Sparse Matrices in MATLAB*P: Design and Implementation

Viral Shah and John R. Gilbert

{viral,gilbert}@cs.ucsb.edu*
Department of Computer Science
University of California, Santa Barbara

Abstract. MATLAB*P is a flexible interactive system that enables computational scientists and engineers to use a high-level language to program cluster computers. The MATLAB*P user writes code in the MATLAB language. Parallelism is available via data-parallel operations on distributed objects and via task-parallel operations on multiple objects. MATLAB*P can store distributed matrices in either full or sparse format. As in MATLAB, most matrix operations apply equally to full or sparse operands. Here, we describe the design and implementation of MATLAB*P's sparse matrix support, and an application to a problem in computational fluid dynamics.

Introduction

MATLAB is a widely used tool in scientific computing. It began in the 1970s as an interactive interface to EISPACK, and LINPACK. Today, MATLAB encompasses several modern numerical libraries such as ATLAS, and FFTW, rich graphics capabilities for visualization, and several toolboxes for such domains as control theory, finance, and computational biology.

Almost all of today's supercomputers are based on parallel architectures. Companies such as IBM, Cray, SGI sell supercomputers with proprietary interconnects. Commodity clusters are omnipresent in research labs today. However, the tools used to program them are still predominantly Fortran and C with MPI or OpenMP.

MATLAB*P brings interactivity to supercomputing. There have been several efforts in the past to parallelize MATLAB. The parallel MATLAB survey [6] discusses most of these projects. Perhaps the most notable project that provides a large scale integration of parallel libraries with a MATLAB interface is NetSolve [2]. NetSolve provides an interfaces by invoking RPC calls through a special MATLAB function, as opposed to MATLAB*P which takes a unique approach to parallelization. The MATLAB*P language is a superset of the MATLAB

* This material is based on research sponsored by Air Force Research Laboratories under agreement number AFRL F30602-02-1-0181. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes not withstanding any copyright notation thereon.

language, and parallelism is propagated through programs using the the dlayout object – p. In the case where these systems use the same underlying packages, we believe that they can all achieve similar performance. However, we believe that the systems have different design goals otherwise and it is unfair to compare them.

Sparse matrices may have dimensions that are often in millions and enough non–zeros that they cannot fit on one workstation. Sometimes, the sparse matrices are themselves not too large, but due to the fill–in caused by intermediate operations (for eg. LU factorization), it becomes necessary to distribute the factors over several processors. Iterative methods maybe a better way to solve such large sparse systems. The goal of sparse matrix support in MATLAB*P is to allow the user perform operations on sparse matrices in the same way as in MATLAB.

1 User's View

In addition to MATLAB's sparse and dense matrices, MATLAB*P provides support for distributed sparse (dsparse) and distributed dense (ddense) matrices. The system design of MATLAB*P and operations on ddense matrices are described elsewhere [12, 7].

The *p* operator provides for parallelism in MATLAB*P. For example, a random parallel dense matrix (ddense) distributed by rows across processors is created as follows:

```
>> A = rand (100000*p, 100000)
```

Similarly, a random parallel sparse matrix (dsparse) also distributed across processors by rows is created as follows: (An extra argument is required to specify the density of non-zeros.)

```
>> S = sprand (1000000*p, 1000000, 0.001)
```

We use the overloading facilities in MATLAB to define a *dsparse* object. The MATLAB*P language requires that all operations that can be performed in MATLAB be possible with MATLAB*P. Our current implementation provides a working basis, but is not quite a drop–in replacement for existing MATLAB programs.

MATLAB*P achieves parallelism through polymorphism. Operations on ddense matrices produce ddense matrices. But once initiated, sparsity propagates. Operations on dsparse matrices produce dsparse matrices. An operation on a mixture of dsparse and ddense matrices produces a dsparse matrix unless the operator destroys sparsity. The user can explicitly convert a ddense matrix to a dsparse matrix using $sparse(A)$. Similarly a dsparse matrix can be converted to a ddense matrix using $full(S)$. A dsparse matrix can also be converted into a MATLAB sparse matrix using `S(:,:)` or `p2matlab(S)`. In addition to the data–parallel SIMD view of distributed data, MATLAB*P also provides a task–parallel SPMD view through the so–called "MM–mode".

