LogGP: Incorporating Long Messages into the LogP Model —

One step closer towards a realistic model for parallel computation

Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106

{*berto,mionescu,schauser,chriss*}@*cs.ucsb.edu*

Abstract

We present a new model of parallel computation—the LogGP model—and use it to analyze a number of algorithms, most notably, the single node scatter (one-to-all personalized broadcast).

The LogGP model is an extension of the LogP model for parallel computation [CKP+93] which abstracts the communication of fixed-sized short messages through the use of four parameters: the communication latency (L), overhead (o), bandwidth (g), and the number of processors (P). As evidenced by experimental data, the LogP model can accurately predict communication performance when only short messages are sent (as on the CM-5) [CKP+93, CDMS94]. However, many existing parallel machines have special support for long messages and achieve a much higher bandwidth for long messages compared to short messages (e.g., IBM SP-2, Paragon, Meiko CS-2, Ncube/2).

We extend the basic LogP model with a linear model for long messages. This combination, which we call the LogGP model of parallel computation, has one additional parameter, G, which captures the bandwidth obtained for long messages. Experimental data collected on the Meiko CS-2 shows that this simple extension of the LogP model can quite accurately predict communication performance for both short and long messages. This paper discusses algorithm design and analysis under the new model, examining the all-to-all remap, FFT, and radix sort.

We also examine, in more detail, the single node scatter problem. We derive solutions for this problem and prove their optimality under the LogGP model. These solutions are qualitatively different from those obtained under the simpler LogP model, reflecting the importance of capturing long messages in a model.

1 Introduction

To guide parallel algorithm designers, a simple but accurate model of parallel computation is needed. The frequently used PRAM model [FW78, KR90] is useful for a gross classification of parallel algorithms, but unfortunately, it is not well suited for predicting performance on actual parallel machines. As a consequence, a variety of alternative models have been developed, each capturing different aspects of real parallel machines, such as memory contention [MV84, KLMadH92], asynchronous execution [Gib89, CZ89], communication latency [PY88, ACS89], or communication bandwidth [ACS90]. Others include network models for a variety of topologies [Lei92], sparse networks [Sny86], models which take the memory hierarchy into account [AC94], or models based on communication primitives more powerful than simple point-to-point messages [Ble87]. There are also special models for shared memory computation, e.g., the CICO model [LCW94].

Some models address several of the above aspects. For example, the BSP [Val90], the Postal model [BNK92], and LogP [CKP+93] all capture both communication latency and bandwidth through parameters.

The LogP model, in addition, captures the communication overhead, i.e., the time it takes for a processor to send or receive a message. The three models are very similar, in that communication is modeled by point-to-point messages and the performance characteristics of communication networks are abstracted through a small number of parameters, ignoring specific details about network topology. This makes it easier to design portable parallel algorithms which do not rely on specific topologies and which can adapt to varying hardware characteristics.

The LogP model reflects the convergence of parallel machines towards systems formed by a collection of complete computers, each consisting of a microprocessor, cache and large DRAM memory, connected by a communication network [CKP+93]. The LogP model for parallel computation models communication performance through the use of four parameters: the communication latency (L), overhead (o), bandwidth (g), and the number of processors (P). Communication is modeled by point-to-point messages of some fixed short size. Thus, the model has implicitly a fifth parameter, the message size w. As evidenced by experimental data collected on the CM-5 [TM94], this model can accurately predict communication performance when only fixed-sized short messages are sent [CKP+93, CDMS94, LC94]. However, many existing parallel machines have special support for long messages which provide a much higher bandwidth than short messages (e.g., IBM SP-2 [BBB+94], Paragon [Pie94], Meiko CS-2 [BCM94], Ncube/2 [SV94]). Even low overhead communication architectures, such as the generic active message specification [CKL+94], support bulk transfers. The LogP model only deals with short messages and does not adequately model machines with support for long messages.

Our work addresses this shortcoming by extending the LogP model with a linear model for long messages. The goal is to design a performance model which captures both short and long messages. We want the new model to be realistic enough to characterize machines and accurately predict performance, but still simple enough for programmers to design and analyze parallel algorithms. Communication models for long messages usually model the time to send an n byte message by a linear model, $t = t_0 + t_B * n$, where t_0 is the startup time and t_B is the time per byte [Hoc94, KGGK94]. The startup time t_0 lumps into a single parameter the overhead and latency differentiated in the LogP model. While combining these parameters may be appropriate for long messages, we believe that this is not sufficiently detailed for short fixed-size messages.

The simplest way for a programmer to incorporate long messages into a program is to group several short messages destined to the same processor into a longer, and thus more efficient, message. Interestingly enough, the ability to send long messages may even change the algorithm and communication structure, trading off extra bandwidth against running time.

The remainder of the paper is structured as follows. In Section 2 we extend the basic LogP model with a linear model for long messages and illustrate its use. This combination, which we call the LogGP model, has one additional parameter, G, which captures the bandwidth obtained for long messages. In Section 3 we present experimental data for communication kernels and complete applications collected on the Meiko CS-2. This data shows that this simple extension to the LogP model can quite accurately predict communication performance for both short and long messages. The data also demonstrates that sending long messages results in much better overall performance than sending short messages. In Section 4 we develop optimal algorithms for the single node scatter problem under the LogGP model. The optimal algorithm leads to a communication structure that is qualitatively different from the one obtained under the LogP model, demonstrating the importance of capturing long messages. We implement our algorithms for the single-item and k-item scatter problems on the Meiko CS-2 and validate our theoretical results by comparing measured performance with that predicted by the model. The results show a close match. Section 5 summarizes the work and concludes.

2 The LogGP model

We use the LogP model [CKP+93] as a basis for our work, and extend it with a simple linear model for long messages. The resulting model, which we call LogGP, can model both short and long messages. Under the new model the processors communicate by point-to-point messages and communication performance is characterized by the following parameters:

- *L*: an upper bound on the *Latency*, incurred in sending a message from its source processor to its target processor.
- *o*: the *overhead*, defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time the processor cannot perform other operations.
- *g*: the *gap* between messages, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of g corresponds to the available per processor communication bandwidth for short messages.
- *G*: the *Gap per byte* for long messages, defined as the time per byte for a long message. The reciprocal of G characterizes the available per processor communication bandwidth for long messages.
- *P*: the number of processor/memory modules.

Just as in the LogP model, all of the parameters are measured as multiples of the processor cycle. For concrete machines we often use actual times (in μs). Implicitly, the model also has another parameter, the size w of small messages. In describing the characteristics of a parallel machine, defining G to be the time per byte is most natural. However, sometimes it may be easier to express G as the gap per item of size w bytes. For example, this is how we use G in our analysis of the scatter problem in Section 4.

Our model assumes that the receiving processor may access a message, or parts of it, only after the entire message has arrived. We also assume a single port model, i.e., at any given time a processor can either be sending or receiving a single message.

2.1 Usage of the Model

The time to send a small message can be analyzed as in the LogP model. Sending a small message between two processors takes $o + L + o$ cycles: o cycles on the sending processor, L cycles for the communication latency, and finally another o cycles on the receiving processor. Under the LogP model, sending a k byte message from one processor to another requires sending $\lceil k/w \rceil$ messages, where w is the underlying message size of the machine. This would take $o + (\lceil k/w \rceil - 1) * max\{g, o\} + L + o$ cycles. In contrast, sending everything as a single large message takes $o + (k-1)G + L + o$ cycles[1] under the new LogGP model, as shown in Figure 1.

Sending a message of k bytes first involves o cycles of sending overhead to get the first byte into the network. Subsequent bytes take G cycles each to go out. The last byte goes out at time $o + (k-1)G$. Each byte travels through the network for L cycles. Thus the last byte exits the network at time $o + (k-1)G + L$. Finally, the receiving processor spends o cycles in overhead, so the entire message is available at the receiving processor at time $o + (k-1)G + L + o$. The sending and receiving processors are busy only during the o cycles of overhead, the rest of the time they can overlap computation with communication.

If a processor wants to send two long messages in a row (of length k_1 and k_2 respectively) it has to wait g cycles after the first message goes out before it can push the first byte of the second message into the

[1]Note, that g does not appear in this equation since only a single message is sent.

