Module 2
Association Rules

VPAC Education Grant Round 2
EDPNMO006/2001
David Taniar and Kate Smith
Monash University
Association Rules

- Discover what items are bought together in a transaction. Refer to as basket data.
- E.g. Customers bought Magazine also bought Sweet.
  
  \[(\text{Magazine}) \rightarrow (\text{Sweet})\]
  
  \[(\text{Coke, Twisty}) \rightarrow (\text{Chewing gum})\]

- Each transaction is associated with a Transaction ID (TID).
- Assume items in a transaction are in lexicographic order

- The term ‘association’ means connection, union or alliance. Within the data-mining environment, association relates to two variables that have associated features. It describes how often two or more variables appear together or how often they are associated together.
- In the example above, what is the frequency of item Magazine and item Sweet are bought together among all the transactions let say within a month.
- The “association” pattern can help business to makes strategic products planning.
- A transaction is associated with a transaction ID. This ID is unique among all the transactions.
- The association rules mining is also called Basket Data mining.
- Each transaction are assumed not to have repeated items and they are sorted in lexicographical order.
Let $I = \{i_1, i_2, ..., i_m\}$ be a set of items.

Let $D$ be a set of transactions, where each transaction $T$ is a set of items such that $T \subseteq I$.

$T$ contains $X$, if $X \subseteq T$. E.g. $T=(a \ b \ c \ d)$ contains $X=(a \ b)$.

An association rule is an implication of the form $X \Rightarrow Y$, where $X \subseteq I$, $Y \subseteq I$ and $X \cap Y = \emptyset$.

The rule $X \rightarrow Y$ has confidence $c$ if $c\%$ of transactions $D$ contain $X$ also contain $Y$.

The rule $X \rightarrow Y$ has support $s$ in the transaction set $D$ if $s\%$ of transactions $D$ contain $X \cup Y$.

• The two main concepts in association rules are the Support and the Confidence.
• These two parameters are provided by the users and the mining algorithm is suppose to find association rules that satisfied the parameters.
Formal Definitions (2)

- For an association rule $R: X \rightarrow Y$
- $\text{confidence}(R) = \frac{\text{support}(X \cap Y)}{\text{support}(X)}$
- where $X$ is called rule antecedent and $Y$ is called rule consequent.
Discover association rules

- Given user specified minimum confidence (minconf) and minimum support (minsup), the task is to find all association rules satisfying minconf and minsup.
- Can be decomposed into two subproblems
  1. Find all sets of items (itemsets) that have have transaction support above minsup. These itemsets are called large itemsets (itemsets).
  2. Use the itemsets to generate desired rules.
     e.g. given itemset \((A \ B \ C \ D)\), then determine if the rule \(A \ B \rightarrow C \ D\) holds by computing the ratio
     \[\text{conf} = \frac{\text{support}(A \ B \ C \ D)}{\text{support}(A \ B)}\]

- There are two main phases in mining association rules, that is, a) finding large candidates itemset and b) generate association rules.
- The large itemset are itemset that has support count equal or greater than the minimum support provided by a user.
- The discovered large itemsets are use to generate association rules that have confidence equal to or greater than the minimum confidence.
Given an large itemset with \( n \) items, there are
\[
\binom{n}{n} + \binom{n}{n-1} + \binom{n}{n-2} + \ldots + \binom{n}{1}
\]
rules.

**Example:** itemset (A B C D)

Possible rules are:
- \( ABC \rightarrow D \), \( ABD \rightarrow C \), \( ACD \rightarrow B \), \( BCD \rightarrow A \)
- \( AB \rightarrow CD \), \( AC \rightarrow BD \), \( AD \rightarrow BC \), \( CD \rightarrow AB \),
- \( BC \rightarrow AD \), \( BD \rightarrow AC \), \( A \rightarrow BCD \), \( B \rightarrow ACD \),
- \( C \rightarrow ABD \), \( D \rightarrow ABC \)

- The formula shows the number of possible association rules that could be generated from an itemset with length \( n \).
- In the example, \( n=4 \), so there are possible \( C(4, 3)+C(4,2)+C(4,1) = 14 \) rules could be generated.
Serial Apriori Algorithm (1)

- Concepts
  - all subsets of a frequent itemset are frequent
  - all supersets of an infrequent itemset are infrequent

- Above are the two main concepts of the apriori algorithm.
- The large itemset are generate level-wise incrementally. The lower level frequent itemsets are use to generate upper level large itemset in a bottom down fashion. It is also possible to generate the frequent itemset other way round from top to down.
Serial Apriori algorithm(2)

1. Discovering large item sets
   - Make multiple passes over the data.
   - In the first pass, count the support of individual items and determine which of them are large (support $\geq$ minsup).
   - Generate candidate itemsets from previous pass large itemsets.
   - Count the support for potential candidate itemsets.

2. Generate rules
   - Generate association rules for large itemsets that has confidence $\geq$ minconf.

• The most famous association rules generation algorithm is the Apriori proposed by Rakesh Argawal 93.
• The Apriori makes multi-passes over the transactions database. In each pass, it will calculate the support of current candidates itemset (a candidate itemset is itemset that is possible to become large itemset).
• Before the support counting started, a set of candidates itemset are generated from previous pass large itemset.
• The length of a candidate itemset at k-th pass is k.
• The candidates generation is actually a lattice based algorithm, where it is possible to implement in top-down (from larger to smaller length) or bottom-up (smaller to larger length) fashion.
• The diagram shows the Apriori algorithm program flow. It is clear from the diagram that the stopping condition is when there is no more large itemset left in the last database pass. When this condition is met, no more candidate itemset could be generated.
• The new set of candidates itemset at each pass were generated by performing self-join.
## Generating candidates itemset (1)

**Pass k = 1,**

Join two 1-large itemsets $p$ and $q$ if $p.item1 > q.item1$.

**Pass k > 1,**

- For all large itemsets $L_k$, join two large itemsets $p, q \in L_k$ to produce candidate itemsets $C_{k+1} = \{p.item1, p.item2, ..., p.item(k-1), q.itemn\}$ if $p.item1 = q.item1, p.item2 = q.item2, ..., p.item(k-1) = q.item(k-1), p.itemk < q.itemk$.
- For all $c \in C_{k+1}$, delete $c$ if some $k$ subset of $c$ is not in $L_k$.

- At the first pass (i.e. $k=1$), the large 1-itemsets are found.
- At pass $k > 1$, the self join is performed with the condition stated above. The join condition is generally saying, two candidates itemset can be join together if the last item in this two itemset are different and one of his last item is larger than the other. The join result will be the extend of the higher order item to the itemset of the another itemset.
- After joining, the pruning step prune any of the candidate itemset which all the length ($k-1$) subsets are not in pass $k$ large itemset.
The example shows how the self-join is performed by joining L2 itself. The join result is in C3. The candidates are further pruned, and finally only one candidate itemset is left. Candidate (bread, milk, oyster sauce) is pruned because it is a subset of (milk, oyster sauce) that is not in L2. The same reason applies to (bread, fish, oyster sauce).
# Counting candidates support (1)

- **Counting candidates support**
  - Use hash-tree (HT) for fast candidate counting.
  - Candidate itemsets $C_k$ are stored in hash-tree.

- **Structures of hash-tree**
  - A node of HT is either contains a list of candidate itemsets (a leaf node) or a hash table (an interior node).
  - **depth** - Number of level between root node and current node. Root node is defined to be at depth 0.
  - In an interior node of depth $d$, each bucket of hash table is points to another node in depth $d + 1$.

