Performance Evaluation of the Matlab PCT for Parallel Implementations of Nonnegative Tensor Factorization

Tabitha Samuel, Master’s Candidate
Dr. Michael W. Berry, Major Professor
What is the Parallel Computing Toolbox?

• Lets you solve computationally and data-intensive problems using MATLAB and Simulink on multicore and multiprocessor computers

• Provides support for data-parallel and task-parallel application development

• Provides high-level constructs such as distributed arrays, parallel algorithms, and message-passing functions for processing large data sets on multiple processors

• Can be integrated with MATLAB Distributed Computing Server for cluster-based applications that use any scheduler or any number of workers
Client and Worker nodes
Application areas of Parallel Computing Toolbox

• Parallel for loops
  Allows individual workers to execute individual loop iterations in parallel

• Offloading work
  Offload work to the worker sessions. This is done asynchronously

• Large Data sets
  PCT allows you to distribute that large arrays among the workers, so that each worker has only a part of that array
## Parallel Computing Toolbox Terminology

<table>
<thead>
<tr>
<th><strong>Job</strong></th>
<th>consists of all tasks that would perform parallel code execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Task</strong></td>
<td>Task is tied to the parallel function to execute</td>
</tr>
<tr>
<td><strong>Worker / Node</strong></td>
<td>Matlab sessions on other cores/ clusters to which parallel work is offloaded</td>
</tr>
<tr>
<td><strong>Client</strong></td>
<td>Matlab session where sequential code is executed and parallel job is setup</td>
</tr>
<tr>
<td><strong>Job Manager</strong></td>
<td>Setup on the scheduler and manages job – node allocation</td>
</tr>
<tr>
<td><strong>Job queue</strong></td>
<td>Sequence of jobs to be executed by the worker nodes</td>
</tr>
</tbody>
</table>
Non Negative Tensor Factorization

- Data mining techniques are commonly used for the discovery of interesting patterns.
- Study sought to identify regions (or clusters) of the earth which have similar short- or long-term characteristics.
- Earth scientists are particularly interested in patterns that reflect deviations from normal seasonal variations.
Patterns from the climate data

Global map of sea surface temperature patterns

Monthly and yearly variations of sea surface temperature patterns
Non Negative Tensor Factorization

- Eigensystem-based analysis driven by principal component analysis (PCA) and the singular value decomposition (SVD) has been used to cluster climate indices.
- Orthogonal matrix factors generated by the SVD are difficult to interpret.
- Among other data mining techniques, Nonnegative Matrix Factorization (NMF) has attracted much attention.
- In NMF, an $m \times n$ (nonnegative) mixed data matrix $X$ is approximately factored into a product of two nonnegative rank-$k$ matrices, with $k$ small compared to $m$ and $n$, $X \approx WH$.
- $W$ and $H$ can provide a physically realizable representation of the mixed data.
- Nonnegative Tensor Factorization (NTF) is a natural extension of NMF to higher dimensional data.
- In NTF, high-dimensional data, such as 3D or 4D global climate data, is factored directly and is approximated by a sum of rank-1 nonnegative tensors.
Non Negative Tensor Factorization

- The ALS approach separates the NTF problem into three semi-NMF sub problems within each iteration, i.e.

- Given **X** and **Y**, we solve for **Z** by
  \[
  \min_Z \phi(Z) = \| T_z - (X \cdot Y)Z \|_F^2
  \]

- Given **X** and **Z**, we solve for **Y** by
  \[
  \min_Y \phi(Y) = \| T_y - (X \cdot Z)Y \|_F^2
  \]

- Given **Z** and **Y**, we solve for **X** by
  \[
  \min_X \phi(X) = \| T_x - (Z \cdot Y)X \|_F^2
  \]
Non Negative Tensor Factorization

- Each data matrix, $T_x$, $T_y$, and $T_z$ are permuted and folded form of the original tensor $T$, illustrated below.

![Diagram of tensor factorization](image)
Non Negative Tensor Factorization

- Given \( A \in R^{m \times n} \geq 0 \) and \( W \in R^{m \times k} \geq 0 \), a semi-NMF problem is defined as

\[
\min_{H} \Phi(H) = \|A - WH\|_F^2 \quad \text{subject to } H \geq 0
\]

- A modified version of the Projected Gradient Descent (PGD) method is used to solve the Semi-NMF problem. It is basically adding a projection function on top of the regular gradient descent method. where the gradient is

\[
H^{(p+1)} = P_+ \left[ H^{(p)} - \alpha_p \nabla \Phi(H^{(p)}) \right]
\]

and \( P_+ \) is the projection function
Non Negative Tensor Factorization

- Only need to use two quadratic forms of $W$ and $A$, i.e. $W^TW$ and $W^TA$
- Comparing the sizes of two quadratic forms, i.e. $m \times k$ and $m \times n$ with the sizes of $W$ and $A$, i.e. $k \times m$ and $n \times m$, and knowing $k \gg n$, we can save memory required to store these matrices.