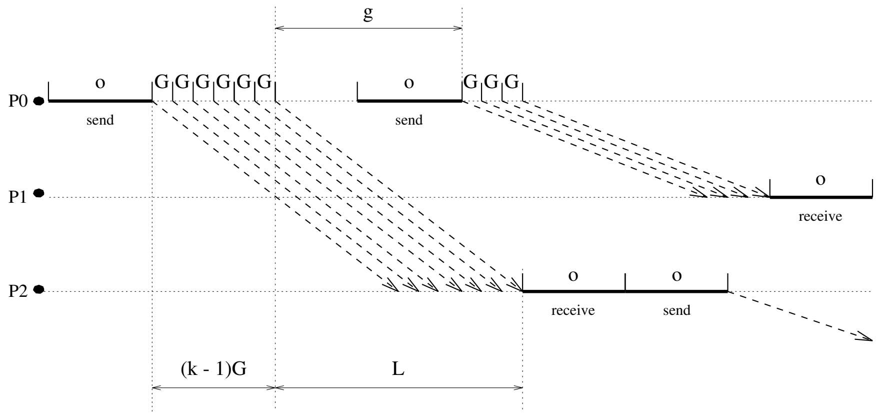


Figure 1: Sending and receiving messages under the LogGP model.

network. As shown in Figure 1, the processor can overlap the sending overhead with the time spent waiting for the network. Thus the first byte of the second message enters the network at time $o + (k_1 - 1)G + g$. The last byte of the second message leaves at time $o + (k_1 - 1)G + g + (k_2 - 1)G$, and arrives at the destination processor $L + o$ cycles later.

2.2 Discussion of the Model

We think that our model is the best way to extend LogP to support sending of long messages. The bandwidth parameter G makes the model much more realistic for long messages. The three parameters, o, g and G, capture different forms of communication bottlenecks: the parameter o captures the time the main processor is involved in sending and receiving; the parameter G reflects the per processor network bandwidth for long messages, while the parameter g captures the startup bottleneck of the network. The startup bottleneck could be due to a variety of reasons, e.g., on the Meiko CS-2 this is due to the communication co-processor opening a communication channel.

As in the original LogP model, it is often possible to work only with a subset of the parameters. For example, if only short messages are being sent, the LogGP model reduces to the LogP model. For very long messages, the time to send the message can be approximated simply as kG. Similarly, on certain architectures, g and o may be close, so they can be combined into a single parameter. On the other hand, simplifying the model by removing any of the current parameters creates loopholes, which, when exploited by algorithm designers, could lead to incorrect performance predictions.

In designing the LogGP model we have considered various alternatives. One alternative to the LogGP model would be to use only a standard linear model, in effect combining the overhead and latency differentiated in the LogGP model into a single parameter. Although this would simplify the model, the current distinction between overhead and latency allows us to design algorithms which overlap communication with computation. Another alternative introduces a separate parameter for the startup cost of long messages. Currently, the startup cost is modeled as $L + 2o$ for both short *and* long messages. The additional parameter would make the model more precise when different communication protocols are used for long and short messages.[2] A third alternative is to assume that sending a message of k bytes takes $kG + 2o + L$ instead of $(k - 1)G + 2o + L$. This slight variation of the LogGP model seems more natural, but has the disadvantage of not reducing to the LogP model for $k = 1$. Our model for $k = 1$ is precisely the LogP model.

[2]For example, on the CM-5, a communication segment has to be set up before sending a long message. Similarly, on the Meiko CS-2, a long active message transfer turns into two operations: a DMA transfer for the long message, followed by a short active message.

Obviously, any machine model will not be able to capture all architectural variations. For example, our model simplifies algorithm design by assuming a fully connected network and a constant bandwidth parameter G. In reality, only high degree networks such as fat trees will have this property, while lower degree networks such as 2D-meshes will be modeled less accurately. A more accurate model could have G as a function of the topology and the number of processors. Furthermore, our model does not specifically deal with the memory hierarchy found on every node. Nevertheless, we feel that our model adequately captures the most important aspect of parallel machines, the interprocessor communication. This forces the algorithm designer to address the issue of data layout and grouping of messages.

2.3 Impact of Long Messages on Algorithm Design

The most straightforward way to make use of the higher bandwidth provided by long messages is to group several short messages into a single long message. To combine multiple short messages into a single long message the algorithm designer may have to adapt the algorithm, i.e., it may be necessary to rearrange the computation to get the data into long messages. For example, as discussed in Section 3 below, in the cyclic-to-blocked remap of the FFT the blocks can directly be sent as a single long message, but do not go into a contiguous location on the receiving side. We need an additional redistribution phase which scatters the long message into the right location. Just the opposite occurs in radix sort. Here, in order to be able to send all of the elements destined to a single processor as a single message, we first have to collect them. This is done by a local sorting phase which is not required for the short message implementation. In both cases, the execution time to obtain the right setup and right distribution is substantial and is more expensive than the actual transfer time.

In these two examples the program transfers the same number of data items under long messages as under short messages. This is not always the case. In Section 4 we discuss an example where long messages may change the structure of an algorithm itself: in the optimal scatter algorithm, we can reduce the running time by increasing the network traffic.

3 Validating the Model

In this section we validate the model by comparing the model's predictions with experimental results.

Our experimental platform is a 64 node Meiko CS-2. The CS-2 consists of Sparc based nodes connected via a fat tree communication network [HM93]. Running a slightly enhanced version of the Solaris 2.3 operating system on every node, it closely resembles a cluster of workstations connected by a fast network. Each node contains a 40 MHz SuperSparc processor with 1 MB external cache and 32 MB of main memory. Each node in a CS-2 contains a special communications co-processor, the Elan processor, which connects the node to the fat tree. The co-processor enables direct user-level communication between the processors. The co-processor contains a *DMA engine* which performs DMA transfers of arbitrary size from the source node to the destination node.

The LogGP parameters for the Meiko CS-2 are shown in Table 1. We show two sets of parameters, corresponding to two communication libraries. One is Meiko's own low-level Elan library [HM93], and the other is an implementation of Split-C [CDG+93] (a parallel extension of C) based on active messages [vECGS92, SS95]. In both cases, we measured the parameters by timing a loop which repeats the relevant communication primitive many times to minimize timer overhead.

To validate the model we focus on two questions. First, given the parameters determined for the Meiko, does the model accurately predict the execution time of parallel programs? And second, is the LogGP model a significant extension of the LogP model?

We begin by examining a simple communication pattern: sending messages between two processors. Figure 2 shows the bandwidth for messages of up to 64 KBytes on the Meiko CS-2 under Split-C. There is

Meiko CS-2	L	o	g	G	BW
Split-C	8.6 μs	1.7 μs	14.2 μs	0.03 μs	33.3 MB/s
Elan lib	10 μs	3.8 μs	13.8 μs	0.023 μs	43 MB/s

Table 1: LogP parameters for the Meiko CS-2 under the Elan library and Split-C.

indeed a substantial difference in bandwidth between long and short messages. We see that the bandwidth for long messages increases rapidly and peaks at 33 MB/s, which corresponds to our $G = 0.030\ \mu s$. In contrast, sending 4 byte short messages yields only 2 MB/s. Figure 3 shows the time needed to send messages of moderate size (64–4K bytes). Ideally, the data points would form a perfect line of slope G. This is close to what we observe; the step function is the result of cache misses.

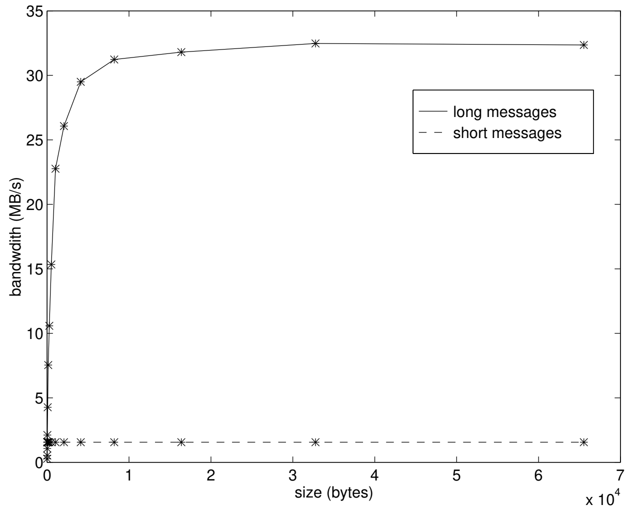


Figure 2: *Bandwidth on the Meiko CS-2 using Split-C.*

We now examine the model under a more complex communication pattern. Figure 4 shows the predicted and measured times for an all-to-all remap. For this test, we transfer from 2K to 512K bytes of data from every processor to all other processors. The communication is staggered such that at stage i ($0 < i < P$), processor p sends to processor $(i+p)$ mod P. In the case of the 2 Kbyte transfer, each processor has an array of $k = 2048 * P$ bytes, and sends a portion of this array to every other processor. There are two versions of this algorithm, which differ in how the data is exchanged: in the first version, short 4 byte messages are used to transfer the data, and in the second version long messages are used.

