Incorporating Cache Effects into a Model of Interprocessor Communication[*]

Ibrahim Hur Calvin Lin
{*ihur, lin*}*@cs.utexas.edu*
The University of Texas at Austin
Austin, Texas 78712

September 19, 1999

Abstract

This paper presents a new communication cost model that incorporates cache effects by estimating the number of unique *memory lines touched* (MLT) when marshalling data. This term improves accuracy because MLT is sensitive to data layout and the order in which data is touched.

We derive formulae to compute MLT, and we build models for both point-to-point communication and parallel prefix operations. Results for benchmarks executing on three parallel computers—the Cray T3E, the IBM SP-2, and the Sun Enterprise 5000—show that our model is significantly more accurate than previous models. Careful statistical analysis confirms these findings. We conclude by showing that our approach can be used to guide program development, as our model can predict the performance of two parallel programs that exhibit a variety of performance tradeoffs, accurately identifying their crossover point.

Keywords. performance modeling, communication cost, cache effects

1 Introduction

Parallel performance modeling and prediction have many uses in algorithm development, compiler optimization, and performance tuning. Unfortunately, such predictions are complicated by the existence of caches, which are notoriously difficult to model because they have complex behavior that depend on many variables. In addition to hardware properties such as line size, cache size, set associativity, and cache miss times, program characteristics such as the relative alignment of data and the order in which data is touched affect the number of cache misses and, ultimately, the application's performance. While many researchers have attempted to model particular aspects of cache behavior for specific purposes—for example, to analyze conflict misses of blocked algorithms [28]—no one has successfully incorporated cache effects into a model of communication performance that can be used for performance prediction.

This paper[1] presents a simple approach to modeling the cache effects of interprocessor communication operations in parallel programs. The key idea is to capture cache effects by understanding how data is laid out in memory and how it is accessed. In particular, we estimate the number of unique memory lines accessed when marshalling data. Therefore, in contrast to previous models, which typically include a term for message startup cost, α, and a term for per-byte transmission cost, β, our model includes a third term that is proportional to the number of memory lines accessed. Here, a *memory line* is defined to be a cache-line-sized block of memory. While other models attempt to include marshalling costs [19, 21], our model is unique in its ability to capture the cache effects of marshalling costs, which can be significant.

To construct our model we first derive a closed form formula for MLT, the number of memory lines accessed when marshalling a message. Once MLT is known, our model can be stated as $\alpha + \beta n + \gamma MLT$, where n is the message length. We can imagine that the third term, which captures cache effects, is a more involved function of MLT, and

[*] This work was supported in part by an NSF Research Infrastructure Award CDA-9624082 and ONR grant N00014-99-1-0402.

[1] A considerably compressed version of this paper has appeared elsewhere [20].

we explore this possibility. To complete our model, we use a training set to measure actual costs for various message sizes and layouts. From these measured values we use curve fitting techniques to determine machine-specific values of α, β, and γ. Thus calibrated, our model can be used to predict communication costs by providing specific values of n and MLT.

This paper makes the following contributions: (1) We introduce a new communication model that includes cache effects; (2) we derive equations to compute MLT for rectangular blocks of data; (3) we demonstrate that the addition of the MLT feature leads to a significantly more accurate model; and (4) we show that our model can be practically applied to guide parallel program construction.

We demonstrate the superiority of our model by first applying it to point-to-point communication operations, since these are the basis upon which all other communication models can be built. To show that our approach works for different machines, we apply it to three very different parallel computers: the Cray T3E, the IBM SP-2 and the Sun Enterprise 5000. To show that our approach works for different languages, we apply it to benchmarks written in MPI [10] and in the ZPL parallel programming language [7, 32]. To show the general utility of our approach, we apply our model to parallel prefix (scan) operations [26], which are more challenging because they include both communication and computation. Finally, to show that our model can be used to guide program design, we show that it can accurately model the performance of two ZPL programs that solve the same problem. In particular, we show that our model accurately predicts the crossover point between these two programs.

This paper is organized as follows. The next section presents related work. Section 3 explains our model, its assumptions, and how it is customized to individual machines. Section 4 provides background material, and is followed by three sections that describe experiments in modeling point-to-point communication, parallel prefix operations, and complete ZPL programs, respectively. We close with some concluding remarks and directions for future research.

2 Related Work

There has been considerable work in parallel performance modeling, including work that uses very different approaches from ours. These include interactive tools [5], task graph-based models [30], queueing theory [22, 23], and trace-based approaches [1, 31]. Others have modeled specific aspects of uniprocessor cache behavior. For example, conflict misses are often used to guide tiling and other loop transformations [28, 34, 8].

Most models of communication performance do not consider cache effects. For example, Ivory [21] creates a model that considers the effects of data copying, but ignores the cache, so the prediction results are inaccurate for non-contiguous data accesses. Fahringer's extensive work on performance prediction [11, 12, 13, 14] includes a static communication cost model for HPF [13], but his method does not consider the effects of cache misses.

Others have modeled specific aspects of cache behavior to guide program optimizations, but none of these is sensitive to data layout. For example, Temam et al. [33] develop an analytical model that focuses primarily on conflict misses of direct mapped caches, and Ferrante et al. [15] derive an upper bound for the number of unique memory lines accessed in a loop.

Recent work by Ghosh et al. [16, 17, 18] develops a new technique, Cache Miss Equations, to estimate precisely both cold misses and replacement misses in loop-oriented scientific codes for uniprocessors. This method extends the notion of reuse analysis to generate linear Diophantine equations that summarize the memory behavior of loops, but it is developed for virtually addressed one level caches. As the authors have noted, extending this technique to deeper cache hierarchies will result in much more complex equations. Also, estimates for conflict misses with the real addressed caches will not be accurate. Finally, speed is a drawback of this approach, as it is reported to be “potentially faster” than simulation.

Amato et al [2] use a training set approach that is similar to ours, but their method of modeling cache behavior is significantly different. They assume different costs for read and write operations, which they use to generate upper and lower bounds for their cost function. Their method is insensitive to differences in data layout, and their method produces upper and lower performance bounds that can differ by as much as two orders of magnitude, which precludes accurate predictions.

3 Our Solution

Figure 1 illustrates two problems in predicting the communication cost of parallel machines. The left graph shows that performance is non-linear due to cache effects: The locations of the discontinuities reflect increases in miss rates for the

L1 cache. The right graph shows that communication behavior is often complicated by other hardware features. Here, the T3E's stream buffers can make it faster to send larger amounts of data than smaller amounts of data, as seen for the $d = 1$ curve. These features, along with hardware features such as data prefetching mechanisms, make analytical modeling of communication almost impossible. From the left graph we conclude that we must incorporate cache effects into any accurate model of communication. From the right graph we conclude that a "benchmarking model" [12] is needed. Benchmarking models first identify a number of significant features to model, and then run a training set on the target machine. By fitting curves to the measured data, we can determine machine-specific coefficients to our basic model. An advantage of this method is that it captures the actual behavior of the system, including all the complexities of the hardware.

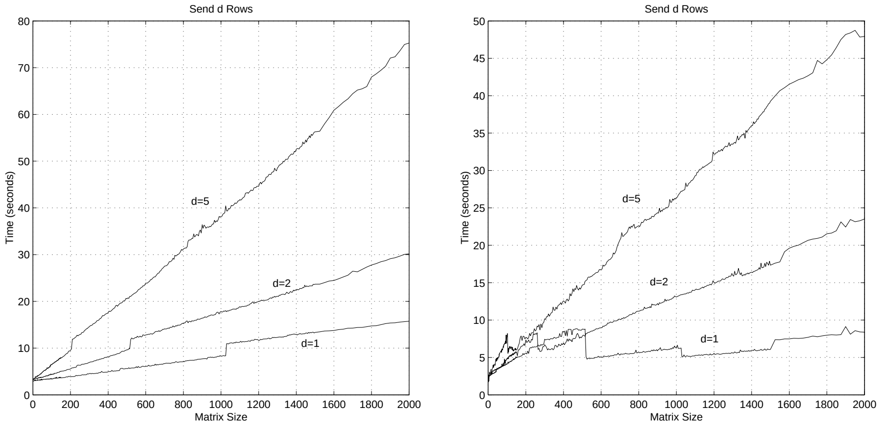


Figure 1: Communication cost on the IBM SP-2 (left) and the Cray T3E (right) for sending d rows of a 2000 × L matrix, as L varies from 1 to 2000. (Results were gathered using a benchmark written in ZPL.)