- A block operation for computing $W^TW$ and $W^TA$, where

$$W^TW = \sum_{i=1}^{p} W_i^T W_i \quad W^T A = \sum_{i=1}^{p} W_i^T A_i$$
Non Negative Tensor Factorization

• Thus, we can partition $X$, $Y$ or $Z$ in column blocks and make calls to the PGD subroutine in parallel
• When calling the PGD subroutine, only the quadratic forms $W^TW$ and $W^TA$ will be used, instead of $W$ and $A$
• The quadratic forms can also be computed locally by partitioning $W$ and $A$, and summed later
• Focus of this PILOT study:
  Parallelize the computation of $W^TA$
## Data Involved

- **6 climate based indices used**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Adjustment</th>
</tr>
</thead>
<tbody>
<tr>
<td>sst</td>
<td>sea surface temperature</td>
<td>+273.15</td>
</tr>
<tr>
<td>ndvi</td>
<td>normalized difference vegetation index</td>
<td>+0.2</td>
</tr>
<tr>
<td>tem</td>
<td>land surface temperature</td>
<td>+273.15</td>
</tr>
<tr>
<td>pre</td>
<td>precipitation</td>
<td></td>
</tr>
<tr>
<td>hg500</td>
<td>geopotential height (elevation) for barometric pressure of 500 millibars</td>
<td>+300</td>
</tr>
<tr>
<td>hg1000</td>
<td>geopotential height (elevation) for barometric pressure of 1000 millibars</td>
<td>+300</td>
</tr>
</tbody>
</table>
Data Involved

- **Preprocessing of data**
  - Shifts to enforce non-negativity
  - Interpolation to counter sparsity of data

- **Each parameter defined by 3-way array**
  - Dimension: 720 x 360 x 252
  - 720 - latitude
  - 360 - longitude
  - 252 - month of reading
  - Time dimension: January 1982 – December 2002 (252 months)
function WtA = computeWtA(X,Y,Z,A)
[p k] = size(X);
[q k] = size(Y);
[r k] = size(Z);
WtA = zeros(k,size(A,4));

f{1} = X;
f{2} = Y;
f{3} = Z;

% sort 'p', 'q' and 'r' in ascending order
[dim c] = sort([p q r]);
f = f(c);
A = reshape(permute(A,[c 4]),[p*q*r size(A,4)]);
M = circDotProd(f{1}, f{2});
for i = 1 : dim(3)
    temp = M .* repmat(f{3}(i,:),[size(M,1) 1]);
    WtA = WtA + temp' * A((i-1)*dim(1)
                          *dim(2)+1:i*dim(1)*dim(2),:);
end;
Approaches Used

- Parfor Loops
- Distributed Jobs with slicing A
- Load and Save with distributed jobs
Setup

- Cluster of 8 dual core processors (16 workers):
  - 4x Dual Core AMD Opteron(tm) Processor 870 (8-core total, 64-bit) Clock speed: 2 GHz
- Each approach was tested with subsets of data and finally with the entire data
- Subsets were created based on the time variable. The subsets used were 12, 24 and 180 months
- Execution time was measured using tic/toc function in Matlab
Parfor (Parallel-for) loops

- Part of the loop is executed on client, rest on the worker
- Data sent from client to workers, calculations are performed on workers, results are sent back to client where they are pieced together
- Used when
  - There are loop iterations that take a long time to execute
- Cannot be used when
  - An iteration in the loop depends on other iterations
  - No advantage when there are only simple calculations to be performed in the loop.
- Example
  
  ```
  x = 0;
  parfor i = 1:10
      x = x + i;
  end
  x
  ```
parfor i = 1 : dim(3)
    temp = M .* repmat(f{3}(i,:),[size(M,1) 1]);
    WtA = WtA + temp' * A((i-1)*dim(1)
                        *dim(2)+1:i*dim(1)*dim(2),:);
end
Sequential code executed at the client

Data sent from client to workers

Parallelizable for loop

Results collected from the workers at the client

Sequential code executed at the client
Execution Times

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>12 months</th>
<th>24 months</th>
<th>180 months</th>
<th>252 months</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without PCT</td>
<td>164.372</td>
<td>414.8772</td>
<td>3.16E+03</td>
<td>4642.5</td>
</tr>
<tr>
<td>4 workers</td>
<td>372.2438</td>
<td>853.8555</td>
<td>5500.8</td>
<td>7477.5</td>
</tr>
<tr>
<td>8 workers</td>
<td>468.7778</td>
<td>988.7908</td>
<td>6501.6</td>
<td>9542</td>
</tr>
<tr>
<td>16 workers</td>
<td>674.7118</td>
<td>1.96E+03</td>
<td>10312</td>
<td>13122</td>
</tr>
</tbody>
</table>
Programming Distributed Jobs

