

To the Graduate Council:

We are submitting herewith a dissertation written by Zhong Tang entitled “Load Balancing in Parallel and Distributed Databases”. We 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 Electrical Engineering.

J. Douglas Birdwell

Co-Major Professor

John Chiasson

Co-Major Professor

We have read this dissertation
and recommend its acceptance:

Hairong Qi

Tse-Wei Wang

Acceptance for the Council:

Anne Mayhew

Vice Chancellor and
Dean of Graduate Studies

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

LOAD BALANCING IN PARALLEL AND DISTRIBUTED DATABASES

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

Zhong Tang
May 2005

To yan zhang, my wife.

Acknowledgments

I would like to thank all the people who helped and supported me throughout the completion of this work. To each of them, I am very grateful.

First and foremost, I would like to thank my advisor, Dr. J. Douglas Birdwell, for his support and guidance throughout my studies at UTK. It is the work under his supervision and in the Laboratory for Information Technologies (LIT) that has shaped my skills as a researcher. Also, I want to thank my co-advisor, Dr. John Chiasson. His advice and direction over these years have been of equal importance. Without both of them, this work would not have been possible. I would further like to thank Dr. Tse-Wei Wang for many valuable discussions with regard to this research and life in general. Her vivid explanations on applied mathematics and molecular biology (especially DNA) will always be in my memory. I greatly appreciate Dr. Hairong Qi as well for her time and input to this dissertation.

Second, I owe many thanks to both staff and students at the LIT lab that I enjoyed working in. Special thanks go to Mark Rader and Dale V. Stansberry for many helpful technical conversations. Many thanks to Ron Tipton for helping me on administering computers. Also, I want to thank my fellow graduate students, John White, Kenny Gilbert, Yongsheng Pan, and Zeqian Shen, for sharing moments in the lab.

Last but not least, I thank my parents for their consistent support through my long time in school, and my wife for her encouragements in these years and sharing everything in my life.

This work was supported by U.S. Department of Justice, Federal Bureau of Investigation under contract J-FBI-98-083, by the National Science Foundation under grant number ANI-0312182, and by a Challenge Grant Award from the Center for Information Technology Research at the University of Tennessee. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. Government.

Abstract

This dissertation addresses the load balancing problem for parallel computations in the presence of *time delays* from a control system design perspective. Extensive experiments in a parallel computing environment have been made to evaluate load balancing strategies, and have driven the continued evolution of the parallel computation model and the load balancing strategies.

The critical feature of the load balancing problem is the delayed receipt of information and transferred load. In this work, load balancing in a parallel computer architecture is first modeled as a deterministic time-delay system with saturations. Three essential properties of the load balancing strategy, including consistency, stability, and conservation of tasks, have been analyzed. The excess load on each sending node is portioned out among the other nodes according to the amounts they are below the sending node's *estimate* of network average load. It is found that the feedback gain in the controller (load balancer) on each node affects the performance of load balancing, and a judicious choice of the feedback gain is necessary to achieve a faster balancing response without breaking into oscillatory behavior.

Load distribution and task processing contend for the same resources on each computational element (node). A deterministic dynamic nonlinear system is developed to model load balancing for parallel computations and incorporate both time delays and resource constraints. This model accounts for the trade-off between using processor resources to process tasks and the advantage of distributing the load evenly between

the nodes to reduce overall processing time. The model is shown to be consistent and the open-loop system is Lyapunov stable, but not asymptotically stable. A distributed closed-loop controller is proposed to balance load dynamically at each node by using not only local estimates of loads at other nodes, but also estimates of the number of tasks in transit to it. Experimental results have demonstrated substantial improvements in performance using a controller based on the *anticipated* estimates of workloads over using a controller based on the local workloads only.

The eventual size and the search requirements for DNA databases necessitate the development of parallel DNA databases. This work presents the design of a parallel DNA database, which is the motivation and an application of the presented load balancing strategies. DNA profiles are assigned in portions to each search engine node of the parallel computer to form distributed databases to be searched in parallel. A *multi-threaded* search server is designed and implemented to achieve high performance. Experimental results for the parallel DNA database and parallel searches with load balancing integrated with the database demonstrate both the efficiency of the parallel database and the efficacy of the load balancing strategy.

Contents

1	Introduction	1
1.1	Background and Literature Review	2
1.1.1	Parallel Processing	2
1.1.2	Load Balancing	9
1.2	Parallel Database	19
1.2.1	Parallel Database Server Architectures	19
1.2.2	Parallel Database Systems	21
1.3	Motivations	23
1.4	Completed Work	27
1.5	Dissertation Organization	30
2	Problem Formulation	31
2.1	Problem Statement	31
2.2	Parallel Machine Measurement and Parameterization	35
2.2.1	Parallel Machine Setup	35

2.2.2	Parameter Values	37
2.3	Summary	40
3	Load Balancing System Model	42
3.1	Linear Model	44
3.1.1	Stability Analysis	47
3.1.2	Brief Summary	49
3.2	Nonlinear Model with Proportional Feedback Control	50
3.2.1	Load Transfer Portions	52
3.2.2	Model Consistency	54
3.2.3	Saturation Controller	57
3.2.4	Brief Summary	63
3.3	Nonlinear Model with Resource Constraints	63
3.3.1	Model Consistency and Stability	66
3.3.2	Feedback Control and Load Transfer	70
3.3.3	Brief Summary	77
3.4	Summary	78
4	Parallel DNA Database	79
4.1	DNA Search Engine	81
4.2	Parallel Search Server	84
4.2.1	Hierarchical Sockets Search Server	85
4.2.2	Multi-threaded Search Server	88

4.3	Parallel Search with Load Balancing	98
4.4	Summary	101
5	Experimental Results	103
5.1	Load Balancing Strategies	104
5.1.1	Nonlinear Model with Proportional Feedback Control	105
5.1.2	Nonlinear Model with Resource Constraints	118
5.2	Parallel Database and Timing Results	142
5.2.1	Parallel Database with Hierarchical Sockets	144
5.2.2	Multi-threaded Parallel DNA Database	149
5.3	Parallel Search with Load Balancing	159
5.3.1	Initial Tasks Only	160
5.3.2	Task Generation	162
5.3.3	Results on Multiple Nodes	169
5.4	Summary	176
6	Conclusions	178
6.1	Completed Work	179
6.2	Directions for Future Research	181
6.2.1	Discrete Event Model	181
6.2.2	Dynamic Resource Allocation	183
6.2.3	Hierarchical Load Balancing	184

Bibliography	186
Appendix	221
Vita	224

List of Tables

4.1	Performance comparison of two search servers	97
5.1	Average search and transaction times on different search servers	157
5.2	Transaction times on the same <i>utss</i> for different client nodes	159
A.1	Configuration of the nodes of the parallel machine and two selected com- puters in LIT.	223

List of Figures

1.1	Parallel database server architectures according to the distribution of memory and disks	20
1.2	A parallel cluster of 24 computing nodes built in the LIT lab	25
1.3	A depiction of the hierarchical structure of a parallel DNA database with load balancing	26
2.1	Model of a flow control system	33
2.2	A queue model at each node for task processing and load balancing. . .	34
2.3	Hardware structure of the parallel machine.	36
2.4	A lab built parallel machine consisting of 2 server nodes and 24 computing nodes	37
2.5	Round trip time versus amount of data transfered in bytes.	39
2.6	Measured bandwidth versus amount of data transfered in bytes.	40
3.1	Illustration of a hypothetical distribution p_{i1} of the load at some time t from node 1's point of view	53

3.2	Summary of the load balance time as a function of the feedback gain. . .	62
3.3	Plot of queue sizes using the controller (3.30) and p_{ij} specified by (3.31). .	76
3.4	Plot of queue sizes using the controller (3.35) and p_{ij} specified by (3.36). .	76
4.1	Flowchart of the parallel search engine.	84
4.2	Schematic diagram of a parallel search server using hierarchical sockets. .	86
4.3	Flowchart of the hierarchical sockets search server.	87
4.4	Time measured for 2,000 random searches on 24 nodes of 48M profiles .	89
4.5	Schematic diagram of a multi-threaded search server.	92
4.6	Multi-threaded search server: requests processing.	95
4.7	Multi-thread search server: results returning.	96
4.8	Time measured for 2,000 random searches on the multi-threaded search server	97
4.9	A depiction of the parallel DNA database with integrated load balancing	99
4.10	Flowchart of <code>MainServerThread</code> for parallel search with integrated load balancing.	100
5.1	Node 1 of a three-node simulation system for load balancing.	106
5.2	Constant p_{ij} nonlinear output responses with $K = 1000$	108
5.3	Constant p_{ij} nonlinear output responses with $K = 5000$	108
5.4	Nonlinear simulation with constant p_{ij} and $K1 = 5221$, $K2 = 4883$, $K3 = 5800$ ($K_z = 0.5$).	109
5.5	Experimental response of the load balancing strategy ($K_z = 0.5$)	111

5.6	Experimental response of the load balancing strategy ($K_z = 0.3$)	112
5.7	Experimental response of the load balancing strategy ($K_z = 0.2$)	112
5.8	Summary of settling time as a function of the feedback gain K_z	113
5.9	Experimental response of the load balancing with $K_z = 0.6$	114
5.10	Experimental response of the load balancing with $K_z = 0.6$: ringing . .	115
5.11	Experimental response of load balancing on Node00	116
5.12	Experimental response of load balancing on Node09	116
5.13	Experimental response of load balancing on Zero	117
5.14	Node 1 of a three-node simulation system with all nonlinearities.	120
5.15	Simulation and experimental results on a three-node system	121
5.16	Average estimates and tracking differences on a three-node system. . . .	122
5.17	Workloads and average estimates on node23.	122
5.18	Experimental results for load balancing with transient suppression. . . .	124
5.19	Workloads and average estimates on node23 with transient suppression.	124
5.20	Results of the load balancing strategy executed at $t_0 = 1\ ms$	127
5.21	Local estimates of the network average load on three nodes	127
5.22	Local queue size relative to its estimate of the network average on each of three nodes	128
5.23	Anticipated estimates of the network average load on three nodes	131
5.24	Local queue size relative to the anticipated estimate of the network av- erage on each of three nodes	132

5.25	Estimates of the queue sizes at node 2	132
5.26	Estimates of the queue sizes at node 3	134
5.27	Comparison of the actual queue size and the anticipated estimate of queue size	134
5.28	Results of closed loop load balancing on three nodes	136
5.29	Tracking difference on each of three nodes in closed loop load balancing	137
5.30	Estimated queue sizes by node 2 in closed loop load balancing	137
5.31	Queue sizes using controller (3.30) and p_{ij} (3.31) based on local waiting times x_i	139
5.32	Queue sizes using controller (3.35) and p_{ij} (3.36) based on anticipated waiting times z_i	139
5.33	Queue sizes on six nodes with initial tasks using controller (3.30) and p_{ij} (3.31) based on x_i	141
5.34	Queue sizes on six nodes with initial tasks using controller (3.35) and p_{ij} (3.36) based on z_i	141
5.35	Queue sizes on six nodes with task generation $\lambda_1 = 6$ using controller (3.30) and p_{ij} (3.31) based on x_i	143
5.36	Queue sizes on six nodes with task generation $\lambda_1 = 6$ using controller (3.35) and p_{ij} (3.36) based on z_i	143
5.37	A running <i>multid</i> server communicates with 24 <i>codisd</i> search engines. .	145

5.38	Command line search tool <i>cline</i> interacts with a search server <i>multid</i> with eight search engines.	146
5.39	Search times of 1,000 random searches on a single node with 2M DNA profiles	147
5.40	Histogram for search times of 1,000 random searches on local node01 . .	147
5.41	Time measured for 2,000 random searches on 24 nodes of 48M profiles .	149
5.42	Time measured for requests with matches found on node01	150
5.43	A multi-threaded search server <i>utss</i> communicates with 24 search engines via PVM and object serialization	151
5.44	Graphical search tool <i>gst</i> communicates with the search server <i>utss</i> . . .	152
5.45	Search times of 1,000 random searches on a single node using the multi- threaded search server <i>utss</i>	153
5.46	Transaction times of 1,000 random searches on a single node using the multi-threaded search server <i>utss</i>	154
5.47	Time measured for 2,000 random searches on 24 nodes using the multi- threaded search server	155
5.48	Time measured for requests with matches found on node01 using the multi-threaded search server	156
5.49	Transaction times on different client nodes communicating with the same server <i>utss</i>	158

5.50	Workloads, average estimates and tracking differences on node3 with initial tasks.	161
5.51	Workloads, average estimates and tracking differences on node1 with initial tasks.	161
5.52	Average estimates on three nodes with initial tasks.	162
5.53	Tracking differences on three nodes with initial tasks	162
5.54	Workloads, average estimates and tracking differences on node1 with tasks generated from a file.	164
5.55	Workloads, average estimates and tracking differences on node2 with tasks generated from a file.	164
5.56	Average estimates on three nodes with tasks generated from a file . . .	164
5.57	Tracking differences on three nodes with tasks generated from a file . . .	164
5.58	Workloads, average estimates and tracking differences on node1 with randomly generated tasks	166
5.59	Workloads, average estimates and tracking differences on node2 with randomly generated tasks	166
5.60	Workloads, average estimates and tracking differences on node3 with randomly generated tasks	167
5.61	Comparison of average estimates on three nodes with randomly generated tasks	168
5.62	Responses of queue sizes on three nodes without load balancing	168

5.63	Workloads, average estimates and tracking differences on node1 with initial tasks on six nodes.	170
5.64	Workloads, average estimates and tracking differences on node2 with initial tasks on six nodes.	170
5.65	Average estimates on six nodes with initial tasks	170
5.66	Tracking differences on six nodes with initial tasks	170
5.67	Workloads, average estimates and tracking differences on node3 with randomly generated tasks	172
5.68	Workloads, average estimates and tracking differences on node5 with randomly generated tasks	172
5.69	Comparison of average estimates on six nodes with randomly generated tasks.	173
5.70	Workloads, average estimates and tracking differences on node1 with tasks initially distributed to node1	175
5.71	Workloads, average estimates and tracking differences on node5 with tasks initially distributed to node1	175
5.72	Comparison of average estimates on six nodes with tasks initially distributed to node1	176

Chapter 1

Introduction

The focus of this research is load balancing in the field of parallel processing and its applications for parallel and distributed databases. This introductory chapter briefly introduces parallel processing and the importance of load balancing. Fundamental concepts are defined, and the existing work related to load balancing and parallel databases is summarized in detail. Next, the motivations of this research are addressed, and the tasks completed in this research are briefly described. Finally, an outline of the remainder of this document is presented as a detailed guide to the research findings and experimental results.

1.1 Background and Literature Review

1.1.1 Parallel Processing

The goals of parallel processing are to reduce the wall-clock time required for computations and to increase the size of solvable problems. Parallel processing, also known as parallel computing, refers to the use of multiple processors to speed-up a problem's solution by dividing the software into multiple fragments that can execute simultaneously on several processors. A program that is extended to run on a system of N processors would ideally execute N times faster than when using a single processor. Though this ideal speed-up is not achievable in most applications, load balancing is one method of distributing the computational workload evenly among processors and thereby optimizing performance.

1.1.1.1 Moore's Law

The computer industry has been one of the fastest growing industries since the middle of the twentieth century. The rapid technological developments in the areas of computer hardware and software have enabled the rapid advancement of information management and have changed many aspects of our lives as well. The technological advances in hardware include chip development and fabrication technologies, as well as high bandwidth and low latency interconnected networks [39]. The software technologies, including operating systems, programming languages, development methodologies, and support tools are also rapidly developing. This growth has enabled the development

and deployment of extremely large applications for scientific and commercial needs.

In 1965, Gordon Moore predicted that the number of transistors on a chip would double every year [158], and this is dubbed as Moore's law. Ten years later, he revised the law to predict doubling every two years. David House, an Intel executive at the time, noted that the changes would cause computer performance to double every 18 months, and this time constant has been widely quoted. The law predicts technological progress and explains why the computer industry has been able consistently to come out with products that are smaller, more powerful and less expensive than their predecessors.

However, Moore's law is not a law of physics. It is merely a reasonably accurate observation about the pace of steady improvements in manufacturing processes. In other words, it is an assessment of what engineers, when organized properly, can do with silicon. Unfortunately, the laws of physics guarantee that the pace cannot continue indefinitely, as there is a physical limit to the density — transistors must be larger than an atom. The fact that materials are made of atoms is a fundamental limit, and we will get close to this limit in the not too distant future.

An important reason for creating and using parallel computers is that parallelism is one of the best ways to overcome the speed bottleneck of a single processor. In addition, the cost/performance ratio of a cluster of parallel computers is much smaller when compared to the economics of a supercomputer. In other words, developing and producing parallel computing systems of moderate speed is much cheaper than implementing a sequential computing system of the equivalent performance. This is one reason modern

supercomputers almost universally employ parallelism.

1.1.1.2 Goals of Parallel Processing

Many scientific problems require huge memory and computational resources. Examples include linear algebra algorithms applied to super size matrices, finite element computing, and searches in very large databases. Such applications not only require Gigabytes to Petabytes of memory, but also billions, or trillions and more, of floating point operations — performance that is larger than can reasonably be expected from a single processor machine. Even when sufficient resources are available, the time required to execute these applications can be unacceptably long on a single processor machine.

The goals of parallel processing are speed-up and scale-up. Speed-up means to reduce the wall-clock time required before a user receives an answer to a computing problem. Scale-up means to solve large problems that do not fit in the limited memory of a single processor computer.

Parallel processing has its roots in the scientific world, and its reach is now extending into the business world. The best candidates for parallel processing are projects that require many computations with similar structure but different data. Single-processor computers perform each computation sequentially. Using parallel processing, a computing system can perform several computations at once, drastically reducing the time it takes to complete a project.

1.1.1.3 Performance Measurement

To estimate performance improvement for parallel design and implementation, Amdahl's law has been widely used by researchers. It is a law governing the speed-up of using parallel computers on a problem, versus using only one serial computer. The speed-up is defined as the time it takes a program to execute in a serial computer divided by the time it takes to execute in a parallel computing system of many processors. If s is the fraction of a calculation that is serial and $1 - s$ the fraction that can be parallelized, then the speed-up that can be achieved using N processors is [10]:

$$\text{Speed-up} = 1/(s + (1 - s)/N),$$

which has a limiting value of $1/s$.

Amdahl's Law is a mathematical concept. It was originally applied to parallel computing, and had consequences throughout all of computer science and engineering. The gist of the law is that an improvement to an infrequent stage in an operation will have little impact on the overall operation. However, the negative way that the original law was stated casts pessimism on the nature of parallel processing if one overestimates the value of the serial fraction.

Researchers in parallel computing fields often observe impressive – almost linear – speed-ups in some large applications. J. Gustafson reported linear speed-ups on a 1024-processor hypercube for three practical applications (with $s = 0.4 - 0.8\%$): 1021 for

beam stress analysis, 1020 for baffled surface wave simulation, and 1016 for unstable fluid flow [95]. An alternative formulation was proposed. If s and p represent serial and parallel (fraction) times spent on the parallel system, then a serial system would require time $s + pN$ to perform the task [95]. Therefore,

$$\textit{Scaled speed-up} = (s + pN)/(s + p) = s + (1 - s)N = N + (1 - N)s,$$

which is known as Gustafson's Law.

Gustafson's Law is a reply to the critics who used Amdahl's Law to argue against parallel systems. The difference is that Amdahl's Law assumes the problem size is fixed, while Gustafson's Law assumes the problem is scaled to the number of processors. It is Gustafson's work that shows the bright future of parallel computing.

In summary, parallel computing can significantly reduce the wall-clock time of many large applications by judiciously using of parallelism, but parallel computing also adds complexity. Even writing and debugging software is quite complicated, and the overhead of parallelism requires resources, including processor cycles, memory, and communication bandwidths. Parallelization of existing software, or design of parallel software implementations, is not automated and does not have as many or as sophisticated tools as those used for sequential software. Therefore, there is also a human cost. However, for large problems and applications, parallel computing can be an enabling technology: Without parallel computing, many current problems can not be solved.

1.1.1.4 Parallel Architectures

Parallel processing extends computing to systems with many processors to obtain speed-up in code execution. The efficiency and effectiveness of the parallelism are largely dependent on the problems to be solved, the selected algorithms, and hardware architectures. Based on load distribution and synchronization implementations, single-instruction multiple-data (SIMD), multiple-instruction multiple-data (MIMD), single-program multiple-data (SPMD), and many other “fashions” have been categorized. SIMD machines are used mostly for special purpose applications. Typical examples are some specially designed digital signal processors. MIMD computers are commonly used and can be classified into [120][39]:

- Parallel Vector Processors (PVP)

A PVP system contains a small number of powerful custom-designed vector processors connected via a custom-designed, high-bandwidth switch network to a number of shared-memory modules. Typical examples include the Cray X1 [57] and the NEC SX-8 [162].

- Massively Parallel Processors (MPP)

An MPP system is a tightly coupled large-scale computing system architecture, which typically consists of several hundred processors interconnected by a high-speed memory bus to a local memory and a high-speed, proprietary communication network. Representative machines have included the Connection Machine,

the Intel Paragon and TFLOPS [153], and the Cray XT3 [57].

- Symmetric Multiprocessors (SMP)

An SMP system has typically 2 to 64 processors with each processor sharing the global memory, bus, I/O, and other resources. A representative machine is the Sun Fire 12K Server [194].

- Clusters of Computers

A cluster is a collection of workstations interconnected via a computer network, which may range from a low-cost commodity network such as Gigabit Ethernet to special-purpose low-latency networks such as Myrinet. Clusters are the low-cost variations of MPPs in some cases. A typical example of clusters is the Virginia Tech Terascale Cluster, which consists of 1,100 Power Mac G5 connected through 24 Mellanox 10 Gbps switches [206].

Many modern parallel computers are built with commercially available, off-shelf hardware and software components. Although these parallel computers rarely match the performance of the most sophisticated leading edge special-purpose designs, they almost always provide the best cost/performance ratio. A number of current installations have demonstrated competitive performance at the highest end of the supercomputing scale [203], such as the Thunder [143] that consists of 1,024 Intel Itanium2 Tiger4 1.4GHz nodes, and the IBM BlueGene/L DD2 beta-System [121] that consists of 32,768 PowerPC 0.7GHz nodes.

1.1.2 Load Balancing

Both supercomputers and clusters of workstations need appropriate tuning to achieve the expected level of performance. When the computations for a parallel application are divided among a fixed number of processes that are to be executed on the parallel nodes, each process performs a certain amount of work. However, it is quite possible that some processors will complete tasks earlier than others and become idle because the work is not evenly divided, or some processors may operate faster than others, and thus finish earlier. Both situations can occur. Effective utilization of a parallel computer architecture requires the computational load to be distributed evenly over the available processors. One of the most important requirements of efficient parallel computing is to keep all the nodes busy; ideally, this coincides with completion of the job in the shortest possible time.

Minimizing load imbalance is a key activity in producing efficient implementations of applications on parallel architectures. Distribution of computational load across available computing resources is referred to as the *load balancing* problem in the literature. The objective of load balancing is to minimize execution time for a program. This is achieved by a strategy that shifts tasks from heavily-loaded nodes to lightly-loaded ones.

1.1.2.1 Load Balancing Strategies

A load balancing strategy needs to resolve four issues:

- what load information (measurements) can be used,

- when to invoke balancing, i.e., conditions to balance,
- which nodes make the load balancing decision,
- how load migrations are to be managed.

There is a large variety of load balancing strategies with different answers to and emphasis upon these four questions.

Load balancing can be done statically before the execution of any process starts, or dynamically during the entire execution process. Static methods examine the global distribution of computational load and assign portions of the workload to resources before processing begins. In contrast, dynamic methods examine the progress of the computation and the expected utilization of resources, and adjust the workload assignments as computation progresses.

Static load balancing depends on locally available resources and usually is efficient at the initial point of execution for some applications. However, because load balancing is done just once at the initialization phase of the computation, static load balancing is not suitable for computations that have a dynamically changing workload throughout execution. Dynamic load balancing refers to redistribution of workload among the parallel processors during the computation process, so that each processor has, more or less, the same amount of work to do.

Dynamic load balancing strategies have an advantage over static strategies: The system does not need an *a priori* determination of the run-time execution behavior. Dynamic load balancing has been a focus of research in the field of parallel computing,

and previous studies suggest dynamic strategies have greater potential for performance improvement than static strategies. These studies are summarized below.

1.1.2.2 Dynamic Load Balancing

Dynamic load balancing is based on the redistribution of load among the processors as the computation progresses. Various taxonomies of load balancing methods exist in the literature [182][189]. Methods to balance computational load may be either deterministic, depending on a predefined strategy, or stochastic, distributing load in a random fashion.

Deterministic Methods

Deterministic methods perform in an iterative manner according to certain predefined strategies. A comparison of several deterministic methods is provided by Willebeek-LeMair and Reeves [214]. Most deterministic methods can be classified into four categories: diffusion [58][212][55], dimension change [58][217], gradient model [146][159], and minimum-direction methods [147][218]. With diffusion methods, a processor exchanges workload with all its neighbors simultaneously at every step. With dimension change methods, a processor goes around the table, exchanging workload with neighbors one at a time and using the new workload for the exchange with the next neighbor. With gradient model methods, a processor transfers the excess load along the direction of the most lightly loaded processor in the system. With minimum-direction methods, the destination processor is chosen as the least loaded processor of the underlying domain (i.e.,

a local minimum) instead of using a global minimum in the gradient model methods. These deterministic methods are briefly discussed below.

- Diffusion

In diffusion methods, each overloaded processor transfers a portion of its excess load to its underloaded neighbors to achieve a local load balance, in which load balancing is modeled as an iterative diffusion process towards a steady balanced state. The processors are arranged in a grid, as if they had spatial coordinates, and load exchanges occur between nearest neighbors. The coordinate system can correspond to parallel communication architectures, where processors “nearer” each other can communicate relatively more easily. Under the synchronous assumption that a processor would not proceed into the next iteration until all the workload transfers of the current iteration have completed, the dynamic behavior of the diffusion method has been studied by modeling it as an iterative process by Cybenko and others [58][212][55]. Specifically, let $L^t = (l_1^t, l_2^t, \dots, l_n^t)$ denote the workload distribution in the network of n nodes at time t , and let $A(i)$ be the set of direct neighbors of the processor i . The change of workload at the processor i in each iteration step of the diffusion method (from time t to time $t + 1$) is given by

$$l_i^{t+1} = l_i^t + \sum_{j \in A(i)} \alpha_{ij}(l_j^t - l_i^t) + \delta_i^{t+1} - \theta_i^{t+1}. \quad (1.1)$$

The parameter α_{ij} ($0 < \alpha_{ij} < 1$) is called the diffusion parameter of i and j , which determines the amount of workload to be exchanged between the two processors;

δ_i^{t+1} and θ_i^{t+1} denote the amounts of workload generated and completed at time $t + 1$, respectively.

With the assumption that no new workload is generated or existing workload completed during the execution of the load balancing procedure (this does not always hold, especially in multi-threaded applications), Cybenko shows that the diffusion method is convergent given any initial workload distribution [58]. Then, without the assumption, he shows that the diffusion method can control the growth of the variance of the unbalanced workload distribution and keep it bounded. Similar results have been obtained in experiments on hypercubes [218] and in mesh networks [220]. Willebeek-LeMair and Reeves study the behaviors of load balancing using Sender Initiated Diffusion (SID) and Receiver Initiated Diffusion (RID) strategies, depending on the determination of active processors [214]. An improved diffusion method is presented in [115] to improve the slow convergence of this class of method using global communication. Other work includes the study on the effectiveness of different diffusive load balancing strategies in [54][212]. The diffusion method is based upon models in which a processor can communicate with all its neighbors simultaneously. Therefore, this works best on hardware architectures that support high-performance parallel communications over the interconnected processors (e.g., SMP). In many other cases, communications among processors are carried out sequentially, and the diffusion method becomes less effective.

- Dimension exchange

The dimension exchange method was initially studied for parallel systems with a hypercube topology, as each processor's neighbors are inspected by following each dimension of the hypercube [58]. Each processor examines and exchanges its workload with those of its neighbors one at a time, and going through all the neighbors (dimensions of the hypercube) once denotes a *sweep* of the iterative process of the load balancing strategy. The dimension exchange method has been extended to arbitrary architectures, other than hypercubes, using edge-coloring of undirected graphs [114].

In the dimension exchange method, an exchange over a dimension of the hypercube (i.e., an edge of the graph) will result in equal workloads in the two processors along the selected dimension. This equal splitting of workload is found to be optimal for hypercubes, but non-optimal for some other structures. Xu and Lau present a generalized dimension exchange method using an exchange parameter β to control the splitting of workload between a pair of directly connected processors [221]: Consider a n -dimensional structure. If processors i and j are direct neighbors, then the change of workload at processor i along the dimension with j is given by

$$l_i^{t+1} = (1 - \beta)l_i^t + \beta l_j^t, \quad (1.2)$$

where $0 < \beta < 1$ is the exchange parameter controlling workload splitting. This

generalized dimension exchange method reduces to the ordinary dimension exchange (on hypercubes) when $\beta = 1/2$. For chain, ring, mesh and torus structures, the optimal exchange parameters are derived in closed form [221].

Loads on processors are treated as real numbers in the work described above for convergence analysis. The direct neighbor method, which uses a discrete load model, is presented in [55]. Both the direct neighbor method and its extended variant by the same authors, which uses non-nearest neighbors, are based upon the dimension exchange philosophy. The dimension exchange method outperforms the diffusion method on small and medium scale parallel architectures in which the cost of the necessary global synchronization phase is not significant.

- Gradient model

In gradient model methods, a contour of the gradients is formed by the differences in workloads of processors in the network. Load redistribution is restricted to occur toward the direction of the most lightly loaded processors in the network [146]. The main difference between the gradient model and the dimension exchange method is that load is migrated to the processor that has the lightest load in the entire network, while only one neighbor processor is considered at each iteration in the dimension exchange method.

The gradient model method has several drawbacks. As heavily loaded processors transfer workload to lightly loaded processors only, much commotion will result when a large portion of moderately loaded processors suddenly becomes lightly

loaded (e.g., with current jobs just completed). And, if there are only a few lightly loaded processors in the system, more than one overloaded processor may transfer some load to the same underloaded processor; this causes an *overflow* effect. The overflow effect may transform the originally underloaded processors immediately to become overloaded. Extensions to the gradient model have been studied in [149] and [159] to alleviate the overflow effect.

- Minimum-direction

In minimum-direction methods, the processor chooses the most underutilized processor within its domain as the destination of load transfer [189]. The domain may range from a small group of two processors to the entire system with all processors involved. Depending on the scope of the domain, the underutilized processor chosen from the domain (i.e., a local minimum) may coincide with the underutilized processor in the entire system, which is a global minimum. The minimum-direction method uses global state information shared throughout the parallel system to balance load, and a load information center or a job dispatcher is therefore required.

A centralized load balancing strategy uses a central job dispatcher (LBC) to maintain the load information and dispatch workload among processors [147]. The drawbacks of such centralized load balancing strategies are the bottleneck caused by the central coordinator and fault-tolerance problems for large systems. A distributed load sharing scheme in [61] shows that distributed strategies are scalable.

Stochastic Methods

Stochastic methods distribute workloads in a random fashion to drive the system into an equilibrium state with high probability. Stochastic methods for load balancing can be classified into three categories: randomized allocation [227], physical optimization based [87][47][17], and queuing theory based methods [164][193][155][91]. Stochastic methods for load balancing are briefly described as follows.

- Randomized allocation

In randomized allocation methods, a random choice of a processor is made to migrate some workload. Several randomized load balancing methods are summarized in [227]; they use a threshold value as a trigger condition of load transfer, and transfer load in a random fashion. These methods are less dependent on system state information than deterministic methods. A processor may be randomly chosen among the direct neighbors only (local random methods) or among all processors including remote neighbors (global random methods). Lüling *et al.* compare local random and global random methods through simulation, and show the superiority of global random methods [149]. While the randomized allocation methods for load balancing show substantial performance improvement over no load balancing in [78][149], a precise theoretical analysis is not yet available.

- Physical optimization

Physical optimization methods map the load balancing problem into a combinatorial optimization problem. Commonly used optimization methods include

simulated annealing [47][87] and genetic algorithms [17][228]. The common drawback of physical optimization based balancing strategies is that the computations are expensive, while dynamic load balancing requires the result yielded in limited time.

- Queuing methods

Queuing methods attempt to solve the load balancing problem based on queuing theory [133] and probabilities. An approximate performance model for queues with load balancing in distributed systems is presented in [148], and a performance analysis of the priority queues is given by Hać in [98]. Nelson and Philips derived an approximation for the mean response time of a multiple queue system in which shortest queue routing is used [164]. Two balancing strategies, Bernoulli splitting and round robin, are used to reduce the mean response time, and the latter shows better performance in simulations. Other work on routing arrivals to M parallel queues are reported in [154][155].

The performance of three load balancing strategies discussed in [78] has been compared to two bounding cases: without load balancing ($M/M/1$ queues) and perfect load sharing at zero cost ($M/M/k$ queues) in homogeneous distributed systems. A queuing theoretic model of optimal load balancing strategy is presented in [193]. For the optimal balancing strategy, in which the load differences, measured in the number of tasks, is less than or equal to 1, in a distributed system consisting of identical processors, results comparable to $M/M/k$ are obtained from the model

and simulations [193]. Recent studies include an approximate analysis of queuing in ATM networks [91] and load balancing web servers [14].

In recent years, applications of control theory have also shown promise in information technologies, including databases [72][108] and web services [4][180]. A recent overview on applications of control and engineering in information technologies is presented in [31]. Other studies on load balancing distributed web servers include methods based upon packet routing [14] or mobile agents [42].

1.2 Parallel Database

The dramatic proliferation of information system technologies and applications in recent years requires much larger databases. The use of parallel processing technology makes it possible to meet those new and challenging requirements. The development of high performance parallel computers holds a bright future for parallel database systems as the next generation high-performance databases.

1.2.1 Parallel Database Server Architectures

Parallel database servers are built on parallel computing architectures. While the hardware architectures of parallel computers have been described in the previous section, parallel database servers are usually divided according to the degree of distribution of memory and disks to simplify the analysis. Parallel database servers can be divided into three categories: shared-everything, shared-disks, and shared-nothing [70][205][89].

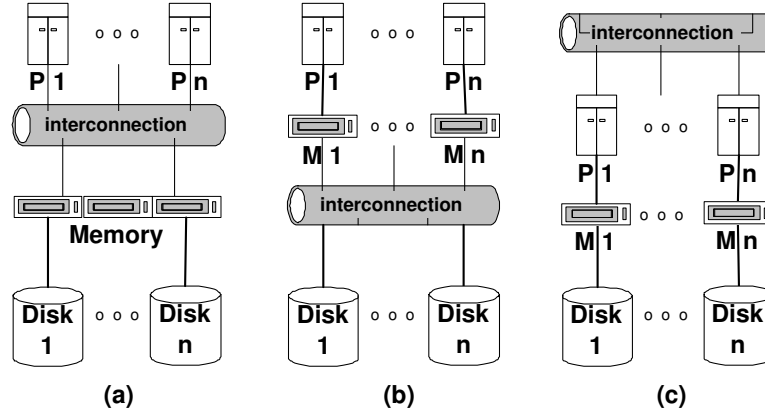


Figure 1.1: Parallel database server architectures according to the distribution of memory and disks. (a) Shared-everything, (b) shared-disks, and (c) shared-nothing.

These three categories of parallel database server architectures are illustrated in Figure 1.1 and are briefly described below.

- **Shared-everything.** All processors share direct access to a global large main memory, and to all disks. Representative designs include the IBM/370 and the SMP systems described above. The advantage for a shared-everything database server is that load balancing can be automatically done on the parallel database in a system of a small number of processors. The drawback for this kind of parallel database servers is that they are difficult to scale up as the number of processors increases. Currently, this architecture has scalability limited to a few tens of processors, i.e., normally ranging from 2 to 64.
- **Shared-disks.** Each processor has its own private memory, but has direct access to all disks. Any processor can request data on any disk. This architecture generally

has better scalability than the shared-everything architecture, while its major drawback is the high traffic of data in the interconnected network.

- **Shared-nothing.** In this architecture, each processor has its own memory and manages its own disk (and data). Mass storage in such an architecture is distributed among the processors, and all processors can access the data in parallel. This architecture for parallel database servers is also known as the cluster of computers described above. As is typical of the systems given above (such as the Tera-scale cluster [206]), this architecture offers good scalability by taking advantage of high-performance commodity processors and network interconnections. The share-nothing architecture has been a trend for parallel database servers. The major drawback of this architecture is that load balancing is difficult.

1.2.2 Parallel Database Systems

This subsection presents a brief overview of three representative parallel database systems. Note that unless otherwise noted, all trademarks are owned by their corresponding companies.

- **Teradata Warehouse**

Teradata developed the first parallel database server, Teradata DBC 1012, widely used to support very large databases in 1984 [89]. The Teradata DBC 1012 is a relational database management system (RDBMS), which uses a shared-nothing parallel architecture and a proprietary interconnection network. (The number

1012 represents one trillion bytes, or a Terabyte.) Since then Teradata has evolved its parallel RDBMS to data warehouses. A Teradata Data Warehouse [201] provides data access and management tools and robust data mining capabilities, while incorporating high-performance parallel database technologies.

- IBM DB2

IBM developed a parallel edition for its relational database system DB2 in 1996. The DB2 Parallel Edition runs on the IBM SP2, which also uses a shared-nothing architecture. The latest database products by IBM include DB2 Universal Database Data Warehouse Editions [122]. The DB2 Universal Database data warehouses provide capabilities including data mining and workload management.

- Oracle Parallel Server

Since the first Oracle parallel database ran on a massively parallel computer (Oracle Parallel Server) in 1991, Oracle has been actively developing parallel database systems. The latest parallel database products by Oracle is Oracle Database 10g, which is designed as a relational database for cluster computing [170]. Oracle Database 10g aims to provide rapid response time by using large numbers of nodes, and to reduce downtime by using a hot secondary database instance in the case of failure of the primary host.

Whereas the parallel database systems described above provide capabilities like data partitioning (for distributed storage), database management, and data mining, these

parallel databases are designed for managing and searching generic data. The parallel implementation can improve the database search performance, but can not specifically improve search performance on any one node. Standard database indexing structures, such as B-trees, are used in each of the parallel databases. Although the indexing structures can provide rapid access to records based upon the indexed values of one or more selected fields, multiple index structures that are computed independently are often required. When stored information based on several indices is retrieved, a time-consuming intersection operation is required to obtain the results. For multidimensional data (e.g., DNA profiles), naturally occurring structures can often be found within the stored data. Search performance can be improved using the naturally occurring structures within the multidimensional data. For example, a decision-tree based DNA profile storage and retrieval method as described in [33] improves the search performance by two orders of magnitude over current software. For large-scale databases of DNA profiles, a parallel DNA database that provides such a rapid search function is desirable, and load balancing is key to producing an efficient implementation. The motivations of the work on load balancing in parallel databases are discussed below.

1.3 Motivations

Modern DNA technology has resulted in the extraordinarily accurate identification of criminals and the exoneration of the innocent at a remarkable rate [38][183]. The Combined DNA Index System (CODIS), established by the FBI Laboratory, blends forensic

science and computer technology into an effective tool for solving violent crimes [81]. DNA databases have been growing rapidly in recent years. The National DNA Index System (NDIS), the top level in the hierarchical CODIS system, became operational in October 1998, and has received and stored over 2 million profiles by December 2004 [81]. It is predicted that the size of DNA databases will increase at a faster rate to an eventual scale of 10^8 profiles. Forensic applications require rapid searches on these DNA database. The anticipated size and the search requirements for DNA databases necessitate the development of parallel DNA databases. New decision-tree based rapid searching methods developed in [32][211][33] not only provide orders of magnitude improvements in performance over current software (5 *ms* vs 5 *sec.*), but also extend to a parallel decomposition of the DNA database search problem.

A parallel machine has been built in the Laboratory for Information Technologies (LIT) as an experimental facility for the parallel DNA database. The parallel machine includes a cluster of 24 computing nodes, which are shown in Figure 1.2. The design and the specification of the parallel machine is described in Section 2.2. In the DNA database application, a DNA search engine is executed on each node of the parallel cluster. The database is hierarchical, and search engine nodes are partitioned into groups that contain identical data. Due to the structure of the search process, search requests can be formulated for any target profile and associated with any node of the index tree. These search requests are created not only by the database clients; the search process itself can also create search requests as the index tree is descended by



Figure 1.2: A parallel cluster of 24 computing nodes built in the LIT lab. Photo courtesy of J. D. Birdwell [27].

any search engine. This creates the opportunity for parallelism; search requests that await processing may be placed in any queue associated with a search engine, and the contents of these queues may be moved arbitrarily among the processing nodes of a group to achieve a balanced load. Figure 1.3 shows the hierarchical structure of the parallel DNA database with load balancing in each group of nodes. The main interest of this research is in load balancing any one group of nodes.

The critical feature of the load balancing problem is the delayed receipt of information and transferred load. Each DNA search task requires little time, typically less than

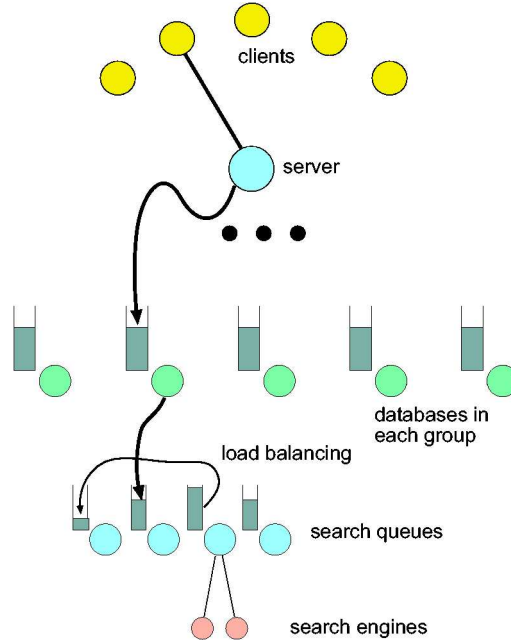


Figure 1.3: A depiction of the hierarchal structure of a parallel DNA database with load balancing. To even out the search queues, load balancing is done between the nodes of a group. Figure courtesy of [29].

1 *ms* for an exact DNA match, and communication delays are therefore important. In a group of nodes with load balancing, each node has only estimates of workloads at other nodes due to *communication delays* that occur when exchanging queue sizes between nodes and *transfer delays* that occur when transferring tasks from one node to another. While existing literature provides quite a large amount of work on the load balancing problem, load balancing for parallel computations in the *presence* of time delays has not been well studied. Moreover, load balancing actions consume both processing time and network bandwidth; that is, both task processing and load balancing contend for the same resources on each node. There is a trade-off between using processor resources

to process tasks and to distribute loads among nodes. This research has studied load balancing for parallel computations with time delays and resource constraints from the control theory point of view, and the results are documented in this dissertation.

1.4 Completed Work

The primary contribution of this research is its focus upon the load balancing problem from a control system design perspective, and extensive evaluation of load balancing strategies through experiments in a parallel computation environment. The experimental components of this research have driven the continued evolution of the parallel computation model and the load balancing strategies. The work documented in this dissertation adds to existing literature by proposing a nonlinear load balancing model incorporating both time delays and resource constraints, and implementation of a distributed controller to balance workloads based anticipated estimates of queue sizes. Development of the models and the theoretical analyses in this research include, for completeness, contributions by J. D. Birdwell and J. Chiasson, as noted in the text. The major tasks completed in this research are briefly summarized in the following.

Nonlinear Model with Proportional Feedback Control

A linear model is developed first to abstract a load balancing system in the presence of time delays [1][2]. As a node would not initiate a negative amount of task transfer to a recipient node in an actual system, a saturation of local workload relative to the

estimated network average must be considered in the feedback controller. A nonlinear (saturation) time-delay system is developed to model the load balancing strategy with saturation in a parallel computing network. This model is shown to be consistent in that the queue lengths on nodes cannot go negative and that the total number of tasks in all the queues and the network is conserved, and to be asymptotically stable in the sense of Lyapunov [28].

