Fourth Symposium on High Performance Computer Architeture, February 1998, Las Vegas, NV

Architectural Implications of a Family of Irregular Applications

David O'Hallaron, Jonathan Richard Shewchuk, and Thomas Gross

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

Abstract

Irregular applications based on sparse matrices are at the core of many important scientific computations. Since the importance of such applications is likely to increase in the future, high-performance parallel and distributed systems must provide adequate support for such applications. We characterize a family of irregular scientific applications and derive the demands they will place on the communication systems of future parallel systems. Running time of these applications is dominated by repeated sparse matrix vector product (SMVP) operations. Using simple performance models of the SMVP, we investigate requirements for bisection bandwidth, sustained bandwidth on each processing element (PE), burst bandwidth during block transfers, and block latencies for PEs under different assumptions about sustained computational throughput. Our model indicates that block latencies are likely to be the most problematic engineering challenge for future communication networks.

1 Introduction

The peak performance of the commodity processors in modern parallel systems is doubling every couple of years. Clearly, communication performance needs to improve as processor performance increases, but the question is how much and in what respects. It is crucial to understand communication requirements because some parts of a high-performance communication system cannot be commodity, and will therefore be expensive.

In general it is important to understand the communication requirements of real applications [5, 10], and these requirements are especially difficult to characterize for the important class of irregular scientific applications that manipulate sparse matrices. Such applications typically model natural phenomena like the flow of air over a wing or the distribution of stresses and strains in a bridge, and tend to be large, complex, and poorly understood.

This paper addresses the following question: As microprocessors in parallel systems continue to improve, how much must communication systems improve to run irregular applications efficiently? Specifically, we address this question for the *Quake applications*, a family of 3D unstructured finite element simulations. The Quake applications, described in Section 2, simulate the motion of the ground during strong earthquakes. They were developed as part of an ongoing project at Carnegie Mellon to model earthquakes in the Los Angeles basin and other alluvial valleys [2].

Running time of the Quake applications is dominated by a sparse matrix-vector product (SMVP) operation that is repeated thousands of times, and the SMVP is the only operation besides I/O that requires the transfer of data between processors. Thus, as modelers we can focus on understanding the behavior of the SMVP operation and abstract away much of the complexity of the application. We derive a simple performance model for operations that consist of separate computation and communication phases—particularly the SMVP—in Section 3.

In Section 4 we apply this model to derive requirements for (1) bisection bandwidth, (2) sustained communication bandwidth per processing element (PE), and (3) block transfer latency and burst bandwidth, under various assumptions about efficiency and local MFLOP rates. (We use *processing element* instead of the common term *node* to avoid confusion with the nodes of finite element meshes.)

Our detailed characterizations of the requirements of the Quake applications reveal that bisection bandwidth is not important for irregular finite element applications; bandwidth at each PE is what matters. For systems with a sustained computational rate of 200 MFLOPS, PEs will need about 300 MBytes/sec of *sustained* bandwidth and 600 MBytes/sec of burst bandwidth to run irregular codes with good efficiency.

Our work is similar in spirit to that of Cypher, Ho, Konstantinidou, and Messina [5], who characterize eight regular and irregular scientific applications in terms of memory, processing, communication, and I/O requirements, and build scalability models for three of the simpler regular applications. However, our approach is different in that our goal is depth rather than breadth. We provide a detailed characterization for a specific family of irregular applications that are real (in the sense that people really care about the results that the Quake applications compute), that we understand completely, that we have complete control over, and that we can make arbitrarily large or small by adjusting the frequency of ground motion.

To show that the generality of our model seems to extend beyond this single family of irregular applications, we observe that there is some evidence that the Quake applications are typical of unstructured finite element simulations. For example, the EXFLOW application from Cypher et al. [5] is a 3D unstructured finite element program that simulates a fluid dynamics problem on 512 PEs. Interestingly, EXFLOW has nearly identical computational properties as a similarly sized Quake application (sf2/128, which resolves a wave with a two-second period on 128 PEs). EXFLOW and Quake each require about 2 MBytes of data on each PE. The communication volume/MFLOP is 144 KBytes for EXFLOW, vs. 155 KBytes for Quake. The average number of messages/MFLOP is 66 messages for EXFLOW, vs. 60 messages for Quake. And the average message size is 2.2 KBytes for EXFLOW, vs. 3.6 KBytes for Quake.

These two unstructured finite element applications are from two different scientific domains, yet each has similar computational properties and differs from regular applications in similar ways. Irregular finite element applications like EXFLOW and Quake have an average total communication volume similar to that of the regular applications studied earlier by Cypher et al. [5], but they transfer more messages having a smaller average size than most of those regular applications. Perhaps our most troublesome conclusion is that because block transfers between PEs tend to be small even for large irregular applications, block latency costs cannot be amortized by large messages, and communication latency will need to be a central focus of future efforts to engineer effective communication networks and software.

2 The family of Quake applications

The Quake applications are unstructured finite element codes that were developed to predict ground motion in the San Fernando Valley of Southern California during earthquakes [2]. There are four Quake applications, denoted sf10, sf5, sf2, and sf1. The "sf" is an abbreviation for San Fernando, and the number indicates the period (in seconds) of the highest frequency wave that the simulation is able to resolve. For example, sf10 resolves waves with 10 second periods, sf5 resolves waves with 5 second periods, etc.

