Characterizing Data Mining Algorithms and Applications:
Do they impact processor and system design?

Speaker
Alok Choudhary
Dept. of Electrical & Computer Engineering
Northwestern University

Project Team Members
Jayaprakash Pisharath
Ying Liu
Wei-keng Liao
Gokhan Memik

Sponsored by Intel Corporation
Introduction

- Tremendous growth in data
- Sophisticated tools to analyze data
- Growth in systems not enough to keep up with it
- Need for smarter systems and algorithms
- Where to start?
- Analyze & Attack
  - Investigate future mining needs
  - Analyze existing tools and algorithms
  - Identify shortcomings and try to meet it

Our current task
Analysis

- Very little is known
- Consistent, steady and phase-driven analysis
- Consider all domains
- First step: construct a benchmark
- To begin with, we look at
  - domains,
  - categories,
  - system features
MineBench

- Data mining benchmark
  - Application suite
  - Popular algorithms (domains)
  - Multiple categories (types) of mining

- Scalability of algorithms
  - Data
  - Algorithmic

- Evaluation options
  - Architecture
  - Measures of interest

- Expansion?
Methodology

- Include things that have not been considered till now (No TPC-H/ SPLASH/ SPEC)
- Eclectic mix of algorithms
  - Clustering
  - Classification
  - Association Rules
- If scalable, improve performance through parallelization
- Study the characteristics
<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Category</th>
<th>Description</th>
<th>Lang.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScalParC</td>
<td>Classification</td>
<td>Decision tree classifier</td>
<td>C</td>
</tr>
<tr>
<td>Naïve Bayesian</td>
<td>Classification</td>
<td>Statistical classifier based on class conditional independence</td>
<td>C++</td>
</tr>
<tr>
<td>K-means</td>
<td>Clustering</td>
<td>Partitioning method</td>
<td>C</td>
</tr>
<tr>
<td>Fuzzy K-means</td>
<td>Clustering</td>
<td>Fuzzy logic based K-means</td>
<td>C</td>
</tr>
<tr>
<td>BIRCH</td>
<td>Clustering</td>
<td>Hierarchical method</td>
<td>C++</td>
</tr>
<tr>
<td>HOP</td>
<td>Clustering</td>
<td>Density-based method</td>
<td>C</td>
</tr>
<tr>
<td>Apriori</td>
<td>ARM</td>
<td>Horizontal database, level-wise mining based on Apriori property</td>
<td>C/C++</td>
</tr>
<tr>
<td>Eclat</td>
<td>ARM</td>
<td>Vertical database, break large search space into equivalence class</td>
<td>C++</td>
</tr>
</tbody>
</table>
System Architecture

Processor 1 -> L2 Cache
Processor 2 -> L2 Cache
... -> L2 Cache
Processor 7 -> L2 Cache
Processor 8 -> L2 Cache

Memory

Disk 130 GB

4 GB Memory

700 Mhz

1 MB L2 Cache
Evaluation Setup

Data Mining Applications

Red Hat Enterprise Linux AS
Evaluation Metrics

- Evaluation perspectives
  - Algorithmic
  - Architectural
- Scalability Study
  - Single Processor, Multiprocessor Analysis
  - Scalable Data Analysis
- Algorithmic
  - Execution Times
  - Operating System Overhead
  - I/O Overhead
  - Synchronization Overhead
- Architectural
  - L1 Cache Analysis
  - L2 Cache Analysis
  - Memory Access Pattern Study
  - CPI Behavior
  - Instruction and Branch Behaviors
## Data Set

- **Clustering**
  - **HOP: ENZO data**
  - **K means, Fuzzy K means**
    - Real image database of 17695 pictures
    - 2 features: Color (9 floating points) and Edge (18 floating points)

### Dataset Parameters

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Classification</th>
<th>Association Rule Mining (ARM)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Parameter</td>
<td>DB Size(MB)</td>
</tr>
<tr>
<td>Small</td>
<td>F26-A32-D125K</td>
<td>27</td>
</tr>
<tr>
<td>Medium</td>
<td>F26-A32-D250K</td>
<td>54</td>
</tr>
<tr>
<td>Large</td>
<td>F26-A64-D250K</td>
<td>108</td>
</tr>
</tbody>
</table>

Fx-Ay-DzK denotes a dataset with Function x, Attribute size y, and Data comprising of z*1000 records. D denotes the number of transactions, T is the average transaction size, and I is the average size of the maximal potentially large itemsets.
Functional Analysis

(Single Processor Analysis)
ScalParC (Decision Tree Classifier)

A decision tree is a class discriminator that recursively partitions example set until each partition consists entirely or dominantly of examples from one class. Each non-leaf node of the tree contains a split point which is a test on one or more attributes and determines how the data is partitioned.