MATLAB*P currently also offers some preliminary graphics capabilities to help users visualize dsparse matrices. This is based upon the parallel rendering

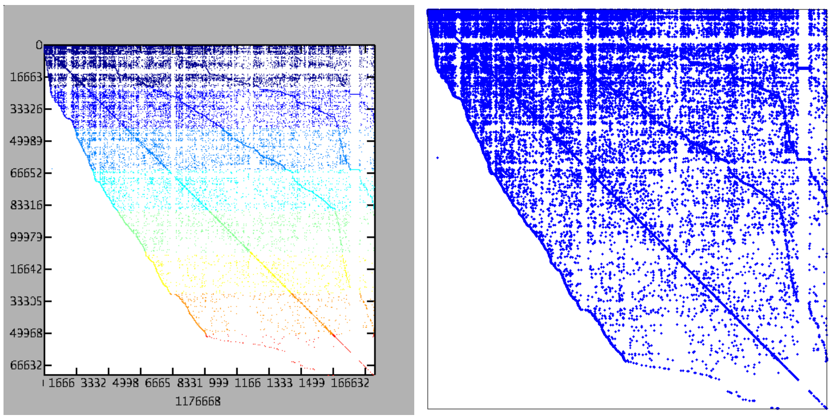

Fig. 1. Matlab and Matlab*P Spy plots of a web crawl dsparse matrix

for ddense matrices [5]. Again, this demonstrates the philosophy that Matlab*P should feel like Matlab. Figure 1 shows the spy plots (showing the non–zeros of a matrix) of a web crawl matrix in Matlab*P and in Matlab.

2 Data Structures and Storage

Matlab stores sparse matrices on a single processor in a Compressed Sparse Column (CSC) data structure [10]. The Matlab*P language allows for matrices to be distributed by block rows or block columns. This is already the case for ddense matrices [12, 7]. The current implementation supports only one distribution for dsparse matrices – by block rows. This is a design choice to prevent the combinatorial explosion of argument types. Block layout by rows makes the Compressed Sparse Row data structure a logical choice to store the sparse matrix slice on each processor. The choice to use a block row layout is not arbitrary, but based on the following observations:

- The iterative methods community largely uses row based storage. Since we believe that iterative methods will be the methods of choice for large sparse matrices, we want to ensure maximum compatibility with existing code.
- A row based data structure also allows efficient implementation of matvec (sparse matrix dense vector product) which is the workhorse of several iterative methods such as Conjugate Gradient and Generalized Minimal Residual.

By default, a dsparse matrix in Matlab*P has the block row layout which would be obtained by ScaLAPACK [3] for a ddense matrix of the same dimensions. This allows for roughly the same number of rows on each processor. The user can override this block row layout in a couple of ways. The Matlab `sparse` function takes arguments specifying a vector of row indices i, a vector of column indices j, a vector of non–zero values v, the number of rows m and the number of columns n as follows:

```
>> S = sparse (i, j, v, m, n)
```

By using a vector *layout* which specifies the number of rows on each processor instead of the scalar m which is simply the number of rows, the user can create a dsparse matrix with the desired layout:

```
>> S = sparse (i, j, v, layout, n)
```

The block row layout of a dsparse matrix can also be changed after creation with:

```
>> changelayout (S, newlayout)
```

The CSR data structure stores whole rows contiguously in a single array on each processor. If a processor has nnz non–zeros, CSR uses an array of length nnz to store the non–zeros and another array of length nnz to store column indices, as shown in Figure 2. Row boundaries are specified by an array of length $m + 1$, where m is the number of rows on that processor.

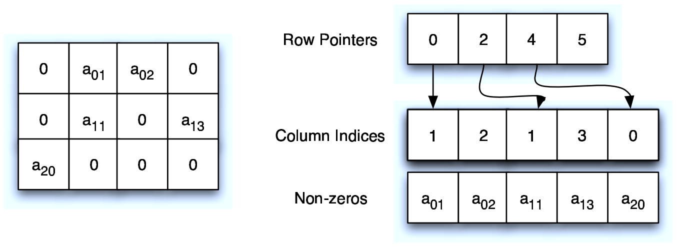

Fig. 2. Compressed Sparse Row (CSR) data structure

Assuming a 32–bit architecture and using double precision floating point values for the non–zeros, an $m \times n$ real sparse matrix with nnz non-zeros uses up $12nnz + 4m$ bytes of memory. Support for complex sparse matrices will be available very soon in Matlab*P.

