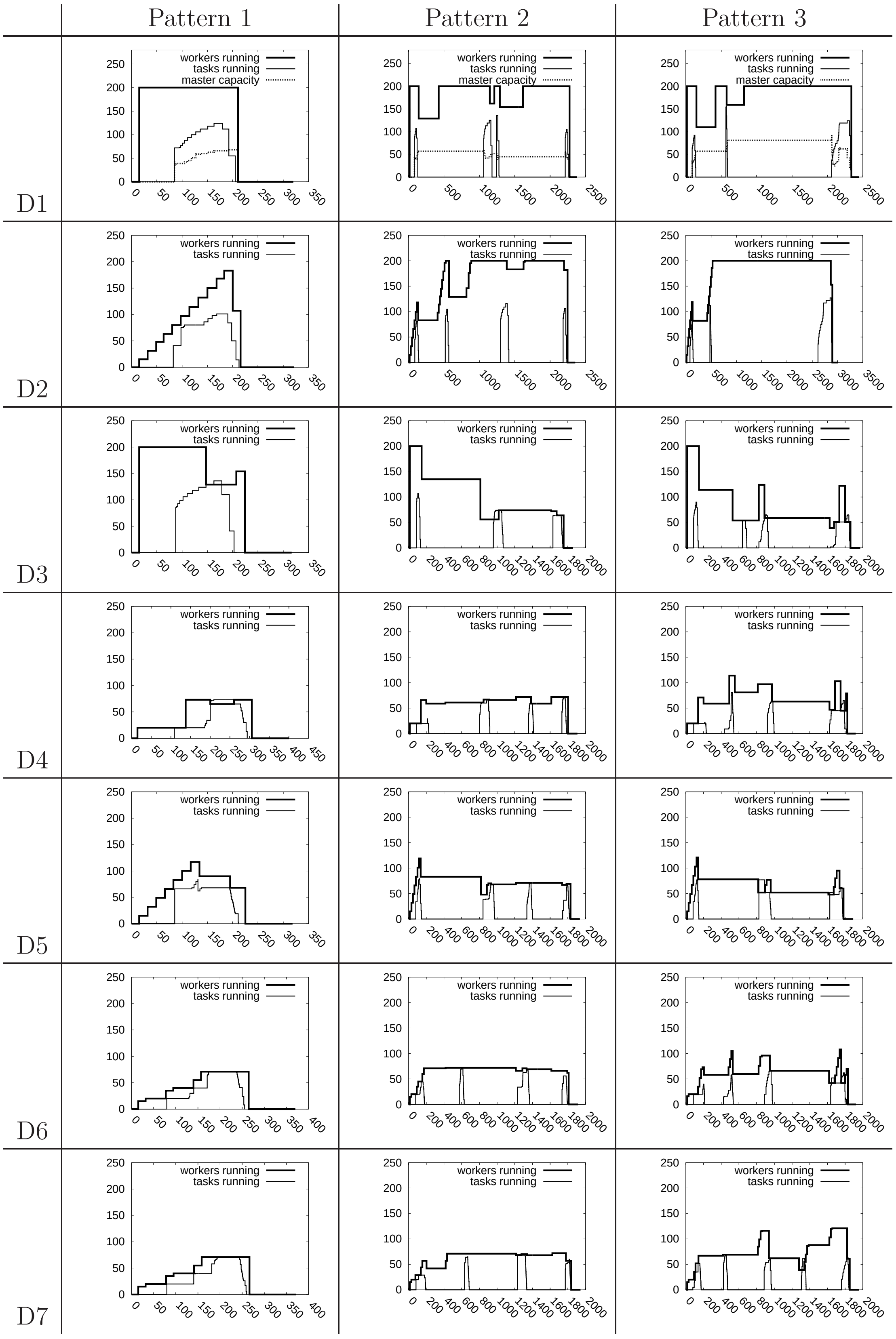


Figure 4.9: Runtime Pool Decision, Tasks Running, and Estimated Capacity for Workload Pattern 1, 2, and 3.

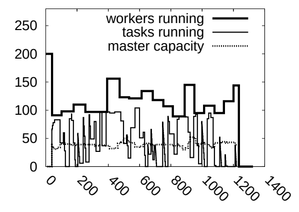
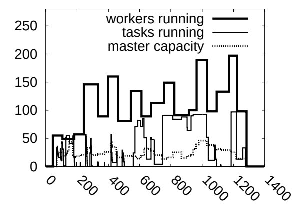
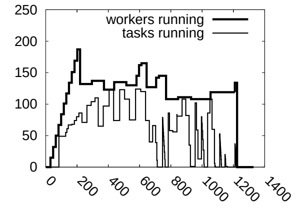
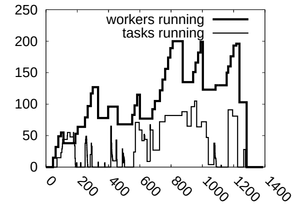
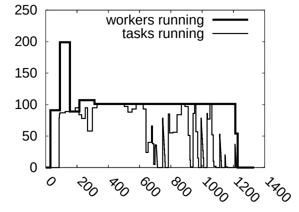
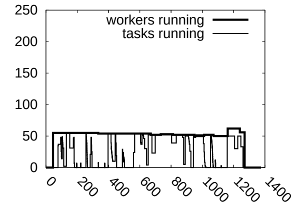
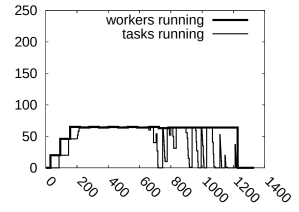
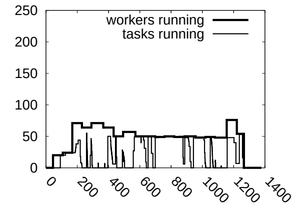
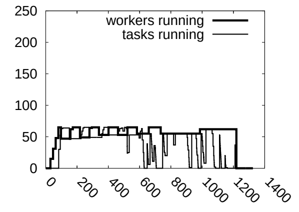
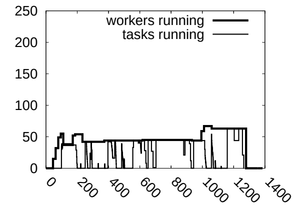
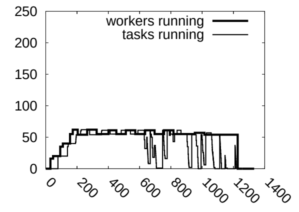
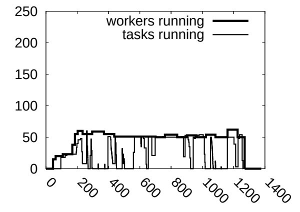
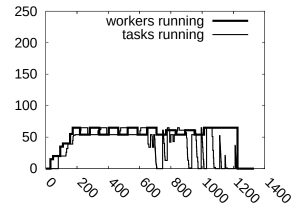
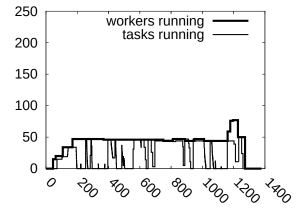
	Pattern 4	Pattern 5
D1		
D2		
D3		
D4		
D5		
D6		
D7		