Figure 4 plots the measured time for the all-to-all transfers using a log-log scale. The expected times are also shown: for the long message version, it is roughly $2kG$, and for the short message version, the expected time is roughly $(1/4)kg$. The factor of 2 in the long message formula occurs because each processor is sending and receiving data, and this can not happen at the same time under the LogGP model. This factor does not appear in the short message version because o is significantly less than g under Split-C; during the time a processor sends a message, it will have time to process another message sent to it. (This is one example that shows that both o and g are important parameters that should not be ignored.) The factor of $1/4$ for the short message version accounts for the 4 byte integers: we send one integer at a time, but measure in bytes. We find that the version with short messages is very accurately predicted. The version using long messages runs within 20% of the expected time for large data sizes. This is probably due to network

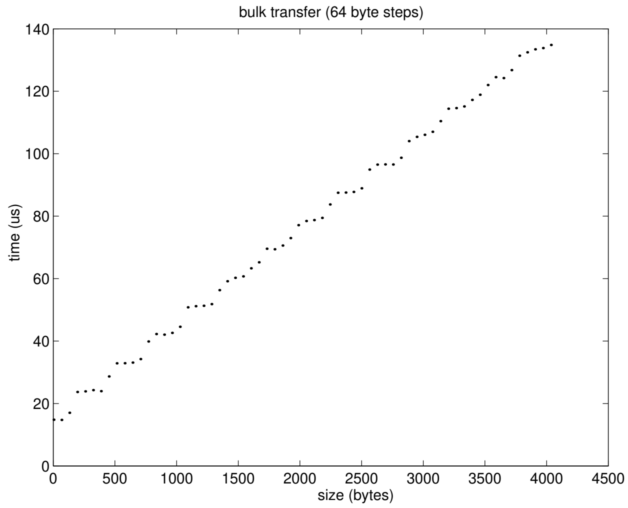


Figure 3: *Time to send bulk messages of moderate size on the Meiko CS-2.*

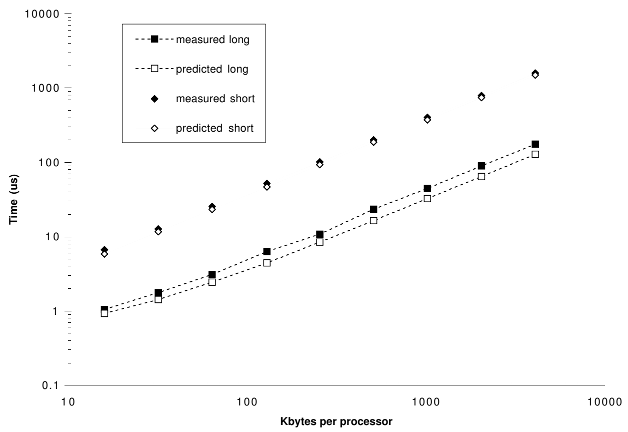


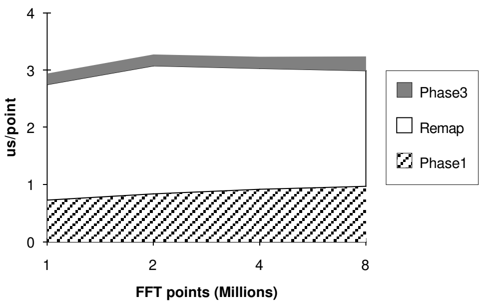
Figure 4: *Predicted and measured times for the all-to-all remap on a Meiko CS-2 (16 processors).*

contention which is not captured by the LogGP model.[3] Overall, we see that the long message version of the all-to-all remap has almost two orders of magnitude improvement over the short message version. The LogP model would fail to explain this improvement.

We now examine two applications: FFT and radix sort. For both applications, we have two versions of the algorithms: the original versions from [CKP+93, CDMS94], based on short messages, and the bulk versions which we restructured to use long messages for data transfer.

A comparison of the two versions of the FFT algorithm is shown in Figure 5. Both our algorithms begin with data in a blocked layout, then perform a single communication phase to map the data into a cyclic layout. The only difference between the two algorithms is how this communication occurs: using short or long messages. For short messages, each message can transfer data directly to its new location in the cyclic layout. For long messages, however, the data within each message must be redistributed to its proper location in memory since the data is no longer contiguous in the cyclic layout. Because of cache misses this local redistribution costs five times as much as the remote communication itself. But in fact, as the

[3] In reality, G increases as network traffic increases, even in a fat tree network such as on the Meiko CS-2. For a communication pattern such as the all-to-all remap, this means that G increases as P increases.



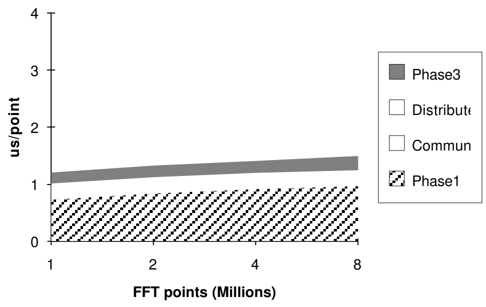


Figure 5: *Execution times for the three phases of the FFT algorithm. The left graph shows the execution times for the short message implementation, and the right graph for the long message implementation, on a 16 processor Meiko CS-2.*

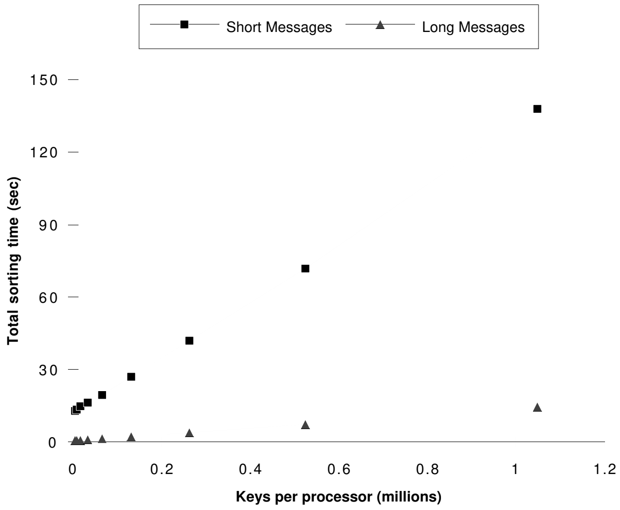


Figure 6: *Execution times for the radix sort algorithm. The two curves show the execution times for both versions of the algorithm (short and long messages), for a 16 processor Meiko CS-2.*

LogGP model predicts, the bulk transfer does save time overall: the long message version finishes the entire computation before the other version even begins the last phase. Note that the LogGP model accurately predicts the time for the data communication, since this is just the all-to-all remap discussed previously and shown in Figure 4.

A comparison of the two versions of the radix sort algorithm is shown in Figure 6. In the original version of the radix sort, each key is sent to the appropriate processor as its position is determined. For the long message version, the positions for all the keys on one processor are first determined by an intermediate sorting step (local). Then all the keys destined for a particular processor are sent at once. This intermediate local sorting step does not occur in the short message version.

4 The Scatter Problem

Collective communication is frequently used in parallel computation and is often built out of point-to-point communication primitives [BBC$^+$94]. In this section we study a specific communication problem in more detail, the single node scatter problem [BT89], also known as the one-to-all personalized broadcast. This problem is of special interest to us because it exhibits the essential difference between the LogGP model and the simpler LogP model. The LogGP algorithms discussed so far are obvious extensions of the corresponding short-message (LogP) algorithms. The approach was to modify an existing short-message based algorithm by grouping short messages into longer ones and thus take advantage of the faster long-message transmission. Here, we present two algorithms which improve the scatter algorithm beyond the simple long-message extension of the short-message algorithm. We achieve a faster running time at the expense of increased network traffic and increased local memory usage: we utilize the high network bandwidth to spread the item distribution work among multiple processors.

The simplest variation of the scatter problem is the single-item scatter. The goal is to distribute $P-1$ items $i_1, i_2, \ldots, i_{P-1}$ from the *source processor*, P_0, to their respective destinations. We would like to emphasize that this is a personalized broadcast problem, i.e., each item, i_j, is in general different from the other items and has a specific recipient, processor P_j. For uniformity of notation we assume that there is an additional item, i_0, which is destined for the source P_0. Of course i_0 need not be transmitted since it is already at its destination.

The more general problem is the k-item scatter problem. Here the source processor has P sets of items, $I_0, I_1, \ldots, I_{P-1}$, each of size k. The destination of the items in set I_j is processor P_j.

4.1 Parameter Simplification

While LogGP is a general model of a parallel machine, the broadcast problems we study in this section are pure communication problems which cannot exploit the possibility of overlapping communication with local computation. Therefore, we can simplify our analysis of the problems by working with a restricted version of LogGP without sacrificing any accuracy. This simplified version has two fewer parameters.

The first step in the simplification is to note that since the broadcast problem requires no overlap of communication and computation, the parameter o is not needed. Instead, the sending and receiving overhead can be incorporated into the latency L to give us a new latency $L' = L + 2o$. As a second step, we eliminate G by normalizing it to 1, and scale L and g accordingly. (For the analysis of our algorithms we measure G with respect to one item.) The final result is a model with only three parameters: latency, L, minimum gap between messages, g, and number of processors, P.

In the remainder of this section we will use the simplified model for our analysis of the scatter problem. We start by describing three algorithms for the (k-item) scatter. After this we derive the optimal algorithm for the single-item scatter problem.

