LoPC: Modeling Contention in Parallel Algorithms

by

Matthew Frank

B.S., University of Wisconsin (1994)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1997

Author .....

Department of Electrical Engineering and Computer Science

November 18, 1996

Certified by .....

Anant Agarwal

Associate Professor of Computer Science and Engineering

Thesis Supervisor

Accepted by .....

Arthur C. Smith

Chairman, Departmental Committee on Graduate Students

LoPC: Modeling Contention in Parallel Algorithms

by

Matthew Frank

Submitted to the Department of Electrical Engineering and Computer Science
on November 18, 1996, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Parallel algorithm designers need computational models that take first order system costs into account, but that are also simple enough to use in practice. This thesis describes the LoPC model, which is inspired by the LogP model, but which accounts for contention in message passing algorithms on a multiprocessor or network of workstations communicating via active messages. While LoPC is based on mean value analysis, it parameterizes architectures and algorithms in exactly the same way as the LogP model. LoPC takes the L, o and P parameters directly from the LogP model and uses them to predict the cost of contention, C, for processing resources.

From LoPC's mean value analysis, which is itself straight forward, we derive several even simpler rules of thumb for common communication patterns. We show that the LoPC model can provide accurate predictions for client-server communication patterns and for algorithms with irregular, but homogeneous, communication patterns. In addition, we demonstrate how to adapt LoPC to deal with systems that include extra protocol processing hardware to implement coherent shared-memory abstractions.

Thesis Supervisor: Anant Agarwal
Title: Associate Professor of Computer Science and Engineering

Acknowledgments

I am grateful to a great many people for their encouragement and help in moving this work to completion. Mary Vernon of the University of Wisconsin has provided guidance and numerous helpful suggestions. Kirk Johnson and Frans Kaashoek have been steady sources of support and advice. My advisor, Anant Agarwal, has pointed me down numerous profitable paths and provided an inspirational and energetic model.

In addition the members of the Alewife group, Ken Mackenzie, Donald Yeung, John Kubiatowicz, Jonathan Babb, Rajeev Barua, Fred Chong, Victor Lee and Walter Lee have generated an exciting environment in which to learn and work. Anne McCarthy has cleared up bureaucratic issues for me on a regular basis, allowing me to concentrate on things I have some chance of understanding.

Larry Rudolph, Charles Leiserson, Donald Yeung and Richard Lethin have provided useful comments on versions of this document.

Finally, I thank my wife, Kathleen Shannon. This work wouldn't exist but for her patience and supportive love.

The author is supported by an NSF Graduate Research Fellowship. The Alewife project is funded in part by ARPA contract #N00014-94-1-0985 and in part by NSF grant #MIP-9504399.

Contents

Chapter 1

Introduction

Light-weight user-level message passing paradigms, like Active Messages [22], are an increasingly popular tool for writing parallel applications. To design effective algorithms, programmers need a simple cost model that accurately reflects first-order system overheads.

The LogP model [7] has been successful at accurately modeling and optimizing algorithms with regular, ordered communication patterns on active-message based systems. The LogP model is simple to use and accounts for network latency and message passing overhead. However, it does not make any prediction about the costs of contention, which can be particularly significant for algorithms with irregular communication patterns. Lewandowski [16] successfully used LogP to analyze a work-pile algorithm with only a relatively small amount of communication. However, when Dusseau used LogP to analyze a variety of sorting algorithms with irregular communication patterns [8], she found that some of her models underestimated execution time and attributed the difference to contention costs. Algorithms that have irregular communication include hash algorithms and applications that use indirect array accesses. Coherent shared-memory systems also often exhibit irregular communication because the home-node for each coherence unit is found using a simple hash function.

Holt *et al* [12] have used LogP as a framework for an experimental study of contention in memory controllers for shared memory. For a variety of SPLASH benchmark applications and a variety of controller speeds and network latencies they find that contention in the memory controller dominates the costs of handler service time and network latency. Holt *et*

al tried to supplement their simulator study with a queueing model but abandoned the effort because they found the errors in their analysis to be unacceptably large (up to 35% of total response time). We believe that the LoPC model extended for non-blocking communication will be applicable to this kind of application-based architectural study for shared memory systems. For the blocking communication patterns we have studied, we have not observed errors in LoPC larger than 6%.

In fact, regular communication patterns can also demonstrate contention. Brewer and Kuszmal [3] measured the communication costs in very regular, all-to-all communication patterns carefully designed on the CM-5 to interleave message arrivals across processors so as to avoid contention. They discovered that the pattern quickly became virtually random, largely due to small variances in the interconnect.

The original LogP paper also notes that the model underestimates the cost of all-to-all communication on the CM-5 unless extra barriers are inserted to resynchronize the communication pattern. However, very low-latency barriers like those on the CM-5 are very expensive relative to other hardware components [19]. Few, if any, current generation multiprocessors or NOWs implement this feature.

The evidence thus suggests that contention for message-processing resources is a significant factor in the total application run time for many *fine-grain* message-passing algorithms (*i.e.*, those that communicate frequently), including those with irregular communication patterns and those that have regular communication patterns but are not tightly synchronized.

The goal of this thesis is to create a new model for analyzing parallel algorithms, LoPC, that provides accurate predictions of contention costs. LoPC is inspired by LogP and, like LogP, is motivated by Valiant's observation [21] that the parallel computing community requires models that accurately account for both important algorithmic operations and realistic costs for hardware primitives. The LoPC approach is to feed the parameters generated for a LogP analysis (network latency, message passing overhead and number of processors) to a simple queueing model to calculate contention costs.

We illustrate the LoPC model for two important classes of algorithms: homogeneous all-to-all communication and client-server workpile applications. We have validated these

models against both an event driven simulation and against synthetic micro-benchmarks running on the MIT Alewife multiprocessor. We find that the queueing model is accurate to within about six percent. Because LoPC is both simple to use and accurately models contention costs we believe it is a tool that will be broadly applicable to studying algorithms and architectural tradeoffs on both current and next generation parallel architectures.

Using the LoPC model we derive a number of interesting insights about the costs of contention in applications with irregular communication patterns. For example, in all-to-all communication patterns we find that on average every message either interrupts an active job or creates processor contention that causes another request to queue. This phenomenon leads to the interesting result that for homogeneous peer-to-peer communication patterns, the cost of contention is approximately equal to the cost of processing an extra message. Thus, in addition to deriving tight bounds on the total cost of contention we are able to develop a simple rule of thumb to accurately predict the run time of an interesting class of algorithms.

Although the simple rule of thumb holds only in the homogeneous case, the LoPC queueing model is itself both simple and computationally efficient, so it can be used in more general cases. For example, we use LoPC to characterize the run time of client-server work-pile applications in which there is no possible contention-free communication pattern. The LoPC analysis allows us to find an optimal allocation of nodes between clients and servers.