Figure 4.10: Runtime Pool Decision, Tasks Running, and Estimated Capacity for Workload Pattern 4 and 5.

TABLE 4.2

TURNAROUND TIME, TOTAL WORKER TIME, AND BILLING CYCLES CONSUMED FOR EACH EXPERIMENT

		P1	P2	P3	P4	P5
	Sum Exe Time (sec)	7500	15000	15000	41985	20505
	Turnaround Time (sec)	209	2264	2336	1219	1276
D1	Sum Worker Time (sec)	57024	361214	398229	174295	183987
	Billing Cycles (count)	232	1075	1120	448	422
	Turnaround Time (sec)	213	2236	2891	1217	1276
D2	Sum Worker Time (sec)	39396	342048	502700	167661	179125
	Billing Cycles (count)	183	1079	1325	317	417
	Turnaround Time (sec)	208	1742	1856	1217	1261
D3	Sum Worker Time (sec)	55665	150326	145712	139886	73050
	Billing Cycles (count)	200	827	837	253	117
	Turnaround Time (sec)	294	1787	1814	1218	1262
D4	Sum Worker Time (sec)	21053	98295	109239	78412	74304
	Billing Cycles (count)	73	368	577	108	155
	Turnaround Time (sec)	212	1817	1778	1218	1265
D5	Sum Worker Time (sec)	27315	114680	103298	78875	67349
	Billing Cycles (count)	117	680	554	131	123
	Turnaround Time (sec)	255	1795	1816	1217	1259
D6	Sum Worker Time (sec)	19595	111012	113029	73397	69411
	Billing Cycles (count)	71	567	619	98	108
	Turnaround Time (sec)	260	1836	1845	1218	1260
D7	Sum Worker Time (sec)	19651	103682	136270	79553	66189
	Billing Cycles (count)	71	264	405	111	124

CHAPTER 5

APPLICATIONS

The tools described in this dissertation have been used to harness distributed computing resources for real world scientific applications. The abstractions developed for static workloads have been applied in applications from bioinformatics and economics. The capacity management architecture is implemented in the Work Queue framework. We show two production systems, one from bioinformatics and one from molecular modeling, that have used the capacity management feature in Work Queue to manage resources for their dynamic workloads.

5.1 Bioinformatics

5.1.1 Static Workloads

The abstractions introduce in Chapter 3 have been applied in several bioinformatics applications. Sequence alignment is one of the most important tasks in bioinformatics and is used in a variety of applications. Common variants of pairwise sequence alignment can be solved using dynamic programming [72] and each requires time proportional to the product of the two sequences considered. Prior parallel implementations have been motivated by either the need to compare a single pair of large sequences [81] or the need to compare many small sequences [75] for tasks such as phylogenetic inference and genome assembly. Previous algorithms have implemented the wavefront problem on dedicated clusters and parallel architectures such as the Cell [92]. Our implementation achieves similar speedups, but requires only sequential coding, and can execute on unreliable, loosely coupled machines.

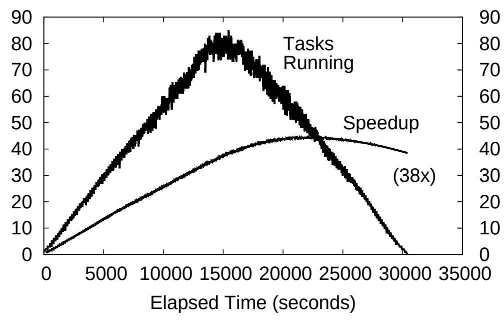


Figure 5.1: 100×100 Wavefront in Bioinformatics
A timeline of a 100×100 Wavefront problem implementing sequence alignment running on non-dedicated multicore Condor pool. 80 cores were available at the peak of the execution. An overall speedup of 38X is achieved, the maximum possible is 50X.

In less than a day, we wrote a single process function in 156 lines of C++ that performed alignment on a substring and propagated the required data for later steps. Distributed sequence alignment was then tested on two large bacteria genomes using wavefront: a non-virulent lab strain of Anthrax (Bacillus anthracis str. Ames; Genbank NC_003997) and its virulent ancestor strain (Bacillus anthracis str. 'Ames Ancestor'; Genbank NC_007530). Each genome is approximately 5.3 million characters long, and the score of an optimal suffix-prefix alignment was computed using only linear-space. An actual alignment (i.e., the path through the dynamic programming matrix) is also attainable based on the divide-and-conquer Hirschberg technique [92], which requires twice as much computation and a more complicated strategy.

Figure 5.1 shows a timeline of this alignment running using a 100×100 partition of the problem. Each task takes about 117 seconds to run on a 1GHz CPU. On the Condor pool, a maximum of 80 tasks running simultaneously was achieved. The overall runtime was reduced from 13 days sequential to 8.3 hours with a speedup of 38X out of the maximum possible 50X.

