ABSTRACTIONS FOR SCIENTIFIC COMPUTING ON CAMPUS GRIDS

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

Christopher M. Moretti

Douglas L. Thain, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
June 2010

ABSTRACTIONS FOR SCIENTIFIC COMPUTING ON CAMPUS GRIDS

Abstract

by

Christopher M. Moretti

Scientific computing users often find it difficult to transform serial domain applications into workloads for large non-dedicated heterogeneous campus grids. Due to hardware and software bottlenecks, a workload that succeeds on 8 nodes can fail disastrously on 128; or even fail on 8 nodes for a different instance of the same problem.

An abstraction is a flexible solution to a pattern of computation that can be used to harness distributed computing resources more easily for non-experts. The users provide the pieces, such as their datasets and serial function, and the workload is constructed and executed for them in an appropriate manner for the environment in order to prevent disastrous configurations and satisfy cost, policy, and performance constraints.

This work presents the design, implementation, and evaluation of a ”toolbox” of middleware and abstractions: All-Pairs, Sparse-Pairs, and Data-Split-Join. These abstractions are used for several problems in bioinformatics, biometrics, and data mining. The discussion of the abstractions includes modeling of the problem, managing input data, organizing computation on the campus grid, and managing output data. Results include the largest known biometrics All-Pairs result of its kind, in which over two years’ worth of computation was executed in 10 days, and a complete alignment of the Human genome using Sparse-Pairs, which completed in 2.5 hours on over 1000 hosts with 952x speedup.

CONTENTS

FIGURES

TABLES

CODE EXCERPTS

ACKNOWLEDGMENTS

I would like to thank my parents for their advice and encouragement throughout my time in graduate school.

I could not have completed this work without the day-to-day support, intellectual stimulation, technical help, and comic relief from my friends and colleagues. In particular I must thank Laurence Brodeur, Kim Lally, Chris Middendorff, Alec Pawling, Phil Snowberger, Deborah Thomas, Hoang Bui, Jeff Hemmes, Li Yu, Mike Albrecht, Peter Bui, Mike Olson, Rory Carmichael, and Tim Dysart.

I also owe a great deal of gratitude to Dr. Douglas Thain for guidance, motivation, and insight throughout this research. His boundless intellectual curiosity has fostered my own, and his keen attention to detail has routinely challenged and sharpened my analytic faculties.

In addition, Dr. Patrick Flynn and my committee members Dr. Nitesh Chawla, Dr. Scott Emrich, and Dr. Christian Poellabauer were supportive and flexible throughout the process. I appreciate their help and guidance on the variety of collaborative projects that make up this dissertation.

CHAPTER 1

INTRODUCTION

Distributed systems are a necessary component of modern science and engineering. Individual computers continue to increase in processor speed, memory and disk capacities. But as larger and more difficult problems are solved with these bigger, faster computers, their successes and conclusions inspire even bigger questions to be answered. If individual computers must be used to solve the problem, this creates a cycle of continual waiting for the next bigger machine to arrive. Even after improved hardware is acquired, there are still drawbacks, including the *application development barrier* [35], in which older applications do not run on new systems, or cannot take advantage of a new system's capabilities. This results in applications having to be redesigned and recoded, further delaying work on the larger versions of the problem. The well-recognized alternative is to utilize many current commodity computers to complete the task instead of waiting for one next generation system to do it alone. To meet these computing needs, many institutions have installed *campus grids* made up of hundreds to tens of thousands of assorted commodity processors [122].

Just providing a campus grid isn't enough, however, because users quickly realize that distributed computing is hard. Many users construct serial versions of their applications, but are unsure how to adapt them to a parallel solution. The applications may require exclusive access to resources, or may expect shared state among subproblems. In constructing the parallel versions of applications, bottlenecks develop within

the workload, for instance at the shared fileserver, network device, or results database. Further, failures are much more common in distributed systems than on local machines, so applications must be expanded to deal with a new variety of errors.

Mishandling these challenges can result in poor performance, outright failure of the application, and abuse of physical resources shared by others. All too often, an end user composes a workload that runs correctly on one machine, then on ten machines, but fails disastrously on one hundred or one thousand machines. For example, consider a non-expert's workload design in which each distributed process starts execution by loading 1GB of required data from a central fileserver. If one processor alone attempts this, it will experience a latency of a few seconds, depending on disk and network speeds. If ten processors start up simultaneously and compete for the central fileserver's bandwidth, they will experience latencies of minutes. If one thousand processors connect at once, this will require a total of 1TB of data transfers – likely resulting in dropped connections for this and other unrelated workloads, very slow access for those transfers that remain active, and network congestion on shared switches that affects users not even using the fileserver.

This dissertation proposes abstractions as the solution to the problem of navigating the complexities of computing on a campus grid.

Abstractions make distributed computing resources more easily used by non-experts. Users must provide well-known pieces of their workload; but the workload is constructed for them in an appropriate manner for the campus grid environment. This improves usability, increases performance, and prevents disastrous configurations.

An abstraction is a device used to improve the ability of non-expert users to solve problems using complex systems. The user provides elements of a problem that he knows, such as multiple sequential programs used to solve the problem serially and the

data that they process, but cedes control of the actual method of execution to a workflow engine. That engine can then manage disk, network, and processor requirements to complete the intended computation while hiding the details of how the workload is realized in the system.

In this work, "non-experts" are those who are not specialists in distributed computing. The users of the abstractions described here have primarily been computer scientists from across the discipline, ranging from advanced undergraduates to research faculty members. Their general computing knowledge and skills are fundamental to their ability to do science at large scales: they must understand what they are doing and why it is the correct course of action. Using an abstraction does not dispense with this requirement, rather it reduces the meticulous load on these users to account for all of the details of specific distributed computing challenges. The abstractions are designed with the goal that, in subsequent work, scientists in other fields who are competent computer users and programmers, even without broad computer science knowledge and capability, will be able to function as effectively as the computer scientists for their own applications of the computing patterns.

Abstractions are specified in terms of their major components: inputs, functions, and outputs. This can be applied, as in an implemented abstraction engine, or higher-level, such as in an abstract problem specification. To introduce the way that abstract problems will be formally defined, and to demonstrate the challenges associated with design and development of abstractions, consider first an example abstraction. The Map abstraction [120] is a very simple pattern that has been applied across a vast array of problems from many disciplines. Map applies a transformation function to each member of the input set, resulting in a new set of the same number of elements:

Map(data $D[i]$, **function** F(**data** x))

returns array R **such that** $R[i] = F(D[i])$

Although Map is a simple loop operation on a single machine, developing abstractions to solve it and problems like it is not trivial. The characteristics of the application are often not well-defined by the general pattern; e.g. F for two instances of Map might take the same input but vary in their execution time or memory requirements by orders of magnitude. Moreover, these differences can require drastically different solutions to the problem; e.g. computing Map on 1,000,000 1KB files where each F takes several hours should be distributed to as many nodes as possible, whereas it would be very inefficient to transfer 1,000,000 1KB files individually to a large number of resources if each F took only a fraction of a second.

Not only are various instances of the same application likely to vary widely, but the computing environment itself can as well. Campus grids are heterogeneous and dynamic, so unlike a cluster in which a solution may be finely tuned for the exact environment in which it will operate, an abstraction for a campus grid must be flexible to a wide range of available resources and configurations. And although the user of the system has access to an enormous number of CPUs, a standard feature of the environment is that his jobs may be preempted without warning when the resource provider reclaims access.

Because an abstraction states a workload in a declarative way, it can be implemented in whatever way satisfies cost, policy, and performance constraints – a critical flexibility required for heterogeneous campus grids. An abstraction may also use the information available to it about the workload and the system to avoid the bottlenecks and other pitfalls that may not be apparent or important to a novice application developer, limiting the opportunity for disasters. Whereas a customized solution to a problem may require

re-engineering to adapt to even the smallest changes – perhaps even to a change in specific inputs! – an abstraction should be flexible to many instances of a problem and a broad set of available resources.

The contribution of this work is a "toolbox" of campus grid computing abstractions that have been implemented and deployed for scientific applications, which are used to present the considerations that must be taken into account in the design and implementation of campus grid abstractions. Abstractions and other high-level programming interfaces are common on clusters, for instance the Map-Reduce [37] abstraction has become a widely-used cluster computing paradigm in recent years. The abstractions presented here work from a similar philosophy and towards a similar goal as Map-Reduce: breaking down large computations into common patterns that can be modeled, planned, and executed in efficient ways without requiring the programmer to individually solve all of the complexities of a distributed computing environment.

Where this work diverges from popular Map-Reduce implementations and other cluster abstractions is the underlying computing environment. Many abstractions for typical cluster environments – which often have homogeneous sets of high-end resources, non-preempting scheduling policies, and a limited number of separate resource owners and administrative domains – do not consider the implications of organizing workloads for the characteristic campus grid challenges mentioned above. Though the core stages of of an abstraction on a cluster versus a campus grid are quite similar (modeling the problem, managing input, coordinating computation, and managing output), the systems challenges for each stage, and thus the abstractions themselves, are fundamentally different because of the disparity in the capabilities and limitations of the environment.

This primary contribution of this dissertation is the design, implementation, and

evaluation of three abstractions for campus grid computing. Specific abstractions are presented for three widely-applicable computation patterns: All-Pairs, Sparse-Pairs, and Data-Split-Join. These problems are of interest to the biometrics, bioinformatics, and data mining communities, and are not well-served by any existing abstraction. The common goal of these three abstractions is to allow users to complete workloads on the campus grid that would otherwise be infeasible, either due to the number of resources required or the complexity of organizing and managing the workload efficiently.

Chapter 2 describes the work that has previously been done on campus grids, particularly in light of the differences between campus grids and other parallel environments that tend to provide a more homogeneous set of resources, a more dedicated environment, or both. It then addresses other workflow systems and computing abstractions for distributed workloads, including the widely-used and highly-influential Map-Reduce [37] abstraction. Workflow systems are more generally adaptable than abstractions for specific patterns of work, but this generality comes at the cost of performance or interface usability for non-experts.

Chapter 3 expands on the properties of campus grid computing environments and the challenges of computing in such an environment in more detail. It is here that the contrast between solving a problem on a traditional tightly-coupled parallel environment with a central fileserver is contrasted with the architecture for computing with abstractions, including the differences in modeling the problem, collocating data and computation, and specifying and managing resources. Chapter 3 also describes the various different middleware systems used by the abstractions, including the custom master/worker middleware, Work Queue, which ameliorates several important disadvantages of batch system computing on campus grids.

All-Pairs is a doubly data parallel problem in which a results matrix for two datasets

is created by applying a function to every pair of elements from those sets. Chapter 4 describes a thorough model for analyzing an instance of an All-Pairs problem in order to ascertain appropriate parameters for executing it on the campus grid. Parameters (such as local versus grid computing, number of computing nodes, and size of workload jobs) are chosen with the model to minimize the overall turnaround time. The implementation focuses on efficient data distribution to all nodes and work allocation given the wide data distribution. Results for the All-Pairs framework on biometric and data mining applications are shown, including the largest known complete iris data comparison, a 58639×58639 All-Pairs problem, which would not have been feasible without an efficient computing abstraction.

Sparse-Pairs is also a problem that computes the function for pairs of items from two data sets, however unlike All-Pairs, not every possible combination must be computed. Instead, a Sparse-Pairs problem instance also includes a list of the pairs that should be computed. Chapter 5 describes the engineering challenges and results for a Sequence Alignment bioinformatics application of the Sparse-Pairs abstraction. The implementation focuses on efficient network and memory management using Work Queue. Results include scalability (measured in terms of speedup and parallel efficiency) experiments for several increasingly large genomic datasets. The largest result is a complete alignment of the Human genome, which completed in 2.5 hours on over 1000 hosts with 952x speedup.

Data-Split-Join is a divide-and-conquer pattern in which a single large data set is split into a number of partitions, a function is applied to those partitions, and the results of those functions are then joined back into a single results set. Chapter 6 describes a distributed abstraction for ensemble classification, which is a data mining data mining Data-Split-Join application that is effective for learning on large data sets because it de-

creases the problem's complexity and increases variety and robustness versus learning on the whole dataset at once. The discussion focuses on different strategies for data partitioning and placement in the distributed system. The results are a thorough examination, using several common learning methods and a variety of real and synthetic datasets, of several parallelization implementations that differ in how they provide the classifier processes with data instances.

For each abstraction, the discussion begins with the general problem in abstract terms and an application of it. Once this has been defined, the implementation is presented in terms of the key theoretical and technical challenges associated with one or more of four tasks for a workload: modeling the system, managing the input data, coordinating the computation, and managing the output data. The abstraction's implementation and performance are compared with those from a conventional alternative, a cluster-based solution, or a different abstraction. Because the separate abstractions naturally stress different components of the workflow, discussing each of them in series permits focusing in each chapter on different parts of an abstraction design and implementation.

Finally, Chapter 7 puts this work into a larger context. This chapter contains discussion of the calculus for choosing between the specific abstractions introduced in this work and between those abstractions and widely-applied general abstractions such as Bag-of-Tasks and Map-Reduce. Although there are several parameters involved in choosing which abstraction to apply to a problem, an abstraction that fits the way the user already looks at his problem is best for minimizing the cost of translation to fit an instance of a problem to an abstraction. Despite the popularity of several general abstractions, even when a problem can be made to fit the general abstraction doing so may yield a lower ceiling for possible performance relative to an abstraction more naturally

aligned to the problem's structure.

This chapter also summarizes the impact of the "toolbox" of abstractions in terms of publications and utility to fields outside distributed systems. Designing software that is used by others to speed up their otherwise-unmodified research projects is a benefit of this work for other fields, however to stop there misses a more fundamental impact. The availability of this software changes not only the *speed* of results generation and the *scale* of approachable problems, but the *processes* by which others do research. Being able to harness large numbers of resources consistently and efficiently can change the scientific design from a focus on one singular final result into an opportunity to routinely measure full-scale or near-full-scale waypoints along a more comprehensive iterative research process.

Using abstraction to give users a high-level interface to complicated systems is not a new idea. For example, assembly language and then later compilers allowed users to provide a high-level specification of a problem without having to worry about the intricate details of making it work on any given system. But as scientific computations evolve to larger and more complicated systems, abstractions must evolve with them. Distributed computing abstractions like those presented here and middleware to support their development are key tools that allow non-expert users to harness available campus grid resources efficiently.

Computing efficiently on complex distributed environments such as campus grids will always require cooperation between experts in various computational science domains and experts in distributed systems. But this cooperation is more equitable and sustainable when each partner is allowed to work primarily within his own area of expertise. Distributed computing experts are better served to allocate most of their effort designing tools and middleware that can be used by many scientists on several differ-

ent applications than to spend all their time re-engineering the non-experts' disparate problem-specific or domain-specific solutions. And the computational scientists are better served concentrating on their own scientific process instead of moonlighting in distributed systems just to get their applications running efficiently.

CHAPTER 2

RELATED WORK

2.1 Computing on a Campus Grid

With the spread of computational research, shared computing grids are now a common feature of most research institutions' campus computing environment. These *campus grids* are made up of a wide variety of computing resources, and have characteristics that can be quite different from traditional clusters, commercial computing clouds, or tera- and peta-scale supercomputers. Using middleware, many disparate clusters and standalone machines may be joined into a single computing system with many providers and consumers. Today, campus grids of approximately 1000 cores are common [122], and larger initiatives grouping resources from several institutions can combine to tens of thousands of cores, such as the 20,000-CPU Indiana Diagrid [112] and the 40,000-CPU Open Science Grid [101].

Computing on a campus grid has a number of differences from traditional parallel environments such as a single many-core machine, a tightly-coupled Beowulf cluster [54, 56], or a BlueGene supercomputer [4, 6].

Perhaps the most widely known tools for parallel processing are parallel languages and libraries such as MPI [41], OpenMP [34], Split C [31], and Cilk [17]. Each of these requires significant adaptation of a user's serial code in order to take advantage of parallel environments. This adaptation often requires careful synchronization of

processes and direct, indirect, and collective interprocess communication. While some problems, particularly those that are naturally parallel, can easily be scaled up with only slight additions to a serial program's code, in many cases scaling a problem to run in parallel on several machines (much less hundreds or thousands) requires significant changes beyond the serial implementation. Many non-expert users are unlikely to want to face the learning curve of these systems in order to scale up their applications.

Even for those who can use one of these parallel tools, a campus grid is a difficult environment for traditional parallel computation. Campus grid resources are heterogeneous, which can make synchronization (e.g. `MPI_Barrier` operations) much more expensive than they would be on a single many-core system or tightly-coupled cluster. Similarly, performance on collective communication (or even regular interprocess communication) can be significantly worse on a campus grid, because unlike InfiniBand clusters [78] and similar tightly-coupled high-throughput/low-latency parallel environments, campus grid resources are linked via *campus area network* connections. Perhaps the largest hurdle is that the resources may be preempted often, unlike traditional parallel environments that give dedicated access for at least some duration; most parallel languages and libraries do not explicitly consider preemption as a normal condition.

Unlike the problems with the traditional parallel languages and libraries, the constraints of a campus grid are a good fit for problems that can be scaled up by managing large numbers of instances of the same serial program. This is a prototypical problem for the many-tasks computing (MTC) paradigm [105], which can generally be applied to resources that are heterogeneous in scale, performance, and networking. As an additional benefit, this instance of MTC meshes nicely with the overall goal of allowing domain scientists to scale up using their own unmodified serial programs.

2.2 Workflow Solutions

A workflow is a set of data and computation patterns arranged in an end-to-end relationship. This set of relationships may consist of all of the activities in a project [39], or be constrained to a certain subset pipeline of processes or services [119]. Workflows are often distributed for intuitive and performance reasons, and can be organized in any of several ways. A common paradigm is to orient a workflow by data-flow – that is, a directed graph with processes as vertices and data inputs and outputs as edges – which is both orderly and amenable to maintaining detailed metadata such as provenance [118].

Workflows may be solved by a continuum of systems and software, ranging from very powerful general solution engines to very specific blackbox solutions to a particular workflow problem [131]. The latter requires an exact specification of the computation and data for a workload on each resource, much like a serial application, and is not easily adaptable to a broader set of problems (or even possibly to other instances of the same problem).

2.2.1 Workflow Languages

The more general languages that design and construct workflows are much more flexible, but also more complex for users. Addis, et al. [3] define the basic requirements for a general workflow language as sequential flows, parallel flows, looping, conditionals, nesting and recursion, and complex data types. This set of capabilities makes workflow languages the most adaptable tool for constructing workflows, but also the most complicated and least predictable. Extensive use of general workflow languages risks potentially exacerbating the original problem that non-expert users are prone to making disastrous mistakes when managing complex distributed workflows.

Between the application-specific and general language endpoints lie workflow sys-

tems that do not provide all of the required capabilities (and thus, are not as flexible) as the languages, but which are still extensible to a large set of problems. Often these systems are simplified interfaces built upon a more complex back-end implementation [79, 91, 134, 136] . This combination allows a solution that is accessible to the non-expert user, but powerful and fully featured.

Swift [136] is a general end-to-end workflow system that relies on a fully-functional workflow language – SwiftScript [129] – to specify an abstract computation and the logical relationships between complex data sets. That computation is then realized, scheduled, and executed via the Karajan execution engine [127]. Swift tasks run on virtual nodes provisioned by Falkon [104] from loosely-coupled distributed systems. Provenance metadata, intermediate data, and final results are stored in a data catalog that may be a virtual node provisioned identically to the computing resources or a separate permanent storage server.

Scufl [114] is a high-level XML-based language that treats each process as an atomic task. Scufl has no variable definitions, so the state must be passed via the input/output pipeline in order to be shared between instances of tasks. Although Scufl does not have explicit looping primitives, process or service nesting is allowed and this can create data flows that accomplish looping, and the language is Turing-complete [51]. Taverna [90, 91] is a workflow system that manages a graph of processes, each of which transforms data input into data output. Most processes are web-services or local Java functions. Taverna workflows are expressed in Scufl, however they are most often manipulated using the Taverna GUI.

Dryad [68] is an extensively expressive programming framework that executed directed acyclic graphs (DAGs) made up of sequential programs as vertices and with one-way data channels as edges. Dryad is a step up from the traditional MPI-like sys-

tems, as the user does not have to be an expert in concurrent programming. Its use of sequential building blocks to construct large parallel workflow solutions is similar in spirit to MTC, as well as the abstractions presented in this dissertation. Dryad itself is a robust environment that is, however, more easily accessible through code built on top of it than via the Dryad framework itself. Examples of workflow systems that are built on Dryad include DryadLINQ [134], which presents a SQL-like declarative programming interface and integration into GUI solutions that allow for easy description of data and process workflow instructions. This allows non-expert programmers to garner automatic parallelism from .Net and C#, and even the ability to describe some other abstraction patterns, such as Map-Reduce, in a concise manner [67].

BPEL [44] is an Oasis standard [72] workflow execution language that defines data interaction with web services. Unlike Scufl, which is data-flow oriented, BPEL is control-flow oriented with explicitly defined XML and WSDL variables instead of implicit data definition based on the input and output of a process. Links in a BPEL graph indicate transfer of control instead of transfer of data; for instance a client will use the *invoke* activity to hand over control to another process. Communication consists of *receive* and *reply* activity pairs or the explicit *assign* statement. BPEL provides a broad set of control logic, including event-driven control, data-conditional control, and loops.

2.2.2 Workflow Management Systems

In addition to the GUIs, wrappers, and other high-level interfaces built atop the workflow languages above, there are other workflow systems that realize and control large workloads without a fully-featured language interface. Because workflows and other specific patterns of computation can often be generalized into DAGs, many of

these systems could be thought of as broadly-expressive DAG abstractions.

Makeflow [120] is a workflow system that exposes a less featureful language as its interface. Based on traditional UNIX make [46], it is intended as an intermediate interface – more general than the specific abstractions in this dissertation, but using a language that users may be more familiar with and thus more likely to try than the workflow languages above. Makeflow described DAG workflows in a dependency controlled list of rules: a user specifies a target to be created, the dependencies it requires, and a command to create the rule's target. The Makeflow engine concretizes the abstract dependencies for a rule and executes the jobs that produce the target (using local multicore processors or remote distributed systems resources). Makeflow will not continue on to the next rule until its dependencies (often a previous target) are produced.

Other workflow systems emphasize other elements of DAG workflows. Pegasus [38], for instance, is a cluster solution that focuses on data deployment. The Pegasus engine converts an abstract DAG into a concrete DAG by locating the various data dependencies and inserting operations to stage input and output data. This DAG is then given to an execution service such as DAGMan [121] for execution. DAGMan itself acts as a metascheduler similar to Makeflow, managing the submission of jobs to Condor based on DAG dependencies.

2.3 Distributed Computing Abstractions

Distributed computing abstractions are intended to allow users to specify workloads in a way that is natural and simple, without concerning themselves with the details of how the workload will be executed on the system. The specificity to a particular pattern of computation or class of workload allows the abstraction to model the system and execute the workload in an efficient manner. Not surprisingly, then, abstractions are

less general than the workflow languages discussed above. Software abstractions are akin to hardware such as systolic arrays [76] and Kress arrays [58], which are highly specialized machines that efficiently complete specific parallel tasks.

Because the next step up in scale beyond a single processor or multicore server is a small cluster, this environment has been a fertile ground for development of computing abstractions. The most basic of these are cluster-wide versions of software systems that abstract away complexities from the user. For example, the simple abstraction provided by file systems has been extended on data-intensive clusters to provide interfaces for search [66], or for distributed data structures like multidimensional arrays [15], matrices [87], trees [80], and hashtables [55]. Similarly, database abstractions common in the implementation of a single local database can be applied across a cluster for performance or efficiency considerations, including sorting record-oriented data [10], and managing table-relational data [26].

Beyond filesystem and database abstractions, a simple general computing abstraction is Bag-of-Tasks [11, 32], in which independent tasks are submitted and executed in parallel. Although primitive, Bag-of-Tasks is so general that is has been the focus for many different types of distributed systems research (scheduling [14, 77], QoS [128], scalability [33], etc.) and is the paradigm employed by many volunteer computing projects such as SETI@Home [8], Folding@Home [126], and Distributed.Net [40].

Other abstractions are designed for higher-level computing problems on clusters and grids. For example, there are abstractions that focus on data management in a cluster, often to collocate data and computation. Although the full scope of Pegasus [38] is a workflow system, when deployed as a end-user data-deployment or data-realization system it could also be considered a cluster data abstraction. Chimera [48] also is a data-realization abstraction, which provides an interface for a user to access data from

a cluster without having to manage staging the data to the cluster beforehand. The user requests the data, which Chimera provides either from a local copy already on the cluster, a remote storage device, or a on-demand data creation pipeline.

Map-Reduce [37] is perhaps the best-known abstraction for distributed computation. The Map-Reduce abstraction considers both the data and computation needs of a workload. The user specifies two functions that transform the data into intermediate sets of name-value pairs (map) and summarize the data into one or more final sets (reduce), respectively. Map and similar operations have been available in functional programming languages such as LISP [115] for many years, and have been considered core operations for parallel programming [16, 71]. The Map and Reduce pairing is also evocative of operations long-seen in databases (several of which are described in Sokolinsky's survey of parallel database architectures [113]). The Map-Reduce abstraction is well suited for analyzing and summarizing large amounts of data, and has a number of implementations [29, 106], of which Hadoop [1] is the most widely deployed, and extensions, such as the data-parallel applications of DryadLINQ [67]. If the desired computation can be expressed in this form, either explicitly using map and reduce operations or in a high level language [97] that defines more complicated combinations of operations, then the computation can be scaled up to thousands of nodes.

FREERIDE [53] is a common abstraction for the parallelization of a variety of data mining workloads. The framework is built on the recognition that basic parallel implementations of many data mining tasks are similar in structure, and thus a framework may exploit parallelism at several well-recognized stages in the computation of a high-level defined workload. But for the same reason, it is also difficult to manage with remote data access or beyond a small number of nodes. FREERIDE-G [52] is an extension on FREERIDE that adapts to larger workloads with data stored in remote

repositories. The framework is a client-server middleware that is responsible for data retrieval from repository disks, distribution to the compute nodes, actual execution of the computation, and data caching on compute nodes where appropriate. FREERIDE-G has been used to complete data mining workloads that scale to several gigabytes of data on small clusters.

CHAPTER 3

ENVIRONMENT AND MIDDLEWARE

Campus grids are made up of both dedicated grid resources (often provided by an institution's research computing facility), and non-dedicated scavenged shared resources. These non-dedicated resources may come from underutilized research clusters owned by faculty, idle desktop machines in classrooms, labs, and offices, or various other assorted machines. Even the non-dedicated machines are often idle up to 90% of the day, providing valuable resources to the campus grid. The machines may vary significantly in capability – from the newest state of the art many-core machines to processors that may be 5-to-10 years old – and are primarily commodity systems with no extraordinary guarantee of reliability. Further, they may be removed from the pool as a matter of policy: for shared resources, the resource owners must be given complete priority to their own systems. Thus, a common policy within a campus grid is preemption [108], in which if a resource owner requires access to a machine, the campus grid job is evicted.

These machines generally have a standard campus internet connection (currently 100Mbps-10Gbps are common), but may reside on a complicated campus area network that has numerous subnets and switches that may be wildly disparate in their capacities and utilization. The machines on the campus grid may or may not have a shared filesystem, and even when available the fileservers are intended for general campus use instead of exclusive service for campus grid computing.

