8-2003

Data Access in Wide Area Networks of Heterogeneous Workstations

Kim Buckner
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a dissertation written by Kim Buckner entitled "Data Access in Wide Area Networks of Heterogeneous Workstations." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

James S. Plank, Major Professor

We have read this dissertation and recommend its acceptance:

Bradley Vander Zanden, Jeffrey Becker, Micah Beck

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a dissertation written by Kim Buckner entitled "Data Access in Wide Area Networks of Heterogeneous Workstations." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

James S. Plank

Major Professor

We have read this dissertation
and recommend its acceptance:

Bradley Vander Zanden

Jeffrey Becker

Micah Beck

Acceptance for the Council:

Anne Mayhew

Vice Provost and Dean of Graduate Studies

(Original signatures are on file with student records.)

Data Access in Wide Area Networks of Heterogeneous Workstations

A Dissertation
Presented for the
Doctor of Philosophy Degree
The University of Tennessee, Knoxville

Kim Buckner
August 2003

Abstract

The accessibility of data in wide area networks can be difficult. This research shows the use of the Internet Backplane Protocol (IBP) along with a modified version of the C standard I/O library that can allow data to be easily accessible without having to make major modifications to legacy code. In fact if legacy programs only use standard input and output routines, they need only be recompiled to effect a homogeneous file system. It also demonstrates that this access is predictable enough to make decisions on what data to access and in what fashion that access is most effective.

DEDICATION

This dissertation is dedicated to my wife, Laura, without whose loyalty and support I would not have been able to stay the course and to my parents, Bonnie and Carl Buckner for the example they set and the encouragement they gave.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The expanding use of wide-area networks requires new ways of scaling applications to take advantage of the available computing resources. Large heterogeneous collections of workstations are becoming readily available to the average business or research department. Many of these organizations have existing software applications which can require long running times to perform their operations. This may be due to the application being very computationally intensive or to its having heavy input/output requirements or some combination of these and other factors. The same organizations which have available to them these relatively large collections of workstations do not have the additional monetary resources to provide dedicated fast networks, high-performance clusters, Symmetric Multiprocessors (SMP) or the like, or high-capacity storage servers. A typical example of such an organization is a small private college or university.

Wide-area settings should be able to provide increased computing power if they can be properly harnessed. In order to harness such settings, applications must be able to execute on some reasonably sized subset of the resources available to the user. Not all applications can be readily adapted to the wide-area.

High-performance parallel numerical applications generally require some uniform communication layer such as Parallel Virtual Machine (PVM) [33] or Message Passing Interface (MPI) [12]. These applications also often require some specific number of processors, for instance a power of two. Such applications usually need a large amount of interprocess communication in the form of scatters, gathers, reductions or other operations required to share state among the different components of the computation. They often do not lend themselves well to situations where the number of processors is arbitrary, the processing environments are heterogeneous, or the communication layer is not uniform. This is not to say that high-performance parallel numerical applications have not been made to perform well in a wide-area environment, only that they are not very flexible.

Serial applications, those whose computational structure show very little potential for parallelization, also seem inappropriate. Having more machines on which to execute these applications does nothing to improve their performance since they are normally designed to operate on a single processor. But some serial applications can accurately process subsets

of the original input data or accurately perform subsets of their total operation, depending on input, and require no interprocess communication.

If existing serial applications are capable of processing a portion of their data accurately or can be modified to do so, then the applications can be expanded to the wide-area, providing that data can be made available to the individual processors in such a manner that data access is not a serious bottleneck. The applications may involve storage and/or access of very large data files, files with sizes larger than two gigabytes. Many operating systems and file systems are becoming capable of handling these large files but physical storage is still often limited to specific disk farms or other repositories such as offered by the Internet 2 Distributed Storage Infrastructure (I2DSI) [5]. Reading and writing of files stored in these limited, often remote, locations can not always be made transparent to user applications because all the resources do not share a uniform file system and many legacy applications are dependent upon the file access methods available when they were designed.

Even though use of wide-area environments presents many problems, the problems are not insurmountable. This research will show that not only can legacy serial applications be expanded into a wide-area setting but that such applications can be modeled with sufficient accuracy to be able to make scheduling decisions based on the model.

The rest of this dissertation is organized as follows: Chapter 2 provides background information. Chapter 3 discusses the chosen application and the software tools used. Chapter 4 covers the application model. Chapter 5 deals with gathering of the model validation data. Chapter 6 covers the results of the initial testing and model verification. Chapter 7 contains the results of large file testing. Chapter 8 gives conclusions and talks about directions for continuing work

Chapter 2

Background

Software is being developed at many commercial and research sites to help take advantage of newly available wide-area resources. So much is being developed in fact that the user is often presented a confusing melange of choices.

2.1 Current work

Heterogeneity has been one of the main problems of wide-area computation. Creating portable software has become easier as operating systems become more similar and begin to support standards such as the Portable Operating Systems Interface (POSIX) [19]. The need for separate binaries still exists but given proper preparation, an application can be compiled to meet the requirements of a new platform fairly quickly, that is if it does not rely upon operating system specific or vendor specific support.

Efforts such as Globus [13] and Legion [16] are middleware environments which attempt to exploit the potential of the "Computational Grid" for high-performance computations. They seek to provide a homogeneous platform on which to execute by supplying uniform resource management and allocation mechanisms for global resources. WebOS [34] provides wide-area operating system services. It allows for remote process invocation and load balancing. It also provides a wide-area file system. WebOS attempts to make execution in a wide-area environment transparent to applications.

Condor [23] exploits the potential of idle resources. It migrates processes among a set of participating processors (workstations). While this is a dynamic environment, it does not support truly parallel applications. The heterogeneity of the environment becomes a problem if a process must be checkpointed and moved. The checkpoint may not be portable or the results of the computation may become unreliable due to different data representations. The process may also remain idle for long periods because binaries are not available for the architectures that are currently free. NetSolve [8] is another project that harnesses available resources. It is a client-server model with an agent that enables remote execution of problems (functions) in a manner similar to Remote Procedure Call (RPC) [31]. The interaction between the clients, servers and agents is architecture independent and the

problems can be customized to operate in almost any setting.

Along with execution environments, data storage and access need to be able to adapt to the wide-area environment. Globus has three components that provide data access: GASS [6] supplies a global name space and an Application Program Interface (API) for remote file access and caching; RIO [14] enables remote I/O for parallel applications; GridFTP [2] [27] is a data transfer protocol based on extensions to the standard FTP protocol. Legion has a global file system, composed of files represented as storage objects (vaults) [21]. Each object can be customized for its use with a particular application. The Jade [28] file system was designed for file access in an Internet. It defines naming conventions and access protocols that allow processes to transparently access local and remote files. Ufo [1] is implemented as a user-level extension to the operating system. It uses the tracing facilities of the operating system to intercept system calls, allowing applications to treat remote files as if they were local. Smart File Objects (SFO) [35] are implemented as middleware to take advantage of the time during the application's compute operations to prefetch data in order to help alleviate the slowdown due to data access overhead. It uses the Legion global file system for data access.

WebFS is a component of WebOS. WebFS is designed to provide flexibility in the wide-area environment and it implements many of the same types of operations as file systems such as the Network File System (NFS) [30], including caching and a global name space (using Uniform Resource Locators (URLs)). xFS [3] does away with the 'standard' file system model of a single server or a small set of servers through which all files must be accessed. Instead it uses a server-less model in which any participating machine in the system can provide any and all file system services. CFS [11] is a peer-to-peer read-only file system based on the secure file system SFS [15]. It uses small storage blocks and block replication to improve performance. IBP, the Internet Backplane Protocol [25], is being developed to provide storage across networks. It provides a very low level set of data storage and retrieval operations and allows the movement of data to be directed by the application. IBP requires that a server or another application be running at the data destination and that sufficient storage space for the data be available.

2.2 Restrictions and Difficulties

Globus and Legion seek to provide a uniform layer upon which to build wide-area computations. However, they can be difficult to port and require that the local area administrators relinquish some autonomy by implementing specific components of the Globus or Legion process control and access structure. WebOS leverages several other software applications to provide its services. One of these relies on the Solaris */proc* file system to provide system security. Even though many applications are restricted to an operating system such as Unix, reliance on vendor specific implementations makes applications such as WebOS inherently non-portable. The */proc* file system is being used by many current operating systems such as Solaris2, Solaris8, and Linux. However there is no standard for its implementation and even the Solaris2 and Solaris8 systems have completely different API's.

Condor harnesses resources (workstations), primarily in a local area, that are currently unused and that some user or owner has allowed to be 'harvested'. It can only access user files from the user's machine (machine on which a particular process was started). If a process is moved, that user's machine is still required for file access, presenting a serious problem if that workstation fails and incurring additional penalties in network overhead for data transfer. NetSolve uses a client-server model to provide RPC-like services. It does not provide a global environment and maintains no state between calls. Data required by the problem is sent from the client to the server and the results are sent back. Each problem that a server can solve must be defined in a *problem description* and may need to include additional code to provide data translation, file access, error handling, et cetera. These *problem descriptions* must be hand crafted by someone familiar with the server's environment, its restrictions, and the application's requirements.

NFS is basically limited to the local area and does not scale well to the wide-area. The Andrew File System (AFS) [17] and the Distributed File System (DFS) [18] can operate in a wider area environment but are kernel based and thus can present a portability problem. For example the DFS server is currently limited to the AIX and Solaris operating systems. Coda [29] was an attempt to expand the capabilities of AFS by replicating file servers to provide scalability. Globus' RIO provides remote I/O operations but requires that the applications use MPI-IO. GASS provides mechanisms that can be used to access remote files in many different modes provided that the sites that are holding the files are participating Globus sites and that the programmers understand the GASS mechanisms well enough to make the correct accessing choices. GridFTP is a protocol that data storage systems must implement in order to provide access to GridFTP clients. Legion can only access files that are in the Legion space. Accessing data outside of this space can be difficult or impossible at runtime. As SFO uses the Legion object space it has the same limitation on data access. Additionally, SFO is currently only implemented for the Mentat system.

xFS uses RAID style data striping across the participating machines. It is only effective in a restricted environment where the participating machines are connected by very fast networks such as Myrinet. IBP is not a file system per se but leverages local file systems to provide application transparent data storage. It does require that a server be running somewhere but the server is accessed via sockets so its location is flexible. Writes to IBP *byte arrays* are append-only so editing data in place is not currently possible. No special access restrictions are enforced. If any process belonging to any user possesses a valid 'capability', that process can do anything to the *byte array* that the capability allows.

Jade also uses local file systems. The Jade library is dynamically linked so it does not require re-compilation. It provides the user a method of mounting resources, remote or local, in a per user directory. But the namespace is also per user, so that exporting a file name does not guarantee access to the file. All remote files are cached in their entirety in the local file system on opening. This can impact availability of local resources seriously in the case of large data sets and leads to cache coherency problems. Ufo, like Jade, personalizes the view of the file system. It ensures that a current local copy of any remote file exists when that file is opened. Ufo does not require re-compilation but relies on attaching to a

process via */proc* and intercepting system calls to give the user transparent access to remote files.

Unlike other systems that require specialized access permissions, operating system kernel changes or other non-portable modifications, the proposed system, combining IBP and the C standard I/O library, simply extends the functionality of the standard I/O library and provides a more user-friendly interface to IBP with few limitations. It can provide a legacy program with access to globally located data by the simple expedient of re-compiling the application. It also provides standard file system access to remote storage that is not normally accessible to a process, such as the local disk of a remote desktop computer. It does require that an IBP server be running on the processor that has access to the actual data storage location, but that is no different from other standard file systems.

Chapter 3

Application and Tools

3.1 FASTA

FASTA is a set of programs based on a fast algorithm for biological sequence comparisons [24]. These sequences are character representations of sets of proteins. The sequences may represent those found in DNA or may be from other sources. The current version of the distribution when this research was begun was 3.2. The particular program used is "fasta3" which can be compiled for serial or parallel (PVM only) use. This application and similar ones are in daily use by many genome projects around the world and there are large data libraries readily available from sources such as the GenBank®genetic sequence database at the National Center for Biotechnology Information part of the National Library of Medicine at the National Institutes of Health [1] .

The different versions of the program can operate from command-line arguments alone, or a mix of command-line and standard input with querying of the user for necessary information as needed. The program as delivered always writes some information to standard output but parts of the output can be sent to a file with command-line arguments.

3.1.1 FASTA Serial Version Description

The serial version "out-of-the-box" performs these operations.

1. It validates the command-line arguments.
2. It opens the query file containing the query sequences (this can be standard input if only one (1) sequence is being tested).
3. Then it opens the "FASTLIB" file, a file containing a set of library description entries. Each consists of:
   (a) a file description,

[1] http://www.ncbi.nlm.nih.gov/Genbank/GenbankOverview.html

(b) a file content type (protein sequence or DNA sequence) indicator,

(c) an arbitrary character assigned by the creator of this FASTLIB file used to allow fasta3 to differentiate between entries or to group files (all files with the same character assigned will be selected for searching),

(d) and the filename of a file that either contains a set of reference DNA or protein sequences for comparison or contains the names of other files of reference sequences.

4. The process extracts the reference data filenames from the FASTLIB file that match the command line character. Only one character is allowed on the command line.

5. It then reads in the first query and begins reading from the reference files to compare the sequences. During each comparison of the query sequence to a reference sequence, a relative score is assigned which represents how well the reference sequence matches the query sequence. The program maintains a list of the best scores and the corresponding file offsets of those reference sequences. The serial version uses standard I/O functions to read the data files.

6. Once the query is compared against all the reference sequences, the output is generated, re-reading portions of the reference files to get the sequences that have been selected as the best matches so that the output can be generated.

7. Repeat from 5) for the remainder of the queries. This version maintains no portion of the reference file in memory so it starts at the beginning of the first reference file for every query sequence processed.

3.1.2 FASTA Parallel Version Description

The parallel version "out-of-the-box" master process works similarly.

1. It validates the command-line arguments.

2. It then determines how many workers are needed from command-line arguments and spawns them using PVM.

3. Contact is established with the workers and each is informed of the filename of the reference file along with which portion of the file each should process. Only a single reference file is allowed in this version.