The model includes a proportional feedback control law. A controller (load balancing method) at each node evaluates the local load relative to the estimate of the network average load. The excess load on each sending node is portioned out among the other nodes according to the amounts they are below the local estimate of network average load. Simulations of this nonlinear model and controller indicate good agreement with an experimental implementation of the load balancing strategy on a parallel computer [29]. Experiments also find that the feedback gain in the controller affects the performance of load balancing [30]. It is desirable to choose the fraction of excess load (to be transferred to other nodes) to be close to 0.5 (feedback gain) to achieve a faster response without breaking into oscillatory behavior [51]. This model does not account for the fact that load distribution and task processing cannot be carried out simultaneously.

Nonlinear Model with Resource Constraints

Load distribution and task processing contend for the same resources on each computational element. A deterministic dynamic nonlinear system has been developed in this research to model load balancing for parallel computations incorporating both time

delays and resource constraints. This model accounts for the trade-off between using processor resources to process tasks and the advantage of distributing the load evenly between the nodes to reduce overall processing time. The model is presented with a proof of model consistency as well as open-loop experiments in [196]. It is also shown that the open-loop system is Lyapunov stable, but not asymptotically stable. The guarantee of asymptotic stability is lost, compared to the previous nonlinear model, because it is possible for the local controllers to spend all available time transferring tasks rather than processing tasks.

A distributed closed-loop controller is proposed to balance load dynamically at each node by using not only local estimates of loads at other nodes, but also estimates of the number of tasks in transit to it [195]. Experiments performed on a parallel computing environment demonstrate substantial improvements in performance using a controller based on anticipated estimates of workloads compared to a controller based on local workloads only.

Parallel Database with Integrated Load Balancing

This dissertation addresses the design and an implementation of a parallel DNA database, which is the motivation and a practical application of load balancing strategies. DNA profiles are stored in portions on every node of the parallel computer to form distributed databases to search in parallel. A DNA search engine running on each node is based upon the decision-tree based search method as described in [33]. A parallel search server using multiple threads, PVM and object serialization is presented,

demonstrating two orders of magnitude improvements in the performance of transaction time over a search server using a publicly available socket library. A parallel database with integrated load balancing is presented as well. Experimental results on the parallel DNA database with integrated load balancing demonstrate both the efficiency of the multi-threaded parallel search server and the efficacy of the load balancing strategy. Initial results for parallel searches with load balancing integrated with a parallel DNA database are documented in [197].

1.5 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter 2 provides a concise statement of the problem and develops the system model for load balancing in a parallel computer architecture with time delays. Chapter 3 shows several evolutions of load balancing models and their related analyses with comparisons and discussion. The design of the parallel DNA database is discussed in Chapter 4, which motivates this research and provides a practical application of load balancing strategies. Experimental results are shown in Chapter 5. These results demonstrate the efficiency of load balancing strategies and the performance on a parallel DNA database. Chapter 6 summarizes the material and provides directions for possible future research.

Chapter 2

Problem Formulation

This chapter formulates the load balancing problem for parallel computing in the presence of time delays. Section 2.1 describes the load balancing problem to be studied, and illustrates the problem using a flow control model. Section 2.2 describes the setup of a parallel machine — a lab-built experimental facility for development and evaluation of load balancing strategies. Parameter values for delays and various packet sizes for transfer of data between nodes are determined by experiments on the parallel machine.

2.1 Problem Statement

A load balancing method is a key ingredient of an efficient implementation of an application on a parallel computer. The computational load needs to be distributed *more or less* evenly across the nodes to obtain the best achievable performance. The qualifier “more or less” is used because the communications required to distribute the load

consume both computational resources and network bandwidth. A point of diminishing returns exists.

To adequately model load balancing problems, several features of the parallel computation environment must be captured: (1) The workload awaiting processing at each computational element (CE); (2) the relative performance of the CEs; (3) the computational requirements of each workload component; (4) the delays and bandwidth constraints of CEs and network components involved in the exchange of workloads and, (5) the delays imposed by CEs and the network on the exchange of measurements. A queuing theory approach is well-suited to the modeling requirements and has been used in the literature by Spies [193], Nelson [164] and others. However, whereas Spies *et al.* assume a homogeneous network of CEs and model the queues in detail, the work as described in this dissertation generalizes the queue length to an expected waiting time, normalizing to account for differences among CEs, and aggregates the behavior of each queue. The load balancing problem for parallel computations with time delays is stated as follows.

Problem 1 *Given n queues containing requests for processing in a parallel network consisting of n nodes, each characterized by expected waiting time of a request inserted at the tail of the queue, minimize the overall completion time (the maximum completion time over all nodes in the network) for processing all jobs, subject to the constraints imposed by time delays that occur when exchanging queue sizes between nodes and when transferring tasks from one node to another.*

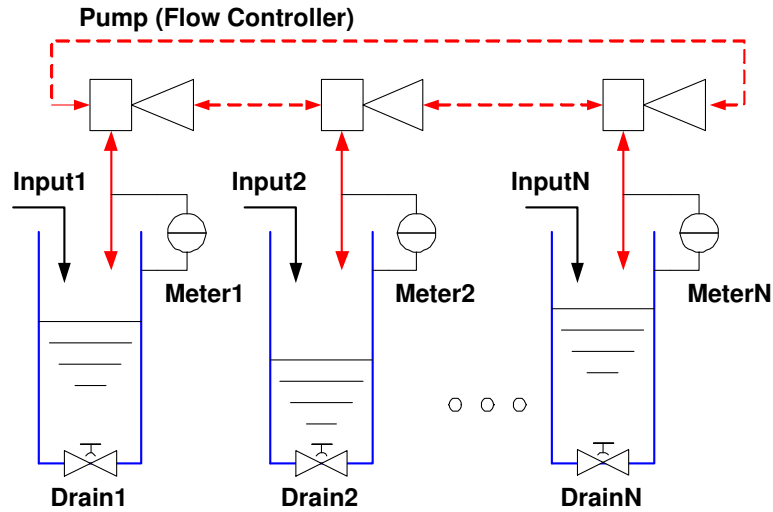


Figure 2.1: Model of a flow control system. Each tank has an input and a drain valve, and connects to each other through pipes. A pump (flow controller) injects water to or ejects water from each tank, depending on the relative level to the average as measured by a meter, to make all tanks run out of water at the same (minimum) time.

To illustrate the load balancing problem, a good analogy in control systems is a flow control model, as described in the following. Figure 2.1 shows a model of a set of water tanks. In the figure, all tanks are connected through pipes. Each tank has a flow input and a drain valve. The valves are not all in the same size; drain rates are different. Flows may arrive from external sources, or be generated (for example, by some reactions) in each tank. Each tank connects with a pump which injects water to or ejects water from the tank, depending on the relative level to the average as measured by a meter. The objective is to make all tanks run out of water at the same (minimum) time. The flow control model in Figure 2.1 is an interesting example of load (water level) balancing. The difficulty for a fine-grained balancing problem is there are delays that occur when

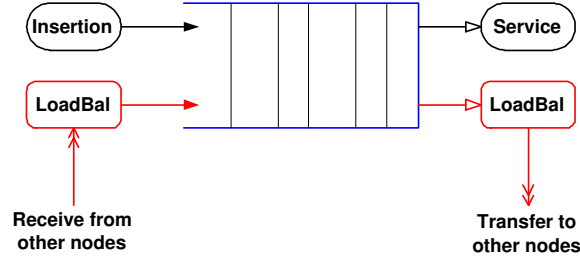


Figure 2.2: A queue model at each node for task processing and load balancing. Tasks are inserted to the queue, and removed for service. Excess load is to be transferred to other nodes according to a local load balancing controller.

estimating water levels of other tanks and when pumping water from one tank to the other, and the delays may affect the balancing performance.

In load balancing for parallel computations, each node may broadcast (or multicast) its queue size to all other nodes, calculate the relative load to its *estimate*¹ of the network average, and transfer the excess load (if any) to other nodes to obtain a balance of computational load in the network. Figure 2.2 shows a queue model for task insertion and service and load balancing at each node in a parallel network. While each node inserts received and generated tasks to a local queue and services them, a local controller (load balancer) at each node determines whether the node has a excess load and how much should be transferred to each of the other nodes. A recipient node continues processing after receiving the tasks transferred from other nodes.

Note that a controller on any node has only estimates of loads at other nodes due to *communication delays* that occur when exchanging queue sizes between nodes and

¹It is an estimate because at any time, each node only has a delayed value of the number of tasks in the other nodes.

transfer delays that occur when transferring tasks from one node to the other. In the parallel DNA database application, each search task requires time of the order of 1 *ms* and, for this reason, delays between nodes are important. This research focuses upon the effects of delays in the exchange of information among nodes, and the constraints these effects impose on the design of a load balancing strategy. Continuous state models are developed to study the load balancing problem for parallel computations with time delays in Chapter 3. A quantitative determination of time delays in a parallel computer network is given in the next section.

2.2 Parallel Machine Measurement and Parameterization

A parallel machine has been built in the LIT lab as an experimental facility for development and evaluation of load balancing strategies. This section discusses the experimental setup of the parallel machine. Experiments and measurements are conducted on the parallel machine to obtain values of parameters in load balancing strategies.

2.2.1 Parallel Machine Setup

The design of the parallel machine is shown in Figure 2.3. A root node communicates with k groups of networked computers. Each of these groups is composed of n nodes (hosts) containing identical data. Within each node, there are either one or two processors. Within each group, any task can be performed by any node, and it is advantageous to distribute load evenly among the nodes to minimize the overall waiting time.

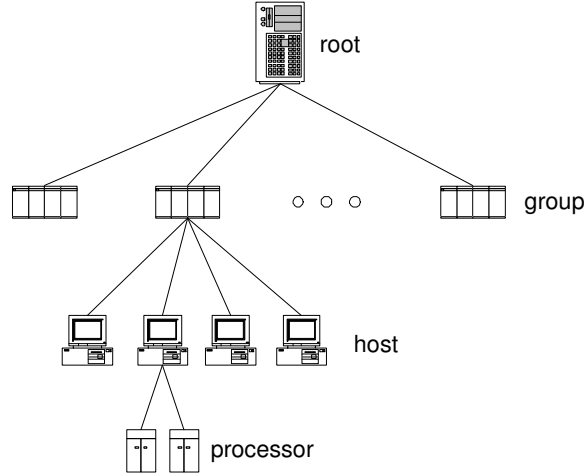


Figure 2.3: Hardware structure of the parallel machine.

Figure 2.4 shows the parallel machine consisting of 2 server nodes and 24 computing nodes.

In the experimental facility, the parallel machine has a root node and a file server node, both of which have dual processor Intel Pentium 4 Xeon architectures. The root node provides communications to other computers within the laboratory and the Internet. The file server node provides approximately 700 Gbytes of file storage with a redundant array of eight drives configured as RAID5 (distributed parity for fault tolerance), which is available to all other nodes of the parallel machine using the Network File System. The parallel machine has, in addition, 24 computer nodes, each of which has 100 Gbytes of local storage (for files and databases). Among these computers, 16 nodes are dual processor nodes with 1 Gbyte of memory each, and the other 8 are single processor nodes with 768 Mbytes of memory each. The dual processor machines use



Figure 2.4: A lab built parallel machine consisting of 2 server nodes and 24 computing nodes. Photo courtesy of J. D. Birdwell [27].

two 1.4 GHz Athlon MP processors, and the single processor machines use a 1.33 GHz Athlon processor each. All run the Linux operating system and are connected using a Foundry Networks FastIron II Gigabit Ethernet switch with non-blocking switching and a backup Netgear Fast Ethernet switch. The interest here is in load balancing any one group.

2.2.2 Parameter Values

In this section, the determination of the parameters on delays and transfer packets for load balancing strategies is discussed.

Congestion in network switches can limit the communication traffic between Parallel Virtual Machine (PVM) [90] nodes in a parallel computation. Prior research in the LIT

lab [63][64] introduced a new benchmark to evaluate the performance of PVM in various networking environments. The benchmark was used to achieve a better understanding of performance limitations in parallel computing that are imposed by the choice of the network. A characterization of network delays and throughput was developed to provide a basis for an informed decision on the networking methods and topology for a parallel DNA database. Experiments are performed on the parallel computer described above to determine the delays incurred when exchanging information between nodes. Round-trip times for messages to be exchanged between nodes are measured, and the average is calculated from 20 repeated measurements for each message size. Experimental procedures found representative values for delays in a Gigabit network for a group consisting of three nodes are $\tau_{ij} = 200 \mu s$ for $i \neq j$, and $\tau_{ii} = 0$. The $\tau_{ii} = 0$ denote that the local queue size can be achieved on each node with negligible delay.

Experiments are also performed to determine the appropriate packet size for data transfers between nodes. Figure 2.5 shows the experimentally determined time to transfer data from one node to another in μs as a function of message size in integers (of 4 bytes). Each host measures the average, minimum and maximum times required to exchange data between itself and the other node in the PVM environment. The node starts the timer when initiating a transfer and stops the timer when it receives the returned data, the round-trip transfer time is measured. This also avoids the problem of synchronizing clocks on two different machines. The data sizes vary from 4 bytes to 4 Mbytes. In order to ensure that anomalies in message timings are minimized, the tests

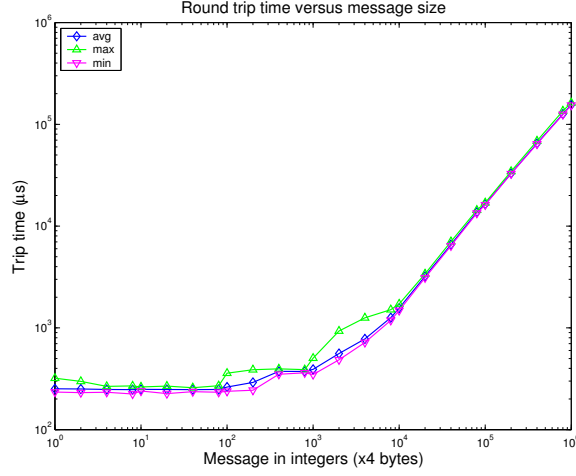


Figure 2.5: Round trip time versus amount of data transfered in bytes.

are repeated 20 times for each message size. The figure shows that increase noticeable for packets of more than about 10^3 integers (4×10^3 bytes) and scale linearly with packet size for packets in excess of 10^4 integers (4×10^4 bytes).

Figure 2.6 shows the experimentally measured bandwidth in Mbps versus the message size in integers. Based on the data, a packet size of about 4×10^4 bytes (10^4 integers) or larger for data transfer should be selected to maximize the data transfer rate (about 400 Mbps). Messages of larger sizes do not improve the bandwidth, but overhead reduces achievable bandwidth for smaller packets. Meanwhile, very high bandwidth means the system must yield computing time for communication time. From these figures, it is apparent that large packet sizes are desirable to maximize data transfer rates, but that excessively large sizes cause large data transfer delays. Here the threshold for data transfer is chosen to be 4×10^3 bytes (10^3 integers) since, as Figure 2.5 shows, this is

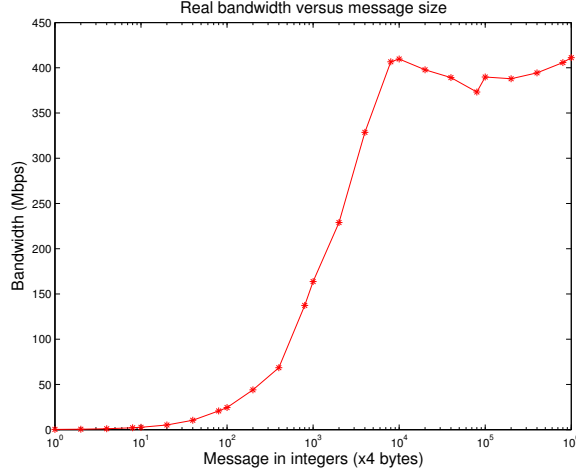


Figure 2.6: Measured bandwidth versus amount of data transfered in bytes.

a trade-off between bandwidth and transfer time. With a typical task (DNA search request) to be transferred of size 400 bytes/task (3,200 bits/task), this means that the threshold is 10 tasks. Note that this threshold for data transfer is determined under the assumption that a search task requires little time in the parallel DNA database application.

2.3 Summary

This chapter provides a concise statement of the problem of load balancing for parallel computations in the presence of time delays. Delays occur when exchanging queue information between nodes and when transferring task data from one node to the other. A parallel machine has been built for development and evaluation of load balancing strategies, and the setup of the parallel machine is presented. Parameter values for

delays in load balancing and various packet sizes for transfer of data between nodes are determined by experiments performed on the parallel machine.

Chapter 3

Load Balancing System Model

In this chapter, continuous time models are developed to model load balancing among a network of computers. A linear model is first described in Section 3.1 to model load balancing strategies that incorporate the presence of delays in communicating between nodes and transferring tasks. For this case, the load balancing strategy is simplified to a linear time-invariant system with delays, and the stability of the resulting closed-loop system is discussed. Two difficulties with the linear model are observed: instabilities above certain thresholds in controller gains, and negative workload queue sizes. To address these problems, a nonlinear time delay model is proposed in Section 3.2 that does not allow the queues to go negative and the number of tasks a node distributes to the other nodes is based on delayed measurements of relative load levels. It is proved that the model preserves non-negativity of queue lengths and is guaranteed to be asymptotically stable. In addition, it is shown that the total number of tasks in all

the queues and in transit is conserved by the load balancing strategy.

Section 3.3 presents a third approach to modeling load balancing strategies in the presence of delays that incorporates processor resource constraints. This model is used to analyze the trade-off in each processor between processing tasks and transferring load. The model evolved from a recognition that a processor can not perform both functions simultaneously. The proposed nonlinear model correctly predicts that the queue lengths are non-negative and that the total number of tasks in all the queues is conserved by the load balancing strategy. It is shown that the (open-loop) model is always stable, but not necessarily asymptotically stable. The design of the feedback control law on a local node and how a node decides to portion out its tasks to the other nodes are addressed. Finally, the differences between feedback controllers based on the local queue information and on the *anticipated* queue information are discussed.

In all cases, a unique feature of this research is its focus upon the load balancing problem from a control system design perspective, and upon extensive evaluation of load balancing strategies through experiments in a parallel computation environment. The experimental components of this research have, to a significant degree, driven the continued evolution of the parallel computation model and the load balancing strategies. For completeness, the model development and the theoretical analysis documented in this chapter include contributions by J. D. Birdwell and J. Chiasson.

3.1 Linear Model

To provide a control system designer's perspective of the load balancing problem, consider a computing network consisting of n computers (nodes) all of which can communicate with each other. The critical feature of the load balancing problem is the delayed receipt of information and transferred load. In control system design problems, the limitations on achievable performance imposed by time delays are well-known [131][49][50]. These limitations are observed, both in theory and experiments, in this distributed control problem. At start up, the computers can be assigned an equal number of tasks. However, when a node executes a particular task it can in turn generate more tasks so that very quickly the loads on various nodes become unequal. To balance the loads, each computer in the network sends its queue size $q_j(t)$ to all other computers in the network. A node i receives this information from node j *delayed* by a finite amount of time τ_{ij} ; that is, it receives $q_j(t - \tau_{ij})$. Each node i then uses this information to compute its local estimate of the average number of tasks in the queues of the n computers in the network.

In this section, a load balancing network of computing nodes is abstracted as a linear time-delay system. With an assumption of symmetric nodes, the stability of the load balancing system is analyzed using system theory. In this model, the simple estimator $\left(\sum_{j=1}^n q_j(t - \tau_{ij})\right) / n$ ($\tau_{ii} = 0$), which is based on the most recent observations, is used. Node i then compares its queue size $q_i(t)$ with its estimate of the network average as $\left(q_i(t) - \left(\sum_{j=1}^n q_j(t - \tau_{ij})\right) / n\right)$, and, if this is greater than zero, the node sends some

of its tasks to the other nodes. Further, the tasks sent by node i are received by node j with a transfer delay h_{ij} . The controller (load balancer) decides how often and quickly to do load balancing (transfer tasks among the nodes) and how many tasks are to be sent to each node.

As explained, each node controller (load balancer) has only *delayed* values of the queue lengths of the other nodes, and each transfer of data from one node to another is received only after a finite time delay. An important issue considered here is the effect of these delays on system performance. Specifically, a continuous time model is developed to capture the effect of the delays in load balancing techniques, and system theoretic methods are used to analyze them.

The mathematical model of a node is given by the following linear differential system of time delays in [2]

$$\begin{aligned}\frac{dx_i(t)}{dt} &= \lambda_i - \mu_i + u_i(t) - \sum_{j \neq i} p_{ij} u_j(t - h_{ij}) \\ y_i(t) &= x_i(t) - \frac{\sum_{j=1}^n x_j(t - \tau_{ij})}{n} \\ u_i(t) &= -K_i y_i\end{aligned}\tag{3.1}$$

where

- $x_i(t)$ is the *expected waiting time* experienced by a task inserted into the queue of the i^{th} node. With $q_i(t)$ the number of *tasks* in the i^{th} node and t_{p_i} the average

time needed to process a task on the i^{th} node, the expected (average) waiting time is then given by $x_i(t) = q_i(t)t_{p_i}$. Note that $x_j/t_{p_j} = q_j$ is the number of tasks in the node j queue. If these tasks were transferred to node i , then the waiting time transferred is $q_j t_{p_i} = x_j t_{p_i}/t_{p_j}$, so that the fraction t_{p_i}/t_{p_j} converts waiting time on node j to waiting time on node i .

- $\lambda_i \geq 0$ is the rate of generation of waiting time on the i^{th} node caused by the addition of tasks (rate of increase in x_i)
- $\mu_i \geq 0$ is the rate of reduction in waiting time caused by the service of tasks at the i^{th} node and is given by $\mu_i = 1$ for all i if $x_i(t) > 0$. If $x_i(t) = 0$ then $\mu_i \triangleq 0$; that is, if there are no tasks in the queue, then the queue cannot possibly decrease.
- $u_i(t)$ is the rate of removal (transfer) of the tasks from node i at time t by the load balancing strategy at node i .
- τ_{ij} is the time delay for communicating the node waiting time x_j to node i ($\tau_{ii} = 0$).
- p_{ij} is the fraction of $u_j(t)$ that node j allocates to node i at time t ; $\sum_{i=1}^n p_{ij} = 1$, ($p_{jj} = 0$).
- h_{ij} is the time delay for task transfer from node j to node i ($h_{ii} = 0$).
- $p_{ij}u_j(t - h_{ij})$ is the rate of removal (transfer) of tasks at time t from node j by (to) node i .
- n is the number of nodes.

All rates are in units of the rate of change of expected waiting time (or time/time, which is dimensionless). This allows nodes to have different performance and is consistent with the implementation and maintenance of a large parallel machine whose nodes may from time to time be replaced. Here $u_i(t) < 0$ means tasks are being sent to other nodes, while $u_i(t) > 0$ means the i^{th} node is receiving tasks from other nodes. A delay h_{ij} is experienced by transmitted tasks before they are received at the other node. The model accurately depicts the behavior of a parallel computing environment when $u_i(t) < 0$, but not when $u_i(t) > 0$. In particular, with $u_i(t) > 0$ the increase in tasks for the i^{th} node is seen immediately, but the decreases in the other nodes where these tasks originated are delayed due to communication times. For the purposes of this analysis, this modeling inaccuracy was considered acceptable since the actual behavior is nonlinear.

The control law $u_i(t) = -K_i y_i$ says that if the i^{th} node output $x_i(t)$ is above the local average $\left(\sum_{j=1}^n x_j(t - \tau_{ij})\right) / n$ then send data to the other nodes, while if it is less than the local average, accept data from the other nodes.

Often, the p_{ij} are functions of the state x_i so as to send a higher fraction of the data to those nodes that have less tasks. However, this is left out in this model to retain linearity of the system so that a stability analysis can be carried out.

3.1.1 Stability Analysis

The stability of this linear time-delay model was first analyzed in [2]. To study the stability of the linear model, a three node ($n = 3$) model is considered with $K_1 = K_2 =$

$K_3 = K$, $p_{ij} = 1/2$, for all i, j , $\tau_{ij} = \tau$ for $i \neq j$, $\tau_{ii} = 0$, $h_{ij} = 2\tau$ for $i \neq j$, $h_{ii} = 0$.

Letting $d_1 = \lambda_1 - \mu_1$, $d_2 = \lambda_2 - \mu_2$, and $d_3 = \lambda_3 - \mu_3$, the Laplace transform of the output response $y_1(t)$ from (3.1) with zero initial conditions is

$$Y_1(s) = \frac{b_1(s, z)}{a_1(s, z)a_2(s, z)}D_1(s) + \frac{zb_2(s, z)}{a_1(s, z)a_2(s, z)}(D_2(s) + D_3(s)) \quad (3.2)$$

where $z \triangleq e^{-\tau s}$ and

$$a_1(s, z) = 3s + K(2 + z)(1 + z^2/2)$$

$$a_2(s, z) = -3s + 2K(1 - z)(-1 + z^2)$$

$$b_1(s, z) = -6s - K(z^2 - 2)(z - 1)(z + 2)$$

$$b_2(s, z) = 3s + Kz(z - 1)(z + 2).$$

The stability of this time delay system is analyzed using the techniques developed in [20][131][111][52][49][3][50]. The range of delay values τ for which (3.2) is stable can be found by separately considering the stability of the transfer functions $1/a_1(s, z)$, $b_1(s, z)/a_2(s, z)$, and $b_2(s, z)/a_2(s, z)$. It is shown in [1] that

$$1/a_1(s, e^{-\tau s}) \text{ is stable for all } \tau \geq 0$$

$$b_1(s, e^{-\tau s})/a_2(s, e^{-\tau s}) \text{ is stable for } \tau < \frac{5\pi}{4K \sin(\pi/3)}$$

$$b_2(s, e^{-\tau s})/a_2(s, e^{-\tau s}) \text{ is stable for } \tau < \frac{5\pi}{4K \sin(\pi/3)}.$$

That is, for a given K , this gives the smallest positive value of the delay for the transfer functions have a zero in the closed right-half plane. Or, in terms of the gain K , the system is stable for

$$K < K_{max} \triangleq \frac{5\pi}{4\tau \sin(\pi/3)},$$

and unstable for $K \geq K_{max}$.

3.1.2 Brief Summary

The model (3.1) is a good attempt to abstract a real load balancing system as a linear time-delay system. Under the assumption of symmetric nodes and controllers (all intercommunication delays are identical, and the controller gains are identical), a systematic procedure was presented to determine the stability of the system by an explicit relationship between the delay values and the control gain. The delays create a limit on the size of the controller gains in order to ensure stability.

In the model (3.1), when $u_i(t) = -K_i y_i(t) < 0$, the controller operates in an intuitive fashion by immediately removing tasks and sending them to the other nodes, where each of those nodes experiences a delay (h_{ij}) in getting these tasks. However, there is a fundamental problem with this linear model. When $y_i(t) < 0$, the controller (load balancer) calculates $u_i(t) = -K_i y_i(t) > 0$; that implies the node is *instantaneously* taking on waiting times (tasks) from the other nodes before those tasks are removed from the other nodes' queues. That is, it is accepting the waiting times (tasks) at the rate $p_{ij} u_j(t)$ from each of the other nodes. There is a finite time delay associated with

this transfer of tasks, and this model ignores this fact. Interestingly, non-negativity of the queues cannot be ensured for the linear model, and it can go unstable for sufficiently large gains K_i .

In other words, a node would not initiate a negative amount of task transfer to a recipient node in a load balancing system. Therefore, the feedback needs to include a saturation of the amount of local workload relative to the estimated network average. In addition, the model (3.1) did not explicitly specify the amount p_{ij} to portion out excess tasks. For example, the p_{ij} were considered constant and equal in the above linear model. It can be useful to use the local information on the waiting times to set the values of the p_{ij} .

3.2 Nonlinear Model with Proportional Feedback Control

To take into account the facts that there is a finite time delay associated with task transferring and a negative amount of tasks can not be transferred to a node, a deterministic dynamic nonlinear time delay model is developed as the following nonlinear differential-delay system [29]

$$\begin{aligned}
\frac{dx_i(t)}{dt} &= \lambda_i - \mu_i + u_i(t) - \sum_{j=1}^n p_{ij} \frac{t_{p_i}}{t_{p_j}} u_j(t - h_{ij}) \\
y_i(t) &= x_i(t) - \frac{\sum_{j=1}^n x_j(t - \tau_{ij})}{n} \\
u_i(t) &= -K_i \text{sat}(y_i(t)),
\end{aligned} \tag{3.3}$$

where $p_{ij} \geq 0$, $\sum_{i=1}^n p_{ij} = 1$, $p_{jj} = 0$, and $\text{sat}(\cdot)$ is a saturation function given by

$$\text{sat}(y) = y \text{ if } y \geq 0, \text{ and } 0 \text{ otherwise.}$$

In this model

- $x_i(t)$ is the expected waiting time at the i^{th} node. λ_i is the rate of increase waiting time, and μ_i is the rate of reduction in waiting time at the i^{th} node. $u_i(t)$ is the rate of removal (transfer) of the tasks from node i at time t by the load balancing strategy at node i . Note that $u_i(t) \leq 0$.
- The quantity $x_i^{avg} = \left(\sum_{j=1}^n x_j(t - \tau_{ij}) \right) / n$ is the *estimate* (due to the delays) by the i^{th} node of the average waiting time of the network, where τ_{ij} is the time delay for communicating the expected waiting time x_j from node j to node i . The local estimate of the average, x_i^{avg} , is referred to as the *local average*.
- The quantity $p_{ij}u_j(t)$ is the rate at which node j sends waiting time (tasks) to node i at time t , where $\sum_{i=1}^n p_{ij} = 1$, $p_{ij} \geq 0$, and $p_{jj} = 0$. That is, the transfer from node j of expected waiting time (tasks) $\int_{t_1}^{t_2} u_j(t)dt$ in the interval of time $[t_1, t_2]$ to the other nodes is carried out with the i^{th} node receiving the fraction $p_{ij} (t_{p_i}/t_{p_j}) \int_{t_1}^{t_2} u_j(t)dt$ where the ratio t_{p_i}/t_{p_j} converts the task from waiting time on node j to waiting time on node i . As $\sum_{i=1}^n \left(p_{ij} \int_{t_1}^{t_2} u_j(t)dt \right) = \int_{t_1}^{t_2} u_j(t)dt$, this results in removing *all* of the waiting time $\int_{t_1}^{t_2} u_j(t)dt$ from node j .
- The quantity $-p_{ij}u_j(t - h_{ij})$ is the rate of increase (rate of transfer) of the expected

waiting time (tasks) at time t from node j by (to) node i , where h_{ij} is the time delay for the task transfer from node j to node i .

In this model, as $u_i(t) \leq 0$, node i can only send tasks to other nodes and cannot initiate transfers from another node to itself. The control law $u_i(t) = -K_i \text{sat}(y_i(t))$ states that if the i^{th} node output $x_i(t)$ is above the local average $\left(\sum_{j=1}^n x_j(t - \tau_{ij})\right) / n$, then it sends data to the other nodes, while if it is less than the local average nothing is sent. The j^{th} node receives the fraction $\int_{t_1}^{t_2} p_{ji} (t_{p_i} / t_{p_j}) u_i(t) dt$ of transferred waiting time $\int_{t_1}^{t_2} u_i(t) dt$ delayed by the time h_{ij} .

3.2.1 Load Transfer Portions

The quantity p_{ij} defines how the controller on node j portions out its excess tasks to other nodes. One approach is to select p_{ij} that are constant and equal:

$$p_{ij} = 1/(n-1) \text{ for } j \neq i \quad (3.4)$$

$$p_{ij} = 0,$$

where it is clear that $p_{ij} \geq 0$, $\sum_{i=1}^n p_{ij} = 1$.

Another approach is to use the measured information about the waiting times $x_i(t)$, $i = 1, \dots, n$ to set their values. Recall that p_{ij} is the fraction of $u_j(t)$ that node j allocates (transfers) to node i at time t , $p_{ij} \geq 0$, $\sum_{i=1}^n p_{ij} = 1$, and $p_{jj} = 0$. The quantity $x_i(t - \tau_{ji}) - x_j^{\text{avg}}$ represents what node j estimates the waiting time in the queue of node i is with respect to the local average of node j . If the queue of node i is above the

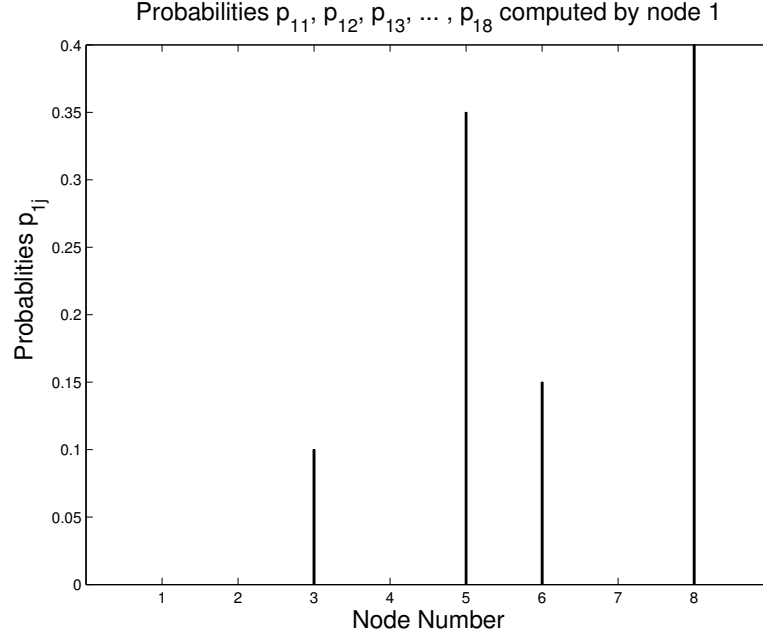


Figure 3.1: Illustration of a hypothetical distribution p_{i1} of the load at some time t from node 1's point of view. Node 1 will send data out to node i in proportion p_{i1} when it estimates node i is below the average, where $\sum_{i=1}^n p_{i1} = 1$ and $p_{11} = 0$.

local average, then node j does not send tasks to it. Therefore $\text{sat}\left(x_j^{avg} - x_i(t - \tau_{ji})\right)$ is an appropriate measure by node j as to how much node i is *below* the local average. Node j then repeats this computation for all the other nodes and portions out its tasks among the other nodes according to the amounts they are below the local average; that is,

$$p_{ij} \triangleq \frac{\text{sat}\left(x_j^{avg} - x_i(t - \tau_{ji})\right)}{\sum_{i \ni i \neq j} \text{sat}\left(x_j^{avg} - x_i(t - \tau_{ji})\right)}. \quad (3.5)$$

If the denominator $\sum_{i \ni i \neq j} \text{sat}\left(x_j^{avg} - x_i(t - \tau_{ji})\right) = 0$, then the p_{ij} are defined to be zero, and no load is transferred. This is illustrated in Figure 3.1 for node 1.

If the denominator

$$\sum_{i \ni i \neq j} \text{sat} \left(x_j^{avg} - x_i(t - \tau_{ji}) \right)$$

is zero, then $x_j^{avg} - x_i(t - \tau_{ji}) \leq 0$ for all $i \neq j$. However, by definition of the average,

$$\begin{aligned} \sum_{i \ni i \neq j} \left(x_j^{avg} - x_i(t - \tau_{ji}) \right) + x_j^{avg} - x_j(t) \\ = \sum_i \left(x_j^{avg} - x_i(t - \tau_{ji}) \right) = 0, \end{aligned}$$

which implies

$$x_j^{avg} - x_j(t) = - \sum_{i \ni i \neq j} \left(x_j^{avg} - x_i(t - \tau_{ji}) \right) > 0.$$

That is, if the denominator is zero, the node j is not greater than the local average, so $u_j(t) = -K_j \text{sat}(y_j(t)) = 0$, and therefore node j is not sending out any tasks.

3.2.2 Model Consistency

This model has been shown to be consistent with load balancing systems in that the queue lengths cannot go negative, and the load balancing strategy cannot create or destroy tasks; it can only move tasks between nodes [28]. Note that in all that follows, the only assumptions on the p_{ij} are that $p_{ij} \geq 0$, and $\sum_{i=1}^n p_{ij} = 1$.

3.2.2.1 Non-Negativity of Queue Lengths

Recall that the queue length of each node is given by $q_i(t) = x_i(t)/t_{p_i}$. The model (3.3) can be rewritten in terms of these quantities as

$$\frac{d}{dt} \left(x_i(t)/t_{p_i} \right) = \frac{\lambda_i - \mu_i}{t_{p_i}} + \frac{1}{t_{p_i}} u_i(t) - \sum_{j=1}^n \frac{p_{ij}}{t_{p_j}} u_j(t - h_{ij}). \quad (3.6)$$

Given that $x_i(0) > 0$ for all i , it follows from the right-hand side of (3.6) that $q_i(t) = x_i(t)/t_{p_i} \geq 0$ for all $t \geq 0$ and all i . To see this, suppose without loss of generality that $q_i(t) = x_i(t)/t_{p_i}$ is the first queue to go to zero, and let t_1 be the time when $x_i(t_1) = 0$. At the time t_1 , $\lambda_i - \mu_i = \lambda_i \geq 0$ by the definition of μ_i and $-\sum_{j=1}^n \frac{p_{ij}}{t_{p_j}} u_j(t - h_{ij}) \geq 0$ for all time by the definition of the u_j . Further, the term $u_i(t_1)$ is negative only if

$$x_i(t_1) > \left(\sum_{j=1}^n x_j(t_1 - \tau_{ij}) \right) / n. \quad (3.7)$$

By supposition (up to time t_1) all the $x_j(t_1 - \tau_{ij}) > 0$ for $j \neq i$ and $x_i(t_1) = 0$, so that $u_i(t_1) = 0$ as the right side of (3.7) is positive at time t_1 . Consequently, at time t_1 all terms on the right-hand side of (3.6) are non-negative. Further, $x_i(t)$ cannot go negative in a neighborhood of t_1 . For if it did, as the right-hand side of (3.7) is continuous, it follows that

$$x_i(t) < 0 < \left(\sum_{j=1}^n x_j(t - \tau_{ij}) \right) / n \quad (3.8)$$

for some $t \in (t_1, t_1 + \delta)$ with $\delta > 0$. Therefore, $u_i(t) = 0$ for all $t \in [t_1, t_1 + \delta]$ and the right-hand side of (3.7) is non-negative for all $t \in [t_1, t_1 + \delta]$, which contradicts $x_i(t) < 0$. Note that $t_1 + \delta$ can be taken to be at least as large as the time at which some x_k goes to zero; that is, $q_k(t_1 + \delta) = 0$, as the right hand side of (3.8) must remain positive for $t \in [t_1, t_1 + \delta]$.

If $x_i(t)$ goes positive after t_1 , then the above argument is repeated at the next time a queue goes to zero. If $x_i(t)$ remains identically zero in the interval $(t_1, t_1 + \delta)$, then the argument is also similar in that at time $t_1 + \delta$, both $x_i(t_1 + \delta)$ and $x_k(t_1 + \delta)$ are then zero. As the remaining $n - 2$ nodes are still positive, the right-hand side of equation (3.8) continues to hold with both x_i and x_k zero at time $t_1 + \delta$, and again a contradiction is obtained if either x_i or x_k goes negative in an interval $(t_1 + \delta, t_2)$. Continuing in this manner, it follows that $q_i(t) = x_i(t)/t_{p_i}$ cannot go negative for all i .

3.2.2.2 Conservation of Queue Lengths

It is now shown that the total number of tasks in all the queues and the network is conserved. To do so, sum up equations (3.8) from $i = 1, \dots, n$ to obtain

$$\begin{aligned} \frac{d}{dt} \left(\sum_{i=1}^n q_i(t) \right) &= \sum_{i=1}^n \left(\frac{\lambda_i - \mu_i}{t_{p_i}} \right) + \sum_{i=1}^n u_i(t)/t_{p_i} \\ &\quad - \sum_{i=1}^n \sum_{j=1}^n \frac{p_{ij}}{t_{p_j}} u_j(t - h_{ij}), \end{aligned} \quad (3.9)$$

which is the rate of change of the total queue lengths on all the nodes. However, the network itself also contains tasks in transit between nodes. The dynamic model of the queue lengths in the network is given by

$$\frac{d}{dt} q_{net_i}(t) = \sum_{j=1}^n \frac{p_{ij}}{t_{p_j}} u_j(t - h_{ij}) - \sum_{j=1}^n \frac{p_{ij}}{t_{p_j}} u_j(t), \quad (3.10)$$

where q_{net_i} is the number of tasks put on the network that are being sent to node i .

This equation says that the j^{th} node is placing tasks on the network to be sent to node i at the rate $\frac{p_{ij}}{t_{p_j}}u_j(t)$, while the i^{th} node is removing the tasks transferred from node j from the network at the rate $\frac{p_{ij}}{t_{p_j}}u_j(t - h_{ij})$. Summing (3.10) over all the nodes, one obtains

$$\begin{aligned} \frac{d}{dt} \left(\sum_{i=1}^n q_{net_i}(t) \right) &= \sum_{i=1}^n \sum_{j=1}^n \frac{p_{ij}}{t_{p_j}} u_j(t - h_{ij}) - \sum_{i=1}^n \sum_{j=1}^n \frac{p_{ij}}{t_{p_j}} u_j(t) \\ &= \sum_{i=1}^n \sum_{j=1}^n \frac{p_{ij}}{t_{p_j}} u_j(t - h_{ij}) - \sum_{j=1}^n \frac{u_j(t)}{t_{p_j}}. \end{aligned} \quad (3.11)$$

Adding (3.9) and (3.11), one obtains the conservation of queue lengths given by

$$\frac{d}{dt} \sum_{i=1}^n \left(q_i(t) + q_{net_i}(t) \right) = \sum_{i=1}^n \left(\frac{\lambda_i - \mu_i}{t_{p_i}} \right). \quad (3.12)$$

In words, the total number of tasks which are in the system (i.e., in the nodes or in the network) can increase only by the rate of arrival of tasks $\sum_{i=1}^n \lambda_i/t_{p_i}$ at all the nodes, or similarly, decrease by the rate of processing of tasks $\sum_{i=1}^n \mu_i/t_{p_i}$ at all the nodes. Load balancing itself cannot increase or decrease the total number of tasks in all the queues and the network.

3.2.3 Saturation Controller

The controller in the model (3.3) is

$$u_i(t) = -K_{i\text{sat}}(y_i(t)), \quad (3.13)$$

where the gains $K_i > 0$ are to be specified. Physically, these gains are limited by the bandwidth constraints in the network. One can also view the p_{ij} as controller parameters to be specified subject to the constraints given above. It is now shown that system (3.3) is asymptotically stable in the sense of Lyapunov for any set of gains $K_i > 0$ and any set of $p_{ij} \geq 0$ with $\sum_{i=1}^n p_{ij} = 1$.

3.2.3.1 Asymptotic Stability of the Controller

The stability analysis for this controller was first given in [28]. For the system described by (3.3) and (3.10) with $\lambda_i = 0$ for $i = 1, \dots, n$ and initial conditions $x_i(0) \geq 0$, $(q_i(t), q_{net_i}(t)) \rightarrow 0$ as $t \rightarrow \infty$.

Recall that the q_{net_i} are non-negative as from (3.10), it follows that

$$q_{net_i}(t) = - \sum_{j=1}^n \frac{p_{ij}}{t_{p_j}} \left(\int_{t-h_{ij}}^t u_j(\tau) d\tau \right) \geq 0. \quad (3.14)$$

Under the given conditions, equation (3.12) becomes

$$\frac{d}{dt} \sum_{i=1}^n (q_i(t) + q_{net_i}(t)) = - \sum_{i=1}^n \frac{\mu_i(q_i)}{t_{p_i}}. \quad (3.15)$$

Let $V(t) \triangleq \sum_{i=1}^n (q_i(t) + q_{net_i}(t))$, and, as the $q_i(t), q_{net_i}(t)$ are non-negative, $V(t) \geq 0$ and is equal to zero if and only if $q_i(t) = 0, q_{net_i}(t) = 0$. Further, as $\mu_i(q_i(t)) = 1$ for $q_i(t) > 0$, and $\mu_i(q_i(t)) = 0$ if only if $q_i(t) = 0$, it follows that $dV/dt = - \sum_{i=1}^n \mu_i(q_i(t))/t_{p_i} \leq 0$.