Campus grid resources are managed by middleware such as Condor [121], SGE [49], and Globus [47]. Though batch schedulers allow jobs to specify the exact resources on which to run, the most common mode of execution is for users to specify only general requirements (for example, operating system and minimum RAM) and allow the scheduler to negotiate where the job will run. In this case, the high-level user interaction with the campus grid is not unlike cloud computing [130].

3.1 Campus Grid Challenges

If workloads were to be completed on a dedicated cluster owned by one user on a switched network, efficient use of resources might not be a concern. However, in a large campus grid shared by many users, a poorly configured workload will consume resources that might otherwise be assigned to waiting jobs. If the workload makes excessive use of the network, it may even halt other unrelated tasks, thus hurting the performance of other users of the system.

Many distributed computing problems run up against the same set of challenges on campus grids, and these obstacles are common to the problems discussed in this work:

Number of Compute Nodes. It is easy to assume that more compute nodes is automatically better. This is not always true. In any kind of parallel or distributed problem, each additional compute node presents some overhead in exchange for extra parallelism. Data must be transferred to that node by some means, which places extra load on the data access system, whether it is a shared filesystem or a data transfer service. More parallelism means more concurrently running jobs for both the engine and the batch system to manage, and a greater likelihood of a node failing, or worse, concurrent failures of several nodes, which consume the attention (and increase the dispatch latency) of the queuing system. For many I/O intensive problems, it may only

make sense to harness ten CPUs, even though hundreds are available.

Data Distribution. After choosing the proper number of servers, it must then be determined how to get the data to each computation. A campus grid usually makes use of an institutional file server or the submitting machine as a file server, as this makes it easy for programs to access data on demand. Often, however, 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 it is much easier to scale up the CPUs of a campus grid than it is to scale up the capacity of a central file server, so 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. If the same input data will be re-used many times, then it makes sense simply to store the inputs on each local disk, getting better performance and scalability. Many dedicated clusters provide fixed local data for common applications (e.g. genomic databases for BLAST [7]). However, in a shared computing environment, there are many different kinds of applications and competition for local disk space, so the system must be capable of adjusting the system to serve new workloads as they are submitted.

Dispatch Latency. The cost of dispatching a job within a campus grid is surprisingly high. Dispatching a job from a queue to a remote CPU requires many network operations to authenticate the user and negotiate access to the resource, synchronous disk operations at both sides to log the transaction, data transfers to move the executable and other details, not to mention the unpredictable delays associated with contention for each of these resources. When a system is under heavy load, dispatch latency can easily be measured in seconds. For batch jobs that intend to run for hours, this is of little

concern. But, for many short running jobs, this can be a serious performance problem. Even under the assumption that a system has no contention for resources and a relatively fast dispatch latency of one second, it would be foolish to run jobs lasting one second: one job would complete before the next can be dispatched, resulting in only one CPU being kept busy. Clearly, there is an incentive to keep job granularity large (relative to total available parallelism) in order to hide the worst case dispatch latencies and keep CPUs busy.

Failure Probability. On the other hand, there is an incentive not to make individual jobs too long. Any kind of computer system has the possibility of hardware failure, but a campus grid also has the possibility that a job can be preempted for a higher priority task, usually resulting in a rollback to the beginning of the job on another CPU. Short runs provide a kind of *de facto* checkpointing, as a small result that is completed need not be regenerated. Long runs also magnify heterogeneity in the pool. For instance, jobs that should take 10 seconds on a typical machine but take 30 seconds on the slowest aren't a problem if batched in small sets. The other machines will just cycle through their sets faster. But, if jobs are chosen such that they run for hours even on the fastest machine, the workload will incur a long delay waiting for the final job to complete on the slowest. Another downside to jobs that run for many hours is that it is difficult to discriminate between a healthy long-running job and a job that is stuck and not making progress. An abstraction has to determine the appropriate job granularity, noting that this depends on numerous factors of the job and of the grid itself.

Resource Limitations. Campus grids are full of unexpected resource limitations that can trip up the unwary user. The major resources of processing, memory, and storage are all managed by high level systems, reported to system administrators, and made known to end users. However, systems also have more prosaic resources. Examples are

the maximum number of open files in a kernel, the number of open TCP connections in a network translation device, the number of licenses available for an application, or the maximum number of jobs allowed in a batch queue. In addition to navigating the well-known resources, an execution engine must also be capable of recognizing and adjusting to secondary resource limitations.

Semantics of Failure. In any kind of distributed system, failures are not only common, but also hard to define. If a program exits with an error code, who is at fault? Does the program have a bug, or did the user give it bad inputs, or is the executing machine faulty? Is the problem transient or reproducible? Without any context about the workload and the execution environment, it is almost impossible for the system to take the appropriate recovery action. But, when using an abstraction that regulates the semantics of each job and the overall dataflow, correct recovery can be straightforward.

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

3.2 Architecture for Computing on Campus Grids

The *conventional* implementation for solving problems on a cluster or similar distributed system executes the specification by simply submitting a series of batch jobs that use a central file server to read data on demand and write outputs into files in the ordinary way. The user specifies what jobs to run *by name*. Each job is assigned a CPU, and does I/O calls on demand with a shared file system. The system has no idea what data a job will access until jobs actually begin to issue system calls. Figure 3.1 shows the difference between using this conventional cluster approach and computing with an

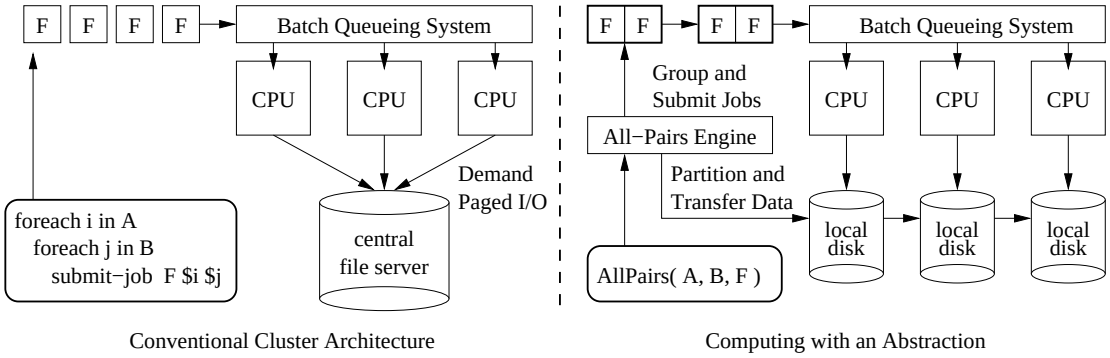


Figure 3.1: Cluster Architectures Compared

When using a conventional computing cluster, the user partitions the workload into jobs, then a batch queue distributes jobs to CPUs where they access data from a central file server on demand. When using an abstraction, the user states the high level structure of the workload, the abstraction engine partitions both the computation and the data access, transfers data to disks in the cluster, and then dispatches computation to execute on the data in place.

abstraction.

When using an *abstraction* implementation, like the three discussed in Chapters 4-6, the user specifies both the data and computation needs, allowing the system to partition and distribute the data in a structured way, then dispatch the computation according to the data distribution. The abstraction implementation exploits the information found in the abstraction by efficiently distributing common data to where it will be used, choosing an appropriate granularity for decomposition, accessing local data copies, and storing the outputs in a manageable way, such as a custom data structure.

Computation within a campus grid can use a hierarchy of abstraction implementations. The abstraction manages the campus-grid-level organization of a workload, but also is applied on each individual resource to manage local resources such as multicore processors. Figure 3.2 shows this hierarchy as a general structure for implementing abstractions that use abstraction engines at multiple layers of a hierarchy. The user in-

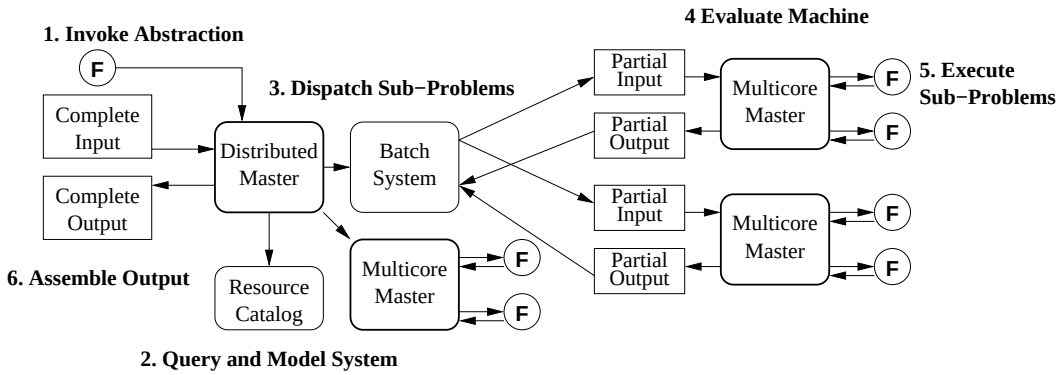


Figure 3.2: Distributed Multicore Implementation
Abstractions can be executed on multicore clusters with a hierarchical technique. The user first invokes the abstraction, stating the input data sets and the desired function. The distributed master process measures the inputs, models the system, and submits 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.

vokes the abstraction by passing the input data and function to the *distributed master* engine. The distributed master engine performs the same tasks it does if submitting functions directly instead of organizing a hierarchical comparison. First, the it examines the size of the input data, the runtime of the function, and models the expected runtime of the workload in various configurations. After choosing a parallelization strategy, the distributed master engine submits sub-problems to the local *batch system*, which dispatches them to available CPUs.

The change in the hierarchical case is that each job consists of a *multicore master* as the executable. When this multicore master is started on a campus grid resource it examines the executing machine, chooses a parallelization strategy, executes the sub-problem, and manages the partial result (either returning to the distributed master or storing directly in the distributed data structure, depending on configuration).

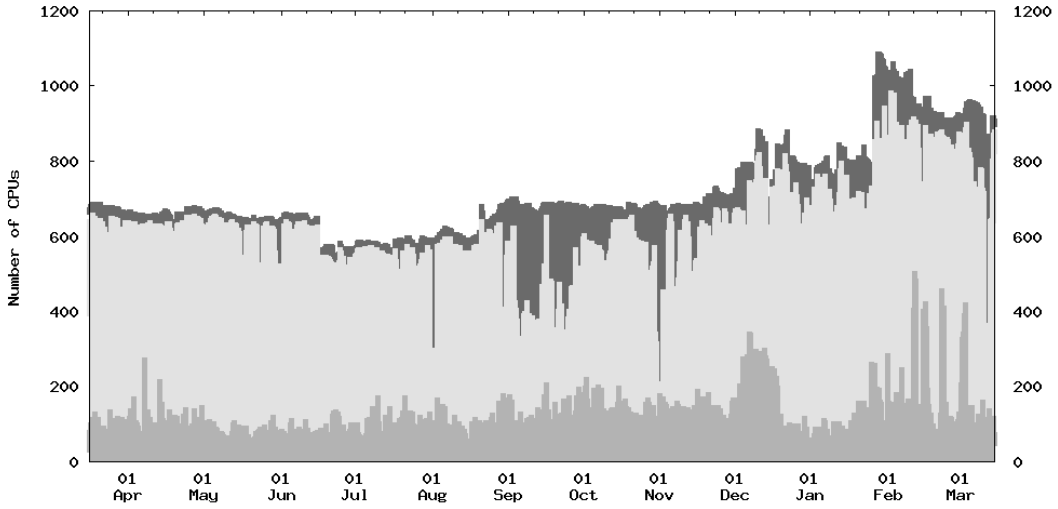


Figure 3.3. Time Variations in a Condor Pool

3.3 Notre Dame Campus Grid

The primary campus grid resources used for this work are from the Notre Dame Condor pool. Over the course of this research, the pool has expanded from approximately 400 cores up to over 1000 cores consistently reporting to the Condor matchmaker.

Additionally, most nodes in the system run a Chirp [123] fileserver to manage access to the local disk. Chirp is a lightweight user-level fileserver that provides a POSIX-like file interface, performs access control, and executes directed transfers between campus grid nodes.

Figure 3.3 shows variations in the number of machines participating in the Condor pool over the course of a full year. The graph plots three curves on top of each other, which creates the equivalent of a stacked column representation. The bottom (medium gray) indicates the of resources that are currently in use by their owners instead of running Condor jobs. Local policy sets machines to this “Owner” state when there is

either significant use of the CPU or keyboard typing detected by the shell. The middle (light gray) indicates the number of machines that are being used by Condor to run jobs. The height of each column is the total number of resources in the pool, so the top (dark gray) indicates the number of resources that are unused – that is, neither in owner state nor being used by Condor. As can be seen, all of these values fluctuate considerably as machines are powered on and off, used during the work day, and harvested for batch workloads. Large dips in the number of resources being used by Condor are often indicative of either a hiccup with the Condor server, or a slight lull as a dominant user's workload is ending and he (or another power user) has not yet submitted the next large workload.

Each processor in the pool has a set of prioritized users, primarily corresponding to the machine's owner. These users, then, have the choice of claiming their resources via direct access to the system or via Condor jobs that will be scheduled there, preempting non-prioritized users' jobs. Where no prioritized user factors into a scheduling decision, priority is determined by the standard Condor priority settings.

At the time that Figure 3.3 was recorded, there were 11 stakeholders who owned the 902 resources operating in the Notre Dame Condor pool. 488 cores were owned and operated by the campus Center for Research Computing, including 164 from the Notre Dame Green Cloud [20]. Assorted nodes owned by the computer science department made up 182 cores, while two computer science professors accounted for 96 and 75 more, respectively. The chemical engineering department contributed 39 cores. The remaining cores in the pool were contributed in small numbers by various other professors.

With that many separate resource contributors, it cannot be a surprise that the machines are highly heterogeneous, as seen in Figure 3.4. New machines are added to

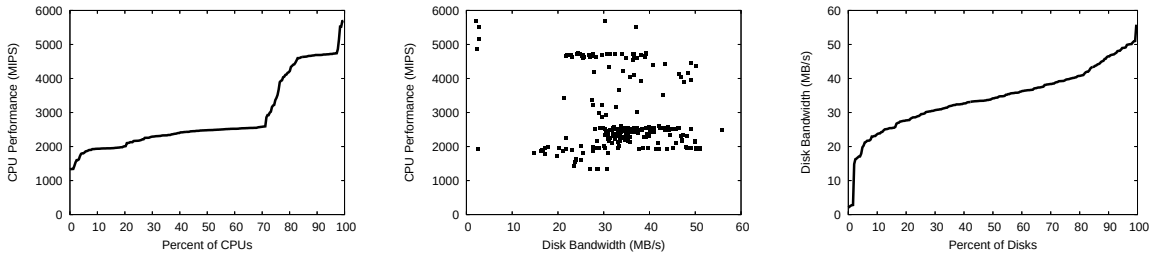


Figure 3.4: Performance Variance in a Campus Grid

A campus grid has a high degree of heterogeneity in available resources. (a) shows the distribution of CPU speed, ranging from 1334 to 5692 MIPS. (c) shows the distribution of disk bandwidth, as measured by a large sequential write, ranging from 2 MB/s (misconfigured controller) to 55 MB/s. (b) shows the weak relationship between CPU speed and disk bandwidth. A fast CPU is not necessarily the best choice for an I/O bound job.

the system and old machines are removed on a daily basis, Often these changes happen as singletons when one machine crashes or is repaired, but sometimes they occur in batches as entire clusters are brought up or decommissioned. As a result, both CPU and disk performance vary widely, with no simple correlation.

3.4 Work Queue

Condor and other batch systems are well-suited and commonly-used to run large numbers of long-running computations such as scientific simulations. But that does not make them well-suited for workloads that have a large number of short-running tasks, very data-intensive tasks, or both of these characteristics. One key reason is that these systems are designed to mediate the needs of many different stakeholders, including the machine owner, the job owner, and the pool manager. This constant coordination means that there is significant latency inherent in the system. Under heavy load, submitting jobs to the queue may take several seconds. And even in a largely unused system, it takes thirty seconds or more from the time a job is submitted until it actually begins running on a machine. This is especially detrimental for short-running jobs, in which

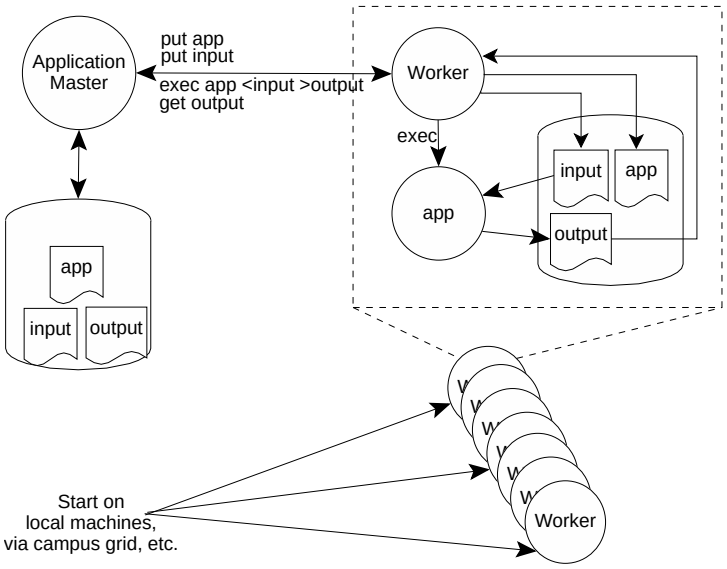


Figure 3.5. Work Queue

the execution time is not much more than the latency.

Additionally, because Condor is careful to clean up thoroughly after a job completes, there is no easy way to maintain state across multiple jobs even if consecutive jobs are directed to the same machine. And if there is contention for resources, although the jobs are fast-running, the large number of small jobs may count against a user's priority, making it even harder to get jobs running.

Work Queue [120] is an intermediate layer of software that is intended to combat these limitations, providing fast execution and data persistence on top of Condor, other batch systems, or arbitrary compute resources. Work Queue's inspiration is a simple observation from Falkon [104] that it is possible to dispatch as a batch job long-running middleware to execute many short-running tasks on a node without having to pay the overhead of submitting each task as a batch job.

3.4.1 Architecture

Work Queue is a general purpose master-worker system with a simple protocol in which the batch job (a "worker" process) connects over the network to a process on a central node (the "master") that dispatches the smaller tasks to run. Figure 3.5 shows how the pieces work together. The worker is a simple standard executable that is the same for all work queue applications. The master is a workload-specific piece of software that coordinates tasks based on the requirements of the workload via the Work Queue API. Most masters perform a variation of the same submit-dispatch-wait-collect cycle.

In practice, the user normally runs the master programs on his or her workstation or server. The worker processes can be submitted to the campus grid, run individually from the command line on nodes where the user has login access, or a combination thereof.

Once running, the worker makes a network connection back to a running Master process, receives files that the master sends over the network, receives a task work order from the master, executes the task as local processes, sends the results back to the master, and waits for the next task.

Likewise, the master determines the work that needs to be done, partitions that work into tasks, assigns a task to a worker, sends the data required by the task to the worker, and collects the results when the worker has finished. Of course, if the batch system decides to evict the worker batch job, it will kill any running processes and delete the local storage. The master is able to detect these evictions, and re-assign tasks to other workers as needed.

When the master collects the results from the worker, it is responsible for verifying the results data and storing it. Making the master responsible for results storage allows

several advantages over having the application or the worker store the results: no globally available shared filesystem is required, worker processes are completely independent of the application, and master processes can interchange methods of verification based on available resources or application-specific workload-level considerations.

3.4.2 Advantages

Task dispatch from the master to a worker is much faster than the dispatch latency in the campus grid queue – milliseconds rather than 30 seconds to several minutes. As contention for campus grid resources increases this advantage is compounded, because subsequent tasks can be started in milliseconds after the first finishes instead of repeating the original latency for starting a batch job.

Another compounded advantage on some systems without preemption is that later pieces of the workload are less influenced by the submitter's degraded priority (which falls as the submitter uses campus grid resources over time). The worker is already running, and thus is not affected by the degraded priority, whereas a new batch job submitted to handle the later work would be chosen to run based on the degraded priority.

Another advantage of Work Queue tasks versus batch jobs is that the workers retain state between tasks, so files needed by many tasks only need to be transferred from the master to a given worker once. And tasks can be assigned to workers based on the amount of data the task requires that is already present on the worker, minimizing data transfer. (By default, Work Queue assigns a task to the next available worker, irrespective of data. However the scheduler can be chosen from among a few simple options using the API).

3.4.3 API

Work Queue masters are written using the Work Queue API. This provides a set of general operations that can be combined to create a complete master process.

The fundamental operations are creating a queue, creating a minimal task data structure, further specifying a task, waiting for tasks to finish, and cleaning up when data structures are no longer needed.

Once a queue has been initialized, the primary driver of the submit-dispatch-wait-collect cycle is the creation of new tasks. The task data structure is instantiated with only the task's command defined. The command is a string representation of a simple shell command line that may consist of a single command or several commands in a pipeline. All other options for a task are separate API functions that modify various fields of this data structure, for example, its `tag` field, which is a user-defined short text description of the task.

All files required by the task – the executable, files passed to the executable on standard input, or other required files such as data, configuration files, or libraries – are specified using the API. Calling the API function to specify a file does not actually place the file on the worker node, rather it adds the requirement to the task data structure's `input_files` field and when the task is dispatched to a node, the master transfers the files as part of the dispatch process.

In order to be transferred to the worker and placed on disk, data does not have to already exist on disk in the master. This is because the API also allows the user to specify that a memory buffer in the master process be copied into a file on the worker.

The last simple field that a user can specify is the output files. This is somewhat unintuitive, as the specification has no bearing on whether the file is actually created. Rather, the `output_files` list field in the task data structure specifies that to be a

properly completed task, the function on the worker must create a certain file, which is then passed back to the master. Thus, the members of the `output_file` list function logically as postconditions for a task. If the file cannot be transferred back to the master when the task completes, the task is marked as a failure.

Once a task is created and all pertinent fields have been specified, the next step for the master's code is to submit the task. This API call does not actually assign the task to a remote worker, rather it submits the task to a queue of tasks. Tasks are not explicitly started via the API, rather they are actually assigned and transferred to workers automatically after the API's `work_queue_wait` instruction is invoked.

The `wait` call indicates that the user is done specifying and submitting tasks for now and will wait for a specified amount of time or until there is a task completed to collect. The `work_queue_wait` operation returns a task data structure (or null if no tasks completed within the wait interval). The master may parse this data structure as needed; generally the postprocessing includes checking the command line's return status for an acceptable value, verifying that the returned output fits the expected format, and copying the output to permanent storage. If the task failed (either in terms of return status or output verification), the user can create and submit a new task with the same specification. The state for a task is finally freed with the `work_queue_task_delete` operation.

Code Excerpt 3.1 shows the typical submit-dispatch-wait-collect cycle within a master via a sample application of compressing a set of files. The queue is created on line 1 with the user specifying the port the queue will listen on for connections from workers. The tasks are created in a loop, where the function `nextFile` is representative of a user-defined function to determine the next quantum of work to be placed into a task. The task data structure is created on line 3 with the `gzip` command line; note

```
1  q = work_queue_create(port,timeout);
2  while(nextFile(input_file,output_file)) {
3      t = work_queue_task_create(
4          ``/usr/bin/gzip -9 < infile > out.gz'');
5      work_queue_task_specify_input_file(t,input_file,
6                                         ``infile'');
7      work_queue_task_specify_output_file(t, ``out.gz'',
8                                          output_file);
9      work_queue_submit(q,t);
10 }
11 while(!work_queue_empty(q)) {
12     t = work_queue_wait(q,10);
13     if(t) work_queue_task_delete(t);
14 }
15 work_queue_delete(q);
```

Code Excerpt 3.1: Submit-Dispatch-Wait-Collect Loop in a Sample Master

that the filenames on the command line are hardcoded into the string for simplicity, but the literal string could be replaced with a constructed string to change the command line between tasks. The input and output files are specified for the task on lines 5 and 7. Line 9 submits the task to the queue.

In this example, all tasks are created before any task will actually begin running, but the creation and execution could be pipelined for workloads with very large numbers of tasks. The `work_queue_empty` call on line 11 checks if there are outstanding tasks (waiting to run, running, or completed), which controls the wait-collect cycle. In this case, because there is no output from `gzip` and the existence of the output file (handled automatically by Work Queue) is sufficient, a task can be deleted as soon as it has been collected by the master. Once all tasks have been completed, the queue can be freed and the master moves on to any application-specific post-processing.

3.4.4 Fast Abort

A large computation can often be significantly slowed down by stragglers. Although slow tasks impact any workload, two particularly noticeable cases are when there are other tasks with dependencies on the slow task, or at the end of a workload, when a slow task is continuing far beyond the completion times of the entire rest of the workload.

Work Queue keeps statistics on task execution times across the workload, and contains a mechanism, "Fast Abort", that proactively cancels and reassigns tasks that have run too long. In initial versions of Work Queue, the criterion for having "run too long" was a parameter of the system, however in the current version allows the user to specify his tolerance for stragglers (defined as a multiplier of the average task completion time, or even a complete deactivation of Fast Abort) via the API.

Fast Abort is studied in considerable detail for the Wavefront and Makeflow abstractions in [133]. Section 5.4.4 evaluates activating Fast Abort near the end of a sequence alignment workload to lessen the chance of one or more slow machines running tasks far beyond the completion of all other tasks in the workload.

CHAPTER 4

ALL-PAIRS ABSTRACTION

4.1 The All-Pairs Problem

The All-Pairs problem is an example of a computation that is trivial to implement for a small computation on one computer, but has applications that would require gigabytes of data running on hundreds of computers for several days. This chapter discusses the aspects of the problem that make implementation non-trivial for scientific computing users, the design and implementation of an abstraction for the problem, and results with the abstraction for applications in several domains.

The All-Pairs problem is easily stated:

All-Pairs(set A, set B, function F) returns matrix M:
Compare all elements of set A to all elements of set B
via function F, yielding matrix M, such that
M[i,j] = F(A[i],B[j]).

This problem is also known as the *Cartesian product* or *cross join* of sets A and B. Variations of All-Pairs occur in many branches of science and engineering, where the goal is either to understand the behavior of a newly created function F on sets A and B, or to learn the covariance of sets A and B on a standard inner product F. The function is sometimes symmetric, in which cases it is enough to compute one half of the matrix using the custom comparison function.

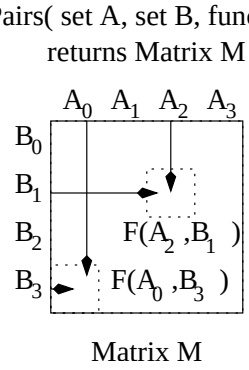


Figure 4.1. The All-Pairs Problem

The All-Pairs abstraction compares all elements of sets A and B together using a custom function F, yielding a matrix M where each cell is the result of $F(A[i], B[j])$.