In a distributed job:

- Tasks do not directly communicate with each other
- A worker may run several of these tasks in succession
- All tasks perform the same function in a parallel configuration
Sequential code executed at the client

Parallelizable code

Parallelizing function called

Scheduler sends the data to workers

Results collected from the workers at the client

Sequential code execution continued at the client
Steps in running a distributed job

1. Find a job manager
2. Create a job
3. Create tasks for the job
4. Submit the job to the job queue
5. Retrieve the results
6. Destroy the job
Steps in running a distributed job

1. **Find a job manager**
   - The `findResource` function identifies available job managers and creates an object representing a job manager in your local MATLAB session.
   - **Syntax:** `jm = findResource('scheduler','type','jobmanager','Name','SamManager','lookupURL','localhost');`

2. **Create a job**
   - Create a job using the available job manager object.
   - **Syntax:** `job1 = createJob(jm)`

3. **Create tasks for the job**
   - Tasks define the functions to be evaluated by the workers during the running of the job.
   - Often, the tasks of a job are all identical.
   - **Syntax:** `createTask(jobname, functionname, # of outputs, {inputs});`
     - **Eg.** `createTask(job1, @rand, 1, {3,3});`
Steps in running a distributed job

- Submit the job to the job queue
  
  To run your job and have its tasks evaluated, you submit the job to the job queue with the submit function

  Syntax: `submit(jobname);`

- Retrieve the results
  
  The results of each task's evaluation are stored in that task object's `OutputArguments` property as a cell array

  Syntax: `results = getAllOutputArguments(jobname);`

- Destroy the job
  
  Destroy removes the job object reference object from the local session, and removes the object from the job manager memory

  Syntax: `destroy(job)`
function WtA = computeWtA(X,Y,Z,A)
.
.
.

numberOfTasks = 8;
count = dim(3)/numberOfTasks;
jm = findResource('scheduler','type',
    'jobmanager','Name','SamManager',
    'lookupURL','localhost');
job1 = createJob(jm);
set(job1,'FileDependencies',
    {'WtAparallel.m'});
job1.MinimumNumberOfWorkers = 1;
    job1.MaximumNumberOfWorkers = 4;
for i = 1 : numberOfTasks
    B = A((i-1)*count*dim(1)*dim(2)
        + 1:(i*count)*dim(1)*dim(2),:);
t(i) = createTask(job1, @WtAparallel, 1,
    {M,f{3},l, count, dim(1),dim(2),B,WtA});
end
submit(job1);
waitForState(job1);
results = getAllOutputArguments(job1);
for j = 1 : numberOfTasks
    WtA = WtA + cell2mat(results(j));
end;
destroy(job1);
function finalWtA= WtAparallel(M,f,i,count,d1,d2,B, WtA)
finalWtA= WtA;
for k = 1 : count,
    temp = M .* repmat(f(k,:),[size(M,1) 1]);
    finalWtA= finalWtA+ (temp' * B((k-1) * d1 * d2 + 1:k*d1 * d2, :));
end;
Load and Save with Distributed Jobs

- Size of matrix A is very large and linear. For entire dataset the size of A is 65318400 x 1
- In Distributed Jobs, A was being passed to the worker node every time a task was created
- This created huge overheads
- In this approach, A is saved to the local workspace of the node, prior to task creation and is reloaded only when there is a change in the value of A
- This minimizes the data overhead every time a task is created
Load and Save code execution at client

Sequential code executed at the client

Check if value of A is same as previous value of A

No

Save value of A to worker node workspace

Yes

Parallelizing function called

Scheduler sends the rest of input data to workers

Parallelizable code

Results collected from the workers at the client

Sequential code execution continued at the client
Load and save code execution at worker node

1. Check if value of A is the same as the previous value of A
   - No
   - Load new value of A into local workspace
2. Yes
   - Make value of A persistent
   - Perform parallel computation
   - Send results back to client
function WtA = computeWtA(X,Y,Z,A)
    .
    .
    A = reshape(permute(A,[c 4]).,[p*q*r size(A,4)]);
oldA = load('array_a.mat','A');
if isequal(A,oldA)
    flagA = 0;
else
    flagA = 1;
    save('array_a.mat','A');
end;
M = circDotProd(f{1}, f{2});
numberOfTasks = 8;
count = dim(3)/numberOfTasks;
jm =
    findResource('scheduler', 'type', 'jobmanager', 'Name', 'SamManager', 'lookupURL', 'local host');
job1 = createJob(jm);
set(job1,'FileDependencies',{'WtAparallel.m' 'array_a.mat'});
job1.MinNumberOfWorkers = 1;
job1.MaxNumberOfWorkers = 4;
for i = 1 : numberOfTasks
    createTask(job1, @WtAparallel, 1, {M,f{3},i, count, dim(1),dim(2), flagA,WtA});
    end
submit(job1);
waitForState(job1);
results = getAllOutputArguments(job1);
for j = 1 : numberOfTasks
    WtA = WtA + cell2mat(results(j));
end;
destroy(job1);
function finalWtA = WtAparallel(M,f,i,count,d1,d2, flagA, WtA)
if flagA==1
    persistent A;
    load('array_a.mat', 'A');
end;