It would be simple to modify this data structure to allow some slack in each row so that element–wise insertion, for example, could be efficient. However, the current implementation uses the simplest possible data–structure for robustness and efficiency in matrix and vector operations.

3 Operations and Implementation

In this section, we describe the implementation of several sparse matrix operations in Matlab*P. All experiments are performed on a cluster of 2.6GHz

Pentium IV Xeon processors with 3GB RAM and a gigabit ethernet interconnect.

3.1 Parallel Sorting

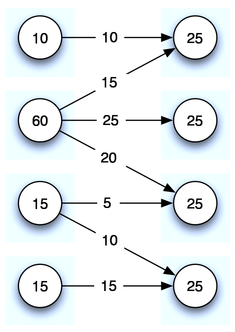

Fig. 3. Starching

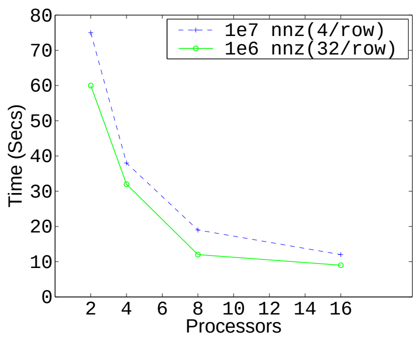

Fig. 4. Scalability of `sparse` (Altix)

Sorting of ddense vectors is an important building block for several parallel sparse matrix operations in MATLAB*P. Sorting is an extremely important primitive for parallel irregular data structures in general. Several parallel sorting algorithms have been surveyed in the literature [4, 8]. Cluster computers have distributed memory, and the interconnect typically has high latency and low bandwidth compared to shared memory computers. As a result, it is extremely important to minimize the communication while sorting. We are experimenting with SampleSort with different sampling techniques, and other median–based sorting ideas. Although we have an efficient parallel sorting algorithm already in MATLAB*P, we are still trying to improve it, given the importance of having fast sorting. We will describe results from our efforts in a future paper.

Several sorting algorithms produce a data distribution that is different from the input distribution. Hence a reshuffling of the sorted data is often required. We refer to this process as Starching, as shown in Figure 3.1. The distribution after sorting is on the left, whereas the desired distribution is on the right in the bipartite graph. The weights on the edges of the bipartite graph show the communication required between pairs of processors during starching. This step is required to ensure consistency of MATLAB*P's internal data structures.

3.2 Constructors

There are several ways to construct parallel sparse matrices in MATLAB*P:

1. `matlab2pp` converts a sequential MATLAB matrix to a distributed MATLAB*P matrix. If the input is a sparse matrix, the result is a dsparse matrix.

```
function s = sparse (i, j, v)

[j, perm] = sort(j);
i = i(perm); v = v(perm);

[i, perm] = sort(i);
j = j(perm); v = v(perm);

starch (i, j, v);
s = assemble (i, j, v);
```

Fig. 5. Implementation of `sparse`

2. `sparse` – `sparse` works with $[i, j, v]$ triples, which specify the row value, the column value and the non–zero value respectively. If i, j, v are ddense vectors with nnz non–zeros, then `sparse` assembles a sparse matrix with nnz non–zeros. If there are duplicate $[i, j]$ indices, the corresponding values are summed. The pseudocode for `sparse` is shown in Figure 3.1. However, in our implementation, we implement this by sorting the vectors simultaneously using row numbers as the primary key, and column numbers as the secondary key.
   The starch phase here is similar to the starching used in the parallel sort, except that it redistributes the vectors so that row boundaries do not overlap among processors and the required block row distribution for the sparse matrix is achieved. The assemble phase actually constructs a dsparse matrix and fills it with the non–zero values. Figure 4 shows the scalability of `sparse` on an SGI Altix 350. Although performance for commodity clusters cannot be as good as that of an Altix, our initial experiments do indicate good scalability on commodity clusters too. We will report more detailed performance comparisons in a future paper.
3. `spones, speye, spdiag, sprand` etc. – Some basic functions implicitly construct dsparse matrices.

3.3 Matrix Arithmetic

One of the goals in designing a sparse matrix data structure is that, wherever possible, it should support matrix operations in time proportional to flops. As a result, arithmetic on dsparse matrices is performed using a sparse accumulator (SPA). Gilbert, Moler and Schreiber [10] discuss the design of the SPA in detail. Matlab*P uses a separate SPA for each processor.