Solving an All-Pairs problem seems simple at first glance. The typical user constructs a standalone program `F` that accepts two files as input, and performs one comparison. After testing `F` on small datasets on a workstation, he or she connects to the campus grid, and runs a script like this:

```
foreach $i in A
    foreach $j in B
        submit_job F $i $j
```

From the perspective of someone who knows how to program but is not an expert in distributed systems this is a perfectly rational way to construct a large workload, because one would do exactly the same thing in a sequential or parallel programming language on a single machine. Unfortunately, it will likely result in very poor performance for the user due to all the challenges of computing on a campus grid discussed in Chapter 3. Figure 4.2 shows a real example of the performance achieved by a user that attempted exactly this procedure at on the Notre Dame campus grid, in which 250

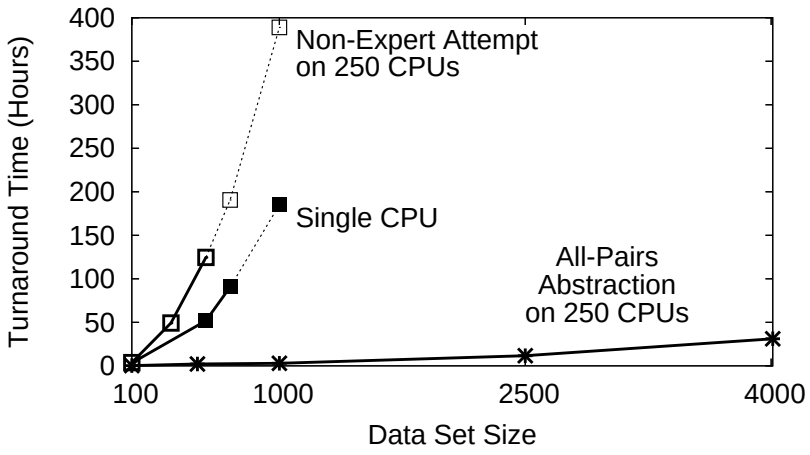


Figure 4.2. Performance of All-Pairs Problem Solutions

The graph shows the performance of an All-Pairs problem on a single machine, on 250 CPUs when attempted by a non-expert user, and on 250 CPUs when using the optimized All-Pairs abstraction.

CPUs yielded *worse* than serial performance.

4.2 Applications

All-Pairs problems occur in several different computing fields. The All-Pairs abstraction has been run and measured on the following applications:

Biometrics is the study of identifying humans from measurements of the body, such as photos of the face, recordings of the voice, or measurements of body structure. A recognition algorithm may be thought of as a function that accepts e.g. two face images as input and outputs a number between zero and one reflecting the similarity of the faces. When a researcher invents a new algorithm for face recognition and writes the code for a comparison function, the accepted evaluation procedure in biometrics is to acquire a known set of images and compare all of them to each other using the

function, yielding a *similarity matrix*. Multiple matrices generated on the same dataset can be used to quantitatively compare different comparison functions.

The biometrics All-Pairs workload benchmark considered here is the comparison of 4010 images of 1.25MB each from the Face Recognition Grand Challenge [96] to all others in the set. This application uses functions that range from 1-20 seconds of compute time, depending on the algorithm in use. This workload requires 185 to 3700 CPU-days of computation, so it must be parallelized across a large number of CPUs in order to make it complete in reasonable time. Unfortunately, each CPU added to the system also needs access to the 5GB of source data. If run on 500 CPUs, the computation alone could be completed in 8.8 hours, but it would require 2.5TB of I/O. Assuming the filesystem could keep up, this would keep a gigabit (125MB/s) network saturated for 5.8 hours, rendering the grid completely unusable by anyone else. Addressing the CPU needs with massive parallelism simply creates a new bottleneck in I/O.

Data Mining is the study of extracting meaning from large datasets. One phase of knowledge discovery is reacting to bias or other noise within a set. In order to improve overall accuracy, researchers must determine which classifiers work on which types of noise. To do this, they use a distribution representative of the data set as one input to the function, and a type of noise (also defined as a distribution) as the other. The function returns a set of results for each classifier, allowing researchers to determine which classifier is best for that type of noise on that distribution of the validation set.

Bioinformatics is the use of computational science methods to model and analyze biological systems. Genome assembly [59, 65] remains one of the most challenging computational problems in this field. A sequencing device can analyze a biological tissue and output its DNA sequence, a string on the set [AGTC]. However, due to physical constraints in the sequencing process, it is not produced in one long string, but in

tens of thousands of overlapping substrings of hundreds to thousands of symbols. An assembler must then attempt to align all of the pieces with each other to reconstruct the complete string. All-Pairs is a natural way of performing the first step of assembly. Each string must be compared to all others at both ends, producing a very large matrix of possible overlaps, which can then be analyzed to propose a complete assembly. Additional All-Pairs applications in bioinformatics are discussed in [102], which notes that All-Pairs (or "doubly data parallel") problems are common in biology.

Other Problems. Some problems may *appear* to fit the common All-Pairs pattern, but may be algorithmically reducible to a smaller problem via techniques such as data clustering or filtering [9, 13, 43, 66]. In these cases, the user's intent is *not* All-Pairs, but sorting or searching, and thus other kinds of optimizations apply. In the All-Pairs problems that are outlined above, it is necessary to obtain *all* of the output values. For example, in the biometric application, it is necessary to verify that like images yield a good score and unlike images yield a bad score. The problem requires brute force, and the challenge lies in providing interfaces to execute it effectively.

4.3 Implementation

The goal of the All-Pairs abstraction is to provide the user an interface such that he can invoke All-Pairs as follows:

```
AllPairs SetA SetB Function Matrix
```

where `SetA` and `SetB` are text files that list the set of files to process, `Function` is the function to perform each comparison, and `Matrix` is the name of a matrix where the results are to be stored.

In the initial versions of the All-Pairs engine, the user's function was required to be essentially a single-CPU implementation of All-Pairs. That is, `Function` is provided

by the end user, and may be an executable written in any language, provided that is has the following calling convention:

```
Function SetX SetY
```

where `SetX` and `SetY` are text files that list a set of files to process, resulting in a list of results on the standard output that name each element compared along with the comparison score:

```
A1 B1 0.53623
A1 B2 2.30568
A1 B3 9.19736
...
```

This was good for performance, because the actual execution time of of a single comparison could be significantly faster than the time needed to invoke an external program. It also improved usability, because the user could easily transition from a small job run on a workstation to a large job run on the campus grid. However, the downside was that while usability improved, the user still was required to rewrite his serial code, and the overall performance still hinged on the user's single-CPU All-Pairs implementation's efficiency.

An improved version of All-Pairs is more amenable to using absolutely unmodified versions of functions that operate on exactly two input items (instead of two sublists of items). To do this it uses a hierarchical scheme in which the abstraction engine coordinates the entire workload, then submits batch jobs that are local masters for the system that can instantiate the computation efficiently on each individual resource in the campus grid. This retains the usability aspect because the original serial 1-versus-1 comparison function can be used in an unmodified form. Although there are multiple

function calls instead of a single call, performance can actually be improved by clever design of the local master (particularly in non-trivial environments such as multicore computers or a memory-bound problem) [132].

Regardless of the version of the function requirements, the term *function* is used 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 abstraction engine relies on its knowledge of the function inputs and the problem structure to provide the necessary data to each node efficiently and arrange the output to be collected efficiently.

Now consider the implementation of All-Pairs in terms of the four stages shared by many distributed computing abstractions and introduced in Chapter 1. For All-Pairs, the problem is modeled to determine an appropriate number of resources to use, input data is distributed to all compute nodes in an efficient manner, the computation is organized by subproblem and computed in a two-level hierarchy, and the output results are stored to a distributed data structure.

4.3.1 Modeling the System

In order to decide how many CPUs to use and how to partition the work, there must be an approximate model of system performance. In a conventional system, it is difficult to predict the performance of a workload, because it depends on factors invisible to the system, such as the detailed I/O behavior of each job, and contention for the network. Both of these factors are minimized when using an abstraction that exploits initial efficient distribution followed by local storage access instead of remote network access.

Previous researchers have studied All-Pairs theoretically [124] and on small clusters [28]. Unlike in those highly predictable environments, achieving optimal perfor-

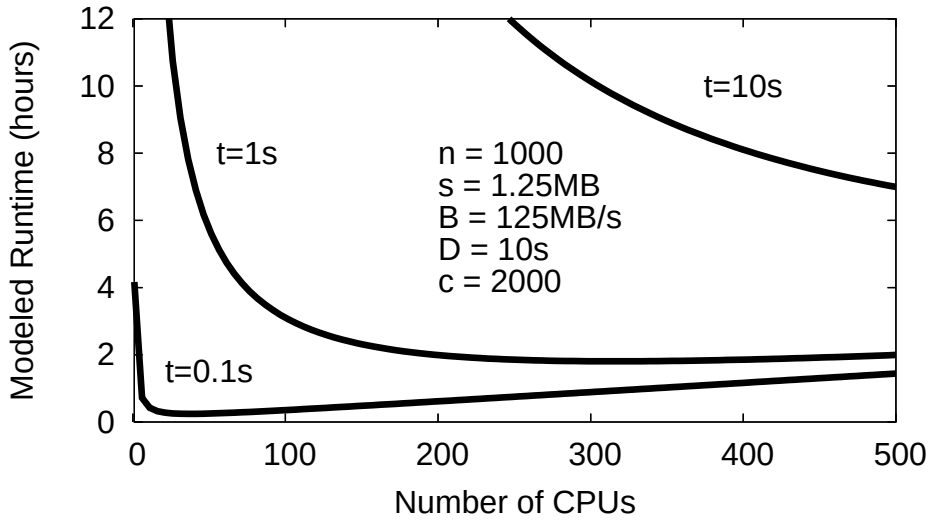


Figure 4.3. Three Workloads Modeled

This graph compares the modeled runtime of three workloads that differ only in the time (t) to execute each function. In some configurations, additional parallelism has no benefit.

mance is essentially impossible in a large dynamic heterogeneous system where nothing is under the user's direct control. Rather, the best that the abstraction can aim for is to avoid disasters by choosing the configuration that is optimal within the model.

The (distributed master) engine measures the input data to determine the size **s** of each input element and the number of elements **n** in each set (for simplicity, assume here that the sets have the same number and size elements). The provided function is tested on a small set of data to determine the typical runtime **t** of each function call. Several fixed parameters are coded into the abstraction by the system operators: the bandwidth **B** of the network and the dispatch latency **D** of the batch software. Finally, the abstraction must choose the number of function calls **c** to group into each batch job, and the number of hosts **h** to harness.

The time to perform one transfer is simply the total amount of data divided by the

bandwidth. Distribution by a spanning tree (described below) has a time complexity of $O(log_2(h))$, so the total time to distribute two data sets is:

$$T_{distribute} = \frac{2ns}{B} log_2(h)$$

The total number of batch jobs is n^2/c, the runtime for each batch job is $D + ct$, and the total number of hosts is h, so the total time needed to compute on the staged data is:

$$T_{compute} = \frac{\frac{n^2}{c}(D + ct)}{h}$$

However, because the batch scheduler can only dispatch one job every D seconds, each job start will be staggered by that amount, and the last host will complete $D(h-1)$ seconds after the first host to complete. Thus, the total turnaround time is:

$$T_{turnaround} = \frac{2ns}{B} log_2(h) + \frac{n^2}{ch}(D + ct) + D(h - 1)$$

Now, the abstraction using a hillclimbing heuristic may choose the free variables c and h to minimize the modeled turnaround time. Some constraints on c and h are necessary. Clearly, h cannot be greater than the total number of batch jobs or the available hosts. To bound the cost of eviction in long running jobs, c is further constrained to ensure that no batch job runs longer than one hour. This is also helpful to enable a greater degree of scheduling flexibility in a shared system where preemption is undesirable. In the original implementation, c was chosen as a multiple of a result row to simplify job partitioning, however in the hierarchical implementation that has replaced it, c is only constrained to be a valid number of cells in a rectangle within the results matrix.

Using the function execution time t, the engine can also model a workload run locally. This allows the engine to compare the predicted turnaround time running locally

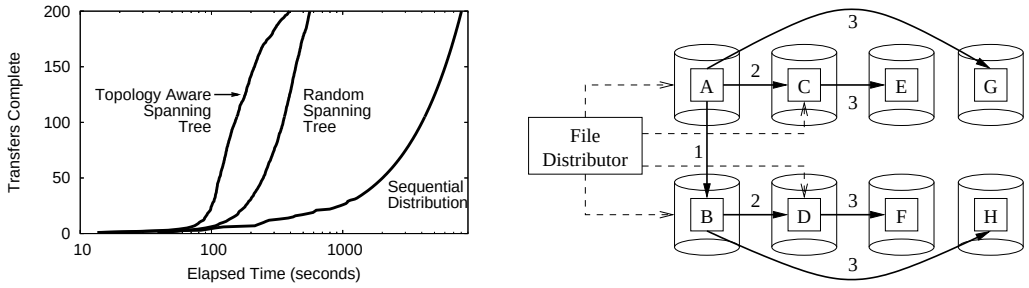


Figure 4.4: File Distribution via Spanning Tree

An efficient way to distribute data to many nodes of a system is to build a spanning tree. In the example on the right, a file distributor initiates transfers as follows: (1) A transfers to B, (2) AB transfer to CD, (3) ABCD transfer to EFGH. The graph on the left compares the performance of transferring data to the first 200 available hosts using sequential transfers, a random spanning tree, and a topology aware spanning tree.

or on the distributed resources and choose the better option. Because the distributed version is more prone to variability, the engine is built to prefer the local execution until there is a significant predicted benefit from distributing. The accuracy of this automatic changeover is discussed in further detail below.

Figure 4.3 shows the importance of modeling the orders of magnitude within the abstraction. Suppose that All-Pairs is invoked for a comparison of 1000x1000 objects of 1.25MB each, on a gigabit Ethernet (125MB/s) network. Depending on the algorithm in use, the comparison function could have a runtime anywhere between 0.1s and 10s. If the function takes 0.1 seconds, the optimal number of CPUs is 38, because the expense of moving data and dispatching jobs outweighs the benefit of any additional parallelism. If the function takes one second, then the system should harness several hundred CPUs, and if it takes ten, all available CPUs.

4.3.2 Managing the Input Data

The model determines how many computing resources should be used. To choose the nodes, the engine consults a *resource catalog* to determine the available machines. Because the number of computing resources can scale far faster than the capacity of a central file server, and the often data-intensive nature of the All-Pairs problem, the All-Pairs implementation prestages data to the computing nodes.

This approach is different from the conventional method, in which batch systems are usually coupled with a traditional file system such that when a job issues I/O system calls, the execution node is responsible for pulling data from the storage nodes into the compute node. Because the abstraction is given the data needs of the workload in advance, it can implement I/O much more efficiently. To deliver all of the data to every node (a design decision discussed below at the end of this subsection), the engine can build a spanning tree which performs streaming data transfers and completes in logarithmic time. Exploiting the local storage on each node avoids the unpredictable effects of network contention for small operations at runtime.

A *file distributor* component is responsible for pushing all data out to a selection of nodes by a series of directed transfers forming a spanning tree with transfers done in parallel. Figure 4.4 shows the algorithm, which is a greedy work list. The data is pushed to one node, which then is directed to transfer it to a second. Two nodes can then transfer in parallel to two more, and so on. The directed transfers are executed by the Chirp fileservers running on each node.

Dealing with failures is a significant concern for pushing data. Failures are quite common and impossible to predict. A transfer might fail outright if the target node is disconnected, misconfigured, has different access controls or is out of free space. A transfer might be significantly delayed due to competing traffic for the shared net-

work, or unexpected loads on the target system that occupy the CPU, virtual memory, and filesystem. Delays are particularly troublesome, because it is uncertain whether a problem will be briefly resolved or delay forever.

A greedy work list is naturally fault tolerant. If any one transfer fails outright or is delayed, the remaining parallel branches of the spanning tree will reach other parts of the campus grid. Because individual nodes that report as available to the catalog may become unavailable during distribution (delaying completion or causing the distribution to to complete with fewer nodes than requested), it is often more effective to simply give the distributor a target number h. This way, the distribution can continue until data has been placed on h hosts, and then it will cancel any outstanding transfers and list the hosts actually reached.

Of course, a campus grid does not have a uniform network topology. Transfers may be fast between machines on one switch, but become slower as transfers reach across routers and other network elements. In the worst case, the file distributor might randomly arrange a large number of transfers that saturate a shared network link, rendering the system unusable to others.

To prevent this situation, the file distributor can be provided with a simplified topology in the form of a "network map", which simply states which machines are connected to the same switch. The file distributor algorithm is slightly refined in two ways. First, the distributor will prefer to transfer data between clusters before transferring within clusters, assuming that the former are slower and thus should be performed sooner so as to minimize the makespan. Second, the distributor will not allow more than one transfer in or out of a given cluster at once, so as to avoid overloading shared network links.

The performance of file distribution is shown in Figure 4.4. Here, a 500MB dataset

is transferred to the first 200 available hosts on the campus grid, recording the elapsed time at which each single transfer is complete. Each of three distribution techniques is performed ten times, and the average at each host is shown. Sequential distribution takes 8482 seconds to complete. A fully random spanning tree takes 585 seconds, while a topology aware tree takes 420 seconds.

To conclude this section, consider whether it is really necessary to distribute *all* of the data to every node? An alternative would be to distribute the minimum amount of data to allow each node to run its job. In fact, however, distributing all of the data via spanning tree is *faster* than distributing the minimum fragment of data from a central server, and it also improves the fault tolerance of the system. Table 4.1 summarizes the result.

Proof: Consider a cluster of h reliable hosts with no possibility of preemption or failure. The *fragment method* minimizes the amount of data sent to each host by assigning each host a square subproblem of the All-Pairs problem. Each subproblem requires only a fragment of data from each set to complete. So, both data sets are divided into into f fragments, where $f = \sqrt{h}$. Each host then needs n/f data items of size s from each set delivered from the central file server. Dividing by the bandwidth B yields the total time to distribute the data fragments:

$$T_{fragment} = \frac{2nsh}{Bf} = \frac{2ns}{B}\sqrt{h}$$

Compare this to the *spanning tree method* described above:

$$T_{distribute} = \frac{2ns}{B} log_2(h)$$

Because $log_2(h) << \sqrt{h}$, the spanning tree method is faster than the minimum fragment method for any number of hosts in a reliable cluster without preemption or

failure. Of course, the total amount of data transferred is higher, and the dataset must fit entirely on a sufficient number of disks. However, as commodity workstation disks now commonly exceed a terabyte and are typically underutilized [42], this has not been a significant problem.

In actuality, however, a campus grid is a highly unreliable environment. The fragment method is even worse when allowing for the possibility of failures. With failures during distribution, a given transfer must be retried in either case, but the spanning tree has the advantage of allowing multiple attempts at once. And because it delivers the minimum amount of data to any one host, there is no other location a job can be scheduled if the data is not available. With the spanning tree method, any job can run on any node with the data, so the entire computation phase is more naturally fault tolerant.

TABLE 4.1

COMPARISON OF DATA DISTRIBUTION TECHNIQUES

Distribution Method	**Total Time**	**Data Transferred**	**Fault Tolerant?**
Fragment	$(ns/B)\sqrt{h}$	$ns\sqrt{h}$	No
Spanning Tree	$(ns/B)log_2(h)$	nsh	Yes

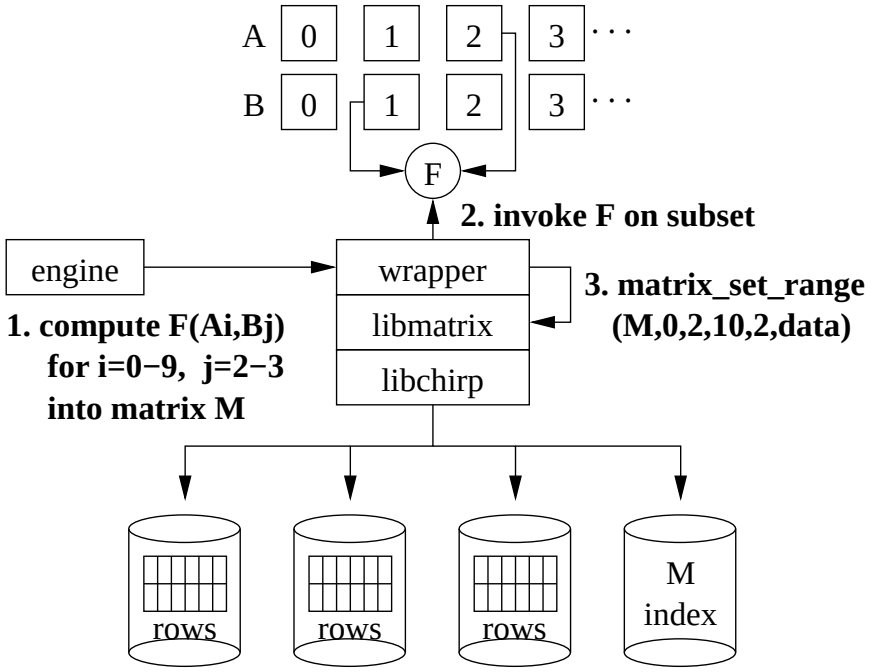


Figure 4.5. Detail of Local Job Execution

The distributed master engine runs on the submitting node, directing each working node to compute a subset of the All-Pairs problem. On each node, the local master invokes the users function, buffers the results in memory, and then updates the distributed data structure with the results.

4.3.3 Coordinating the Computation

After transferring the input data to a suitable selection of nodes, the All-Pairs engine then constructs batch submit scripts for each of the grouped jobs, and queues them in the batch system with instructions to run on those nodes where the data is available. Each batch job consists of the user's function and the local master, shown in Figure 4.5.

Although the abstraction relies heavily on the batch system to manage the workload at this stage, the framework still has two important responsibilities: local resource management and error handling.

The All-Pairs engine is responsible for managing local resources on the submitting

machine. If a workload consists of hundreds of thousands of partitions, it may not be a good idea to instantly materialize all of them as batch jobs for submission. Each materialized job requires the creation of several files in the local filesystem, and consumes space in the local batch queue. Although Condor is capable of queuing hundreds of thousands of jobs reliably, each additional job slows down queue management and thus scheduling performance. When jobs complete, there is the possibility that they produce some large error output or return a core dump after a crash. Instead of materializing all jobs simultaneously, the engine throttles the creation of batch jobs so that they queue only has twice as many jobs as CPUs. As jobs complete, the engine deletes the output and ancillary files to manage the local filesystem.

The engine and the local master together are responsible for handling a large number of error conditions. Again, Condor itself can silently handle problems such as the preemption of a job for a higher priority task or the crash of a machine. However, because it has no knowledge of the underlying task, it cannot help when a job fails because the placed input files have been removed, the execution machine does not have the dynamic libraries needed by the function, or a brief network outage prevents writing results to the matrix. Although events like this sound very odd, they are all too common in workloads that run for days on hundreds of machines. To address this, the local master itself is responsible for checking a number of error conditions and verifying that the output of the function is well formed. If the execution fails, the local master informs the engine through its exit code, and the engine can resubmit the job to run on another machine.

The engine can also handle the problem of jobs that run for too long. This may happen because the execution machine has some hardware failure or competing load that turns a 30 minute job into a 24 hour job. Although the runtime of an arbitrary batch

job is impossible to predict in the general case, the engine has access to a model of the workload, as well as a distribution of runtimes, so it can cancel and resubmit jobs that fall far out of the range of normal execution times. A variant of this procedure, "fast abort", is discussed at greater length in Chapter 3. To improve the "long tail" of jobs at the end of an execution, it could also submit duplicate jobs as in Map-Reduce [37], although this approach has not been attempted.

4.3.4 Managing the Output Data

The output produced by an All-Pairs run can be very large. A 60,000 by 60,000 comparison, approximately the size of the largest production biometric workload in Section 4.5, will produce 3.6 billion results. Each result must, at a minimum, contain an eight-byte floating point value that reflects the similarity of two items, for a total of 28.8GB of unformatted data. If there is additional data such as troubleshooting information for each comparison, the results may balloon to several hundred gigabytes. Users run many variations of All-Pairs, so the system must be prepared to store many such results.

Although current workstation disks are one terabyte and larger, and enterprise storage units are much larger, several hundred gigabytes is still a significant amount of data that must be handled with care. It is not likely to fit in memory on a workstation and applying improper access patterns will result in performance many orders to magnitude slower than necessary. A user that issues many All-Pairs runs will still fill up a disk quite quickly.

The user who applies existing interfaces in the most familiar way will run into serious difficulties. The non-expert users of this work have been inclined to store each result as a separate file. Of course, this is a disastrous way to use a filesystem, because

each eight byte result will consume one disk block, one inode, and one directory entry. If each row of results is stored in a separate file in the function's text output format, this is still storing several hundred gigabytes of data, and still has a sufficiently large number of files that directory operations are painfully slow. Such a large amount of data would require the user to invest in a large storage and memory intensive machine in order to manipulate the data in real time.

Instead, the abstraction must guide users toward an appropriate storage mechanism for the workload. Output from All-Pairs jobs goes to a *distributed data structure* provided by the system. The data structure is a matrix whose contents are partitioned across a cluster of reliable storage nodes maintained separately from the campus grid. Data in the matrix is not replicated for safety, because the cluster is considered an intermediate storage location in which results are analyzed and then moved elsewhere for archival. In the event of failure leading to data loss, the All-Pairs run can easily be repeated.

The full design and performance results for the matrix data structure are reported in [87], but are summarized here.

Because of the underlying storage, row-major access is most efficient: a row read or write results in a single sequential I/O request to one host. Column-major access is still possible: a column read or write results in a strided read or write performed on all hosts in parallel, taking advantage of hardware parallelism. Individual cell reads are also possible, but are inefficient.

4.4 Evaluation and Results

Configurations. Evaluation is based on the two implementations of All-Pairs mentioned above: *abstraction* and *conventional*. For the conventional configuration, the central file server was a dedicated 2.4GHz dual-core Opteron machine with 2GB of

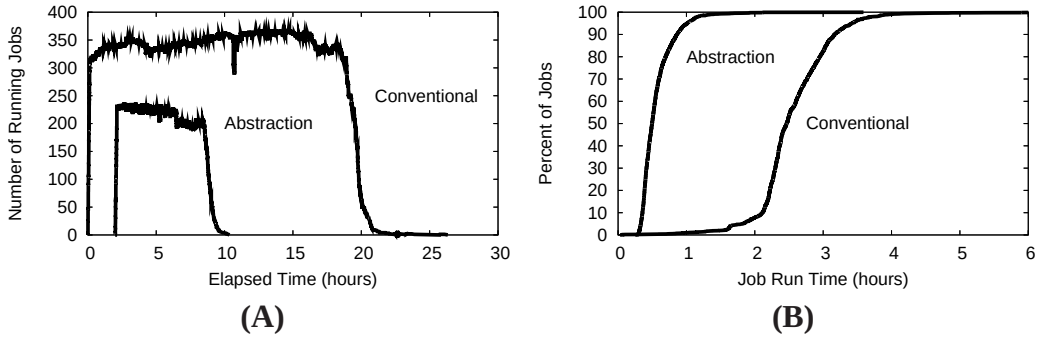


Figure 4.6: Challenges in Evaluating Grid Workloads
Evaluating workloads in a campus grid is troublesome, because of the variance of available resources over time. 4.6(A) shows the CPUs assigned to two variants of the same 2500x2500 All-Pairs problem run in conventional and abstraction modes. 4.6(B) shows the distribution of job run times for each workload. In this case, the abstraction mode is significantly better, but a quantitative evaluation is complicated by the variance in number of CPUs and the long tail of runtimes that occurs in a distributed system. To accommodate this variance, the resource efficiency *described in the text is an effective metric.*

RAM, also running a Chirp fileserver.

Note that these cannot easily be compared against a kernel-level distributed filesystem like NFS [109]. This campus grid spans multiple administrative domains and firewalls; gaining access to modify kernel level configurations is impossible in this kind of environment. Both configurations use the exact same software stack between the end user's application and the disk, differing only in the physical placement of jobs and data. In any case, the precise filesystem hardware and software is irrelevant, because the conventional configuration saturates the gigabit network link.

Metrics. Evaluating the performance of a large workload running in a campus grid has several challenges. In addition to the heterogeneity of resources, there is also significant time variance in the system. The number of CPUs actually plugged in and running changes over time, and the allocation of those CPUs to batch users changes according to local priorities. In addition, the two different modes (*abstraction* and

conventional) will harness different numbers of nodes for the same problem. How can an algorithm be evaluated quantitatively in this environment?

