Resource-Constrained Load Balancing Controller for a Parallel Database

J. Douglas Birdwell, Zhong Tang, John Chiasson, Chaouki T. Abdallah and Majeed M. Hayat

Abstract— **The critical features of the load balancing problem are the delayed receipt of information and transferred load. Load distribution and task processing contend for the same resources on each computational element. This paper documents experimental results using a previously reported deterministic dynamic nonlinear system for load balancing in a cluster of computer nodes used for parallel computations in the presence of time delays and resource constraints. The 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 control law is implemented as a distributed closed-loop controller to balance the load at each node using not only local estimates of the queue sizes of other nodes, but also estimates of the number of tasks in transit to each node. Experimental results using a parallel DNA database show the superiority of using the controller based on the anticipated work loads to a controller based on local work loads.**

I. INTRODUCTION

Parallel computing, which uses multiple interconnected computational elements to solve a single problem, can be applied to large-scale parallel databases. For example, DNA databases have been growing rapidly in recent years, and are predicted to increase to an eventual scale of 10^8 profiles. Forensic applications require rapid searches on these DNA databases. The anticipated size and the search requirements for DNA databases necessitate the development of parallel DNA databases. New methods developed by Wang and Birdwell [1][2][3] lead naturally to a parallel decomposition of the DNA database search problem while providing orders of magnitude improvements in performance over current software. Distributing the load evenly on parallel architectures is a key activity required for an efficient implementation.

Distribution of computational load across available resources is referred to as the *load balancing* problem in the literature. This work uses a generalization of queue length of tasks to expected waiting time, normalizing to account for differences among computational elements (CEs), and aggregates the behavior of each queue. Previous results by the authors study the effects of delays in the exchange of information among CEs and the performance of a load balancing strategy [4][5][6][7], and provide some historical background. This paper documents experimental results using a parallel DNA database that demonstrates the efficacy of the controller.

Computational loads need to be distributed more or less evenly over the available CEs to effectively utilize a parallel computer architecture. The qualifier "more or less" is used because the communications required to distribute the load consume both computational resources and network bandwidth. It is not difficult to imagine scenarios in which load distribution occurs so frequently that tasks are shifted around a parallel architecture without being computed. Our work in [8] discusses a mathematical model that captures processor resource constraints in a load balancing system. This open loop model was shown to be self consistent and (Lyapunov) stable, and was validated using Simulink simulations and comparison with simple experiments using time delays to model database activities. Initial results showing an extension to closed loop control are presented in [9].

This work addresses closed loop control for a resource-constrained load balancing problem in the presence of time delays. While the author's prior publications document experimental work using a time delay to emulate a database search, this paper documents results using implementation of DNA profile databases containing several million profiles. Closed loop control requires knowledge of work loads throughout the system. 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 *transfer delays* that occur when transferring tasks from one node to another. The control law based on *delayed* information from other nodes can cause unnecessary data transfers between nodes (the queue lengths oscillate), and prolong the completion time [9]. In order to increase efficiency, a control law has been proposed that uses estimates of *anticipated* workloads, which includes not only local estimates of the queue sizes at the other nodes, but also estimates of the number of tasks in transit to it [9]. In this manner, each node has an estimate of the status of both CEs and tasks in transit to the CEs when making its control decision. Experiments on a parallel DNA database are presented in this paper that document both the implementation strategy and the efficacy of the load balancing strategy using anticipated work loads.

This paper is organized as follows. Section II describes

The work of J. D. Birdwell, J. Chiasson, and Z. Tang was supported by the National Science Foundation under grant number ANI-0312182. The work of C. T. Abdallah and M. M. Hayat was supported by the National Science Foundation through grant ANI-0312611. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. Government.

J. D. Birdwell and J. Chiasson are with the Dept. of Elec. & Comp. Engr., University of Tennessee, Knoxville, TN 37996-2100, USA, {birdwell,chiasson}@utk.edu

Z. Tang is with Dept. R46S, Advanced Technologies, Abbott Laboratories, Abbott Park, IL 60064-3500, USA, tang@lit.net. The work was done when the author was at the University of Tennessee.

