LoPC: Modeling Contention in Parallel Algorithms

Matthew I. Frank Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology
{mfrank,agarwal}@lcs.mit.edu

Mary K. Vernon
Computer Sciences Department
University of Wisconsin-Madison
vernon@cs.wisc.edu

Abstract

Parallel algorithm designers need computational models that take first order system costs into account, but are also simple enough to use in practice. This paper introduces the LoPC model, which is inspired by the LogP model but accounts for contention for message processing resources in parallel algorithms on a multiprocessor or network of workstations. LoPC takes the L, o and P parameters directly from the LogP model and uses them to predict the cost of contention, C.

This paper defines the LoPC model and derives the general form of the model for parallel applications that communicate via active messages. Model modifications for systems that implement coherent shared memory abstractions are also discussed. We carry out the analysis for two important classes of applications that have irregular communication. In the case of parallel applications with homogeneous all-to-any communication, such as sparse matrix computations, the analysis yields a simple rule of thumb and insight into contention costs. In the case of parallel client-server algorithms, the LoPC analysis provides a simple and accurate calculation of the optimal allocation of nodes between clients and servers. The LoPC estimates for these applications are shown to be accurate when compared against event driven simulation and against a sparse matrix computation on the MIT Alewife multiprocessor.

1 Introduction

Light-weight user-level message passing paradigms, like Active Messages [33], 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. In this paper we are particularly interested in algorithms that are loosely synchronized or that have irregular communication patterns, including hash algorithms, client-server applications, 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.

The LogP model [10] has been successful at accurately modeling and optimizing tightly synchronized 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.

In *Proceedings of the SIXTH ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*,
Las Vegas, Nevada, June 18th-21st, 1997

There is evidence 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. For example, Dusseau *et al* used LogP to analyze a variety of sorting algorithms with irregular communication patterns [11]. They found that some of their models underestimated execution time and attributed the difference to contention costs. Furthermore, Holt *et al* [18] 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.

Regular communication patterns can also demonstrate contention. Brewer and Kuszmal [5] measured the communication costs in 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 regular all-to-all communication on the CM-5 unless extra barriers are inserted to resynchronize the communication pattern. However, low-latency barriers like those on the CM-5 are very expensive relative to other hardware components [28]. Few, if any, current generation multiprocessors or NOWs implement this feature.

The goal of this paper is to create a new model for analyzing parallel algorithms, LoPC, that provides algorithm running times that include accurate predictions of contention costs. LoPC is inspired by LogP and, like LogP, is motivated by Valiant's observation [30] 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 use the parameters required for a LogP analysis (network latency, message processing overhead and number of processors) to analyze a queueing model to calculate execution time inflation due to contention. The challenge is to generate an accurate yet simple and efficient analysis that yields *insight* into algorithm behavior and trade-offs. In particular, the questions addressed in this research are:

1. Can we develop an approximate model that is sufficiently accurate yet also yields insights about contention for message processing resources?
2. How significant is the contention in important classes of algorithms?
3. What insights can we obtain about the contention or the algorithms?

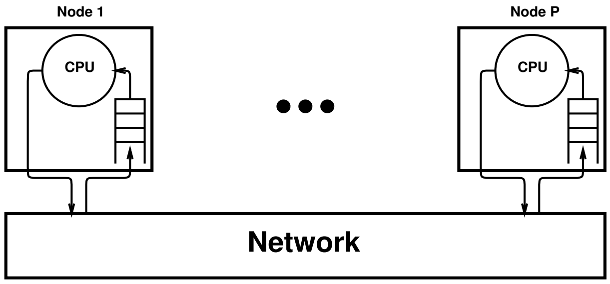

Figure 1: Parallel Architecture with Active Message Communication

We illustrate the LoPC model for two important classes of parallel algorithms: applications with homogeneous all-to-any communication, and homogeneous client-server applications. The former class includes dense and sparse matrix computations, and in that case the LoPC model is validated against a sparse matrix application running on the MIT Alewife multiprocessor. The LoPC models for both classes of algorithms are also shown to be accurate by comparing against results from detailed event driven simulations.

Using the LoPC model we derive a number of interesting insights about the costs of contention in parallel applications that are not tightly synchronized. For example, for algorithms with homogeneous all-to-any communication patterns we derive tight bounds on the total cost of contention and find that the cost of contention per message is approximately equal to the cost of processing an extra message. Thus, we are able to develop a simple rule of thumb to accurately predict the run time of this broad class of algorithms. Furthermore, the LoPC queueing model is itself both simple and computationally efficient, and can be used to compute algorithm run times in more general cases.

In the case of homogeneous client-server applications, such as workpile algorithms, the LoPC analysis allows us to find an optimal allocation of nodes between clients and servers.

Because LoPC is both simple to use and accurately models contention costs, we believe it is a tool that could be broadly applicable to studying algorithmic and architectural tradeoffs on both current and next generation parallel architectures. Given a small number of parameters that represent the algorithm and architecture under study, a numerical solution of the LoPC equations generates the runtime of the application including the cost of contention for processing resources.

The next section discusses the architectural assumptions in the LoPC model. Section 3 describes the parameters and analysis of the LoPC model. Section 4 contains a complete LoPC analysis for algorithms with homogeneous all-to-any communication and derives bounds on the total cost of contention. Section 5 uses LoPC to find the optimal allocation between clients and servers in homogeneous client-server applications. Section 6 discusses related work. Section 7 concludes. A detailed description of the general LoPC model is contained in Appendix A.

2 Architectural Assumptions

The systems modeled by LoPC consist of a set of processing nodes each with an interface to a high speed interconnect, (see Figure 1.) Each node may send a message to any other node. The message contains a pointer to a handler and some amount of data (typically around eight words for the systems modeled in this paper). When that message arrives at the destination node, it interrupts the running job. The destination processor atomically runs the handler 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. The Alewife machine [2], used to validate the analyses in this paper, provides hardware network input queues that can hold up to 512 bytes of data.

Parameter	Description
n	Number of requests made by each processor
W	Average local work per request
V_{ij}	Fraction of messages from node i directed to node j

Table 1: Algorithmic Parameters

This type of communication model using messages, called *Active Messages* [33], is general enough to implement more complex communication and synchronization protocols, which we believe makes it a good basis for algorithm analysis in modern parallel systems. A number of existing machines provide efficient support for communication with active messages [2, 9, 14, 20, 24, 28, 32].