Figure 4.6(A) shows this problem. The graph compares the two configurations of an All-Pairs run of 2500x2500 on a biometric workload. The *conventional* mode uses all available CPUs, while the *abstraction* mode chooses a smaller number. Both vary considerably over time, but it is clear that *abstraction* completes much faster using a smaller number of resources. Figure 4.6(B) shows the distribution of job run times, demonstrating that the average job run time in *abstraction* is much faster, but the long tail rivals that of *conventional*.

To accommodate this, for each result consider two quantitative results. The *turnaround time* is simply the wall clock time from the invocation to completion. The *resource efficiency* is the total number of cells in the result (the number of function invocations), divided by the cumulative CPU-time (the area under the curve in Figure 4.6(A). For both metrics, smaller numbers are better.

Results. Figure 4.7 shows a comparison between the two implementations for a biometric comparing face images of 1.25MB each in about 1s each. For workload above 1000x1000, the abstraction is twice as fast, and four times more efficient. Figure 4.8 shows a data mining application comparing datasets of 700KB in about 0.25s each. Again, the execution time is almost twice as fast on large problems, and seven times more resource efficient on the largest configuration. The third application is a synthetic application with a heavier I/O ratio: items of 12.5MB with 1s of computation per comparison. Although this application is synthetic, chosen to have ten times the biometric data rate, it is relevant as processor speed is increasing faster than disk or network speed, so applications will continue to be increasingly data hungry. Figure 4.9 shows for this third workload another example of the abstraction performing better than

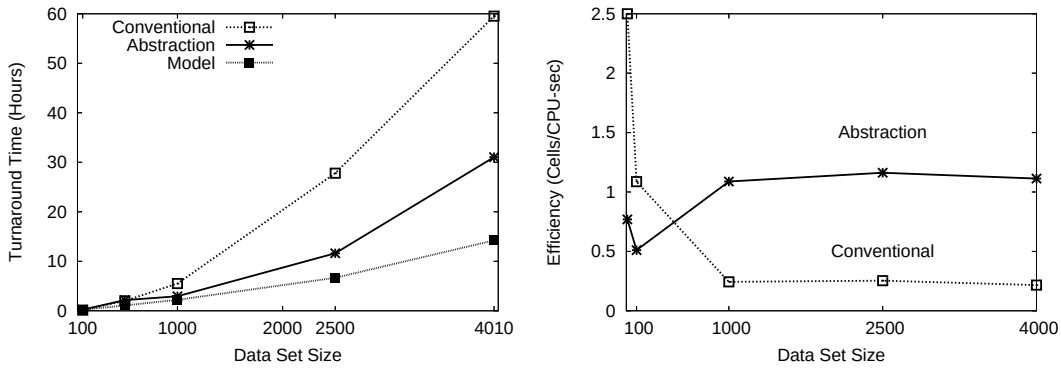


Figure 4.7: Performance of a Biometric All-Pairs Workload
The biometric face comparison function takes 1s to compare two 1.25MB images.

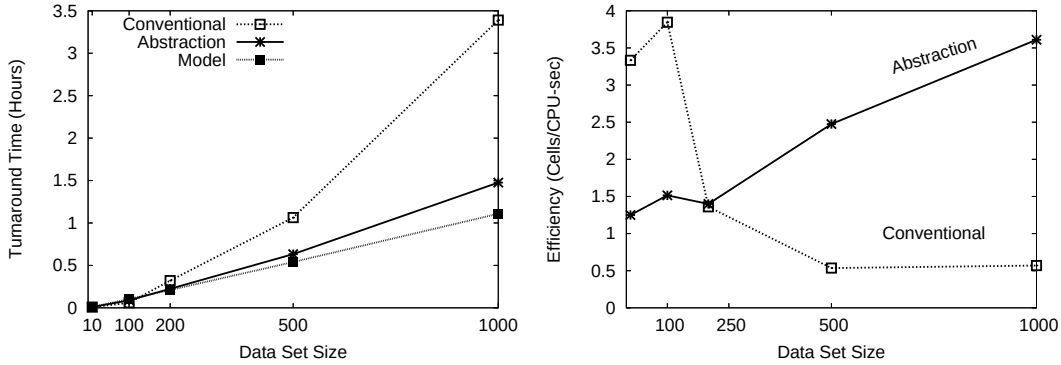


Figure 4.8: Performance of a Data Mining All-Pairs Workload
The data mining function takes .25 seconds to compare two 700KB items.

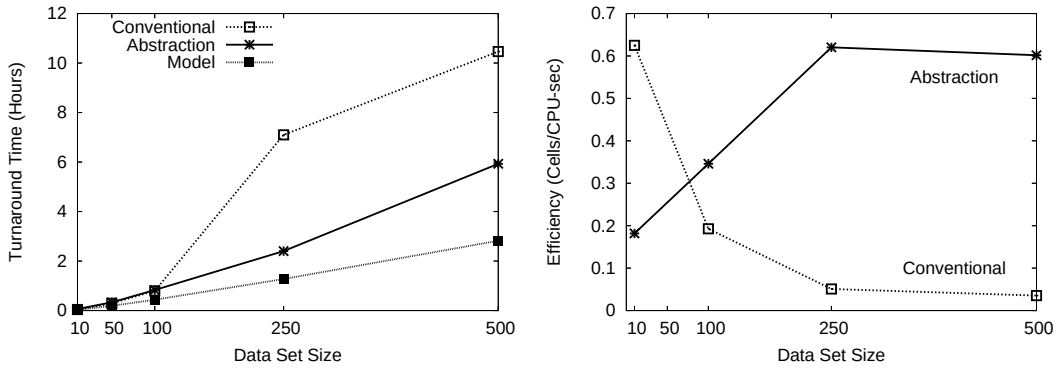


Figure 4.9: Performance of a Synthetic All-Pairs Workload
This function takes 1s for two 12.5MB data items, 10 times the biometric data rate.

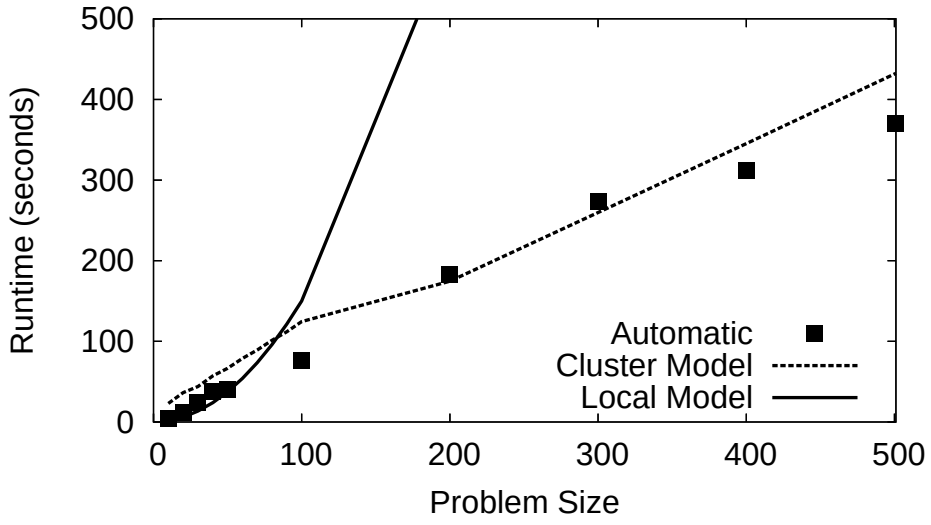


Figure 4.10. Selecting An Implementation Based on the Model

This graph overlays the modeled multicore and cluster performance on problems of various sizes for All-Pairs. 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.

the conventional mode on all non-trivial data sizes.

For comparison, the graphs also show the execution time predicted by the model for the abstraction. As expected, the actual implementation is often much slower than the modeled time, because it does not take into account failures, preemptions, competition for resources, and the heterogeneity of the system.

For small problem sizes on each of these three applications, the completion times are similar for the two data distribution algorithms. The central server is able to serve the requests from the limited number of compute nodes for data sufficiently to match the data staging step in the application.

For larger problem sizes, however, the conventional algorithm is not as efficient because the aggregate I/O rate (hs/t) exceeds the capacity of the network link to the

central file server, which has a theoretical maximum of 125 MB/s. If there were exactly 300 CPUs are in use at once (easily feasible on the campus grid), the aggregate I/O rate is 375 MB/s in Figure 4.7, 820 MB/s in Figure 4.8, and 3750 GB/s in Figure 4.9. To support these data rates in a single file server would require a massively parallel storage array connected to the cluster by a high speed interconnect such as Infiniband. Such changes would dramatically increase the acquisition cost of the system. The use of an abstraction allows us to exploit the aggregate I/O capacity of local storage, thereby achieving the same performance at a much lower cost.

Figure 4.10 compares the multicore and cluster models and demonstrates actual performance achieved when selecting the implementation at runtime. The model is sufficiently accurate that it can be used to choose the appropriate implementation at runtime based on the properties given to the abstraction.

4.5 Production Workloads

This implementation of All-Pairs has been used in a production mode for over two years to run a variety of workloads in biometrics. All-Pairs has been used to explore matching algorithms for 2-D face images, 3-D face meshes, and iris images. The largest single production run so far explored the problem of matching a large body of iris images. A more detailed overview of iris biometrics is given by Daugman [36].

A conventional iris biometric system will take a grayscale iris image and extract a small binary representation of the texture in the iris, called an *iris code* [18]. The iris code is a small (20KB) black and white bitmap designed to make comparisons as fast as possible. To compare two iris codes, the comparison function is the normalized Hamming distance, which measures the fraction of the bits that differ. Two random binary strings would likely differ in about half of their bits, and would therefore have

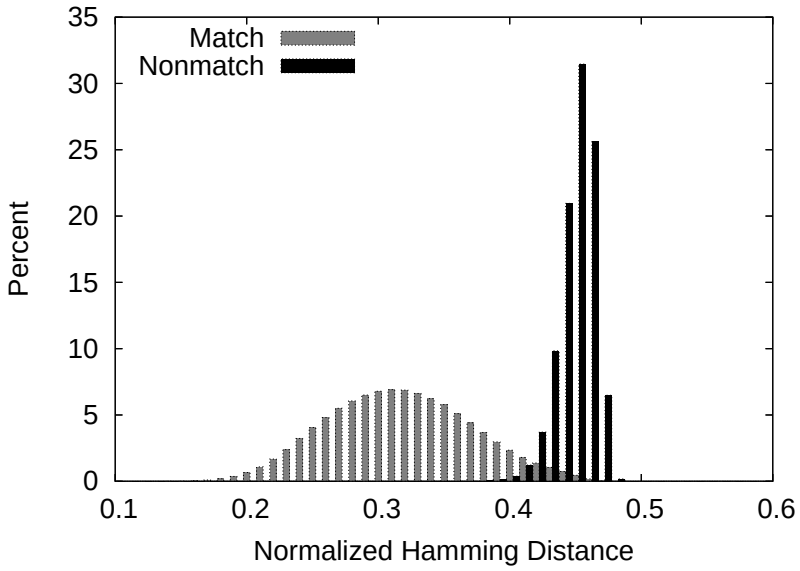


Figure 4.11. Results From Production Run

The results of All-Pairs on 58,639 iris codes. The gray indicates comparison of irises from the same person (match). The black indicates comparison of irises from different people (nonmatch).

a Hamming distance score around 0.5. Two iris codes corresponding to two different images of the same person's eye should not differ in as many bits, and thus have a Hamming distance closer to 0. A comparison between two different images of the same iris is called a *match*, and comparison between images of two different eyes is called a *nonmatch*. Ideally, all match comparisons should yield lower Hamming distance scores than all nonmatch comparisons.

The largest single run computed Hamming distances between all pairs of 58,639 20KB iris codes from the ICE 2006 [89] data set. The next largest publicly available iris data set is CASIA 3 [25], about three times smaller, on which no results have been published on complete comparisons. This is the largest such result ever computed on a publicly available dataset, as of the time of completion.

Figure 4.11 shows the end result of this workload. A histogram shows the frequency of Hamming distances for *matches* and *nonmatches*. As can be seen, the bulk of each curve is distinct, so an expert system for matching might use a threshold of about 0.4 to determine whether two irises are distinct. However, these results also indicate a group of non-matching comparisons that significantly overlap the matches. External examination by a biometrics expert discovered that these low scores occur when one of the images in the comparison is partially occluded by eyelids so that only a small amount of iris is visible and available for the comparison. This observation has has spurred development of mechanisms to account for such irregularities in data acquisition and processing [22]. Without the ability to easily perform large scale comparisons, such an observation could not have been made.

The pool's fastest single machine for this experiment could perform 50 comparisons per second, and would take about 800 days to run the entire workload sequentially. The All-Pairs implementation ran in 10 days on a varying set of 100-200 machines, for a parallel speedup of about 80. The speedup is imperfect because the pool does not uniformly consist of equivalent machines, and because it cannot maintain ideal conditions over the course of ten days due to competition for resources and unreliability of resources. Table 4.2 summarizes all of the failures that occurred over that period, grouped by the component that observed and responded to the failure.

As discussed above, the Condor batch system handles a large fraction of the failures, which are preemptions that force the job to run elsewhere. However, the number of failures handled by the rest of the system is still large enough that they cannot be ignored. All are cases that are not supposed to happen in a well regulated system, but creep in anyhow. Despite extensive debugging and development, the user's function still crashes when exposed to unexpected conditions on slightly different machines. A

TABLE 4.2

SUMMARY OF FAILURES IN PRODUCTION RUN

Failure Type	**Observer**	**Count**
Job killed with signal 15.	engine	4161
Job killed with signal 9.	engine	372
Inputs not accessible.	wrapper	5344
Failed to store output.	wrapper	17
Dynamic linking failed.	wrapper	45
Function returned 255.	wrapper	20
Function returned 127.	wrapper	300
Job preempted.	batch system	14560

number of machines were wiped and re-installed during the run, so input files were not always found where expected. There are a surprisingly large number of instances where the job was forcibly killed with a signal; only a local system administrator would have permission to do so. These data emphasize the point that anything can and will happen in a campus grid, so every layer of system is responsible for checking errors and ensuring fault tolerance – this task cannot be delegated to any one component.

Despite these challenges, this production workload has demonstrated that the All-Pairs abstraction takes a computation that was previously infeasible to run, and makes it easy to execute in a matter of days, even in a uncooperative environment. Using this abstraction, a new computer science graduate student was able to break new ground in biometrics research without first becoming an expert in parallel computing.

CHAPTER 5

SPARSE-PAIRS ABSTRACTION

5.1 Sparse-Pairs Abstract Problem

There are many workloads that involve the comparison of large sets of objects, but do not require *all possible* comparisons. One specific pattern within this group is the Sparse-Pairs problem.

Sparse-Pairs(**data** A, **data** B, **function** F(**data** x, **data** y), **pairs** P)
returns array R **such that** $F(A[P[i].x], B[P[i].y])$

The Sparse-Pairs abstraction applies a function F to pairs of elements in sets A and B given by the set P, yielding a result set R. Sparse-Pairs fits between the one-dimensional array abstraction of Map, and the two-dimensional array abstraction of All-Pairs. In this way it is a bit like superimposing Bag-of-Tasks on top of the one-dimensional structure of a Map abstraction [120].

Data considerations differentiate Sparse-Pairs from both Map and All-Pairs. Although the pairs are sparse, each sequence is still used many times throughout the workload. Thus, while the pairs to be computed could be written in full to files in which every pair was a single element, and Map could then be run using that input, this is inefficient. And although Sparse-Pairs result is a subset of a corresponding All-Pairs result, it is unnecessary to complete an entire All-Pairs problem for every case of

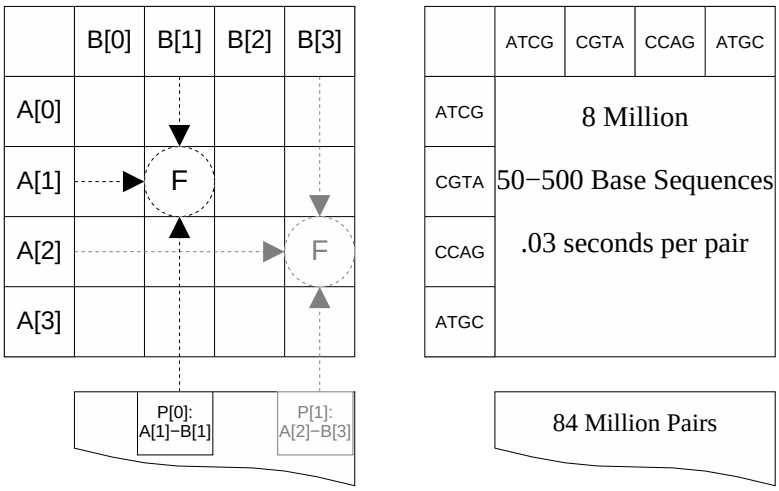


Figure 5.1. The Sparse-Pairs Abstraction Applied to Bioinformatics

Sparse-Pairs, and for particularly sparse sets of pairs, it may be very inefficient to do so even if the All-Pairs abstraction is highly optimized.

Further, even disregarding the problem of unneeded computations, Sparse-Pairs problems also do not have the regular structure that makes All-Pairs easy to interface. The regular structure of All-Pairs allows the interface to the abstraction to require only the function and the names of the full sets. For Sparse-Pairs the usage is less uniform even for the same input set size, thus it is less beneficial to prestage all data to all nodes and assign computation to arbitrary identically prepared resources.

5.2 Application of Sparse-Pairs in Bioinformatics

Many bioinformatics problems are naturally data-parallel and thus lend themselves to distributed computing at large scales [111]. The genome assembly problem presents both naturally data-parallel problems that can be scaled up to thousands of nodes and other problems that use much smaller levels of parallel computation to distribute disk

or memory requirements. This section describes the genome assembly pipeline, the structure, data, and algorithms for a naturally parallel application within that pipeline, and how the Sparse-Pairs can be applied to that application.

5.2.1 Assembly Pipeline

Genome sequencing is the laboratory process of determining an organism's DNA string from a biological sample. A DNA string is a long series of *bases* (A, G, T, and C); however, no current sequencing process is capable of producing an organism's entire string of millions or billions of bases. Instead, the physical process produces a large number of randomly located substrings known as *reads*. Individually, these reads have limited scientific value, as they are very short – 25 to 1000 bases each [98]. *Genome assembly* is the computational process of arranging reads in the correct order to produce the largest possible contiguous strings known as *contigs*. There are many assemblers [12, 60, 63, 88, 100] that solve the problem in a variety of ways – often performing similar conceptual steps, but organizing and naming them differently. For this discussion, the organization will be simplified to three phases: candidate selection, alignment, and consensus. These phases are illustrated in Figure 5.2: (a) is the set of reads produced by genome sequencing; (b) shows the candidate pairs that were determined as potential matches in candidate selection; (c) shows the best overlaps, as determined by alignment; and (d) illustrates ordering the reads based on overlaps at the beginning of the consensus phase.

The *alignment* step is the process of finding overlaps between the suffix of one read and the prefix of another. To ensure that enough reads will overlap, a genome sequencing project oversamples from the DNA in the cell by a factor of 5-10. In principle, every single read should be compared to every other read; however the $O(n^2)$ algorithm is is

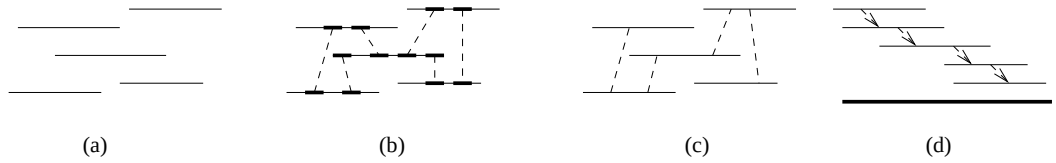


Figure 5.2. Stages of the Genome Assembly Pipeline

computationally infeasible for large numbers of reads.

To avoid this problem, *candidate selection* is performed first to find candidate pairs or reads that are likely to overlap. One common heuristic for this asserts that pairs without an exact short match are unlikely to be well-aligned for a longer prefix or suffix, and thus it is possible to discard pairs of sequences that do not share a short (usually 20-30 bases) exact match. This heuristic will filter out the vast majority of the $O(n^2)$ comparisons [99]

Finally, the assembler lays out reads in the proper order from alignment, creates one or more combined sequences, and then forms them together into larger structures called *scaffolds*. In most assemblers these processes are divided into many separate steps, however the rest of this chapter will refer to them jointly as the *consensus* step.

To use a layman's analogy, genome assembly is something like putting together a jigsaw puzzle. One method for solving the puzzle would be to check every edge of every piece against every edge from all the other pieces. However, this is an inefficient jigsaw puzzle technique, and instead heuristics such as discarding pairs of pieces with drastically different colors can be employed to reduce the number of pieces that have to be compared – candidate selection. The remaining possible connections are tried, and pieces are connected into small clusters – alignment. These clusters can be joined together to form larger and larger contiguous pieces – scaffolding and consensus.

5.2.2 Sequence Alignment

Sparse-Pairs problems occur frequently in the field of bioinformatics. One such example is the alignment step discussed above. Most previous approaches to parallelizing assembly have focused on programming models and hardware architectures for tightly-coupled parallelism, requiring dedicated high performance clusters or massively parallel supercomputers. The pattern of computation, however, is amenable to execution on a campus grid. The alignment step is the naturally parallel, requiring millions of pairs of sequences to be compared using a self-contained alignment algorithm. No task requires inter-computation communication or has dependencies on prior tasks.

In principle, one could run an All-Pairs abstraction to compare all fragments to each other, and then match up the pieces with the best scores. However, as noted above, for a sufficiently large problem, this is computationally infeasible – for the Human genome discussed below, for instance, an All-Pairs comparison of reads would require nearly 1 *quadrillion* alignments.

The candidate selection phase of genome assembly greatly reduces the problem, so instead of an application of All-Pairs, sequence alignment becomes an application of Sparse-Pairs. The list of "candidate" sequences remaining after candidate selection becomes the P set for a Sparse-Pairs workload, as shown in Figure 5.1.

It is natural to ask what good an aligner is without the other portions of the assembly pipeline. However, this work latches on to the growing trend of developing modular genome assembly components. Also, it was was developed in conjunction with a distributed candidate selection framework [86, 92] as part of the Scalable Assembly at Notre Dame (SAND) software package (`http://cse.nd.edu/~ccl/software/sand`).

The trend for modular assembly components is being approached from both sides:

the UMDOverlapper [107], for instance, can reliably work with several common assemblers; while the AMOS consortium [98] is actively developing an open source, modular assembly pipeline with the intent that others will contribute new and different approaches to each of the pipeline stages. Typical parallel solutions to genome assembly have tightly coupled alignment with the other stages of the assembly. These highly specific assemblers have relied on batch processing, complex MPI programming or specialized hardware such as BlueGene/L [73], FPGAs [116], and the Cell processor [110] to speed up alignment, but the new modular approaches are agnostic to the mechanisms of the individual modules. This presents a perfect opportunity for distributed abstractions to be supplied as modules.

5.2.3 Genomic Data and Algorithms

This chapter considers four different genomic datasets, shown in Table 5.1. The smallest dataset consists of the all the reads from the largest scaffold of *Anopheles gambiae S*, the next largest is the entire *A. gambiae S* genome. The *A. gambiae* genome was sequenced using traditional Sanger sequencing, which has longer read lengths, but is more expensive and time consuming. The large dataset is a set of simulated reads of the *Sorghum bicolor* genome [93], generated by extracting reads of 500-1000 bases from the finished *S. bicolor* genome with randomized starting positions. The largest genome is the Venter human genome [125], which is used in this work to demonstrate scalability to state-of-the-art sized data sets.

An important choice in any assembler is the algorithm used for alignment of the reads. In this work, two approaches are considered. The primary algorithm is the simple Smith-Waterman (SW) alignment commonly taught in bioinformatics textbooks [57]. This algorithm computes alignments in time proportional to the lengths of the sequences

by computing progressive overlap scores in a dynamic programming matrix. The reasons for using SW are twofold: it can be implemented very easily, highlighting the ability of the abstraction to be reused by scientists who are capable but not familiar with distributed systems programming; and its increased sensitivity may be required in certain cases, such as in SNP discovery programs like MOSAIK [61] and in short-read sequence assemblers. The second approach is a simple banded alignment, also introduced in [57], in which only a narrow band of the SW dynamic programming matrix is computed. In this case, the amount of data remains the same while the execution time of each task decreases significantly. This is used as an example of the various heuristics that are not as sensitive much run much more quickly than SW.

TABLE 5.1: GENOMES USED IN THIS CHAPTER

	Dataset	Number Reads	Average Read Size	Candidate Pairs	Uncomp. Size	Task Data Size	Comp. Size	Task Data Comp. Size
Small	*A. gambiae scaffold*	101617	764.22	738838	80.2MB	684.2MB	21.9MB	187.6MB
Medium	*A. gambiae complete*	1801181	763.66	12645128	1.4GB	13.2GB	0.4GB	3.6GB
Large	*S. bicolor simulated*	7915277	747.57	121321821	5.7GB	127GB	1.7GB	34.6GB
Huge	*H. sapiens complete*	31257852	654.49	327025224	20.0GB	299GB	5.5GB	79.7GB

5.3 Implementation

The sequence alignment application of Sparse-Pairs is built with a Master-Worker paradigm using the Work Queue API. Figure 5.3 shows how the master and worker pieces of the work together in SAND. The piece considered in this chapter is the middle operation: Alignment. Typical of the general case, the master is responsible for transferring the serial executable and task-specific data to the worker – in this case, the task-specific data is a pair of sequences for which an alignment score should be calculated. The worker completes the assigned computations and transfers the results back to the master, which sanity checks them and releases them to archival storage.

The alignment master is executed by the worker with few requirements:

```
sand_align_master align.exe cands.cand seqs.cfa results.ovl
```

The user supplies only his serial alignment executable, two inputs, and a target output file that will store a list of sequence pairs that overlap and data about the quality of the overlap and where the alignments occur within the sequences. The two inputs are the list of candidate pairs generated by the candidate selection step, and the actual library of reads. The candidates can be pre-computed by a candidate selection program or, with a runtime option, computed and supplied concurrently in a pipeline.

With the very large size of genomic datasets, management of data both before and during computation is the critical challenge for this application of Sparse-Pairs.

5.3.1 Managing the Input Data

The principal complication for Sparse-Pairs is that it is not generally feasible to optimize a bulk transfer of data files to many nodes because while each data item is used multiple times, the number of repetitive uses may be far less than the number of nodes. Additionally, input datasets are quite large and the target campus grid resources

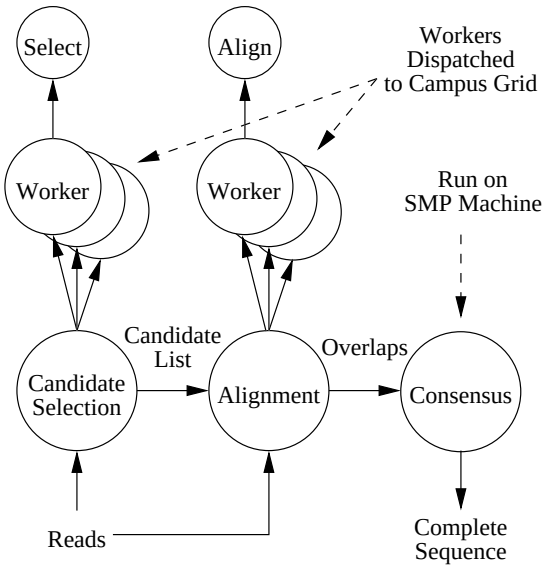


Figure 5.3. A Scalable Modular Assembler

are neither persistent nor reliable. The former limits the effectiveness or ability to prestage all the tasks' data to every compute node. The latter limits the effectiveness or ability to carefully craft exactly which tasks will run on which resources and prestage the appropriate task input files accordingly.

