Parallel Numerical Algorithms

Chapter 12 – Eigenvalue Problems

Prof. Michael T. Heath

Department of Computer Science
University of Illinois at Urbana-Champaign

CS 554 / CSE 512

Outline

Eigenvalues and Eigenvectors

- Given $n \times n$ matrix $\boldsymbol{A}$, find scalar λ and nonzero vector $\boldsymbol{x}$ such that

$$\boldsymbol{A}\boldsymbol{x} = \lambda \boldsymbol{x}$$

- λ is *eigenvalue* and $\boldsymbol{x}$ is corresponding *eigenvector*
- $\boldsymbol{A}$ always has n eigenvalues, but they may be neither real nor distinct
- May need to compute only one or few eigenvalues, or all n eigenvalues
- May or may not need corresponding eigenvectors

Problem Transformations

- *Shift*: for scalar σ, eigenvalues of $\boldsymbol{A} - \sigma \boldsymbol{I}$ are eigenvalues of $\boldsymbol{A}$ shifted by σ, $\lambda_i - \sigma$
- *Inversion*: for nonsingular $\boldsymbol{A}$, eigenvalues of $\boldsymbol{A}^{-1}$ are reciprocals of eigenvalues of $\boldsymbol{A}$, $1/\lambda_i$
- *Powers*: for integer $k > 0$, eigenvalues of $\boldsymbol{A}^k$ are kth powers of eigenvalues of $\boldsymbol{A}$, λ_i^k
- *Polynomial*: for polynomial $p(t)$, eigenvalues of $p(\boldsymbol{A})$ are values of p evaluated at eigenvalues of $\boldsymbol{A}$, $p(\lambda_i)$

Similarity Transformations

- B is *similar* to A if there is nonsingular T such that

$$B = T^{-1}AT$$

- Then

$$By = \lambda y \;\Rightarrow\; T^{-1}ATy = \lambda y \;\Rightarrow\; A(Ty) = \lambda(Ty)$$

  so A and B have same eigenvalues, and if y is eigenvector of B, then $x = Ty$ is eigenvector of A

- Similarity transformations preserve eigenvalues, and eigenvectors are easily recovered

Similarity Transformations

Forms attainable by similarity transformation

A	T	B
distinct eigenvalues	nonsingular	diagonal
real symmetric	orthogonal	real diagonal
complex Hermitian	unitary	real diagonal
normal	unitary	diagonal
arbitrary real	orthogonal	real block triangular (real Schur)
arbitrary	unitary	upper triangular (Schur)
arbitrary	nonsingular	almost diagonal (Jordan)

Preliminary Reduction

- Eigenvalues easier to compute if matrix first reduced to simpler form by similarity transformation
- Diagonal or triangular most desirable, but cannot always be reached in finite number of steps
- Preliminary reduction usually to tridiagonal form (for symmetric matrix) or Hessenberg form (for nonsymmetric matrix)
- Preliminary reduction usually done by orthogonal transformations, using algorithms similar to QR factorization, but transformations must be applied from both sides to maintain similarity

Parallel Algorithms for Eigenvalues

- Algorithms for computing eigenvalues and eigenvectors employ basic operations such as
  - vector updates (`saxpy`)
  - inner products
  - matrix-vector and matrix-matrix multiplication
  - solution of triangular systems
  - orthogonal (QR) factorization
- In many cases, parallel implementations will be based on parallel algorithms we have already seen for these basic operations, although there will sometimes be new sources of parallelism

Power Iteration

- Simplest method for computing one eigenvalue-eigenvector pair is *power iteration*, which in effect takes successively higher powers of matrix times initial starting vector

- $\boldsymbol{x}_0 =$ arbitrary nonzero vector
  for $k = 1, 2, \ldots$
  $\quad \boldsymbol{y}_k = \boldsymbol{A}\boldsymbol{x}_{k-1}$
  $\quad \boldsymbol{x}_k = \boldsymbol{y}_k / \|\boldsymbol{y}_k\|_\infty$
  end

- If $\boldsymbol{A}$ has unique eigenvalue λ_1 of maximum modulus, then power iteration converges to eigenvector corresponding to dominant eigenvalue

Power Iteration

- Convergence rate of power iteration depends on ratio $|\lambda_2/\lambda_1|$, where λ_2 is eigenvalue having second largest modulus
- It may be possible to choose shift, $\boldsymbol{A} - \sigma \boldsymbol{I}$, so that ratio is more favorable and yields more rapid convergence
- Shift must then be added to result to obtain eigenvalue of original matrix