Each Quake application consists of (1) a three-dimensional unstructured finite element model of the ground beneath San Fernando, and (2) a parallel finite element program that simulates the propagation of seismic waves through the model for 60 seconds of simulated time.

2.1 Quake finite element models

Each Quake application employs a 3D unstructured mesh, composed of thousands or millions of tetrahedra (i.e., pyramids with triangular bases), and models a volume of earth roughly 50 km x 50 km x 10 km in size (Figure 1). Each tetrahedron 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. To ensure that the simulation is numerically stable, the size of elements in any region of the mesh must be matched to the wavelength of ground motion, which is shorter in softer soils and longer in hard rock. Thus, softer soils need a higher density of smaller elements.

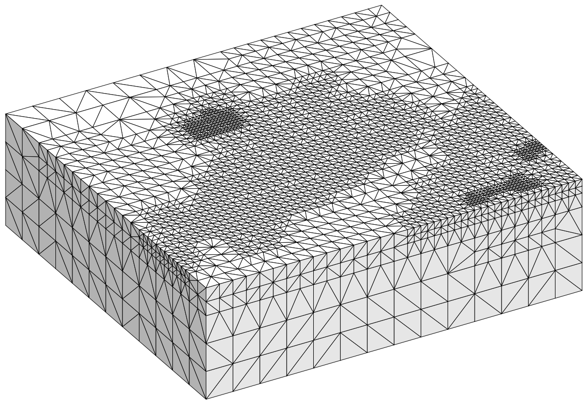


Figure 1. Finite element mesh for the sf10 model of the San Fernando Valley.

Figure 2 records the sizes of the San Fernando meshes. When the wave's period is halved, its frequency doubles, and the number of nodes increases by a factor of nearly eight—a factor of two for each of three dimensions. As a general rule, for each node in the mesh, a simulation uses about 1.2 KByte of memory at runtime to accommodate the storage of several vectors and sparse matrices. For example, sf2 requires about 450 MBytes of memory at runtime.

2.2 Parallel finite element simulations

During each of 6000 time steps, a Quake finite element simulation executes a sparse matrix-vector product (SMVP) operation of the form $y = Kx$, where x and y are vectors of

Number of...	sf10	sf5	sf2	sf1
nodes	7,294	30,169	378,747	2,461,694
elements	35,025	151,239	2,067,739	13,980,162
edges	44,922	190,377	2,509,064	16,684,112

Figure 2. Sizes of the Quake meshes.

length $3n$ (here n is the number of nodes), and K is a sparse $3n \times 3n$ *stiffness matrix*. Because an explicit time-stepping method is used, there are no other parallel operations (such as dot products or preconditioning). The vectors x and y have length $3n$ because each vector represents three degrees of freedom—x, y, and z displacements—for each node of the mesh. K can be likened to an adjacency matrix of the nodes of the mesh; K contains a 3×3 submatrix for each pair of nodes that are 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 of K contains an average of $14 \times 3 = 42$ nonzero floating point numbers [15].

The simulations are parallelized using Archimedes, a domain-specific tool chain for finite element problems [2, 18]. 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*; a one-to-one mapping is established between PEs and subdomains. The program that partitions each mesh into subdomains is based on a recursive geometric bisection algorithm [12] that divides the elements equally among the subdomains while attempting to minimize the total number of nodes that are shared by multiple subdomains, and hence the total communication volume. The geometric partitioning algorithm has provable asymptotic upper bounds on the number of shared nodes, and in practice generates partitions that are competitive with those produced by other modern partitioning algorithms [7, 3, 8].

2.3 The Parallel SMVP

The running time of the Quake applications is dominated by the execution of SMVP operations, which consume over 80% of the total running time in the sequential case. Furthermore, the SMVP operations are the only operations (besides I/O) that require interprocessor communication. So even though the Quake applications are complicated, we can abstract away most of the complexity and focus on the performance of a simple well-defined parallel SMVP kernel.

To compute the global SMVP operation $y = Kx$ on a set of PEs, we must consider the data distribution by which vectors and matrices are stored. The vectors x and 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 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.

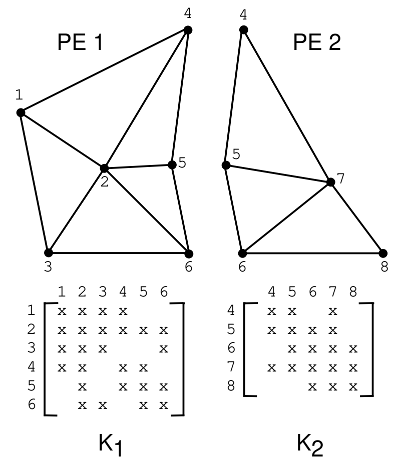


Figure 3. A finite element mesh and corresponding stiffness matrix K, distributed between two PEs. Each X represents a 3×3 submatrix.

With this method of distributing data, the global SMVP $y = Kx$ is performed in two steps. First, each PE computes a local SMVP over the subdomain that resides on that PE. Second, PEs that share nodes communicate and sum their nodal y values into correct global values for each node. In Figure 3, PEs 1 and 2 must communicate with each other to resolve the values of the shared nodes 4, 5, and 6.