We also explored the application of a heuristic for bioinformatics problems similar to sequence alignment. SSAHA (Sequence Search and Alignment by Hashing

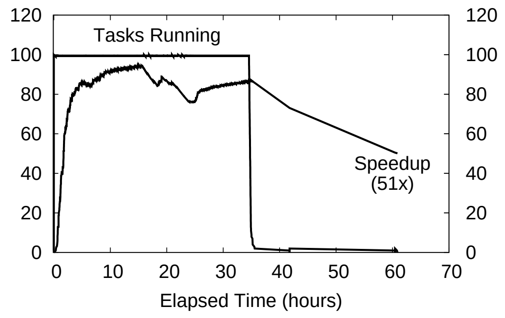


Figure 5.2: Makeflow without Fast Abort
A timeline of SSAHA execution on 100 simultaneous workers without Fast Abort. As can be seen, the long tail is almost as long as the peak computation period.

Algorithm) [74] is a bioinformatics tool designed to map one set of genetic data onto another set of data. SSAHA is very similar to the popular bioinformatics tool BLAST [7] because it creates a hash table for a set of subject sequences to speed up the search of query sequences for matches. Unlike BLAST, SSAHA computes the complete mapping and therefore can be used to discover detailed differences between sequences and individuals. SSAHA is a publicly available sequential application. Our implementation involves running the sequential application many times in parallel using the Makeflow and Work Queue abstractions. This allows us to harness the Condor pool to complete our computation in a reasonable time.

Our implementation mapped 11.5 million sequences consisting of 11 billion bases onto the genome *Sorghum bicolor* [77] (738.5 million bases). This is a large bioinformatics workload with the majority of execution time for each job dedicated to mapping the queries and a small portion dedicated to generating hash tables. The abstraction split a large sequential execution into nearly 2300 smaller sequential computations that were run in parallel on workers submitted to our Condor pool. Figure 5.2 shows the execution of this job on a maximum of 100 simultaneous workers without Fast Abort. There is an extremely prominent long-tail effect that nearly doubles the total execution time. Figure 5.3 shows the same workload run with fast

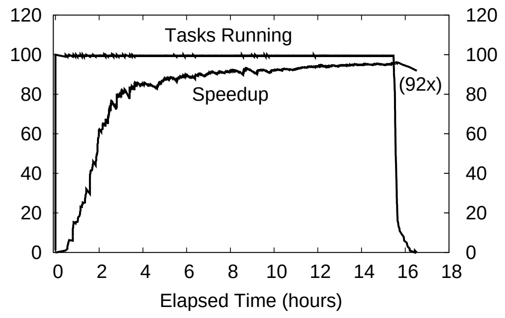


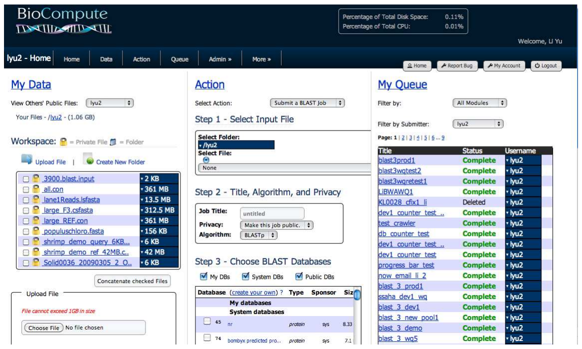
Figure 5.3: Makeflow with Fast Abort
A timeline of SSAHA execution on 100 simultaneous workers with Fast Abort. Compared to the above figure, the tail is mostly eliminated.

abort enabled, which nearly eliminated the long-tail effect and more than halved our total run time. The implementation using Fast Abort required 16 hours of runtime compared to the sequential runtime of 65 days with a total speedup of 92X.

5.1.2 Dynamic Workloads

The capacity management architecture has been implemented in the Biocompute system [20] – a production system that facilitates biology researchers to run select bioinformatics applications on campus distributed computing resources. Running a bioinformatics workload is as easy as selecting a desired bioinformatics application such as BLAST [7], answering a few application specific questions, and then clicking a submit button. The web user interface of the Biocompute system is shown in Figure 5.4. The web portal then generates a Makeflow script which describes a workflow according to the user's answers and the Makeflow script is executed by the Makeflow engine using the Work Queue framework. Upon job completion, the users are able to view and share their data, job information, and analytical results within the web portal. As application specific code is confined to the web portal, the backend is completely generic and its improvements are broadly applicable.

Demand for the computing resources provided by Biocompute is primarily met



Percentage of Total Disk Space: 0.11%
Percentage of Total CPU: 0.01%
Welcome, Li Yu

My Data

View Others' Public Files: lyu2

Your Files - /lyu2 - (1.06 GB)

Workspace: = Private File = Folder

3900.blast.input	2 KB
all.con	361 MB
lane1Reads.lsfasta	13.5 MB
large F3.csfasta	312.5 MB
large REF.con	361 MB
populuschloro.fasta	156 KB
shrimp demo query 6KB...	6 KB
shrimp demo ref 42MB.c..	42 MB
Solid0036 20090305 2 O..	6 KB

Upload File

File cannot exceed 1GB in size

No file chosen

Action

Select Action: Submit a BLAST Job

Step 1 - Select Input File

Select Folder: /lyu2
Select File: None

Step 2 - Title, Algorithm, and Privacy

Job Title: untitled
Privacy: Make this job public.
Algorithm: BLASTp

Step 3 - Choose BLAST Databases

☑ My DBs ☑ System DBs ☑ Public DBs

	Database (create your own) ?	Type	Sponsor	Siz
	My databases			
	System databases			
45	nr	protein	sys	8.33
74	bombyx predicted pro...	protein	sys	7.1

My Queue

Filter by: All Modules

Filter by Submitter: lyu2

Page: 1 | 2 | 3 | 4 | 5 | 6 ... 9

Title	Status	Username
blast3prod1	Complete	lyu2
blast3wqtest2	Complete	lyu2
blast3wqretest1	Complete	lyu2
LiBWAWQ1	Complete	lyu2
KL0028 cfix1 li	Deleted	lyu2
dev1 counter test ..	Complete	lyu2
test crawler	Complete	lyu2
db counter test	Complete	lyu2
dev1 counter test ..	Complete	lyu2
dev1 counter test	Complete	lyu2
progress bar test	Complete	lyu2
now email li 2	Complete	lyu2
blast 3 prod1	Complete	lyu2
ssaha dev1 wq	Complete	lyu2
blast 3 dev1	Complete	lyu2
blast 3 new pool1	Complete	lyu2
blast 3 demo	Complete	lyu2
blast 3 wq5	Complete	lyu2

Figure 5.4: Biocompute Web Portal

through the Condor batch system. This resource has contributed 62.8 CPU years since early 2010 with the majority of cycles shared among the top 30 users spanning the departments of biology, biochemistry, and computer science. The largest job on Biocompute–requiring 9.3 CPU years and involving more than a million independent tasks–completed in just 16 days.