4.2 The Optimal k-Item Algorithm Under LogP

Under the LogP model we can only use short messages consisting of one item. This makes the optimal k-item scatter algorithm under LogP very simple: the source processor sends each of the $(P-1)k$ items to its destination as a separate message (Figure 7). Since this is the fastest way for the source processor to send out all the items, the algorithm is optimal. We refer to this algorithm as the *Short-Message* algorithm. Its running time is given in Table 2.

4.3 The Simple Extension for LogGP

The Short-Message algorithm is not optimal under the LogGP model. It is easy to improve it by sending all the items destined for the same processor as a long message of k items instead of sending k separate ones. The running time of this algorithm, which we call the *Simple Long-Message* algorithm, is also given in Table 2. Although the new algorithm is significantly faster than the Short-Message algorithm for $k > 1$ items, it is still not optimal. Furthermore, for $k = 1$ it takes exactly as much time as the Short-Message algorithm.

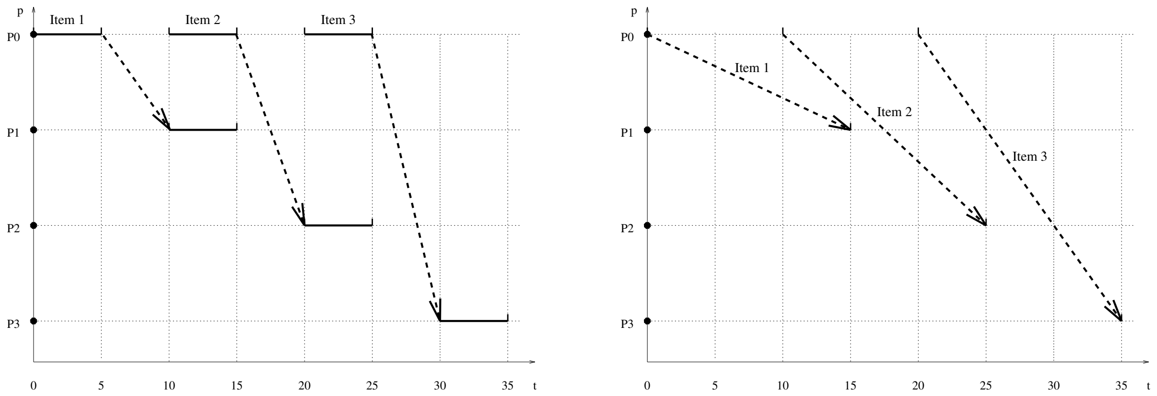


Figure 7: The Short-Message algorithm is optimal under LogP. In the example $k = 1$ item and $P = 4$ processors. The graph on the left shows the algorithm as viewed through the full LogGP model ($L = o = 5, g = 10, G = 1$); the graph on the right uses the simplified LogGP ($L' = 15, g' = 10$).

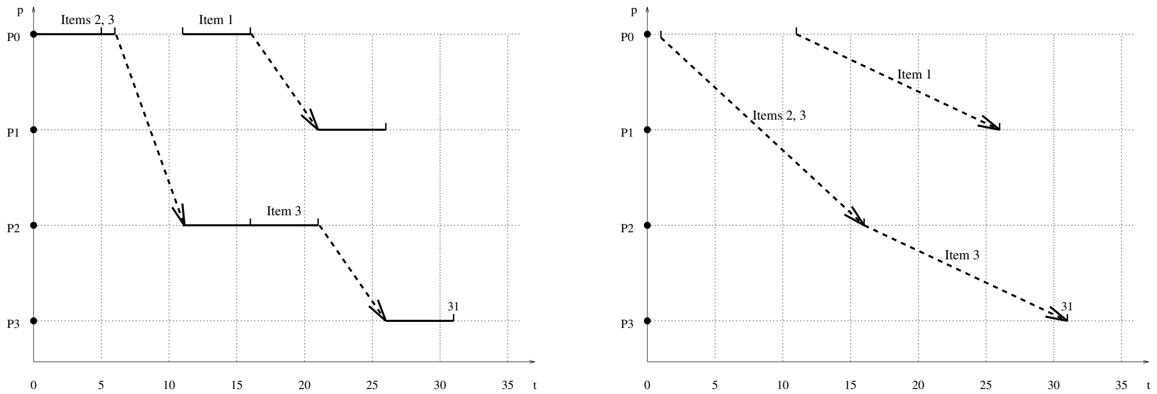


Figure 8: The Binomial Tree algorithm for the single item scatter problem for $k = 1$ item on $P = 4$ processors under the LogGP model. The graph on the left shows the algorithm as viewed through the full LogGP model ($L = o = 5, g = 10, G = 1$); the graph on the right uses the simplified LogGP ($L' = 15, g' = 10$).

4.4 The Binomial Tree Algorithm

The intuition behind the next algorithm is that we want to take full advantage of the long messages by grouping more items in a single message. We are willing to put items destined for different processors in a single message in order to "get rid" of them as fast as possible. The receiving processor will have to complete the delivery of the items that are not destined for it.

The single-item scatter works as follows (Figure 8). Assume that the number of processors, P, is a power of 2. In the beginning P_0 sends to processor $P_{\frac{P}{2}}$ all the items destined for processors $P_{\frac{P}{2}}, P_{\frac{P}{2}+1}, \ldots, P_{P-1}$. Thus the processors are split in two groups of equal size and the broadcast problem is reduced to two independent broadcast problems of half size. Processor P_0 is the source of the broadcast for the first group and $P_{\frac{P}{2}}$ is the source for the second group. If the number of processors is not divisible by two, the number of items sent is $\lfloor \frac{P}{2} \rfloor$.

The algorithm generalizes easily for all values of k. At each step the source processor simply sends half of the item groups as a long message to a new processor thus reducing the problem to two subproblems of half size.

Observe that the Binomial Tree algorithm achieves a better runing time at the expense of increased network traffic. In Figure 8, for example, item 3 is sent twice over the network. Since such items must be buffered at the intermediate nodes before retransmission, this algorithm uses additional local memory compared to the Simple Long-Message algorithm. For example, when P is a power of 2, processor $P_{\frac{P}{2}}$ needs to buffer $k(\frac{P}{2} - 1)$ items destined for other processors, $P_{\frac{P}{4}}$ must buffer $k(\frac{P}{4} - 1)$ items, and so on.

The running time for the Binomial Tree algorithm, if P is a power of 2, is given in Table 2. (The formula for arbitrary P is given and proved later in Subsection 4.8, where it is used as a lemma to bound the optimal algorithm.) Table 3 shows the predicted running times of the three algorithms discussed so far for two sets of parameters. For comparison the table includes the single-item optimal algorithm described in the following section.

Algorithm	Time complexity
Short-Message	$((P-1)k-1)g+L$
Simple Long-Msg.	$(P-2)g+(P-1)(k-1)+L$
Binomial Tree	$\max\{L,g\}(\log_2 P-1)+L+(P-1)k-\log_2 P$

Table 2: Comparison of the three algorithms for the k-item scatter problem. (For the Binomial Tree formula, we assume that P is a power of 2.)

Algorithm	k = 1 g = 10 L = 30	k = 1 g = 100 L = 300	k = 10 g = 10 L = 30	k = 10 g = 100 L = 300	k = 100 g = 10 L = 30	k = 100 g = 100 L = 300
Short-Message	10,250	102,500	102,320	1,023,200	1,023,020	10,230,200
Simple Long-Message	10,250	102,500	19,457	111,707	111,527	203,777
Binomial Tree	1,313	4,013	10,520	13,220	102,590	105,290
Optimal	1,171	2,860	10,358	11,819	102,419	103,688

Table 3: Predicted performance of the three algorithms for the k-item scatter problem for $P = 1024$. The parameters are for the simplified LogGP model.

4.5 The Optimal Algorithm for the Single-Item Scatter Problem

The Binomial Tree algorithm is faster than the Simple Long-Message algorithm but nevertheless is generally suboptimal. In this section we derive the optimal algorithm for the scatter of $k = 1$ item. Later we extend the algorithm for arbitrary k.

Examining the Binomial Tree algorithm, we can observe that dividing the items into two equal halves may lead to load imbalance. Because of the difference between L and g, the sending processor may be able to start sending a new message before the receiving processor can send out its first message. This happens when $g < L$. In these cases it may be advantageous for the sending processor to do more work by delegating a little less than half of the items to the new processor and keeping a little more than half of the items for itself. The new algorithm, which we denote A_t, runs in time $t(P)$ where P is the number of processors and

$t(P)$ is as follows:[4]

$$
\begin{aligned}
t(1) &= 0 \\
t(P) &= \min_{0<s<P}\{(s-1)+\max\{L+t(s), g+t(P-s)\}\}
\end{aligned}
$$

The number of items to be sent in the first split is $S(P)$, where $S(P)$ is the value of s which minimizes the formula for $t(P)$. To ensure that $S(P)$ is unique we pick the smallest s if there are multiple values for which the minimum is achieved.