4. It sends all workers the first query once they all check back in indicating they have their reference data.

5. The master prepares the next query to send.

6. It then waits on the workers.

7. As they finish it receives each worker's results.
8. It tabulates the results and then sequentially requests from the appropriate worker the reference sequence data for the selected best matching sequences.
9. After all the data to generate the output has been received, it sends the next query to the workers and writes the output.
10. The master repeats from 5) until done.

The parallel worker operation is markedly different.

1. Each receives the initialization information from the master process.
2. It then opens the reference file and reads from that file its portion of the reference data using non-buffered input/output (**open()** and **read()**).
3. It blocks waiting for the next communication from the master.
4. If the communication is a request for sequence data, the worker sends the data else it goes on to the next step else it exits.
5. The worker then compares the query to the in-memory portion of the reference file.
6. Last it sends the results to the master and blocks at step 3) waiting for a response.

3.2 IBP

File access over a wide-area setting is an obvious problem. There is no guarantee of a uniform name space or uniform file access semantics. IBP can be used to address this problem. It provides for very primitive data storage and retrieval. Data is treated as simple arrays of bytes with no notion of machine-dependent format. This means that applications using IBP must decide on internal data representations if access will be across platforms or use readily available translations such as eXternal Data Representation (XDR) [32]. IBP allows for byte array creation, removal, write (append only), read, and remote store. It also gives the user access to FIFOs (usually for inter-process communication), a uniform data access API and global name space. The storage that IBP uses in any local area network is determined beforehand and can consist of any standard file storage space in the local file system. The amount of space to be used is determined when the server is started so local administrators can control IBP's resource usage. Unlike NFS, IBP has no special access protection. It manages its data as local files with whatever access permissions are granted to the server process via its owner. That means the owner of the server process can then access the data stored in IBP byte arrays created by that server regardless of who stored the data. But just as the IBP server will have to be trusted by the local administrator, either the owner of the server will have to be trusted by the client or the data in byte arrays can be encrypted.

3.3 NetSolve

Another problem is remote invocation of copies of the application. Utilities such as remote shell (rsh) and secure shell (ssh) can be used but are fairly limited in scope. NetSolve can be leveraged to provide easier access to remote resources. NetSolve, like IBP, only requires a server to be running on the machine where one desires to execute. The server accepts jobs from clients who become aware of the server via an intermediary agent. This agent is known to both the server and the client but need not be near either one. The server accepts jobs based on the problem to be solved and any restrictions placed on it when it was started. Once completed, the results of the job are returned to the client, and the server maintains no state. Again, like IBP, any files created and any processes spawned by the server only have the access capabilities granted the owner of the process, modified by any restrictions specified to the server. NetSolve starts the applications in a simple manner using fork-and-exec. Access to the servers and agents is via sockets and access control can be specified at the server by a configuration file which can limit access to certain domains or addresses much like Unix's *rhosts*.

3.4 IBP_STDIO

In general, applications can be accessed via NetSolve without any code modification or recompilation. This is because its 'problem descriptions' can be as simple as a call to the Unix **system**() function. Rewriting applications to use IBP instead of the local file system is not usually difficult providing the applications rely on such basic file operations as **read**() and **write**(): simply replace those system calls with the corresponding IBP calls. Because the serial version of FASTA, like many other programs, primarily uses the standard input/output (stdio) library functions such as **fopen**(), **fread**(), **fgetc**() and **ungetc**(), IBP_STDIO, a modified version of the FreeBSD Unix C stdio library, was developed as part of this dissertation. This allows access to IBP byte arrays while still allowing use of the underlying system libraries to access local files if desired. The creation of IBP_STDIO resulted in very few changes being made to the FASTA code because all the file access semantics, including function names and arguments, remain the same. The IBP_STDIO library is statically linked to the application's code which only to has to have some *#include*'s changed. Otherwise its use is transparent to the application. Depending on the name of the file, operations are performed on IBP byte arrays, or they are passed to the appropriate standard library functions for operations in the local file system.

Chapter 4

Modeling and Testing the Application

4.1 Application Model

Modeling the application in terms of physical properties of the networks, processors and files is the focus of this chapter. These are some of the more basic components that will be used in the model:

D_{RE} = Apparent disk read bandwidth for the executable file, in megabytes per second.

D_{RQ} = Apparent disk read bandwidth for the query file, in megabytes per second.

D_{RR} = Apparent disk read bandwidth for a reference file, in megabytes per second.

D_{WL} = Apparent disk write bandwidth for the log file, in megabytes per second (using write())

D_{WO} = Apparent disk write bandwidth for the output file, in megabytes per second (using fprintf())

S_L = Size of a log file in megabytes

S_O = Size of an output file in megabytes

S_Q = Size of the query file in megabytes

S_R = Size of a reference file in megabytes

The data and file sizes used are all in megabytes. The bandwidths for the equations are all in megabytes per second.

4.1.1 Basic Equation

The basic model of the program can be written

$$T_T = T_1 + T_2 + T_3 + T_4 + T_5 + T_6 \tag{4.1}$$

where

T_T = total execution time.

T_1 = time in which the task is spawned, the program validates the command line and then prepares to read the first query.

T_2 = time to read the queries into memory

T_3 = time to read the reference file(s) into memory.

T_4 = time it takes to compare the queries to the references.

T_5 = time to compute the final values for the selected sequences, prepare and write the output.

T_6 = time to write final statistics to files and perform cleanup prior to exiting.

4.1.2 Breakdown of Times

Equation (4.1) is overly simplistic of course and does not show the many different factors involved. Here are the details of those factors.

1. T_1 has a number of separate components.

$$T_1 = S_E/D_{RE} + S_{O_{LH}}/D_{WO} + F_{O_{open}} + F_{O_{fopen}} \tag{4.2}$$

   is the equation for T_1 in which

   (a) S_E = the size of the executable file that is read from disk to memory by the operating system.
   (b) $S_{O_{LH}}$ = the size of the header written to the log file.
   (c) $F_{O_{fopen}}$ = the time to open and create an output file using fopen().
   (d) $F_{O_{open}}$ = the time to open and create a log file using open().

   In the tests run for validation, both log and output files reside on the same disk. This time is typically dominated by the time to read the executable.

2. T_2 is composed of the time is takes to read the first query, rewind the file, and read all the query sequences. The first query is used to make an initial determination of the type of sequences being used, either protein or DNA. The query file is expected to be uniform, all queries of one sequence type. Because query sequences are usually smaller than the size of the buffer that standard I/O libraries use (usually 8192 bytes), the second read of this sequence can be safely ignored. The value can then be expressed in terms of file size and apparent disk access time as:

$$T_2 = S_Q / D_{RQ}. \tag{4.3}$$

3. T_3 is the time it takes to read the data from the reference file(s) multiplied by the number of queries. This multiplication is necessary because the references are not saved in memory as they are read, and when a new query is started the reading of the reference file(s) starts at the beginning. However, when using NFS or another local file system which may cache file data or IBP_STDIO with read caching enabled, repetitive reads of cached data do not require remote disk access, rather they access local memory or possibly swap space on local disk. The amount of data that can be cached is dependent on several factors, the most important being:

   (a) the amount of physical memory available,

   (b) the current usage of local memory by processes,

   (c) and the amount of reference data read.

   For instance, in the tests performed to determine the disk I/O statistics, it was seen that if multiple reads of the same 189 megabyte file were performed using Sun Ultra 2 or Sun Ultra 5 machines, both of which had 256 megabytes of RAM, the subsequent reads of the data, even by a new process, took significantly less time, by at least a factor of 10. Sun Ultra 1 machines which also had 256 megabytes of RAM did not appear to cache that much data or cache it for any significant time between process invocations. Repetitive reads on these machines showed much more uniform results. However if the amount of data was reduced to approximately half of the size of the RAM, then all three sets of machines showed similar results.

   The amount of data read is the total amount of reference data divided by the number of processors involved. With 70 processors and 189 megabytes of reference data, each processor reads approximately 2.7 megabytes. On all machines in the test set this is cached for the life of the process either in local memory or in local swap space.

   Thus, for a given processor on which the amount of reference data accessed fits into some file cache, T_3 can be expressed as:

$$T_3 = \sum_{i=1}^{N_R} (S_{R_i} / D_{RR_i}) + (N_q - 1) * \sum_{i=1}^{N_R} (S_{R_i} / D_{RC_i}), \tag{4.4}$$

where it is assumed that each processor has a cache separate from all other processors. For those processors on which the data will not all fit into the cache and assuming LRU cache replacement policy we have:

$$T_3 = N_q * \sum_{i=1}^{N_R} (S_{R_i}/D_{RR_i}), \tag{4.5}$$

where

(a) N_R = the number of reference files accessed by the processor.

(b) S_{R_i} = the size of the data from reference file **i**.

(c) D_{RR_i} = the apparent disk read bandwidth for the disk on which reference file **i** resides.

(d) D_{RC_i} = the apparent disk read bandwidth for the disk on which cached reference reference data from file **i** resides.

(e) N_q = the number of queries.

4. T_4 can be expressed as:

$$T_4 = N_q * \left(\sum_{i=1}^{N_R} S_{R_i} \right) / C_p, \tag{4.6}$$

where

(a) C_p = the time the it takes to compare a query to one megabyte of reference sequence data on processor **p**.

(b) N_R = the number of reference files.

C_p is a constant value for each type of processor. It can vary from machine to machine depending on the current computational load but in the long run, these deviations average out. For the FASTA process, the bulk of the operations are integer comparisons and the computation for the score is very small in terms of floating point operations.

5. In T_5 each selected best sequence (the tests used a default of 20) is re-read from the reference file(s) twice, once while preparing the best scores and once while preparing the alignment of the reference sequences with the query sequence.

   This time then becomes a function of the total time to print (write) the output, represented as

$$P_T = P_{O_H} + P_{O_{BT}} + P_{O_{AT}}, \tag{4.7}$$

   plus the time to re-read the selected reference sequences twice

$$R_T = \sum_{i=1}^{N_q} \left(2 * \sum_{j=1}^{N_{R_i}} \left(S_{S_{ij}}/D_{RR_{ij}} \right) \right). \tag{4.8}$$

The print times can be further broken down to the apparent disk write bandwidth times the amount of data being written:

$$P_H = S_{O_A}/D_{WO}, \tag{4.9}$$

$$P_{BT} = \left(\sum_{i=0}^{N_q} S_{O_{Bi}}\right)/D_{WO}, \tag{4.10}$$

$$P_{AT} = \left(\sum_{i=0}^{N_q} S_{O_{Ai}}\right)/D_{WO}. \tag{4.11}$$

These combined yield:

$$P_T = \left(S_{O_H} + \sum_{i=1}^{N_q} (S_{O_{Bi}} + S_{O_{Ai}})\right)/D_{WO} \tag{4.12}$$

Combining equations (4.8) and (4.12) leads to

$$T_5 = \left(S_{O_H} + \sum_{i=1}^{N_q} (S_{O_{Bi}} + S_{O_{Ai}})\right)/D_{WO} + \sum_{i=1}^{N_q} \left(2 * \sum_{j=1}^{N_{R_i}} \left(S_{S_{ij}}/D_{RR_{ij}}\right)\right) \tag{4.13}$$

where

(a) N_{R_i} = the number of selected best reference sequences for query **i**.
(b) P_{O_H} = the time to print the output header.
(c) $P_{O_{BT}}$ = the total time to print the best scores for all queries.
(d) $P_{O_{AT}}$ = the total time to print the alignment for all queries.
(e) $S_{S_{ij}}$ = the size in megabytes of reference sequence **j** matching query **i**.
(f) $D_{RR_{ij}}$ = the apparent disk read bandwidth for reference file **j** of query **i**.
(g) S_{O_H} = the size in megabytes of the output header.
(h) $S_{O_{Bi}}$ = the size in megabytes of the output best scores for query **i**.
(i) $S_{O_{Ai}}$ = the size in megabytes of the output alignment for query **i**.

6. T_6 includes printing a small program statistics summary to the output file, printing statistics to the log file, and closing the query file. The file close operation time is so small as to be unmeasurable because the file buffers are flushed at the end of the T_5. This means that the file close operation is simply a local software operation and

requires no real disk access. This component can then be expressed in terms of the printing operations,

$$T_6 = P_{OS} + P_{LS}. \tag{4.14}$$

Here

(a) P_{OS} = time to print program (output) statistics and

(b) P_{LS} = time to print log statistics

and these can be broken down to

$$P_{OS} = S_{OS}/D_{WO} and \tag{4.15}$$

$$P_{LS} = S_{LS}/D_{WL} \tag{4.16}$$

resulting in

$$T_6 = S_{LS}/D_{WL} + S_{OS}/D_{WO} \tag{4.17}$$

4.1.3 Final Model

There are two cases of the final model. The first is the case of no read caching. For this, substituting equations (4.2), (4.3), (4.5), (4.6), (4.13) and (4.17) into equation (4.1) gives

$$\begin{aligned} T_T = S_E/D_{RE} + S_{O_{LH}}/D_{WO} + F_{O_{open}} + F_{O_{fopen}} + \\ [S_Q/D_{RQ}] + \left[N_q * \sum_{i=1}^{N_R} (S_{R_i}/D_{RR_i})\right] + \\ \left[N_q * \left(\sum_{i=1}^{N_R} S_{R_i}\right)/C_p\right] + \left(S_{O_H} + \sum_{i=1}^{N_q} (S_{O_{Bi}} + S_{O_{Ai}})\right)/D_{WO} + \\ \sum_{i=1}^{N_q} \left(2 * \sum_{j=1}^{N_{R_i}} \left(S_{S_{ij}}/D_{RR_{ij}}\right)\right) + [S_{LS}/D_{WL} + S_{OS}/D_{WO}]. \end{aligned} \tag{4.18}$$

