Working paper, December, 1996.

Communication Performance of Irregular Applications and the Implications for Graph Partitioners

David R. O'Hallaron and Jonathan Richard Shewchuk

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{droh, jrs}@cs.cmu.edu

Abstract

We derive a simple performance model for a family of sparse irregular applications and use the model to predict running times on the Cray T3D. Based on insights gained from applying the model to the applications, we identify some limitations of modern graph partitioners and suggest some improvements. In particular, we suggest an approach whereby a partitioner could use our model to attempt to minimize running time on a specific system.

1 Introduction

In general it is important to understand the communication requirements of real applications [6], and these requirements are especially difficult to understand for the important class of irregular scientific applications that manipulate sparse matrices. Irregular applications include those that model processes in nature like the flow of air over a wing, or the distribution of stresses and strains in a bridge. Characterizing realistic irregular applications is difficult, primarily because they simulate complex physical phenomena operating over complex physical domains, and thus tend to be large, complex, and poorly understood.

This paper characterizes the communication performance of a family of large irregular applications and uses this characterization to identify some limitations in modern graph partitioning approaches. Specifically, we are concerned with a family of *Quake applications* that model the motion of the ground during strong earthquakes in the San Fernando Valley of Southern California. The Quake applications, described in Section 2, were developed over a period of two years by civil engineers at Carnegie Mellon as part of an ongoing project to model the entire Los Angeles basin [1].

A nice property of the Quake applications is that their running time is dominated by a sparse matrix-vector product (SMVP) operation that is repeated thousands of times. Further, the SMVP is the only operation in the application that requires the transfer of data between processors. Thus, we can focus on understanding the behavior of the SMVP operation and abstract away much of the complexity of the application.

Starting from a simple performance model of the SMVP operation, we describe an approach for deriving the application-specific and system-specific parameters in the model, and then use them to predict the running time on the Cray T3D of one of the Quake applications. Our performance model for the SMVP is described in Section 3, and the model is validated in Section 4.

This research was sponsored in part by the Advanced Research Projects Agency/CSTO monitored by SPAWAR under contract N00039-93-C-0152, in part by the National Science Foundation under Grant CMS-9318163, and in part by a grant from the Intel Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either express or implied, of the U.S. government.

Then in Section 5 we use the model to identify a limitation in current graph partitioners and to suggest a different approach. In general, modern graph partitioners attempt to minimize the global volume of data transferred, subject to the constraint that each PE has an equal amount of work. This is a convenient metric for algorithm designers, but it is not clear how closely related it is to actual running time of parallel programs, because of two shortcomings: (1) It ignores the number of neighbors that each PE must communicate with, and (2) it does not attempt to balance communication across PEs. Shortcoming (1) is significant because, according to our model, real applications can be either dominated by block latency, dominated by burst bandwidth during block transfers, or somewhat balanced between the two. The choice depends on the properties of the underlying system. In cases where the communication phase is latency-dominated, the partitioner should attempt to minimize the maximum number of neighbors of each subgraph (and thus the number of distinct messages that each PE must send). Shortcoming (2) is significant because the communication time can vary significantly from PE to PE when using modern graph partitioners, and the running time of the slowest PE determines the overall running time.

The general problem is that modern graph partitioners do not attempt to minimize actual running time. In Section 5 we suggest a system-specific approach based on our simple performance model. With this approach, the partitioner uses the model in Section 3 to attempt to minimize overall running time, using the method outlined in Section 4 to determine the system-specific bandwidth and latency parameters. We are not outlining a new partitioning algorithm, but we are suggesting a system-dependent objective function that future partitioners should seek to optimize.

2 The Quake applications

The paper is motivated by our experience with a family of four *Quake applications*, denoted sf10, sf5, sf2, and sf1. Each is a finite-element simulation of the motion of the ground in the San Fernando Valley during a strong earthquake [1]. The digit in each name indicates the highest frequency wave (in seconds) that the simulation is able to resolve. For example, sf10 resolves waves with 10 second periods, sf5 resolves waves with 5 second periods, and so on. Each application simulates 60 seconds of shaking as shock waves travel through a model of the San Fernando Valley. Each model employs a three-dimensional unstructured finite element mesh composed of thousands or millions of tetrahedra (i.e., pyramids with triangular bases). The model for sf10 is illustrated in Figure 1. The model corresponds to a volume of earth roughly 50 km x 50 km x 10 km, and the smallest distance between nodes is roughly 50 m. Beverly Hills is in the lower right-hand corner. The town of San Fernando is in the midst of the darkly shaded region near the upper left corner.

