UNIVERSITY OF CALIFORNIA
SANTA CRUZ

DSOARS: A USER-LEVEL LIBRARY FOR DYNAMIC REPLICATED STORAGE

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Preethy Vaidyanathan

March 2002

The thesis of Preethy Vaidyanathan
is approved:

Professor Tara Madhyastha, Chair

Professor James Whitehead

John May, Ph.D.

Terry Jones, M.S.

Frank Talamantes
Vice Provost and Dean of Graduate Studies

Contents

List of Figures

List of Tables

Acknowledgments

I wish to thank my advisor, Tara Madhyastha for all her guidance and unwavering support in my research goals. I am also extremely grateful to Terry Jones for his insights and suggestions to my ideas and for being a wonderful host for me last summer. I am also thankful to all my committee readers for agreeing to evaluate my thesis and their valuable feedbacks.

I would like to thank Jim Kent for providing me with the source of his genome alignment codes and also his constant and I should add patient feedbacks to all my queries. Special thanks to Patrick Gavin and Jorge Garcia, the CBSE cluster tech support staff at UCSC. I would also like to thank all the scientists at Lawrence Livermore National Laboratory we had interactions with for their valuable feedbacks.

Thanks to all the members of the STAR group, past and present for their assistance and friendship. Many thanks in particular to Ivan Dramaliev, Miriam Sivan-Zimet and Hong Bo for their infinite patience and support. Special thanks to Leslie Anne Clark, Jessica Masters and Vidhya Jayakrishnan for proof-reading my thesis.

And finally, I thank my family in particular my aunt, uncle and Walter for their generosity and my parents and my sister for believing in me unconditionally.

This research was partially funded by NSF Grant No. CCR-0093051 and Lawrence Livermore National Labs.

Chapter 1

Introduction

With the advancements in processor technology in the last 20 years, CPU performance has been doubling every 18 months in accordance with Moore's law [29]. Disk speeds have not kept pace with processor growth [51]. Another important computing trend is the shift from supercomputers to clusters. Clusters, built using off-the-shelf components, are increasingly used to provide large computational resources to solve complex problems. The setup of a cluster is orders of magnitude lower than that of supercomputers. The recent TOP500 list, as of November 10, 2001, listed 8.6% of the fastest processing machines in the world to be clusters [18]. An emerging technology with similar characteristics to clusters is the wide-area distributed computing or grid technology.

Computational biology, or bioinformatics, is an extremely important and growing research area that can utilize clusters or grid technology. One challenge faced by this field is to understand the makeup of the human genome, revolutionizing our understanding of the human developmental processes and our ability to treat and diagnose diseases.

The typical input/output (I/O) access patterns of these applications are large dataset reads, in the orders of MBs or GBs, and varying amounts of writes (KBs to GBs). These applications have different access patterns from other scientific applications and require large amounts of storage and processing power. As a specific example, consider the Human Genome project, the goal of which is to discover all the approximate 30,000 human genes (the human genome) and sequence the 3 billion chemical base pairs making up the human genome [7]. The input/output requirements of this project are staggering, with the genbank database doubling every 14 months and is currently 67.6 GB (approximately) [9].

We examine the scalability of two embarrassingly parallel computational biology applications for sequence alignment, psLayout and WABA, that played an important role in the mapping of the human genome [25, 34]. These applications are responsible for over 50% of the UCSC cluster use.

The characterization study of the two computational biology applications on a low-cost UCSC computational biology cluster shows that file location is a major factor affecting performance of applications. To generalize such a conclusion without being specific to one loosely coupled cluster architecture, we compared results from a supercomputer and a different cluster architecture. The three architectures provide a range of multiprocessor designs.

We present in this thesis the design of a library for location-transparent storage in a tightly or loosely coupled clusters. This library is responsible for managing replicas to improve performance. This library is flexible and can be extended to be used in a grid architecture. As a first pass, we implemented a user-level library for dynamic selection of the optimal file location. Sometimes the best solution might be to replicate the dataset onto the local disk and

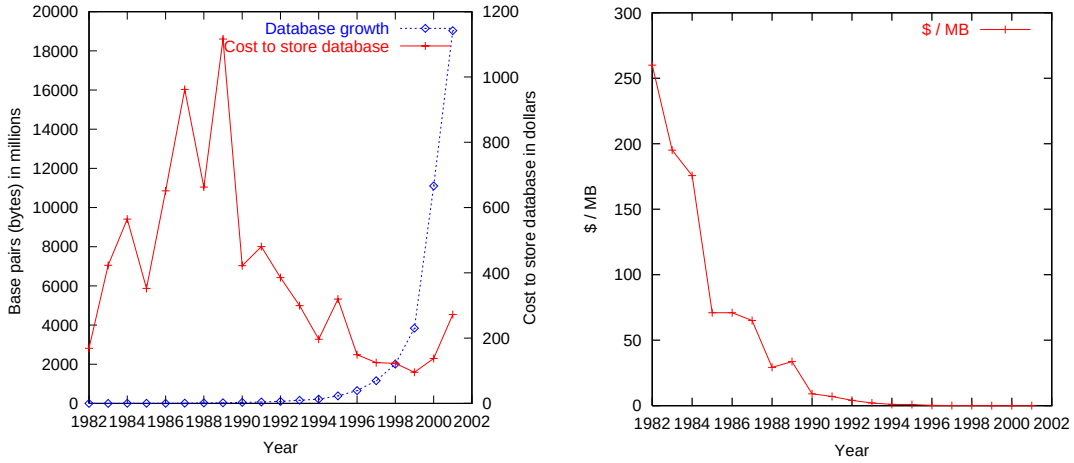


Figure 1.1: (a) Cost to store the growing genbank database (b) Cost of disk over time.

use that copy.

1.1 Motivation

Intelligent replication seems reasonable given the trends in technology and data volume. The cost of disk storage has been decreasing exponentially, primarily by improvements in areal density [11]. Figure 1.1a shows the growth of the genbank database [5] and the cost to store it [6, 37] over the last 20 years. During this time, the size of the genbank database has been approximately doubling every 14 months [9]. The curve showing the cost to store the database is jagged because the rate of growth of genbank and the rate of decline of disk cost are not proportional. There is a very significant difference of the genbank size for the years 1982 and 2001, whereas there is relatively very small difference between the cost to store genbank in 1982 and 2001. The reason for marked lack of difference between the costs to store

the database is the exponentially decreasing disk costs as shown in Figure 1.1b. The shape of the curve depicting the cost to store the database is therefore governed by the relative rates of database growth and decreasing storage cost.

Current disk drive technology cannot continue this trend without bound. As bits become smaller, the probability that they will spontaneously reverse polarity increases; this is called the superparamagnetic effect. Although the precise density at which this effect will have an impact is unknown, it is motivating a variety of research on alternative storage technologies. IBM predicts current growth will continue for the next four years and then decline [27]. Nevertheless, the volume of bioinformatics data available to researchers has been growing exponentially. Scientists are using hundreds of data formats that are rapidly changing with new technology. In the next ten years, we will most likely see the storage cost of genomic data increase, making data management and scalability a serious problem for this class of applications. A user level library to manage all the replicated copies and to provide data reliability with performance improvement addresses this problem.

1.2 Outline

The remainder of the thesis is organized as follows. In Chapter 2, we describe related work. We describe the applications characterized in Chapter 3. In Chapter 4 we describe the different architectures and the experimental characterization efforts that show performance to be a factor of input file location. This motivates the design and implementation of an I/O library for automatic storage replication, Dynamic Storage Replicator and Selector (dSOARS),

described in Chapter 5. We evaluate the performance of dSOARS in Chapter 6. Finally we conclude with directions for future work in Chapter 7.

Chapter 2

Related Work

The I/O bottleneck is not a new problem and many researchers have characterized the I/O behavior of scientific applications. Because of decreasing disk costs, replication can be used effectively to improve performance. Much research has been done on distributed computing architectures that use replication and adaptation techniques.

The remainder of this chapter is described as follows: in §2.1 we discuss several I/O characterization efforts with an emphasis on I/O-intense applications. In §2.2 we describe different approaches to improve performance through replication and adaptation.

2.1 I/O Characterization

Many researchers have studied the I/O behavior of important high-performance applications out of growing concern over the increasing gap between I/O and processor performance. The CHARISMA project [41] has examined system-level input/output accesses on the

iPSC/860 Concurrent File System (CFS) and the CM5 Scalable Disk Array to obtain some generalizations of access patterns in production parallel input/output workloads. They have observed predominantly write accesses, small request sizes, and generally sequential requests. The Pablo group did extensive performance characterization of parallel and I/O-intense applications. Some example application areas from these characterization efforts include modeling of electron-molecule collisions, a 3-D numerical simulation of the Navier-Stokes equations, an implementation of the Hartree-Fock self consistent field method to calculate the electron density around a molecule, and quantum chemical reaction dynamics [26, 48, 49]. Difficulties obtaining high performance from general I/O application interface, led to the development of MPI-IO [40].

These characterization efforts revealed I/O to be a significant component of execution time, but they did not focus specifically on computational biology. A study of the NWS gene sequencing algorithm [22] showed that I/O patterns could be described as a work queue, where each process would compute on some portion of data for either a very short or extremely long period of time, depending on the possibility of a match. Yap *et al* [36] studied the efficiency of parallel algorithms for homologous sequence searching and multiple sequence alignment, demonstrating the importance of load balancing. Another approach to solve a similar gene sequencing problem, taken by Spring *et al* [50], is to use an Application Level Scheduler (AppLeS). Load balancing is provided by static allocation and dynamic reallocation of jobs based on a benchmark and the CPU utilization. This gene sequencing characterization work focus on load balancing.

2.2 Replication and Adaptation

Many researchers have considered replication as a way to improve I/O performance while providing reliability. River [31], a project from Berkeley, focuses on a data-flow programming environment to provide maximum performance in a heterogeneous environment. A simple RAID mirroring policy is adopted for fault tolerance against disk failures. The River project proposes a model which dynamically modifies the choice of the mirror copy and the percentage of information accessed from each copy. The data-flow model between producers and consumers also provides dynamic load balancing by regulating the flow of records (termed as messages) in the environment. For example, if a consumer is getting I/O from two producers and one is slower than the other, this model would get more out of the fast producer, thereby not taxing the slower producer and at the same time increase the number of records to obtain from the fast producer.

ABACUS provides a programming model and a run-time system to partition functionality among producers and consumers for effective utilization of cluster resources [21]. When a client requests data, instead of going through a store-and-forward node like a server, it accesses a Network Attached Secure Disk (NASD) device, increasing the bandwidth of data transfer. The programming model supports coding each functionality as an object. Each object is functionally independent, with its own checkpoint and restoration algorithm if failure or migration occurs. The data-intensive applications are programmed using the integration of various objects. The run-time system monitors the resource utilization and this information is fed into a cost-benefit model that determines where functionalities should be placed. This

design corrects itself even when at first the functionality is placed at the wrong place. Hence functionality is moved back and forth to provide load balancing for effective cluster resource utilization.

The need for a large pool of computational resources has spurred the interests in data grid technology. The goal of The Globus project from Argonne National Laboratory (ANL) is to facilitate access to mass storage and computational resources like supercomputers, by providing a base level of services [17]. Other areas dealing with petabyte-scale data resources requiring data grids for potentially thousands of users is the climate modeling problem. On this large a scale, load balancing and performance become very significant issues. The Earth System Grid prototype deals with this technology supporting high-performance and data replication [20]. Services are very important in wide-area distributed systems and another research project focusing on replication is the Distributed Parallel Storage System (DPSS) [38]. This model provides higher data access throughput, using a network-aware DPSS master and replication of data. The client program requesting data communicates to the DPSS master, which determines which replicated data copy is to be used, based on the current network configuration. In all the above projects, replication is done similar to mirroring and no intelligent heuristics are used.

Chapter 3

Applications

The computational biology department at UCSC is working on several interesting and highly innovative projects; however, the most visible is the mapping of the human genome [25, 7]. The human genome mapping will help us to better understand the human body, biological processes responsible for disease, and differences among species.

Our characterization efforts focus on the application psLayout, a program that finds alignments. This program represents the second and most time-consuming step of six [34] of the human genome assembly process, and utilizes more than 50% of the CBSE cluster time. This is an I/O-intensive application. To get a broader perspective of the typical computational biology applications at UCSC, we also choose a relatively more compute-intensive alignment application, WABA.

The rest of this chapter is organized as follows: in §3.1, we describe the alignment process. We discuss the heuristics and the I/O access patterns of psLayout in §3.2 and WABA in §3.3.

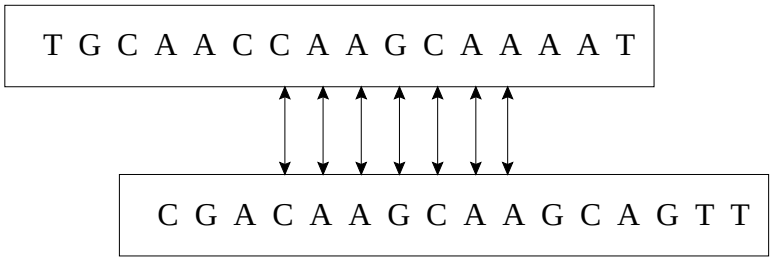


Figure 3.1: Alignment process: the two horizontal boxes are the input sequence and the vertical lines show the alignment.

3.1 Alignment Overview

The basic carrier of genetic information is Deoxyribonucleic acid (DNA), which can be represented as a sequence of nucleotide bases: A-Adenine, C-Cytocine, G-Guanine and T-Thymine. Thus, a DNA molecule is stored as a string over an alphabet of four characters {A,T,G,C} (nucleotides). To form a draft of the human genome, individual sequences generated from a variety of distributed sources need to be aligned in their positions on a chromosome map and assembled.

Combinations of the four basic carriers and unknowns, represented by blanks or 'holes', make up a genome sequence. The genome sequence can be viewed as a string of characters and the alignment problem can be reduced to a string matching problem as shown in Figure 3.1.

This is not a trivial problem. Figure 3.2 illustrates some of the difficulties in this process [33] using a nursery rhyme as an example. Figure 3.2a shows an input, or sequence, which contains repeating elements. Our goal is to align this input with itself. Unfortunately, this input comes in subsequence fragments, as shown in Figure 3.2b; many of these fragments overlap and have repeated subsequences, complicating the alignment process, because the

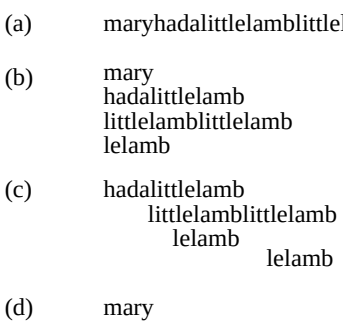


Figure 3.2: Alignment problems: (a) An input containing repeating elements to be aligned with itself (b) Deconstructing the input (c) Assembling overlapping pieces with placement uncertainty (d) Piece does not fit in the assembly.

matches could occur at several places (Figure 3.2c) or the sequence fragment may not align (Figure 3.2d). The complexity increases when there are unknowns in the sequences.

Though complex, alignments provide more information about subsequences than when they are studied separately. Alignments can also be used to identify characteristics of an unknown sequence based on its match with a known sequence. For these reasons, alignments are a very interesting and important problem and we have chosen to examine two alignment applications, psLayout and WABA.

Genome sequences are to be processed based on inherent characteristics of the genome. This is implemented using application-level library read calls. We study these calls for our characterization efforts. The significant difference between the two applications are the application-level I/O-intensity of these applications and the heuristics used for the alignment process. The applications, based on their heuristics, spend different amounts of time doing library I/O, chiefly reads, interleaved with the alignment process. With psLayout this constituted nearly 99% of total execution time and for WABA, 5%-75% (approximately), depending on the number of processors. Thus, read time measured at the application level includes some

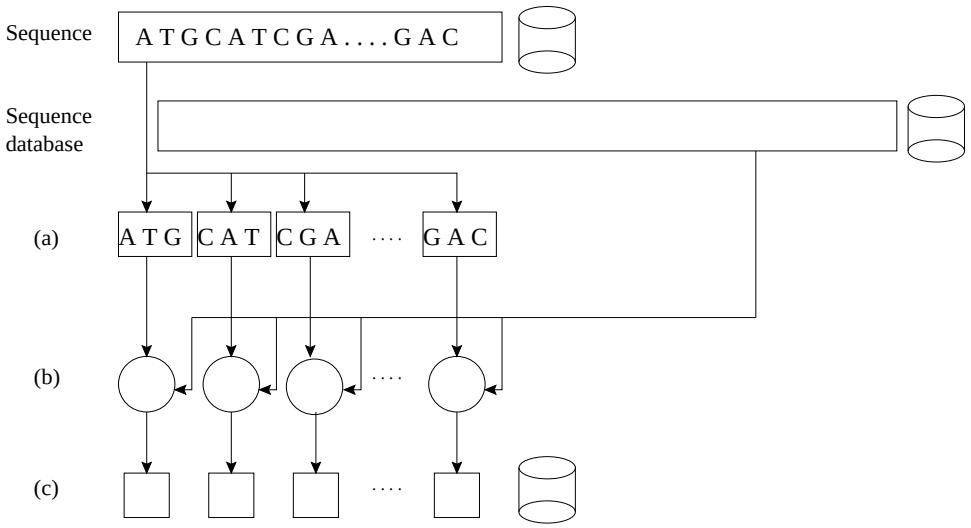


Figure 3.3: PsLayout execution flow (a) The split of one of the inputs to *n* jobs (sequences) (b) Compute nodes in the cluster aligning sequence to the sequence database, stitching together local sequences using dynamic programming (c) Output local sequences.

alignment.

