Towards the Effective Parallel Computation of Matrix Pseudospectra [*]

C. Bekas, E. Kokiopoulou
Computer Engineering and Informatics Department
University of Patras Greece
knb@hpclab.ceid.upatras.gr
kokiopou@ceid.upatras.gr

I. Koutis
Computer Science Department
Carnegie Mellon University
jkoutis@cs.cmu.edu

E. Gallopoulos
Computer Engineering and Informatics Department
University of Patras Greece
stratis@hpclab.ceid.upatras.gr

ABSTRACT

Given a matrix A, the computation of its pseudospectrum $\Lambda_\epsilon(A)$ is a far more expensive task than the computation of characteristics such as the condition number and the matrix spectrum. As research of the last 15 years has shown, however, the matrix pseudospectrum provides valuable information that is not included in other indicators. So, we ask how to compute it efficiently and build a tool that would facilitate engineers and scientists to make such analyses? In this paper we focus on parallel algorithms for computing pseudospectra. The most widely used algorithm for computing pseudospectra is embarassingly parallel; nevertheless, it is extremely costly and one cannot hope to achieve absolute high performance with it. We describe algorithms that have drastically improved performance while maintaining a high degree of large grain parallelism. We evaluate the effectiveness of these methods in the context of a MATLAB-based environment for parallel programming using MPI on small, off-the-shelf parallel systems.

Keywords

Pseudospectra, MPI, MATLAB, NOWs

1. INTRODUCTION AND MOTIVATION

Let $A \in \mathbb{C}^{n\times n}$ have singular value decomposition (SVD) $A = U\Sigma V^*$, where Σ is diagonal with nonnegative elements $\sigma_j, j = 1, ..., n$, called the singular values of A, and U, V unitary matrices having as columns the left and right singular vectors of A. Let also $\Lambda(A)$ be the set of eigenvalues of A. The $\epsilon-$pseudospectrum $\Lambda_\epsilon(A)$ (pseudospectrum for short) of a matrix describes the locus of eigenvalues of $\Lambda(A+E)$, for all possible E such that $\|E\| \leq \epsilon$ for given ϵ. The $\epsilon-$pseudospectra are regions of the complex plane that show where the eigenvalues of a matrix could go when the matrix is subject to perturbations and for this reason have many interesting properties; see Fig. 1 for examples of pseudospectra of specific matrices. When A is normal (i.e. satisfies the relation $AA^* = A^*A$), the regions are readily computed from the eigenvalues and classical matrix theory, that predicts that the pseudospectrum will consist of the union of the disks of radius ϵ surrounding each eigenvalue of A. The pseudospectrum becomes of interest, on its own or as an alternative to standard eigenvalue analysis, when A is not normal (e.g. nonsymmetric); see [17, 19]. An important barrier in making pseudospectra a standard engineering tool is the expense involved in their calculation.

[*]This work has been partially supported by the Greek General Secretariat for Research and Development, Project ΠΕΝΕΔ 99-07

1. Define mesh Ω_h on a region of the complex plane that contains $\Lambda(A)$.
2. Compute $s(z) := \sigma_{\min}(zI - A)\ \forall z \in \Omega_h$.
3. Plot ϵ contours of $s(z)$.

Table 1: GRID method for computing pseudospectra.

In the sequel we assume that we use the spectral norm $\sigma_{\max}(A) = \|A\|_2$; we also define $s(z) := \sigma_{\min}(zI - A)$, the minimum singular value of matrix $zI - A$. It is known that at points where the minimum singular value is simple, function $s(z)$ is real analytic, that is it can be expanded as Taylor series of two variables (x, y) where $z = x + iy$. In the sequel, even if we write $s(z)$, we really mean $s(x, y)$. It is known that $\Lambda_\epsilon(A)$ can also be defined as the $z \in \mathbb{C}$ that satisfy

$$\Lambda_\epsilon(A) = \{z \in \mathbb{C} : s(z) \leq \epsilon\}, \qquad (1)$$

where $\sigma_{\min}(\cdot)$ denotes the smallest singular value of its argument matrix.

The standard algorithm (GRID) for the computation of $\Lambda_\epsilon(A)$ is presented in Table 1. Two important features of GRID are its straightforward simplicity and robustness. Its cost is typically modeled by

$$C_{GRID} = |\Omega_h|\ C_{\sigma_{\min}} \qquad (2)$$

where $|\Omega_h|$ denotes the number of nodes of Ω_h and $C_{\sigma_{\min}}$ is a measure of the average cost for the computation of $s(z)$. The total cost quickly becomes prohibitive with the increase of either the number of nodes or the size of A. Given that the cost of computing $s(z)$ is at least $O(n^2)$ and that a typical grid could easily contain $O(10^4)$ points, the cost can be dramatic, even for matrices of small size. To get an idea of the expense, we note that the MATLAB function `pscont` from [9] that implements this algorithm took more than 2000 sec to compute the pseudospectrum of a matrix of size 200 on a 100×100 grid.

Cost formula (2) readily indicates two major classes of methods for accelerating the computation, based on the mathematics and numerics of the problem: a) Reducing the number of nodes z and hence the number of evaluations of $\sigma_{\min}$, and b) reducing the cost of each evaluation of $s(z)$. We would be referring to methods that belong to category (a) as *domain-based* and to those that belong to (b) as *matrix-based*. The above can be combined with system-level approaches, such as the exploitation of hierarchical memory and parallel processing. All these approaches are the subject of intense active research; see [18] for a comprehensive survey of recent efforts as well as the valuable Oxford Web site [16]. Not surprisingly, given the computational complexity of the problem, it is expected that a successful approach must combine all of the above techniques. Moreover, in addition to accuracy and speed, a practical package would also be evaluated on the basis of its user friendliness. Pursuing this work, this paper contributes with the following:

i) Parallel algorithms for two recent methods, Pseudospectrum Descent and Inclusion-Exclusion /Modified Grid, for computing pseudospectra.

ii) Static assignment schemes as alternatives to more expensive dynamic policies for load balancing when computing pseudospectra.

iii) MATLAB implementations based on MPI and parallel multitasking on networks of uniprocessor PC's.

1.1 Computational environment

We used cheap off-the-self computing equipment: Single processor PCs, running Windows 2000, connected over fast a Ethernet network. We developed our programs using MATLAB and conducted our experiments over a novel environment that allows the concurrent operation of MATLAB; see the next subsection. We use standard test matrices, drawn from the Test Matrix Toolbox ([9]) and the Harwell-Boeing collection ([7]), throughout our discussion.