The next section discusses the architectural assumptions we make. Section 3 describes how to parameterize the LoPC model. Section 4 introduces the contention model we use. Section 5 goes through a complete LoPC analysis for the case of homogeneous all-to-all communication and derives bounds on the total cost of contention. Section 6 uses LoPC to find the optimal allocation between clients and servers in a work-pile algorithm. Finally, Section 7 concludes.

Chapter 2

Architectural Assumptions

The systems we model consist of a set of processing nodes each with an interface to a high speed interconnect, (see Figure 2-1.) Each node may send a message to any other node. The message contains a pointer to a handler and some small amount of data (typically around eight words). When that message arrives at the destination node, it interrupts the running job. The destination processor atomically runs the handler, which can perform arbitrary computation, and then returns to its background job. If additional requests arrive while the atomic handler is running, they are queued in a hardware FIFO. When the first handler finishes, the processor is again interrupted for each additional message in the queue.

This type of communication model using messages, called *Active Messages* [22], is general enough to implement arbitrarily complex communication and synchronization protocols. For example, a typical blocking request might begin with a node sending a message and then spinning on a counter variable. At the destination the message handler runs, perhaps loading or storing some data, and then sends a reply message back to the requester.

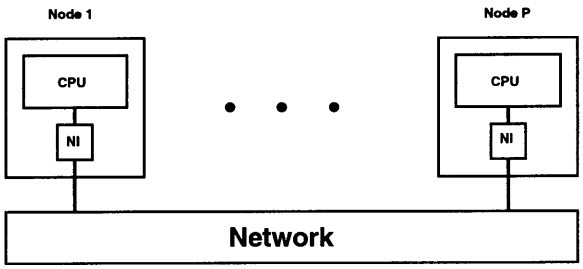

Figure 2-1: Architecture

When the reply reaches the requester, it interrupts the spinning background job and runs a reply handler. The reply handler does some work, decrements the counter variable, and then exits. When the background thread resumes, it finds that the counter has changed, finishes spinning, and continues with its work.

Handlers in the active message model are assumed to run at user level and in the application address space. This has two consequences. First, the operating system must provide some concept of *handler atomicity*. If a message arrives while a handler is running it must be queued until the previous handler finishes. The class of machines we model typically provide some hardware support for maintaining atomicity [1, 6, 10, 17, 19]. In particular the Alewife machine provides hardware network input queues which can hold up to 512 bytes of data, and an application controlled interrupt enable flag in the network controller.

An additional consequence is that the operating system must provide support for multiple applications sending messages. There are a number of ways to avoid the problem of messages arriving for a process that is not currently scheduled. One solution would be to have the operating system buffer messages when they arrive and then redirect them to the appropriate application when it becomes scheduled. Unfortunately, the high cost of buffering will typically be unacceptable to fine-grain parallel applications. The typical solution to this problem is to *coschedule* the machine so that message arrivals will coincide with the correctly scheduled application. This is the approach assumed in this thesis.

We make two simplifications to make our model tractable. First we assume that the interconnect is contention free. We model contention only for processor resources. Second we assume that the hardware message buffers at the nodes are infinitely large. We find that these assumptions don't affect our results for the short messages (less than about 8 data words) and low cost handlers that we used in our validations.

While validating our model, we compared results on the Alewife multiprocessor [1] against our event driven simulator, which has a contention free network and infinitely large message buffers. The simulator gives results accurate to within about 1% for all the communication patterns discussed in this thesis.

The next section gives an example of how the LogP and LoPC models are parameterized.

Chapter 3

Parameterization for LoPC

The process of parameterizing the LoPC model follows exactly the same lines as parameterizing a LogP analysis and uses both an algorithmic characterization and an architectural characterization. The model predicts total application run times from these two characterizations. In this section we will discuss both of these parameterizations.

Algorithmic Parameters Algorithmic characterization using either LogP or LoPC starts by finding the total number of arithmetic and communication operations performed by the algorithm. The differences between the LoPC and LogP models lie not in how they are parameterized nor in how the basic algorithmic analysis proceeds. LoPC simply extends the LogP analysis by calculating the average cost of contention, C, using the LogP parameters.

As with the LogP model, the method for deriving parameters varies from algorithm to algorithm. To illustrate the technique we will calculate the number of arithmetic and communication operations for a straight forward matrix-vector multiply routine.

A LoPC algorithm characterization produces an average time between requests, W, and the total number of messages sent by each node, n. Suppose we have an $N \times N$ matrix, A, that is cyclically distributed across P processors such that row i of the matrix is assigned to processor i mod P, and a vector x that is replicated on each processor. We wish to multiply $A \times x$ and replicate the resulting vector, y across all processors. Each processor will be responsible for the N/P dot products corresponding to the rows of A that are assigned to it.

After a processor computes the dot product of row A_i with x to produce the value y_i,

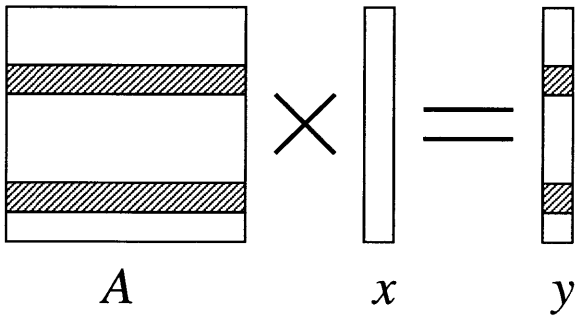

Figure 3-1: Matrix-vector multiply with matrix A cyclically distributed

LoPC	LogP	Description
S_l	L	Average wire time (latency) in the interconnect
S_o	o	Average cost of message dispatch
-	g	Peak processor to network bandwidth
P	P	Number of processors
C_o^2	-	Variability in message processing time (optional)

Table 3.1: Architectural Parameters of the LoPC Model

that value must be communicated to each of the other $P-1$ processors. We will assume that the values are communicated with *put* operations. A message is sent to a remote node containing the value and an address. The handler on the remote node stores the value in memory and sends an acknowledgment message back to the originator. The node that originates the request is blocked until the acknowledgment message returns.

The total work done by each node, then, consists of $m = N/P \times N$ multiply-add operations to calculate the dot-products and $n = N/P \times (P-1)$ *put* operations. These are exactly the quantities required to parameterize the LogP model.

The LoPC model requires one further step to calculate the average work done between remote requests, $W = \frac{m}{n}$. For this algorithm W will be the cost of $\frac{m}{n} = N/(P-1)$ multiply-add operations (or $\frac{1}{P-1}$ the cost of an N element dot product.) Using this value, the LoPC model will calculate the average run time of the algorithm, including the costs of contention for processor resources.