The key issue here is the determination of the model features. Our approach uses three features: the number of messages sent (m), the message size (n), and the number of unique memory lines touched (MLT). In particular, we model execution time, t, as follows:

$$t = \alpha m + \beta n + f(MLT) \tag{1}$$

Communication costs are often given as a per-message cost, which can be stated as follows:

$$t = \alpha + \beta n + f(MLT) \tag{2}$$

Here, α is a machine-specific value for the per message overhead and β is a machine-specific per-byte cost. We conjecture that $f(MLT)$ represents cache miss behavior as a simple function of the number of lines touched, most likely of the form γMLT, but we will explore other possibilities as well (see Section 4).

3.1 Why Our Model Works

To understand why our model works, consider the transmission of two n-byte messages, one contiguous in memory and the other not. For example, the contiguous message might correspond to a row of a matrix and the non-contiguous message might correspond to a column of a matrix, assuming memory is allocated in row major order. The conventional model ($\alpha + \beta n$) does not distinguish between these two messages, yet the row-transfer will fill the cache with useful data while the column-transfer will only use one word of useful data per cache line, assuming that the width of the matrix is larger than one cache line. Our solution is to compute MLT, the number of distinct cache lines to which the message would be mapped. Our model should capture the effects of both compulsory misses and capacity misses, since both are proportional to the number of unique memory lines touched. Our model does ignore other features, such as conflict misses, that might improve the accuracy of our model at the expense of increased model complexity. We choose to focus on compulsory and capacity misses to see if a simple model will suffice.

To confirm the significance of including cache effects and data layout in a performance model, we measured the performance difference for row transfers and column transfers. Figure 2 shows these results on the IBM SP-2 (using our ZPL benchmark). We show results for $d = 1, d = 2$ and $d = 5$. The left graph shows the time for sending columns where $1 \leq L \leq 4000$, and the middle graph shows the time for sending rows. The arrays hold 4 byte integers, and the cache line size is 64 bytes, so when one column is transferred ($d = 1$), only 4 bytes of each cache line will hold useful data, while 60 bytes will be loaded from memory unnecessarily. By contrast, when one row is transferred, the entire cache line will hold useful data, so we expect 16 times as many cache misses when sending one column than when sending one row. This factor of 16 in cache misses translates to a factor of 8 in overall communication performance, as seen in the right graph of Figure 2, which shows the ratio of column transfer time to row transfer time.

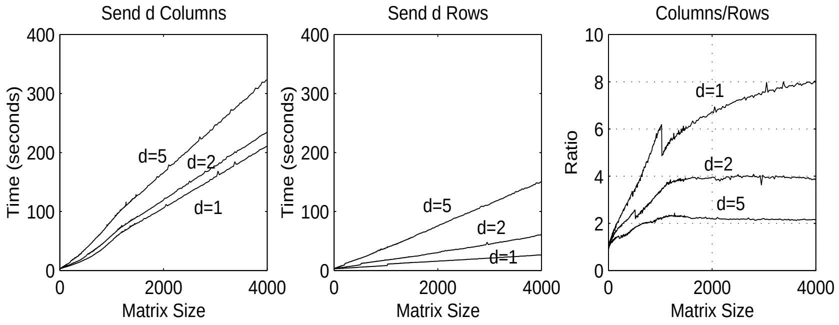


Figure 2: Measured results for our ZPL benchmark on the IBM SP-2.

3.2 Calculating the Model Parameters

This section examines the parameters of our model for messages that correspond to rectangular slices of 2D arrays. These results can be generalized in a straightforward manner to higher dimensional arrays and arbitrary paralleliped slices. We will use the following assumptions and definitions:

The processor mesh is assumed to be $P_r \times P_c$, where $P_r \geq 1$ and $P_c \geq 1$.
p: total number of processors ($p = P_r \cdot P_c$)
b_r: height of the data on one processor ($b_r = \lceil \frac{L_R}{P_r} \rceil$)
b_c: width of the data on one processor ($b_c = \lceil \frac{L_C}{P_c} \rceil$)
l**:** cache line size

We assume that each communication operation is performed for a matrix of size $L_R \times L_C$, i.e. L_R rows and L_C columns (in bytes), and we assume that data block and matrix sizes are given in bytes. For these rectangular messages, the number of bytes (n) transmitted from one processor to another is $n = b_c d$ for the transfer of d rows, and $n = b_r d$ for the transfer of d columns.

3.2.1 Number of Unique Memory Lines Touched, MLT

This section derives formulae to calculate MLT, the number of unique memory lines touched when accessing d rows or d columns of an array. Two communication operations that send the same amount of data can have very different values of MLT depending on whether rows or columns are transferred. Therefore we investigate these two cases separately. In addition to the definitions and assumptions of the previous sections, we add the following assumptions:

All the processors hold data of the same size.
For the column transfers $b_c > 2l$.
Memory is allocated in row major order.

For row transfers MLT is trivial to calculate. We simply divide the number of bytes transferred by the cache line size, l. However, since there is no way to know if the data to transfer starts on a cache line boundary, we give lower and upper bounds, LB_R and UB_R.

$$LB_R = \left\lfloor \frac{b_c d}{l} \right\rfloor \tag{3}$$

$$UB_R = \left\lceil \frac{b_c d}{l} \right\rceil + 1 \tag{4}$$

We can then use $MLT_R = (LB_R + UB_R)/2$ to approximate MLT for row transfers.

Calculating MLT for column transfers is more complicated. The intuition behind our derivation is as follows. If only one column is transmitted, there will be b_r distinct lines accessed, because there are b_r rows in the array and each row will be touched once. If the leftmost d columns of an array are transmitted, where $d > 1$, the number of cache lines touched would be b_r if each row fell within a single cache line. The number of cache lines touched will be larger than b_r if any row spans both sides of a cache line boundary. For example, Figure 3 shows how the elements of an array might map to different cache lines; we see that when $d = 9$, the first row of the data spans two distinct cache lines. Thus, to count the number of cache lines touched, we need to compute the number of cache line boundaries that reside within the first d columns of data; this number can be added to b_r to get the total number of cache lines touched. Theorem 1 will show that the horizontal spacing between these cache line boundaries is a constant, q, that can be easily computed, and that the value of q is constant across any d contiguous columns of data. Theorem 2 will then use q to compute lower and upper bounds for the number of cache lines touched.

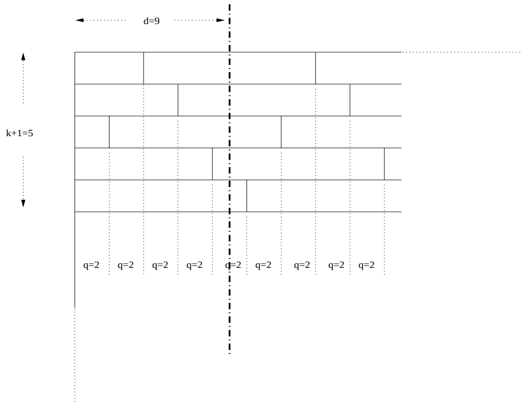


Figure 3: Relations among the line boundaries in the rows where $l = 10$, $q = \gcd(b_c, l) = 2$. Only the first group is shown. See the proof of Theorem 2 for more explanation.

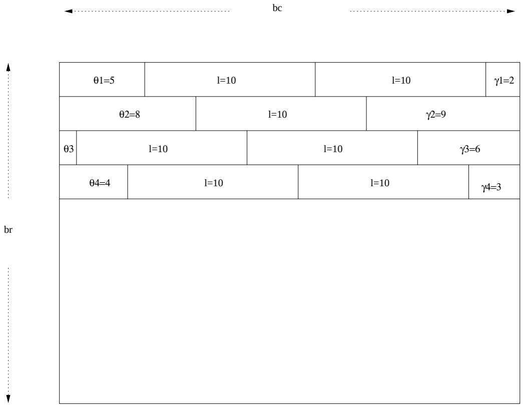


Figure 4: This figure shows the alignment of the first four lines in an array where $b_c = 27$, $l = 10$. Notice the wrap around at the end of the rows. The first byte of the array starts at the sixth byte of the first line. See the proof of the Theorem 2 for more explanation.

