UNSTRUCTURED MESH COMPUTATIONS ON NETWORKS OF WORKSTATIONS*

MARK T. JONES AND PAUL E. PLASSMANN†

Abstract. Unstructured mesh technology can be used to create highly efficient scientific and engineering application software. Networks of workstations (NOWs) are a cost-effective platform for the timely solution of large problems in science and engineering. The performance of unstructured mesh computations on NOWs is investigated in this paper. Several parallel unstructured mesh algorithms are informally shown to have computation and communication characteristics similar to those of parallel sparse matrix-by-vector multiplication. These characteristics are discussed, and the requirements they place on an interconnection network are described. Experimental data are given to summarize the communication parameters of four different NOW configurations. Finally, extensive empirical results are given to characterize the performance of unstructured mesh computations on the four NOWs.

Key words. Finite Elements, Networks of Workstations, Parallel Computing, Scalable Algorithms, Unstructured Meshes

1. Introduction. Parallel computers such as the IBM SP2 and the Intel Paragon are extremely powerful tools; however, their cost is beyond the means of most small to medium science and engineering groups. On the other hand, most groups do have access to a network of workstations (NOW). A NOW can be very cost efficient for the solution of mid-sized problems that are too large for a single workstation yet do not require a parallel computer with hundreds of processors. A potential disadvantage, however, is that the latencies incurred during interprocessor communication on a NOW are typically much larger than those for a dedicated parallel computer. For good performance on a NOW, careful attention must be paid to the demands placed by the algorithm on the interconnection network.

The use of unstructured mesh technology has been shown to reduce computation time and memory requirements in many scientific applications, including computational fluid dynamics and structural analysis. An unstructured mesh has no regular connections among the vertices in the mesh; in a structured mesh, excluding boundary vertices, the local connection pattern of each vertex is identical. An example of each type of mesh is given in Figure 1.1. Programs for structured meshes tend to be easier to write; however, unstructured meshes can be constructed to conform to virtually any geometry and are, for many problems, much more efficient in terms of memory use and CPU requirements.

Even with the use of unstructured mesh technology, many scientific and engineering applications require meshes with millions of vertices, particularly for three-dimensional problems. Parallel computing reduces the wall-clock time necessary to

* The first author received support from NSF grants ASC-9501583, CDA-9529459, and ASC-9411394. The second author was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

†The address of the first author is Computer Science Department, University of Tennessee, Knoxville, TN 37996. The address of the second author is Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439.

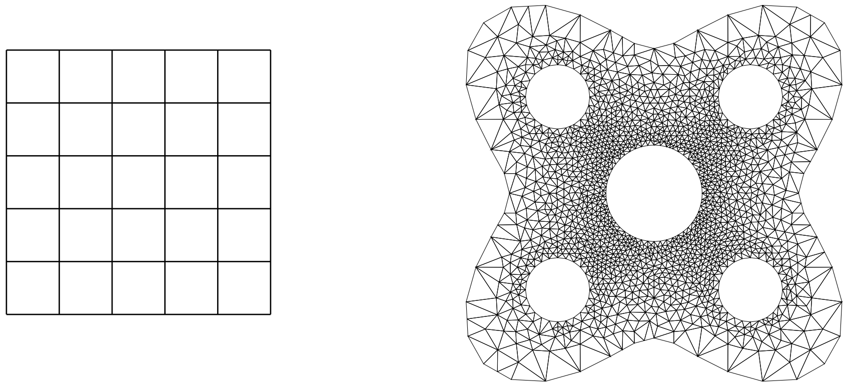

FIG. 1.1. *On the left a structured mesh with regular local vertex connection patterns, on the right an unstructured mesh without regular vertex connections or placement*

solve such problems and provides access to the required memory. An integrated approach to parallel algorithms and software for unstructured mesh computations, as well as computational results from the 512-processor Intel DELTA, given in [8].

In this paper we examine the performance of several aspects of unstructured mesh computations on four different NOW configurations. Further, we define the characteristics of these computations that are salient to performance and then delineate the requirements of the interconnection networks that effectively support these computations.

The remainder of the paper is organized as follows. In Section 2 we give an overview of the main computational tasks required in a parallel implementation of unstructured mesh methods. In Section 3 we examine the communication characteristics of the NOWs used in our experiments. Based on these characteristics, in Section 4 we define a problem class into which many parallel unstructured mesh algorithms fall, and we define the requirements of parallel architectures to support the scalable performance of this algorithm class. Computational results are given in Section 5. Finally, we give conclusions and suggestions for future work in Section 6.

2. Parallel Unstructured Mesh Computations. In Figure 2.1 we give the general algorithm for a typical unstructured mesh application; the computational tasks of interest in this paper are underlined. After each problem solution, areas of the mesh determined to have high discretization error are refined to decrease the error; a sequence of such meshes is shown in Figure 2.2. If the mesh has changed significantly, it must be repartitioned across the processors. Note that the sparse linear system solution may be part of a more complex solution process (e.g., a nonlinear system solution, an optimization problem, or an eigenvalue problem); however, it is typically the most computationally intensive and the most difficult part of the solution process to parallelize. In the following subsections, we will briefly discuss several key aspects of unstructured mesh applications.

2.1. Mesh Partitioning. To allow parallel computation, the mesh must be partitioned across the processors. When the mesh is altered (e.g., vertices are added or deleted) it may be necessary to repartition the mesh during the computation. In

Construct a mesh M_0 that conforms to the input geometry
If not already partitioned **then**
    Partition M_0 across the p processors
Endif
$i = 0$
Repeat
    Assemble a sparse matrix K_i from M_i
    Solve the linear system $K_i u_i = f_i$
    Estimate the error on M_i
    If maximum error estimate on M_i is too large **then**
        Based on the error estimates, refine M_i to get M_{i+1}
        If partitioning is not satisfactory for M_{i+1} **then**
            Repartition M_{i+1} across the p processors
        Endif
    Endif
    $i = i + 1$
Until maximum error estimate on M_i is satisfactory

FIG. 2.1. *A general algorithm for typical unstructured mesh applications. Operations discussed in this paper are underlined.*

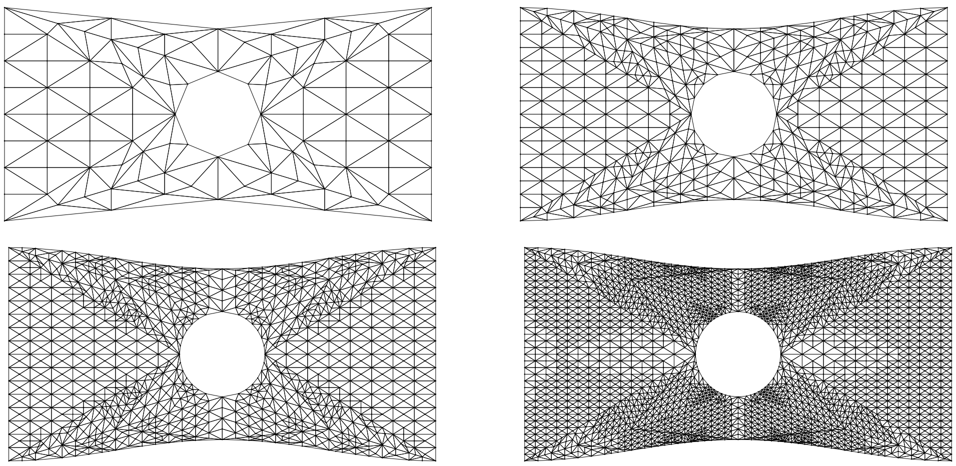

FIG. 2.2. *A sequence of meshes generated by the procedure in Figure 2.1.*

these mesh computations, we seek to partition the mesh such that every processor has approximately the same number of vertices and elements. Further, we want to minimize the communication between processors; hence, we should try to minimize the number of edges in the mesh that connect vertices on different processors (we refer to these edges as *cross edges*). In Section 5 we use the percentage of edges that are cross edges as one indicator of the computation-to-communication ratio; the lower the percentage, the better the ratio. Another indicator is the number of processors with which a processor shares a cross edge; this indicates the number of messages that a processor is required to send.

In our mesh partitioning/storage scheme, each vertex is owned by one processor; however, copies of vertices may exist on other processors to reduce communication. Similarly, each element is owned by one processor with copies on other processors. By storing copies, we significantly reduce or eliminate the cost of communication for some operations. We give an example of a mesh partitioning in Figure 2.3. We note that maintaining the integrity of the copies is the responsibility of algorithms that modify the mesh, for instance, the mesh refinement algorithm.