Architectural Parameters The architectural parameters used by the LoPC model are also very similar to those used by the LogP model. An algorithmic analysis under the LogP model depends on four architectural parameters, shown in table 3.1. L represents the

network *latency*. This is the time that the message spends in the interconnect between the completion of message injection by the processor, and arrival of the message at the remote node. It does not include any processing costs for the message. The processor overheads for injecting and receiving messages are covered in other parameters.

The o parameter represents the *overhead* of sending or receiving a message. This parameter corresponds with costs on the CM-5 multiprocessor [15] for which the LogP model was targeted. The cost of sending on the CM-5 is relatively quite large compared to more advanced message passing systems. The costs of interrupts on the CM-5 are also quite high, so the LogP model assumes that all message notification is done through polling. Interrupts are not modeled.

The g parameter is also somewhat peculiar to the CM-5 system. It represents the minimum *"gap"* between message sends and is the inverse of the peak processor to network bandwidth. This parameter is necessary on the CM-5 because the processor to network bandwidth is quite small. We expect that, in fact, most message passing platforms will have network interfaces with balanced bandwidth, *i.e.*, the gap will be 0.

The P parameter, finally, is the number of processors available for use in the system.

The LoPC parameters are similar to the LogP parameters. S_l, the average service time in the network, corresponds exactly to the L parameter from the LogP model. The P parameter, likewise, represents the number of processors in both models.

The S_o parameter corresponds approximately to the o parameter in the LogP model in that it measures message processing overhead. While the LogP model assumes a polling model with relatively expensive sends, the LoPC model assumes an interrupt model with relatively low costs for sending a message. S_o represents the cost of taking a message interrupt and handling the corresponding request. On most machines, the majority of this cost will be devoted to the interrupt.

The g parameter, which, in LogP, accounts for the peak processor to network bandwidth, is not included in the LoPC model because we have not yet found it to be relevant. We believe that on most current and future multiprocessors and NOWs the bandwidth between the node and interconnect will be roughly balanced with the rate at which the processor can compose messages.

Finally, LoPC also optionally permits the use of a parameter, C_o^2, that represents the squared coefficient of variation of service times for message handlers. The default LoPC model assumes exponential distributions, (equivalent to assuming $C_o^2 = 1$). We include this parameter because many message handlers consist of short instruction streams with low variability. These handlers have service time distributions that are closer to constant than exponential. We can represent this in the LoPC model by setting $C_o^2 = 0$.

Except for these few minor differences in architectural parameters, the parameterization of the LoPC and LogP models is very similar.

Chapter 4

The LoPC Model

LoPC extends LogP by calculating the average cost of contention for processor resources using the LogP parameters. Our computational model assumes a message passing machine with P nodes which can communicate through a high-speed interconnect. Each node, i, runs a thread T_i. These threads do some local work and then after an average service time W they make a blocking request to some other node and begin waiting for a reply. Section 3 discusses how to derive the parameter W from the algorithm being modeled. Each request travels through the interconnect, which is assumed to be contention free, at an average delay of S_l and arrives with some probability, V_{ij}, at one of the $P-1$ other nodes, j, see Figure 4-1.

At the point when a request arrives it interrupts the thread, T_j, running on the destination node, j, and runs a high-priority request handler, H_q, for an average delay of S_o (including the cost of taking the interrupt). When the handler finishes it sends a message through the interconnect, again with delay S_l, to the requesting node. Finally, when the message arrives back at its home it interrupts the processor and runs a high-priority reply handler, H_y, with an average delay of S_o, to unblock the local thread, which returns to work. Figure 4-2 shows the control flow for a complete request (without any contention).

The LoPC model calculates the runtime of an application from the parameters derived in Section 3. These include the algorithm specific parameters, n and W and the architecture specific parameters, S_l, S_o and P. Given the average computation time between requests, W, and the total number of requests, n, LoPC simply derives R, the response time of a

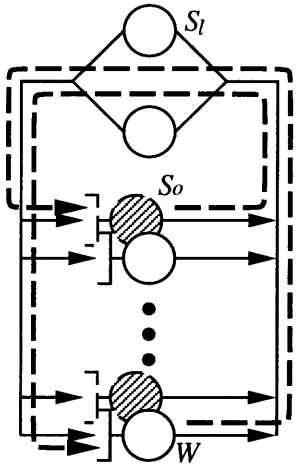

Figure 4-1: Queueing Model

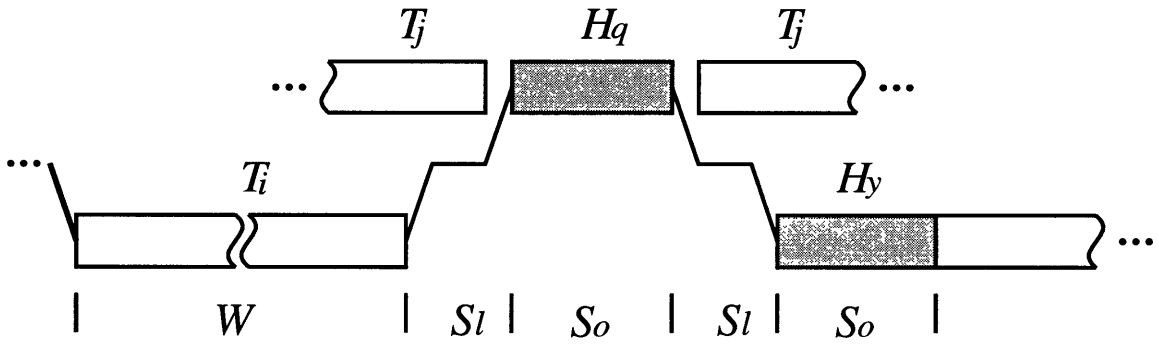

Figure 4-2: Timeline for a (Contention Free) Request

V	Portion of request handlers that run on a particular node
R_q	Response time of high-priority request handlers
R_y	Response time of high-priority reply handlers
R_w	Response time of computation thread
S_o	Average service requirement for a handler
S_l	Average latency in the interconnect
W	Average service requirement between blocking requests
Q_q	Average number of requests queued at a node
Q_y	Average number of replies queued at a node
U_q	Processor utilization by requests
U_y	Processor utilization by replies
X	System throughput
C_o^2	Coefficient of variation in service time of handlers
P	Number of processors in the system
P_s	Number of processors acting as servers
P_c	Number of processors acting as clients

Table 4.1: Notation. In general, terms related to request handlers are subscripted with a q and terms related to reply handlers are subscripted with a y

complete compute/request cycle, including the cost of contention, to get the total application runtime, nR. Contention is suffered by the computation thread, T_i, because of interference from request handlers, which have higher priority. The request and reply handlers, H_q and H_y, suffer contention delays due to queueing while other handlers complete.

To predict the costs of contention for processor resources we follow the general techniques of Mean Value Analysis. The key idea is that the average queue length at any node can be derived from the average response time at that node, while the average response time can be derived from the average queue length. From the system and algorithm parameters of the LogP model we can derive a system of equations, the solution of which gives the total running time of the algorithm. Our notation and presentation largely follows the derivations in [14].