Parallel Power Iteration

- Power iteration requires repeated matrix-vector products, which are easily implemented in parallel for dense or sparse matrix, as we have seen
- Additional communication may be required for normalization, shifts, convergence test, etc.
- Though easily parallelized, power iteration is often too slow to be useful in this form

Inverse Iteration

- *Inverse iteration* is power iteration applied to $\boldsymbol{A}^{-1}$, which converges to eigenvector corresponding to dominant eigenvalue of $\boldsymbol{A}^{-1}$, which is reciprocal of *smallest* eigenvalue of $\boldsymbol{A}$
- Inverse of $\boldsymbol{A}$ is not computed explicitly, but only factorization of $\boldsymbol{A}$ (and only once) to solve system of linear equations at each iteration
- $\boldsymbol{x}_0 =$ arbitrary nonzero vector
  for $k = 1, 2, \ldots$
  Solve $\boldsymbol{A}\boldsymbol{y}_k = \boldsymbol{x}_{k-1}$ for $\boldsymbol{y}_k$
  $\boldsymbol{x}_k = \boldsymbol{y}_k / \|\boldsymbol{y}_k\|_\infty$
  end

Inverse Iteration

- Shifting strategy can greatly accelerate convergence
- Inverse iteration is especially useful for computing eigenvector corresponding to approximate eigenvalue, since it converges rapidly when approximate eigenvalue is used as shift
- Inverse iteration also useful for computing eigenvalue closest to any given value β, since if β is used as shift, then desired eigenvalue corresponds to smallest eigenvalue of shifted matrix

Parallel Inverse Iteration

- Inverse iteration requires initial factorization of matrix A and solution of triangular systems at each iteration, so it appears to be much less amenable to efficient parallel implementation than power iteration
- However, inverse iteration is often used to compute eigenvector in situations where
  - approximate eigenvalue is already known, so using it as shift yields very rapid convergence
  - matrix has previously been reduced to simpler form (e.g., tridiagonal) for which linear system is easy to solve

Simultaneous Iteration

- To compute many eigenvalue-eigenvector pairs, could apply power iteration to several starting vectors simultaneously, giving *simultaneous iteration*

  $\boldsymbol{X}_0 =$ arbitrary $n \times q$ matrix of rank q
  for $k = 1, 2, \ldots$
  $\quad \boldsymbol{X}_k = \boldsymbol{A}\boldsymbol{X}_{k-1}$
  end

- span($\boldsymbol{X}_k$) converges to invariant subspace determined by q largest eigenvalues of $\boldsymbol{A}$, provided $|\lambda_q| > |\lambda_{q+1}|$

- Normalization is needed at each iteration, and columns of $\boldsymbol{X}_k$ become increasingly ill-conditioned basis for span($\boldsymbol{X}_k$)

Orthogonal Iteration

- Both issues addressed by computing QR factorization at each iteration, giving *orthogonal iteration*

  $\boldsymbol{X}_0 =$ arbitrary $n \times q$ matrix of rank q
  for $k = 1, 2, \ldots$
  Compute reduced QR factorization
  $\hat{\boldsymbol{Q}}_k \boldsymbol{R}_k = \boldsymbol{X}_{k-1}$
  $\boldsymbol{X}_k = \boldsymbol{A}\hat{\boldsymbol{Q}}_k$
  end

- Converges to block triangular form, and blocks are triangular where moduli of consecutive eigenvalues are distinct

Parallel Orthogonal Iteration

- Orthogonal iteration requires matrix-matrix multiplication and QR factorization at each iteration, both of which we know how to implement in parallel with reasonable efficiency

QR Iteration

- If we take $\boldsymbol{X}_0 = \boldsymbol{I}$, then orthogonal iteration should produce *all* eigenvalues and eigenvectors of $\boldsymbol{A}$
- Orthogonal iteration can be reorganized to avoid explicit formation and factorization of matrices $\boldsymbol{X}_k$
- Instead, sequence of unitarily similar matrices is generated by computing QR factorization at each iteration and then forming reverse product, giving *QR iteration*

$\boldsymbol{A}_0 = \boldsymbol{A}$
for $k = 1, 2, \ldots$
    Compute QR factorization
        $\boldsymbol{Q}_k \boldsymbol{R}_k = \boldsymbol{A}_{k-1}$
    $\boldsymbol{A}_k = \boldsymbol{R}_k \boldsymbol{Q}_k$
end