This then implies that

$$V(t) = V(0) - \int_0^t \sum_{i=1}^n \frac{\mu_i(q_i(t))}{t_{p_i}} dt \geq 0 \quad (3.16)$$

is monotonically decreasing. As $V(t)$ is bounded from below, we have $V(t) \downarrow V_f \geq 0$, or

$$\lim_{t \rightarrow \infty} \int_0^t \sum_{i=1}^n \frac{\mu_i}{t_{p_i}} dt = V(0) - V_f \geq 0. \quad (3.17)$$

The quantity $\mu_i(q_i(t))$ is either 1 or 0 depending on whether $q_i(t)$ is positive or zero, so $\mu_i(q_i(t))$ can be viewed as a set of pulses of unit height and varying widths. The integral $\int_0^\infty \mu_i(q_i(t)) dt$ is finite by (3.17), which implies that the widths of the unit-height pulses making up $\mu_i(q_i(t))$ must go to zero as $t \rightarrow \infty$. So, even if a $q_i(t)$ ($= x_i(t)/t_{p_i}$) continues to switch between zero and positive non-zero, the time intervals for which it is non-zero must go to zero as $t \rightarrow \infty$. Summarizing, the $q_i(t)$ are non-negative and continuous functions, bounded by the non-negative monotonically decreasing function $V(t)$, and the intervals for which the $q_i(t)$ are non-zero go to zero as $t \rightarrow \infty$. Further, as

$$\begin{aligned} u_i(t) &= -K_i \text{sat} \left(x_i(t) - \frac{\sum_{j=1}^n x_j(t - \tau_{ij})}{n} \right) \\ &= -K_i \text{sat} \left(t_{p_i} q_i(t) - \frac{\sum_{j=1}^n (t_{p_i}/t_{p_j}) q_j(t - \tau_{ij})}{n} \right), \end{aligned}$$

it follows that the time intervals for which the *bounded* functions $u_i(t)$ are non-zero must go to zero as $t \rightarrow \infty$. Consequently, by equation (3.14), $q_{net_i}(t) \rightarrow 0$ as $t \rightarrow \infty$.

We now show that the monotonically decreasing function $V(t)$ must go to zero; that is, $\lim_{t \rightarrow \infty} V(t) = V_f = 0$. Suppose not, so that $V_f > 0$. As $q_{net_i}(t) \rightarrow 0$, choose t_1 large enough so that $0 \leq \sum_{i=1}^n q_{net_i}(t) < \epsilon V_f$ for $t > t_1$ where $0 < \epsilon < 1$. Then

$$V(t) = \sum_{i=1}^n \left(q_i(t) + q_{net_i}(t) \right) \geq V_f \text{ for all } t$$

and

$$\sum_{i=1}^n q_i(t) \geq (1 - \epsilon) V_f > 0 \text{ for } t > t_1.$$

This in turn implies that at least one of the $q_i(t) > 0$ for all $t > t_1$, and therefore $\sum_{i=1}^n \mu_i(q_i(t))/t_{p_i} dt \geq \min\{1/t_{p_i}\}$ for all $t > t_1$. By equation (3.15), we then have

$$V(t) = V(t_1) - \int_{t_1}^t \sum_{i=1}^n \frac{\mu_i(q_i(t))}{t_{p_i}} dt \leq V(t_1) - \int_{t_1}^t \min\left\{\frac{1}{t_{p_i}}\right\} dt. \quad (3.18)$$

As the right side of (3.18) eventually becomes negative, we have a contradiction, and therefore $V_f = 0$. As it has already been shown that $q_{net_i} \rightarrow 0$ for all i , $V(t) \rightarrow 0$ then implies that the non-negative functions $q_i(t) \rightarrow 0$ for all i .

3.2.3.2 Controller Gains

The controller gain K_i is related to the actual implementation by how fast the load balancing can be carried out and how much (fraction) of the load is transferred. Recall that the waiting time is related to the number of tasks as $x_i(t) = q_i(t)t_{p_i}$ where t_{p_i} is the average time to carry out a task. Also, $y_i(t) = \left(q_i(t) - \left(\sum_{j=1}^n q_j(t - \tau_{ij}) \right) / n \right) t_{p_i} =$

$r_i(t)t_{p_i}$, where $r_i(t)$ is the number of tasks above the estimated (local) average number of tasks. When $y_i(t) > 0$, the control law (3.13) gives the rate of decrease of waiting time $x_i(t)$ by $u(t) = -K_i y_i(t)$. Consequently, the gain K_i represents the rate of reduction of waiting time per second in the continuous time model.

Define K_z as the fraction of the excess node to be transferred. With Δt the time interval between successive executions of the load balancing strategy, the control law says that a fraction of the queue $K_z r_i(t)$ ($0 < K_z < 1$) is removed in the time Δt so the rate of reduction of waiting time is $u(t) = -K_z r_i(t)t_{p_i}/\Delta t$, and

$$u(t) = -\frac{K_z y_i(t)}{\Delta t} \implies K_i = \frac{K_z}{\Delta t}. \quad (3.19)$$

Comparison of simulations on the model with the load balancing experiments are shown in Section 5.1. In the load balancing experiments on a parallel computer, Δt is different each time the load is balanced. As a consequence, the value of Δt used in (3.19) for a simulation is an average value for the experimental run.

From the previous subsection, the system described by the model (3.3) is asymptotically stable, and load balancing does not create or destroy tasks. For currently available tasks on all nodes, it takes some time for the system to enter a balanced state after load balancing actions are carried out. The settling time refers to the time it takes the system to settle, after which no further balancing action is needed for current queues. The settling time indicates how well the load balancing performs on a system of given queues of tasks, and is affected by a choice of gain K_z , the fraction of the excess load

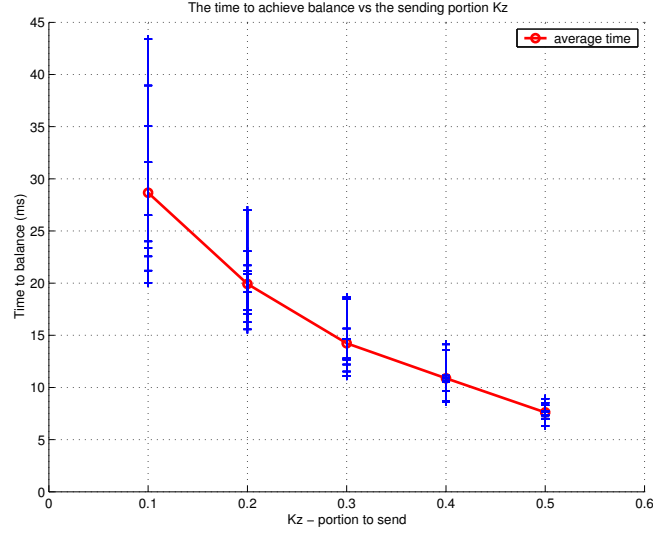


Figure 3.2: Summary of the load balance time as a function of the feedback gain.

to be transferred. Experiments were conducted to study the effect of feedback gains on the performance of load balancing, and the results were first reported in [30].

Figure 3.2 summarizes the data from several experimental runs on the parallel computer. For $K_z = 0.1, 0.2, 0.3, 0.4, 0.5$, ten runs were made, and the settling times were determined. These are marked as small horizontal ticks in Figure 3.2. For all such runs, the initial queues were the same, and set to $q_1(0) = 600, q_2(0) = 400, q_3(0) = 200$. For each value of K_z , the average settling time for these ten runs was computed and is marked as a dot in the figure. For values of $K_z = 0.6$ and higher (with increments of 0.1 in K_z), consistent results could not be obtained. In many cases, ringing extended throughout the experiment's complete time interval. That is, the response was so oscillatory that it was not possible to accurately determine a settling time (see experimental

results shown in Section 5.1). From Figure 3.2, it is desirable to choose the feedback gain to be close to 0.5 to achieve a faster response without breaking into oscillatory behavior. The experimental results were summarized in [51].

3.2.4 Brief Summary

A nonlinear time-delay system with proportional feedback control was developed to model the load balancing strategy. The model (3.3) was first presented in [29]. The model was shown to be consistent in that the queue lengths cannot go negative and that the total number of tasks in all the queues and the network is conserved. Further, it is shown that the proposed controller is asymptotically stable in the sense of Lyapunov [28]. Simulations of the nonlinear model are shown in Chapter 5 and are compared with an experimental implementation of the load balancing strategy on a parallel computer. Experiments found that the feedback gain affects the performance of load balancing [30]. It was found in experiments that the fraction of excess load to be transferred should be chosen close to 0.5 to achieve a faster response without breaking into oscillatory behavior [51].

3.3 Nonlinear Model with Resource Constraints

The load balancing operation requires processor time which is not being used to process tasks. Consequently, there is a trade-off between using processor time to process tasks and therefore reduce load, and using network bandwidth to achieve the advantage of

distributing the load evenly between the nodes to reduce overall processing time. The fact that the simulations of the model (3.3) in [28] showed the load balancing to be carried out faster than the corresponding experimental results motivated refinement of the model to account for processor constraints. A new deterministic nonlinear time-delay system model is presented here that captures these constraints. This model was proposed in [196] with a stability analysis as well as open-loop simulations and experiments.

Further, a control law is presented that uses not only its estimate of the queue size of the other nodes, but also measurements of the number of tasks in transit to it. It is shown that this controller tends to reduce unnecessary data transfers between nodes. The closed loop control of this model is presented in [195], and experiments demonstrate the superiority of the controller using anticipated work loads over the controller using only local work load estimates. Section 3.3.1 briefly discusses the model's consistency and stability. Section 3.3.2 describes the feedback control law on a local node and how a sending node decides to portion out its excess load to the other nodes. Finally, Section 3.3.3 concludes this work with a brief summary.

A mathematical model of a computing node for load balancing taking into account processor resource constraints is given by

$$\begin{aligned} \frac{dx_i(t)}{dt} = & \lambda_i - \mu_i (1 - \eta_i(t)) - U_m(x_i)\eta_i(t) \\ & + \sum_{j=1}^n p_{ij} \frac{t_{p_i}}{t_{p_j}} U_m(x_j(t - h_{ij}))\eta_j(t - h_{ij}), \end{aligned} \quad (3.20)$$

where $p_{ij} \geq 0$, $\sum_{i=1}^n p_{ij} = 1$, $p_{jj} = 0$, and

$$U_m(x_i) = \begin{cases} U_{m0} > 0 & \text{if } x_i > 0 \\ 0 & \text{if } x_i \leq 0. \end{cases}$$

In this model

- $x_i(t)$ is the expected waiting time experienced by a task inserted into the queue of the i^{th} node. λ_i is the rate of increase waiting time, and μ_i is the rate of reduction in waiting time at the i^{th} node.
- $\eta_i = 0$ or 1 is the *control input*, which specifies whether tasks (waiting time) are processed on a node ($\eta_i = 0$) or tasks are transferred to other nodes ($\eta_i = 1$).
- U_{m0} is the limit on the rate at which data can be transmitted from one node to another and is a constraint on network bandwidth.
- $p_{ij}U_m(x_j)\eta_j(t)$ is the rate at which node j sends waiting time (tasks) to node i at time t . That is, the transfer from node j of expected waiting time (e.g. $\int_{t_1}^{t_2} U_m(x_j)\eta_j(t)dt$ in the interval of time $[t_1, t_2]$) to the other nodes is carried out with the i^{th} node being sent the fraction p_{ij} of this waiting time ($p_{ij} \int_{t_1}^{t_2} U_m(x_j)\eta_j(t)dt$).
- The quantity $p_{ij}U_m(x_j(t - h_{ij}))\eta_j(t - h_{ij})$ is the rate of transfer of the expected waiting time (tasks) at time t from node j by (to) node i where h_{ij} ($h_{ii} = 0$) is the time delay for the task transfer from node j to node i .

In this model, the control input η_i is introduced to capture the processor constraints. Node i processes tasks in its queue when $\eta_i = 0$. As $\eta_i = 1$, node i can only send tasks to other nodes and cannot initiate transfers from another node to itself. The quantity p_{ij} is the probability that a certain waiting time to be transferred from node j to node i with $p_{ij} \geq 0$, $\sum_{i=1}^n p_{ij} = 1$ and $p_{jj} = 0$. A straightforward approach is to choose them as constant and equal; that is,

$$p_{ij} = 1/(n-1) \text{ for } j \neq i \text{ and } p_{ii} = 0. \quad (3.21)$$

Other approaches to set p_{ij} based on the available information on the state of the network are given in the Section 3.3.2. The analysis of consistency and stability of the model is given in the following subsection.

3.3.1 Model Consistency and Stability

It is now shown that the model (3.20) is consistent with actual working systems in that the queue lengths cannot go negative and the load balancing strategy cannot create or destroy tasks; it can only move tasks between nodes.

3.3.1.1 Non-Negativity of the Queue Lengths

To show the non-negativity of the queue lengths, recall that the queue length of each node is given by $q_i(t) = x_i(t)/t_{p_i}$. The model (3.20) is rewritten in terms of t_{p_i} as

$$\begin{aligned}
\frac{d}{dt} \left(x_i(t)/t_{p_i} \right) &= \frac{\lambda_i - \mu_i (1 - \eta_i(t))}{t_{p_i}} - \frac{1}{t_{p_i}} U_m(x_i) \eta_i(t) \\
&\quad + \sum_{j=1}^n \frac{p_{ij}}{t_{p_j}} U_m(x_j(t - h_{ij})) \eta_j(t - h_{ij}). \tag{3.22}
\end{aligned}$$

Given that $x_i(0) > 0$ for all i , it follows from the right-hand side of (3.22) that $q_i(t) = x_i(t)/t_{p_i} \geq 0$ for all $t \geq 0$ and all i . To see this, suppose without loss of generality that $q_i(t) = x_i(t)/t_{p_i}$ is the first queue to go to zero, and let t_1 be the time when $x_i(t_1) = 0$. At the time t_1 , $\lambda_i - \mu_i (1 - \eta_i(t)) = \lambda_i \geq 0$ as $\mu_i(x_i) = 0$ if $x_i = 0$. Also, $\sum_{j=1}^n \frac{p_{ij}}{t_{p_j}} U_m(x_j(t - h_{ij})) \eta_j(t - h_{ij}) \geq 0$, as $\eta_j \geq 0$. Further, the term $U_m(x_i) = 0$ for $x_i \leq 0$. Consequently

$$\frac{d}{dt} \left(x_i(t)/t_{p_i} \right) \geq 0 \text{ for } x_i = 0,$$

and thus the queues cannot go negative.

3.3.1.2 Conservation of Queue Lengths

It is now shown that the total number of tasks in all the queues and the network is conserved. To do so, sum up equations (3.22) from $i = 1, \dots, n$ to get

$$\begin{aligned}
\frac{d}{dt} \left(\sum_{i=1}^n q_i(t) \right) &= \sum_{i=1}^n \left(\frac{\lambda_i - \mu_i(x_i) (1 - \eta_i)}{t_{p_i}} \right) - \sum_{i=1}^n \frac{U_m(x_i(t))}{t_{p_i}} \eta_i \\
&\quad + \sum_{i=1}^n \sum_{j=1}^n \frac{p_{ij}}{t_{p_j}} U_m(x_j(t - h_{ij})) \eta_j(t - h_{ij}), \tag{3.23}
\end{aligned}$$

which is the rate of change of the total queue lengths on all the nodes. The dynamic model of the queue lengths in the network is given by

$$\frac{d}{dt}q_{net_i}(t) = -\sum_{j=1}^n \frac{p_{ij}}{t_{p_j}} U_m(x_j(t-h_{ij}))\eta_j(t-h_{ij}) + \sum_{j=1}^n \frac{p_{ij}}{t_{p_j}} U_m(x_j(t))\eta_j(t). \quad (3.24)$$

Here $q_{net_i} \triangleq \sum_{j \neq i}^n q_{net_{ij}}$ is the number of tasks put on the network that are being sent to node i from the other nodes. This equation says that the j^{th} node is inserting the number of tasks $q_{net_{ij}}$ into the network to be sent to node i at the rate $d(q_{net_{ij}})/dt = (p_{ij}/t_{p_j}) U_m(x_j(t))\eta_j(t)$, while the i^{th} node is taking these tasks from node j off the network at the rate $-(p_{ij}/t_{p_j}) U_m(x_j(t-h_{ij}))\eta_j(t-h_{ij})$. Summing (3.24) over all the nodes, one obtains

$$\begin{aligned} & \frac{d}{dt} \left(\sum_{i=1}^n q_{net_i}(t) \right) \\ &= -\sum_{i=1}^n \sum_{j=1}^n \frac{p_{ij}}{t_{p_j}} U_m(x_j(t-h_{ij}))\eta_j(t-h_{ij}) + \sum_{i=1}^n \sum_{j=1}^n \frac{p_{ij}}{t_{p_j}} U_m(x_j(t))\eta_j(t) \\ &= -\sum_{i=1}^n \sum_{j=1}^n \frac{p_{ij}}{t_{p_j}} U_m(x_j(t-h_{ij}))\eta_j(t-h_{ij}) + \sum_{j=1}^n \frac{U_m(x_j(t))\eta_j(t)}{t_{p_j}}. \end{aligned} \quad (3.25)$$

Adding (3.23) and (3.25), one obtains

$$\frac{d}{dt} \sum_{i=1}^n \left(q_i(t) + q_{net_i}(t) \right) = \sum_{i=1}^n \left(\frac{\lambda_i - \mu_i(1 - \eta_i)}{t_{p_i}} \right). \quad (3.26)$$

In words, the total number of tasks which are in the system (i.e., in the nodes or in the network) can increase only by the rate of arrival of tasks $\sum_{i=1}^n \lambda_i/t_{p_i}$ at all the

nodes, or similarly, decrease by the rate of processing of tasks $\sum_{i=1}^n \mu_i (1 - \eta_i) / t_{p_i}$ at all the nodes. The load balancing strategy itself cannot increase or decrease the total number of tasks in all the queues.

3.3.1.3 Stability of the Model

Combining the results of the previous two subsections, one can show Lyapunov stability of the model. For the system described by (3.20) and (3.24) with $\lambda_i = 0$ for $i = 1, \dots, n$ and initial conditions $x_i(0) \geq 0$, the system is found Lyapunov stable for any choice of the switching times of the control input functions $\eta_i(t)$.

First note that the q_{net_i} are non-negative as

$$q_{net_i}(t) = \sum_{j=1}^n \frac{p_{ij}}{t_{p_j}} \left(\int_{t-h_{ij}}^t U_m(x_j(\tau)) \eta_j(\tau) d\tau \right) \geq 0. \quad (3.27)$$

By the non-negativity property of the q_i , the linear function

$$V(q_i(t), q_{net_i}(t)) \triangleq \sum_{i=1}^n (q_i(t) + q_{net_i}(t))$$

is a positive definite function. Under the conditions of the theorem, equation (3.26)

becomes

$$\frac{d}{dt} \sum_{i=1}^n (q_i(t) + q_{net_i}(t)) = - \sum_{i=1}^n \frac{\mu_i(q_i)}{t_{p_i}} (1 - \eta_i), \quad (3.28)$$

which is negative semi-definite. By standard Lyapunov theory (e.g., see [132]), the system is Lyapunov stable.

The model (3.20) is only Lyapunov stable, and asymptotic stability must be ensured by judicious choice of the feedback. Compared to the stability equation (3.15) of the nonlinear model with saturations (3.3), the guarantee of asymptotic stability is lost. As the control input η_i introduced in equation (3.28) indicates, it is possible for the local controllers to spend all available time transferring tasks rather than processing tasks.

3.3.2 Feedback Control and Load Transfer

In this subsection, feedback control strategies are addressed to dynamically balance loads across a network of nodes. The controller at each node decides when a balancing action is needed and how much of its load is to be transferred out. At each sending node, how it portions out its excess load to be transferred to other nodes is given by a load transfer probability p_{ij} . Expected waiting times and other available information in the network can be used in the controller and the load transfer probability.

3.3.2.1 Controller based on the x_i

A feedback law based on local estimates of work loads at other nodes is proposed here. The controller (load balancer) at each node i uses the information of expected waiting time $x_i(t)$ and the *delayed* values $x_j(t - \tau_{ij})$ ($j \neq i$) from the other nodes. Here τ_{ij} ($\tau_{ii} = 0$) denote the time delays for communicating the expected waiting time x_j from node j to node i . Define

$$x_{i_avg} \triangleq \left(\sum_{j=1}^n x_j(t - \tau_{ij}) \right) / n \quad (3.29)$$

to be the *local average* at node i , which is the i^{th} node's *estimate* of the average of all the nodes. (This is only an estimate due to communication delays). Further, define

$$y_i(t) \triangleq x_i(t) - x_{i_avg}(t) = x_i(t) - \frac{\sum_{j=1}^n x_j(t - \tau_{ij})}{n}$$

to be the expected waiting time relative to the estimate of the network average by the i^{th} node.

A control law for model (3.20) based on x_i is given by

$$\eta_i(t) = h(y_i(t)) = h(x_i(t) - x_{i_avg}(t)), \quad (3.30)$$

where $h(\cdot)$ is a function given by

$$h(y) = \begin{cases} 1 & \text{if } y \geq y_0 \\ 0 & \text{if } y < y_0. \end{cases}$$

The control law basically states that if the i^{th} node's waiting time $x_i(t)$ is above the estimate of the network average $x_{i_avg}(t)$ ($(\sum_{j=1}^n x_j(t - \tau_{ij})/n)$) by the threshold amount y_0 , then the i^{th} node sends data to the other nodes; while if it is less than its estimate of the network average, nothing is sent.

The quantity p_{ij} defines how to portion the tasks to be sent out from each sending node. Instead of setting p_{ij} to be constant and equal as in (3.21), it is useful to use the local values of expected waiting times $x_i(t), i = 1, \dots, n$. Recall that p_{ij} is the fraction

of $\int_{t_1}^{t_2} U_m(x_j)\eta_j(t)dt$ in the interval of time $[t_1, t_2]$ that node j allocates (transfers) to node i , and conservation of the tasks requires $p_{ij} \geq 0, \sum_{i=1}^n p_{ij} = 1$. The quantity $x_i(t - \tau_{ji}) - x_{j_avg}$ represents what node j estimates the waiting time in the queue of node i to be relative to node j 's estimate of the network average. If the queue of node i is above the network average load as estimated by node j , then node j does not send any tasks to it.

Define a saturation function by

$$\text{sat}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0. \end{cases}$$

Then $\text{sat}(x_{j_avg} - x_i(t - \tau_{ji}))$ is node j 's estimate of how much node i is *below* the network average. Node j then repeats this computation for all the other nodes and portions out its tasks among the other nodes according to the amounts they are *below* its estimate of the network average; that is,

$$p_{ij} = \frac{\text{sat}(x_{j_avg} - x_i(t - \tau_{ji}))}{\sum_{i \ni i \neq j} \text{sat}(x_{j_avg} - x_i(t - \tau_{ji}))}. \quad (3.31)$$

If the denominator is zero, the expected waiting time on node j is not greater than its estimate of the network average, and therefore it is not sending out any tasks.

3.3.2.2 Controller based on the z_i

The controller (3.30) at each node is based on the *delayed* values of expected waiting times from the other nodes. Communication delays τ_{ij} are much smaller than the delays h_{ij} of the actual tasks transfer. Unnecessary task transfers could be initiated on highly-loaded nodes due to delayed queue sizes from other nodes (as illustrated in Section 5.1.2). However, there is additional information that can be made available to the nodes — specifically, the information about the tasks that are in the network being sent to the i^{th} node, q_{net_i} , or equivalently, the waiting time $x_{net_i} \triangleq t_{p_i} q_{net_i}$, can be sent to the receiving nodes.

Here, the proposed controller at node i is based upon not only the local estimate of the work loads q_i from other nodes, but also the number of tasks q_{net_i} in transit to node i [195]. Node j broadcasts to each node k in the network information on the number of tasks $q_{net_{kj}}$ it is sending to each of the other nodes. This way the other nodes can take this information into account (without having to wait for the actual arrival of the tasks) in making their control decisions. The communication of the number of tasks $q_{net_{kj}}$ being sent from node j to node k is much faster than the actual transfer of the tasks. Furthermore, each node i broadcasts its total (anticipated) amount of tasks, i.e., $q_i + q_{net_i}$ where $q_{net_i} = \sum_{j=1}^n q_{net_{ij}}$, to the other nodes so that they have a more current estimate of the tasks on each node (rather than having to wait for the actual transfer of the tasks). The information that each node has will be a more up to date estimate of the state of network using this scheme.

Define

$$z_i \triangleq x_i + x_{net_i} = t_{p_i} (q_i + q_{net_i}) \quad (3.32)$$

to be the *anticipated* waiting time at node i . Further, define

$$z_{i_avg} \triangleq \left(\sum_{j=1}^n z_j(t - \tau_{ij}) \right) / n \quad (3.33)$$

to be the i^{th} node's estimate of the average anticipated waiting time of all the nodes in the network. This is still an estimate due to the communication delays. Therefore,

$$w_i(t) \triangleq x_i(t) - z_{i_avg}(t) = x_i(t) - \frac{\sum_{j=1}^n z_j(t - \tau_{ij})}{n} \quad (3.34)$$

is the anticipated waiting time relative to the estimate of average (anticipated) waiting time in the network by the i^{th} node. By using expected waiting time $x_i(t)$ in (3.34), we avoid migration of nonexistent tasks (waiting time) from the queue on node i (i.e., tasks in transit to it). A control law based on the anticipated waiting time is chosen as

$$\eta_i(t) = h(w_i(t)). \quad (3.35)$$

Similarly, the p_{ij} can be specified using the anticipated waiting time z_j of the other nodes. The quantity $z_{j_avg} - z_i(t - \tau_{ji})$ represents what node j estimates the network's average anticipated waiting time is relative to its estimate of the anticipated waiting time in the queue of node i . If the estimate of the queue of node i ($z_i(t - \tau_{ji})$) is above

what node j estimates the network's average (z_{j_avg}) is, then node j sends tasks to node i . Otherwise, node j sends no tasks to node i . Therefore $\text{sat}(z_{j_avg} - z_i(t - \tau_{ji}))$ is a measure by node j as to how much node i is *below* the local average. Node j then repeats this computation for all the other nodes and portions out its tasks among the other nodes according to the amounts they are below its estimate of the network average; that is,

$$p_{ij} = \frac{\text{sat}(z_{j_avg} - z_i(t - \tau_{ji}))}{\sum_{i \ni i \neq j} \text{sat}(z_{j_avg} - z_i(t - \tau_{ji}))}. \quad (3.36)$$

It is obvious that $p_{ij} \geq 0$, $\sum_{i=1}^n p_{ij} = 1$, and $p_{jj} = 0$. All p_{ij} are defined to be zero, and no load is transferred if the denominator is zero, or concisely stated,

$$\sum_{i \ni i \neq j} \text{sat}(z_{j_avg} - z_i(t - \tau_{ji})) = 0.$$

Experiments on a network of three nodes are presented here to compare the two controllers discussed above. The following experiments show the responses using closed loop load balancing with an initial queue distribution of $q_1(0) = 600$ tasks, $q_2(0) = 200$ tasks and $q_3(0) = 100$ tasks. The average time to do a search task is $400 \mu s$. In these experiments the inputs were set as $\lambda_1 = 0$, $\lambda_2 = 0$, $\lambda_3 = 0$. After initial communications, the closed loop load balancing strategy was initiated using, in the first case, the controller (3.30) and p_{ij} based on local waiting times x_i , and, in the second case, the controller (3.35) and p_{ij} based on anticipated waiting times z_i .

Figures 3.3 and 3.4 show the responses of the queues versus time using the controller

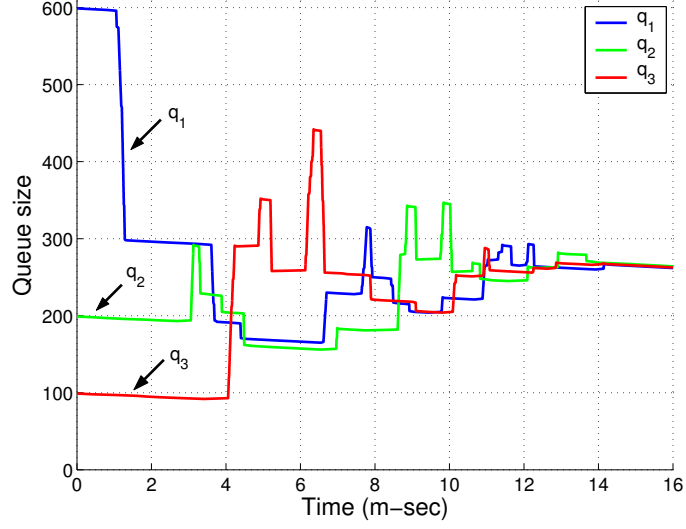


Figure 3.3: Plot of queue sizes using the controller (3.30) and p_{ij} specified by (3.31).

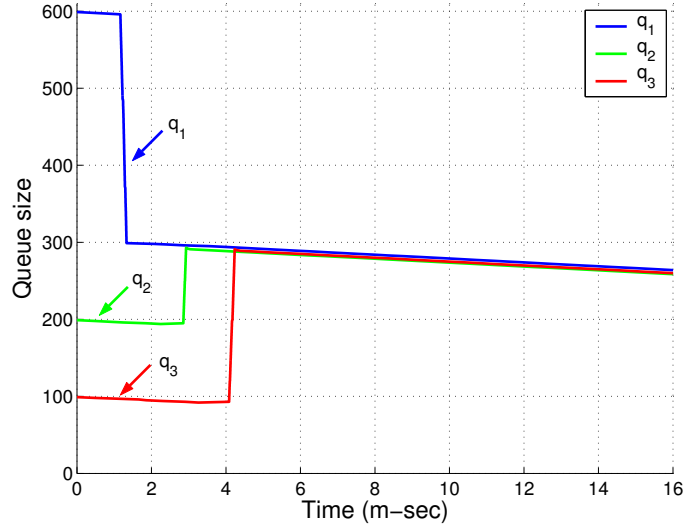


Figure 3.4: Plot of queue sizes using the controller (3.35) and p_{ij} specified by (3.36).

(3.30) with p_{ij} specified by (3.31), and the controller (3.35) with p_{ij} specified by (3.36), respectively. Note the substantial difference in the load balancing performance between these two schemes. In Figure 3.3, there are unnecessary exchanges of tasks back and forth between nodes. Although the system reaches a balanced condition at around $t = 14 \text{ ms}$, those additional transfers cost processing time and prolong the completion time. In Figure 3.4, the system reaches the balanced state much faster by using the anticipated waiting times. More experimental results on the two controllers are presented in Section 5.1.2.

3.3.3 Brief Summary

A model for load balancing strategy that incorporates processor resource constraints was developed as a nonlinear time-delay system. This model was presented with a stability analysis and open loop experiments in [196]. It was shown that the model was consistent in that the total number of tasks was conserved and that the queues were always non-negative. It was also shown the system was always Lyapunov stable, but not necessarily asymptotically stable. The guarantee of asymptotic stability is lost, compared to the nonlinear model with proportional feedback control in Section 3.2, because it is possible for the local controllers to spend all available time transferring tasks rather than processing tasks. A closed loop controller was proposed based on the local queue size and the estimate of the tasks being sent to the queue from other nodes [195]. Experimental results on a parallel machine demonstrated the efficacy of the load balancing strategy and are presented in Section 5.1.2. Substantial improvements using

the controller based on the anticipated work loads can be achieved over the controller based on the local work loads only.

3.4 Summary

This chapter discussed continuous time models developed in this research to model load balancing strategies in a parallel computing network. The load balancing strategy was first simplified to a linear time-invariant system with delays, and the stability of this linear model was analyzed by Chiasson *et al.* [1][2]. To account for the two problems left out in the linear model — instabilities above certain thresholds in controller gains and negative queue sizes, a nonlinear time-delay system with saturations was developed to model the load balancing strategy [29]. Three essential properties of the load balancing strategy, including consistency, stability, and conservation of tasks, were analyzed [28]. Experiments found that the feedback gain in the controller affects the performance of load balancing [30]. A judicious choice of feedback gain is necessary to achieve a faster balancing response without breaking into oscillatory behavior [51]. A third approach to modeling load balancing strategies was proposed that incorporates both time delays and processor resource constraints [196]. This model accounts for the trade-off in each processor between processing tasks and transferring load to reduce the overall processing time [196]. The control law was implemented as a distributed closed-loop controller to balance the load at each node using not only local estimates of loads at other nodes, but also estimates of the number of tasks in transit to each node [195].

Chapter 4

Parallel DNA Database

Modern DNA technology has resulted in the extraordinarily accurate identification of criminals and the exoneration of the innocent at a remarkable rate [38]. Since its introduction in the mid-1980s, DNA typing has significantly changed forensic science and the ability of law enforcement to identify perpetrators from minute crime scene evidence [183]. The Combined DNA Index System (CODIS), established by the Federal Bureau of Investigation (FBI) Laboratory, blends forensic science and computer technology into an effective tool for solving violent crimes [81]. CODIS is implemented as a distributed database with three hierarchical tiers – local, state, and national. The National DNA Index System (NDIS) is the top level in the CODIS hierarchy, and enables participating DNA laboratories to exchange and compare DNA profiles on a national level.

DNA databases have experienced rapid growth in recent years. NDIS became operational in October 1998, and it has received and stored over 2 million (convicted offenders)

profiles as of December 2004 [81]. It is predicted that the size of DNA databases will increase at a faster rate to an eventual scale of 10^8 profiles. Forensic applications require rapid searches on these DNA database. Currently there are about 180 DNA crime labs in the U.S., and they all have almost one year's backlog of work [76]. The rapid growth and increasing demand for fast searches of DNA databases necessitate the development of parallel DNA databases. New methods developed in [32][211][33] not only provide orders of magnitude improvements in performance over current CODIS software (5 *ms* vs 5 *sec.*), but also extend to a parallel decomposition of the DNA database search problem.

This chapter presents the design of a parallel DNA database. DNA profiles are stored in distributed databases, with a portion of profiles in each, on the nodes of a parallel machine. Database operations, such as search and store requests, are distributed by a search server, and are executed in parallel by the search engines running on each node. Although the focus of this work is on large-scale DNA databases, the ideas on the design of the parallel database also apply to other databases of multidimensional data with naturally occurring patterns within the stored data.

This chapter is outlined as follows: In Section 4.1, the previous work by the Laboratory for Information Technologies on the tree-structured indexing method for DNA information storage and retrieval is briefly described. This method provides a solid foundation for a highly efficient search engine in the parallel DNA database. Section 4.2 addresses the designs of search servers for the parallel DNA database. A search

server using hierarchical sockets is first presented in Section 4.2.1, and Section 4.2.2 discusses the design of a multi-threaded search server using PVM and object serialization. Section 4.3 documents a parallel database with integrated load balancing. Section 4.4 concludes this chapter with a brief summary.

4.1 DNA Search Engine

Deoxyribonucleic acid (DNA) consists of nucleic acids made of complex molecules that carry (in coded form) instructions for passing on hereditary characteristics [79]. A DNA profile consists of allele information at one or more DNA loci (or sites). Typically 10 or more loci are used for DNA identification [183]. An individual may have either one or two alleles at each locus (with rare cases of three alleles); forensic samples containing DNA (mixture) from two or more individuals can have more alleles. DNA identification requires locating all matched DNA profiles within the database that satisfy a set of user-selected match criteria when compared against a target DNA profile. The match criteria allow for the possibility of missing allele or locus data, the inexact match of allele information at a specified locus, an error tolerance in the number of base pairs in matching alleles, and the specification of equivalent alleles. The match criteria can also define groups of loci that must be present in matching DNA profiles and a maximum number of matching profiles to be returned to the requesting user [33].

The complexity of the DNA match specifications can not be implemented using standard database indexing methods, such as those found in commercial database en-

gines. Changes in the way that DNA information is stored and retrieved are required to implement search strategies that scale better than linearly with the size of the database. Although standard database indexing structures, such as B-trees, can help to rapidly access records based upon the indexed value of a selected field, they are not able to take advantage of naturally occurring structure in the data. More than one field may be indexed, but multiple index structures that are computed independently are often required. When stored information based on several indices is retrieved, a time-consuming intersection operation is required to obtain results.

Previous work by Birdwell *et al.* in [32][211][33] developed a DNA profile storage organization mechanism that minimizes the time required to find DNA matches by using a tree-structured indexing method. First, a decision tree is formed to hold all DNA profiles. A test is made upon DNA profile information at each node of the tree, so that one or more branches are selected from the node based on the test results. In this manner, the search is narrowed down to a smaller scope, by ruling out many other portions of the database. This tree-structured method results in a complexity on the order of $\log(N)$, $O(\log(N))$, where N is the total number of DNA profiles in the database. The performance is achieved by utilization of naturally occurring patterns and clusters within stored DNA data. Both entropy-adjacency partitioning and multivariate statistical clustering can be used to divide the profiles into subsets of roughly equal size at each level of the tree, and therefore minimize the worst-case search time [33]. This rapid search method provides a solid foundation for a high-performance search engine

for the parallel DNA database.

To evaluate the parallel DNA database, a parallel computer has been built for use as an experimental facility. The design of the parallel machine is illustrated in Figure 2.3 of Section 2.2. A photo of the parallel machine is shown in Figure 2.4, and more are available at [27]. The hosts are connected via a Gigabit Ethernet network, and a Fast Ethernet network that is used for software installations and a backup connection. It is anticipated that the implementation will scale by multiples of eight computers, and the upper limit of this design appears to be on the order of 10^8 DNA profiles due to current memory limitations of the systems and available network bandwidth.

Figure 4.1 shows a flowchart of the parallel search engine running on each host of the parallel computer. The search engine program includes three modules: **BuildDNATree**, **MainServerThread**, and **RequestClientThread**. Module **BuildDNATree** builds the tree structure for DNA profiles loaded from a local MySQL database or an input file. MySQL is a popular open source database and provides a less complicated solution suitable for widespread application deployment at great reliability and reduced costs compared to other commercial database engines [160]. A MySQL database is preferred over a file for persistent storage and convenient maintenance. Module **MainServerThread** spawns a thread which resides in memory and listens for incoming requests distributed by the parallel search server. Depending on the design and implementation of the parallel search server (in the next section), **MainServerThread** communicates with the search server via sockets or PVM. Module **RequestClientThread** spawns a thread that pro-

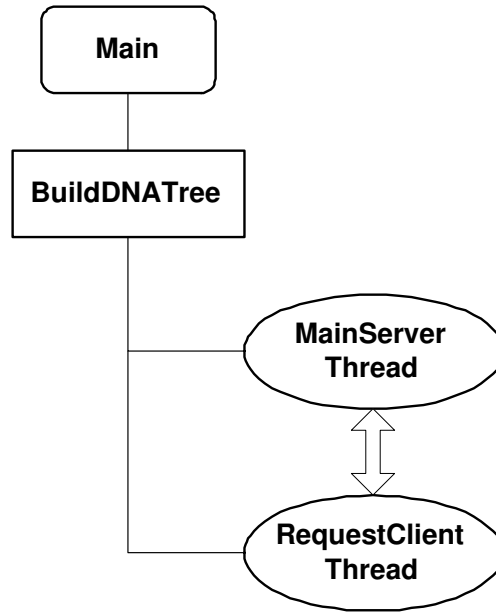


Figure 4.1: Flowchart of the parallel search engine.

cesses requests whenever a local queue of requests is not empty. Requests are inserted to the local queue by `MainServerThread`, and the results are sent back to the search server via `MainServerThread`.

4.2 Parallel Search Server

The search engine software serves as a back-end search agent that runs on each node of the parallel computer. A search server communicates with clients, accepts incoming requests, and returns results. Clients do not interact directly with parallel nodes, and see a single huge database with rapid search capability. The distribution of searches across the nodes of the parallel machine and the collection of results are transparent to

the client. This section documents the design of the search server for the parallel DNA database. Section 4.2.1 discusses a search server that uses hierarchical sockets and the legacy code of a single host DNA database. This search server extends a single host DNA database previously developed in the Laboratory for Information Technologies to a parallel DNA database on multiple nodes. The server provides fast searches, but does not perform well on transaction times, which are the processing times seen by clients. In Section 4.2.2, the design of a multi-threaded search server that reduces the loss of performance in transaction times by using threads, PVM, and object serialization is documented. Experiments using synthetic data of up to 48M (48,000,000) DNA profiles are performed to evaluate the performance of the parallel DNA database.

4.2.1 Hierarchical Sockets Search Server

A socket represents one end of a computer communication link and holds or points to all the information associated with the link. This information includes the communication protocol to use, source and destination addresses, data buffers and option flags [178]. Sockets are either connection-oriented or connectionless. Connection-oriented sockets allow for data to flow back and forth as needed, while connectionless sockets (also known as datagram sockets) allow only one message at a time to be transmitted, without an open connection. Connection-oriented sockets are used in this research for stream communications. The Simple Sockets Library [40] provides an application programming interface to streaming sockets communications that resembles the standard C library's file handling interface. Previous work in LIT by J. D. Birdwell *et al.* enables a client

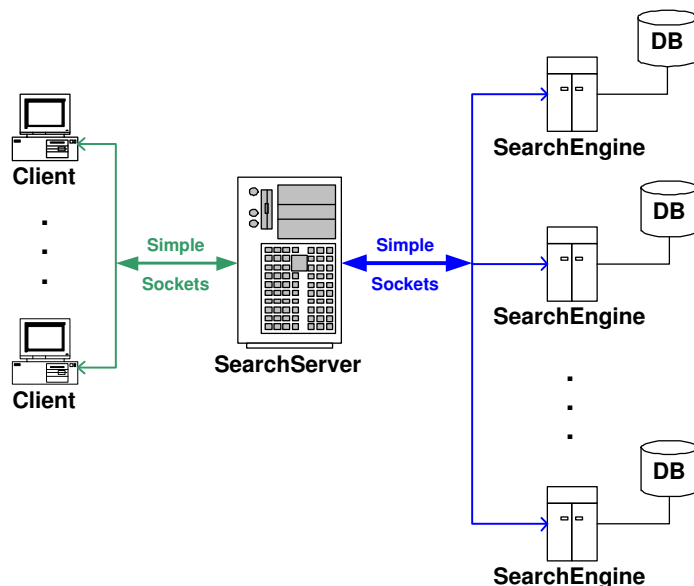


Figure 4.2: Schematic diagram of a parallel search server using hierarchical sockets.

program (either on Windows or Linux) to communicate with a single host DNA database via simple sockets. An extension to the legacy code is made to support the parallel DNA database on multiple hosts. The simple sockets library is used for socket communications between the client and the search server, and for communications between the search server and the back-end search engines. A block diagram of the parallel search server using hierarchical sockets is shown in Figure 4.2.

The search server listens for incoming requests from a client, parses the requests, and distributes them to all search engines in the parallel computer for service. Figure 4.3 shows a flowchart of the parallel search server using hierarchical sockets. The search server includes three modules: `ReadConfig`, `MultiServer`, and `RequestHandlers`. Module `ReadConfig` reads and parses a configuration file that specifies the number of search engines to be run and the nodes of the parallel computer to be used. Module

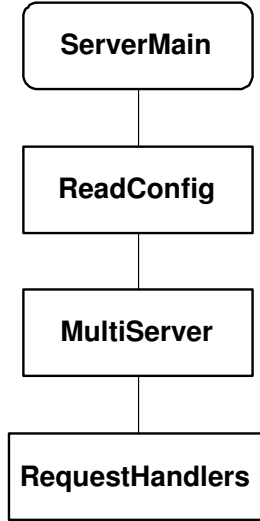


Figure 4.3: Flowchart of the hierarchical sockets search server.