We begin our analysis by breaking down the total average response time, R, of a *compute/request cycle*. The cycle starts with the cost, R_w, of servicing, T_i, the sending thread. Then the thread makes a request, which suffers a contention free delay of S_l for latency in the interconnect. Next the request handler, H_q, arrives at a remote node where the response time (cost of service plus any queueing) is given by R_q. Finally, a reply message is sent back through the interconnect, again with delay S_l, and arrives back at the home

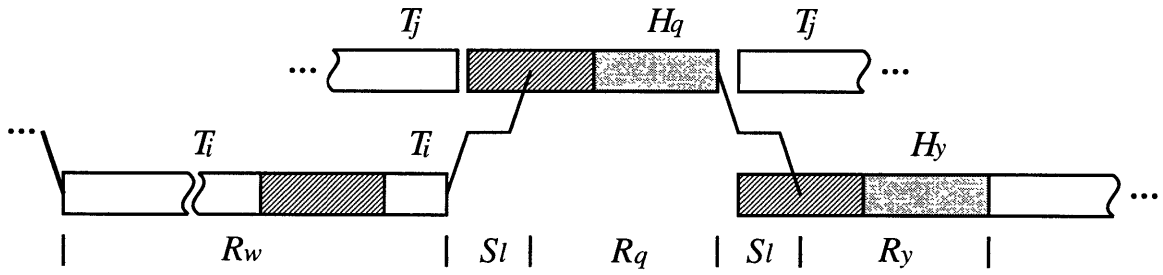

Figure 4-3: Timeline of a compute/request cycle including contention

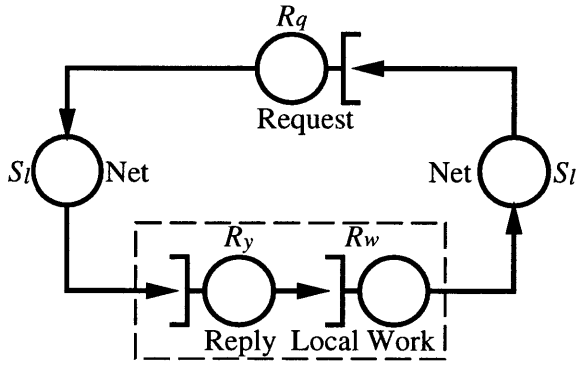

Figure 4-4: Breakdown of a compute/request cycle

node where a reply handler, H_y, is run at a cost of R_y. Figure 4-3 shows the breakdown of the compute/request cycle including the costs of contention. Total average response time, R, is given by:

$$R = R_w + S_l + R_q + S_l + R_y \tag{4.1}$$

Figure 4-4 shows a pictorial representation of the compute/request cycle. The total response time consists of four main parts. First is the residence time, R_w, of running the computation thread, including interference from requests made by other nodes. Next there is the delay, S_l, of two trips through the interconnect, once for the request and once for the reply. Next is the residence time for the request, R_q, which includes the overhead of waiting for the completion of any handlers that might already be queued. Finally there is the cost of running the reply handler, R_y, including the overhead of waiting for the completion of any handlers that might already be queued. Once we have calculated R, the response time per compute/request cycle, we can calculate the total application runtime by multiplying R by n, the total number of requests made by each thread.

To estimate the residence time at processor resources, we follow the general techniques of Mean Value Analysis [18]. Mean Value Analysis relies on Little's result, which states that for any queueing system the average number of customers in the system is equal to the product of the throughput and the average residence time in the system. In the LoPC model equations, we will most often use Little's result in the form $N = XR$, where N is the number of threads in a particular system or subsystem, X is throughput and R is the average response time for any particular thread. Little's result is very general, and makes no assumptions about scheduling discipline, maximum queue length, specific service time distributions, or the behavior of external system components. We use Little's result to calculate the utilization of each node in the system, to find the average number of messages waiting for service at each node and to compute the total system throughput.

The key element of Mean Value Analysis, the Arrival Theorem [13, 20], claims that for a broad class of queueing networks the average queue length observed by an arriving customer is equal to the average steady state queue length of a network with one fewer customers. To remove this recursion on the number of customers in the system, we use an approximation to the arrival theorem, due to Bard [2], which states that the average queue length at request arrival time is approximately equal to the average queue length. This approximation will slightly overestimate the average observed queue lengths and response times, and underestimate throughput. This error diminishes asymptotically as N increases. The key advantage of Bard's approximation is that its simplicity allows us to derive several simple and useful rules of thumb for contention costs.

Appendix A gives the LoPC model in its general form, including the ability to model "multi-hop" requests. In the next two sections we take advantage of the simplicity of Bard's approximation to derive simple closed form solutions for two important special cases of the LoPC model. In Section 5 we derive tight bounds on the contention costs of homogeneous all-to-all communication patterns. In Section 6 we give a simple and accurate closed form expression for the optimal number of servers in a client-server algorithm.

Chapter 5

All-to-All Communication

The general LoPC model, shown in detail in Appendix A, produces a system of equations that can be solved numerically. In this section we show that for an important special case, homogeneous all-to-all communication, we can make use of the homogeneity to simplify the model and derive very tight bounds on the cost of contention. We walk through the derivation step-by-step as an example of how to perform a LoPC analysis in general. In Section 6 we show how to use LoPC to derive the optimal distribution of clients and servers for a work-pile algorithm.

As demonstrated in Section 4, any LoPC analysis begins by deriving the total response time, R, for a compute/request cycle in terms of its subcomponents, R_w, the response time for the compute thread, S_l, the network latency, R_q, the request handler response time, and R_y, the reply handler response time. (See Equation 4.1.) In this section we will show how to derive each of these subcomponents. Although the situation is simplified somewhat, because we take advantage of the system's homogeneity, the same analysis can be applied to any communication pattern.

The next section goes through the analysis in detail. Section 5.2 explains how we model arbitrary service time distributions. Finally Section 5.3 gives results.

5.1 The LoPC Model

We begin by calculating the total system throughput, X, by Little's result, given the total response time for a compute/request cycle, R. There are P threads, each of which is making a request per time R so:

$$X = \frac{P}{R} \tag{5.1}$$

This is the throughput of the system as a whole. We will denote the throughput directed to any particular node as VX where V is the fraction of total throughput directed to the node. In a homogeneous algorithm the traffic is evenly divided between the nodes.

$$V = \frac{1}{P} \tag{5.2}$$

Again by Little's result, we calculate the queue length, Q_k of high-priority handlers on any node k.

$$Q_k = VXR_k \tag{5.3}$$

Likewise, the utilization, U_k of any node by either a request or reply handler can be calculated as:

$$U_k = VXS_o \tag{5.4}$$

We can calculate the response time, R_k, of an individual request handler at a given node k by noting that the response time is given by the cost of servicing this request plus all the requests that were in the network queue when this request arrived. By Bard's approximation to the Arrival Theorem [20], however, we can approximate the queue length at arrival time by the steady state queue length.