Partition (Example Set S)
- if (all examples in S are of the same class) then return;
- for each attribute A do
  evaluate splitting point on attribute A;
- use the best splitting found to partition S into S₁ and S₂;
- Partition (S₁);
- Partition (S₂);

* Use Gini index for splitting point selection measure:

\[
Gini(S) = 1 - \sum_{i=1}^{c} p_i^2 \quad Gini(S_1, S_2) = \frac{n_1}{n} Gini(S_1) + \frac{n_2}{n} Gini(S_2)
\]

- Find splitting point with minimum Gini
Function Profiling

- Splitting procedure (ParClassify) causes the maximum bottleneck (high resource and data stalls)
- Simple GINI index calculation could get complex (second most resource consuming, and time consuming operation)
- Index comparison (VRCompare) can also be optimized further (less instructions retired, but comparatively longer clock ticks)
- Main does just the delegation work
Naïve Bayesian classifier, a statistical classifier, assumes the effect of an attribute on a given class is independent of the other attributes. It predicts the probability that a given example belongs to a particular class.

Suppose there are m classes, \( C_1, C_2, \ldots, C_m \), assume \( P(C_1) = P(C_2) = \ldots = P(C_m) \), data example \( X = (x_1, x_2, \ldots, x_n) \), \( s_i \) is the number of training examples belonging to \( C_i \), \( s_{ik} \) is the number of training examples of class \( C_i \) having the value \( x_k \) for attribute \( k \).

• Scan the training data set, calculate \( s_i \) and \( s_{ik} \) for categorical attributes, calculate mean \( \mu_{Ci} \) and standard deviation \( \sigma_{Ci} \) for continuous attributes.

• To classify an unknown example \( X \), assign \( X \) to the class \( C_i \) with maximum \( P(X|C_i)P(C_i) \).
Functional Profiling

- **nbc_add** is used while scanning the dataset. A tuple is read (added to the list), an update is done to the class distribution list, and counters (sum, mean). Even though this operation has less data ref (13%), it results in high L1 misses (87%) and significant L2 misses. Memory is accessed too often => room for cache optimization.

- **tfs_getfld** retrieves fields (columns). This takes up considerable amount of the computation time. Lot of fields are retrieved (reason for high data references). Whereas reading attributes (as_read + att_ualadd) is much quicker – but can be improved still (misses).

---

<table>
<thead>
<tr>
<th>Function</th>
<th>Resource Related Stalls</th>
<th>L1 Misses</th>
<th>L2 Misses</th>
<th>Bus Transactions</th>
<th>Bus Memory Transactions</th>
<th>Data Memory References</th>
<th>Branch Instructions</th>
<th>Branch Mispredictions</th>
<th>Instructions Retired</th>
<th>Clockticks</th>
</tr>
</thead>
<tbody>
<tr>
<td>nbc_add</td>
<td>24.87</td>
<td>87.22</td>
<td>30</td>
<td>53.59</td>
<td>43.2</td>
<td>13.84</td>
<td>9.96</td>
<td>13.65</td>
<td>11.69</td>
<td></td>
</tr>
<tr>
<td>att_valadd</td>
<td>28.87</td>
<td>3.62</td>
<td>35</td>
<td>23.04</td>
<td>25.49</td>
<td>17.15</td>
<td>16.06</td>
<td>7.42</td>
<td>12.67</td>
<td>13.53</td>
</tr>
<tr>
<td>as_read</td>
<td>19.51</td>
<td>4.19</td>
<td>10</td>
<td>11.11</td>
<td>14.81</td>
<td>20.77</td>
<td>16.8</td>
<td>6.22</td>
<td>18.26</td>
<td>15.43</td>
</tr>
<tr>
<td>tfs_getfld</td>
<td>26.34</td>
<td>3.51</td>
<td>10</td>
<td>7.52</td>
<td>9.47</td>
<td>48.01</td>
<td>56.93</td>
<td>74.63</td>
<td>55.2</td>
<td>59.02</td>
</tr>
<tr>
<td>nbc_setup</td>
<td>0.01</td>
<td>0.79</td>
<td>5</td>
<td>0.82</td>
<td>0.97</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.2</td>
</tr>
</tbody>
</table>
K–Means (Clustering)

The k-means algorithm takes the input parameter, \( k \), and partitions a set of \( n \) objects into \( k \) clusters so that the resulting intra-cluster similarity is high but the inter-cluster similarity is low.