We make two further assumptions in the LoPC model that appear to result in very little loss in accuracy yet great gain in simplicity. First, we assume that the hardware message buffers at the nodes are infinitely large. Second, we assume that the interconnect is contention free. We model contention only for message processing resources in the processor nodes. The model can be extended to include analysis of contention in the network, as noted in Section 3.2. However, a number of researchers have found that for many real applications contention in current interconnection networks accounts for only a minimal portion of total runtime [10, 18, 26]. Furthermore, for the algorithms investigated in this paper we found that network contention is insignificant.

To simplify the explanation of the LoPC model, the analyses in this paper will assume: 1) a single CPU per node, 2) a single computation thread per CPU, 3) message handlers run on the processor (interrupting local computation), 4) message send (i.e., *put*) operations block for a reply and 5) messages are relatively short (around eight words for the algorithms discussed in this paper). The model can be generalized in fairly straightforward ways to include multiple CPUs per node, message co-processors, and asynchronous send operations. Alexandrov *et al* have extended LogP to model much longer messages [3] and we believe that such extensions can also be incorporated into LoPC. Validating such models is the subject of future research.

The next section gives an example of how the LoPC model is parameterized and explains how the analysis is carried out.

3 The LoPC Model

LoPC extends the LogP parameters and analysis to calculate the average cost of contention, C. In this section we describe the LoPC model parameters and then the analysis that yields the algorithm execution time including contention costs.

3.1 Parameterization for LoPC

The parameters of the LoPC model are very similar to the parameters for the LogP model, with some simple extensions that enable calculation of message processing contention costs. The parameters involve both an algorithmic characterization and an architectural characterization. The model predicts total application run times from these two characterizations. Both of these parameterizations are discussed below.

3.1.1 Algorithmic Parameters

Algorithmic characterization for either LogP or LoPC analysis starts with specifying the total number of arithmetic and communication operations performed by the algorithm as a function of the problem size and the number of processors. 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 sparse matrix-vector multiply routine. The algorithmic parameters for the model are shown in Table 1.

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 also cyclically distributed, as shown in Figure 2. We wish to multiply $A \times x$ to produce the vector y, again cyclically distributed. Each processor will be responsible for the N/P dot products corresponding to the rows of A that are assigned to it.

To compute the dot product of row A_i with x, the processor requires the value of the element x_j corresponding to each non-zero value A_{ij}. We will assume that these values are requested with *get* operations. A message is sent to the remote node with the address of the required value. The handler on the remote node loads the value from memory and sends it in a message back to the originator. The node that originates the request is blocked until the reply message returns. We expect that, on average, $1/P$ of the required values from x will be assigned locally. For these values no remote request operation is required.

Given the number of non-zero values, b, in the matrix A the average amount of work done by each node consists of $m = b/P$ multiply-add operations and $n = b/P \times (P-1)/P$ *get* operations[1]. The average local work done between remote requests, W, is equal to $\frac{m}{n} = P/(P-1)$ multiply-add operations. These are exactly the quantities required to parameterize the LogP model.

The LoPC model requires one further set of parameters, namely the fraction of messages from node i that are directed to node j, V_{ij}. For this algorithm, $V_{ij} = 1/(P-1)$ for all i and j. Using these values, the LoPC model will calculate the average run time of the algorithm, including the costs of contention for message processing resources.

3.1.2 Architectural Parameters

The architectural parameters used by the LoPC model are very similar to those used by the LogP model, as shown in Table 2. In both models P represents the number of processors in the system.

The average service time in the network, S_l, corresponds exactly to the network latency parameter, L in the LogP model. This is the time that the message spends in the interconnect between the completion of message injection by the processor, and the arrival of the message at the remote node. It does not include any processing costs for the message. The processor overheads for handling messages are covered in other parameters.

[1]These calculations assume that the non-zero elements are approximately evenly distributed among the processors, as expected for many large sparse matrix computations. The parameters would be modified in the obvious way for uneven distribution of the work.

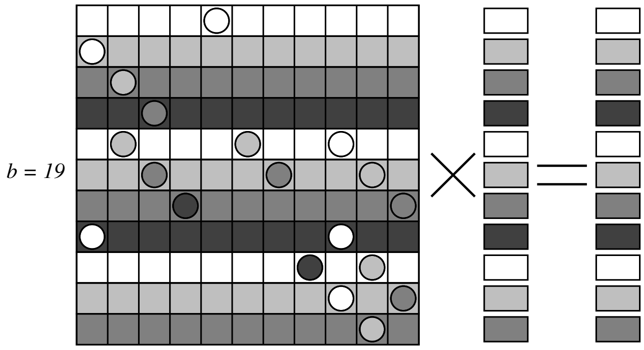

Figure 2: Sparse Matrix-vector Multiply, matrix A cyclically distributed among four nodes; circles indicate non-zero elements of A; colors indicate the node assignments.

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

Table 2: Architectural Parameters of the LoPC Model. An optional parameter, C_o^2, can be used to specify variability in message processing time.

LoPC's S_o parameter corresponds approximately to the o parameter in the LogP model in that it measures message processing overhead. The LogP model assumes a polling model with relatively expensive sends; thus the o parameter in LogP represents both the cost of a send and the cost of processing an incoming message. In contrast, 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 LoPC analyses in this paper assume that message send operations have zero cost. The model can easily be extended to include non-negligible send cost.

We have not included LogP's g parameter in LoPC. The g, or "gap" parameter represents the maximum rate at which a processor can inject messages into the network. Most current generation machines are *balanced*, in the sense that they can accept new messages into the network as fast as the processor can compose them. If, in the future, network interface bandwidth again becomes a bottleneck (as on the CM-5), the g parameter could be added back to the LoPC model, but we did not find it useful for the analyses and validations in this paper.

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

In summary, the parameterization of the LoPC and LogP models

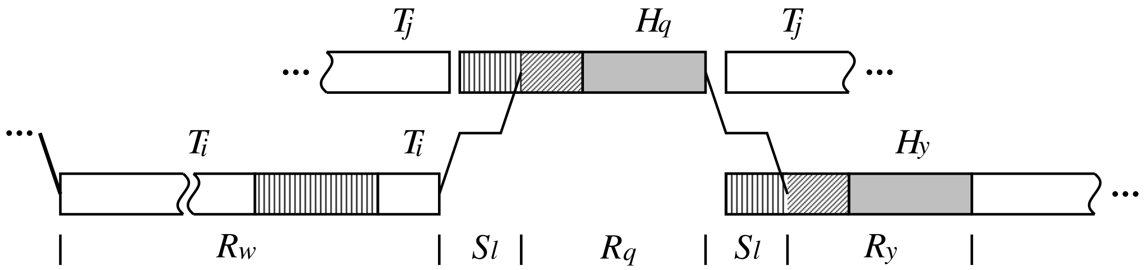

Figure 3: Timeline of a Compute/request Cycle, Including Contention.

is very similar. LoPC has a somewhat different interpretation of the o architectural parameter, the additional visit count parameters, V_{ij}, and the optional coefficient of variation in request processing time, C_o^2.