C. T. Abdallah and M. M. Hayat are with the Dept. of Elec. & Comp. Engr., University of New Mexico, Albuquerque, NM 87131-1356, USA, {chaouki,hayat}@ece.unm.edu

a nonlinear time-delay model of a load balancing algorithm for a computer network that incorporates both time delays required to communicate between nodes and transfer tasks and processor resource constraints. Section II-B examines the feedback control law on a local node and how a sending node portions out its tasks to other nodes. Section III documents the implementation of a parallel database and its load balancing method. Section IV presents experiments on the parallel DNA database using the closed loop controller based on anticipated work load. Section V concludes this work.

II. MATHEMATICAL MODEL

To provide a control system design perspective for the load balancing problem, consider a computing network consisting of n computers (nodes) all of which can communicate with each other. At start up, the computers may be assigned an equal number of tasks. However, some nodes may operate faster than others, and 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.

A simple approach to load balancing would be to have each computer in the network broadcast 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 can then use this information to compute its local estimate of the average number of tasks in the queues of the n computers in the network. 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, can be used as the network average. Node i compares its queue size $q_i(t)$ with its estimate of the network average, and only if this is greater than zero or some positive threshold, 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 delay h_{ij}. The task transfer delay h_{ij} depends on the number of tasks to be transferred and is much greater than the communication delay τ_{ij}. The controller (load balancing algorithm) decides how often and fast to do load balancing (transfer tasks among the nodes) and how many tasks are to be sent to each node. It has been shown that this straightforward controller leads to unnecessary task transfers (the queue lengths oscillate) due to the time delays, and that a modification of this controller can be used to avoid unnecessary task transfers [9].

The critical features of the load balancing problem are the delayed receipt of information and transferred load. As explained, each node controller (load balancing algorithm) 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. In addition, both load distribution and task processing require processor time on each node. An important issue considered here is the effect of these delays and resource constraints on system performance. The model used here captures the effect of the delays in load balancing techniques as well as the processor constraints so that system theoretic methods can be used for analysis.

A. Basic Model

The mathematical model of the task load dynamics at a given computing node for load balancing is given by

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

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

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

In this model,

- n is the number of nodes.
- $x_i(t)$ 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 $x_i(t) = q_i(t)t_{p_i}$.
- $\lambda_i \geq 0$ 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 $\mu_i \equiv (1 \times t_{p_i})/t_{p_i} = 1$ for all i if $x_i(t) > 0$, while if $x_i(t) = 0$ then $\mu_i \triangleq 0$.
- $\eta_i = 0$ or 1 is the *control input* which specifies whether tasks (waiting time) are processed on a node or tasks (waiting time) are transferred to other nodes.
- U_{m0} is the limit on the rate at which data can be transmitted from one node to another and is basically a bandwidth constraint.
- $p_{ij}U_m(x_j)\eta_j(t)$ 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 ($\int_{t_1}^{t_2} U_m(x_j)\eta_j(t)dt$ 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 (i.e., $p_{ij}\int_{t_1}^{t_2} U_m(x_j)\eta_j(t)dt$).
- 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 to node i where h_{ij} $(h_{ii}=0)$ is the time delay for the task transfer from node j to node i.
- The factor t_{p_i}/t_{p_j} converts the waiting time from node j to waiting time on node 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 $q_j t_{p_i} = x_j t_{p_i}/t_{p_j}$.

All rates are in units of the rate of change of expected waiting time, or *time/time* which is dimensionless. Generalizing the queue length to an expected waiting time (normalization) helps to account for differences among CEs. The control input η_i is introduced to capture the processor constraints. Node i processes tasks in its queue when $\eta_i = 0$. If $\eta_i = 1$, node i can only send tasks to other nodes and cannot initiate transfers from another node to itself. The quantity p_{ij} is the fraction of waiting time above the estimate of the network

average to be transferred from node j to node i with $p_{ij} \geqslant 0, \sum_{i=1}^{n} p_{ij} = 1$ and $p_{jj} = 0$. One approach is to choose them as constant and equal, i.e.,

$$p_{ij} = 1/(n-1) \text{ for } j \neq i \text{ and } p_{jj} = 0. \tag{2}$$

Another approach defines p_{ij} based on the estimated state of the network, and is given in the following subsection.