Given a set of parameters, L and g, we can determine for any P a single solution to the recurrence relation for $t(P)$. Algorithm A_t works recursively as follows: Assume processor P_i has P data items for P processors. Processor P_i splits the data into two groups: One group, of size $s = S(P)$, is sent to a new source processor, P_j. The other group, of size $P - s$, is kept by processor P_i. Sending out the s items takes $(s-1)$ time. P_j receives the items with a delay of L and then broadcasts them in $t(s)$ time using the optimal algorithm for s processors. Meanwhile, processor P_i must wait for g cycles and then distributes its remaining $P-s$ items in $t(P-s)$ time using the optimal scatter algorithm for $P-s$ processors. Algorithm A_t finishes with the slower of the two independent broadcast trees.

Algorithm A_t is guided by the values $S(P)$ which determine how many items to send out and how many to keep during each split. For a given set of LogGP parameters, $S(P)$ depends only on the number of processors P. Thus, for a specific machine with given L and g, $S(P)$ can be precomputed and used during the actual broadcast. Note that A_t is superior to the Binomial Tree algorithm, which can be obtained by always choosing $S(P) = P/2$ instead of selecting the best value.

The remainder of this section is dedicated to proving that A_t is an optimal algorithm. First we show that there is at least one "well-behaved" optimal algorithm. We define the following two restrictions on the actions an algorithm can take.

Definition 1

- *Restriction $R1$: No processor sends out its own item.*
- *Restriction $R2$: No processor sends the same item more than once.*

Lemma 1 *There is an optimal algorithm for the single item scatter problem that complies with restrictions $R1$ and $R2$.*

> **Proof:** Let A_{opt} be an optimal algorithm for the single item scatter problem and let $t_{opt}(P)$ be the time A_{opt} takes. If A_{opt} violates $R1$ then we can remove from each message the item that is destined for the sending processor. The newly obtained algorithm complies with $R1$ and runs no slower than $t_{opt}(P)$ and therefore is optimal.
>
> Assume that A_{opt} violates $R2$, i.e., there is a processor P_j which sends out an item i_l more than once. One of the several copies of i_l is the first to arrive at the destination P_l. Obviously the other copies of i_l need not have been sent. If we modify A_{opt} by not sending the unnecessary copies of i_l we obtain a correct algorithm. The new algorithm complies with $R2$ and runs no slower that $t_{opt}(P)$. $\square$

[4] We need to define the finishing time of an algorithm precisely. One can think of computation finishing time, which is the point at which all processors become free to start local computation, and communication finishing time, which is the point when all processor becomes free to start a communication operation. Note that the communication finishing time is always bigger than, or equal to, the computation finishing time. Throughout the remainder of this paper we only use the communication finishing time to measure the running time of an algorithm. In particular an algorithm which sends a single item from P_0 to P_1 finishes at time $\max\{L, g\}$ even when $g > L$, i.e., we say that P_0 finishes at time g (its communication finishing time), rather than 0 (its computation finishing time).

Having proved Lemma 1, without loss of generality we will only consider algorithms that comply with $R1$ and $R2$. Now we describe how all these algorithms work. Consider the implications of Lemma 1. Since no processor can send the same item more than once, we can view each item as a token that is *passed* from processor to processor without being replicated. At a given point in time each processor *owns* a set of items. In the beginning of the algorithm, processor P_0 owns all the items and the other processors own none. At the end each processor, P_j, owns only its item, i_j. Items are passed by processor P_i to a new processor, P_j, when P_i sends a message to P_j. At this point the ownership of the items contained in the message is transferred from P_i to P_j. Thus at any time each item is owned by exactly one processor.

Now we prove that A_t is optimal. To do this we show that A_t is optimal for a more general problem from which it automatically follows that A_t is optimal for the more restricted problem. Consider the generalized single-item scatter problem for P processors with the modification that in addition to the P processors there is an infinite number of *external* processors $P_P, P_{P+1}, \ldots$ which conceivably can be used to speed up the broadcast. Let A_{opt}^{gen} be an optimal algorithm for the generalized problem and let t_{opt}^{gen} be its running time. It is simple to see that Lemma 1 holds for the generalized problem as well, thus we can assume that A_{opt}^{gen} complies to $R1$ and $R2$. The following is our main result in the analysis of the scatter problem.

Theorem 1 *A_t is an optimal algorithm for the generalized single-item scatter problem.*

Proof: Consider an optimal algorithm A_{opt}^{gen} with running time $t_{opt}^{gen}(P)$ that complies with $R1$ and $R2$. We prove by induction on P that $t(P) \leq t_{opt}^{gen}(P)$.

Base: When $P = 2$ the optimal algorithm is unique and it is A_t (processor P_0 sends item i_1 to P_1). Thus $t(2) \leq t_{opt}^{gen}(2)$.

Hypothesis: Assume that $t(P) \leq t_{opt}^{gen}(P)$ for all $P \leq n$.

Inductive step: Consider how A_{opt}^{gen} works for $P = n+1$. The first step is the sending of a message M_{send} from P_0 to some other processor P_j. Let s be the number of items in M_{send}. Observe that $1 \leq s \leq n$, because M_{send} cannot be empty and P_0 cannot send out all $n+1$ items (it must keep item i_0 because A_{opt}^{gen} complies with $R1$). After the message is received by P_j, the items in M_{send} are owned by P_j and P_0 loses them. Denote the set of the $P - s$ items that remain at P_0 with M_{keep}. At this point the algorithm can be viewed as composed of two broadcasts; the first rooted at P_0 delivers items M_{keep} to their respective destinations in time $T_{keep}(n+1-s)$, and the second rooted at P_j delivers items M_{send} to their destinations in time $T_{send}(s)$. Note that in general, unlike in A_t, the two broadcasts can intersect (i.e, have processors in common). The total time of the algorithm is

$$\begin{aligned} t_{opt}^{gen}(n+1) &= (s-1) + \\ &\quad \max\{L + T_{send}(s), g + T_{keep}(n+1-s)\} \end{aligned}$$

We establish a lower bound on $T_{send}(s)$ and $T_{keep}(n+1-s)$. We do this by assuming that the two broadcasts do not interfere with one another. Consider the broadcast for M_{keep}. Even if there is no interference from the broadcast for M_{send}, the items in M_{keep} cannot be delivered to their destinations faster than $t_{opt}^{gen}(n+1-s)$. But $s > 1$ which means that $n+1-s \leq n$ and thus $t_{opt}^{gen}(n+1-s) \geq t(n+1-s)$ from the inductive hypothesis. Therefore $T_{keep}(n+1-s) \geq t(n+1-s)$.

Now consider the second broadcast tree independent of the first one. Excluding item i_j, P_j has to distribute either $s-1$ or s items, depending on whether its item i_j was in message M_{send}. Note that if P_j is an "external" processor (i.e., $j \geq n+1$) there is no item i_j. If $i_j \in M_{send}$ then $s-1$ items must be distributed (excluding i_j). $T_{send}(s) \geq t(s)$ because $s < n+1$ and $t(s)$ is the fastest way to distribute $s-1$ items according to the hypothesis. In the second case ($i_j \notin M_{send}$), P_j has s items to distribute but it does not have its own item. Obviously this cannot be done faster than the optimal time $t_{opt}^{gen}(s+1)$. Observe that in this case $s \leq n-1$, because P_0's item and P_j's item were not in M_{send}. Thus $s+1 \leq n$ and by the inductive hypothesis $t(s+1) \leq t_{opt}^{gen}(s+1)$, therefore $T_{send}(s) \geq t(s+1) \geq t(s)$ (because $t(P)$ is a monotone function and thus $t(s+1) \geq t(s)$).

Thus we established that $T_{keep}(n+1-s) \geq t(n+1-s)$ and $T_{send}(s) \geq t(s)$. Now, from the formula for $t(P)$ we derive that

$$t_{opt}^{gen}(n+1) \quad \geq \quad (s-1) + \max\{L + t(s), g + t(n+1-s)\}$$

But the right-hand side of this equation is exactly the expression that is minimized in the formula for $t(P)$ for $P = n+1$. Therefore

$$t(n+1) \quad \leq \quad (s-1) + \max\{L + t(s), g + t(n+1-s)\}$$

and thus $t_{opt}^{gen}(n+1) \geq t(n+1)$.

This completes the inductive proof. We established that for any P $t(P) \leq t_{opt}^{gen}(P)$, i.e., algorithm A_t is an optimal algorithm for the generalized single-item scatter problem. □

As already mentioned, since A_t is optimal for the more general problem using an infinite number of processors, it is optimal for the single-item scatter problem on P processors.