Each tetrahedron in Figure 1 is called an *element*, and the vertices of the tetrahedra are called *nodes*. Some finite element simulations use structured meshes constructed from regular grids; however, the Quake applications require *unstructured* meshes, which can accommodate the wildly varying density of the soils in the valley. Wavelengths are shorter in softer soils, thus softer soils need a higher density of elements.

Figure 2 lists the properties of the San Fernando meshes. When the wave period is halved, its frequency doubles, and the number of nodes increases by a factor of nearly eight—a factor of two in each of three dimensions. As a general rule, for each node in the mesh a simulation uses about 2 KBytes of memory at runtime to accommodate the storage of several vectors and sparse matrices. Thus sf10 requires about 15 MBytes and sf1 requires about 5 GBytes.

During each of the 6000 time steps, a Quake simulation executes three sparse matrix-vector product (SMVP) operations of the form $\mathbf{y} = \mathbf{K}\mathbf{x}$, where $\mathbf{x}$ and $\mathbf{y}$ are vectors of length $3n$ (representing three degrees of freedom—x, y, and z displacements—for each node of the mesh), and $\mathbf{K}$ is a sparse $3n \times 3n$ *stiffness matrix*. $\mathbf{K}$ can be likened to an adjacency matrix of the nodes of the mesh; $\mathbf{K}$ contains a 3×3 submatrix for each pair of nodes connected by an edge of the mesh (including self-edges). The stiffness matrix is extremely sparse; each node is connected to an average of 13 neighbors (in addition to itself), so each row

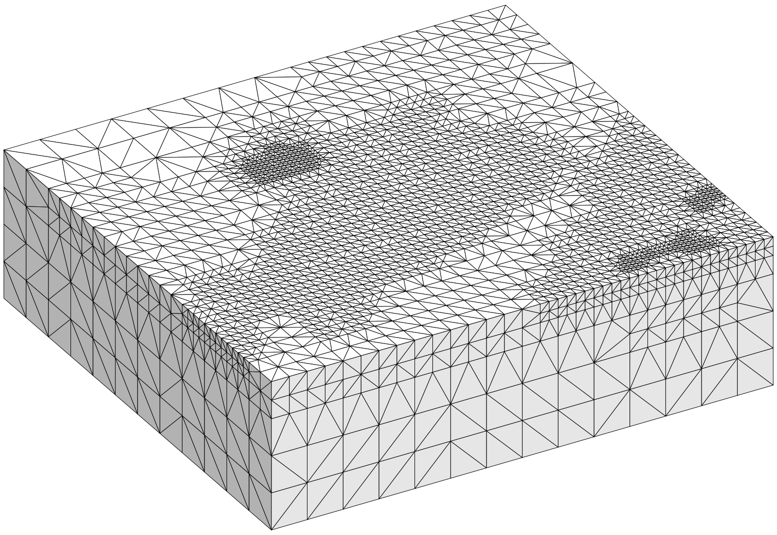

Figure 1: Finite element mesh for the sf10 model of the San Fernando Valley.

property	sf10	sf5	sf2	sf1
nodes	7,294	30,169	378,747	2,461,694
edges	44,922	190,377	2,509,064	16,684,112
faces	72,698	311,514	4,198,057	28,202,581
elements	35,025	151,239	2,067,739	13,980,162

Figure 2: Properties of the Quake applications.

of $\mathbf{K}$ contains an average of $14 \times 3 = 42$ nonzero numbers [9].

The simulations are parallelized using a domain-specific tool chain for finite element problems called Archimedes [1]. To generate a simulation that will run on p PEs, Archimedes partitions the mesh into p disjoint sets of elements. Each set is called a *subdomain* and is assigned to some PE. The partitioner is based on a recursive geometric bisection algorithm [8] that divides the elements equally among the subdomains while attempting to minimize the total number of nodes that are shared by subdomains, and hence the total communication volume. The geometric partitioning algorithm has provable upper bounds on the separator sizes and in practice usually generates partitions that are as good as those produced by other modern partitioning algorithms [2, 4, 5].