3 An SMVP performance model

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 PE it shares nodes with (the communication phase). The computation and communication phases are synchronous, i.e., each phase begins with a global barrier synchronization.[1] Further, we assume (1) that during the communication phase, the summing operations are overlapped with the exchanges, and (2) that the

[1] It is in principle possible, with difficult modifications to the applications, to improve performance by overlapping the computation and communication phases, but the implementations of the Quake application do not do this, so the models in this section follow suit. (This paper undertakes to discuss how architects can accommodate scientific users, and not vice versa.) By not modeling any overlap, we obtain conservative bandwidth and latency estimates, and avoid possibly slowing the program by complicating the runtime system.

time to perform the exchanges determines the running time of the communication phase. These assumptions are reasonable, 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

$$T_{smvp} = T_{comp} + T_{comm},$$

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.)

Parameters for Equation (1)	
F	flops per PE per SMVP
C_{max}	maximum communication words per PE per SMVP
E	target efficiency
T_f	amortized time per flop (inverse of sustained flops)
Parameters for Equation (2)	
B_{max}	maximum communication blocks per PE per SMVP
C_{max}	maximum communication words per PE per SMVP
T_l	time per communication block (block latency)
T_w	time per additional block word (inverse of burst bandwidth)
Computed quantities	
T_{smvp}	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
T_c	amortized time per comm word (inverse of sustained bwidth)
β	upper bound on relative overestimate of communication time
M_{avg}	average message size (words)

Figure 4. Summary of symbols.

3.1 Model of the computation phase

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 the 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 PE's 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 flop requires an average time of T_f, then the running time for a local SMVP is FT_f. Modern graph partitioners do an excellent job of distributing computation evenly across PEs [15], so we will assume that each PE performs $F = 2m$ flops and thus the running time for the computation phase 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. For example, measurements of the local SMVPs from the Quake applications show a steady $T_f = 30$ ns for the Cray T3D (150 MHz Alpha 21064, cc -O3) and $T_f = 14$ ns for the Cray T3E (300 MHz Alpha 21164, cc -O3).

3.2 High level model of the communication phase

The hardware part of the communication system of a PE is modeled as a network interface (NI) with an input link and output link, as shown in Figure 5.

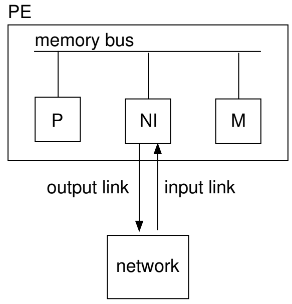


Figure 5. Processing element (PE) model (P: processor, M: memory system, NI: network interface).

The running time of the communication phase is given by

$$T_{comm} = C_{max}T_c$$

where C_{max} is the maximum number of words communicated by any PE during the communication phase and T_c is the average time per communication word. The rate T_c^{-1} is the *sustained bandwidth* across the links of the network interface during the communication phase. T_c includes the time to transfer data in and out of a PE, as well as all software and hardware overheads.

We define the *efficiency* E to be the proportion of SMVP time devoted to computation; that is, $E = T_{comp}/T_{smvp}$. Given this definition, $T_{comm}/T_{comp} = (1-E)/E$. We have seen that $T_{comm}/T_{comp} = (C_{max}T_c)/(FT_f)$, and hence

$$T_c = \frac{F}{C_{max}}\left(\frac{1-E}{E}\right)T_f. \tag{1}$$

Equation (1) describes at a high level the relation between sustained computation bandwidth (T_f^{-1}) and sustained communication bandwidth (T_c^{-1}). The model is interesting because it cleanly separates the various factors that relate computation performance to communication performance: The *computation/communication ratio* F/C_{max} is a property of the application and the partitioner; T_f is a property of the processor architecture and compiler; E is a target efficiency imposed by the user. This separation of factors is similar in spirit to the familiar CPI model for uniprocessor performance [9].

In Section 4 we use Equation (1) to investigate the following question: given a target efficiency E for an SMVP running on PEs with local computational rates of T_f^{-1}, what is the required communication performance of the system? The answers to this question provide insight, for this class of applications, into the sustained performance we need from our communication systems as processor speeds continue to increase.

3.3 Low level model of the communication phase

Equation (1) provides a high level model of sustained performance in terms of streams of words being transferred between memory and the network interface. However, in a lower level and more realistic view of communication, the unit of work during 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 *block transfer time*. A block is a transfer unit, 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 [11], a message in a message passing system [13], a bulk asynchronous data transfer between two PEs' memories [17], or a page in a software distributed shared memory system [1].

The transfer time for a block i of l_i words is $T_l + l_i T_w$, 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 an interconnection network. We only model the overhead of transferring data between the network interface and local memory, in view of previous findings that most of the cost of communication on modern systems is incurred at the individual PEs [19]. In essence, we assume that the interconnection network has infinite capacity and constant latency. In an expanded version of this paper [16], we argue empirically that this is a reasonable assumption for the SMVP running on tightly coupled systems.