3.4 Indexing, Assignment and Concatenation

The syntax of matrix indexing in Matlab*P is the same as in Matlab. It is of the form $A(p, q)$. p and q can each be either a range $(1 : n)$, or a permutation vector or scalars. Depending on the context, however, this can mean different things.

```
>> B = A(p,q)
```

In this case, the indexing is done on the right side of "=" which specifies that B is a submatrix of A. This is the `subsref` operation in MATLAB.

```
>> B(p,q) = A
```

On the other hand, indexing on the left side of "=" specifies that A should be stored in a submatrix of B. This is the `subsasgn` operation in MATLAB.

If p and q are both integers, $A(p, q)$ directly accesses the dsparse data structure. If p or q are vectors or a range, $A(p, q)$ calls `find` and `sparse`. `find` is the reverse of `sparse` – it converts the matrix from CSR to $[i, j, v]$ format. In this format, it is very easy to find $[i, j]$ pairs which satisfy the indexing criteria. The resulting submatrix is then assembled by simply calling `sparse` .

MATLAB also supports horizontal and vertical concatenation of matrices. The following code, for example, concatenates A and B horizontally, C and D horizontally, and finally concatenates the results of these two operations vertically.

```
>> S = [ A B; C D ]
```

The basic primitives, `find` and `sparse` are used to provide support for concatenation operations in MATLAB*P.

3.5 Matvec

The matvec operation multiplies a dsparse matrix with a ddense column vector, producing a ddense column vector as a result. Matvec is the kernel for many iterative methods.

For the matvec, $y = Ax$, we have A and x distributed across processors by rows. The submatrix of A at each processor will need a piece of x depending upon its sparsity structure. When matvec is invoked for the first time on a dsparse matrix A, MATLAB*P computes a communication schedule for A and caches it. When more matvecs are performed using A, this communication schedule does not need to be recomputed, which saves some computing and communication overhead, at the cost of extra space required to save the schedule. MATLAB*P also overlaps the communication and computation during matvec. This way, each processor starts computing the result of the matvec whenever it receives a piece of the vector from any other processor. Figure 6 also shows how matvec scales in MATLAB*P, since it forms the main computational kernel for conjugate gradient.

Communication in matvec can be reduced by performing graph partitioning of the graph of the sparse matrix. If fewer edges cross processors, lesser communication is required during matvec. MATLAB*P can use several of the available tools for graph partitioning. However, by default, MATLAB*P does not perform graph partitioning during matvec. The philosophy behind this decision is similar to that in MATLAB, that reorganizing data to make later operations more efficient should be possible, but not automatic.

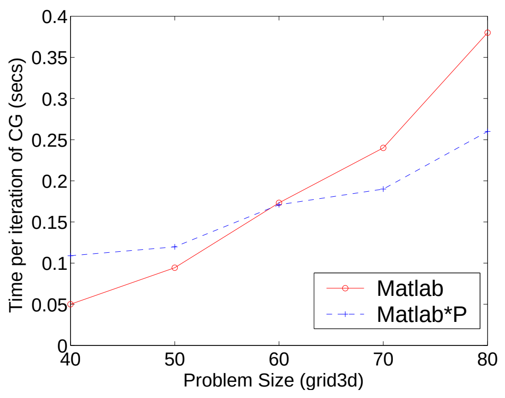

Fig. 6. Time per iteration of CG, scalability of matvec

3.6 Solutions of Linear Systems

MATLAB solves the linear system $Ax = b$ with the matrix division operator, $x = A\backslash b$. In sequential MATLAB, $A\backslash b$ is implemented as a polyalgorithm [10], where every test in the polyalgorithm is cheaper than the next one.

1. If A is not square, solve the least squares problem.
2. Otherwise, if A is triangular, perform a triangular solve.
3. Otherwise, test whether A is a permutation of a triangular matrix (a "morally triangular" matrix), permute it, and solve it if so.
4. Otherwise, if A is Hermitian and has positive real diagonal elements, find a symmetric minimum degree ordering p of A, and perform the cholesky factorization of $A(p,p)$. If successful, finish with two sparse triangular solves.
5. Otherwise, find a column minimum degree order p, and perform the LU factorization of $A(:,p)$. Finish with two sparse triangular solves.