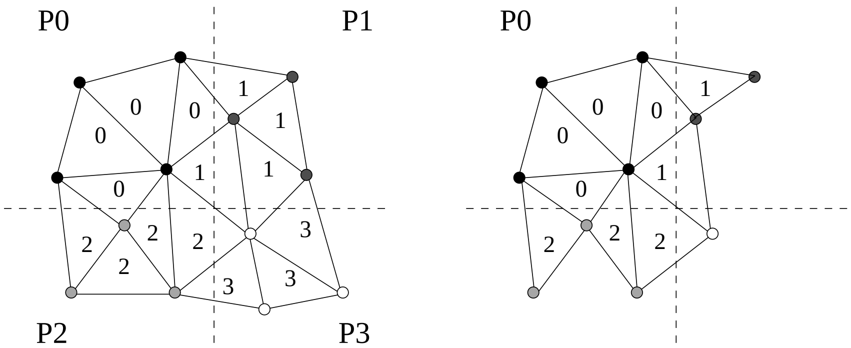

FIG. 2.3. *On the left, we give a geometric partitioning of a mesh among 4 processors. Vertex ownership is indicated by the fill pattern and triangle ownership by number. On the right, we show the vertices/elements that must be stored on* P0*, including copies of vertices/elements owned by other processors. Note that every triangle that contains a vertex owned by* P0 *is stored on* P0*.*

We have chosen to first partition the vertices using a geometric partitioning scheme described in [8]. This scheme is an extension of the orthogonal recursive bisection [1, 16] and has been shown experimentally to have desirable properties that scale with the problem size [11]. This partitioning algorithm is itself a parallel algorithm. The runtime of the implementation used in this paper increases as the number of processors increases linearly with the problem size.

2.2. Mesh Refinement. The mesh refinement algorithm is based on the bisection of triangles (or tetrahedra) [15]. A set of triangles (tetrahedra) is marked for refinement based on local error estimates. The parallel refinement algorithm selects independent subsets of triangles (tetrahedra) from this marked set that can be bisected simultaneously while still maintaining the integrity of the mesh data structures [11]. The computation for this algorithm includes evaluating the local error estimator on each element and, if necessary, bisecting that element. Given proper load balancing, the computation time is proportional to the number of elements assigned to a processor. Required communication consists of informing neighboring processors of bisection operations and keeping "copies" of vertices/elements up to date. The communication time is proportional to the number of elements on the boundary of a processor's partition.

2.3. Matrix Assembly. The global matrix assembly based on the unstructured mesh is done in two phases.[1] The first phase is the construction of the nonzero structure of the global matrix. The entire sparse column(s) associated with a vertex is stored on the processor owning that vertex. The column structure associated with a vertex is determined solely by the elements that contain that vertex. Because of the vertex and element copies maintained on each processor, no interprocessor communication is required during the construction of the nonzero structure (see Figure 2.3).

The second phase is the evaluation of the element submatrices and their incorporation into the global matrix. Like the first phase, this requires no interprocessor communication. Because all elements on a processor are evaluated, including copies, an element may be evaluated on more than one processor. An alternative approach would be to evaluate an element submatrix on only one processor and then to communicate portions of the submatrix to the required processors. Our approach trades redundant computation for savings in interprocessor communication. We note that the percentage of redundant computation is larger for higher-order elements than lower-order elements. Clearly, there exists some combination of computation-to-communication cost ratio and element type for which this tradeoff is not advantageous. The tradeoff is hardware and application dependent.

Note that the computation time for each of these algorithms is proportional to the number of elements assigned to a processor. The communication time in the alternative algorithm is proportional to the number of elements on the boundary of a processor's partition. The efficiency of the algorithm we have used is similar to the alternative algorithm; the number of redundant element evaluations is proportional to the number of elements on the boundary of a processor.

2.4. Matrix Solution. Parallel solution of sparse linear systems of equations has been widely studied [3, 5, 6, 9, 14]. The iterative method used in this paper is a parallel conjugate gradient method preconditioned by an incomplete factorization implemented in the BlockSolve95 software package [10]. The parallel incomplete factorization method is based on graph coloring and is described in [7, 9]. BlockSolve95 analyzes and takes advantage of the local structure of the mesh; it achieves high execution rates for problems with multiple unknowns per vertex and for higher order elements.

The parallel performance of an iterative method such as the conjugate gradient method preconditioned by incomplete factorization involves two aspects: execution efficiency and total solution time. The execution efficiency of a single iteration can typically be measured by the number of floating-point operations executed per processor per second. Two computations dominate each iteration: (a) sparse matrix-by-vector multiplication, and (b) sparse triangular matrix solution. Given a good partitioning, the execution efficiency of a single iteration remains constant as the problem size is scaled with the number of processors. The number of iterations typically scales with the square root of the problem size in two dimensions and the cube root of the prob-

[1] In this paper we use matrix solution methods that require the global stiffness matrix to be assembled. We note that methods do exist that do not require assembly of the global stiffness matrix.

lem size in three dimensions. Therefore, even with perfect efficiency on a per iteration basis, the linear system solution time will increase as the problem size increases.

3. Networks of Workstations. For the experimental results presented in this paper, we consider two NOWs, which we refer to as NOW1 and NOW2, as the basis for comparison. For each NOW, we compare two different network technologies.

NOW1 is a group of 24 Sun SPARCstation 5 model 70's, each with 96 megabytes of RAM. The workstations are connected by a 10 Mbps shared ethernet in addition to a 100 Mbps switched ethernet. The switched ethernet topology has two Bay (Synoptics) model 28115 10/100 ethernet switches. A schematic of the switch interconnect is shown in Figure 3.1.

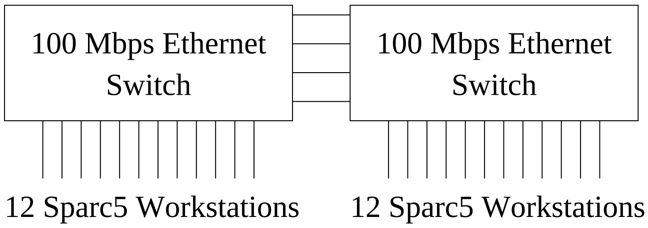

FIG. 3.1. *The 100 Mbps ethernet-based NOW1. The switches are connected by four 100 Mbps ethernet lines, and each of the workstations is connected to the switch by a 100 Mbps ethernet line.*

The second set of workstations, NOW2, is a group of 12 Sun Enterprise 2 Model 2170 computers, each with 256 Megabytes of RAM.[2] The computers are connected by a 10 Mbps shared ethernet in addition to a 155 Mbps ATM network. The ATM network uses a 16-port Bay (Synoptics) Model 10114A switch.

Further, as a reference point for the communication performance on a high-end parallel machine, we give message-passing timing results from an IBM SP computer. This particular system, located at Argonne National Laboratory, uses SP-1 compute nodes and a TB2 (IBM proprietary) switched, interconnection network.

To characterize the performance of these five configurations, we experimentally examined two quantities of interest: latency and bandwidth as a function of the number of processors and the message length. We wrote a short C program that sent and received messages between pairs of processors using MPI [13]. We timed 100 repetitions of that task for varying message sizes and for varying numbers of processor pairs operating simultaneously. A barrier function was used to synchronize the processors. While somewhat imprecise, this method gives a rough estimate of the performance to which an application will have access in each configuration.

In Figures 3.2 through 3.4 we display the message times for each of the five networks. Based on these timings, we computed a least squares fit to the average latency and average bandwidth for these networks. These values are reported in Table 3.1. As one might expect, the shared 10Mbps media performance degrades significantly as the number of processors increases; the bandwidth of the network is

[2]Although each computer has two processors, we ran only one process per computer because of software limitations.