To develop lower and upper bounds, LB_C and UB_C, for MLT, we use the definition of addition modulo l to define the *additive group modulo* l as $(Z_l,+_l)$. The size of this group is $|Z_l| = l$. Consider an element b_c. The subgroup that can be generated from b_c by the group operation, denoted by $< b_c >$, is defined as

$$< b_c >= \{{b_c}^{(m)} : m \geq 1\},$$

where

$${b_c}^{(m)} = mb_c \pmod l.$$

We can now state the following theorem.

Theorem 1 *For any positive integers b_c and l, if $q = \gcd(b_c, l)$, then $< b_c >= \{0, q, 2q, ..., ((l/q) - 1)q)\}$, and $| < b_c > | = l/q$.*

Proof A proof of this theorem is given by Cormen et al.[2] [9].

We can now derive closed form formulae for lower and upper bounds for column transfers.

Theorem 2 *For a rectangular array with b_r rows and b_c columns, which is stored in row-major order in memory, the lower and upper bounds, LB_C and UB_C, for the number of distinct lines accessed, MLT, when transferring d columns of the array are*

$$LB_C = \left\lfloor \frac{b_r \gcd(b_c, l)}{l} \right\rfloor \left(\frac{l}{\gcd(b_c, l)} + \left\lceil \frac{d}{\gcd(b_c, l)} \right\rceil - 1 \right) \tag{5}$$

$$UB_C = \left\lceil \frac{b_r \gcd(b_c, l)}{l} \right\rceil \left(\frac{l}{\gcd(b_c, l)} + \left\lceil \frac{d}{\gcd(b_c, l)} \right\rceil \right) \tag{6}$$

where l is the number of bytes in a cache line, and $\gcd(b_c, l)$ is the greatest common divisor of b_c and l.

As mentioned above, when multiple columns are transmitted, the number of distinct lines touched will depend on the alignment of the lines in the array. The following proof shows that the effect of this alignment can be calculated exactly.

Proof Define θ_i and γ_i to be the number of bytes in the first and last memory lines in row i, respectively (See Figure 4). Notice that we make no assumptions about the position of the first byte of the array. Since $\theta_1 \leq l$, we can define γ_i and θ_i as follows:

$$\gamma_i = (ib_c - \theta_1) \pmod l = ib_c \pmod l - \theta_1, i \geq 1$$
$$\theta_i = (l - \gamma_{i-1}) = ib_c \pmod l + (l - \theta_1), i \geq 2$$

Because of modulo arithmetic we can always replace a value with its smallest nonnegative equivalent in the same equivalence class. By Theorem 1, we can conclude that

$$\theta = \{(l - \theta_1), q + (l - \theta_1), 2q + (l - \theta_1), ..., kq + (l - \theta_1)\}$$
$$\gamma = \{-\theta_1, q - \theta_1, 2q - \theta_1, ..., kq - \theta_1\}$$

Since $-\theta_1 = (l - \theta_1) \pmod l$ we obtain

$$\theta \equiv \gamma = \{t, q + t, 2q + t, ..., kq + t\}$$

where t is a constant, $q = \gcd(b_c, l)$, and $k = (l/q) - 1$.

[2] Section 33.4, page 820.

In other words, the difference between the consecutive elements of the sets θ and γ is $q = \mathrm{gcd}(b_c, l)$. Since these are sets, the i^{th} element in these sets does not necessarily correspond θ_i or γ_i. Clearly, if $b_r > k$, the θ_i's and γ_i's will repeat. The lower and upper bounds for the number of the repeating groups are $\lfloor A \rfloor = \lfloor b_r/(k+1) \rfloor$ and $\lceil A \rceil = \lceil b_r/(k+1) \rceil$.

Now, consider the θ_i's of the first group. Assume that we want to transfer the d leftmost columns of the array. If $d = 1$ then for one group the number of distinct lines accessed is $MLT_1 = k + 1$. If $d > 1$ then MLT_1 is increased once at every q increase of d. Therefore, for one group

$$MLT_1 = \frac{l}{\mathrm{gcd}(b_c, l)} + \left\lceil \frac{d}{\mathrm{gcd}(b_c, l)} \right\rceil - 1$$

We need to add 1 to MLT_1 to make it an upper bound for the cases where the array does not start on a line boundary. Hence, $LB_C = \lfloor A \rfloor MLT_1$ and $UB_C = \lceil A \rceil (MLT_1 + 1)$. If $(k+1)$ divides b_r and the array starts on the line boundary then the lower and upper bounds become equal.

□

4 Background and Methodology

This section explains how we generate performance models for specific parallel computers. We first briefly describe the architectures used in our experiments. We then explain how we develop linear regression models for point-to-point communication cost, and we discuss the metrics used to statistically evaluate our models. Although the discussion here focuses on the modeling of point-to-point communication costs, the same principles apply to the other operations as well.

4.1 Architectures

Our experiments use three parallel machines, the IBM SP-2, the Cray T3E, and the SUN E5000. The IBM SP-2 consists of 16 RS-6000 nodes each running at 67MHz. Each node has two levels of caches. The communication channel has a peak rate of 40 MB/second. The Cray Research T3E employs 68 nodes which are embedded in a three-dimensional toroidal memory interconnect. Each node has 8KB data and 8KB instruction caches, and a 96KB on chip level two cache. The communication channel has a peak bandwidth of 300 MB/second. The Sun E5000 is an 8 processor bus-based machine with superscalar processors, 16KB data caches, 16KB instruction caches, and a level two cache.

4.2 Languages

To show that our approach is not specific to any particular language, we use benchmark programs written in two languages: C with calls to vendor-supplied implementations of the MPI message passing standard [10], and ZPL, a dataparallel array language [7, 32].

4.3 Regression Models

Linear regression can be used to develop performance models by setting up a system of equations where the known values are measured communication times and calculated values of the model features, and the unknowns are the model coefficients. Solving this system produces the values of the model coefficients that we are looking for.

The data used to determine unknown coefficients in regression analysis will be referred to as the *training set*, and the data used for testing the performance of models is known as the *test set*. Linear regression models for the communication cost can be defined as

$$y_i = \beta_0 + \beta_1 \Phi_{i1} + \beta_2 \Phi_{i2} + ... + \beta_p \Phi_{ip}, \qquad i = 1, 2, ..., n, \tag{7}$$

where n is the number of elements in the training set, p is the number of coefficients less one (the degrees of freedom) in the model, and y_i is the measured communication cost. This equation can also be stated in matrix form as

$$\mathbf{y} = \mathbf{\Phi} \beta \tag{8}$$

The elements of the $\mathbf{\Phi}$ matrix are known. Each column of this matrix (sometimes called basis functions) represents one feature of the model. For example, for our model (equation 1) the first column represents the number of messages sent, the second column the number of bytes transferred, and the third column the number of memory lines touched. The values of $\mathbf{y}$ are the measured communication times from our training set. To find the value of the β vector, the coefficients of our model, we use a least squares method, which is defined as

$$\beta = \mathbf{\Phi}^{+}\mathbf{y} \tag{9}$$

where $\mathbf{\Phi}^{+}$ is the pseudo-inverse of $\mathbf{\Phi}$ [4].

Our three model features have different scales, so we normalize the columns of the Φ matrix to improve the numerical properties of the system.

The models we have discussed thus far are called first-order regression models, because the exponent of each Φ_j is one. Alternatively we can define second-order models which include quadratic, Φ_j^2, and cross-product, $\Phi_j\Phi_k$, terms. Higher order models may sometimes provide better fit, but they might not generalize well. This happens because any non-zero error in model predictions arises for two distinct reasons. If the model is different from the "true model" that we're looking for, we have *bias*. If the model parameters are too sensitive to a particular training set and do not give good predictions for data outside the training set, we have *variance.* The overall goal in regression analysis is to find a good compromise between bias and variance [4]. Generally one decreases as the other increases. One way to reduce both bias and variance at the same time is to use larger training sets. We explore this effect in Section 5.2.

Model Name	**Regression Function**
S1 (Standard Model)	$\alpha + \beta\mathbf{n}$
S2	$\alpha + \beta n + \beta_1 n^2$
S3	$\alpha + \beta n + \beta_1 n^2 + \beta_2 n^3$
M1 (Our Model)	$\alpha + \beta\mathbf{n} + \gamma\mathbf{MLT}$
M2	$\alpha + \beta n + \gamma MLT + \gamma_1 nMLT$
M3	$\alpha + \beta n + \gamma MLT + \gamma_1 nMLT + \beta_1 n^2 + \gamma_2 MLT^2$

Table 1: Regression models for per-message point-to-point communication cost.