1.1.1 Programming environment

We used the Cornell Multitask Toolbox [20], developed by J.A. Zollweg and A. Verma at the Cornell Theory Center, for MATLAB (version 5.3), the widely used problem solving environment from Mathworks. The multitasking toolbox enables multiple copies of MATLAB to run simultaneously on a network of workstations and to exchange matrices, thus facilitating parallel computations. MATLAB copies are started on each machine using a command of the form

```
mpirun -wd mydir -np size matlabdir\bin\matlab.exe,
```

Cluster				
CPU	@ MHz	Cache Kb	RAM Mb	Network
16 PIII	450-550	512	128	100 Mbit

Table 2: Hardware environment: PIII stands for Pentium III.

where *mydir* is the working directory and *size* is the number of Matlab copies to be started. The Cornell Multitask toolbox makes message-passing functions available to the programmer so that parallel programs can be written as m-files. This convention is similar to that adopted by the MPI Forum; the toolbox runs on top of a message passing environment such as MPI/Pro. It is worth noting, that if one wants to make use of the system, he would lose most of the support available in the form of MATLAB's intrinsic function library. In our case, however, the algorithms we use are of large granularity and justify the use of this system: In all cases, each processor is assigned the computation of one or more instances of $s(z)$.

1.1.2 Hardware Environment

Experiments were run on a 16-node cluster of `Pentium III` PCs with clock speeds ranging from 450-550 MHz. The network was a 100 `Mbit` fast Ethernet. The configuration is summarized in Table 2.

2. PARALLEL GRID AND LOAD BALANCING

As described in the Introduction, `GRID` is an embarassingly parallel algorithm; this is assuming that we take the calculation of a single singular value as its basic computational granule. Consider the application of `GRID` for a typical matrix. Model (2) predicts that the total cost is equal to the number of gridpoints times the *average cost of computing $s(z)$ per gridpoint z*. On the basis of this model, we conclude that on P processors, where $P \ll |\Omega_h|$, the parallel cost would be equal to $T(\texttt{GRID}, P) \approx T(\texttt{GRID}, 1)/P$.

If the time needed by the chosen method to compute $s(z)$ at any point of the grid is nearly constant, then the parallel version of `GRID` can trivially use any static assignment, in which every processor is allocated $|G_j| \approx |\Omega_h|/P$ gridpoints (no other constraints need to be satisfied), to obtain nearly perfect parallel performance. For this to hold, however, we must assume that the workload for each SVD is balanced across gridpoints. For example, let us take the case of a typical matrix used in pseudospectra computations, like `grcar` of size $n = 500$ [9]. Then if we use MATLAB on a Pentium III @ 700 MHz and apply its intrinsic function `svd` that computes (all) singular values using a direct method, the average runtime of each computation for a grid of size $|\Omega_h| = d \times 2d$, where $d = 25$, is $C_{\sigma_{\min}} = 4.6$ sec while the minimum and maximum costs of the calculation at each point z are in the interval $[4.57, 4.65]$. The conclusion is straightforward: If we use a direct method to compute $s(z)$, then equidistribution of gridpoints to the processors is sufficient for near perfect speedups for `GRID`.

The situation changes drastically if we assume that the size and sparsity structure of the problem force us to use an it-

21.36	12.92	4.34	16.90	75.46	13.39	2.48	5.25	11.28	15.74
23.47	11.14	2.50	22.16	22.11	354.50	14.97	2.48	6.95	12.05
21.53	13.04	3.45	262.12	6.94	6.93	42.62	2.49	4.32	10.41
24.02	17.14	9.63	2.60	2.50	2.51	6.90	15.06	2.59	7.86
27.96	20.61	17.19	12.98	13.04	2.48	6.91	19.80	2.47	6.98

Table 3: Runtimes (sec) **using `svds` on 700 MHz Pentium III.**

0.92	0.58	0.28	0.34	0.38	0.25	0.24	0.43	0.61	0.57
0.95	0.58	0.22	0.38	0.40	0.38	0.24	0.21	0.41	0.58
0.91	0.57	0.29	0.36	0.68	0.41	0.37	0.25	0.29	0.38
0.93	0.57	0.39	0.31	0.24	0.25	0.38	0.31	0.21	0.40
0.89	0.93	0.55	0.56	0.91	0.24	0.67	0.36	0.24	0.40

Table 4: Runtimes (sec) **using `lansvd` on 700 MHz Pentium III.**

erative method to compute each $s(z)$. Consider the same matrix as before, and the variability, with z, in the number of iterations of the iterative method, `svds` contained in MATLAB. Then the times for the same grid are tabulated in Table 3. The high variability implies that we have to be careful about the assignment of gridpoints to processors and the effects of load balancing. We also note that the above numbers immediately motivate research into *matrix based* methods for speeding up the computation of each $s(z)$. We thus opted, in the course of this research, to use a Lanczos based procedure that offered significantly better performance than the `svds` procedure from Mathworks. In particular, in the tradition of earlier work ([4]), we used a recent, effective, Lanczos based algorithm, proposed and implemented in MATLAB by R.M. Larsen [11]. Algorithm `lansvd` uses Lanczos bidiagonalization and partial reorthogonalization to compute the singular values of large sparse matrices. `lansvd` represents a development over approaches that are based on the explicit approximation of the eigenvalues of A^*A or those of the augmented system $[0, A; A^*, 0]$. As an indication of its performance, we tabulate in Table 4 the runtimes corresponding to the use of `lansvd` for $s(z)$ at the same points where we used `svds` in Table 3. Observe now that even though these numbers are much lower, they occasionally differ by a factor of 3. Clearly, with any of these iterative methods, the allocation of gridpoints to processors must be done in a manner that utilizes efficiently the resources. Unfortunately, however, we do not have a priori information regarding the coordinates of points where the costs will be much higher than others. So what could we do? The major idea, discussed in the few papers that have appeared on the parallel computation of pseudospectra [8, 15], is to use a dynamic allocation scheme, in which gridpoints are entered in a centralized queue that is managed by some processor; as processors become available, they pick a gridpoint, z, from the queue and go ahead with the computation of $s(z)$; the entire computation of $s(z)$ can be achieved without any need for communication with other processors. Such a mechanism can be implemented at little extra cost in a shared memory environment. In this case, however, our main interest is for NOW type environments, where parallelism originates mostly from the interconnection of PC's, some of which might contain more than one processors. On the other hand, depending on the details of the system, there is the possibility of overhead resulting from the communication between processors and the processors that runs the queue as well as because of contention in accessing the queue. This dynamic scheme was used in the literature because it was assumed that we had no a priori information regarding the time needed for the computation of each $s(z)$. But is it so? In the sequel, we show that a heuristic strategy, based on the mathematics of the problem, appears to often offer satisfactory load balance with a static assignment, or else, to be a good initial guess for the implementation of a hybrid policy.