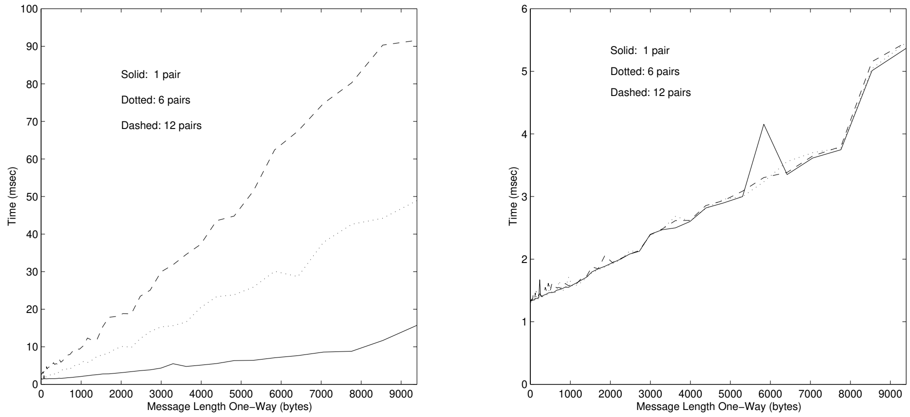

FIG. 3.2. *The average time (in milliseconds) to send and receive a message between two, twelve, and twenty-four processors on NOW1. The graph on the left shows the results for the 10Mbps ethernet, and the 100Mbps results are shown on the right.*

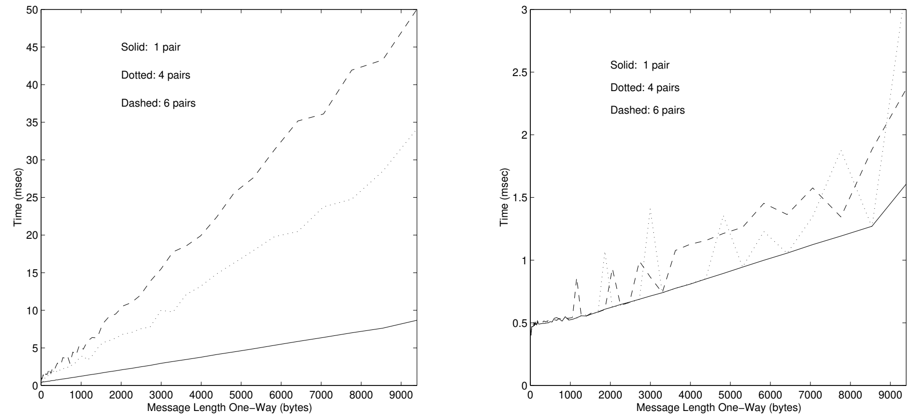

FIG. 3.3. *The time (in milliseconds) to send and receive a message between two, eight, and twelve processors for the 10Mbps ethernet and the 155Mbps ATM on NOW2.*

not proportional to the number of processors. However, the switched interconnection networks scale well with p, the number of processors. As we show in following section, the ability of the network bandwidth to scale with p is essential to support the scalable performance of many of the tasks required in an unstructured mesh framework.

The latency in the NOWs is largely a function of processor speed. This fact is an indication that the overhead incurred in sending a message is largely software based. For example, the SPARC Ultra processor in NOW2 is approximately three times as fast as the SPARC 5 processor in NOW1 for the unstructured mesh operations described in Section 2. The MPI message-passing software for the SP is more streamlined than for a NOW (it is built upon the underlying EUI communication layer). This fact is reflected in its lower latency, even though the SP-1 processor is generally slower than the newer SPARC Ultra processor. Hence, one expects better efficiency for smaller problem sizes on the SP than on the other interconnection networks.

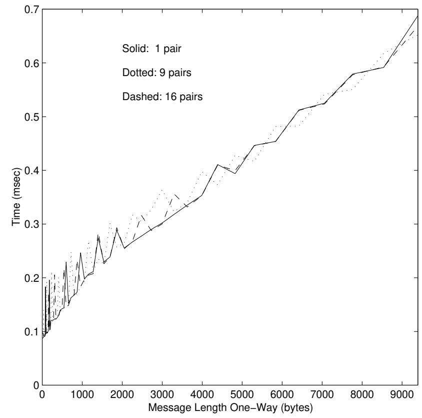

FIG. 3.4. *The time (in milliseconds) to send and receive a message between two, eighteen, and thirty-two processors on the IBM SP.*

TABLE 3.1
Statistics for latency and bandwidth for the 5 networks.

Network	Network Type	Number of Processors	Average Latency (msec)	Average Bandwidth (Mbps)
NOW1	10Mbs	2	1.12	6.95
NOW1	10Mbs	12	0.90	9.70
NOW1	10Mbs	24	1.46	9.76
NOW1	100Mbs	2	1.27	21.23
NOW1	100Mbs	12	1.26	127.49
NOW1	100Mbs	24	1.30	258.77
NOW2	10Mbs	2	0.40	9.39
NOW2	10Mbs	12	0.47	9.32
NOW2	ATM	2	0.45	81.34
NOW2	ATM	12	0.44	296.97
SP	TB2	2	0.12	132.14
SP	TB2	12	0.12	800.34
SP	TB2	24	0.12	1597.71
SP	TB2	32	0.12	2114.73

4. Matching Algorithms to Parallel Architectures. As discussed in §2, the tasks required to implement applications based on unstructured mesh discretizations are diverse. Parallel algorithms have been developed for these tasks, and their scalable performance has been demonstrated on high-end machines such as the Intel DELTA [8]. The important question is the determination of the essential network characteristics required to support the scalable performance of these algorithms on more modest NOWs.

In this section we observe that although these algorithms appear quite different, their underlying communications requirements are often similar. Based on this observation, we define an abstract problem class that encapsulates the computation and communication required in a representative problem arising in typical finite-element applications: the sparse matrix-vector product. Based on an analysis of the communication and computation requirements of this problem class, we can describe the network characteristics required to support the scalable performance of algorithms for this problem class.

The unstructured mesh algorithms considered in this paper are all data parallel algorithms based on a partitioning of the mesh with n vertices and its associated work onto p processors. We can assume that if a good partitioning algorithm is used on a finite-element mesh, each partition will have

1. $\frac{n}{p}$ vertices and a proportional number of elements;
2. C neighboring partitions, where C is independent of n and p; and
3. $O((\frac{n}{p})^{(D-1)/D})$ vertices/elements on the boundary of the partition, where D is the number of dimensions.

These assumptions appear to be empirically valid for a wide range of computational domains and different partitioning schemes.

4.1. SMVP equivalent algorithms. We define a class of algorithms called *SMVP-equivalent* (i.e., sparse matrix-vector product equivalent) if they have computation and communication of the same relative order as those required in sparse matrix-by-vector multiplication. Such algorithms have the following properties:

1. if the computation time required on a single processor is W, the parallel computation time is $O(W/p)$;
2. each processor will send $O(C)$ messages, C independent of n and p; and
3. there exists a nonzero, monotonic function $f(W/p)$ such that the total data sent by each processor is at least $f(W/p)$, and $f(W/p)$ is $o(W/p)$.

Naturally, a sparse matrix-vector product is SMVP equivalent if the partitioning is good. Each processor communicates only with its nearest neighbors; no significant global communication occurs. The total work required is $O(n)$ for the matrix arising from a finite-element matrix obtained from a bounded-degree mesh with n vertices. Further, the computation is load balanced and proportional to $\frac{n}{p}$; the number of messages sent by a processor is a constant independent of n and p. The total length of the messages sent by one processor is $O((\frac{n}{p})^{(D-1)/D})$, which is monotonic and $o(n/p)$. Examples of other SMVP equivalent algorithms include explicit time-stepping methods, some iterative methods for solving linear systems, finite-element stiffness matrix assembly, and unstructured mesh enrichment and improvement.

We can now specifically consider the main tasks required in unstructured matrix calculations as presented in §2 and determine whether they are SMVP equivalent. The problems are as follows:

- **Mesh Partitioning** – The runtime of the implementation used in this paper is not scalable; the parallel algorithm suffers an overhead of $O(\log(p))$ relative to the sequential version of the algorithm and is, therefore, not SMVP equivalent.

- **Mesh Refinement** – Given proper load balancing, the computation time is proportional to the number of elements assigned to a processor. Required communication consists of informing neighboring processors of bisection operations and keeping "copies" of vertices/elements up to date. The communication time is proportional to the number of elements on the boundary of a processor's partition. A coloring of the element dual graph can be maintained to ensure that the number of messages sent is bounded by a constant independent of the number of processors [11]. Thus, this problem is SMVP equivalent.
- **Matrix Assembly** – The matrix assembly algorithm that does not use redundant element evaluation is SMVP equivalent as only the boundary nonzeros need to be communicated in the assembly process. Because it involves no communication and has a computational overhead resulting from redundant element evaluation, the algorithm that we have used is not SMVP equivalent.
- **Matrix Solution** – As discussed in §2, two tasks are required by the preconditioned conjugate gradient iteration used for the results in this paper. The two tasks are sparse matrix-by-vector multiplication and sparse triangular matrix solution. The first task is SMVP equivalent by definition. The second is SMVP equivalent when a graph coloring is used to reorder the unknowns for parallelism [9].