- arbitrarily choose \( k \) objects as the initial cluster centers;
- repeat
  - for each object \( p_i \) do
    - for each center \( c_j \) do
      - calculate distance between \( p_i \) and \( c_j \);
      - find the cluster center \( c \) nearest to \( p_i \);
      - assign \( p_i \) to the cluster centered by \( c \);
      - update the cluster center (calculate the mean value of objects in each cluster);
  - until no change;
## Function Profiling

<table>
<thead>
<tr>
<th>Function</th>
<th>Resource Related Stalls</th>
<th>L1 Misses</th>
<th>L2 Misses</th>
<th>Bus Transactions</th>
<th>Bus Memory Transactions</th>
<th>Data Memory References</th>
<th>Branch Instructions</th>
<th>Branch Mispredictions</th>
<th>Instructions Retired</th>
<th>Clockticks</th>
</tr>
</thead>
<tbody>
<tr>
<td>euclid_dist</td>
<td>56.4</td>
<td>86.55</td>
<td>88.76</td>
<td>89.38</td>
<td>89.64</td>
<td>54.72</td>
<td>69.24</td>
<td>72.14</td>
<td>68.63</td>
<td>67.68</td>
</tr>
<tr>
<td>find_nearest_point</td>
<td>15.09</td>
<td>6.18</td>
<td>0.09</td>
<td>0.01</td>
<td>0.06</td>
<td>17.56</td>
<td>11.48</td>
<td>0.37</td>
<td>10.33</td>
<td>10.43</td>
</tr>
<tr>
<td>main</td>
<td>0.06</td>
<td>0.27</td>
<td>0.25</td>
<td>0.34</td>
<td>0.31</td>
<td>0.01</td>
<td>0.04</td>
<td></td>
<td></td>
<td>0.04</td>
</tr>
</tbody>
</table>

- Euclidean distance calculation leads the pack.
  - Too many cache misses (no locality in access pattern)
  - Many data memory references as point coordinates are stored in memory (and hence max bus transactions)
  - Lot of these instructions (68.3%)
  - More room for automation of distance calculation. No need for big cache. Memory resident distance calculation would work best.

- Finding nearest point (center) is cache efficient (significant data ref, but less L2 misses). Lot of points are read at any given point of time (array).

- Clustering function does the work of assigning a point to cluster and recalculating the center. Main is just a block that delegates work.
Fuzzy K–Means (Clustering)

Rather than having a precise cutoff that an object is or is not a member of a particular cluster, Fuzzy K-means assumes that an object can have a degree of membership in each cluster.

- The Fuzzy K-means assigns each pair of object and cluster a probability that indicates the degree of membership of the object in the cluster.
- For each object, the sum of the probabilities to all clusters equals to 1.
- The assignment iterates until the sum of all membership values converges.

The flexibility of assigning objects to multiple clusters is necessary to generate better clustering qualities.
## Function Profiling

