Spark98: Sparse Matrix Kernels for Shared Memory and Message Passing Systems

David R. O'Hallaron

October 8, 1997

CMU-CS-97-178

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Spark98 is a collection of sparse matrix kernels for shared memory and message passing systems. Our aim is to provide system builders with a set of example sparse matrix codes that are simple, realistic, and portable. Each kernel performs a sequence of sparse matrix vector product operations using matrices that are derived from a family of three-dimensional finite element earthquake applications. We describe the computational structure of the kernels, summarize their performance on a parallel system, and discuss some of the insights that such kernels can provide. In particular we notice that efficient parallel programming of sparse codes requires careful partitioning of data references, regardless of the underlying memory system. So on one hand, efficient shared memory programs can be just as difficult to write as efficient message passing programs. On the other hand, shared memory programs are not necessarily less efficient than message passing programs.

Effort sponsored in part by the Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287, in part by the National Science Foundation under Grant CMS-9318163, and in part by a grant from the Intel Corporation. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government. Author's email address: droh@cs.cmu.edu

Keywords: benchmark, finite element, parallel computing, sparse matrix

Contents

About this Report

The Spark98 kernels (including an electronic copy of this report) can be obtained from the Web page at www.cs.cmu.edu/~quake/spark98.html.

1 Introduction

The multiplication of a sparse matrix by a dense vector is central to many computer applications, including scheduling applications based on linear programming and applications that simulate physical systems. For example, the Quake project at Carnegie Mellon University uses a sequence of more than 16,000 sparse matrix-vector product (SMVP) operations to simulate the motion of the ground during the first 40 seconds of an aftershock of the 1994 Northridge earthquake in the San Fernando Valley [4, 3]. The sparse matrix consists of over 13 million rows and 180 million nonzero entries, where each nonzero entry is a dense 3×3 submatrix of double precision floating point numbers. Repeated executions of a single SMVP routine account for over 70% of the simulation's computation time.

A sparse computation like the SMVP is fundamentally different from a dense matrix computation because its memory reference pattern is irregular and depends on the nonzero structure of the sparse matrix, which in turn depends on the physics of the underlying system and the goals of the simulation. For example, the number and distribution of mesh nodes in an earthquake simulation is determined by both the density of the soil and the frequencies of interest to the user.

The data-dependent nature of sparse computations presents difficulties for system builders who need access to sparse codes in order to evaluate design options. A kernel must be simple enough that designers can understand it without having to wade through masses of obscure code. If the code is simple then it can be modified, and thus becomes a useful tool for experimentation. A kernel must also be realistic in that it represents a larger class of applications. Finally, a kernel must also be portable so that it can be run without modification on a large number of platforms.

Existing sparse matrix collections like the Harwell-Boeing suite [8] are important tools for designers of numerical algorithms and partitioning algorithms. And existing software libraries that support sparse matrix computations are useful tools for application programmers [2, 7, 11]. However, to date the designers of parallel systems have not had access to a set of public domain sparse matrix kernels that are simple, realistic, and portable. Towards that end, we introduce the Spark98 kernels, a set of 10 SMVP kernels for shared memory and message passing systems.[1] We expect that builders of cache coherent shared memory multiprocessors [15, 6], distributed memory systems [22], message passing libraries [18], and distributed shared memory libraries [1, 25] will find that the Spark98 kernels are useful tools for understanding the performance of irregular codes on their systems. In our own experience with the Spark98 kernels we notice that efficient parallel programming of sparse codes requires careful partitioning of data references, regardless of the underlying memory system. Because equal care must be given to the partitioning of references, we find that for sparse codes, efficient shared memory programs are just as difficult to write as efficient message passing programs. On the other hand, shared memory programs are not necessarily less efficient than message passing programs.

Each Spark98 kernel consists of a C program and a partitioned finite element mesh, which is described by an input file. There are five programs (SMV, LMV, RMV, MMV, and HMV) and two meshes (sf10 and sf5), for a total of ten kernels, which are summarized in Figure 1. In the figure, n is the number of nodes, nz is the number of nonzero entries in the corresponding sparse matrix (where each entry is a 3×3 submatrix of double precision floating point numbers), and p is the number of processing elements (PEs). (We use the term *processing element* instead of the more common term *node* to avoid confusion with the nodes of finite element meshes.) For reference, all of the symbols used in this paper are summarized in Figure 2.

[1] Spark is a loose acronym for "sparse kernels", but actually the name reflects our hope that the kernels will spark interest in sparse computations among system designers.

Kernels based on mesh sf10 ($n = 7,294$, $nz = 97,138$)		
Kernel name	Description	Number of PEs (p)
SMV/sf10	sequential	$p = 1$
LMV/sf10	parallel, shared memory using locks	$p \geq 1$
RMV/sf10	parallel, shared memory using reductions	$p \geq 1$
MMV/sf10	parallel, message passing	$p = 1, 2, 4, 8, \ldots, 128$
HMV/sf10	parallel, hybrid shared memory and message passing	$p = 1, 2, 4, 8, \ldots, 128$

Kernels based on mesh sf5 ($n = 30,169$, $nz = 410,923$)		
Kernel name	Description	Number of PEs (p)
SMV/sf5	sequential	$p = 1$
LMV/sf5	parallel, shared memory using locks	$p \geq 1$
RMV/sf5	parallel, shared memory using reductions	$p \geq 1$
MMV/sf5	parallel, message passing	$p = 1, 2, 4, 8, \ldots, 128$
HMV/sf5	parallel, hybrid shared memory and message passing	$p = 1, 2, 4, 8, \ldots, 128$

Figure 1: The Spark98 kernels.

Symbol	Description
p	number of subdomains or PEs
L	number of locks
n	nodes in mesh (and rows in matrix)
e	edges in mesh (including self-edges)
nz	nonzero 3 × 3 submatrices, assuming nonsymmetric storage.
n_i	nodes in mesh (and rows in matrix) on subdomain i
e_i	edges in subdomain i (including self-edges)
nz_i	nonzero 3 × 3 submatrices in the matrix on subdomain i, assuming nonsymmetric storage

Figure 2: Summary of symbols used in this paper.