4.2. SMVP supporting architectures. Often, the performance of a parallel algorithm is measured by scaling the problem size linearly with the number of processors. For SMVP equivalent algorithms with good partitionings, this implies that the computation time is independent of p, the total number of messages sent by all processors is $O(p)$, and, because $O(W/p)$ is fixed, the total length of messages sent by all processors is $O(p)$. For a parallel architecture to be *SMVP supporting*, it must, at a minimum, be capable of supporting without degradation message traffic in which

- the total number of messages sent grows linearly with p,
- the total length of the messages sent grows linearly with p, and
- the number of messages sent and received by a single processor does *not* exceed a constant value independent of p.

Examples of interconnection networks that fit into this class given a suitable mapping of mesh partitions to processors include crossbars, rings, and mesh-connected arrays. We note that from a pragmatic standpoint, any interconnection network is SMVP supporting if the bisection bandwidth is $O(p)$ and the message-passing latency grows modestly with p, for example $O(\log(p))$. A fixed bandwidth, bus-based network would not be SMVP supporting.

4.3. Bandwidth and Latency Dominated Implementations. Suppose the amount of data, in bytes, communicated by each processor by an SMVP equivalent algorithm is bounded by the function $f(W/p)$. The efficiency, $\mathcal{E}$, of an SMVP equivalent algorithm on an SMVP supporting architecture can be bounded below by the ratio of optimal computation time to a bound on the total execution time as

$$\mathcal{E} \geq \frac{\frac{W}{p}}{\alpha C + \beta f(\frac{W}{p}) + \frac{W}{p}}, \tag{4.1}$$

where α is the message startup cost, and β is the incremental cost to send a byte of data.

Based on this expression, we offer the following three informal propositions.

PROPOSITION 4.1. *The efficiency of a SMVP equivalent algorithm on a SMVP supporting parallel architecture is constant with respect to p for constant W/p.*

PROPOSITION 4.2. *The effect of message startup cost on efficiency can be hidden for sufficiently large W/p.*

PROPOSITION 4.3. *The efficiency is limited by the bandwidth of the interconnection network. Further, this effect is more pronounced in three dimensions than it is in two dimensions.*

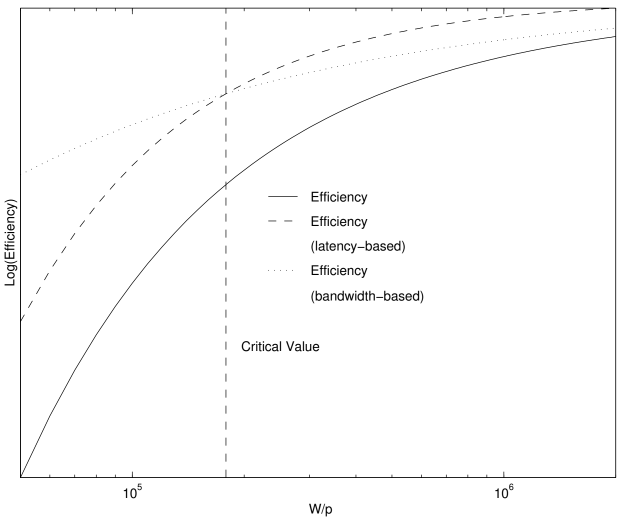

FIG. 4.1. *The parallel efficiency as a function of the local problem size, W/p. The critical value, $(W/p)^*$, is the value of W/p at which the bandwidth, β, rather than the latency, α, begins to dominate the efficiency. The bandwidth-based function shows the efficiency without the latency cost. The latency-based function shows the efficiency without the bandwidth cost.*

Proposition 4.1 is true because, by definition, $f(W/p)$ is $o(W/p)$. The final two propositions follow from the fact that $f(W/p)$ is monotonically increasing. Thus, for any α and β and increasing W/p, the efficiency eventually will be dominated by the incremental message cost associated with β. We say that the problem is *bandwidth dominated* for $W/p > (W/p)^*$, where

$$\alpha C = \beta f((W/p)^*). \tag{4.2}$$

For values of W/p less than this critical value, we say that the problem is *latency dominated*.

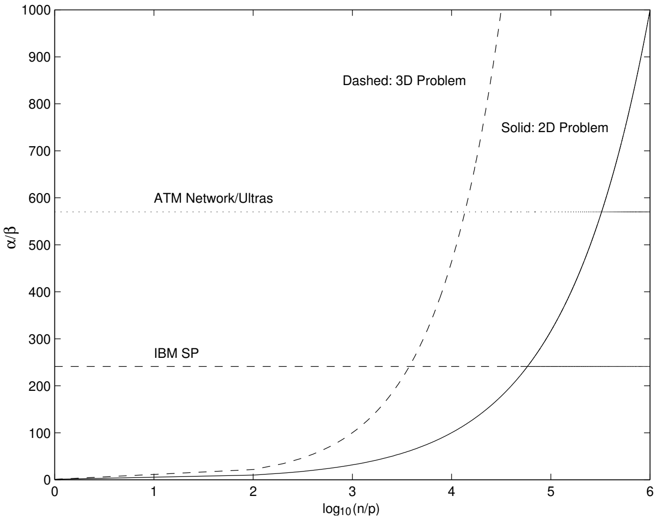

FIG. 4.2. *Approximate values of n/p when the efficiency the specific algorithm, sparse matrix-vector product, goes from being latency dominated to being bandwidth dominated as a function of the network parameters α/β. Two horizontal lines show the approximate values for α/β for the IBM SP system and the NOW2 system as discussed in the text. The dashed curve shows the results for a canonical three-dimensional problem, and the solid line shows the results in two dimensions. Values of n/p to the right of the curves are bandwidth dominated; to the left of the curves, the problems are latency dominated.*

We illustrate this property in Figure 4.1, where we schematically plot the parallel efficiency as a function of W/p. Two regions, latency dominated and bandwidth dominated, are divided by a critical value at which the efficiency of the algorithm is primarily determined by the bandwidth of the architecture as opposed to the latency.

In Figure 4.2 we illustrate that SMVP equivalent algorithms on NOWs can easily be bandwidth dominated as opposed to latency dominated. If we consider an algorithm for the specific sparse matrix vector product problem, we get the approximation

$$(\frac{n}{p})^* \approx (\frac{\alpha}{\beta})^{\frac{D}{D-1}}. \tag{4.3}$$

Assuming that we are sending eight-byte, double-precision values, we get the curves for problems from two and three dimensions shown in Figure 4.2. Using the data from Table 3.1, we get the approximate α/β values shown as horizontal lines in the figure. Thus, for modest numbers of unknowns per processor relative to the amount of memory available on most NOWs, one can expect to be bandwidth dominated. For example, for a three-dimensional problem with more than approximately 5,000 unknowns per processor on the NOW2/ATM system (the configuration with the largest ratio α/β) the problem will be bandwidth dominated. Such a problem size is not large relative to the amount of memory available to the processor.

5. Computational Results. We present experimental results to demonstrate the effect of several variables on the performance of the unstructured mesh computations described in Section 2. The variables are *problem type*, *problem size*, *network type*, and *number of processors*. Each of these variables affects the ratio of computation to communication, which in turn determines the efficiency of the parallel algorithms.

We will look at four basic problems in this study: linear elasticity in two dimensions (LE2), linear elasticity in three dimensions (LE3), Poisson's equation in two dimensions (PE2), and Poisson's equation in three dimensions (PE3). For each of these problem types, the basic finite element is a triangle (tetrahedron) with either linear or quadratic basis functions (denoted by adding either an "L" or a "Q" to the problem type). In each problem we begin with a coarse initial grid on a relatively simple geometry and, using the basic algorithm in Figure 2.1, selectively refine the mesh until the local error estimate is satisfactory on every element. By manipulating the error tolerance, the size of the final mesh can be changed. For a more detailed description of the problems and the solution approach see [11]. A brief description of each of the problems is given in Table 5.1.

On each configuration we used an identical set of application programs. Each program is written primarily in C and uses the Message Passing Interface (MPI) standard for interprocessor communication [13].

TABLE 5.1
Information on the smallest instance of each of the problem types.