The model (1) is shown in [8] to be self consistent in that the queue lengths are always nonnegative and the total number of tasks in all the queues and the network are conserved (i.e., load balancing can neither create nor destroy tasks). The model is only (Lyapunov) stable, and asymptotic stability must be insured by the choice of the feedback law.

B. Feedback Control

In [8], a feedback law at each node i was based on the value of $x_i(t)$ and the *delayed* values $x_j(t-\tau_{ij})$ $(j \neq i)$ from the other nodes, where τ_{ij} $(\tau_{ii} = 0)$ denote the time delays for communicating the expected waiting time x_j from node j to node i. This controller caused unnecessary task transfers back and forth between nodes due to delayed information of queues at other nodes (see [9]). However, there is additional information that can be made available to the controllers at every node – specifically, the information on q_{net_i}, which is the number of tasks that are in the network being sent to the i^{th} node, or equivalently, the waiting time $x_{net_i} \triangleq t_{p_i} q_{net_i}$.

A new control law was proposed by the authors in [9] that uses not only the local estimate of the work loads q_i on the other nodes, but also the number of tasks q_{net_i} in transit to it. Each node j sends to each node i in the network information on the number of tasks $q_{net_{ij}}$ it has decided to send to each of the other nodes. This way the other nodes can take into account this information (without having to wait for the actual arrival of the tasks) in making their control decision. The communication of the number of tasks $q_{net_{ij}}$ being sent from node j to node i is much faster than the actual transfer of the tasks. Furthermore, each node i also broadcasts its total (anticipated) amount of tasks, i.e., $q_i + q_{net_i}$ to the other nodes so that they have a more current estimate of the tasks on each node (rather than have to wait for the actual transfer of the tasks). The information that each node has is a more up to date estimate of the state of network using this scheme.

Define

$$z_i \triangleq x_i + x_{net_i} = t_{p_i}(q_i + q_{net_i}) \tag{3}$$

which is the *anticipated* waiting time at node i. Further, define

$$z_{i\_avg} \triangleq \left(\sum_{j=1}^{n} z_j(t-\tau_{ij}) \right) / n \tag{4}$$

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. Define

$$w_i(t) \triangleq x_i(t) - z_{i\_avg}(t) = x_i(t) - \frac{\sum_{j=1}^{n} z_j(t-\tau_{ij})}{n} \tag{5}$$

to be the expected 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 (5) we avoid transferring nonexistent tasks in its queue (i.e., tasks in transit to it) from a node. A control law based on the anticipated waiting time is chosen as

$$\eta_i(t) = h(w_i(t)), \tag{6}$$

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

$$h(w) = \begin{cases} 1 & \text{if} \quad w \geqslant 0 \\ 0 & \text{if} \quad w < 0. \end{cases}$$

The load transfer portions, i.e., p_{ij}, can be specified using the anticipated waiting time z_j of the other nodes as follows.

$$p_{ij} = \frac{\text{sat}(z_{j\_avg} - z_i(t-\tau_{ji}))}{\sum_{i \ni i \neq j} \text{sat}(z_{j\_avg} - z_i(t-\tau_{ji}))}, \tag{7}$$

where $\text{sat}(\cdot)$ is defined as

$$\text{sat}(x) = \begin{cases} x & \text{if} \quad x \geqslant 0 \\ 0 & \text{if} \quad x < 0. \end{cases}$$

The quantity $\text{sat}(z_{j\_avg} - z_i(t-\tau_{ji}))$ is a measure by node j as to how much node i is *below* node j's estimate of the network average (anticipated) waiting time. Node j then portions out its tasks among the other nodes according to the amounts they are below its estimate of the network average waiting time. It is obvious that $p_{ij} \geqslant 0, \sum_{i=1}^{n} p_{ij} = 1$ and $p_{jj} = 0$. All p_{ij} are defined to be zero, and no load is transferred, if the denominator is zero.

