Eigensolvers
Eigenvalue problems

- The standard eigenvalue problem $Ax = \lambda x$
  - $A$ is a square $nxn$ matrix
  - Any vector $x$ that satisfies the equation is an eigenvector of $A$ (usually the eigenvectors are normalized after determination)
  - $\lambda$ is an eigenvalue corresponding to $x$
- $A$ has $n$ linearly independent eigenvectors
- Matrix $X$ whose columns are the eigenvectors will diagonalize $A$ in a similarity transformation $A' = X^{-1}AX$
  - $A'$ will be a diagonal matrix with the eigenvalues of $A$ in its diagonal
Basic power method

- The power method is a simple method for finding the maximum eigenvalue.
- The main idea is to obtain successive iterates from $x^{k+1} = cAx^k$
  - $c$ is a normalization constant.
  - When $k$ becomes large, $x^{k+1}$ → $v$, where $v$ is the eigenvector corresponding to the maximum eigenvalue.

(initialize $x$)

```
do k = 1, n
    x = matmul(A, x)
    x = x/maxval(x)
end do
```

- The largest eigenvalue can be obtained as the maximum value (or an inner product $v^TAv$ if L2 norm was used in the iteration) from the converged iterate.

Also other norms can be used for the normalization, e.g. the L2 norm.
Power method
Inverse power method

• To selectively compute the minimum eigenvalue an inverse power method can be applied
  
  \[
  x = 1 \\
  \text{do } k = 1, n \\
  \quad y = \text{solve}(A, x) \\
  \quad x = y / \text{norm}(y) \\
  \text{end do}
  \]

  Solution of \( Ax = y \)

  • It is beneficial to form e.g. the LU decomposition of \( A \) before the iteration for a faster solution of the linear equations

  • The smallest eigenvalue is obtained from the converged iterate as \( 1 / \lambda \approx x_{m-1} \cdot y_m \)

  • Better convergence is obtained with a proper shift, i.e. to modify the iteration to read \( (A - \sigma I)x^{k+1} = cx^k \)
Inverse shifted power method

- Better convergence for the inverse power method is obtained with a proper shift, i.e. to modify the iteration to read 
  \[(A-\sigma I)x^{k+1}=cx^k\]

- The thus found eigenvalues are also shifted, i.e. iterations converge to the eigenvector corresponding to \(|\lambda-\sigma|\)
Deflation

• Assume we have determined the smallest/largest eigenvalue with a power method

• Before we can reapply these and compute another eigenvalue, we need to “deflate” the matrix
  ▪ This means forming a matrix from which the “impact” of the largest/smallest eigenvalue has been neglected
  ▪ For example a matrix $\tilde{A} = A - \lambda_1 v_1 v_1^T$ has the eigenvalues $0, \lambda_2, \lambda_3, \ldots$

• It should be noted that the deflation methods are efficient for finding a couple of eigenvalues, for a larger number global eigensolvers should be used
Householder deflation

• An always stable deflation method is the Householder deflation, a process consisting of the following steps
  ▪ Compute the maximum eigenvalue and the corresponding eigenvector
  ▪ Obtain a Householder matrix $H$ using the eigenvector $w$ as
    \[ H = I - 2 \frac{ww^T}{w^Tw} \quad \forall \; w \neq 0 \]
  ▪ Form a matrix $G = HAH^T$
  ▪ Extract the eigenvalue at $G_{11}$
  ▪ Define $A_{n-1} = G_{2:n,2:n}$
  ▪ Repeat the process on the newly defined matrix $A_{n-1}$
The QR eigensolver

- Global semidirect method
- Construct a sequence of similar matrices \( \{A_k\}, \ k=1,\ldots,n, \) where \( A_1=A \) and \( A_n \) approaches, as \( k \) gets large, an upper triangular matrix with the eigenvalues on the diagonal

\[
T = A \\
do k = 1, n \\
\quad \text{call QR_decomposition}(T,Q,R) \\
\quad T = \text{matmul}(R,Q) \\
end do
\]

- A computationally more feasible method is obtained by starting by transforming \( A \) into the tridiagonal form or into a Hessenberg form
More global eigensolvers

- Orthogonal iteration
- Krylov subspace methods
  - Lanczos method
  - Arnoldi method
- Divide-and-conquer
Literature recommendations

Application performance
General considerations

• Select the algorithm carefully
• Choose the right machine (type) for your code
• Be aware of the scaling properties of your code and the problem you are looking at
• Use the machine properly
Understanding parallel scalability

• Parallel overhead
  – additional operations which are not present in serial calculation
  – synchronization
  – redundant computation
  – communication

• Load balance
  – distribution of workload to different cores
  – for an application to scale, all processes must do an equal amount of work

• Improving parallel performance = minimizing parallel overhead
Understanding serial performance

- Several stages are required to execute one instruction, for example:
  - Instruction fetch and decode (IF)
  - Read data (RD)
  - Execution (EX)
  - Write-back (WB).