The Biocompute system was originally configured to execute the generated Makeflow scripts directly with the Condor batch system. That is, all tasks in the Makeflow scripts are submitted as individual Condor jobs. However, this setup suffers from slow startup overhead inherent in the Condor system. And, the benefit of vast off-campus resources (e.g. thousands of computing nodes from Wisconsin-Madison and Purdue) is usually spoiled by the low transfer rate. Furthermore, it is difficult to adjust the resource allocation for submitted workflows at runtime. Last but not least, submitting each task as a Condor job is equivalent of matching the number of resources to the number of tasks, which might cause resource waste as shown in the previous capacity management chapter. These problems inspired the switch to Work Queue, which has the superior resource management capability.

In the current Biocompute system, each Makeflow script is invoked as a Work Queue master. That is, any workflow constructed by the web portal runs a Work Queue master in the system. An external worker pool, managed by the Biocompute

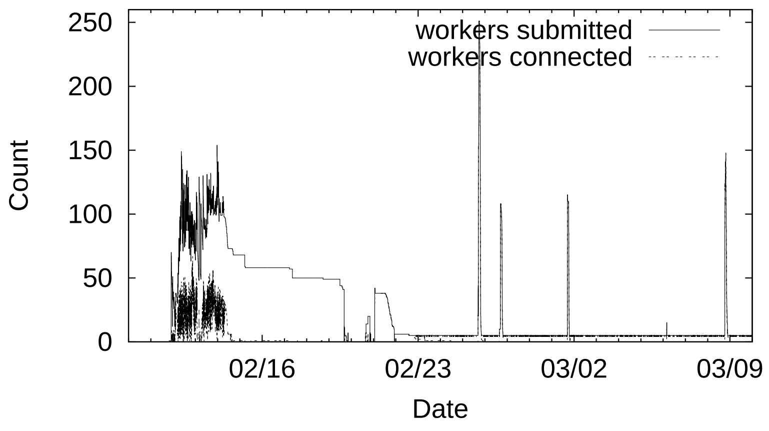


Figure 5.5: Biocompute Worker Pool Activity in 2013

system administrator, is responsible for allocating resources for the Biocompute masters. Since the capacity management architecture is fully implemented in the Work Queue framework, the Biocompute system is now taking advantage of this resource management capability. All the Biocompute masters advertise their own status to the catalog server deployed at Notre Dame. The Biocompute worker pool is configured to respect the master capacity and will allocate resources accordingly. And because the workflow runs as a Work Queue master, the user is now able to add additional grid or cloud resources to the execution of the workflow at runtime. The user only needs to start some extra workers on the grid or the cloud platform (with tools provided in the CCTools software) and point them to the desired master's hostname and port, which could be obtained from the catalog server.

Figure 5.5 shows the activity of the Biocompute worker pool from Feb 11, 2013 to Mar 25, 2013. The solid curve shows the number of submitted workers over time. The dotted curve shows the number of workers that are actually connected to the biocompute masters over time. As can be seen, the worker pool is able to automatically request for resource when there are workloads present and reduce them as the workloads finish. Starting from Feb 23, the worker pool is configured to always keep 5 workers submitted, as can be seen from the solid curve to the left of the Feb 23 point. These 5 constantly running workers allows workers to connect to a new workload immediately when the workload starts. Without these workers, the new workload would have to wait for the underlying resource management system (e.g. Condor) to allocate the resources for the workers requested by the worker pool, which typically involves a delay ranges from tens of seconds to a few minutes. The 5 workers may timeout and quit when there are no Biocompute masters, in which case, the worker pool would resubmit workers to compensate for the timed out ones.

5.2 Economics

The wavefront abstraction can represent a number of dynamic economic problems. Consider, for example, the competition between two microprocessor vendors. Each firm produces microprocessors and engages in R&D to improve the clock speed. That game ends when they reach limits imposed by physics. Economic models examining such dynamic games would discretize the problem by assuming that there are N possible efficiencies and each firm begins with efficiency level 1. The state of a two-player game is denoted by the vector of efficiencies, (i,j). At each such state, each firm competes for sales of the chips of those efficiencies but each firm also wants to improve its efficiency. When the game reaches the state (N,N) the dynamics are done and we have reached a static situation which can be computed directly. If the state of the game is $(N-1,N)$ then firm 1 still works to improve its efficiency and its incentives to work on R&D are affected by the anticipated profits it receives when the game goes to (N,N). This is also true for player 2 in the state $(N,N-1)$. Hence, the solution at (N,N) allows us to solve $(N-1,N)$ and $(N,N-1)$. Similarly, those solutions allow us to solve $(N-2,N)$, $(N-1,N-1)$ and $(N,N-2)$. The wavefront abstraction sweeps through the states until we have solved the dynamic game at all states (i,j), $1 \leq i,j \leq N$.

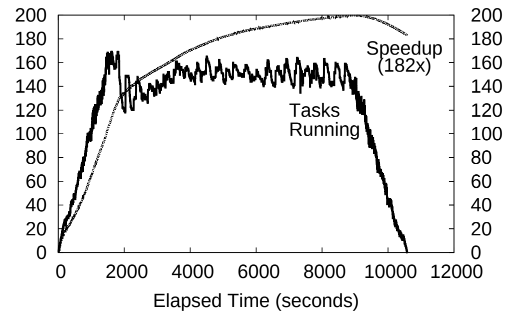


Figure 5.6: 500×500 Wavefront in Economics
A timeline of a 500×500 Wavefront problem in economics running on non-dedicated multicore Condor pool. Because many of the remote CPUs were faster than the submitting CPU, the overall speedup of 180X is greater than the number of CPUs.