2.1 Orderings for static assignment

We next consider the computation of the pseudospectrum for matrices `fish` and `triangle` of size $n = 300$ on half of a 50×50 grid. The names of these matrices originate from their pseudospectra, depicted in Fig. 1. The runtimes per grid point are depicted in Fig. 2. As before, it is clear that there is little that can be said, a priori, regarding the distribution and one would tend to opt immediately for fully dynamic load balancing. On the other hand, we observe that runtimes do not vary abruptly from point to point, therefore we expect that a block assignment would likely handle points of similar computational difficulty for $s(z)$, therefore it can run into load balancing problems, e.g. if all points of the block are of similar difficulty (say high) whereas another block handles only points that are easier. In addition, the multicolor ordering has the property that every processor corresponds to gridpoints lying on the antidiagonals of the domain, and is thus likely to deal with a more representative sample from the entire grid.

Consider, now, an ad-hoc static assignment based on multicoloring (MC), reminiscent of orderings used in PDE solvers. In particular, we colored gridpoints so that every P gridpoints lying adjacent on the same row or column are colored differently. The coloring is simple to implement, e.g. labeling each gridpoint z_{jk} with its indices (j, k) and then using color numbers $0 \leq c \leq P-1$ for any gridpoint whose indices satisfy $\texttt{mod}(j+k, P) = c$. The case $P = 2$ corresponds to the well-known red-black coloring. We should note, however, that there is a fundamental difference from the grid multicoloring applied in PDEs, since in MC, each processor is assigned all gridpoints with the same color.

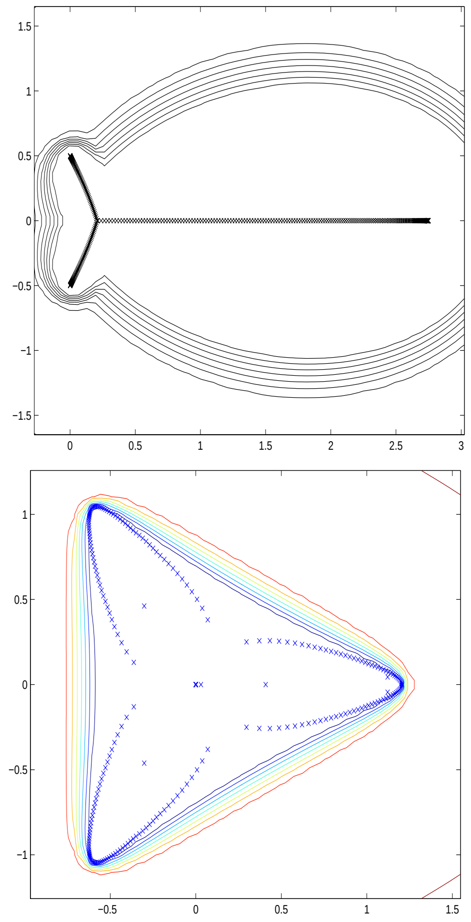

Figure 1: Pseudospectra of matrices `fish` and `triangle` of size $n = 300$ and ϵ from 10^{-1} to 10^{-9}.

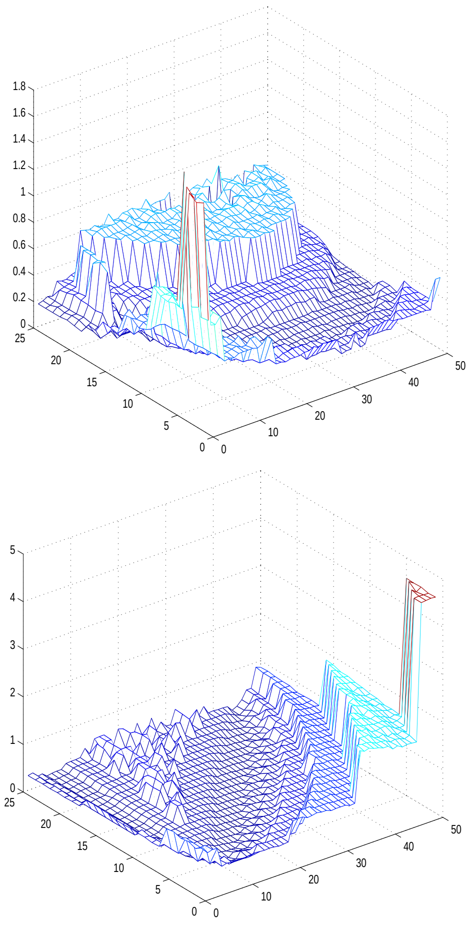

Figure 2: Runtimes (sec) for pseudospectra of matrices `fish` and `triangle` of size $n = 300$.

We experimented, with this multicolored static assignment scheme (abbreviated MC) and compared its performance with the standard static block assignment (SB), in which each block consisted of a set of adjacent vertical lines of gridpoints. We also compared with a scheme in which load balancing is achieved dynamically, by means of a central queue (QUEUE) from which tasks are dispatched to free processors. An assignment such as SB would have been a natural partition, had we decided to compute $s(z)$ with a direct method. Table 5 depicts the runtimes of the above schemes. The times for the queue policy are listed in the row corresponding to the number of MATLAB processes actually computing $s(z)$ and not handling the queue. For example, the algorithm spent 227 sec to compute the pseudospectrum of matrix `triangle(300)` using five processors, four of which were actually computing $s(z)$ and the fifth one was handling the queue. Notice that for the case of $P = 1$ we used a 450MHz node. We observe that MC achieves far superior performance than SB. On the other hand, dynamic assignment starts by being superior to both for low processor counts, but its performance deteriorates and becomes similar to MC when P increases. The situation is expected to deteriorate for QUEUE as the number of CPUs grows due to the contention at the queue and the increased number of messages.