- Apriori uses a special data structure called Hash Tree for support counting.
- The hash tree is use to stores candidates itemset.
- A hash tree is a multi-branches tree, where the nodes is an interior node or a leaf node.
- An interior node contains a hash table and a leaf node contains a list of candidates itemset.
- The bucket in the hash table of an interior node is use to point to the next level (depth) child node.
Counting candidates support (2)

- Candidate itemsets are stored in the leaf node.
- Branching factors (BF) - the size of the hash table for an interior node.

- Slide above shows the structures of a hash tree.
- As you can see, the interior contains hash table that store pointer to a child node.
- The leaf node contains a list of candidates itemset.
Counting candidates support (3)

Adding candidates itemset
- All node are initially created as leaf node. We convert leaf node to interior node when the number of candidate itemsets is larger then a threshold (T) and current depth is less than maxdepth.
- New hash tree is created at the beginning of each pass (k > 1).
- Start with the root node, we hash on item d-th of candidate itemset until we reach a leaf node (in this case maxdepth ≤ k; k > 1).
- If we reached a leaf node, add the candidate itemset into the list of candidates.

- A candidate itemset is added to the hash tree by hashing on it items start from the root node. In this way a path will found to a leaf node.
- When we reached leaf node, the candidate itemset is added to the itemset list.
- Initially the candidate hash tree is created with one root node (or leaf node). After adding more candidates itemset, the root node is converted into interior node when the number of itemset is greater than a threshold T (i.e. counts).
- To convert an leaf node into an interior node, we hash on the d-th item of all the itemsets. The hashed bucket index will be pointed to a new child node, in which the itemset will be added to that child node.
- The depth of hash tree is constrained by the maximum depth. When a leaf node is at depth max depth, it will not be converted into an interior although the number of candidates itemset is more than the threshold value.
Counting candidates support (4)

- Given a hash function \( h(x) = (x \% BF) \), maxdepth=1, \( T=1 \), BF=2
- Adding (2 4 5), (1 4 7), (4 8 9), (1 6 7)

![Diagram showing the process of adding candidates and converting nodes](image)

- The slide shows example on how to add the 4 candidates itemset.
- In (1), the root node is converted into an interior node because the number of candidate is greater than \( T=1 \).
- In (2), the left leaf node is not converted to an interior node because of the maximum maxdepth constraint.
Counting candidate support (5)

- Subset function - How to find all (k+1) subsets of a transaction T?
  1. Start hashing from the root node on items in T.
  2. If it is leaf node, then find all candidate itemsets that is contained in T and update the support count. Mark this node as VISITED by using TID.
  3. At root node (interior node), we hash on every items in T using the same hash function as adding candidate itemsets.
  4. If we reached an interior node by hashing on item i in T, we apply hash function to every items after i.

- After adding the candidates to the hash tree, we start support counting.
- The subset function is to find all subset of a transaction t and update it support count.
- To do so, we traverse the hash tree from the root node until we reach to leaf node. How to do it?
- At root node we hash on items 0 through (n – k + 1) using the same hash function as when adding candidates, where n is the length of t and k is the current pass no.
- If we reach depth d by hashing on item i then we hash on items i+1 through (n-k+1).
- To avoid a transaction from updating more than once in a leaf node, we mark the updated leaf node as VISITED by using the transaction ID.
Counting candidates support (6)

Example: Given $h(x) = \begin{array}{c} 2,4,6,8 \\ 1,3,5,7,9 \end{array}$

$T=(1\ 2\ 3\ 5\ 8\ 9)$

- Above is an example of how to update candidate count operation is done. $h(x)$ is the hash function, with items 2, 4, 6, 8 go to left child and other items go to right child.
- Here we assume that at depth $d$, we hash on each items in the itemset after index $d$. The underlined items mean the items that are hashed to reach the next level.
- The number after $ht$ “:” is the support count for that candidate itemset.
- $T$ is a transaction from database and the left hand side hash tree hold the candidate itemsets.
**Candidates itemset pruning**

- Recall-delete all candidate itemsets $C_{k+1}$ that has some k-subset not in $L_k$.
- Fast pruning - use hash table.
  1. Hash each $L_k$ large itemsets into hash table.
  2. Probe hash table to find each k-subset of $c \in C_{k+1}$.
  3. Delete candidate $c$ if first k-subset is not found.

- The hash table is use for candidate itemset pruning. In the k-1-th pass, the large candidates itemset are hashed into a hash table. In the current k-th iteration, the hash table is probe to find whether a subset exist of not. If a length k-1 subset of a candidate itemset is not exist, then that candidate is discarded.
Association Rules Generation (1)

- Generates association rules
  - for all \( a \in \) large itemset
  - find all subsets \( s \) of \( a \), where \( s \neq \emptyset \)
  - output rules \( s \rightarrow (a - s) \) if \( \text{support}(a)/\text{support}(s) \% \geq \text{minconf} \)

- Rules properties
  - Given \( s' \subset s \), if confidence \( R1:s \rightarrow (a - s) < \text{minconf} \), then confidence \( R2:s' \rightarrow (a - s') \leq \text{conf}(R1) < \text{minconf} \).

- Association rules are generated based on the two main concepts of this algorithm.
- The logic is as below,
- if a rule \( R1:s=>(a-s) \) less than minimum confidence, then the confidence of the subsets \( s' \) of \( s \) must be less than the minimum confidence. Why?
  - from the definition above
  - confidence\((R1) = \text{support}(a)/\text{support}(s)\), but \( \text{support}(s') \leq \text{support}(s) \), so from the equation, confidence\((R2) \leq \text{confidence}(R1) < \text{minconf} \).
- On the other way round, we can say that in order for an association rule to has a minimum confidence, all of its subset must have minimum confidence too.
- In the implementation, we start with rule antecedent of length 1 and gradually create candidates antecedent with greater length, until the antecedent length is equal to \( k-1 \) or no more candidate antecedent could be generated.
- Example:
  - Let ABCDE is a large itemset and only three ABCD=>E, ABC=>D, and ACD=>B satisfied the minimum confidence constraint.
  - Then the candidate two items rules consequent are generated from B, D, and E, which results in BD, BE, DE. The possible rules that will only needs to consider here are ACE=>BD, ACD=>BE, and ABC=>DE
Association Rules Generation (2)

- From this, for confidence R3: \((a - c) \rightarrow c\) to hold, all rules of the form \((a - c') \rightarrow c'; c' \subset c\), must also hold. For example, if the rule \(AB \rightarrow CD\) holds, then the rules \(ABC \rightarrow D\) and \(ABD \rightarrow C\) must also hold.
Parallel Apriori Implementations

- Use shared-nothing architecture, where each of \( N \) processors has a private memory and a private disk.
- Each processor is connected by a communication network and can communicate by messages passing.
- Inter-processor communication using Message Passing Interface (MPI) primitives.

The parallel Apriori uses the shared nothing architecture.
Processors are communicate through MPI communications libraries.
The Counts Distribution parallel Apriori algorithm was implemented. This method using data parallelism concept, in which the data is partitioned among all processor. Each processor perform the same serial Apriori algorithm with incomplete information on support counts. Only local support counts (based on local partition) is available to a processor. A global support counts needs to be established through MPI communication. The above diagram show the ideas of the Counts Distribution algorithm.
Parallel Apriori - Counts Distribution (2)

Steps:
1. Partition data with equal size to each processors $P_i$ $(1 \leq i \leq N)$.
2. Pass $K = 1$
   - Each processor $P_i$ scan its local partition $D_i$ to count support for 1-itemsets.
   - Each processor $P_i$ perform MPI_Allreduce to gets global 1-itemsets support count.
   - Each processor $P_i$ asynchronously determine the large 1-itemsets.