III. Parallel Database with Load Balancing

A parallel computer has been built as an experimental facility to evaluate load balancing strategies on parallel databases. A root node (search server) communicates with k groups of networked computers. Each of these groups is composed of n nodes holding identical copies of a portion of the database. Any pair of groups correspond to different databases, which are not necessarily disjoint. In the experimental facility, all machines run the Linux operating system. 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.

A. Parallel DNA Database

In the DNA database application, a database search engine is executed on each node of the parallel machine. The parallel 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 accesses tree-structured indices to locate database records that match search requests, as described in [3]. 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. Search requests that await processing may be placed in any queue associated with a search engine containing the same data, and the contents of these queues may be moved arbitrarily among the processing nodes of a group to achieve a balance of the load. Each node also runs a load balancing thread to exchange queue information with the other nodes and redistribute the tasks depending on the relative workload.

B. Multi-threaded Search Server

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. Figure 1 shows a multi-threaded search server using threads, PVM [10], and object serializations. The multi-threaded search server starts a listening thread (`LThread`) which listens for connection requests and manages a pool of service threads (`SThread`'s) that service these connections. Requests are distributed and results are gathered 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 [11] database for administration. Each search engine node also maintains a MySQL database for saving and restoring states.

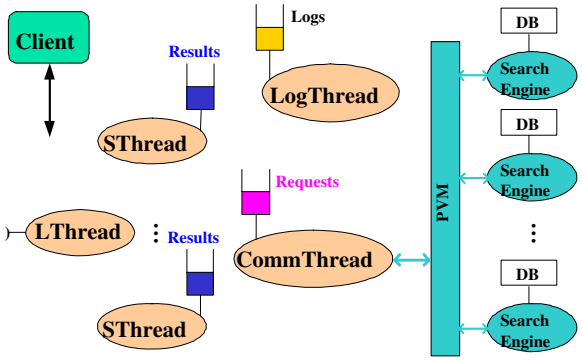

Fig. 1. Diagram of a multi-threaded search server.

IV. EXPERIMENTAL RESULTS

Experimental results for parallel searches with load balancing integrated with a parallel DNA database are presented here. The first set of experiments is conducted to evaluate the parallel database with integrated load balancing with an initial task distribution and no arriving tasks. The second set of experiments shows results with randomly generated task arrivals in the search engines and compares this to searching the parallel database with load balancing disabled. The third experiment shows results with load balancing on a larger network consisting of six nodes. These experimental results demonstrate the efficacy of the load balancing strategy using anticipated waiting times on a parallel DNA database.

A. Queues of Initial Tasks

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. 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 (7), and broadcasts the number of search requests to be transferred to each of the other nodes, which include the (anticipated) number of tasks being sent to `node1` and `node2` (66 each). Figure 2 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 $q_{1\_diff} = q_1 - q_{1\_avg} = -67$, 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 load balancing algorithm, which uses a closed loop controller based on anticipated work loads, works quite well in this situation.

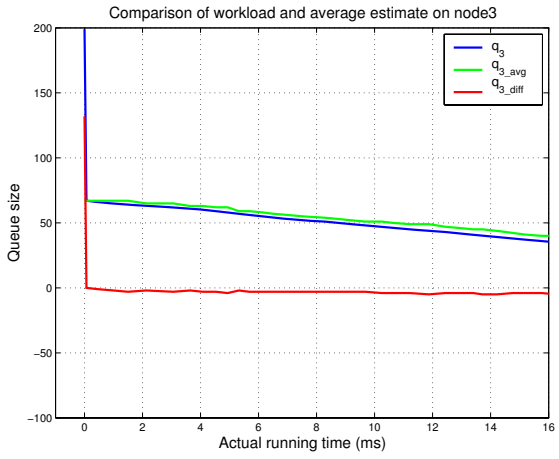

Fig. 2. Workloads and average estimates on `node3`.

Figure 3 compares the tracking differences between local workloads and average estimates on the three nodes. The local workloads track the 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`.