This kind of game arises in many dynamic economic problems. See [42, 97, 96] for original papers on the learning curve, [85, 87, 86, 30] for examples of dynamic R&D races, and [88] for an example from the exhaustible resources literature. All of these results are limited in scope because a sequential implementation dramatically limits the number of parameters. For example, the learning and R&D papers assume only two firms and a small number of steps. This is an unreasonable assumption since there are many firms in each industry, particularly at the early stages where innovation is rapid and many firms are competing to be one of the few survivors. These models are essential for a serious examination of antitrust policies that limit how fiercely firms may compete and tax policies that are supposedly designed to encourage innovation.

Using the wavefront abstraction, we can easily carry out problems many orders of magnitude larger than have been attempted before. With less than a day of coding, we ported a Nash equilibrium function for two players with four parameters from Mathematica into a 77-line C program usable with Wavefront. On a single input, this function requires about 7.6 seconds to complete on a 1GHz CPU.

Figure 5.6 shows a timeline of this workload running on the Condor pool. The workload quickly reached the maximum available parallelism of between 120 and 160 CPUs. An overall speedup of 182X was achieved, reducing the sequential runtime from 22 days to 2.9 hours. The speedup achieved was faster than ideal because many of the remote CPUs were faster than the submitting machine on which the function was benchmarked.

5.3 Molecular Modeling

Molecular modeling is a research area that uses theoretical methods and computational techniques (simulations) to model or mimic the behavior of molecules. It usually requires large amount of computing power, however, traditional simulation techniques lack the scalability to take advantage of the vastly available grid and cloud computing resources. Accelerated Weighted Ensemble or AWE package provides a Python library for adaptive sampling of molecular dynamics. This method requires only a large number of short calculations and incurs minimal communication between computing nodes, which creates the potential to scale up the computation onto a wide range of distributed computing platforms.

A team of researchers from the University of Notre Dame and Stanford University created a protein folding simulation system that uses the AWE technique to run thousands of short Gromacs and Protomol simulations in parallel with periodic resampling to explore the rich state space of a molecule. Using the Work Queue framework, these simulations are executed by workers (managed by worker pools) distributed across thousands of CPUs and GPUs drawn from the Notre Dame, Stanford, and commercial cloud providers. At the scale of thousands of cooperative computing nodes, the resulting system was able to simulate the behavior of a protein at an aggregate sampling rate of over 500 ns/hour, covering a wide range of behavior in days rather than years. Figure 5.7 shows some scientific results obtained through the AWE system – two Fip35 folding pathways.

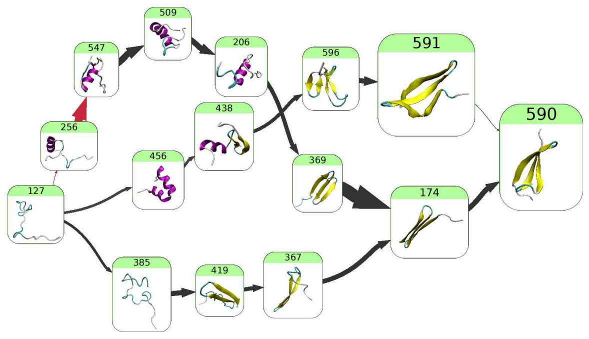


Figure 5.7: Fip35 Folding Pathways found from the AWE Network
Colors blue, grey, and red represent unfolded, intermediate, and folded conformations, respectively.

To demonstrate the scalability on heterogeneous resources, we show a timeline of three AWE applications started at different points over a three day period along with the resources that are provided to them in Figure 5.8. All three masters share the same set of workers (managed by multiple worker pools) and receive a certain amount of workers according to their changing needs. Workers from the same resource provider is managed by the same single worker pool. The workers are requested from the following resource platforms: Notre Dame SGE cluster of 6000 cores, Notre Dame Condor pool of 8000 cores (with the ability to request resources from Condor pools at Purdue University and the University of Wisconsin-Madison), Stanford ICME(a dedicated cluster at Stanford University, consisting of about 200 CPUs and 100 NVIDIA C2070 GPUs), Amazon EC2, and Microsoft Azure. The latter two platforms are commercial cloud platforms which provide virtually unlimited amount of virtual machines.

Figure 5.8 contains three graphs that show the number of connected workers over time for each of the three masters. The progression of the execution is as follows: Master 0 (M0) is started and runs alone for 11 hours. The second master – M1 is then started. Because M0 has entered the straggling phase which does not need as many workers, the workers that used to work for M0 starts to migrate to M1. The

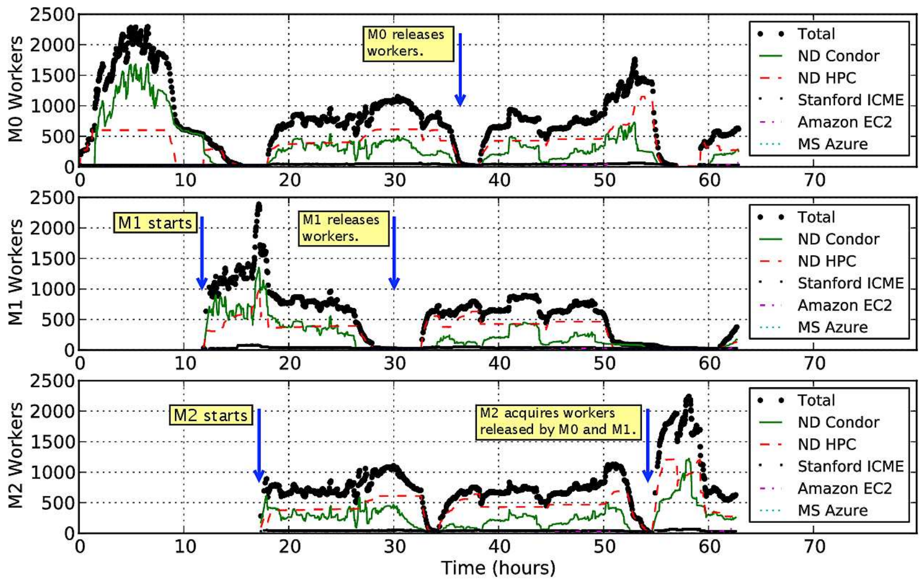


Figure 5.8: Multiple Worker Pools Providing Workers to Three AWE Applications

third master – M2 started at hour 18. New workers are started by the pool and some workers from M0 and M1 are migrated to M2. As can be seen through the entire timeine, when one master enters a slow phase (needs less workers), others are able to share the available resources.