3.2 PsLayout

PsLayout is a program that finds alignments. It represents the second and most time consuming step of six [34] of the human genome assembly process. PsLayout aligns sequence data that may have "holes" in it with the sequence database. The heuristics in pslayout are used to align mRNA with a genome, or to align genomes, though poorly.

PsLayout is an embarrassingly parallel application. The two sequences are given in two input files. The first input, the *sequence data*, is a collection of FASTA files [28] (ASCII files that represent sequences and their descriptions as text strings) containing up to 5 million bases. The second input, the *sequence database*, is a single FASTA file. This can be either genomic data or mRNA with no restriction on the number of the bases.

The input sequence can be split into *n* pieces and be aligned with the sequence database. These *n* individual alignments can be combined to produce the same result as the non-partitioned alignment. Figure 3.3a illustrates this experimental setup. PsLayout runs in parallel as separate applications on different nodes (Figure 3.3b). For each run, the output is written to a separate output file as shown in 3.3c.

The heuristics of this application are described below. The input sequence is split into overlapping pieces that are stored in an index. The index also stores where this piece appears in the complete sequence. The sequence database is split into non-overlapping pieces. Each segment of the sequence database is looked up in the index table, and if present in the index it is considered a hit. There is an alignment if the match is above a certain threshold value. If it is not present in the index, it is a miss and is ignored.

Once all the hits are obtained, they must be recombined, which is done using a dynamic program. The hits are projected on the target file when they are 500 bases apart. Thus, the final alignment in this application is obtained by combining the smaller alignments that have been written to individual output files.

The I/O access pattern of psLayout, is the input sequence reads interleaved with the making of the index table, followed by the reads of the sequence database (interleaved with the alignment process) and the writing of the output file.

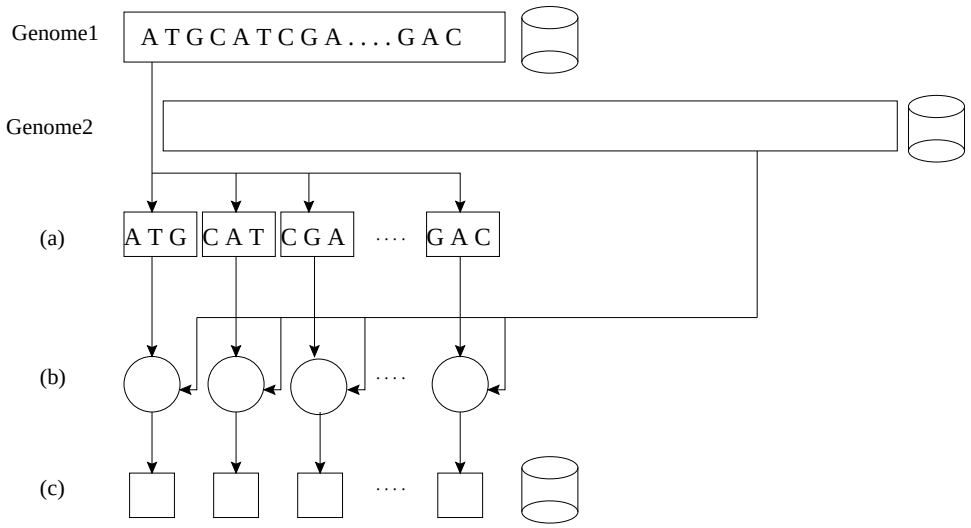


Figure 3.4: WABA execution flow (a) The split of the smaller genome sequence to *n* jobs (b) Compute nodes in the cluster aligning genome1 subsequence to genome2 (c) Output local sequences matches.

3.3 WABA

Wobble Aware Bulk Aligner (WABA) is a genome-to-genome alignment application. The scoring method of psLayout when used to align genome-to-genome is not efficient. WABA is extremely efficient in handling genome alignments and is considered more tolerant than BLAST, another popular genome-to-genome aligning application [35].

The two genomes to be aligned are given in two input files. The first genome, typically the largest of the two, is split into *n* pieces, as shown in Figure 3.4a. Each *genome1* subsequence piece is now aligned with the *genome2* sequence in Figure 3.4b. The output is then written to the individual output files in the global store as shown in Figure 3.4c. Like psLayout, WABA, runs in parallel as separate applications on different nodes.

WABA has three major passes, as shown in Figure 3.5. The first pass, the fastest, identifies possible target positions of alignment. The first file (genome1) is read at once into

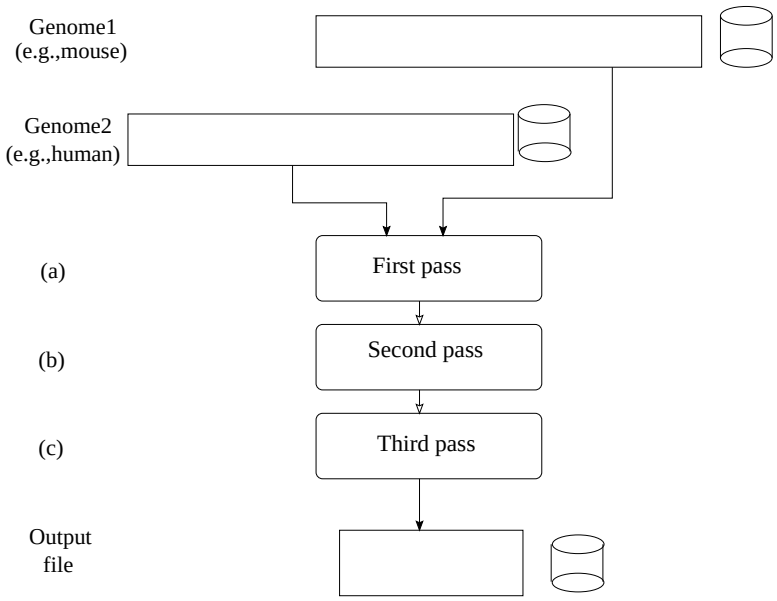


Figure 3.5: WABA application overview. (a) The initial possible positions are marked (b) HMM is used to get the new score (c) Stitches smaller alignments into large alignments.

a buffer, which is in turn read one line at a time. The first input sequence is broken into overlapping 8-base pieces, popularly known as 8-mer, to form the index table. Prior to the index table creation, to conserve memory space, the sequence is modified by packing the file to store 8-bases in a 16-bit word. The second file (genome2) is read in one pass, using one-line reads and aligned with genome1. The tricky part here is that the genome alignment results in a lot of hits, and hence the hit list grows very fast. So the hit list needs to be consistently monitored to check for a long homogeneous hit ranging through a wide range of the sequence. The alignment has a large error margin and the output is written to a file.

The second pass, which is much slower, has more tolerance to mismatches. The output from the first pass is now the input. The input to second pass is read, one line at a time, and using a seven-state Hidden Markov Model (HMM), a different score for the entire

alignment is generated and written to a file.

The first two passes are done using small input window sequences with a scoring value for each alignment. In the third pass, smaller alignments are merged and the output is written to the output file. The output files from the first and second passes are temporary files. Thus, there are three sequential passes and each pass is characterized by reads interleaved with alignment heuristics followed by writes.

Chapter 4

Characterization of Applications

Computational biology is a growing research field with different I/O access patterns from other scientific applications. In this chapter, we characterize the I/O performance of the two applications, psLayout and WABA, described in Chapter 3. In our characterization efforts, we focus on the factors affecting the performance of the application when it is scaled on a number of processors.

The remainder of this chapter is organized as follows. In §4.1, we describe the architectures we use. In §4.2, we describe our approach to application instrumentation and analysis. In §4.3 we explain the different experiments and their results. In §4.4, we discuss how a user-level library can improve the performance of these applications.

4.1 Architectures

We characterized the applications on a range of architectures. The first characterization study was done on the Center for Biomolecular Science and Engineering (CBSE) cluster at Santa Cruz. The next two architectures were at Lawrence Livermore National Labs (LLNL) and they include the supercomputer ASCI Blue-Pacific and Vivid cluster.

4.1.1 CBSE Cluster

The computational biology group at UCSC uses a cluster with 93 Linux nodes. Each node of the 93-node cluster has an 850 MHz Pentium III processor with 256 MB RAM and a 20 GB IDE drive. The nodes are internetworked with 100 Base-T Ethernet in two subclusters. The topology of this cluster is illustrated in Figure 4.1. Clocks on all the nodes are synchronized using NTP.

4.1.1.1 File System

There is no parallel file system, and files are shared on two global store NFS servers. The computational biology group is currently replicating all the genome data onto each local disk. The central database from which the data is replicated to the nodes is in the NFS server. Their central database is updated from the genbank website once every 2-3 months. They perform these updates manually, which is an extremely time-consuming task (typically 8-12 hours).

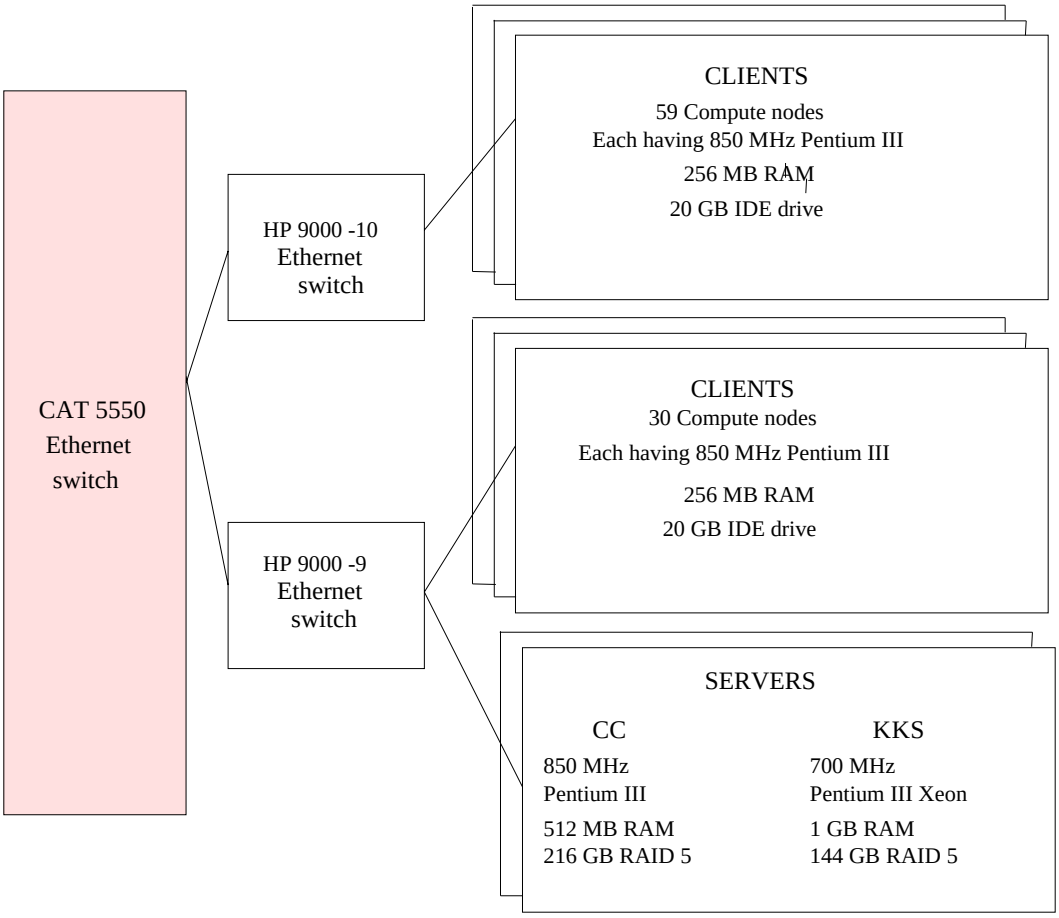


Figure 4.1: 93-node CBSE cluster topology.

4.1.1.2 Scheduler

The computational biology group at UCSC uses Condor [3] for job scheduling in the 93-node cluster. Condor checks for nodes with idle CPU cycles and assigns jobs from a central pool to those nodes. Condor is designed for computing using collections of distributed resources, as opposed to parallel computing on a homogeneous cluster; however, many of the computational biology applications lend themselves well to a work-queue programming model. Because the computational biology applications at UCSC have no inter-process communication, Condor provides a fast and effective mechanism for spawning jobs to various nodes in the cluster.

4.1.1.3 Cluster Usage

More than half of the cluster time is used to run embarrassingly parallel alignment applications. A typical application on the cluster is to build libraries of Hidden Markov Model (HMM) and support vector machines classification of G-protein coupled receptor superfamily [13]. Protein secondary structure predication using HMMs is another important application run on the cluster. Some of the other specific applications run on the cluster include serving web queries for database search and sequence alignment, finding short DNA sequences corresponding to sequence-tagged sites (STS) on the draft genome assembly, and aligning STS primers to the entire genome.

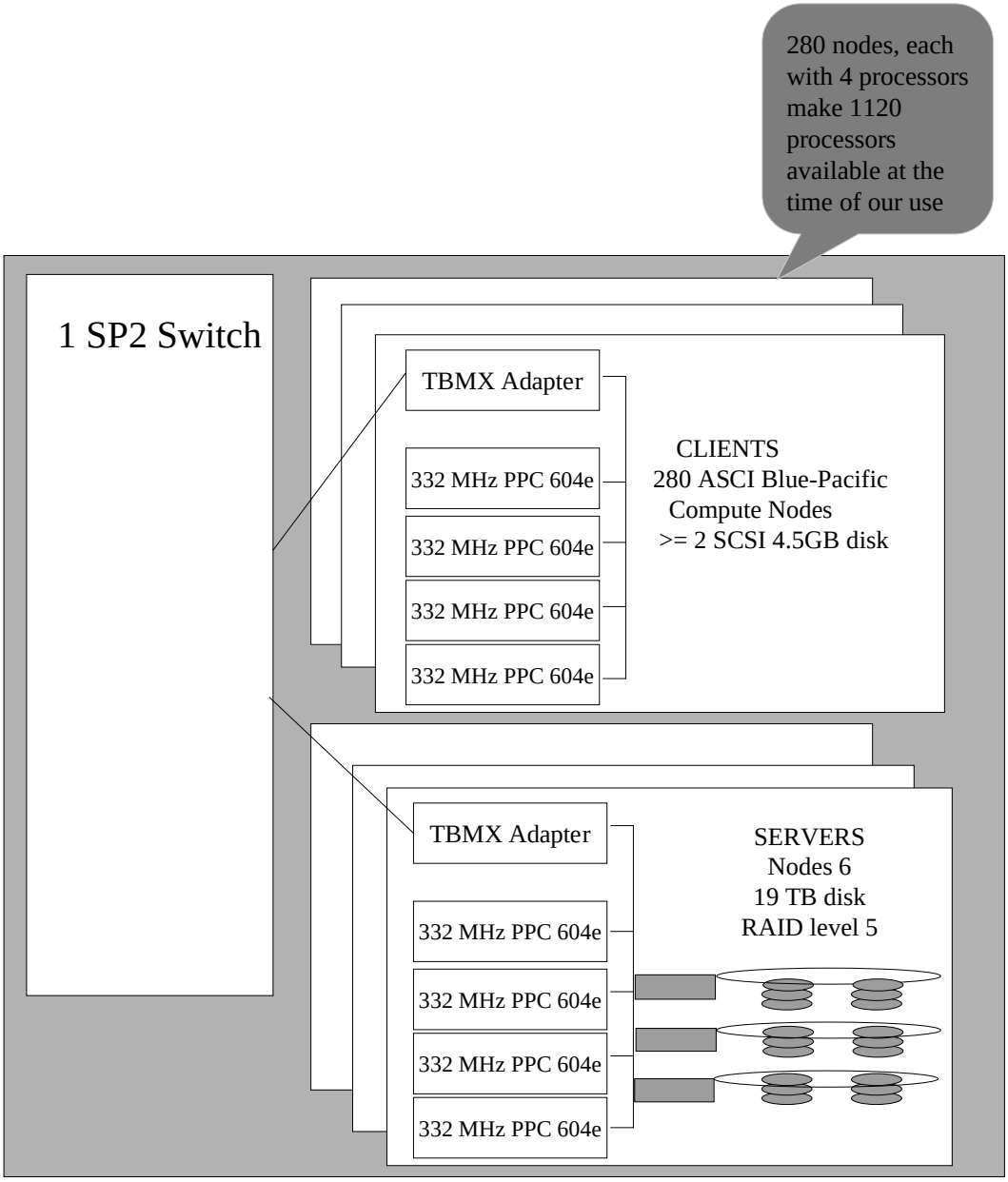


Figure 4.2: ASCI Blue-Pacific topology.