MultiServer listens for incoming requests from clients through stream sockets. When requests arrive, module **RequestHandlers** parses requests and distributes them to one or all search engines through stream sockets. There are four kinds of requests: **search**, **find**, **delete**, and **store**. For **store** requests, module **RequestHandlers** randomly picks a node in the parallel computer to store the DNA profile, and therefore balances the storage of DNA profiles across the parallel computer. Module **RequestHandlers** sends the other three kinds of requests to all search engines. Then the module **RequestHandlers** waits for results from the search engines, gathers results and sends back to the client.

An experiment using 2,000 random searches is run to evaluate the performance of the parallel DNA database on the cluster of 24 nodes. Each parallel node has a 2M-profile portion of a 48M-profile DNA database. A client program randomly selects 2,000 profiles from the parallel database (of 48M profiles), and then sends the 2,000 search

requests to the search server one at a time, which distributes each request to all 24 search engines for service. The timing results for 2,000 random searches are shown in Figure 4.4. In this figure, the top part shows the average search time versus the node index (`node01` through `node24`). The search time refers to the time to service a search request by searching the decision tree to find an exact match. The average search time across the parallel machine is $431.4 \mu s$ ¹. This search time is at least three orders of magnitude faster than that of the current CODIS software. The bottom part of Figure 4.4 shows the average transaction time of 2,000 random searches versus the node index. The transaction time is the round-trip time required to send a search request from a client, process the request, and return the results to the client. The average transaction time across the parallel machine is $410.3 ms$.

From Figure 4.4, the search server using hierarchical sockets shows good search time performance, but the transaction times are poor. Improvements to both the search engine and the search server are addressed in the next section.

4.2.2 Multi-threaded Search Server

To improve the transaction times of the parallel DNA database, the design of the search engine is optimized on each parallel node. Originally it makes additional function calls to find the profiles as results to return to clients after the search process finds the indices of all matched profiles. A straightforward improvement is made by integrating storage of

¹The timing data is measured on the 24 computing nodes of the parallel machine, which consisting of 8 single processor nodes and 16 dual processor nodes with specifications given in Appendix A.

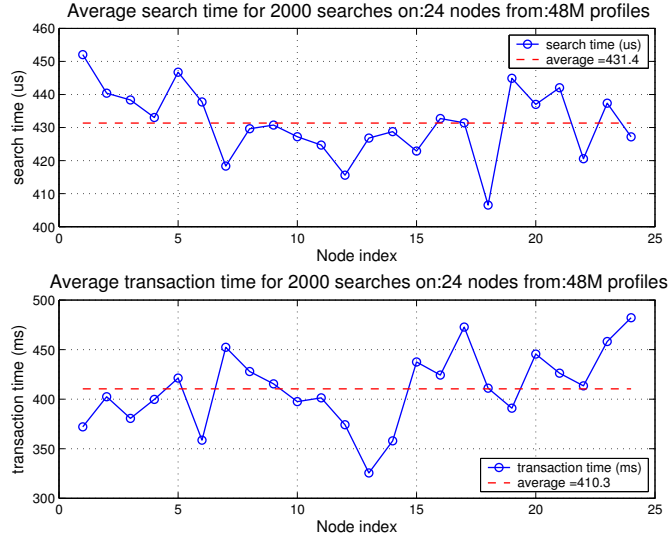


Figure 4.4: Time measured for 2,000 random searches on 24 nodes of 48M profiles. Top: average search times, bottom: average transaction times.

the matched profiles to the search process (as profiles are used when calculating matches in leaf nodes of the DNA tree structure). The storage of the matched profiles is used as the results to return to clients without calling additional functions to find the profiles based on the indices of the matched profiles. Another way is to design a multi-threaded search server to eliminate the overhead of the simple sockets communication library. The simple sockets client communicates with the socket server in clear text twice for each packet: the first transaction provides the packet size (to avoid a buffer overflow), and the second transaction provides the data. This can be replaced by communication with binary buffers using PVM and object serialization when the architectures of all nodes of the parallel machine use the same data format. This is the case in the experimental

facility, where all nodes conform to the Intel i386 architecture.

4.2.2.1 Key Terms

The multi-threaded search server presented in this section relies on the advantages of threads, PVM, object serialization, and queues of requests and results. These key terms to be used later are briefly described as follows.

A *thread* is a lightweight process. A *process* is an instance of a program in execution [179]. Each process has a full address space and operating system's state. While multiple threads share the same address space within a single process, the context switch cost between threads is much cheaper than that between processes [175]. A *multi-threaded* program runs in parallel and therefore makes a more efficient use of resources on multiprocessor computers than its single thread (process) counterpart [145].

A *pool of threads* is a collection of threads created to service incoming tasks. Any available thread (i.e., idling) may be selected to service an incoming task. A typical selection method is *round robin*, in which new tasks are assigned in a cyclic fashion among the pool of threads. That is, the threads are labeled $0, 1, \dots, m-1$ and the K th incoming task is assigned to the thread labeled $K \bmod m$.

PVM (Parallel Virtual Machine) is a portable programming system that provides message passing and process management functions. PVM is designed to link separate host machines to form a “virtual machine”, which is a single, manageable computing resource [140]. The system is portable to a wide variety of architectures, including workstations, multiprocessors, supercomputers and PCs [90].

Object serialization is a representation mechanism for storing and transmitting objects, in which objects are written in a serialized form to a sequence of bytes (bits) stored in memory or a file. The bytes can be reconstructed as they are read when the content is needed later or after the objects are sent across a network.

Requests refer to incoming tasks from clients. There are four types of requests in the DNA database application: **SearchRequest**, **FindRequest**, **StoreRequest**, and **DeleteRequest**, which corresponds to search (using specified match criteria), list (to retrieve a profile by an index number), store, and delete DNA profiles, respectively. A *queue of requests* is a collection of tasks to be executed by the DNA search engines.

Results stand for the resulting data obtained on DNA search engines for the incoming requests and to be returned to clients. There are four types of results in the DNA database application: **SearchResult**, **FindResult**, **StoreResult**, and **DeleteResult**; each is the result for the corresponding request. A *queue of results* is a collection of results to be returned to clients by the search server.

4.2.2.2 Multi-threaded Search Server

Figure 4.5 shows the structure of a multi-threaded search server using threads, PVM², and object serialization. The multi-threaded search server *utss* starts a thread (**LThread**) that listens on the network for connection requests and manages a pool of service threads (**SThreads**) that serve these connections. Requests are distributed and results are gath-

²PVM is used, rather than MPI [11], because PVM is more robust to node failure. Whereas PVM can tolerate the failure of any node other its root node, the failure of any node of a MPI cluster will cause MPI to fail.

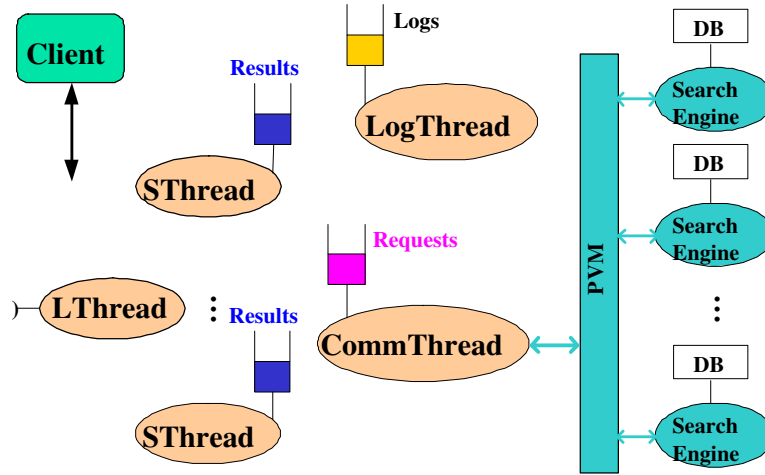


Figure 4.5: Schematic diagram of a multi-threaded search server.

ered using a communication thread (**CommThread**), which communicates with the search engines via PVM and serialization. A logging thread (**LogThread**) records events, such as connection time and request information, into a MySQL database for administration. Each search engine node also maintains a MySQL database for saving and restoring states.

When a client program makes a socket connection to send in a request, a sequence of events occur in the search server as follows:

- **LThread** listens for connection requests from clients.
- **LThread** assigns the requested connection to an available **SThread**, which is selected *round robin* from a pool of m **SThreads**. If no **SThread** is available, **LThread** waits until one **SThread** completes its last request and becomes available.
- The assigned **SThread** parses the request data from the client and puts the request

object to `CommThread`.

- `CommThread` distributes the request to the DNA search engines running the nodes of the parallel machine for service.

Components for the multi-threaded search server, `LThread`, `SThread`, and `CommThread`, are briefly described below.

`LThread` opens a socket on a port specified by the symbol `ParallelDaemonPort` as the server side. The socket server listens for requests from clients, and `LThread` manages a pool of m running `SThreads`. The number of `SThreads` (m) ranges from 1 to the maximum available number allowed by the operating system (the default limit number is 1023 on Linux). When a connection is requested, `LThread` establishes the connection and selects an available `SThread`, in round robin, to serve the connection. Once an `SThread` is assigned to service the client, `LThread` comes back to listen for the next client request and assigns another available `SThread` to service.

`SThread` connects to a requesting client via a socket and the socket stays open until it receives a packet from the client that indicates an end of requests. While the socket is open, `SThread` repeats the following sequence:

- Get request data from the client.
- Separate the request data into two parts: `command` and `target`. The `command` may be one of four types: `search`, `find`, `store`, and `delete`, while the `target` refers to the DNA profile to search against.

- Parse the `command` and the `target` DNA profile.
- Create a request object based on the parsed `command` and `target` profile.
- Put the request object in the `CommThread`'s request queue.
- Wait for results from `CommThread`, and return these to the client.
- Accept the next request until an end packet is received.

`CommThread` distributes requests to and gathers results from all search engines on the parallel computer. Figure 4.6 shows (in red) the request processing steps performed by the `CommThread`. The `SThread` assigned by the `LThread` to a client connection accepts the request data from the client, and pushes the request object onto the request queue of `CommThread` after parsing. A request object is transmitted across the network using binary serialization and deserialization. The serialization sequentially packs sizes and data for all members of a request object into one binary package, while the deserialization unpacks the package (after receiving) in the order reverse to the packing sequence to ensure correct receiving of the sizes and the data of all members of the object. `CommThread` takes a request object from its queue of requests, and distributes it to one or more PVM nodes depending on the request type of the object. There are four kinds of requests: `SearchRequest`, `FindRequest`, `DeleteRequest`, and `StoreRequest`. While `StoreRequests` are sent to a randomly chosen node to balance DNA data load on the parallel computer, the other three types of requests are sent to all search engines for service.

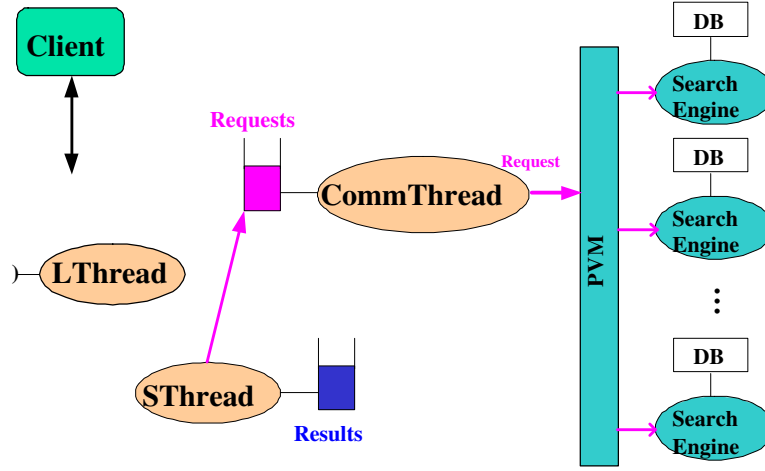


Figure 4.6: Multi-threaded search server: requests processing.

Each search engine on the parallel computer enrolls into PVM after building its DNA decision tree. Each search engine spawns a main server thread to listen for incoming requests, and a request client thread to service the received requests. After searching through the decision tree for matches (or other actions), the search engine sends the result in a serialized buffer using PVM to the `CommThread` of the search server. Figure 4.7 shows (in blue) how results are returned to the client. `CommThread` waits for the serialized results from all search engines. There are four kinds of results: `SearchResult`, `FindResult`, `DeleteResult`, and `StoreResult`. `CommThread` gathers results after receiving the serialized data from all search engines (only one for `StoreResult`), and pushes the results objects to the requesting `SThread`'s results queue according to the index stored at the `LThread`. The `SThread` returns the results to the client via the dedicated socket connection.

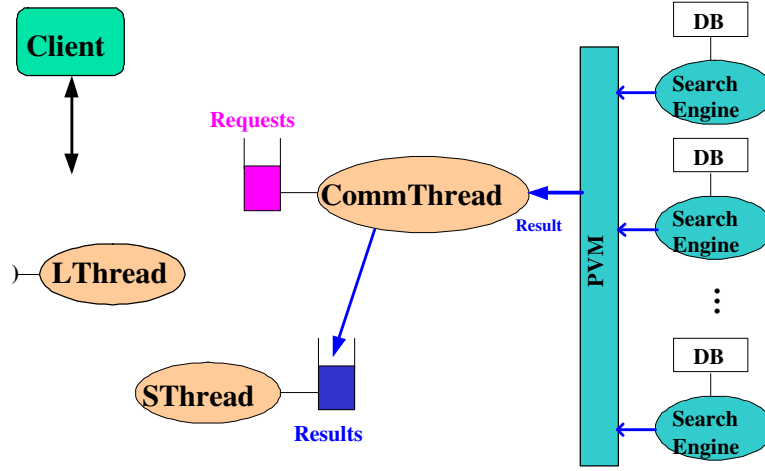


Figure 4.7: Multi-thread search server: results returning.

4.2.2.3 Performance Results

An experiment using 2,000 random searches is conducted to test the performance of the multi-threaded search server. The DNA database has 48M profiles stored across 24 nodes of the parallel computer, with a 2M-profile portion on each. Figure 4.8 shows the measured search times and transaction times for 2,000 random searches. In this figure, the top part shows the average search time versus the node index (**node01** through **node24**). The average search time across the parallel machine is $318.3 \mu s$, which is slightly better than the average search time of $431.4 \mu s$ in Figure 4.4. The bottom part shows the average transaction time for 2,000 random searches as a function of the node index. The average transaction time across the parallel machine is $3.7 ms$. Compared to the average transaction time of $410.3 ms$ in Figure 4.4, this is about 110-fold smaller.

Table 4.1 compares the average search time and the average transaction time on two search server implementations. The performance boost in transaction times by the

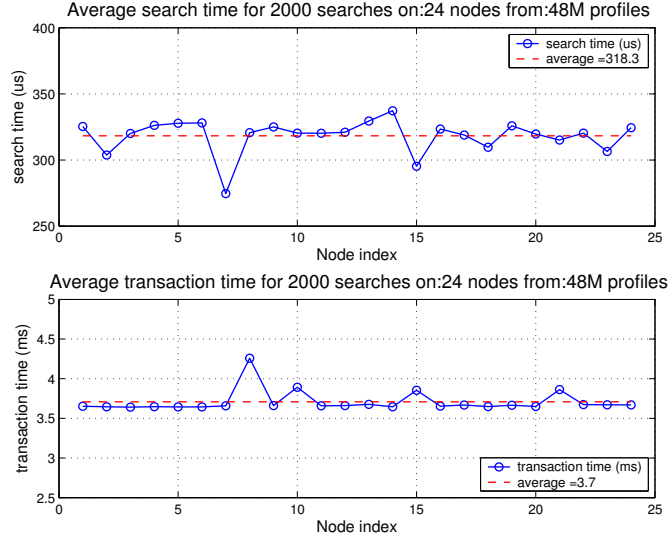


Figure 4.8: Time measured for 2,000 random searches on the multi-threaded search server. Top: average search time, bottom: average transaction time.

Table 4.1: Performance comparison of two search servers

	Search time	Transaction time
Hierarchical sockets search server	431.4 μs	410.3 ms
Multi-threaded search server	318.3 μs	3.7 ms

multi-threaded search server is primarily due to the replacement of the simple sockets library by a multi-threaded design using PVM and binary communication via object serialization. More experimental results are shown in Section 5.2 that demonstrate the high performance of the multi-threaded search server.

4.3 Parallel Search with Load Balancing

The parallel database described in the previous section provides a practical application for the load balancing strategy discussed in Chapter 3. This section presents the design of parallel searches with integrated load balancing. The parallel database is implemented as a set of queues with associated search engine threads, typically assigned one per node of the parallel machine, as presented in the previous section. Each search engine accesses a tree-structured index to locate database records that match search requests.

Figure 4.9 shows a schematic structure of the parallel database with load balancing. Nodes on the parallel computer are divided into many groups to form a distributed storage structure for DNA profiles. Each group is assigned a portion of the DNA profiles, and all nodes of a group share the same data. Load balancing actions are performed between search engines running on the nodes in the same group. Due to the structure of the search process, search requests can be formulated for any target profile and associated with any node of the index tree. These search requests are created not only by the database clients; the search process itself can also create search requests as the index tree is descended by any search thread, according to the search mechanism

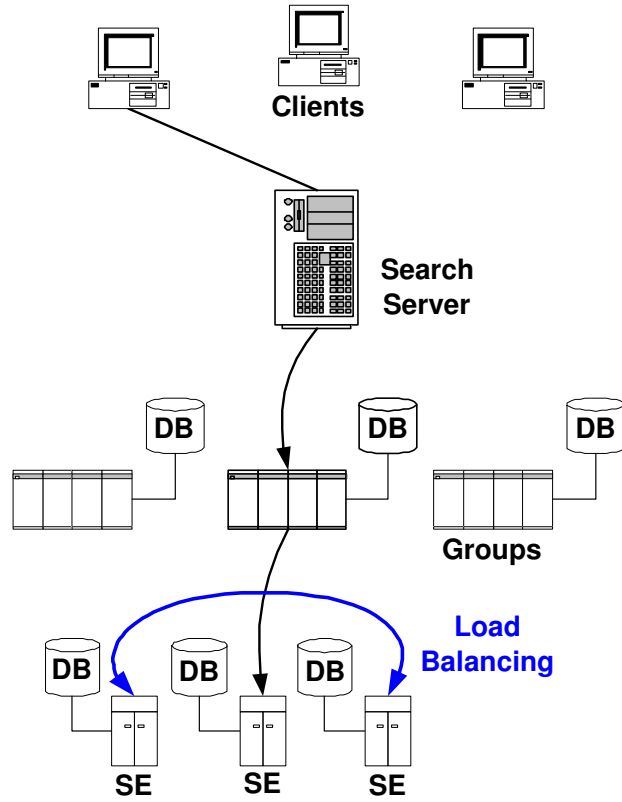


Figure 4.9: A depiction of the parallel DNA database with integrated load balancing. All search engines on the nodes of each group use an identical DNA database. Load balancing is done between the nodes of a group to even out the queues of requests.

described in [33]. Search requests that await processing may be placed in any queue associated with a search engine having the same stored data, and the contents of these queues may be moved arbitrarily among the processing nodes of a group to achieve a balanced load.

The parallel DNA database with load balancing uses the multi-threaded search server’s structure discussed in Section 4.2.2. While the structure of the new search engine software is similar to that shown in Figure 4.1, changes to the software, primar-

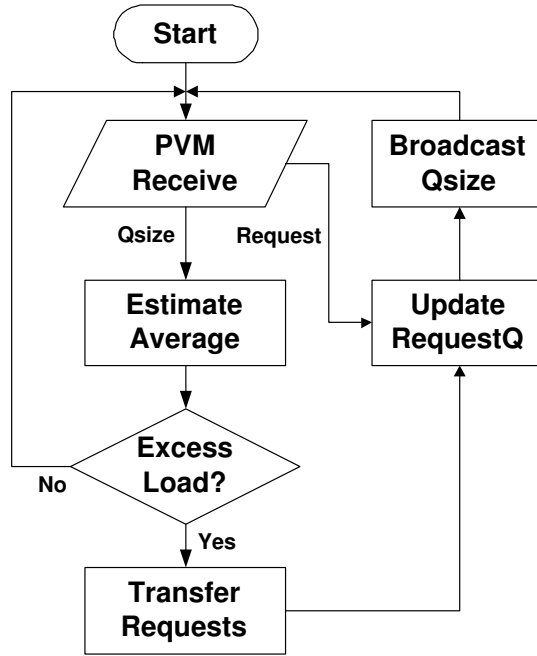


Figure 4.10: Flowchart of `MainServerThread` for parallel search with integrated load balancing.

ily in the `MainServerThread` module, have been made to integrate the load balancing strategy with the search engine. Figure 4.10 shows a flowchart of `MainServerThread` for the search engine with integrated load balancing. Note that the current PVM software [140] is not multithread safe, which implies all PVM communications must be handled in a single thread. This could cause a bottleneck when intensive communications are needed in a cluster consisting of a large number of nodes.

The `MainServerThread` module repeats the following sequence:

- Receive PVM data in a non-blocking³ manner.

³Non-blocking means the routine always returns immediately either with the message or with the information that the message has not yet arrived at the local `pvmd` [90].

- If the received data are requests, either distributed by the search server or transferred from other nodes, insert the requests into a local queue and broadcast the updated queue size to the other nodes of the group.
- If the received data are queue sizes on other nodes, update the local estimate of the average workload in the group.
- Evaluate the local workload relative to an estimate of the group's average workload.
- If there is no excess workload, go to the starting point to listen for incoming data.
- If the excess workload is greater than zero (or a predefined positive threshold), transfer enough requests to other nodes to return the local workload to the estimate of the group's average workload. The load balancing controller updates the local queue of requests, and broadcasts the current queue size to the other nodes.

Experimental results are shown in Sections 5.2 and 5.3 and demonstrate both the efficacy of the parallel DNA database and the efficiency of the load balancing strategy.

4.4 Summary

In this chapter, two designs for the parallel DNA database have been presented. DNA profiles are stored in a distributed fashion in both designs so that a portion of the profiles is stored in a local component database residing on each node of the parallel computer. A search engine based on the tree-structured index for stored DNA profiles

is briefly described. Two implementations of search servers for the parallel database are presented. The multi-threaded search server using PVM and object serialization demonstrates two orders of magnitude improvements in transaction time over the search server using hierarchical sockets. A parallel database design with an integrated load balancing method has also been presented, and provides a practical application for the load balancing methods discussed in Chapter 3.

Chapter 5

Experimental Results

This chapter presents the experimental results for the proposed load balancing strategies and the search results on parallel DNA databases. Section 5.1 demonstrates how the load balancing strategy works and compares experiments and simulations for the two nonlinear time delay models presented in Chapter 3. Section 5.2 shows the performance of parallel searches using a hierarchical sockets search server, and using a multi-threaded search server that provides significant performance improvement. The DNA database is constructed using a synthetic population of up to 48M (48,000,000) DNA profiles (based on the human gene statistics). Finally, in Section 5.3, the search results on parallel databases with integrated load balancing are presented.

The experimental results utilize the parallel computer cluster and other computers in the Laboratory for Information Technologies. Not all computers, purposely, have the same configuration or performance. The configuration of the computers used in these

experiments and their interconnecting network are documented in Appendix A. The computers used in these experiments are indicated by their node (host) names.

5.1 Load Balancing Strategies

Load balancing distributes computational load among multiple processors such that all processors finish their tasks at approximately the same time, thus reducing idle time in any of the processors. The theory for several load balancing strategies has been outlined in Chapter 3. In this section, simulation results using MATLAB [202] and experimental results using two implementations on a parallel machine are presented. A parallel machine has been built and used as an experimental facility for evaluation of load balancing strategies in parallel databases. The experimental setup of the parallel machine has been described in Section 2.2.1. The load balancing strategy is implemented in C++ using STL [127] on a Red Hat Linux platform. The experimental data is logged on each parallel node for evaluation and then plotted in MATLAB or Gnuplot.

Subsection 5.1.1 presents simulations and experimental results for the nonlinear model with proportional feedback control (3.3). Simulations for the nonlinear model with processor constraints (3.20) are shown in subsection 5.1.2, and results are compared with that of load balancing experiments on the parallel computer. Both simulations and experiments demonstrate the superiority of the controller using anticipated work loads over the controller using only local work load estimates.

5.1.1 Nonlinear Model with Proportional Feedback Control

A nonlinear time-delay model (3.3) was developed in Section 3.2 to characterize task transfer delays and no transferring of negative amount of tasks in a load balancing network. This model was first presented in [28] and summarized in [51]. Simulations and experimental results are documented here. The initial conditions for expected waiting times in simulations were $x_1(0) = 0.24$, $x_2(0) = 0.16$ and $x_3(0) = 0.08$, corresponding to an initial queue distribution as $q_1(0) = 600$, $q_2(0) = 400$, $q_3(0) = 200$ in experiments on the parallel machine. The average task processing time t_{p_i} is taken to be equal to $400 \mu s$ (average DNA search time).

5.1.1.1 Nonlinear Model Simulations

Simulations for load balancing with time delays are presented here. Figure 5.1 is a block diagram of one node in the load balancing simulation for the nonlinear model (3.3). Node 1 processes tasks at a rate of μ_1 and receives tasks (if any) transferred from node 2 and 3 with transfer delays h_{12} and h_{13} , respectively. The load balancing strategy calculates an estimate of average workload at node 1 using its local load x_1 and expected waiting times from other nodes (x_2 and x_3) with communication delays τ_{12} and τ_{13} , respectively. Then the local tracking difference y_1 is computed by comparing the estimate of the network average with local load x_1 . If $y_1 > 0$, the controller at node 1 portions out its excess load as u_{21} and u_{31} and sends to node 2 and node 3, respectively. By using the proportional (saturation) controllers $u_i = K_i \text{sat}(y_i)$ (in 3.3),

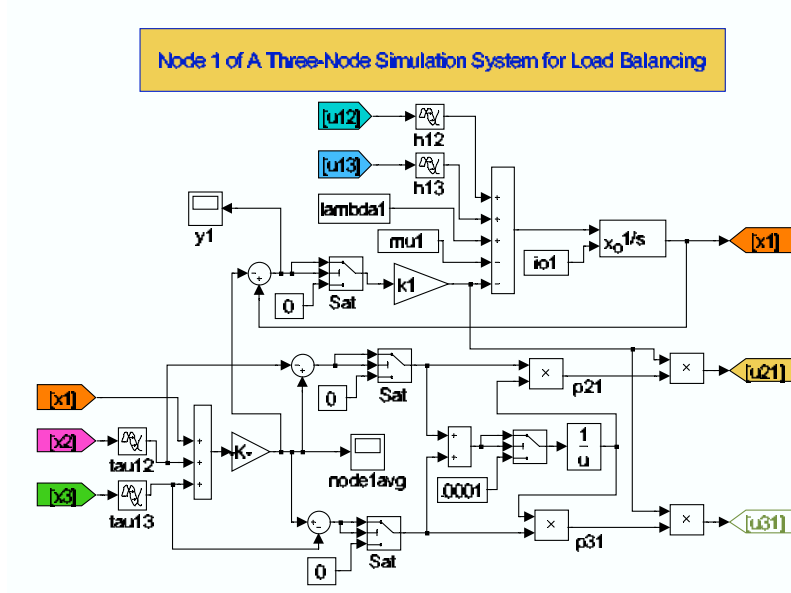


Figure 5.1: Node 1 of a three-node simulation system for load balancing.

the nonlinear model ensures no negative amount of tasks to be transferred to a recipient node.

An important point is that the actual delays experienced by the network traffic in the parallel machine are *random*. Work has been performed to characterize the bandwidth and delay on unloaded and loaded network switches, in order to identify the delay parameters of the analytic models in [63]. Experimental procedures found representative values for delays in a Gigabit network with three nodes are $\tau_{ij} = \tau = 200 \mu s$ for $i \neq j$, $\tau_{ii} = 0$, and $h_{ij} = 2\tau = 400 \mu s$ for $i \neq j$, $h_{ii} = 0$. The initial conditions for simulations were $x_1(0) = 0.24$, $x_2(0) = 0.16$ and $x_3(0) = 0.08$, while the inputs were set as $\lambda_1 = 3\mu_1$, $\lambda_2 = 0$, $\lambda_3 = 0$, $\mu_1 = \mu_2 = \mu_3 = 1$. The t_{p_i} 's were taken to be equal to $400 \mu s$ (average DNA search time) in this case.

Figure 5.2 shows the responses with the gains set as $K = 1000$ at each of the three simulation nodes. Recall that tracking errors y_i denote local load relative to its estimate of network average on each node. Load transfer probabilities p_{ij} are set to be $1/(n - 1) = 1/2$. From Figure 5.2 it can be seen the tracking errors go to zero slowly after balancing actions. Figure 5.3 shows the responses with the gains set as $K = 5000$ at each node with constant load transfer probabilities p_{ij} at $1/2$. The tracking errors go to zero quickly, but with oscillations. These figures indicate that the value of the gain K has a significant effect on the response of the system. It was found that the system did not go unstable, which is consistent with the stability analysis given in Section 3.2. However, for low values of the gains (e.g. $K = 1000$ in Figure 5.2), the response was sluggish; while for high values of the gains (e.g. $K = 5000$ in Figure 5.3), the response was oscillatory.

Oscillations in the simulations correspond to tasks transfer back and forth between nodes in the actual system. It is important to choose an appropriate controller gain to achieve load balance without oscillations. To compare with the experimental results, Figure 5.4 shows the output responses with the gains set at $K1 = 5221$, $K2 = 4883$, $K3 = 5800$, respectively. It is different from Figure 5.3 in that each node uses a slightly different gain K_i (each from a measurement in the experimental system).

5.1.1.2 Nonlinear Implementation Results

Experimental results are presented in this subsection to compare with the simulations from the model (3.3). The value $\tau = 200 \mu s$ used in the above simulations represents

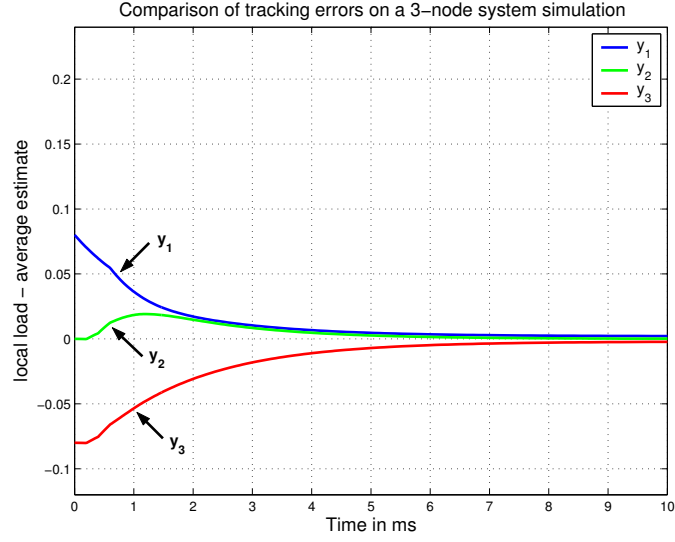


Figure 5.2: Constant p_{ij} nonlinear output responses with $K = 1000$.

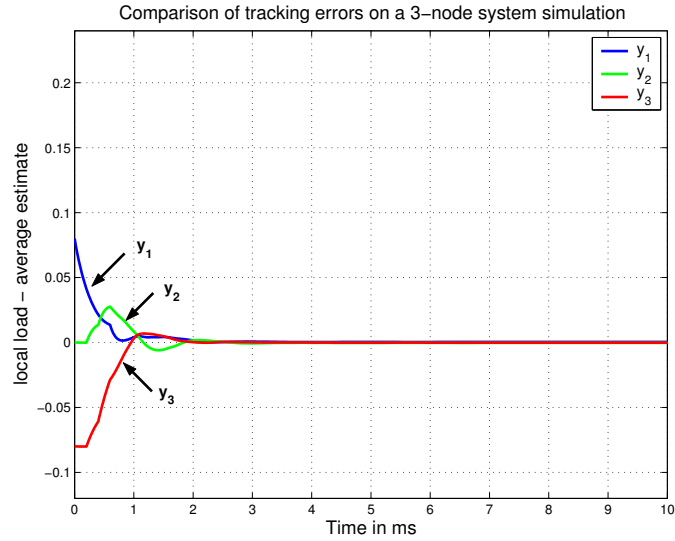


Figure 5.3: Constant p_{ij} nonlinear output responses with $K = 5000$.

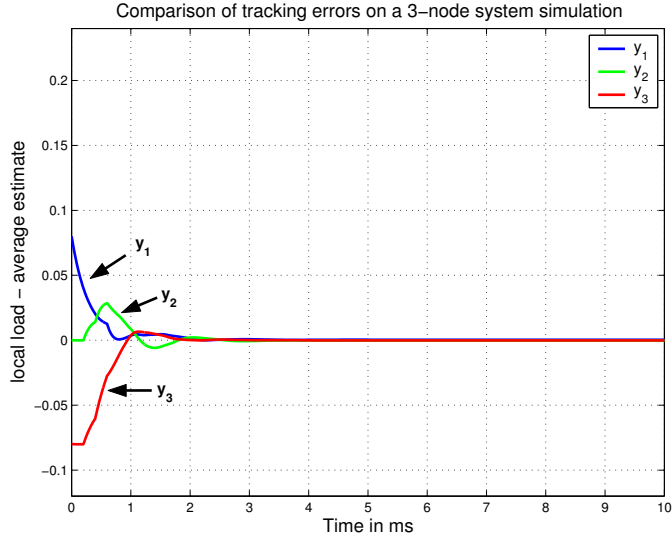


Figure 5.4: Nonlinear simulation with constant p_{ij} and $K1 = 5221$, $K2 = 4883$, $K3 = 5800$ ($K_z = 0.5$).

an average value for the delays in the experiments. Experiments are conducted to study the effect of feedback gains on the performance of load balancing. The findings of this work were first presented in [30].

To explain the connection between the control gain K_i and the actual implementation, recall that the waiting time is related to the number of tasks as $x_i(t) = q_i(t)t_{p_i}$ where t_{p_i} is the average time to carry out a task. The continuous time control law is $u_i(t) = -K \text{sat}(y_i(t))$, where $u_i(t)$ is the rate of decrease of waiting time $x_i(t)$ per unit time. Consequently, the gain K_i represents the rate of reduction of waiting time per second in the continuous time model. As the interest here is in the case where $y_i(t) > 0$, consider $u(t) = -K_i y_i(t)$. Also, $y_i(t) = \left(q_i(t) - \left(\sum_{j=1}^n q_j(t - \tau_{ij}) \right) / n \right) t_{p_i} = r_i(t) t_{p_i}$ where $r_i(t)$ is simply the number of tasks above the estimated (local) average num-

ber of tasks. With $\Delta t > 0$ the time interval between successive executions of the load balancing strategy, the control law says that a fraction of the queue $K_z r_i(t)$ ($0 < K_z < 1$) is removed in the time Δt so the rate of reduction of *waiting time* is $u(t) = -K_z r_i(t) t_{p_i} / \Delta t = -K_z y_i(t) / \Delta t$, and

$$K_i = K_z / \Delta t.$$

This shows that the gain K_i is related to the actual implementation by how fast the load balancing can be carried out and how much (fraction) of the load is transferred. In the experimental work reported here, Δt is different each time the load is balanced, so an average value for the experimental run is used as the value of Δt for a simulation when compared with an experiment. The average time t_{p_i} to process a task is the same on all nodes and is equal to $400\mu s$ while the time it takes to ready a load for transfer is about $5\mu s$. The initial conditions were taken as $q_1(0) = 600, q_2(0) = 400, q_3(0) = 200$ (corresponding to $x_1(0) = q_1(0)t_{p_i} = 0.24, x_2(0) = 0.16, x_3(0) = 0.08$). All of the experimental responses were carried out with constant $p_{ij} = 1/2$ for $i \neq j$.

Figure 5.5 is a plot of the responses $r_i(t) = q_i(t) - \left(\sum_{j=1}^n q_j(t - \tau_{ij}) \right) / n$ for $i = 1, 2, 3$ (recall that $y_i(t) = r_i(t)t_{p_i}$) with $K_z = 0.5$. The average Δt 's were measured as $95.7\mu s, 102.4\mu s, 86.2\mu s$ on each of three nodes, and this results in the values of the gains were ($K_z = 0.5$) $K_1 = 0.5/95.7\mu s = 5221, K_2 = 0.5/102.4\mu s = 4883, K_3 = 0.5/86.2\mu s = 5800$. The plot shows that the load balancing took about $6ms$ to settle. Figure 5.5 compares favorably with the simulation results shown in Figure 5.4 except for

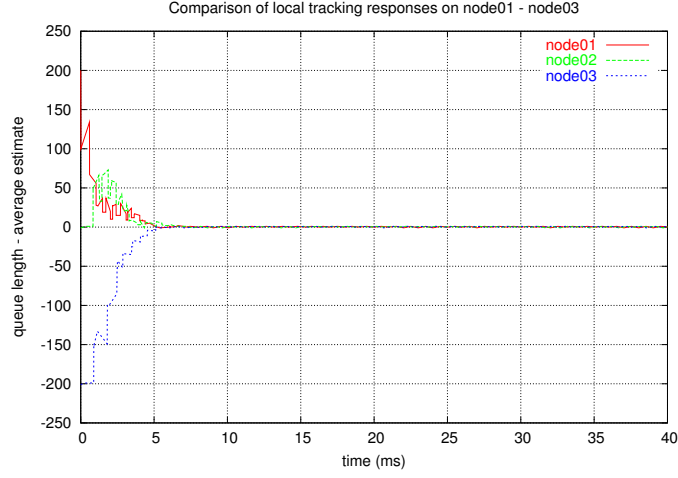


Figure 5.5: Experimental response of the load balancing strategy ($K_z = 0.5$). The average value of the gains are $K_1 = 5221$, $K_2 = 4883$, $K_3 = 5800$ with constant p_{ij} . The system settles at about 6 ms.

the time scale being off; that is, the experimental responses are slower. The explanation for this is that the time Δt is *not* modeled in the continuous time simulations. That is, the continuous time model does not stop processing jobs (at the average rate t_{p_i}) while it is transferring tasks to do the load balancing.

Figure 5.6 shows the plots of the response for the case of $K_z = 0.3$. The measured average Δt 's on three nodes were 93.1 μs , 96.6 μs , 89.9 μs , and the values of the gains were given by $K_1 = 0.3/93.1 \mu s = 3222$, $K_2 = 0.3/96.6 \mu s = 3106$, $K_3 = 0.3/89.9 \mu s = 3338$. The plot shows that the load balancing took about 15 ms to settle.

Figure 5.7 shows the plots of the response for the case of $K_z = 0.2$. The measured average Δt 's on three nodes were 90 μs , 117 μs , 100 μs , and therefore the gains were $K_1 = 2220$, $K_2 = 1706$, $K_3 = 1998$. The load balancing took about 22 ms to settle.

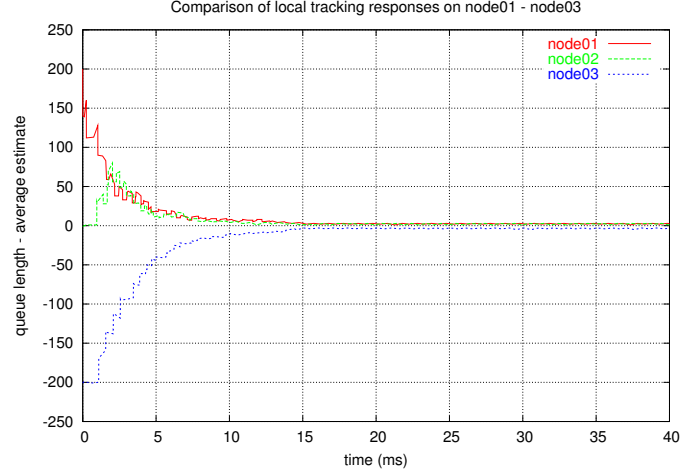


Figure 5.6: Experimental response of the load balancing strategy ($K_z = 0.3$). The average value of the gains are $K_1 = 3222$, $K_2 = 3106$, $K_3 = 3338$ with constant p_{ij} . The system settles at about 15 *ms*.

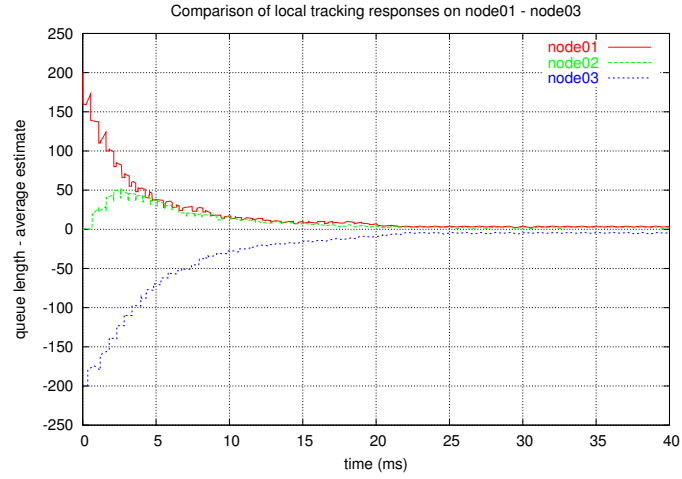


Figure 5.7: Experimental response of the load balancing strategy ($K_z = 0.2$). The average value of the gains are $K_1 = 2220$, $K_2 = 1706$, $K_3 = 1998$ with constant p_{ij} . The system settles at about 22 *ms*.

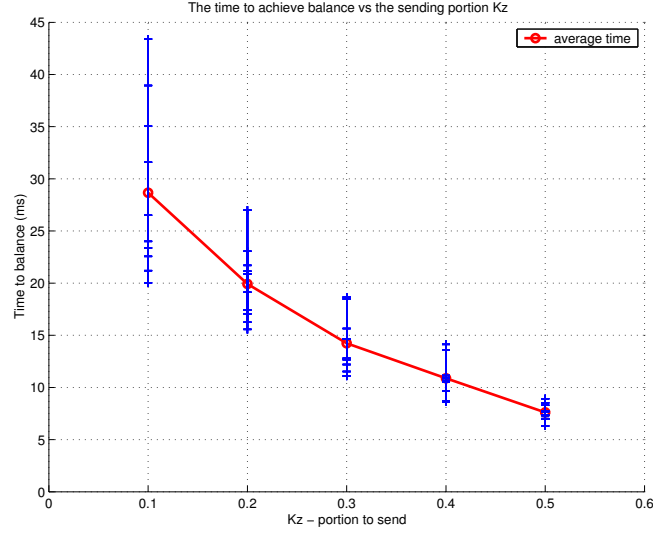


Figure 5.8: Summary of settling time as a function of the feedback gain K_z .

Figure 5.8 summarizes the data from several experimental runs of the type shown in Figures 5.5, 5.6, and 5.7. For $K_z = 0.1, 0.2, 0.3, 0.4$, and 0.5 , ten runs were made and the settling times (time to load balance) were determined. These are marked as small horizontal ticks in Figure 5.8. For all such runs, the initial queues were the same and equal to $q_1(0) = 600, q_2(0) = 400, q_3(0) = 200$. For each value of K_z , the average settling time for these ten runs was computed and is marked as a dot on the Figure 5.8. For values of $K_z = 0.6$ and higher with increments of 0.1 in K_z , consistent results could not be obtained. In many cases, ringing extended throughout the 200 ms time interval for the experiments.

For example, Figure 5.9 shows the plots of the queue length less the local queue average for an experimental run with $K_z = 0.6$ where the settling time is approximately

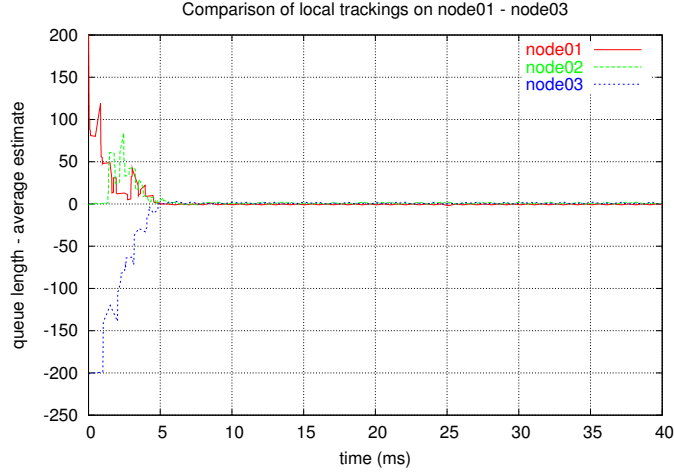


Figure 5.9: Experimental response of the load balancing with $K_z = 0.6$. The settling time is approximately 7 *ms*.