Notes: Assume each processor know the largest item value in the data sets.
The items is assumed to be integer value that correspond to products ID.
In this case, each processor $P_i$ simply create an array of size $\text{max item value}$ and perform the counting.

• The data is initially partitions to all processor by the root processor.
• At pass $k=1$, all processor find the local support count of 1-itemset. These local support counts were gathered by all the processors. Then, they can independently find the large 1-itemset.
Parallel Apriori - Counts Distribution (3)

3. Pass K>1
   - Each processor $P^i$ generates candidates itemsets using large itemsets in iteration K-1.
   - $P^i$ asynchronously extracts local counts for $C_k$ into a count array $LCntArr$ (size max item value).
   - Synchronous all processors and perform $MPI_{Allreduce}$ on $LCntArr$ with $MPI_{SUM}$ binary function to get global support count $GCntArr$.
   - $P^i$ updates its count for $C_k$ using $GCntArr$.
   - $P^i$ gets large itemsets from $C_k$ (i.e. support $\geq$ minsup).

- The slide above shows the step taken for pass $K > 1$
- The general steps are similar to pass $K=1$ which involves global support count gathering and updating local hash tree support counts.
- After obtaining the global support counts, each processor can then locally generate the association rules by assigning the large candidates itemset to a processor in a round-robin manner (e.g. the i-th large itemset assigned to the j-th processor).
Parallel Apriori - Counts Distribution (4)

- \( P^i \) parallel generation of association using *round-robin* method.
• Example of global support counting for two processors \( p_1 \) and \( p_2 \) at the first pass.
• The transactions are partitioned into 2 transactions for processor \( p_1 \) and 3 transactions for processor \( p_2 \).
• The local support counts are then established for all 1-itemset (table (1) & (2)).
• The global support counts after the MPI_Allreduce function calls is shown in table (3) and (4).
### Parallel Apriori - Counts Distribution (5)

<table>
<thead>
<tr>
<th>2-itemset</th>
<th>Count</th>
<th>2-itemset</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 2)</td>
<td>0</td>
<td>(1 2)</td>
<td>1</td>
</tr>
<tr>
<td>(1 3)</td>
<td>1</td>
<td>(1 3)</td>
<td>1</td>
</tr>
<tr>
<td>(1 5)</td>
<td>0</td>
<td>(1 5)</td>
<td>1</td>
</tr>
<tr>
<td>(2 3)</td>
<td>1</td>
<td>(2 3)</td>
<td>1</td>
</tr>
<tr>
<td>(2 5)</td>
<td>1</td>
<td>(2 5)</td>
<td>2</td>
</tr>
<tr>
<td>(3 5)</td>
<td>1</td>
<td>(3 5)</td>
<td>2</td>
</tr>
</tbody>
</table>

Pass $k > 1$

<table>
<thead>
<tr>
<th>2-itemset</th>
<th>Count</th>
<th>2-itemset</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 2)</td>
<td>1</td>
<td>(1 2)</td>
<td>1</td>
</tr>
<tr>
<td>(1 3)</td>
<td>2</td>
<td>(1 3)</td>
<td>2</td>
</tr>
<tr>
<td>(1 5)</td>
<td>1</td>
<td>(1 5)</td>
<td>1</td>
</tr>
<tr>
<td>(2 3)</td>
<td>2</td>
<td>(2 3)</td>
<td>2</td>
</tr>
<tr>
<td>(2 5)</td>
<td>3</td>
<td>(2 5)</td>
<td>3</td>
</tr>
<tr>
<td>(3 5)</td>
<td>3</td>
<td>(3 5)</td>
<td>3</td>
</tr>
</tbody>
</table>

Pass $k > 1$

- for pass $K > 1$, the table are update in the same fashion as in pass $K=1$. Readers can trace the support counts by hand to be more understand on how it work.
Data Distributions

- In count distributions, each processor with exact set candidate itemsets. Thus, redundancy operating on the same candidate itemsets.
- In data distributions, each processor working on distinct set of candidate itemset Ci. The union of candidate itemsets \( \cup Ci \) is the complete set.
- Due to the partition of data, a processor has no complete set of the data allocated in others processor. Thus, data transfer between processor are necessary.
- Suited for implementation environment that has high communications bandwidth because the data transfer time may significant.
Data Distributions Algorithm (1)

Steps

Pass k=1
- Same as count distributions.

Pass k > 1
1. Processor P\textsuperscript{i} generates C\textsubscript{k} from L\textsubscript{k-1}. It retains only 1/Nth of the candidate itemsets subsets C\textsubscript{k}\textsuperscript{i}. Each processor has distinct set of candidate itemsets.
2. Processor P\textsuperscript{i} count the support counts for local candidates using the data from both local and other processors.
3. Each processors Pi find it local large candidate itemset L\textsubscript{k} using the local C\textsubscript{k} using the local C\textsubscript{k}\textsuperscript{i}, where \bigcup L\textsubscript{k} = L\textsubscript{k}.

- The essential steps of data distribution algorithm are to distribute equally the candidate itemsets to each processor and exchanging the data at each processor.
- To assign 1/N of the candidates to each processor, we use the round robin methodology, where the ith candidate itemset is assign to the (i mod P)-th processor. Where P is the processor size.
Data Distributions Algorithm (2)

3. Processors exchange $L'_k$ so that every processor has the complete $L'_k$ for generating $C_{k+1}$ for the next pass. Each processor can independently decide whether to terminate or continue on the next pass.
The data distribution algorithm partition both the data and the candidate itemsets. Each processor will have a difference hash tree with different set of $C_k$. The local large itemsets $L_k$ are gathered from all processors. Whereas the local data partition in each processor are send to other processors.
Data Distribution

- Large Itemsets at P0 and P1

<table>
<thead>
<tr>
<th>Large Itemset</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2, 3)</td>
<td>3</td>
</tr>
<tr>
<td>(2, 5)</td>
<td>4</td>
</tr>
<tr>
<td>(3, 5)</td>
<td>2</td>
</tr>
<tr>
<td>(4, 6)</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Large Itemset</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2, 6)</td>
<td>3</td>
</tr>
<tr>
<td>(3, 5)</td>
<td>2</td>
</tr>
<tr>
<td>(8, 9)</td>
<td>5</td>
</tr>
<tr>
<td>(10, 17)</td>
<td>8</td>
</tr>
</tbody>
</table>

- After all gathered the large itemsets

<table>
<thead>
<tr>
<th>Large Itemset</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2, 3)</td>
<td>3</td>
</tr>
<tr>
<td>(2, 5)</td>
<td>4</td>
</tr>
<tr>
<td>(3, 5)</td>
<td>2</td>
</tr>
<tr>
<td>(4, 6)</td>
<td>10</td>
</tr>
<tr>
<td>(2, 6)</td>
<td>3</td>
</tr>
<tr>
<td>(3, 6)</td>
<td>2</td>
</tr>
<tr>
<td>(8, 9)</td>
<td>5</td>
</tr>
<tr>
<td>(10, 17)</td>
<td>8</td>
</tr>
</tbody>
</table>

- Each processor (P0 and P1) will have different set of large itemsets after find the their support counts.
- Each processor will then gather all the large itemsets from all processor so that it can use to generate the new set of candidate itemset in the next iteration.
• Show the flow chart for the data distribution algorithm.
• The algorithm post P-1 asynchronous receive MPI functions to receive data from other processors.
• If data is (are) received itemset from other processor, it will process the received data first before processor it local itemset. This will avoid data congestion in the communication line or buffer. If no data is receive, the processor will get one itemset from local partition and update the candidate itemset count.
• After all data have been use (local and distributed data) to update the local candidate itemset, the next step will be to find the large candidate itemset for further processing.
Asynchronous Receive and Sending (1)

- Each processor needs to send its local data to others.
- Cannot use blocking point-to-point communication because it will waste most of the time waiting for data to arrive.
- Can take benefit from the asynchronous non-blocking point-to-point communication to receive and send data.
- Use MPI_Isend() and MPI_Irecv() MPI functions.
Asynchronous Receive and Sending (2)

- Receive data
  - post P-1 MPI_IRecv() requests.
  - check if any of the receive buffers have received data using MPI_Test or MPI_Testany() functions.
  - repost requests for buffer that are free.