Different issues arise in parallel polyalgorithms. For example, morally triangular matrices and symmetric matrices are harder to detect in parallel. One also expects to be able to use iterative methods. Design for the right polyalgorithm for \ in parallel is an active research problem. For now, MATLAB*P uses a parallel general direct sparse solver for \, which is SuperLU_DIST [14] by default, although a user can choose to use MUMPS [1] too.

The open question at this point is, should MATLAB*P use preconditioned iterative methods to solve sparse linear systems instead of direct methods. Currently, iterative methods are not usable as a black box, and not yet suitable for MATLAB*P.

3.7 Iterative methods - Conjugate Gradient

Conjugate Gradient is an iterative method used to solve a symmetric, positive definite system of equations. The same code is used for MATLAB*P and matlab, except that the input is dsparse in the MATLAB*P case. In Fig 6, `grid3d(k)` is a routine used from the meshpart [9] toolbox, which returns a $k^3 \times k^3$ symmetric positive definite matrix A with the structure of the $k \times k \times k$ 7–point grid.

4 An Application in Computational Fluid Dynamics

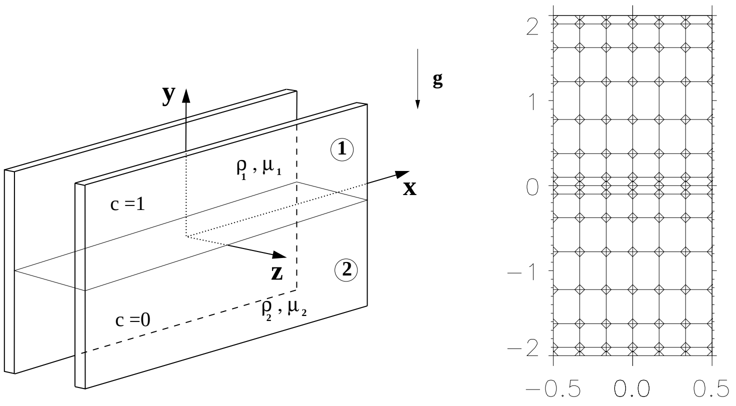

Fig. 7. Geometry of the Hele-Shaw cell (left). The heavier fluid is placed above the lighter one. Either one of the fluids can be the more viscous one. Mesh point distribution in the computational domain (right). A Chebyshev grid is employed in the y–direction, and compact finite differences in the z–direction.

We are using a prototype version of MATLAB*P in collaboration with a number of domain scientists for applications in computational science and engineering. We describe an application here.

Goyal and Meiburg [11] are studying the influence of viscosity variations on the density–driven instability of two miscible fluids. The two fluids, of different density and viscosity are in a vertical Hele–Shaw cell as shown in figure 7. This problem is used to model porous media flows and finds applications in enhanced oil recovery, fixed bed regeneration and groundwater flows.

Fig. 7 shows the discretization of the problem, which yields an algebraic system of the form $A\phi = \sigma B\phi$. The eigenvalue σ represents the growth rate of the perturbations, while the eigenvector ϕ reflects the shape of the perturbations. A positive (negative) eigenvalue indicates unstable (stable) behavior. The system has a 5×5 block structure reflecting the 5 variables at each mesh point (3 velocity components u, v and w, relative concentration of the heavier fluid c, and pressure p).

A discretization of 165×25 points turns out to be sufficient for this problem. Since we solve for 5 variables at each grid point, the matrix A is of the

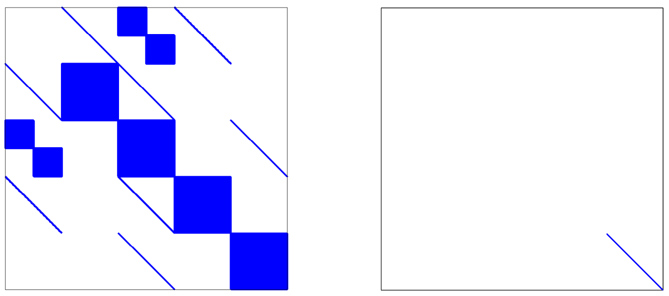

Fig. 8. Spy plots of the matrices A and B

size $20,625 \times 20,625$. The number of non–zeros is $3,812,450$. The matrix is unsymmetric, both in values and nonzero structure, as shown in the spy plots in Figure 4. In order to calculate the largest eigenvalue, we use the power method with shift and invert in Matlab*P.

```
function lambda = peigs (A, B, sigma, iter)

  [m n] = size (A);
  C = A - sigma * B;
  y = rand (n*p, 1);

  for k=1:iter
    q = y ./ norm (y);
    v = B * q;
    y = C \ v;

    theta = dot (q, y);
    res = norm (y - theta * q);
    if res <= 0.0001,  break;   end
  end

lambda = 1 / theta + sigma;
```