7 *ms*. In contrast, Figure 5.10 shows the experimental results under the same conditions where persistent ringing regenerates for 40 *ms*. It was found the response was so oscillatory that it was not possible to accurately determine a settling time. However, Figure 5.8 shows that it is desirable to choose the gain to be close to 0.5 to achieve a faster response time without breaking into oscillatory behavior.

5.1.1.3 Results on Randomly Generated Tasks

Instead of using the initial tasks only, randomly generated tasks are used to test the balancing strategy's performance. This requires dynamic load balancing to even the tasks on all queues. This experiment shows the load balancing results for randomly generated tasks on three dual processor nodes: Node00, Node09 and Zero. The three nodes are connected via a Gigabit network.

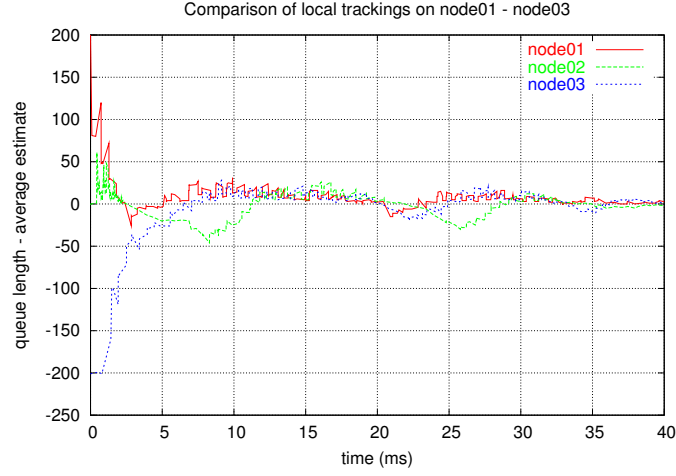


Figure 5.10: Experimental response of the load balancing with $K_z = 0.6$: ringing. The same conditions as Figure 5.9, but now the ringing persists, due to random network traffic delays. Note that the system settles beyond 40 *ms*.

Figure 5.11 is a plot of the responses for workload, local estimate of average and tracking difference on Node00. The upward transitions correspond to task insertions in a search queue while the downward transitions correspond to the removal of tasks. When the workload is higher than the average estimate, the load balancer will shift some amount of tasks to other nodes. When the workload is lower than the average, the node is more likely to receive tasks transferred from other nodes. The experimental responses for workload, local estimate of average and tracking difference on Node09 and Zero are shown in Figure 5.12 and Figure 5.13 respectively. From the figures it can be seen that the three nodes each peak loading at similar times, and the local averages track the workloads quite well.

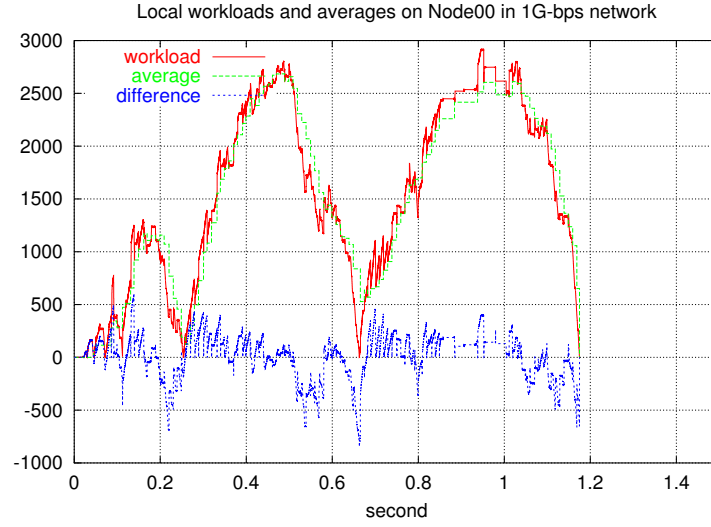


Figure 5.11: Experimental response of load balancing on Node00. Workload: red line, local average: green dash, and tracking difference: blue dash.

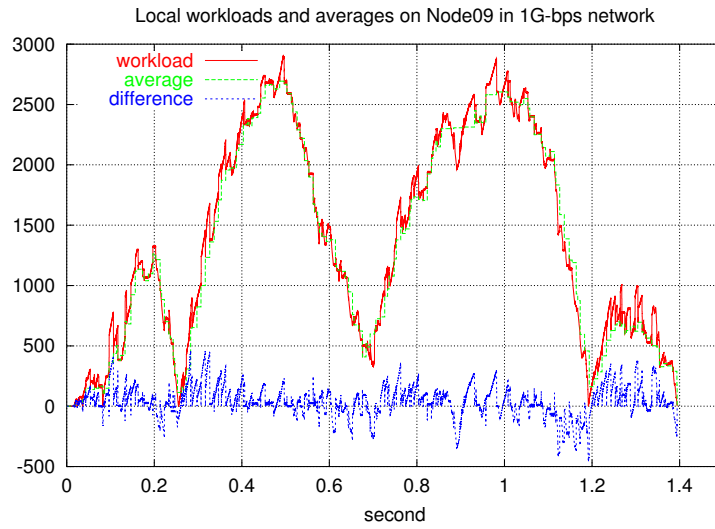


Figure 5.12: Experimental response of load balancing on Node09. Workload: red line, local average: green dash, and tracking difference: blue dash.

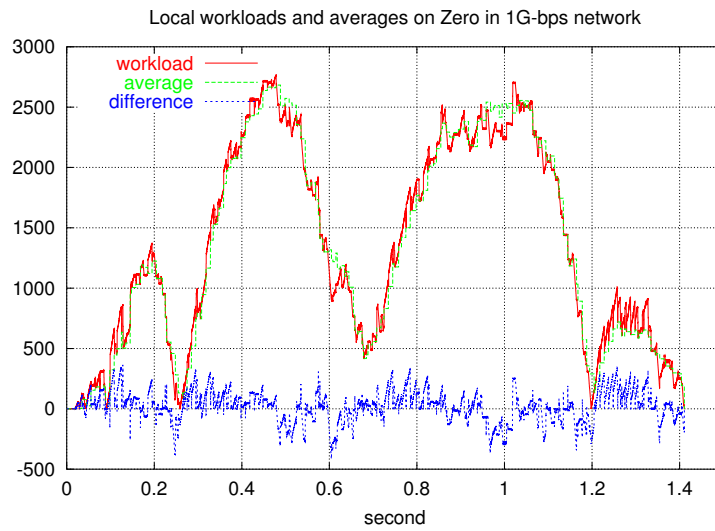


Figure 5.13: Experimental response of load balancing on Zero. Workload: red line, local average: green dash, and tracking difference: blue dash. Similar behavior as that shown in Figures 5.11 and 5.12.

5.1.2 Nonlinear Model with Resource Constraints

Load balancing operation requires processor time, which can not be used to process tasks. Consequently, there is a trade-off between using processor time to process tasks and using processor time/network bandwidth to distribute the load evenly between the nodes to reduce overall processing time. The simulations of the previous model (Figure 5.4) showed that the load balancing would be carried out faster than occurred in the corresponding experimental results (Figure 5.5). This motivates a refinement of the model to account for processor constraints. A new deterministic nonlinear time-delay system model was developed to capture these constraints in [196], and the closed loop load balancing strategy was summarized in [195].

Experimental results are presented in this section to study the effects of time delays and resource constraints in load balancing. Subsection 5.1.2.1 presents a Simulink model for the nonlinear time-delay system (3.20) discussed in Section 3.3. Experiments show a correlation of the Simulink model with the actual implementation. In addition, transients found in local average estimates and tracking differences can be suppressed by a strategy of packing the updated queue size with transferred tasks on a sending node.

Subsection 5.1.2.2 explains the system dynamics including the effects of time delays in load balancing. Experimental results for the open loop load balancing are also presented to illustrate the difference between two load transfer probability functions (3.31) and (3.36): p_{ij} based on local estimates of expected waiting times x_i and p_{ij} based on

anticipated waiting times z_i , respectively. In subsection 5.1.2.3, the closed loop controller (3.30) with the p_{ij} specified by (3.31) is compared with the closed loop controller (3.35) with the p_{ij} specified by (3.36). Load balancing results on a larger network of multiple nodes are presented in subsection 5.1.2.4. The first set of experiments uses queues of initial conditions with no subsequent tasks. The second set of experiments is conducted using a dynamic task generation scheme. These experiments demonstrate the superiority of the controller based on anticipated waiting times z_i over that based on local estimates of waiting times x_i .

5.1.2.1 Simulink Model and Transient Suppression

This subsection presents Simulink simulations for the nonlinear time-delay model for load balancing with resource constraints. Figure 5.14 is a block diagram of one node in the load balancing simulation system for this nonlinear model (3.20). Node 1 processes tasks at a rate of μ_1 and receives tasks (if any) transferred from node 2 and 3 with transfer delays h_{12} and h_{13} , respectively. Node 1 receives expected waiting times of other nodes experienced communication delays τ_{12} and τ_{13} , and uses the delayed values to estimate average load across the network. The load balancing strategy at node 1 evaluates tracking errors y_1 using local load x_1 relative to the average estimate. A hysteresis controller η_i is introduced to capture processor constraints. When $y_1 > 0$, node 1 has excess load to be transferred out, $\eta_1 = 1$, processor time is used to transfer tasks (such as packing task data and copying to a sending buffer); no tasks can be processed at the same time. When $\eta_1 = 0$, processor time is used to process tasks, and

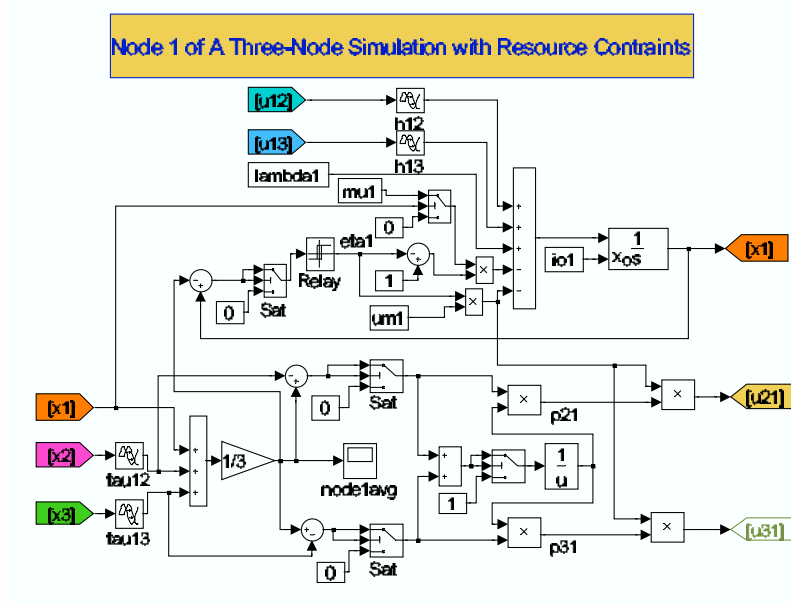
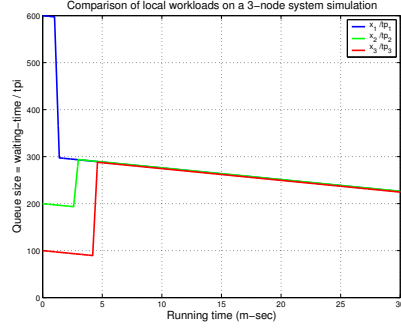


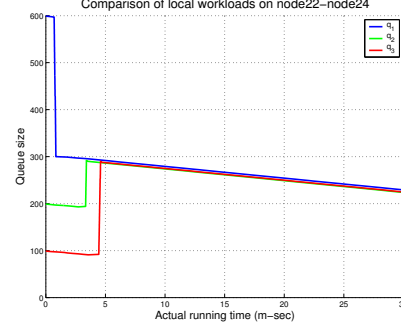
Figure 5.14: Node 1 of a three-node simulation system with all nonlinearities.

no tasks are exchanged.

Initial queues as (600, 200, 100). This shows an experiment with initial conditions (600, 200, 100) for the queue sizes on node22, node23 and node24, respectively. The (open-loop) load balancing strategy was executed at $t_0 = 1 \text{ ms}$. The delay values are $h_{21} = 1.6 \text{ ms}$, $h_{31} = 3.2 \text{ ms}$. The communication delay $\tau_{ij} = 150 \text{ } \mu\text{s}$ and the task processing time $tp_i = 400 \text{ } \mu\text{s}$. A simulation under the same set of conditions was done using the Simulink model shown in Figure 5.14. The simulation workloads and experimental results are shown in Figures 5.15(a) and 5.15(b) respectively. From the figures, it is evident that the simulation shows a correlation with the actual implementation of load balancing.



(a) simulation workloads

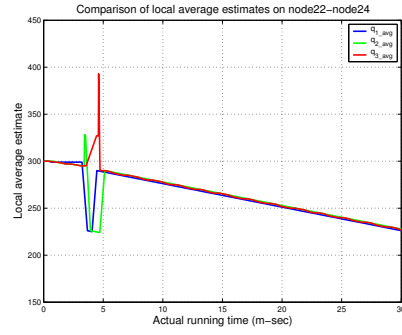


(b) experimental results

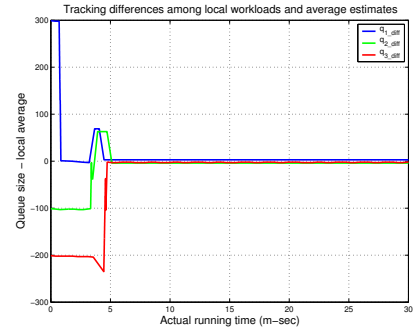
Figure 5.15: Simulation and experimental results on a three-node system. Initial queue sizes on node22, node23 and node24 are (600, 200, 100).

Figures 5.16(a) and 5.16(b) show the average estimates and tracking differences on the experimental three-node system. Due to delayed information of loads at other nodes, the averages deviate from the ideal average value 300, as shown in Figure 5.16(a). For the same reason, the tracking differences on three nodes in Figure 5.16(b) do not go to zero (ideally right after the balancing action) until they receive new broadcasts of queue sizes from other nodes.

Figure 5.17 shows local workloads and average estimates on node23 (a recipient node) from the experiment using initial queue sizes of (600, 200, 100). From the figure, it can be seen there is a transient on the curve of local average estimates. After node21 sends 100 tasks to node23 and 200 tasks to node24 to balance the load, node23 will receive 100 tasks from node22 and update its queue.



(a) average estimates



(b) tracking differences

Figure 5.16: Average estimates and tracking differences on a three-node system.

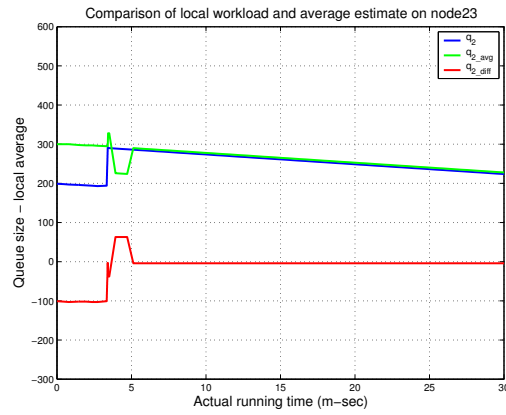


Figure 5.17: Workloads and average estimates on node23.

In Figure 5.17, due to a communication delay, node23 updates its average estimate as a transient until it receives broadcast of queue sizes on other nodes. This causes a skewed estimate of average workload of the system and impacts the performance of load balancing. An approach to suppress transients in load balancing is presented as follows.

Transient suppression. Transients are apparent in the above tests (Figures 5.16 and 5.17), and these are caused by the delays between each transfer of tasks and the next broadcast of queue status. One approach that can reduce these transients is to pack the queue size (say, updated size on node 1 after removing tasks from its queue for transfer) with the transferred task data, so that the recipient node can update its estimate of the transmitting node's status as it unpacks task data without waiting for the next broadcast from that node. The following experiment demonstrates the effect of this strategy. Initial queue sizes on node22, node23 and node24 are set to be (600, 200, 100), respectively. The load balancing strategy executes at $t_0 = 1 \text{ ms}$.

Local estimates of average loads and tracking differences for load balancing with transient suppression are shown in Figures 5.18(a) and 5.18(b), respectively. Note the differences from the load balancing results without transient suppression (as shown in Figures 5.16(a) and 5.16(b)). By packing queue size into task transfer, each recipient node updates its estimate of average load (and the corresponding tracking difference) without transients. The local workloads and average estimates on node23 under this test are shown in Figure 5.19. Compared with Figure 5.17 for node23, there are no transients shown on the plots of average estimates and tracking difference.

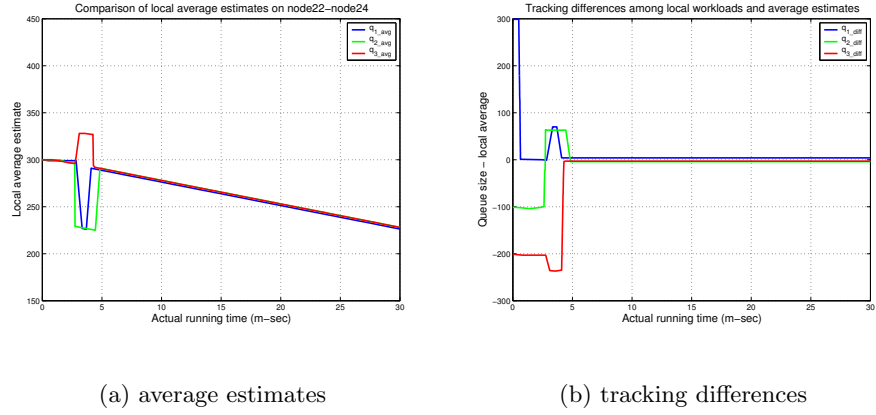


Figure 5.18: Experimental results for load balancing with transient suppression. Initial queue sizes on node22, node23 and node24 are (600, 200, 100).

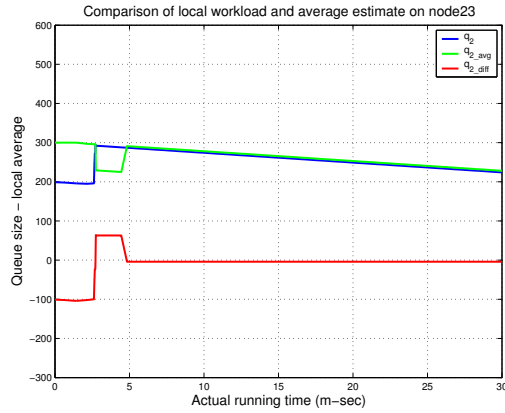


Figure 5.19: Workloads and average estimates on node23 with transient suppression.

The transient suppression helps to avoid skewed estimates of average loads, but cannot prevent initiation of unnecessary transfers (due to delayed queue sizes) on the highly-loaded node. This difficulty for load balancing is addressed in the next subsection, and is solved by using anticipated waiting times of other nodes.

5.1.2.2 Open Loop Experiments

In this set of experiments, load balancing is performed *once at a fixed time* and is referred to as an “open loop” run. This open loop run is done to facilitate an explanation of the system dynamics with the effects of the delays. In the open loop load balancing experiments, two cases of load transfer probabilities are used, and the balancing performances are compared. Recall that the load transfer probability p_{ij} defines how a sending node portions out its excess load to other nodes. The first case uses a load transfer probability p_{ij} specified by (3.31), which is based on local estimates of loads at other nodes x_i . The second case uses a load transfer probability p_{ij} specified by (3.36), which is based on the anticipated values of loads at other nodes z_i . In the following experiments, an initial queue distribution is $q_1(0) = 600$ tasks, $q_2(0) = 200$ tasks, and $q_3(0) = 100$ tasks. The average time to do a search task is $t_{p_i} = 400 \mu s$, and the inputs are set as $\lambda_1 = 0, \lambda_2 = 0, \lambda_3 = 0$.

Case 1: Load transfer probability p_{ij} based on x_i . This experiment shows the response of the queues versus time using p_{ij} based on local waiting times x_i , with an initial queue distribution of $q_1(0) = 600$ tasks, $q_2(0) = 200$ tasks and $q_3(0) = 100$

tasks. The average time to do a search task is $t_{p_i} = 400 \mu s$, and the inputs are set as $\lambda_i = 0, i = 1, 2, 3$. In this open loop experiment, the software was written to execute the load balancing strategy at $t_0 = 1 ms$ using the load transfer probability p_{ij} as specified by (3.31), which is based on local estimates of loads at other nodes x_i . Figure 5.20 shows queue sizes versus time on three nodes, and that the data transfer delay from node 1 to node 2 is $h_{21} = 1.8 ms$, while the data transfer delay from node 1 to node 3 is $h_{31} = 2.8 ms$.

Figure 5.21 is a plot of each nodes' estimate of the network average, i.e.,

$$q_{i_avg}(t) \triangleq \left(\sum_{j=1}^n q_j(t - \tau_{ij}) \right) / n$$

where $q_{i_avg}(t) = x_{i_avg}(t)/t_{p_i}$ (see (3.29)). Figure 5.22 is a plot of the queue size relative to its estimate of the network average, i.e.,

$$q_{i_diff}(t) \triangleq q_i(t) - q_{i_avg}(t) = q_i(t) - \left(\sum_{j=1}^n q_j(t - \tau_{ij}) \right) / n$$

for each of the nodes.

Note the effect of the delay in terms of what each local node *estimates* as the queue average and therefore whether it determines its load to be above or below the average. Figure 5.21 is now discussed in detail as follows. All time symbols for this experimental run are shown in Figure 5.22.

At the time of load balancing $t_0 = 1 ms$, node 1 computes its queue size *relative* to

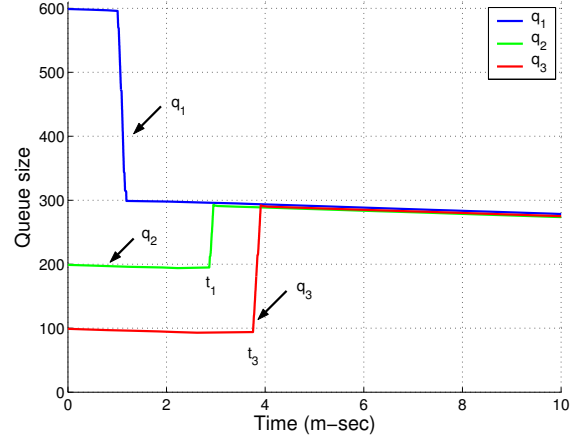


Figure 5.20: Results of the load balancing strategy executed at $t_0 = 1$ ms. The system is balanced after q_3 at node 3 receives the load transferred from q_1 at node 1.

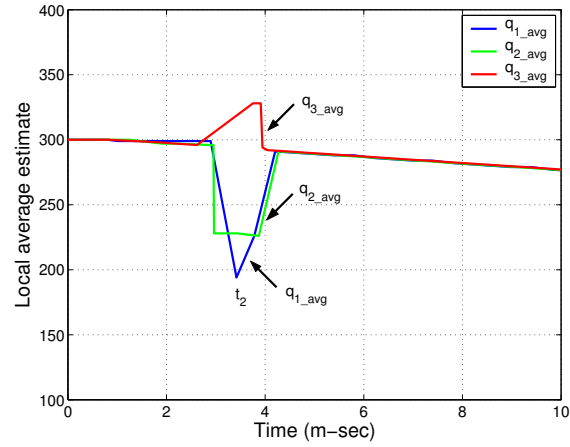


Figure 5.21: Local estimates of the network average load on three nodes. $q_{i_avg}(t)$ for node i , $i = 1, 2, 3$ in an open loop load balancing experiment.

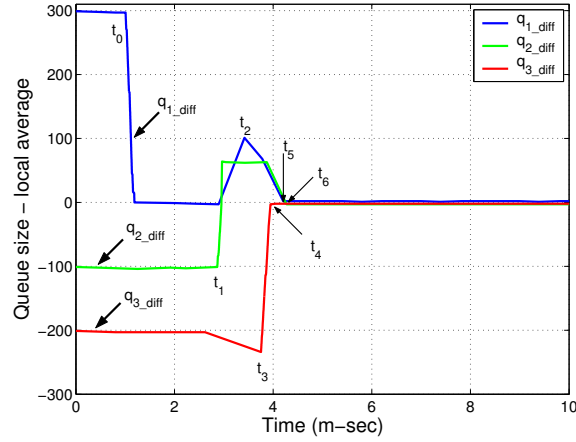


Figure 5.22: Local queue size relative to its estimate of the network average on each of three nodes. $q_{i_diff}(t) = q_i(t) - q_{i_avg}(t)$ for $i = 1, 2, 3$ using p_{ij} specified by (3.31). Node 1 portions out the excess load at $t = t_0$, but the tracking differences $q_{i_diff}(t)$ are not equal to zero until $t = t_6$ due to delays.

its estimate of the network average q_{1_diff} to be 300, node 2 computes its queue size *relative* to its estimate of the network average q_{2_diff} to be -100 and node 3 computes its queue size *relative* to its estimate of the network average q_{3_diff} to be -200 . Node 1 sends tasks out according to load transfer probability equation (3.31), which transfers about 100 and 200 tasks to node 2 and node 3 respectively.

At time t_1 node 2 receives 100 tasks from node 1 and updates its local queue size to be 300 as well as its estimate of node 1's queue size to be (about) 300, so that node 2's estimate of the network average is now $(300 + 300 + 100)/3 \approx 233$, and its estimate of its queue size relative to the estimate of the network average is now $q_{2_diff} \approx 300 - 233 = 67$. That is, node 2 is not aware of the 200 tasks being sent from node 1 to node 3.

At time t_2 node 1 has already sent out the tasks to nodes 2 and 3 and updated to

its local queue size to 300. Node 1 receives the broadcast of two queue sizes of nodes 2 and 3 which are still 200 and 100 respectively, as the task transfers from node 1 are still on the way to nodes 2 and 3. Thus, node 1's estimate of the network average is about $(300 + 200 + 100)/3 = 200$ making the computation of its queue size relative to the estimate of the network average now $q_{1_diff} \approx 300 - 200 = 100$.

At time t_3 , node 3 receives the queue size of node 2 (which has already increased to about 300 as shown in Figure 5.20). Node 3 now computes q_{3_diff} to be about $(100 - (600 + 300 + 100)/3) \approx -233$.

At time t_4 , node 3 finally receives the 200 tasks from node 1 and updates its queue size and its estimate of node 1's queue size to be about 300. The network average computed by node 3 is now about $(300 + 300 + 300)/3 = 300$ so that $q_{3_diff} \approx (300 - 300) = 0$.

At time t_5 , node 1 receives the communications for the queue sizes of node 2 and node 3 (which is now about 300 - see Figure 5.20). Then node 1 computes $q_{1_diff} \approx (300 - (300 + 300 + 300)/3) = 0$.

Finally, at time t_6 , node 2 receives the queue size of node 3 (which is now about 300 - see Figure 5.20). Node 2 now computes its queue size relative to local average $q_{2_diff} \approx (300 - (300 + 300 + 300)/3) = 0$.

The transfer of 200 tasks to node 3 takes more time than that of 100 tasks to node 2. It is the task transfer delay that delays node 2's receipt of the new queue size of node 3 at time t_1 , and that delays node 3's receipt of the tasks until $t_4 > t_3$. It is the

communication delay that delays node 1's receipt of the new queue size of node 2 at time t_2 . The effect of both delays could cause the unnecessary transfers for node 2 at time t_1 and node 1 at time t_2 if the load balancing is done in closed loop.

Case 2: Load transfer probability p_{ij} based on z_i . The node that transfers tasks computes the amounts to be sent to other nodes according to how far below the network average its estimate of each recipient node's queue is. Therefore, it is feasible to send the amounts of its next task transfers to each of other nodes before actually transferring tasks. Such communications are efficient; the communication delay of each transferred measurement is much smaller than the actual task transfer delays. A key finding of this research is that knowledge of the anticipated queue sizes can be used to compensate the effect of delays.

Figure 5.23 is a plot of the local average of queue sizes on all nodes using measurements of anticipated transfers

$$q_{i_avg}^{est}(t) \triangleq \left(\sum_{j=1}^n q_j^{est}(t - \tau_{ij}) \right) / n = \left(\sum_{j=1}^n q_j(t - \tau_{ij}) + q_{net_j}(t) \right) / n$$

where $q_{net_j}(t) = \sum_{i=1}^n q_{net_{ji}}(t - \tau_{ij})$. The nodes have an initial queue distribution of $q_1(0) = 600$ tasks, $q_2(0) = 200$ tasks and $q_3(0) = 100$ tasks. The average time to do a search task is $t_{p_i} = 400 \mu s$, and the inputs are $\lambda_1 = 0, \lambda_2 = 0, \lambda_3 = 0$. Note that $q_{i_avg}^{est}(t) = z_{i_avg}(t)/t_{p_i}$ (see (3.33)). In this experiment, the load balancing strategy was executed at $t_0 = 1 ms$ using the load transfer probability p_{ij} as specified by (3.36), which

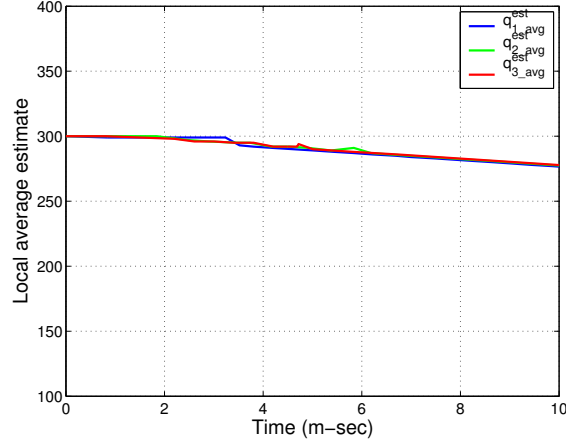


Figure 5.23: Anticipated estimates of the network average load on three nodes. $q_{i_avg}^{est}(t) = \left(\sum_{j=1}^n q_j^{est}(t - \tau_{ij}) \right) / n$ for $i = 1, 2, 3$ using p_{ij} specified by (3.36). Note that the anticipated estimates of the network average on three nodes are close to each other.

is based on anticipated values of loads at other nodes z_i . Compared with Figure 5.21, the estimates by the nodes of the network average are substantially improved.

Figure 5.24 is a plot of the queue size relative to the local average

$$q_{i_diff}^{est}(t) \triangleq q_i(t) - q_{i_avg}^{est}(t) = q_i(t) - \left(\sum_{j=1}^n q_j^{est}(t - \tau_{ij}) \right) / n$$

for each of the nodes. Compared with Figure 5.22, the tracking differences are near or below zero, and no further unnecessary transfers are initiated.

Figures 5.25(a) and 5.25(b) show the estimates of the queue sizes by node 2 when using x_i and when using z_i , respectively. In Figure 5.25(a), after receiving the tasks from node 1, node 2 computed its queue size relative to its local average to be 67.

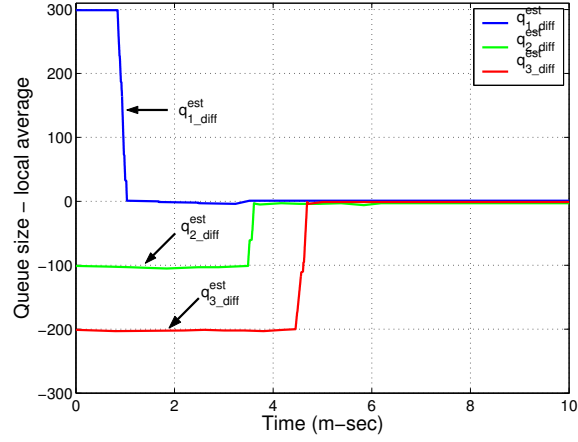
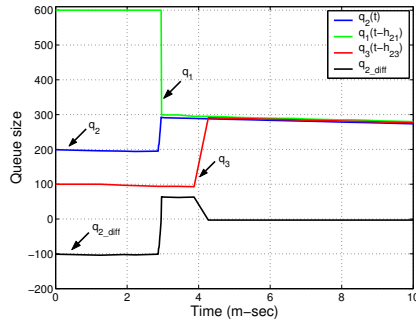
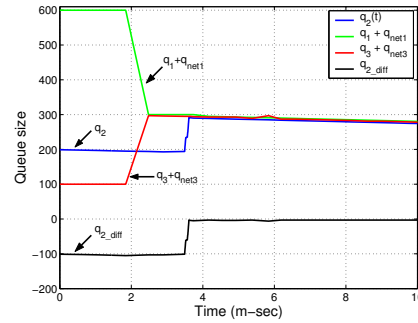


Figure 5.24: Local queue size relative to the anticipated estimate of the network average on each of three nodes. $q_{i_diff}^{est}(t) = q_i(t) - q_{i_avg}^{est}(t)$ for $i = 1, 2, 3$ using p_{ij} specified by (3.36). Note that the tracking differences $q_{i_diff}^{est}(t)$ are below or equal to zero after node 1 portions out its excess load.



(a) node 2: using x_i



(b) node 2: using z_i

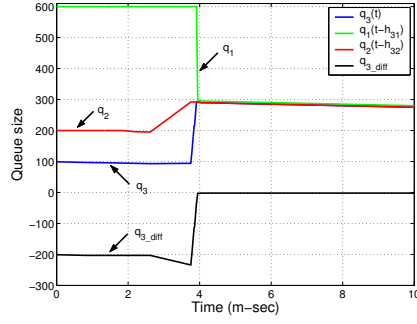
Figure 5.25: Estimates of the queue sizes at node 2. (a) Using x_i and (b) using z_i . Note that the tracking difference q_{2_diff} when using x_i is above zero due to transfer delays.

In Figure 5.25(b), node 1 sends the numbers of tasks to be transferred before actually transferring tasks to the other nodes; node 2 receives the estimate of node 3's queue size and updates its local queue size relative to its estimate of the network average to be zero.

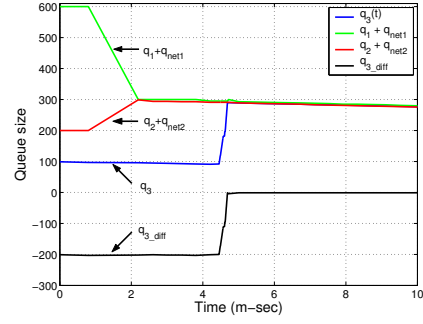
Figures 5.26(a) and 5.26(b) show the estimates of the queue sizes by node 3 when using x_i and when using z_i , respectively. In Figure 5.26(a), when node 2 broadcasts its updated queue size to node 3, node 3 has not received the tasks sent from node 1 due to transfer delays. Node 3 updates its queue size relative to its estimate of the network average to be -233 . In Figure 5.26(b), node 1 sends the numbers of tasks to be transferred before transferring actual tasks to the other nodes; node 3 receives the anticipated estimate of the queue sizes on other nodes, and computes its queue size relative to its estimate of the network average to -200 . When node 3 receives the tasks transferred from node 1, it updates its local queue and computes the amount relative to the network average to be zero. Compared with Figures 5.25 and 5.26 shown above, these anticipated estimates significantly compensate the effect of delays of task transfers. The comparison of local queue sizes and anticipated estimates on the two recipient nodes 2 and 3 are shown in Figures 5.27(a) and 5.27(b) respectively.

5.1.2.3 Closed Loop Experiments

The previous subsection discussed differences between using a load transfer probability p_{ij} based on local estimates of loads at other nodes x_i and using a p_{ij} based on anticipated values of loads at other nodes z_i . Experimental results of open loop load balancing

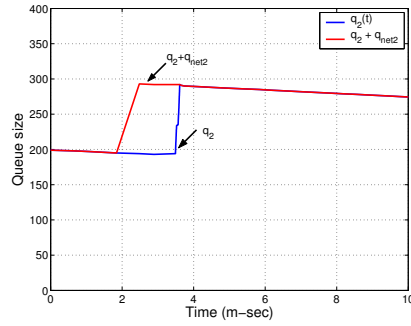


(a) node 3: using x_i

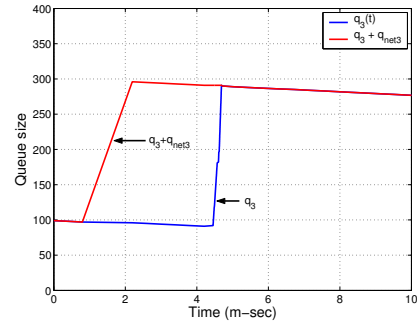


(b) node 3: using z_i

Figure 5.26: Estimates of the queue sizes at node 3. (a) Using x_i and (b) using z_i . Note that the tracking difference $q_{2,diff}$ when using x_i is above zero due to transfer delays.



(a) node 2

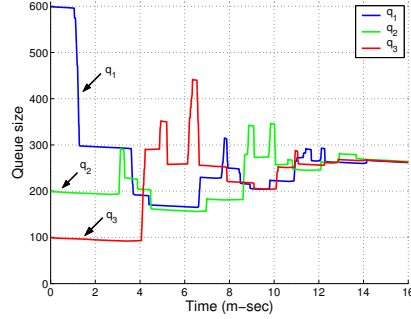


(b) node 3

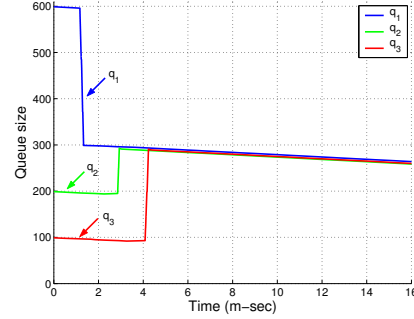
Figure 5.27: Comparison of the actual queue size and the anticipated estimate of queue size. (a) At node 2 and (b) node 3. Note that each node has an anticipated estimate of load by taking into account the tasks being transmitted in the network.

showed that using p_{ij} based on z_i can help to bring the estimate of average workload on each node together and avoid unnecessary transfers caused by skewed tracking differences (due to delayed information of loads at other nodes). For dynamic load balancing, balancing actions need to be run in closed loop. Two feedback controllers based on x_i and z_i were addressed in Section 3.3.2. Here, experiments are conducted to evaluate closed loop load balancing using the two schemes with different load transfer probability equations. The first experiment shows the balancing results on three nodes with initial tasks only, and the second experiment shows the balancing results for the case with task generation.

Closed Loop: Initial tasks only. The following experiments show the responses using closed loop load balancing with an initial queue distribution of $q_1(0) = 600$ tasks, $q_2(0) = 200$ tasks and $q_3(0) = 100$ tasks. The average time to do a search task is $400 \mu s$. In these experiments the inputs were set as $\lambda_1 = 0, \lambda_2 = 0, \lambda_3 = 0$. That is, there are only initial tasks to be processed and balanced. After initial communications, the closed loop load balancing strategy (see Section 3.3) was initiated using the p_{ij} as specified by (3.31) and by (3.36), respectively. Figures 5.28(a) and 5.28(b) show the responses of the queue sizes versus time using p_{ij} specified by (3.31) and p_{ij} specified by (3.36) in closed loop balancing, respectively. Note the substantial difference in the load balancing performance between these two schemes. In Figure 5.28(a), there are unnecessary task transfers due to delayed information of loads at other nodes, while the system settles quickly in Figure 5.28(b) by using anticipated values of loads at other nodes.



(a) queue sizes: using x_i

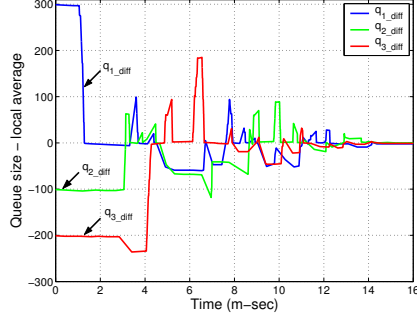


(b) queue sizes: using z_i

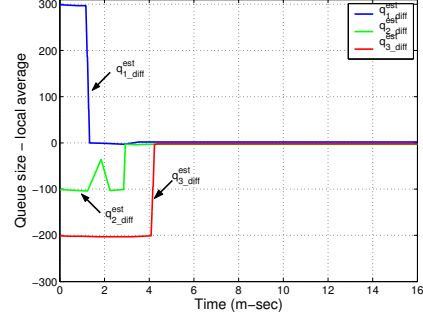
Figure 5.28: Results of closed loop load balancing on three nodes. (a) Queue size using x_i and p_{ij} (3.31), and (b) queue sizes using z_i and p_{ij} (3.36).

The tracking differences between local queue sizes and estimated network averages (from delayed values of other nodes) using p_{ij} by (3.31) are shown in Figure 5.29(a). There are unnecessary exchanges of tasks back and forth among nodes. Although the system reaches a balanced condition at around $t = 14 \text{ ms}$, those additional transfers cost processing time and prolong the completion time. Figure 5.29(b) shows the tracking differences between local queue sizes and anticipated estimates of the network averages using p_{ij} by (3.36) based on z_i .

Figure 5.30(a) shows node 2's estimates of the queue sizes in the network of closed loop load balancing using p_{ij} based on the x_i . Node 2 estimates the network average using the delayed information from other nodes only, and its controller 3.30, based on its queue size relative to the estimated average, causes the unnecessary exchanges of tasks back and forth, as shown in Figure 5.30(a). Figure 5.30(b) shows node 2's estimates of

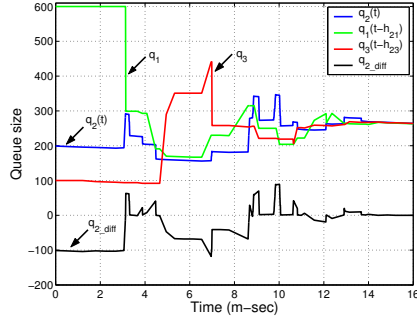


(a) $q_{i-diff}(t)$ by (3.31) based on x_i

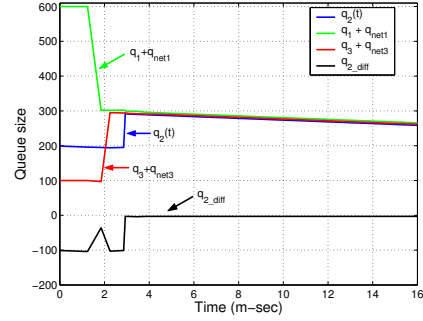


(b) $q_{i-diff}^{est}(t)$ by (3.36) based on z_i

Figure 5.29: Tracking difference on each of three nodes in closed loop load balancing. (a) $q_{i-diff}(t) = q_i(t) - \left(\sum_{j=1}^n q_j(t - \tau_{ij}) \right) / n$ and (b) $q_{i-diff}^{est}(t) = q_i(t) - \left(\sum_{j=1}^n q_j^{est}(t - \tau_{ij}) \right) / n$ for $i = 1, 2, 3$.



(a) node 2: using x_i



(b) node 2: using z_i

Figure 5.30: Estimated queue sizes by node 2 in closed loop load balancing. (a) Using x_i and p_{ij} (3.31), and (b) using z_i and p_{ij} (3.36).

the anticipated queue sizes in the network of closed loop load balancing using p_{ij} based on the z_i . Node 1 sends the numbers of tasks before actually transferring tasks to the other nodes. Node 2 receives the anticipated estimate of node 3's queue size and updates its queue size relative to its estimate of the network average as zero. From Figure 5.30, it is evident that the anticipated estimates are used to compensate the effect of delays of task transfers so that there are no unnecessary task transfers initiated.

Closed Loop: Task generation. The previous experiments presented the load balancing on a network of three nodes with queues of initial tasks only. It is interesting to see how the load balancing dynamically performs in the case of task generation. The inputs in this experiment are $\lambda_1 = 3\mu_1 = 3$, $\lambda_2 = 0$, and $\lambda_3 = 0$. That is, a number of new tasks (λ_1) is generated on node 1 in each processing loop. This creates increases of tasks in the queues for a dynamic load balancing. The initial queue distribution is $q_1(0) = 600$ tasks, $q_2(0) = 200$ tasks and $q_3(0) = 100$ tasks.