Modern graph partitioners do an excellent job of balancing computation (F) and a good job of minimizing the global volume of communication. Unfortunately they do less well in balancing the total number of blocks B_i and the total volume in words sent and received by each PE i [15]. As a simplifying assumption, we pessimistically assume that the PE that transfers the maximum number of words (C_{max}) is the same PE that transfers the maximum number of blocks (B_{max}). In this case, the same PE has the longest communication phase, and thus

$$T_{comm} = B_{max} T_l + C_{max} T_w.$$

Finally, since $T_c = T_{comm}/C_{max}$, we have

$$T_c = \frac{B_{max}}{C_{max}} T_l + T_w. \tag{2}$$

T_l and T_w are properties of the communication system, including the architecture, hardware implementation, and software libraries. Elsewhere [16], we describe a simple methodology for estimating these parameters on real systems. For example, for the Cray T3E, we have $T_l = 22\ \mu$s and $T_w = 55$ ns. For message-passing systems, B_{max} (the maximum number of blocks transferred by one PE) is purely a property of the application and the partitioner, but on shared-memory systems it is also a property of the architecture, as the data sent from one PE to another may have to be broken up into multiple blocks (i.e., cache lines). Thus, Equation (2) characterizes sustained communication bandwidth during the communication phase of the SMVP in terms of basic properties of the communication system and the application. In Section 4, we use this model to explore bandwidth and latency tradeoffs in communication systems as the base microprocessors continue to improve.

The models in Equations (1) and (2) are similar in some ways and different in others to the LogP model [4]. LogP is a general performance model for bulk-synchronous parallel (BSP) computations [20], where a program is viewed as a series of *supersteps* separated by barrier synchronizations. During a superstep, each PE performs local computation and transfers a limited number of messages. The models we have developed are for a restricted class of BSP programs where each superstep (SMVP) consists of a distinct computation phase followed by a distinct communication phase. In general our models view performance at a lower level of abstraction than LogP. The parameters T_f, T_w, F, B_{max}, and C_{max} have no counterparts in LogP. On the other hand, our T_l parameter is similar to the overhead parameter o in LogP.

3.4 An error bound

In (2) we assume that the PE with the most words to communicate also has the most blocks to communicate. In practice this assumption may not be true. The question is, by how much do we overestimate T_c (and thus T_{comm}) as a result? Elsewhere [16], we show that our model does not overestimate T_{comm} by more than a factor of

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

The bound β is an application property, independent of any target machine, that is equal to one when C_{max} and B_{max} pertain to the same PE, and larger otherwise (but never larger than two). Figure 6 shows the computed values of β for the Quake applications. Since β is close to one for each of these applications, (2) appears to be a reasonable model.

Subdomains	sf10	sf5	sf2	sf1
4	1.00	1.00	1.00	1.00
8	1.00	1.00	1.00	1.00
16	1.09	1.10	1.07	1.00
32	1.01	1.01	1.15	1.00
64	1.03	1.08	1.11	1.05
128	1.03	1.04	1.04	1.11

Figure 6. Computed relative error bounds β on T_c for each Quake application.

Subdomains		sf10	sf5	sf2	sf1
4	F	453,924	1,899,396	24,640,110	162,372,024
	C_{max}	2,352	7,746	55,338	186,162
	B_{max}	6	6	6	6
	M_{avg}	369	1,290	8,682	27,540
	F/C_{max}	193	245	445	872
8	F	235,566	970,740	12,414,006	81,602,442
	C_{max}	2,550	7,080	35,148	151,764
	B_{max}	12	12	10	14
	M_{avg}	237	699	4,152	13,761
	F/C_{max}	92	137	353	538
16	F	122,742	496,872	6,278,076	41,116,374
	C_{max}	2,208	5,292	28,482	119,280
	B_{max}	18	20	16	18
	M_{avg}	159	342	1,920	7,434
	F/C_{max}	56	94	220	345
32	F	64,980	257,004	3,191,436	20,740,734
	C_{max}	2,172	4,476	24,018	87,228
	B_{max}	30	30	26	26
	M_{avg}	87	213	1,239	4,044
	F/C_{max}	30	57	133	238
64	F	34,956	134,424	1,632,708	10,511,586
	C_{max}	1,764	4,296	20,520	73,062
	B_{max}	38	40	36	38
	M_{avg}	57	135	765	2,712
	F/C_{max}	20	31	80	144
128	F	18,954	70,956	838,224	5,332,806
	C_{max}	1,740	3,360	16,260	51,048
	B_{max}	62	52	50	46
	M_{avg}	36	135	459	1,515
	F/C_{max}	11	21	52	104

Figure 7. Quake SMVP properties. F: flops per PE. C_{max}: maximum communication words on any one PE. B_{max}: maximum communication blocks on any one PE. M_{avg}: Average message size (words). F/C_{max}: computation/communication ratio.

4 Communication requirements

In this section, we use our performance models to address the following question: As the base commodity microprocessors in parallel systems get faster, what kind of performance will be needed from communication systems in order to run the Quake SMVPs efficiently?