So the conventional approach [64, 88] is to prestage the work locally, split the problem up into as many tasks as there are resources, submit those tasks as batch jobs to the campus grid , and require the batch system to transfer the task input data with the batch job.

An issue with this solution, however, is its voracious consumption of local state. As most batch systems require all files to be in place on submission and remain in place (because of the likelihood of latency, out-of-order execution, or eviction) the framework would have to prestage locally a file corresponding to every task. For workloads in which sequences appear in many different candidates this means that the master must

have enough disk space for many times the total data set size. As an example, Table 5.1 shows the sequence library and required task data sizes for the four workloads used in this chapter. The task data corresponds to the amount of data that must be sent over the network.

To prevent excessive consumption of disk space and slow filesystem access to many small files, at runtime the master process reads the genetic sequences into a hash table in memory for fast lookup based on the sequence identifier. The abstraction engine (Master) can construct tasks on the fly as the workload advances, streaming data from memory buffers across the network.

The hash table can be extremely memory consuming on the Master, which also must be active in transferring data. This potentially creates a single bottleneck at the Master's outgoing network link. Both the memory consumption and the network bottleneck can be alleviated with compressed data – in bioinformatics, the alphabet $\{ACGT\}$ can easily be compressed to two bits per basepair – or multiple Master processes running on different nodes.

For fast-finishing functions, even if the Master has sufficient bandwidth the network latency may be too great to keep a sufficient number of Workers satiated. To prevent task submission latency from limiting effective parallelism, the input data (the sequence ID, the sequence metadata, and the sequence data for each candidate pair) for many separate instances of the serial program are grouped together into task buffers.

To decrease total data sent over the network in tasks that consist of many separate instances, the candidate list is sorted. This allows the master to easily group together pairs sharing a first sequence, abbreviating the task buffer so that the shared sequence is copied only once in a task buffer instead of once for every pair that includes it.

5.3.2 Coordinating the Computation

Because of the data-intensive nature of the bioinformatics application, the natural parallelism of the actual computation tasks, and the simplicity of Work Queue's Master-Worker framework, most of the challenges with this abstraction application are data management. However, there is one particular computational challenge that impacts users' satisfaction.

All candidate pairs are independently computable, and thus during a workload even very slow machines do useful work. At the end of a workload, however, slow machines may take work that would be completed faster on other available resources. In the worst case, this can hold up completion of the workload significantly and cause a long-tail effect at the end of the workload. Although unpredictable, this situation was not uncommon in the initial sets of experiments.

Even beyond the performance impact of these longs tails, the patience for slow (but eventually completing) tasks decreases significantly at the end of a workload. Users following the progress of their workload are anxious to see the results, and may get concerned about the correctness of the system if a few individual tasks appear to be hanging.

Avoiding long tails is the intent of the Work Queue fast abort mechanism. In all previous applications of fast abort, however, the mechanism was either enabled or disabled throughout the workload – generally based on whether there were computation dependencies throughout the workload. In this implementation, however, the fast abort mechanism is enabled *during* the workload at a certain point as an end-game strategy. This allows slow machines to contribute during most of the workload, but lessens the chance that they will result in a long tail in finishing.

5.3.3 Managing the Output Data

When Work Queue tasks complete, the worker send results back to the master to be written to persistent storage (in this case, the OVL record file provided as a command line argument). Because the master may run for many hours or days, it includes a recovery mechanism for starting back up a workload during which the master has crashed. The recovery mechanism in the master scans the completed OVL results (in linear time to the number of completed pairs) for the partially-completed workload. Because candidates may complete out-of-order, it is not possible to simply start in the candidate file at the next candidate beyond the last completed result from the results file.

For each completed result, a tag consisting of the two sequence identifiers is loaded into a recovery hash table in memory. Once tags are loaded for every completed result, the master mimics starting a new workload, with one key difference. As the master scans the candidate list to create and buffer new tasks, it checks each candidate pair against the tags in the recovery hash table. The master only needs to create tasks for those candidate pairs that are not yet completed (that is, those that don't have a tag in the recovery hash table). Because the recovery hash table may require significant memory, and this application does not have repeated tasks, the recovery mechanism removes completed tasks from the hash table as they are reached in the candidate file. This process gradually reclaims memory as each of the completed results is reached; and once all completed pairs have been scanned (so the hash table is empty), the hash table itself is freed. The remainder of the workload continues unhindered as though the recovery mechanism was never activated.

5.4 Evaluation and Results

Candidate selection for each of the first three datasets described above was completed on the reads using the complementary SAND module. The memory required for candidate selection was reduced from 18GB on a single core to less than 2GB per core across the cluster throughout the workload, ensuring consistent access to data without costly paging to disk, and garnering speedup [86]. After this process, sequence alignment on the datasets were benchmarked, varying the number of resources provisioned from the Notre Dame campus grid. Those results are presented in this section. Performance at larger scales (in terms of both data set size and number of workers) is examined in the next section.

5.4.1 Task Size

In the benchmarks below, each task contained 5000 alignments. However, when running on a sufficiently fast network, such as a local cluster, task size does not have a significant effect on performance, which can be seen in Figure 5.4.

Task size becomes more important when many nodes are further away over the network, as the transfer time for each task does not scale linearly with the size of the task. Larger task sizes pay the same overhead while sending more data, and utilize the workers better, resulting in faster run times and better speedup. However, there are two major downsides to increased task size. First, if the system is especially volatile, more work is lost when a worker is evicted. Second, the master queues a large amount of tasks to ensure that the it can quickly dispatch tasks to workers that have returned a completed task, or to new workers that join. A larger task size will take up more memory per task, increasing the memory consumption. The effects of excessive memory consumption are discussed in more detail in Section 5.5.

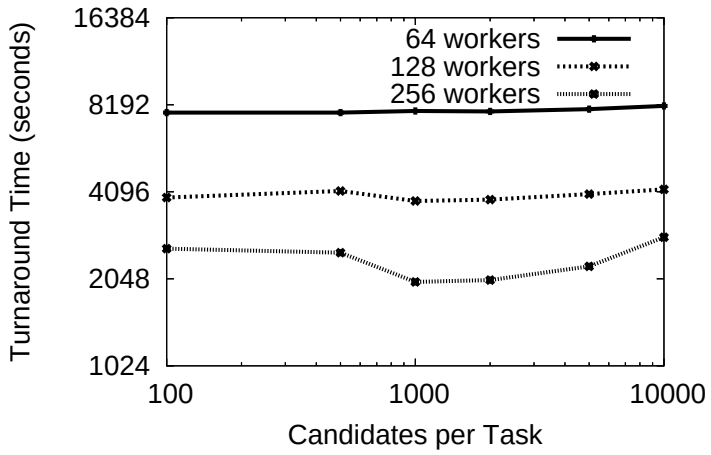


Figure 5.4. Alignment Candidates Per Task

There is little difference in workloads with different numbers of alignments per Work Queue task, extreme task sizes may be inefficient with many workers.

5.4.2 Scalability Benchmarks

A workload that indicates good strong-scaling efficiency will, for a constant workload problem size, see its speedup scale by the same factor as the increase in number of processors. A workload that indicates good weak-scaling efficiency will keep a constant turnaround time if both the problem size and the number of nodes are increased by the same scaling factor. Sequence alignment demonstrates both strong-scaling and weak-scaling in the benchmarks in this section.

Calculating conventional parallel speedup (the sequential wall-clock turnaround time divided by the parallel wall-clock turnaround time) for a heterogeneous and dynamic set of resources is not meaningful, because the serial resource has little performance relation to the parallel resources. Further, because the benchmarks were so large and contained so many alignments it was not feasible to simply run all the alignments sequentially. Instead, the speedup metric in this work uses the workload's average

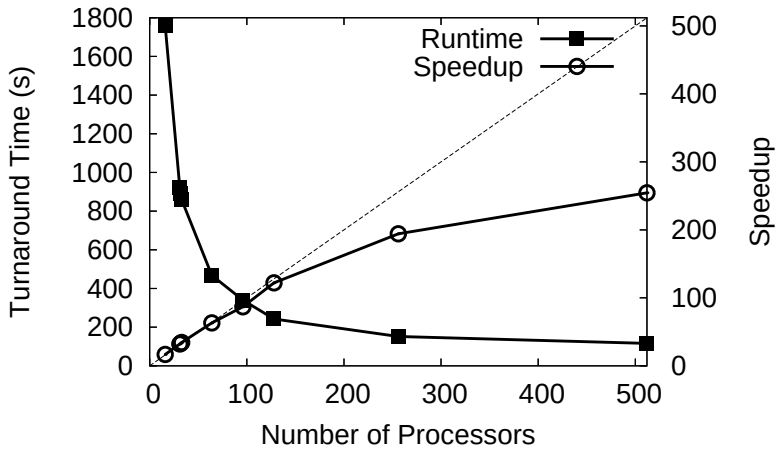


Figure 5.5. Scalability of Alignment on Small Genome

The small genome scales efficiently to 128 local campus grid nodes. Beyond that, the problem is not large enough to exploit additional parallelism.

execution time across all tasks, multiplied by the number of tasks completed as the sequential runtime for the parallel speedup computation. Note that in Figures 5.11, 5.12 and 5.10, which consider both problematic and corrected instances of workloads, the average run time from the corrected version is used to calculate speedup as a function of time.

Almost all of the benchmarks exhibit scaling speedup. However, each benchmark has features that shed light on the strengths and weaknesses of the system. For the smallest dataset, benchmarks achieved near linear speedup until about 128 workers (Figure 5.5). Because this is the smallest dataset, with too many nodes all the work is completed before some nodes receive a task.

The medium dataset (Figure 5.6) yielded better results; the dropoff in speedup did not occur until 512 nodes were used. The large dataset (Figure 5.7) displayed similar scalability to the previous dataset. It was able to run on 512 cores in only 9595 seconds,

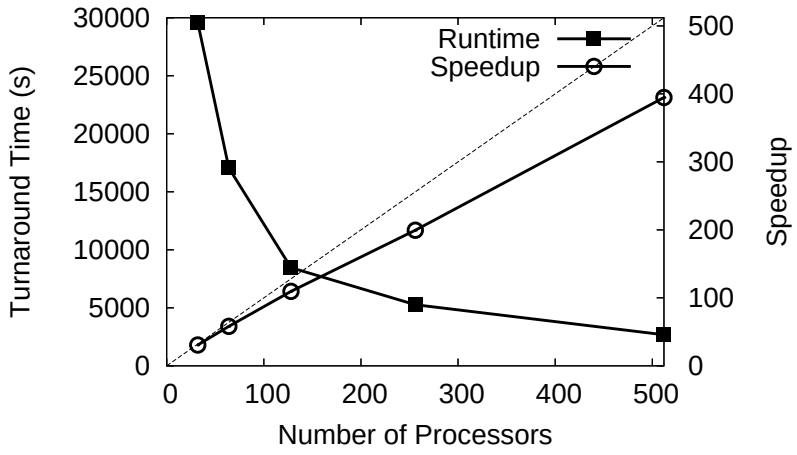


Figure 5.6. Scalability of Alignment on Medium Genome

The medium genome scales efficiently to 128 campus grid nodes at the same institution as the master, and scales to 512 nodes while retaining 80% efficiency.

for a speedup of 455x. This dataset did highlight some of the challenges of the assembly problem and of distributed computing in general. These are discussed in detail in Section 5.5.

5.4.3 Banded Alignment

One of the primary advantages of the abstraction framework is its ability to substitute any alignment algorithm for the one used in the above benchmarks. So, in addition to the benchmarks using SW, Figure 5.8 shows execution of the medium dataset with Banded Alignment to consider how the framework adapts to alignment programs that are considerably faster. As a result of the increased relative overhead, scalability should decrease, and the results confirm this – scaling speedup up to 64 workers, beyond which are diminishing returns.

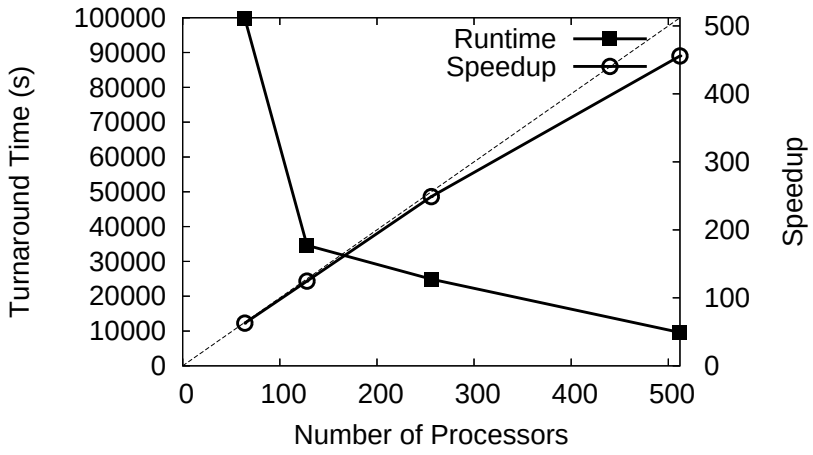


Figure 5.7. Scalability of Alignment on Large Genome

The large genome scales efficiently to 256 campus grid nodes at the same institution as the master, and scales to 512 nodes while retaining almost 90% efficiency.

5.4.4 Preventing Long Tails

Although long tails are common, even identical workloads on identical resources within a campus grid vary too much for long-tail conditions to be predictable. Because of this, in order to evaluate the fast abort mechanism a set of identical resources were picked from a 64-node cluster in which one of those nodes was handicapped to take 5-10x longer to complete tasks than the other nodes. This environment is much more prone to substantial delays in the workload due to a single very slow node.

Figure 5.9 shows a histogram of completion times for 38 workloads with the Small dataset on this set of resources. The white boxes show counts of workloads in which fast abort is not activated and the dark boxes are counts of those in which fast abort was activated after all tasks had been submitted. Though variations in workload timings didn't result in long tails every time without fast abort, it is clear that a significant amount of the trials took much longer to complete. Upon inspection, every one of these

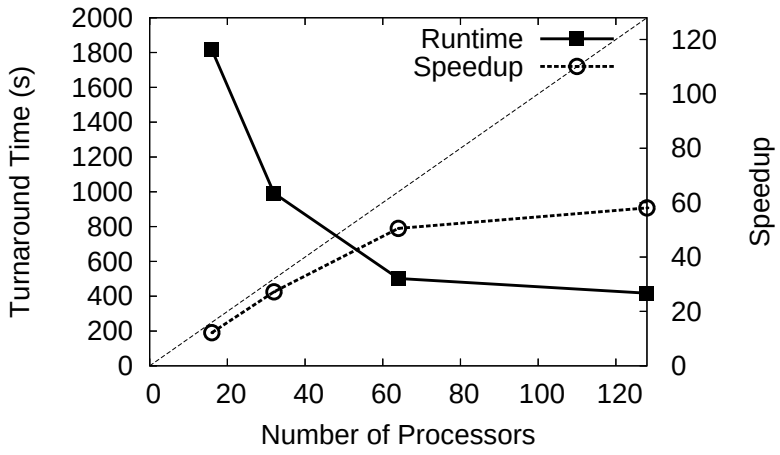


Figure 5.8. Effect of Faster Alignment

Many applications do not need the precision of a complete Smith-Waterman alignment and can use faster alignment heuristics. This graph of a workload on the medium dataset with faster alignments shows decreased scalability, but retains significant speedup over a serial solution of approximately 60x.

delays resulted from having one remaining task being computed on the slow worker while all other workers were idle. The version with fast abort enabled to cut off a worker after it has exceeded the average completion time by 50% does not suffer from these extreme tails.

5.5 Production Workloads on the Grid

For very large problems, the computational resources required exceed the capacity of the clusters comprising Notre Dame's campus grid. This section explores the ramifications of running on multi-institutional resources such as remote Condor pools or the Open Science Grid [2]. The primary experiments run on the large dataset for sufficient available parallelism, using Condor's flocking mechanism as an example of using remote grids. This section illustrates the problems that led to design decisions discussed

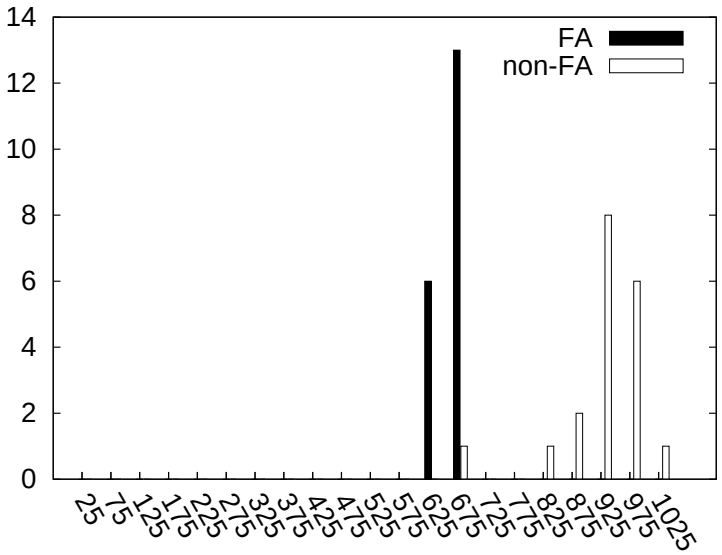


Figure 5.9. Using fast abort to prevent long tails

A heterogeneous environment is prone to long tails at the end of workloads. This histogram groups the runtimes of 38 identical workloads in such an environment, half with the fast abort countermeasure in place and half without. Almost all of the runs without fast abort take significantly longer than the ones in which it was turned on.

in Section 5.3, which often cannot be seen at benchmark scales, and examples of large workloads that run efficiently when these problems are eliminated.

The last two subsections below describe two types of highly-scaled workloads that demonstrate the capabilities of the abstraction framework to run on a wide variety of scales and use-cases. The first of these is a contrived (but realistic) scenario that demonstrates how a scientist may use the system to test a new method in production and then quickly scale the workload up to a multi-institutional grid to generate complete results. The second is an even larger run coordinated from the start to use more than 1000 cores to complete a workload of particular interest to the bioinformatics community and the public at large.

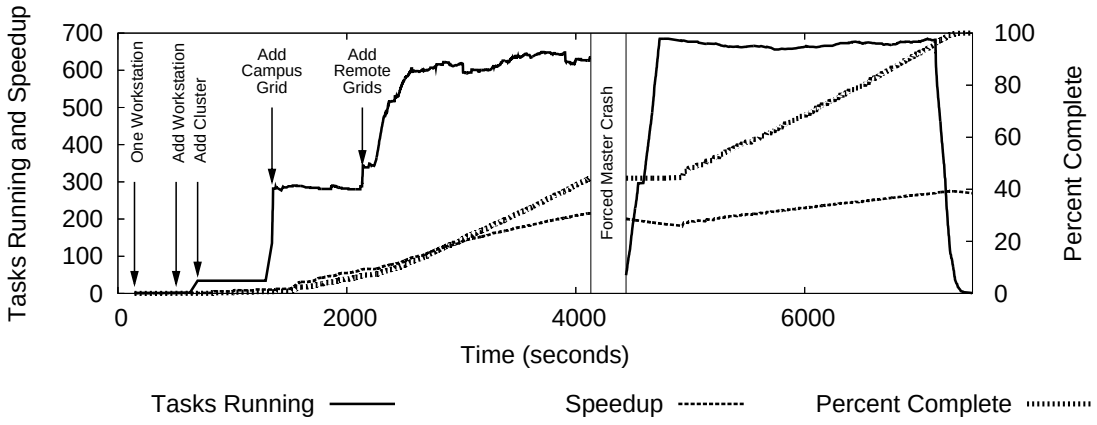


Figure 5.10: Scaling Up to the Grid

This figure shows the timeline of a large assembly run on a system grown progressively from a single workstation up to a large scale grid including resources at the University of Notre Dame, Purdue University, and the University of Wisconsin. The master is forcibly killed halfway through to demonstrate failure recovery.

5.5.1 Out-of-Core Task Data

Complete alignment on the large dataset scales at nearly linear speedup up to 256 workers, but saw a marked decrease in performance when using 512 workers. The biggest problem with running such a large dataset was memory. Although the master was running on a machine with 8GB of memory, the large dataset was 5.7GB. This is loaded into memory to achieve the best retrieval times when building tasks. Additionally, the master buffers a significant number of tasks in memory, and this number scales up with the number of connected workers.

With 512 workers, the additional buffered tasks caused the master to exceed physical memory. When the master began to need paging for its task management, performance began to degrade. The effect of this can be seen in Figure 5.11(A). Because it takes significantly longer to create the number of tasks required, workers must wait longer to receive their task. When running with many workers, the amount of time

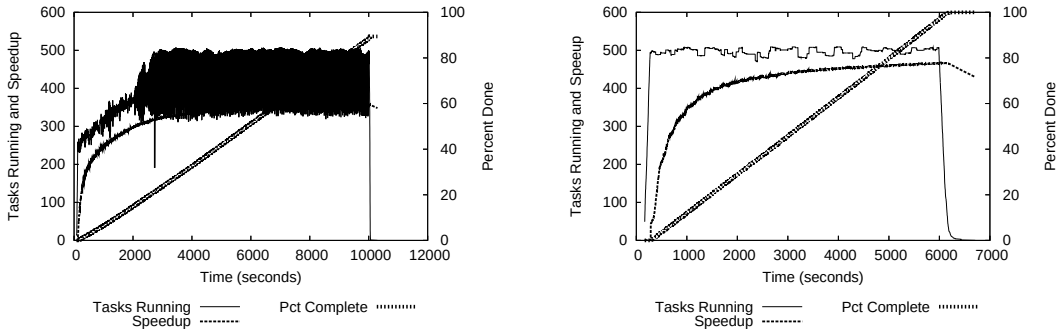
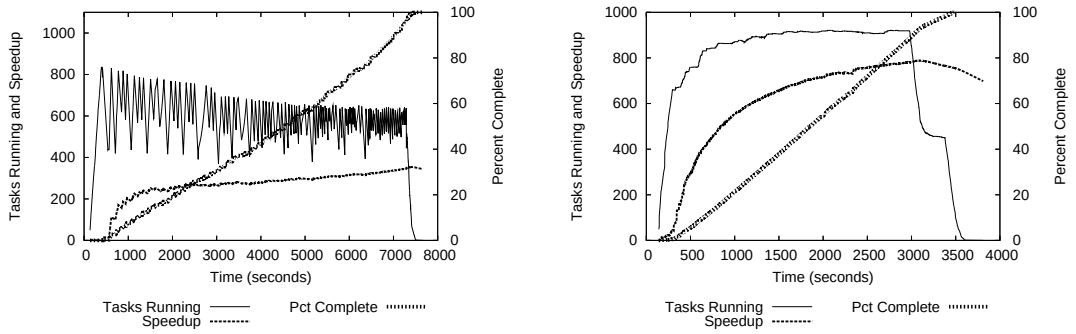


Figure 5.11: The Effect of Data Compression.
These graphs show the effect of data compression on the master's ability to dispatch tasks using the large dataset. Each shows a timeline of a single run, with the number of tasks running, the cumulative speedup, and the percent complete over time. Figure 5.11(A) does not use data compression, and oscillates between 300 and 400 tasks running at once, reaching a speedup of slightly better than 300x. Figure 5.11(B) uses compression, and stabilizes at about 500 workers with a speedup of about 500x.

necessary to give tasks to all the workers is longer than the amount of time it takes a worker to complete this task. This creates a convoy effect, where workers are spending more time waiting to be processed by the master than they spend actually working. This explains the large variation in the number of tasks working.

Figure 5.11(B) shows how the same job ran on 512 workers with compression enabled. Once the amount of memory needed can be kept within the physical memory, the master is easily able to keep up with the workers requesting tasks. In this case, the number of workers running at any time remains relatively constant, subject only to minor fluctuations, mostly caused by changes in the number of workers active.

This continues to be necessary even as resources scale up with the data set sizes. For example, the master for the human dataset was run on a machine with 32GB of RAM, which was enough for its 20GB requirement. Without compression, however, that requirement would have been far exceeded even the large memory machine's core memory.



(A) Single Master **(B) Dual Masters**

Figure 5.12: The Effect of Splitting Masters.
When using a sufficiently large number of workers on the large dataset, the master does not have enough network bandwidth to keep all of them busy. These figures show a timeline of a single run with approximately 950 workers using one master (A) and two masters (B). With a single master, workers complete faster than the master can dispatch new work, so not all nodes can be kept busy processing at once, and the speedup reaches less than 400x. With dual masters, peak speedup reaches 790x before settling out about 700x. Note that the unequal rate of completing work in (B) causes the dropoff beyond 3000s.

5.5.2 Waiting for Task Assignment

When a master has many workers connected to it, it takes the master longer to assign tasks to all the workers in round-robin fashion. If task assignment is slow, it can take the master longer to assign tasks to all workers in the pool than it takes for an individual worker to finish its task. The same symptoms appear as in the memory case above: workers spend more time waiting to be given new tasks than they spend working, and efficiency suffers. In this case, the main problem is waiting for the master to transfer task data to every worker. To exceed the number of machines available in Notre Dame's Condor pool, machines are available from other computing pools on campus and other institutions' Condor pools, particularly Purdue University and the University of Wisconsin.

Resources beyond the local campus grid, however, introduce significant data trans-

fer complications due to reduced throughput capacity. While data transmission to machines at Notre Dame averaged of 42.29 MB/s (meaning data for a task could be transfered in only a few hundredths of a second), data to Purdue took an average of .36s, and data to Wisconsin was even slower, at .53s per transfer. In a with 900 submitted workers for a single master with 5000 candidates per task, the average transfer time was 0.27s. 835 workers completed tasks, with the others failing to find an available campus grid resource or exiting after starvation. This means the average time to transfer files to all 835 workers was 225s, which is greater than the typical task completion time.

Ideally more nodes with fast connections could be added in place of machines at other institutions, however this is not always feasible – campus grids are often at the whim of voluntary contributors and institutional budgets. In order to take advantage of remote computing resources that have slower network transfer times without compromising the efficiency of a workload, one mechanism is to divide the workers between two controlling masters, balancing the slow connections between them. This is not an ideal approach, as it requires splitting the list of candidate pairs half and running the master program on two separate machines with the same list of sequences but different halves of the candidate pairs list. When using two masters on the above workload, sending data to 450 workers each averaging 0.27s per task takes only 121s, so both masters were able to work efficiently.

Figure 5.12(A) shows a timeline of workers waiting rather than actively computing associated with this problem for a similar job with 950 submitted workers, while Figure 5.12(B) shows the smoother two-master version of the same workload. The maximum number of workers running tasks at a time was 921 with two masters.

The multiple-master technique is not limited in application to workloads with large number of workers with slow connections. Various other system resources limitations

can cause workers to experience starvation even if network speeds are fast enough to support all workers. For instance, many Linux systems have hard limits on file descriptors open by a process (usually 1024), and users might not have root access to increase this limit. Using multiple masters multiplies the number of connections, and thus supportable workers, in this case.

In the future, instead of running multiple masters as separately invoked user programs, the masters themselves could be Work Queue tasks. In this case, a single hierarchical master would be invoked by the user, which would be responsible for starting masters as Work Queue tasks and dividing work between them.