Fig. 9. Matlab*P `code for power method with shift and invert`

The original non–Matlab*P code used LAPACK with ARPACK [13], while the Matlab*P code is using SuperLU_DIST with the power method as shown in figure 4. We use a guess of 0.1 to initialize the power method and it converges to 0.0194 which is enough precision for linear stability analysis. We use a cluster with 16 processors to solve the generalized eigenvalue problem. Each node has a 2.6GHz Pentium Xeon CPU, 3GB of RAM and a gigabit ethernet connection. Results are presented in Table 1.

Table 1. Time to solve the generalized eigenvalue problem

No. of processors	Time (seconds)
4	90
8	39
16	33

As a next step we want to incorporate a variable viscosity net flow through the Hele–Shaw cell to incorporate the potentially destabilizing effects of viscous fingering into play, so that the possibility of complex interactions between density- and viscosity-driven instabilities arises. The existence of a more complex flow field necessitates a finer grid and a larger domain size for the linear stability calculations as compared to the previous case discussed where the two fluids were essentially at rest with respect to each other. We expect that we will require about 10 times more computing resources (CPU and memory) to tackle these challenges .

5 Conclusion

The implementation of sparse matrices in MATLAB*P is work in progress. Current available functionality includes being able to construct sparse matrices, perform element–wise arithmetic and indexing operations on them, multiply a sparse matrix with a dense vector and solve linear systems. This level of functionality allows us to implement several algorithms such as conjugate gradient and the power method.

Much remains to be done. A complete implementation of sparse matrices requires matrix–matrix multiplication and several factorizations (Cholesky, QR, SVD etc). Improvements in the sorting code can lead to general improvements in many parts of MATLAB*P. It is also important to make existing graph partitioners available in MATLAB*P – Meshpart and ParMetis. Several preconditioning methods also need to be implemented for MATLAB*P, since iterative methods might possibly be the way to solve large linear systems.

The goal of sparse matrix support in MATLAB*P is to provide an interactive environment for users to perform operations on large sparse matrices in parallel, while being compatible with MATLAB. Our current implementation is ready to be used for simple real life problems.

6 Acknowledgements

David Cheng made useful contributions to the sorting code. We had several useful discussions with Parry Husbands, Per–Olof Persson, Ron Choy, Alan Edelman, Nisheet Goyal and Eckart Meiburg.

References

1. Patrick Amestoy, Iain S. Duff, and Jean-Yves L'Excellent. Multifrontal solvers within the PARASOL environment. In *PARA*, pages 7–11, 1998.
2. D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.
3. L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide.* SIAM, Philadelphia, PA, 1997.
4. Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine cm-2. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 3–16. ACM Press, 1991.
5. Oskar Bruning, Jack Holloway, and Adnan Sulejmanpasic. Matlab *p visualization package. 2002.
6. Long Yin Choy. Parallel Matlab survey. 2001. http://theory.csail.mit.edu/~cly/survey.html.
7. Long Yin Choy. MATLAB*P 2.0: Interactive supercomputing made practical. *M.S. Thesis, EECS*, 2002.
8. D. E. Culler, A. Dusseau, R. Martin, and K. E. Schauser. Fast parallel sorting under LogP: from theory to practice. In *Proceedings of the Workshop on Portability and Performance for Parallel Processing*, Southampton, England, July 1993. Wiley.
9. John. R. Gilbert, Gary L. Miller, and Shang-Hua Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.
10. John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.
11. Nisheet Goyal and Eckart Meiburg. Unstable density stratification of miscible fluids in a vertical hele-shaw cell: Influence of variable viscosity on the linear stability. *Journal of Fluid Mechanics*, (To appear), 2004.
12. P. Husbands and C. Isbell. MATLAB*P: A tool for interactive supercomputing. *The Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.
13. R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK Users Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods.* SIAM, Philadelphia, 1998.
14. Xiaoye S. Li and James W. Demmel. Superlu_dist: A scalable distributed–memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Softw.*, 29(2):110–140, 2003.