Each Spark98 program consists of roughly $1,000$ lines of code, most of which serve to parse the input files; the actual SMVP code is quite small. Kernels such as LMV and RMV use trivial partitioning algorithms that are computed at runtime, and thus can run on an arbitrary number of PEs, as indicated in the rightmost column of Figure 1. Other kernels like MMV and HMV rely on a partition that is precomputed offline and stored in a file. For these kernels we provide precomputed partitions for $1, 2, 4, 8, \ldots, 64, 128$ PEs.

The Spark98 meshes are called sf10 and sf5. They were developed by the Quake project at Carnegie Mellon to model earthquake-induced ground motion in the San Fernando Valley of Southern California. Each mesh is an unstructured three-dimensional finite element model of the earth underneath the San Fernando Valley. Sf10 is a small-sized mesh ($n = 7,294$, $nz = 97,138$) and sf5 is moderate-sized ($n = 30,169$, $nz = 410,923$).

The Spark98 programs are written in ANSI C and consist of a baseline sequential program (SMV) and four parallel programs (LMV, RMV, MMV, and HMV). Each program computes a sequence of SMVP pairs, $w = Kv$ followed by $w' = K'v'$, where K and K' are symmetric $n \times n$ sparse matrices with identical nonzero structures, and v, v', w, and w' are $n \times 1$ dense vectors. Each vector entry is a 3 × 1 subvector of doubles. Each of the nz nonzero matrix entries is a 3×3 submatrix of doubles. In order to conserve precious memory, matrices are stored in a compressed sparse row (CSR) format where only the submatrices along the diagonal and in the upper triangle are actually stored in memory. As we will see in Section 3, exploiting symmetry in this way makes the kernels much harder to parallelize because each diagonal nonzero submatrix contributes to two subvectors of the output vector.

LMV and RMV are simple parallel shared memory programs based on locks and reductions, respectively. Each uses a trivial partitioning algorithm whereby each thread is assigned a contiguous set of matrix rows with roughly the same number of nonzeros. For LMV, each thread updates a shared output vector w, which is protected by locks. For RMV, each thread updates its own private copy of the output vector. These private vectors are then summed to produce the final result vector w. Because of the simple partitioning algorithm, both LMV and RMV can run on an arbitrary number of processing elements (PEs).

MMV is a parallel message passing program that uses a partition precomputed by a sophisticated geometric recursive bisection mesh partitioning algorithm [17]. HMV is a hybrid shared memory program that employs the same mesh partition as MMV, but uses shared memory copy operations rather than sends and receives to transfer data. Precomputed partitions are supplied for these kernels for $1, 2, 4, \ldots, 128$ PEs. The shared memory programs can be compiled to use either the Posix or SGI thread interfaces. The message passing program is based on the industry standard MPI interface [18].

In sum, Spark98 is a set of sparse matrix kernels that are designed for system builders who want to evaluate the performance of sparse codes on their systems. The kernels are small and easy to understand, realistic in that the underlying meshes were developed by an application group, and portable in that they are written in C using standard shared memory and message passing interfaces.

The remainder of the paper is organized as follows. Sections 2 and 3 describe the Spark98 meshes and programs in more detail. Section 4 summarizes the performance of the kernels on some parallel systems and discusses some of the insights from using the kernels.

2 The Spark98 meshes

The Spark98 kernels are based on a pair of three-dimensional unstructured finite element meshes, called sf10 and sf5, that model a volume of earth under the San Fernando Valley roughly 50 km x 50 km x 10 km in size [16]. They are *unstructured* in the sense that the neighbors of a node cannot be determined implicitly and must be determined explicitly from adjacency information stored in memory. Sf10 and sf5 are artifacts of the CMU Quake project, which has developed computer techniques for predicting earthquake-induced ground motion in large basins. The "sf" in the names is an abbreviation for San Fernando, and the digit indicates the highest frequency seismic wave (in seconds) that the mesh is able to resolve. For example, sf10 resolves waves with 10 second periods and sf5 resolves waves with 5 second periods. The sf10 mesh is illustrated in Figure 3. Beverly Hills is in the lower right-hand corner. The town of San Fernando is in the midst of the darkly shaded region near the upper left corner.

Sf10 and sf5 are composed of different sized tetrahedra (i.e., pyramids with triangular bases). Each tetrahedron is called an *element*, and the vertices of the tetrahedra are called *nodes*. Edges are undirected, and each node has an implied self-edge. Figure 4 summarizes the basic topological properties of the Spark98 meshes and their corresponding sparse matrices.

The Spark98 meshes are extremely sparse, with only 13-14 nonzero submatrices per row on average and 30 nonzero submatrices per row at most. As we will see in the next section, the number of nonzero

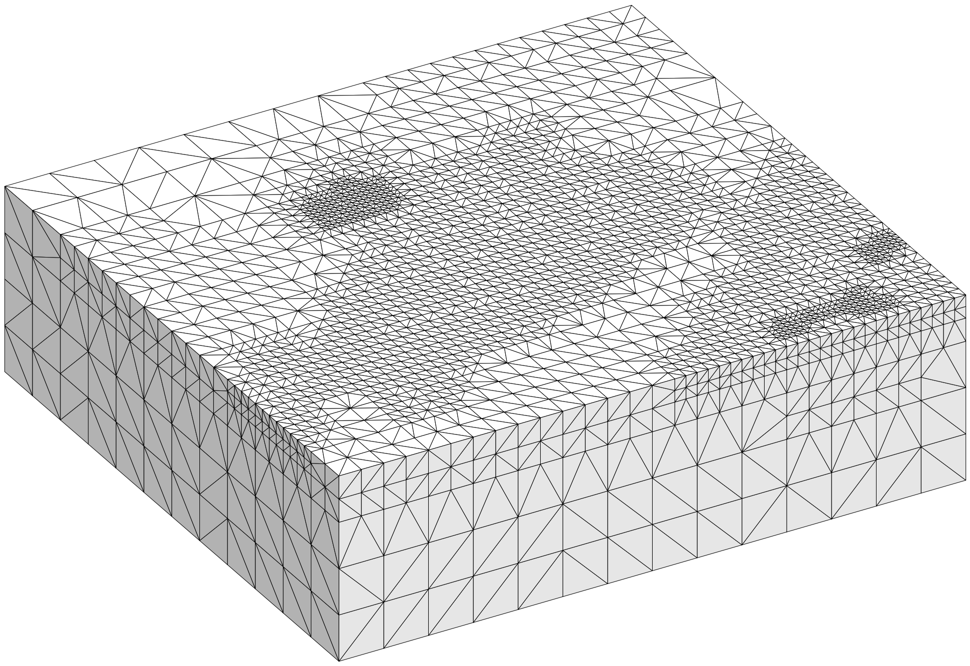

Figure 3: The Spark98 10-second San Fernando mesh.