- Send data
  - Create a buffer to send local itemsets.
  - If send buffer is full, post one MPI_Isend() requests to each processor.
  - If not more local itemset, post End of Transfer (EOT) to all processor indicates no more data.
Counts/Data Distribution implementations

■ Assume knowledge of list standard template library and C++.

1. Database file
   • ASCII text file
   • Transactions format:
     \[ \text{TID Num item}_1 \text{ item}_2 \ldots \text{ item}_\text{Num} \]
     where Num is the number of items.
     \[ \text{TID and item are 2 bytes integer value.} \]

2. Configuration file
   • Number of transaction and max item value.
### Input Format Example

- **Filename “input.txt”** *
  - 0: 5 2 4 9 18 20
  - 1: 7 4 5 8 9 18 19 20
  - 2: 5 1 10 13 19 20
  - 3: 3 9 17 20
  - 4: 1 3
  - 5: 7 2 5 7 9 12 14 18

- **Configuration file “config.txt”** *
  - 100 ← Total number of transactions
  - 20 ← Maximum item number

---

EDPNMO006/2001 (D.Taniar & K.Smith)
Output Format

- Output files
  - assocxx.rules where xx is the processor rank number.

- Outputs
  - \[7 \implies 2\]
  - Conf=40.74% Support=11.00% (11)

Conf – rule confidence
11.00% is the support counts in percentages and (11) is the support count in number.
Source Files

- Source files
  - Apriori.cpp
  - Bcalcdb.cpp
  - Hashtab.cpp
  - HashTree.cpp
  - Itemnode.cpp
  - Itemset.cpp

- Header files
  - Apriori.h
  - Bcalcdb.h
  - Hashtab.h
  - HashTree.h
  - Itemnode.h
  - Itemset.h

Slide above shows the source files and header files for the parallel Apriori program.
Global Variables

- Apriori.h, Apriori.cpp
  
  int DBASE_NUM_TRANS;  /* total number of transaction */
  int DBASE_MAXITEM;    /* product ID max value */
  int MIN_SUPPORT;       /* minsup */
  float MIN_CONFIDENCE;  /* minconf */
  int processor_size;    /* total number of processor */
  int my_rank;           /* processor rank number */
  list<Itemset *> freq_itemset;  /* set of large itemset so far */
  Hash_tbl * htbl;      /* hash table to store all large itemsets */
1. class Itemset {
    private:
        int support_count;  // support count for this itemset.
        list<int> * lst_item;  // item in the itemset.
    
    public:
        Itemset();
        Itemset(int);
        ~Itemset();
        Itemset(Itemset & iset);  // copy constructor

    int & support() { return support_count; }
    int subset(list<int> & iv);  // subset method.
    void clear() { lst_item->erase(lst_item->begin(), lst_item->end()); }  // clear the itemset.
    void insert(int new_i) { lst_item->push_back(new_i); }
    int get_size() { return lst_item->size(); }

    list<int>& get_item() const { return *lst_item; }
    // operators use for stl container class.
    friend ostream& operator<< (ostream& os, Itemset& iset);
}
const Itemset& operator=(Itemset& iset);
bool operator>(const Itemset& iset);
bool operator<(const Itemset& iset);
20. bool operator==((const Itemset& iset));
bool operator!=(const Itemset& iset);
bool operator<=(const Itemset& iset);
bool operator>=(const Itemset& iset);
int operator[](int index) const;
25. static int compare(const void*a, const void*b);
);

• This object class store an itemset. The important private variables are the support count and items list.
• Other methods are use to support private members access and operator operation.
• For the [] operator, the first item number in the list is 0.
Object Classes Definitions (3)

- `Itemnode.h`
  
  ```cpp
  class ItemNode
  {
  ...
  }
  ```

- Store a list of itemsets in `list<Itemset*> * theItemList;`

```
A
B
C

A
B
D
```

1. class ItemNode {
   
   private:
   
   list<Itemset *> *theItemList;
   
   public:

   5. ItemNode();
   
   ~ItemNode();
   
   ItemNode(ItemNode & inode);
   
   list<Itemset *> & items_list() { return theItemList; }

   void insert(Itemset * new_item);

   10. int item(int index);

   Itemset * operator[](int index) const;
   
   int get_size() { return theItemList->size(); }

   friend ostream& operator<<(ostream& os, ItemNode & i_node);

};
```

• Provides Itemsets access, insertion, and support operator.
1. class Hash_tbl {
   private:
   int hash_func ( const Itemset * );
   list < Itemset * > * * buckets;
   long element_count; //number of element hashed
   int size; //size of buckets
   
   public:
   Hash_tbl ( int = 997 ); //constructor
   ~Hash_tbl(); //destructor
   friend ostream & operator < (ostream & os, Hash_tbl & htbl); //friend ostream operator< (ostream & os, Hash_tbl & htbl);
   void insert ( Itemset * & );
   int lookup ( Itemset * & );
};

5. Method hash_func is the hash function.

10. The lookup method is use to find the existence of an itemset and return it support count if found.

- Provides chained hash table functions.
- Items are chained using list<Itemset *> list template.
- Does not provide memory limitation checking.
Object Classes Definitions (5)

- Bcalcdb.h
  
  ```
  class Dbase_Ctrl_Blk
  {...}
  ```

- Provides methods to read a transaction, and read configuration file parameters into global variables.

1. class Dbase_Ctrl_Blk{
   private:
   int numtrans; /*last read transaction number*/
   ifstream fd; /*file pointer*/

   public:
   Dbase_Ctrl_Blk(char *infile, char *confile);
   ~Dbase_Ctrl_Blk();
   int get_next_trans(int &tid, int &nitems, list<int> *&buf);
};

- method get_next_trans() read in a transaction from the database file fd and return the itemset in buf, the number of items in nitems and transaction id in tid.
- the input filename and the configuration filename is pass to the class constructor (i.e. infile & confile variables).
Object Classes Definitions (6)

- HashTree.h
  
  ```cpp
  class HashTree
  {
  ...
  }
  ```

- Provides the hash tree constructs and supporting methods.

- Two important storage variables are the hash table (interior node) and the list of ItemNode which hold the itemsets (leaf node).