The second case considers the situations where caching of read data is expected to have a significant impact. Here equation (4.4) is used in place of (4.5) giving:

$$T_T = S_E/D_{RE} + S_{O_{LH}}/D_{WO} + F_{O_{open}} + F_{O_{fopen}} + [S_Q/D_{RQ}] + \left[\sum_{i=1}^{N_R}(S_{R_i}/D_{RR_i}) + (N_q - 1) * \sum_{i=1}^{N_R}(S_{R_i}/D_{RC_i})\right] + \left[N_q * \left(\sum_{i=1}^{N_R} S_{R_i}\right)/C_p\right] + \left(S_{O_H} + \sum_{i=1}^{N_q}(S_{O_{Bi}} + S_{O_{Ai}})\right)/D_{WO} + \sum_{i=1}^{N_q}\left(2 * \sum_{j=1}^{N_{R_i}}\left(S_{S_{ij}}/D_{RR_{ij}}\right)\right) + [S_{LS}/D_{WL} + S_{OS}/D_{WO}]\,. \quad (4.19)$$

Because writing the entire log file and the entire output file are both accounted for, the terms relating to output are combined and represented with

$$[S_O/D_{WO} + S_L/D_{WL}] = [S_{O_{LH}}/D_{WO}] + \left[\left(S_{O_H} + \sum_{i=1}^{N_q}(S_{O_{Bi}} + S_{O_{Ai}})\right)/D_{WO}\right] + [S_{LS}/D_{WL} + S_{OS}/D_{WO}] \quad (4.20)$$

The input files, query and reference, are stored on the same disk for the tests so, occurrences of D_{RQ} and D_{RR} are replaced with D_R. Because the caching of the input, if any, will all be done to the same storage device for all input processed by any one processor, D_{RC_i} is replaced with D_{RC}. By substituting in these replacements equation (4.18) is further simplified to

$$T_T = \left[S_E/D_{RE} + F_{O_{open}} + F_{O_{fopen}}\right] + [S_Q/D_R] + \left[N_q * \sum_{i=1}^{N_R}(S_{R_i}/D_R)\right] + \left[N_q * \left(\sum_{i=1}^{N_R} S_{R_i}\right)/C_p\right] + \left[\sum_{i=1}^{N_q}\left(2 * \sum_{j=1}^{N_{R_i}}\left(S_{S_{ij}}/D_R\right)\right)\right] + [S_L/D_{WL} + S_O/D_{WO}] \quad (4.21)$$

and equation (4.19) to

$$T_T = \left[S_E/D_{RE} + F_{O_{open}} + F_{O_{fopen}}\right] + \left[S_Q + \sum_{i=1}^{N_R}(S_{R_i}/D_R) + (N_q - 1) * \sum_{i=1}^{N_R}(S_{R_i}/D_{RC})\right] + \left[N_q * \left(\sum_{i=1}^{N_R} S_{R_i}\right)/C_p\right] + \left[\sum_{i=1}^{N_q}\left(2 * \sum_{j=1}^{N_{R_i}}\left(S_{S_{ij}}/D_R\right)\right)\right] + \left[S_L/D_{WL} + S_O/D_{WO}\right]. \quad (4.22)$$

Combining terms in equation (4.21) yields

$$T_T = F_{O_{open}} + F_{O_{fopen}} + [S_E/D_{RE}] + [S_Q/D_R] + \left[\left(N_q * \sum_{i=1}^{N_R} S_{R_i}\right) + \left(2 * \sum_{i=1}^{N_q}\sum_{j=1}^{N_{R_i}} S_{S_{ij}}\right)\right]/D_R + \left[N_q * \left(\sum_{i=1}^{N_R} S_{R_i}\right)/C_p\right] + [S_O/D_{WO}] + [S_L/D_{WL}]. \quad (4.23)$$

The same operation on equation (4.22) results in

$$T_T = F_{O_{open}} + F_{O_{fopen}} + [S_E/D_{RE}] + \left[S_Q + \sum_{i=1}^{N_R} S_{R_i} + \left(2 * \sum_{i=1}^{N_q}\sum_{j=1}^{N_{R_i}} S_{S_{ij}}\right)\right]/D_R + \left[(N_q - 1) * \sum_{i=1}^{N_R} S_{R_i}\right]/D_{RC} + \left[N_q * \left(\sum_{i=1}^{N_R} S_{R_i}\right)/C_p\right] + [S_O/D_{WO}] + [S_L/D_{WL}]. \quad (4.24)$$

The computation of the predicted running times will be based on equations (4.23) and (4.24).

4.2 Initial Testing

The FASTA serial version was modified so that it would accept all arguments from the command line. This removed dependencies on environment variables and/or operator input. The reason behind these modifications is that when running tasks in the background using remote shells, there are sometimes conflicts caused because some environment variables may not be set. Also in these cases operator input is impossible. These modifications changed nothing about the actual operation of the program.

FASTA was then changed so that all the output would go to a single file and none of the output would be sent to standard output. The original code always wrote some data to standard output regardless of options selected at execution. This simply involved changing when the output file was opened and which function was used to write the data in question. Minor functions to log execution and time different portions of the program were added and the program was changed so that the user could specify a beginning file offset and an ending file offset for the query file, the reference file(s), or both. Doing this simply extended already existing capabilities by using functions normally limited to the PVM (parallel) version. These modifications allow the serial version of the program to be run as multiple parallel processes with no interprocess communication.

FASTA testing began with a subset of the DNA data bank from GenBank®. This subset is a file of approximately 189 megabytes. It contains 401,113 reference DNA sequences, a total of 144,592,971 characters which require comparison against any query DNA sequence. The query sequences are approximately 300 characters long, chosen randomly from GenBank.

In order to start the multiple applications and specify which portion of the reference data each processed, two driver programs were created. These simply take command line arguments, divide the data among the specified number of processors, and cause a task to spawn on each machine in the test group. NetSolve was used to start the remote tasks. This required that a NetSolve server be started on each machine in the test group but as the servers use very few cpu cycles when idle they can be left to run indefinitely.

The results of the initial testing are from tests using 70 desktop computers in the Computer Science department for application execution. Data was stored for these tests in four locations, three within the department and the fourth in North Carolina. The number of processors was chosen for mainly logistical reasons. It is large enough to include a reasonable population of different architectures and it is small enough so that it can almost always be assured that there are enough machines available for program execution.

Chapter 5

Validation Data Acquisition

Validation of the models in the previous chapter requires data. To obtain that data, a series of tests was performed and the pertinent data were extracted. All of the results here are from tests that were performed one process at a time and so that there is no contention for resources by multiple processes.

5.1 Processors

The test processes were run from different machines belonging to one of the following five sets.

1) Pent3 : a set of Pentium III processor computers running Linux or Windows NT, connected in a cluster and belonging jointly to the Innovative Computation Laboratory (ICL) and Oak Ridge National Labs. Only machines currently running Linux were used.

2) Sparc5 : various Sun SPARCstation 5's through out the department.

3) Ultra1 : the department Cetus lab consisting of Sun Ultra 1's.

4) Ultra2 : the department Gemini lab consisting of Sun Ultra 2's.

5) Ultra5 : the department Hydra lab consisting of Sun Ultra 5's.

The data storage was provided by one of the following machines.

1) Wangzot : a Sun Ultra5 400MHz machine with a Sun StorEdge ®A1000 Array with 6 18GB disk drives configured as RAID-5 used as an NFS file server.

2) Plank : a Sun Sparc2 workstation with an attached 8 GB Sun disk pack used as an NFS file server.

3) DSI-UT and DSI-NC: IBM RS6000 MultiProcessor Internet File Servers with attached Tape robots for a maximum storage capacity of approximately 600 gigabytes. One located in the CS department at UT and one located at Chapel Hill, North Carolina.

5.2 S_E

This was simply a matter of determining the size of the executable file that is loadable using the 'size' utility. The values are in Table 5.1.

5.3 S_Q

Table 5.2 has the size of the files containing the queries.

5.4 S_L

The sizes of the log files are in Table 5.3. The difference in sizes of these files is primarily due to the length of the strings representing the file names and the number of arguments. Those using files stored in IBP generally have the strings representing all the byte array capabilities (not just read or write) which are considerable longer than the paths of the NFS files used. The table has averages of all the log files by test for each set of tests run. The largest log files are for the tests using local IBP servers, which had the log and output files stored in byte arrays as well as the query and reference files. The smallest set of files is for the test using NFS for all the storage because the path names are very short.

5.5 S_O

The size of the individual process output files varies considerably. FASTA assigns a value to every reference sequence compared to the query sequence based on a number of factors. It then selects the, at most, *(*n) reference sequences which have a value within a predetermined range. There may be no reference sequences selected and only selected The smallest output file for the multiple query tests had 5 queries which had no matches and in the largest output file all queries had *(*n) matches. In the computations of predicted running times the average sizes of the output files for single and multiple queries from Table 5.4 were used.

5.6 S_S

The reference sequence files used contained Expressed Sequence Tag (EST) sequences. These normally have fairly short (300-500) base proteins in the sequence and a line of arbitrary length of descriptive data. FASTA re-reads the selected reference sequences twice from the input files, once to compute the alignment with the query sequence and again to

Table 5.1: Executable size read

Machine	O/S	Megabytes
Pent3 550MHz	Linux 2.2.15	0.2598
SPARC5	SunOS 5.7	0.3275
Ultra1	SunOS 5.7	0.3275
Ultra2	SunOS 5.5.1	0.3223
Ultra5	SunOS 5.7	0.3275

Table 5.2: Query file size

Queries	Megabytes
One	0.0004
Seven	0.0029

Table 5.3: Average log file sizes

Test Type	Megabytes
Using NFS storage	0.00059
Using IBP storage	0.00086
Prepositioning in IBP	0.00108

Table 5.4: Average output file sizes

Queries	Megabytes
One	0.018
Seven	0.129

generate the output. It was assumed that the maximum number of sequences (20) would always be selected. Table 5.5 shows the average number of characters for the sequences and descriptions. Table 5.6 contains the 20 sequence total and are the values used in the validation.

5.7 S_R

The sizes of the reference files are in Table 5.7. The EST file on DSI is actually two files but FASTA accesses it as though it were one file. It is split because this version of FASTA, and in fact the Linux operating system used for these tests, has not yet been modified to handle file sizes in excess of $2^{31} - 1$ bytes.

5.8 C

The data in Table 5.8 was extracted from multiple runs of the same tests used for the validation. The FASTA program is instrumented so that the time to read the reference data is excluded from the time to compute the match. It was felt that this was a better approach than developing a modified version of the program that reads some large piece of reference data into memory and then processes a query against that stored data.

5.9 F_O

This is the time it takes to create/open a file for writing, or open it for reading. (See Table 5.9.) Both **open()** and **fopen()** were tested. The logging functions use **open()** and **write()** and the rest of the output is generated using **fopen()** and **fprintf()**.

5.10 D_W

D_W is the apparent disk write access time in megabytes per second. The test results in Table 5.10 represent a single process writing to a file. The tables show times for both write() and fprintf().

5.11 D_{R_X}

Apparent disk read access bandwidth shown in Table 5.11 is in megabytes per second. The test which gathered the data used fgets(), a C standard I/O function, in the same fashion the FASTA program does. The input file consists of the same type of data used in the reference library file, so each line is seldom more than 80 characters long. The query data is in the same format as the reference data so these test results will be used for all read operations.

Table 5.5: Average sequence lengths

File	Avg Seq	Avg Desc
EST human subset	360	129
EST human on DSI	390	131

Table 5.6: 20 sequence totals

File	Megabytes
EST human subset	0.0093
EST human on DSI	0.0080

Table 5.7: Reference file sizes

File	Megabytes
EST human subset	188.991
EST Human on DSI	2050.613

Table 5.8: Speed of comparing queries to sequences

Machine	Avg Megabytes/Sec
Pent3	0.385
Sparc5	0.031
Ultra1	0.067
Ultra2	0.086
Ultra5	0.182

Table 5.9: File open test results in seconds

Machine	Disk	Create		No create
		open()	fopen()	fopen()
Pent3	IBP (Ultra2)	0.025	0.040	0.011
Sparc5	IBP (Ultra2)	0.049	0.081	0.044
Ultra1	IBP Ultra2)	0.032	0.054	0.017
Ultra2	IBP (Ultra2)	0.031	0.041	0.011
Ultra5	IBP (Ultra2)	0.025	0.038	0.017
Pent3	IBP (DSI-NC)	0.376	0.520	0.458
Sparc5	IBP (DSI-NC)	0.398	0.430	0.236
Ultra1	IBP (DSI-NC)	0.376	0.425	0.235
Ultra2	IBP (DSI-NC)	0.382	0.397	0.198
Ultra5	IBP (DSI-NC)	0.374	0.397	0.210
Pent3	IBP (DSI-UT)	0.788	0.793	0.391
Sparc5	IBP (DSI-UT)	0.041	0.066	0.035
Ultra1	IBP (DSI-UT)	0.017	0.033	0.012
Ultra2	IBP (DSI-UT)	0.015	0.027	0.009
Ultra5	IBP (DSI-UT)	0.012	0.026	0.010
Pent3	IBP (local)	0.002	0.004	0.001
Sparc5	IBP (local)	0.103	0.283	0.020
Ultra1	IBP (local)	0.032	0.052	0.004
Ultra2	IBP (local)	0.024	0.049	0.004
Ultra5	IBP (local)	0.005	0.016	0.002
Pent3	NFS (plank)	0.022	0.042	0.007
Sparc5	NFS (plank)	0.043	0.049	0.034
Ultra1	NFS (plank)	0.024	0.042	0.009
Ultra2	NFS (plank)	0.022	0.036	0.007
Ultra5	NFS (plank)	0.022	0.043	0.017
Pent3	NFS (wangzot)	0.004	0.010	0.001
Sparc5	NFS (wangzot)	0.012	0.016	0.013
Ultra1	NFS (wangzot)	0.006	0.011	0.007
Ultra2	NFS (wangzot)	0.006	0.007	0.004
Ultra5	NFS (wangzot)	0.005	0.006	0.010