Mesh	Description	Mesh Entities			Nonzero submatrices		
		Nodes (n)	Edges (e)	Elements	total(nz)	avg/row	max/row
sf10	10-second San Fernando mesh	7,294	52,216	35,025	97,138	13	26
sf5	5-second San Fernando mesh	30,169	220,546	151,239	410,923	14	30

Figure 4: Basic properties of the Spark98 meshes and their matrices.

submatrices in each row determines the trip count of the inner loop of the basic SMVP algorithm. Sparse rows result in short inner loops, which are hard to pipeline or vectorize.

Partitioned versions of the Spark98 meshes are provided for $1, 2, 4, \ldots, 64, 128$ PEs. These partitioned meshes provide a wide range of computation to communication ratios (i.e., the number of floating point operations per PE divided by the number of communicated words per PE), ranging from 500:1 for small numbers of PEs down to 50:1 for large numbers of PEs [19]. The partitioned meshes also provide a challenging communication pattern for the SMVP that lies somewhere between a simple nearest neighbor pattern and a complete exchange [13]. See [19] for a complete characterization of the computation and communication properties of the sf10 and sf5 partitioned meshes, along with some even larger meshes that could appear in future versions of the Spark kernels.

3 The Spark98 programs

Each of the five Spark98 programs executes a sequence of SMVP pairs of the form

```
<read input mesh>
<generate sparse matrices and vectors>
loop i = 1 to iters
   w = K * v
   w' = K' * v'
endloop
```

where K and K' are sparse symmetric *stiffness matrices* and v, v', w, and w' are dense vectors. Each iteration consists of a pair of SMVP operations, rather than a single SMVP operation, to approximate the worst case cache behavior of the SMVP when it is used in a real finite element application. In a real application, SMVP operations are separated by references to other data structures and I/O operations which can replace the cache lines associated with the matrix. So in the worst case, each SMVP operation in the real application must reload its data into the cache, and alternating SMVP operations with different sparse matrices in the kernels approximates this behavior.

A stiffness matrix 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 connected by each of the e edges of the mesh (including self-edges). If the mesh has n nodes, then K has dimension $3n \times 3n$ and each vector has dimension $3n \times 1$. (The constant factors of 3 are due to the fact that the underlying quake simulations model the displacement of the ground in each of three directions.) All scalars are double-precision floating-point numbers.

A stiffness matrix consists of $nz = 2e - n$ nonzeros, where e and n are the number of edges and nodes respectively in the underlying mesh. However, because simulations are often memory bound, the matrix coefficients are stored in a symmetric compressed sparse row (CSR) format where only the e nonzero submatrices on the diagonal and in the upper triangle are actually stored in memory. These e nonzero submatrices are stored row-wise in an e-submatrix coefficient vector A.

The nonzero structure of the stiffness matrix is represented by two integer vectors, $Acol$ and $Aindex$. $Acol$ is an e-element vector such that $Acol[i]$ is the column number of submatrix $A[i]$, and $Aindex$ is an $(n + 1)$-element vector such that $Aindex[i]$ is the starting position in A of row i. $Aindex[n]$ marks the end of the last row and is always set to $e + 1$. Note that while each Spark98 kernel requires two distinct coefficient vectors, one for K and the other for K', it requires only a single pair of $Acol$ and $Aindex$ vectors because K and K' have identical sparsity structure.

Figure 5 shows the sequential SMVP routine that lies at the heart of all of the Spark98 kernels. Note that *local_smvp()* actually peforms the operation $w \leftarrow w + Kv$ rather than $w \leftarrow Kv$, because the former is easier to implement than the latter. However, w is always initially zero, so the effect is the same.

The pattern of data references in *local_smvp()* is ideal in some ways and less than ideal in others. On the one hand, it is ideal in the sense that it scans sequentially through the A coefficient vector (indexed by variable *Anext* in line 25 of Figure 5). The sequential access pattern provides good spatial locality and minimizes memory bank conflicts in systems with interleaved memory systems.