CHAPTER 6

CONCLUSION

This work has been focusing on the question of how many computing resources should be allocated for a given workload. We have answered this question for two types of workloads – static and dynamic workloads. We have described abstractions as a solution for regular static workloads and capacity management as a solution for dynamic workloads. And we have shown the applications of these solutions in bioinformatics, economics, and molecular modeling.

For static workloads, we have demonstrated how simple high level abstractions can be used to scale regularly structured problems up to clusters of multicore computers. We have made the following key observations:

- It is feasible to accurately model the performance of large scale abstractions across a wide range of configurations, allowing for the rational selection of appropriate resources.
- Processes are a realistic alternative to threads for programming multicore systems, even on I/O intensive tasks.
- Abstractions are easy for non-experts to program, provided there is a good match between the application structure and the application.
- The All-Pairs and Wavefront abstractions can be scaled up to hundreds of cores, achieving good performance even under adverse conditions.
- General abstractions, like Makeflow, are able to deal with more kinds of application structures; however, they might not achieve the same performance as specific abstractions.

For dynamic workloads, we have shown how resource waste can arise when running elastic applications in a distributed computing environment and presented the capacity management architecture as a solution to avoid such waste. Our solution uses an external resource allocator to allocate and manage computing resources for elastic applications based on their runtime performance and capacity measurements. In addition to the application's runtime measurements, we have identified the following factors that can affect the effectiveness of the resource allocator: the limit on the speed of allocating new resources, the default capacity for applications that do not yet have capacity estimations, and the billing cycle of the computing resources. We have shown how these factors, individually or combined, can affect the quality of the resource allocation. By evaluating the resource allocator's performance with different workload patterns ranging from highly homogeneous to completely random, we have demonstrated that our solution can significantly reduce resource waste without sacrificing application performance.

There are many avenues of future work. For static workloads, we have outlined a two-level hierarchy of implementations for abstractions, but the system could be generalized to support solving very large problems across the wide area with deeper nesting. Additional implementations of abstractions on specialized architectures such as the Cell or FPGAs might be effective ways of transparently adding such devices to large computations. For dynamic workload, there are several potential improvements that are worth studying for the capacity management architecture. In the current implementation, the workers would only exit the system after a fixed timeout. If this timeout can be determined at runtime based on some cost variables, the overall resource waste reduction may be further improved. New capacity estimation algorithms can be constructed measure the resource needs even more precisely. And the system may apply different capacity estimation algorithm on different types of application, if the type of application could be somehow determined in advance. Another direction is to reduce the possibilities of false negative feedback, as seen in the cache bounce effect, so that the resource allocator can make more aggressive decisions (reduce workers connected to a master immediately after its capacity decreases) without fall into oscillations. Last but not least, we can exploit various resource providers' specialties, such as Amazon EC2's spot instance, to make more cost-effective resource allocation strategies.

BIBLIOGRAPHY

1. Amazon elastic compute cloud (ec2). http://aws.amazon.com/ec2, July 2013.

2. Amazon auto scaling. http://aws.amazon.com/autoscaling/, July 2013.

3. Nimbus home page. http://www.nimbusproject.org, July 2013.

4. Rightscale. http://www.rightscale.com, July 2013.

5. B. Abdul-Wahid, L. Yu, D. Rajan, H. Feng, E. Darve, D. Thain, and J. A. Izaguirre. Folding Proteins at 500 ns/hour with Work Queue. In *8th IEEE International Conference on eScience (eScience 2012)*, 2012.

6. I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management*, pages 423–424, 2004.

7. S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 3(215):403–410, Oct 1990.

8. D. P. Anderson. Boinc: A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10. IEEE, 2004.

9. D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

10. N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. Ourgrid: An approach to easily assemble grids with equitable resource sharing. In *Job scheduling strategies for parallel processing*, pages 61–86. Springer, 2003.

11. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

12. M. Baker and R. Buyya. Cluster computing: The commodity supercomputing. *SOFTWAREPRACTICE AND EXPERIENCE*, 1(1):1–4, 1988.

13. D. Bakken and R. Schlichting. Tolerating failures in the bag-of-tasks programming paradigm. In *IEEE International Symposium on Fault Tolerant Computing*, June 1991.

14. L. Beernaert, M. Matos, R. Vilaça, and R. Oliveira. Automatic elasticity in openstack. In *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*, page 2. ACM, 2012.

15. D. Bellenger, J. Bertram, A. Budina, A. Koschel, B. Pfänder, C. Serowy, I. Astrova, S. G. Grivas, and M. Schaaf. Scaling in cloud environments. *Recent Researches in Computer Science*, 2011.

16. A. Bialecki, M. Cafarella, D. Cutting, and O. OMALLEY. Hadoop: a framework for running applications on large clusters built of commodity hardware. *Wiki at http://lucene. apache. org/hadoop*, 11, 2005.

17. R. S. Boyer and J. S. Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the ACM (JACM)*, 31(3):441–458, 1984.

18. T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, et al. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed computing*, 61(6):810–837, 2001.

19. P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain. Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications. In *Workshop on Python for High Performance and Scientific Computing (PyHPC) at the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing)* , 2011.

20. R. Carmichael, P. Braga-Henebry, D. Thain, and S. Emrich. Biocompute: towards a collaborative workspace for data intensive bio-science. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 489–498. ACM, 2010.

21. E. Caron, F. Desprez, and A. Muresan. Forecasting for grid and cloud computing on-demand resources based on pattern matching. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 456–463. IEEE, 2010.

22. T. Cheatham, A. Fahmy, D. Siefanescu, and L. Valiani. Bulk synchronous parallel computing-a paradigm for transportable software. In *Hawaii International Conference on Systems Sciences*, 2005.

23. H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, 1989.

24. W. Cirne, F. Brasileiro, J. Sauvé, N. Andrade, D. Paranhos, E. Santos-neto, R. Medeiros, and F. C. Gr. Grid computing for bag of tasks applications. In *In Proc. of the 3rd IFIP Conference on E-Commerce, E-Business and EGovernment.* Citeseer, 2003.