<table>
<thead>
<tr>
<th>Function</th>
<th>Resource Related Stalls</th>
<th>L1 Misses</th>
<th>L2 Misses</th>
<th>Bus Transactions</th>
<th>Bus Memory Transactions</th>
<th>Data Memory References</th>
<th>Branch Instructions</th>
<th>Branch Mispredictions</th>
<th>Instructions Retired</th>
<th>Clockticks</th>
</tr>
</thead>
<tbody>
<tr>
<td>fuzzy_kmeans_cluster</td>
<td>54.21</td>
<td>83.04</td>
<td>5.22</td>
<td>6.42</td>
<td>5.36</td>
<td>59.97</td>
<td>51.48</td>
<td>3.67</td>
<td>63.69</td>
<td>57.61</td>
</tr>
<tr>
<td>euclid_dist</td>
<td>39.9</td>
<td>11.08</td>
<td>88.75</td>
<td>88.22</td>
<td>88.49</td>
<td>32.93</td>
<td>38.33</td>
<td>94.43</td>
<td>27.22</td>
<td>32.89</td>
</tr>
<tr>
<td>sum_fuzzy_members</td>
<td>5.86</td>
<td>5.4</td>
<td>5.49</td>
<td>4.89</td>
<td>5.89</td>
<td>7.09</td>
<td>10.19</td>
<td>1.88</td>
<td>9.08</td>
<td>9.47</td>
</tr>
<tr>
<td>main</td>
<td>0.01</td>
<td>0.39</td>
<td>0.36</td>
<td>0.23</td>
<td>0.08</td>
<td>1.88</td>
<td>9.08</td>
<td>9.47</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Different from K-means even though goal is the same (bottlenecks have changed)
- `fuzzy_kmeans_cluster` involves fuzzy membership calculation. Lot of computation involved (high # of instructions retired) due to varying membership of each object.
- Euclid distance calculation is not the main bottleneck (but still significant). Cache is still a bottleneck. Lot of branch mispredictions as well – can be avoided if compiler assembles code more smartly.
- Convergence and check for convergence is really not a bottleneck (as we would expect). Implies, a lot more room for parallelization.
BIRCH (Clustering)

BIRCH is an integrated hierarchical clustering method, effective for incremental or dynamic clustering. *Clustering Feature tree (CF tree)* is used to summarize cluster representations. The non-leaf node in *CF tree* stores the *clustering feature (CF)* of its descendents.

\[
CF = (N, \overline{LS}, SS)
\]

- \(N\) number of objects in the subcluster
- \(\overline{LS}\) linear sum on \(N\) objects
- \(SS\) square sum of \(N\) objects
- \(B\) branching factor
- \(T\) threshold

**Phase 1:** for each object \(p\) in database do
- insert \(p\) into the closest leaf node (subcluster) \(l\);
- if the diameter of the subcluster stored in \(l > T\) then
  - split \(l\);
  - update the information in \(l\) and its ancestors;

**Phase 2:** apply a clustering algorithm (i.e. K-means) to cluster the leaf nodes of the *CF tree*. 
Software Operators (overridden in C++) are used for distance calculation purposes. Clearly indicates that software operators can be a considerable overhead. Hardware optimization could help (hardware vector processing).

Very high misses during redistribution of points to its closest seed (to obtain a new set of clusters). Caused due to the nature of tree access.
HOP (Clustering)

HOP, a clustering algorithm proposed in astrophysics, identifies groups of particles in N-body simulation. HOP clustering can be applied to applications involving neighborhood searching geology, astronomy, molecular pattern Recognition, etc.

• **Constructing a KD tree:**
  Recursively bisect the particles along the longest axis, and move nearby particles into the same sub-domain;

• **Generating density:**
  for each particle $p$ do
  traverse the tree to find $N_{dens}$ neighbors; assign an estimated density to $p$;

• **Hopping:**
  for each particle $p$ do
  associate $p$ with its densest neighbor; hop to its densest neighbor till $p$ reaches a particle that is its own densest neighbor;

• **Grouping:**
  Define particles associated to the same densest neighbor as a group;
  Refine and merge groups;
smBallGather: Gathering densest and nearby neighbors in ("Hopping" step) stalls for resources to the worst extent (lot of data ref).

But execution time of smBallSearch (finding nearest neighbor and generating density) is more than smBallGather. They are similar in functionality. Creates maximum misses.

Density calculation takes the maximum time (used by smBallSearch). But data efficient as such – less misses. But not instruction efficient – lot of room for improvement in branch mispredictions (88% of mispredictions).
Apriori (ARM)

Apriori is a level-wise search method to find the frequent itemsets in database records, where $k$-itemsets are used to explore $(k+1)$-itemsets. Apriori property, every subset of a frequent itemset has to be frequent, is used to prune many Candidate itemsets.

- $L_1 =$ frequent 1-itemsets;
- for $(k = 2; L_{k-1} \neq \emptyset; k++)$ do
  - $L_{k-1}$ is used to generate candidates set $C_k$;
  - eliminate those candidates $c$ having a subset that is not frequent from $C_k$;
  - for each transaction $t \in$ Database
    - for each $c \in C_k$ do
      - if ($c$ is a subset of $t$) $c$.count++;
    - $L_k = \{c \in C_k | c$.count $\geq min\_sup\}$
- return $L = \bigcup_k L_k$;
## Function Profiling

Database_readfrom produces the maximum stalls (lesser execution times). Data ref are less, but still misses and stalls are a lot (due to absence of locality as the access pattern is uniform). Even after modifying the access to bulk-load data to the memory (instead of contiguous access), it still remains a bottleneck.