Table 5.10: Disk write test bandwidth

Machine	Disk	Mbytes/Sec	
		write()	fprintf()
Pent3	IBP (Ultra2)	0.085	0.461
Sparc5	IBP (Ultra2)	0.058	0.249
Ultra1	IBP (Ultra2)	0.095	0.527
Ultra2	IBP (Ultra2)	0.091	0.562
Ultra5	IBP (Ultra2)	0.101	0.659
Pent3	IBP (DSI-UT)	0.092	0.508
Sparc5	IBP (DSI-UT)	0.060	0.270
Ultra1	IBP (DSI-UT)	0.106	0.623
Ultra2	IBP (DSI-UT)	0.104	0.704
Ultra5	IBP (DSI-UT)	0.092	0.717
Pent3	IBP (DSI-NC)	0.003	0.011
Sparc5	IBP (DSI-NC)	0.003	0.011
Ultra1	IBP (DSI-NC)	0.003	0.011
Ultra2	IBP (DSI-NC)	0.003	0.011
Ultra5	IBP (DSI-NC)	0.003	0.011
Pent3	IBP (local)	0.009	0.071
Sparc5	IBP (local)	0.012	0.079
Ultra1	IBP (local)	0.058	0.368
Ultra2	IBP (local)	0.097	0.601
Ultra5	IBP (local)	0.081	0.609
Pent3	NFS (plank)	0.100	0.142
Sparc5	NFS (plank)	0.763	0.734
Ultra1	NFS (plank)	0.941	0.837
Ultra2	NFS (plank)	0.952	0.862
Ultra5	NFS (plank)	0.955	0.888
Pent3	NFS (wangzot)	1.726	1.961
Sparc5	NFS (wangzot)	1.022	0.999
Ultra1	NFS (wangzot)	6.373	5.927
Ultra2	NFS (wangzot)	8.501	7.943
Ultra5	NFS (wangzot)	8.015	7.768

Table 5.11: Disk read test bandwidth

Machine	Disk	Mbytes/Sec
Pent3	IBP (Ultra2)	0.674
Sparc5	IBP (Ultra2)	0.340
Ultra1	IBP (Ultra2)	0.748
Ultra2	IBP (Ultra2)	0.740
Ultra5	IBP (Ultra2)	0.840
Pent3	IBP (DSI-UT)	0.730
Sparc5	IBP (DSI-UT)	0.369
Ultra1	IBP (DSI-UT)	0.829
Ultra2	IBP (DSI-UT)	0.916
Ultra5	IBP (DSI-UT)	0.996
Pent3	IBP (DSI-NC)	0.017
Sparc5	IBP (DSI-NC)	0.017
Ultra1	IBP (DSI-NC)	0.017
Ultra2	IBP (DSI-NC)	0.017
Ultra5	IBP (DSI-NC)	0.017
Pent3	IBP (local)	2.306
Sparc5	IBP (local)	0.096
Ultra1	IBP (local)	0.495
Ultra2	IBP (local)	0.911
Ultra5	IBP (local)	0.644
Pent3	NFS (plank)	0.851
Sparc5	NFS (plank)	0.828
Ultra1	NFS (plank)	0.928
Ultra2	NFS (plank)	0.929
Ultra5	NFS (plank)	0.922
Pent3	NFS (wangzot)	5.622
Sparc5	NFS (wangzot)	1.572
Ultra1	NFS (wangzot)	5.843
Ultra2	NFS (wangzot)	10.452
Ultra5	NFS (wangzot)	9.580

5.12 D_{RC}

In both the NFS and IBP cases, the apparent disk read access time when caching is in use is affected by a number of factors. One, which cannot be accurately modeled, is the current usage of the local disk by processes. This includes disk swap space. However, for the general case, testing of the data access time during normal working hours and using a mix of machines gives a good approximation of the validation testing conditions.

Another factor which has perhaps more impact is the amount of swap space for data caching. When using NFS this is be approximately one half of the available swap space. This is tunable by the system administrators but is usually not changed once the initial installation is complete. IBP_STDIO caching is currently configured so that the amount of cache space is a minimum of 1 megabyte and a maximum of 100 megabytes. If that minimum amount is not available in either volatile or stable storage, no caching is done. Once a storage area is chosen and a maximum size determined, neither is changed for the life of that process. Both NFS and IBP use their cache space in a LRU manner, so that if the amount being cached exceeds the space available and the data is reread from the beginning each time, the access time devolves to that of reading initially from the remote storage. As caching for IBP is equivalent to reading the data from the original file on the first pass followed by reading the data from local IBP storage on all subsequent reads, the data necessary to compute this is in Table 5.11.

5.13 Example Calculation

Assume that a test process is running on a Sparc 5 machine. It is processing one query against the EST human subset reference file. The query file, the reference file and the output, both log file and FASTA results, are using IBP on the DSI server at UT. The executable is residing in NFS on wangzot. Note that in this situation there is only one reference file and one query. Using equation 4.23 and values from the tables we have

$$\begin{aligned} T_T = 0.041sec + 0.066sec + [0.3275mb/1.572mb/sec] + \\ [0.0004mb/0.369mb/sec] + \\ [(1 * 188.991mb) + (2 * 0.0093mb)] /0.369mb/sec + \\ [1 * 188.991mb/0.031mb/sec] + [0.018mb/0.270mb/sec] \\ + [0.00086mb/0.060mb/sec] \end{aligned} \quad (5.1)$$

$$\begin{aligned} T_T = 0.041sec + 0.066sec + 0.208sec + 0.001sec + 512.221sec + \\ 6096.484sec + 0.067sec + 0.014sec \end{aligned} \quad (5.2)$$

$$T_T = 6609.102sec \quad (5.3)$$

Chapter 6

Testing for Validation

This chapter contains the results of putting the data and equations from the previous sections to work. The intent is to see if the predictive model is reasonable. The best way to represent the comparison between the predicted and actual results is graphically. The two following sections document that comparison for the case of using a single query sequence and the case of using 7 query sequences. In each case, 70 processors were used with the same 189 megabyte reference file. The test data was collected for five different sets of machines (list in Table 6.1) and the results for each set are combined. The different test sub-cases correspond to the locations of the reference data and the output files.

6.1 Single query results

The first eight figures in this chapter are graphs of the results of using a single query. The data storage areas were varied to more completely test the system and the model. The primary storage server was IBP. However, the local NFS file-system was used for storage during preliminary testing to validate the test programs and the program model. It is not the intent to show that IBP can compete with NFS in a local area network, this information is only included for completeness. Regardless, when the input and output files are stored in local NFS the model does a fair job of predicting the running time of the processes, see

Table 6.1: Testing Machines

Name	Processor	Speed	O/S	Number
Pent3	Dual Pentium III	550 MHz	RedHat Linux 7.1	8
Sparc5	Sun Sparc	85 MHz	Solaris 2	20
Ultra1	Sun Sparcv9	143 MHz	Solaris 7	30
Ultra2	Dual Sun Sparcv9	167 MHz	Solaris 2	12
Ultra5	Sun Sparcv9	400 MHz	Solaris 7	30

Figure 6.1.

Next the input data was pre-positioned on each processing machine on its local disk using an IBP server (Figure 6.2). In this situation IBP allows access to data storage resources that may not normally be available to an external process. The data for this test is divided between the processors and stored prior to the application starting. Output data is also written to this local storage and then moved by the driver program after the job completes. This is done only for ease of data collection and is not strictly necessary. Both graphs show that the computation time (T4) is the dominant component of overhead.

No buffering was used for the results in Figures 6.3 - 6.5. These tests simply used the IBP storage as if it were a standard file system. No read buffering is involved because the IBP_STDIO library will only do that for byte arrays if an IBP server exists on the local host and no such server was allowed to exist.

The IBP buffering tests of Figures 6.6 - 6.8 use a local IBP server accessing the local disk to buffer the read of data from the reference files. The buffering is transparent to the process and is managed by the IBP_STDIO library. All that is required is that a local IBP server be started (by the driver program) before the application. The storage for the local IBP servers is **/var/tmp** instead of **/tmp** because **/var/tmp** is normally the larger of the two.

The Figures 6.6 and 6.8 show that the actual total execution time is greater than the predicted results, primarily in the time required to read the reference data. The reason is even though the processes are contending for storage access, the IBP_STDIO library uses a buffer of one megabyte. The functions (such as fread()) that the process uses to read the reference data have a buffer size of one (1) kilobyte. This means that for the size of data an individual machine reads in this test (2.7 megabytes) there are approximately 2765 reads from the remote storage without buffering and 3 remote reads for the case with buffering. Then there is an additional read of the data from the local disk. This means that even though there is contention for disk and network access in all the remote access tests, it is relatively speaking a smaller proportion of the time to read the data in the buffering tests.

Figure 6.7 has the opposite result. This is due to the manner in which the data is transferred from remote storage to the buffer. In determining the data transfer rates (the table values of chapter 5) the amount requested per read was one kilobyte. For each of these reads a round-trip network connection must be established then the data transferred. Because of the large number of reads the overhead of establishing the network connection (latency) is quite large. The test version of IBP uses a maximum data transfer size of one megabyte. This virtually assures that the majority of network traffic will be transfers of the largest possible size. In the case of the connection to a remote site having a large network latency this can result in a significant increase in overall transfer rate. This increase in transfer rate is also occurring in the other buffering tests, it is just not as significant a factor.

In each set of test results there are anomalies. For instance in Figure 6.1 the result for the actual computation on the Sparc5 machines is much longer than the predicted time.

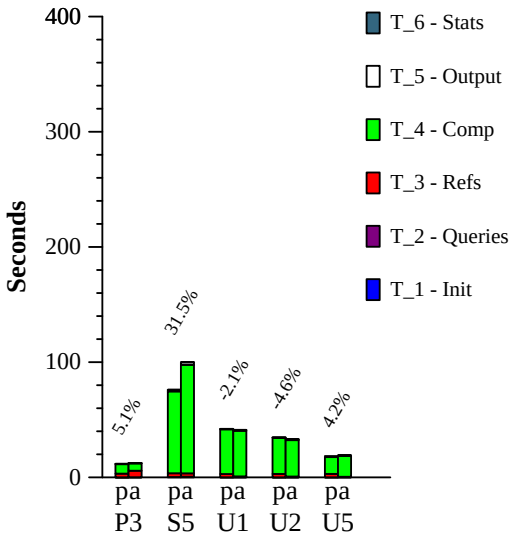

Figure 6.1: One Query, Using NFS.

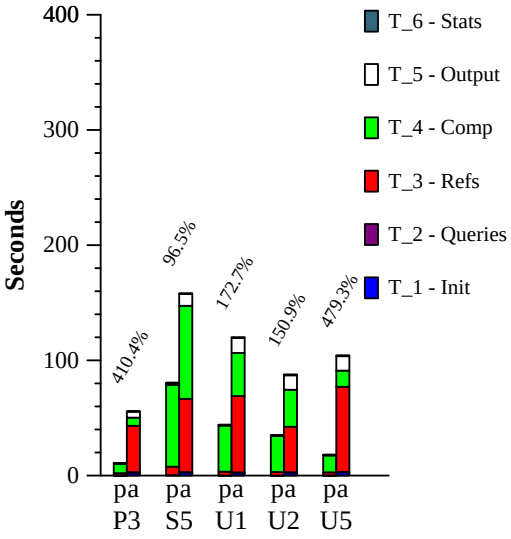

Figure 6.3: One Query, No Buffering Using DSI-UT

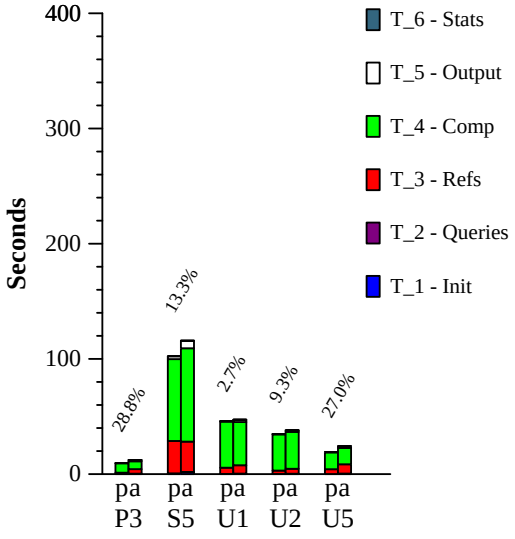

Figure 6.2: One Query, Using Local Storage (IBP).

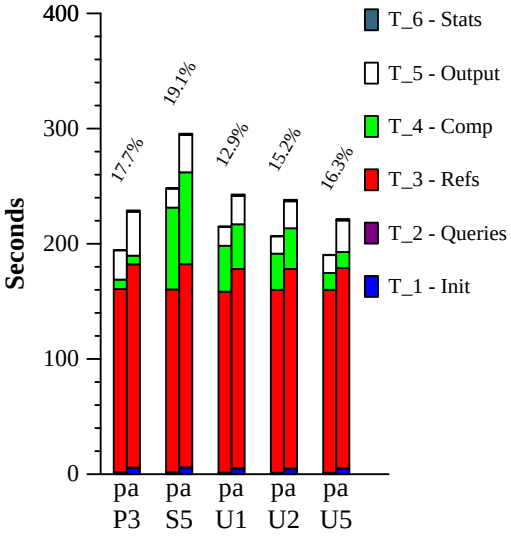

Figure 6.4: One Query, No Buffering Using DSI-NC.

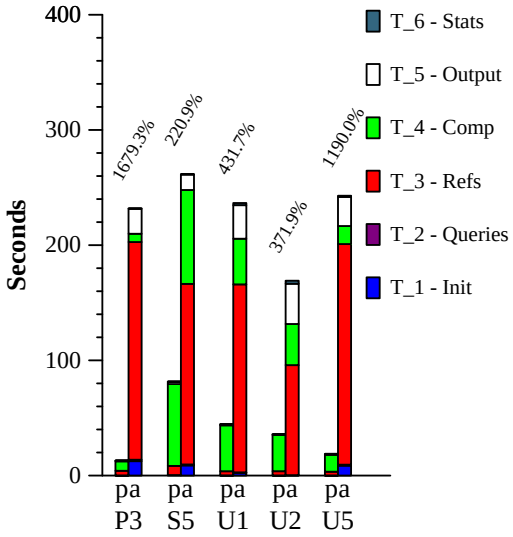

Figure 6.5: One Query, No Buffering Using Ultra2.