Name	Problem Type	Dim.	Element Type	No. of Vertices	No. of Elements	No. of Unknowns
LE2L	Linear Elasticity	2	linear	3794	7346	7588
LE2Q	Linear Elasticity	2	quadratic	14804	7288	29608
LE3L	Linear Elasticity	3	linear	4360	22106	13080
LE3Q	Linear Elasticity	3	quadratic	3925	2541	11775
PE2L	Poisson's Eq.	2	linear	2690	5245	2690
PE2Q	Poisson's Eq.	2	quadratic	5267	2582	5267
PE3L	Poisson's Eq.	3	linear	1845	9759	1845
PE3Q	Poisson's Eq.	3	quadratic	13057	9337	13057

In the rest of this section we describe the effects of these variables on the computational efficiency of the algorithms, including the total problem solution time in the next subsection.

5.1. Total Execution Time. In this section we look at the total solution time for the LE2L problem and focus on the problem size and network type variables. Figures 5.1 and 5.2 illustrate the effect of higher network bandwidth that scales with p on total solution time. It is evident that the restricted bandwidth of the 10 Mbps networks is not sufficient to allow scalable execution of SMVP-equivalent algorithms.

We note that the performance for all of the configurations is closer to the ideal when the problem size is larger. This is due to the better ratio of computation to communication for the larger problem sizes. To illustrate this, we plot the percentage of cross edges as well as the maximum number of neighbors of a processor in Figure 5.3. The percentage is much smaller for the large problems and for small numbers of processors. Note that a property of the partitioning algorithm is that the number of neighbors for a partition is not a function of the problem size; it is a characteristic of the problem. Typically, the number of neighbors rises until approximately 16 processors, after which it remains constant [11].

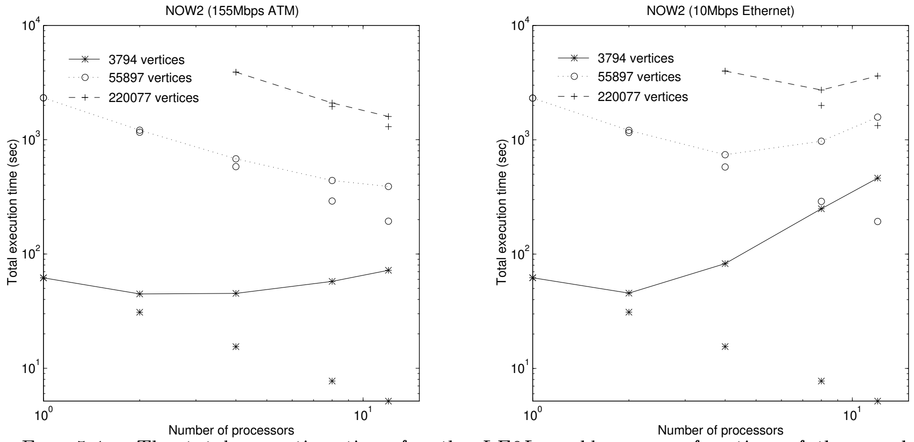

FIG. 5.1. *The total execution time for the LE2L problem as a function of the number of processors for three problem sizes; the left graph is for NOW2 with ATM, and the right graph is for NOW2 with 10Mbps ethernet. The actual results are given by the lines; points off the lines reflect the ideal results with no communication cost and perfect load balance.*

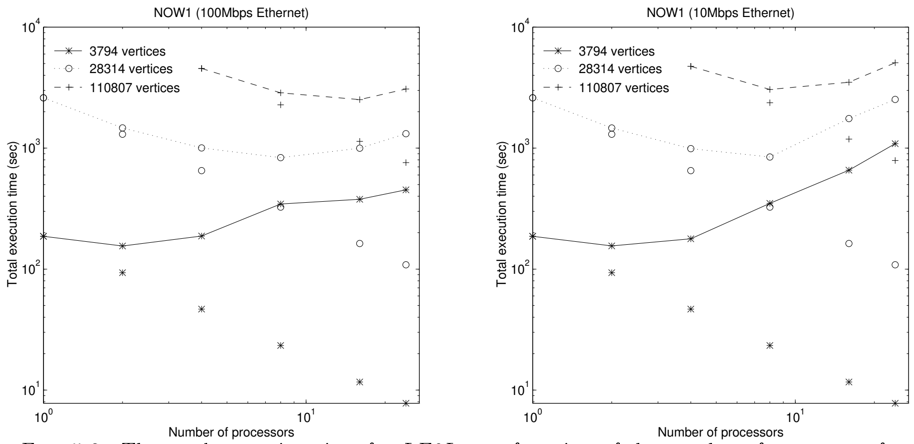

FIG. 5.2. *The total execution time for LE2L as a function of the number of processors for three problem sizes; the left graph is for NOW1 with 100 Mbps ethernet, and the right graph is for NOW1 with 10Mbps ethernet. The actual results are given by the lines; points off the lines reflect the ideal results with no communication cost and perfect load balance.*

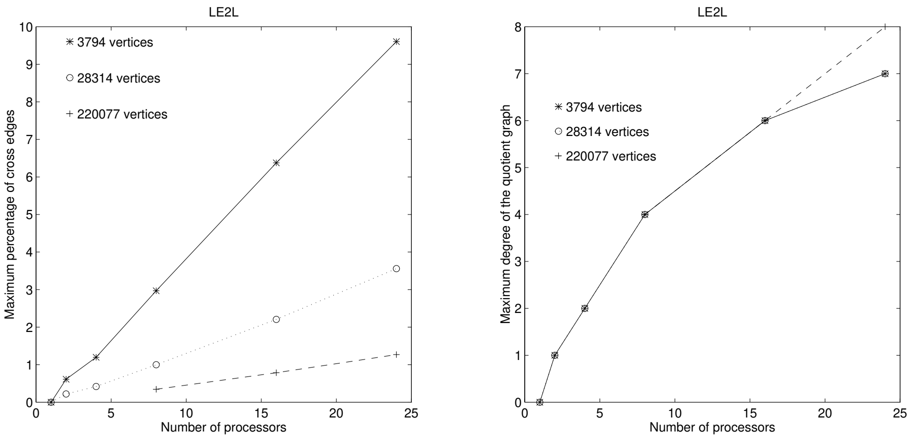

FIG. 5.3. *For the LE2L problems, the maximum percentage of cross edges and the maximum degree of the quotient graph as a function of the number of processors for different problem sizes.*

5.2. Partitioning. In Section 2 we noted that the performance of the parallel implementation of the partitioning algorithm was not expected to scale well with the number of processors. In effect, the implementation does twice as much work to partition a problem of size $2X$ into $2p$ pieces as it does to partition a problem of size X into p pieces. In Figure 5.4, as expected, we see that the number of vertices partitioned by the p processors per second is essentially constant for the ATM network after 4 processors. However, we note that the penalty for the slower 10 Mbps network is not nearly as severe as it was for the total solution times. Further, we note that in Figure 5.5, the partitioning rate in three dimensions is lower than that for two dimensions; however, this difference is not as large as for the refinement computation because the runtime of this partitioning algorithm is dependent not on the percentage of cross edges but rather on the number of vertices, processors, and dimensions.

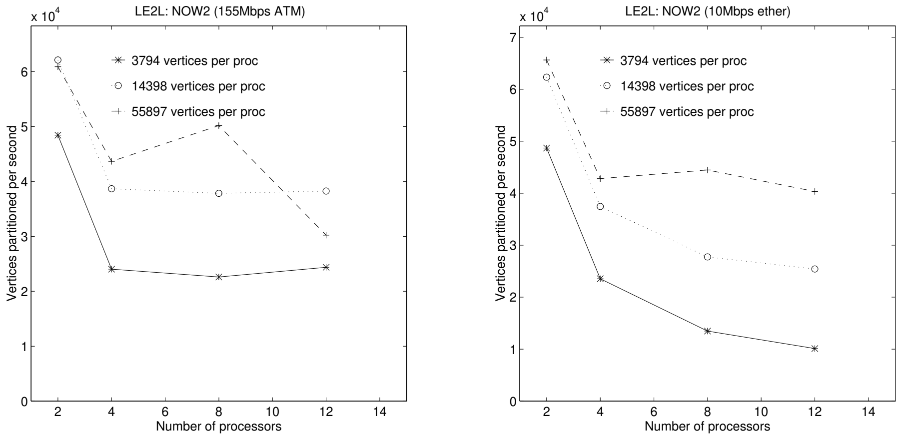

FIG. 5.4. *A comparison of the partitioning rate as a function of the number of processors for different sizes of the LE2L problems; the left graph is for a NOW2 with 155 Mbps ATM, and the right graph is NOW2 with 10Mbps ethernet.*

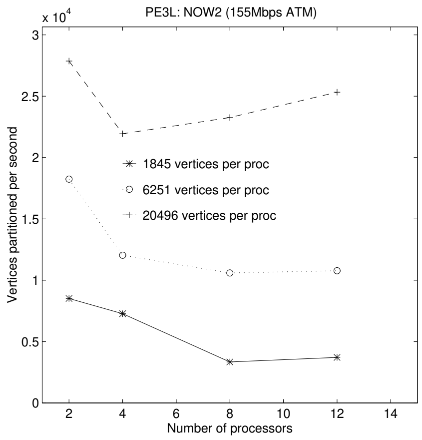

FIG. 5.5. *The partitioning rate as a function of the number of processors for the PE3L problem on NOW2 with 155 Mbps ATM.*

5.3. Refinement. We now examine the parallel efficiency of the refinement computation as a function of problem size and of the problem type, specifically the dimension of the problem. We measure the parallel efficiency of refinement as the maximum number of vertices added per second per processor [11]. This measure should be nearly constant as the problem size and/or the number of processors increases *if* the cross-edge percentage (and the maximum number of partition neighbors) remains constant. As noted before, of course, the cross-edge percentage increases as the number of processors increases if the problem size is fixed. However, the cross-edge percentage will stabilize after a few processors if the problem size increases linearly with the number of processors.