On the other hand, the access pattern is less than ideal in the following sense. Suppose we are manipulating some submatrix A_{ij} during the course of the sequential access of A. Since we are using a storage scheme that exploits symmetry, then we must be sure to use A_{ij} to record the contribution of both A_{ji} and A_{ij} to the result. In particular, $w_i \leftarrow w_i + A_{ij}v_j$ (lines 18-20, 27-29) and $w_j \leftarrow w_j + A_{ji}v_i$ (lines 22-24). The important implication is that some sets of references scan through v and w sequentially (i.e., the references to v_i and w_i), but other accesses are irregular (i.e., the references v_j and w_j). The impact of the irregular accesses of v_j and w_j is tempered somewhat by the fact that each vector entry is really three numbers (recall that the underlying simulation has three degrees of freedom) and thus there is a

```
void local_smvp(int nodes, double (*A)[3][3], int *Acol, int *Aindex,
                double (*v)[3], double (*w)[3], int firstrow, int numrows) {
  int i;
  int Anext, Alast, col;
  double sum0, sum1, sum2;

  for (i = firstrow; i < (firstrow + numrows); i++) {
    Anext = Aindex[i];
    Alast = Aindex[i + 1];

    sum0 = A[Anext][0][0]*v[i][0] + A[Anext][0][1]*v[i][1] + A[Anext][0][2]*v[i][2];
    sum1 = A[Anext][1][0]*v[i][0] + A[Anext][1][1]*v[i][1] + A[Anext][1][2]*v[i][2];
    sum2 = A[Anext][2][0]*v[i][0] + A[Anext][2][1]*v[i][1] + A[Anext][2][2]*v[i][2];

    Anext++;
    while (Anext < Alast) {
      col = Acol[Anext];
      sum0 += A[Anext][0][0]*v[col][0] + A[Anext][0][1]*v[col][1] + A[Anext][0][2]*v[col][2];
      sum1 += A[Anext][1][0]*v[col][0] + A[Anext][1][1]*v[col][1] + A[Anext][1][2]*v[col][2];
      sum2 += A[Anext][2][0]*v[col][0] + A[Anext][2][1]*v[col][1] + A[Anext][2][2]*v[col][2];

      w[col][0] += A[Anext][0][0]*v[i][0] + A[Anext][1][0]*v[i][1] + A[Anext][2][0]*v[i][2];
      w[col][1] += A[Anext][0][1]*v[i][0] + A[Anext][1][1]*v[i][1] + A[Anext][2][1]*v[i][2];
      w[col][2] += A[Anext][0][2]*v[i][0] + A[Anext][1][2]*v[i][1] + A[Anext][2][2]*v[i][2];
      Anext++;
    }
    w[i][0] += sum0;
    w[i][1] += sum1;
    w[i][2] += sum2;
  }
}
```

Figure 5: Spark98 SMVP routine.

limited amount of spatial locality. Still, these irregular accesses are a fundamental reason that the SMVP is a challenging and interesting computational kernel.

The SMVP becomes much easier to compute if we do not exploit symmetry and instead store the entire matrix. In this case, each row of the matrix contributes to a unique subvector of w. If we assign each thread a disjoint set of matrix rows, then there is no contention among either the reads of A or the writes to w, and the only possible contention occurs during the reads of v. However, experience with applications groups makes it very clear that memory efficiency is crucial, and thus the Spark98 kernels are based on the more computationally interesting representation that exploits symmetry.

3.1 SMV: baseline sequential program

SMV is the baseline sequential program. Each SMVP $w = Kv$ is performed with the following procedure calls:

```
zero_vector(w, 0, n);
local_smvp(n, A, Acol, Aindex, v, w, 0, n);
```

Zero_vector(w, j, m) zeros out subvectors $w[j], \ldots, w[j+m-1]$, i.e., the first m subvectors of w starting at position j.

Speedup numbers for the parallel Spark98 programs should always be reported with respect to SMV, not the single-PE version of the parallel program.

3.2 LMV: shared memory program using locks

LMV is a parallel program based on a shared memory model, where a program consists of a collection of p threads that share a single address space. The number of threads is arbitrary and can be specified at runtime. Each thread i is assigned a disjoint and contiguous set of rows of the sparse matrix so that each set has roughly the same number of nonzeros. Since there are on the order of thousands of rows and only tens of nonzeros per row, it is trivial to generate good partitions. Thread i performs the SMVP on its set of rows, using the following call sequence:

```
barrier();
zero_vector(w, firstrow[i], numrows[i]);
barrier();
local_smvp(n, A, Acol, Aindex, v, w, firstrow[i], numrows[i]);
```

Firstrow[i] and *numrows[i]* describe the set of rows assigned to thread i, *barrier()* performs a barrier synchronization among all of the threads. Since different threads might update the same subvector of w, a set of $L > 0$ locks guarantees consistent updates of w:

```
setlock(col % L);
w[col][0] += A[Anext][0][0]*v[i][0] + ...
w[col][1] += A[Anext][0][1]*v[i][0] + ...
w[col][2] += A[Anext][0][2]*v[i][0] + ...
unsetlock(col % L);
```

```
setlock(i % L);
w[i][0] += sum0;
w[i][1] += sum1;
w[i][2] += sum2;
unsetlock(i % L);
```

If lock k is unset, then *setlock(k)* atomically sets lock k and returns. Otherwise, *setlock(k)* waits until the lock becomes unset, then atomically sets it. *Unsetlock(k)* unsets lock k. In general, more locks means fewer synchronizations between threads, with the best case being a unique lock for each subvector. However, most systems limit the number of locks. Thus the number of locks L can be specified at runtime.

It is interesting to observe that the *only* reason we need locks in the first place is because we are exploiting the symmetry of the coefficient matrix by storing only the diagonal and upper triangle. If we were to store the entire sparse matrix, then each thread would update a disjoint portion of the result vector, and the program would be trivially parallelized. This is a classic space/time tradeoff that makes the symmetric version of the SMVP much more interesting than the usual nonsymmetric case.

3.3 RMV: shared memory program using reductions

A simple modification to the LMV program eliminates the need for the locks, but increases the memory requirement and introduces an extra reduction operation. The idea is for each thread i to update its own private w vector. These p private vectors are then summed (reduced) to form the final result vector w.

```
barrier();
zero_vector(privatew[i], 0, n);
local_smvp(n, A, Acol, Aindex, v, privatew[i], firstrow[i], numrows[i]);
barrier();
add_vectors(privatew, w, firstrow[i], numrows[i]);
```

```
void add_vectors(double (**tmpw)[3], double (*w)[3], int firstrow, int numrows) {
   int i, j;
   double sum0, sum1, sum2;

   for (i = firstrow; i < (firstrow+numrows); i++) {
      sum0 = sum1 = sum2 = 0.0;
      for (j=0; j< threads; j++) {
         sum0 += tmpw[j][i][0];
         sum1 += tmpw[j][i][1];
         sum2 += tmpw[j][i][2];
      }
      w[i][0] = sum0;
      w[i][1] = sum1;
      w[i][2] = sum2;
   }
}
```

Figure 6: Reduction operation for RMV.

Privatew is an array of p private w vectors, where are summed by the *add_vectors()* routine to produce the final result vector w. The code for the *add_vectors()* routine is shown in Figure 6. Notice that each thread sums the subvectors in n/p rows of p private vectors, for a total of n subvectors per thread, independent of the number of threads. This suggests that RMV will not scale well as we increase the number of threads.