For request handlers we take into account the contention caused by both other request handlers and reply handlers. The average response time is given by:

$$R_q = S_o(1 + Q_q + Q_y) \tag{5.5}$$

Since only one thread is assigned to each node, only one reply message can queue at any given node, so we only need to account for contention caused by requests.

$$R_y = S_o(1 + Q_q) \tag{5.6}$$

Finally, we model the response time, R_w, for the computation threads by using the preempt resume (BKT) priority approximation [4, 5, 9]. We use the BKT approximation because, for our purposes, it is more accurate than the simpler shadow server approximation. We are unable to use the Chandy-Lakshmi priority approximation [4, 9], which is often more accurate than BKT, because it requires information about queue lengths in a system with $P - 1$ customers.

Only one computation thread is assigned to each node, so R_w includes no queueing delay for interference from other computation threads. In addition the computation thread runs only when the reply handler finishes, so there is no interference from reply handlers. The high-priority request handlers do, however, interfere with the computation thread.

First, when the reply handler finishes there may be additional requests queued. Since these have higher priority than the computation thread they will run first. In addition, once the computation thread does resume, additional request messages may arrive, interrupting the computation thread. The BKT approximation models this interference as:

$$R_w = \frac{W + S_o Q_q}{1 - U_q} \tag{5.7}$$

Modeling Shared Memory A shared memory machine can be thought of as a message passing system with special hardware, sometimes called a *protocol processor* to handle requests and replies. In such a system request handlers will not interfere with computation threads. In essence shared memory systems introduce an extra degree of parallelism into each node so that request handler processing can proceed simultaneously with computation. In this case we simply model R_w as W.

The rest of the shared-memory model remains unchanged from the message-passing model. In particular, request handlers still contend with each other for protocol processor resources and reply handlers still suffer queueing delays from request handlers.

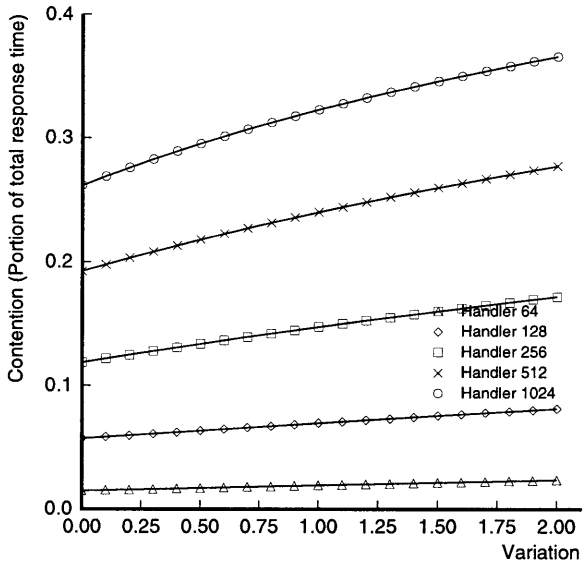

Figure 5-1: Effect of Coefficient of Variation on Contention, $W = 1000$

The set of Equations 4.1 through 5.7 completely characterize the contention suffered by a homogeneous all-to-all communication pattern. Section 5.3 discusses the solution of this non-linear system. The next section explains how to extend LoPC to deal with non-exponential service time distributions.

5.2 Modeling Uniform Service Time Distributions

The model presented above assumes exponential distributions. Our experience, however, is that most handlers consist of relatively short instruction streams with homogeneous cache behavior across invocations and few, if any, branches. For many applications then, the service time distributions for handlers will be much closer to a constant distribution. This section discusses how to extend the model to account for arbitrary service time distributions.

Suppose the service time distribution for handlers has a squared coefficient of variation given by C_o^2. Then if a message arrives at an arbitrary node k there is a probability, U_k, given by the utilization, that it will find a handler currently in service at that node. The *residual life*, of that handler will be given by $\frac{1+C_o^2}{2} S_o$. When a message arrives at a queue, it is delayed by the residual life of the first message in the queue and the full service time of the rest of the handlers in the queue. The total delay caused by the handlers queued when

a message arrives at a processor, k, can then be given as:

$$S_o(Q_k - U_k + \frac{1 + C_o^2}{2} U_k) = S_o(Q_k + \frac{C_o^2 - 1}{2} U_k) \tag{5.8}$$

We then modify the response time equations as follows:

$$R_q = S_o(1 + Q_q + Q_y + \frac{C_o^2 - 1}{2}(U_q + U_y)) \tag{5.9}$$

$$R_y = S_o(1 + Q_q + \frac{C_o^2 - 1}{2} U_q) \tag{5.10}$$

Interestingly, the equation for R_w does *not* change. This is because the thread restarts *exactly* at the point when the high-priority reply handler finishes. The thread therefore observes the complete service times of any request handlers left in the FCFS queue when the reply handler finishes.

In addition, because there is exactly one computation thread assigned to each node there is never any contention from other threads' computation. The variation in the service requirement for computation threads, therefore, does not affect the result. Likewise, in a contention free network there is never any interference between jobs so the average wire time is all we need to characterize the response time in the network.

Figure 5-1 shows LoPC's prediction of the percentage of total response time devoted to contention in a variety of homogeneous all-to-all communication patterns. In the figure W is held constant at 1000 cycles and the variation, C_o^2, is varied from 0 to 2 for a variety of possible values of handler occupancy, S_o. The difference between the values predicted for a constant distribution, $C_o^2 = 0$, and an exponential distribution, $C_o^2 = 1$, is about 6%.

5.3 Results

Solving the model given in Sections 5.1 and 5.2 requires solving a quartic equation. Typically the simplest way to do this is to use an equation solver to find a numerical solution. Here we take a different approach. We derive a recursive definition for R and then find limits on the fixed point. The result is that for a homogeneous all-to-all communication

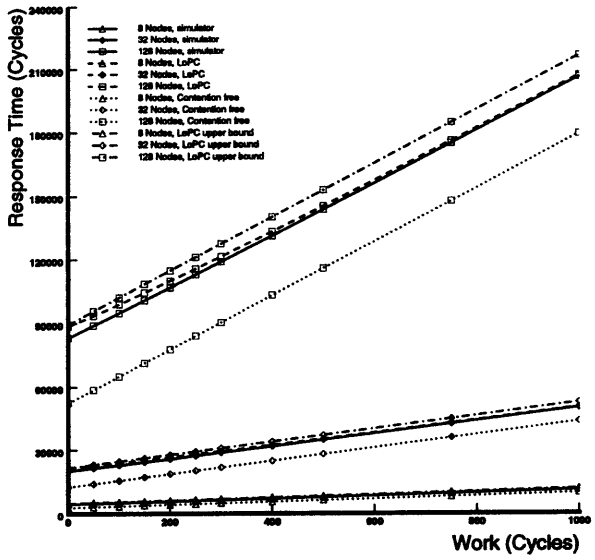

Figure 5-2: Response time of all-to-all communication with a handler time of 200 cycles, $C_o^2 = 0$