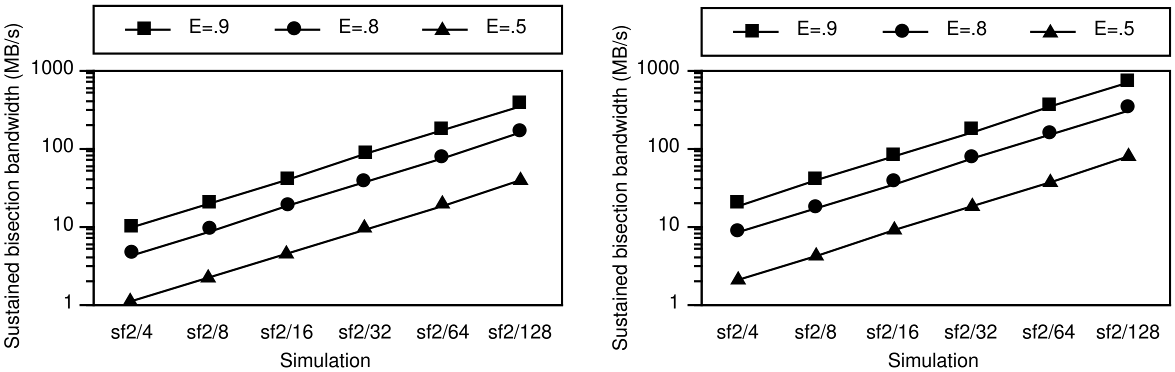
We will investigate this question for two hypothetical machines: A "current" machine that runs the local SMVP at a sustained rate of 100 MFLOPS ($T_f = 10$ ns), and a "future" machine that runs the local SMVP at a rate of 200 MFLOPS ($T_f = 5$ ns). (Specifications are for 64-bit floating-point values.) Computational rates of 100 to 200 MFLOPS may seem low, but they reflect the reality that the actual performance of irregular applications is far less than peak rated performance, largely because of irregular memory reference patterns and because the data structures are too large to fit in cache. For example, a single PE of a Cray T3E (300 MHz Alpha 21164, cc -O3) runs the local SMVP from the Quake applications at approximately 70 MFLOPS ($T_f = 14$ ns), which is only 12% of the peak rated performance of 600 MFLOPS.

The sf2 SMVP will serve as a running example. Sf2 is a good example for a number of reasons. First, it is clearly large enough to qualify as a real problem by any standard (the sparse coefficient matrix is 1.14M $\times$1.14M in size). Second, even though it is quite large, it has a wide range of computation/communication ratios (F/C_{max}), from a high of 450:1 when partitioned into four subdomains, down to 50:1 for 128 subdomains.

4.1 Properties of the Quake applications

Figure 7 summarizes the properties of the various Quake SMVP instances. (See O'Hallaron and Shewchuk [15] for a complete characterization of the applications.) The values of B_{max} and C_{max} in this table are always even, because each message from PE i to PE j is matched by a message from j to i of equal length. (The values of C_{max} are also divisible by three, because there are three degrees of freedom per node.) The values of B_{max} indicate the degree of subdomain adjacency; for instance, if $B_{max} = 46$, then each subdomain shares mesh nodes with at most 23 other subdomains, and the corresponding PE must communicate with 23 other PEs. These values assume that blocks may be arbitrarily large, and hence each PE sends at most one block to each other PE. On a fine-grained shared memory machine, where the unit of interchange between PEs may be as small as a cache line, B_{max} may be much larger.

There are some interesting points to make about the numbers in Figure 7. First, conventional wisdom holds that sparse codes like the SMVP are communication intensive. However, this is not always the case. As we see for sf2, which is a reasonably large problem, the computation/communication ratios F/C_{max} vary from large (450 for sf2/4) to moderate (50 for sf2/128). (We use the notation sf$x$/y to denote an instance of application sf$x$ partitioned into y subdomains.) This common misconception about sparse



(a) $T_f = 10$ ns (100 MFLOPS).

(b) $T_f = 5$ ns (200 MFLOPS).

Figure 8. Sustained bisection bandwidths required for sf2.

codes is probably due to the fact that researchers have not had the opportunity to run sufficiently large problems.

Second, while the computation/communication ratios are reasonably high for large problems, as the problem sizes grow by a factor of ten, we see that the computation/communication ratios grow only by a factor of two. This is not surprising; consider that a good partition of an n-node 3D mesh will produce $\mathcal{O}(n^{2/3})$ shared nodes (for the same reason that an $\mathcal{O}(n)$-node cube has a surface area of $\mathcal{O}(n^{2/3})$ nodes). Hence, the computation/communication ratio is $\mathcal{O}(n^{1/3})$, and a factor-of-ten increase in n yields roughly a factor-of-two increase in that ratio. Our point is that while large SMVPs indeed have reasonable computation/communication ratios, these ratios do not increase quickly with increasing problem size, as they do for cubic problems like dense matrix multiply. Thus, we cannot rely on simply increasing the problem size to guarantee good efficiency.

Third, even when blocks are as large as possible (i.e., each PE sends at most one block to any other PE), the mean block size M_{avg} is surprisingly small. For example, the largest mesh (sf1) running on 128 PEs has an average message size of only about 1,500 words. Thus, we cannot rely on large blocks to amortize high block latencies.

Finally, for sf1/128 each PE communicates with up to 20% of the other PEs. Thus, the Quake SMVP occupies an interesting middle ground between difficult applications like the 2D FFT that require an all-to-all communication, and simple applications like regular grid problems wherein PEs communicate with at most four neighbors.