5.5.3 Growing From Desktop to Grid

This subsection presents an example that is contrived – that is, an actual domain scientist did not proceed through this set of steps attributed to him, instead application developers performed actions typical of an exploratory use case for the abstraction framework. This scenario serves not only to illustrate a typical user's actions, but also to demonstrate all of the features and flexibility of the framework: adaptability to many types of resources (local execution, execution as a cluster job, execution on a campus batch system, execution as part of a multi-institutional resource pool); fault-tolerance to failures on the worker nodes; and fault tolerance to failures on the master node.

As in many fields, research in bioinformatics is highly exploratory. An active researcher may test many slight variations upon an algorithm, generating a number of tests of various sizes before proceeding to analyze an entire dataset. Because Work Queue does not require a predetermined set of workers, a user may slowly generate small results, then progressively add resources as confidence is gained. Figure 5.10 graphs such progressive growth for this contrived example:

TABLE 5.2

SUMMARY OF MULTI-INSTITUTIONAL WORKLOAD

	Tasks	Average Runtime (s)
Total	16936	184.1 ± 53.8
Notre Dame	7998	215.3 ± 46.4
Purdue	7760	154.0 ± 40.8
Wisconsin	1232	170.1 ± 56.2

With the master running a scientist started a worker process on his workstation. After a few minutes, he surveyed the progress and determined that the results were promising, but serial execution would not be sufficient, so he asked a coworker to start a worker on her own machine, and also prepared and submitted some batch jobs to his research group's 32-node cluster. As these jobs started running, speedup increased accordingly. Hoping to finish the alignments that afternoon, he submitted jobs to the campus computing grid at Notre Dame, followed by submissions to Condor-based grids at Purdue University and the University of Wisconsin. About halfway through the complete assembly, however, he accidentally killed off the master, causing the computation to halt. Fortunately, when the master was restarted, it loaded all of the complete results, accepted connections from the still-running workers, and continued where it left off. The entire assembly completed in just over two hours, with a speedup of 269x and a maximum of 680 CPUs in use at once. Note that the low speedup should not be alarming, because of the gradual nature in which the workers were added, and because it includes the unproductive time during the crash in the middle of the job.

Table 5.2 summarizes the work distribution across sites. The tasks running at home

were slower and exhibited more runtime outliers, because the local campus grid includes a large number of scavenged resources compared with more homogeneous dedicated grid resources at the other sites.

Even making many connections over the WAN, the master was still able to maintain a steady task throughput with machines at three different institutions. The scalability is strong – taking into account that the final speedup is not reflective of the final state of the workload – and with an improved wide area network connection or a larger number of local resources available even more resources could be harnessed. Additionally the multiple-masters technique used before to demonstrate a solution to insufficient network bandwidth will still be advantageous.

5.5.4 Many-Node Runs on the two Largest Datasets

When there are not enough local machines, so resources from other institutions are used (thus significantly increasing the master's transfer times to each worker), using two masters on a single workload shows scalability beyond that seen in this section's first scenario. Using multi-institutional resources, Smith-Waterman alignments were computed for the large dataset – 121 million candidate pairs from a set of 8 millions sequences – in under one and a half hours. For comparison, the same workload serially would take over 57 days on an average resource from the campus grid pool. Figure 5.13 shows a peak of almost 1300 resources harnessed, sustained levels above 1000 for an hour during the workload, and a final speedup of 927x at 71.3% parallel efficiency.

Finally, the last workload demonstrates scaling beyond even the large workload. A complete alignment of the Human genome [125] – 327025224 candidate pairs from a set of 31257852 sequences – was computed in 2.5 hours using 1024 nodes with one master. The pool of resources was limited to 1024 nodes because this was the maximum

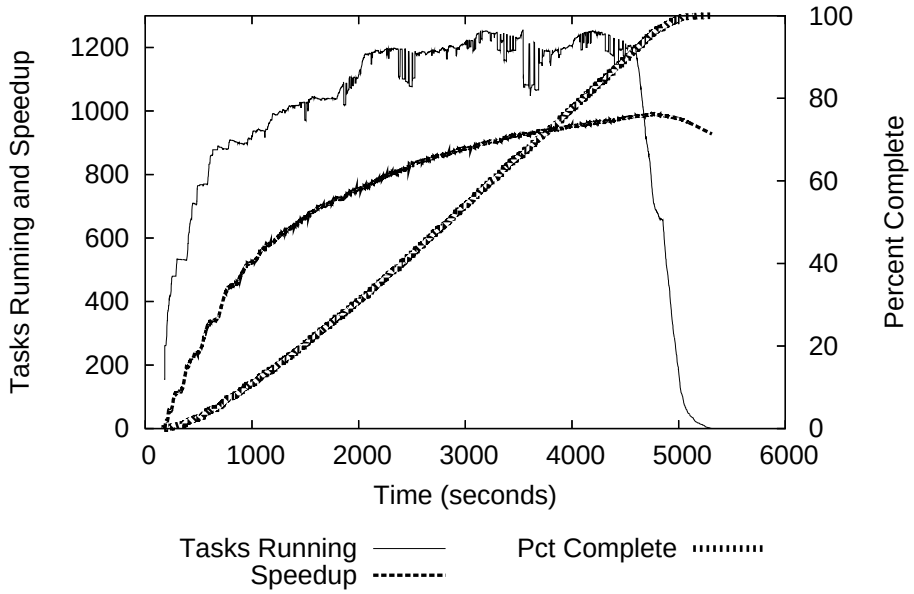


Figure 5.13. Multiple masters at grid scale

This figure shows the timeline of a 121M candidate run on the large sequence set using approximately 1300 workers at two institutions separated by a WAN. Two masters support almost 1300 at peak and use 1000 or more workers consistently for most of the 90-minute runtime, totaling a speedup of 927x.

supportable number of connections on the machine where the master was run, but the master process could have supported 3000-4000 workers in terms of network transfer performance. The resources were all located on the local campus grid at one institution, so there was no detrimental effect of transferring data over the WAN, although they came from several different resource pools on campus. Figure 5.14 shows a peak of over 1000 cores harnessed (the theoretical maximum given the number of available file descriptors), sustained peak levels once the peak is reached, and a final speedup of 952x at 93.0% parallel efficiency.

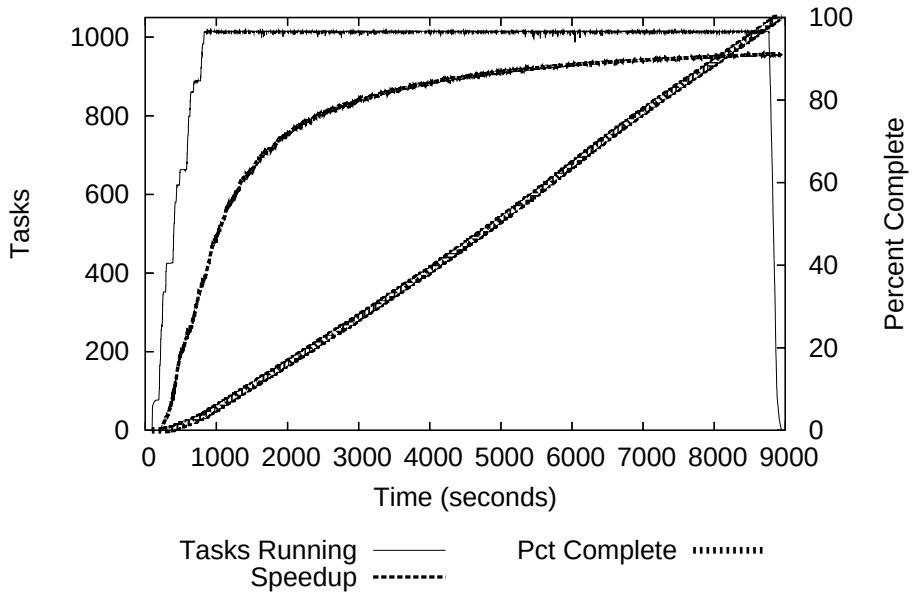


Figure 5.14. Human genome at scale

This figure shows the timeline of a 327M candidate run on the huge sequence set using 1024 workers on the Notre Dame campus grid. The master supports all workers at peak, and sustains peak performance for over two hours during the run, totaling a speedup of 952x.

CHAPTER 6

DATA-SPLIT-JOIN ABSTRACTION

6.1 Data-Split-Join Abstract Problem

The Data-Split-Join problem is an example of a computation that is straightforward to write up on a chalkboard, but not so easy to implement for applications that must manage gigabytes of data splits to campus grid nodes, hundreds of computation tasks, and a summary operation that of a collective join across a campus grid. This chapter discusses the design, implementation, and deployment of an abstraction for Data-Split-Join, particularly as applied to a general data mining application.

An abstract Data-Split-Join problem can be defined as follows:

Data-Split-Join(D, T, P, N, F, C) returns R:

D - Primary (Split) Dataset: list of (name,properties)

T - Secondary (Join) Dataset: list of (name,properties)

P - Partitioning method.

N - Number of partitions.

F - Function.

C - Collection process.

R - Result set: list of (name,class)

The Data-Split-Join workload starts by dataset D into process P, which creates N partitions D1...DN. These partitions are the initial input into N copies of F in parallel.

Once a function has completed its computation on the partition of D, it then completes a computation on set T, generating results R1...RN. Results are joined by collection process C into a final result R returned to the user. The function F is simply an existing sequential function with the following signature:

F(D, T) returns R:

D - Partition of primary dataset: list of (name,properties)

T - Full secondary dataset: list of (name,properties)

R - Result set: list of (name,class)

Although Data-Split-Join has many more parameters as an abstract problem than All-Pairs or Sparse-Pairs, its implementation can be broken down in the same way. The first consideration is how to appropriately partition the subsets (that is, manage the input data) for learning. As with the previous abstractions, modeling the problem, managing the input data, and coordinating the computation are intertwined – part of what makes a general problem solvable with a computing abstraction is that the design decisions are interrelated in clear patterns. Thus, the subset partitioning approaches will often be coupled with corresponding approaches for coordinating the computation.

6.2 Application of Data-Split-Join

6.2.1 Challenges of Data Mining Large Datasets

In recent years increasingly massive datasets have become available from scientific research and data collection on numerous real-world applications. The large increases in dataset size and complexity taxes data mining algorithms and the computer systems they run on. Dataset sizes that exceed the memory capacity of a desktop computer, in particular, are a continually expanding challenge if data grow much faster than memory

capacity.

Parallel and distributed data mining [74] systems have held back the wall of data with scalable implementations of various learning algorithms, allowing a capability to scale to massive datasets. As a benefit, sampling [30] and ensemble methods [27] can even gain a significant improvement in accuracy in such systems. Building on these successes, workloads have been applied across even larger distributed systems [24, 53, 62, 75, 95, 117].

But, as with many such developments, there have been two nearly-disjoint approaches. The projects that have successfully scaled to larger systems [21, 50], by and large, have done it with application-specific designs and implementations. The implementations are often limited to highly-reliable clusters, or complicated to design without expertise in distributed computing. On the other side, general-purpose systems may require less effort from the programmer and/or user but still cannot scale beyond several Gigabytes of data [52].

6.2.2 Ensemble Methods for Classification

Ensemble classification is a general divide-and-conquer data mining technique in which a classification decision is reached through the coalescence of several independent classifiers, each of which was learned on a different subset of the training data. This problem lends itself to parallelization. In the parallel case, a dataset is partitioned across a group of processors. Each of those processors learns a classifier concurrently, and a central processor coalesces these disparate classifiers as an ensemble.

There are numerous variations for how to construct the ensemble from the separate classifiers. For this work it is asserted that the independent classifiers are applied to the testing set on the same parallel nodes they are already running. Their votes are

collected by a process running on the central processor, which coalesces them with a majority vote. By structuring the problem this way, ensemble classification matches the signature of the Data-Split-Join abstraction; the training set is the primary dataset D and the testing set is the secondary dataset T.

Ensemble classification can benefit both performance and accuracy of a workload. An attractive characteristic of ensembles is that they reduce the computational complexity of the problem (often n problems of size m are easier to solve than one problem of size mn) which also decreases the hardware requirements for solving the problem. Smaller training sets also decrease the risk of an inductive learner overfitting as it tries to model the entire training set. And because each independent classifier learned on a different small subset of the data, the classifiers are diverse, which can also improve overall accuracy.

Parallelization of ensemble classification doesn't diminish any of these advantages, and it gives the additional benefit of computing the small independent classifiers at the same time. The Data-Split-Join abstraction allows data miners to scale up the general pattern of ensemble classification efficiently on several scales of resources from a campus grid.

6.3 Implementation

6.3.1 Managing the Input Data

There are many possible ways to implement Data-Split-Join in a parallel or distributed system. An implementation must choose how to use nodes for computation, how to use nodes for data, and how to connect the two. Figure 6.1 shows several possibilities, differing only in where data is placed in the system.

Like All-Pairs, data movement and placement is critical, because network and file

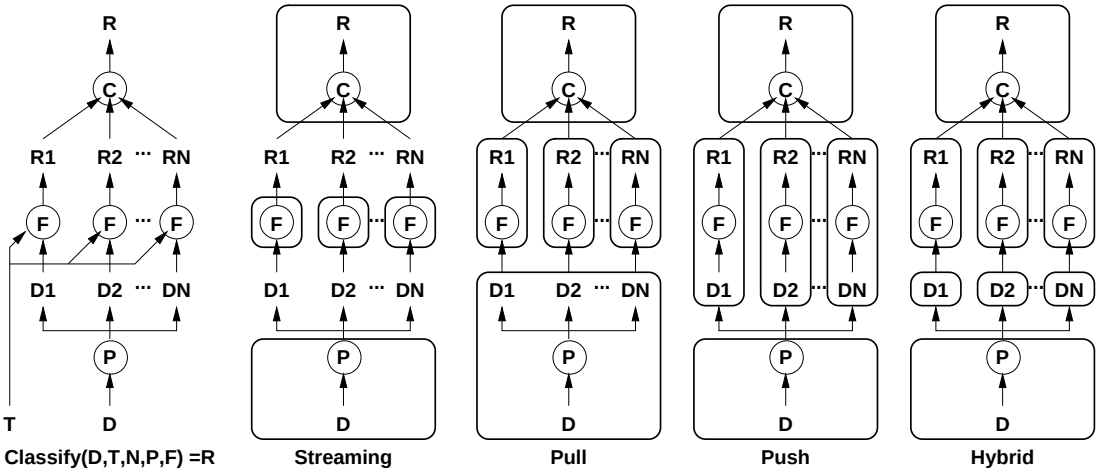


Figure 6.1: Four Implementations of the Data-Split-Join Abstraction
This figure shows four possible ways of implementing the Data-Split-Join abstraction by varying the placement of data and functions on the nodes of the system. Rounded boxes show the boundaries of one node in the system, which has both a CPU and local storage. For example, in the Pull implementation, the partition function P reads the training data D and writes the partitions D1...DN back to the same node. Each of the functions F run on separate nodes and pull the data over the network. But in Push, the partition function P reads the data D from one node and writes the partitions directly to the execution nodes, where the functions F read the local copy. Full details are given in Section 6.3.

server access is a key limitation at large scales. Unlike All-Pairs, there is little to gain from putting all data everywhere, because each partition requires only a subset of the full data.Even if many separate partitions will be computed on each node, this still requires solving something akin to the partitioning problem on each individual node, because the function must be able to address and access its specific partition from within the full dataset.

Streaming. The simplest implementation of Data-Split-Join connects each process in the system at runtime via a *stream* such as a TCP connection or a named pipe. Data only exists in memory between processes and, except for some minimal buffering, a writer must block until a reader clears the buffer of data. The simplicity of avoiding the

disk, however, results in the requirement that all processes be ready to run simultaneously. It also affords no simple recovery from failure – if one process or stream fails, the abstraction has two options. It can either perform a significant collective communication to determine what data has been distributed, then compare that to the entire data set to determine the contents of the lost partition, or it can give up and retry from the beginning. Neither of these is attractive if failures are common, thus, Streaming is appropriate only in very select cases. One such case is an implementation for a multicore machine with the number of partitions less than or equal to the number of processes. Except for very small workloads, Streaming is not practical for larger clusters or a campus grid where the possibility of network or node failure is very high. To make the abstraction robust, the implementation must make use of some storage between processes.

Pull. In this implementation, P reads data from the source node and writes partitions back to the same node. When the various Fs are assigned to CPUs, they connect to the source node and *pull in* the proper partition. This provides maximum runtime flexibility as there is no constraint on where an F may run. Because each partition is stored on disk, individual Fs may fail and restart without affecting the rest of the computation. This places a significant I/O burden on the source node in both the partitioning and computing stages, however. The technique may be appropriate for a cluster with a large central file server, but is not likely to scale to a campus grid of any significant size.

Push. In this implementation, P chooses *in advance* which nodes will be responsible for working on each partition. As it reads data items from the training set, they are *pushed out* directly to the assigned nodes. The Fs are then dispatched for execution. In "Pure Push", each F must run only on the node where data is located. This may not be possible in the absence of dedicated resources, as that node may have been dynam-

ically assigned to an unrelated task. "Relaxed Push" is a slight variation that resolves this, where each F prefers to run on the node with its partition but may also run on another node and access that partition remotely. This technique can improve the performance of partitioning and the overall I/O rate as the number of nodes increases. The value of the Relaxed version is particularly significant when contention for resources is high.

Note that relying on nodes from the campus grid for data access will increase the exposure of the system to failed, slow, or otherwise misbehaving disks, which are surprisingly common across a large computing pool. This is a key tradeoff between the reliable (but possibly underprovisioned) central server and a set of remote resources.

Hybrid. To address the *limitations* of Push and Pull, a fourth implementation, Hybrid, is designed with the *strengths* of each. In this mode, P chooses a small set of intermediate nodes known to be fast, reliable, and of sufficient capacity to write the partitioned data. At runtime, each F then reads its partition over the network from these nodes. This combines advantages of Pull (flexible allocation of CPUs, reliable partitioning) with advantages of Push (increased I/O performance). However, it requires the implementation to have some knowledge of the reliability of the underlying system, which may not always be possible.

Even once a general pattern for data distribution to the location of the computation is establishes, there is still the matter of the actual partitioning mechanism. There are a number of different partitioning techniques for the training set, each again with certain tradeoffs. *Shuffle* selects data items one at a time and sends each to a random partition, resulting in roughly equal-sized partitions. These partitions are unlikely to be corrupted by the structure of the input data (if the instances are sorted, for instance), however this comes at the cost of having to make a separate decision for every single instance. A

shuffle partition may also be *M-overlapping*, in which an item may appear in M partitions, allowing for more accurate sampling of minority classes but increasing data sizes and runtimes. *Chop*, on the other hand, does not make separate placement decisions for each instance, but rather divides the training set into equal pieces, preserving the existing order. For the ensemble classification application, this is typically only appropriate when the data is pre-randomized, or when the user wishes to reproduce runs exactly, as any inherent structure or organization in the training set may corrupt classifiers.

6.3.2 Coordinating the Computation

The source node running the Data-Split-Join abstraction is responsible for several tasks: partitioning the data, configuring local state to define the batch jobs, submitting the batch jobs, and collecting the results after all jobs have completed. The remote nodes on the campus grid are responsible for executing the function instances and generating the prediction output.

Local state requirements include an execution directory, the primary and secondary dataset definitions required by all functions, and the batch job definition files. The secondary dataset (and other shared metadata, such as the `.names` dataset definition) is not replicated on the local disk, but rather shared efficiently. The job definition files are created after the data partitioning, and the batch jobs are submitted using these definitions.

Within the batch jobs themselves, there is a hierarchical architecture of processes. The batch job that is run on each remote node is the *wrapper*, a standard piece of code that is the same for all instances of the ensemble classification application of Data-Split-Join. The wrapper is responsible for setting up the execution environment on the remote compute node, then executing the *function*. The function is a user-provided

application-specific piece of translational middleware. The function executes the underlying executable (the *application*) and maps application-specific output to the structure expected by the wrapper. The function allows execution of any underlying application without having to change core pieces of the abstraction framework. After the function is complete, the wrapper is again responsible for ensuring that all items are in the required places to be picked up by the batch system.

6.3.3 Managing the Output Data

Collection is the process of managing the output data, which consists primarily of the results from each function. In the case of ensemble classification, the results are the votes from each individual classifier on each instance. This section considers two approaches for collection, each of which is vaguely analogous to one of the partitioning methods in terms of the order in which it accesses and manipulates the data.

The first, *by-file*, is analogous to chop partitioning. The algorithm completes the entire results file for one function at a time, maintaining a plurality-determining data structure for each instance in the secondary dataset. After all files are processed, each data structure contains the combined final result. The overall accuracy, accuracy per class, and other important data mining statistics can be computed from these data structures. As the number of instances in the secondary dataset increases, this version needs more memory to maintain data structures for each instance. Memory requirements scale by a factor of the product of the number of secondary dataset instances and the number of classes in the dataset.

The alternative, collecting *by-instance*, is akin to shuffle partitioning. The results files for all the functions are accessed concurrently, and only one data structure is needed as each instance is tallied serially across all results files. Memory for this

version remains constant as the number of instances increases, since the memory requirement is only a factor of the number of classes in the dataset. On the other hand, it requires more files open at once and accesses individual results files less efficiently.

An abstraction may decide the tradeoff between file resources accessed concurrently and memory used for concurrent tallying data structures. For datasets that have few classes, concurrent data structures for each partition fit in memory easily even when the test set is large. However, for very large numbers of classes or very large numbers of instances in the secondary dataset, it is possible for the collection to exceed main memory capacity.

Because the largest collection memory requirement of any dataset tested in the evaluation below was less than 100MB, all of the results use by-file collection. Note that another concern could be the transfer of all prediction files back to the submitting node, however because the largest set of prediction files was still less than 10MB of output it was not necessary to use separate file server or distributed filesystem.

6.4 Evaluation and Results

This section summarizes the results of the large number of experiments – across datasets, algorithms, and system sizes – to evaluate the performance and scalability of the Data-Split-Join abstraction implementation as applied to ensemble classification.

The platform used as testbed to evaluate the performance and scalability characteristics of Data-Split-Join is the same institutional condor pool as described in Chapter 3. Although the pool as a whole is a campus grid with limited control for the user, in conducting the experiments a 48-node subset allowed direct control. This allowed more power over the environment (e.g. reliability of resources, priority status for execution). The machines in this dedicated cluster are dual-core 64-bit x86 architectures with ei-

TABLE 6.1

ATTRIBUTES OF DATASETS

Dataset	**Training Instances (Size on Disk)**	**Test Instances (Size on Disk)**	**Attributes**
Protein	3,257,515 (170 MB)	362,046 (20 MB)	20
KDDCup	4,898,431 (700 MB)	494,021 (71 MB)	41
Alpha	400,000 (1.8 GB)	100,000 (450 MB)	500
Beta	400,000 (1.8 GB)	100,000 (450 MB)	500
Syn-SM	10,000,000 (5.4 GB)	100,000 (55 MB)	100
Syn-LG	100,000,000 (54 GB)	100,000 (55 MB)	100

ther 2GB or 4GB of total memory (1GB or 2GB per core, respectively). Jobs were instructed to prefer this cluster over other nodes when available.

6.4.1 Datasets

The data for these experiments is a combination of real and synthetic datasets with varying dimensions covering a wide range of sizes. The Protein dataset is real data describing the folding structure of different amino acids; the task is to predict the structure of new sequences. The second dataset stems from the 1999 KDD-Cup (`http://www.sigkdd.org/kddcup/index.php`) and contains real network data; the task is to distinguish the "good" instances of network traffic from the "bad" instances (intrusions). The next two datasets, Syn-SM and Syn-LG, were produced with the QUEST generator [5] using a perturbation factor of 0.05 and function 1 for class assignment. The last two datasets, Alpha and Beta, are taken from the Pascal Large

Scale Learning Challenge[1]. These were included in order to have an appropriate set for support vector machines, as the other datasets required significant SVM parameter changes even on much smaller subsamples than were used with the other algorithms.

6.4.2 Algorithms

Three traditional learning methods were used to evaluate the abstraction framework's scalability:

- Decision trees (popular C4.5 implementation [103])
- SVMs (efficient implementation [70])
- K-nearest neighbor classification (implementation by Karsten Steinhaeuser)

The algorithms cover a range of computational complexities and rank among the most popular learning methods. For decision trees and support vector machines, they were configured with the default parameters provided by the respective implementations. For k-nearest neighbor classification there were $k = 5$ neighbors. All of the algorithms were compiled for 32-bit x86 systems with g++ v3.4.6 using optimization -$O3$.

These algorithms naturally fit the distribute-compute-collect paradigm. However, it is worth noting that with only minor modifications to the abstraction, other learning methods could be accommodated, such as Distributed K-Means Clustering [69] or finding frequent itemsets using Apriori-Based methods [135], which may require multiple distributed stages.

The scalability experiments covered the range from 1 to 128 nodes for the five smaller datasets. With Syn-LG the memory requirements for each individual partition are much larger, so 48 to 256 nodes are used with that dataset. For k-nearest neighbor

[1] http://largescale.first.fraunhofer.de/

classification, the test set size was 1,000 instances for the synthetic datasets and to 10,000 instances for all other datasets to keep computation feasible within the system.

6.4.3 Partitioning and Collection

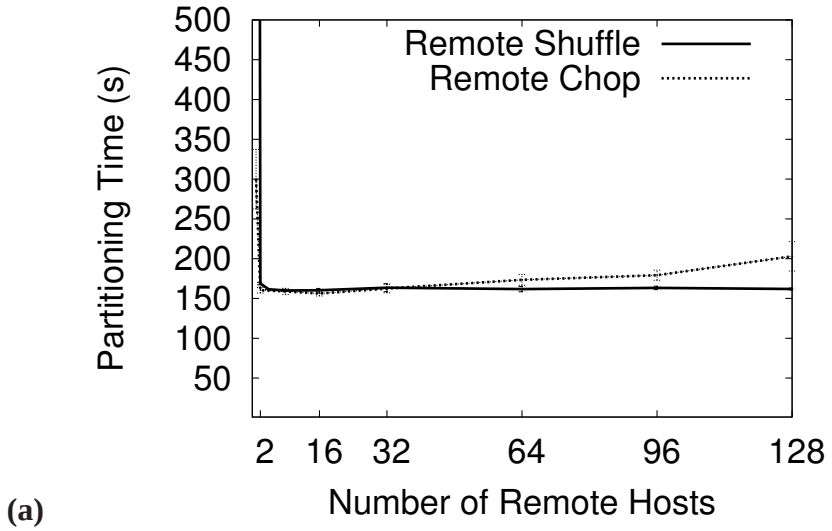
In large clusters or across a campus grid, data would ideally be Pushed to a number of remote nodes equal to the number of partitions to maximize parallelism. Figure 6.2(a) shows, however, that chop partitioning to a large number of remote resources begins to reduce performance due to moving beyond homogeneous clusters and encountering a greater variety of hardware. Shuffle partitioning has its own drawback in the larger environment, because it requires remote connections to remain open to every remote node throughout the entire partitioning.

Figure 6.2(b) shows that remote partitioning even to a modest set of reliable nodes is faster than local partitioning, without the pitfalls of Pushing data to unreliable environments.