4.1.2 ASCI Blue-Pacific

ASCI Blue-Pacific (Blue), manufactured by IBM, is a supercomputer at LLNL with 336 nodes, each with four 332 MHz PowerPC604e processors. At the time of our usage, the CPU resource pool consisted of 280 nodes. Each node has a minimum of two SCSI disks (each 4.5 GB) and 1.5 GB RAM. The server consists of 6 nodes each with four 332 MHz PowerPC604e processors, internally connected by the TBMX adapter. The server has a disk capacity of 19 TB, of which 3 TB is local disk space and the rest is the GPFS file system. The nodes are interconnected by a SP2 switch [2]. ASCI Blue-Pacific theoretical peak performance is 3868 GFlops and the maximum attained performance is 2144 GFlops. Blue ranks fifth in the TOP500 listing of 2001 [18].

4.1.2.1 File System

The file system is General Parallel File System (GPFS) with a total capacity of 16 TB. GPFS is the product version of the Tiger Shark file system of IBM. The global store GPFS provides high performance to run parallel applications by striping files across multiple disks achieving an aggregate bandwidth of 2.5 GB/s [4, 12]. GPFS has all data including the metadata on each node, preventing a metadata bottleneck. For increasing throughput of the actual data access, GPFS uses Virtual Storage Disk (VSD) servers that do not physically hold the data, allowing an application to still be able to use that server as if it physically had the data. Hence data can be physically shared or can be virtually shared through the software simulations of the storage area network. By having more than one VSD server, data is virtually replicated, providing robustness to the file system in spite of node failure of the VSD servers.

Parallel System Support Program (PSSP) for AIX is used to provide a fault tolerance mechanism [16]. PSSP is a collection of software tools, for the IBM SP cluster that provide good scalability and performance efficiency. Also it provides system recovery and problem management, making GPFS automatically recover in the event of a node failure.

GPFS is made highly scalable mainly due to the distributed locking feature [47]. Distributed locking provides cache consistency between nodes. Read throughput scales proportionally with the scaling of the nodes. The write throughput also scales well due to the token server mechanism. The token server issues tokens to processors that request them. The typical reason for a processor to request for a token is to update a resource, typically a shared file, and the update is possible only after it receives the token. Since there is only one token, updates are synchronized and consistency is maintained. Using heuristics the communication between the process and the token server is minimized. In the event of failure of the token server, another node becomes the token server and by issuing messages to every node, it is able to obtain the information about which processor currently holds the token and can begin servicing from that point on. Through the token server mechanism, the throughput for I/O is high irrespective of whether many processors are accessing many files simultaneously or many processors are accessing one file at the same time.

4.1.2.2 Scheduler

The `psub` command of the Distributed Production Control System (DPCS) is used to submit a DPCS job script. The parameters required by DPCS to run a batch job are provided through the script. Typical parameters include the executable, the number of nodes, number

of tasks per node, Estimated Time of Completion (ETC) for the job, standard error and output files. When the requested resources are available, the DPCS will pass the queued job to the Load Leveler queue [1]. Load Leveler is a distributed network-wide job management facility to dynamically schedule and manage jobs for IBM SPs [14]. The scheduler is meant for all kinds of workloads, parallel or serial, and to efficiently use all the resources available, irrespective of the workload distribution and arrival rate. The scheduler selects the processor(s) from the available pool according to the requirements of the clients, and submits and starts the execution of the job on the chosen processor(s).

4.1.2.3 Supercomputer Usage

ASCI Blue is used to solve a variety of scientific calculations by using parallel applications. Some of these applications are sPPM to solve compressible turbulence problems [19], MPQC to search the existence of polymeric forms of nitrogen [44], JEEP [30], IMPACT, a coupled atmospheric modeling simulation [46] and Ardra to simulate the flux of fusion neutrons that comes out of the Nova laser target chamber [23]. All these applications are characterized as writes mostly, with an ability to restart from datasets of intermediate calculations.

4.1.3 Vivid Cluster

Vivid cluster is a 16-node Linux cluster each with two 800 MHz Pentium III processors. Each node has a local SCSI disk with a capacity of 18 GB and a 0.5 GB RAM memory. The connection between nodes is through a Gigabit Ethernet. This cluster is part of the visualization group at LLNL. The server is a 400 MHz Alpha that is connected to all the nodes by

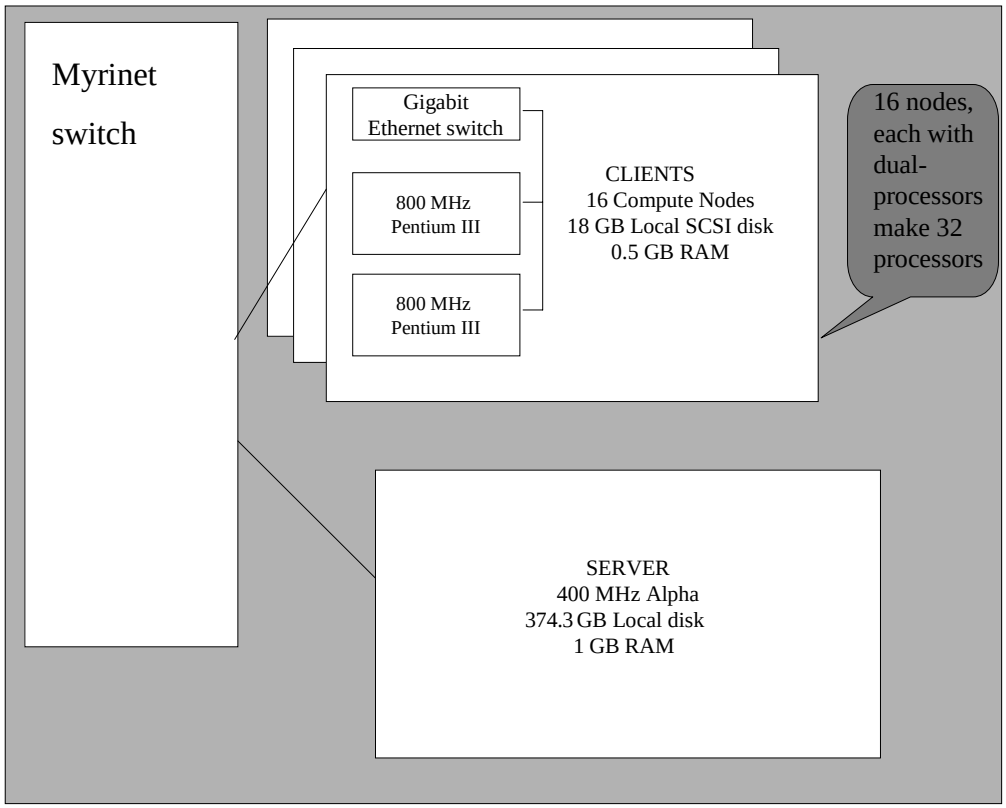


Figure 4.3: Vivid cluster topology.

Myrinet [8].

4.1.3.1 File System

There are two global file systems: a Network Appliance (NetApp) NFS server and a Parallel Virtual File System (PVFS). NetApp NFS server has a filer for performance improvements. NetApp filer storage appliances are the building blocks for scalable network storage [15]. NetApp filers are robust while providing high data access throughput and easy scalability. The Vivid cluster uses the F740 filer for the NFS file server, which can scale up to 1 TB. High throughput, availability, and reliability are provided by the built-in RAID system. A battery-backed NVRAM provides additional data protection. The server is scalable in terms of the number of users as well as the storage capacity of the file server.

The PVFS project [45] started as the popularity and usage of PC clusters increased. The goal of the project was to create a parallel file system for clusters. PVFS offers a global name space, striping data across multiple I/O nodes. Data is physically striped across disks, thereby decreasing the bottleneck of accessing from one disk, increasing data access throughput. By striping files over the local disks of the cluster nodes, one can obtain better performance than an NFS server [39, 24].

4.1.3.2 Scheduler

There is no explicit job scheduler in vivid cluster. Parallel jobs are implemented using MPI [10] on selected nodes. Users negotiate for nodes by emailing each other.

4.2 Application Instrumentation

The goal of our instrumentation was to study the I/O behavior of the two applications, similar to the characterization work done by Smirni *et al* [49]. The access pattern of the applications consists of small bytes of reads, interleaved with computation, followed by writes. Application reads are implemented using special-purpose library. The application-level read calls incorporate some of the alignment process, and are not strictly I/O. The writes were implemented as standard system calls with no special library.

The performance and access pattern of these applications can be studied using the Pablo [43] I/O instrumentation library, that supports data capture and analysis. The goal of the Pablo project was the development of a portable performance data analysis environment to be used by many massively parallel systems. The Pablo instrumentation library is a set of wrappers around standard I/O calls to collect trace data. Because the chosen applications had a very large number of small (each typically a few bytes) buffered I/O operations, the trace perturbation was very large. Hence we decided to trace the special purpose library operations, instead of tracing the individual character buffered I/O calls.

PsLayout and WABA are run in parallel as separate applications on different nodes, creating several individual trace files, one for each portion of the job. Tracing overhead was negligible. Self-Defining Data Format (SDDF) tools of Pablo library can be used to combine these files to achieve a global temporal ordering.

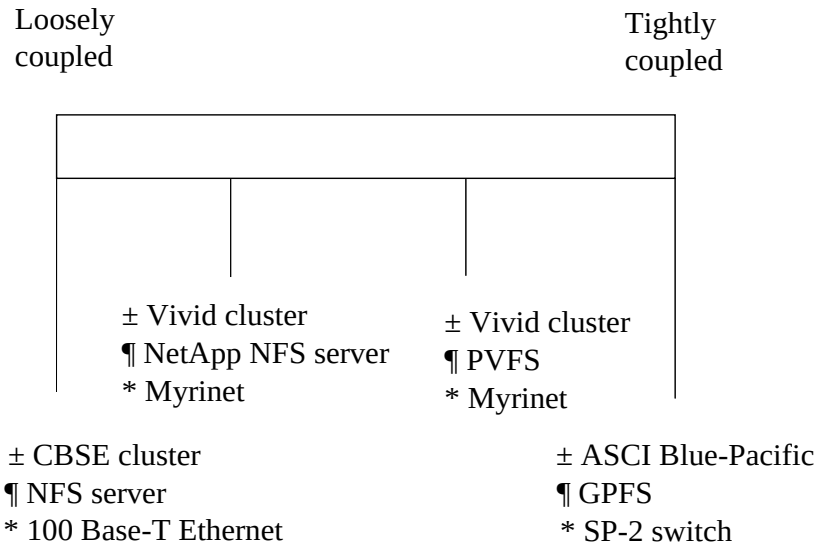


Figure 4.4: The spectrum map of the architectures(±) in terms of file system(¶) performance and interconnect speed(∗). Note that the terms loosely and tightly coupled indicates the parallel processing support of the architecture.

4.3 Experimental Setup

We characterized psLayout and WABA to understand their performance of different file systems. The goal of this experiment is to characterize their behavior when scaled with different file systems and input file locations. We examined performance on three different architectures and four different file systems for psLayout and one architecture and file system for WABA. The first architecture is a low-cost cluster (CBSE) with a reliable server (NFS) and a slow interconnect (100 Base-T Ethernet). The second architecture, ASCI Blue-Pacific, is at the opposite extreme to the first. This architecture has a highly scalable file system (GPFS) and a very fast interconnect (SP-2 switch). The third architecture, Vivid, falls somewhere between the two extreme cases, with two file systems and a fast interconnect (Myrinet). The first file system is robust (NetApp NFS), and the second has support for parallel applications (PVFS).

The three architectures provide a range of multiprocessor designs. Imagine these architectures (represented by ±) to be in a spectrum, as shown in Figure 4.4. Moving from left to right, the performance of the file system for parallel applications (indicated by ¶), increases. The interconnect speed (labeled by ∗), increases as well.

The two experiments with psLayout on the three architectures differ in the input file sizes and the number of processors. In the smaller data set experiment, the sequence file is a 4 MB chromosome sequence and the 411 MB sequence database is BAC Ends (also called bac ends) taken from Bacterial Artificial Chromosome (BAC). Both are read-only files. This experiment was scaled to 10 processors. The large data set experiment uses a 26 MB sequence file consisting of sequenced BAC data and the same 411 MB sequence database file. This data set was scaled to 50 processors. The WABA experiment implements a human to mouse genome alignment. The read-only input file sizes are 48MB and 22 KB. They represent a snippet of human and mouse genome respectively. This experiment was scaled to 10 processors on the CBSE cluster.

The pre-computation process for both applications involves the splitting of the input sequence file or genome1 (4 MB, 26 MB, 48 MB) to the right number of portions corresponding to the scaling of the number of processors. This is shown in Figure 3.3a or Figure 3.4a. For a one-processor run, there is no pre-computation step. For example for the psLayout alignment of a 4 MB file with the 411 MB file on four processors, the pre-computation stage would involve the split of the 4 MB file to four equal portions of approximately 1 MB each. Then each 1 MB file, or sequence, is aligned with the sequence database. The size of the split files are approximate because the file format used in the human genome project to represent the genes

(FASTA) can only be split in certain safe places called “markers” [28]. For a one-processor run for the same data set, the sequence is not split or split into one part.

We consider the performance impact of different file locations as we scale the application to more processors. The read-only input files may be located at the global store, or one on the global store and the other at local disk, or both on local disk, creating four combinations: global/global, local/global, global/local and local/local.

4.3.1 PsLayout with Small Data Set

We implemented the alignment between chromosome sequence (4 MB) and the bac ends taken from Bacterial Artificial Chromosome (BAC), which is 411 MB using psLayout. We characterized this alignment on all the three architectures with different input file locations and scaled to 10 processors.

For each architecture, the storage resources include the global store and the local disk. For the input file to be at the local disk, it must be copied there. Hence the tradeoff for replication to local disk is the fast access time verses copy cost. The output is written to the global store.

PsLayout is embarrassingly parallel, with no communication between nodes except through I/O, and should ideally scale very well. However, as shown by Figure 4.5a, it does not. Figure 4.5a shows speedup of psLayout on the CBSE cluster using the small input data sets, both located on the global store. With 10 processors, the speedup is approximately 1.65. This run took about 50 minutes to complete on a single CBSE processor.

In Figure 4.5b, we examine the breakdown of the execution time for psLayout. Here

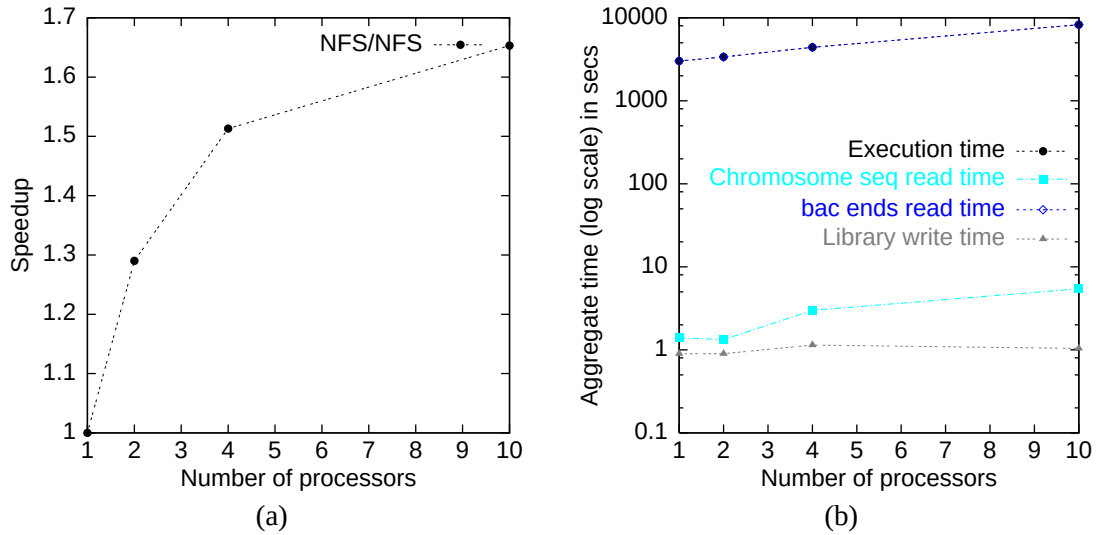


Figure 4.5: PsLayout run on CBSE cluster with 4 MB and 411 MB input files on NFS servers (a) Speedup (b) Aggregate time breakdown.

we compare the aggregate of individual execution times on separate nodes as we scale the number of processors. The difference between aggregate execution and sequence database (bac ends) read time is negligible, indicating that most of the alignment computation is interleaved with the character by character read of the sequence database. The write time represents a very small fraction of the total execution time (less than 0.05%).