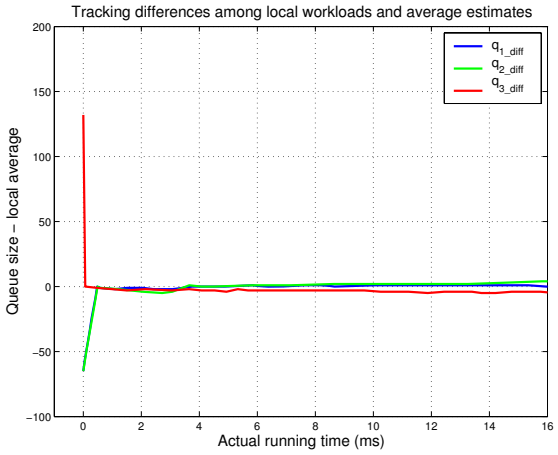

Fig. 3. Tracking differences on three nodes with initial tasks: q_{1_diff}, q_{2_diff}, q_{3_diff} for `node1`, `node2`, and `node3` respectively.

B. Randomly Generated Requests

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 $5ms$ 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 $5ms$, 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 $140ms$. 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 4 shows the workload, average estimate, and tracking difference on a representative `node2`. 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.

Figure 5 shows a comparison of average estimates computed on three nodes with randomly generated requests. When a new block of search requests arrives, the receiving node updates its average, which creates a step transient that is visible in the figure. The load balancing algorithm then evens out the tasks and brings the average estimates together. For

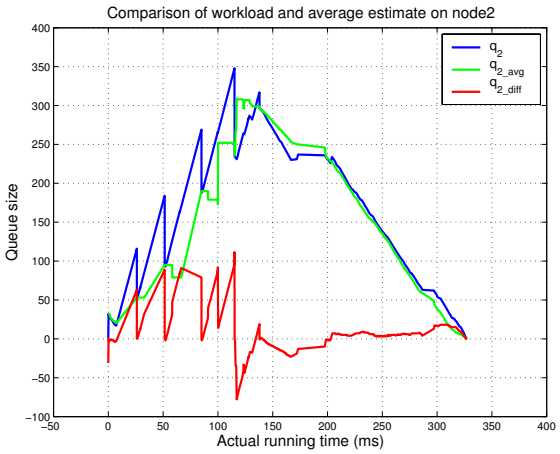

Fig. 4. Workloads, average estimates and tracking differences on `node2` with random requests. Blue curve: workload q_2, green: average estimate q_{2_avg}, and red: tracking difference q_{2_diff}.

$t > 200ms$, no new search requests arrive. The system settles to a balanced state, and the average estimates on three nodes closely follow each other.

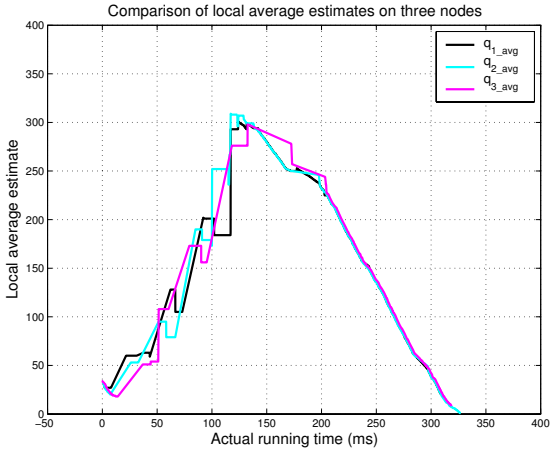

Fig. 5. Comparison of average estimates on three nodes with randomly generated requests. The black, cyan and pink curve stands for q_{1_avg}, q_{2_avg}, q_{3_avg} on `node1`, `node2`, and `node3` respectively.

Figure 6 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.