Figures 5.31 and 5.32 show the responses of the queues versus time using the p_{ij} specified by (3.31) and by (3.36), respectively, with task generation in the process of execution. The staircase-like increases of queue size corresponds to task generation on node 1 with $\lambda_1 = 3$. Note the difference in the load balancing performance between these two schemes using different controllers. In Figure 5.31, unnecessary transfers cause tasks to be exchanged back and forth among the three nodes. The transfers cost additional processing time and result in a longer completion time. In Figure 5.32, the system settles to a balanced state much faster by using the anticipated waiting times.

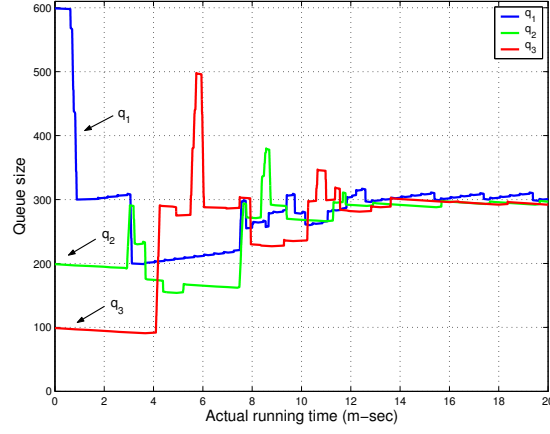


Figure 5.31: Queue sizes using controller (3.30) and p_{ij} (3.31) based on local waiting times x_i . Load balancing is performed in closed loop with task generation parameter $\lambda_1 = 3$.

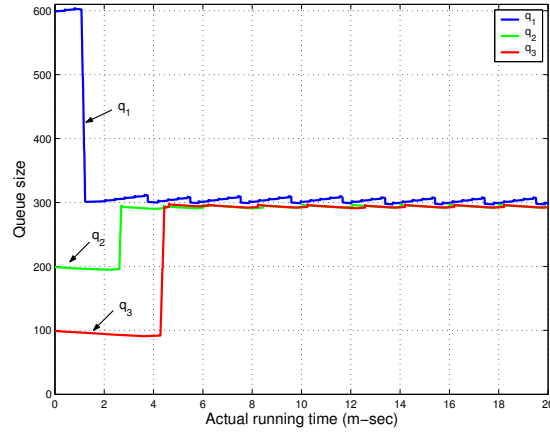


Figure 5.32: Queue sizes using controller (3.35) and p_{ij} (3.36) based on anticipated waiting times z_i . Load balancing is performed in closed loop with task generation parameter $\lambda_1 = 3$.

5.1.2.4 Results on Multiple Nodes

The following experiments are presented to show the load balancing performance on a larger network of multiple nodes ($n = 6$). The first set of experiments shows the load balancing on six nodes with queues of initial tasks only, while the load balancing results on six nodes with task generation is shown in the second set of experiments.

Multiple Nodes: Initial tasks only. In this experiment, the initial queue distribution is set to be $q_1(0) = 650$ tasks, $q_j(0) = 50$ tasks on node j where $j = 2, 3, 4, 5, 6$. The average time to do a search task is $t_{p_i} = 400 \mu s$, and the inputs are set as $\lambda_i = 0, i = 1, \dots, 6$. That is, there are only initial tasks in the queues to be processed and balanced.

Figure 5.33 shows the responses of the queue sizes versus time using the p_{ij} specified by (3.31) in a network of six nodes. In Figure 5.33, the closed loop controller based only upon the delayed values of waiting times x_i on other nodes. The estimate of the network average load on each node could deviate far from the ideal average load due to the delays, and this may cause exchanges of tasks back and forth among nodes as shown in Figure 5.33. Those unnecessary transfers cost processing time and prolong the overall completion time. Figure 5.34 shows the responses of the queue sizes in the function of time using the p_{ij} specified by (3.36) on six nodes. In Figure 5.34, the system reaches a balanced state much faster by using the controller based upon the anticipated waiting times z_i , and has a shorter completion time.

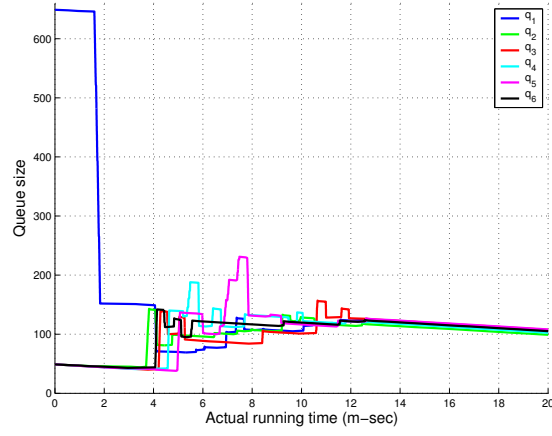


Figure 5.33: Queue sizes on six nodes with initial tasks using controller (3.30) and p_{ij} (3.31) based on x_i .

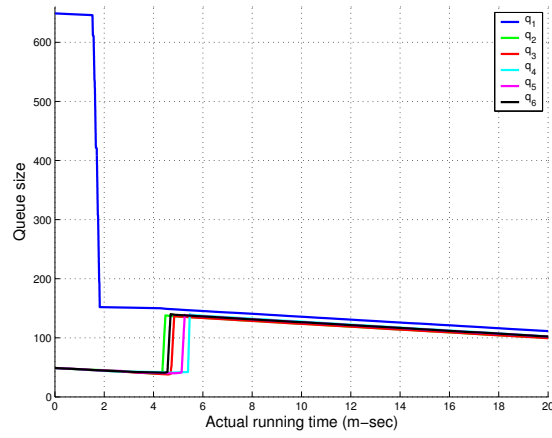


Figure 5.34: Queue sizes on six nodes with initial tasks using controller (3.35) and p_{ij} (3.36) based on z_i .

Multiple Nodes: Task generation. Instead of setting all inputs to zero, the inputs in this set of experiments are $\lambda_1 = 6$, and $\lambda_j = 0$ where $j = 2, 3, 4, 5, 6$. That is, a number of new tasks (λ_1) is generated on node 1 in each processing loop. This creates an increase of tasks for a dynamic load balancing. The initial queue distribution is $q_1(0) = 650$ tasks, $q_j(0) = 50$ tasks on node j where $j = 2, 3, 4, 5, 6$.

Figures 5.35 and 5.36 show the responses of the load balancing with task generation on a network of six nodes, using the controller and p_{ij} based on x_i and z_i , respectively. Note the difference in the load balancing performance between these two schemes. The staircase-like increases of queue sizes in the figures correspond to task generation on node 1 at a rate of $\lambda_1 = 6$. In Figure 5.35, the controller (3.30), based only on the delayed values of queue sizes on other nodes, causes exchanges of tasks back and forth among nodes. The unnecessary transfers need additional balancing operations and cost more processing time. In Figure 5.36, as the controller (3.35) is based on the anticipated queue sizes on other nodes, the system settles much faster and this results in a shorter completion time.

5.2 Parallel Database and Timing Results

The anticipated size and the search requirements for DNA databases have necessitated the development of parallel DNA databases. Methods developed by Birdwell *et al.* [32][33] and Wang *et al.* [211] extend to a parallel decomposition of the DNA database search problem while providing orders of magnitude improvements in performance over

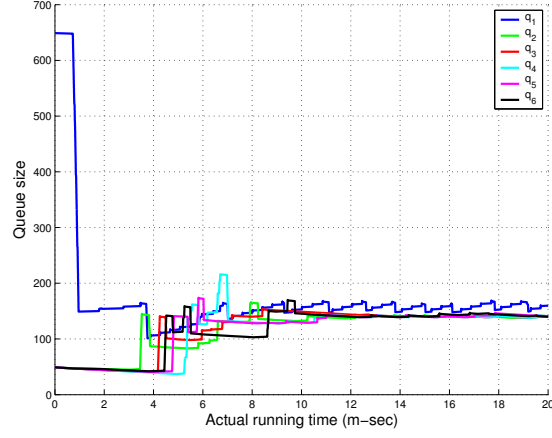


Figure 5.35: Queue sizes on six nodes with task generation $\lambda_1 = 6$ using controller (3.30) and p_{ij} (3.31) based on x_i .

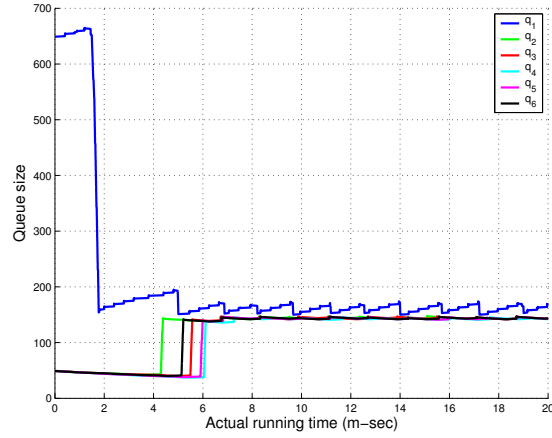
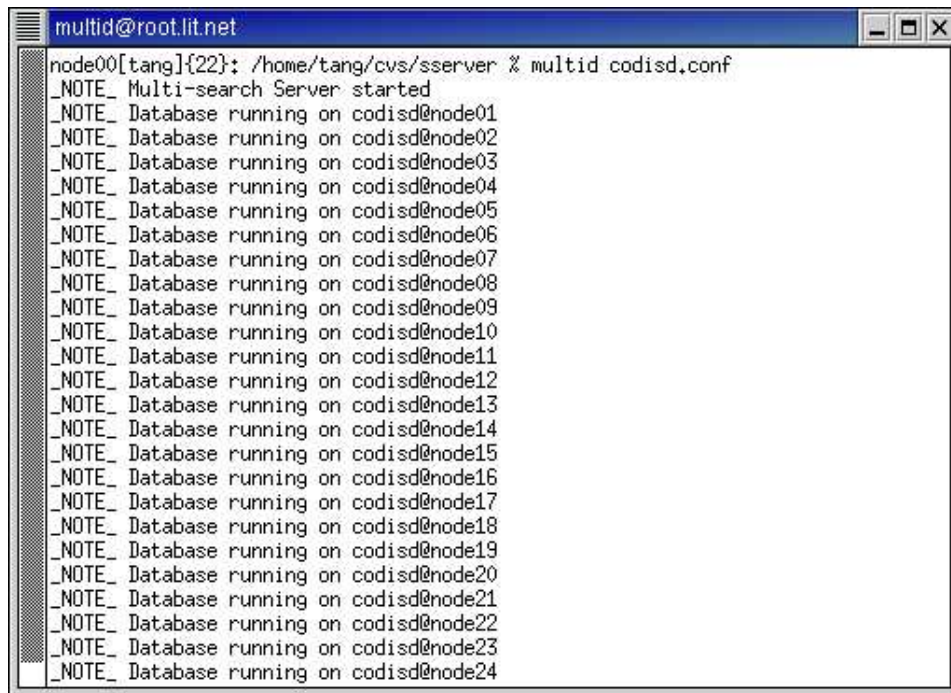


Figure 5.36: Queue sizes on six nodes with task generation $\lambda_1 = 6$ using controller (3.35) and p_{ij} (3.36) based on z_i .

the current software. The designs of the parallel DNA database and two parallel search servers have been addressed in Chapter 4. One search server uses hierarchical sockets and is implemented based on the legacy code. The other is a multi-threaded search server using PVM and object serialization, which aims to improve the performance in transaction times. This section presents experimental results on the parallel DNA database using synthetic data of up to 48M (48,000,000) DNA profiles. The timing results achieved on the parallel DNA database using the hierarchical sockets search server appear in Section 5.2.1. Section 5.2.2 presents the timing results when using the multi-threaded search server, and demonstrates a significant improvement in performance.

5.2.1 Parallel Database with Hierarchical Sockets

The design of a parallel DNA database using hierarchical sockets was illustrated in Section 4.2. Performance results from this implementation are documented in this section. The database is implemented in C++ on a Linux platform using the Simple Sockets Library [40]. This extends the previous work about a single host DNA database to a parallel DNA database on multiple hosts. The software includes (1) *codisd*, which runs on each node of the parallel computer as a DNA search engine, (2) *multid*, which runs as a search server listens for incoming requests and communicates with the search engines, and (3) *cline*, which interacts with the search server as a command line search tool. The *codisd* is a DNA search engine that searches a DNA decision tree to find matches for a target DNA profile, as described in Section 4.1. A screen shot of a running *multid* server with 24 *codisd* search engines is shown in Figure 5.37. The number of



```
multid@root.lit.net
node00[tang]{22}: /home/tang/cvs/sserver % multid codisd.conf
_NOTE_ Multi-search Server started
_NOTE_ Database running on codisd@node01
_NOTE_ Database running on codisd@node02
_NOTE_ Database running on codisd@node03
_NOTE_ Database running on codisd@node04
_NOTE_ Database running on codisd@node05
_NOTE_ Database running on codisd@node06
_NOTE_ Database running on codisd@node07
_NOTE_ Database running on codisd@node08
_NOTE_ Database running on codisd@node09
_NOTE_ Database running on codisd@node10
_NOTE_ Database running on codisd@node11
_NOTE_ Database running on codisd@node12
_NOTE_ Database running on codisd@node13
_NOTE_ Database running on codisd@node14
_NOTE_ Database running on codisd@node15
_NOTE_ Database running on codisd@node16
_NOTE_ Database running on codisd@node17
_NOTE_ Database running on codisd@node18
_NOTE_ Database running on codisd@node19
_NOTE_ Database running on codisd@node20
_NOTE_ Database running on codisd@node21
_NOTE_ Database running on codisd@node22
_NOTE_ Database running on codisd@node23
_NOTE_ Database running on codisd@node24
```

Figure 5.37: A running *multid* server communicates with 24 *codisd* search engines.

search engines to be started and on what nodes is specified by a configuration file (*codisd.conf*). Figure 5.38 shows a screen shot of the command line search tool *cline*, which interacts with the server *multid* over eight search engines. The program *cline* provides a convenient tool to database operations by specifying commands and DNA profiles in command line arguments.

5.2.1.1 Random Searches on a Single Node

To evaluate the performance of the parallel database, an experiment consisting of random searches is executed on a single node. DNA profiles are first selected in a random

```

cline - command line tool to search
node00[tang]{27}: /home/tang/cvs/sserver % cline multid@root "(Search ProfileCD
1)" "(1 (csf1po 10 12) (d13s317 8 14) (d16s539 11 13) (d18s51 18 19) (d21s11 30)
(d3s1358 18 19) (d5s818 12) (d7s820 11) (d8s1179 14 15) (fga 19 21) (th01 7 9,3
) (tpox 8 10) (vwa 14 16) (dqa1pha 1,3 3) (dis80 24 31) (data 11 12))"
2123456 ,309 : NONE : NONE : NONE : NONE : NONE : NONE : NONE
node00[tang]{28}: /home/tang/cvs/sserver %

```

Figure 5.38: Command line search tool *cline* interacts with a search server *multid* with eight search engines.

fashion from a MySQL database. A client program use these profiles to make random search requests that are sent to the search engine. Figure 5.39 shows the search times of 1,000 random searches performed on a single node, node01, holding 2M (2,000,000) DNA profiles. The search time refers to the time to service a search request by searching the decision tree to find an exact match. As shown in Figure 5.39, the mean search time is 441.6 μs , and the standard deviation computed for this sample set is 115.2 μs .

A histogram of these 1,000 search times for the database executed on node01 appears in Figure 5.40. The distribution of search times shows that the bulk of the searches require between 300 μs and 700 μs to complete. The mean value of search times is 441.6 μs , which is at least two orders of magnitude faster than that of current CODIS software (ranging from several *ms* to several *sec*.).

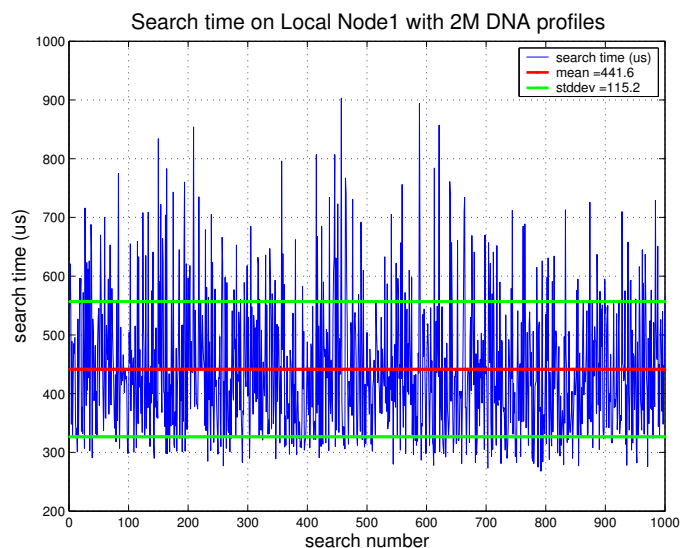


Figure 5.39: Search times of 1,000 random searches on a single node with 2M DNA profiles. Mean search time is 441.6 μs and standard deviation value is 115.2 μs .

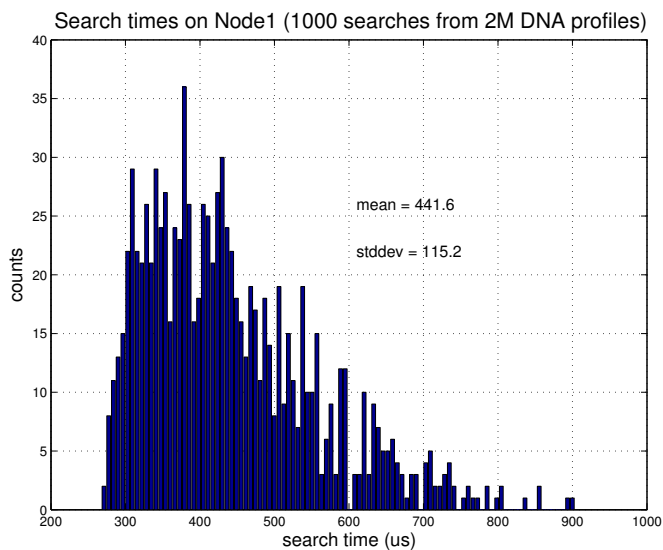


Figure 5.40: Histogram for search times of 1,000 random searches on local node01. Most DNA search requests require between 300 μs and 700 μs to complete.

5.2.1.2 Random Searches on the Parallel Cluster

This experiment shows the performance for 2,000 random searches in the parallel cluster of 24 nodes. Each parallel node has a 2M-profile portion of a 48M-profile DNA database. A client program randomly selects 2,000 profiles from the parallel database (of 48M profiles), and then sends the 2,000 search requests to the *multid* search server, which distributes each request to all 24 search engines for service.

The timing results for these 2,000 random searches are shown in Figure 5.41. In this figure, the top part shows the average search time versus the node index (node01 through node24). Node01 has the largest average search time of 452 μs , and node18 has the smallest average search time at about 407 μs . This is partially due to the hardware difference (single processor versus dual processor) and the different current status of each computer. The average search time across the parallel machine is 431.4 μs . The bottom part of Figure 5.41 shows the average transaction time of 2,000 random searches versus the node index. The transaction time stands for a round-trip time for sending a search request from a client and returning the searched result to the client. From the figure node13 has the smallest average transaction time of 320 ms , while node24 has the largest average transaction time at about 490 ms . That is partially due to random completion times for different requests and sequential communications in the Simple Sockets protocol. The average transaction time across the parallel machine is 410.3 ms . The differences of both search times and transaction times among nodes also necessitate the load balancing for parallel searches.

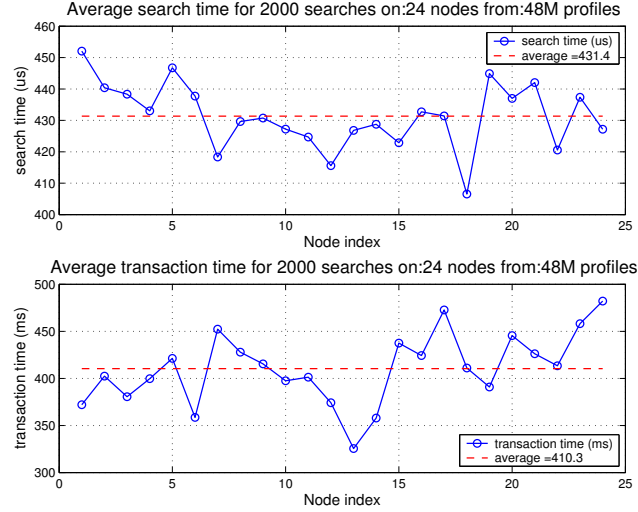


Figure 5.41: Time measured for 2,000 random searches on 24 nodes of 48M profiles. Top: average search times, bottom: average transaction times.

An evaluation is also made to check the performance for those searches with matches found on a local node. The variations of search times and transaction times for those requests with matches found on node01 are shown in Figure 5.42. The average transaction time is much slower than the average search time; this is addressed in the next section.

5.2.2 Multi-threaded Parallel DNA Database

From the previous experiment using the hierarchical sockets search server, the transaction times for this parallel search server are not satisfactory. One method of improvement is optimization of the search engine on each parallel node. A straightforward improvement has been made by integrating the return of profiles into the search process

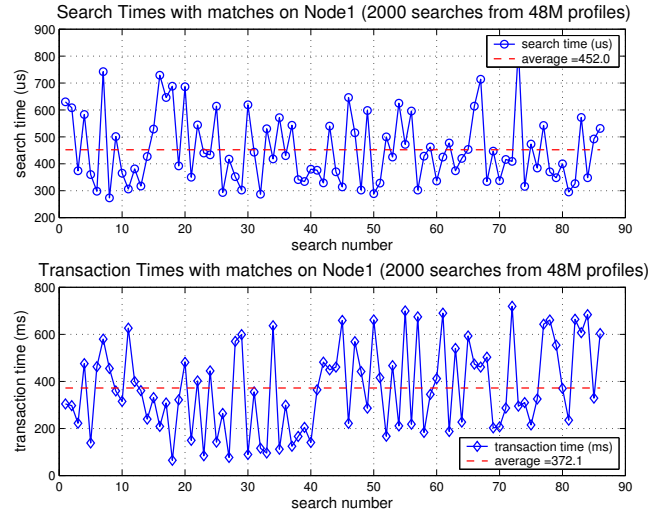
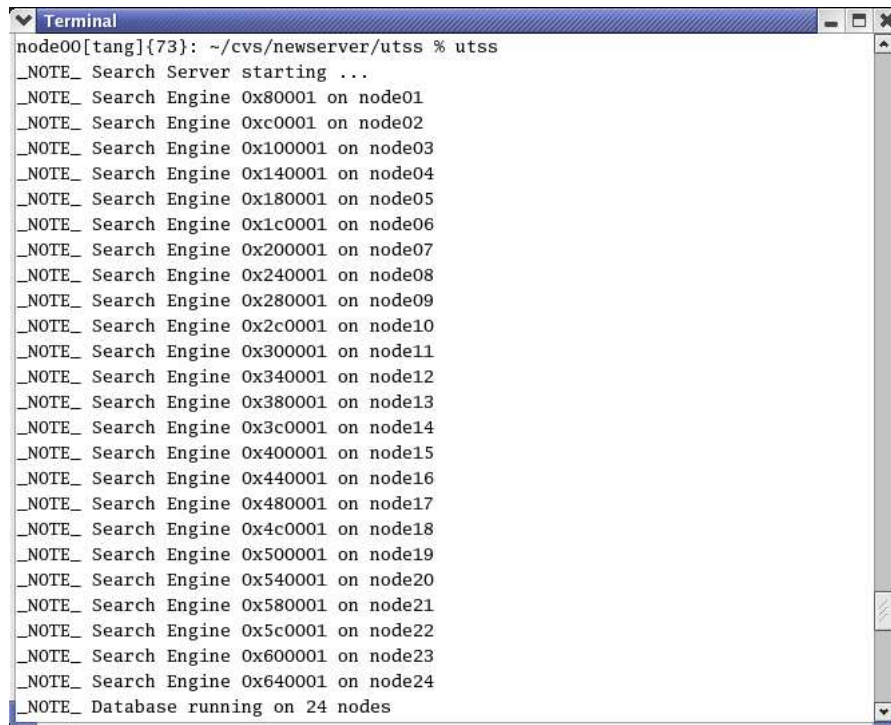


Figure 5.42: Time measured for requests with matches found on node01 (from 2,000 random searches on 24 nodes of 48M profiles). Top: search times with an average of 452.0 μs , bottom: transaction times with an average of 372.1 ms .

when matches are found. Another way is to design a multi-threaded search server to eliminate the loss of performance due to the simple sockets layer communication library. The simple sockets client communicates with the socket server in clear text twice for each packet: the first is for the packet size (to avoid a buffer overflow), and the second is for the real packet.

A multi-threaded search server using PVM communications and serialized buffers has been designed and implemented, as illustrated in Section 4.2. The multi-threaded search server *utss* starts a listening thread (**LThread**), which listens for connection requests and manages a pool of service threads (**SThreads**) that service these connections. Requests are distributed and results are gathered using a communication thread (**CommThread**),

A terminal window titled "Terminal" with a blue header bar. The window shows the execution of the `utss` command. The output consists of a series of messages: "_NOTE_ Search Server starting ...", followed by 24 lines of "_NOTE_ Search Engine [hex address] on node[01-24]", and finally "_NOTE_ Database running on 24 nodes". The hex addresses range from 0x80001 to 0x640001 in increments of 0x10000. The terminal window has standard OS window controls (minimize, maximize, close) in the top right corner.

```
node00[tang]{73}: ~/cvs/newserver/utss % utss
_NOTE_ Search Server starting ...
_NOTE_ Search Engine 0x80001 on node01
_NOTE_ Search Engine 0xc0001 on node02
_NOTE_ Search Engine 0x100001 on node03
_NOTE_ Search Engine 0x140001 on node04
_NOTE_ Search Engine 0x180001 on node05
_NOTE_ Search Engine 0x1c0001 on node06
_NOTE_ Search Engine 0x200001 on node07
_NOTE_ Search Engine 0x240001 on node08
_NOTE_ Search Engine 0x280001 on node09
_NOTE_ Search Engine 0x2c0001 on node10
_NOTE_ Search Engine 0x300001 on node11
_NOTE_ Search Engine 0x340001 on node12
_NOTE_ Search Engine 0x380001 on node13
_NOTE_ Search Engine 0x3c0001 on node14
_NOTE_ Search Engine 0x400001 on node15
_NOTE_ Search Engine 0x440001 on node16
_NOTE_ Search Engine 0x480001 on node17
_NOTE_ Search Engine 0x4c0001 on node18
_NOTE_ Search Engine 0x500001 on node19
_NOTE_ Search Engine 0x540001 on node20
_NOTE_ Search Engine 0x580001 on node21
_NOTE_ Search Engine 0x5c0001 on node22
_NOTE_ Search Engine 0x600001 on node23
_NOTE_ Search Engine 0x640001 on node24
_NOTE_ Database running on 24 nodes
```

Figure 5.43: A multi-threaded search server *utss* communicates with 24 search engines via PVM and object serialization.

which communicates with the search engines *codisdnews* via PVM and serialization. A screen shot showing the start-up of the multi-threaded search server *utss* appears in Figure 5.43.

After building a decision tree for the profiles loaded from a MySQL database (or a huge file), the search engine *codisdnew* on each parallel node enrolls into PVM to wait for search requests distributed by *utss*, unpacks incoming serialized packages to service the requests, and then returns the serialized results to the server via PVM. The search server *utss* gathers results and returns these to the appropriate client. Besides a set of

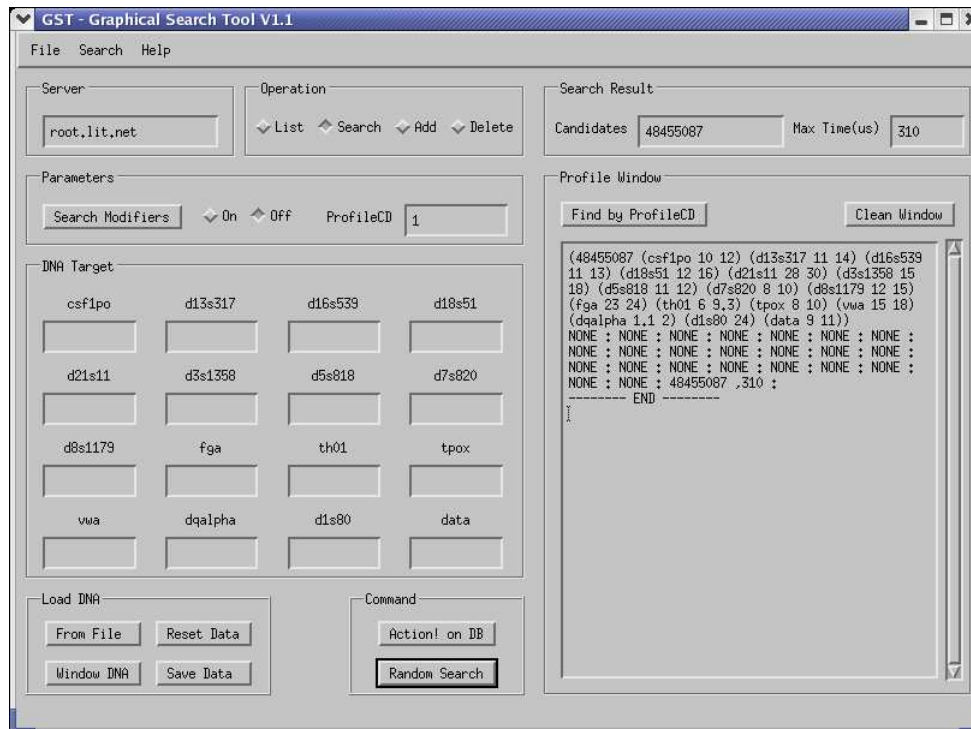


Figure 5.44: Graphical search tool *gst* communicates with the search server *utss*.

command line tools for database operations, a graphical search tool *gst* has also been implemented to explore the parallel DNA databases. A screen shot of the user interface for the graphical search tool appears in Figure 5.44.

5.2.2.1 Random Searches on a Single Node

An experiment is first run using 1,000 random searches on the multi-threaded search server *utss* with a single search engine. This checks database performance with minimal effect of multiple communications. 1,000 DNA profiles are randomly selected from a local MySQL database of 2M DNA profiles, and are used to generate random search

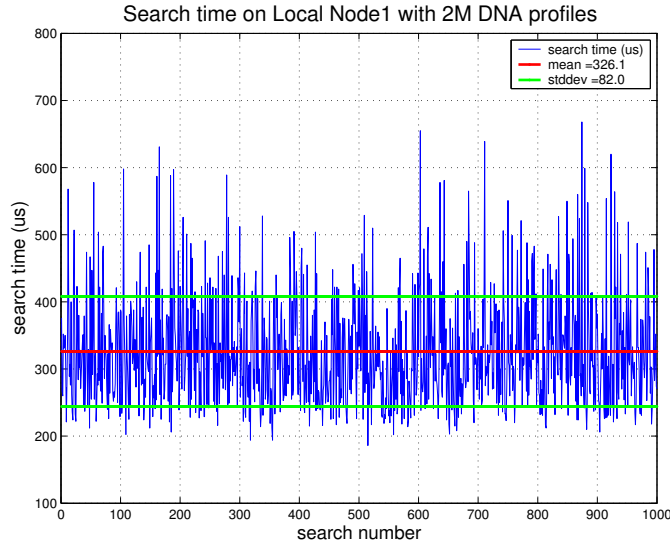


Figure 5.45: Search times of 1,000 random searches on a single node using the multi-threaded search server *utss*. Mean search time is 326.1 μs and standard deviation value is 82.0 μs .

requests to be sent to the search engine.

Figure 5.45 shows the search times of 1,000 random searches on a single node (node01) with 2M DNA profiles. As shown in the figure, the mean search time is 326.1 μs , and the standard deviation computed for the sample set is 82.0 μs . The slight improvement on the search speed is primarily due to a updated Linux kernel and an improved implementation of the search engine *codisdnew*.

The transaction times for these 1,000 random searches on the multi-threaded search server are shown in Figure 5.46. The mean transaction time is 2.851 ms , and the sample set's standard deviation value is 0.169 ms . Comparing with the previous results on transaction times shown in the bottom part of Figure 5.42, the performance boost

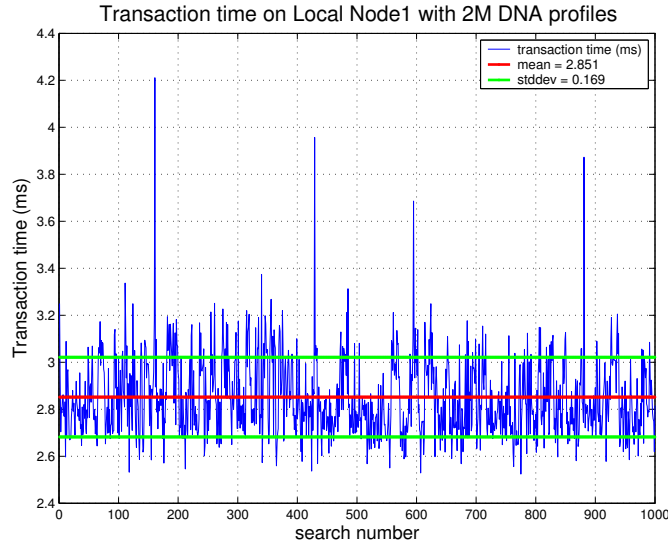


Figure 5.46: Transaction times of 1,000 random searches on a single node using the multi-threaded search server *utss*. Mean transaction time is 2.851 *ms* and standard deviation value is 0.169 *ms*.

in transaction time is over two orders of magnitude.

5.2.2.2 Random Searches on the Parallel Cluster

An experiment using 2,000 random searches and the parallel cluster of 24 nodes is performed. A client program randomly selects 2,000 profiles from the parallel database (of 48M profiles), and then sends the generated 2,000 requests to the search server *utss*, which distributes the requests to the 24 search engines (*codisdnews*).

Figure 5.47 shows the measured search times and transaction times for 2,000 random searches. In this figure, the top part shows the average search time versus the node index (node01 through node24). Node14 has the largest average search time at about 340 μs ,

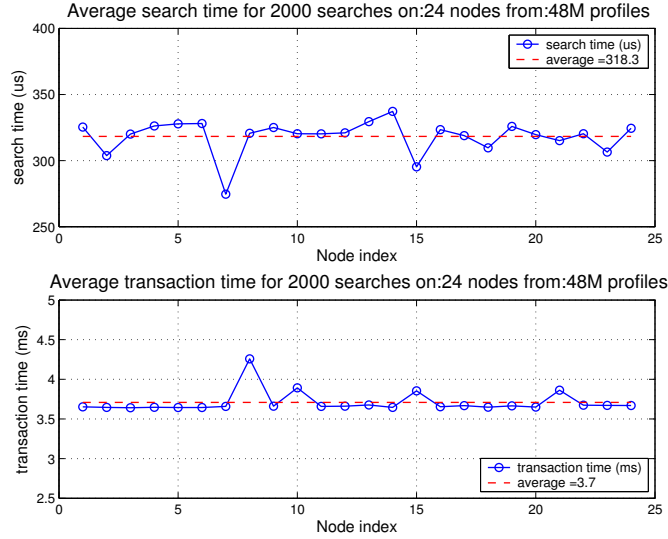


Figure 5.47: Time measured for 2,000 random searches on 24 nodes using the multi-threaded search server. Top: average search time, bottom: average transaction time.

while node07 has the smallest average search time at about $270 \mu s$. This variation is a function of random effects within each node and variations in the database's index structure. The average search time across the parallel machine is $318.3 \mu s$, which is slightly better than the average search time of $431.4 \mu s$ in Figure 5.41. The bottom part shows the average transaction time for 2,000 random searches as a function of the node index. The average transaction time across the parallel machine is $3.7 ms$. Since the search server *utss* must gather results from 24 search engines, a transaction time at $3.7 ms$ for a search request is quite fast. Compared with the the average transaction time of $410.3 ms$ in Figure 5.41, this is about 110-fold smaller.

The local performance of a parallel node can also be evaluated in this experiment.

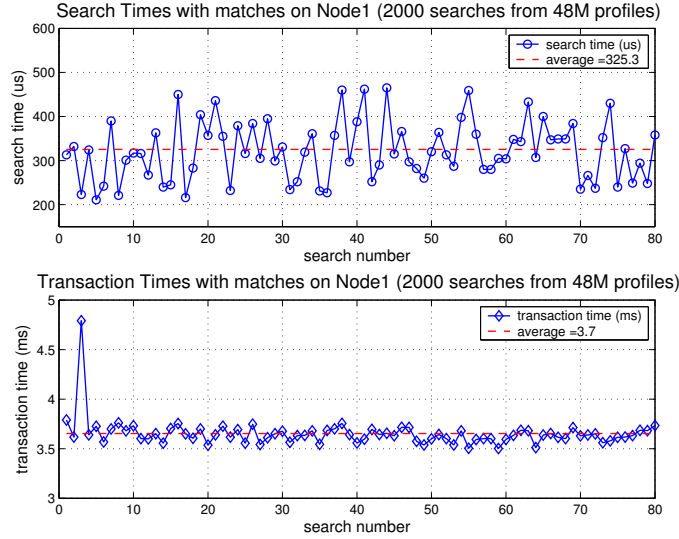


Figure 5.48: Time measured for requests with matches found on node01 using the multi-threaded search server (from 2,000 random searches on 24 nodes). Top: search times with an average of $325.3 \mu s$, bottom: transaction times with an average of $3.7 ms$.

The variations of search times and transaction times for those requests with matches found on node01 are shown in Figure 5.48. While the average search time of $325.3 \mu s$ is comparable to the average of $452.0 \mu s$ in Figure 5.42, the average transaction time of $3.7 ms$ is 100-fold smaller than that average of $372.1 ms$. A detailed comparison of search and transaction times for the two search servers is listed on Table 5.1. From the table it is evident that the multi-threaded search server *utss* completes a transaction two orders of magnitude faster than the hierarchical sockets search server *multid*, while the new search engine *codisdnew* provides a slightly faster search speed than the original search engine *codisd*.

Table 5.1: Average search and transaction times on different search servers

Search server	Search engines	Average search time	Average transaction time
<i>multid</i>	24 nodes	431.4 μs	410.3 ms
	node01	452.0 μs	372.1 ms
<i>utss</i>	24 nodes	318.3 μs	3.7 ms
	node01	325.3 μs	3.7 ms

5.2.2.3 Different Client Nodes

The transaction times on different client nodes communicating with the same parallel search server *utss* (on Root) for a 48M profiles database running on 24 nodes are investigated in this experiment. The experiment measures the relative effects of different network paths between a client and the parallel search server (*utss*) in these cases. Figure 5.49 shows the transaction times for clients on Root (the same host as the search server), Node09 (one of the 24 parallel nodes, and is in the same subnet as Root), and Holly (one node in a different network connected with Root via Fast Ethernet). 1,000 random searches are performed, and the transaction times are measured for each of the three cases. The transaction times are marked as short horizontal bars, while the average transaction time for each case is computed and marked as a circle in Figure 5.49.

The tests show the network location of the client node, for these three cases, does not have much effect on the average transaction time. However, the range of transaction times varies by roughly a factor of 6. This is to be expected, since the presence of other network traffic on a non-dedicated network (client on Holly) can greatly impact the delay experienced by a network packet. This, however, is not likely to occur very often

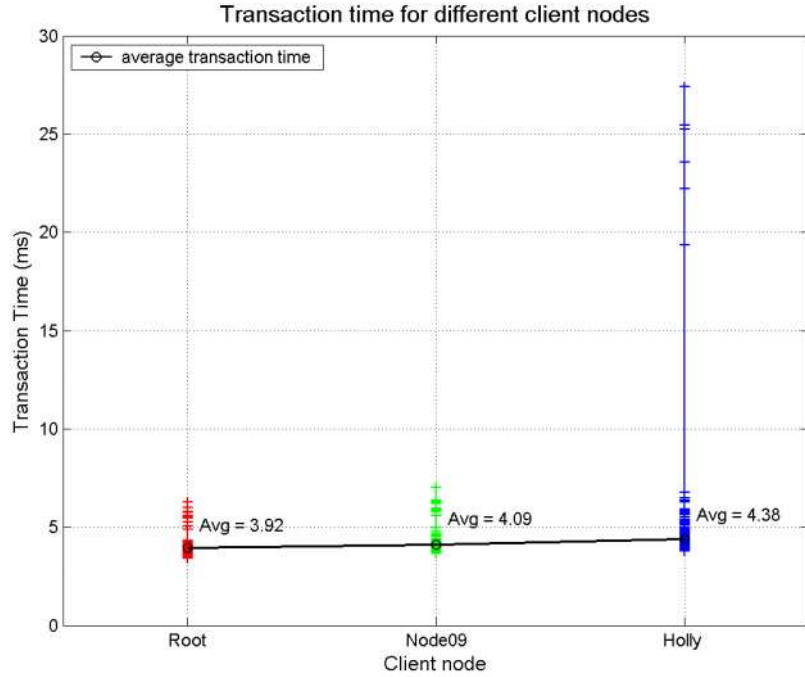


Figure 5.49: Transaction times on different client nodes communicating with the same server *utss*. The *utss* search server runs on Root with a 24-node DNA database of 48M profiles. Average transaction time is 3.92 *ms* on Root (the same host), 4.09 *ms* on Node09 (in the same subnet), and 4.38 *ms* on Holly (in a different network).

unless the network is heavily loaded. From the figure, the average transaction time is 3.92 *ms* for a client on the same host with the search server, 4.09 *ms* for a client in the same subnet but on a different host, and 4.38 *ms* for a client in a different network. There are 6 isolated points (of the 1,000 tests) for the client in a different network (Holly), and this is probably due to other network traffic on the shared portion of the network. The comparison of average transaction times on different client nodes is listed on Table 5.2.

Table 5.2: Transaction times on the same *utss* for different client nodes

Client node	Average transaction time	Relationship with server
<i>Root</i>	3.92 <i>ms</i>	same host
<i>Node09</i>	4.09 <i>ms</i>	same subnet
<i>Holly</i>	4.38 <i>ms</i>	different network

5.3 Parallel Search with Load Balancing

In this section, results for parallel searches with a load balancing method running on each node are presented. The database is implemented as a set of queues with associated search engine threads, typically assigned one per node of the parallel machine. The search engine threads access tree-structured indices to locate database records that match search requests. This hierarchical structure has been illustrated in Figure 1.3.

Nodes on the parallel computer are divided into many groups to form distributed storages of the parallel DNA database. Each group is assigned to a portion of DNA profiles. Load balancing actions are performed between search engines running on the nodes in the same group. As a group of load balancing nodes uses the identical DNA database, search requests can be transferred between the nodes in a group and the workload can be balanced. The parallel DNA database with load balancing uses a multi-threaded search server discussed in Section 4.2.2. The first set of experiments is conducted to evaluate parallel searches with integrated load balancing on a network of three nodes of initial tasks only. The second set of experiments shows with task generation. Results on a larger network of six nodes are presented in the third set of

experiments. These experiments demonstrate that parallel searches with load balancing integrated with the parallel DNA database work quite well.

5.3.1 Initial Tasks Only

In this experiment, the performance of load balancing for a 3-node group with an initial unbalanced condition and no new arrivals is evaluated. Each of the nodes (labeled **node1**, **node2**, and **node3**) runs a search engine with an identical DNA database. Note that the nodes labeled **node1**, **node2**, and **node3** correspond to the hosts named node22, node23, and node24, respectively, on the parallel machine. The initial conditions used for the task queues (q_1, q_2, q_3) are $(0, 0, 200)$.