Subset calculation is the most active routine. Max data references, but comparatively less L2 misses.

c.count++ is implemented as a separate function (increment) – takes up 4% of the time and considerable overheads. Even if a function is simple, compiler overheads do arise and hit the architecture.

<table>
<thead>
<tr>
<th>Function</th>
<th>Resource Related Stalls</th>
<th>L1 Misses</th>
<th>L2 Misses</th>
<th>Bus Transactions</th>
<th>Bus Memory Transactions</th>
<th>Data Memory References</th>
<th>Branch Instructions</th>
<th>Branch Mispredictions</th>
<th>Instructions Retired</th>
<th>Clockticks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database_readfrom</td>
<td>43.82</td>
<td>59.3</td>
<td>51.67</td>
<td>66.67</td>
<td>66.67</td>
<td>15.26</td>
<td>67.95</td>
<td>62.8</td>
<td>13.86</td>
<td>13.86</td>
</tr>
<tr>
<td>subset</td>
<td>18.74</td>
<td>21.65</td>
<td>18.44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main_proc</td>
<td>30.41</td>
<td>5.37</td>
<td>18.17</td>
<td>20.62</td>
<td>4.22</td>
<td>0.68</td>
<td>3.91</td>
<td>0.05</td>
<td>0.73</td>
<td>0.73</td>
</tr>
<tr>
<td>increment</td>
<td>2.44</td>
<td>3.87</td>
<td>3.29</td>
<td>4.22</td>
<td>4.22</td>
<td>0.01</td>
<td>0.08</td>
<td>0.05</td>
<td>0.73</td>
<td>0.73</td>
</tr>
<tr>
<td>find_in_list</td>
<td>0.54</td>
<td>3.95</td>
<td>4.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rehash</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>string</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3/15/2004 CUCIS 25
ECLAT (ARM)

Eclat is an association rule mining algorithm based on equivalence classes. An equivalence class is a potential maximal frequent itemset. Efficient lattice traversal techniques are used to identify all the true maximal frequent itemsets. All the frequent subsets of the maximal frequent itemsets are generated during traversing.

**Phase 1:** finding equivalence classes
- generate frequent $k$-itemsets $L_k$ (i.e. $k=2$);
- create equivalence classes $F_k$ by joining itemsets in $L_k$, based on their common $k-1$ length prefix;

**Phase 2:** bottom-up traversing lattice
- generate candidate set $C_{k+1}$ based on $F_k$;
- for each transaction $t \in \text{Database}$ do
  - for each $c \in C_{k+1}$ do if ($c$ is a subset of $t$) $c$.count++;
  - $L_{k+1} = \{c \in C_{k+1} \mid c$.count $\geq \text{min}\_\text{sup}\}$
  - create equivalence classes $F_{k+1}$;
- if ($F_{k+1} \neq \emptyset$) then $\text{Bottom}\_\text{Up}(F_{k+1})$;
Function Profiling

- add routine creates an equivalence class (used multiple times in both phases). Less data ref, but most stalls and misses. Better creation of equivalence class is needed.
- get_intersect involves finding common items in order to create a longer itemset. Lot of branches and many are mispredicted as well. Better reorganization of instructions might help.
- process_invert and partition_get_lidxsup are used during the subset calculation (not much of a bottleneck).
Scalability Analysis

Multiprocessor, Scalable Data Analysis
Parallelization

- Goal: to test the scalability of data mining applications on SMPs
- Parallel versions of our benchmark applications.
- Currently 5 parallel applications (out of 8)
  - ScalParC (Zaki’s SMP algorithm),
  - K-means (data parallelism),
  - Fuzzy K-means (data parallelism),
  - HOP (data parallelism), and
  - Apriori (common candidate partitioned database strategy).
- Data sets – as described earlier (Small, Medium, Large).
Execution Times