4.2 Bisection bandwidth

Bisection bandwidth is a popular measure of communication system performance, but it proves to be unimportant for Quake SMVPs. To compute the bisection bandwidth requirements for a Quake SMVP running on p PEs, we create a symmetric $p \times p$ matrix m such that m_{ij} is the number of 64-bit words transferred from PE i to PE j. If we assume that PEs $0, \ldots, p/2-1$ are on one side of the bisection and PEs $p/2, \ldots, p-1$ are on the other side, then

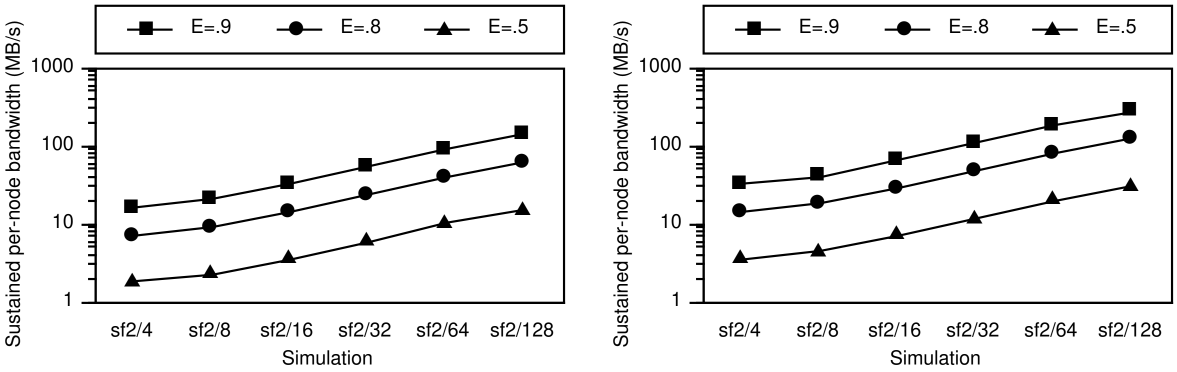
$$V = 2 \sum_{i=0}^{p/2-1} \sum_{j=p/2}^{p-1} m_{ij}$$

words cross the bisection during the communication phase, and the sustained bisection bandwidth is $V/(C_{max}T_c)$. Figure 8 shows the required bisection bandwidths given different assumptions of PE performance and overall efficiency. The worst case of 700 MBytes/sec ($E = 0.9$ and $T_f^{-1} = 200$ MFLOPS) is quite modest, on the order of the bandwidth of a couple of links in a modern system. Bisection bandwidth is unlikely to be an issue for SMVPs as local PE performance increases.

4.3 Sustained PE bandwidth

Figure 9 plots the sustained bandwidth for each PE (T_c^{-1}) that is required for the sf2 SMVP under different assumptions of PE performance and overall efficiency. The required bandwidths are computed using Equation (1) and the properties from Figure 7. On a system with 100-MFLOP PEs, maintaining a sustained rate of 120 MBytes/sec per PE during the communication phase is sufficient to run all instances of the sf2 SMVP at 90% efficiency. On systems with 200-MFLOP PEs, a sustained PE bandwidth of about 300 MBytes/sec will be required to run all of the sf2 SMVPs at 90% efficiency.

A sustained PE bandwidth of 300 MBytes/sec seems like an easy performance target until one remembers that it in-



(a) $T_f = 10$ ns (100 MFLOPS).

(b) $T_f = 5$ ns (200 MFLOPS).

Figure 9. Sustained PE bandwidth (T_c^{-1}) required for sf2.

cludes all overheads, including software overheads, local strided read and write copies, and other latencies; we are not concerned here with peak link bandwidth ratings. For example, even though the optimal throughput of strided copies on the Cray T3D is 30–40 MBytes/sec [19], current implementations of sf2 achieve at best a measured and sustained bandwidth of 10 MBytes/sec using the C interface to the vendor-supplied MPI library [2].

Even more daunting is the goal of running the Quake applications at say 80% efficiency on high-speed networks of workstations. This goal is attractive because it would make it possible to obtain enough memory to run much larger simulations, but demands sustained per-PE bandwidths of about 100 MBytes/sec.

4.4 Bandwidth and latency tradeoffs

There is an interesting tradeoff between the burst bandwidth and latency that are necessary to achieve some target sustained bandwidth on each PE. If block latency is made smaller, then burst bandwidth can be larger, and vice-versa. Equation (2) quantifies this tradeoff, which is shown in Figure 10 for the sf2 SMVP running on 128 200-MFLOP PEs. Any point along a diagonal line represents a combination of burst bandwidth (T_w^{-1}) and latency (T_l) that meets the sustained PE bandwidth requirement (T_c^{-1}) for sf2/128 from Figure 9(b). The curves are generated by first using Equation (1) to compute the required inverse sustained bandwidth T_c, and then using Equation (2) to determine the appropriate combinations of T_l and T_w.