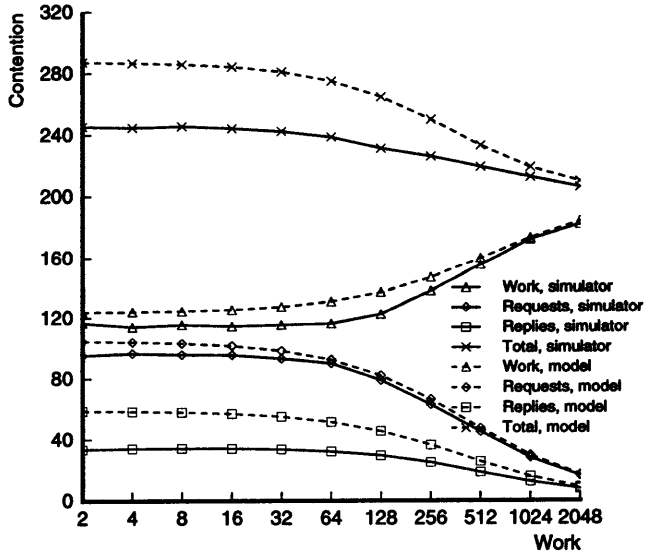

Figure 5-3: Components of contention for 32 node all-to-all communication with a handler time of 200 cycles, $C_o^2 = 0$

pattern with handlers that have little variation in service time, we can use the LoPC model to derive tight bounds on the total response time.

We derive a simple rule of thumb for the homogeneous all-to-all case when $C_o^2 = 0$. We begin by taking the AMVA model of Section 5.1 and solving for Q_q and Q_y in terms of R. Then we plug the result into the definition of R to get:

$$\begin{aligned} F[R] \;=\; & \frac{RW}{R - S_o} + 2S_l + 2S_o + \\ & \frac{5S_o^2}{2(R - S_o)} + \\ & \frac{2S_o^3}{R^2 - RS_o - S_o^2} + \\ & \frac{3S_o^4}{(R - S_o)(R^2 - RS_o - S_o^2)} \end{aligned} \tag{5.11}$$

The fixed points of $F[R]$ are the solutions of R. We note the following about $F[R]$.

- $F[R]$ is continuous and strictly decreasing when $R > W + 2S_l + 2S_o$
- $\lim_{R\to\infty} F[R] = W + 2S_l + 2S_o$

Therefor $F[R]$ has a stable fixed point at some value greater than $W + 2S_l + 2S_o$. We find further that $F[W + 2S_l + 3.46S_o] < W + 2S_l + 3.46S_o$, so

$$W + 2S_l + 2S_o < R^* < W + 2S_l + 3.46S_o \tag{5.12}$$

where R^* is the fixed point of $F[R]$.

This technique is generally applicable for arbitrary C_o^2. Only the constants will change. See Section 5.2 for more information about the effect of increased variability in handler service time.

The lower bound of this range represents the contention free cost of all-to-all communication. The upper bound represents the maximum value for the numerical solution of the LoPC model. Figure 5-2 shows these bounds along with the numerical solution to the LoPC model and the values we measured in our simulator.

Figure 5-3 shows the breakdown of contention costs for one compute/request cycle in an all-to-all communication pattern on a 32 node machine, as measured on the simulator and predicted by LoPC. To a first approximation the cost of contention is equal to the cost of an extra handler.

We can get some intuitive idea of why this should be so by looking at the cases where W is very large or very small. If W is very large then the probability is very large that a request handler arrives while the computation thread is working, so W is expanded to include an extra handler time. If, on the other hand, W is very small (say 0), and $S_l \ll S_o$ then the average queue length for handlers throughout the system is about 1 and the utilization by handlers is quite high (nearly 1). As a result an arriving handler usually has to queue for about the length of a residual life of a handler ($S_o/2$). Since each cycle requires both a request and a reply handler the cost of queueing adds another factor of S_o to the total response time.

LoPC gives a slightly pessimistic estimate of runtime. This is due to Bard's approximation, which overestimates the queue length at the time of message arrival. In the worst case, where $W = 0$, LoPC overestimates the cost of contention by 17%. Most of this error is in the contention faced by reply handlers, which LoPC over predicts by 76%. LoPC overestimates total runtime by 6% in the worst case, with the error asymptotically decreasing to 0 as the work between requests increases. In contrast, the contention free model, such as a naive application of LogP, in the worst case under predicts total run time by 37%. In addition, the total error of the contention free model (about equal to the cost of running a handler) remains constant even as the work between requests increases, so that even when $W = 1024$ the error of the contention free model is still 13%.

Chapter 6

Client-Server Communication

In this section we use the LoPC model to derive the optimal number of servers for a work-pile algorithm on a machine with P processors. As in the previous section we take advantage of application specific features to simplify the analysis. In particular, we are able to show that in the optimal case the mean queue length at the servers is equal to one. This, in addition to the observation that the clients only communicate with the servers, and that the servers never initiate a request allows us to simplify the model dramatically.

The objective of work-pile algorithms is to achieve load balance for algorithms in which there are a large number of relatively independent chunks of work to be processed and where the amount of work required to process each chunk is highly variable. The problem with work-pile algorithms is that if too few nodes are allocated as servers then the servers will become a bottleneck. If too many nodes are allocated as servers, on the other hand, then the servers won't create a bottleneck but there will be too few clients to actually do the work.

The machine is partitioned into P_c client nodes, which will actually perform the work, and $P_s = P - P_c$ server nodes which will be used to distribute work to the clients. Because the client nodes all communicate with the servers at the same average rate, and the compute threads on the servers don't communicate at all, the model of Figure 4-1 reduces to that shown in Figure 6-1. Each client node will process a chunk of work and when finished with that chunk, will request another chunk from a randomly chosen server. The system has P_c threads running (one per client), some of which will be working and some of which will be in the process of making a request for more work from one of the servers. We would like to