4.6 Approximating the Optimal Algorithm

One may wish to approximate the optimal algorithm by selecting a fixed splitting ratio α, $0 < \alpha < 1$, and always chosing $S(P) = \max\{\lfloor P\alpha \rfloor, 1\}$. The resulting algorithm would be easier to implement than the optimal one because it does not require the use of a precomputed table. For example, the binomial tree algorithm is derived when $\alpha = \frac{1}{2}$ which is optimal whenever $L = g$. In general, however, it is not possible to obtain an optimal algorithm by using a fixed α. The reason is that when the number of items a processor owns is large, then the optimal splitting ratio for the first several messages is very close to $\frac{1}{2}$. On the other hand at the final stages of the broadcast when the number of items a processor has to distribute is small, the splitting ratio may be very far from $\frac{1}{2}$ due to the difference between L and g. For example suppose that $L = 30$ and $g = 10$. One can easily verify that when a processor has 5 items it must send exactly one of them in the first message and keep the remaining 4 in order to achieve optimal time. This means that $\alpha < \frac{2}{5}$, otherwise $\lfloor P\alpha \rfloor$ would not be 1. On the other hand when the number of items owned by a processor is 320 then optimal running time can only be achieved if the first message contains exactly $s = 150$ items. This implies that $\frac{150}{320} \leq \alpha$. Since $\frac{2}{5} < \frac{150}{320}$, it is not possible to pick a fixed value for α that will produce an optimal algorithm.

4.7 The Algorithm for k Items

The k-item scatter problem is more difficult to analyze because each processor must receive multiple items and each item could conceivable arrive through a different route. We do not know of an optimal algorithm for the k-item case. Nevertheless, we can extend the single-item-optimal algorithm to work for k-items by simply treating each set of items I_j as an indivisible larger item. That is, we do not allow splitting of item sets among different messages. Denote the new algorithm with A_t^k. It runs in time $t^k(P)$ where

$$\begin{aligned} t^k(1) &= 0 \\ t^k(P) &= \min_{0<s<P} \{(sk-1) + \max\{L + t^k(s), g + t^k(P-s)\}\} \end{aligned}$$

The number of sets of items to be sent in the first split is $S^k(P)$, where $S^k(P)$ is the value of s which minimizes the above formula. To ensure that $S^k(P)$ is unique we pick the smallest s if there are multiple values for which the minimum is achieved.

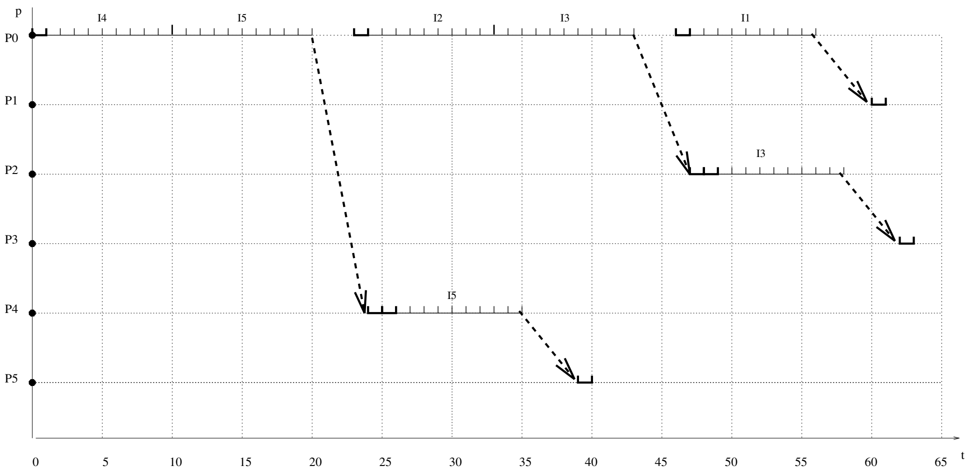


Figure 9: The schedule for scatter problem given by Algorithm A_t^k under the full LogGP model. The parameters are $L = 4, o = 1, g = 4, G = 1, P = 6$, and each processor receives $k = 10$ items. The overhead o is represented by a thick box, while the sending of each byte is represented by a thinner box. The algorithm finishes at time $t = 63$.

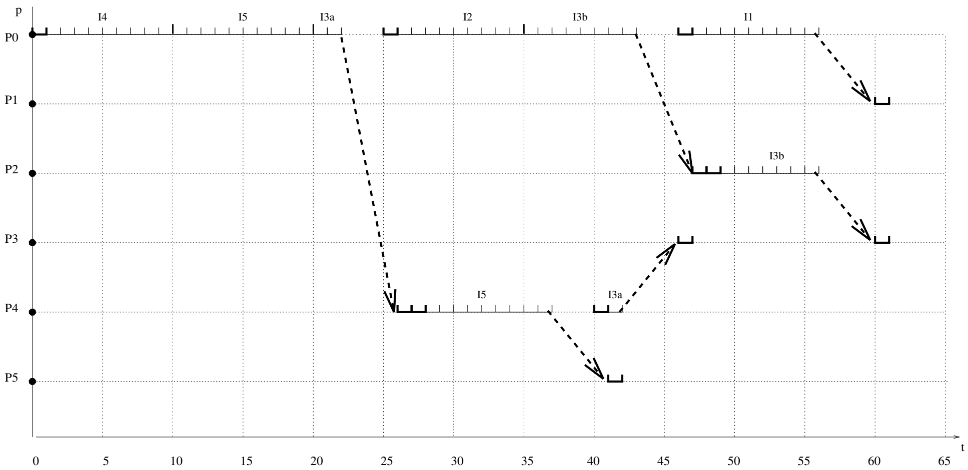


Figure 10: A modification of the schedule shown in Figure 9. The message for processor $P3$ is split and handled by two intermediate processors, which results in a shorter schedule. It finishes at time $t = 61$, two time steps less than the algorithm without splitting.

Similar to the $k = 1$ case, the algorithm is determined by choosing the best split ratio $\frac{s}{P}$ at each step.

Algorithm A_t^k is optimal among all algorithms that do not split item sets among messages. In general, though, this is not optimal since there are cases in which splitting item sets can produce a slightly faster algorithm. An example is shown in Figures 9 and 10. Figure 10 shows an "optimal" schedule based on the algorithm A_t^k, when 10 items are scattered ($k = 10$) among 6 processors. The LogGP parameters are $L = 4, o = 1, g = 4, G = 1, P = 6$, and the schedule finishes at time $t = 63$. The critical path in this case involves processor $P2$ sending 10 items to processor $P3$. If some of those items are sent by processor $P5$ instead, we derive a faster schedule. This case is shown in Figure 10, were 2 of the 10 items destined for processor $P3$ are sent to $P5$, while the remaining 8 items are still handled via processor $P2$. The total time for this schedule is $t = 61$ clock cycles, two cycles less than the algorithm without splitting.

4.8 Bounding the Optimal k-Item Algorithm

In this section we relate the running time of the optimal algorithm for the k-item scatter to the Binomial Tree algorithm for the special case when $L = g$ (in the simplified LogGP model, or equivalently $L + 2o = g$ in the complete LogGP model). Even though we do not know the running time of the optimal k-item scatter, we show that, at least when $L = g$, it is within a constant factor of the Binomial Tree algorithm's running time.

In order to prove this, we need the exact running time of the Binomial Tree algorithm for arbitrary P. We prove this for the case when $L \geq g$. The proof could easily be generalized to include the case $L < g$, but this encumbers the equations and is not needed for the main result of this subsection.

Lemma 2 *If $L \geq g$, the Binomial Tree algorithm runs in time $t^k(P) = L\lceil \log_2 P \rceil + (P-1)k - \lceil \log_2 P \rceil$.*

Proof: The Binomial Tree algorithm is obtained from the optimal algorithm by choosing $s = \lfloor \frac{P}{2} \rfloor$ at each step instead of selecting the optimal s. Thus, the recurrence relation for the time complexity of the Binomial Tree algorithm is:

$$\begin{aligned} t^k(1) &= 0 \\ t^k(P) &= \left\lfloor \frac{P}{2} \right\rfloor k - 1 + L + t^k(\left\lceil \frac{P}{2} \right\rceil) \end{aligned}$$

Let $f^k(P) = L\lceil \log_2 P \rceil + (P-1)k - \lceil \log_2 P \rceil$. We prove by induction that $t^k(P) = f^k(P)$.

Base: For $P = 1$, $t^k(1) = 0 = f^k(1)$ by definition.

Hypothesis: Suppose that $t^k(P) = f^k(P)$ for all $P < n$.

Induction step: Let $P = n$. We consider two cases, depending on whether n is even or odd.

If n is even, then $\lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil = \frac{n}{2}$. Therefore

$$\begin{aligned} t^k(n) &= \frac{n}{2}k - 1 + L + t^k(\frac{n}{2}) \\ &= \frac{n}{2}k - 1 + L + L\lceil \log_2 \frac{n}{2} \rceil + (\frac{n}{2} - 1)k - \lceil \log_2 \frac{n}{2} \rceil \\ &= L(1 + \lceil \log_2 \frac{n}{2} \rceil) + (n-1)k - (1 + \lceil \log_2 \frac{n}{2} \rceil) \end{aligned}$$

Note that $1 + \lceil \log_2 \frac{n}{2} \rceil = \lceil 1 + \log_2 \frac{n}{2} \rceil = \lceil \log_2 2\frac{n}{2} \rceil = \lceil \log_2 n \rceil$. We therefore proved that $t^k(n) = f^k(n)$ for even n.