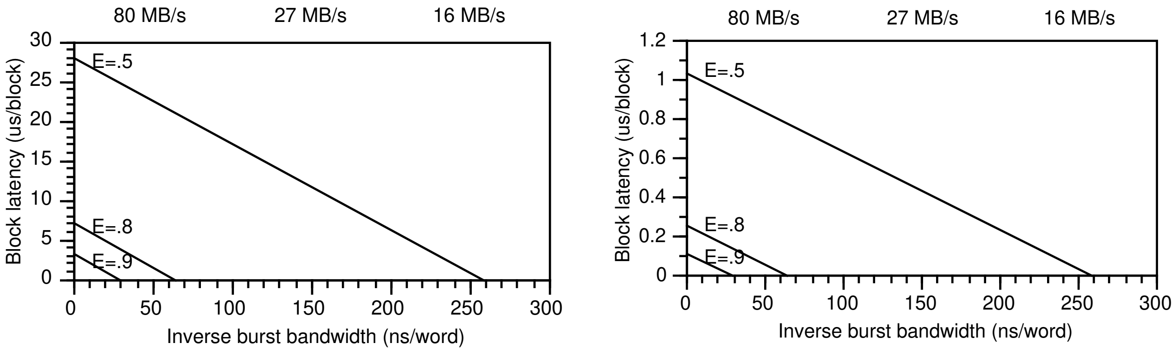
Figure 10(a) shows the burst bandwidth and latency tradeoffs under the assumption that blocks are as large as possible, and thus each PE sends at most one block to any other PE. This is the case in message passing systems or DSMs that aggregate blocks [6]. The interesting point about this graph is that latency matters for the SMVP. Even if burst bandwidth is driven to infinity, observed block latency must not exceed 3 μs if the code is to run at 90% efficiency.

We can also use Equation (2) to get a sense of the latency and bandwidth tradeoffs if blocks are small, fixed-sized objects like cache lines. To model the tradeoff for fixed-sized blocks consisting of four 64-bit words, we set $B_{max} = C_{max}/4$ and then plot block latency as a function of burst bandwidth as before, in Figure 10(b). Here, block latency is even more crucial, as expected. For example, if burst bandwidth is infinite, then observed block latency must not exceed 100 ns if the SMVP is to run at 90% efficiency.

For a given application, a reasonable network design goal is to achieve values for T_l and T_w^{-1} such that half of the time for the communication phase is due to block latency and the other half to burst bandwidth. We refer to a burst bandwidth chosen thusly as the *half-bandwidth*, and its corresponding latency as the *half-bandwidth latency*. These figures are useful targets for architects of communication systems because overengineering a network's bandwidth cannot relieve the latency constraint by more than a factor of two, and vice-versa. Figure 11 plots the half-bandwidths and half-bandwidth latencies for the entire space of sf2 SMVPs, for both the maximal block size and the four-word block size.

The bottom portion of the graph shows the requirements when blocks are fixed-sized four-word transfer units, as might occur in a shared-memory architecture. For the easiest case, running at 50% efficiency on 100-MFLOP PEs, we need a burst bandwidth of about 3 MBytes/sec with a block latency of about 10 μs. In the most difficult case, running at 90% efficiency on 128 200-MFLOP PEs, we would need a burst bandwidth of about 600 MBytes/sec and a block latency of about 70 ns.

The upper portion of the graph shows the requirements when blocks are made as large as possible (i.e., each PE



(a) Arbitrarily large block size.

(b) Four-word block size.

Figure 10. burst bandwidth and latency tradeoffs for sf2/128 (200 MFLOP PE).

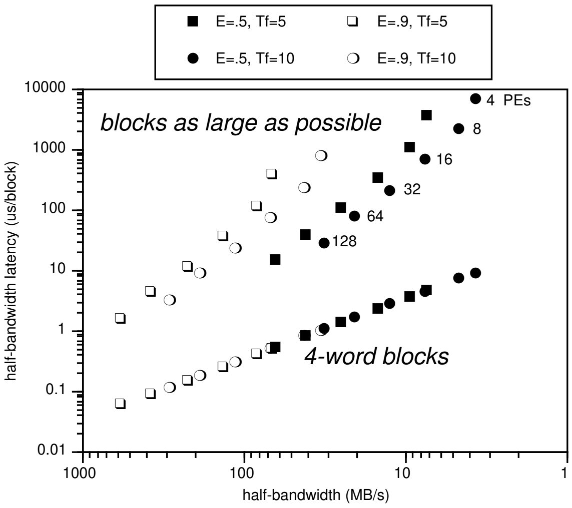


Figure 11. Half-bandwidths and latencies for the sf2 SMVP

sends at most one block to any other PE), as on a message-passing multiprocessor or network. For the simplest case, running at 50% efficiency on four 100-MFLOP PEs, we would need a burst bandwidth of about 3 MBytes/sec and a half-bandwidth latency of about 8 ms. However, the most demanding case, where we are running at 90% efficiency on 128 200-MFLOP PEs, requires a burst bandwidth of 600 MBytes/sec and a block latency of about 2 μs.

5 Concluding remarks

Irregular applications have been poorly understood in the past, in large part because researchers and system builders have not had access to large realistic example applications. This paper represents a step toward understanding the implications of large-scale irregular codes on the architecture of high-end systems. We have chosen to study unstructured finite element simulations of strong earthquake-induced ground motion because they constitute real and significant engineering applications currently in daily use, and because their performance is characterized by the performance of a simple SMVP kernel. By examining the behavior of the SMVP, we can diagnose the requirements placed on communication systems in future high-end computers and networks as the performance of commodity PEs continues to increase.