4.4 Generating Training and Test Data

The best way to evaluate the performance of a model is to use test sets that are independent from the training set. Thus, we collect a number of measurements and use 100 of these measurements as the training set and the rest of the measurements as the test set. Section 5.2 shows that such training sets are sufficiently large. Our measurements represent data along two dimensions: the size of the matrix and the number of rows or columns to transfer. To generate measurements, we take matrix sizes between 1×1 and 4000×4000, and we assume that the number of rows or columns to be transferred is between 1 and 200.

4.5 Statistical Analysis

Many different statistical measures can be used to assess the adequacy of a fitted model, and it is generally recommended that more than one statistic be used [25, 29]. We use two measures. The first is *estimated error standard deviation*, $s_e = MS_E$, where

$$MS_E = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n - p - 1} \tag{10}$$

y_i: i^{th} measured value in the test set
$\hat{y}_i$: model prediction for y_i
n: number of elements in the test set (training and test sets have the same size)

A small value of MS_E indicates a good fit between predicted and measured results. The second statistic is the *coefficient of determination*, R^2, which is probably the most extensively used measure of goodness for regression models. There are various definitions of R^2, each with its potential pitfalls [25]. We use the following definition, as suggested by Mason et al. [29]:

$$R^2 = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)}{\sum_{i=1}^{n} (y_i - \bar{y}_i)} \tag{11}$$

In assessing model accuracy, R^2 equals unity when the model is as good a predictor of the target data as the simple model $\hat{y} = \bar{y}$, and it equals to zero if the model predicts the data values exactly [4]. For regression problems we need an R^2 value that is less than 0.01.

5 Modeling Point-to-Point Communication

Using the three model features described in the previous section, we define six different linear regression models (See Table 1). S1 corresponds to models discussed in the literature, which only use the number of messages and message length as model features [3, 24, 12]. We refer this as the *standard model*. M1 is our proposed model, the simplest model that includes MLT as a feature. The other models are higher order models that we test for completeness.

5.1 Experimental Results

Figures 5 and 6 illustrate the difference between model predictions for S1, M1, and M3 on the SP-2 using the ZPL and C+MPI benchmarks. In these figures bold lines (dots) represent measured values and thin lines represent predicted values. We see that S1 predicts badly, particularly for column transfers, while M1 is quite accurate, and M3 is slightly better yet. (The other models that do not include MLT (S2, and S3) produce results that are comparable to S1, and those that include MLT (M2) produce results that are comparable to M1. All six models are statistically analyzed in the next subsection).

Note that these graphs show results for transferring one column or one row of a matrix. Recall from Figure 2 that cache effects are most prominent when $d = 1$, so we are showing the worst case for the standard model. However, an informal survey of ZPL programs shows that $d = 1$ represents the most common type of communication. Results for the other machines are similar, so we do not include them here.

Finally, Figure 7 shows how model M1 predicts performance on the SP-2 and T3E for the motivating graphs that we showed in Section 3. While our model cannot predict all of the non-linearities of the observed behavior, the model does capture overall performance quite well. For comparison purposes, Figure 8 shows how the standard model, S1, predicts the same performance, and Figure 9 shows how our model can be improved using quadratic terms (M3).

5.2 Statistical Assessment

Table 2 shows the accuracy of the models for six sets of measured data, namely, using C+MPI and ZPL on three different architectures. Each entry in the table statistically summarizes an entire set of measured data points. We see that all of the models that include MLT are superior to the standard model. For example, the R^2 or MS_E values of S1 model are 2-6 times larger than corresponding M1 values. As mentioned above, a value smaller than 0.01 is

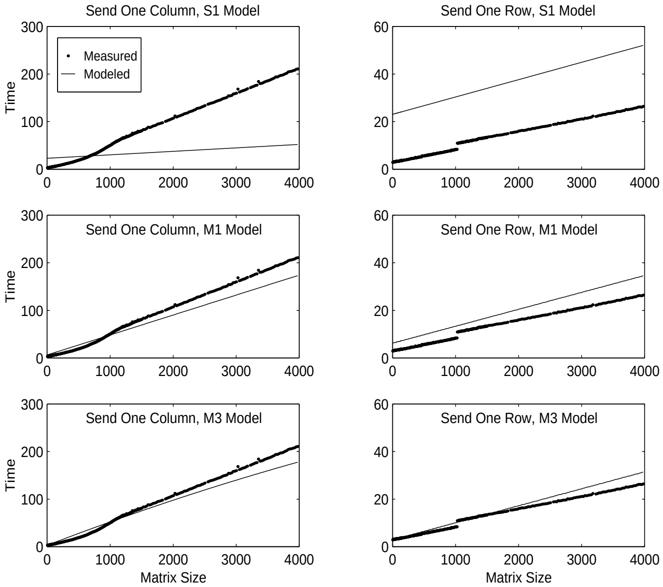


Figure 5: Measured vs. predicted results for ZPL on the IBM SP-2. Our models, M1 and M3, give much better predictions than the standard model, S1.

assumed acceptable for R^2 in regression operations, and M1 satisfies this condition in all cases. M3, a second order model which includes MLT, gives the best results in all cases. We believe that M3 is superior because it implicitly models cache conflict misses by its inclusion of the MLT^2 term, which we have computed to be correlated to MLT^2. We do not model these conflict misses with a separate parameter because this would complicate the model.

Model Name	**IBM SP-2**				**CRAY T3E**				**SUN E5000**			
	ZPL		**C+MPI**		**ZPL**		**C+MPI**		**ZPL**		**C+MPI**	
	R^2 $(\times10^{-3})$	MS_E $(\times10^3)$	R^2 $(\times10^{-3})$	MS_E $(\times10^3)$	R^2 $(\times10^{-3})$	MS_E $(\times10^3)$	R^2 $(\times10^{-3})$	MS_E $(\times10^3)$	R^2 $(\times10^{-3})$	MS_E $(\times10^3)$	R^2 $(\times10^{-3})$	MS_E $(\times10^3)$
S1	**3.54**	**1.84**	**2.81**	**0.48**	**9.05**	**2.23**	**10.12**	**0.59**	**25.87**	**3.21**	**12.02**	**0.52**
S2	3.45	1.79	2.75	0.47	8.89	2.19	9.71	0.57	25.74	3.19	7.45	0.32
S3	3.39	1.76	2.72	0.46	8.85	2.18	9.56	0.56	25.54	3.17	6.02	0.26
M1	**0.60**	**0.31**	**0.29**	**0.05**	**3.29**	**0.81**	**2.34**	**0.14**	**13.41**	**1.66**	**9.60**	**0.42**
M2	0.59	0.30	0.29	0.05	2.87	0.71	2.21	0.13	12.83	1.59	4.24	0.18
M3	0.44	0.23	0.26	0.04	2.87	0.71	2.01	0.12	8.12	1.01	3.76	0.16

Table 2: Accuracies of the six models. Small values represent better accuracy. Our model, M1, and its derivatives, M2 and M3, provide better accuracy than the standard model, S1, and its derivatives, S2 and S3.