3.4 MMV: message passing program

The MMV program is based on a *message passing model* where PEs have private memories and use standard MPI message passing primitives to transfer data between the memories [18]. Unlike LMV and RMV, which are parallelized at runtime using a trivial partitioning algorithm, MMV is parallelized at compile time by a sophisticated partitioning algorithm that partitions the mesh into p disjoint sets of tetrahedral elements. Each such set is called a *subdomain* and is assigned to exactly one PE (we use the terms subdomain, PE, and thread interchangeably). The partitioner is based on a recursive geometric bisection algorithm that divides the elements equally among the subdomains while attempting to minimize the total number of nodes that are shared by subdomains, and hence minimize the total communication volume [17, 21]. The algorithm enjoys provable upper bounds on the volume of communication, and in practice [10] generates partitions that are comparable to those produced by other modern partitioners [5, 9, 12, 20, 23].

To compute $w = Kv$ on a set of PEs, we must consider the data distribution by which vectors and matrices are stored. The vectors v and w 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 v_i and w_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 7 demonstrates this method of distributing data.

Given this method of distributing data, the multiplication $w = Kv$ is performed in two steps: (1) *Computation step:* each PE independently computes a local matrix-vector product over the subdomain that resides on that PE. (2) *Communication step:* PEs that share nodes communicate and sum their nodal w values into correct global values for each node. In Figure 7, PEs 1 and 2 must exchange the values of the interface nodes 4, 5, and 6. This exchange-and-add process is also referred to as *full assembly*. The code for each PE has the following form:

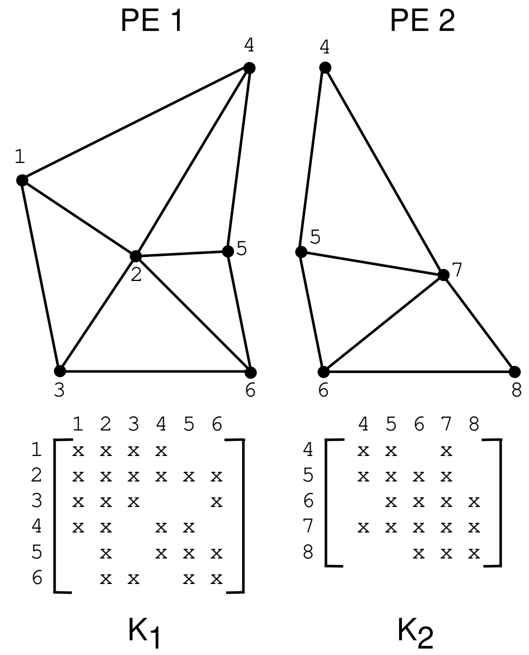

Figure 7: A finite element mesh and corresponding stiffness matrix K, distributed among two PEs. Xs represent nonzero 3×3 submatrices. Note that some nodes are shared by both PEs, as are some submatrices (corresponding to shared edges).

```
zero_vector(w, 0, n);
local_smvp(n, A, Acol, Aindex, v, w, 0, n);
full_assemble(w);
```

The variable n on PE i is the number of nodes n_i assigned to PE i. Similarly, the vector A holds the nonzero submatrices of sparse coefficient matrix K that are assigned to PE i.

All communication between PEs occurs during the *full_assemble()* routine, which resolves the values of the interface nodes using MPI message passing primitives. *Full_assemble()* has access to communication schedules on each PE that tell it which nodes are shared with other PEs, which PEs those nodes are shared with, and how to map between nodes and values included in each message.

Full_assemble() consists of two steps. Step 1: After an initial barrier synchronization (to ensure that the previous computation step has finished on each PE), each PE posts an asynchronous receive operation for each message that it expects to receive from other PEs. Messages are maximally aggregated, and thus each PE receives at most one message from any other PE. The asynchronous receive operations are non-blocking, and their purpose is to tell the message passing system where to store arriving messages in user memory. Each message is stored in its own memory; messages do not share storage. Step 2: After another barrier synchronization (to ensure that all PEs have posted their receive operations), each PE sends a message to every PE with which it shares mesh nodes. After the messages are sent, each PE waits for all of the expected messages to arrive, and then sums the values in the messages into the appropriate nodal values. This two-step communication pattern, which is an example of *deposit model communication* [24], ensures that user-level memory exists for every message and thus avoids unnecessary copying by the message passing system.

3.5 HMV: hybrid message passing/shared memory program

MMV is distinguished by the aggressive way that it partitions its data references. HMV is a hybrid shared memory program that achieves an equivalent effect by using arrays to simulate the private local data structures. For example, the local coefficient matrix A on PE i in the MMV message passing program becomes $A[i]$ in the shared memory program. Similarly, the number of nodes n on PE i becomes $n[i]$. If we use shared memory copy operations instead of message passing transfers to implement the full assembly phase, then the SMVP on each thread has the following form:

```
zero_vector(w[i], 0, n[i]);
local_smvp(n[i], A[i], Acol[i], Aindex[i], v[i], w[i], 0, n[i]);
shmem_full_assemble(w, i);
```

HMV is a hybrid in the sense that it combines aggressive partitioning of references from the message passing program with the shared memory mechanism for the full assembly step.

For both MMV and HMV, the computational load on each PE (or thread) is proportional to the number of nonzero submatrices assigned to each PE. If there are e_i edges and n_i nodes assigned to subdomain i, then there are a total of $nz_i = 2e_i - n_i$ nonzero submatrices. Since each nonzero submatrix requires a 3×3 dense matrix-vector product operation, subdomain i performs a total of $18nz_i$ floating point operations. Figure 8 shows the distribution of nonzeros for partitioned meshes distributed with the Spark98 kernels. However, since symmetry is exploited in the implementation, only e_i nonzero submatrices are actually stored in memory.