Our main conclusions are: (1) Bisection bandwidth is not an issue. (2) Block transfers tend to be small, even for large applications. Thus we cannot expect to amortize block latency costs with large messages; other latency hiding techniques must be used, or latency must be reduced. (3) Systems with sustained computational throughput of 200 MFLOPS and maximally aggregated blocks will need about 300 MBytes/sec of sustained bandwidth, 600 MBytes/sec of burst bandwidth, and a block latency under 2 μs to run unstructured finite element applications with 90% efficiency.

The bandwidth requirements per PE for future systems are aggressive, especially since they must include all hardware and software overheads. Yet they seem achievable. More worrisome are the block latency requirements (2 μs for large blocks, 7 ns for small blocks), which appear much more difficult to achieve. The numbers in this paper provide a strong quantitative argument for reducing latency in future systems, either with more efficient interfaces, latency hiding techniques, or both.

Postscript: Tools for further experimentation

The Spark98 suite [14], a collection of 10 portable sequential and parallel SMVP kernels written in C and based on the sf10 and sf5 meshes described in this paper, is available at www.cs.cmu.edu/∼quake/spark98.html. The complete set of partitioned San Fernando meshes is available from www.cs.cmu.edu/∼quake/meshsuite.html.

Acknowledgments

Many thanks to our Quake project partners from the CMU Department of Civil Engineering: Jacobo Bielak, Omar Ghattas, Hesheng Bao, Loukas Kallivokas, and Jifeng Xu, who developed the San Fernando simulations and generously shared their extensive knowledge. Gary Miller provided insighful comments. Joe Brandenburg and Mike Barton at Intel sparked our interest in characterizing large finite element simulations.

The Pittsburgh Supercomputing Center provided time on the Cray T3D and T3E sytems. Funding was provided in part by the National Science Foundation under Grant CMS-9318163, by the Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287, and by grants from Intel Corporation and Digital Equipment Corporation.

References

[1] C. Amza, A. Cox, S. Dwarkadas, C. Hyams, Z. Li, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[2] H. Bao, J. Bielak, O. Ghattas, D. O'Hallaron, L. Kallivokas, J. Shewchuk, and J. Xu. Earthquake ground motion modeling on parallel computers. In *Proc. Supercomputing '96*, Pittsburgh, PA, Nov. 1996. See also www.cs.cmu.edu/∼quake/.

[3] S. Barnard and H. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. Technical Report RNR-92-033, NASA Ames Research Center, Nov. 1992.

[4] D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):79–85, Nov. 1996.

[5] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architetural requirements of parallel scientific applications with explicit communication. In *Proc. 20th Intl. Symp. Computer Arch.*, pages 2–13. ACM/IEEE, May 1993.

[6] S. Dwarkadas, A. Cox, and W. Zwaenepoel. 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)*, pages 186–197, Boston, Oct. 1996. ACM.

[7] J. Gilbert, G. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. In *9th International Parallel Processing Symposium*, pages 418–427, Santa Barbara, CA, Apr. 1995. IEEE.

[8] B. Hendrickson and R. Leland. The Chaco user's guide Version 2.0. Technical Report SAND95-2344, Sandia National Laboratories, July 1995.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach (2nd Edition)*. Morgan Kaufman, 1995.

[10] C. Holt, J. Singh, and J. Hennessy. Application and architectural bottlenecks in large scale distributed shared memory machines. In *Proc. 23nd Intl. Symp. Comp. Arch.*, pages 134–145, Philadelphia, PA, May 1996. ACM.

[11] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, Mar. 1992.

[12] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Automatic Mesh Partitioning. In A. George, J. R. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*. Springer-Verlag, New York, 1993.

[13] MPI Forum. MPI: A Message Passing Interface. In *Proc. Supercomputing '93*, pages 878–883, Portland, OR, Nov. 1993. ACM/IEEE.

[14] D. O'Hallaron. Spark98: Sparse matrix kernels for shared memory and message passing systems. Technical Report CMU-CS-97-178, School of Computer Science, Carnegie Mellon University, Oct. 1997.

[15] D. O'Hallaron and J. Shewchuk. Properties of a family of parallel finite element simulations. Technical Report CMU-CS-96-141, School of Computer Science, Carnegie Mellon University, Dec. 1996.

[16] D. O'Hallaron, J. Shewchuk, and T. Gross. Architectural implications of a family of irregular applications. Technical Report CMU-CS-97-189, School of Computer Science, Carnegie Mellon University, Nov. 1997.

[17] S. Scott. Synchronization and communication in the T3E multiprocessor. In *Proc. 7th. Intl. Conf. on Arch. Support for Prog. Lang. and Oper. Systems*, pages 26–36, Boston, MA, Oct. 1996. ACM.

[18] J. Shewchuk. *Delaunay Refinement Mesh Generation*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1997. Also available as CMU Tech Report CMU-CS-97-137.

[19] T. Stricker and T. Gross. Optimizing memory system performance for communication in parallel computers. In *Proc. 22nd Intl. Symp. Comp. Arch.*, pages 308–319, Santa Marguerita di Ligure, Italy, June 1995. ACM.

[20] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.