Figure 10 shows how the training set size affects accuracy for the IBM SP-2 with C+MPI. We use two graphs because of the scale differences. Results for the Cray T3E and SUN E5000 are very similar, as are those for the ZPL benchmarks. As mentioned before, as the model complexity (number of coefficients to be determined) increases, more data points are needed to fit the model. *We see that a training set size of 100 is always sufficient, and that beyond 100, better predictions can only be obtained by developing better models, not by taking more measurements.*

6 Modeling Parallel Prefix Operations

To be generally useful, our approach should apply to communication operations other than to point-to-point messages. This section extends our model to parallel prefix operations [26, 27], which are both challenging and significant. They are challenging because they combine both communication and computation costs, and they are significant because of

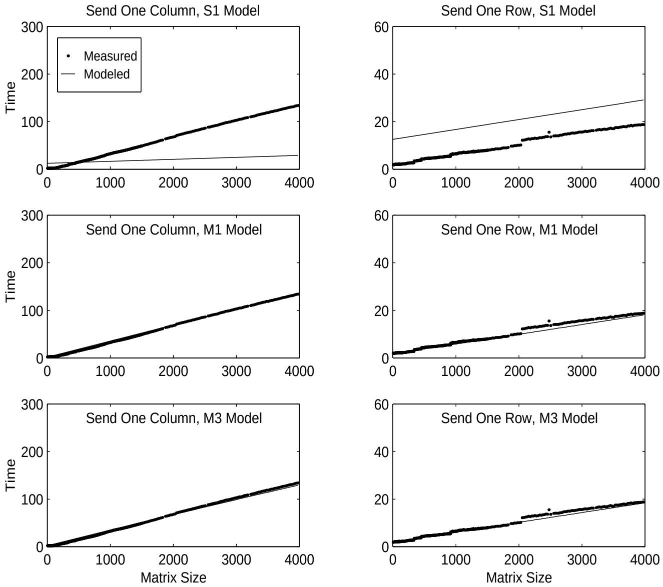


Figure 6: Measured vs. predicted results for C+MPI on the IBM SP-2. Our models, M1 and M3, give much better predictions than the standard model, S1.

their usefulness in parallel algorithms. For example, parallel prefix operations can be used to solve problems such as ranking, sorting (using radix sort), carry lookahead addition, the solution of linear recurrences, polynomial evaluation and interpolation, and pattern matching. For more information, Lakshmivarahan and Dhall provide a comprehensive analysis of the prefix problem [27].

The parallel prefix problem is defined as follows:

> Let A be a set and $\circ$ be a binary operation defined over the elements of A. Consider $d = (d_1, d_2, d_3, ..., d_N)$ where $d_i \in A$, for $1 \leq i \leq N$. Now, the problem of computing $x_i = x_{i-1} \circ d_i$ for $2 \leq i \leq N$, where $x_1 = d_1$, is called the prefix problem. Prefix operation is sometimes called as the scan operation.

In this study we consider horizontal and vertical scan operations in two dimensional arrays. In a scan operation, a processor that operates on an $L \times L$ array of data, does L^2 arithmetic operations. It also performs point-to-point communication for at most $2L$ elements. In other words, it sends and/or receives one row or one column of the array. We expect horizontal scans to be more expensive than vertical scans, because a horizontal scan transfers one non-contiguous column while a vertical scan transfers one contiguous row. Indeed, our measurements indicate that vertical scans are up to 40% more expensive than their horizontal counterparts.

In natural extensions of the point-to-point communication models, we can model scans by adding a new term, *the number of arithmetic operations*, to our point-to-point communication models. We performed experiments to compare the scan models on different processor configurations for various matrix sizes. We used seven different processor configurations, with matrix sizes ranging from 50×50 to 2000×2000. The standard model always underpredicts the cost of the vertical scans and overpredicts the cost of the horizontal scans. While our model is not perfect, it better matches the observed behavior. For example, on the Cray the R^2 value for the standard model is an unacceptable 48.12×10^3, whereas with our model it is reduced to 0.36×10^3. Adding cross product and second order terms to the standard model improves its prediction accuracy only insignificantly. While the standard model does not benefit from higher order terms of the model features, our model does, so we use cross product and second order terms. Figure 11 shows these results graphically for matrix sizes ranging from 500×500 to 1000×1000.

7 Modeling Whole Programs: Uniform Linear Convolution

To test the practical benefits of our approach, we see if it can accurately model the performance of two different programs for solving a single problem. The problem is the uniform linear convolution, and the two programs use different

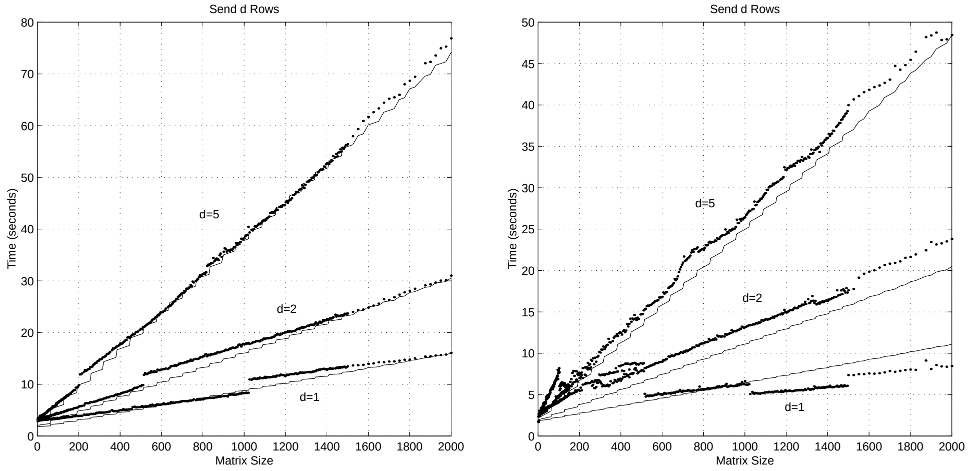


Figure 7: Predicted cost using our model (M1) on the IBM SP-2 (left) and the Cray T3E (right) for sending d rows of a 2000 × L Matrix, as L varies from 0 to 2000.

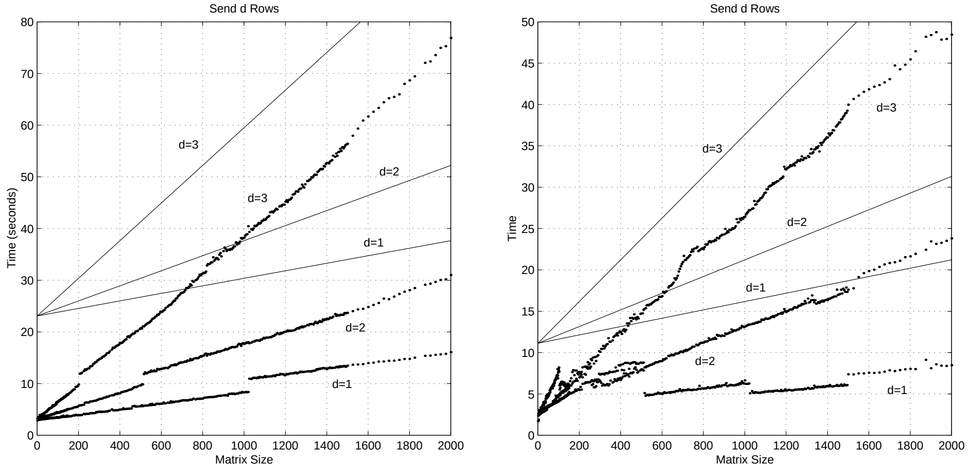


Figure 8: Predicted cost using the standard model (S1) on the IBM SP-2 (left) and the Cray T3E (right) for sending d rows of a 2000 × L Matrix, as L varies from 0 to 2000.

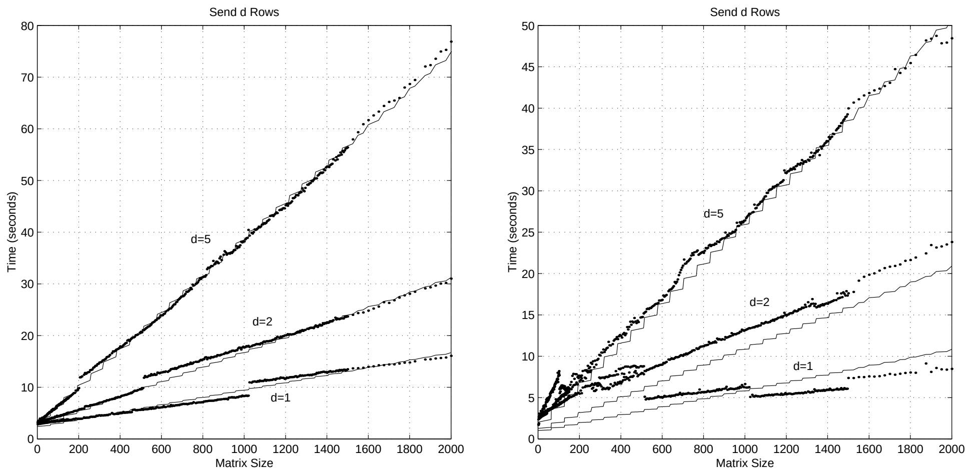


Figure 9: Predicted cost using our model with quadratic terms (M3) on the IBM SP-2 (left) and the Cray T3E (right) for sending d rows of a 2000 × L Matrix, as L varies from 1 to 2000.