If n is odd, then $\lfloor \frac{n}{2} \rfloor = \frac{n-1}{2}$ and $\lceil \frac{n}{2} \rceil = \frac{n+1}{2}$. Therefore

$$\begin{aligned} t^k(n) &= \frac{n-1}{2}k - 1 + L + t^k(\frac{n+1}{2}) \\ &= \frac{n-1}{2}k - 1 + L + L\lceil \log_2 \frac{n+1}{2} \rceil + (\frac{n+1}{2} - 1)k - \lceil \log_2 \frac{n+1}{2} \rceil \\ &= L(1 + \lceil \log_2 \frac{n+1}{2} \rceil) + (n-1)k - (1 + \lceil \log_2 \frac{n+1}{2} \rceil) \\ &= L\lceil \log_2 (n+1) \rceil + (n-1)k - \lceil \log_2 (n+1) \rceil \\ &= L\lceil \log_2 n \rceil + (n-1)k - \lceil \log_2 n \rceil = f^k(n) \end{aligned}$$

Note that $\lceil \log_2 n \rceil = \lceil \log_2 (n+1) \rceil$ because n is odd in this case.

Thus, we proved that $t^k(n) = f^k(n)$ for both even and odd n. It follows by induction that $t^k(P) = f^k(P)$ for all values of P. □

Again, the proof could be generalized to include the case when $L < g$, essentially by substituting $\max\{L, g\}$ for all occurrences of L, and accounting for the very last message send, which always involves

L, not g. The complete formula for the general case becomes: $t^k(P) = \max\{L, g\}(\lceil \log_2 P \rceil - 1) + L + (P-1)k - \lceil \log_2 P \rceil$. However, we only need the simpler formula for the proof below.

Now we proceed to prove that the Binomial Tree algorithm is within $L-1$ of the optimal algorithm for the special case where $L = g$. Note that in this case, the Binomial Tree is optimal for the scatter of $k = 1$ item (Theorem 1). That is, algorithm A_t^k reduces to the Binomial Tree algorithm for the special case when $g = L$ in the simplified LogGP model (or equivalently $g = L + 2o$ in the full model). Based on this and Lemma 2 we prove the following:

Theorem 2 *Given that $L = g$, the running time of any any k-item scatter algorithm is at least $t^k(P) - (L-1)$, where $t^k(P) = L\lceil \log_2 P \rceil + (P-1)k - \lceil \log_2 P \rceil$ is the running time of the Binomial Tree algorithm.*

Proof: Let B_u^k be an arbitrary scatter algorithm for k items that runs in time $u^k(P)$. Using $B_u^k(P)$ (i.e., the algorithm for P processors), we construct a new algorithm $C_v^1(kP)$ which is a single item scatter algorithm for kP processors. C_v^1 works in two phases. First, it splits the kP processors into P logical groups of k processors each. The processors in group i, $0 \le i \le P-1$, have numbers $G_i = \{ik, ik+1, ik+2, \ldots, ik+(k-1)\}$. Each group G_i has a designated leader, processor number ik.

The first phase of C_v^1 is a k-item scatter that involves only the leaders of the P groups. Each leader receives the k items destined for the processors in its group. This phase is a run of algorithm $B_u^k(P)$.

The second phase consists of P independent single-item scatter trees that run in parallel: the leader distributes the k items among the k processors in its group. For this phase we use the optimal single-item scatter algorithm A_t, which happens to be precisely the Binomial Tree for $L = g$.

The total running time of C_v^1 is the sum of the times for the two phases:

$$v(kP) \quad = \quad u^k(P) + t^1(k)$$

Note that C_v^1 is a single-item scatter algorithm for kP processors and therefore runs no faster that $t^1(kP)$ (the running time of the Binomial Tree algorithm for 1 item and kP processor). Thus

$$\begin{aligned}
u^k(P) &= v(kP) - t^1(k) \\
&\ge t^1(kP) - t^1(k) \\
&= (L\lceil \log_2 kP \rceil + (kP-1) - \lceil \log_2 kP \rceil) \\
&\quad -(L\lceil \log_2 k \rceil + (k-1) - \lceil \log_2 k \rceil) \\
&= (L-1)(\lceil \log_2 kP \rceil - \lceil \log_2 k \rceil) + (P-1)k \\
&= (L-1)(\lceil \log_2 k + \log_2 P \rceil - \lceil \log_2 k \rceil) + (P-1)k \\
&\ge (L-1)(-1 + \lceil \log_2 k \rceil + \lceil \log_2 P \rceil - \lceil \log_2 k \rceil) + (P-1)k \\
&= -(L-1) + (L-1)\lceil \log_2 P \rceil + (P-1)k \\
&= t^k(P) - (L-1)
\end{aligned}$$

Since B_u^k was an arbitrary k-item scatter algorithm, any k-item scatter algorithm runs in time at least $t^k(P) - (L-1)$.□

4.9 Discussion

There has been a substantial amount of work on developing parallel algorithms for broadcast problems given only point-to-point communication primitives under a fully-connected model. For example, the single item and k-item (non-personalized) broadcast problem has been studied under the Postal [BNK92] and LogP model [KSSS93]. The scatter problem has received much less attention. It has been studied for specific topologies such as the mesh and hypercube assuming fixed size packets [JH89, SS89, BOS$^+$91].

The Binomial Tree algorithm for the scatter problem is not new. In [JH89] the problem was studied assuming a hypercube network and a linear model for large message transfer ($\tau + nt_c$). Their binomial tree algorithm is identical to ours if they assume a sufficiently large packet size, and thus, has the same complexity. They prove that their algorithm's execution time is within a factor of 2 of optimal for the hypercube network.

To our knowledge, our single-item optimal algorithm A_t and its k-item, extension A_t^k, have not been previously proposed. These algorithms are in general faster than the Binomial Tree under LogGP due to the difference between L and g which allows the sending processor to initiate a new transfer before the receiving processor can start transmitting.

4.10 Experimental Results

To verify the accuracy of the LogGP model for the scatter problem, we implemented the three long-message scatter algorithms (Simple Long-Message, Binomial Tree, and optimal) on the Meiko CS-2 and measured their running time on 64 processors for different number of items k. The algorithms are implemented in C using Meiko's Elan communication library (see Table 1). The results are shown in Figure 11 where both measured and predicted times are plotted. The figure shows that the predicted times closely match the measured. One can note that the optimal algorithm is exactly the Binomial Tree algorithm for the set of parameters for a 64 processor Meiko CS-2. These two algorithms differ slightly in the measured graph due to experimental error. To detect a significant difference between the optimal and Binomial Tree algorithms, we would need a larger number of processors, at least on the Meiko CS-2.

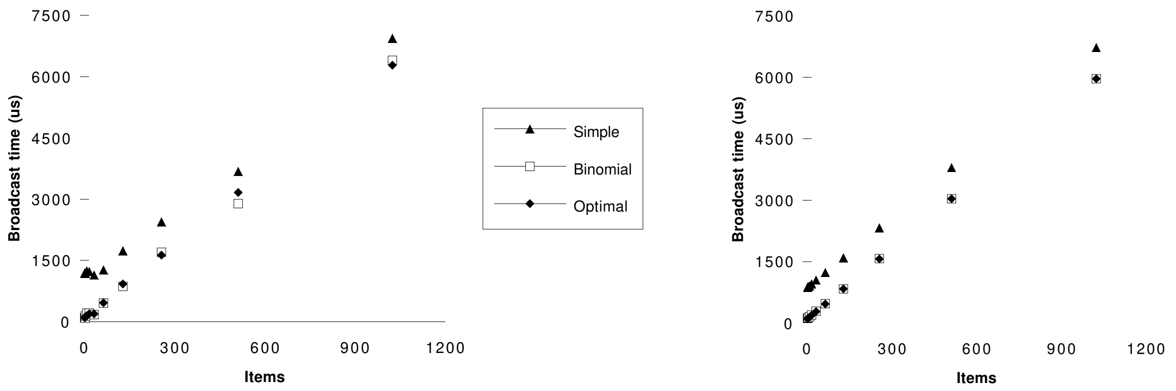


Figure 11: *Measured (left graph) and predicted (right graph) times for the scatter algorithms for $P = 64$ processors. The algorithms are implemented using the Elan communications library (see Table 1).*

The results of a second experiment are shown in Figure 12. We implemented a scatter algorithm that approximates the optimal one by always choosing a fixed splitting ratio *alpha*. The figure shows a close match between predicted and measured times. Observe that the optimal *alpha* is 0.5 which corresponds to the Binomial Tree algorithm.