We conclude this discussion with two additional schemes for enhancing performance: The first is *continuation*, in which, the minimum left singular vector at z is used as starting point for the computation of $s(z')$, for values z' close to z; see [13] and [5] for a parallel scheme. Continuation can also be applied in the static and dynamic schemes applied above. In the case of MC, for instance, continuation can be used because gridpoints lying along the antidiagonals have the same color and use singular value information from $z = (x_i, y_j)$ to compute singular values at (x_{i+1}, y_{j-1}) and (x_{i-1}, y_{j+1}). The second is that we can combine static interleaving with a *distributed* queue strategy. The idea is to organize gridpoints into groups, just as in multicoloring, and assign each group to one processor. We let the gridpoints in each group correspond to a local queue, from which the corresponding processor takes its next task. When the queue empties, if some processors still have enough tasks in their queues, we could use task stealing to further balance the load.

3. BEYOND GRID

In this section we consider the parallel implementations of two recent powerful methods for the computation of pseudospectra, Inclusion-Exclusion and Pseudospectrum Descent.

	triangle(300)			fish(300)		
P	SB	MC	QUEUE	SB	MC	QUEUE
1	933	-	-	530	-	-
2	632	572	453	306	320	254
4	454	265	227	166	146	127
8	289	130	114	125	68	66
15	171	61	62	66	38	36
16	166	60	-	60	34	-

Table 5: Runtimes (sec) of GRID for static block assignment (SB), static multicolored assignment (MC) and dynamic assignment (QUEUE) for triangle(300) (cols 2-4), fish(300) (cols 5-7).

A common characteristic of both methods is that like GRID, they both offer significant large grain parallelism and thus are suitable for the programming environment offered by the Cornell Multitask Toolbox. We first briefly review these methods and then discuss their parallel features and implementation.

3.1 Pseudospectrum descent method (PsDM)

The Pseudospectrum Descent Method (PsDM) was proposed in [1] and is intimately connected with work described in [3]. The idea behind the method is to use points from an already existing pseudospectrum level curve approximated by means of a set of N points $z_k \in \Lambda_\epsilon(A), k = 1, ..., N$, to generate in parallel points of a nearby level curve, $\partial\Lambda_\delta(A)$, where $\delta < \epsilon$. The initial curve can be computed by existing parallel methods (e.g. [2, 15]). It was shown in [1] that this process can be applied repeatedly to approximate several pseudospectrum level curves. The advantage of PsDM is that it is embarrassingly parallel, since there are as many independent tasks as the number of points defining $\partial\Lambda_\epsilon$. Moreover, it adjusts to the geometric characteristics of the pseudospectrum, capturing disconnected components that present a difficulty for some domain-based methods. Figure 3 illustrates the scheme. We start with N points $z_k \in \partial\Lambda_\epsilon(A)$. At the end of a single sweep, each point z_k is corrected to a corresponding point $y_k \in \partial\Lambda_\delta(A)$, $\delta < \epsilon$. The correction is achieved by Newton iteration on function $G(x, y) = s(x, y) - \delta$. The key ingredient is that the gradient $\nabla G(x, y)$, necessary for the Newton iteration, can be easily computed, since $\nabla G(x, y) = (\Re(v^*u), \Im(v^*u))$, where v, u are the right and left singular vectors corresponding to the smallest singular value of $A - zI$, [6] and $\Re, \Im$ denote the real and imaginary parts respectively. This sweep is embarrassingly parallel: each correction can be carried out completely independently from all others. Furthermore, the sweep can be repeated so as to compute a number of contours, moving towards the spectrum of A. This repeated application constitutes PsDM.

3.2 Inclusion-Exclusion and modified grid method

At each point of the domain of interest, GRID computes $s(z)$ and uses that information in order to classify the point z as belonging to $\Lambda_\epsilon(A)$ or not. In that sense, GRID makes only "pointwise" use of the information it computes at each z. It was shown in [10], based also on work in [3], that knowledge of the minimum singular triplet $[\sigma_{\min}(zI - A), u_{\min}, v_{\min}]$

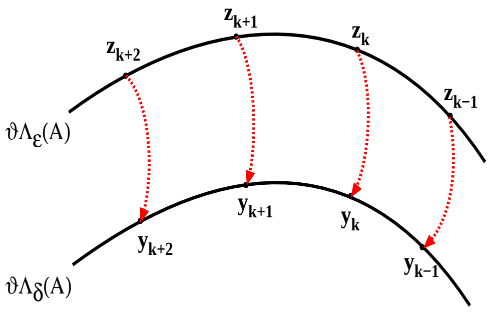

Figure 3: Computing an inner contour using PsDM.

1. Obtain an inclusion region $\hat{\Omega} \supset \Lambda_\epsilon(A)$.
2. Compute set of exclusion regions Δ_j intersecting $\hat{\Omega}$, i.e. $\hat{\Omega} \cap \Delta_j \neq \emptyset$, so that $\Lambda_\epsilon(A) \cap \Delta_j = \emptyset, j = 1 : n$ and set $\Omega = \hat{\Omega} \backslash \cup_{j=1:n} \Delta_j$.
3. Discretize Ω and call the resulting grid $\Omega_h \subset \Omega$.
4. Compute the $\sigma_{\min}$'s on Ω_h.

Table 6: General *Inclusion-Exclusion* methodology

at $z \in \mathbb{C}$ provides much more information, that can be used effectively to locate the pseudospectrum. In particular, from every z where we compute the triplet, it is possible to construct "exclusion disks" that do not intersect the pseudospectrum. The resulting algorithm was called MoG in [10] and is based on the following theorem:

THEOREM 3.1. *If* $s(z) = r > \epsilon$ *then*

$$D^\circ(z, r - \epsilon) \cap \Lambda_\epsilon(A) = \emptyset,$$

where $D^\circ(z, r - \epsilon)$ is the open disk centered at z with radius $r - \epsilon$ (refer to [14] for relevant theorems). We call this an exclusion disk for the pseudospectrum, because it provably does not intersect with it. This above theorem was used in the context of an "inclusion-exclusion" methodology, presented in Table 6, to implement the rapid and judicious pruning of the initial domain enclosing the pseudospectrum. Despite its apparent simplicity, the above theorem provides an extremely effective tool for computing pseudospectra; the method is not only faster than GRID, but as it is based on the same simple calculations, it is equally robust and basically renders GRID obsolete. In [10], the above theorem is further refined to give larger exclusion regions of the form $D^\circ(z, \alpha(r - \epsilon))$, for some factor $\alpha \geq 1$, and the theoretical issues related to the method are investigated. As has been observed in [10], the method is embarassingly parallel, but presents the designer of a parallel algorithm with many interesting challenges. We illustrate the application of the above approach, as described in the above theorem, for matrix triangle, in Figure 4. The effectiveness of the method is seen by the number of computations of $s(z)$ that it required: Only 676 compared to 2500 for GRID.