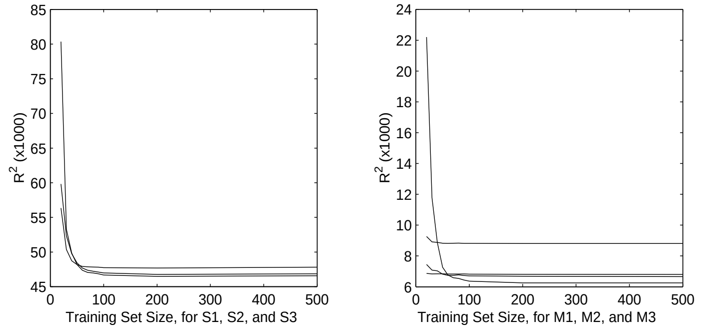


Figure 10: The effect of training set size on accuracy, for C+MPI on the IBM SP-2. A training set size of 100 is always sufficient.

algorithms that exhibit complex tradeoffs that depend upon both machine-specific and problem-specific parameters.

Convolution is one of the most frequently used signal and image processing operations, with applications to circuit theory, optics, and digital filtering [6]. The uniform linear convolution accepts as inputs an $L \times L$ image and a block size, b. The output is an $L \times L$ image where each pixel is the weighted sum of the $b \times b$ block of image values for which that pixel is the lower right corner.

Figure 19 shows the Conv-Scan program, written in ZPL, that computes the uniform convolution using two parallel prefix operations and three point-to-point shift operations. Line 27 computes the parallel prefix of the image, stored in the array `Im`, along the second, or horizontal, dimension, and Line 28 computes the parallel prefix of the resulting array along the first, or vertical, dimension. As a result, each element of the `BoxSum` array contains the sum of all elements to its left and top (See Figure 12). Given these prefix sums, Line 30 then computes the uniform convolution by adding and subtracting values of `BoxSum` as shown in Figure 13. Complete descriptions of the ZPL language are available elsewhere [7, 32].

Figure 18 shows the Conv-Shift program, again written in ZPL, that computes the same result by repeatedly shifting the image to the right to accumulate the sums along each row, and then repeatedly shifting the values down each column to compute the sum of the entire $b \times b$ block. Without describing the full details of ZPL's syntax and semantics, it suffices to understand that the `@` operator in line 32 is used to shift the image data, held in the array `T`, b times to the right, with each shift incurring point-to-point communication between adjacent processors. Similarly, the loop on line 38 shifts the value of `T` b times in the vertical direction to complete the summation of the values inside of a box. Given ZPL's array semantics, these statements compute the uniform convolution for each point of the original

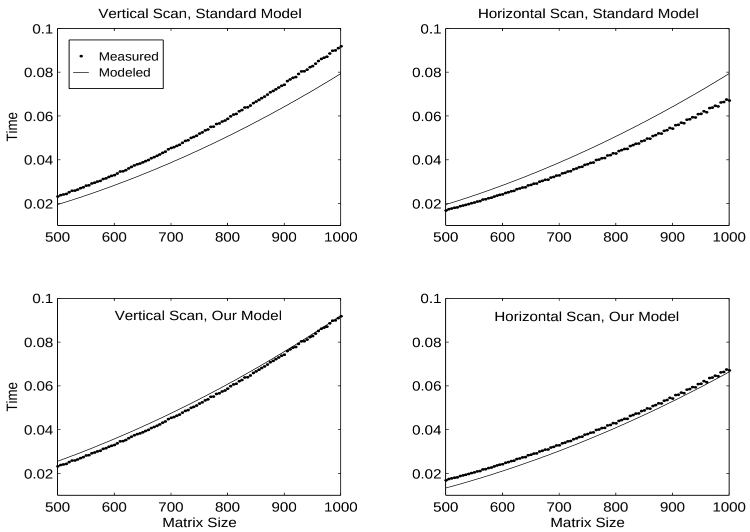


Figure 11: Comparison of the model precisions for scan operations on the Cray T3E with ZPL using a 2×2 processor mesh. The top two figures show that the standard model underpredicts the cost of the vertical scans and overpredicts the cost of the horizontal scans. Our model performs much better.

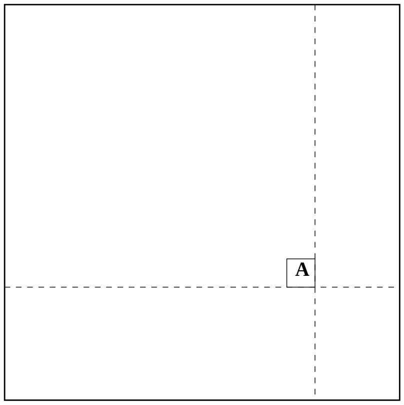


Figure 12: After executing lines 27 and 28 in Figure 19, each element of the `BoxSum` array contains the sum, A, of all elements to its left and top.

image.

In general, it's unclear which of the two ZPL programs will run faster because there are many factors to consider:

- The Conv-Shift program uses $2 \cdot (b - 1)$ shift operations and no scan operations.
- The Conv-Scan program uses a fixed number of communication operations: two scans and three shifts.
- Shifts require only point-to-point communication, while scans require communication with all other processes in the row or column of the processor mesh.
- As the box size increases, the number of shifts required in Conv-Shift also increases.
- As the number of processors increases, the cost of scan operations will also grow, albeit slowly ($O(\log \sqrt{n})$).
- The relative performance is machine-dependent, since the relative costs of shift and scans will differ depending on the number of processors and their communication and computation power.

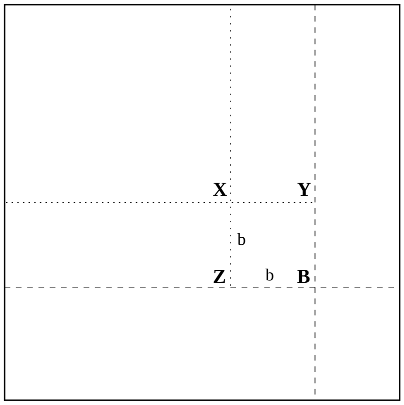


Figure 13: The uniform convolution of the $b \times b$ box can be computed from the values of the BoxSum array by taking the value B, subtracting Z and Y, and then adding the value at X to account for the fact that X has been subtracted twice.

We hypothesize that the Conv-Scan algorithm is generally faster, but to evaluate our performance model we'd like to see if it accurately predicts the tradeoffs between the two programs. We thus modeled the costs of these programs using the standard model and our model.

To model these programs, we first need a method of modeling computation costs such as the addition of two arrays and the copying of one array to another.

7.1 Modeling Computation Cost

To model the computation costs of ZPL statements that do not induce communication, we model the execution time of two types of statements: $A = x$ and $A = x \; op \; y$, where x represents either a scalar or an array, and where *op* represents either addition or multiplication. Other operations, such as division, can be modeled similarly. We decompose the computation cost, t_{comp}, into two parts: the cost of arithmetic operations, t_{arith}, and the cost of the memory overhead, t_{memory}.

$$t_{comp} = t_{arith} + t_{memory} \tag{12}$$

Here, t_{memory} represents the cost of all load and store operations including cache miss penalties. Since t_{memory} includes all memory overhead, we can model the cost of arithmetic operations as: $t_{arith} = \delta p$, where p is the number of arithmetic operations and δ is the model unknown.

We model t_{memory} using the same models that we used for modeling communication cost (See Table 1). In these models MLT has the same meaning, but n means the total number of load and store operations rather than the number of bytes transferred. Our results for computation cost are very similar to the results for communication cost. For example on the Cray T3E, R^2 metric for $A = A + B$ is 17.4 with the standard model whereas we reduce it to 5.3 with our model.

7.2 Experimental Results

To compare our model against the standard model, we performed experiments for block sizes ranging from 1 to 10 and image sizes ranging from 100 × 100 to 1000 × 1000. Figure 14 shows our results on a 2 × 2 configuration of the Cray T3E. The y-axis is the execution time of the Conv-Shift algorithm divided by that of the Conv-Scan algorithm, so a value smaller than 1.0 indicates that Conv-Shift runs faster. Each point along the x-axis represents a different combination of block size and matrix size, with only the block sizes labeled in the figure. The top graph shows how the standard model (thin line) compares against actual running times (heavy dots), and the bottom graph shows the same information for our model. We see that the standard model consistently underpredicts this ratio whereas our model gives very accurate results.