To compute $\mathbf{y} = \mathbf{K}\mathbf{x}$ on a set of PEs, we must consider the data distribution by which vectors and matrices are stored. The vectors $\mathbf{x}$ and $\mathbf{y}$ are stored in a distributed fashion according to the mapping of nodes to PEs induced by the partition of elements among PEs. If a node i resides in several PEs (because i is a vertex of several elements mapped to different PEs), the values x_i and y_i are replicated on those PEs. The matrix $\mathbf{K}$ is distributed so that K_{ij} resides on any PE on which nodes i and j both reside. Figure 3 demonstrates this method of distributing data.

Given this method of distributing data, the multiplication $\mathbf{y} = \mathbf{K}\mathbf{x}$ is performed in two steps. First, each PE computes a local matrix-vector product over the subdomain that resides on that PE. Second, PEs that share nodes communicate and combine their nodal $\mathbf{y}$ values into correct global values for each node. In

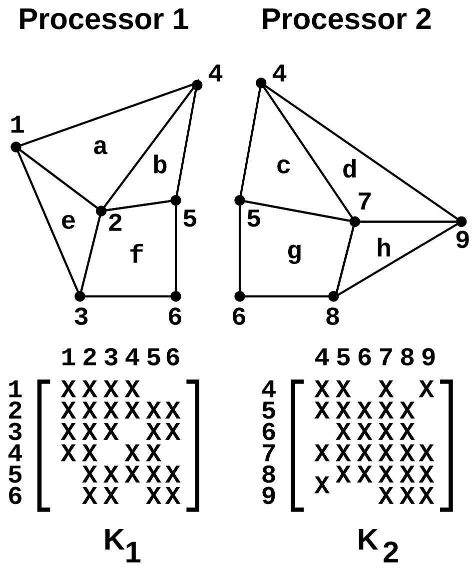

Figure 3: A finite element mesh and corresponding stiffness matrix $\mathbf{K}$, distributed among two PEs. Xs represent nonzero 3×3 submatrices. Note that some nodes are shared by both PEs, as are some stiffness matrix entries (corresponding to shared edges).

Figure 3, PEs 1 and 2 must communicate to resolve the values of the shared nodes 4, 5, and 6.

3 Performance model

As we have seen, each global SMVP consists of a simple sequence, executed simultaneously on each PE: a local SMVP (the computation phase) followed by an operation in which each PE exchanges and sums data with each of its neighbors (the communication phase). The computation and communication phases are synchronous, i.e., each phase begins with a global barrier synchronization. (It is in principle possible, with much difficulty, to improve performance by overlapping the computation and communication phases, but the real implementations of the Quake SVMP do not do this, so the models in this section will follow suit.) Further, we assume that during the communication phase, the summing operations are overlapped with the exchanges, and that the running time of the communication phase is determined by the time to perform the exchanges. This is a reasonable assumption, as the summing operations can be handled by the PE and the exchanges by a network interface. Given these assumptions, the running time for each global SMVP is

$$T_{mv} = T_{comp} + T_{comm}$$

where T_{comp} is the time required for all PEs to finish their computation phases and T_{comm} is the time required for all PEs to finish their communication phases. (For reference, Figure 4 summarizes all symbols introduced in this section.)

The unit of work for the computation phase is a floating-point operation (*flop*), which is either a scalar add or multiply. Although the flop is not an ideal way to measure work in every application, it is reasonable for our SMVP operation because the (minimum) number of flops each PE must perform can be counted exactly and is independent of the compiler and its optimization level. If a PEs local matrix has m nonzeros, then the local SMVP requires at least $F = 2m$ flops; one add and one multiply for each nonzero. If each

Symbol	Description
T_{mv}	running time for sparse matrix-vector product (SMVP)
T_{comp}	running time for computation phase of SMVP
T_{comm}	running time for communication phase of SMVP
F	flops per processing element (PE)
C	number of communication words per PE
B	number of blocks per PE
β	communication balance
T_f	time per flop (inverse of sustained FLOPS)
T_c	time per communication word (inverse of sustained bandwidth)
T_l	time per block (block latency)
T_w	time per additional block word (inverse of block bandwidth)

Figure 4: Symbols.