In Figure 5.6 we compare the refinement rate for a fixed problem size and a problem size that scales with the number of processors. Note that the refinement rate stabilizes after four processors for the scaled problem size. In Figures 5.7 and 5.8 the refinement rate is much higher for the two-dimensional case. This is caused by two factors: (a) bisection in three dimensions is more complex, and (b) the cross-edge percentage in three dimensions is much higher.[3] In Figure 5.9 we compare the percentage of cross edges for the two-dimensional and three-dimensional problems; as expected, the percentage of cross edges grows much more rapidly for the three-dimensional problem. As noted in Section 4, a three-dimensional problem must be much larger than a two-dimensional problem to achieve the same cross-edge percentage.

5.4. Matrix Assembly. Unlike the other tasks, the matrix assembly is independent of the network type because no communication is involved. However, as noted in Section 2, this lack of communication is traded off against the redundant evaluation of some element submatrices. In this subsection we will measure the num-

[3]For all four problem types, the problem size is increased successively by approximately a factor of two when scaled. As a consequence, for the 12- and 24-processor cases, the number of vertices per processor will be 1.5 times larger than for the 8- and 16-processor cases, respectively, for figures with scaled problem sizes.

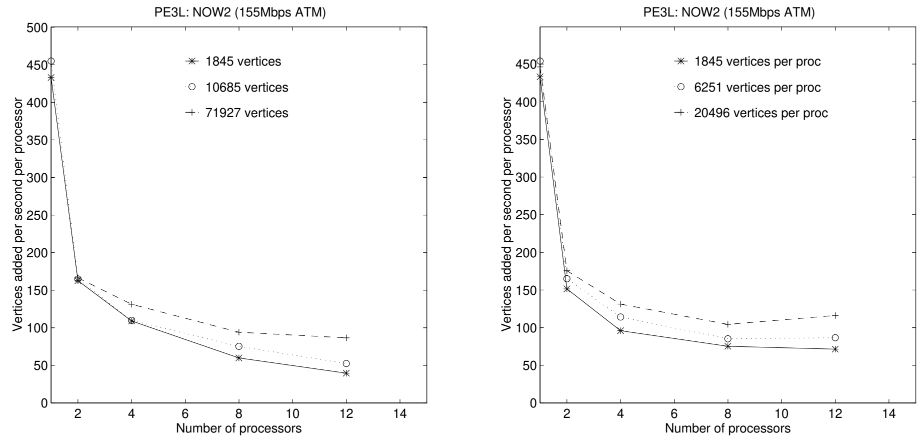

FIG. 5.6. *The maximum rate of refinement per processor as a function of the number of processors for different problem sizes; the left graph is for a fixed problem size, and the right graph is for a problem size that scales with the number of processors. These results are for NOW2 with 155 Mbps ATM on the PE3L problems.*

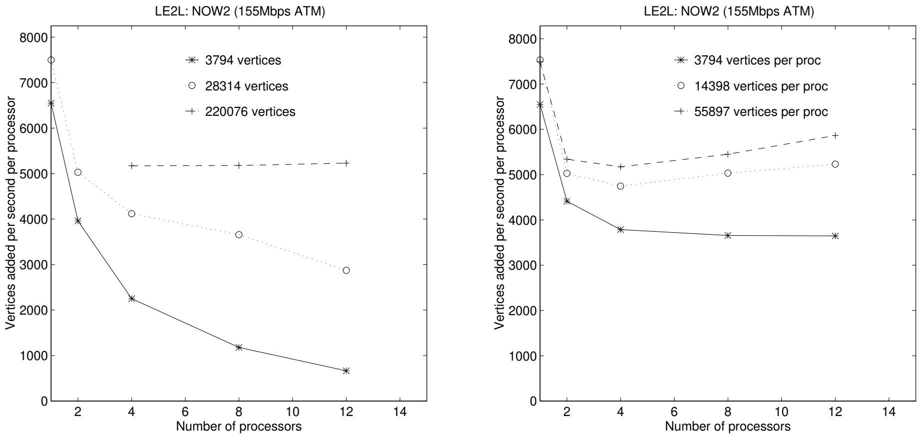

FIG. 5.7. *The maximum rate of refinement per processor as a function of the number of processors for different problem sizes; the left graph is for a fixed problem size, and the right graph is for a problem size that scales with the number of processors. These results are for NOW2 with 155 Mbps ATM on the LE2L problems.*

ber of *nonredundant* submatrix evaluations per second per processor. In Figures 5.10 and 5.11 we see that the assembly rate *per processor* remains essentially constant as the problem size is scaled. The slight decrease in assembly rate as the number of processors increases for the smaller problems is due to the larger proportion of redundant evaluations for these problems. Note the expected slower rate for the same problems with quadratic elements in Figure 5.11.

The elements for the PE2L problem are much simpler to evaluate; this fact is reflected in the assembly rates in Figure 5.12.

5.5. Matrix Solution. The most time-consuming computation for these problems is the solution of the sparse linear systems. These computations are generally

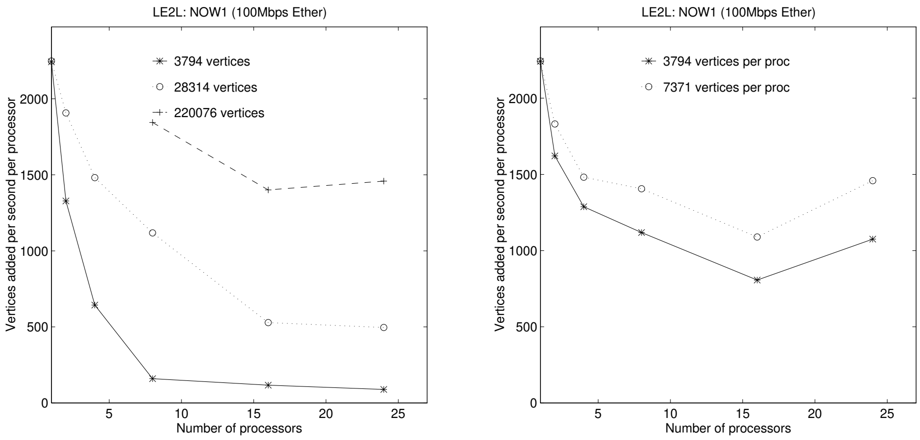

FIG. 5.8. *The maximum rate of refinement per processor as a function of the number of processors for different problem sizes; the left graph is for a fixed problem size and the right graph is for a problem size that scales with the number of processors. These results are for NOW1 with 100 Mbps ethernet on the LE2L problems.*

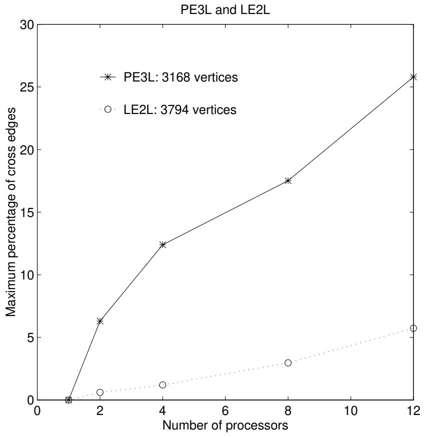

FIG. 5.9. *A comparison of the maximum percentage of cross edges as a function of the number of processors for the LE2L and PE3L problems. Note that the three-dimensional problem has a far greater percentage of cross edges.*

more tightly coupled than the other computations and hence place greater demands on the network.