5 Conclusions

In this paper we study the impact of long messages on algorithm design and propose a refinement of the LogP model, the LogGP model, that better captures the advantages of bulk transfers. Many existing parallel machines have special support for long messages and achieve a substantially higher bandwidth for long messages compared to short messages. The LogP model ignores long messages and, therefore, is not

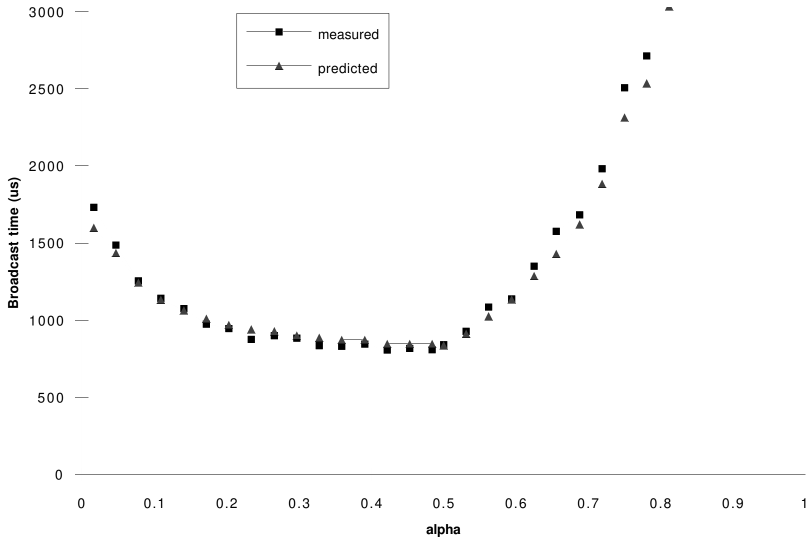


Figure 12: *Measured and predicted times for the scatter algorithm using a fixed splitting ratio equal to* alpha *for* $k = 128$ *items (512 bytes) on* $P = 64$ *processors. The implementation uses the Elan communications library.*

adequate for modeling such machines. The LogGP model addresses this shortcoming by introducing a new parameter, G, which captures the bandwidth for long messages. The ratio g/G is an excellent indicator of the gain that can be obtained by grouping data into long messages. We believe that LogGP is a realistic model which captures the most important features of modern parallel machines. Furthermore, it is still simple enough to be used for algorithm design and analysis, especially since it is often possible to work only with a subset of the parameters.

In this paper we specify the LogGP model and use it to design and analyze several algorithms including FFT, radix sort, and the single node scatter problem. Experimental results collected on the Meiko CS-2 show the value of the model in designing algorithms and predicting their execution times. For the scatter problem we present several algorithms that are fundamentally different from, and significantly faster than, the LogP (short-message) optimal algorithm. We prove optimality for our single-item scatter algorithm. Although experimental results are primarily presented for Meiko CS-2, the same model is applicable to other existing parallel machines that support long messages, such as the IBM SP-2, Paragon, and Ncube. We feel that the LogGP model will serve as an important tool for both algorithm design and analysis.

Acknowledgments

We would like to thank Abhijit Sahay, Seth Copen Goldstein, Martin Rinard, Pedro Diniz, and the anonymous referees for their comments. Computational support at UCSB was provided by the NSF Infrastructure Grant number CDA-9216202, with support from the College of Engineering and UCSB Office of Research. Klaus Erik Schauser received research support from the Department of Computer Science at UCSB.

References

[AC94] B. Alpern and L. Carter. Towards a Model for Portable Parallel Performance: Exposing the Memory Hierarchy. In T. Hey and J. Ferrante, editors, *Portability and Performance for Parallel Processing*. Wiley, 1994.

[ACS89] A. Aggarwal, A. K. Chandra, and M. Snir. On Communication Latency in PRAM Computation. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*. ACM, June 1989.

[ACS90] A. Aggarwal, A. K. Chandra, and M. Snir. Communication Complexity of PRAMs. In *Theoretical Computer Science*, March 1990.

[BBB+94] V. Bala, J. Bruck, R. Bryant, R. Cypher, P. de Jong, P. Elustondo, D. Frye, A. Ho, C-T. Ho, G. Irwin, S. Kipnis, R. Lawrence, and M. Snir. The IBM external user interface for scalable parallel systems. *Parallel Computing*, 20(4), April 1994.

[BBC+94] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C-T. Ho, S. Kipnis, and M. Snir. CCL: a portable and tunable collective communication library for scalable parallel computers. In *8th International Parallel Processing Symposium*, April 1994.

[BCM94] E. Bartson, J. Cownie, and M. McLaren. Message passing on the Meiko CS-2. *Parallel Computing*, 20(4), April 1994.

[Ble87] G. E. Blelloch. Scans as Primitive Parallel Operations. In *Proceedings of International Conference on Parallel Processing*, August 1987.

[BNK92] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, June 1992.

[BOS+91] D. P. Bertsekas, C. Ozveran, G. D. Stamoulis, P. Tseng, and J. N. Tsitsiklis. Optimal Communication Algorithms for Hypercubes. *Journal of Parallel and Distributed Computing*, 11, 1991.

[BT89] D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation*. Prentice Hall, 1989.

[CDG+93] D. E. Culler, A. Dusseau, S. C. Golstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. of Supercomputing*, November 1993.

[CDMS94] D. E. Culler, A. Dusseau, R. Martin, and K. E. Schauser. Fast Parallel Sorting under LogP: from theory to practice. In T. Hey and J. Ferrante, editors, *Portability and Performance for Parallel Processing*. Wiley, 1994.

[CKL+94] D. E. Culler, K. Keeton, L. T. Liu, A. Mainwaring, R. Martin, S. Rodrigues, and K. Wright. Generic Active Message Interface Specification. UC Berkeley, November 1994.

[CKP+93] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[CZ89] R. Cole and O. Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *Proceedings of the Symposium on Parallel Architectures and Algorithms*, June 1989.

[FW78] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the 10th Annual Symposium on Theory of Computing*, May 1978.

[Gib89] P. B. Gibbons. A More Practical PRAM Model. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*. ACM, June 1989.

[HM93] M. Homewood and M. McLaren. Meiko CS-2 Interconnect Elan-Elite Design. In *Proc. of Hot Interconnects*, August 1993.

[Hoc94] R. Hockney. Performance Parameters and Results for the Genesis Parallel Benchmarks. In T. Hey and J. Ferrante, editors, *Portability and Performance for Parallel Processing*. Wiley, 1994.

[JH89] S. L. Johnsson and C. T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9), September 1989.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*. Benjamin Cummings, 1994.

[KLMadH92] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM Simulation on a Distributed Memory Machine. In *Proceedings of the Twenty-Fourth Annual ACM Symposium of the Theory of Computing*, May 1992.

[KR90] R. M. Karp and V. Ramachandran. Parallel Algorithms for Shared-Memory Machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier Science Publishers, 1990.

[KSSS93] R. Karp, A. Sahay, E. Santos, and K. E. Schauser. Optimal Broadcast and Summation in the LogP Model. In *5th Symp. on Parallel Algorithms and Architectures*, June 1993.

[LC94] L. T. Liu and D. E. Culler. Measurements of Active Messages Performance on the CM-5. Technical Report UCB/CSD 94-807, CS Div., UC Berkeley, May 1994.

[LCW94] J. R. Laurus, S. Chandra, and D. A. Wood. CICO: A Practical Shared-Memory Programming Performance Model. In T. Hey and J. Ferrante, editors, *Portability and Performance for Parallel Processing*. Wiley, 1994.

[Lei92] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufman, 1992.

[MV84] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21(4), November 1984.

[Pie94] P. Pierce. The NX message passing interface. *Parallel Computing*, 20(4), April 1994.

[PY88] C. H. Papadimitriou and M. Yannakakis. Towards an Architecture-Independent Analysis of Parallel Algorithms. In *Proceedings of the Twentieth Annual ACM Symposium of the Theory of Computing*. ACM, May 1988.

[Sny86] L. Snyder. Type Architectures, Shared Memory, and the Corollary of Modest Potential. In *Ann. Rev. Comput. Sci.* Annual Reviews Inc., 1986.

[SS89] Y. Saad and M. H. Schultz. Data communication in hypercubes. *Journal of Parallel and Distributed Computing*, 6(1), February 1989.

[SS95] K. E. Schauser and C. J. Scheiman. Experience with Active Messages on the Meiko CS-2. In *9th International Parallel Processing Symposium*, April 1995.

[SV94] M. Schmidt-Voigt. Efficient parallel communication with the nCUBE 2S processor. *Parallel Computing*, 20(4), April 1994.

[TM94] L. W. Tucker and A. Mainwaring. CMMD: Active messages on the CM-5. *Parallel Computing*, 20(4), April 1994.

[Val90] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8), August 1990.

[vECGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, May 1992.