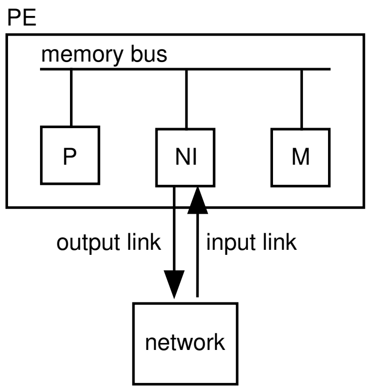

Figure 5: Processing element (PE) model (P: processor, M: memory system, NI: network interface).

flop requires an average time of T_f, then the running time for a local SMVP is

$$T_{comp} = FT_f.$$

Since T_f includes all hardware and software overheads (e.g., loads, stores, various miss penalties, pipeline stalls, etc.) it is difficult to predict. However, T_f can be precisely measured for a particular application running on a particular system.

The hardware part of the communication system on a processing element (PE) is modeled as a network interface (NI) and an input link and output link, as shown in Figure 5. (We use *processing element* instead of the common term *node* to avoid confusion with the nodes of finite element meshes.) The unit of work for the communication phase is the transfer of a *block* between the network and the local memory system of the PE. The time for this transfer is called the *transfer time*. A block is a group of words that move together from one PE to another. Blocks may be fixed-sized or variable-sized. For example, a block might be a cache line in a CC-NUMA system [7], a bulk asynchronous data transfer between two PEs' memories [10], or a page in a software DSM system [3].

The transfer time for a block i of l_i words is $T_l + l_iT_w$, where T_l is the constant *block latency*, T_w is the constant marginal cost of transferring each additional word, and T_w^{-1} is the *burst bandwidth*. Notice that the block latency does not include any delay through a network. We are only modeling the overhead of transferring data between the network and local memory, which reflects previous findings that most of the cost of communication on modern systems is incurred at the individual PEs [11].

If a PE transfers B blocks with a total volume of $C = \sum l_i$ communication words, then the running time of the communication phase on that PE is at least

$$T_{comm} = BT_l + CT_w.$$

Assume for the moment that F, C, and B are the same for each PE. Then the running time for the SMVP is

$$T_{mv} = T_{comp} + T_{comm} = FT_f + BT_l + CT_w$$

and each word is transferred in an average time of

$$T_c = \frac{T_{comm}}{C} = \frac{B}{C}T_l + T_w.$$

The rate T_c^{-1} is the *sustained bandwidth* during the communication phase. The sustained bandwidth includes all block transfers in and out of a PE, as well as all software and hardware overheads. We will use sustained bandwidth as the performance metric for the communication phase.

Modern graph partitioners do an excellent job of balancing computation (F), and an excellent job of minimizing the global volume of communication, but they do less well in balancing the amount of communication (C and B) on each PE [9]. So let us relax our assumption that C and B are the same on each PE. As C and B vary among PEs, we assume that the length of the communication phase is determined by the PE for which T_{comm} is largest. As a simplifying assumption, we assume that the PE with the largest communication volume in words (denoted C_{max}) is the PE that transfers the greatest number of blocks (denoted B_{max}); clearly, this *bottleneck PE* has the largest T_{comm} and its sustained bandwidth is given by

$$T_c = \frac{B_{max}}{C_{max}}T_l + T_w \tag{1}$$

How reasonable is our assumption that the PE that transfers the most words also transfers the most blocks? Intuitively, we would expect the two measures to be correlated, and our assumption is conservative—it never underestimates the sustained bandwidth we require—although it may overestimate it. Regardless of the tradeoff between burst bandwidth and latency, one can show (with some difficulty) that we will never overestimate the sustained bandwidth by more than

$$\beta = 1 + \min_{i \text{ is a PE}} \left[\max \left\{ \frac{(B_{max} - B_i)C_{max}}{B_{max}C_i}, \frac{(C_{max} - C_i)B_{max}}{B_i C_{max}} \right\} \right]. \tag{2}$$

The metric β is equal to one when C_{max} and B_{max} pertain to the same PE, and larger otherwise (but never larger than two).[1] In our sf meshes, partitioned among various numbers of processors, we have never seen β exceed 1.15, and it is often unity; hence, we are confident that our model approximates reality well.