On each node, a load balancing thread broadcasts its queue size (when the queue's size changes) to the other nodes in the network, and also receives information on their queues' sizes. After loading the initial 200 search requests (tasks), **node3** calculates its estimate of network average load as $q_{3_avg} = (200+0+0)/3 \approx 67$, and its workload relative to the network average as $q_{3_diff} = 200 - 67 = 133$. Next, **node3** calculates the portions of search requests (tasks) to be transferred according to (3.36), and broadcasts the number of search requests to be transferred to each of the other nodes, which includes the (anticipated) number of tasks being sent to **node1** and **node2** (66 each). Figure 5.50 shows the local workloads, average estimates and tracking differences computed by **node3**. **Node1** receives the values broadcast from **node3**, and updates its estimate of average (anticipated) workload to $q_{1_avg} = ((200 - 132) + 66 + 66)/3 \approx 67$. **Node1** then calculates its workload relative to the network average as $q_{1_diff} = q_1 - q_{1_avg} = -67$,

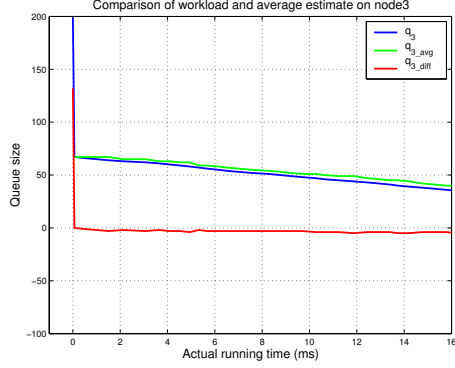


Figure 5.50: Workloads, average estimates and tracking differences on **node3** with initial tasks.

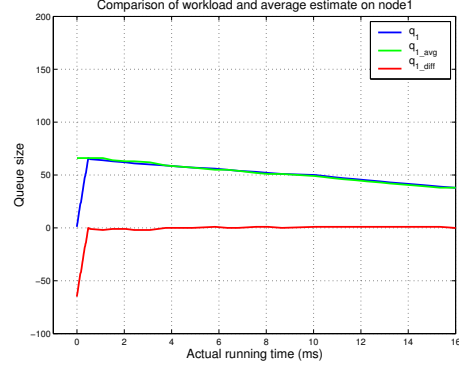


Figure 5.51: Workloads, average estimates and tracking differences on **node1** with initial tasks.

and so $\text{sat}(q_1 - q_{1_avg}) = 0$. In this manner, **node1** has a more up to date estimate of the (anticipated) workload at **node2**, and unnecessary transfers are avoided. Upon receiving the 66 requests transferred from **node3**, **node1** inserts the search requests to its queue and continues processing them. The local workloads and average estimates on **node1** are shown in Figure 5.51. The data collected for **node2** are quite similar to those for **node1** in this test. The results shown in Figures 5.50 and 5.51 demonstrate that the load balancing method, which uses a closed loop controller based on anticipated work loads, works quite well in this situation.

A comparison of the average estimates on each of the three nodes as a function of time is shown in Figure 5.52. The average estimates on the three nodes are similar. The deviations are due to variations in the times for different search requests and random delays in network traffic. Figure 5.53 compares the tracking differences between local workloads and average estimates on the three nodes. The local workloads track the

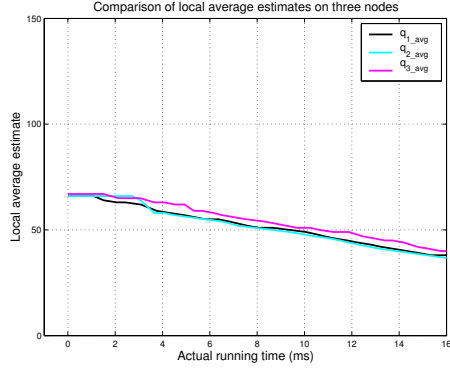


Figure 5.52: Average estimates on three nodes with initial tasks. q_{1_avg} , q_{2_avg} , q_{3_avg} for **node1**, **node2** and **node3** respectively.

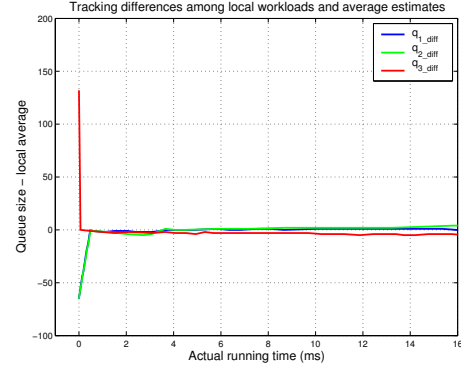


Figure 5.53: Tracking differences on three nodes with initial tasks. q_{1_diff} , q_{2_diff} , q_{3_diff} for **node1**, **node2** and **node3** respectively.

average estimates very well, and the system settles quickly. Note that the database searches are running in parallel and asynchronously on each search engine node. Only the changes of queue states on each node are logged. Task processing (insertions and removals of tasks) on **node1**, as well as **node2**, starts after receiving the tasks transferred from **node3**.

5.3.2 Task Generation

The previous experiment balances search tasks in initial queues of parallel nodes. In the following set of experiments, load balancing performance is assessed for tasks generated from client programs, which are distributed to parallel search engines (with a load balancer running on each) as they are received.

5.3.2.1 Profiles Loaded from a File

For this case, search requests are generated using the DNA profiles loaded from a data file. The search server has been configured to randomly pick a parallel node and then send the search requests using PVM and serialization. A load balancer on each search engine periodically evaluates its workload relative to the network average, and sends excess workload to the other nodes according to the closed loop controller based on anticipated waiting time z_i , as discussed in Chapter 3. In this experiment, 200 search requests generated with profiles from a file are distributed to **node1** by the search server. A search engine thread on **node1** starts processing after receiving the search requests, while a load balancing thread on **node1** communicates with other nodes in the group and balances the workload. Figure 5.54 shows the workload, average estimate, and tracking difference achieved on **node1**. An obvious difference from the case using initial tasks shown in Figure 5.50 is that the search engine has started processing requests before the load balancing thread starts the first balancing operation. The workloads and average estimates on **node2** are shown in Figure 5.55, and the results from data on **node3** are quite similar to those on **node2** in this experiment.

Figure 5.56 shows a comparison of average estimates on three nodes. The average estimates on the three nodes are very close to each other. Again, the slight deviations result partially from differences in random completion times of search requests. Figure 5.57 shows a comparison of tracking differences on three nodes. The local workloads track the average estimates quite well and the system settles quickly.

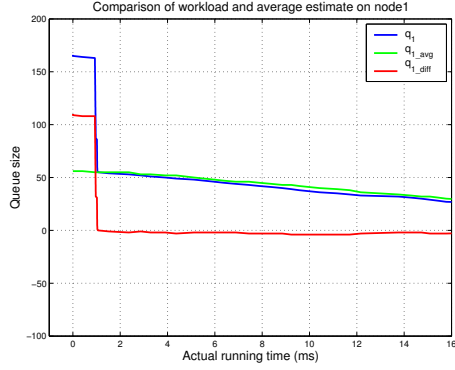


Figure 5.54: Workloads, average estimates and tracking differences on **node1** with tasks generated from a file.

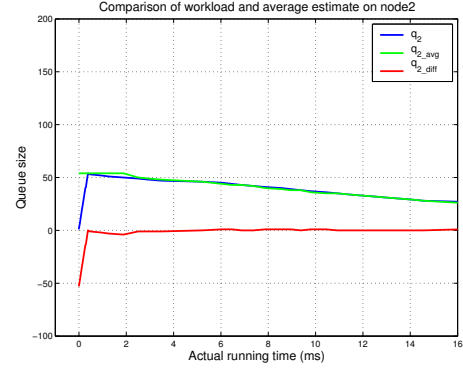


Figure 5.55: Workloads, average estimates and tracking differences on **node2** with tasks generated from a file.

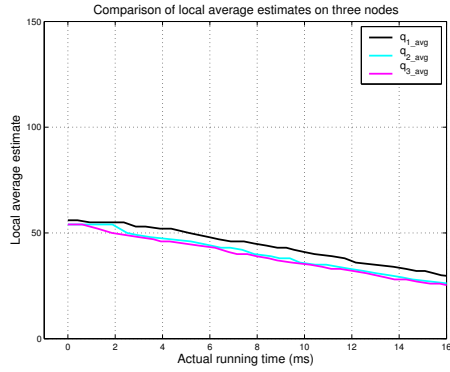


Figure 5.56: Average estimates on three nodes with tasks generated from a file. $q1_{avg}$, $q2_{avg}$, $q3_{avg}$ for **node1**, **node2** and **node3** respectively.

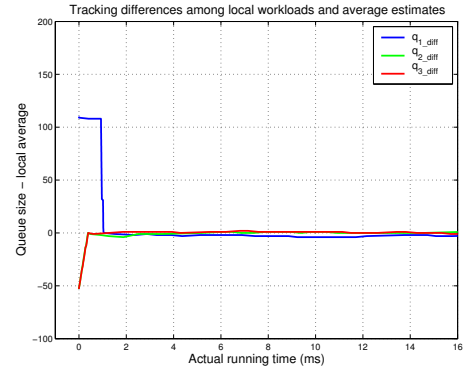


Figure 5.57: Tracking differences on three nodes with tasks generated from a file. $q1_{diff}$, $q2_{diff}$, $q3_{diff}$ for **node1**, **node2** and **node3** respectively.

5.3.2.2 Random Profiles from a Database

This experiment simulates a series of random search requests from a client. The search requests are then distributed to search engines by the search server. Consider the tasks to be collected together into blocks of 100 each by the search server. To illustrate the queuing up of tasks on the search engine nodes, these blocks of 100 tasks each are then sent randomly to the nodes in the network. In this experiment, which also uses a group of three nodes, a total of 1,000 search requests (tasks) are generated by a client program by randomly selecting DNA profiles to be used as targets for a search. Every 5 *ms* the search server randomly selects a search engine node and sends a block of 100 tasks. This is to illustrate the queuing up of tasks on the nodes. Although 100 tasks are sent every 5 *ms*, this rate exceeds the rate at which each queue can receive tasks and insert them into a local queue. Thus, the tasks are received over a period of about 140 *ms*. While the search engine thread on each node processes requests in its local queue, each node exchanges queue information with the other nodes and redistributes the tasks depending on the relative workload by running the load balancing thread.

Figure 5.58 shows the workload, average estimate, and tracking difference on **node1**. The large upward transitions are caused by task arrivals (a block of 100 tasks) from the client, while small upward transitions are caused by received search requests and queue insertions. The downward transitions are caused by removal of tasks from a queue for service or to be transferred (in blocks) to other nodes. The workloads and average estimates on **node2** and **node3** are shown in Figure 5.59 and Figure 5.60, respectively.

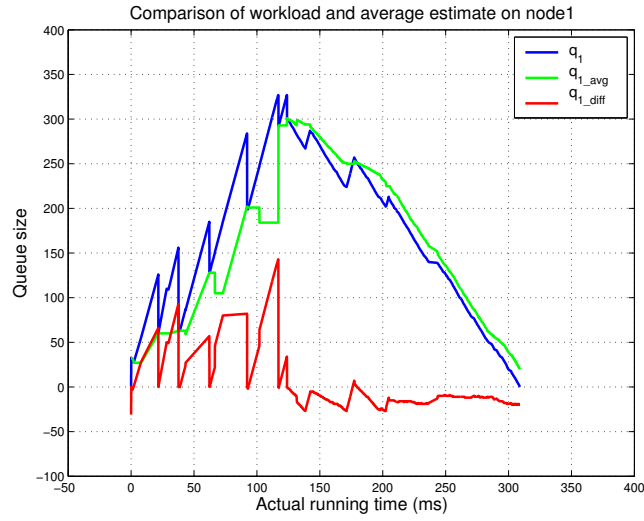


Figure 5.58: Workloads, average estimates and tracking differences on **node1** with randomly generated tasks. Blue curve: workload q_1 , green: average estimate q_{1_avg} , and red: tracking difference q_{1_diff} .

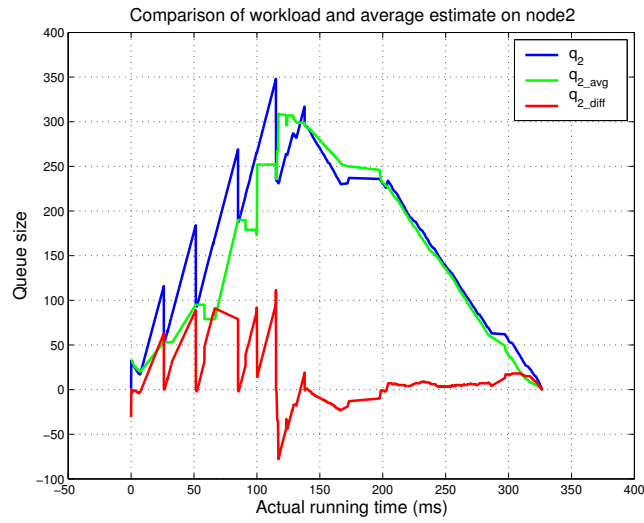


Figure 5.59: Workloads, average estimates and tracking differences on **node2** with randomly generated tasks. Blue curve: workload q_2 , green: average estimate q_{2_avg} , and red: tracking difference q_{2_diff} .

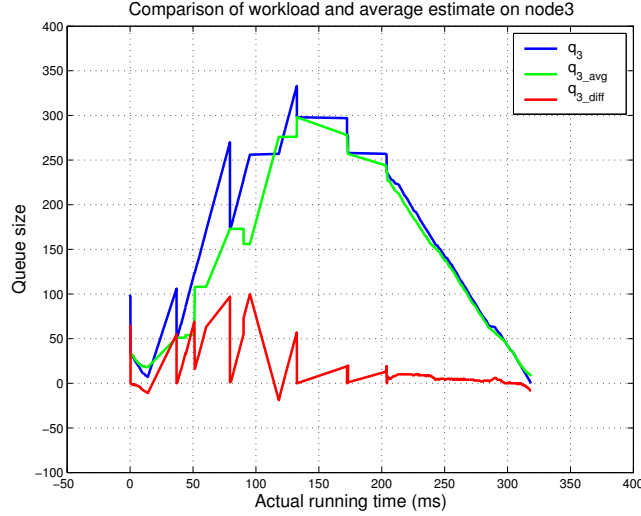


Figure 5.60: Workloads, average estimates and tracking differences on **node3** with randomly generated tasks. Blue curve: workload q_3 , green: average estimate q_{3_avg} , and red: tracking difference q_{3_diff} .

Figure 5.61 shows a comparison of average estimates measured on these three nodes with randomly generated tasks. When a new block of search requests arrives, the receiving node updates its average, which creates a transient that is visible in the figure. Then the load balancing method evens out the tasks and brings the average estimates together. For $t > 200\text{ ms}$, no new search requests arrive. The system settles to a balanced state, and the average estimates on three nodes closely follow each other.

Figure 5.62 shows the responses for 1,000 tasks arriving in 10 blocks on three nodes when the load balancing thread is disabled. The search server randomly selects a search engine node for each block transfer. The queues on the nodes are not balanced. This leads to different completion times and a larger completion time for the group.

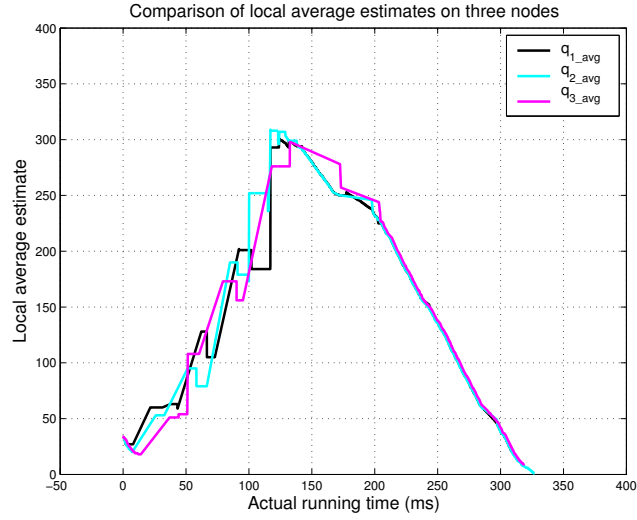


Figure 5.61: Comparison of average estimates on three nodes with randomly generated tasks. The black, cyan and pink curves stand for q_{1_avg} , q_{2_avg} , q_{3_avg} on **node1**, **node2**, and **node3** respectively.

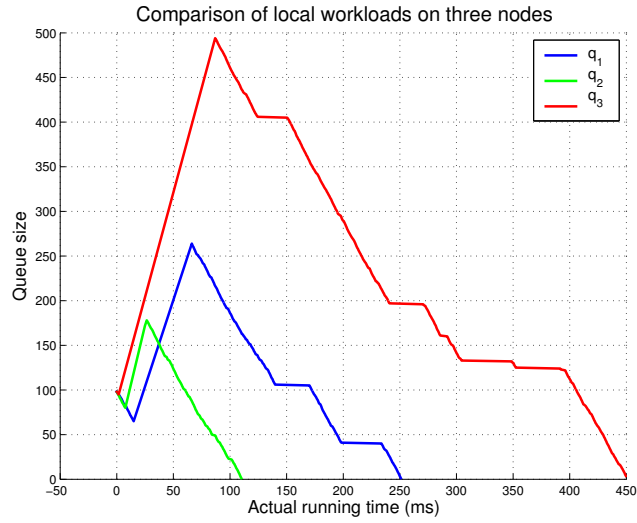


Figure 5.62: Responses of queue sizes on three nodes without load balancing. The search server randomly selects a search engine node for each of 10 block transfers (with 100 tasks each) when the load balancing thread is disabled.

5.3.3 Results on Multiple Nodes

The experiments in the previous subsections were conducted on three nodes. This subsection presents the experimental results for parallel search with load balancing on a larger network of six nodes ($n = 6$). Note that the nodes labeled **node1**, **node2**, **node3**, **node4**, **node5**, and **node6** correspond to the hosts named node22, node23, node24, node19, node20, and node21, respectively, on the parallel machine.

5.3.3.1 Initial Tasks

Here, the first set of experiments is designed to test the performance of the parallel database with load balancing on a network of six nodes. The initial queue distribution is $q_1(0) = 600$ tasks, $q_j(0) = 0$ where $j = 2, 3, 4, 5, 6$. Figure 5.63 shows the local workloads, average estimates and tracking differences on **node1**. **Node2** inserts the incoming 100 requests transferred from **node1** to its queue, updates its estimate of average workload, and continues processing those search requests. The local workloads and average estimates on **node2** are shown in Figure 5.64.

A comparison of average estimates on the six nodes is shown in Figure 5.65. The average estimates on the six nodes are very close to each other after a load balancing action. Figure 5.66 compares tracking errors on the six nodes. From the figure, the local workloads track the average estimates very well, and the system settles quickly. Again, only changes in queue size are logged on each node, and task processing on **node2** and other nodes starts after receiving the transfer from **node1**.

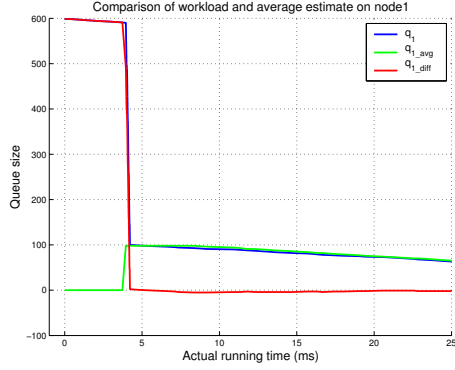


Figure 5.63: Workloads, average estimates and tracking differences on **node1** with initial tasks on six nodes.

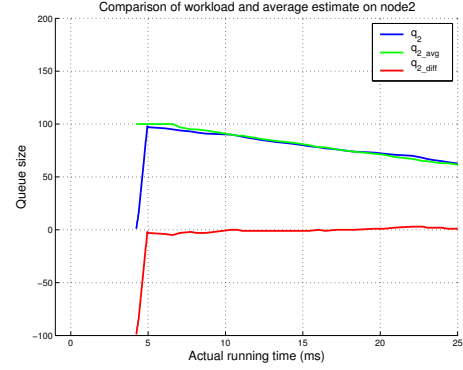


Figure 5.64: Workloads, average estimates and tracking differences on **node2** with initial tasks on six nodes.

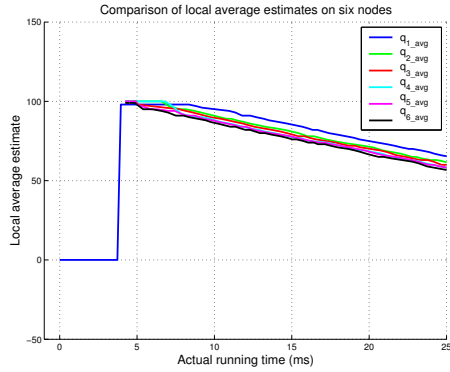


Figure 5.65: Average estimates on six nodes with initial tasks. q_{1_avg} , q_{2_avg} , q_{3_avg} , q_{4_avg} , q_{5_avg} , q_{6_avg} for each of the six nodes respectively.

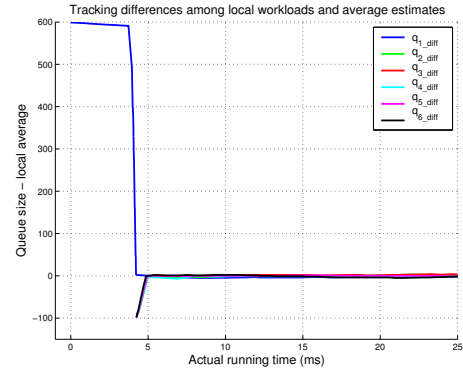


Figure 5.66: Tracking differences on six nodes with initial tasks. q_{1_diff} , q_{2_diff} , q_{3_diff} , q_{4_diff} , q_{5_diff} , q_{6_diff} for each of the six nodes respectively.

5.3.3.2 Randomly Generated Tasks

This set of experiments, which also uses a group of six nodes, simulates randomly generated search requests from a client and being distributed by the search server. The client generates a series of random requests by selecting profiles from a MySQL database, and sends the requests to a search engine through PVM using object serialization.

Tasks to a Randomly Selected Node In this experiment, a total of 2,000 tasks are randomly generated by a client program in 20 blocks of 100 tasks each. A block of 100 requests is randomly distributed by the search server every 5 *ms* to a search engine node for service. The load balancing threads on six nodes communicate with each other and even out the workloads. Responses on a representative node **node3** are shown here to demonstrate the load balancing strategy. Figure 5.67 shows workloads, average estimates and tracking differences computed on **node3** in a network of six nodes with randomly generated tasks. **Node3** receives requests transferred from other nodes, shown as small upward transitions in the figure, and continues processing them. Three incoming blocks of requests (100 each) from the client are distributed to **node3**, shown as three large upward transitions in the figure, and **node3** then balances the requests among the other five nodes according to the amount of they are below **node3**'s estimate of network average load. Figure 5.68 shows the local workloads, average estimates and tracking differences on another representative node **node5**.

Figure 5.69 shows a comparison of average load estimates measured on six nodes. When a new block of search requests arrives, the receiving node updates its average,

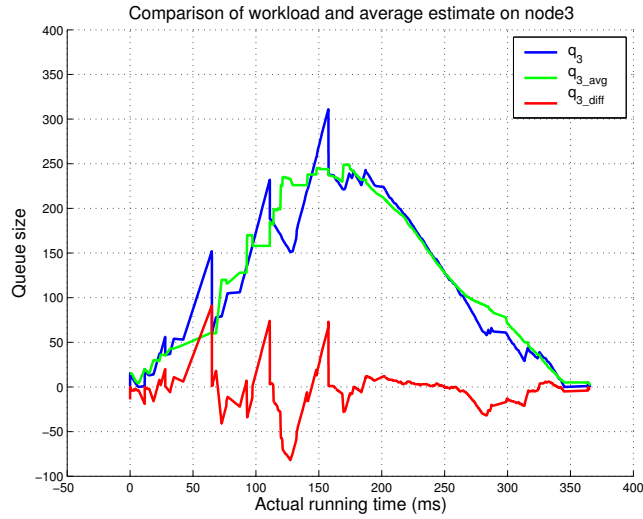


Figure 5.67: Workloads, average estimates and tracking differences on **node3** with randomly generated tasks. Blue curve: workload q_3 , green: average estimate q_{3_avg} , and red: tracking difference q_{3_diff} .

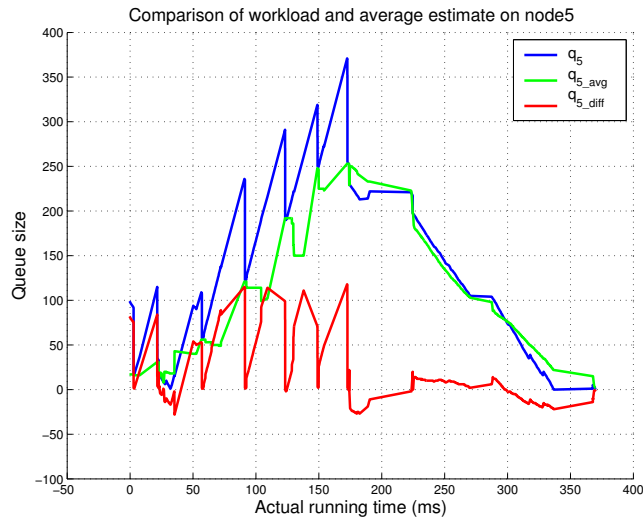


Figure 5.68: Workloads, average estimates and tracking differences on **node5** with randomly generated tasks. Blue curve: workload q_5 , green: average estimate q_{5_avg} , and red: tracking difference q_{5_diff} .

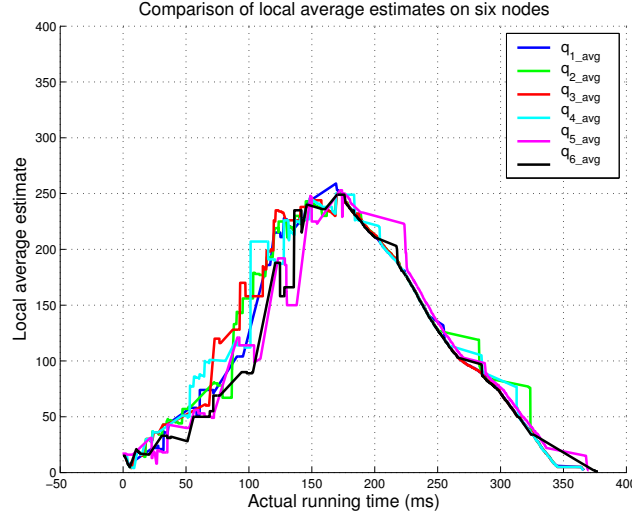


Figure 5.69: Comparison of average estimates on six nodes with randomly generated tasks.

which creates a step transient as shown in Figure 5.69. The load balancing method then evens out the tasks and brings the average estimate close to that on other nodes. For $t > 200 \text{ ms}$, no new search requests arrive. The system settles to a balanced state, and the average estimates on six nodes closely follow each other.

Notice that the previous experiment in subsection 5.3.2.2 used a group of three nodes for the incoming 1,000 randomly generated tasks (10 blocks of 100 tasks each), and this experiment uses a group of six nodes to balance and service the incoming 2,000 randomly generated tasks (20 blocks of 100 tasks each). The overall waiting time to complete all 2,000 tasks (the maximum completion time in a group) is 377.4 ms in this experiment (see Figure 5.69), while it took 327.1 ms to complete all 1,000 tasks on three nodes in the previous experiment (see Figure 5.61). For this case, the speed-up is

73% when the number of nodes is doubled.

Tasks to a Single Node This experiment simulates the extreme case of load balancing on a parallel database. In this experiment, which also uses a group of six nodes, a total of 1,000 tasks are randomly generated by a client program in 20 blocks of 50 tasks each. Every 5 *ms* a block of 100 search requests is sent to a single node, **node1** in this case, for service. This is to illustrate the worst case for load balancing on search engines of a parallel database. The load balancing threads on six nodes communicate with each other and even out the workloads. Figure 5.70 shows the local workloads, average estimates and tracking differences on **node1**. From the figure, **node1** receives the incoming search requests in blocks with 50 each (corresponding to the large upward transitions), and balances the tasks to the other five nodes (corresponding to the large downward transitions). As the 20 blocks of 50 tasks each are sent every 5 *ms*, this rate exceeds the rate at which **node1** can receive tasks and insert them to process, and the tasks are received over a period of about 200 *ms*. Responses on **node5**, a representative node of the other five nodes, are shown in Figure 5.71. When it receives the tasks transferred from **node1**, **node5** (one of the five recipient nodes) continues processing those tasks and returning results. In Figure 5.71, the queue size at **node5** decreases toward zero as it completes tasks prior to the arrival of the next block of requests balanced out from **node1**. All five recipient nodes follow the same pattern.

A comparison of average estimates computed at six nodes for this case is shown in Figure 5.72. As the tasks are distributed to **node1** each time, the average estimate at

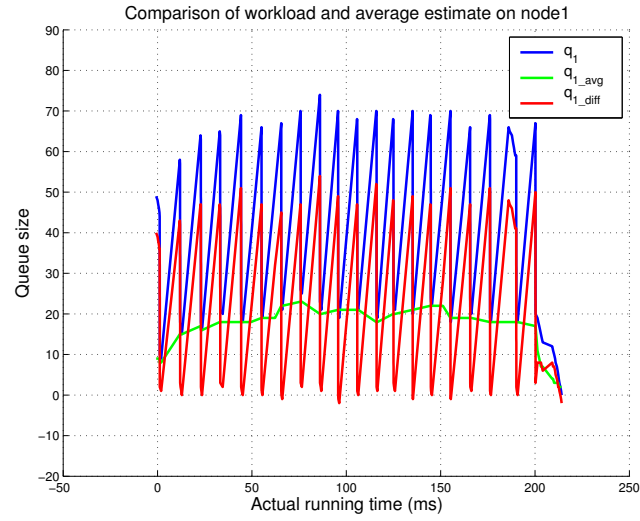


Figure 5.70: Workloads, average estimates and tracking differences on **node1** with tasks initially distributed to **node1**. Blue curve: workload q_1 , green: average estimate q_{1_avg} , and red: tracking difference q_{1_diff} .

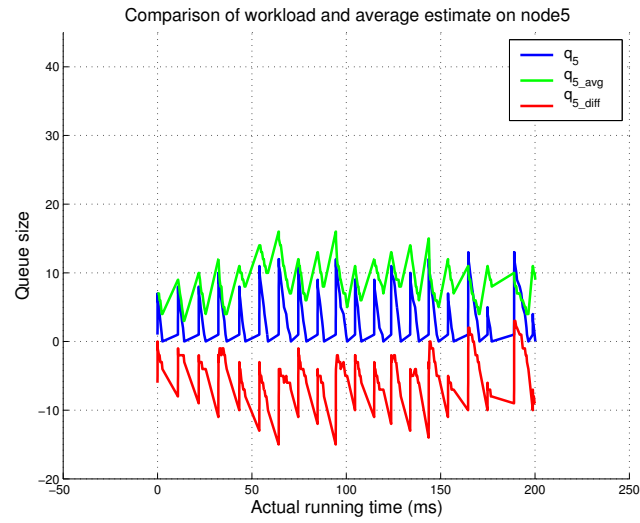


Figure 5.71: Workloads, average estimates and tracking differences on **node5** with tasks initially distributed to **node1**. Blue curve: workload q_5 , green: average estimate q_{5_avg} , and red: tracking difference q_{5_diff} .

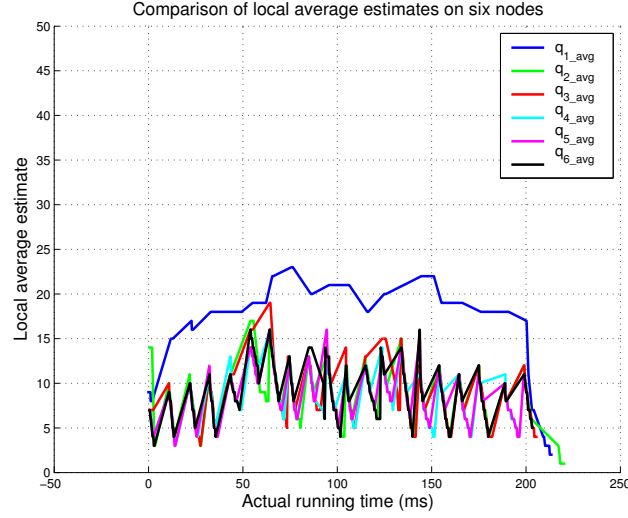


Figure 5.72: Comparison of average estimates on six nodes with tasks initially distributed to **node1**. q_{1_avg} , q_{2_avg} , q_{3_avg} , q_{4_avg} , q_{5_avg} , and q_{6_avg} are the average estimate at each of six nodes respectively.

node1 remains larger than those on the other nodes. When **node1** receives a block of requests from the search server, it balances its excess load to the other five nodes in the group, updates its average estimate, and brings its tracking difference to zero. After all 20 blocks of requests are received at about 200 *ms*, the load balancing method brings all average estimates close to each other (within a threshold that triggers a balancing action), and the system settles to a balanced state.

5.4 Summary

In this chapter, simulations and experimental results for load balancing strategies have been presented. For the nonlinear time delay model with proportional feedback control,

experiments found that a judicious choice of the feedback gain is necessary to achieve a faster balancing response without oscillatory behavior. For the nonlinear time delay model that incorporates processor resource constraints, a distributed closed-loop controller is implemented to balance the load at each node using not only local estimates of loads at other nodes, but also estimates of the number of tasks in transit to each node. Experimental results demonstrated substantial improvement in performance using a controller based on anticipated work loads over a controller based on local work loads only. Two implementations of parallel search servers have been discussed. Searches on a parallel DNA database showed a multi-threaded search server provided two orders of magnitude improvements in transaction times over a search server using hierarchical Simple Sockets. Experiments were performed on the parallel DNA database with integrated load balancing, and results demonstrated both the efficiency of the parallel DNA database and the efficacy of the load balancing strategy.

Chapter 6

Conclusions

This research addresses the problem of load balancing in parallel computing and its applications in parallel and distributed databases. The load balancing on a parallel computer has been modeled as deterministic nonlinear time-delay systems, and has been analyzed using system theoretic methods. The design and an implementation of a parallel DNA database, which was used to evaluate the load balancing strategies, have been discussed. Both simulations and experimental results on the parallel computer have been presented to support the accuracy of the nonlinear time-delay models and to demonstrate the efficacy of the load balancing strategies. This chapter provides a summary of the completed work and a short discussion of possible future directions for research.

6.1 Completed Work

Recall that the objective of this research is to study the load balancing problem for parallel computations. The critical feature of the load balancing problem is the delayed receipt of information and transferred load. Moreover, there is a trade-off between using processor resources to process tasks and the advantage of distributing the load evenly between the nodes to reduce overall processing time. The primary contribution of this research is its focus upon the load balancing problem from a control system design perspective, and its extensive evaluation of load balancing strategies through experiments in a parallel computation environment. For completeness, the model development and the theoretical analysis include contributions by J. D. Birdwell and J. Chiasson. The major tasks completed in this research are briefly summarized as follows.

- **Nonlinear Time-delay Model with Proportional Feedback Control.**

To account for the fact that a node would not initiate a negative amount of task transfer to a recipient node in an actual system, a nonlinear time-delay system with saturations was developed to model the load balancing strategy in a parallel computing network. Three essential properties of the load balancing strategy, including consistency, stability, and conservation of tasks, have been analyzed [28]. The excess load on each sending node was portioned out among the other nodes according to the amounts they are below the local estimate of network average load. Simulations of this nonlinear model indicated good agreement with an experimental implementation of the load balancing strategy on a parallel computer

[29]. Experiments also found that the feedback gain in the controller (load balancer) affects the performance of load balancing [30]. It is desirable to choose the fraction of excess load (to be transferred to other nodes) to be close to 0.5 to achieve a faster response without breaking into oscillatory behavior. The experimental results documented in this dissertation and similar behaviors observed in experiments on PlanetLab were summarized in [51].

- **Nonlinear Model with Resource Constraints.**

As load distribution and task processing contend for the same resources on each computational element in a parallel computer, a deterministic dynamic nonlinear system was presented to model load balancing for parallel computations incorporating both time delays and resource constraints. The model was presented with a proof of model consistency and open-loop simulations and experiments in [196]. It was also shown that the system was Lyapunov stable, but not necessarily asymptotically stable. A distributed closed-loop controller was proposed to balance load dynamically at each node by using not only local estimates of loads at other nodes, but also estimates of the number of tasks in transit to the local node [195]. Experimental results demonstrated substantial improvements in performance using a controller based on the anticipated estimates of workloads over a controller based on the local workloads only.

- **Parallel DNA Database with Load Balancing.**

The design and the implementation of a parallel DNA database were documented

in this dissertation. DNA profiles are stored in portions on each node of the parallel computer to form a distributed storage and to search in parallel. A DNA search engine running on each node is based upon the tree-structured rapid search method as described in [33]. A parallel search server using multi-threads, PVM and object serialization was presented, demonstrating two orders of magnitude improvements in transaction time over a search server using hierarchical sockets. The parallel database with integrated load balancing was presented as well. Experimental results on the parallel DNA database with integrated load balancing demonstrated both the efficiency of the multi-threaded parallel search server and the efficacy of the load balancing strategy. Initial results for parallel searches with load balancing integrated with a parallel DNA database are documented in [197].

6.2 Directions for Future Research

The ideas, concepts and experimental results presented in this dissertation offer interesting avenues for future research. Several major directions for future research are briefly discussed as follows.

6.2.1 Discrete Event Model

In this work, the load balancing problem in parallel computing has been modeled as nonlinear time-delay systems in continuous time. The advantage of using a continuous state load balancing model is that dynamic properties such as stability and model

consistency can be analyzed, and precise results can be obtained. However, delays in practice are not constant when broadcasting queue information, and are not linear in the size of packets when transferring tasks. The delays in a parallel computer network depend on many random factors such as network availability and software execution at each node. In other words, the actual delays are *random*. In addition, load balancing actions can not be performed continuously (or at exactly the same intervals), but are performed in a discrete fashion (depending on relative load level each time). All of these characteristics of the actual system impact performance, and have not been precisely captured by the presented continuous state models. An approach to modeling using a discrete event or hybrid state formulation that accounts for block transfers that occur after random intervals is advantageous in analyzing the network and load balancing.

To study these effects, a discrete event model using OPNET Modeler has been developed by J. White in the LIT lab. The OPNET Modeler [169] is a tool suite for the creation and analysis of discrete event simulations of computer networks. The OPNET simulations using finite state machine to model a process of load balancing can be seen in [31], and preliminary results are compared with load balancing experiments in [199]. Both discrete event simulations and experiments demonstrate the superiority of using the closed loop controller based on anticipated estimates of workloads. An analytical proof is advantageous to obtain a precise analysis, which is an interesting topic for future research. Other future work includes development of a more efficient scheme for load balancing in the presence of random delays, possibly by merging scheduling and

optimization methods into the discrete event controller.

6.2.2 Dynamic Resource Allocation

Each node running a load balancing strategy can serve as a hot spare node for every other in the same group by replication of stored data in multiple nodes. The load balancing strategy for the parallel databases in this work can be extended to a broader scope: dynamic resource allocation. The objective of dynamic resource allocation is to allocate the available system resources under dynamic conditions (such as node failures, system maintenance, database growth, and hardware/software upgrades) in such a way to maintain suitable performance. The entire parallel search system consists of a server and a group of nodes connected through a network switch. Resources of this parallel system that can be allocated include processors (or processing time) on each node, memory on each node and bandwidth across the network. The complete database of DNA profiles can be broken down into smaller component databases, which are then distributed among the nodes running a DNA search engine on each. This allows for these components of the database to be stored in a distributed fashion and to be searched in parallel.

Issues that need to be taken into consideration in assigning the DNA profiles among the nodes include: (1) the initial size of the entire database, and the initial size of a portion of DNA profiles allocated to each of the nodes depending on the available memory size and the processor speed of the respective nodes; (2) the rate of growth of the DNA database, (3) the number of node failures that can (or are allowed to)

be handled, (4) the database size that would be redistributed in case of a failure of a particular node, (5) the database size that will be reassigned when new nodes (or resources) are introduced, (6) the role and the number of hot spare nodes and, (7) overlapping or non-overlapping database portions. In addition, maintenance of nodes in a parallel system is required from time to time, and there should be a smooth process that does so without severely impacting the service time of the requests.

The resources of the parallel computer and the network are subject to constraints. Preliminary work on dynamic resource allocation has been conducted in the LIT lab to analyze the relationships between the resources and their constraints, and can be used to determine the amount of resources required and their allocation. This can be an interesting topic for future research based upon the present work.

6.2.3 Hierarchical Load Balancing

In a group of nodes with load balancing, every node broadcasts (by multicasting) its queue information to the other nodes, calculates its relative load level and load transfer portions, and then sends excess load (if any) to lightly-loaded nodes. It is not difficult to imagine that communication overhead is an issue when the scale of a group increases to hundreds or even thousands of nodes for very intensive computations. The experiments on the parallel machine indicate that the communication overhead increases greatly when a group has more than 16 nodes. The anticipatory control method can improve the performance by reducing unnecessary task transfers, but cannot avoid the communication overhead. An approach to balancing load across a large group of nodes is to

divide nodes into several hierarchies. Each hierarchy, which has several groups of nodes, is an autonomous system running a *group-wise load balancing* strategy. In each group, a *node-wise load balancing* strategy is executed to balance loads across the group only. This way, the communication overhead increases linearly with the scale of the system and can be reduced to a logarithmic order of the number of nodes in the group. By dividing a parallel system into hierarchical groups, workload across a parallel system of a large number of nodes can be balanced using hierarchical load balancing method.

Future work includes scaling up the load balancing methodology for larger networks. For example, while each node broadcasts both its queue size and the number of tasks in transit to other nodes, the local controller could perhaps send out tasks only to the nodes corresponding to the m largest values of load transfer portions (i.e., the m nodes with the largest p_{ij}). This would alleviate the communication cost (time) required to transfer tasks to all other nodes in the network. Several other extensions are also good topics for future research. For instance, it will be useful to extend the design of the parallel DNA database to other large databases of multi-dimensional data with naturally occurring patterns within the stored data, and to extend the load balancing strategies to large-scale web servers to obtain the best achievable performance.

Bibliography

Bibliography

- [1] C. T. Abdallah, N. Alluri, J. D. Birdwell, J. Chiasson, V. Chupryna, Z. Tang, and T. Wang. A linear time delay model for studying load balancing instabilities in parallel computations. *International Journal of System Science*, 34(10-11):563–573, August-September 2003.
- [2] C. T. Abdallah, J. D. Birdwell, J. Chiasson, V. Chupryna, Z. Tang, and T. Wang. Load balancing instabilities due to time delays in parallel computations. In *Proceedings of the 3rd IFAC Workshop on Time Delay Systems*, pages 198–202, December 2001. Sante Fe, NM, USA.
- [3] C. T. Abdallah, P. Dorato, J. Benitez-Read, and R. Byrne. Delayed positive feedback can stabilize oscillatory systems. In *Proceedings of 1993 the American Control Conference*, pages 3106–3107, June 1993. San Francisco, CA, USA.
- [4] T. Abdelzaher, Y. Lu, R. Zhang, and D. Henriksson. Practical application of control theory to web services. In *Proceedings of the 2004 American Control Conference*, pages 1992–1997, June 2004. Boston, MA, USA.

- [5] M. Adler, S. Chakrabarti, M. Mitzenmacher, and L. Rasmussen. Parallel randomized load balancing. *Random Structures & Algorithms*, 13(2):159–188, 1998.
- [6] J. Aguilar and E. Gelenbe. Task assignment and transaction clustering heuristics for distributed systems. *Information Sciences*, 97(1-2):199–219, 1997. Informatics and Computer Science.
- [7] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, M. Sam, N. Veronika, D. Quesnel, and S. Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5):749–771, May 2002.
- [8] E. Altman, T. Basar, and R. Srikant. Nash equilibria for combined flow control and routing in networks: Asymptotic behavior for a large number of users. *IEEE Transactions on Automatic Control*, 47(6):917–930, June 2002.
- [9] E. Altman and H. Kameda. Equilibria for multiclass routing in multi-agent networks. In *Proceedings of the 40th IEEE Conference on Decision and Control*, pages 604–609, December 2001. Orlando, FL, USA.
- [10] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, April 1967. Atlantic City, NJ, USA.
- [11] ANL Mathematics and Computer Science. Message Passing Interface, 2004. Available: <http://www-unix.mcs.anl.gov/mpi/>.