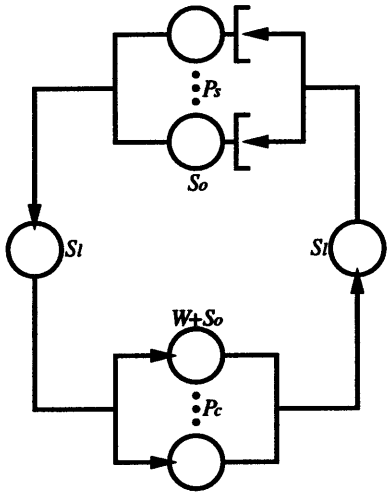

Figure 6-1: work-pile Model

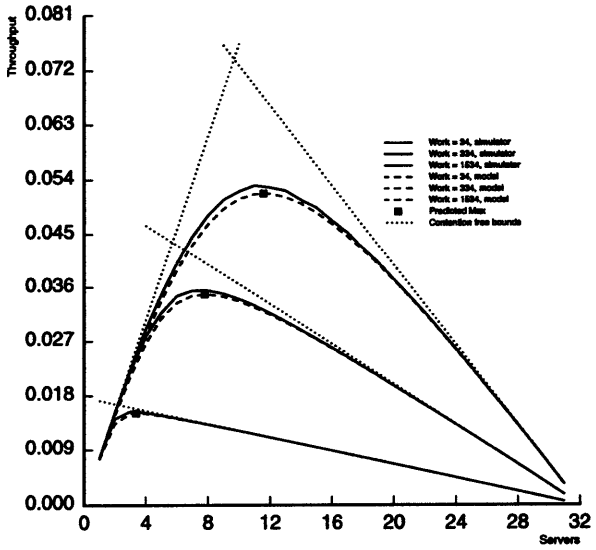

Figure 6-2: Throughput on a 32 Node machine with handler time of 131 cycles

determine the proper distribution of nodes between clients and servers such that throughput (chunks processed per unit time), is maximized. We will show that the maximum system throughput will occur with an allocation such that the average number of requests being handled by each of the servers is 1.

Suppose that there are P_s nodes working as servers and $P_c = P - P_s$ nodes working as clients. Then if there are on average only $P_s - 1$ threads requesting service, then one of the servers will, on average, be idle and we could get higher throughput if that node were acting as a client. Suppose, on the other hand, that on average $P_s + 1$ customers are at the servers. Then on average at least one customer must be waiting for service at a server that is already in use. If we reduce the total number of customers to $P_c - 1$ and increase the number of servers to P_s we will achieve higher throughput. At the optimal number of servers, then, the average number of customers at the servers is P_s and the average queue length at each individual server is $P_s/P_s = 1$.

We can use this information to find a closed form solution for the optimal number of servers given a machine with P processors, network latency S_l, handler occupancy S_o, handler variation C_o^2 and the algorithmic parameter W for the amount of work done by the client between requests. By Little's result we can calculate the queue length, Q_s, at each individual server in terms of the total system throughput, X, and the average response time at the servers, R_s.

$$Q_s = \frac{X}{P_s} R_s = 1 \Rightarrow X = \frac{P_s}{R_s} \tag{6.1}$$

Again by Little's result we can determine the total system throughput in terms of the total number of threads and the average response time, R, to process a chunk of work (including time at both the client and server).

$$X = \frac{P_c}{R} = \frac{P - P_s}{R} \tag{6.2}$$

We can now combine Equations 6.1 and 6.2 to determine the optimal number of servers in terms of average response times.

$$P_s = \frac{PR_s}{R + R_s} \tag{6.3}$$

Again by Little's result and Equation 6.1 we can determine the utilization at the servers.

$$U_s = \frac{X}{P_s} S_o = \frac{S_o}{R_s} \tag{6.4}$$

By combining the terms for utilization and queue length with Bard's approximation to the Arrival Theorem, we determine the average response time for a request at any of the servers.

$$R_s = S_o(1 + Q_s + \frac{C_o^2 - 1}{2} U_s) = S_o(2 + \frac{(C_o^2 - 1)S_o}{2R_s}) \tag{6.5}$$

This equation can be simplified by solving for R_s.

$$R_s = S_o(1 + \frac{\sqrt{2(C_o^2 + 1)}}{2}) \tag{6.6}$$

Now that we have determined the cost of a request at the servers, we can calculate the total response time of a complete compute/request cycle. This includes the cost of doing a chunk of work at the client, a trip through the network from client to server, the cost of making a request at the server, a return trip through the network and finally the cost of the reply handler at the client.

$$R = W + S_l + R_s + S_l + S_o \tag{6.7}$$

Finally by combining Equations 6.6 and 6.7 with Equation 6.3 and solving for P_s, we find the optimal number of servers.

$$P_s = \frac{P(1 + \frac{\sqrt{2(C_o^2+1)}}{2})S_o}{W + 2S_l + (3 + \sqrt{2(C_o^2 + 1)})S_o} \tag{6.8}$$

Figure 6-2 shows the predictions of this model. The throughput for an event driven simulation of a work-pile algorithm running on a 32 processor machine is shown for each combination of P_s servers from 1 to 31 and $P_c = 32 - P_s$ clients. In the worst case

LoPC predicts a value that is conservative by 3%. In addition, the black squares show the predictions of Equation 6.8.

By examining the maximum throughputs of both the clients and the servers and ignoring other contention, as we might do in a LogP analysis, we can find somewhat weaker optimistic bounds on the throughput of the work-pile algorithm. The work-pile algorithm will be server bound when the server utilization approaches 1. By Little's result this implies that $X_s \leq P_s S_o$.

At most, the client can processes one chunk of work for every compute/request cycle. The minimum time for a complete compute/request cycle, on the other hand is given by $W + S_l + S_o + S_l + S_o$, assuming that the thread suffers no contention at the server. For a system with P_c clients this means that the throughput, $X_c \leq \frac{P_c}{W+2S_l+2S_o}$. These bounds are shown in Figure 6-2 as dotted lines. These bounds are asymptotically correct, but, unfortunately, only in the range where the work-pile algorithm achieves poor parallelism.

Chapter 7

Conclusion and Future Work

LoPC is an extension to the LogP model, based on approximate mean value analysis, which permits accurate analysis of parallel algorithms with irregular communication patterns. LoPC uses the LogP architectural and algorithmic parameters to compute total application runtime including contention for processor resources. No additional parameters are required. This thesis describes the LoPC model and shows how to use LoPC to analyze the contention behavior of two common communication patterns.

For homogeneous all-to-all communication we show that the total contention costs are bounded by a small constant factor and, to a first approximation, the cost of contention is equal to the cost of an extra handler. For client-server communication we find a simple closed form expression that gives the optimal allocation of machine nodes between clients and servers. We have validated our model against an event driven simulation and shown that it produces results for response time and throughput that are accurate within 6%.

Because LoPC is both simple to use and accurately models contention costs we believe it is a tool that will be broadly applicable to studying algorithms with irregular communication patterns and architectural tradeoffs on both current and next generation parallel architectures. Although we have only validated the message passing version of the model, we have also shown how to model communication contention in shared-memory machines.

Our ongoing work with LoPC includes extending the model, using a technique pioneered by Heidelberger and Trivedi [11], to model non-blocking requests. With this extension we plan to use LoPC to evaluate architectural and cost-performance tradeoffs between shared-

memory and message-passing communication primitives.