```cpp
1. class HashTree {
   private:
     HashTree **theHTable; //Hash table for the internal node.
     list<ItemNode *> * theEList;
   5. int theFlg;       //flag for VISITED.
     int theDepth;      //depth of this node.
     static int theCount;
     static int theThreshold; //maximum number of itemset in a leaf
                               //node before rehash/split.
     static int theFanOut;  //branching factor for the hash table
   10. static int theMaxDepth; //maximum allowable depth for the hash
                               //tree.
     void rehash(); //procedure for converting leaf node to an
                    //interior node
   public:
     HashTree(int=0);
     ~HashTree();
   15. int hash(int);       //hash function
     void add_element(ItemNode *);
  }
```
int isLeaf(){ return (theHTable == NULL); }
int& flag(){return theFlg; }    //store transaction ID.
int& depth(){return theDepth; }   //this hastree depth.
20. HashTree **&htable(){return theHTable;} //get the Hash table
void globalsupport(int *&i, int &j, int &k);
list<ItemNode *> *&get_list(){return theEList;}
static int& count(){ return theCount; }
static int& fanout(){ return theFanOut; }
25. static int& threshold(){ return theThreshold; }
static int& maxdepth(){ return theMaxDepth; }
static void calc_fanout(int num, int dep){
    theMaxDepth = dep;
    theFanOut = (int)ceil(pow(num/theThreshold, 1.0/dep));
    if (theFanOut < 1) theFanOut = 1;
}
friend ostream& operator << (ostream&, HashTree&);
};

- Lines 3 is the hash table for interior node and Line 4 is the itemnode list for leaf node.
- Lines 5 to 10 are the variables for Maximum depth, Maximum threshold, branching factor, current depth, and the visited flag.
- Lines 27 to 30 are method to calculate the branching factor and the maximum depth in the hash tree. The parameters for this method are the total number of possible itemset (from previous iteration) and dep is the current iteration number.
- Explanation of the branching factor calculation can refers to http://www.supercomp.org/sc96/proceedings/SC96PROC/ZAKI/INDEX.HTM.
- The global_support() method traverse the hash tree in a depth first order and update/return the support counts of all leaf node using/in an array.
Data partitioning (1)

- The original input file is partitioned into $P$ parts. Where $P$ is the number of processors.
- The partition is done by the $\text{partition()}$ function.
- The partitioned file name is “par$\times$.txt” where $\times$ is the processor rank number.
- Each processor will also have their own configuration file param$\times$.txt.

```cpp
1. void partition(Dbase_Ctrl_Blk * & db, long parse)
{
    ofstream * fout;
    ofstream * fconf;
3.    int numtrans, tid;
4.    list<int> * buf = new list<int>; // input buffer
5.    char filename[50]; // output file name for each processor
6.    char config[50]; // configuration file name for each processor
7.    long i;
8.    int proc_rank;
9.    if (processor_size == 1)
10.    return;
11.    i =0;
12.    proc_rank =1;
14.    sprintf(filename,"par%d.txt",proc_rank);
15.    fout = new ofstream(filename,ios::out);
16.    sprintf(config,"param%d.txt",proc_rank);
17.    fconf = new ofstream(config,ios::out);
18.    *fconf << parsez << endl;
19.    *fconf << (DBASE_MAXITEM-1);
20.    fconf->close();
```
while(db->get_next_trans(tid,numtrans,buf))
{
    *fout << tid << " " << numtrans << " " << *buf << endl;
    i++;
    if (i >= parsize)
    {
        if (proc_rank < (processor_size -1))
        {
            fout->close();
            sprintf(filename,"par%d.txt",++proc_rank);
            fout->open(filename,ios::out);
            sprintf(config,"param%d.txt",proc_rank);
            fconf->open(config,ios::out);
            *fconf << parsize << endl;
            *fconf << (Dbase_MAXITEM -1);
            fconf->close();
            i=0;
        } else {
            fout->close();
            break;
        }
    }
} / / while
//get the remaining transaction for root processor
45. fout->open("par0.txt",ios::out);
fconf->open("param0.txt",ios::out);

    *fconf << ((DBASE_NUM_TRANS % processor_size) + parsize)
    << endl;
    *fconf << (DBASE_MAXITEM -1);
fconf->close(); delete fconf;

50. while(db->get_next_trans(tid,numtrans,buf))
{
    *fout << tid << " " << numtrans << " " << *buf << endl;
}
fout->close();
55. delete fout;
}

• Lines 2-9 are local variables declarations
• Lines 15-17 creates the initial input and configuration files for processor rank 1.
• Lines 28-38 creates a new input and configuration files after finished partition 1/P of the input data.
• Lines 45-54 put the remaining input data for the root processor.
• Note that, the partition() function only calls by the root processor. Other processor can only continue execution after finished data partitioning.
Parallel Apriori Implementations

Main functions (Apriori.h)

- `list<ItemNode *> * get_F1_support(Dbase_Ctrl_Blk &);`
- `void get_tree_freq(HashTree *, list<ItemNode *> * &);`
- `void get_freq(int, HashTree *, list<ItemNode *> * &);`
- `Itemset * new_cand(Itemset *, Itemset *);`
- `int get_iilist(HashTree *, ItemNode *);`
- `int generate_cand(int, HashTree *, list<ItemNode *> * &);`
- `void increment_cand_count(HashTree &, list<int> *);`
- `void count_support(HashTree &, int, int, list<int> *, int);`
- `void get_FK_support(Dbase_Ctrl_Blk &, list<ItemNode *> *);`
- `void genrules(ofstream * &);`
- `void apgenrules(Itemset * & , list<ItemNode *> * & , int , int &);`

- `get_F1_support()` - find the local support count for 1-itemset and synchronize to establish global support counts. The result is a set of large 1-itemsets.
- `get_tree_freq()` - get the large itemsets from a hash tree.
- `get_freq()` - call `get_tree_freq()` to find the large itemsets. For k=1, function `get_tree_freq()` is not called.
- `new_cand()` - create a new candidate itemset of length k from two length (k-1) itemset.
- `get_iilist()` - generate all candidates itemset of length k from k-1 iteration that have similar (k-1) prefix. These itemset are stored in an ItemNode.
- `generate_cand()` - call `get_iilist()` function to generate all candidate itemsets and stored in ItemNode. All ItemNode is store in the HashTree.
- `increment_cand_count()` - increment the support counts of all itemset in a leaf node that contained in a transaction.
- `count_support()` - update hash tree candidates itemset support by calling `increment_cand_count()` function.
- `get_FK_support()` - main loop of Apriori algorithm.
- `genrules()` - generate association rules antecedents
- `apgenrules()` - generate association rules
Counts Distributions (1)

- Main loop of count distribution
  - void get_FK_support
    - dbc - database file object class
    - F1 - 1-large itemsets

- Functions calling sequences
  - call generate_cand() to generate candidates
  - call count_support() to find supports
  - call get_freq() to get large itemsets
  - call genrules() to generate association rules

1. void get_FK_support (Dbase_Ctrl_Blk &dbc, list<ItemNode *> *F1)
   {
       HashTree *Cand = NULL;
       list<ItemNode *> * FK = F1;
       list<ItemNode *>::iterator nodeitr;
       list<Itemset *>::iterator isetitr;
       int cands;
       int * sup_arr;
       int index, flag;
       list<int> * buf = new list<int>;
       int iter;
       int transid;
       int numitem;
       ofstream * rulefile;
       char rulefname[50];
       get_freq(1, NULL, F1);
       store_large_itemset(Freq_itemset);
       sprintf(rulefname,"rule%d.txt",my_rank);
       rulefile = new ofstream(rulefname,ios::out);
       iter=1;
while (Freq_itemset.size() > 0) {
    iter++;
    cands = generate_cand(iter, Cand, FK);
    while (dbc.get_next_trans(transid, numitem, buf)) {
        count_support(*Cand, 0, numitem - iter + 1, buf, transid);
    }
    sup_arr = new int[cands];
    assert(sup_arr != NULL);
    index=0;
    flag=0;
    Cand->global_support(sup_arr, index, flag);
    int *globalcount = new int[cands];
    assert(globalcount != NULL);
    MPI_Barrier(io_comm);
    MPI_Allreduce(sup_arr, globalcount, cands, MPI_INT,
                  MPI_SUM, io_comm);
    index=0;
    flag = 1;
    Cand->global_support(globalcount, index, flag);
    delete [] sup_arr;
    delete [] globalcount;
    get_freq(iter, Cand, FK);
    delete Cand;
    Cand = NULL;
    store_large_itemset(Freq_itemset);
    genrules(rulefile);
} rulefile->close();
delete rulefile;
• Lines 1-14 are the local variables declarations
• In line 15, the get_freq() function is called to get all the 1-large itemsets.
• In line 16, these 1-large itemsets are hashed into hash table.
• In line 17, a output file rule.txt is created to store the association rules, where is the processor rank number.
• The main loop start at line 20 and end at line 46.
• In line 23, the generate_cand() function is called to generate the candidates itemset in from the previous iteration large itemset. The hash tree that stores the new candidates itemset is pass back in Cand variable. The total number of candidates itemset is returned in cands variable.
• Lines 24-26 find the support counts of all candidate itemsets using the count_support() function.
• Line 27 create an array with size cands that use to stored the local support counts. This was achieved by calling the the global_support() method in line 31. Here we set the flag to 0 the indicate retrieval.
• In line 32, another array (globalcount) with the same size is created as a placeholder of the global support counts.
• In lines 34 and 35, all processors are synchronized and the MPI_Allreduce() MPI function is called to obtains the global support count.
• In line 38, the global_support() method with flag 1 (updating) is called to update the candidate itemsets with global support counts.
• In line 41, the get_freq() function is calls to find the large itemsets.
• In line 44, these large itemsets are stored in the hash table along with their support counts.
• In line 45, the genrules() function is calls to generate association rules from the large itemsets.
Find Large 1-itemset (1)

Apriori.cpp

- Function get_F1_support()
  - Input parameter - Dbase_Ctrl_Blk type that contains the database file pointer.
  - Counts support for all 1-itemset
  - Establish local large 1-itemsets (line 10-20)
  - Get global support counts (line 22)
  - Independently determines global large 1-itemset (line 23-30)