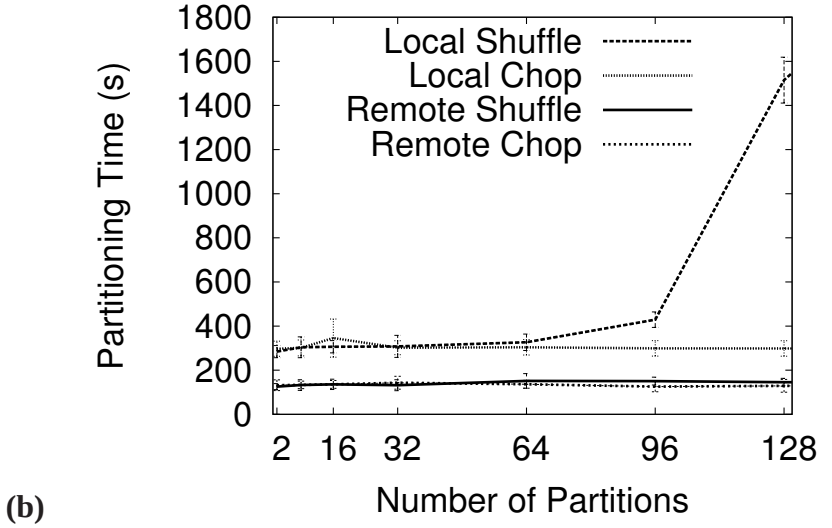
Figure 6.3 shows the time required to collect results of a distributed ensemble of classifiers using these two approaches, varying the number of partitions. The input data is the set of prediction files from a run of the KDDCup data, chosen because it the largest by-file memory requirement among the datasets (approximately 91MB). However, even this dataset does not result in significant concern due to prediction files being too large to collect in-core.

6.4.4 Campus Grid Execution

The primary thrust lies in the scalability analysis moving beyond the component benchmarks to larger executions. Figure 6.4 shows the execution time for decision trees, k-nearest neighbor classification, and support vector machines on multiple datasets for



(a)



(b)

Figure 6.2: Performance of Partitioning

6.2(a) shows the time to partition 5.4GB of data into 256 partitions on a single local disk or a varying number of remote disks. Figure 6.2(b) shows the time to partition 5.4GB of data into a varying number of partitions, using a single local disk and writing to 16 remote disks

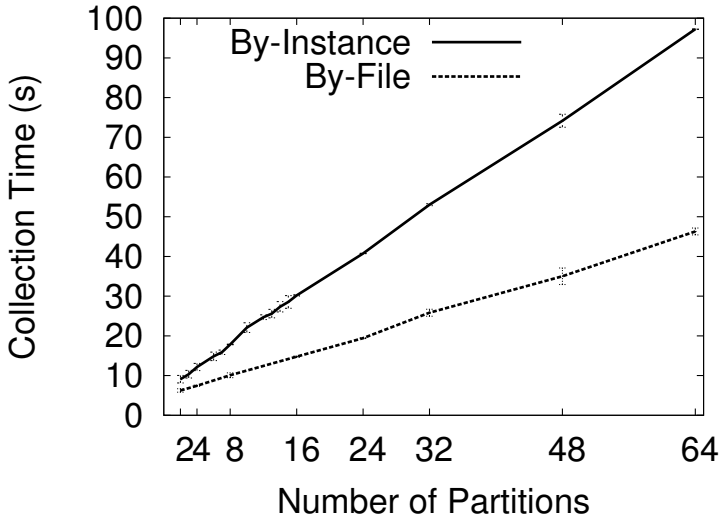


Figure 6.3. Performance of Collecting

This figure shows the time to collect classifier output (3.2MB per partition) from each of a varying number of remote disks. By-file collection uses 91MB of memory, while by-instance uses less than 1KB.

varying number of partitions. Within the grid of plots, rows correspond to datasets and columns correspond to learning algorithms. Each individual plot contains three lines for the different data distribution methods.

The results for Syn-LG with decision trees and k-nearest neighbors are omitted for space reasons as the trends observed are very similar to Syn-SM, albeit at a larger scale. In addition, for massive datasets it is difficult to measure Push partitioning. This task is feasible for smaller datasets and controlled environments, but becomes more difficult as the size of the dataset or number of hosts and diversity of the system increases.

Decision Trees. The first column of Figure 6.4 shows strong parallelizability of decision trees across all datasets. In most of the experiments, the data distribution does not significantly influence the execution time through 16 or 32 partitions, demonstrating extensive, though not exclusive, use of the 48-node dedicated cluster. Beyond that

threshold, performance diverges as jobs begin utilizing unreliable, heterogeneous nodes from the campus grid. Even beyond the cluster/grid threshold, however, there are improved turnaround times for several algorithms using the Hybrid approach.

As an example of a case where additional parallelism did *not* provide any added benefit, the KDDCup plot for decision trees shows that no improvements in execution time are achieved beyond 32 partitions. For decision trees in particular, the small workloads result in very minimal classifier training times. In addition, smaller jobs yield more relative overhead and higher costs to complete the serial stages of the process. It is unsurprising, then, that almost exactly the same amount of time is required for the execution phases when exceeding 32 partitions. For instance, doubling the collection time (twice as many predictions to process per instance) requires more time than is saved by the marginal improvement in execution time afforded by the resources.

Another factor impacting the scalability of executions is the data set size. The Syn-SM set continues to improve execution time using Hybrid through 128-way parallelism, whereas a smaller dataset, Beta, achieves limited further improvement beyond 32 nodes. The primary difference here is that for small data sets, further partitioning results in no effective gain when balancing batch job execution time against additional overhead from greater parallelism (partitioning, collection, and batch system overhead).

For almost all configurations the Hybrid approach yielded shortest turnaround times, and Pull yielded the longest turnaround times. Combining the advantages (and mitigating the disadvantages) of the Push and Pull techniques is particularly apparent for the larger datasets and as the number of partitions gets larger.

K-Nearest Neighbor Classification. The results in the second column of Figure 6.4 also show encouraging trends in execution time with respect to the number of partitions. All datasets observe consistent improvements in execution time while staying within the

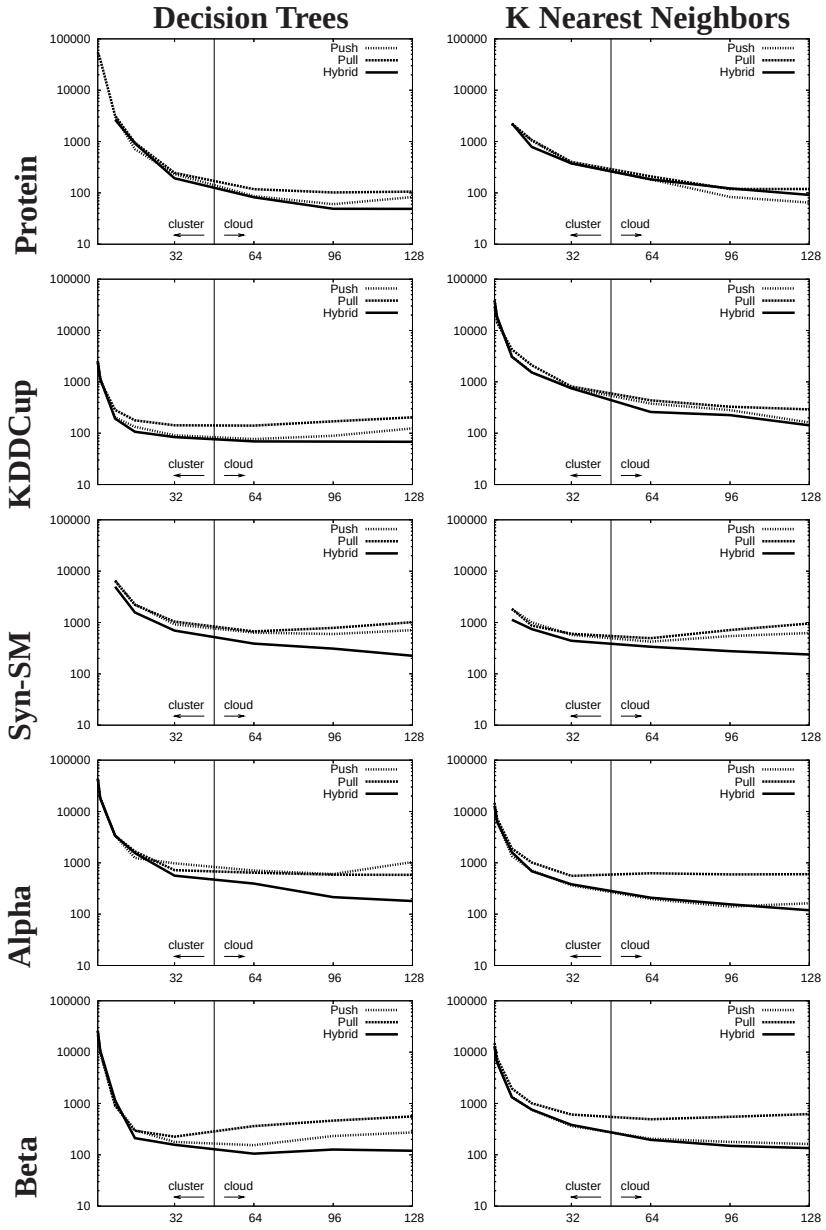


Figure 6.4: Scalability of Classifiers from a Cluster Subset to the Campus Grid

This figure shows the runtime of executing Data-Split-Join on five different datasets with decision tree and k-nearest neighbor classifiers. Each configuration is scaled up from 1 to 32 nodes on a homogeneous reliable cluster, and then up to 128 nodes on a campus grid. Each abstraction is run in three different configurations: Push, Pull, and Hybrid, as shown in Figure 6.1. Each graph shows the number of hosts on the X axis and the execution time in seconds on the Y axis. Generally speaking, the hybrid implementation is the most robust across the various configurations.

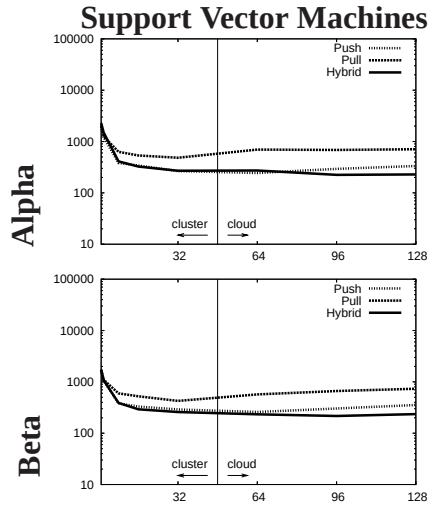


Figure 6.5: Scalability of Support Vector Machines

This figure shows the runtime of executing Data-Split-Join on the Alpha and Beta datasets. Results for SVM are not shown on the first three datasets from Figure 6.4, because the algorithm does not converge.

small cluster (up to 32 nodes) and with one exceptions also with 64 partitions. Only for 128 partitions is there increased execution times in several cases, most notably for the Push method. This behavior is due to some jobs getting placed on slower machines in the campus grid. In addition, the plots only show times for successful runs, but it is worth noting that with Push it sometimes took several attempts to complete the task without experiencing a failure.

The aforementioned tradeoffs are also apparent in these results, in particular with dataset Syn-SM. Neither Push nor Pull are able to improve beyond 64 partitions, and in fact both achieve significantly worse performance. However, the flexibility of the Hybrid method allows it to efficiently distribute data and computation, resulting in additional gains when going to 128 partitions.

Dataset size should also be taken into consideration when determining the appropriate configuration for a given problem. For smaller datasets, the choice of data distri-

bution method is largely irrelevant, as all three lines exhibit very similar behavior. But for large problems the Push and especially Hybrid models are better suited as using the maximum number of available partitions achieves the best performance and therefore is advisable.

Support Vector Machines. As shown in the right column of Figure 6.4, support vector machines exhibit behavior different from the other algorithms. Most notably, the majority of experiments do not achieve the best execution time for the largest number of partitions. And with SVMs this is not only due to heterogeneity in the campus grid, but also to the strong dependency of the algorithm runtime on the characteristics of the data.

Once again, the data distribution method is less of a factor than the amount of parallelism in determining the execution time, although the pull method is consistently the worst performer. In the actual executions there was also a tendency towards a smaller number of partitions to achieve the best result than the other algorithms. More specifically, the best performance was achieved with 8 to 16 partitions in all configurations.

6.4.5 Accuracy

It is generally established that ensemble learning can result in improved accuracy [27]. The fundamental goal in this chapter is to work with that assumption and evaluate the systems aspects of distributed data mining. For the experiments there are primarily synthetic datasets, and therefore observe only modest improvements.

Figure 6.6 shows the trends for each classifier on all applicable datasets. In most cases, accuracy is quite stable with an increasing number of partitions. Exceptions are increased accuracy for decision trees on the Alpha and Syn-SM datasets, and decreases for decision trees on the Beta and the 8-partition k-nearest neighbor for Syn-SM.

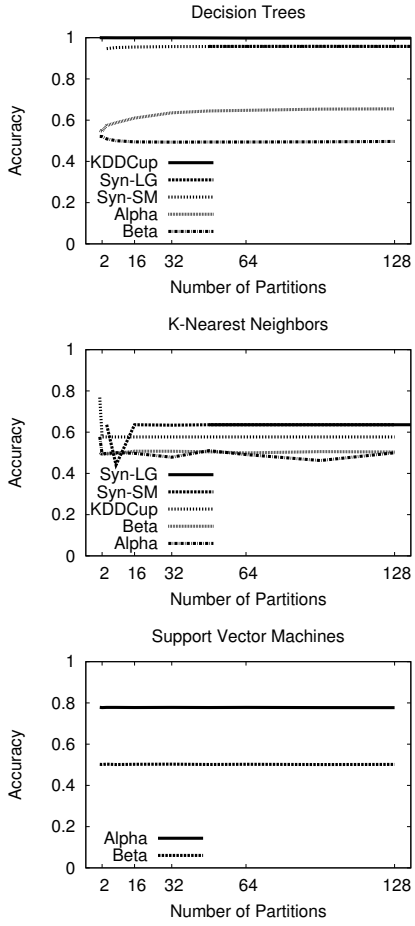


Figure 6.6: Trends in Accuracy with a Varying Number of Partitions.

6.4.6 Generalization

Let the evaluation conclude with Table 6.2, a set of general observations about the tradeoffs in switching from a well-controlled subset of a campus grid to the entire pool at large.

TABLE 6.2: EMPIRICAL ANALYSIS OF TRADEOFFS BETWEEN DIFFERENT CRITERIA

	Cluster	**Campus Grid**
Pull	- chop is necessary for large number of partitions - for large clusters, submitting node can become a bottleneck as the data server - worst turnaround time in most experiments	- chop is necessary for large number of partitions - for large clusters, submitting node can become a bottleneck as the data server - less concern about heterogeneity (fast nodes run bigger share), reliability (data not on remote nodes)
Hybrid	- shuffle is preferred partitioning method (can randomize, overlap, etc.) - less risk of bottleneck in large clusters where submitting node has limited resources - sweet spot trading off parallelism for robustness	- shuffle is preferred partitioning method (can randomize, overlap, etc.) - not reliant on central file server during execution - best choice for turnaround for most configurations (mitigates disadvantages of the other two methods)
Push	- good for small runs with limited parallelism available - shuffle is preferred partitioning method (can randomize, overlap, etc.) - good for algorithms with super-linear complexity - brittleness less concern in controlled environment	- tradeoff between partitioning robustness (chop) and performance (shuffle) - tradeoff between parallelism and reliability (more available resources but less reliable on full campus grid)

CHAPTER 7

CONCLUSIONS AND BROADER IMPACT

As distributed computing, particularly cloud and grid computing, has become more widespread, there has been an increase in interest in abstractions for scaling up repeatable patterns of work to larger systems for tackling larger problems. Patterson [94] has proposed that abstractions will be the assembly language for large distributed systems. This dissertation has explored how abstractions can be used to improve the usability, performance, and efficiency of a campus grid to scientists with large, sometimes data-intensive, computational workloads.

Unlike arbitrary workloads, abstractions are designed with the high level structure of a workload in mind, and it is feasible to accurately model the performance of large scale abstractions across a wide range of configurations. Some abstractions will be able to target provably optimal solutions, particularly on predictable systems. In general, though, these models aim for an execution that avoids disastrous configurations that get poor performance, waste resources on unproductive tasks, and potentially slow or disable resources shared with other users. Computing with an abstraction is more likely to result in an efficient execution that fits the data and computation requirements.

This work also examines several considerations that must be made when designing any abstraction for campus grid computing. Resource selection, data distribution, memory and disk management, job size selection, recovery from failure, and other topics addressed within the context of the specific abstractions in Chapters 4-6 will be

encountered by abstraction frameworks for almost any problem. This work discusses in particular some of the building blocks – and stumbling blocks – for designing abstractions for computing on a campus grid. While cluster computers have been well-studied, and grid and commercial cloud computers have recently been a popular field, campus grids have emerged as an architecture available to most institutions with minimal additional infrastructure required beyond the computing resources they already own for various purposes.

7.1 Choosing the Right Abstraction

The All-Pairs, Sparse-Pairs, and Data-Split-Join abstractions provide high level interfaces to a distributed system, improving both performance and usability compared to the conventional solutions that are likely to be developed by scientists without distributed computing expertise. These are not universal abstractions, however, and there are other abstractions that satisfy other kinds of applications for which the three presented here would not suffice. For example, Wavefront [120, 133] is an abstraction for a recurrence relation pattern that has different properties from these problems such as task interdependencies. There *are* problems, however, that could be solved by multiple different abstractions: so how can a user decide which abstraction to choose?

The formal relationship between different abstractions, and how to choose amongst them, remains an open problem in the field and an opportunity for future work. How, then, can a user choose which one to use for a given problem? So far, the abstractions toolbox has been developed by working closely with potential users to choose and develop the appropriate abstraction for their needs. With the growing suite of abstractions, though, it is becoming important that users in various fields can select the right abstraction from the toolbox based on their knowledge of their own problem.

The intent of providing abstractions is for the user to define a large workload in a simple manner. The user should be able to use codes that are very similar or identical to their serial implementations. The user should be able to garner good performance without having to separately implement complicated resource management, data management, and fault tolerance mechanisms into each application.

Abstractions on the whole shield the user from difficult details about executing a workload in a distributed environment. However, it is often the case that the abstraction that fits the problem best – either due to the design of the abstraction or the way a user has defined the problem – will be more efficient due to less transformation required to scale up to the cloud and because of greater possibilities for problem-specific solution optimizations.

The general suggestion is that a user should choose the abstraction that fits the way he already thinks about his problem. This most easily fulfills the intent of running a workload as-is, and simply scaling up to a cloud while abstracting away the messier details of the larger scale. This also usually requires the least amount of user overhead to handle the details of transforming his serial application into an entirely different problem before scaling it up.

An example of additional work required to transform the problem is seen when comparing a Sparse-Pairs problem to a general DAG or Bag-of-Tasks workflow. A particular piece of a computation within the more specific abstraction can be referenced simply by coordinates of the two input sets. That ordered pair, when combined with the problem definition, is sufficient to enumerate all incoming and outgoing edges in the DAG. The more general DAG abstraction would need to define the problem in a less efficient manner, costing execution time to complete the transcription into the more general definition and also the disk/memory resources to store it. Even then, when exe-

cuting, a general abstraction would still not have the advantages of automatically being able to optimize disk and memory management to the rigid patterns of a specific problem. Likewise, it only makes sense for a user who is already looking at his workload as a Sparse-Pairs problem to use the abstraction that is most specific for that problem – because it fits with how he has already designed his approach, and transforming a more general problem to an instance of the more specific pattern can be equally as costly as transforming the opposite direction.

This is, however, only a general suggestion, and must be reevaluated even when scaling up the same workload. An example of a case in which this is important was shown above when discussing the Sparse-Pairs problem. A scientist may start with a fairly dense set of pairs to compute between two sets, and decide to use the All-Pairs problem. However, as the problem is scaled up and the set of pairs becomes sparser, even though the All-Pairs abstraction is still available and will still solve the problem, it no longer is the appropriate choice. Generalizing an arbitrary set of computation pairs into the superset of computation pairs will increase the amount of work he requires significantly. Not only will it require much more time to compute all the extraneous pairs that he isn't interested in, but the abstraction solving that problem will provision more remote resources (data and worker nodes, for instance) to solve the larger version.

7.2 General Abstractions as Alternatives

As mentioned above, some abstractions can be interchanged with each other, with the cost as some loss of efficiency. Is it possible, though, that there are abstractions that could perform several of these patterns with similar efficiency? In this section Bag-of-Tasks and Map-Reduce are considered for the general patterns attacked by All-Pairs, Sparse-Pairs, and Data-Split-Join.

Bag-of-Tasks is a powerful abstraction for computation-intensive tasks, but ill-suited for All-Pairs problems. If a user attempts to map an All-Pairs problem into a Bag-of-Tasks abstraction by e.g. making each comparison into a task to be scheduled, this will end up with all of the problems described in the naïve solution to All-Pairs. Bag-of-Tasks is insufficient for a problem in which the abstraction achieves its greatest gains via data management, as Bag-of-Tasks does not recognize the overarching workload structure to exploit data patterns. Similarly, Bag-of-Tasks is less effective for Sparse-Pairs workloads because its model does not recognize the data reusage pattern and thus will likely read repeated data items from disk instead of maintaining them in memory like the specific Sparse-Pairs abstraction. Like the other two abstractions, Bag-of-Tasks *could* be used to solve a Data-Split-Join problem, but not with similar efficiency. Using Bag-of-Tasks, the inherent data pipeline from the split to the computation and from the computation to the join would be lost by treating them as completely separate tasks joined only by a order-of-completion dependency, and thus all data would have to be written to disk instead of pipelined directly between memory buffers.

Another common abstraction that targets a very general computation pattern is Map-Reduce [37], which encapsulates both the data and computation needs of a workload. This abstraction allows the user to apply a `map` operator to a set of name-value pairs to generate several intermediate sets, then apply a `reduce` operator to summarize the intermediates into one or more final sets. Map-Reduce allows the user to specify a very large computation in a simple manner, while exploiting system knowledge of data locality.

Hadoop [1] is a widely-used open source implementation of Map-Reduce. Although Hadoop has significant fault-tolerance capabilities, it has developed out of original assumptions that it is the primary controller of a dedicated cluster, so it does not thrive

in a campus grid environment made up of volunteered resources where policy and preemption mechanisms are a critical necessity.

Even setting aside the fundamental differences between cluster abstraction assumptions and campus grid environment realities, can one express an All-Pairs problem using the Map-Reduce abstraction? It is possible, but an efficient mapping is neither trivial nor obvious. A pure Map can only draw input from one partitioned data set, so it might itemize the Cartesian product into a set like $S = ((A_1, B_1), (A_1, B_2)...)$ then invoke $Map(F, S)$. Obviously, this would turn a dataset of n elements into one of n^2 elements, which would not be a good use of space. If set A is smaller, it would be better to package A with F and define $F^+ = Map(F, A)$ and then compute $Map(F^+, B)$, relying on the system to partition B. However, this would result in the sequential distribution of one set to every node, which would be highly inefficient. A more efficient method might be to add the All-Pairs spanning tree mechanism for data distribution alongside Hadoop, and then use the Map-Reduce to simply invoke partitions of the data by name. However this already departs significantly from the pure Map-Reduce model, and requires running multiple abstractions developed for different purposes side-by-side. While this is possible to orchestrate, this hybrid solution increases the complexity for an end user instead of decreasing it!

Data-Split-Join also appears similar to Map-Reduce. The assignment of tasks F onto $D1...DN$ is completed by the Mapper function, and C, the collection of independent distributed results into a final result, is the job of the Reducer function. But several components of Data-Split-Join are not strictly accounted for by the Map-Reduce abstraction – that is to say, the problem cannot be represented as Map-Reduce in its present form. The Map-Reduce model does not consider logical partitioning as a first-class component of the model, rather it delegates partitioning as an implementation

detail of physical partitioning of the underlying filesystem. The inclusion of additional files in each partition's computation (for example, the testing set in the data mining ensemble classification problem) also does not fit into the Map-Reduce abstraction model.

Some Map-Reduce implementations [1, 29, 106] adapt the Map-Reduce model to recognize logical partitioning in various ways, such as allowing for custom partitioning algorithms or actually including partitioning as primitive in their adjusted models. Mapping logical partitions onto physical partitions within the filesystem, however, remains a characteristic highly dependent on the implementation rather than strictly defined within the Map-Reduce abstraction.

The various Map-Reduce implementations also offer relaxed notions of what data can be computed at which stage of the workflow. But even this accommodation means that included files such as the testing set must either be encapsulated in the Mapper and Reducer functions or be stored on the distributed filesystem. The former is a rather significant design change associated with deploying the Mapper or Reducer tasks, while the former is potentially costly in terms of performance because of multiple replicas and significant metadata for each instance of various sparsely-used files (many of which have short lifetimes).

Data-Split-Join is a good fit when the key to success is careful consideration of data placement and access patterns, as it has been designed and implemented to consider workflow elements relating to data placement directly as first-class components of the abstraction model. So although it may be possible to complete a specific ensemble classification workload using Map-Reduce, it is difficult to do a thorough examination of the separate abstract parts of a Data-Split-Join workload while using a strict Map-Reduce paradigm. Even setting this aside, when implementations allow users to leverage a large number of options in setting parameters, this devolves into the origi-

nal problem of requiring non-expert users to appropriately configure complicated distributed systems.

An overarching observation of the difference between the three abstractions presented here and the more general Bag-of-Tasks and Map-Reduce abstractions is the notion of task planning and allocation as the primary control exerted by the abstraction. In the specific abstractions, the problem is modeled in order to plan specific allocations of data and computation such that they will be executed efficiently. In the more general abstractions, computations are placed into a larger system and run with little or no sense of planning for logical tasks. An example where this is most clearly evident is in the desire for an active storage computation: instead of active storage being a direct result of the abstraction's explicit coordination of data and computation, using a the general abstraction collocation happens more as byproduct of other parameters such as mirroring in the underlying filesystem. Thus, though it is possible in various implementations to tune parameters to make the system behave more like the result of the planning and allocation, the model is still quite different. This makes general abstractions prone to suffering from high translational overhead, and less adaptable to the differences in where the greatest benefit of parallelization can be harnessed in different workloads.

7.3 Lessons Learned

In Section 3.1, a number of general challenges associated with computing on a campus grid were laid out, many with subtle difficulties that tend to elude users (both expert and non-expert) at first glance. In this section, several items from the list of challenges are revisited to discern lessons learned that proved key to solving these specific issues and may be carried on to future projects. The last part of this section considers the sociological and group dynamics challenges of distributed systems research, which is

multidisciplinary by the nature of the systems as tools for wide-ranging applications.

Although seemingly obvious at face value that using the maximum available number of compute nodes is not always advisable, the data-intensive operations in this work emphasize the point. Even though All-Pairs and Sparse-Pairs are naturally parallel, there were significant limitations to the available parallelism due to factors beyond the structure of the problem. The resource cost of data management before and after computation is a necessary component to any model that, if considering only computation, would be likely to scale – on paper – to arbitrarily many nodes.

I/O patterns are one of the most natural starting points for exploiting a problem's regularity with abstractions. Because of the significant difference in memory versus disk bandwidth, exploiting I/O patterns to introduce streaming in and out of memory buffers instead of between disks is critical when dealing with data intensive tasks. This was a critical piece in the development of the Split-Map-Join abstraction, and it is an important missing piece of Work Queue, which can manage buffers in the master but realizes data exclusively as files on worker nodes.

When disks are required, active storage is generally an attractive option for two major reasons: computation on local data is generally much more efficient than computation on remote data; and computation is generally much less costly to relocate than data. In some systems and for some problems data must be moved, however these cases should lean heavily on avoiding disk accesses, partitioning the work to keep as much data in the whole of distributed memory as possible, and reusing data once it has been moved.