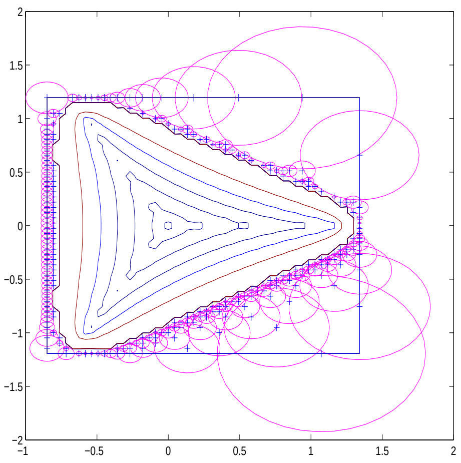

Figure 4: Using $\mathcal{IE}$ `MoG` to compute the pseudospectrum of `triangle` (32) for $\epsilon = 1e-1$.

4. PARALLEL PsDM, GRID COARSENING AND LOAD BALANCING

At its simplest form, `PsDM` offers large grain parallelism, like `GRID`. The difference is that in `PsDM`, the number of independent tasks at each step is much smaller as these correspond to the independent calculation of singular values for points approximating a one dimensional curve rather than a two dimensional domain. Therefore, as in `GRID`, the performance may suffer when we use an iterative method to compute each $s(z)$, the cost of which differs significantly from point to point. It was shown in [1] that an interleaved assignment of the N gridpoints that discretize the initial contour leads to better balanced workloads. The algorithm was implemented in MPI and tested on an 8 CPU SGI Origin 2000 system, using `ARPACK` to compute singular values and vectors [12]. However, as noted in [1], when we need to compute $\Lambda_\epsilon(A)$ for many values of ϵ, it is likely that the number of points needed to define curves that correspond to different values of ϵ that are relatively distant from each other, can also vary. Typically, for example, curves lying in the interior might need fewer points. Therefore, in [1] we proposed schemes, called `MEAN` and `GREEDY`, which have as goal to coarsen the one-dimensional grid defining the curve; it was shown that these policies lead to significant performance gains on a sequential environment, since their implementation is of low complexity and their effect could be a reduction of the total number of points that need to be computed. In a parallel environment, however, the application of such schemes is likely to alter the original load assignments, and hence lead to potential load imbalance and lower performance. In this section we examine `PsDM` when it is enhanced by the aforementioned point reduction strategies. Both schemes are based on the evolution of the smallest relative distance between adjacent gridpoints on the curve. Assuming that we wish to compute M contours starting from N points on the initial contour, they can be expressed as described in Table 7. Assuming that we have already computed l_{k-1} points $z_i^{(k-1)} \in \partial\Lambda_{\epsilon_{k-1}}$ at step $k-1$, let d_{k-1} to be the smallest distance between any two consecutive points $z_i^{(k-1)}, z_{i+1}^{(k-1)}$. *DropPolicy* selects $l_k \leq l_{k-1}$ points in accordance with one of the following rules:

```
1. Start with $N_0$ points of the initial curve $\partial\Lambda_{\epsilon_0}(A)$
2. $l_0 = N$
3. for k=1,M
      3.1 Compute $l_{k-1}$ points on $\partial\Lambda_{\epsilon_k}(A)$
      3.2 Apply DropPolicy
      3.3 Keep $l_k \leq l_{k-1}$ points of $\partial\Lambda_{\epsilon_k}(A)$
4. end
```

Table 7: `PsDM` incorporating grid coarsening.

	NR		MEAN		GREEDY	
P	SB	MC	SB	MC	SB	MC
1	380	-	350	-	283	-
2	188	195	169	170	143	142
4	99	98	75	89	80	74
8	50	48	31	44	54	39
16	23	21	18	24	22	22

Table 8: Runtimes (sec) for parallel `PsDM` for `triangle(300)`. i) no grid coarsening (cols. 2, 3) ii) MEAN point reduction scheme (cols. 4, 5) and iii) GREEDY point reduction (cols. 6, 7).

MEAN: If $(|z_i^{(k)} - z_{i+1}^{(k)}| + |z_{i+1}^{(k)} - z_{i+2}^{(k)}|)/2 < d_{k-1}$ then, unless $z_i^{(k)}$ has been already dropped, drop $z_{i+1}^{(k)}$ and set $l_k = l_{k-1} - 1$.

GREEDY: If $|z_i^{(k)} - z_{i+1}^{(k)}| < d_{k-1}$ then, unless $z_i^{(k)}$ has been already dropped, drop $z_{i+1}^{(k)}$ and set $l_k = l_{k-1} - 1$.

The above policies use d_{k-1}; to compute it, we need to access all distances between adjacent points. In the case of the block assignment, this can be done in a distributed fashion, computing minimum distances for each processor's share of the points, and then computing the final result from the local intermediate ones. Nevertheless, because this is a computation of low complexity, that we cannot implement on the interleaved distribution of gridpoints, we let point dropping be handled by the master process. In particular, processors send back points to the master processor who applies the point reduction policy, redistributes points to the processors and undertakes itself a segment of the points. Therefore, we have a *send - compute - gather - reduce* cycle for the master processor, and a *receive - compute - send* cycle for the remaining, slave processors. After the master marks which points remain, points could be dispatched, one by one, to the slave processors for processing on a first come first served basis. As in `GRID`, in the context of NOW and COW configurations with a non-dedicated network, we consider instead static assignment policies. Therefore, at each step, we assign $\lfloor \frac{l_k}{p} \rfloor$ points to each processor; our experiments apply both the block and the multicolor (interleaved) arrangement. Note that in this case, interleaving is trivial as it is one-dimensional.