C. Balancing on Multiple Nodes

This set of experiments show results for parallel searches with load balancing on a larger network of multiple nodes ($n = 6$). 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

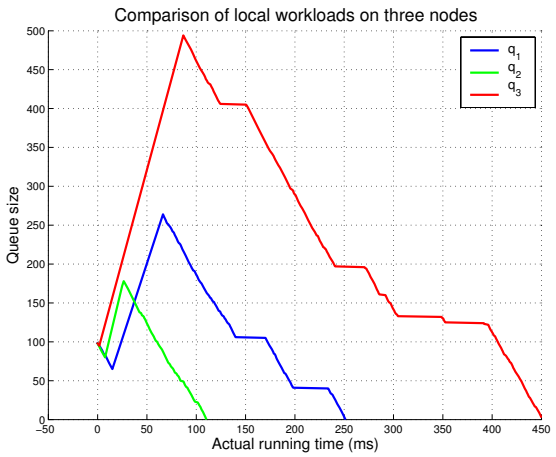

Fig. 6. Responses of queue sizes on three nodes without load balancing.

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.

Figure 7 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, which creates a step transient as shown in Figure 7. The load balancing algorithm then evens out the tasks and brings the average estimate close to that on other nodes. For $t > 200ms$, no new search requests arrive. The system settles to a balanced state, and the average estimates on the six nodes closely follow each other.

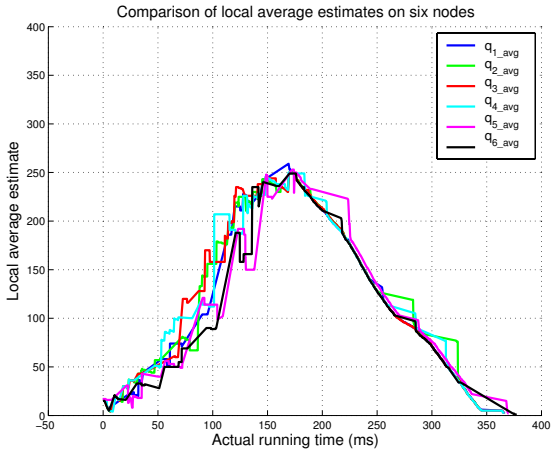

Fig. 7. Comparison of average estimates on six nodes with randomly generated requests.

Notice that the previous experiment 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 7), while it took 327.1*ms* to complete all 1,000 tasks on three nodes in the previous experiment (see Figure 5). For this case, the speed-up is 73% when the number of nodes is doubled.

V. SUMMARY

A load balancing algorithm for parallel computing is modeled as a nonlinear dynamic system incorporating both time delays and processor resource constraints. A closed loop controller is implemented that uses not only the local queue size, but also an estimate of the number of tasks in transit to the queue from other nodes. Experiments on a parallel DNA database demonstrate the efficacy of the load balancing strategy.

REFERENCES

[1] T. W. Wang, J. D. Birdwell, P. Yadav, D. J. Icove, S. Niezgoda, and S. Jones, "Natural clustering of DNA/STR profiles," in *Tenth International Symposium on Human Identification*, September 1999. Orlando, FL, USA.

[2] 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 *Tenth International Symposium on Human Identification*, September 1999. Orlando, FL, USA.

[3] 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.

[4] 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," *The International Journal of System Science*, vol. 34, pp. 563–573, August-September 2003.

[5] 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 *Advances in Time-Delay Systems* (S.-I. Niculescu and K. Gu, eds.), vol. 38 of *Lecture Notes in Computational Science and Engineering*, pp. 355–370, Springer-Verlag, 2003.

[6] M. 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 *Advances in Time-Delay Systems* (S.-I. Niculescu and K. Gu, eds.), vol. 38 of *Lecture Notes in Computational Science and Engineering*, pp. 371–385, Springer-Verlag, 2003.

[7] 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*, pp. 582–587, December 2003. Maui, HI, USA.

[8] Z. Tang, J. D. Birdwell, J. Chiasson, C. T. Abdallah, and M. M. Hayat, "A time delay model for load balancing with processor resource constraints," in *Proceedings of the 43rd IEEE Conference on Decision and Control*, pp. 4193–4198, December 2004. Paradise Island, Bahamas.

[9] 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 *Advances in Communication Control Networks* (S. Tarbouriech, C. Abdallah, and J. Chiasson, eds.), vol. 308 of *Lecture Notes in Control and Information Sciences*, pp. 245–268, Springer, 2004.

[10] 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*. Cambridge, MA: MIT Press, 1994.

[11] MySQL, "MySQL database server," 2004. Available: http://www.mysql.com/products/.