3.2 The LoPC Analysis

The LoPC analysis assumes a message passing machine with P nodes that can communicate through a high-speed interconnect. Each node, i, runs a thread T_i. 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.

The LoPC model calculates the runtime of an application, including the cost of contention for message processing resources, from the parameters specified in Section 3.1. These include the algorithm specific parameters, n, W, and V_{ij}, 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, the LoPC analysis derives R, the mean round trip time of a complete compute/request cycle, to get the total application runtime, nR.

The contention delays are computed using Approximate Mean Value Analysis. LoPC generates a system of equations that can be solved to produce the total runtime of the algorithm including processor contention. This section discusses the general form of those equations, which in the general case are solved numerically. However, the approximations permit dramatic simplification of the system of equations for important special cases. For example the equations reduce to "rules of thumb" for algorithms with homogeneous all-to-any communication and for homogeneous client-server systems, as shown in sections 4 and 5.

The LoPC analysis begins with breaking down the average round trip time, R_i, of a *compute/request cycle* at each node i. The timeline for a compute/request cycle for a given thread, T_i, is shown in Figure 3. The cycle starts with the cost, R_{wi}, of running T_i, including the average local work W plus the cost of handling interrupting requests that have higher priority. Section 3.1.1 discusses how to derive the parameter W for the algorithm being modeled. After the compute time W is complete, the thread makes a blocking request to some other node and begins waiting for a reply. 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.

At the point when a request arrives at node j, it waits for the completion of any handlers that might already be queued, and then interrupts the thread, T_j, running on the destination node to run a high-priority re**q**uest handler, H_q, for an average delay of S_o including the cost of taking the interrupt. The mean response time at the remote node, including average time for handler processing plus mean queueing delay, is denoted by R_{qj}. 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 queues behind waiting requests and then interrupts the processor to run a high-priority repl**y** handler, H_y, for an average delay for queueing plus service of R_{yi}. The reply handler unblocks the local thread, which returns to work.

Thus, the total average round trip time for a compute/request cycle, R_i, is given by:

$$R_i = R_{wi} + S_l + \sum_j V_{ij} R_{qj} + S_l + R_{yi} \tag{1}$$

Figure 4 shows a pictorial representation of the queueing delays in the compute/request cycle. Assuming a load-balanced application, once we have calculated R_i, the average round trip time for a compute/request cycle of a thread, we can calculate the expected total application runtime by multiplying R_i by n, the total number of requests made by the thread[2].

To predict the interference and queueing delays at processor resources (i.e., to compute R_{wi}, R_{qj}, and R_{yi}), we follow the general techniques of Approximate Mean Value Analysis (AMVA). The notation is given in Table 3. Appendix A derives the equations in their general form, including the ability to model "multi-hop" requests. Due to careful selection of approximations in the analysis, the equations simplify for important classes of algorithms that have homogeneous threads. For example, sections 4 and 5 derive the simpler equations that result in *closed form solutions* for two such classes of algorithms. Section 4 derives tight bounds on the contention costs of algorithms with homogeneous all-to-any communication, and Section 5 derives a simple and accurate closed form expression for the optimal number of servers in a client-server algorithm. Below we discuss one of the MVA approximations that make the closed form expressions possible.

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 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 round trip time for a 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,

[2] Load-imbalance and possible associated synchronization costs can be accounted for in the calculation of total application runtime using ad hoc techniques that have been proposed in the performance modeling literature. Discussion and application of such techniques is beyond the scope of this paper.

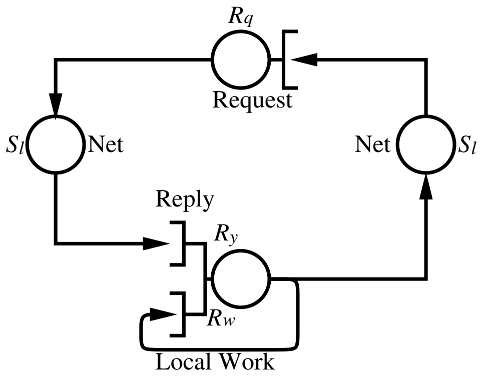

Figure 4: Queueing delays in a Compute/request Cycle

R	Average round trip time for a compute/request cycle
X	System throughput
R_q	Average response time of high-priority re**q**uest handler
R_y	Average response time of high-priority repl**y** handler
R_w	Average residence time of computation thread
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

Table 3: Notation. Terms related to re**q**uest handlers have subscript q; terms related to repl**y** handlers have subscript y.

to find the average number of messages waiting for service at each node and to compute the total system throughput.

The cornerstone of Mean Value Analysis, the Arrival Theorem [19, 29], states 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 in a network with the arriving customer removed. In a queueing network with multiple classes of customers, like LoPC, where each thread is represented by a different customer class, the recurrence is complex, requiring a recursive solution that is exponential in the number of threads.

To remove this recursion on the number of customers in the system, we use an approximation to the arrival theorem, due to Bard [4], which assumes that the average queue length at request arrival time is approximately equal to the average queue length. This approximation will overestimate the average observed queue lengths and response times, and underestimate throughput. However, the error diminishes asymptotically as the number of threads, 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. If more precise estimates of contention are required, the exact MVA equations, or the Bard-Schweitzer approximation, can be used instead, at the cost of reduced insight.

Due to Bard's approximation, the general form of the LoPC model, derived in Appendix A, is much less complex than the exponential system that would be given by the "exact" form of the Arrival Theorem. The resulting system of equations can be solved numerically, or for communication patterns that exhibit homogeneity the system can be further simplified to produce "rules of thumb" to guide system or algorithm design, as shown in the next two sections.

Note that neither LoPC nor LogP models contention for *network* resources. Again at the cost of additional complexity in the model, one could incorporate contention at network links or switches into LoPC. For the communication patterns studied in this paper, network contention costs are minimal compared to contention for message processing resources.

4 Algorithms with Homogeneous All-to-Any Communication

The general LoPC model, discussed in Section 3 and derived in Appendix A, produces a system of equations that can be solved numerically. In this section we consider an important special case, namely parallel algorithms with homogeneous all-to-any communication. In these algorithms each thread performs the same average amount of work per compute/request cycle and each request is equally likely to visit any of the other processors. For such algorithms, we can make use of the homogeneity and the model approximations to derive closed form expressions that tightly bound the cost of contention. We derive the equations for this special case in detail as an example of how to perform a LoPC analysis. In Section 5 we show how to use LoPC to derive the optimal allocation of nodes to clients and servers for a work-pile algorithm.