<table>
<thead>
<tr>
<th>Program</th>
<th>Data set = S</th>
<th>Data set = M</th>
<th>Data set = L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P1</td>
<td>P4</td>
<td>P8</td>
</tr>
<tr>
<td>HOP</td>
<td>6.30</td>
<td>3.50</td>
<td>5.25</td>
</tr>
<tr>
<td>K-means</td>
<td>5.70</td>
<td>2.85</td>
<td>4.38</td>
</tr>
<tr>
<td>Fuzzy</td>
<td>164.10</td>
<td>3.01</td>
<td>6.22</td>
</tr>
<tr>
<td>K-means</td>
<td>3.50</td>
<td>31.70</td>
<td>4.90</td>
</tr>
<tr>
<td>BIRCH</td>
<td>51.00</td>
<td>3.78</td>
<td>4.90</td>
</tr>
<tr>
<td>Apriori</td>
<td>6.10</td>
<td>2.03</td>
<td>2.35</td>
</tr>
<tr>
<td>Eclat</td>
<td>11.80</td>
<td>81.50</td>
<td>-</td>
</tr>
</tbody>
</table>

- ScalParC scales very well, Fuzzy K-Means performs well.
- Balanced partitioning of data in ScalParC => very less memory contention (for shared variables). SMP architecture is utilized.
- Apriori has issues with SMP: hash tree is common and is accessed too frequently.
Overheads increase on parallelization to more processors.
Mainly from OpenMP overheads, program locks.
OpenMP programming environment adds extra execution cycles => affects execution times. Unless speedup is good, OpenMP could be destructive.
- Very small I/O overheads in the program – Bayesian is an exception.
- Bulk read (ScalParC) vs. Character based (Bayesian) read operation. Character reads prove to be costlier in terms of I/O.
- I/O overheads increase with data size
Synchronization time in CPU cycles for all applications. The synchronization time increases when computation is scaled to multiple processors due to increased contention for shared variables.

Synchronization time in CPU cycles for all applications for different datasets. The synchronization time increases when data size is increased (as shared data also relatively increases).
Applications are drastically different in their L1 cache behavior (two types). Random data distribution is seen.

Individual processor study suggested master processor is at times overloaded.

Data size is proportional to data misses (due to limited size of L1).
Erratic behavior
- Data distribution is random as we use dynamic scheduling for parallelization of our applications.
- Load is also unbalanced
- Misses increase as data sizes are increased (cache not evenly used). There is a lot of contention.
Memory access times increase when more processors are used due to the repeated data accesses arising from
- Locks
- L2 cache misses
- Synch time
CPI is less for some applications. Attributed to contentions (program locks, OMP barriers).

Some applications are able to overcome it by reducing total synch times.

<table>
<thead>
<tr>
<th>Programs</th>
<th>P1</th>
<th>P4</th>
<th>P8</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOP</td>
<td>1.53</td>
<td>1.36</td>
<td>1.45</td>
</tr>
<tr>
<td>K-means</td>
<td>1.82</td>
<td>1.56</td>
<td>1.72</td>
</tr>
<tr>
<td>Fuzzy K-means</td>
<td>1.36</td>
<td>1.53</td>
<td>1.61</td>
</tr>
<tr>
<td>BIRCH</td>
<td>1.29</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ScalParC</td>
<td>2.96</td>
<td>2.63</td>
<td>2.61</td>
</tr>
<tr>
<td>Bayesian</td>
<td>1.20</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Apriori</td>
<td>3.83</td>
<td>2.66</td>
<td>3.44</td>
</tr>
<tr>
<td>Eclat</td>
<td>9.59</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Preliminary Conclusions

- MineBench is representative.
- MineBench is diverse: two association rule mining algorithms, two classification algorithms, and four clustering algorithms.
- Data mining algorithms are scalable (SMP).
- OS overhead, the synchronization overhead, and the I/O time are usually small in MineBench applications.
- L1 data cache miss rates are small.
- L2 cache miss rates are high, which results in small instruction-level parallelism (measured in CPI).
- Improvements in the performance of processors are likely to have a significant impact on the overall performance of data mining systems.
- Data fetch techniques like prefetching should also improve the performance of the processor considerably.
- Lot of room for algorithmic and architectural optimizations (targeting emerging scenarios) in existing data mining algorithms.
What more is needed?

- More evaluation – more architectures
- Newer measurement tools, measures of interest
- More algorithms
- More domains, categories
Where do we go after this?

- Function profiling
- Kernel identification
  - Individual
  - Common ones across categories/domains
- Kernel optimizations. Also, individual kernel parallelization.
- More detailed characterization
  - More measures of interest
  - More innovative evaluation schemes/tools
Questions?

Thank you