As shown in Figure 4.6, some large application reads occur in the beginning of execution, followed by many small writes. This confirms our description of I/O activity given in §3.2. Some of the reads are exceptionally long; this variance is caused by the lengths of the sequences and the difficulty of alignment.

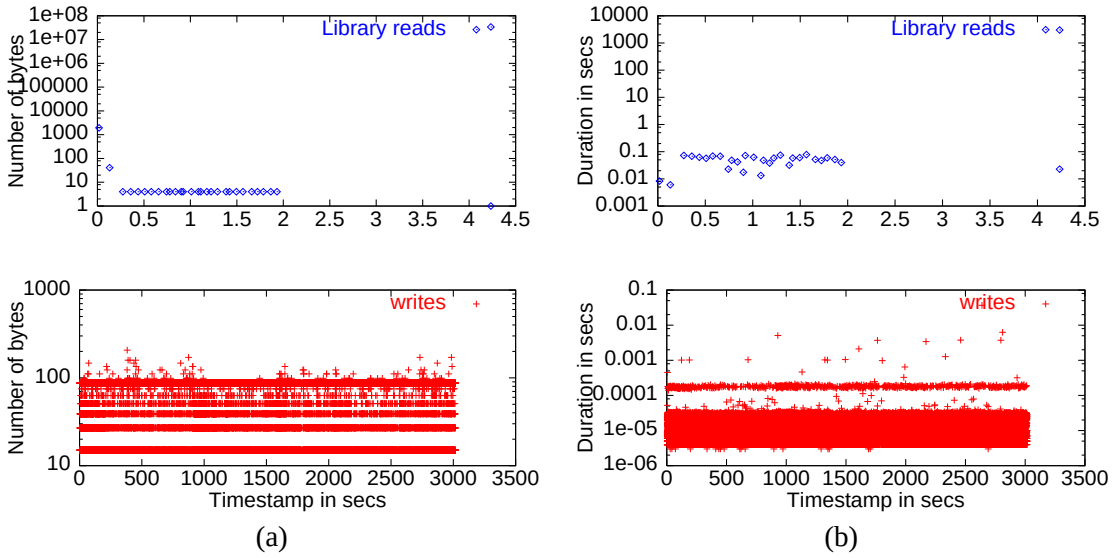


Figure 4.6: Timeline of psLayout on CBSE cluster for 4 MB and 411 MB NFS file inputs (a) Number of bytes accessed through library reads and writes (b) Duration of these library reads and writes. Note that the vertical axes are in logarithmic scale and the x-axis for all the figures is in seconds. There is a library read operation at the top right corner of the graphs, apart from the legend.

4.3.1.1 Input File Location

The major suspect in the poor scalability of the CBSE cluster shown in Figure 4.5a is file location. Files are shared on the CBSE cluster using NFS, but we know this performs poorly under concurrent requests, evident from Figures 4.6a and 4.6b. For convenience, databases are kept at the global store; however, for performance reasons, we could consider replicating either one or both of the inputs at the local node disks. Now we have the choice of using any one of the replicated copies.

We implement PsLayout using the same input files but consider the effect of replicating them from global to local store, to understand the impact of file location on performance. For any architecture, the combination of input files are global/global, local/global, global/local

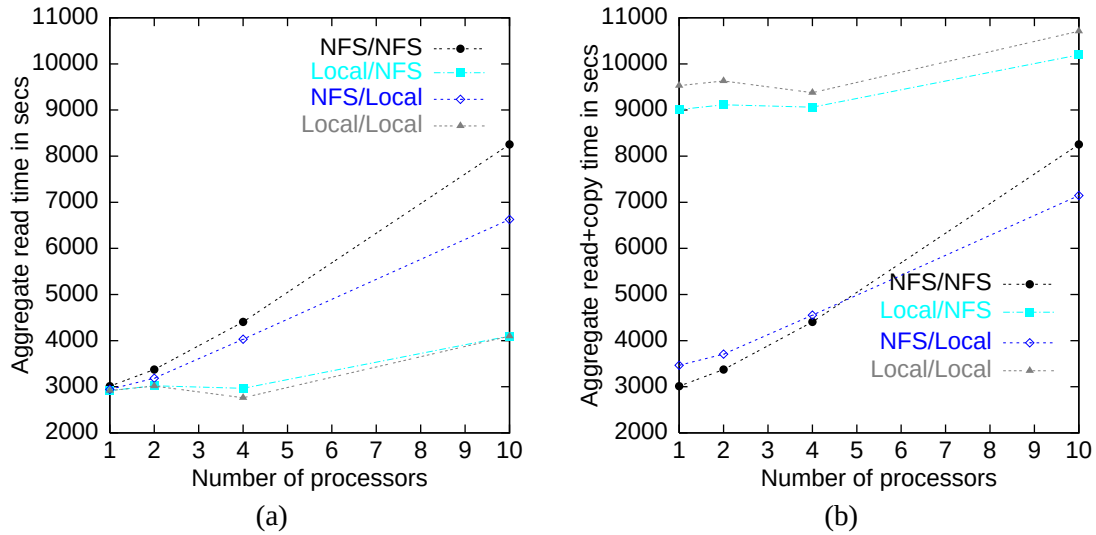


Figure 4.7: Comparison of read times with increased levels of parallelism of psLayout on CBSE cluster for 4 MB and 411 MB file inputs and different file locations (a) excluding copy time overhead (b) including copy time overhead.

and local/local. For the last three combinations, we incur a copy time overhead for copying one or more input file(s) to the local store. For the CBSE cluster, files must be copied to all nodes in the cluster because we do not know initially, to what node Condor is going to assign the job. We use an optimized binary tree copy program that understands the cluster topology. The copy time is significant for the CBSE cluster because the files are copied to all the nodes in the cluster through a slow interconnect. For Blue and Vivid, we need copy only to those processors doing the alignment. In these cases, the copy time is insignificant primarily because of their faster interconnects and also because files are copied to only the specific processors doing the alignment process. For example, for the 411 MB file, copy time to the 93 nodes on the CBSE cluster is 6093 secs, in contrast to 62 secs on Vivid cluster to copy it to one node, and 82 seconds to copy it to 30 nodes, a difference of two orders of magnitude.

Figure 4.7a and 4.7b show aggregate application read time for psLayout excluding and including copy overhead, respectively, for different file locations on the CBSE cluster when scaled to different numbers of processors. Depending on the precise parallelism of the input file, there are actually slight variances in the total time to perform the alignment; this is why local/local and NFS/local for four processors takes slightly less time than for two processors and slightly more for ten. The library read calls that includes reading and the alignment process, account for more than 99% of the total execution time. Because we compute the aggregate execution times, ideally the curves in Figure 4.7a and 4.7b should be close to horizontal. Instead we see in Figure 4.7a that the local/NFS (4 MB/411 MB) and NFS/NFS curves begin to increase significantly even at four processors. NFS file server performs poorly on concurrent operations. In this application, reads to the NFS file from all the nodes are concurrent. Hence NFS is as predicted a bottleneck. For a small degree of scaling, the cost of copying files can erode the performance improvement, but as the number of nodes increases, NFS becomes a very significant bottleneck, and the benefit of replication becomes clear.

Vivid represents a compromise between the CBSE cluster and Blue; it has a fast network and fast NFS server. Figures 4.8a and 4.8b show aggregate execution and copy time from Vivid.[1] The possible input file locations are PVFS, NetApp NFS and local disk. For the local disk copy, the file is copied from the NetApp NFS server to the local disk. With a very small copy overhead and efficient file server performance on concurrent access, results from Vivid show good scalability. The NFS/local and PVFS/PVFS combinations perform best when scaled to ten processors. The PVFS/PVFS input file combination performs slightly better than

[1] Data from the local/local four processor run is unavailable.

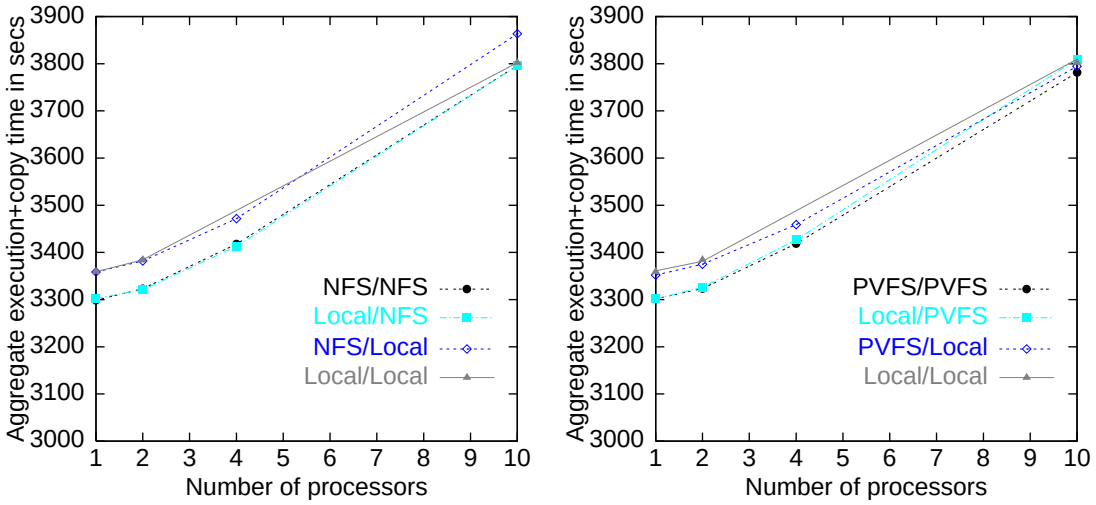


Figure 4.8: Comparison of execution times with increased levels of parallelism of psLayout on Vivid cluster for 4 MB and 411 MB file inputs and different file locations using (a) NFS server and Local disk (b) PVFS and Local disk.

the NFS/local one for 10 processors.

ASCI Blue provides a robust architecture with fast network infrastructure and a file system highly tuned for parallelism. Figure 4.9 shows the four input file combinations for ASCI Blue, with GPFS and local disk as possible file locations. All the four combinations perform similarly, and are comparable to the CBSE cluster without considering copy overhead. For the local copy, the file is copied from the NetApp NFS server to only those processors implementing the alignment.

4.3.1.2 Load Balancing

Even when all I/O is local, psLayout does not scale very well. We can see from Figure 4.10 that this is because of problems with load balancing. There is one node among the 10 processors whose alignment takes at least more than double the average time. The input to

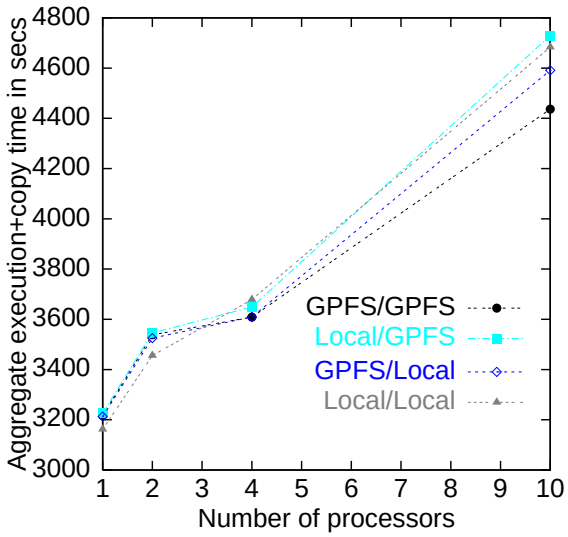


Figure 4.9: Comparison of execution times with increased levels of parallelism of psLayout on ASCI Blue-Pacific for 4 MB and 411 MB file inputs and different file locations.

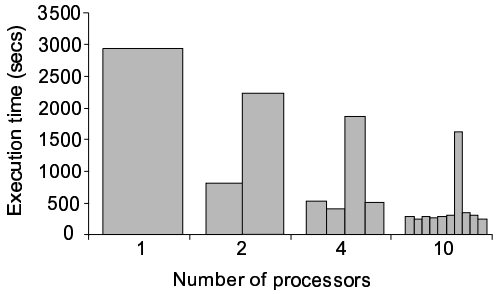


Figure 4.10: PsLayout workload distribution for 4 MB and 411 MB local file inputs on the CBSE cluster.

Architecture	Optimal input file combination
CBSE cluster	NFS / local
ASCI Blue-Pacific	GPFS / GPFS
Vivid cluster	local / NetApp NFS
Vivid cluster	PVFS / PVFS

Table 4.1: Summary of optimal file location for psLayout run with a small data set.

this node has a lot of repeats in its sequence, causing delay in the alignment, because several sequences match for each alignment. In practice, scientists at UCSC manually balance the cluster load by timing the submission of their jobs.

4.3.1.3 Summary

PsLayout, a typical computational biology application, has characteristics very different from many scientific applications. It is embarrassingly parallel, with all communication through the shared global store. We instrumented the application-level I/O calls using the Pablo performance environment, which supports user-level performance data capture and analysis. PsLayout is structured so that the alignment computation is inextricably interleaved with the I/O. Most of the application time (approximately 98-99%) is spent in user-level I/O libraries doing buffered reads, memory allocation, and string comparisons. The write time is insignificant by comparison.

We characterized a small data set experimental run of psLayout on three architectures. Because each node running the program needs to access the two input files, a scalable shared store (either a parallel file system or network file system) is necessary. Scalability of

this global store is crucial to performance. Even with a scalable file system, due to the nature of the algorithm, load balancing is a problem and ideal speedup is not attained.

Another factor affecting performance and scalability of this application is the input file location. We calculate the optimal file location combination as the combination that takes the minimum aggregate execution time. Table 4.1 shows the optimal file location combination for psLayout when scaled to 10 processors.

4.3.2 WABA

WABA was implemented to find the alignment between the human and the mouse genome. By knowing similarity between the two genes, we will be able to predict what a particular sequence in the human genome might correspond to, based on the known sequences of the mouse genome. WABA was characterized on the CBSE cluster. Since it is a production system and also because WABA is a compute-intensive algorithm, we used a small problem set. The input size used was 48 MB and 22 KB, representing a snippet of the human and mouse genome, respectively. The four possible input file combinations are global/global, local/global, global/local and local/local. The output was written to the NFS server. The same experiment was not tested on the other two architectures because it is not I/O-intensive.

WABA, like psLayout has no inter-process communication except through the I/O. The experiment when both the files are at NFS server does not show ideal scaling as seen in Figure 4.11a. The execution speedup obtained is approximately 1.5 on scaling to 10 processors.

The breakdown of the aggregate execution times is shown in Figure 4.11b. The

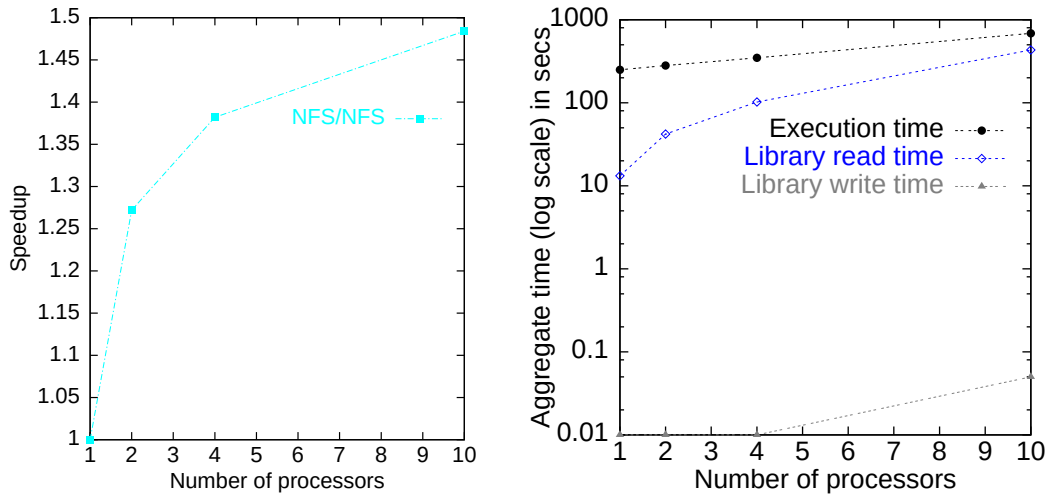


Figure 4.11: WABA with 48 MB and 22 K at NFS server (a) Execution and I/O speedup,where I/O speedup is the upper bound, assuming perfect parallelism (b) Aggregate times breakdown

library reads is the application-level reads that are interleaved with the alignment process. The library read time for one processor run are insignificant, approximately 5%. But the library read times are significant when more processors are used, up to 75% of execution time for ten processors. Each pass computes and writes standard outputs, irrespective of the input size. The way the passes reads the date is dependent on the heuristics of the pass. With ten processors, second and third pass read overhead increases by more than a linear factor. Hence the reason for the increasing read percentage as we scale is because of the additional overheads of second and third pass reads in the algorithm.