4 Model validation on Cray T3D

In this section we outline a simple method for estimating the system-specific parameters T_l and T_w, and then we use the method to derive the parameters for a Cray T3D using MPI to handle communication. Using the parallel SMVP in a real application (the sf2 Quake code), we scale the size of each message by some factor x and then measure and plot the elapsed time y of the communication phase as a function of

[1]The final version of the paper will include a proof as an appendix.

x. (Note that changing the message sizes in this manner does not necessarily yield a working application. However, by measuring the change in communication time as a function of changing message length, we can discriminate between bandwidth-related and latency-related overheads.) The parameter T_l is determined from the y-intercept of the resulting line, and T_w is determined from its slope. These parameters values can then be substituted into Equation (1) to predict the sustained bandwidth (and thus the total running time) of the communication phase of an SMVP operation.

Figure 6 lists the absolute running times for the computation and communication phases of various sf2 SMVP operations. Notice that for the T3D, the communication phase represents less than 10% of the total running time. However, as we shall see, this percentage becomes significantly larger if we replace our multicomputer with a network of workstations, and hence incur a much larger block latency T_l.

	16 PEs	32 PEs	64 PEs
comp (ms)	195.8	99.3	51.4
comm (ms)	4.9	4.5	4.1
total (ms)	17.9	109.3	59.2

Figure 6: Running time of sf2 matrix-vector product on Cray T3D (ms)

Figure 7 graphs the measured running times on T3D of the sf2 SMVP communication phase, as a function of different message size scaling factors. The numeric values are given in Figure 8.

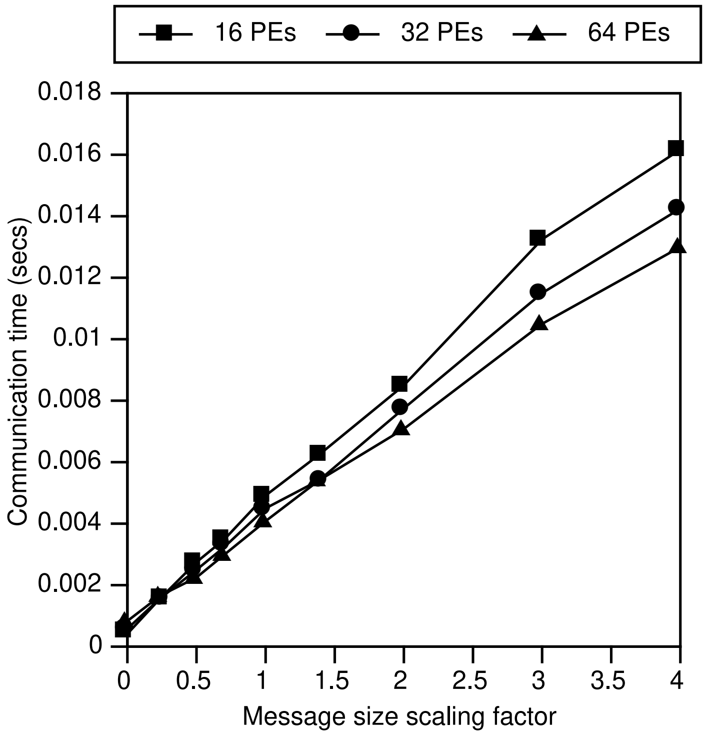

Figure 7: Running time for the communication phase of the sf2 SMVP on the Cray T3D

Each curve in Figure 7 corresponds to the running time of the sf2 application as a function of the scaling factor. Let x be a scaling factor and let $y(x)$ be the running time as a function of x. There are several

scaling factor	Communication times (us)		
	16 PEs	32 PEs	64 PEs
0	462	577	848
1/4	1606	1592	1699
1/2	2789	2517	2255
$1/\sqrt{2}$	3510	3302	3011
1	4939	4524	4092
$\sqrt{2}$	6263	5418	5454
2	8499	7776	7098
3	13251	11513	10535
4	16152	14231	13003

Figure 8: Raw data for Figure 7

interesting aspects of Figure 7. First, we see that the communication times for different numbers of PEs scale nearly linearly, which indicates that network congestion is not an issue. If it were, then we would see concave curves rather than nearly straight lines.