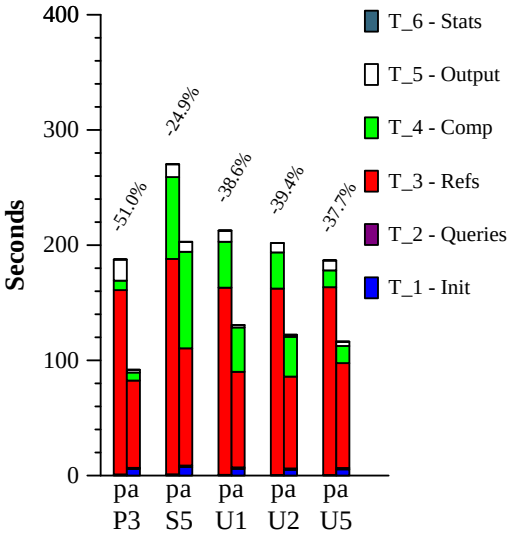

Figure 6.7: One Query, Buffering Using DSI-NC.

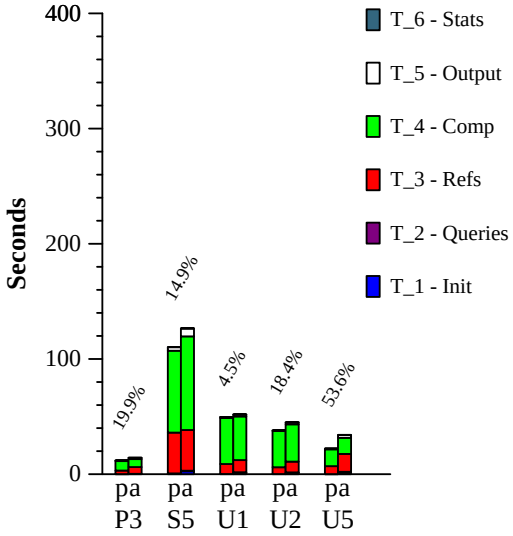

Figure 6.6: One Query, Buffering Using DSI-UT.

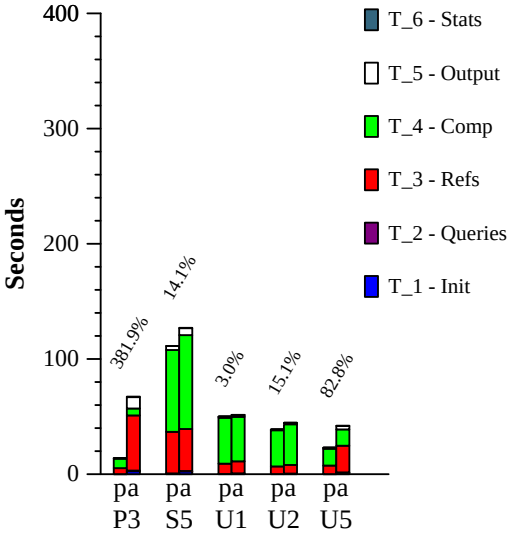

Figure 6.8: One Query, Buffering Using Ultra2.

Similarly in the rest of the graphs there is some component that is noticeably different from the predicted values. This is typically a by-product of the fact that these machines and their networks are in active use by others. Also the model sets (Table 6.1) are relatively small, the largest being the Ultra5 and Ultra1 machines. The others have 20 or fewer members, not all of which were available for any given test. This means that as the number of tests for any one configuration was small the impact of even a single machine being heavily loaded can be significant.

Also note that even though the Ultra2 machines are physically part of the same LAN as the rest of the actual processing machines, when the reference data is stored on one of them, the differences between predicted and actual access times are much larger than would seem to be warranted. This is ascribed to the configuration of the switches which connect this cluster of workstations to the rest of the network. They were originally installed to research different networking strategies and methods, particularly when performing parallel processing and were not intended to be used other than as a cluster. On an individual basis they respond as might be expected of any other workstation but the cluster connection to the LAN is the bottleneck when large numbers of external accesses are made.

6.2 Multiple query results

Multiple query results are given in the same order as the single query results. These comparisons are of the predicted values to the actual values for the cases where multiple (seven) queries were used in the testing. Figures 6.9 through 6.16 are the graphs for these results.

Again we see that for the NFS and Local Storage tests the predicted results and the actual results are very close. From this it is inferred that the model is accurate. The tests that use IBP buffering also have results that very closely match the model, more so even than the single query tests.

On the other hand, the tests that use no buffering and only read directly from IBP show that some factor has not been correctly accounted for. Specifically, this is the impact of multiple processes reading from a single IBP server. This is not a uniform factor. Each IBP storage array is affected by such things as the load on the local processor running the server, the disk access semantics of the particular hardware architecture, the amount of local disk cache available on that processor, how network connections are handled by the O/S kernel, the network speed and type that connects that server to the client.

As can be seen in the multiple query tests the multiple-access effect has significant impact on the predictability of the process running time. The use of IBP buffering (and we assume any local data buffering) can minimize this impact in the long run. The multiple query tests without buffering that are accessing DSI-NC seem to indicate that the multiple-access effect has small impact. Actually, the latency of the connection is so large that it masks that impact.

A comparison of the single and multiple query results illustrates the multiple-access effect. Except for the case of DSI-NC, the single query times are not much different than

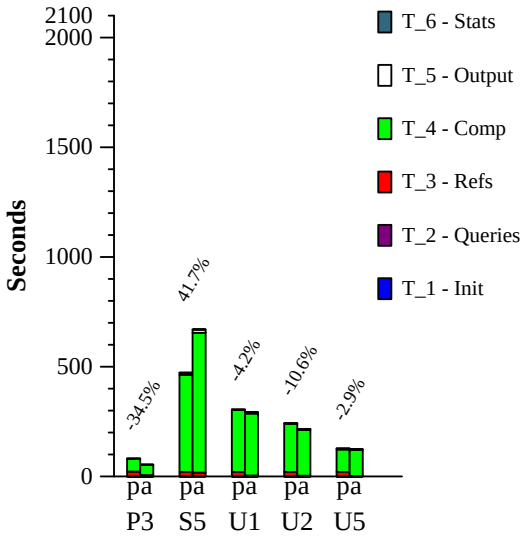

Figure 6.9: Seven Queries, Using NFS.

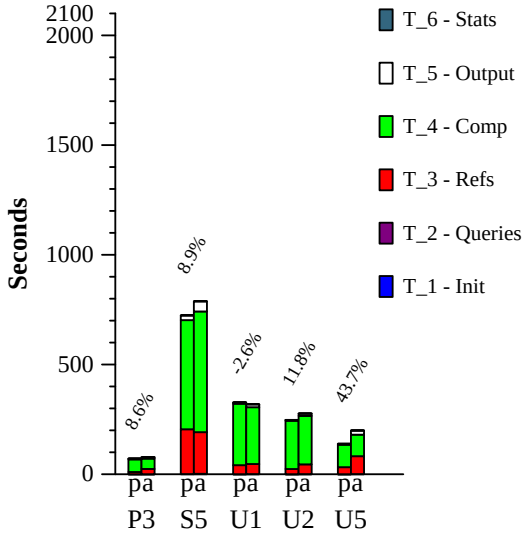

Figure 6.11: Seven Queries, Buffering Using DSI-UT.

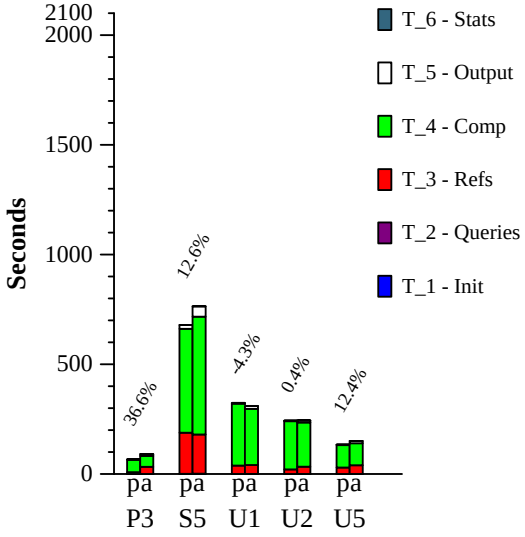

Figure 6.10: Seven Queries, Using Local Storage.

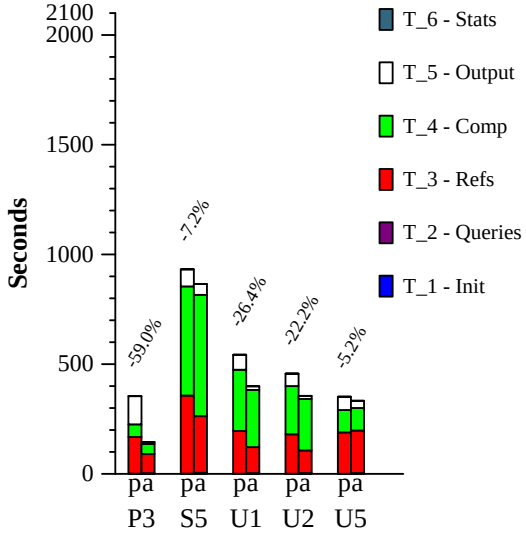

Figure 6.12: Seven Queries, Buffering Using DSI-NC.

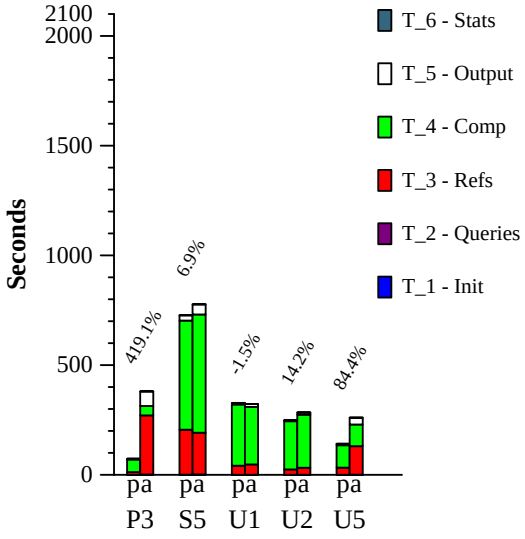

Figure 6.13: Seven Queries, Buffering Using Ultra2.

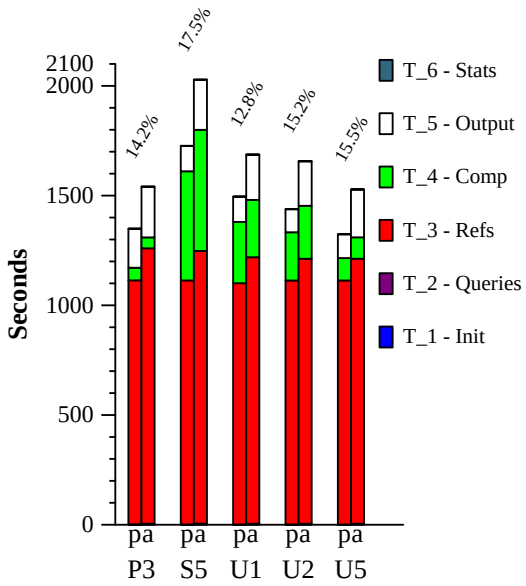

Figure 6.15: Seven Queries, No Buffering Using DSI-NC.

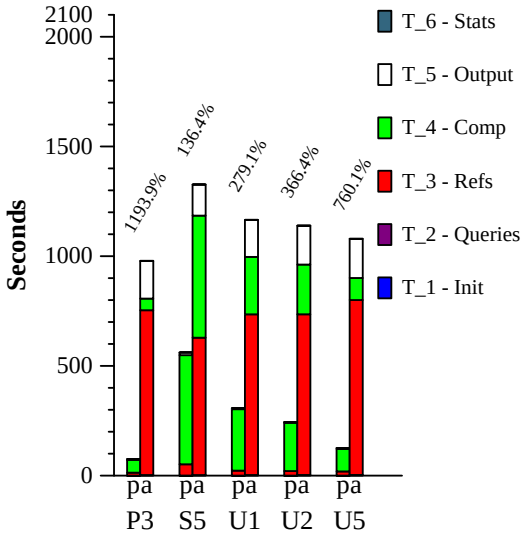

Figure 6.14: Seven Queries, No Buffering Using DSI-UT.

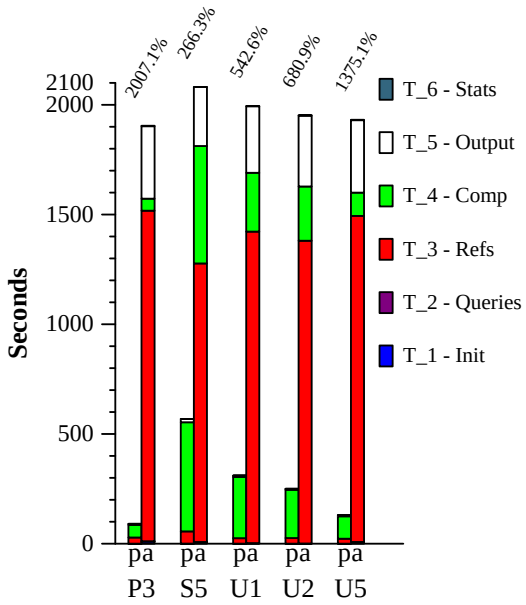

Figure 6.16: Seven Queries, No Buffering Using Ultra2.

the unbuffered results. This is because the buffering only has an impact on those reads that occur after the initial access to some block of data. In the multiple query case, all accesses to the data after the first query are from local disk, so that the longer the process runs (the more queries that are processed at a single time), the less impact the multiple-access effect has.

Chapter 7

Continuing Experiments Using Large Files

7.1 Introduction

Preliminary testing showed that the original hypothesis appears accurate and that the model seems to be consistent. Testing began on a large input reference. The original test reference file was a subset of the EST human database from GenBank®. This file was 189 megabytes and contained 401,113 DNA sequences. As seen in the previous chapter, running times for processing queries against this reference file are reasonable and in some sense predictable. The large input we use in this chapter is the entire EST human DNA sequence set from GenBank ®stored in two files residing on DSI file servers. The reference was divided into two files because of the 2 gigabyte limit currently imposed on file size by some compilers and operating systems. The total size of this reference EST database is 2.003 gigabytes and it contains 4,084,858 DNA sequences.