As expected, the block size affects the relative performance of the two algorithms. On the Cray, the Conv-Shift algorithm is superior for block sizes smaller than 3. Our model can make this distinction and predicts the better

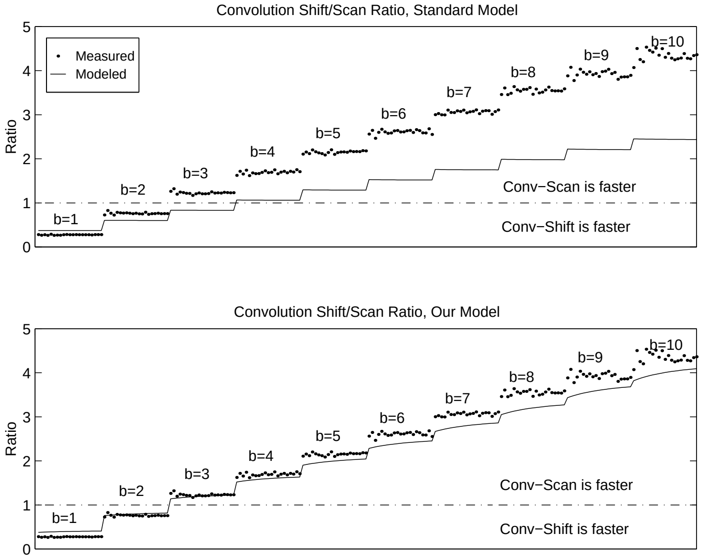


Figure 14: Modeling the relative performance of two uniform convolution programs on the Cray T3E with a 2×2 processor mesh. Each point along the x-axis represents a different block size (b) and different image size. The y-axis gives the ratio of the execution times for the two programs. Ratios above 1.0 indicate that Conv-Scan is faster, while ratios below 1.0 indicate that Conv-Shift is faster. Note that the standard model incorrectly predicts that Conv-Shift is faster when $b = 3$.

algorithm in all cases. By contrast, the standard model would choose the incorrect algorithm when the block size is 3, predicting a ratio that is off by about 25%.

Figure 15, shows the details for $b = 3$. The two top graphs show that the standard model can mispredict execution time of the Conv-Shift algorithm by a factor of two and the Conv-Scan algorithm by a factor of 4. The inaccuracy of the standard model is not as evident in Figure 14 because the ratio of the execution times factors out some of the overprediction. The bottom two graphs of Figure 15 show that our model is quite accurate.

While the crossover occurs at $b = 3$ on the Cray, the crossover differs depending on the architecture. Figure 17 shows that on the SUN E5000 the threshold value is 7 and that the standard model chooses the inferior algorithm for $b = 5$ and $b = 6$. On the IBM SP2, the crossover occurs at $b = 6$, and the standard model mispredicts for $b = 4$ and for some instances where $b = 5$.

Note that our prediction results cannot be duplicated by the approaches mentioned in the Related Work section. The performance models that do not consider cache effects would produce results no better than the standard model. Approaches that only model specific aspects of cache behavior, such as the number of cache misses, do not provide estimates of overall execution time, so even if these approaches could accurately predict the number of cache misses, it's unlikely that the relative performance of the two uniform convolution programs could be predicted solely by comparing cache misses. Moreover, the Cache Miss Equations, which precisely compute the number of cache misses, would be prohibitively expensive to compute since a separate set of equations would have to be solved for each data point in the graph. Finally, the performance bounds produced by Amato's approach are so large (two orders of magnitude) that their model could not detect the crossover point between the two uniform convolution programs.

8 Conclusions

This paper has described how cache effects can be incorporated into a communication cost model by including a term that is a function of MLT, the number of unique memory lines touched. This term is significant because it is sensitive to data layout, effectively modeling the number of compulsory and capacity cache misses that are incurred in marshalling and unmarshalling messages. The benchmarking approach that we use is also crucial, because as it customizes the model to individual machines, it implicitly captures details such as cache access times.

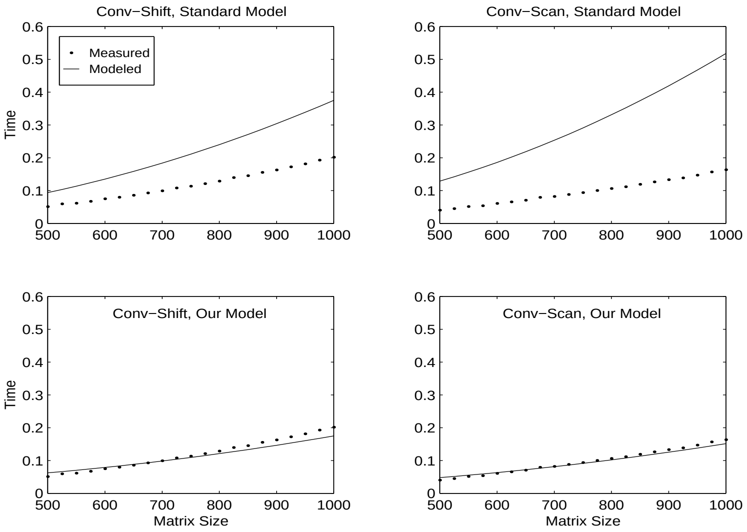


Figure 15: Standard model vs our model for the two uniform convolution algorithms for the block size of 3. Measurements are done on the Cray T3E with a 2×2 processor mesh. The standard model always overpredicts the running times of both of the algorithms, whereas our model gives accurate results.

We have derived formulae for computing MLT for rectangular subarrays and have shown that our model is significantly more accurate than the standard model for both point-to-point communication and parallel prefix operations. The former are important because they are the basis upon which all communication models can be built. The latter are interesting because they combine both communication and computation in the same operation. Finally, we have conducted an experiment to predict the performance of two ZPL programs that compute the uniform convolution using different algorithms. Our model correctly predicts the crossover point between these two programs, whereas the standard model fails in this regard.

We are currently studying ways to model cache conflict misses, in an attempt to understand the importance of this as a model feature. Ultimately, our goal is to use our model to guide compiler optimizations, for example, to trade off load balance for extra communication and to choose optimal block sizes for block-cyclic data distributions.

Acknowledgments. We thank E Christopher Lewis for his useful comments on earlier versions of this paper.

References

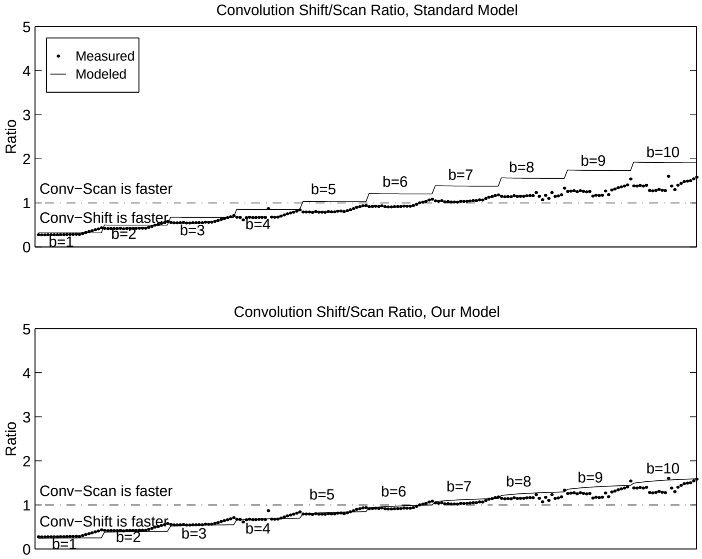
[1] Anant Agarwal, Mark Horowitz, and John Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, May 1989.

[2] Nancy M. Amato, Andrea Pietracaprina, Geppino Pucci, Lucia K. Dale, and Jack Perdue. A cost model for communication on a symmetric multiprocessor. Technical Report 98-004, Department of Computer Science, Texas A&M University, January 1998.

[3] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A static performance estimator to guide data partitioning decisions. Technical Report COMP TR90-136, Department of Computer Science, Rice University, Houston, TX, October 1990.

[4] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[5] Robert J. Block, Sekhar Sarukkai, and Pankaj Mehra. Automated performance prediction of message-passing parallel programs. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference, San Diego*, December 1995.