Second, given some curve y and some scaling factor x, $y(0)$ is the running time associated with the block latency T_l and $y(x) - y(0)$ is the running time associated with the marginal cost T_w of transferring an additional word. In particular,

$$T_l = \frac{y(0)}{B_{max}} \tag{3}$$

and

$$T_w = \frac{y(x) - y(0)}{xC_{max}} \tag{4}$$

For a particular x, T_w is directly proportional to the slope of y evaluated at x.

Both T_l and T_w can be computed quite easily once we know the application-specific parameters C_{max} and B_{max}. These parameters for the sf2 SMVP, partitioned for execution on 16, 32, and 64 PEs, are summarized in Figure 9 for a range of scaling factors between zero and four. The parameters are determined directly from properties of the partitioned meshes that induce the SMVP operations [9].

We can use Equations (3) and (4), and the application-specific parameters in Figure 9 to arrive at reasonable estimates for T_l and T_w for the T3D. The average T_l thus derived for 16, 32, and 64 PEs is roughly 25 ns, and we use that as an estimate of the block latency on the T3D. Similarly, the average T_w for 16, 32, and 64 PEs is roughly 160 ns, and we use that as the estimate for the marginal cost of transferring a block word. While not exact, they give excellent predictions of actual running time for the communication phase of the sf2 SMVP operation, as shown in Figure 10.

5 Implications for Graph Partitioners

Modern graph partitioners typically try to minimize global communication volume (i.e., the sum of the communication volumes, C, on each PE) subject to the constraint that each PE gets an equal amount of computational work. While this is a convenient metric for algorithm designers, it is not clear how closely this metric is related to the actual running time of parallel programs. This is mainly due to the facts that (1) it ignores the number of neighbors that each PE must communicate with, and (2) it does not attempt to someow balance communication across PEs.

Scaling factor	16 PEs B_{max}	16 PEs C_{max}	32 PEs B_{max}	32 PEs C_{max}	64 PEs B_{max}	64 PEs C_{max}
0	16	0	26	0	36	0
$1/4$	16	7120	26	6004	36	5130
$1/2$	16	14241	26	12009	36	10260
$1/\sqrt{2}$	16	19937	26	16812	36	14363
1	16	28482	26	24018	36	20520
$\sqrt{2}$	16	39874	26	33625	36	28727
2	16	56964	26	48036	36	41040
3	16	85446	26	72054	36	61560
4	16	113928	26	96072	36	82080

Figure 9: Application parameters for the sf2 SMVP

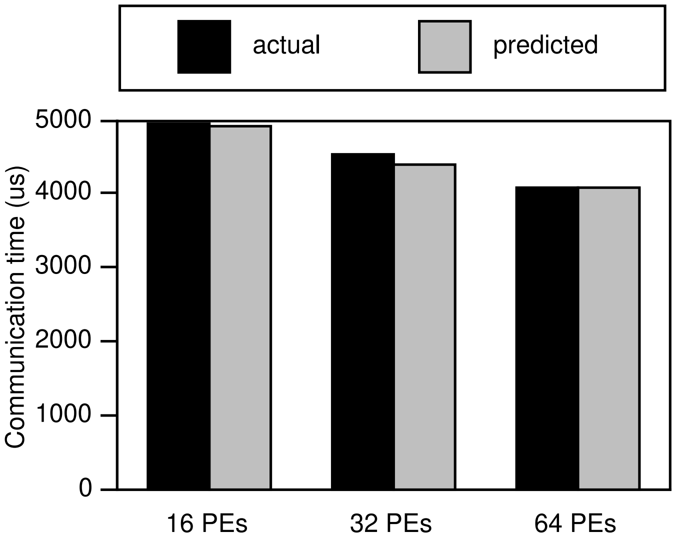

Figure 10: Predicted vs actual communication performance for the sf2 SMVP ($T_l = 25us$, $T_w = 160ns$)