- Independent instructions can be executed concurrently in the different pipe-line stages

- Works like an factory assembly-line
Understanding serial performance

- Memory hierarchy
  - Only the data in registers can be accessed within one CPU cycle
  - Otherwise it has to be fetched from memory

- Improving serial performance = minimizing the amount of needed CPU cycles
  - Amount of instructions
  - Avoiding empty cycles
  - Utilizing special registers

Memory hierarchy:
- L1 data cache: 64 KB
- L2 cache: 0.5 MB
- L3 cache: 2 MB
- Memory (DDR2): 12.8 GB/s

Registers:
- 16 SSE2 128-bit
- 16 64-bit
- 2x8 bytes per clock (=39 GB/s)
- 2 loads/2 stores/1 load & 1 store
- 8 bytes per clock (=19 GB/s)

Diagram of memory hierarchy.
Four easy steps to decent performance

- Utilize tuned libraries everywhere
- Employ compiler optimization
- Find suitable settings for environment variables
- Mind the I/O
Numerical libraries

- Numerical libraries usually yield significant performance improvement
- Do not reinvent the wheel but use a performance library correspond of a mathematical operation whenever applicable!
- The libraries tuned for a system or a processor outperform the generic open source implementations
Enabling compiler optimization

- Data dependencies (actual or potential) and conditionals forbid the compiler to optimize the code
- To obtain better optimization by the compiler
  - Avoid bulky loops
  - Write simple and readable structures
  - Avoid conditionals inside loops
  - Avoid function/subroutine calls from loops
  - Guide the compiler with pragmas
  - Use local constants
Enabling compiler optimization

• “An optimization flag improved the performance but changed the results”
• Find out where the error arises
• Does that piece of code take a significant portion of the execution time?
  ▪ No: optimize elsewhere on the code with the beneficial but problematic flag
  ▪ Yes: try to restructure the loop such that the result remains correct
### Compiler optimization flags

<table>
<thead>
<tr>
<th></th>
<th>PGI</th>
<th>Pathscale</th>
<th>GNU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td><code>-On</code></td>
<td><code>-On</code></td>
<td><code>-On</code></td>
</tr>
<tr>
<td></td>
<td><code>-tp arch</code></td>
<td><code>-march=arch</code></td>
<td><code>-march=arch</code></td>
</tr>
<tr>
<td><strong>Interprocedual</strong></td>
<td><code>-Mipa=fast</code></td>
<td><code>-ipa</code></td>
<td><code>-finline-functions</code></td>
</tr>
<tr>
<td></td>
<td><code>-Minline</code></td>
<td><code>-inline</code></td>
<td></td>
</tr>
<tr>
<td><strong>SSE2</strong></td>
<td><code>-Mvect=sse</code></td>
<td><code>-LNO:simd=n</code></td>
<td><code>-ftree-vectorize</code></td>
</tr>
<tr>
<td></td>
<td><code>-Mscalarsse</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Faster floating</strong></td>
<td><code>-Mfprelated</code></td>
<td><code>-ffast-math</code></td>
<td><code>-ffast-math</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>-OPT:fast_math=ON</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Quick guide for impatient people

<table>
<thead>
<tr>
<th></th>
<th>PGI</th>
<th>Pathscale</th>
<th>GNU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic</strong></td>
<td>-O2</td>
<td>-O2</td>
<td>-O2</td>
</tr>
<tr>
<td><strong>Intermediate</strong></td>
<td>-fastsse</td>
<td>-03</td>
<td>-03</td>
</tr>
<tr>
<td></td>
<td>-OPT:Ofast</td>
<td>-LNO:simd=1</td>
<td></td>
</tr>
<tr>
<td><strong>Aggressive</strong></td>
<td>-fastsse -03</td>
<td>-0fast -LNO:simd=2</td>
<td>-03</td>
</tr>
<tr>
<td></td>
<td>-Mipa=fast,inline</td>
<td>-ffast-math</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-Mfprelaxed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Efficient I/O

• I/O is always an issue - make sure that you ask only for the minimum amount of I/O
  ▪ Do you need all that checkpointing?
  ▪ Do you need all that output?

• Use parallel I/O
  ▪ Create natural partitioning of data so that it will go to disk in a way that makes sense
  ▪ The easiest way for parallel I/O with good performance is to use MPI-I/O or some high-level library (HDF5, netCDF)

• Filesystem parameters (e.g. Lustre striping) have a major role
Application optimization

1. Choose algorithms, data structures, and parallelization strategy
   - Apply compiler optimization
     - Tune for the processor
     - Optimize I/O

2. Develop code
   - Apply parallelization

3. Unoptimal, correct parallel code
   - Identify performance bottlenecks
     - No → Optimized code
     - Yes

4. Assess scalability
   - Sufficient?
     - No → Yes → Converged?
     - Yes → Measure single-core performance

5. Measure single-core performance
   - Sufficient?
     - No → Yes → Converged?
     - Yes → Measure parallel performance

6. Measure parallel performance
   - Reduce overhead from communication and load imbalance
   - Assess scalability
     - Yes → Converged?
     - No → Yes → Converged?
Further information

- If interested in code optimization, participate
  - Code optimization workshop at CSC March 22-24
    http://www.csc.fi/english/csc/courses/archive/code-opt-10
  - CSC Summer School June 12-20
    http://www.csc.fi/hpc/summerschool