The increase in write time represents a much smaller percentage by comparison. The write overhead increases for the same reason, but because writes represent a very small portion of the execution time, the increase is not significant.

The timeline graphs of WABA when both the files are at the NFS server on one pro-

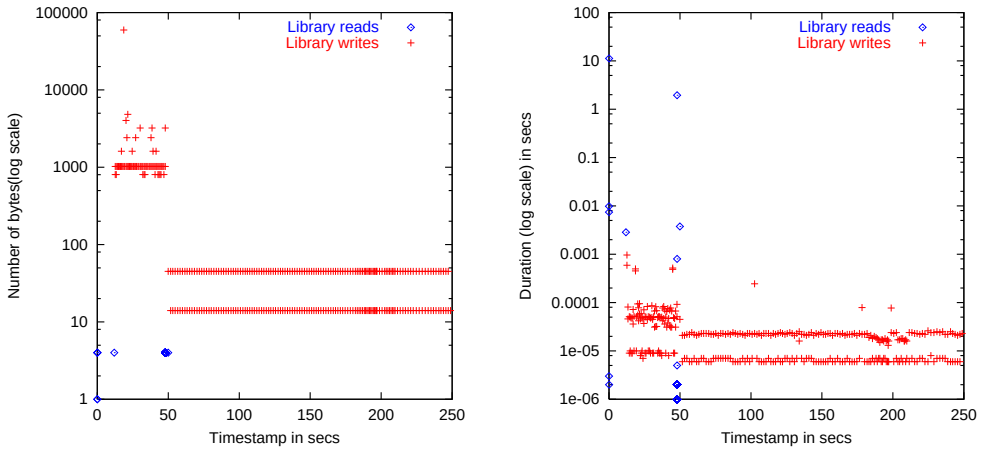


Figure 4.12: Timeline of WABA for 48 MB and 22 KB NFS file inputs at NFS server. (a) Bytes accessed (b) Operation durations.

cessor are shown in Figure 4.12. We can see the initial I/O operation are reads, corresponding to the reads in the first pass. Then the high-error margin alignments are written to a temporary file. This is followed by reads, corresponding to the second pass reads, and the third pass writes. This confirms our description of I/O activity in §3.3. The number of bytes written is higher initially due to the high error margin of the first pass. Subsequently by the use of more stringent heuristics in the second pass, a lower error margin is observed and only the most accurate data is written, decreasing the bytes written.

4.3.2.1 Input File Location

As in the previous experiment, the input file copies can be located at either the NFS server or the local disk. Local disk copies must be made to all 93 nodes. We saw from the small data set experiment with psLayout (99% I/O-intense) that the file location affects scalability (§4.3.1). The goal of this experiment is to test if that conclusion is true of a relatively less

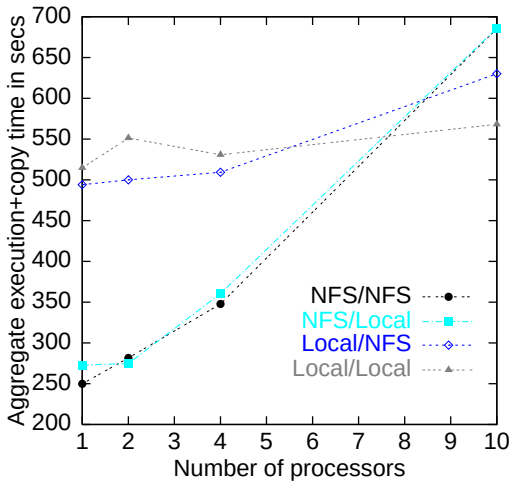


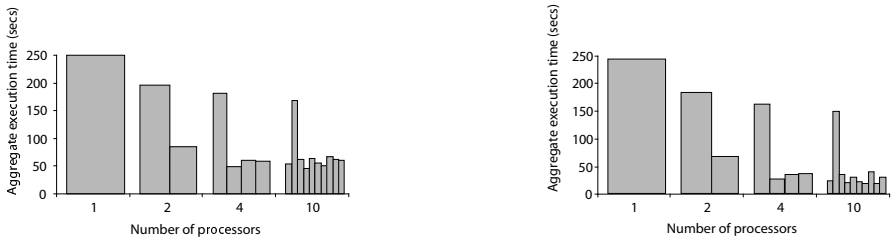
Figure 4.13: Comparison of execution times with increased levels of parallelism of WABA on CBSE cluster for 48 MB and 22 KB file inputs and different input file locations.

I/O-intense (4% - 75%) application.

Figure 4.13 shows the different execution times including the copy time for the four input file combinations. The NFS server provides good performance when scaled to a small number of processors. For Local/NFS and Local/Local combinations, the copy time incurred exceeds the execution time of one processor run. As the number of processors increases, the NFS access time increases and the best combination for 10 processors is local/local.

4.3.2.2 Load Balancing

The load balancing graph in Figure 4.14, clearly shows the contention at the NFS server and why local disk access is preferred. Figure 4.14a shows the execution times when both the files are at NFS server. We see that there is one job for every run that takes a long time to complete. This is due to input having a lot of repeats and this algorithm takes more time on handling more repeats. Figure 4.14b shows the execution times when the larger file is at local



(a) (b)

Figure 4.14: Workload distribution of WABA for 48 MB and 22 KB file inputs and file locations at (a) both NFS server (b) Local disk and NFS server respectively.

Number of processors scaled	Optimal input file combination
1	NFS/NFS
2	NFS/Local
4	NFS/NFS
10	Local/Local

Table 4.2: Summary of optimal file location of WABA for different numbers of processors

disk and the smaller file is at NFS server. There is the same one node that takes a lot longer than other nodes in each run, causing load imbalance. The execution time is less when the local disk copy is accessed (Figure 4.14b) on comparison to NFS access (Figure 4.14a). This justifies the local disk copy to be accessed for the 48 MB file. The speedup is not significant because of the lone job that takes up a long time.

4.3.2.3 Summary

WABA, a genome alignment application, uses complex heuristics to align genome sequences. This application is compute-intense and the percentage of execution spent on I/O increases when scaled to large numbers of processors. We characterized the performance of

WABA on the CBSE cluster with varying input file location and number of processors. The performance of WABA when run with a small problem size varies dramatically depending on the file location and the number of processors scaled. As an example, Table 4.2 shows the optimal input file combination, assuming the same definition as before for optimal.

From the two characterization efforts on psLayout and WABA, we see that irrespective of the I/O intensity of the application, the performance of the computational biology applications run on the CBSE cluster depends on the file location and the number of processors scaled.

4.3.3 PsLayout with Large Data Set

PsLayout was implemented to align the sequenced BAC file (26 MB) and the bac ends taken from Bacterial Artificial Chromosome (BAC), which is 411 MB. This run was relatively large. For larger experiments with larger numbers of processors, NFS global storage is a bottleneck and it is obvious that there is a crossover point where the additional copy overhead is insignificant compared to the overhead caused by the NFS bottleneck. Because the CBSE cluster is a production cluster, we could not create a bottleneck at the NFS server with 50 processors reading the 411 MB file. Hence we ran the experiment with the 411 MB file at the local disk and 26 MB file at the NFS server. The file location selection imitates how the experiments are run in the cluster. When there are two read-only input files, the smaller one is accessed through the NFS server and the larger file from the local disk. The Vivid cluster has 33 nodes, hence this run was scaled up to 30 processors on Vivid. The same input file combination as CBSE is run on Vivid to make a comparison. All the four input file

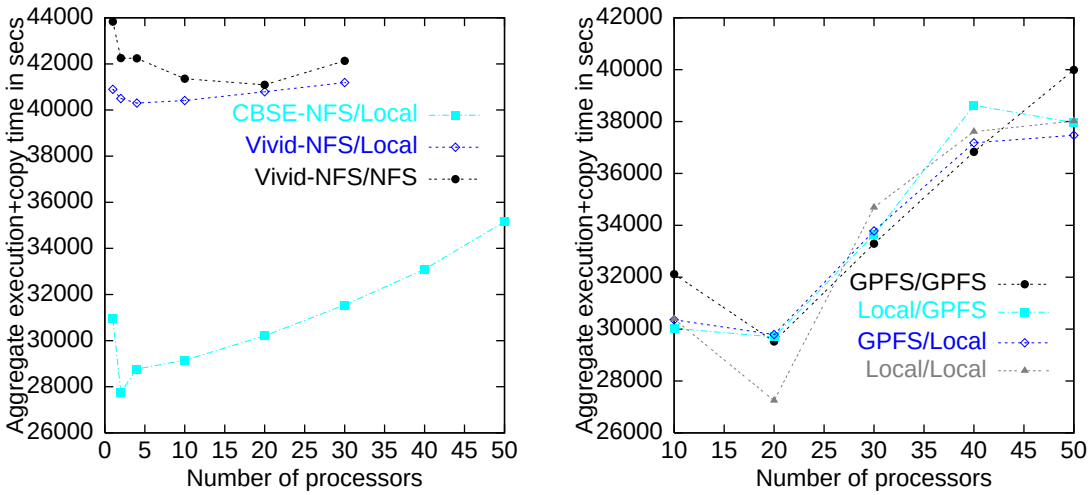


Figure 4.15: Aggregate execution time of psLayout using 26 MB and 411 MB inputs (a) NFS and local file input on CBSE and Vivid clusters (b) Comparison of execution times for different file locations on ASCI Blue-Pacific.

combinations were run on ASCI Blue-Pacific.

Figure 4.15a shows the scalability of the 26 MB and the 411 MB input file run. For this larger run, we examined only the NFS/local combination on the CBSE cluster and the NFS/local and NFS/NFS combinations on the Vivid cluster, with the 26 MB file at the NFS server and the 411 MB file at the local disk. The aggregate execution time is relatively constant as the number of processors increases, indicating good scalability. The NFS/NFS combination on Vivid is slower than the NFS/local, although not significantly. The CBSE cluster has better performance than Vivid because of the faster processors on the CBSE cluster.

We examine the scalability of this application on Blue in Figure 4.15b for different combinations of local and global store. All combinations scale well for 50 processors. For different numbers of processors, different combinations are slightly better or worse. Hence

this supports our claim that file location and number of processors for a given architecture and file system are important factors affecting performance of this application.

4.4 User-Level Library

From the characterization efforts of psLayout and WABA, we see that performance of computational biology applications is dependent on many factors and one of them is file location. This conclusions is relevant to most of the other computational biology applications at UCSC, which have similar characteristics to either psLayout or WABA. Currently, the optimal file location choice is made manually by the scientists at UCSC. Our characterizations showed the choice of optimal file location to be important and non-static.

Chapter 5

Design and Implementation of dSOARS

From Chapter 4 we see that I/O scalability problems are evident with even a small degree of parallelism. Programmers at UCSC alleviate bottlenecks by manually replicating databases to improve locality, and this approach works. Given the low cost of storage, replicating databases is a reasonable solution for improving performance, but managing these replicas is difficult and time consuming. As explained in Chapter 1, storage cost trends and genomic data trends are such that indiscriminate replicas are not a cost-effective solution.

To address this problem, we are developing a user-level I/O library for a new model of location-transparent storage. This library maintains read-only replicas of records and information about access times. Therefore, a read access to a record may be redirected to the most appropriate location. Unlike a traditional cache, where there is a strict hierarchy of access

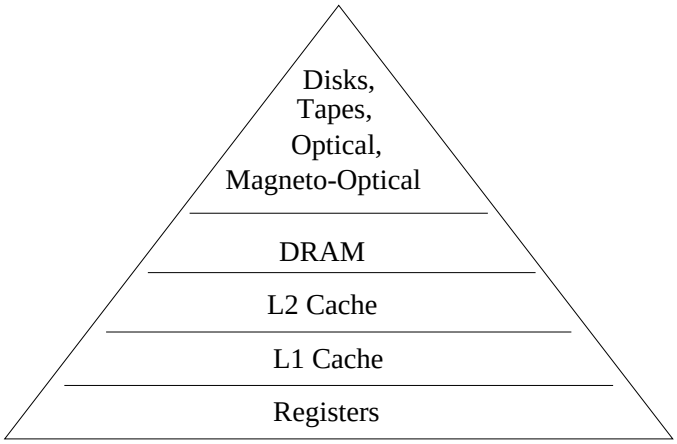


Figure 5.1: Memory Hierarchy.

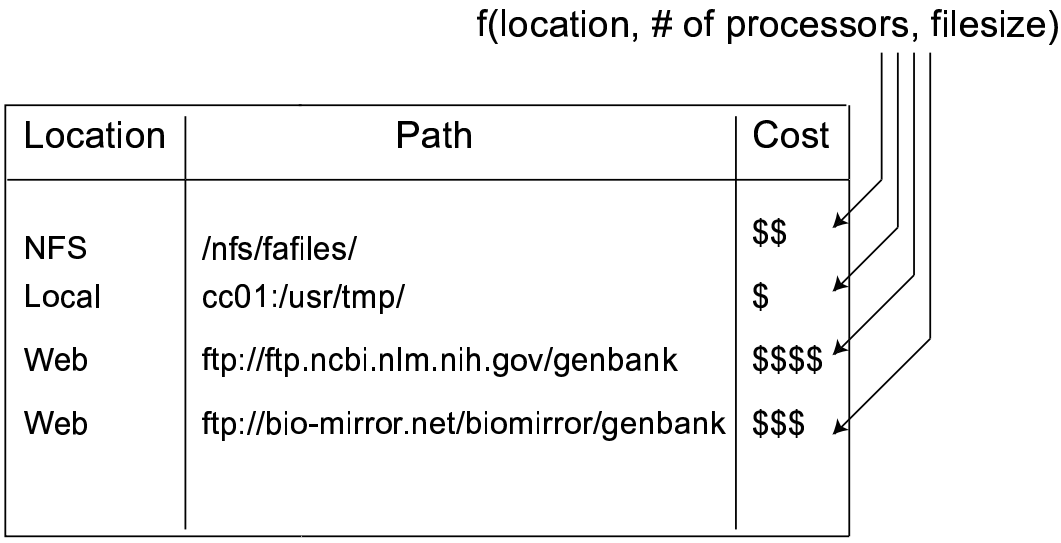


Figure 5.2: Example cache table entry.

times (that usually differ by an order of magnitude or more) as shown in Figure 5.1, access times to local disk or network storage change based on load and network conditions and may not retain a strict ordering.

Figure 5.2 shows an example cache table entry for a genomic data file. Here, the cost for accessing data at each location is calculated as a simple function of the number of processors, the file location and the file size. Although these parameters are fixed at the start of application execution, the cost function may be based on parameters that vary continuously. For example, as network links break or bandwidth is limited, it will be more expensive to access a file on the Web, and this can be reflected with this model. Ultimately, we envision linking replication with a dynamic run-time performance model that can provide performance data of the execution environment on-the-fly to calculate access costs.

The remainder of this chapter is organized as follows. In §5.1 we give the overview of the design issues and we discuss the different components of our design in §5.2. In §5.3 we discuss the library component interactions with the client program. In §5.4 we describe how the cost function is computed. We describe our implementation decisions in §5.5. Finally we give an example to illustrate library operation in §5.6.

5.1 dSOARS Design

In distributed computing environments, data placement and access are two of the major factors affecting performance. Replication can effectively provide access time improvements. Dynamic Storage Replicator and Selector (dSOARS) provides a layer of abstraction

to the file locations accessed by the client user, along with dynamic performance calculations, while managing the replicas. Only the file opens of read-only files are considered, because in the computational biology application area, read-only input files are often used [42]. Moreover for files that are opened for writing, consistency becomes an issue that we have not yet considered.

The first design consideration is to consider granularity of access at the file level. The other possible choice is record-level access. We did not implement record-level access because of the overhead of intercepting each record call. With the exponential increase of data for computational biology applications (Figure 1.1a), the number of intercepting calls would grow exponentially.

The next important design issue is the choice of where the layer of abstraction that determines the optimal file location should be placed. The choices are to intercept all the file open calls, to modify the kernel, or to require the application to make an explicit library call. In typical computational biology applications, small, temporary files are opened and used, as we saw in the case of WABA in §3.3. Optimizing location for files with very small lifetimes may not be beneficial. Moreover we are only focusing on read-only files. Hence optimizing all file opens may not be the ideal design choice. To use an explicit library call, we need to include the library header file in the client program. We call the optimization function as a dSOARS wrapper call and this design seems to fit our requirements. This particular design was chosen to provide flexibility, where some files can be opened with optimization and some are opened without.