The qualitative significance of (1) and (2) for real applications can be be seen quite clearly by using the model in (1) to separate the relative impact of block latency (T_l) and burst bandwidth (T_w^{-1}) on the running time. Consider Figure 11, which shows the time due to block latency (BT_l) versus the time due to burst bandwidth (CT_w) for the sf2 SMVP operation partitioned for execution on 32 PEs. Each point in Figure 11 represents the latency and bandwidth components (BT_l, CT_w) of the total communication time $BT_l + C_Tw$ for a PE involved in the SMVP operation. There are three clusters of points, with each cluster corresponding to a different assumption about the block latency T_l. (1) $T_l = 25us$: This represents the low-latency case, and is the measured block latency for the Cray T3D using MPI. (2) $T_l = 200us$: This represents an average-latency case, such as we might find on a high-overhead message passing system like some PVM implementations. (3) $T_l = 1000us$: This represents the high-latency case, such as we might find on a network of workstations where a message passing operation requires a supervisor call. Notice for the low-latency case that the absolute communication time is a small fraction of the total SMVP time. However, for the medium and high latency cases the communication time is a significant fraction of the

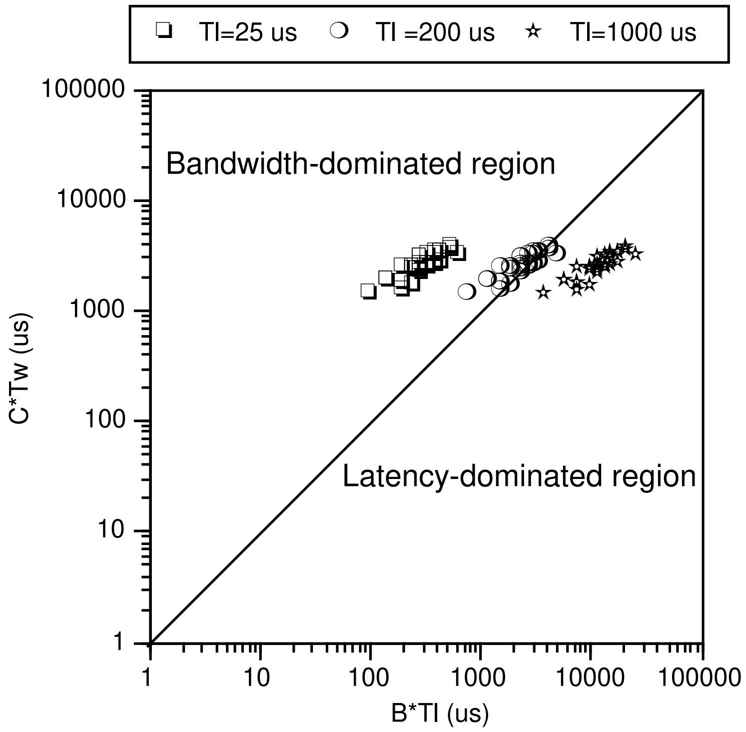

Figure 11: Impact of block latency and burst bandwidth on communication time for sf2 running on 32 PEs ($T_w = 160ns$)

overall SMVP time, and thus it is important to reduce this time.

Since the slowest PE determines the running time, we would like to minimize the maximum running time on any PE so that the the communication time is evenly balanced across all the PEs. If the communication phase were perfectly balanced, then all the points in a cluster in Figure 11 would fall on a 45-degree diagonal extending from the top left corner to the bottom right corner of the graph. In practice, however, we see that the communication times are not balanced across the PEs and that both the latency and bandwidth components of the individual PE running times can vary significantly. The point is that by paying attention only to the global communication volume, the partitioner produces an SMVP communication phase with unbalanced communication.

Another interesting aspect of Figure 11 is the degree to which different assumptions on block latency can cause the communication phase of the same application to be either dominated by the block latency component, dominated by the burst bandwidth component, or balanced between the two. Applications that are latency-dominated lie in the lower triangle, those that are bandwidth-dominated lie in the upper triangle, and those that are balanced between the two lie along the diagonal. For the sf2 SMVP, we see that the low-latency case ($T_l = 25ns$) is bandwidth-dominated, that the high-latency case ($T_l = 1000ns$) is latency-dominated, and that the medium-latency case is influenced equally by both latency and bandwidth. Thus, latency can be a significant component of communication performance for real applications. And by using global communication volume as the only optimization metric, the partitioner is ignoring the impact of an important component in the running time of the communication phase.

The main point is that modern partitioners are pursuing an optimization goal (minimizing the global communication volume) that does not necessarily minimize communication time because it ignores the impact on communication time of both unbalanced communication and block latencies. Global communication volume is certainly a convenient optimization criterion because it is a single measure and it corresponds directly to nice mathematical properties like separator sizes. Unfortunately, it doesn't reflect the reality that block latency and communication balance matter.