Appendix A

The General LoPC Model

In this section we present the LoPC model in its most general form. Although we don't show any derivations of algorithms with non-homogeneous communication patterns, the technique very closely follows the analysis given in Section 5. The main difference is that while in that derivation we were able to take advantage of the inherent similarity between threads to simplify the analysis, here we must derive a complete set of equations for each individual thread.

We are given a system with P processors, each of which has a thread assigned to it. For each thread c (the thread assigned to processor c) we are given that the thread requires W_c service on the local processor and then makes a blocking request. The processor will require, on average, a fraction of service V_{ck} at each node k. Note that in general we permit $\sum_{k=1}^{P} \geq 1$, so we can easily model communication patterns that require "multi-hop" requests.

By Little's result we can determine the throughput of each thread c as:

$$X_c = \frac{1}{R_c} \qquad c = 1, \ldots, P \tag{A.1}$$

Where R_c is the average response time for thread c.

In addition, we can find the average throughput for each thread c through each node k as:

$$X_{ck} = V_{ck} X_c \qquad c, k = 1, \ldots, P \tag{A.2}$$

Again by Little's result we can determine, for each node, k, the utilization of that node by request handlers.

$$U_{qk} = S_o \sum_{c=1}^{P} X_{ck} \qquad k = 1, \ldots, P \tag{A.3}$$

And similarly, we can find the utilization of each node, k, by reply handlers.

$$U_{yk} = X_k S_o \qquad k = 1, \ldots, P \tag{A.4}$$

Once again by Little's result we can find the average queue lengths on each node, k, of request and reply handlers

$$Q_{qk} = R_{qk} \sum_{c=1}^{P} X_{ck} \qquad k = 1, \ldots, P \tag{A.5}$$

$$Q_{yk} = X_k R_{yk} \qquad k = 1, \ldots, P \tag{A.6}$$

Next, by Bard's approximation to the arrival theorem we calculate the average response times for request and reply handlers at each node from the average queue lengths at the node.

$$R_{qk} = S_o(1 + Q_{qk} + Q_{yk}) \qquad k = 1, \ldots, P \tag{A.7}$$

$$R_{yk} = S_o(1 + Q_{qk}) \qquad k = 1, \ldots, P \tag{A.8}$$

And by the BKT approximation, combined with Bard's approximation (more of which is described in Section 5.1), we can calculate the response time for each computation thread.

$$R_{wk} = \frac{S_o Q_{qk} + W_k}{1 - U_{qk}} \qquad k = 1, \ldots, P \tag{A.9}$$

LoPC can model machines with protocol-processor support by avoiding modeling contention between handlers and the computation threads by instead using $R_{wk} = W_k$.

Finally, we put all the parts together to arrive at the total response time for a compute/request cycle. Note that this is slightly more complicated than the derivation shown in Section 4 to account for the possibility of requests that require multiple hops through the network.

$$R_c = R_{wc} + \sum_{k=1}^{P} V_{ck}(S_l + R_{qk}) + S_l + R_{yc} \qquad c = 1, \ldots, P \tag{A.10}$$

In addition, the model can be extended in a straightforward way to deal with handler service time distributions other than exponential. For details see Section 5.2.

Bibliography

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[2] Yonathan Bard. Some Extensions to Multiclass Queueing Network Analysis. In M. Arato, A. Butrimenko, and E. Gelenbe, editors, *Performance of Computer Systems*. North-Holland, 1979.

[3] Eric A. Brewer and Bradley C. Kuszmaul. How to Get Good Performance from the CM-5 Data Network. In *Proceedings of the 1994 International Parallel Processing Symposium*, April 1994.

[4] Raymond M. Bryant, Anthony E. Krzesinski, M. Seetha Lakshmi, and K. Mani Chandy. The MVA Priority Approximation. *ACM Transactions on Computer Systems*, 2(4):335–359, November 1984.

[5] Raymond M. Bryant, Anthony E. Krzesinski, and P. Teunissen. The MVA Pre-empt Resume Priority Approximation. In *Proceedings of the 1983 ACM Sigmetrics Conference*, pages 12–27, 1983.

[6] Derek Chiou, Boon S. Ang, Arvind, Michael J. Beckerle, Andy Boughton, Robert Greiner, James E. Hicks, and James C. Hoe. StarT-NG: Delivering Seamless Parallel Computing. In *Proceedings of the EURO-PAR '95*, pages 101–116, Stockholm, Sweden, August 1995.

[7] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the 4th Symposium on Principles and Practices of Parallel Programming*, pages 1–12, May 1993.

[8] Andrea C. Dusseau. Modeling Parallel Sorts with LogP on the CM-5. Masters Report UCB/CSD-94-829, UC Berkeley Computer Science Department, May 1994. To appear in IEEE Transactions on Parallel and Distributed Systems.

[9] Derek L. Eager and John N. Lipscomb. The AMVA Priority Approximation. *Performance Evaluation*, 8(3):173–193, June 1988.

[10] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine Multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Ann Arbor, MI, 1995.

[11] Philip Heidelberger and Kishor S. Trivedi. Queueing Network Models for Parallel Processing with Asynchronous Tasks. *IEEE Transactions on Computers*, C-31(11):1099–1109, November 1982.

[12] Chris Holt, Mark Heinrich, Jaswinder Pal Singh, Edward Rothberg, and John Hennessy. The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors. Technical Report CSL-TR-95-660, Stanford Computer Systems Laboratory, January 1995.

[13] S.S. Lavenberg and M. Reiser. Stationary State Probabilities of Arrival Instants for Closed Queueing Networks with Multiple Types of Customers. *Journal of Applied Probability*, December 1980.

[14] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantatative System Performance, Computer System Analysis Using Queueing Network Models*. Prentice Hall, 1984.

[15] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, Daniel Hillis, Bradley C. Kuszmal, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The Network Architecture of the Connection Machine CM-5. *The Journal of Parallel and Distributed Computing*, 33(2):145–158, March 1996.

[16] Gary Lewandowski. LogP Analysis of Parallel Branch and Bound Communication. *Submitted to IEEE Transactions on Parallel and Distributed Systems*, April 1994.

[17] Kenneth Mackenzie, John Kubiatowicz, Matthew Frank, Walter Lee, Anant Agarwal, and M. Frans Kaashoek. UDM: User Direct Messaging for General-Purpose Multiprocessing. Technical Memo MIT-LCS-TM-556, MIT Laboratory for Computer Science, March 1996.

[18] M. Reiser and S.S. Lavenberg. Mean Value Analysis of Closed Multichain Queueing Networks. *Journal of the ACM*, 27(2):313–322, April 1980.

[19] Steven L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[20] K. C. Sevcik and I. Mitrani. The Distribution of Queuing Network States at Input and Output Instants. *Journal of the ACM*, 28(2):358–371, April 1981.

[21] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[22] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Shauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992. ACM Sigarch.