25. K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. Resource management architecture for metacomputing systems. In *IPPS/SPDP Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998.

26. D. da Silva, W. Cirne, and F. Brasilero. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Euro-Par*, 2003.

27. W. Dawoud, I. Takouna, and C. Meinel. Elastic virtual machine for fine-grained cloud resource provisioning. In *Global Trends in Computing and Communication Systems*, pages 11–25. Springer, 2012.

28. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large cluster. In *Operating Systems Design and Implementation*, 2004.

29. E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, B. Berriman, J. Good, A. Laity, J. Jacob, and D. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13(3), 2005.

30. U. Doraszelski. An R&D race with knowledge accumulation. *Bell Journal of Economics*, 34:19–41, 2003.

31. R. C. Dorf. *Modern control systems*. Addison-Wesley Longman Publishing Co., Inc., 1991.

32. J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

33. E. Elmroth and P. Gardfjall. Design and evaluation of a decentralized system for grid-wide fairshare scheduling. In *e-Science and Grid Computing, 2005. First International Conference on*, pages 9–pp. IEEE, 2005.

34. S. I. Feldman. Makea program for maintaining computer programs. *Software: Practice and experience*, 9(4):255–265, 1979.

35. I. Foster and C. Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Access Online via Elsevier, 2003.

36. J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5 (3):237–246, 2002.

37. M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache oblivious algorithms. In *Foundations of Computer Science (FOCS)*, 1999.

38. M. R. Garey and D. S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal on Computing*, 4(4):397–411, 1975.

39. M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1(2):117–129, 1976.

40. N. Geddes. The national grid service of the uk. In *e-Science and Grid Computing, 2006. e-Science'06. Second IEEE International Conference on*, pages 94–94. IEEE, 2006.

41. W. Gentzsch. Sun grid engine: Towards creating a compute power grid. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, 2001.

42. P. Ghemawat and A. M. Spence. Learning curve spillovers and market performance. *The Quarterly Journal of Economics*, 100:839–852, 1985.

43. C. T. Gibson. Time-sharing in the ibm system/360: model 67. In *Proceedings of the April 26-28, 1966, Spring joint computer conference*, pages 61–78. ACM, 1966.

44. R. Grossman and Y. Gu. Data mining using high performance data clouds: experimental studies using sector and sphere. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 920–927. ACM, 2008.

45. M. Hategan, J. Wozniak, and K. Maheshwari. Coasters: uniform resource provisioning and access for scientific computing on clouds and grids. *Proc. Utility and Cloud Computing*, 2011.

46. R. Henderson and D. Tweten. Portable batch system: External reference specification. Technical report, NASA, Ames Research Center, 1996.

47. Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez, and M. Humphrey. Early observations on the performance of Windows Azure. In *Proceedings of HPDC*, 2010.

48. D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34:429–732.

49. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data parallel programs from sequential building blocks. In *Proceedings of EuroSys*, March 2007.

50. S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. Task scheduling strategies for workflow-based applications in grids. 2005.

51. N. P. Jouppi and D. W. Wall. *Available instruction-level parallelism for superscalar and superpipelined machines*, volume 17. ACM, 1989.

52. H. T. Kung. Why Systolic Architectures? *IEEE Computer*, 15:37–46, January 1982.

53. S. M. Larson, C. D. Snow, M. Shirts, et al. Folding@ home and genome@ home: Using distributed computing to tackle previously intractable problems in computational biology. 2002.

54. E. Laure and B. Jones. Enabling grids for e-science: The egee project. *Grid Computing: Infrastructure, Service, and Applications, CRC Press*, pages 55–74, 2009.

55. H. Li and R. Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

56. C.-C. Lin, J.-J. Wu, J.-A. Lin, L.-C. Song, and P. Liu. Automatic resource scaling based on application service requirements. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 941–942. IEEE, 2012.

57. J. Linderoth, S. Kulkarni, J.-P. Goux, and M. Yoder. An enabling framework for master-worker applications on the computational grid. In *IEEE High Performance Distributed Computing*, pages 43–50, Pittsburgh, Pennsylvania, August 2000.

58. A. Luckow, L. Lacinski, and S. Jha. SAGA BigJob: An extensible and interoperable pilot-job abstraction for distributed applications and systems. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 135–144. IEEE, 2010.

59. J. MacLaren, R. Sakellariou, K. T. Krishnakumar, J. Garibaldi, and D. Ouelhadj. Towards service level agreement based scheduling on the grid. In *Proceedings of the Workshop on Planning and Scheduling for Web and Grid Services*, pages 100–102, 2004.

60. T. Maeno. PanDA: distributed production and distributed analysis system for ATLAS. *Journal of Physics: Conference Series*, 119, 2008.

61. M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

62. M. Mao, J. Li, and M. Humphrey. Cloud auto-scaling with deadline and budget constraints. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 41–48. IEEE, 2010.

63. P. Marshall, K. Keahey, and T. Freeman. Elastic site: Using clouds to elastically extend site resources. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 43–52. IEEE Computer Society, 2010.

64. P. Mell and T. Grance. The nist definition of cloud computing (draft). *NIST special publication*, 800(145):7, 2011.

65. R. G. Michael and D. S. Johnson. Computers and intractability: A guide to the theory of np-completeness. *WH Freeman & Co., San Francisco*, 1979.

66. C. Moretti, J. Bulosan, D. Thain, and P. Flynn. All-Pairs: An Abstraction for Data Intensive Cloud Computing. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–11, 2008.

67. C. Moretti, K. Steinhaeuser, D. Thain, and N. V. Chawla. Scaling Up Classifiers to Cloud Computers. In *IEEE International Conference on Data Mining (ICDM)*, pages 472–481, 2008.

68. C. Moretti, A. Thrasher, L. Yu, M. Olson, S. Emrich, and D. Thain. A Framework for Scalable Genome Assembly on Clusters, Clouds, and Grids. *IEEE Transactions on Parallel and Distributed Systems*, 23(12), 2012.

69. A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *Parallel and Distributed Systems, IEEE Transactions on*, 12(6):529–543, 2001.