Instead of minimizing global communication volume, we recommend an approach, based on the results in this paper, where partitioners directly try to minimize total communication time. We have seen from the model in (1) that running time can be accurately predicted by the metric $B_{max}T_l + C_{max}T_w$. The partitioner can compute the application parameters B_{max} and C_{max} directly from the graph as it is being partitioned, and the system parameters can be derived using the techniques outlined in Section 4 and then passed as run-time arguments to the partitioner. Again, we emphasize that we are not proposing a new partitioning algorithm. Instead we are suggesting a system-dependent objective function that future partitioners should seek to optimize.

6 Conclusions

We have developed a simple performance model for sparse matrix-vector product (SMVP) operations such as those required by unstructured finite element applications. The parameters that characterize the block latency and burst bandwidth of the underlying system can be measured in a straightforward way and used to accurately predict the running time of the communication phase of the SMVP operations from a real application.

Experience with the model on real applications has provided us with the following insight into the limitations of current graph partitioning technology: by attempting to minimize only global communication volume, modern graph partitioners ignore the impact of block latency and overall communication imbalance. Partitioners will be more effective if they can reduce the maximum degree of adjacency of the subgraphs (especially for networks of workstations), and balance the communication volume among processors. We propose a attempting to directly minimize running time on a particular system, based on some simple application-specific and system-specific parameters.

We have been able to quantify the extent to which a real application can be both dominated and unbalanced by block latency; our approach attacks these problems directly. However, we have not measured how much such a direct approach can improve running time, compared with the existing approach of minimizing communication volume. We intend to pursue these questions, and encourage other researchers to do the same.

References

[1] BAO, H., BIELAK, J., GHATTAS, O., O'HALLARON, D. R., KALLIVOKAS, L. F., SHEWCHUK, J. R., AND XU, J. Earthquake ground motion modeling on parallel computers. In *Proceedings of Supercomputing '96* (Pittsburgh, PA, Nov. 1996).

[2] BARNARD, S., AND SIMON, H. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. Tech. Rep. RNR-92-033, NASA Ames Research Center, Nov. 1992.

[3] DWARKADAS, S., COX, A., AND ZWAENEPOEL, W. An integrated compile-time/run-time software distributed shared memory system. In *Proc. Sixth Intl. Conf. on Architectural Support for Prog. Languages and Operating Systems (ASPLOS VI)* (Boston, Oct. 1996), ACM, pp. 186–197.

[4] GILBERT, J., MILLER, G., AND TENG, S.-H. Geometric mesh partitioning: Implementation and experiments. In *9th International Parallel Processing Symposium* (Santa Barbara, April 1995), IEEE, pp. 418–427.

[5] HENDRICKSON, B., AND LELAND, R. The chaco user's guide version 2.0. Tech. Rep. SAND95-2344, Sandia National Laboratories, July 1995.

[6] HOLT, C., SINGH, J., AND HENNESSY, J. Application and architectural bottlenecks in large scale distributed shared memory machines. In *Proc. 23nd Intl. Symposium on Computer Architecture* (Philadelphia, PA, May 1996), ACM, pp. 134–145.

[7] LENOSKI, D., LAUDON, J., GHARACHORLOO, K., WEBER, W., GUPTA, A., HENNESSY, J., HOROWITZ, M., AND LAM, M. The Stanford Dash multiprocessor. *IEEE Computer 25*, 3 (Mar. 1992), 63–79.

[8] MILLER, G., TENG, S., THURSTON, W., AND VAVASIS, S. Automatic mesh partitioning. Tech. Rep. CTC92TR112, Cornell Theory Center, Nov. 1992.

[9] O'HALLARON, D., AND SHEWCHUK, J. Properties of a family of parallel finite element simulations. Tech. Rep. CMU-CS-96-141, School of Computer Science, Carnegie Mellon University, Nov. 1996.

[10] SCOTT, S. Synchronization and communication in the T3E multiprocessor. In *Proc. 7th. International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, Oct. 1996), ACM, pp. 26–36.

[11] STRICKER, T., AND GROSS, T. Optimizing memory system performance for communication in parallel computers. In *Proc. 22nd Intl. Symposium on Computer Architecture* (Santa Marguerita di Ligure, Italy, June 1995), ACM, pp. 308–319.