Examples of algorithms that exhibit homogeneous all-to-any communication are the load-balanced sparse matrix-vector multiplication discussed in Section 3.1, algorithms where the communication pattern is dependent on a complex graph structure, and algorithms like radix sort, where the communication pattern is dependent on the results of performing a hash on each input data element.

As discussed in Section 3.2, a LoPC analysis derives the total time, R_i, for a compute/request cycle at node i in terms of its subcomponents: R_{wi}, the average time the thread computes; S_l, the network latency; R_{qj}, the average request handler response time at each node j; and R_{yi}, the reply handler mean response time. (See equation 1.) In this section we will show how to derive each of these terms for the class of algorithms where we can drop the subscripts i and j because all nodes are statistically homogeneous. Although the equations are simplified due to the homogeneity among threads, the analysis uses the same techniques as the general model derived in Appendix A.

The next section goes through the analysis in detail. Sections 4.2 and 4.3 explain how to extend the model for shared memory communication and non-exponential distributions of message processing time, respectively. Section 4.4 compares the results of LoPC to an event driven simulation of a synthetic micro-benchmark. Finally, Section 4.5 completes the LoPC analysis of the sparse matrix-vector multiply discussed in Section 3.1 and compares results against the measured costs of running this algorithm on the MIT Alewife multiprocessor.

4.1 The LoPC Equations

We begin by calculating the total system throughput, X, in terms of the total time for a compute/request cycle, R. There is one thread per processor in the system, each with throughput $1/R$, so the total system throughput is given by:

$$X = \frac{P}{R}. \tag{2}$$

Since each node receives an equal fraction of the request messages, the rate at which request messages arrive at each node is given by X/P. Using Little's result, we can calculate the average message input queue lengths, Q_q for request messages and Q_y for reply messages, as follows:

$$Q_q = \frac{X}{P} R_q, Q_y = \frac{X}{P} R_y. \tag{3}$$

Likewise, the utilizations, U_q and U_y of a node by request and reply handlers, respectively, can be calculated using Little's result:

$$U_q = U_y = \frac{X}{P} S_o. \tag{4}$$

We can calculate the average response time for an individual request at a given node j by noting that the response time is given by the cost of servicing this request plus all the requests that were in the queue when this request arrived. By Bard's approximation to the Arrival Theorem, we approximate the queue length at arrival time by the steady state queue length. For request handlers we take into account the contention caused by other request handlers and by reply handlers, as follows:

$$R_q = S_o(1 + Q_q + Q_y). \tag{5}$$

Since only one thread is assigned to each node and all send operations are blocking operations, only one reply message can queue at any given node; thus, we only need to account for contention caused by requests:

$$R_y = S_o(1 + Q_q). \tag{6}$$

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. We compute the average total time for the thread computation including handler interrupts, R_w, by using the BKT preempt-resume priority approximation [6, 7, 13]:

$$R_w = \frac{W + S_o Q_q}{1 - U_q}. \tag{7}$$

We use the BKT approximation because it is more accurate than the simpler shadow server approximation, and it yields a simpler result than the Chandy-Lakshmi priority approximation [6, 13].

The set of equations (1) through (7) completely characterize the execution time of a compute/request cycle, including contention for processor resources, for algorithms with homogeneous all-to-any communication. The algorithm running time is computed by multiplying R by the LogP algorithmic parameter n, the average number of requests made by each thread, defined in Section 3.1.1. Section 4.4 discusses the solution of this non-linear system. The next two sections explain how to extend the LoPC analysis to deal with shared memory communication and non-exponential service time distributions, respectively.

4.2 Shared Memory Communication

A shared memory system 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.

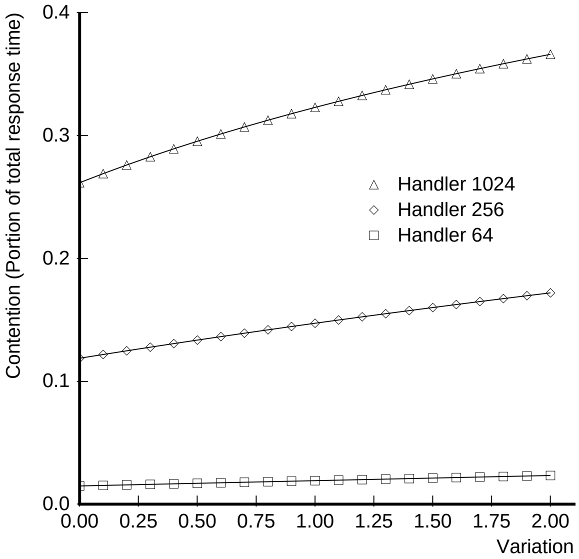

Figure 5: Fraction of Execution Time Devoted to Contention as Handler Variance Changes (P = 32, S_l = 6, W = 1000)

One other change is required if the shared memory algorithm includes non-blocking (or asynchronous) communication. Approximate MVA techniques for modeling such asynchronous communication [17] are well-known and have been employed in validated AMVA models of shared memory architectures [8, 21, 31]. Otherwise, the shared-memory model for the specified class of algorithms is the same as the message-passing model. In particular, handlers still contend with each other for protocol processor resources.

4.3 Non-exponential Message Processing Times

The analysis presented above holds for any distribution of the local computation time between requests, but assumes exponential message handler processing times. In our experience, message handlers often consist of relatively short instruction streams with similar cache behavior across invocations and few, if any, branches. Thus, for many applications the service time distributions for handlers will be much closer to a constant distribution. This section discusses how to extend the model with an approximation, due to Reiser and Lavenberg [27], to account for arbitrary handler service time distributions, with squared coefficient of variation given by C_o^2. For most systems it will be appropriate to assume either $C_o^2 = 0$ or $C_o^2 = 1$.

When a message arrives at a given node, there is a probability that it will find a handler currently in service at that node. This probability is approximated by the utilization, U_q or $U_q + U_y$, for a reply or request message arrival, respectively. The *residual life* of the handler in service is given by $\frac{1+C_o^2}{2} S_o$, which assumes a random arrival instant. The arriving message is delayed by this residual life of the message at the head of the queue, and by the full service time of the rest of the handlers in the queue. The total delay caused by the handlers queued when a reply message arrives at a node is thus:

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

We then modify the response time equations as follows:

$$R_q = S_o(1 + Q_q + Q_y + \frac{C_o^2 - 1}{2}(U_q + U_y)), \qquad (5')$$

$$R_y = S_o(1 + Q_q + \frac{C_o^2 - 1}{2} U_q). \qquad (6')$$

Note that the equation for R_w does *not* change. Since the thread restarts at the point when the high-priority reply handler finishes, the thread observes the complete service times of any request handlers left in the FCFS queue when the reply handler finishes.