As in the previous case of `GRID` we use 1 450 MHz node when $P = 1$ up to a full configuration of $P = 16$ 450-500 MHz nodes. Our first experiment was with matrix `triangle` of size 300. The initial contour was $\partial\Lambda_{0.1}(A)$ approximated by 72 points and `PsDM` computed 20 curves

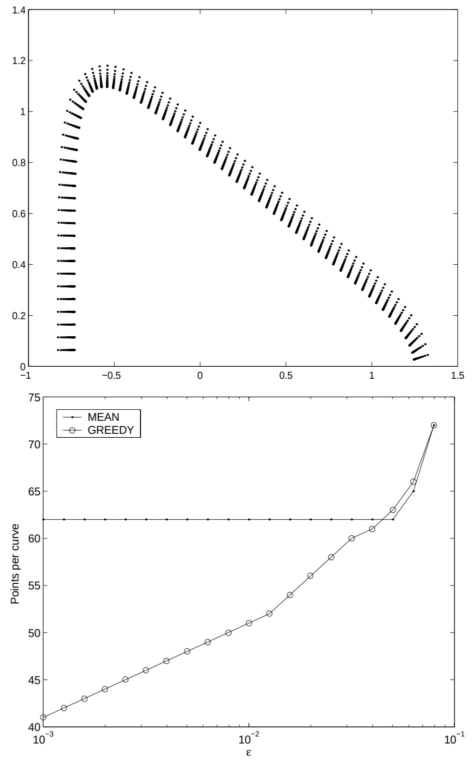

Figure 5: Top: Points computed by `PsDM` for 20 pseudospectrum curves of `triangle(300)`. Bottom: Number of points on each ϵ-pseudospectrum curve with MEAN and GREEDY reduction policies.

$\partial\Lambda_{\epsilon_k}(A)$, $\log_{10}\epsilon_k = -1.1 : -0.1 : -3$. The upper part of figure 5 illustrates the resulting discretized contours, using no point reduction strategy, and the lower part the decreasing number of points per curve, achieved by the two reduction strategies. Table 8 depicts the runtimes for i) no point reduction policy (NR), ii) MEAN and iii) GREEDY point reduction policies. We first observe the reduced runtimes for point reduction relative to the "no drop" policy as well as the overall satisfactory speedups. On the other hand, in contrast to `GRID` and results in [1], the interleaved assignment even though it outperforms the static assignment when we consider the GREEDY dropping policy, demonstrates relatively similar performance to SB in the case of MEAN. An explanation for this is that the performance of the underlying iterative solver, `lansvd`, is far smoother at the curve points than the `ARPACK` code used in [1].

In the second experiment we used matrix `pores_2` of size 1224 (9613 non-zero elements) from the Harwell-Boeing collection, scaled by 10^{-7}. The initial contour was drawn for $\epsilon = 0.1$ and was approximated by 64 points, from which we applied `PsDM` to compute 20 curves, corresponding to values of ϵ such that $\log_{10}\epsilon_k = -1.1 : -0.1 : -3$. Runtimes are shown in Table 9.

We observe again the beneficial effect of grid coarsening in a parallel environment. We should note that we present performance in terms of time rather than speedup because

	NR		MEAN		GREEDY	
P	SB	MC	SB	MC	SB	MC
1	7979	-	4751	-	3321	-
2	3892	4137	2358	2364	1694	1761
4	2035	2041	1177	1180	1070	857
8	1009	1011	585	584	966	442
16	448	450	660	349	650	269

Table 9: Parallel `PsDM` on 10^{-7}`pores_2`. Runtimes (sec) with i) no point reduction (cols. 2,3) ii) MEAN point reduction scheme (cols. 4,5) and iii) GREEDY point reduction scheme (cols. 6,7).

in a heterogeneous system , "speedup" can have a different value, depending which processor is used to obtain T_1, the "single CPU runtime". We observe again a behavior similar to the previous experiment, that suggests choosing MC over SB, especially when many processors are employed.

5. PARALLEL MoG AND LOAD BALANCING

As discussed above, for any $z \in \mathbb{C}^n$, the computation of $s(z)$ results in one of the following two actions: Either $s(z) \leq \epsilon$, therefore $z \in \Lambda_\epsilon(A)$; otherwise, all gridpoints $\hat{z} \in \hat{Z} := \{\hat{z} | \hat{z} \in \Omega_h, |\hat{z} - z| < s(z) - \epsilon\}$ are excluded and need not be considered any further. Therefore, if we apply parallelism to speed up the computation in the same manner we did with `GRID`, allocating to processors gridpoints z and letting them compute the corresponding $s(z)$ values, every computation is likely to exclude points that lie within an exclusion disk centered at z. It is worth noting, and this is one more point making `MoG` very interesting, that the actual exclusions depend on the order in which the gridpoints are swept. Consequently, if we adopt a static allocation policy such as block partitioning, a single computation within one processor could exclude many points allocated to the same processor. In some respects, we have an action reverse from locality: If we compute $s(z)$ then it is likely that we will not have to compute $s(z+\Delta z)$ at all! This means that any static policy that assigns neighboring points to the same processor is likely to lead to load imbalance, since those points will be excluded if they happen to lie outside the pseudospectrum. For this reason, we consider here an implementation in which processes have access to a common area that maintains up-to-date information about the status of each point. In particular, points are classified in one of three categories: *Active*, in the sense that their $s(z)$ still needs to be computed, *inactive*, in the sense that they have been excluded, and *fixed*, in the sense that $s(z) \leq \epsilon$. The pseudospectrum contour is plotted based on the values of the fixed points. As a result, we opt for a dynamic policy.

The parallel algorithm is designed based on the following steps. We start with an initial discretization Ω_h of a rectangular region Ω that is guaranteed to contain the pseudospectrum. Ω_h is defined using its two antidiagonal vertices α, γ, and the number of its rows and columns. Initially, we define the set of active nodes $\mathcal{N}$ as the set of all mesh nodes. During the course of the algorithm, this set is updated to contain only those mesh nodes that have not been excluded in previous steps. The algorithm proceeds in three phases.

In the first phase we aim to find the smallest bounding box $\mathcal{B}$ of $\partial\Lambda_\epsilon(A)$, the perimeter of which does not contain any active points. The second phase performs a collapse of the perimeter of $\mathcal{B}$ towards $\partial\Lambda_\epsilon(A)$. At the end of this phase no further exclusion can be performed. The set of active nodes $\mathcal{N}$ contains only mesh nodes $z_{in} \in \Lambda_\epsilon(A)$. Finally, in the third phase all remaining $s(z),\ z \in \mathcal{N}$ are computed.

We provide a detailed description, starting with the definition of neighboring points in the set of active nodes $\mathcal{N}$.