QR Iteration

- In simple form just given, each iteration of QR method requires $\Theta(n^3)$ work
- Work per iteration is reduced to $\Theta(n^2)$ if matrix is in Hessenberg form, or $\Theta(n)$ if symmetric matrix is in tridiagonal form
- Preliminary reduction is usually done by Householder or Givens transformations
- In addition, number of iterations required is reduced by preliminary reduction of matrix
- Convergence rate also enhanced by judicious choice of shifts

Parallel QR Iteration

- Preliminary reduction can be implemented efficiently in parallel, using algorithms analogous to parallel QR factorization for dense matrix
- But subsequent QR iteration for reduced matrix is inherently essentially serial, and yields little parallel speedup for this portion of algorithm
- This may not be of great concern if iterative phase is relatively small portion of total time, but it does limit efficiency and scalability

Krylov Subspace Methods

- *Krylov subspace* methods reduce matrix to Hessenberg (or tridiagonal) form using only matrix-vector multiplication
- For arbitrary starting vector x_0, if

$$\boldsymbol{K}_k = \begin{bmatrix} \boldsymbol{x}_0 & \boldsymbol{A}\boldsymbol{x}_0 & \cdots & \boldsymbol{A}^{k-1}\boldsymbol{x}_0 \end{bmatrix}$$

then

$$\boldsymbol{K}_n^{-1}\boldsymbol{A}\boldsymbol{K}_n = \boldsymbol{C}_n$$

where C_n is upper Hessenberg (in fact, companion matrix)

Krylov Subspace Methods

- To obtain better conditioned basis for span($\boldsymbol{K}_n$), compute QR factorization

$$\boldsymbol{Q}_n \boldsymbol{R}_n = \boldsymbol{K}_n$$

so that

$$\boldsymbol{Q}_n^H \boldsymbol{A} \boldsymbol{Q}_n = \boldsymbol{R}_n \boldsymbol{C}_n \boldsymbol{R}_n^{-1} \equiv \boldsymbol{H}$$

with $\boldsymbol{H}$ upper Hessenberg

Krylov Subspace Methods

- Equating kth columns on each side of equation $\boldsymbol{AQ}_n = \boldsymbol{Q}_n\boldsymbol{H}$ yields recurrence

$$\boldsymbol{A}\boldsymbol{q}_k = h_{1k}\boldsymbol{q}_1 + \cdots + h_{kk}\boldsymbol{q}_k + h_{k+1,k}\boldsymbol{q}_{k+1}$$

  relating $\boldsymbol{q}_{k+1}$ to preceding vectors $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_k$

- Premultiplying by $\boldsymbol{q}_j^H$ and using orthonormality,

$$h_{jk} = \boldsymbol{q}_j^H \boldsymbol{A}\boldsymbol{q}_k, \quad j = 1, \ldots, k$$

- These relationships yield *Arnoldi iteration*, which produces upper Hessenberg matrix column by column using only matrix-vector multiplication by $\boldsymbol{A}$ and inner products of vectors

Arnoldi Iteration

$\boldsymbol{x}_0 =$ arbitrary nonzero starting vector
$\boldsymbol{q}_1 = \boldsymbol{x}_0 / \|\boldsymbol{x}_0\|_2$
for $k = 1, 2, \ldots$
    $\boldsymbol{u}_k = \boldsymbol{A}\boldsymbol{q}_k$
    for $j = 1$ **to** k
        $h_{jk} = \boldsymbol{q}_j^H \boldsymbol{u}_k$
        $\boldsymbol{u}_k = \boldsymbol{u}_k - h_{jk}\boldsymbol{q}_j$
    end
    $h_{k+1,k} = \|\boldsymbol{u}_k\|_2$
    if $h_{k+1,k} = 0$ **then** stop
    $\boldsymbol{q}_{k+1} = \boldsymbol{u}_k / h_{k+1,k}$
end

Arnoldi Iteration

- If

$$\boldsymbol{Q}_k = \begin{bmatrix} \boldsymbol{q}_1 & \cdots & \boldsymbol{q}_k \end{bmatrix},$$

  then

$$\boldsymbol{H}_k = \boldsymbol{Q}_k^H \boldsymbol{A} \boldsymbol{Q}_k$$

  is upper Hessenberg matrix

- Eigenvalues of $\boldsymbol{H}_k$, called *Ritz values*, are approximate eigenvalues of $\boldsymbol{A}$, and *Ritz vectors* given by $\boldsymbol{Q}_k \boldsymbol{y}$, where $\boldsymbol{y}$ is eigenvector of $\boldsymbol{H}_k$, are corresponding approximate eigenvectors of $\boldsymbol{A}$