7.2 Hardware

Between the original tests to validate the operation of the software systems and the model and these large file tests, the department's hardware experienced significant changes. Operating systems were upgraded, older hardware was phased out and newer machines were installed. The new machine sets are reduced to four: Ultra1, Ultra5, Sparc5 and Pent3. Their characteristics are in Table 7.1.

These changes also required re-examination of the original data obtained for chapter 5. Most remained the same but the file access data showed some significant changes, generally due to network reconfigurations and improvements in physical connections. Additionally, some of the preliminary testing was performed using reference data stored on an I2-DSI server located at Chapel Hill, North Carolina which is no longer available due to hardware failures. The new reference data is contained in Tables 7.2, 7.3, and 7.4.

Table 7.1: New Testing Machines

Name	Processor	Speed	O/S	Number
Pent3	Dual Pentium III	550 MHz	RedHat Linux 7.1	8
Sparc5	Sun Sparc	85 MHz	Solaris 8	20
Ultra1	Sun Sparcv9	143 MHz	Solaris 8	30
Ultra5	Sun Sparcv9	400 MHz	Solaris 8	30

Table 7.2: File open test results in seconds

Machine	Disk	Create		No create
		open()	fopen()	fopen()
Pent3	IBP (DSI-NC)	0.015	0.018	0.322
Sparc5	IBP (DSI-NC)	0.040	0.057	0.131
Ultra1	IBP (DSI-NC)	0.029	0.059	0.148
Ultra5	IBP (DSI-NC)	0.018	0.029	0.138
Pent3	IBP (DSI-UT)	0.011	0.012	0.009
Sparc5	IBP (DSI-UT)	0.046	0.105	0.031
Ultra1	IBP (DSI-UT)	0.037	0.068	0.030
Ultra5	IBP (DSI-UT)	0.015	0.029	0.016
Pent3	IBP (sinrg)	0.013	0.258	0.490
Sparc5	IBP (sinrg)	0.050	0.154	1.000
Ultra1	IBP (sinrg)	0.033	0.075	0.782
Ultra5	IBP (sinrg)	0.015	0.030	0.391
Pent3	IBP (local)	0.002	0.012	0.001
Sparc5	IBP (local)	0.069	0.204	0.017
Ultra1	IBP (local)	0.041	0.095	0.009
Ultra5	IBP (local)	0.008	0.026	0.003
Pent3	NFS (wangzot)	0.004	0.003	0.003
Sparc5	NFS (wangzot)	0.007	0.010	0.007
Ultra1	NFS (wangzot)	0.005	0.010	0.004
Ultra5	NFS (wangzot)	0.003	0.004	0.003

Table 7.3: Disk read test bandwidth

Machine	Disk	Mbytes/Sec
Pent3	IBP (DSI-UT)	1.033
Sparc5	IBP (DSI-UT)	0.428
Ultra1	IBP (DSI-UT)	0.572
Ultra5	IBP (DSI-UT)	0.954
Pent3	IBP (DSI-NC)	0.026
Sparc5	IBP (DSI-NC)	0.026
Ultra1	IBP (DSI-NC)	0.026
Ultra5	IBP (DSI-NC)	0.027
Pent3	IBP (sinrg)	0.504
Sparc5	IBP (sinrg)	0.403
Ultra1	IBP (sinrg)	0.590
Ultra5	IBP (sinrg)	1.051
Pent3	IBP (local)	2.306
Sparc5	IBP (local)	0.088
Ultra1	IBP (local)	0.228
Ultra5	IBP (local)	0.481
Pent3	NFS (wangzot)	21.089
Sparc5	NFS (wangzot)	1.986
Ultra1	NFS (wangzot)	3.660
Ultra5	NFS (wangzot)	14.188

Table 7.4: Disk write test bandwidth

Machine	Disk	Mbytes/Sec	
		write()	fprintf()
Pent3	IBP (DSI-UT)	0.157	0.869
Sparc5	IBP (DSI-UT)	0.027	0.165
Ultra1	IBP (DSI-UT)	0.051	0.293
Ultra5	IBP (DSI-UT)	0.029	0.565
Pent3	IBP (DSI-NC)	0.124	0.709
Sparc5	IBP (DSI-NC)	0.070	0.306
Ultra1	IBP (DSI-NC)	0.077	0.372
Ultra5	IBP (DSI-NC)	0.120	0.688
Pent3	IBP (sinrg)	0.055	0.227
Sparc5	IBP (sinrg)	0.026	0.147
Ultra1	IBP (sinrg)	0.051	0.283
Ultra5	IBP (sinrg)	0.114	0.665
Pent3	IBP (local)	0.313	2.959
Sparc5	IBP (local)	0.012	0.081
Ultra1	IBP (local)	0.029	0.191
Ultra5	IBP (local)	0.083	0.533
Pent3	NFS (wangzot)	7.661	9.774
Sparc5	NFS (wangzot)	1.147	1.146
Ultra1	NFS (wangzot)	2.029	2.046
Ultra5	NFS (wangzot)	9.734	10.114

7.3 Initial Tests

The testing with reference files in excess of 2 gigabytes was begun with the EST reference data stored on the DSI server at UT Knoxville. As in the preliminary testing, there are three basic test configurations and all tests were performed using 70 machines located within the Computer Science Department. The tests are:

1) all input reference and query data and log and output files are located on the same remote server and accessed using IBP

2) all input reference and query data are located on the remote server, data reads are buffered on the local machine using an IBP server much like NFS file buffering, and log and output files are also written to the local IBP server

3) all input reference and query data are prepositioned on the local machine using IBP and the log and output files are written to the local IBP server

The single query results are again given first. Figure 7.1, "Using Local Storage (IBP)", serves to support the contention that the model of the computation is basically correct given that the various factors involved in the computation can be accurately measured. The Pent3 results when buffering is used, Figure 7.2, seem rather anomalous. The predicted time in this test sequence is much greater than the actual time.

These machines are always the first to start. The reason for this is the driver program which accesses a list of potential server machines from an agent. Along with the names of the servers the driver receives integer and floating point benchmark values for each machine. Because the FASTA3 application performs basic integer comparisons, the driver sorts the machines using the integer benchmark value, fastest first. This should mean that those processors which run first, if they are fast enough, can access their data without being as seriously impacted by the effects of multiple-access as the other processors. To test this theory, the order of execution is reversed, the fastest 70 processors are still selected out of the available servers but the slowest of these is started first and the fastest last. Figure 7.3 shows that even though there is a small (approximately 6%) increase in read time for the Sparc5, Ultra1 and Ultra5 machines, the Pent3 machines have something on the order of a 430% increase in the time to read the reference data.

The remaining two single query graphs, Figures 7.4 and 7.5, indicate that in the case where the data access is directly from the remote storage, the order of execution has little effect. These graphs do show that the effect of multiple-access to a storage area is even more pronounced for long running processes. The multiple query (seven) results in Figures 7.6 through 7.10 show results similar to the single query tests.

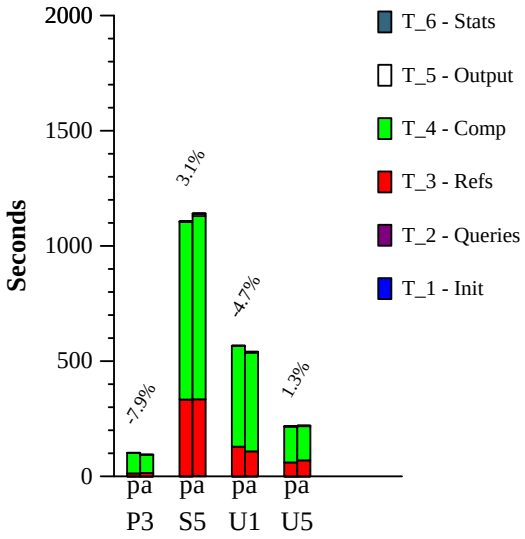

Figure 7.1: One Query, Large Files, Using Local Storage (IBP).

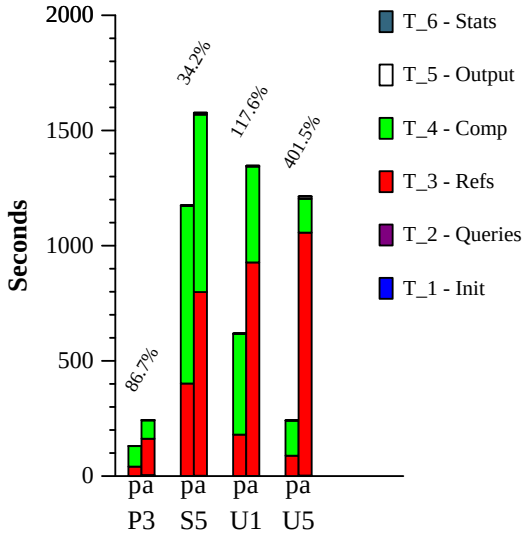

Figure 7.3: One Query, Large Files, Reversed Order, Buffering Using DSI-UT.

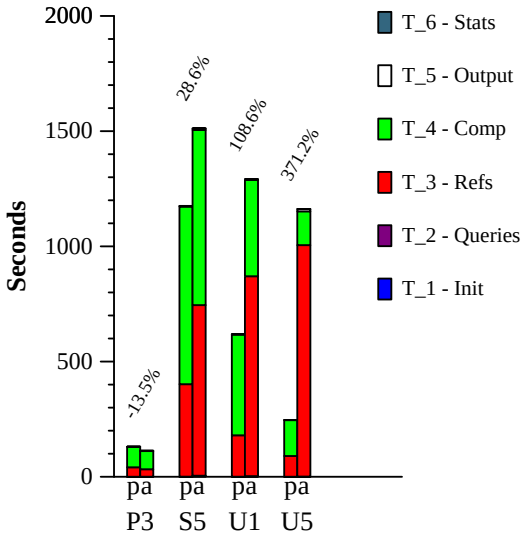

Figure 7.2: One Query, Large Files, Buffering Using DSI-UT.

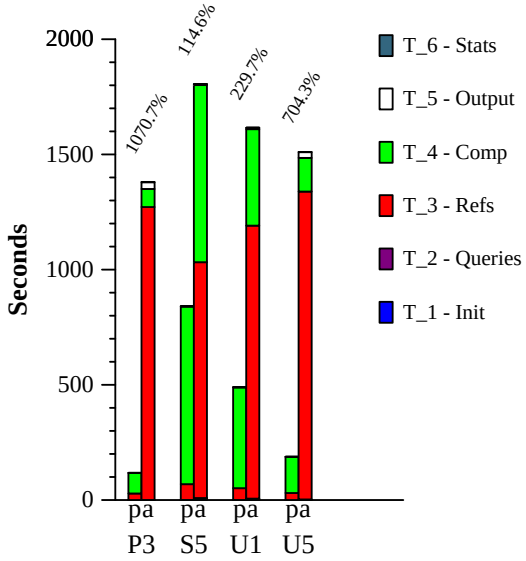

Figure 7.4: One Query, Large Files, No Buffering Using DSI-UT.

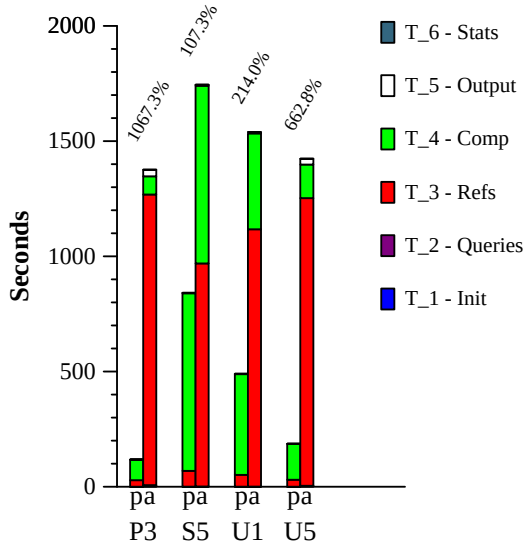

Figure 7.5: One Query, Large Files, Reversed Order, No Buffering Using DSI-UT.

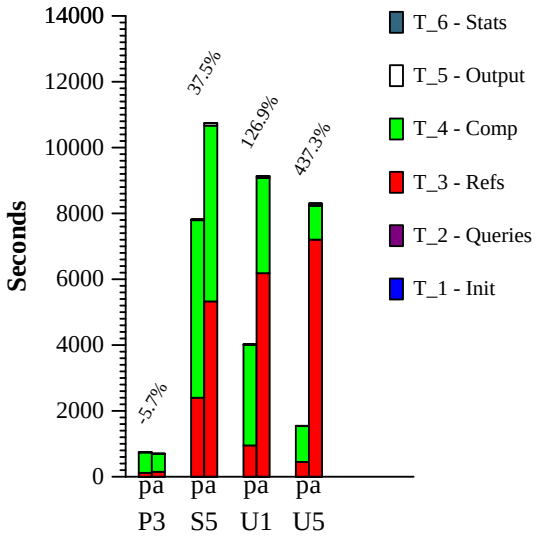

Figure 7.7: Seven Queries, Large Files, Buffering Using DSI-UT.

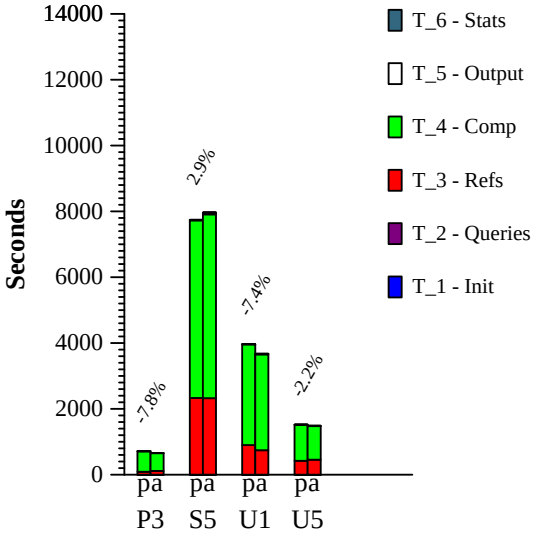

Figure 7.6: Seven Queries, Large Files, Using Local Storage.