In Figure 8, we see that the load is reasonably well distributed for smaller numbers of processors, but that for larger numbers of processors, the load between two PEs can differ by as much as 20%. This imbalance, which is due to the fact that the partitioner distributes mesh elements rather than mesh edges, highlights an interesting tradeoff. On the one hand, if the partitioner distributes mesh elements, then the local sparse matrix coefficients can be generated independently on each PE using legacy sequential code. On the other hand, the amount of work on each processor is more directly related to the number of nonzeros on each processor than the number of elements, which suggests that the partitioner should distribute mesh edges rather than mesh elements. The Quake project opted to simplify the generation of the local stiffness matrices by distributing mesh elements. This is the approach we have taken with the Spark98 kernels, at the cost of non-optimal load balancing.

Another important mesh property is the number of nonzero submatrices per row, which determines the length of the inner loop of the local SMVP routine. Longer inner loops are generally more efficient because the loop and pipeline overheads can amortized better. The statistics for the partitioned meshes distributed with the Spark98 kernels are shown in Figure 9, assuming nonsymmetric storage. The important point about these numbers is that they show us how sparse the matrices really are: no row in any matrix has more than 30 nonzero submatrices, and on average has a paltry 13.

The duration of the communication phase is determined by both the number and size of the messages transferred by each PE. These statistics for the partitioned meshes distributed with the Spark98 kernels are summarized in Figures 10 and 11. The interesting point is that the communication phase tends to consist of a large number of fairly short messages. Thus, message overhead matters, and we cannot count on huge messages to amortize this cost.

subdomains		sf10	sf5
1		97,138	410,923
2	min	48,488	207,201
	avg	49,493	208,226
	max	49,498	209,252
4	min	25,043	104,584
	avg	25,218	105,522
	max	25,403	107,135
8	min	12,849	52,609
	avg	13,087	53,930
	max	13,225	54,740
16	min	6,608	26,927
	avg	6,819	27,604
	max	7,108	28,227
32	min	3,447	13,953
	avg	3,610	14,278
	max	3,784	14,811
64	min	1,806	7,123
	avg	1,942	7,468
	max	2,158	8,232
128	min	967	3,696
	avg	1,053	3,942
	max	1,208	4,278

Figure 8: Distribution of nonzero 3×3 submatrices ($nz_i = 2e_i - n_i$) per subdomain, assuming nonsymmetric storage. Only e_i nonzero submatrices are actually stored in memory.

subdomains		sf10	sf5
1	min	4	5
	avg	13	13
	max	26	30
2	min	4	4
	avg	13	13
	max	26	30
4	min	4	4
	avg	13	13
	max	26	28
8	min	4	4
	avg	13	13
	max	25	27
16	min	4	4
	avg	12	12
	max	23	26
32	min	4	4
	avg	12	12
	max	23	23
64	min	4	4
	avg	10	11
	max	19	22
128	min	4	4
	avg	9	11
	max	22	24

Figure 9: Distribution of nonzero submatrices per row for the subdomain with the fewest average submatrices per row (assuming nonsymmetric storage).

4 Discussion

The Spark98 programs are points on a continuum that is defined by how well each program partitions its data references among computational threads. At one end of the continuum is SMV, which by definition makes no attempt to partition references because it consists of a single thread. For the SMVP operation $w = Kv$, LMV has a better partition than SMV, with disjoint references of K and non-disjoint references of v and w. RMV has a better partition than LMV because it replaces non-disjoint references of w with disjoint ones; however, references to v are still non-disjoint. Finally, MMV and HMV use the best partitions, with disjoint references to K, u, and v.

The quality of the partition is directly related to the difficulty of developing and parallelizing the program. For example, the sequential SMV program is the simplest program to develop because by definition it does not partition any references. LMV is only a little harder to write than SMV, requiring only simple changes to SMV: assigning rows of K to threads, creating the threads, and protecting updates to the result vector with locks. RMV requires only simple modifications to LMV: defining an array of vectors and modifying the call to the local SMVP routine. MMV and HMV are much more difficult to develop because they require a sophisticated partitioner, computation of global-to-local node and element mappings for each PE, computation of the sparse matrix structure for each PE, and a communication schedule for each PE.

subdomains		sf10	sf5
2	min	2	2
	avg	2	2
	max	2	2
4	min	4	2
	avg	5	4
	max	6	6
8	min	4	4
	avg	8	7
	max	12	12
16	min	4	4
	avg	10	11
	max	18	20
32	min	6	6
	avg	16	14
	max	30	30
64	min	6	8
	avg	19	18
	max	38	40
128	min	6	8
	avg	25	21
	max	62	52

Figure 10: Number of messages per subdomain.

subdomains		sf10	sf5
2	min	1,716	5,052
	avg	1,716	5,052
	max	1,716	5,052
4	min	1,344	2,592
	avg	1,839	5,160
	max	2,352	7,746
8	min	1,152	3,258
	avg	1,905	4,893
	max	2,550	7,080
16	min	942	2,028
	avg	1,617	3,849
	max	2,208	5,292
32	min	750	1,926
	avg	1,362	3,045
	max	2,172	4,476
64	min	492	1,026
	avg	1,110	2,382
	max	1,764	4,296
128	min	396	762
	avg	885	1,797
	max	1,740	3,360

Figure 11: Communication words per subdomain.

The quality of the partition is also directly related to performance. This can be seen quite clearly in Figure 12 for the Spark98 sf5 kernels. The kernels were run on 1, 2, 4, and 8 PEs of a 12-PE SGI Power Challenge, a bus based cache coherent shared memory system that supports both shared memory and MPI message passing primitives. The LMV program was run using $k = 1, 2, 4, \ldots, 512$ locks, where 512 is the maximum (power of two) number of locks allowed by the system. The performance of the baseline sequential SMV program is shown as a single dot in the lower lefthand corner of the graph. Performance is reported in absolute throughput units of Mflop/s (millions of floating point operations per sec).[2]

Figure 12 shows how performance improves as programs do a better job of partitioning their data references. MMV and HMV are faster than RMV, which is faster than all instances of LMV. Although the single-PE LMV kernel is slower than the purely sequential SMV program because of lock overheads, given enough locks and threads its performance quickly surpasses that of SMV. However, for small numbers of locks, LMV actually experiences a slowdown as the number of threads increase, due to excessive contention for locks. Also, notice how poorly the performance of RMV scales, as we predicted in Section 3.