- Eigenvalues of $\boldsymbol{H}_k$ must be computed by another method, such as QR iteration, but this is easier problem if $k \ll n$

Arnoldi Iteration

- Arnoldi iteration expensive in work and storage because each new vector q_k must be orthogonalized against all previous columns of Q_k, which must be stored
- So Arnoldi process usually restarted periodically with carefully chosen starting vector
- Ritz values and vectors produced are often good approximations to eigenvalues and eigenvectors of A after relatively few iterations
- Work and storage costs drop dramatically if matrix symmetric or Hermitian, since recurrence then has only three terms and H_k is tridiagonal (so usually denoted T_k), yielding *Lanczos iteration*

Lanczos Iteration

$\boldsymbol{q}_0 = \boldsymbol{0}$
$\beta_0 = 0$
$\boldsymbol{x}_0 =$ arbitrary nonzero starting vector
$\boldsymbol{q}_1 = \boldsymbol{x}_0 / \|\boldsymbol{x}_0\|_2$
for $k = 1, 2, \ldots$
    $\boldsymbol{u}_k = \boldsymbol{A}\boldsymbol{q}_k$
    $\alpha_k = \boldsymbol{q}_k^H \boldsymbol{u}_k$
    $\boldsymbol{u}_k = \boldsymbol{u}_k - \beta_{k-1}\boldsymbol{q}_{k-1} - \alpha_k \boldsymbol{q}_k$
    $\beta_k = \|\boldsymbol{u}_k\|_2$
    if $\beta_k = 0$ **then** stop
    $\boldsymbol{q}_{k+1} = \boldsymbol{u}_k / \beta_k$
end

Lanczos Iteration

- α_k and β_k are diagonal and subdiagonal entries of symmetric tridiagonal matrix $\boldsymbol{T}_k$
- As with Arnoldi, Lanczos iteration does not produce eigenvalues and eigenvectors directly, but only tridiagonal matrix $\boldsymbol{T}_k$, whose eigenvalues and eigenvectors must be computed by another method to obtain Ritz values and vectors
- If $\beta_k = 0$, then algorithm appears to break down, but in that case invariant subspace has already been identified (i.e., Ritz values and vectors are already exact at that point)

Lanczos Iteration

- In principle, if Lanczos algorithm is run until $k = n$, resulting tridiagonal matrix is orthogonally similar to $\boldsymbol{A}$
- In practice, rounding error causes loss of orthogonality, invalidating this expectation
- Problem can be overcome by reorthogonalizing vectors as needed, but expense can be substantial
- Alternatively, can ignore problem, in which case algorithm still produces good eigenvalue approximations, but multiple copies of some eigenvalues may be generated

Krylov Subspace Methods

- Virtue of Arnoldi and Lanczos iterations is ability to produce good approximations to extreme eigenvalues for $k \ll n$
- Moreover, they require only one matrix-vector multiplication by $\boldsymbol{A}$ per step and little auxiliary storage, so are ideally suited to large sparse matrices
- If eigenvalues are needed in middle of spectrum, say near σ, then algorithm can be applied to matrix $(\boldsymbol{A} - \sigma \boldsymbol{I})^{-1}$, assuming it is practical to solve systems of form $(\boldsymbol{A} - \sigma \boldsymbol{I})\boldsymbol{x} = \boldsymbol{y}$

Parallel Krylov Subspace Methods

- Krylov subspace methods composed of
  - vector updates (`saxpy`)
  - inner products
  - matrix-vector multiplication
  - computing eigenvalues/eigenvectors of tridiagonal matrices
- Parallel implementation requires implementing each of these in parallel as before
- For early iterations, Hessenberg or tridiagonal matrices generated are too small to benefit from parallel implementation, but Ritz values and vectors need not be computed until later

Jacobi Method

- *Jacobi method* for symmetrix matrix starts with $\boldsymbol{A}_0 = \boldsymbol{A}$ and computes sequence

$$\boldsymbol{A}_{k+1} = \boldsymbol{J}_k^T \boldsymbol{A}_k \boldsymbol{J}_k$$

  where $\boldsymbol{J}_k$ is plane rotation that annihilates symmetric pair of off-diagonal entries in $\boldsymbol{A}_k$

- Plane rotations are applied repeatedly from both sides in systematic sweeps through matrix until magnitudes of all off-diagonal entries are reduced below tolerance