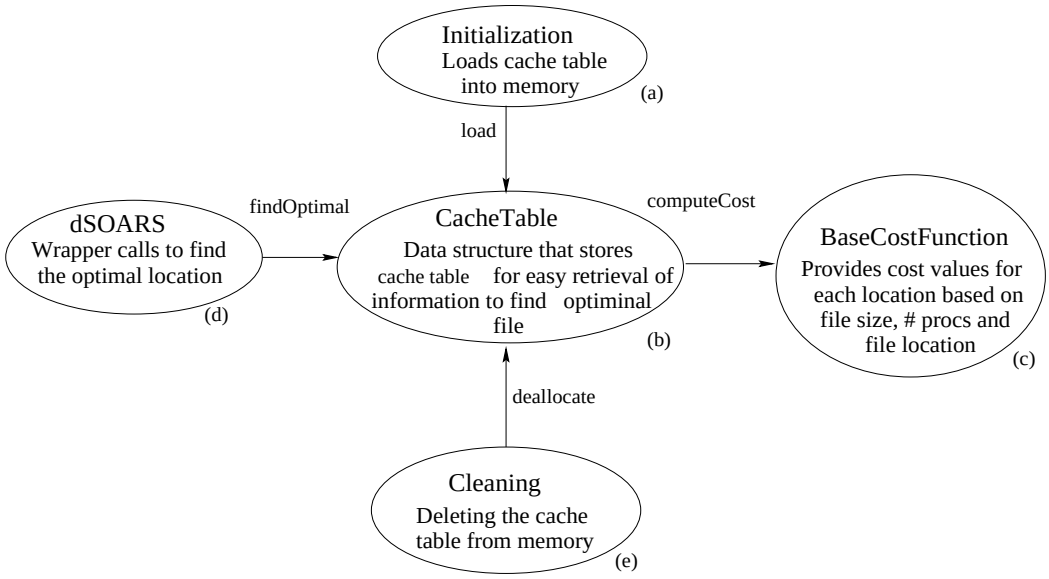


Figure 5.3: dSOARS components.

5.2 User-Level Library Overview

The library determines the file location that provides best performance, making replicated copies as necessary. To design such a library, we make a few assumptions. The assumption previously discussed in §5.1 is that we consider only read-only files. We consider all the input files to be at a default global location. The library determines if the global file access gives good performance; if not, it creates a local replica. This decision is done using a cost function, which is computed from a benchmark result. As part of future work, we want to track replicated copies and feed this information to the job scheduler. When the file needs to be accessed in another task, the job can be assigned to the node with the replicated local disk copy.

Our first pass design has five main components, as illustrated in Figure 5.3. We discuss the design choices of each component in detail in the sections §5.2.1 to §5.2.5.

5.2.1 Initialization

The major task that the user library should provide is to choose the optimal location. First, the cache table, which holds the cost function parameter values associated with each file, is loaded in memory. The design decisions were to either load the cache table during the initialization phase or with every dSOARS file optimization call. For the former choice, the advantage is we only need to load and deallocate the cache table once. The downside of this choice is that if any changes made to the cache table during execution are not used. Moreover if concurrently running application creates replicas of files, the additional replica information is not reflected during the current program execution. The latter choice incurs a lot of overhead that is associated with loading and deallocating the cache table from memory for each dSOARS file optimization. In this design, since we do not consider dynamic network parameters, the changes to the cache table are infrequent. We choose to ignore the current replication information and each program execution will work with the data from the previous program execution. Hence we choose to load the cache table in memory during the initialization as shown in Figure 5.3a.

5.2.2 Cache Table

The cache table is the data structure that manages the replicas along with the cost for accessing them. Hence the cache table should contain the file name, the location, and the cost to access it. The elements representing a unique key are the file name, the cache level (location), the file path and the operation (e.g. read). The operation element is included for easy extensibility of this library to other operations. For example, we have not considered write

operation and the consistency issues, and in this design, we can easily modify to incorporate writes.

The value elements associated with the key are the file size and the cost value for different number of processors for the particular operation, obtained from the benchmark result. The reason for having file size is because file size is one of the parameters of the cost function. The other option would have been to make a 'stat' system call for each file optimization. The file permissions are governed by the underlying file system.

The cache table data structure gets loaded into memory in the initialization phase, and when the cost function value of a file is needed, a search function is implemented as shown in Figure 5.3b. This function returns the cost function value for that instance.

5.2.3 Cost Function

The cost function is the factor that determines the selection of the file location. The cost function is a quantitative value, which is computed for each cache level of the file as shown in Figure 5.2. The cost is computed as a function of the file size, number of processors, and the file location (Figure 5.3c). We assume the cache level associated with the minimum cost value corresponds to the optimal location. The cost function is not a simple equation and the parameters we consider are for our first-pass implementation and as future work, we want to add more dynamic parameters to the cost function.

5.2.4 dSOARS

The dSOARS class provides location-transparency to the client program. dSOARS provides a wrapper call, `findOptimal`, that takes the base name of a file as input to determine the optimal location. For optimized file opens, before the file opens calls are made in the client program, the optimal file location should be found. The right location is determined by the cache table component, based on the cost component. This location is returned to the client program as shown in Figure 5.3d.

5.2.5 Cleaning

When the application has completed its task, before it completes execution, the cache table object in memory should be deallocated (Figure 5.3e).

5.3 Interaction with the Client Program

The interaction between the client program and the user level library components is illustrated in Figure 5.4. The interactions are described in two phases: initialization and the dSOARS file location optimization. Part I in Figure 5.4 shows the initialization phase. When the client program initializes dSOARS, the cache table is loaded in memory. Initialization is done once at the start of the application, before any library file optimization calls. Part II in Figure 5.4 shows the optimization phase. For optimized file opens, the client program calls the `findOptimal` method of the dSOARS object with the base name of the file to find the right location. This invokes the `findOptimal` method of CacheTable. The algorithm that

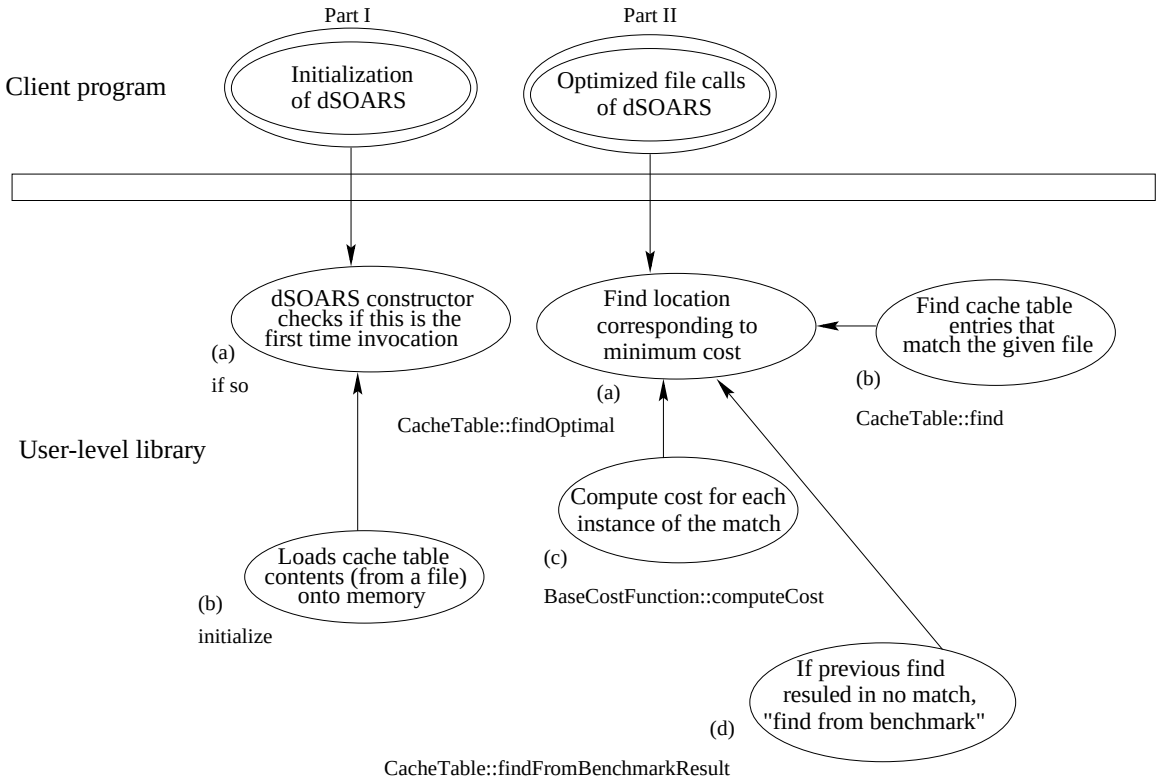


Figure 5.4: dSOARS component interaction to the client program.

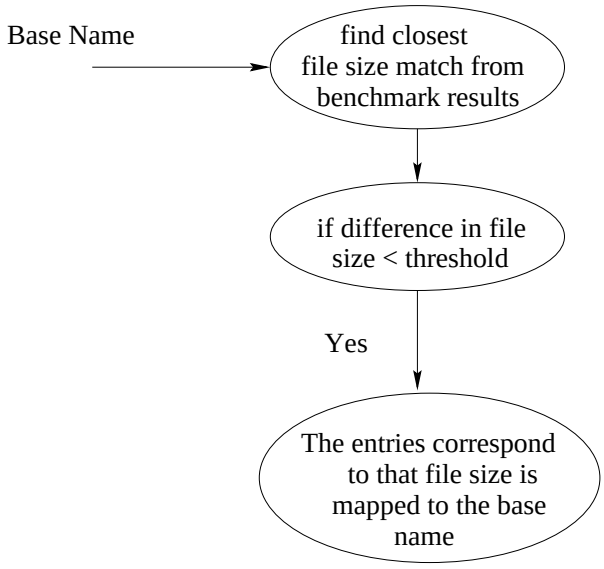


Figure 5.5: Mapping of the files from the benchmark results.

the user-level library uses to identify the optimal location is as follows:

- Find all entries in the cache table that match the specified base name (Figure 5.4 PartIIb).
- If no matching entry is found from the cache table, find a match from the benchmark result (Figure 5.4 PartIId). This decision procedure is shown in Figure 5.5 and is the `findFromBenchmarkResult` method. We assume that all the files we are referring to are at a global location. The size of the base name file is found using a 'stat' system call. This file size is matched to the nearest file size from the benchmark result. When the difference falls within a threshold of 10KB, the access cost corresponding to the benchmark match is mapped to the base name.
- The access cost for the specified base name for different locations are obtained. The `computeCost` method computes the access cost value corresponding to the number of processors of that instance as the cost value for each location. This is shown in

Figure 5.4 PartIIa.

- Return location with minimum cost to the dSOARS file optimization call. The `findOptimal` method of dSOARS returns the optimal file location to the client program.

5.4 Computing the Access Cost Function

The user level library determines the best location to read a file, based on a cost function. To calculate this cost function, we used a microbenchmark. The goal of this benchmark is to provide a cost value associated with a read operation on a given architecture with respect to different parameter considerations. In our first pass, we choose the most important static parameters for the benchmark from our characterization results discussed in §4.3. The inputs to the microbenchmark are the cache level (location), file size, number of processors and the type of architecture. The microbenchmark works as described in Figure 5.6 and the algorithm used is as follows:

- First a file of the specified file size is created on the default NFS location (Figure 5.6a). Creation of the file at NFS, mimics the behavior of how the CBSE works currently, when they replicate the dataset from genbank to NFS.

- If the cache level is NFS, the file is present at the right location and if the cache level is Local, then it has to be first copied to the local disk.

- Every processor reads 8K blocks from the file at NFS or local file system, until the EOF is reached.

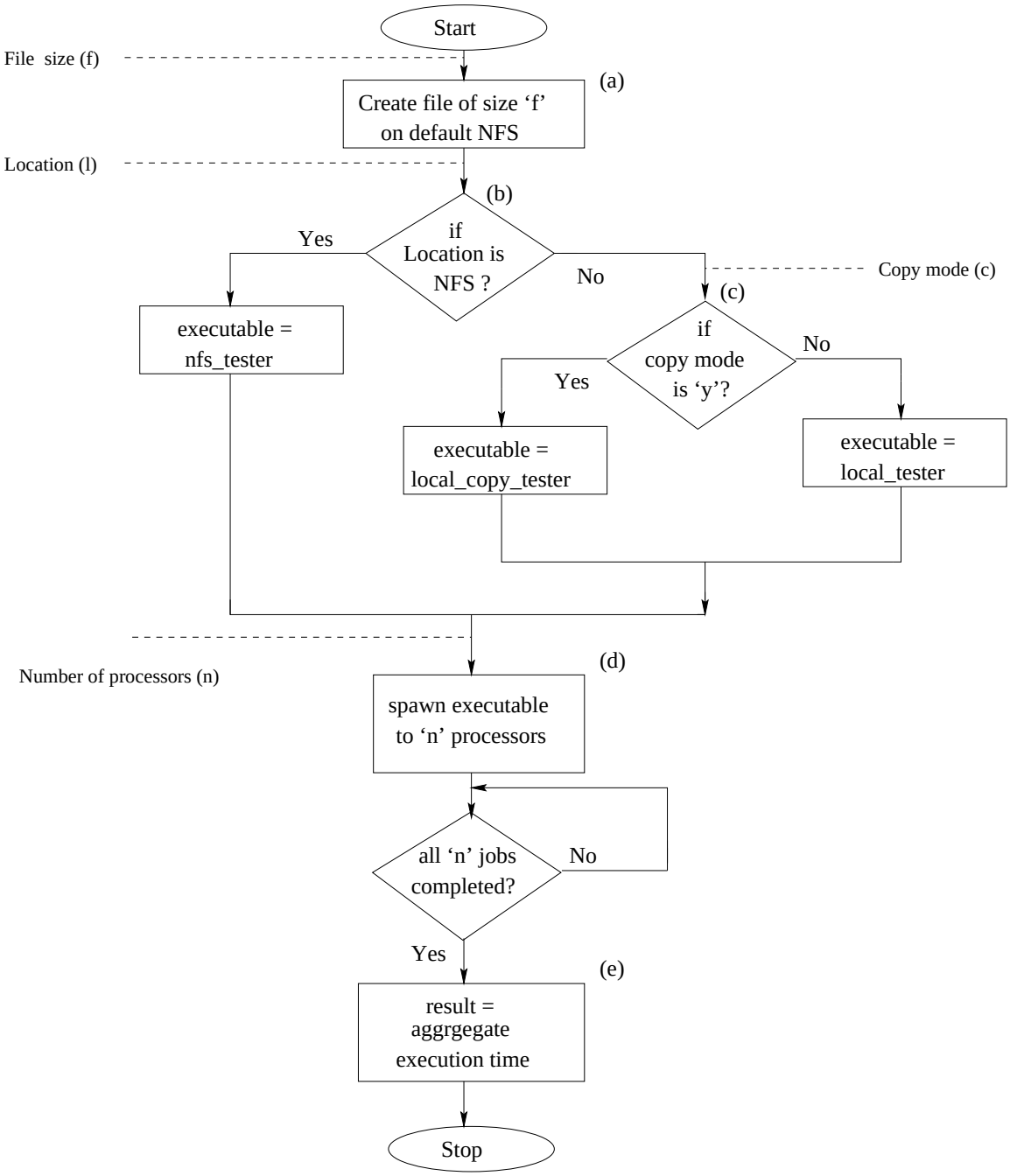


Figure 5.6: Calculation of benchmark results.

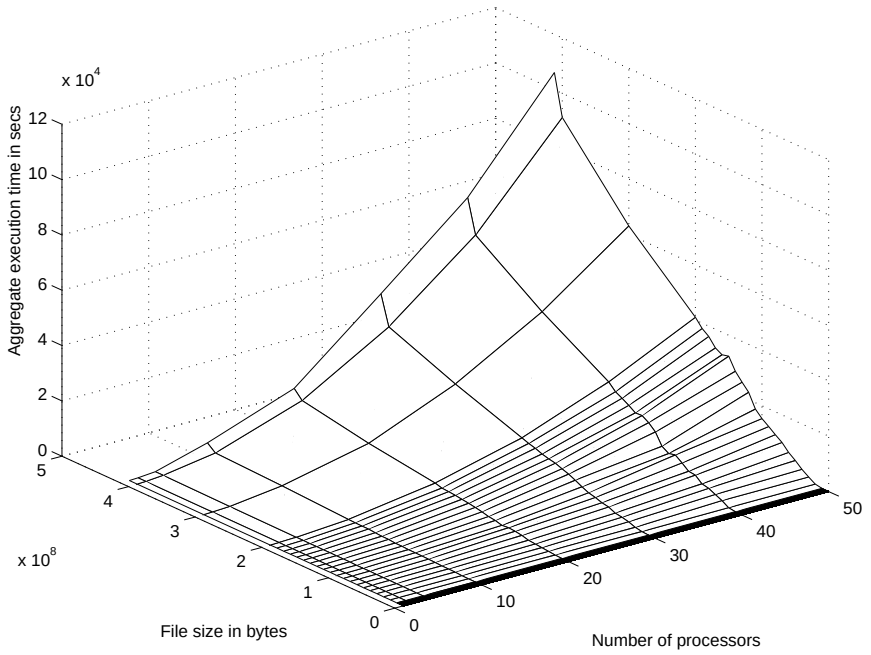


Figure 5.7: Benchmark time to read on CBSE cluster from NFS.