70. H. Muller, M. Pitkanen, X. Zhou, A. Depeursinge, J. Iavindrasana, and A. Geissbuhler. Knowarc: enabling grid networks for the biomedical research community. *Studies in health technology and informatics*, 126:261, 2007.

71. A. Nagavaram, G. Agrawal, M. A. Freitas, K. H. Telu, G. Mehta, R. G. Mayani, and E. Deelman. A cloud-based dynamic workflow for mass spectrometry data analysis. In *E-Science (e-Science), 2011 IEEE 7th International Conference on*, pages 47–54. IEEE, 2011.

72. S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

73. G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–177, 2006.

74. Z. Ning, A. J. Cox, and J. C. Mullikin. Ssaha: a fast search method for large dna databases. *Genome research*, 11(10):1725–1729, 2001.

75. T. Oliver, B. Schmidt, D. Nathan, R. Clemens, and D. Maskell. Using reconfigurable hardware to accelerate multiple sequence alignment with clustalw. *Bioinformatics*, 21:3431–3432, 2005.

76. J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

77. A. H. Paterson, J. E. Bowers, R. Bruggmann, I. Dubchak, J. Grimwood, H. Gundlach, G. Haberer, U. Hellsten, T. Mitros, A. Poliakov, et al. The sorghum bicolor genome and the diversification of grasses. *Nature*, 457(7229): 551–556, 2009.

78. X. Qiu, M. Hedwig, and D. Neumann. SLA Based Dynamic Provisioning of Cloud Resource in OLTP Systems. In *E-Life: Web-Enabled Convergence of Commerce, Work, and Social Life*, pages 302–310. Springer, 2012.

79. M. Rahman, S. Venugopal, and R. Buyya. A dynamic critical path algorithm for scheduling scientifc workflow applications on global grids. 2007.

80. I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a Fast and Light-weight tasK executiON framework. In *IEEE/ACM Supercomputing*, 2007.

81. S. Rajko and S. Aluru. Space and time optimal parallel sequence alignments. *IEEE Transactions on Parallel and Distributed Systems*, pages 1070–1081, 2004.

82. C. Ramamoorthy and H. F. Li. Pipeline architecture. *ACM Computing Surveys (CSUR)*, 9(1):61–102, 1977.

83. R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, pages 140–146. IEEE, 1998.

84. T. Redkar and T. Guidici. *Windows Azure Platform.* Apress, 2011.

85. J. Reinganum. Dynamic games of innovation. *Journal of Economic Theory*, 25: 21–41, 1981.

86. J. Reinganum. A dynamic game of R&D: Patent protection and competitive behavior. *Econometrica*, 50:671–688, 1982.

87. J. Reinganum. Corrigendum. *Journal of Economic Theory*, 35:196–197, 1985.

88. J. Reinganum and N. Stokey. Oligopoly extraction of a common property natural resource: The importance of the period of commitment in dynamic games. *International Economic Review*, 26:161–174, 1985.

89. N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 500–507. IEEE, 2011.

90. S. M. Rumble, P. Lacroute, A. V. Dalca, M. Fiume, A. Sidow, and M. Brudno. Shrimp: accurate mapping of short color-space reads. *PLoS computational biology*, 5(5):e1000386, 2009.

91. J. Saltz, S. Oster, S. Hastings, S. Langella, T. Kurc, W. Sanchez, M. Kher, A. Manisundaram, K. Shanbhag, and P. Covitz. cagrid: design and implementation of the core architecture of the cancer biomedical informatics grid. *Bioinformatics*, 22(15):1910–1916, 2006.

92. A. Sarje and S. Aluru. Parallel biological sequence alignments on the cell broadband engine. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2008.

93. I. Sfiligoi. GlideinWMS – a generic pilot-based workload management system. *Journal of Physics: Conference Series*, 119, 2008.

94. M. Shirts, V. S. Pande, et al. Screen savers of the world unite. *COMPUTING*, 10:43, 2006.

95. M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: the complete reference*. MIT press, 1995.

96. A. M. Spence. The learning curve and competition. *Bell Journal of Economics*, 12:49–70, 1981.

97. A. M. Spence. Cost reduction, competition, and industry performance. *Econometrica*, 52:101–121, 1984.

98. C. Team. The directed acyclic graph manager. http://www.cs.wisc.edu/condor/dagman, 2002.

99. D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. In F. Berman, A. Hey, and G. Fox, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.

100. D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.

101. K. B. Theobald and G. R. Gao. An efficient parallel algorithm for all pairs examination. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 742–753, New York, NY, USA, 1991. ACM. ISBN 0-89791-459-7. doi: http://doi.acm.org/10.1145/125826.126175.

102. A. Thrasher, Z. Musgrave, D. Thain, and S. Emrich. Shifting the Bioinformatics Computing Paradigm: A Case Study in Parallelizing Genome Annotation Using Maker and Work Queue. In *IEEE International Conference on Computational Advances in Bio and Medical Sciences*, 2012.

103. A. Tsaregorodtsev, M. Bargiotti, N. Brook, A. C. Ramo, G. Castellani, P. Charpentier, C. Cioffi, J. Closier, R. G. Diaz, G. Kuznetsov, et al. Dirac: a community grid solution. In *Journal of Physics: Conference Series*, volume 119, page 062048. IOP Publishing, 2008.

104. D. Warneke and O. Kao. Exploiting dynamic resource allocation for efficient parallel data processing in the cloud. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6):985–997, 2011.

105. D. Warneke and O. Kao. Exploiting Dynamic Resource Allocation for Efficient Parallel Data Processing in the Cloud. January 2011.

106. W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *Computers, IEEE Transactions on*, 100(8):949–960, 1987.

107. Y. Zhao, J. Dobson, L. Moreau, I. Foster, and M. Wilde. A notation and system for expressing and executing cleanly typed workflows on messy scientific data. In *SIGMOD*, 2005.

108. S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: a load sharing facility for large, heterogenous distributed computer systems. *Software: Practice and Experience*, 23(12):1305–1336, December 1993.

This document was prepared & typeset with LaTeX 2_ε, and formatted with NDdiss2_ε *classfile (v3.2013β[2013/01/31]) provided by Sameer Vijay and updated by Megan Patnott.*