Figure 5 shows the LoPC estimates of the fraction of algorithm execution time devoted to contention versus the coefficient of variation in handler processing time, C_o^2, for a variety of values of average handler occupancy, S_o. In the figure, W is equal to 1000 cycles. For handler processing times that are a significant fraction of W, the fraction of execution time devoted to contention is significant, and there is a nontrivial difference between the estimated execution time for constant message handling times, $C_o^2 = 0$, as compared to exponential handling times, $C_o^2 = 1$.

4.4 LoPC Results: Contention Cost Insight

By solving equations (3), (4), (5') and (6') for Q_q and Q_y and then plugging the results back into equations (5'), (6') and (7) we can derive equations for R_q, R_y and R_w entirely in terms of R, the total time for a compute/request cycle, and the model inputs S_o, S_l and W. If we then assume $C_o^2 = 0$ and use these derived forms of R_q, R_y and R_w in the definition of R, equation (1), we find that:

$$\begin{aligned} R \;=\; & \frac{W}{1 - S_o/R} + 2S_l + 2S_o + \\ & \frac{5S_o^2}{2(R - S_o)} + \\ & \frac{2S_o^3}{R^2 - RS_o - S_o^2} + \\ & \frac{3S_o^4}{(R - S_o)(R^2 - RS_o - S_o^2)}. \end{aligned} \qquad (8)$$

Completing the LoPC analysis thus requires solving a quartic equation. The straightforward way to proceed is to use an equation solver to find a numerical solution, as was done to generate the results in Figure 5. Here we take a different approach that leads to tight bounds on the total response time. From these bounds we will derive a simple rule of thumb for algorithm running time.

The fixed points of equation (8) are the solutions of R. We note the following about the right hand side of equation (8), $F[R]$:

- $F[R]$ is continuous and strictly decreasing in the feasible range of R, $R > W + 2S_l + 2S_o$
- $\lim_{R\to\infty} F[R] = W + 2S_l + 2S_o$

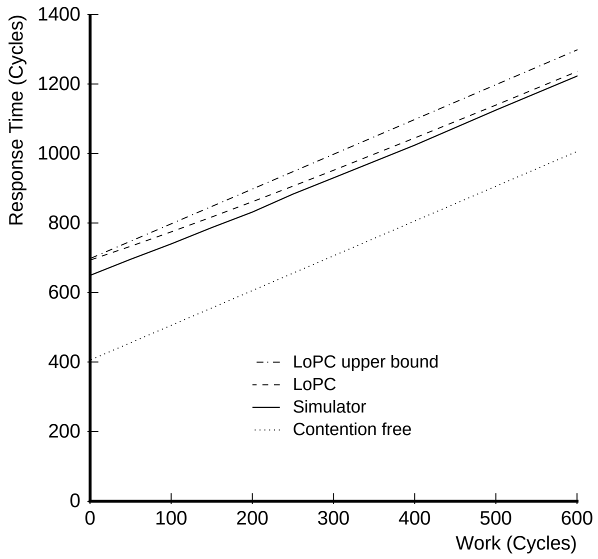

Figure 6: Runtime of a Compute/request Cycle for Algorithms with Homogeneous All-to-any Communication, $P = 32$, $S_o = 200$ cycles, $S_l = 6$, $C_o^2 = 0$.

Therefore equation (8) has exactly one fixed point that is 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

$$W + 2S_l + 2S_o < R^* < W + 2S_l + 3.46S_o \qquad (9)$$

where R^* is the desired fixed point of equation (8). This technique is applicable for arbitrary C_o^2. Only the constants will change.

The lower bound for R^* represents the contention free execution time, or LogP cost estimate, for algorithms with all-to-any communication. The upper bound represents the maximum value for the numerical solution of the LoPC model. An interesting feature of these bounds is that the upper bound on contention is equal to 1.46 times the handler processing time.

To check that the LoPC predictions are reasonable, we have built a simple event driven simulator with which to compare our results. The simulator models the state of each of the P simulated machine nodes, including the message queues. Local work and high priority handlers are modeled in the simulator as having constant cost. The network latency between any two nodes is taken to be the Manhattan distance between the two nodes given their positions on a two dimensional mesh. The destination of request messages are selected with a random number generator according to the visit counts V_{ij}.

Figure 6 shows the bounds on the running time of a compute/request cycle (R) along with the running time estimates from numerical solution of the LoPC model, and the running times measured in our simulator. The LoPC cost estimates agree closely with the simulation results. Also, as might be expected, the average running time is closer to the upper bound than to the contention-free (LogP) lower bound.

We can get some intuitive idea of why the actual cycle time is closer to the upper bound or why, to a first approximation, the total cost of contention is equal to the cost of an extra handler per

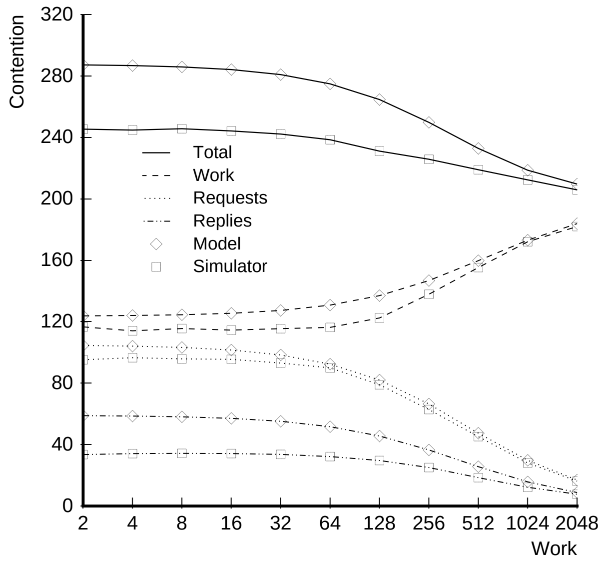

Figure 7: Contention or Interference Cost per Compute/request Cycle, $P = 32$, $S_o = 200$ cycles, $S_l = 6$, $C_o^2 = 0$