Definition 5.1. *Define as $\mathcal{NB_N}(z)$ the set of all neighboring mesh nodes $z_\mathcal{N} \in \mathcal{N}$ of node $z \in \mathcal{N}$ (the diagonal directions are not considered).*

Phase 1 (Exclusion) : Initialize the set $\mathcal{B}$ to contain the points at the perimeter of Ω_h. Select P of these points, where P is the number of available processors, that are equidistributed along $\mathcal{B}$. Assign each one, say z_k, to an available processor and compute the corresponding $s(z_k)$. The master processor gathers all $s(z_k)$ and computes the set $\mathcal{E}$, of points rendered inactive because of the exclusion:

$$\mathcal{E} = \{z \in \mathcal{N} :\ |z - z_k| < s(z_k) - \epsilon\} \tag{3}$$

The master also updates the set of active points:

$$\mathcal{N} = \mathcal{N} \setminus \mathcal{E} \tag{4}$$

Redefine $\mathcal{B}$: $\alpha_B = (\min(\Re(\mathcal{N})), \max(\Im(\mathcal{N})))$, $\gamma_B = (\max\{\Re(\mathcal{N})\}, \min\{\Im(\mathcal{N})\})$. Repeat the above until $\mathcal{B}$ contains no active nodes.

Phase 2 (Collapse) : Any node z that excludes only itself is marked as "fixed". Define $\mathcal{F}$ as those points that are currently marked as such. Initially, $\mathcal{F} = \emptyset$. We also define the hull $\mathcal{H}$ of $\mathcal{N}$ as

$$\mathcal{H} = \{z \in \mathcal{N} : (|\mathcal{NB_N}(z)| \leq 3) \wedge z \notin \mathcal{F}\} \tag{5}$$

Initially $\mathcal{H} = \mathcal{B}$. Distribute each one of P points $z_k \in \mathcal{H}$, one to each of the processors and compute the corresponding z_k. The master processor gathers all $s(z_k)$, computes the set $\mathcal{E}$ of points rendered inactive because of exclusion as in Formula (3) above and updates the set of active points as in formula (4) above. The processor also updates $\mathcal{H}$ and $\mathcal{F} : |\mathcal{H}| = 0$.

Phase 3 (Grid) : Compute all remaining $s(z),\ z \in \mathcal{N}$

Some explanations are in order: The motivation behind the first phase of the algorithm is to attempt large exclusions that have small overlap. Since the exclusion disks are not known a priori, the heuristic is that gridpoints that lie on the outer convex hull enclosing the set of active points will offer a better chance for larger exclusions. The second phase of the algorithm is designed to sieve through those points that should remain active from those that should be rendered inactive but have not been so in phase 1 because of their special location; e.g. phase 1 would not exclude points that lie deep inside some concave region of the pseudospectrum. An example of this, for matrix `grcar` (1000), is shown in Figure 6, where we labeled with dots ('·') those points that remain active after the end of phase 1 and with circle ('o') points that remain active after phase 2 is completed. A master node takes on the responsibility of gathering the singular values and distributing the points, on the bounding box $\mathcal{B}$ (Phase 1) and on the hull $\mathcal{H}$ (Phase 2). The administrative task assigned to the master node (exclusions and construction of the hull) is light; it is thus feasible to also use the master node as a singular value calculator. In conclusion, phases 1 and 2 adopt a synchronized queue model: The master waits for all processes, including itself, to finish computing $s(z)$ for their currently allocated point and only then proceeds to dispatch the next set.

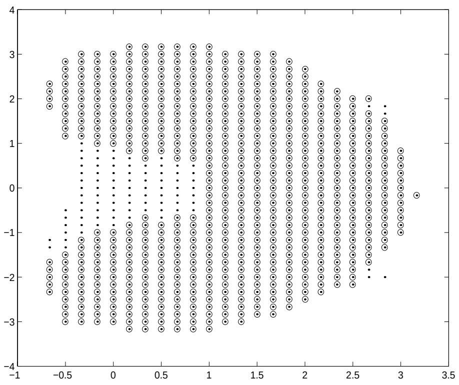

Figure 6: $\mathcal{IE}$ `MoG` outcome of Phases 1 (dots '·') and 2 (circle 'o')for matrix `grcar` (1000) on 50×50 grid.

Phase 3 is similar to the parallel implementation of `GRID`, the only difference being that the points where we need to compute $s(z)$ are known to satisfy $s(z) \leq \epsilon$. For this reason, we experimented with the same policies described in Section 2.1, namely static assignments (SB and MC), as well as dynamic allocation (QUEUE). As in Section 2.1, for the latter case we used an additional MATLAB process as queue manager.

The first test matrix was `grcar(1000)` on a 50×50 grid for $\epsilon = 0.1$. Table 10 depicts the performance of the method for all assignment policies. In each row, we write the number of processors used and then the times taken by phases 1 and 2 which are common to all allocations. We then list the time taken by phase 3 as well as the total time for each allocation. Remember that when using QUEUE in phase 3, P available processors correspond to $P-1$ working processes and 1 queue monitoring process.

Commenting on the results, we first note that all these numbers represent an extreme improvement over the use of `GRID`, where 50×50 computations of the minimum singular value of a matrix of size 1000 would have been required, necessitating more than 2.5 hours of calculation on one processor compared to less than 1 hour for one processor and 4 minutes for 16 processors using $\mathcal{IE}$ `MoG`. We next note the reduction in time as the number of processors increases. Phases 1 and 2 take the least time but also lead to smaller speedups than phase 3, which is embarassingly parallel, like `GRID`. The second test was with matrix `pores_2`. Results are shown in Table 11. We note that the performance of QUEUE is slightly

P	Phase1	Phase 2	SB		MC		QUEUE	
			Ph. 3	sum	Ph. 3	sum	Ph. 3	sum
1	150	326	1434	1910	-	-	-	-
2	101	181	707	989	703	985	669	951
4	62	90	369	521	361	513	338	490
8	120	25	193	338	178	323	167	312
15	43	17	112	172	100	160	90	150
16	115	16	110	239	96	227	-	-

Table 10: Runtimes (sec) **for parallel $\mathcal{IE}$ `MoG` for `grcar(1000)` using i) block (SB) ii) multicolor (MC) and iii) QUEUE.** $\Omega = [-4, 4] \times [-4, 4]$ **using** 50×50 **grid and** $\epsilon = 0.1$.