finalWtA = WtA;
for k = 1 : count,
    l = i+k;
    temp = M .* repmat(f(l,:),[size(M,1) 1]);
    finalWtA = finalWtA+ (temp' * A((l-1) * d1 * d2 + 1:l*d1 * d2, :));
end;
Execution Times with Load and Save

<table>
<thead>
<tr>
<th>Time (Seconds)</th>
<th>12 months</th>
<th>24 months</th>
<th>180 months</th>
<th>252 months</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without PCT</td>
<td>164.372</td>
<td>414.8772</td>
<td>3160</td>
<td>4642.5</td>
</tr>
<tr>
<td>4 workers</td>
<td>1.31E+03</td>
<td>1.81E+03</td>
<td>8.59E+03</td>
<td>1.19E+04</td>
</tr>
<tr>
<td>8 workers</td>
<td>1.40E+03</td>
<td>2.00E+03</td>
<td>9.47E+03</td>
<td>1.24E+04</td>
</tr>
</tbody>
</table>
Overall Comparison

<table>
<thead>
<tr>
<th>Method</th>
<th>Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without PCT</td>
<td>4642.5</td>
</tr>
<tr>
<td>Parfor 4 workers</td>
<td>7447.5</td>
</tr>
<tr>
<td>Distributed Job 8 workers, 16 nodes</td>
<td>18040</td>
</tr>
<tr>
<td>Load and Save 8 workers, 8 tasks</td>
<td>12363</td>
</tr>
</tbody>
</table>
Conclusions drawn

- Parfor loops
  - By far, the best performance among the three methods used
  - The easiest to use in terms of code modification
  - Data overhead is minimal when compared to other two methods
Conclusions drawn

• Distributed jobs
  – Except for the load and save method, there is no way of controlling the workspace of worker node
  – Workers cannot share a workspace with the client, hence all input must be available to all workers
  – Cannot determine node – task allocation, it is done by the scheduler
  – Inputs have to be bound to the task at the time of creation, cannot be bound to the task at a later point of time
  – Task execution is not staggered i.e. there is no time lag between the start of tasks at worker nodes
Conclusions drawn

- **Load and Save**
  - Can bind a variable to a node’s workspace for the length of the job, this eliminates the need to send it as a part of input while creating the task
  - The “persistent” function saves the value of a variable for the duration of the job
Conclusions drawn

• Parallel Computing Toolbox – Overall
  – Parallel Computing Toolbox does not lend itself to linear inputs and relatively less complex parallel code
  – On experimental runs with more regular square matrix data there was significant improvement over sequential execution of code
    • Eg. FFT and InverseFFT code run on two matrices of size 500*500 and 900 * 900
    • Distributed Jobs with 8 worker nodes: 179.5767s
    • Sequential execution of code: 456.4300s
``Parallel Nonnegative Tensor Factorization Algorithm for Mining Global Climate Data,''
Q. Zhang, M.W. Berry, B.T. Lamb, and T. Samuel, Proceedings of the International
Conference on Computational Science (ICCS 2009) GeoComputation Workshop, Baton
Rouge, LA, Lecture Notes in Computer Science (LNCS) 5545, G. Allen et al. (Eds.),

"Scenario Discovery Using Nonnegative Tensor Factorization", Brett W. Bader, Andrey
A. Puretskiy, and Michael W. Berry, in Progress in Pattern Recognition, Image Analysis
and Applications, Proceedings of the Thirteenth Iberoamerican Congress on Pattern
Recognition, CIARP 2008, Havana, Cuba, Lecture Notes in Computer Science (LNCS)
5197, Jos'e Ruiz-Shulcloper and Walter G. Kropatsch (Eds.), Springer-Verlag, Berlin,

``Discussion Tracking in Enron Email Using PARAFAC", Brett W. Bader, Michael W.
Berry, and Murray Browne, in Survey of Text Mining II: Clustering, Classification, and
147-163.

``Nonnegative Matrix and Tensor Factorization for Discussion Tracking", Brett W. Bader,
Michael W. Berry, and Amy N. Langville, in Text Mining: Theory, Applications, and
Visualization, A. Srivastava and M. Sahami (Eds.), Chapman & Hall/CRC Press, (2010), to
appear.