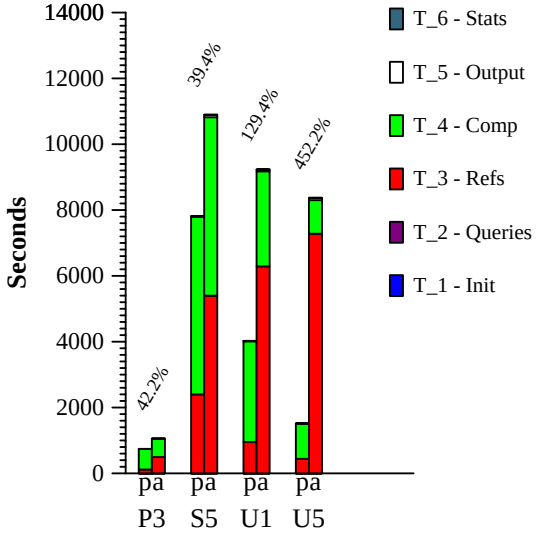

Figure 7.8: Seven Queries, Large Files, Reversed order, Buffering Using DSI-UT.

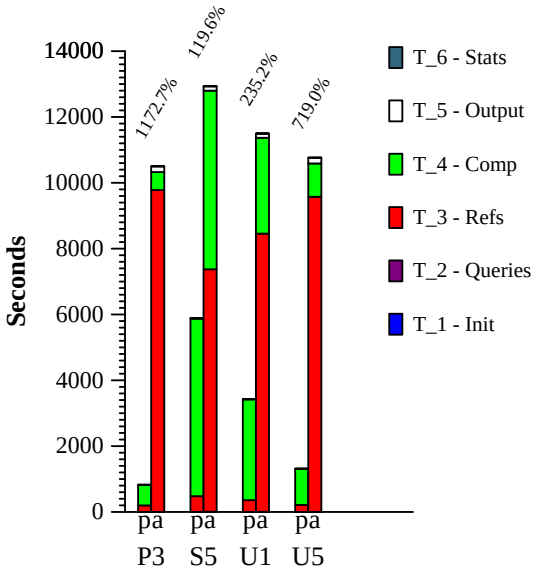

Figure 7.9: Seven Queries, Large Files, No Buffering Using DSI-UT.

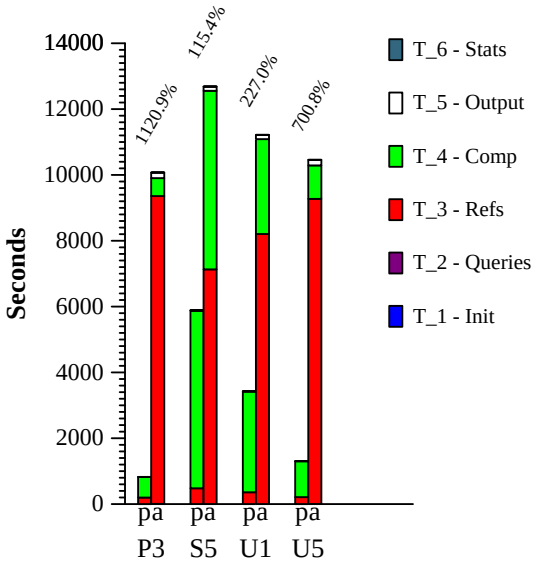

Figure 7.10: Seven Queries, Large Files, Reversed order, No Buffering Using DSI-UT.

7.4 Distributed Input Tests

The question now arises: can reducing the number of processes accessing the data significantly improve the predictability of the running time? To test this, the large input files have been divided into ten approximately equally sized files and each placed on a separate machine, using IBP. One problem here is finding sufficient storage space for the data. Even though many of the desktop machines being used have large local disks with partitions in excess of 2 gigabytes, these partitions are often administratively restricted to specific users or uses or are shared by many users. This means that it can be difficult for an arbitrary user to store files of 200 megabytes or more on multiple desktop units.

Storage for the files for these tests was provided by a cluster of Sun Enterprise 220R servers which are dual 450MHz UltraSparc-II, 64-bit processors with 512 megabytes of ram and four megabytes of cache, part of the Scalable Intracampus Research Grid (SInRG) [20] project. The standardization tests have been performed with these machines and those results are included in Tables 7.2, 7.3, and 7.4. The discovery of storage resources will not be dynamic but will follow the pattern already established, file names or IBP capabilities are passed to the driver program via command line arguments.

Any more testing using data prepositioned on the processing machines would serve no purpose. It has already been seen that this case is predictable. Therefore the tests using the SInRG machines is limited to the buffered and unbuffered cases.

The improvement in overall execution time seen in the distributed input tests is significant. Examination of Figures 7.11 through 7.18 and comparison with their counter-parts from Figures 7.2 through 7.5 and 7.7 through 7.10 shows that by distributing the data storage we have significantly reduced the data access time. This corresponds to a reduction in the multiple-access effect as evinced by fact that the actual times more closely approach the predicted times in these tests.

Another fact that comes from the examination is that neither the fastest nor the slowest processor sets were significantly affected by reversing the order in which the processors were used. The Ultra1 and Ultra5 sets do show a marked change. With 70 processors and 10 separate data storage machines, 7 processors now access each data storage area. In general there are 7 Pent3 machines per test so they either access the first machine or the last. The same basic thing holds for the Sparc5, only with approximately 14 machines. That means the Ultra1 and Ultra5 machines are contending for the remainder of the storage areas. It appears that the reverse ordering in this case improves the access patterns by reducing conflicts for those machines.

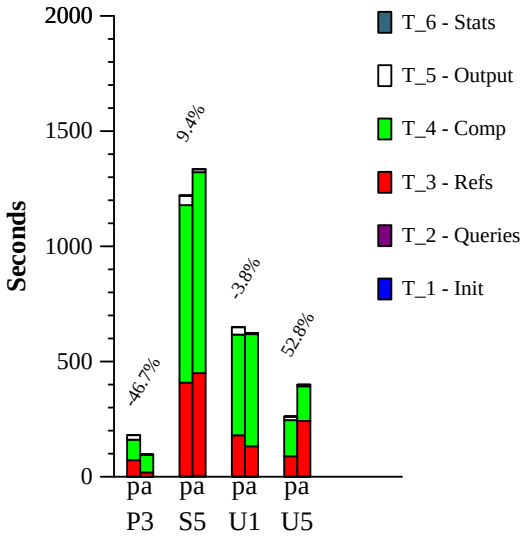

Figure 7.11: One Query, Large Files, Buffering Using SInRG.

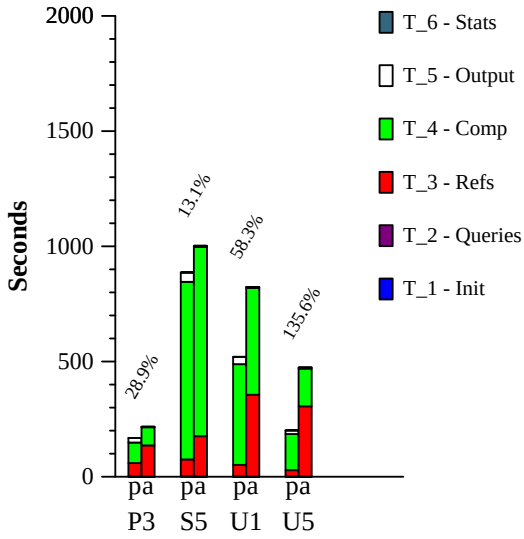

Figure 7.13: One Query, Large Files, No Buffering Using SInRG.

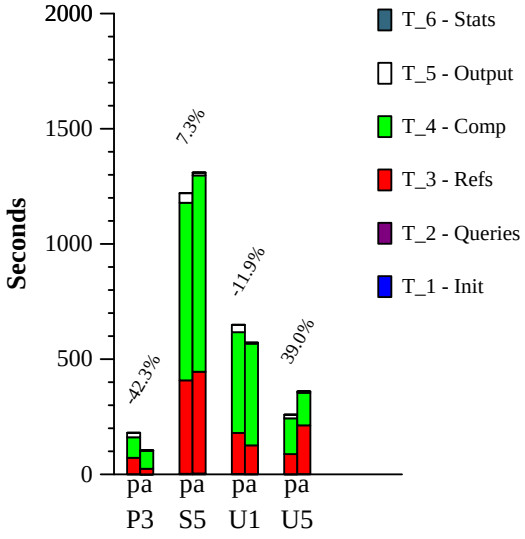

Figure 7.12: One Query, Large Files, Reversed Order, Buffering Using SInRG.

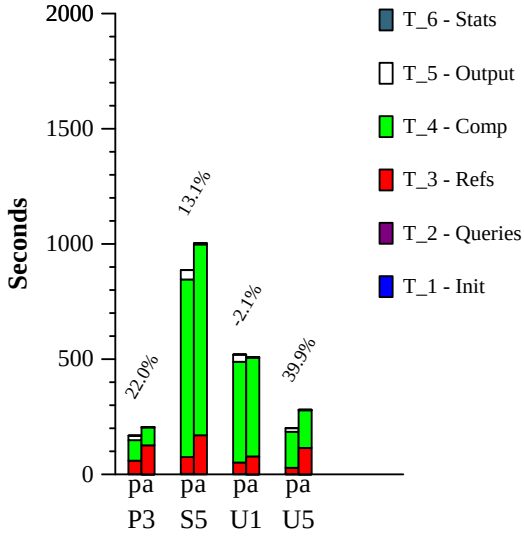

Figure 7.14: One Query, Large Files, Reversed Order, No Buffering Using SInRG.

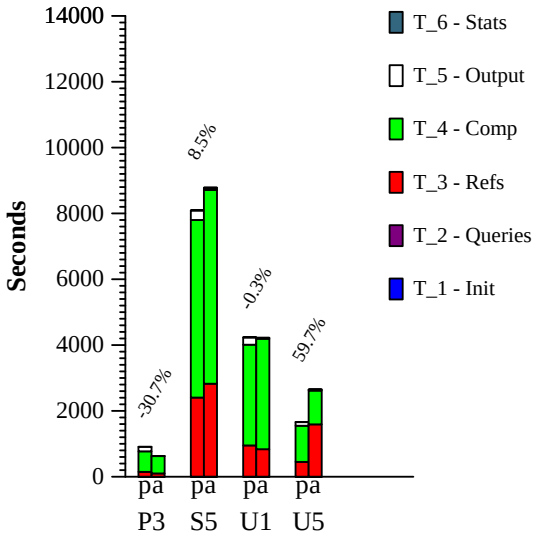

Figure 7.15: Seven Queries, Large Files, Buffering Using SInRG.

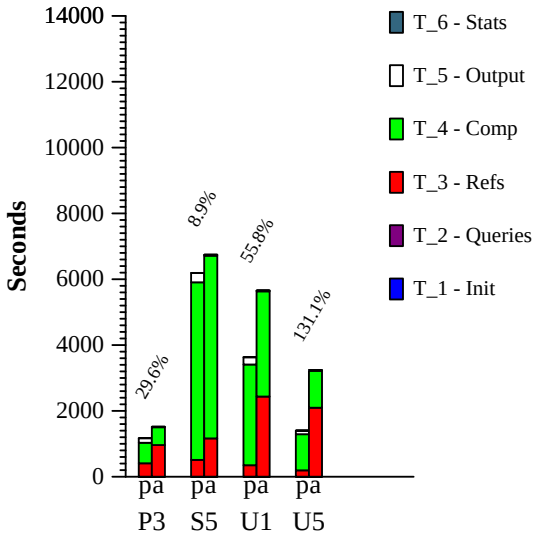

Figure 7.17: Seven Queries, Large Files, No Buffering Using SInRG.

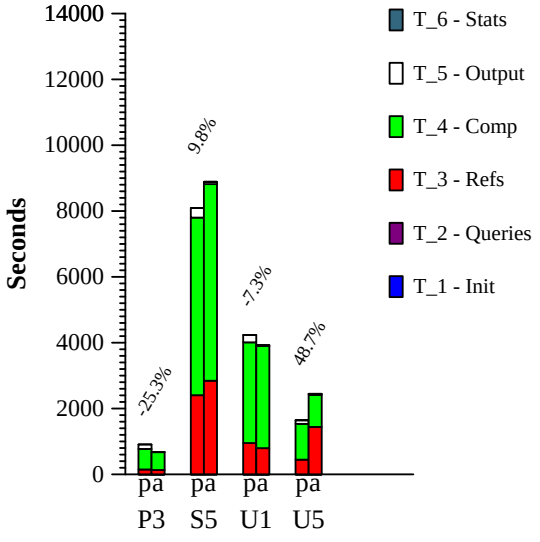

Figure 7.16: Seven Queries, Large Files, Reversed Order, Buffering Using SInRG.

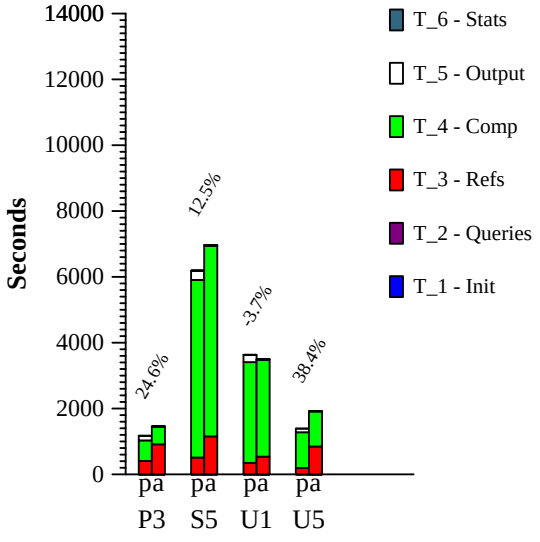

Figure 7.18: Seven Queries, Large Files, Reversed Order, No Buffering Using SInRG.