- Resulting diagonal matrix is orthogonally similar to original matrix, so diagonal entries are eigenvalues, and eigenvectors given by product of plane rotations

Parallel Jacobi Method

- Jacobi method, though slower than QR iteration serially, parallelizes better
- Parallel implementation of Jacobi method performs many annihilations simultaneously, at locations chosen so that rotations do not interfere with each other (analogous to parallel Givens QR factorization)
- Computations are still rather fine grained, however, so effectiveness is limited on parallel computers with unfavorable ratio of communication to computation speed

Bisection or Spectrum-Slicing

- For real symmetric matrix, can determine how many eigenvalues are less than given real number σ
- By systematically choosing various values for σ (*slicing spectrum* at σ) and monitoring resulting count, any eigenvalue can be isolated as accurately as desired
- For example, by computing inertia using $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T$ factorization of $\boldsymbol{A} - \sigma\boldsymbol{I}$ for various σ, individual eigenvalues can be isolated as accurately as desired using interval bisection technique

Sturm Sequence

- Another spectrum-slicing method for computing individual eigenvalues is based on *Sturm sequence* property of symmetric matrices
- Let $p_r(\sigma)$ denote determinant of leading principal minor of $\boldsymbol{A} - \sigma \boldsymbol{I}$ of order r
- Zeros of $p_r(\sigma)$ strictly separate those of $p_{r-1}(\sigma)$, and number of agreements in sign of successive members of sequence $p_r(\sigma)$, for $r = 1, \ldots, n$, gives number of eigenvalues of $\boldsymbol{A}$ strictly greater than σ
- Determinants $p_r(\sigma)$ easy to compute if $\boldsymbol{A}$ transformed to tridiagonal form before applying Sturm sequence technique

Parallel Multisection

- Spectrum-slicing methods can be implemented in parallel by assigning disjoint intervals of real line to different tasks, and then each task carries out bisection process using serial spectrum-slicing method for its subinterval
- Both for efficiency of spectrum-slicing technique and for storage scalability, matrix is first reduced (in parallel) to tridiagonal form, so that each task can store entire (reduced) matrix
- Load imbalance is potential problem, since different subintervals may contain different numbers of eigenvalues

Parallel Multisection

- Clustering of eigenvalues cannot be anticipated in advance, so dynamic load balancing may be required to achieve reasonable efficiency
- Eigenvectors must still be determined, usually by inverse iteration
- Since each task holds entire (tridiagonal) matrix, each can in principle carry out inverse iteration for its eigenvalues without any communication among tasks
- Orthogonality of resulting eigenvectors cannot be guaranteed, however, and thus communication is required to orthogonalize them

Divide-and-Conquer Method

- Another method for computing eigenvalues and eigenvectors of real symmetric tridiagonal matrix is based on *divide-and-conquer*
- Express symmetric tridiagonal matrix $\boldsymbol{T}$ as

$$\boldsymbol{T} = \begin{bmatrix} \boldsymbol{T}_1 & \boldsymbol{O} \\ \boldsymbol{O} & \boldsymbol{T}_2 \end{bmatrix} + \beta \boldsymbol{u}\boldsymbol{u}^T$$

- Can now compute eigenvalues and eigenvectors of smaller matrices $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$
- To relate these back to eigenvalues and eigenvectors of original matrix requires solution of *secular equation*, which can be done reliably and efficiently

Divide-and-Conquer Method

- Applying this approach recursively yields divide-and-conquer algorithm that is naturally parallel
- Parallelism in solving secular equations grows as parallelism in processing independent tridiagonal matrices shrinks, and vice versa
- Algorithm is complicated to implement and difficult questions of numerical stability, eigenvector orthogonality, and load balancing must be addressed

References

- Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, eds., *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, SIAM, 2000
- J. W. Demmel, M. T. Heath, and H. A. van der Vorst, Parallel numerical linear algebra, *Acta Numerica* 2:111-197, 1993
- P. J. Eberlein and H. Park, Efficient implementation of Jacobi algorithms and Jacobi sets on distributed memory architectures, *J. Parallel Distrib. Comput.*, 8:358-366, 1990
- G. W. Stewart, A Jacobi-like algorithm for computing the Schur decomposition of a non-Hermitian matrix, *SIAM J. Sci. Stat. Comput.* 6:853-864, 1985
- G. W. Stewart, A parallel implementation of the QR algorithm, *Parallel Comput.* 5:187-196, 1987