P	Phase1	Phase 2	SB		MC		QUEUE	
			Ph. 3	sum	Ph. 3	sum	Ph. 3	sum
1	794	1324	4756	6874	-	-	-	-
2	574	644	2290	3508	2436	3654	2132	3350
4	271	347	1216	1834	1205	1823	1072	1690
8	183	158	540	881	576	917	523	864
15	102	93	299	494	300	495	280	475
16	81	104	275	460	278	463	-	-

Table 11: Runtimes (sec) **for `MoG` on matrix** 10^{-7}`pores_2`. $\Omega = [-2.5, 1.5] \times [-1.5, 1.5]$ **using a** 50×50 **grid for** $\epsilon = 0.1$

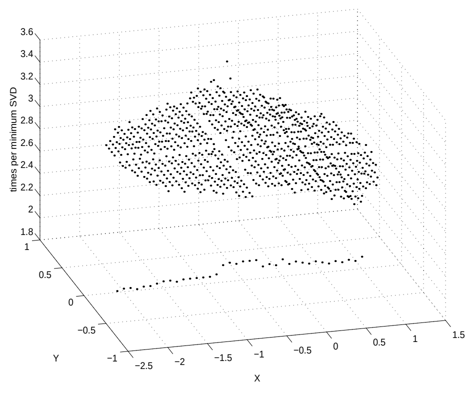

Figure 7: Times to compute $s(z)$ at each one of the points remaining in phase 3 of `MoG`.

better than that of the policies using static allocation. We note, however, that these experiments were conducted under very low load conditions for the network. Since the static allocation regimes give similar performance with the QUEUE, we opt for the former. Regarding the static schemes we note that the advantage of MC over SB is far inferior to the one observed in `GRID`. This is because the runtimes of `lansvd` at the active points demonstrate very little variation. This is illustrated in Figure 7, where we plot the time needed to compute $s(z)$ with `lansvd` at each one of the points remaining in Phase 3 for matrix `pores_2`. The only cases where times are significantly different is for those points lying on the real axis.

6. CONCLUSIONS

Pseudospectrum calculations are enormously time consuming and represent an important computational challenge that requires numerical and computational advances. We have described initial designs of parallel state-of-the-art algorithms and have demonstrated that their use significantly improves the performance; we also highlighted the importance of proper data assignment and load balancing for their effective implementation. Given the large grain of the independent computations, we expect that the patterns observed in this paper will remain valid as long as the number of processors remains moderate compared to the number of gridpoints. We also note that at this stage of our work, we concentrated on the parallelism available at the level of the calculation instead of the way this parallelism is serviced by the hardware platform, for which more emphasis must be paid to load balancing at nodes of differing capacities. We finally note that this work is part of our efforts to build algorithms and software for computing pseudospectra that can be used for applications originating in Chemical Engineering.

Acknowledgement

We thank our colleagues from the Parallel Systems group and the Computer Center of the Computer Engineering and Informatics Department for their support in these experiments.

7. REFERENCES

[1] C. Bekas, and E. Gallopoulos. Parallel Computation of Pseudospectra by fast Descent. Presented at International Workshop on Parallel Matrix Algorithms and Applications (PMAA'00), Neuchatel, Switzerland. In URL `www.ceid.upatras.gr/scgroup/Reports`. Submitted for publication, Nov. 2000.

[2] C. Bekas, and E. Gallopoulos. Cobra: Parallel path following for computing the matrix pseudospectrum. *Parallel Computing* (to appear). Available at `www.hpclab.ceid.upatras.gr` in `/faculty/stratis/Papers`.

[3] C. Bekas, and I. Koutis. Parallel Algorithms for the Computation of Pseudospectra. Master's thesis, Computer Engineering and Informatics Department, University of Patras, June 1998. In Greek.

[4] M.W. Berry. Large scale singular value decomposition. *Int. J. Supercomp. Appl. 6* (1992), 13–49.

[5] T. Braconnier. Fvpspack: A Fortran and PVM Package to Compute the Field of Values and Pseudospectra of Large Matrices. Numerical Analysis Report No. 293, Manchester Centre for Computational Mathematics, Manchester, England, Aug. 1996.

[6] M. Brühl. A curve tracing algorithm for computing the pseudospectrum. *BIT 33*, 3 (1996), 441–445.

[7] I.S. Duff, R.G. Grimes, and J.G. Lewis. User's guide for the Harwell-Boeing sparse matrix collection (Release I). Tech. Rep. TR/PA/92/86, CERFACS, Toulouse Cedex, France, Oct. 1992.

[8] V. Frayssé, L. Giraud, and V. Toumazou. Parallel computation of spectral portraits on the Meiko CS2. In *LNCS: High-Performance Computing and Networking* (1996), H. L. *et al.*, Ed., vol. 1067, Springer-Verlag, pp. 312–318.

[9] N.J. Higham. The Test Matrix Toolbox for MATLAB (version 3.0). Tech. Rep. 276, Manchester Centre for Computational Mathematics, Sept. 1995.

[10] I. Koutis, and E. Gallopoulos. Exclusion regions and fast estimation of pseudospectra. Submitted, 2000.

[11] R. M. Larsen. *SVD of sparse or structured matrices using Lanczos bidiagonalization with partial reorthogonalization.* PhD thesis, University of Aarhus, October 1998.

[12] R. Lehoucq, D.C. Sorensen, and C. Yang. *Arpack User's Guide: Solution of Large-Scale Eigenvalue Problems With Implicitly Restarted Arnoldi Methods.* SIAM, Philadelphia, 1998.

[13] S.H. Lui. Computation of pseudospectra with continuation. *SIAM J. Sci. Comput. 18*, 2 (1997), 565–573.

[14] L.N. Trefethen M. Embree. Generalizing eigenvalue theorems to pseudospectra theorems. Tech. Rep. 00/12, Oxford University Computing Laboratory, Numerical Analysis Group, June 2000.

[15] D. Mezher, and B. Philippe. Parallel computation of the pseudospectrum of large matrices, Nov. 2000. Submitted for publication.

[16] Pseudospectra Gateway. At the Oxford University site http://web.comlab.ox.ac.uk/projects/pseudospectra.

[17] L.N. Trefethen. Pseudospectra of matrices. In *Numerical Analysis 1991, Proc. 14th Dundee Conf.*, D. Griffiths and G. Watson, Eds. Essex, UK: Longman Sci. and Tech., 1991, pp. 234–266.

[18] L.N. Trefethen. Computation of pseudospectra. In *Acta Numerica 1999*, vol. 8. Cambridge University Press, 1999, pp. 247–295.

[19] L.N. Trefethen, A.E. Trefethen, S.C. Reddy, and T.A. Driscoll. Hydrodynamic stability without eigenvalues. *Science 261* (July 1993), 578–584.

[20] J.A. Zollweg, and A. Verma. The Cornell Multitask Toolbox. Directory `Services/Software/CMTM` at URL http://www.tc.cornell.edu.