```cpp
1. list<ItemNode *> * get_F1_support (Dbase_fCtrl_Blk& DB) {
    int i;
    int tid, numitems;
    list<int> * buf = new list<int>;

5. list<ItemNode *> * F1 = new list<ItemNode *>;
    ItemNode * itemnode = new ItemNode;
    int * itemcount = new int[DBASE_MAXITEM + 1];
    int * result = new int[DBASE_MAXITEM + 1];
    for (i = 0; i <= DBASE_MAXITEM; i++)
10. { itemcount[i] = 0;
         result[i] = 0;
    }
}
while (DB.get_next_trans(tid, numitems, buf))
{ list<int>::iterator itr = buf->begin();
15. while (itr != buf->end())
         itemcount[*itr]++; itr++;
} //while
```
MPI_Allreduce(itemcount,result,DBASE_MAXITEM+1,MPI_INT,MPI_SUM,io_comm);
for(i=0;i<= DBASE_MAXITEM;i++)
20. { if (result[i] >= MIN_SUPPORT)
    { Itemset *fit = new Itemset();
      fit->insert(i);
      fit->support() = result[i];
      itemnode->insert(fit);
    }
}
   F1->push_back(itemnode); delete []itemcount; delete [] result;
delete buf; return F1;}

• Lines 18, the MPI_allreduce() function is call to obtain the global support count.
• Lines 19 to 26 find all large 1-itemset and store in an ItemNode and put into the variable F1.
Generate Candidate Itemsets (1)

- Generating candidates
  - `generate_cand()`
  - `gen_ilist()`

- How candidates are organized in an ItemNode?
  - Each ItemNode stores candidate itemset that all have the same (k-1) items, where k is the current iteration number.
  - For example, AB, AC, AD & AE are large itemsets in iteration 2. The candidates at iteration 3 are,

\[
\begin{align*}
(A \ B \ C) & \rightarrow (A \ C \ D) & \rightarrow (A \ D \ E) \\
(A \ B \ D) & \rightarrow (A \ C \ E) & \\
(A \ B \ E) & 
\end{align*}
\]

```
1. int generate_cand(int iter, HashTree *Cand, list<ItemNode *> *FK) {
    int candcnt = 0;
    Cand = new HashTree;

5. int newhsz = Freq_itemset.size() * (Freq_itemset.size() -1) /2;
    Cand->calc_fanout(newhsz, iter-1);
    list<ItemNode *>::iterator itr = FK->begin();
    while(itr != FK->end()) {
        10. candcnt += gen_ilist(Cand, *itr);
            itr++;
    }

    return candcnt; }
```

- This function create a new hash tree at start of an iteration (lines 4) and calling the calc_fanout() function to the find the branching factor of hash tree. In line 10, the gen_ilist() function is calls to generate the new candidates itemset from each ItemNode. In line 7, the itr variable is an iterator to the first ItemNode that is in the list from previous iteration.
1. int gen_ilist(HashTree * Cand, ItemNode * inode)
   ItemNode *tempnode;
   int ccnt = 0;
   Itemset * fit;
5. list<Itemset *>::iterator itr1 = inode->items_lst()->begin();
   list<Itemset *>::iterator itr2;
   while (itr1 != inode->items_lst()->end())
   {
      itr2 = itr1;
10.   itr2++;
      tempnode = NULL;
      while (itr2 != inode->items_lst()->end())
      {
         fit = new_cand(*itr1, *itr2);
15.      if (fit != NULL)
         {
            if (tempnode == NULL) {
               tempnode = new ItemNode();
               assert(tempnode != NULL); }
         20.      tempnode->insert(fit);
            ccnt++;
         }
         itr2++;
         } //while
         if (tempnode != NULL)
25.   {
            //add new candidate into hash tree.
            Cand->add_element(tempnode); }
         itr1++;
      }
30. return ccnt;
    }

• In this function the itemsets in an ItemNode are self joined. In line 14, if two itemsets could be joined, the new_cand() function return the new candidate itemset else it returns NULL. On line 26, the new candidates itemset are hashed into hash tree.
• The total number of candidate itemsets are returned by this function (lines 30).
Adding Candidate Itemsets to Hash Tree (1)

- Adding candidate ItemNode into hash tree
  - using add_element() method of HashTree class.
  - recursively calls itself until reach the leaf node
  - at leaf node, the candidate ItemNode is added to the ItemNode list
  - convert leaf node into interior node if number of ItemNode’s greater than threshold and current depth less than maximum depth.

1. void HashTree::add_element(ItemNode *hel)
   
   assert(hel != NULL);
   if (isLeaf()){
      if (theEList == NULL)
         theEList = new list<ItemNode *>;
         theEList->push_back(hel);
         if (theEList->size() > theThreshold)
            if (theDepth + 1 <= theMaxDepth)
               5.    rehash();
   }
   10. else { // interior node
           int hval = hash((**hel)[0]) [theDepth]);
           15. if (theHTable[hval] == NULL)
                    theHTable[hval] = new HashTree(theDepth + 1);
                    // add ItemNode to the child node.
                    theHTable[hval]->add_element(hel);
               }
   }

• Lines 4, check whether current node is a leaf node or not.
• Lines 8-10, checks the maximum threshold and maximum depth constraint.
• Lines 14, hash on item number current depth in an itemset to find the child node to traverse to.
• Lines 17, recursive call.
Support Counts (1)

- Support counting
  - **void count_support()**
    - HashTree& ht - hash tree
    - int start - start hashing index for an itemset
    - int end - end hashing index for an itemset
    - list<int>* trans - a transaction
    - int tid - transaction ID
  - **increment_cand_count()**
    - HashTree &ht - hash tree
    - list<int>* trans - transaction

- The count_support() function is recursively called to perform support counts
- In a leaf node the function increment_cand_count() is called to update support counts

```
1. void count_support(HashTree& ht, int start, int end,
        list<int>* trans,int tid)
    {
        int i,hval;
        if(ht.isLeaf())
            {
6.           if ((ht.flag() != tid) && (ht.get_list())
7.              && (ht.get_list())->size() > 0))
8.                {
9.                    ht.flag() = tid;
10.                   increment_cand_count(ht,trans);
11.                }
12.            } else if (end <= (int)trans->size())
13.                {
14.                    list<int>::iterator itr = trans->begin();
15.                        for(i=start;i< end;i++)
16.                            {
17.                            hval = ht.hash(*itr);
18.                            if((hthtable())[hval] != NULL)
19.                               count_support((hthtable())[hval]),
20.                                   i+1, end+1,trans,tid);
```
• The counts support function recursively call it self to update the support count of all subset of a transaction.
• On lines 15, the hash function hash on an item in the transaction.
• On lines 17, the recursive call is done by pass the child node in the hash table bucket as parameter.
• On lines 5-9, if the child node is a leaf node, the increment_cand_count() function is call to update all candidates itemset that contained in the transaction. Below shows the increment_cand_count function code.