The efficiency of the iterative method used in these computations increases with the order of the element and the number of unknowns per vertex. Therefore, we expect that the linear elasticity problem with quadratic elements will have a higher efficiency than the Poisson problem with linear elements. In Figure 5.13 we note that for a fixed problem size, the solution time decreases substantially as the number of processors increases on the ATM network; however, performance is poor on the 10 Mbps network. We see a similar situation for the quadratic elements in Figure 5.14.

As a measure of parallel efficiency, we can look at the number of floating-point operations per second executed by the iterative method. In Section 2 we stated that

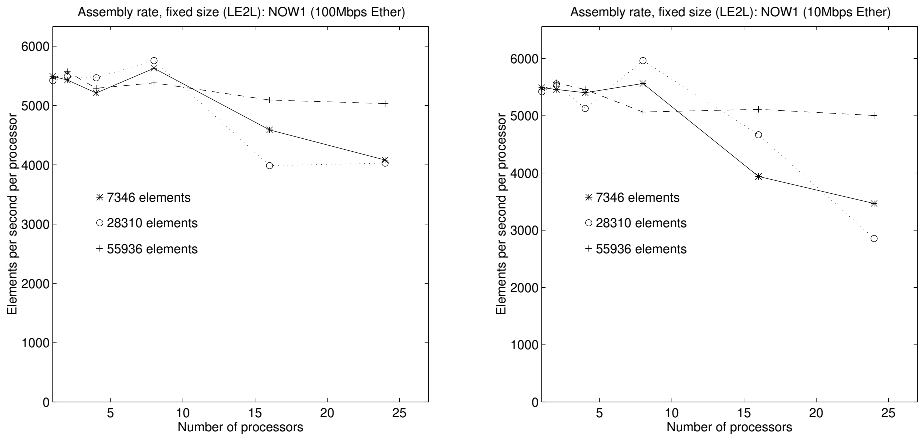

FIG. 5.10. *A comparison of the matrix assembly rate as a function of the number of processors for different sizes of the LE2L problems; the left graph is for NOW1 with 100 Mbps ethernet, and the right graph is for NOW1 with 10 Mbps ethernet.*

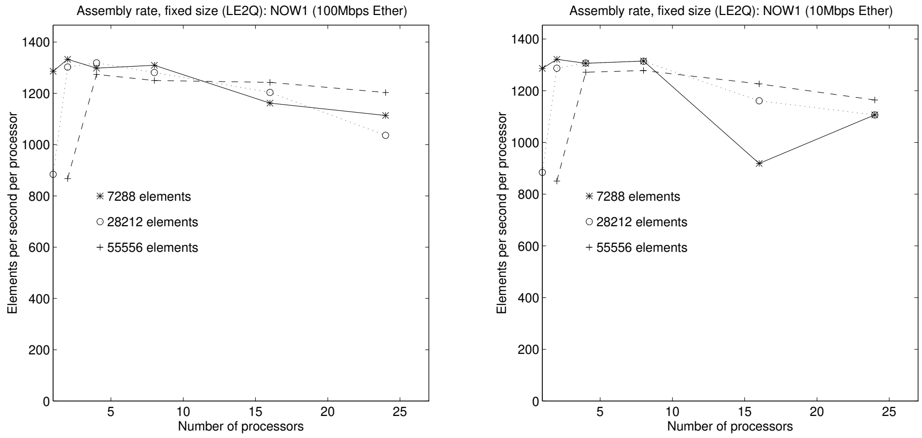

FIG. 5.11. *A comparison of the matrix assembly rate as a function of the number of processors for different sizes of the LE2Q problems; the left graph is for NOW1 with 100 Mbps ethernet, and the right graph is for NOW1 with 10 Mbps ethernet.*

the efficiency of the method should remain nearly constant if the problem size is scaled linearly with p; note that we expect some degradation in these problems because the cross-edge percentage typically increases until $p = 16$. We give the rates for scaled versions of the LE2 problem on NOW2 for both the linear and quadratic problems in Figure 5.15. In Figure 5.16, we give the rates on the LE3 problem on NOW2; these rates are largely governed by the percentage of cross edges given in Figure 5.17.[4] Note that these rates do not include the time either to reorder the matrix or to factor it; however, the sum of these times is typically less than 5% of the solution time.

[4]The rate for the scaled LE3L on NOW2 for 12 processors is adversely affected by the need to use virtual memory for this problem.

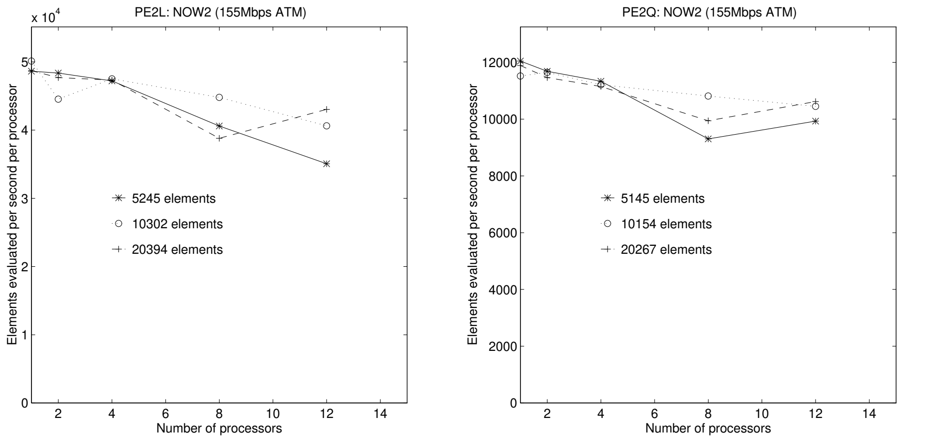

FIG. 5.12. *A comparison of the matrix assembly rate as a function of the number of processors for different sizes of the PE2 problem on NOW2 with 155 Mbps ATM; the left graph is for linear elements, and the right graph is for quadratic elements.*

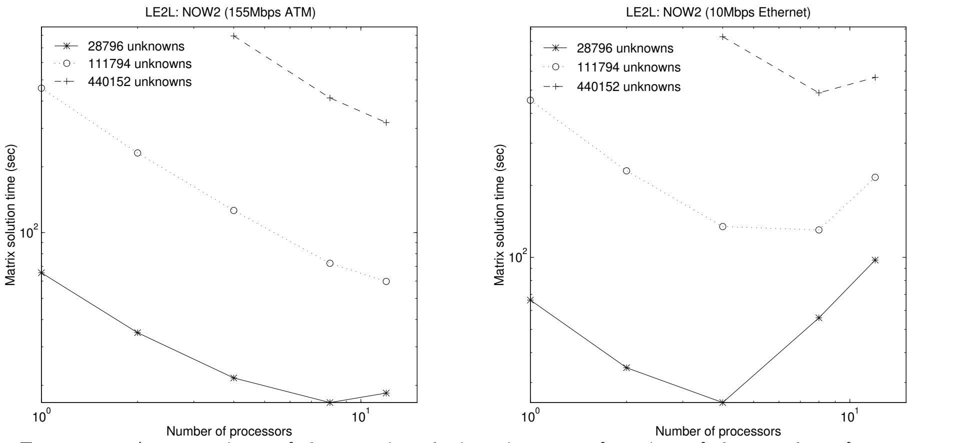

FIG. 5.13. *A comparison of the matrix solution time as a function of the number of processors for different sizes of the LE2L problems; the left graph is for NOW2 with 155 Mbps ATM, and the right graph is for NOW2 with 10 Mbps ethernet.*

6. Concluding Remarks. We have shown that parallel unstructured mesh computations can perform effectively on a network of workstations using off-the-shelf components. We characterized the performance of these computations and described the type of interconnection network necessary for their scalable performance. The primary determinants of efficient performance were found to be the problem size per processor and type of interconnection network used. Specifically, we found that efficient execution was dependent on

- individual workstations having substantial available RAM to allow large problem sizes, and
- an interconnection network whose capacity scaled with p, for example, a switched 100 Mbps ethernet network and a 155 Mbps ATM network.