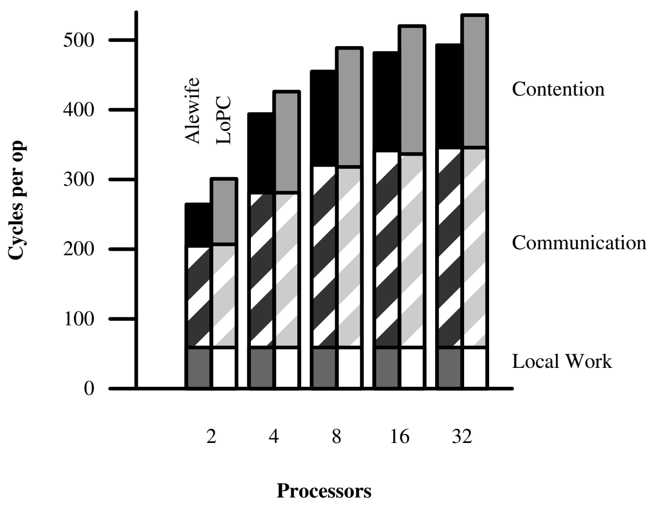

Figure 8: Breakdown of Runtime Costs for Sparse Matrix-vector Multiply Running on MIT Alewife Machines of a Variety of Sizes. For each machine size the left hand bar shows the measured runtime on Alewife, the right hand bar shows the LoPC prediction.

compute/request cycle, by considering the cases where W is very large or very small. Recall that requests arrive at rate $\frac{X}{P} = \frac{1}{R}$. If W is very large, $W \approx R$ and, on average, one interrupting request will arrive each time the local thread runs. In this case W is expanded to include one 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 nearly 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.

Figure 7 shows the contention costs for one compute/request cycle in algorithms with homogeneous all-to-any communication on a 32 node machine, as measured on the simulator and estimated by LoPC. The figure gives the average total contention in one compute/request cycle, as well as the average contention for the request message, reply message, and local processing. In this and the previous figure LoPC gives systematically pessimistic estimates. This is due to the Bard and BKT approximations that overestimate the queue length at the time of message arrival and the interference of high priority requests on the node, respectively. In the worst case observed in these validation experiments, when $W = 0$, LoPC overestimates the total cost of contention by 17%. Most of this error is in the contention faced by reply handlers that LoPC overestimates by 76%. However, LoPC overestimates *total* runtime by only 7% when $W = 0$, with the error asymptotically decreasing to 0 as the work between requests (W) increases. In contrast, the contention free LogP model underestimates total runtime by 37% when $W = 0$. Furthermore, 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 $W = 1000$ the error in total runtime of the contention free model is still 13%.

4.5 Results: Sparse Matrix-Vector Multiply

An example algorithm that exhibits homogeneous all-to-any communication is the sparse matrix-vector multiply discussed in Section 3.1. In this section we compare the measured running time of an implementation of a sparse matrix-vector multiply running on the MIT Alewife machine against the estimates of the LoPC model. The input matrix for this experiment was 13000×13000 with 654800 non-zero entries.

The measured cost of each multiply-add operation on Alewife was 59 cycles. The latency through the network on Alewife is relatively small, about 6 cycles on a 32 node machine. We measured the overhead, S_o, for handling a message including the cost of the interrupt, at about 145 cycles. Message handling times are approximately deterministic (i.e., $C_o^2 = 0$).

Figure 8 shows the breakdown of runtime per non-zero element in the sparse matrix. We ran the algorithm on Alewife configurations with a varying number of processors from 2 to 32. Each left hand bar in the graph shows the time as measured on Alewife, while each right hand bar shows the time predicted by LoPC. As in Section 4.4 we find that LoPC's predictions are slightly pessimistic with a percentage error that diminishes as P increases. LoPC overestimates total runtime by about 14% on a machine with just two nodes and about 9% on a machine with 32 nodes. These experimental results confirm that the insight regarding the impact of contention is valid.

5 Client-Server Communication

In this section we use the LoPC model to derive the optimal number of servers for parallel client-server algorithms, such as work-pile algorithms, on a system with P processors. Both this model and the model in the previous section are special cases of the general LoPC model derived in Appendix A. As in the previous

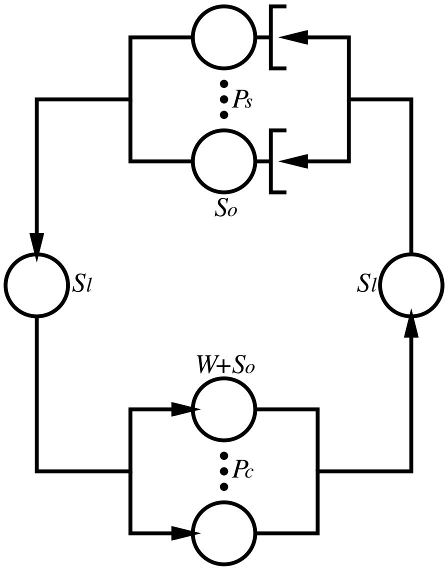

Figure 9: Queueing Delays for Client-Server Algorithms

section we take advantage of application specific features, including thread homogeneity, to simplify the equations.

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 machine is partitioned into P_c client nodes that actually perform the work, and $P_s = P - P_c$ server nodes that distribute work to the clients. Each client node will process a chunk of work and when finished with that chunk, will request another chunk from a randomly chosen server. Because each client node issues requests to the servers at the same average rate, and the compute threads on the servers simply respond to the client requests, the model of Section 3.2 reduces to that shown in Figure 9.

The system has P_c threads running (one per client). At any given point in time, some of these threads will be working and some of them will be in the process of making a request for more work from one of the servers. A key system design issue is how many nodes to allocate as servers. If too few nodes are allocated as servers then the servers will become a bottleneck. On the other hand, if too many nodes are allocated as servers, there will be too few clients to actually do the work. We would like to determine the proper distribution of nodes between clients and servers such that throughput (chunks processed per unit time) is maximized.