```cpp
void increment_cand_count(HashTree &ht, list<int> *trans) {
    list<ItemNode> :: iterator itr1;
    list<Itemset> :: iterator itr2;
    itr1 = ht.get_list() -> begin(); // head of ItemNode
    while (itr1 != ht.get_list() -> end()) {
        itr2 = (*itr1) -> items_lset() -> begin(); // head of Itemset
        while (itr2 != (*itr1) -> items_lset() -> end()) {
            if ((*itr2) -> subset(*trans)) {
                (*itr2) -> support() ++;
                itr2 ++; // next itemset.
            }
            itr1 ++; // next ItemNode.
        }
    }
}
```
Find large k-itemsets (1)

- Find large itemsets
  - `get_freq()`
  - `get_tree_freq()`
- Function `get_freq()` determine the large itemset from the hash tree support by `get_tree_freq()` function.
- Function `get_tree_freq()` performs recursive calls to until reached leaf node. At leaf node, the large itemset is saved into global variable (list).

```c
1. void get_freq(int iter, HashTree *Candidate, list<ItemNode *> * &FK) {
    Itemset *new_item;
    itemset for the next iteration.
    if (Freq_itemset.size() != 0)
        Freq_itemset.erase(Freq_itemset.begin(), Freq_itemset.end());
    if (iter == 1)
        { list<ItemNode *>::iterator itr1 = FK->begin();
          list<Itemset *>::iterator itr2;
          while (itr1 != FK->end())
            { itr2 = (*itr1)->items_lst()->begin();
              while (itr2 != (*itr1)->items_lst()->end())
                { new_item = *itr2;
                  Freq_itemset.push_back(new_item);
                  itr2++;
                }
              itr1++;
            } //while
```
• On lines 5, the previous iteration large itemsets were cleared.
• On lines 6-17, the large itemset for 1-itemset is found.
• On lines 23, the get_tree_freq() function is called.

1. void get_tree_freq(HashTree *ht, list<ItemNode *> *&FK)
   { int i;
     Itemset * fit;
     list<ItemNode *> ::iterator itr1;
     list<Itemset *> ::iterator itr2;
     if (ht->isLeaf())
     { //check if the ItemNode list is empty or not.
       if ((ht->get_list() != NULL) && (ht->get_list()->size() > 0)) //if-1
         //for all ItemNode in leaf hash tree node.
       for (itr1 = ht->get_list()->begin();
            itr1 != ht->get_list()->end(); itr1++) //for-1
         ItemNode * new_item_node = NULL;
       //for all Itemset in the ItemNode.
       for (itr2 = (*itr1)->items_lst()->begin();
            itr2 != (*itr1)->items_lst()->end(); itr2++) //for-2
         { //support of Itemset >= minimum support?
           if (((*itr2)->support() >= MIN_SUPPORT)
             { if (new_item_node == NULL)
                 new_item_node = new ItemNode();
               fit = new Itemset((**itr2));
               new_item_node->insert(fit);
Freq_itemset.push_back(fit);
}
  } //for-2
25.  if (new_item_node != NULL) {
    if (FK == NULL) FK = new list<ItemNode *>
    FK->push_back(new_item_node);
  }
  } //for-1
30. } //if-1
else { //internal node
  for (i=0;i<ht->fanout();i++)
    if ((ht->htable())[i]) //got child node
      get_tree_freq((ht->htable())[i],FK);
35. }
}

- On lines 6-24, the each itemset contained in the ItemNodes are traversed to check for their support counts. If their support count satisfy the minimum support threshold, it is added to the large itemset list template.
- If it is an interior node, the function will call itself recursively (line 34). Each hash table bucket that is node empty is traverse to the child node.
Data Distributions (1)

- Extras functions
  - DataDistribute() - distribute local data and receive data from other processor. At the same time, call count_support() function to count supports.
  - copybuffer() - copy a transaction to send buffer
  - CopyLargeSet() - copy local large itemset into send buffer
  - SaveLargeSet() - save received large itemsets into list template.
Data Distributions (2)

Main loop
- get_FK_Support()
  - call generate_cand()
  - call count_support()
  - call get_freq()
  - call DataDistribute()
  - call CopyLargeIset()
  - call SaveLargeIset()
  - call genrules()

1. void get_FK_support(Dbase_Ctrl_Blk &dbc, list<ItemNode *> *F1)

   {  ... //variables declarations and initialization
     allyotitem = (int*)malloc(sizeof(int) * processor_size);
     recvdisp = (int*) malloc(sizeof(int) * processor_size);
      iter = 1;
      totlitemset = (*FK->begin())->get_size();
      while (totlitemset>0)
      {
         iter++;
         cands = generate_cand(iter,Cand,FK,totlitemset);
         DataDistribute(&dbc,Cand,iter);
         get_freq(iter, Cand, FK);
         delete Cand;
         Cand = NULL;
         mysize = 0;
         if((FK != NULL) && (FK->size() > 0))
            mysize= CopyLargeIset(FK, ItemSetBuf, iter);
         else
            FK = new list<ItemNode*>;
         MPI_Allgather(&mysize,1, MPI_INT, alltotitem, 1,
                         MPI_INT,io_comm);
totsize = 0;
for(i=0; i < processor_size; i++)
    totsize += alltotitem[i];
totitemset =0;
25. if(totsize > 0)
    {   ...
        recvbuffer = (int*) malloc(sizeof(int) * totsize);
        MPI_Allgatherv(ItemSetBuf, mysize, MPI_INT, recvbuffer,
                        alltotitem, recvdisp, MPI_INT, io_comm);
        SaveLargeIset(FK, recvbuffer, totsize, iter,totitemset);
30. free(ItemSetBuf);
    free(recvbuffer);
    ItemSetBuf=NULL; recvbuffer=NULL;
    }
    genrules()
35. }//for
    rulefile->close();
    delete rulefile;
}

• Lines 7: Start of main loop
• Lines 10: generate candidate itemsets using previous iteration large itemsets (FK).
• Lines 11: Data distribution of local data and update support counts. Also receive complete data from other processor.
• Lines 12: get frequent itemset.
• Lines 16: Copy all large itemset into send buffer so that can be used to exchange with other processors.
• Lines 20: Gather from all the size of send buffer. This information is useful to allocate the size of receiving buffer.
• Lines 22-23: Calculate total receive buffer size.
• Lines 28: Exchanging large itemsets.
• Lines 29: Save the global large itemsets into FK list data structure.
• Lines 34: Generate association rules.
```c
1. void DataDistribute(Dbase_Ctrl_Blk * db, HashTree * cand, int iter) {
   ... // variables declarations and initializing
   size = processor_size -1;
   recvstatus = (MPI_Status*)malloc(sizeof(MPI_Status) * size);
   sendstatus = (MPI_Status*)malloc(sizeof(MPI_Status) * size);
   recvrequest = (MPI_Request*)malloc(sizeof(MPI_Request) *size);
   sendrequest = (MPI_Request*)malloc(sizeof(MPI_Request) *size);
   lst = new list<int>;
   buf = new list<int>;
10. for(i=0; i < size; i++)
    {
        recvrequest[i] = MPI_REQUEST_NULL;
        sendrequest[i] = MPI_REQUEST_NULL;
    }
15. recvbuffer = (int**) malloc(sizeof(int*) * size);
    sendbuffer = (int*) malloc(sizeof(int) * BUFFERSIZE);
    for(i=0; i < size; i++)
    {
        recvbuffer[i] = (int*) malloc(sizeof(int) * BUFFERSIZE);
20.    }
    allsend = false; allreceive = false;