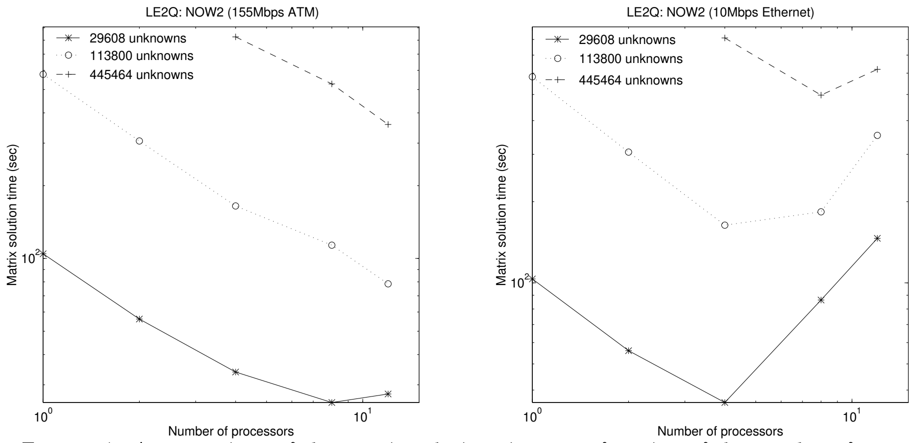

FIG. 5.14. *A comparison of the matrix solution time as a function of the number of processors for different sizes of the LE2Q problems; the left graph is for NOW2 with 155 Mbps ATM, and the right graph is for NOW2 with 10 Mbps ethernet.*

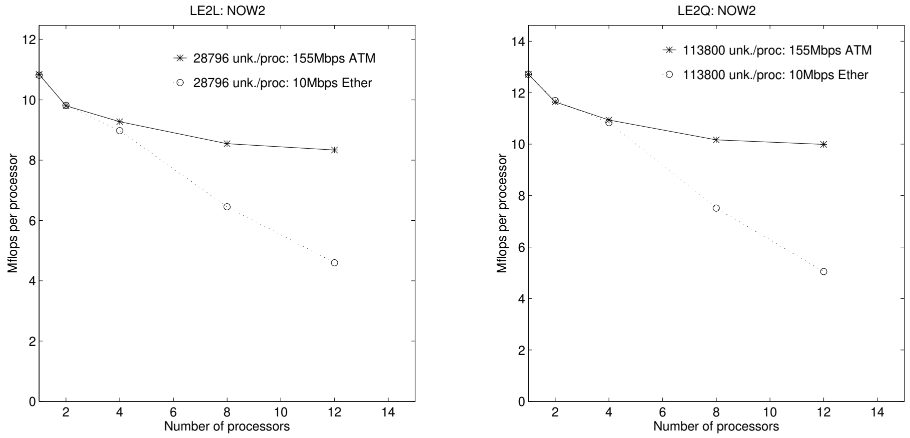

FIG. 5.15. *A comparison of the matrix computation rates as a function of the number of processors for a scaled version of the LE2L problem on the left and the LE2Q problem on the right.*

The majority of the algorithms and software implementations in the paper behaved in a scalable and effective fashion on the NOWs examined. The algorithms were tolerant of the high latencies found in NOWs when the local problem size was large enough. However, the algorithms required high total bandwidth and, therefore, were intolerant of the restricted total bandwidth found in the 10Mbps ethernet. Of the parallel algorithms employed, only the performance of the partitioning algorithm was found to be wanting. The authors are investigating a novel implementation of this algorithm to increase its scalability.

New technologies for NOWs, such as active messages [12] and the SHRIMP project [2], offer the promise of increasing the effective bandwidth available to applications. Advances in operating systems for NOWs such as Condor [4] will improve the performance of applications during times when some workstations on the network are being used intensively.

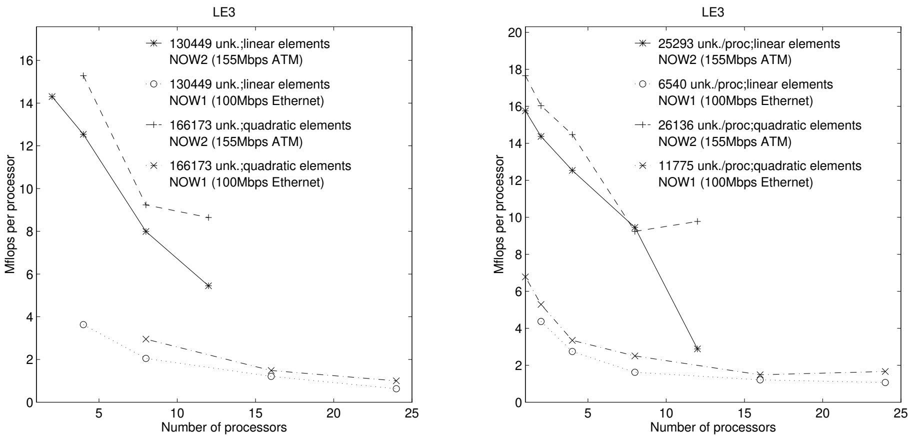

FIG. 5.16. *A comparison of the matrix computation rates as a function of the number of processors for fixed versions of the LE3 problem on the left and a scaled versions of the LE3 problem on the right.*

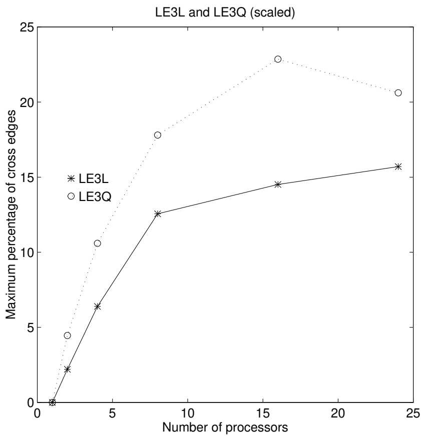

FIG. 5.17. *A comparison of the maximum percentage of cross edges as a function of the number of processors for scaled versions of the LE3L and LE3Q problems.*

REFERENCES

[1] M. J. BERGER AND S. H. BOKHARI, *A partitioning strategy for nonuniform problems on multiprocessors*, IEEE Transactions on Computers, C-36 (1987), pp. 570–580.

[2] M. A. BLUMRICH, K. LI, R. ALPERT, C. DUBNICKI, E. W. FELTEN, AND J. SANDBERG, *Virtual memory mapped network interface for the shrimp multicomputer*, in Proceedings of the 21st Annual International Symposium on Computer Architecture, April 1994, pp. 142–153.

[3] I. S. DUFF, *Parallel implementation of multifrontal schemes*, Parallel Computing, 3 (1986), pp. 193–204.

[4] D. EPEMA, M. LIVNY, R. VAN DANTZIG, X. EVERS, AND J. PRUYNE, *A worldwide flock of condors: Load sharing among workstation clusters*, Journal on Future Generations of Computers Systems, 12 (1996).

[5] A. GEORGE, M. T. HEATH, AND J. LIU, *Parallel Cholesky factorization on a shared-memory multiprocessor*, Linear Algebra and its Applications, 77 (1986), pp. 165–187.

[6] M. HEATH, E. NG, AND B. PEYTON, *Parallel algorithms for sparse linear systems*, SIAM Review, 33 (1991), pp. 420–460.

[7] M. T. Jones and P. E. Plassmann, *A parallel graph coloring heuristic*, SIAM Journal on Scientific Computing, 14 (1993), pp. 654–669.

[8] ———, *Computational results for parallel unstructured mesh computations*, Computing Systems in Engineering, 5 (1994), pp. 297–309.

[9] ———, *Scalable iterative solution of sparse linear systems*, Parallel Computing, 20 (1994), pp. 753–773.

[10] ———, *BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems*, ANL Report ANL-95/48, Argonne National Laboratory, Argonne, Ill., December 1995.

[11] ———, *Parallel algorithms for adaptive mesh refinement*, SIAM Journal on Scientific Computing, 18 (1997), pp. 686–708.

[12] L. T. Liu and D. E. Culler, *Evaluation of the Intel Paragon on active message communication*, in Proceedings of the Intel Supercomputer Users Group Conference, June 1995.

[13] Message Passing Interface Forum, *MPI: A message-passing interface standard*, International Journal of Supercomputing Applications, 8 (1994).

[14] D. P. O'Leary and R. White, *Multi-splittings of matrices and parallel solution of linear systems*, SIAM Journal of Algebraic Discrete Methods, 6 (1985), pp. 630–640.

[15] M.-C. Rivara, *Mesh refinement processes based on the generalized bisection of simplices*, SIAM Journal of Numerical Analysis, 21 (1984), pp. 604–613.

[16] R. D. Williams, *Performance of dynamic load balancing algorithms for unstructured mesh calculations*, Concurrency: Practice and Experience, 3 (1991), pp. 457–481.