Although dispatch latencies of seconds seem easy to work around for systems of any significant size, when combined with start latencies (particularly with contention for resources) these delays can cripple performance. One way that has worked is increasing

job size so as to keep hold of the resource, however this has the downside of costly preemption. Instead, it is often advantageous to use Work Queue as a mechanism for holding onto a resource for many successive tasks while still allowing tasks to complete and be archived (avoiding large costs of preemption). In this way, even problems that require no collective communication can benefit from the master/worker paradigm.

The driving force behind this work has been the last and most important of the campus grid computing challenges, the usability of a computing system. Many times during discussions of this work, colleagues have initially scoffed (or even objected) to the characterizations of naïve users. However, the more experience that colleagues get with designing and computational science tools intended to be used by domain scientists, the less often that they claim the descriptions are strawmen. Using only one distributed system there are plenty of users who, through confusion or rather simple misunderstanding much more so than lack of intellect, continually make many similar critical errors. Of course using multiple systems compounds this. Abstractions take away some of a user's power, but also the user's power to make many of the disastrous mistakes. The ability for abstractions built atop Work Queue to operate seamlessly with several rather different underlying systems is a further step in the right direction to abstract away things that users easily get wrong.

The most difficult challenge of this work has not been science nor engineering, but communication. It is still at times jarring to note the complete disconnect amongst very intelligent people from different disciplines or even different areas of computer science, especially in that the disconnects often don't pertain to technical details but rather definition of what the general problems are. Communication is critical, and in this work it has not been rare, by any means, for many, many early discussions to be spent talking past each other. Collaborative research depends on experts from disparate

fields working together to attack a joint problem. However, when experts from opposite sides of the table fundamentally misunderstand the problems of the other side – or perhaps the two sides can't agree on a joint set of problems! – collaborative progress cannot proceed. Because distributed systems research generally stems from new and innovative uses of the system by non-distributed computing experts, a fundamental challenge will always be breaking a collaboration down to its smallest pieces in order to find initial agreements in order to begin communication and true collaboration.

7.4 Impact

Software developed as part of this work remains in use by researchers from several groups that have scientific computing needs. Additionally, further research continues on broadening the abstractions "toolbox" to attack new patterns of computation.

The user group that has benefited the most from the All-Pairs abstraction is the Notre Dame Computer Vision Research Laboratory (CVRL). Advances in the field of biometrics advance the state of the art several real-world applications, including personal security (such as biometric locks on laptops or doors) and national security (such as face recognition at airports).

An initial prototype for All-Pairs was used to evaluate new algorithms for 3D face image identification and feature detection [45]. This initial comparison served to create a single set of results on a completed algorithm, which was the normal mode of operation. However later work using the complete All-Pairs abstraction engine integrated the same large-scale comparison as a core evaluation within the development and enhancement of new algorithms [81]. Thus, access to this computing abstraction fundamentally changed the pattern of research. Instead of being constrained to testing experimental versions of algorithms on small subsets of the intended target data (waiting to run a

lengthy full workload with the final version), the biometrics researcher could complete the more informative full workload on each successive iteration along the way. This developmental process, akin to a parameter sweep, consumed more than 2 million CPU hours on the Notre Dame campus grid over more than two years – becoming the largest user of the campus grid resources (and in one year consuming over 50% of the cycles individually).

All-Pairs has also been used by the CVRL to study iris comparisons, as discussed in Section 4.5. The All-Pairs comparison of 58,639 irises presented here is believed to be the largest complete comparison on a publicly-available dataset, and the final result yielded knowledge of ranges of Hamming distances that contained both matching and non-matching pairs. This knowledge, which has repercussions on the efficacy of setting a particular cutoff for judging a match, would not have been possible on a less-than-complete set of comparisons.

Work Queue is a core tool for the BioCompute [19] project at Notre Dame, which is a web-portal-based tool that uses campus grid resources for solving large bioinformatics problems. Work Queue is also an underlying infrastructure for Makeflow [133], which is used in BioCompute and other workflow applications.

All-Pairs is an example of an abstraction that can be used with Weaver [23], a Python-based high-level framework for data processing workflows in Makeflow. Emerging work such as Weaver may further increase campus grid usability by providing an interface to use several optimized abstractions working together within a single workflow. All-Pairs is also used by BXGrid [22], which is a very large online repository for biometrics data that can also facilitate computation on that data through web portals and command line tools.

The Sparse-Pairs abstraction has been used as part of work on assembly and valida-

tion of large genomes [92], and currently bioinformatics researchers at Notre Dame are applying the SAND tools to complete comparisons of mosquito genomes.

7.4.1 Publications and Software

A precursor to the All-Pairs work was published at PCGRID07 [83]. The All-Pairs abstraction was introduced in a poster at GRID 2007 [82], published at the 2008 IEEE/ACM International Parallel and Distributed Processing Symposium [84], and expanded for the *IEEE Transactions on Parallel and Distributed Systems* [87]. Sparse-Pairs was first published in the 2009 Workshop on Many-Task Computing on Grids and Supercomputers [86]. The implementation of Sparse-Pairs is now released as an open-source package as part of the SAND project. Data-Split-Join was originally presented at the IEEE International Conference on Data Mining [85].

7.5 Conclusion

The opening chapter observes that scientists who are not experts in distributed computing are often faced with the dilemma of completely redesigning their applications to fit the often complicated and system-specific requirements of large parallel resources or giving up on scaling their applications to larger and more interesting problems. Finding the former too difficult or too time-consuming, some opt for the latter option – limiting themselves to problems within their current grasp. Others enlist the help of a distributed computing expert who can complete the redesign, but are then beholden to repeating this process every time they need to adapt to a new or different set of resources. Abstractions are a guide away from this inefficient cycle.

Abstractions are manageable for scientists to use, and often work with their unmodified serial applications. Abstractions can use serial UNIX processes and can run

on commodity hardware generally found in campus grids. General middleware APIs that abstract away the messy details of campus grid systems provide a set of tools that allow capable users to program their own abstractions (with the experts still there for guidance in design).

This more equitable cycle can also be more sustainable, because the user who is most interested in these problems has more resources at hand to develop improved solutions to them. And these implementations can last for generations of systems upgrades seamlessly hidden by the middleware (which is much easier for the expert to update than many separate implementations).

With the ability for countless users to add useful contributions to the campus grid computing "toolbox", the number of such tools will grow. This leaves a real opportunity for researchers to repeat the process at the next level of abstraction: identifying common patterns that connect these building-blocks together to enable new forms of discovery.

BIBLIOGRAPHY

1. The Hadoop Project. http://hadoop.apache.org, July 2009.

2. The Open Science Grid. http://www.opensciencegrid.org.

3. M. Addis, J. Ferris, M. Greenwood, P. Li, D. Marvin, T. Oinn, and A. Wipat. Experiences with e-Science workflow specification and enactment in bioinformatics. In S. Cox, editor, *e-Science All Hands Meeting 2003*, pages 459–466, 2003.

4. N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. Bright, J. Brunheroto, C. Caşcaval, J. Casta nos, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. Cipolla, P. Crumley, K. Desai, A. Deutsch, T. Domany, M. Dombrowa, W. Donath, M. Eleftheriou, C. Erway, J. Esch, B. Fitch, J. Gagliano, A. Gara, R. Garg, R. Germain, M. Giampapa, B. Gopalsamy, J. Gunnels, M. Gupta, F. Gustavson, S. Hall, R. Haring, D. Heidel, P. Heidelberger, L. Herger, D. Hoenicke, R. Jackson, T. Jamal-Eddine, G. Kopcsay, E. Krevat, M. Kurhekar, A. Lanzetta, D. Lieber, L. Liu, M. Lu, M. Mendell, A. Misra, Y. Moatti, L. Mok, J. Moreira, B. Nathanson, M. Newton, M. Ohmacht, A. Oliner, V. Pandit, R. Pudota, R. Rand, R. Regan, B. Rubin, A. Ruehli, S. Rus, R. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B. Steinmacher-Burow, K. Strauss, C. Surovic, R. Swetz, T. Takken, R. Tremaine, M. Tsao, A. Umamaheshwaran, P. Verma, P. Vranas, T. Ward, M. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. Krolak, C. Li, T. Liebsch, J. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. Wait, J. Wittrup, M. Bae, K. Dockser, L. Kissel, M. Seager, J. Vetter, and K. Yates. An overview of the BlueGene/L supercomputer. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–22, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

5. R. Agrawal, T. Imielinski, and A. Swami. *IEEE Trans. Knowl. Data Eng.*, 5(6): 914–925, 1993.

6. F. Allen, G. Almasi, W. Andreoni, D. Beece, B. J. Berne, A. Bright, J. Brunheroto, C. Cascaval, J. Castanos, P. Coteus, P. Crumley, A. Curioni, M. Denneau,

W. Donath, M. Eleftheriou, B. Fitch, B. Fleischer, C. J. Georgiou, R. Germain, M. Giampapa, D. Gresh, M. Gupta, R. Haring, H. Ho, P. Hochschild, S. Hummel, T. Jonas, D. Lieber, G. Martyna, K. Maturu, J. Moreira, D. Newns, M. Newton, R. Philhower, T. Picunko, J. Pitera, M. Pitman, R. Rand, A. Royyuru, V. Salapura, A. Sanomiya, R. Shah, Y. Sham, S. Singh, M. Snir, F. Suits, R. Swetz, W. C. Swope, N. Vishnumurthy, T. J. C. Ward, H. Warren, and R. Zhou. *IBM Syst. J.*, 40(2):310–327, 2001.

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

8. D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. *Communications of the ACM*, 45(11):56–61, 2002.

9. A. Andoni and P. Andyk. *CACM*, 51(1).

10. A. Arpaci-Dussea, R. Arpaci-Dusseau, and D. Culler. High performance sorting on networks of workstations. In *SIGMOD*, May 1997.

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

12. S. Batzoglou et al. *Genome Res.*, 12(1):177–189, January 2002. doi: http://dx.doi.org/10.1101/gr.208902. URL `http://dx.doi.org/10.1101/gr.208902`.

13. R. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *World Wide Web Conference*, May 2007.

14. A. Benoit, L. Marchal, J.-F. Pineau, Y. Robert, and F. Vivien. *IEEE Transactions on Computers*, 59:202–217, 2009. doi: http://doi.ieeecomputersociety.org/10.1109/TC.2009.117.

15. M. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. Middleware for filtering very large scientific datasets on archival storage systems. In *IEEE Symposium on Mass Storage Systems*, 2000.

16. G. E. Blelloch. *IEEE Transactions on Computers*, C-38(11), November 1989.

17. R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *ACM SIGPLAN Notices*, volume 30, August 1995.

18. K. Bowyer, K. Hollingsworth, and P. Flynn. *Computer Vision and Image Understanding*, 110(2):281–307, 2007.

19. P. Braga-Henebry. Biocompute: Harnessing Distributed Systems for Bioinformatics, 2009.

20. P. Brenner, D. Thain, and D. Latimer. Grid Heating Clusters: Transforming Cooling Constraints Into Thermal Benefits. In *The Uptime Institute Green Enterprise IT Award Paper*, pages 1–7, 2009.

21. G. Buehrer, S. Parthasarathy, S. Tatikonda, T. Kurc, and J. Saltz. Toward terabyte pattern mining. In *Proceedings of ACM SIGPLAN PPoPP*, pages 2–12, 2007.

22. H. Bui, M. Kelly, C. Lyon, M. Pasquier, D. Thomas, P. Flynn, and D. Thain. *Journal of Cluster Computing*, 12(4):373, 2009.

23. P. Bui, L. Yu, and D. Thain. Weaver: Integrating distributed computing abstractions into scientific workflows using python. In *Challenges of Large Applications in Distributed Environments (CLADE) 2010*, Chicago, IL, 2010.

24. M. Cannataro, A. Conguista, A. Pugliese, D. Talia, and P. Trunfio. *IEEE Systems, Man, and Cybernetcis, Part B*, 34(6):2451–2465, 2004.

25. Center for Biometrics and Security Research. CASIA iris image database http://www.cbsr.ia.ac.cn/english/Databases.asp, accessed Apr 2008.

26. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, , and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Operating Systems Design and Implementation*, 2006.

27. N. V. Chawla, L. O. Hall, K. W. Bowyer, and W. P. Kegelmeyer. *Journal of Machine Learning*, 5:421–451, 2004.

28. W. Chen, A. Radenski, and B. Norris. A generic all-pairs cluster-computing pipeline and its applications. In E. H. D'Hollander, J. R. Joubert, F. J. Peters, and H. Sips, editors, *Parallel Computing: Fundamentals & Applications, Proceedings of the International Conference ParCo'99, 17-20 August 1999, Delft, The Netherlands*, pages 366–374. Imperial College Press, 2000. URL `citeseer.ist.psu.edu/radenski99generic.html`.

29. C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, 2007.

30. S. Cong, J. Han, J. Hoeflinger, and D. Padua. A sampling-based framework for parallel data mining. In *Proceedings of ACM SIGPLAN PPoPP*, pages 255–265, 2005.

31. D. Culler, A. Dusseau, S. Goldstien, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split c. In *Supercomputing*, November 1993.

32. 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.

33. F. A. B. da Silva and H. Senger. *Parallel Comput.*, 35(2):57–71, 2009. doi: http://dx.doi.org/10.1016/j.parco.2008.09.013.

34. L. Dagum and R. Menon. *IEEE Computational Science and Engineering*, 1998.

35. P. Dasgupta, V. Karamcheti, and Z. Kedem. Transparent distribution middleware for general purpose computations. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, June 1999.

36. J. Daugman. *IEEE Transactions on Circuits and Systems for Video Technology*, 14(1):21–30, 2004.

37. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large cluster. In *Operating Systems Design and Implementation (OSDI)*, 2004.

38. 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. *Scientific Programming Journal*, 13(3), 2005.

39. E. Denis Howe. The free on-line dictionary of computing. http://www.foldoc.org, accessed March 2010.

40. Distributed.net. Distributed.net home page. http://www.distributed.net.

41. J. J. Dongarra and D. W. Walker. *Supercomputer*, pages 56–68, January 1996.

42. J. Douceur and W. Bolovsky. A large scale study of file-system contents. In *Measurement and Modeling of Computer Systems (SIGMETRICS)*, Atlanta, Georgia, May 1999.

43. T. Elsayed, J. Lin, and D. Oard. Pairwise document similarity in large collections with mapreduce. In *48th Annual Meeting of the Association for Computational Linguistics*, 2008.

44. W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. L. Price. *Journal of Grid Computing*, 3:283–304, September 2005.

45. T. C. Faltemier. Flexible and Robust 3D Face Recognition, 2007.

46. S. Feldman. *Software: Practice and Experience*, 9:255–265, November 1978.

47. I. Foster and C. Kesselman. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

48. I. Foster, J. Voeckler, M. Wilde, and Y. Zhou. Chimera: A virtual data system for representing, querying, and automating data derivation. In *14th Conference on Scientific and Statistical Database Management*, Edinburgh, Scotland, July 2002.

49. W. Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 35, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1010-8.

50. C. Giannella, H. Dutta, K. Borne, R. Wolff, and H. Kargupta. Distributed data mining for astronomy catalogs. In *SDM Workshop on Scientific Data Mining*, 2006.

51. T. Glatard and J. Montagnat. Implementation of turing machines with the scufl data-flow language. In *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 663–668, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3156-4. doi: http://dx.doi.org/10.1109/CCGRID.2008.52.

52. L. Glimcher, R. Jin, and G. Agrawal. FREERIDE-G: Supporting applications that mine remote data repositories. In *ICPP*, pages 109–118, 2006.

53. L. Glimcher, R. Jin, and G. Agrawal. *Journal of Parallel and Distributed Computing*, 68(1):37–53, 2008.

54. M. K. Gobbert. *Computing in Science and Engineering*, 7:14–26, 2005. doi: http://doi.ieeecomputersociety.org/10.1109/MCSE.2005.29.

55. S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *USENIX Operating Systems Design and Implementation*, October 2000.

56. W. Gropp, E. Lusk, and T. Sterling. *Beowulf cluster computing with Linux*. MIT Press, Cambridge, MA, USA, 2003. ISBN 0-262-69292-9.

57. D. Gusfield. *Algorithms on strings, trees, and sequences : computer science and computational biology*. Cambridge Univ. Press, January 2007. ISBN 0521585198.

58. R. W. Hartenstein and R. Kress. A datapath synthesis system for the reconfigurable datapath architecture. In *ASP-DAC '95: Proceedings of the 1995 Asia and South Pacific Design Automation Conference*, page 77, New York, NY, USA, 1995. ACM. ISBN 0-89791-766-9. doi: http://doi.acm.org/10.1145/224818.224959.

59. P. Havlak, R. Chen, K. J. Durbin, A. Egan, Y. Ren, X.-Z. Song, G. M. Weinstock, and R. Gibbs. *Genome Research*, 14:721–732, 2004.

60. P. Havlak et al. *Genome Res*, 14(4):721–732, April 2004. doi: http://dx.doi.org/10.1101/gr.2264004. URL `http://dx.doi.org/10.1101/gr.2264004`.

61. L. W. W. Hillier et al. *Nat Methods*, January 2008.

62. J. Hofer and P. Brezany. Digidt: Distributed classifier construction on the grid data mining framework gridminer-core. In *ICDM Workshop on Data Mining and the Grid*, 2004.

63. X. Huang and A. Madan. *Genome Res.*, 9(9):868–877, September 1999. doi: http://dx.doi.org/10.1101/gr.9.9.868. URL `http://dx.doi.org/10.1101/gr.9.9.868`.

64. X. Huang, J. Wang, S. Aluru, S.-P. Yang, and L. Hillier. *Genome Res.*, 13(9): 2164–2170, September 2003. doi: http://dx.doi.org/10.1101/gr.1390403. URL `http://dx.doi.org/10.1101/gr.1390403`.

65. X. Huang, J. Wang, S. ALuru, S.-P. Yang, and L. Hillier. *Genome Research*, 13: 2164–2170, 2003.

66. L. Huston, R. Sukthankar, R. Wickremesinghe, M.Satyanarayanan, G. R.Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *USENIX File and Storage Technologies (FAST)*, 2004.

67. M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 987–994, 2009.

68. 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.

69. G. Ji and X. Ling. *Emerging Technologies in Knowledge Discovery and Data Mining*, chapter Ensemble Learning Based Distributed Clustering, pages 312–321. LNCS, Springer, 2007.

70. T. Joachims. Training linear svms in linear time. In *KDD*, pages 217–226, 2006.

71. S. L. P. Jones. *The Computer Journal*, 32:175–186, April 1989.

72. D. Jordan and J. Evdemon. Web services business process execution language version 2.0. OASIS Standard, April 2007.

73. A. Kalyanaraman, S. Emrich, P. Schnable, and S. Aluru. *Journal of Parallel and Distributed Computing*, 67(12):1240 – 1255, 2007. Best Paper Awards: 20th International Parallel and Distributed Processing Symposium (IPDPS 2006).

74. H. Kargupta, K. Bhaduri, and K. Liu. The distributed data mining bibliography. http://www.cs.umbc.edu/~hillol/DDMBIB/, 2006.

75. A. Kumar, M. Kantardzic, and S. Madden. *IEEE Internet Computing*, 10(4):15–17, 2006.

76. H. T. Kung. *IEEE Computer*, 15:37–46, January 1982.

77. Y. C. Lee and A. Y. Zomaya. *IEEE Transactions on Computers*, 56(6):815–825, 2007. doi: http://dx.doi.org/10.1109/TC.2007.1042.

78. J. Liu, A. Vishnu, and D. K. Panda. Building multirail InfiniBand clusters: MPI-level design and performance evaluation. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, page 33, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2153-3. doi: http://dx.doi.org/10.1109/SC.2004.15.

79. D. Lui and M. Franklin. GridDB a data centric overlay for scientific grids. In *Very Large Databases (VLDB)*, 2004.

80. J. MacCormick, N. Murphy, M. Najork, C. Thekkath, and L. Zhou. Boxwood: Abstractions as a foundation for storage infrastructure. In *Operating System Design and Implementation*, 2004.

81. R. McKeon. *Three-dimensional Face Imaging and Recognition: A Sensor Design and Comparative Study*. PhD thesis, University of Notre Dame.

82. C. Moretti, J. Bulosan, D. Thain, and P. J. Flynn. Poster: All-Pairs: An Abstraction for Data Intensive Computing. In *IEEE/ACM Grid Computing*, 2007.

83. C. Moretti, T. Faltemier, D. Thain, and P. Flynn. Challenges in executing data intensive biometric workloads on a desktop grid. In *Workshop on Large Scale and Volatile Desktop Grids*, Long Beach, CA, March 2007.

84. C. Moretti, J. Bulosan, D. Thain, and P. Flynn. All-pairs: An abstraction for data-intensive cloud computing. In *IEEE/ACM International Parallel and Distributed Processing Symposium*, April 2008.

85. C. Moretti, K. Steinhaeuser, D. Thain, and N. V. Chawla. Scaling up classifiers to cloud computers. In *International Conference on Data Mining (ICDM)*, 2008.

86. C. Moretti, M. Olson, S. Emrich, and D. Thain. Highly Scalable Genome Assembly on Campus Grids. In *IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS09)*, 2009.

87. C. Moretti, H. Bui, K. Hollingsworth, B. Rich, P. Flynn, and D. Thain. *"IEEE" Transactions on Parallel and Distributed Systems*, 21(1):33–46, 2010.

88. E. W. Myers et al. *Science*, 287(5461):2196–2204, March 2000. doi: http://dx.doi.org/10.1126/science.287.5461.2196. URL `http://dx.doi.org/10.1126/science.287.5461.2196`.

89. National Insitute of Standards and Technology. Iris challenge evaluation data http://iris.nist.gov/ice/, accessed Apr 2008.

90. T. Oinn and et al. *Bioinformatics*, 20(17):3045–3054, 2004.

91. T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. *Concurr. Comput. : Pract. Exper.*, 18 (10):1067–1100, 2006. doi: http://dx.doi.org/10.1002/cpe.v18:10.

92. M. Olson. New Methods for Assembly and Validation of Large Genomes, 2009.

93. A. H. Paterson et al. *Nature*, 457(7229):551–556, January 2009. doi: 10.1038/nature07723. URL `http://dx.doi.org/10.1038/nature07723`.

94. D. Patterson. *Communications of the ACM*, 51, January 2008.

95. M. S. Perez, A. Sanchez, V. Robles, P. Herrero, and J. M. Pena. *Future Generation Computer Systems*, 23:42–47, 2007.

96. P. Phillips and et al. Overview of the face recognition grand challenge. In *IEEE Computer Vision and Pattern Recognition*, 2005.

97. R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. *Scientific Programming Journal*, 13(4):227–298.

98. M. Pop and S. L. Salzberg. *Trends in Genetics*, 24(3):142–149, March 2008. URL `http://www.sciencedirect.com/science/article/B6TCY-4RTCPK7-1/2/9d9f9874f725d0365f40e6bf6c106d73`.

99. M. Pop, S. L. Salzberg, and M. Shumway. *Computer*, 35(7):47–54, 2002. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2002.1016901.

100. M. Pop et al. *Computer*, 35(7):47–54, 2002. doi: 10.1109/MC.2002.1016901. URL `http://dx.doi.org/10.1109/MC.2002.1016901`.

101. R. Pordes and et al. *Journal of Physics: Conference Series*, 78, 2007.

102. X. Qiu, J. Ekanayake, S. Beason, T. Gunarathne, G. Fox, R. Barga, and D. Gannon. Cloud technologies for bioinformatics applications. In *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, pages 1–10, 2009.

103. J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman Publishers, 1993.

104. 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.

105. I. Raicu, I. Foster, and Y. Zhao. Many-Task Computing for Grids and Supercomputers. In *IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08)*, 2008.

106. C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Symposium on High-Performance Computer Architecture (HPCA)*, 2007.

107. M. Roberts et al. *Journal of Computational Biology*, 11(4):734–752, 2004. doi: 10.1089/cmb.2004.11.734. URL `http://dx.doi.org/10.1089/cmb.2004.11.734`.

108. A. Roy and M. Livny. Condor and preemptive resume scheduling. Kluwer Academic Publishers, 2004.

109. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *USENIX Summer Technical Conference*, pages 119–130, 1985.

110. A. Sarje and S. Aluru. Parallel biological sequence alignments on the cell broadband engine. pages 1–11, April 2008.

111. M. Schatz. *Bioinformatics (Online Advance Access)*, April 2009.

112. P. M. Smith, T. J. Hacker, and C. X. Song. Implementing an industrial-strength academic cyberinfrastructure at purdue university. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–7. IEEE, 2008.

113. L. B. Sokolinsky. *Program. Comput. Softw.*, 30(6):337–346, 2004.

114. J. Sroka and J. Hidders. *Fundamenta Informaticae*, 92(3):279–299, 2009.

115. G. Steele. *Common LISP: The Language*. Digital Press, Woburn, MA, 1990.

116. O. Storaasli and D. Strenski. Exploring accelerating science applications with FPGAs. July 2007.

117. D. Talia, P. Trunfio, and O. Verta. *Concurr. Comp.-Pract. E.*, 2008.

118. W. Tan, P. Missier, R. Madduri, and I. Foster. *ICSOC 2008 International Workshop on Service-Oriented Computing*, pages 118–129, 2009.

119. Taverna. What is a workflow? http://www.taverna.org.uk, accessed March 2010.

120. D. Thain and C. Moretti. Abstractions for cloud computing with Condor. In S. Ahson and M. Ilyas, editors, *Cloud Computing and Software Services*. CRC Press, 2009.

121. 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.

122. D. Thain, T. Tannenbaum, and M. Livny. *Concurrency and Computation: Practice and Experience*, 18:1989–2019, 2006.

123. D. Thain, C. Moretti, and J. Hemmes. *Journal of Grid Computing*, 7(1):51–72, 2009.

124. K. B. Theobald and G. R. Gao. An efficient parallel algorithm for All Pairs examination. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 742–753, 1991.

125. J. C. Venter et al. *Science*, 291(5507):1304–1351, February 2001.

126. e. a. Vijay S. Pande. Folding@home. http://folding.stanford.edu.

127. G. von Laszewski, M. Hategan, and D. Kodeboyina. *Workflows for e-Science: Scientific Workflows for Grids*. Springer-Verlag New York, Inc., 2007.

128. C. Weng and X. Lu. *Future Generation Computer Systems*, 21(2):271–280, 2005.

129. M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. *IEEE Computer*, November 2009.

130. L. Youseff, M. Butrico, and D. Da Silva. Toward a unified ontology of cloud computing. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, Nov. 2008. doi: 10.1109/GCE.2008.4738443.

131. J. Yu and R. Buyya. *Journal of Grid Computing*, 3:171–200, 2006.

132. L. Yu, C. Moretti, S. Emrich, K. Judd, and D. Thain. Harnessing Parallelism in Multicore Clusters with the All-Pairs and Wavefront Abstractions. In *IEEE High Performance Distributed Computing*, pages 1–10, 2009.

133. L. Yu, C. Moretti, A. Thrasher, S. Emrich, K. Judd, and D. Thain. *to appear in Journal of Cluster Computing*, 2010.

134. Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 247–260, 2009.

135. M. J. Zaki. *IEEE Concurrency*, 7(4):14–25, 1999.

136. 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.

This document was prepared & typeset with LaTeX 2$_\varepsilon$, and formatted with NDdiss2$_\varepsilon$ *classfile (v3.0[2005/07/27]) provided by Sameer Vijay.*