We begin the analysis with the following argument that the maximum system throughput will occur for an allocation such that the average number of requests being handled by each server is one. Suppose instead that on average only $P_s - 1$ threads are requesting service. Then, on average, one of the servers will 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 the algorithmic parameter W for the average amount of work done by the client between requests, and a machine with P processors, network latency S_l, handler occupancy S_o, and handler service time variation C_o^2. 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:

$$Q_s = \frac{X}{P_s} R_s; \quad Q_s^{opt} = 1 \Rightarrow X^{opt} = \frac{P_s^{opt}}{R_s^{opt}}. \tag{10}$$

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

$$X = \frac{P_c}{R} = \frac{P - P_s}{R} \tag{11}$$

We can now combine equations (10) and (11) to determine the optimal number of servers in terms of average residence times:

$$P_s^{opt} = \frac{P R_s^{opt}}{R^{opt} + R_s^{opt}}. \tag{12}$$

By Little's result and equation (10) we can determine the utilization at the servers:

$$U_s = \frac{X}{P_s} S_o; \quad U_s^{opt} = \frac{S_o}{R_s^{opt}}. \tag{13}$$

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:

$$R_s = S_o(1 + Q_s + \frac{C_o^2 - 1}{2} U_s), \tag{14}$$

or

$$R_s^{opt} = S_o(2 + \frac{(C_o^2 - 1) S_o}{2 R_s^{opt}}). \tag{15}$$

This equation is simplified by solving for R_s^{opt}:

$$R_s^{opt} = S_o(1 + \frac{\sqrt{2(C_o^2 + 1)}}{2}). \tag{16}$$

Substituting equation (16) for R_s^{opt} into equation (10) gives a closed form expression for the optimal system throughput.

Now that we have determined the average response time of a request at the servers, we can calculate the total average round trip 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:

$$R = W + S_l + R_s + S_l + S_o. \tag{17}$$

Finally, by substituting equations (16) and (17) into equation (12), we find the optimal number of servers in terms of the model input parameters:

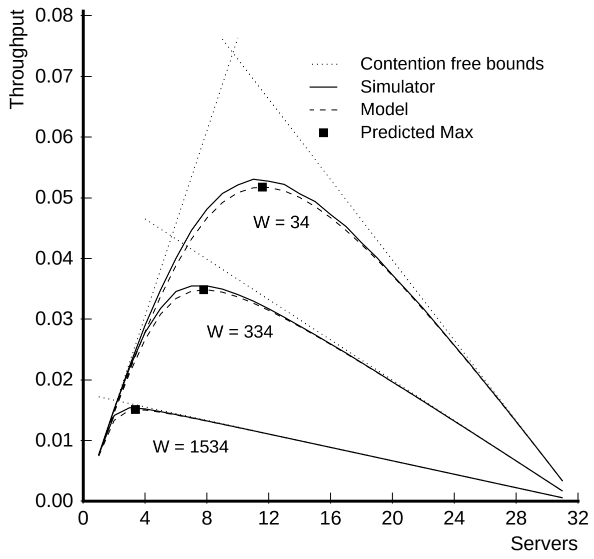

Figure 10: Client-Server Throughput (P = 32, $S_o = 131$ cycles)

$$P_s^{opt} = \frac{P(1 + \frac{\sqrt{2(C_o^2+1)}}{2})S_o}{W + 2S_l + (3 + \sqrt{2(C_o^2+1)})S_o}. \quad (18)$$

Figure 10 shows the throughput estimates of this LoPC model as compared with an event driven simulation of a work-pile algorithm running on a 32 processor machine, for each number of servers, $P_s = 1$ to 31, and clients, $P_c = 32 - P_s$. The black squares on the LoPC curves show the estimated optimal number of servers from equation (18). The LoPC throughput estimates are conservative, with worst case error between observed and predicted optimal throughput equal to 3%.

By examining the throughputs of both the clients and the servers ignoring contention, as in a LogP analysis, we can find optimistic bounds on the throughput of the client-server algorithm. First, consider the upper bound on throughput due to the servers. Since each server can have throughput at most $1/S_o$, $X_s \leq P_s/S_o$.

Next consider the upper bound on throughput due to the clients. The minimum time for a complete compute/request cycle 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 with dotted lines in Figure 10. The contention-free bounds are asymptotically correct, but, unfortunately, the error is quite high in the range where the client-server algorithm has optimal parallelism. The contention-free analysis also underestimates the number of nodes that should be allocated to the servers for optimal throughput.

6 Related Work

As noted in Section 1, several studies have used LogP as a framework for studying applications with asynchronous communication patterns. Lewandowski [22] successfully used LogP to analyze a parallel branch and bound algorithm with a relatively small amount of communication relative to processing. Dusseau *et al* [11] compared LogP analyses of a variety of parallel sorting algorithms with implementations of those algorithms running on a CM-5 with Active Messages. For those algorithms with irregular communication patterns, radix sort and sample sort, they found that their LogP models underestimate communication costs and attribute the difference to contention.

Holt *et al* [18] have performed an empirical study of the sensitivity of several of the SPLASH benchmarks, running on coherent shared memory machines, to the L, o and P parameters of the LogP model. They found that application performance is highly dependent on the cost of contention in the message coprocessor. As in our study, they found that their applications were not sensitive to the g, (gap), parameter of the LogP model.

A recent study by Martin *et al* [25] finds similar empirical results for a variety of fine grain message passing benchmarks running on a network of workstations. They also find that as message overhead increases application runtime increases by a factor greater than predicted by a simple contention free model.

Contention has also been studied in a more formal framework. For example, Dwork *et al* [12] and Gibbons *et al* [15] have extended the PRAM model, traditionally used for parallel algorithm complexity analysis, to account for contention. However this extended PRAM model assumes that each individual memory location has a queue associated with it, whereas LoPC makes the more realistic assumption that queues are associated with message processing resources.

A study by Liu *et al* [23] models more restricted message passing systems in which there is only a single, finite length, queue per processor. They find, as do the studies with more relaxed resource constraints, that for several algorithms contention is bounded by a constant factor. One goal of the LoPC model is to correctly predict the constant factors that are of concern to applications programmers.

Finally, two recent works [1, 16] propose analytic models to predict parallel program performance, including communication and contention costs. The thesis work by Adve [1] models a parallel program with a deterministic task graph and uses mean value analysis to predict mean task execution times, including contention. The work by Harzallah and Sevcik [16] breaks parallel program execution into phases and uses mean value analysis to predict the execution time of each phase, including contention. These studies also illustrate the accuracy of mean value analysis in analyzing the contention due to communication in parallel algorithms, but they have different architectural and algorithmic abstractions than LoPC, and they have not attempted to derive direct insights from the analytic equations.

7 Conclusion and Future Work

This paper has defined a new model called LoPC, which is based on the LogP model and uses a small number of parameters to analyze total application runtime including the impact of contention for message processing resources. The LoPC analysis requires a simple set of algorithmic parameters (n, W, and V_{ij}) and architectural parameters (L, o, and P). Carefully selected approximations in the mean value analysis that is used to analyze contention yield closed form results for at least two important classes of algorithms with two different common communication patterns. Appendix A provides the general equations that can be solved quickly for any parallel algorithm that communicates using active messages. The extensions needed for shared memory systems were identified in Section 4.2.