Conventional wisdom maintains that shared memory programs are easier to write, but less efficient, than message passing programs. This statement is certainly true for LMV and RMV, which are simpler to write than MMV and provide less than half the performance. However, the statement is not true for HMV, which is just as difficult to write as MMV and has equivalent performance. Thus, on the one hand an efficient shared memory program is not necessarily easier to write than a message passing program, but on the other hand it is not necessarily slower either.

Although MMV and HMV enjoy the best performance, Figure 12 shows that the performance curve is flattening as the number of PEs increases to 8 PEs, a very small number of PEs indeed. This type of

[2] We use Mflop/s rather than MFLOPS, as suggested by [14].

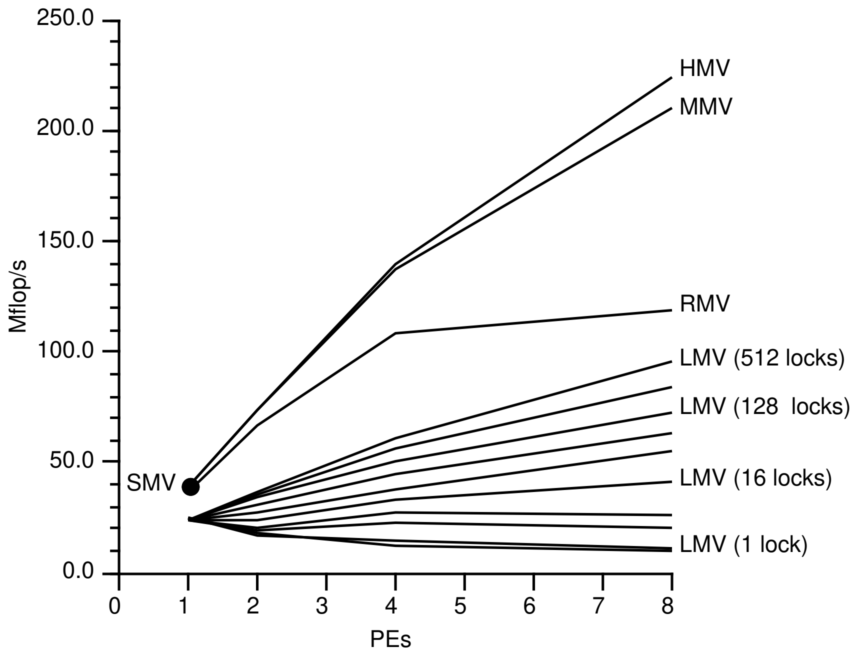

Figure 12: Performance of the Spark98 sf5 kernels on an SGI Power Challenge (195 MHz MIPS R10000, 32 Kbyte Icache, 32 Kbyte L1 Dcache, 2 Mbyte L2 Dcache, cc -O2). Sample points are at 1, 2, 4, and 8 PEs. LMV/sf5 was measured using 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512 locks. SMV/sf5 performance is 39 Mflop/s.

scalability problem does not arise in every system, as shown in Figure 13 for the identical MMV/sf5 kernel running on the Cray T3E, which uses a torus instead of a shared bus to connect the PEs. Designers can use the kernels in this way to help them identify limitations in the communication systems of their parallel computers.

Finally, the sequential SMV program illustrates an interesting property of modern compilers and processors. It is not unreasonable to wonder if more aggressive use of temporary variables in the *local_smvp()* routine might improve performance by eliminating the possibility of pointer aliasing. Interestingly enough, though, loading every input value into a temporary scalar variable makes almost no difference in performance on a modern system. For example, on an SGI Power Challenge, rewriting *local_smvp()* to use as many temporaries as possible raises the measured performance of the SMV/sf5 kernel from 39 Mflop/s to 40.5 Mflop/s, an improvement of only 4%.

5 Concluding remarks

Our purpose in developing Spark98 is to give system builders access to parallel sparse matrix kernels that are realistic, and yet fairly small and easy for one person to understand, modify, and experiment with. The kernels are based on a baseline sequential SMVP program, three shared memory programs, and a message passing program, along with two unstructured three-dimensional meshes from an earthquake ground modeling application. The programs represent points along a continuum that is defined by the care with which data references are partitioned among PEs.

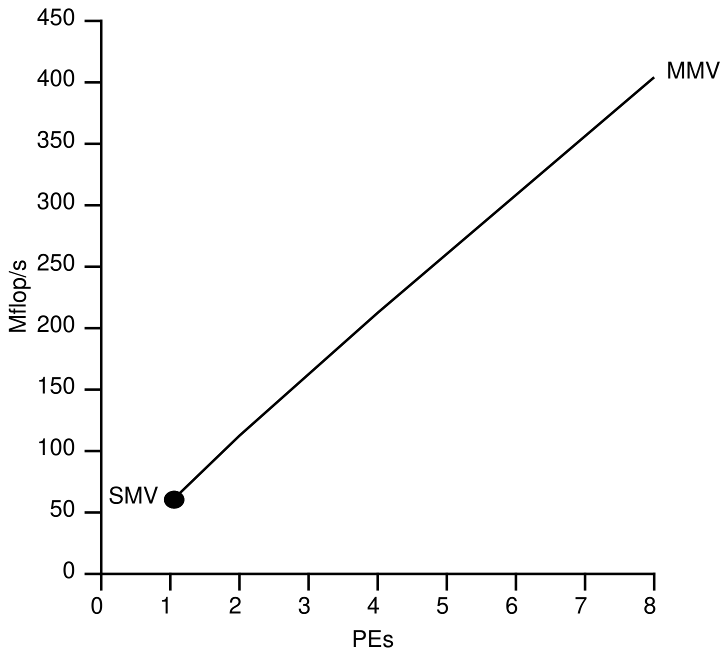

Figure 13: Performance of the Spark98 SMV/sf5 and MMV/sf5 kernels on a Cray T3E (450 MHz Alpha 21164, 8 Kbyte Icache, 8 Kbyte L1 Dcache, 96 Kbyte L2 Dcache, cc -O2). Sample points are at 1, 2, 4, and 8 PEs. SMV/sf5 performance is 57 Mflop/s.