Convolution Shift/Scan Ratio, Standard Model

Convolution Shift/Scan Ratio, Our Model

Figure 16: Modeling the relative performance of two uniform convolution programs on the Sun E5000 with a 2×2 processor mesh. Each point along the x-axis represents a different block size (b) and different image size. The y-axis gives the ratio of the execution times for the two programs. Ratios above 1.0 indicate that Conv-Scan is faster, while ratios below 1.0 indicate that Conv-Shift is faster. Note that the standard model incorrectly predicts that Conv-Shift is faster when $b = 5$ and $b = 6$.

[6] C.S. Burrus and T.W. Parks. *DFT/FFT and Convolution Algorithms, Theory and Implementation*. John Wiley and Sons, 1985.

[7] Bradford Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. The case for high level parallel programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–86, July-September 1998.

[8] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 279–290, June 1995.

[9] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.

[10] Jack J. Dongarra, Steve W. Otto, Marc Snir, and David Walker. An introduction to the MPI standard. *Communications of the ACM*, to appear.

[11] Thomas Fahringer. Estimating and optimizing performance for parallel programs. *IEEE Computer*, pages 47–56, November 1995.

[12] Thomas Fahringer. *Automatic Performance Prediction of Parallel Programs*. Kluwer Academic, 1996.

[13] Thomas Fahringer. Compile-time estimation of communication costs for data parallel programs. *Journal of Parallel and Distributed Computing*, 39(1):46–65, November 1996.

[14] Thomas Fahringer. Estimating cache performance for sequential and data parallel programs. In *Proceedings of HPCN'97*, April 1997.

[15] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 328–343, August 1991.

[16] Somnath Ghosh, Margaret Martonosi, and Shared Malik. Cache miss equations: An analytical representation of cache misses. In *ICS'97, Vienna, Austria*, pages 317–324, 1997.

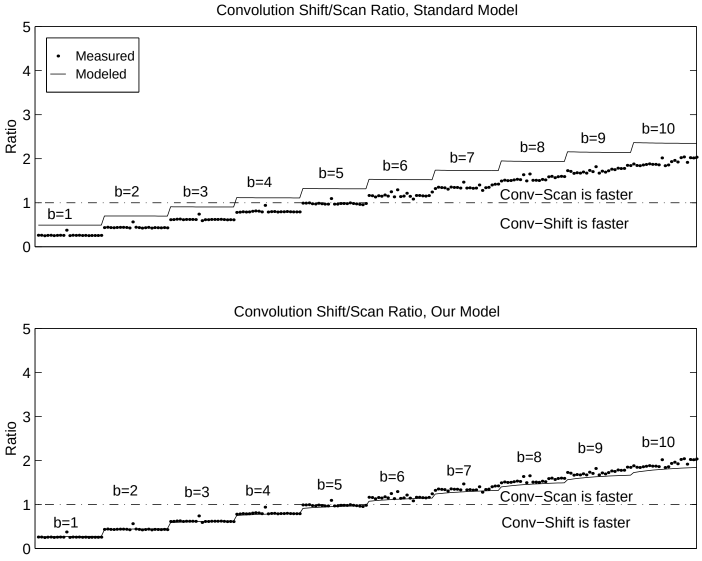


Figure 17: Modeling the relative performance of two uniform convolution programs on the IBM SP2 with a 2×2 processor mesh. Each point along the x-axis represents a different block size (b) and different image size. The y-axis gives the ratio of the execution times for the two programs. Ratios above 1.0 indicate that Conv-Scan is faster, while ratios below 1.0 indicate that Conv-Shift is faster. Note that the standard model incorrectly predicts that Conv-Shift is faster when $b = 4$ and $b = 5$.

[17] Somnath Ghosh, Margaret Martonosi, and Shared Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the Workshop on Interaction between Compilers and Computer Architectures (HPCA-3), San Antonio, Texas*, February 1997.

[18] Somnath Ghosh, Margaret Martonosi, and Shared Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the 8th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII), San Jose, California*, October 1998.

[19] John Gunnels, Calvin Lin, Greg Morrow, and Robert van de Geijn. A flexible class of parallel matrix multiplication algorithms. In *Proceedings of the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, March 1998.

[20] Ibrahim Hur and Calvin Lin. Modeling the cache effects of interprocessor communication. In *11th IASTED International Conference on Parallel and Distributed Computing and Systems*, October 1999.

[21] Melody Y. Ivory. Modeling communication layers in portable parallel applications for distributed-memory multiprocessors. Master's thesis, University of California at Berkeley, 1996.

[22] H. Jonkers. *Performance Analysis of Parallel Systems: A Hybrid Approach*. PhD thesis, Delft Technical University, Netherlands, October 1995.

[23] H. Jonkers, A.J.C. van Gemund, and G.L. Reijns. A probabilistic approach to parallel system performance modeling. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, volume 2, January 1995.

[24] Ken Kennedy, Nathaniel McIntosh, and Kathryn McKinley. Static performance estimation in a parallelizing computer. Technical Report CRPC TR92294-S, Center for Research on Parallel Computation, Rice University, Houston, TX, April 1992.

[25] Tarald O. Kvalseth. Cautionary note about r2. *The American Statistician*, 39(4):279–285, November 1985.

```
program conv_scan;

config var N : integer = 512;
           w : integer = 2;
constant   M : integer = -10;
region     I = [1..N, 1..N];

var       Im     : [I] integer;
          BoxSum : [I] integer;
          t      : double;
          fp     : file;

direction west  = [0, M];
          north = [M, 0];
          nw    = [M, M];

procedure conv_scan();
[I]          begin
                   fp:=open("scan.dat","a+");

                   Im:=1;
[west of I]        BoxSum:=0;
[nw of I]          BoxSum:=0;
[north of I]       BoxSum:=0;

                   BoxSum:=+||[2] Im;       -- compute parallel prefix
                   BoxSum:=+||[1] BoxSum;   -- compute parallel prefix

                   BoxSum:= BoxSum - BoxSum@north - BoxSum@west + BoxSum@nw;

                   BoxSum:= BoxSum*w;
             end;
```

Figure 18: ZPL program that computes an $M1 \times M2$ uniform convolution using parallel prefix operations.

[26] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the Association for Computing Machinery*, 27(4):831–838, October 1980.

[27] S. Lakshmivarahan and Sudarshan K. Dhall. *Parallel Computing Using the Prefix Algorithm*. Oxford University Press, 1994.

[28] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Fourth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[29] Robert L. Mason, Richard F. Gunst, and James L. Hess. *Statistical Design and Analysis of Experiments*. John Wiley & Sons, 1989.

[30] Jens Simon and Jens-Michael Wierum. Accurate performance prediction for massively parallel systems and its applications. In *Proceedings of the European Conference on Parallel Processing (EuroPar'96)*, pages 675–688, August 1996.

[31] Jaswinder Pal Singh, Harold S. Stone, and Dominique F. Thiebaut. A model of workloads and its use in miss-rate prediction for fully associative caches. *IEEE Transactions on Computers*, 41(7):811–825, July 1992.

[32] Lawrence Snyder. *A Programmer's Guide to ZPL*. MIT Press, 1999.

[33] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling Computer Systems*, 1994.

[34] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

```
program conv_shift;

config var  N : integer = 15;
            boxb : integer = 4;
            boxh : integer = 4;
            w     : integer = 2;
region      I = [1..N, 1..N];

var         Im        : [I] integer;
            BoxSum, T : [I] integer;
            t         : double;
            fp        : file;

direction   west  = [0, -1];
            north = [-1, 0];

procedure conv_shift();
var i:integer;
[I]       begin
               fp:=open("shift.dat","a+");

               Im:=1;
               T:=Im;

[west of I]    T:=0;
[north of I]   T:=0;

               BoxSum:=Im;
               T:=Im;

               for i:=1 to boxb-1 do    -- Shift and accumulate values of T
                   T:=T@west;
                   BoxSum:=BoxSum+T;
               end;

               T:=BoxSum;

               for i:=1 to boxh-1 do    -- Shift and accumulate values of T
                   T:=T@north;
                   BoxSum:=BoxSum+T;
               end;

               BoxSum:=BoxSum*w;
           end;
```

Figure 19: ZPL program that computes an $M1 \times M2$ uniform convolution using shift operations.