Using the LoPC model, we quantified the impact of contention for message processing resources for algorithms with homogeneous all-to-any communication (e.g., sparse matrix-vector multiply). The LoPC analysis showed that the total contention cost is bounded by a small constant factor and, to a first approximation, the cost of contention is equal to the cost of an extra handler per compute/request cycle.

For client-server algorithms with homogeneous clients and homogeneous servers, the LoPC analysis yields simple closed form results for the maximum system throughput and the optimal allocation of machine nodes between clients and servers. We also showed that an analysis that ignores contention for message processing resources, such as LogP, overestimates the maximum system throughput and underestimates the optimal server node allocations for this class of algorithms.

The LoPC models were validated against event driven simulations and against a sparse matrix benchmark running on the MIT Alewife machine. In the experiments performed in this paper the LoPC model produces estimates of application running time and throughput that are highly accurate; the maximum observed error in total application runtime was 9% on a 32 node machine. For the usual reasons, system throughput predictions proved even more accurate than total runtime predictions.

Because LoPC is both simple to use and accurately models contention costs, it is a tool that could be broadly applicable to studying algorithms and architectural tradeoffs on both current and next generation parallel architectures. Although only the message passing version of the model has been validated, we expect that the model extensions for communication contention in shared-memory machines will also validate well.

Ongoing work with LoPC includes analysis of further parallel applications and classes of applications, as well as extending the model, using a technique pioneered by Heidelberger and Trivedi [17], to model non-blocking requests such as those that occur in shared memory systems. With this extension we plan to use LoPC to evaluate cost-performance tradeoffs between shared-memory and message-passing communication primitives.

Acknowledgments

We thank Kirk Johnson, Frans Kaashoek, and Ken Sevcik for valuable discussions related to this work, and Larry Rudolph, Charles Leiserson, Donald Yeung, Richard Lethin, Daniel Xjiang, Victor Lee and Kathleen Shannon for insightful comments on earlier versions of this paper.

This research was supported in part by ARPA contract #N00014-94-1-0985, NSF grants #MIP-9504399, #CDA-9024618, #CCR-9024144 and #GER-9550429 and by Matthew Frank's NSF Graduate Research Fellowship.

A The General LoPC Model

The notation used in this appendix is defined in Table 3. We are given a system with P processors, each of which has a thread assigned to it. For each thread i (the thread assigned to processor i) we are given that the thread requires W_i service on the local processor and then makes a blocking request. The request will require, on average, V_{ik} visits to each node k. Note that in general we permit $\sum_{k=1}^{P} V_{ik} \geq 1$, so we can easily model communication patterns that require "multi-hop" requests.

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

$$X_i = \tfrac{1}{R_i} \qquad i = 1, \ldots, P \tag{19}$$

Where R_i is the average response time for thread i.

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

$$X_{ik} = V_{ik} X_i \qquad i, k = 1, \ldots, P \tag{20}$$

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

$$U_{qk} = S_o \sum_{i=1}^{P} X_{ik} \qquad k = 1, \ldots, P \tag{21}$$

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

$$U_{yk} = X_k S_o \qquad k = 1, \ldots, P \tag{22}$$

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

$$Q_{qk} = R_{qk} \sum_{i=1}^{P} X_{ik} \qquad k = 1, \ldots, P \tag{23}$$

$$Q_{yk} = X_k R_{yk} \qquad k = 1, \ldots, P \tag{24}$$

Next, using 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:

$$R_{qk} = S_o(1 + Q_{qk} + Q_{yk}) \qquad k = 1, \ldots, P \tag{25}$$

$$R_{yk} = S_o(1 + Q_{qk}) \qquad k = 1, \ldots, P. \tag{26}$$

By the BKT priority approximation, combined with Bard's approximation, we calculate the response time for each computation thread:

$$R_{wk} = \tfrac{S_o Q_{qk} + W_k}{1 - U_{qk}} \qquad k = 1, \ldots, P. \tag{27}$$

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:

$$\begin{aligned} R_i \quad = \quad & R_{wi} + S_l + R_{yi} + \\ & \sum_{k=1}^{P} V_{ik}(S_l + R_{qk}) \\ & i = 1, \ldots, P \end{aligned} \tag{28}$$

Note that this is more general than equation (1) to account for the possibility of requests that require multiple hops through the network. In addition, the model can be extended to represent handler service time distributions other than exponential, as discussed in Section 4.3.

References

[1] Vikram S. Adve. *Analyzing the Behavior and Performance of Parallel Programs*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, October 1993.

[2] 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.

[3] Albert Alexandrov, Mihai Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating Long Messages into the LogP Model. In *Proceedings of the SPAA'95*, pages 95–105, Santa Barbara, CA, July 1995.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Men-Chow Chiang and Guri Sohi. Evaluating Design Choices for Shared Bus Multiprocessors. *IEEE Transactions on Computers*, 41(3):297–317, March 1992.

[9] 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.

[10] 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.

[11] Andrea C. Dusseau, David E. Culler, Klaus Erik Schauser, and Richard P. Martin. Fast Parallel Sorting Under LogP: Experience with the CM-5. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):791–805, 1996.

[12] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in Shared Memory Algorithms. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, pages 174–183, San Diego, California, May 1993.

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

[14] 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.

[15] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. The QRQW PRAM: Accounting for Contention in Parallel Algorithms. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 638–648, Philadelphia, Pennsylvania, 23–25 January 1994. To appear SIAM Journal on Computing.

[16] Karim Harzallah and Kenneth C. Sevcik. Predicting Application Behavior in Large Scale Shared-memory Multiprocessors. In *Supercomputing '95*, December 1995.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Scott Leutenegger and Mary K. Vernon. A Mean Value Performance Analysis of a New Multiprocessor Architecture. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurment and Modeling of Computer Systems*, pages 167–176, Santa Fe, New Mexico, May 1988.

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

[23] Pangfeng Liu, William Aiello, and Sandeep Bhatt. An Atomic Model for Message-Passing. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 154–163, June 1993.

[24] 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.

[25] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[26] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurment and Modeling of Computer Systems*, pages 48–60, Santa Clara, CA, May 1993.

[27] M. Reiser and S.S. Lavenberg. Mean Value Analysis of Closed Multichain Queueing Networks. Report RC-7023, IBM T.J. Watson Research Center, March 1978.

[28] 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.

[29] 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.

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

[31] Mary K. Vernon, Edward D. Lazowska, and John Zahorjan. An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 308–315, Honolulu, June 1988.

[32] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 40–53, December 1995.

[33] 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.