Although we expect that the current set of kernels will be quite useful, there are some limitations that we might want to address in the future. First, as we have already discussed, the Spark98 programs are points along a continuum defined by how well they partition data references among their computational threads. We don't claim to have provided complete coverage of parallel SMVP algorithms, and there are surely other approaches that would be interesting to add to the mix of programs.

Second, it might be useful to include other computations beside the SMVP in the kernels. For example, examples of direct methods for solving sparse linear systems would be helpful.

Finally, the current set of kernels are based on two finite element meshes from the CMU Quake project. It might be useful to include meshes from other types of applications such as computational fluid dynamics.

Acknowledgments

The Spark98 meshes were created by my colleagues on the CMU Quake project: Jacobo Bielak, Omar Ghattas, Loukas Kallivokas, Hesheng Bao, and Jifeng Xu. The meshes were generated, partitioned, and parceled using programs from the Archimedes tool chain that were written by Jonathan Shewchuk while he was a graduate student at Carnegie Mellon. Jifeng Xu also wrote part of an octree program that generates the initial distribution of mesh nodes. Thomas Gross gave helpful perspective on how to develop and organize the kernels. Computer time on the Cray T3E and SGI Power Challenge was provided by the Pittsburgh Supercomputing Center.

References

[1] AMZA, C., COX, A., DWARKADAS, S., HYAMS, C., LI, Z., AND ZWAENEPOEL, W. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer 29*, 2 (Feb 1996), 18–28.

[2] ATLAS, S., BANERJEE, S., CUMMINGS, J., HINKER, P., SRIKANT, M., REYNDERS, J., AND THOLBURN, M. POOMA: A high performance distributed simulation environment for scientific applications. In *Proceedings of Supercomputing '95* (Washington, D.C., Nov. 1995).

[3] BAO, H., BIELAK, J., GHATTAS, O., O'HALLARON, D., KALLIVOKAS, L., SHEWCHUK, J., AND XU, J. Earthquake ground motion modeling on parallel computers. In *Proceedings of Supercomputing '96* (Pittsburgh, PA, Nov. 1996). See also www.cs.cmu.edu/˜quake/.

[4] BAO, H., BIELAK, J., GHATTAS, O., O'HALLARON, D., KALLIVOKAS, L., SHEWCHUK, J., AND XU, J. Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers. *Computer Methods in Applied Mechanics and Engineering* (1997). To appear.

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

[6] BLUMRICH, M., LI, K., ALPERT, K., DUBNICKI, C., FELTEN, E., AND SANDBERG, J. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proc. 21th Intl. Symp. on Computer Architecture* (May 1994), ACM, pp. 142–153.

[7] DONGARRA, J., LUMSDAINE, A., POZO, R., AND REMINGTON, K. A sparse matrix library in C++ for high performance architectures. In *Proceedings of the Second Object Oriented Numerics Conference* (1994), pp. 214–218.

[8] DUFF, I. S., GRIMES, R. G., AND LEWIS, J. G. Sparse matrix test problems. *ACM Transactions on Mathematical Software 15*, 1 (Mar. 1989), 1–14.

[9] FARHAT, C. A simple and efficient automatic FEM domain decomposer. *Comp. & Struct. 28*, 5 (1988), 579–602.

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

[11] GROPP, W., AND SMITH, B. The design of data-structure-neutral libraries for the iterative solution of sparse linear systems. *Scientific Programming 5* (1996), 329–336.

[12] HENDRICKSON, B., AND LELAND, R. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Comput. 16*, 2 (1995), 452–469.

[13] HINRICHS, S., KOSAK, C., O'HALLARON, D., STRICKER, T., AND TAKE, R. An architecture for optimal all-to-all personalized communication. In *Proc. SPAA '94* (Cape May, NJ, June 1994), ACM, pp. 310–319.

[14] HOCKNEY, R. *The Science of Computer Benchmarking*. SIAM, Philadelphia, PA, 1996.

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

[16] MAGISTRALE, H., MCLAUGHLIN, K., AND DAY, S. A geology-based 3-D velocity model of the Los Angeles basin sediments. *Bulletin of the Seismological Society of America 86* (1996), 1161–1166.

[17] MILLER, G. L., TENG, S.-H., THURSTON, W., AND VAVASIS, S. A. Automatic Mesh Partitioning. In *Graph Theory and Sparse Matrix Computation*, A. George, J. R. Gilbert, and J. W. H. Liu, Eds. Springer-Verlag, New York, 1993.

[18] MPI FORUM. MPI: A Message Passing Interface. In *Proc. Supercomputing '93* (Portland, OR, November 1993), ACM/IEEE, pp. 878–883.

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

[20] POTHEN, A., SIMON, H., AND LIOU, K. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications 11* (1990), 430–452.

[21] SCHWABE, E., BLELLOCH, G., FELDMANN, A., GHATTAS, O., GILBERT, J., MILLER, G., O'HALLARON, D., SHEWCHUK, J., AND TENG, S. A separator-based framework for automated partitioning and mapping of parallel algorithms for numerical solution of PDEs. In *Proc.1992 DAGS/PC Symposium* (June 1992), pp. 48–62.

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

[23] SIMON, H. Partitioning of unstructured problems for parallel processing. *Comput. Sys. Engr. 2*, 3 (1991), 135–148.

[24] STRICKER, T., STICHNOTH, J., O'HALLARON, D., HINRICHS, S., AND GROSS, T. Decoupling synchronization and data transfer in message passing systems of parallel computers. In *Proc. of 9th Intl. Conference on Supercomputing* (Barcelona, Spain, July 1995), ACM, pp. 1–10.

[25] ZHOU, Y., IFTODE, L., SINGH, J., LI, K., TOONEN, B., SCHOINAS, I., HILL, M., AND WOOD, D. Relaxed consistency and coherence granularity in DSM systems: A performance evaluation. In *Proceedings of the SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (Las Vegas, NV, June 1997), ACM, pp. 193–205.