- [12] B. Ataşlar, P.-F. Quet, A. İftar, H. Özbay, T. Kang, and S. Kalyanaraman. Robust rate-based flow controllers for high-speed networks: the case of uncertain time-varying multiple time-delays. In *ACC*, volume 4, pages 2804–2808, June 2000. Chicago, IL, USA.
- [13] M. Athans and P. Falb. *Optimal Control*. McGraw-Hill, New York, NY, USA, 1966.
- [14] L. Aversa and A. Bestavros. Load balancing a cluster of web servers using distributed packet rewriting. In *Proc. 19th IEEE Int’l Performance, Computing, and Communications Conference*, Phoenix, AZ, USA, February 2000. <http://www.cs.bu.edu/faculty/best/res/papers/ipccc00.pdf>.
- [15] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal of Computing*, 29(1):180–200, 1999.
- [16] F. Baccelli and P. Bremaud. *Elements of Queuing Theory: Palm Martingale Calculus and Stochastic Recurrences*. Springer-Verlag, New York, NY, USA, 1994.
- [17] J. Baumgartner, D. J. Cook, and B. Shirazi. Genetic solutions to the load balancing problem. In *Proc. International Conference on Parallel Processing*, pages 72–81, 1995. Urbana-Champaign, IL, USA.
- [18] W. Becker. Dynamic balancing complex workload in workstation networks – challenge, concepts and experience. In *Proc. High-Performance Computing and*

- Networking*, volume 919 of *Lecture Notes in Computer Science*, pages 407–412. Springer, May 1995. Milan, Italy.
- [19] W. Becker and G. Waldmann. Exploiting inter task dependencies for dynamic load balancing. In *Proc. Third IEEE International Symposium on High Performance Distributed Computing*, pages 157–165, August 1994. San Francisco, CA, USA.
 - [20] R. Bellman and K. L. Cooke. *Differential-Difference Equations*. Academic Press, New York, NY, USA, 1963.
 - [21] D. P. Bertsekas and D. A. Castanon. Parallel asynchronous Hungarian methods for the assignment problem. *ORSA Journal on Computing*, 5(3):261–74, 1993.
 - [22] D. P. Bertsekas and D. A. Castanon. Parallel primal-dual methods for the minimum cost flow problem. *Computational Optimization and Applications*, 2(4):317–36, December 1993.
 - [23] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, Englewoods Cliffs, NJ, USA, 1989.
 - [24] M. A. Bhandarkar, R. K. Brunner, and L. V. Kalé. Run-time support for adaptive load balancing. In J. Rolim, editor, *Proceedings of IPDPS 2000 Workshop on Parallel and Distributed Processing*, volume 1800 of *Lecture Notes in Computer Science*, pages 1152–1159. Springer-Verlag, March 2000.

- [25] V. Bharadwaj, D. Ghose, and V. Mani. Optimal sequencing and arrangement in distributed single-level tree networks with communication delays. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):968–976, September 1994.
- [26] V. Bharadwaj, D. Ghose, V. Mani, and T. G. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, 1996. Los Alamitos, CA, USA.
- [27] J. D. Birdwell. Photos of the parallel computing cluster, December 2001. Available: <http://hickory.engr.utk.edu/~birdwell/parcomp3/index.html>.
- [28] J. D. Birdwell, J. Chiasson, C. T. Abdallah, Z. Tang, N. Alluri, and T. Wang. The effect of time delays in the stability of load balancing algorithms for parallel computations. In *Proceedings of the 42nd IEEE Conference on Decision and Control*, pages 582–587, December 2003. Maui, HI, USA.
- [29] J. D. Birdwell, J. Chiasson, Z. Tang, C. T. Abdallah, M. Hayat, and T. Wang. Dynamic time delay models for load balancing Part I: Deterministic models. In S.-I. Niculescu and K. Gu, editors, *Advances in Time-Delay Systems*, volume 38 of *Lecture Notes in Computational Science and Engineering*, pages 355–370. Springer-Verlag, 2003.
- [30] J. D. Birdwell, J. Chiasson, Z. Tang, C. T. Abdallah, and M. M. Hayat. The effect of feedback gains on the performance of a load balancing network with time

- delays. In *Proc. IFAC Workshop on Time Delay Systems – TDS03*, September 2003. Rocquencourt, France.
- [31] J. D. Birdwell, J. N. Chiasson, C. T. Abdallah, M. M. Hayat, Z. Tang, and J. White. The information age: Challenges and opportunities for control and engineering. In *to appear in Proceedings of the 2nd International Workshop on Software Cybernetics*, July 2005. Edinburgh, Scotland.
- [32] J. D. Birdwell, R. D. Horn, D. J. Icove, T.-W. Wang, P. Yadav, and S. Niezgoda. A hierarchical database design and search method for CODIS. In *Proceedings of the Tenth International Symposium on Human Identification*, Orlando, FL, USA, 1999. Available: <http://www.promega.com/geneticidproc/ussymp10proc/abstracts.html>.
- [33] J. D. Birdwell, T.-W. Wang, R. D. Horn, P. Yadav, and D. J. Icove. Method of indexed storage and retrieval of multidimensional information. *U.S. Patent 6,741,983*, 2004.
- [34] J. D. Birdwell, T.-W. Wang, and M. Rader. The University of Tennessee’s new search engine for CODIS. In *The 6th CODIS Users Conference*, February 2001. Arlington, VA, USA.
- [35] Y. P. Boglaev. Exact dynamic load balancing of MIMD architectures with linear programming algorithms. *Parallel Computing*, 18:615–623, 1992.

- [36] A. Boukerche and S. K. Das. Dynamic load balancing strategies for conservative parallel simulations. In *Proc. 11th Workshop on Parallel and Distributed Simulation*, pages 20–28. IEEE Computer Society Press, June 1997. Lockenhaus, Austria.
- [37] A. Boukerche and S. K. Das. Experimental studies in load balancing. In D. J. Pritchard and J. Reeve, editors, *Euro-Par '98 Parallel Processing, Proc. 4th International Euro-Par Conference*, volume 1470 of *Lecture Notes in Computer Science*, pages 318–321. Springer, September 1998. Southampton, UK.
- [38] J. M. Butler. *Forensic DNA Typing – Biology & Technology behind STR Markers*. Academic Press, London, UK, 2001.
- [39] R. Buyya. *High Performance Cluster Computing: Architectures and Systems, Volume 1*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1999.
- [40] C. E. Campbell and T. McRoberts. The simple sockets library, 1998. Available: <http://mysite.verizon.net/astronaut/ssl/index.html>.
- [41] L. M. Campos and I. Scherson. Rate of change load balancing in distributed and parallel systems. In *Proc. 13th Int'l Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, volume 2, pages 701–707. IEEE Computer Society Press, April 1999. San Juan, Puerto Rico.

- [42] J. Cao, Y. Sun, X. Wang, and S. K. Das. Scalable load balancing on distributed web servers using mobile agents. *Journal of Parallel and Distributed Computing*, 63(10):996–1005, October 2003.
- [43] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14:141–154, February 1988.
- [44] O. Catoni. Solving scheduling problems by simulated annealing. *SIAM Journal of Control and Optimization*, 36(5):1539–1575, 1998.
- [45] D. Cavendish, S. Mascolo, and M. Gerla. SP-EPRCA: an ATM rate based congestion control scheme based on a Smith predictor. Technical report, UCLA, 1997. <http://citeseer.csail.mit.edu/cavendish97speprca.html>.
- [46] S. Chakrabarti, J. Demmel, and K. Yelick. Models and scheduling algorithms for mixed data and task parallel programs. *Journal of Parallel and Distributed Computing*, 47(2):168–184, 1997.
- [47] H. Chang and W. J. Oldham. Dynamic task allocation models for large distributed computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1301–1315, 1995.
- [48] J. Chen and A. Miranda. A polynomial time approximation scheme for general multiprocessor job scheduling. *SIAM Journal of Computing*, 31(1):1–17, 2001.

- [49] J. Chiasson. A method for computing the interval of delay values for which a differential-delay system is stable. *IEEE Transactions on Automatic Control*, 33(12):1176–1178, December 1988.
- [50] J. Chiasson and C. T. Abdallah. Robust stability of time delay systems: Theory. In *Proceedings of the 3rd IFAC Workshop on Time Delay Systems*, pages 125–130, December 2001. Sante Fe, NM, USA.
- [51] J. Chiasson, Z. Tang, J. Ghanem, C. T. Abdallah, J. D. Birdwell, M. M. Hayat, and H. Jerez. The effect of time delays on the stability of load balancing algorithms for parallel computations. *IEEE Transactions on Control Systems Technology*, 2004. Accepted, to appear.
- [52] J. N. Chiasson, S. D. Brierley, and E. B. Lee. A simplified derivation of the Zeheb-Walach 2-d stability test with applications to time-delay systems. *IEEE Transactions on Automatic Control*, 30(4):411–414, April 1985.
- [53] P. Christen. A parallel iterative linear solver with dynamic load balancing. In *Proc. International Conference on Supercomputing*, pages 7–12, July 1998. Melbourne, Australia.
- [54] A. Corradi, L. Leonardi, and F. Zambonelli. On the effectiveness of different diffusive load balancing policies in dynamic applications. In *Proc. High-Performance Computing and Networking*, volume 1401 of *Lecture Notes in Computer Science*, pages 274–283. Springer, April 1998. Amsterdam, The Netherlands.

- [55] A. Corradi, L. Leonardi, and F. Zambonelli. Diffusive load-balancing policies for dynamic applications. *IEEE Concurrency*, 7(1):22–41, 1999.
- [56] A. Cortés, A. Ripoll, M. Senar, F. Cedó, and E. Luque. On the stability of a distributed dynamic load balancing algorithm. In *Proc. International Conference on Parallel and Distributed Systems*, pages 435–446, December 1998. Taiwan, China.
- [57] Cray. Cray products, 2004. Available: <http://www.cray.com/products/>.
- [58] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, 1989.
- [59] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proc. 10th IEEE Symposium on High Performance Distributed Computing*, pages 181–194, August 2001. San Francisco, CA, USA.
- [60] D. J. Daley and D. Vere-Jones. *An introduction to the theory of point processes*. Springer-Verlag, New York, 1988.
- [61] S. P. Dandamudi and M. Lo. A hierarchical load sharing policy for distributed systems. In *Proc. 5th Int’l Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 3–10, January 1997. Haifa, Israel.

- [62] S. K. Das, D. J. Harvey, and R. Biswas. Dynamic load balancing for adaptive meshes using symmetric broadcast networks. In *Proc. 1998 International Conference on Supercomputing*, pages 417–424, July 1998. Melbourne, Australia.
- [63] P. Dasgupta. *Performance Evaluation of Fast Ethernet, ATM and Myrinet under PVM*. Master’s thesis, University of Tennessee, Knoxville, May 2001.
- [64] P. Dasgupta, J. D. Birdwell, and T.-W. Wang. Timing and congestion studies under PVM. In *Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computation*, March 2001. Portsmouth, VA, USA.
- [65] P. Dasgupta, A. K. Majumder, and P. Bhattacharya. V-THR: An adaptive load balancing algorithm. *Journal of Parallel and Distributed Computing*, 42(1):101–108, 1997.
- [66] R. Datko. A procedure for determination of the exponential stability of certain differential-difference equations. *Quarterly Applied Mathematics*, 36:279–292, 1978.
- [67] J. de Ronde, A. Schoneveld, P. M. Sloot, N. Floros, and J. Reeve. Load balancing by redundant decomposition and mapping. In *Proc. High-Performance Computing and Networking*, volume 1067 of *Lecture Notes in Computer Science*, pages 555–561. Springer, April 1996. Brussels, Belgium.
- [68] T. Delica. Modeling of some plain load distribution strategies for jobs in a multi-computer system. *Information Sciences*, 97(1-2):35–43, 1997.

- [69] X. Deng, N. Gu, T. Brecht, and K. Lu. Preemptive scheduling of parallel jobs on multiprocessors. *SIAM Journal of Computing*, 30(1):145–160, 2000.
- [70] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.
- [71] S. Dhakal, B. S. Paskaleva, M. M. Hayat, E. Schamiloglu, and C. T. Abdallah. Dynamical discrete-time load balancing in distributed systems in the presence of time delays. In *Proceedings of the 42nd IEEE Conference on Decision and Control*, pages 5128–5134, December 2003. Maui, HI, USA.
- [72] Y. Diao, J. L. Hellerstein, A. J. Storm, M. Surendra, S. Lightstone, S. Pareky, and C. Garcia-Arellano. Using MIMO linear control for load balancing in computing systems. In *Proceedings of the 2004 American Control Conference*, pages 2045–2050, June 2004. Boston, MA, USA.
- [73] O. Diekmann, S. A. van Gils, S. M. V. Lunel, and H. O. Walther. *Delay Equations*. Springer-Verlag, 1995.
- [74] R. Diekmann, A. Frommer, and B. Monien. Efficient schemes for nearest neighbour load balancing. *Parallel Computing*, 25:789–912, 1999.
- [75] H. Dietz. Linux parallel processing HOWTO, 1998. Available: <http://www.tldp.org/HOWTO/Parallel-Processing-HOWTO.html>.
- [76] DOJ. Survey of DNA crime laboratories, 2004. Available: <http://www.ojp.usdoj.gov/bjs/abstract/sdnacl01.htm>.

- [77] S. Dutt and N. R. Mahapatra. Scalable load balancing strategies for parallel techniques for parallel A* algorithms. *Journal of Parallel and Distributed Computing*, 22(3):488–505, 1994.
- [78] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, May 1986.
- [79] H. Echols. *Operators and Promoters – The Story of Molecular Biology and Its Creators*. University of California Press, Berkeley, CA, USA, 2001.
- [80] S. Elmohamed. Parallel programming concepts, 2003. Available: <http://www.tc.cornell.edu/Services/Edu/Topics/ParProgCons/>.
- [81] FBI. Combined DNA Index System program, 2004. Available: <http://www.fbi.gov/hq/lab/codis/>.
- [82] D. G. Feitelson and L. Rudolph. Parallel job scheduling: Issues and approaches. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – Proc. IPPS’95 Workshop*, volume 949 of *Lecture Notes in Computer Science*, pages 1–18. Springer, April 1995. Santa Barbara, CA, USA.
- [83] M. D. Feng and C. K. Yuen. Dynamic load balancing on a distributed system. In *Proc. Sixth IEEE Symposium on Parallel and Distributed Processing*, pages 318–325. IEEE Computer Society Press, October 1994. Dallas, TX, USA.

- [84] A. Ferrari and V. S. Sunderam. TPVM: Distributed concurrent computing with lightweight processes. In *Proc. Fourth IEEE International Symposium on High Performance Distributed Computing*, pages 211–219, August 1995. Washington D.C., USA.
- [85] A. Ferrari and V. S. Sunderam. Multiparadigm distributed computing with TPVM. *Concurrency: Practice and Experience*, 10(3):199–228, 1998.
- [86] J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Adaptive local refinement with octree load balancing for the parallel solution of three-dimensional conservation laws. *Journal of Parallel and Distributed Computing*, 47(2):139–152, 1997.
- [87] G. Fox, W. Furmanski, J. Koller, and P. Simic. Physical optimization and load balancing algorithms. In *Proc. Fourth Conference on Hypercubes, Concurrent Computers and Applications*, pages 591–594, March 1989. Monterey, CA, USA.
- [88] M. A. Franklin and V. Govindan. A general matrix iterative model for dynamic load balancing. *Parallel Computing*, 22:969–989, 1996.
- [89] A. A. Freitas and S. H. Lavington. *Mining Very Large Databases with Parallel Processing*. Kluwer Academic Publishers, Boston, MA, USA, 1998.
- [90] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A Users’ Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994.

- [91] E. Gelenbe and A. Ghanwani. Approximate analysis of coupled queueing in ATM networks. *IEEE Communications Letters*, 3(2):31–33, February 1999.
- [92] J. Ghanem, C. T. Abdallah, M. M. Hayat, S. Dhakal, J. D. Birdwell, J. Chiasson, and Z. Tang. Implementation of load balancing algorithms over a local area network and the internet. In *Proceedings of the 43rd IEEE Conference on Decision and Control*, pages 4199–4202, December 2004. Paradise Island, Bahamas.
- [93] B. Ghosh, F. T. Leighton, B. M. Maggs, S. Muthukrishnan, C. G. Plaxton, R. Rajaraman, and A. W. Richa. Tight analysis of two local load balancing algorithms. *SIAM Journal of Computing*, 29(1):29–764, 1999.
- [94] V. Govindan and M. A. Franklin. Effect of control parameters on dynamic load balancing. In *Proc. International Conference on Parallel Processing*, pages 452–460. IEEE Computer Society Press, 1994. Raleigh, NC, USA.
- [95] J. Gustafson. Reevaluating Amdahl’s law. *Communications of the ACM*, 31(5):532–533, May 1988.
- [96] H. Guyennet, B. Herrmann, L. Philippe, and F. Spies. Simulation of dynamic load-balancing algorithms for multicomputers. *International Journal of Computers and Applications*, 19(2):70–76, 1997.
- [97] H. Guyennet and F. Spies. A cooperative algorithm for load balancing in interconnected transputer network. In M. Cosnard and R. Puigjaner, editors, *Decentral-*

- ized and Distributed Systems*, volume A-39 of *IFIP Transactions*, pages 305–313, Palma de Mallorca, Spain, September 1993.
- [98] A. Hać. Performance analysis of the priority queues for the load-building algorithms. *Journal of Systems and Software*, 19:69–74, 1992.
 - [99] A. Hać and X. Jin. Dynamic load balancing in a distributed system using a sender-initiated algorithm. *Journal of Systems and Software*, 11(2):79–94, 1990.
 - [100] A. Hać and X. Jin. A decentralized algorithm for dynamic load balancing with file transfer. *Journal of Systems and Software*, 16:37–52, 1991.
 - [101] A. Hać and T. J. Johnson. Sensitivity study of the load balancing algorithm in a distributed system. *Journal of Parallel and Distributed Computing*, 10(1):85–89, 1990.
 - [102] T. Hagerup. Allocating independent tasks to parallel processors: An experimental study. *Journal of Parallel and Distributed Computing*, 47(2):185–197, 1997.
 - [103] J. K. Hale and S. M. V. Lunel. *Introduction to Functional Differential Equations*. Springer-Verlag, 1993.
 - [104] M. Hamdi and C.-K. Lee. Dynamic load balancing of data parallel applications on a distributed network. In *Proc. International Conference on Supercomputing*, pages 170–179, July 1995. Barcelona, Spain.

- [105] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. In *Proc. ACM Sigmetrics Conference on Measurement & Modeling of Computer Systems*, pages 13–24, May 1996. Philadelphia, PA, USA.
- [106] M. Hayat, C. T. Abdallah, J. D. Birdwell, and J. Chiasson. Dynamic time delay models for load balancing Part II: a stochastic analysis of the effect of delay uncertainty. In S.-I. Niculescu and K. Gu, editors, *Advances in Time-Delay Systems*, volume 38 of *Lecture Notes in Computational Science and Engineering*, pages 371–385. Springer-Verlag, 2003.
- [107] J. R. Heath and B. Sivasena. Development, analysis and verification of a parallel hybrid dataflow computer architectural framework and associated load-balancing strategies and algorithms via parallel simulation. *Simulation*, 69(1):7–25, July 1997.
- [108] J. L. Hellerstein. Challenges in control engineering of computing systems. In *Proceedings of the 2004 American Control Conference*, pages 1970–1979, June 2004. Boston, MA, USA.
- [109] M. Herlihy, B. hong Lim, and N. Shavit. Scalable concurrent counting. *ACM Transactions on Computer Systems*, 13(4):343–364, November 1995.
- [110] D. Hertz, E. Jury, and E. Zeheb. Stability independent and dependent of delay for delay differential systems. *Journal of Franklin Institute*, 318(3):143–150, September 1984.

- [111] D. Hertz, E. I. Jury, and E. Zeheb. Simplified analytic stability test for systems with commensurate time delays. *IEE Proceedings, Part D*, 131(1):52–56, January 1984.
- [112] Y. C. Ho, R. S. Sreenivas, and P. Vakili. Ordinal optimization of discrete event dynamic systems. *Journal of Discrete Event Dynamic Systems*, 2(2):61–88, 1992.
- [113] R. D. Horn and J. D. Birdwell. Real-time object oriented intelligent control environment. In *Proceedings of the 31st IEEE Conference on Decision and Control*, pages 2984–2985, December 1992. Tucson, AZ, USA.
- [114] S. H. Hosseini, B. Litow, M. Malkawi, J. McPherson, and K. Vairavan. Analysis of a graph coloring based distributed load balancing algorithm. *Journal of Parallel and Distributed Computing*, 10(2):160–166, 1990.
- [115] Y. Hu and R. Blake. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing*, 25:417–444, 1999.
- [116] Y. F. Hu, R. J. Blake, and D. R. Emerson. An optimal migration algorithm for dynamic load balancing. *Concurrency: Practice and Experience*, 10(6):467–483, 1998.
- [117] K. A. Hua and W. Tavanapong. Performance of load balancing techniques for join operations in shared-noting database management systems. *Journal of Parallel and Distributed Computing*, 56:17–46, 1999.

- [118] C. Hughes and T. Hughes. *Object-Oriented Multithreading Using C++*. John Wiley & Sons, New York, NY, USA, August 1997.
- [119] C.-C. Hui and S. T. Chanson. Hydrodynamic load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 10(11):1118–1137, November 1999.
- [120] K. Hwang and Z. Xu. *Scalable Parallel Computing: Technology, Architecture, Programming*. McGraw-Hill, New York, NY, USA, 1998.
- [121] IBM. Blue Gene/L DD2 beta-Systems, 2004. Available: <http://www.research.ibm.com/bluegene/>.
- [122] IBM. Db2 universal database data warehouse, 2004. Available: <http://www-306.ibm.com/software/data/db2/udb/dwe/>.
- [123] J. Jansen and F. Sijstermans. Parallel branch-and-bound algorithms. *Int'l J. Future Generation Computer Systems*, 4(1):271–279, 1988.
- [124] W. Joosen, Y. Berbers, and P. Verbaeten. Dynamic load balancing in transputer applications with geometric parallelism. *Microprocessing and Microprogramming*, 30:77–84, 1990.
- [125] W. Joosen, S. Bijmens, B. Robben, J. V. Oeyen, and P. Verbaeton. Flexible load balancing software for parallel applications in a time-sharing environment. In *Proc. High-Performance Computing and Networking*, volume 919 of *Lecture Notes in Computer Science*, pages 398–406. Springer, May 1995. Milan, Italy.

- [126] W. Joosen, S. Bijmens, and P. Verbaeten. A reusable load balancer for parallel search problems. *Microprocessing and Microprogramming*, 38:205–212, 1993.
- [127] N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, Boston, MA, USA, 1999.
- [128] L. V. Kale. Comparing the performance of two dynamic load distribution methods. In *Proc. International Conference on Parallel Processing*, pages 8–12, University Park, PA, USA, August 1988.
- [129] H. Kameda, E.-Z. S. Fathy, I. Ryu, and J. Li. A performance comparison of dynamic versus static load balancing policies in a mainframe. In *Proceedings of the 39th IEEE Conference on Decision and Control*, pages 1415–1420, December 2000. Sydney, Australia.
- [130] H. Kameda, J. Li, C. Kim, and Y. Zhang. *Optimal Load Balancing in Distributed Computer Systems*. Springer-Verlag, London, UK, 1997.
- [131] E. Kamen. Linear systems with commensurate time delays: Stability and stabilization independent of delay. *IEEE Transactions on Automatic Control*, 27:367–375, April 1982.
- [132] H. K. Khalil. *Nonlinear Systems: Third Edition*. Prentice Hall Press, New Jersey, 2002.
- [133] L. Kleinrock. *Queueing Systems Volume 1: Theory*. John Wiley & Sons, New York, NY, USA, 1975.

- [134] C. Knessl and C. Tier. Two tandem queues with general renewal input I: Diffusion approximation and integral representation. *SIAM Journal of Applied Mathematics*, 59(6):1917–1959, 1999.
- [135] C. Knessl and C. Tier. Two tandem queues with general renewal input II: Asymptotic expansions for the diffusion model. *SIAM Journal of Applied Mathematics*, 59(6):1960–1997, 1999.
- [136] R. B. Konuru, S. W. Otto, and J. Walpole. A migratable user-level process package for PVM. *Journal of Parallel and Distributed Computing*, 40(1):81–102, 1997.
- [137] D. W. Krumme, G. Cybenko, and K. N. Venkataraman. Gossiping in minimal time. *SIAM Journal of Computing*, 21(1):111–139, 1992.
- [138] R. Kumar and M. Shayman. Centralized and decentralized supervisory control of nondeterministic systems under partial observation. *SIAM Journal of Control and Optimization*, 35(2):363–383, March 1997.
- [139] V. Kumar, A. Y. Grama, and N. R. Vempaty. Scalable load balancing techniques for parallel computers. *Journal of Parallel and Distributed Computing*, 22(1):60–79, 1994.
- [140] O. R. N. Laboratory. PVM: Parallel Virtual Machine, 2004. Available: <http://www.csm.ornl.gov/pvm/>.
- [141] Z. Lan, V. E. Taylor, and G. Bryan. Dynamic load balancing for adaptive mesh refinement application: Improvements and sensitivity analysis. In *Proc. 13th In-*

- ternational Conference on Parallel and Distributed Computing and Systems*, pages 142–147, August 2001. Anaheim, CA, USA.
- [142] Z. Lan, V. E. Taylor, and G. Bryan. Dynamic load balancing for structured adaptive mesh refinement application: Improvements and sensitivity analysis. In *Proc. International Conference on Parallel Processing*, pages 571–579, September 2001. Valencia, Spain.
- [143] Lawrence Livermore National Lab. Linux Cluster Thunder, 2004. Available: <http://www.llnl.gov/linux/thunder/>.
- [144] E. Lee and L. Markus. *Foundations of Optimal Control Theory*. Krieger Publishing, Melbourne, FL, USA, 1986. Reprint Edition.
- [145] B. Lewis and D. J. Berg. *Threads Primer: A Guide to Multithreaded Programming*. SunSoft Press, Mtn. View, CA, USA, 1995.
- [146] H. Lin and R. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, 13(1):32–38, 1987.
- [147] H. Lin and C. S. Raghavendra. A dynamic load-balancing policy with a central job dispatcher (LBC). *IEEE Transactions on Software Engineering*, 18(2):148–158, 1992.
- [148] H. T. Liu and J. Silvester. An approximate performance model for load-dependent interactive queues with application to load balancing in distributed systems. In

- Proc. 7th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 956–965, March 1988. New Orleans, LA, USA.
- [149] R. Lüling, B. Monien, and F. Ramme. Load balancing in large networks: a comparative study. In *Proceedings of 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 686–689, December 1991. Dallas, TX, USA.
- [150] N. R. Mahapatra and S. Dutt. Random seeking: a general, efficient, and informed randomized scheme for dynamic load balancing. In *Proc. 10th International Parallel Processing Symposium*, pages 407–412. IEEE Computer Society Press, April 1996. Honolulu, HI, USA.
- [151] N. R. Mahapatra and S. Dutt. Scalable global and local hashing strategies for duplicate pruning in parallel A* graph search. *IEEE Transactions on Parallel and Distributed Systems*, 8(7):738–756, 1997.
- [152] F. Matthijs, Y. Berbers, and P. Verbaeten. A flexible I/O framework for parallel and distributed systems. In *Proc. International Workshop on Object Orientation in Operation Systems*, pages 187–190. IEEE Computer Society Press, August 1995. Lund, Sweden.
- [153] T. Mattson and G. Henry. An overview of the Intel TFLOPS supercomputer, 1998. Available: http://www.intel.com/technology/itj/q11998/articles/art_1.htm.

- [154] R. A. Milito and E. Fernández-Gaucherand. Open-loop routing of N arrivals to M parallel queues. *IEEE Transactions on Automatic Control*, 40(12):2108–2114, 1995.
- [155] R. A. Milito and E. Fernández-Gaucherand. Routing arrivals to M parallel queues. In *Proc. IEEE Conference on Decision and Control*, pages 1415–1420, 1995. New Orleans, LA, USA.
- [156] M. Mitzenmacher. Load balancing and density dependent jump Markov processes. In *37th Annual Symposium on Foundations of Computer Science*, pages 213–222. IEEE Computer Society Press, October 1996. Burlington, VT, USA.
- [157] M. Mitzenmacher. On the analysis of randomized load balancing schemes. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 292–301, June 1997. Newport, RI, USA.
- [158] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965. Available: <ftp://download.intel.com/research/silicon/moorespaper.pdf>.
- [159] F. J. Muniz and E. J. Zaluska. Parallel load-balancing: an extension to the gradient model. *Parallel Computing*, 21:287–301, 1995.
- [160] MySQL. MySQL database server, 2004. Available: <http://www.mysql.com/products/>.

- [161] K. Nam, J. Seo, S. Lee, and J. Kim. Synchronous load balancing on hypercube multicomputers with faulty nodes. *Journal of Parallel and Distributed Computing*, 58(1):26–43, 1999.
- [162] NEC. High performance computing – SX series, 2004. Available: <http://www.hpce.nec.com/572.0.html>.
- [163] R. D. Nelson and T. K. Philips. An approximation to the response time for shortest queue routing. In *ACM SIGMETRICS Performance Evaluation Review*, volume 17, pages 181–189, May 1989. Berkeley, CA, USA.
- [164] R. D. Nelson and T. K. Philips. An approximation for the mean response time for shortest queue routing with general interarrival and service times. *Performance Evaluation*, 17(2):123–139, 1993.
- [165] D. M. Nicol and G. Ciardo. Automated parallelization of discrete state-space generation. *Journal of Parallel and Distributed Computing*, 47(2):153–167, 1997.
- [166] S. I. Niculescu and C. T. Abdallah. Delay effects on static output feedback stabilization. In *Proceedings of the 39th IEEE Conference on Decision and Control*, pages 2811–2816, 2000. Sydney, Australia.
- [167] L. Oliker and R. Biswas. Efficient load balancing and data remapping for adaptive grid calculations. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 33–42, June 1997. Newport, RI, USA.

- [168] L. Olikar, R. Biswas, and H. N. Gabow. Performance analysis and portability of the PLUM load balancing system. In D. J. Pritchard and J. Reeve, editors, *Euro-Par '98 Parallel Processing, Proc. 4th International Euro-Par Conference*, volume 1470 of *Lecture Notes in Computer Science*, pages 307–317. Springer, September 1998. Southampton, UK.
- [169] OPNET Technologies Inc. OPNET Modeler, 2004. Available: <http://www.opnet.com/products/modeler/home.html>.
- [170] Oracle. Oracle database products, 2004. Available: <http://www.oracle.com/database/index.html>.
- [171] A. Overkamp and J. H. V. Schuppen. Maximal solutions in decentralized supervisory control. *SIAM Journal of Control and Optimization*, 39(2):492–511, May 2000.
- [172] P. Perona and J. Malik. Scale space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(7):629–639, July 1990.
- [173] C. Philips, C. Stein, and J. Wein. Task scheduling in networks. *SIAM Journal of Discrete Mathematics*, 10(4):573–598, 1997.
- [174] PlanetLab Consortium. PlanetLab, 2004. Available: <http://www.planet-lab.org>.
- [175] S. Prasad. *Multithreading Programming Techniques*. McGraw-Hill, New York, NY, USA, 1997.

- [176] E. Rahm. A framework for workload allocation in distributed transaction processing systems. *Journal of Systems and Software*, 18(2):171–190, 1992.
- [177] P. Ramadge and W. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, 1987.
- [178] B. Rieken and L. Weiman. *Adventures in UNIX Network Applications Programming*. John Wiley & Sons, New York, NY, USA, 1992.
- [179] K. A. Robbins and S. Robbins. *Practical UNIX Programming – A Guide to Concurrency, Communication, and Multithreading*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1996.
- [180] A. Robertsson, B. Wittenmark, M. Kihl, and M. Andersson. Design and evaluation of load control in web server systems. In *Proceedings of the 2004 American Control Conference*, pages 1980–1985, June 2004. Boston, MA, USA.
- [181] R. Robinett, B. Petterson, and J. Fahrenholtz. Lag-stabilized force feedback damping. *Journal of Intelligent and Robotic Systems*, 21(3):277–285, March 1998.
- [182] H. G. Rotithor. Taxonomy of dynamic task scheduling schemes in distributed computing systems. *IEE Proceedings of Computer Digital Technology*, 141(1):1–10, 1994.
- [183] N. Rudin and K. Inman. *An Introduction to Forensic DNA Analysis*. CRC Press, Boca Raton, FL, USA, December 2001. Second edition.

- [184] R. Saeks. On the decentralized control of interconnected dynamical systems. *IEEE Transactions on Automatic Control*, 24(2):269–271, April 1979.
- [185] B. A. Sanders. An asynchronous, distributed flow control algorithm for rate allocation in computer networks. *IEEE Transactions on Computers*, 37(7):779–787, July 1988.
- [186] P. Sanders. Analysis of nearest neighbor load balancing algorithms for random loads. *Parallel Computing*, 25:1013–1033, 1999.
- [187] C. M. P. Santos and J. S. Aude. PM-PVM: A portable multithreaded PVM. In *Proc. 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, volume 1, pages 191–195. IEEE Computer Society Press, April 1999. San Juan, Puerto Rico.
- [188] K. Schloegel, G. Karypis, and V. Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *Journal of Parallel and Distributed Computing*, 47(2):109–124, 1997.
- [189] M. A. Senar, A. Cortesé, A. Ripoll, L. Hluchý, and J. Astalos. Dynamic load balancing. In J. C. Cunha, P. Kacsuk, and S. C. Winter, editors, *Parallel Program Development for Cluster Computing*, pages 69–95. Nova Science Publishers, New York, NY, USA, 2001.

- [190] J. Seo, S. Lee, and J. Kim. Synchronous load balancing in hypercube multi-computers with faulty nodes. In *Proc. International Conference on Parallel and Distributed Systems*, pages 414–421, 1997.
- [191] B. S. Siegell and P. Steenkiste. Automatic generation of parallel programs with dynamic load balancing. In *Proc. IEEE International Symposium on High Performance Distributed Computing*, pages 166–175, 1994.
- [192] B. S. Siegell and P. A. Steenkiste. Automatic selection of load balancing parameters using compile-time and run-time information. *Concurrency: Practice and Experience*, 9(4):275–317, 1997.
- [193] F. Spies. Modeling of optimal load balancing strategy using queueing theory. *Microprocessing and Microprogramming*, 41:555–570, 1996.
- [194] Sun Microsystems. High-End Servers: Sun Fire 12K Server, 2004. Available: <http://www.sun.com/servers/highend/sunfire12k/index.xml>.
- [195] Z. Tang, J. D. Birdwell, J. Chiasson, C. Abdallah, and M. Hayat. Closed loop control of a load balancing network with time delays and processor resource constraints. In S. Tarbouriech, C. Abdallah, and J. Chiasson, editors, *Advances in Communication Control Networks*, volume 308 of *Lecture Notes in Control and Information Sciences*, pages 245–268. Springer, 2004.
- [196] Z. Tang, J. D. Birdwell, J. Chiasson, C. Abdallah, and M. Hayat. A time delay model for load balancing with processor resource constraints. In *Proceedings of*

- the 43rd IEEE Conference on Decision and Control*, pages 4193–4198, December 2004. Paradise Island, Bahamas.
- [197] Z. Tang, J. D. Birdwell, J. Chiasson, C. Abdallah, and M. Hayat. Resource-constrained load balancing controller for a parallel database. In *submitted to the 44th IEEE Conference on Decision and Control*, December 2005. Seville, Spain.
- [198] Z. Tang and R. T. Whitaker. Modified anisotropic diffusion for image smoothing and enhancement. In *Nonlinear Image Processing and Pattern Analysis XII, Proceedings of SPIE*, volume 4304, pages 318–325, January 2001.
- [199] Z. Tang, J. White, J. Chiasson, J. D. Birdwell, C. Abdallah, and M. Hayat. Closed-loop load balancing: Comparison of a discrete event simulation with experiments. In *to appear in Proceedings of the 24th American Control Conference*, June 2005. Portland, OR, USA.
- [200] E. Tärnvik. Dynamo – a portable tool for dynamic load balancing on distributed memory multicomputers. *Concurrency: Practice and Experience*, 6(8):613–639, December 1994.
- [201] Teradata. Teradata warehouse, 2004. Available: <http://www.teradata.com/t/page/42711/index.html>.
- [202] The Mathworks, Inc. MATLAB and Simulink for Technical Computing, 2004. Available: <http://www.mathworks.com/>.

- [203] TOP500.org. TOP500 supercomputer sites, November 2004. Available: <http://www.top500.org/>.
- [204] J. Tsitsiklis, D. Bertsekas, and M. Athans. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE Transactions on Automatic Control*, 31(9):803–812, September 1986.
- [205] P. Valduriez. Parallel database systems: Open problems and new issues. *Distributed and Parallel Databases*, 1(2):137–165, 1993.
- [206] Virginia Tech. Virginia Tech Terascale computing facility, 2003. Available: <http://www.tcf.vt.edu/>.
- [207] P. R. D. Waal and J. H. V. Schuppen. A class of team problems with discrete action spaces: optimality conditions based on multimodularity. *SIAM Journal of Control and Optimization*, 38(3):875–892, 2000.
- [208] G. C. Walsh, H. Ye, and L. G. Bushnell. Stability analysis of networked control systems. *IEEE Transactions on Control Systems Technology*, 10(3):438–446, May 2002.
- [209] C. Walshaw and M. Berzins. Dynamic load-balancing for PDE solvers on adaptive unstructured meshes. *Concurrency: Practice and Experience*, 7(1):17–28, 1995.
- [210] C. Walshaw, M. Cross, and M. G. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 47(2):102–108, 1997.

- [211] T.-W. Wang, J. D. Birdwell, P. Yadav, D. J. Icove, S. Niezgoda, and S. Jones. Clustering study of STR allele distribution patterns. In *Proceedings of the Tenth International Symposium on Human Identification*, Orlando, FL, USA, 1999. Available: <http://www.promega.com/geneticidproc/ussymp10proc/abstracts.html>.
- [212] J. Watts and S. Taylor. A practical approach to dynamic load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):235–248, 1998.
- [213] R. T. Whitaker and S. M. Pizer. A multi-scale approach to nonuniform diffusion. *CVGIP: Image Understanding*, 57(1):99–110, 1993.
- [214] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, 1993.
- [215] H. S. Witsenhausen. On information structures and causality. *SIAM Journal of Control and Optimization*, 9(2):149–160, May 1971.
- [216] S. C. Wong. Group-based optimisation of signal timings using parallel computing. *Transportation Research*, 5C(2):123–139, April 1997.
- [217] C. Z. Xu and F. C. Lau. Analysis of the generalized dimension exchange method for dynamic load balancing. *Journal of Parallel and Distributed Computing*, 16(1):385–393, 1992.

- [218] C.-Z. Xu and F. C. Lau. Optimal parameters for load balancing using the diffusion method in k -ary n -Cube networks. *Information Processing Letters*, 47(4):181–187, 1993.
- [219] C.-Z. Xu and F. C. Lau. Iterative dynamic load balancing in multicomputers. *Journal of Operational Research Society*, 45(7):786–796, 1994.
- [220] C.-Z. Xu and F. C. Lau. Optimal parameters for load balancing with the diffusion method in mesh networks. *Parallel Processing Letters*, 4(1-2):139–147, 1994.
- [221] C.-Z. Xu and F. C. Lau. The generalized dimension exchange method for load balancing in k -ary n -Cubes and variants. *Journal of Parallel and Distributed Computing*, 24(1):72–85, 1995.
- [222] C. Z. Xu and F. C. Lau. *Load Balancing in Parallel Computers: Theory and Practice*. Kluwer Academic, 1997. Boston, MA, USA.
- [223] C.-Z. Xu, B. Monien, R. Lüling, and F. C. Lau. Nearest-neighbor algorithms for load-balancing in parallel computers. *Concurrency: Practice and Experience*, 7(7):707–736, 1995.
- [224] X. Yuan, B. He, D. Baalsara, and R. Melhem. A load balancing package for domain decomposition on distributed memory systems. In *Proc. High-Performance Computing and Networking*, volume 1067 of *Lecture Notes in Computer Science*, pages 547–554. Springer, April 1996. Brussels, Belgium.

- [225] F. Zambonelli. Exploiting biased load information in direct-neighbour load balancing policies. *Parallel Computing*, 25:745–766, 1999.
- [226] H. Zhou and A. Geist. LPVM: a step towards multithread PVM. *Concurrency: Practice and Experience*, 10(5):407–416, 1998.
- [227] S. Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, 14(9):1327–1341, September 1988.
- [228] A. Y. Zomaya, C. Ward, and B. Macey. Genetic scheduling for parallel processor systems: comparative studies and performance issues. *IEEE Transactions on Parallel and Distributed Systems*, 10(8):795–812, 1999.

Appendix

Appendix A

This appendix summarizes the parameters that govern performance of the parallel machine and other selected computers in the Laboratory for Information Technologies (LIT). The parallel machine consists of one server node **root**, one file server node **shrek**, and 24 computing nodes, consisting of 8 single processor nodes (**node01–node08**) and 16 dual processor nodes (**node09–node24**). **Zero** is the file system server for LIT, while **holly** is a client computer which connects to both **zero** and the parallel machine. Table A.1 lists the specifications of the parallel machine and two selected computers in the LIT lab. All these computers run the Linux operating system. All the nodes of the parallel machine, including **root** and **shrek**, are connected using a Foundry FastIron II 32-port Gigabit Ethernet switch, which is a wire speed (non-blocking) switch. A Fast Ethernet network also connects these nodes, using two Netgear 18-port Fast Ethernet switches. **Root**, **zero**, and **holly** are connected using a D-Link DGS-3224TGR Gigabit Ethernet switch at gigabit speeds (also wire speed), except for **holly**, which is connected at 100-megabit speed.

Table A.1: Configuration of the nodes of the parallel machine and two selected computers in LIT.

Host	CPU type	Clock speed	CPUs	RAM	Disk size	NICs
root	Intel Xeon	2.4 GHz	2	2048MB	120GB	Intel Pro1000 Intel Pro100 Intel Pro1000
shrek	Intel Xeon	2.4 GHz	2	2048MB	700GB (RAID5)	Intel Pro1000 Intel Pro100
node01– node08	AMD Athlon	1.33 GHz	1	768MB	80GB	Intel Pro1000 Intel Pro100
node09– node24	Athlon MP	1.4 GHz	2	1024MB	100GB	Intel Pro1000 Intel Pro100
zero	Intel Xeon	2.4 GHz	2	3072MB	80GB 700GB (RAID5)	Intel Pro1000
holly	Athlon XP	1.5 GHz	1	512MB	80GB	Realtek 100

Vita

Zhong Tang was born in JiangSu Province, China, on December 1, 1973. After completing his higher secondary education at Taixing High School in July 1990, he entered the Department of Automatic Control Engineering, Huazhong University of Science and Technology, where he was awarded a Bachelor of Science degree in June 1994. Fascinated by the control systems and computers, he began his graduate study in the Graduate School at Huazhong University of Science and Technology (HUST), and was awarded a Master of Science degree in June 1997. He then worked as a graduate researcher in the Institute of Multimedia and Database Technologies at HUST.

In December 1998, he enrolled in the Ph.D. program majoring in Electrical Engineering at Auburn University, AL, USA. He transferred to the University of Tennessee, Knoxville in August 1999 and then worked in the former Vision Group. He joined the Laboratory of Information Technologies in August 2000, and has been working under the supervision of Dr. J. Douglas Birdwell. He will complete his graduate study and achieve the Ph.D. degree in Spring 2005 with a specialization in control systems and parallel computing.