Chapter 8

Conclusions and Directions

8.1 Conclusions

This research has shown that a legacy serial application can easily be made to function in a wide-area setting. Further it has shown that such applications can be modeled so that the application's performance can be predicted with some accuracy. This can be of great benefit to researchers or others who do not have locally available resources for storage of large data sets or computation but do have access to sufficient wide-area resources.

The research has also shown the usefulness of IBP as a medium for data access in wide-area networks. For such diverse, heterogeneous systems it can be used to provide a relatively simple and predictable "file system". By combining IBP with the C standard I/O library we now have a method by which legacy code can be easily modified to operate in the wide-area environment. This can result in the saving of many man-hours over trying to rewrite such code.

Perhaps as importantly, it has shown that restrictions such as file size limits and disk storage space can be overcome by the use of IBP. The data can be easily divided up and stored in separate locations even though this adds somewhat to the complexity of the data access. With IBP there is now a consistent way to access such data stored on practically any machine that can be accessed through a network connection.

The research has also provided insight into predicting data access times in wide-area program networks. Modeling of the application is not necessarily simple, but even in the case of legacy programs it can be done fairly accurately. In the simplest uses of the model, such as those where the data is prepositioned on the local machine, data access prediction based on basic bandwidth tests is reasonable. But in situations where multiple processes are attempting access of the same data at arbitrarily overlapping intervals, this is not enough.

For any wide-area network there are a large number of dynamic components, such as switches and routers, that contribute to the overhead of the data access. In addition there is the contention for the network medium by outside processes. And, as has been seen, something as simple as the order of execution of the wide-area programs can markedly impact overall performance. The point being that static predictions can only supply a general

idea of the running time of a process in this type of environment. But such predictions and initial testing based on them can supply information on how to pre-configure the system for good average performance in the long run.

8.2 Future Directions

A major drawback to the system as tested for this research is its static nature. All measurements and data storage decisions are made before any test is begun and no attempt is made to update information after each test. This naturally points to research into dynamic performance data acquisition for use in the prediction model. Information gained through this could give some better idea of the overall performance of networks under varying load conditions.

Another research avenue is that of dynamic selection of storage resources. The Logistical Backbone (L-Bone) component of the Network Storage Stack [26] is being developed to provide dynamic information on availability of IBP servers and corresponding live network bandwidth. With this more accurate prediction of the data access time, total execution time predictions are improved. This improvement would lead to the ability to make a choice of storage location for a particular computational resource to minimize overall running time. This would in turn lead to the ability to make well-informed load balancing decisions, i.e. the faster processors should reasonably be asked to process more data than the slower processors.

Fault tolerance in the system as tested was not considered. In the tested system the results of each processor were independent of all other results. With the ability to dynamically select duplicate data storage locations, failure of one portion of the computation due to inaccessibility of data can be easily rectified by restarting that portion of the computation using a different data location. Information to support this dynamic selection process could be made available through resources such as the exNode [4] component of the Network Storage Stack.

Bibliography

Bibliography

[1] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Extending the operating system at the user level: The Ufo global file system. *Proceedings of the USENIX Annual Technical Comference*, pages 77–90, Jan 1997.

[2] W. Allcock, J. Bresnahan, I. Foster, L. Liming, J. Link, and P. Plaszczac. Gridftp update january 2002. Technical report, The Globus Project, Jan 2002.

[3] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, Feb 1996.

[4] S. Atchley, S. Soltesz, J. S. Plank, M. Beck, and T. Moore. Fault-tolerance in the network storage stack. In *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, Ft. Lauderdale, FL, April 2002.

[5] M. Beck and T. Moore. The Internet2 Distributed Storage Infrastructure Project: An architecture for internet content channels. *Computer Networks and ISDN Systems*, 30(22–23):2141–2148, Nov 1998.

[6] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A data movement and access service for wide area computing systems. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, May 1999.

[7] R. L. Burden, J. D. Faires, and A. C. Reynolds. *Numerical Analysis*. PWS Publishers, Boston, second edition, 1981.

[8] H. Casanova and J. Dongarra. NetSolve: A network server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.

[9] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. Submitted to Netstore '99, Oct 1999.

[10] A. Choudhary, M. Kandemir, H. Nagesh, J. No, X. Shen, V. Taylor, S. More, and R. Thakur. Data management for large-scale scientific computations in high performance distributed systems. In *Proceedings of the 8th IEEE International Symposium*

on High Performance Distributed Computing, pages 263–274. IEEE Computer Society Press, Aug 1999.

[11] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACMSymposium on Operating System Princples (SOSP '01)*, Chateau Lake Louise, Banff, Canada, Oct 2001.

[12] Message Passing Interface Forum. MPI: A message passing interface standard. Version 1.1, Jun 1995.

[13] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[14] I. Foster, D. Kohr, R. Krishnaiyer, and J. Mogill. Remote I/O: Fast access to distant storage. In *Proceedings of the Fourth Workshop on I/O in Parallel and Distributed Systems*, pages 14–25, 1997.

[15] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems*, 20(1):1–24, Feb 2002.

[16] A. S. Grimshaw, W. W. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds. Legion: The next logical step toward a nationwide virtual computer. Department of Computer Science Technical Report CS-94-21, University of Virginia, Jun 1994.

[17] IBM. *AFS User Guide, Version 3.6*. International Business Machines Corporation, first edition, Apr 2000. Publication Number GC09-4561-00, copyright 1989, 2000 IBM Corp.

[18] IBM. *IBM DFS for AIX and Solaris Administration Guide, Version 3.1*. International Business Machines Corporation, Apr 2000. copyright 2000 IBM Corp.

[19] JTC 1/SC 22. Information technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language]. ISO/IEC 9954-1:1996, Aug 2000.

[20] Innovative Computing Labratory. SInRG, complete project narrative. Avaliable at http://icl.cs.utk.edu/ sinrg/docs/sinrg-narrative.pdf, Jan 2002.

[21] M. J. Lewis and A. Grimshaw. The core Legion object model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 551–561. IEEE Computer Society Press, Aug 1996.

[22] D. Libes. Expect: Curing Those Uncontrollable Fits of Interaction. In *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, CA, Jun 1990.

[23] M. Litzkow, M. Livny, and M. W. Mutka. Condor - A hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, Jun 1988.

[24] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. In *Proceedings of the National Academy of Sciences of the United States of America*, volume 85, pages 2444–2448, Apr 1998.

[25] J. S. Plank, M. Beck, W. R. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the network. NetStore '99: Network Storage Symposium, Internet2, Oct 1999.

[26] J. S. Plank, M. Beck, and T. Moore. Logistical networking research and the network storage stack. In *Work-in-progress report, FAST 2002, Conference on File and Storage Technologies*. USENIX, January 2002.

[27] The Globus Project. GridFTP: Universal data transfer for the grid. Available at http://www.globus.org/datagrid/deliverables/C2WPdraft3.pdf, Sep 2000.

[28] H. C. Rao and L. L. Peterson. Accessing files in an internet: The Jade file system. *IEEE Transactions on Software Engineering*, 19(6), Jun 1993.

[29] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, Apr 1990.

[30] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS version 4 protocol. RFC 3010, Dec 2000.

[31] R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831, Aug 1995.

[32] R. Srinivasan. XDR: External Data Representation Standard. RFC 1832, Aug 1995.

[33] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, Dec 1990.

[34] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Cullen, P. Eastham, and C. Yoshikawa. WebOS: Operating system services for wide area applications. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, pages 52–63. IEEE Computer Society Press, Jul 1998.

[35] J. B. Weissman. Smart File Objects: A remote file access paradigm. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 89–97. ACM Press, May 1999.

Appendix

Appendix A

Remote Invocation

During the initial validation of the model, it was noted that the time required to farm out the test processes was longer than seemed reasonable. It was theorized that NetSolve was the source of this overhead.

After examining the data from the preliminary tests closely, it became apparent that using NetSolve to start a large number of jobs incurred a significant penalty. This is attributed to several factors.

1) The initial connection to a NetSolve agent results in a proxy of the client process being created. All communication to and from the client must go through this proxy.

2) Every connection to the NetSolve agent results in a new agent process being created, via fork() and exec(), to handle the connection.

3) Every connection to a NetSolve server results in at least one more server process being created (also with fork() and exec()) to handle the connection even if the task is refused for such reasons as insufficient capability or no such task on the server..

4) When a server initially accepts a task, information about the acceptance is not returned to the agent immediately nor is the impending load on the server correctly predicted. Specifically, even though the load is expected to be increased by some amount for every task a server performs, this information is based on metrics that assume mathematical computations based on the sizes of the input data and the projected speed of the local machine and assumes floating-point operations. If a task such as FASTA is started, this metric results in totally inaccurate initial load values.

5) The agent does not make any assumptions about which, if any server may have accepted the task. Consequently the next client request for the same task may get the previous list of servers in the same order.

The last item means that instead of being able to rely on the agent to provide a list that is ordered in terms of least loaded server to most loaded server as implied by the

documentation, the list defaults to alphabetical order. The ordering of this list and the fact that no machine is removed from it during the initialization of all the remote tasks, means that left to itself , NetSolve would start all the jobs on a single server.

Because this is an unwanted situation, the configuration file for each server was modified so that each only accepts a single task at time. This means that the set of servers available for any particular task could change radically if another process was attempting to access them. Even though this sufficed to ensure one task per server it did not change the way the agent responded. The agent still provided a list of all the potentially available servers and the NetSolve client code was required to contact each in turn until one was found that would accept the job. The time to start a particular task then is severely impacted by that task's place in the list of tasks to start. Each request by a client means opening a network connection to a local NetSolve Proxy process which in turn contacts the remote server. That server then spawns a process to handle the request. Any request in these tests was potentially refused due to the limit of one task per process which means that this entire set of operations must be repeated until some server accepts the task. The final testing scenario settled on 70 machines, this being a number that was large enough to be significant and could usually be relied upon to be available. Using NetSolve this means that in order to start 70 tasks 2485 requests have to be sent out by the client to servers.

Assuming that T_s is the time to contact a server and get an accept/reject response, T_e is the time to actually start a task on a server, then the total time, T_t, to start task m is

$$T_t = T_{e_m} + \sum_{i=1}^{m} T_{s_i}, \tag{A.1}$$

And the time to start N servers is

$$T_t = \sum_{j=1}^{N} T_{e_j} + \sum_{i=1}^{j} T_{s_i} \tag{A.2}$$

Assuming that the time to start a task on any particular machine is a constant, T_e and that the time to connect to any particular server and get an accept/reject response is a constant T_s then (A.2) reduces to

$$T_t = N * T_e + (N\,(N+1)) * T_s. \tag{A.3}$$

During the validation testing the time to start a task was approximately 0.3 seconds and and the time to contact a server for the accept/reject response was approximately 0.2 seconds. This meant that the time to start the 70 tasks using NetSolve took on the order of 70 * 0.3 + (70(71) / 2) * 0.2 = 70 + 2485 = 518 seconds or approximately 8.6 minutes. This is not a serious problem in and of itself but it is enough to be noticeable and could significantly impact the prediction of total running time.

A.1 New Client-Server Software

The problems that thus far encountered with NetSolve were, overhead, difficulty of crafting problem descriptions, and difficulty in modifying the system. To overcome these a more

basic client-server system for remote processing was needed. The most important principle focused on during the creation of this new system simplicity.

The primary advantage that the resulting client-server system has over the NetSolve code is this simplicity. When a server starts it opens a configuration file and searches a set of paths specified in that file for executable programs. These programs are the only resources that it uses. The server then acquires the current processor load as reported by a basic Linux/Unix utility, uptime. Last, it runs a simplified benchmark suite to derive floating point and integer operation per second values.

There is same problem encountered with NetSolve that in order to be used a server must exist on the remote host. In either case this can be done by by individually logging in to the remote host or through the use of a scripting language such as Expect [22] to automate the procedure. Once the servers are in place they report to an agent. The report contains their status, the set of tasks they can execute and the processor statistics. They update the agent periodically (the default is five minutes) with the current cpu load.

In order to execute a task, the client first must explicitly request a list of servers from the agent. The list is supplied with the processor statistics. This allows the client to determine which servers to use and in what order to use them. The client then contacts the selected server directly and passes a string which contains the name of the executable and its command line arguments. The server validates the task name and if correct, causes the task to be executed through fork() and execv(). This is an asynchronous set of operations. There is no concept of waiting for the results after the initial contact. The client queries the server for the status of the task when it chooses.

This all means that there is no specific mechanism for transmitting parameters to or receiving results from a task. The client is responsible for ensuring that data is available where the task can access it and for ensuring some method for returning results is established ahead of time. But this is really no different from NetSolve. Even though NetSolve has a mechanism for sending and receiving data, it is not effective when talking about tens or hundreds of megabytes. As a consequence the user must modify the basic RPC-like concept no NetSolve. This is the point at which NetSolve's load balancing system breaks down, with nothing specified as an argument but a file name string, the built-in system for determining server load has nothing to work with.

Fortunately, FASTA was very easily accommodated to this new client-server system. Because FASTA can write to any file, part of the front-end program's function is to use the new client-server mechanism to start IBP_servers on the remote hosts, if needed, establish local byte arrays for prepositioning data or for output files and then delete the byte arrays after the data is saved and terminate the serves when no longer needed.

The efficacy of this new system is seen in that unless there is a failure of a server, there is no need to contact more than one server to initiate a task. Therefore, using the same notation as the previous section and again assuming that the times to contact a server and start a task are constant, the time to start N tasks

$$T_t = N * (T_e + T_s)\,. \tag{A.4}$$

Using the same values of 0.2 and 0.3 seconds respectively, the total time for 70 tasks is on the order of 70 * (0.2 + 0.3) = 35 seconds or 0.6 minutes, a significant improvement over the 8.6 minutes used by NetSolve..

Vita

Kim Buckner was born in Republic, Washington on February 7, 1954. He attended public schools in Ferry County and graduated from Republic High School in 1970. After periods of work and junior college he joined the U.S. Navy in 1975. In 1990 he resumed college study and received a Bachelor of Science in Computer Science in 1993. Retiring from the Navy in 1995, he began at the University of Tennessee, Knoxville. He completed a Master of Science in Computer Science in May, 1998 and received the Doctor of Philosophy Degree in August, 2003.