- The benchmark time is calculated by aggregating the read times at all the processors. If cache level is local, the benchmark time can be the aggregate read time of all processors, including or excluding copy overhead time.

The result of this benchmark on the CBSE cluster for different file sizes and number of processors for the cache level NFS is shown in Figure 5.7. All files are at NFS with no replication. This graph shows an increase in read time with the increase of the number of processors and the file size. The Local cache level benchmark results, excluding and including copy times are shown in Figures 5.8 and 5.9 respectively. The graphs shows increase in read time with the increasing file size and number of processors. We can also see a significant difference in the execution time between Figures 5.8 and 5.9. When copy time is included, the aggregate execution time (Figure 5.9), varies in proportion to the read time increase as well as

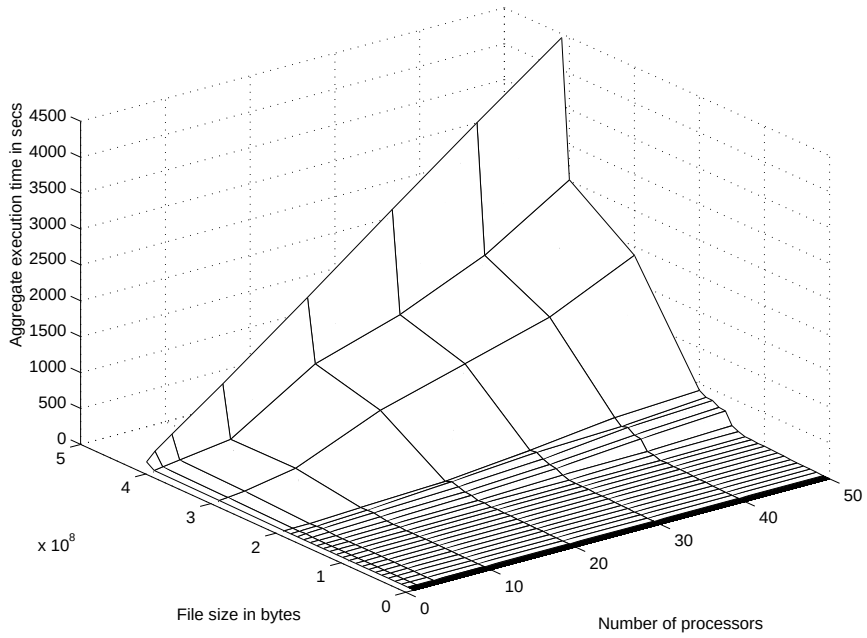


Figure 5.8: Benchmark time to read on CBSE cluster from Local disks, excluding copy overhead.

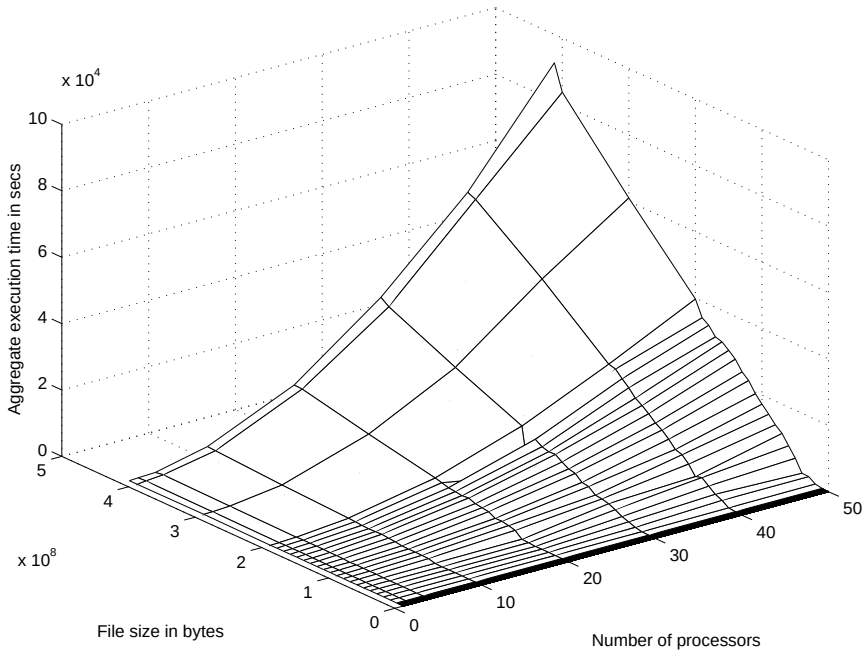


Figure 5.9: Benchmark time to read on CBSE cluster from Local disks, including copy overhead.

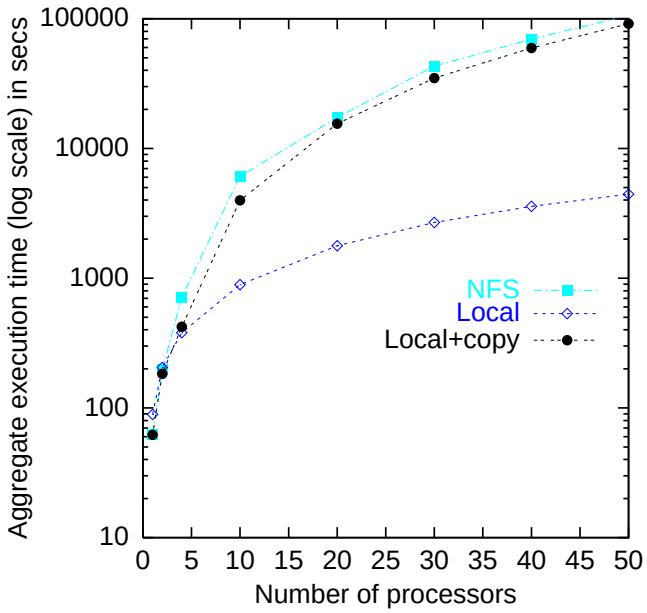


Figure 5.10: Benchmark time to read on the CBSE cluster from different locations for 400 MB file size.

the copy time increase.

To give a better perspective of the benchmark, we compare the times for a particular file size for different cache levels. Figures 5.10 show the benchmark times for a 400 MB file on different locations as we scale the number of processors. We see that local copy is not always optimal. For example, for one processor, NFS access is close to local disk access. If there is no local existing copy, NFS is best until 4 processors (Figures 5.10), after which it is more efficient to copy. Thus, the benchmark results are used to estimate read times based on file size, number of processors and file location. We are assuming the network bandwidth and latency remain constant. This is often not the case and as future work, we want to study dynamic network behavior.

5.5 dSOARS Implementation

dSOARS is implemented as a user-level library that the computational biology applications can link their applications with. dSOARS is implemented mainly using C++, with some parts like `findFromBenchmarkResult` using Perl. This software environment is available on a large platform of workstations and clusters. The main components of the library are C++ objects.

The client program needs to instantiate a dSOARS object on every process and the `findOptimal` wrapper call of dSOARS must be called for every optimized file open. In this implementation we need to explicitly specify the number of processors and give it as an argument to the dSOARS object. Explicit passing is needed because we cannot always get information about the number of processors from the job scheduler. If we had used MPI to spawn jobs, we can get the number of processor information during run-time. MPI library provides ease of use for message passing among applications. We did not use MPI because this class of applications has no inter-process communication and doing so would have required an overhead of linking the application with MPI libraries.

The user-level library design incorporates location-transparency and dynamic selection of the optimal file location. We discuss the components implementation choices in detail in the five sections from §5.5.1 to §5.5.5.

Field Name	Data Type	Examples
baseName	string	“bacEnds.fa”, “mouse_genome”,“human_genome”
cacheLevel	enum	NFS, cc01, cc12, Local+Copy
filePath	string	“/nfsserver/file/path/”, “/var/tmp/”
oper	enum	read, write, seek

Table 5.1: Cache Table Key elements and their data types with examples.

Field Name	Data type	Examples
fileSize	int	411751352
costFunctionValue	cost[]	{10,20,30,40,50}

Table 5.2: Cache Table Value elements and their data types with examples.

5.5.1 Initialization

The cache table content is loaded into memory from a pre-defined file. This file is at NFS server, a central location, accessible by all the nodes in the cluster. The cache table data structure is discussed in the next subsection §5.5.2.

5.5.2 Cache Table

The Cache Table is implemented using the Standard Template Library (STL) data structure, map. Map is an associated array represented by a (key,value) pair. Each key is a unique entity and can be mapped to one value.

The elements representing a unique key are the file name, the cache level, the file path and the operation. The data types of these elements and examples are shown in the Table 5.1. The value elements associated with the key are the file size and the access cost for

different number of processors. The value elements and their respective types with examples are shown in Table 5.2. In this implementation the cost data type is an integer.

The operations performed on this data structure are load, during the initialization phase and search, to find the entries corresponding to the base name during optimization phase. Given *n* entries in the table and *m* optimized file open calls, insert is $O(n)$ and search is $O(mn)$. In this implementation, we insert into the cache table only once during initialization of dSOARS and we search for each optimization call. Hence search is the operation with the highest frequency of occurrence. We chose the map data structure because retrieve is $O(log(mn))$.

The cache table object has the `findOptimal` member function that uses the retrieve or search function of map to identify the optimal file location. This function needs the cost function component to provide the cost value as discussed in §5.5.3.

5.5.3 Cost Function

The cost function is not a simple equation and it can have many dynamic parameters associated with it. In this implementation, the cost function is mapped from the benchmark result. This mapping, implemented using Perl, is written in a plain-text file in a central location, accessible from all the nodes in the cluster. To make the library extensible, we implemented the cost as a base virtual class with derived classes implementing the `computeCost` virtual method that over-rides the base class virtual function. Future cost functions can incorporate other parameters.

5.5.4 dSOARS

The client program interacts with the user-level library to provide location transparency. dSOARS class provides the interaction functionality. The `findOptimal` method of the dSOARS class determines the optimized file location by using the cache table and cost function classes. Before this method execution, the cache table has to be loaded in memory, which by default happens when initializing the dSOARS class object.

The client program creates the dSOARS class object and calls the `findOptimal` method to find the optimal location before any optimized file open calls in the client program. The library computes the best location and returns that as a string object to the client program. The client program can easily convert the string to a char* object and use it in the standard UNIX file open system call.

5.5.5 Cleaning

Since allocated objects are automatically deleted at the end of the execution, cleaning operation is implemented implicitly.

5.6 Implementation Example

The implementation is done such a way that the inclusion of the library into the client program can be done with minimum changes to the source code. The changes made are shown in Figure 5.11. First the client program needs to create an object of dSOARS, specifying the number of processors. The declaration of the dSOARS object initializes and

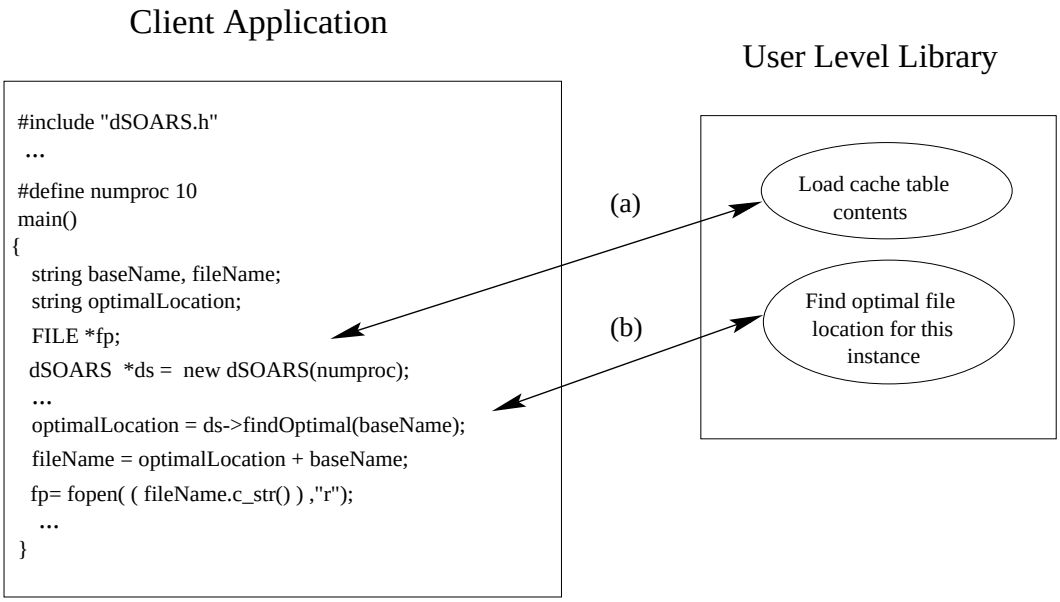


Figure 5.11: dSOARS usage example (a) dSOARS object initialization (b) finding optimized file location using dSOARS library calls.

loads the cache table into memory (Figure 5.11a). Before opening a file, the application calls `findOptimal` as shown in Figure 5.11b. This function returns the optimal location as a string. By concatenating this with the base name we get the file path, which can now be opened using the standard UNIX system call.

Chapter 6

Performance Evaluation of dSOARS

DSOARS can improve the I/O performance of applications by dynamically selecting the best location for data, replicating files as necessary. We evaluate dSOARS using three applications on the CBSE cluster.

The first application is a parallel implementation of grep. Grep is a UNIX command to search for a pattern in the specified file(s). A list of files is given as input and is searched for a match of the given pattern. The list of files can be split to make parallel implementation of the sequential search program. This application is simple, data parallel, and easily load-balanced. The second and third applications are psLayout and WABA.

The rest of this chapter is organized as follows. We describe the experimental design in §6.1. Grep results are discussed in §6.2. In §6.3, we describe the psLayout experimental results. WABA results are described in §6.4. We summarize the performance of the user-level library in §6.5.

6.1 Experimental Design

We evaluated the performance of dSOARS in the CBSE cluster. Since it is a production system, no nodes and server load assumptions are made. Also, the number of processors scaled are up to 20, because we had low priority on the cluster and we could only get a small portion of the cluster. We are comparing the behavior of the application when the client makes the choice of the file location and when dSOARS determines the optimal location for the client dynamically. The possible file locations are NFS and local. In dSOARS implementation, the client program specifies the base name and the library determines the optimal file location for that instance. The overhead on including the user-level library is negligible.

All the experiments were run with only one existing copy of the file, at NFS. All the replicated copies at the local disks were removed before each test. Hence the test indicates the upper-bound behavior of the library, assuming uniform network behavior in the cluster.

6.2 Grep

The first application we examined is the parallel implementation of grep, a search utility. We choose this particular program to test because grep is a simple program whose execution time is a factor of the I/O to be done. Evaluating the performance of this program shows the performance gain of making the right I/O calls, through the dSOARS library.

Grep, the UNIX command does pattern matching of expressions. The inputs for this command are the file(s) to be searched and the expression. The output for this command is the list of the lines from the specified file(s), with their line numbers, that match the specified

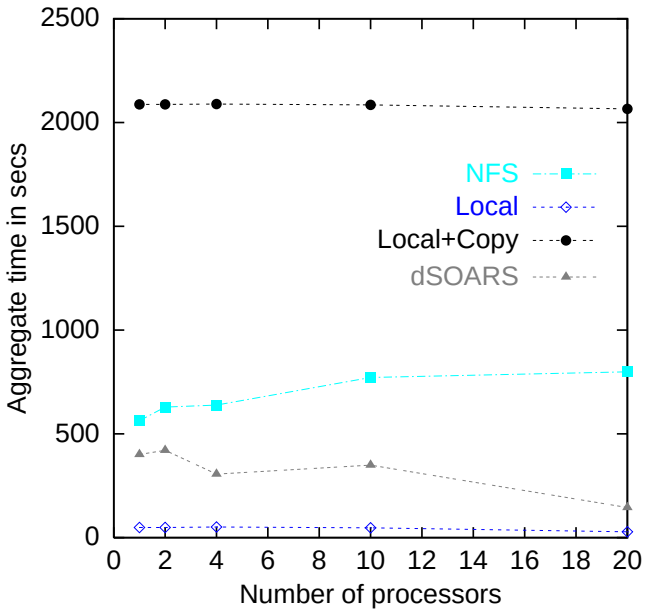


Figure 6.1: Performance of dSOARS implementation on a parallel implementation of grep.

expression. In our implementation, the input to the program are the file that contains the list of files to search for the expression and the search expression. We split the list of files and giving a portion of the work to different processors. The split was random with no uniformity and so each processor may do more or less work depending on the number of files and their respective sizes. In this experiment, we are not trying to perform load balancing, instead, we want to study the behavior of dSOARS implementation. There are 11 input files with a random mix in file sizes totaling 75 MB. The smallest and the largest file size are 1 MB and 20 MB, respectively. This experiment is parallelized to 20 processors.