    sendcount = 0; recvcnt = 0; nextpos =0;
    k=0;
    do{
25.        if(recvcnt < size)
            {
                for(i=0; i < size; i++)
                    {
                        if(recvrequest[i] != MPI_REQUEST_NULL)
20.                            continue;
                        MPI_Irecv(&recvbuffer[i][0], BUFFERSIZE, MPI_INT,
                                   MPI_ANY_SOURCE, MPI_ANY_TAG, io_comm, &recvrequest[i]);
                    }
                noreceived = true;
                for(i=0;i < size; i++)
30.            {
                if(recvrequest[i] == MPI_REQUEST_NULL)
                    continue;
                MPI_Test(&recvrequest[i], &flag, &recvstatus[i]);
```
if (flag)
40.    {  noreceived = false;

        MPI_Get_count(&recvstatus[i],MPI_INT,&outcount);
        k=0;
        do{

        45.            tid1 = recvbuffer[i][k++];
            if (tid1 == -111 || tid1 == EOT)
                break;
            numitem1 = recvbuffer[i][k++];
            for (l=0;l < numitem1; l++)
                buf->push_back(recvbuffer[i][k++]);
        50.            count_support(*cand,0,numitem1 - iter + 1, buf,tid1);
                buf->erase(buf->begin(),buf->end());
        }while(k < outcount);
            if (tid1 == EOT)
        55.                recvcount++;
        }
        if (recvcount == size)
            {
                allreceive = true;
        60.                    break;
        }
    }//for
    if (sendcount > 0)
        {
        for(i=0;i < size; i++)
        65.            {
                if (sendrequest[i] != MPI_REQUEST_NULL)
                    {
                        MPI_Test(&sendrequest[i], &flag, &sendstatus[i]);
                        if(flag == true)
                            sendcount--;
        70.                        }
                    }
        }//for
    }//if
    if (sendcount == 0 && allsend && noreceived)
        {
        if(lst->size() > 0)
        75.            count_support(*cand,0,numitem - iter + 1, lst,tid);
                copybuffer(lst,tid,numitem, sendbuffer,nextpos,BUFFERSIZE-2);
            } else
{ if(db->get_next_trans(tid,numitem,lst))
    { count_support(*cand,0,numitem - iter + 1, lst,tid);
      if(copybuffer(lst,tid,numitem,sendbuffer,nextpos, 
        BUFFERSIZE-2) == 0)
        { j=0;
          sendbuffer[nextpos] = -111;
          count = nextpos+2;
          for(i=0; i < processor_size; i++)
            { if(i==my_rank)
                continue;
                MPI_Isend(sendbuffer, count, MPI_INT, i, i, 
                io_comm, &sendrequest[j]);
              j++;
            }
          sendcount = size;
          nextpos =0;
        }
      } else
        { //EOT
          sendbuffer[nextpos] = EOT;
          count = nextpos + 1;
          j=0;
          for(i=0; i < processor_size; i++)
            { if(i==my_rank)
                continue;
                MPI_Isend(sendbuffer, count, MPI_INT, i, i, 
                io_comm, &sendrequest[j]);
              j++;
            }
          allsend = true;
        }
      } //else
    }
  }while(!allsend || !allreceive);
  for(i=0; i < size; i++)
    { if(recvrequest[i] != MPI_REQUEST_NULL)
        MPI_Cancel(&recvrequest[i]);
    }
  free(sendbuffer); free(recvbuffer); free(recvrequest);
  free(sendrequest);}
• Lines 4-7: Allocate send and receive request handles (MPI_Request). Also allocate their status handler.
• Lines 10-14: Init send and receive handles to NULL.
• Lines 15-20: Allocate a fixed memory size for send buffer. Whereas allocate a memory of size BUFFERSIZE for receiving buffer from each others processor.
• Lines 24: Start of looping to alternate between receiving data, send data and count support.
• Lines 25: Check if any of the receive buffer is free.
• Lines 27-32: Issuing an asynchronous receive from any source (MPI_ANY_SOURCE) and with any message ID (MPI_ANY_TAG).
• Lines 34-38: check if any of the request handles have finished receiving data. Use MPI_Test() MPI function to do so.
• Lines 42: If received a complete data stream, use MPI_Get_Count the determine how much data has been received.
• Lines 44-53: Process the received data. Perform support count.
• Lines 54: Check if the last received value is the end of transfer symbol. If it is, increment the number of count of completed receive data.
• Lines 63-72: Check if any of the send request pending by using MPI_Test().
• Lines 73: if there is still data haven't send, don't disturb the send buffer.
• Lines 75-80: Update support count using local data and then copy then to send buffer. If the buffer if full, then append the end of buffer symbol (-111) and send the data to each processor except itself (85-91).
• Lines 96-107: All local data have been processed, so append the end of transfer symbol to the last send buffer and send to each others processor.
• Lines 110: The termination conditions for send and receive are all local data has been sent and all data has been received.
• Lines 111-114: Check if any request handles still waiting. If it is, then release them.
**Exchange Large Itemsets (1)**

- Send/Receive buffer format:

  
  | m | L1 | ... | Lm | ... |
  
  m is the number itemset in the itemnode. L1...Lm are the large itemset for the Itemnode.

- At iteration k, the size of each large itemset is k. So, the receiver knows how to extract the received buffer.

- All processor needs to construct back the received large itemset in the same order so that the their will ensure the generated candidate itemsets are distributed correctly.

- The large itemsets buffer is array of integer. The support count of each itemset was appended after each itemset.

- Example:
  - Let an ItemNode consisted of (1, 3, 4):5, (1, 3, 5):2, where the number after “:” is the support count.
  - The array buffer will be [2 | 1 | 3 | 4 | 5 | 1 | 3 | 5 | 2 ].
int CopyLargeIsSet(list<ItemNode*> *linode, int *& buffer, int iter)
{
    int inodebeg = linode->begin();
    int inodeend = linode->end();
    buffer = (int*) malloc(sizeof(int) * cursize);
    i = 0;

    while (inodebeg != inodeend)
    {
        int isetbeg = (*inodebeg)->items_lst()->begin();
        int isetend = (*inodebeg)->items_lst()->end();
        int isetsize = (*inodebeg)->get_size();
        if ((i + (isetsize * iter + 2)) > cursize)
        {
            cursize += (100 + (isetsize * iter + 2));
            buffer = (int*) realloc(buffer, cursize);
        }
        buffer[i++] = isetsize;
        while (isetbeg != isetend)
        {
            ItemNodeItemST* itembeg = (*isetbeg)->get_item().begin();
            ItemNodeItemST* itemend = (*isetbeg)->get_item().end();
            while (itembeg != itemend)
            {
                buffer[i++] = *itembeg;
                itembeg++;
            }
            isetbeg++;
        }
        delete *inodebeg;
        inodebeg++;
    }
    of->close();
    linode->erase(linode->begin(), linode->end());
    return i;
}
End of Module 2
Association Rules