We ran the same grep program with all the input files to be searched from NFS or all from local disk. For all the files to be at the local disk, they have to be copied there first. Since we do not know what node would be assigned by Condor to do the search, we need to copy the

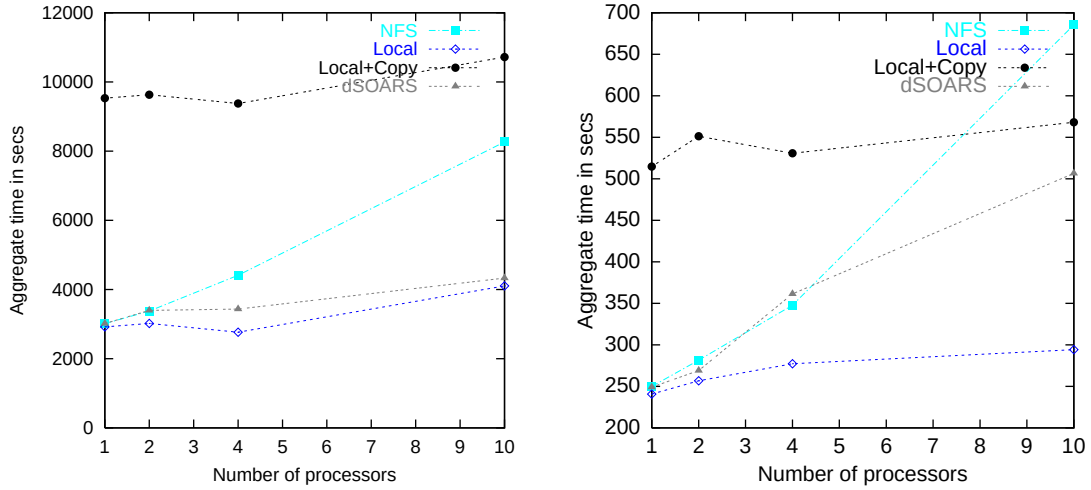


Figure 6.2: Performance of dSOARS implementation in (a) psLayout (b) WABA.

input files to all the nodes. The result of this experiment is shown in Figure 6.1. As we can see from the graph, NFS and Local locations provide good scalability although NFS access took 500 to 750 seconds longer. If files are replicated on all the local disks to increase performance, the search time is less (Local), but when the copy overhead is considered (Local+Copy), the execution time increases. In dSOARS, the right location for the files is determined and they are replicated, if the cost function favors an additional copy. For example, for the one and twenty processor runs, the number of files copied to local disk on each processor were 4 and 8 respectively. dSOARS implementation shows good performance, close to the local disk access performance, but without a huge copy overhead.

6.3 PsLayout

PsLayout, described in §3.2, takes two read-only files as input. The data set consists of 4 MB and 411 MB input files and is scaled to 10 processors. The 4 MB dataset is a collection of 52 FASTA files, each typically in the order of several KB. The library searches for each of the FASTA files to find the right location. The 411 MB is one large file and it involves one library file search. The performance of the dSOARS implementation is tested by comparing the results of dynamic replication with all the files access to be NFS or Local (including and excluding copy overhead). We did not test the larger run of psLayout scaled to 50 processors, as we had limited access to the resources.

The results are shown in Figure 6.2a. With small number of processors (one and two), the dSOARS implementation performs equivalently to NFS and Local access, without the copy overhead. With the increase in processors, the dSOARS implementation performs close to the local disk access with a small copy overhead. The dSOARS library replicates the input files to the local disk, if the cost function favored that decision. The number of replications of this run is dependent on the number of files replicated and the number of processors. For instance, no file is replicated for the one-processor run and the 411 MB file is replicated on both the processors for the two-processor run. For the ten-processor run the 411 MB and one other file is replicated on all ten nodes.

6.4 WABA

WABA, described in §3.3, takes two input files of sizes 22 KB and 48 MB. Both the files are one large FASTA file. The 22 KB file is split across safe "marker" positions to parallelize the application. Results using dSOARS are shown in Figure 6.2b.

Local disk access shows good scalability. When the files are accessed from the NFS server location, the execution time increases with the increase in the number of processors. In the dSOARS implementation, the number of replicated files for the first input is dependent on the file size and the number of processors. The number of replications is zero for one-processor run and one for each processor in the ten-processor run. Figure 6.2b shows reasonable performance of dSOARS when scaled. Local disk access is only considered with the added copy cost, if the cost function favors replication. Sometimes the dSOARS implementation shows worse performance, than all NFS reads (4 processor) because of the reads to temporary files created in the first and second passes, which are not optimized using dSOARS.

6.5 Summary

We have tested the dSOARS performance in CBSE cluster using a parallel grep implementation and a small experimental data set for psLayout, and WABA. The possible location choices were NFS and local disk. All the local copies were deleted before each run, hence the results are an upper bound and considerable performance gain is clearly seen as the number of processors increases. dSOARS is able to provide performance improvements by smart replication of data, providing good trade-off between server bottleneck and copy over-

head. The number of replicated copies depend on the number of processors and the file size. dSOARS shows good scalability across different file systems (NFS and Local) and processor configurations.

Chapter 7

Conclusions

We present in this thesis the design of a user-level library, dSOARS, providing location-transparent storage. dSOARS dynamically determines the optimal file location, making replicas if necessary. The library determines the optimal location based on a cost-function. The cost-function is complex, and as a first pass we have considered three parameters, file size, number of processors and file location, which we thought were most important. The access cost for each location is based on the results from a benchmark. The location that has the minimum access cost for that instance determines the optimal location. This implementation can be linked easily with program and we tested it on three applications. Our test results show that our first pass implementation show reasonable performance, and achieves a good trade-off between the server bottleneck and the replicated copy overhead.

7.1 Discussion

We characterized the performance of psLayout, a computational biology application that performs genomic alignment, on three architectures. The architectures varied in their interconnect speed and the file system. We also characterized the performance of WABA, a genome versus genome alignment application on CBSE cluster. In contrast to psLayout, WABA is compute-intensive. There are three passes in this application, and each pass reads the temporary output file written from the previous pass. Due to these temporary reads and writes, I/O intensity increases with the scaling of the number of processors.

We determined that although it is embarrassingly parallel, psLayout has poor scalability due to I/O contention and poor load balancing. We assessed scalability on a range of file systems and architectures ranging from the low-end CBSE cluster to ASCI-Blue. For psLayout, the best-performing combination of input databases for 10 processors with input file sizes 4 MB and 411 MB is different for each architecture: NFS/Local for CBSE, NFS/Local and PVFS/PVFS for Vivid and GPFS/GPFS for ASCI-Blue. Input file location is a major factor affecting the aggregate execution time of this application. We validated this conclusion by running a larger problem size with input file sizes 26 MB and 411 MB and scaling up to 50 processors. The data locality also affect performance of WABA on the CBSE cluster.

Computational biology is an important application area with different I/O needs than other scientific applications. The performance of these applications running on such large-scale environments depend on not only the processing power, but also data locality. The wrong data placement might incur large remote I/O overheads that ultimately degrade the per-

formance, which would defeat the purpose of using a distributed system. Hence data locality and access is a crucial issue for distributed systems. Although a fast network and parallel file system or a scalable NFS server can service the clusters and loads, we believe that replication of data will play an increasing role in scalability of this class of applications.

dSOARS is a library that implements smart replication based on a cost function to provide performance improvements. The design of this library consists of five major components: initialization, cache table, cost function, dSOARS and cleaning. The cache table maps the file locations with their access costs. When the base name is passed to the optimization function, the location with the minimum access cost is considered best. Sometimes, this results in an additional replicated copy. The client program can link the user level library and use this functionality by making a few modifications to the source code. When the dSOARS class object is initialized in the client program, the cache table is loaded. Before each optimization file open call in the client program, the dSOARS wrapper call must be called with the base name to determine the optimal location.

We tested the library by running the applications, parallel implementation of grep, a pattern matching program, psLayout and WABA in the CBSE cluster, but instead of manually providing the file location, we only specified the base names and the library chose the optimal location. We compared dSOARS performance with the performance obtained when the client selects the file location. Our first pass implementation shows good performance and scalability, with a good tradeoff between the NFS bottleneck and the copy overhead.

7.2 Future Work

To the current framework, we want to first add more dynamic parameter considerations. In addition to the benchmark result, by having a current network monitoring system, we may be able to predict behavior much more accurately.

As an alternative to the current implementation, the library function can be incorporated into the kernel level. There are several ways in which you can approach this problem. The best choice would be in the middle layer of the kernel at the vnode, this implementation is more often used currently and seems to have good trade-off between the performance gained and the semantic modifications needed for file system hacking [32]. Another alternative is to incorporate the library in a job scheduler. Typically a job scheduler matches hardware resources from the clients request to the available resource pool. We believe that data locality becomes a serious issue and it should be part of the matching function that determines the right processor to assign the job. Our ultimate goal is to design a system that provides data availability with improved performance through smart replication.

Bibliography

[1] ASCI Blue-Pacific. http://www.llnl.gov/asci/sc99fliers/blue_pacific_pg1.html, 9th July 2001.

[2] ASCI Blue-Pacific Bonus Links. http://www.llnl.gov/asci/platforms/bluepac/bonus_links.html, 14th Sept 2001.

[3] Condor High Throughput computing. http://www.cs.wisc.edu/condor/, 29th Mar 2001.

[4] General Parallel File System for AIX. http://www-1.ibm.com/servers/eserver/pseries/software/sp/gpfs.html, 16th Sept 2001.

[5] Growth of Genbank. http://www.ncbi.nlm.nih.gov/Genbank/genbankstats- .html, 20th July 2001.

[6] Historic notes about the cost of hard drive storage space. {http://www.alts.net/ns1625/winchest.html, 20th July 2001.

[7] Human Genome Project Information. http://www.ornl.gov/hgmis/, 9th Apr 2001.

[8] Myrinet Overview. http://www.myri.com/myrinet/overview/, 14th Sept 2001.

[9] NCBI Databases. http://www.ncbi.nlm.nih.gov:80/Database/index.html, February 2001.

[10] Parallel Code Support-ASCI Blue-Pacific. http://www.llnl.gov/asci/platforms/bluepac/parallel.html#MPI, 14th Sept 2001.

[11] Areal density increases data rates. http://www.storage.ibm.com/hdd/library/density.htm, 21st Feb 2002.

[12] GPFS Parallel File System. http://www.almaden.ibm.com/cs/gpfs.html, 20th Jan 2002.

[13] Human Genome Project Information. http://www.cse.ucsc.edu/ rachelk, 2nd Jan 2002.

[14] LoadLeveler. http://www-1.ibm.com/servers/eserver/pseries/software/sp/load- leveler.html, 3rd Jan 2002.

[15] NetApp Filers. http://www.netapp.com/products/filer/, 31st Jan 2002.

[16] Parallel System Support Programs for AIX. http://www-1.ibm.com/server/eserver/pseries/software/sp/pssp.html, 6th Feb 2002.

[17] The Globus Project. http://www.globus.org, 3rd Jan 2002.

[18] TOP500 List for November 2001. http://www.top500.org/lists/2001/11, 9th Jan 2002.

[19] A.A.Mirin, R.H.Cohen, B.C.Curtis, W.P.Dannevik, A.M.Dimits, M.A.Duchaineau, D.E.Eliason, D.R.Schikore, S.E.Anderson, D.H.Porter, P.R.Woodward, L.J.Shieh, and S.W.White. Very High Resolution Simulation of Compressible Turbulence. In *Supercomputing 99 conference*, number UCRL-JC-134237, 1999.

[20] Bill Allcock, Ian Foster, Veronika Nefedova, Ann Chervenak, Ewa Deelman, Carl Kesselman, Jason Lee, Alex Sim, Arie Shoshani, Bob Drach, and Dean Williams. High-Performance Remote Access to Climate Simulation Data: A Challenge Problem for Data Grid Technologies. In *SC2001*, 2001.

[21] Khalil Amiri, David Petrou, Gregory Ganger, and Garth Gibson. Dynamic Function Placement in Active Storage Clusters. Technical Report CMU-CS-99-140, CMU Parallel Data Laboratory, 1999.

[22] James W. Arendt. Parallel genome sequence comparison using a concurrent file system. Technical Report UIUCDCS-R-91-1674, University of Illinois at Urbana-Champaign, 1991.

[23] Brown, Peter, Britton Chang, Keith Grant, Ulf R. hanebutte, Carol S.Woodward, and Thomas A.Brunner. ARDRA:Scalable Parallel Code System to Perform Neutron and Radiation Transport Calculations. In *SC99*, 1999.

[24] Philip H. Carns, Walter B.Ligon III, Robert B.Ross, and Rajeev Thakur. PVFS:A Parallel File System for Linux Clusters. http://parlweb.parl.clemson.edu/pvfs/el2000/extreme2000.html#simitci:frame- work, 14th Sept 2001. Parallel Architecture Research Laboratory.

[25] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, pages 860–921, February 2001.

[26] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Characterization of a Suite of Input/Output Intensive Applications. In *Proceedings of Supercomputing '95*, December 1995.

[27] D.A.Thompson and J.S.Best. The future of magnetic data storage technology. http://www.research.ibm.com/journal/rd/443/thompson.html, 2000. Volume 44,Number 3.

[28] Dr.M.Hill. The Human Genome- About FASTA files. http://anatomy.med.unsw.edu.au/cbl/GENOME/about/aboutfasta.htm, 29th Mar 2001.

[29] Gordon E.Moore. Progress in digital integrated electronics . In *IEEE Digital Integrated Electronic Device Meeting*, 1975.

[30] Fattebert, Jean Luc, and Francois Gygi. A continuum solvation model for ab initio molecular dynamics. In *American physical society*, 2001.

[31] Remzi H.Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E.Culler, Joseph M.Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River:Making the Fast Case Common. In *Input/Output for Parallel and Distributed Systems*, 1999.

[32] Dave Hitz, James Lau, and Micheal Malcolm. File system design for an NFS File Server Appliance. Technical Report TR3002, Network Appliance Inc.

[33] William Kent. The Human Genome Project and UCSC. Web Page – http://www.soe.ucsc.edu/ kent/presentations/ScholarsDay2001/, 28th Sept 2001.

[34] W.James Kent. Gigassembler: An algorithm for the initial assembly of the human genome working draft, http://genome.ucsc.edu/goldenPath/algo.html. Technical report, School of Engineering,University of California,Santa Cruz, 2000.

[35] W.James Kent and Alan M.Zahler. Conservation,regulation,synteny, and introns in a large-scale c.briggsae-c.elegans genomic alignment. In *Genome Research*, August 2000.

[36] Tieng K.Yap, Ophir Frieder, and Robert L.Martino. Parallel computation in biological sequence analysis. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):283–294, March 1998.

[37] John L.Hennessy and David A.Patterson. *Compter Architecture a quantitative approach*. Morgan Kaufmann Publishers, Inc., 1996.

[38] Brian L.Tierney, Jason Lee, Brian Crowley, Mason Holding, Jeremy Hylton, and Jr Fred L.Drake. A Network-Aware Distributed Storage Cache for Data Intensive Environments. Technical Report LBLN-42896, Lawrence Berkeley National Laboratory.

[39] Marshall Kirk McKusick, Keith Bostic, Michael J.Karels, and John S.Quarterman. *he Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley , 1996.

[40] MPI-IO: a parallel file I/O interface for MPI. The MPI-IO Committee, April 1996. Version 0.5.

[41] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, October 1996.

[42] Personal communication. Thomas R. Slezak, Lawrence Livermore National Labs, September 2001.

[43] Daniel A. Reed, Ruth. A. Aydt, Roger J. Noe, Philip C. Roth, Keith A. Shields, Bradley Schwartz, and Luis F. Tavera. Scalable performance analysis: The pablo performance analysis environment. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer, October 1993.

[44] R.Manaa. Towards a new energy-rich molecular systems: from N10 to N60. In *Chemistry Physics letters*, 2000.

[45] Rob Ross. The Parallel Virtual File System. http://parlweb.parl.clemson.edu/pvfs/, 14th Sept 2001.

[46] Rotman, C.Atherton, D.Bergmann, P.Cameron-Smith, C.Chuang, P.Connell, J.Dignon, A.Franz, K.Grant, A.Mirin, C.Molenkamp, and J.Tannahill. IMPACT, A coupled tropospheric/stratospheric chemistry model:Analysis and comparison of results to observations. In *American Geophysical Union Annual Fall meeting*, 2000.

[47] Frank Schmuck and Roger Haskin. GPFS-A Shared Disk File System for Large Computing Clusters. In *File And Storage Technologies*, 2002.

[48] Evgenia Smirni, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. I/O Requirements of Scientific Applications: An Evolutionary View. In *Fifth International Symposium on High Performance Distributed Computing*, pages 49–59, 1996.

[49] Evgenia Smirni and Daniel. A. Reed. Workload characterization of input/output intensive parallel applications. In *Modelling Techniques and Tools for Computer Performance Evaluation*, June 1997.

[50] Neil Spring and Rich Wolski. Application Level Scheduling of Gene Sequence Comparison on Metacomputers. In *ACM International Conference on Supercomputing*, 1998.

[51] Jon William Toigo. Avoiding A Data Crunch. *Scientific American*, 282:58–74, May 2000.