A Hardware Implementation of Hierarchical Clustering for Text Documents

Dan Legorreta  
dan.legorreta@charter.net

Moshe Looks  
moshe@metacog.org

Shobana Padmanabhan  
sp3@arl.wustl.edu

Abstract—Clustering is a form of unsupervised learning where a collection of unlabeled data points is partitioned into groups (clusters) according to some similarity metric, maximizing similarity intracluster while minimizing similarity intercluster. In hierarchical clustering, the groups are clustered into subgroups, etc., organizing the data as a tree. Many clustering algorithms are superlinear, while pertinent datasets are growing in size exponentially. Since clustering algorithms also tend to be highly parallelizable, there is a strong motivation for speedup via hardware acceleration. We present a simple hierarchical clustering algorithm for text documents, adapt it for hardware acceleration, and implement on the FPX platform. To our knowledge, this is the first work to present a hierarchical clustering algorithm designed for straightforward hardware implementation.

I. MOTIVATION AND OVERVIEW

As databases of interest grow exponentially larger, data-mining algorithms are struggling to keep up (cf. [10], [4]). This is an broad phenomenon, affecting at least the following areas:

- Gene sequences and other bioinformatics data sources
- Sales and other business transactions
- The www
- Various data sources utilized within the intelligence community

Thus, there is compelling motivation to move data-mining algorithms into hardware in order to accelerate them (i.e., take advantage of specialized circuitry). In this paper, we focus on a particular data-mining problem, the hierarchical clustering of text documents based on document similarity. To our knowledge, this is the first work to present a hierarchical clustering algorithm designed for straightforward hardware implementation.¹

Section II briefly reviews the clustering problem in general, with a particular focus on the clustering of text documents. Section III describes our clustering algorithm. The FPX platform and LEON2 Processor, with we have chosen for our implementation, are introduced in Section IV, and their selection justified. In Section VI we then describe how the primary computation of the algorithm (scoring) may be offloaded onto a specialized circuit, describe the speedup in Section VII, and conclude in Section VIII.

II. THE CLUSTERING PROBLEM

The clustering problem is to partition a given set of items into subsets, typically maximizing intracluster similarity while minimizing intercluster similarity, according to a given similarity metric. In hierarchical clustering, the clusters are themselves recursively clustered. Well-known “flat” (as opposed to hierarchical) clustering algorithms include $k$-means and its generalization, expectation maximization [7].

A well-studied special case is where the items correspond to text documents and the similarity metric is based on viewing a document as a “bag-of-words”, or “bag-of-word-disjunctions” [16], [17]. That is, documents are points in a high-dimensional space where each dimension corresponds to the frequency/occurrence of one or more words in the document.

There are a number of important factors to consider in crafting an approach to text document clustering which have been identified in the literature:

¹A number of versions of the flat (non-hierarchical) $k$-means algorithm have been implemented in hardware, most frequently for image clustering (cf. [5]).
The similarity measure used is as important as the algorithm chosen; normalized measures such as the cosine measure,

\[ s(\vec{x}, \vec{y}) = \frac{\vec{x} \cdot \vec{y}}{||\vec{x}||_2 \cdot ||\vec{y}||_2}, \tag{1} \]

are better suited than unnormalized distance metrics (i.e., Euclidean or Manhattan) [17].

Hierarchical top-down approaches where the set of documents is recursively bisected outperform flat clustering algorithms such as k-means, as well as hierarchical bottom-up clustering methods based on agglomeration [16].

Approaches based on the occurrence of phrases (n-grams) generally do not outperform simpler approaches based on single words [3].

The dimensionality of the space utilized may be reduced by a number of means without adversely impacting the quality of the results [15].

III. CLUSTERING VIA HIERARCHICAL PARTITIONING

Hierarchical Partitioning (HP) clustering may be defined as follows:

- **Input**: A set of \(N\) \(k\)-dimensional points (vectors) in a binary space.
- **Output**: A binary tree where each point corresponds to exactly one leaf.

Partition the points (from the root of the tree downward) into sets \(\text{left}\) and \(\text{right}\), maximizing \(\text{score(}\text{left}\text{)} + \text{score(}\text{right}\text{)}\). The algorithm terminates when \(\text{left}\) and \(\text{right}\) are individual points. The function \(\text{score}\) maps sets of points to reals, and measures cohesiveness. Specifically, for a set of points \(C = \{\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n\}\), let

\[ \text{sum}(C) = \sum_{i=1}^{n} \vec{a}_i \tag{2} \]

in

\[ \text{score}(C) = \frac{1}{n} \sum_{i=1}^{n} \frac{\text{sum}(C) \cdot \vec{a}_i \cdot k}{|\vec{a}_i|}, \tag{3} \]

Figure 1 shows the clustering results for a subset of 160 documents from the popular 20-newsgroup dataset [1] (the first 40 documents each from the groups rec.sport.hockey, rec.sport.baseball, talk.politics.mideast, and rec.motorcycles). The tree has been automatically pruned based on cluster quality at each level. Data dimensionality is reduced to 4000-wide vectors (from a dimensionality in the hundreds of thousands) via a word mapping table as described in [10].

A. Performance Considerations

To approximate the optimal partitioning, hill-climbing is used; the set of points is randomly partitioned by randomly assigning each point to either \(\text{left}\) or \(\text{right}\). Moves of a single point between \(\text{left}\) and \(\text{right}\) which increase the sum of the scores are taken until no more such moves are possible (i.e., a local minimum is reached).

To avoid using floating-point numbers, \(\text{score}(C)\) is approximated by

\[ \left(\frac{1}{n} \sum_{i=1}^{n} \frac{\text{sum}(C) \cdot \vec{a}_i \cdot k}{|\vec{a}_i|}\right). \tag{4} \]

The magnitudes, \(|\vec{a}_i|\), may be easily precomputed and cached in an array. For each computation of \(\text{score}(C)\), we only need to compute \(\text{sum}(C)\) once. Each entry in the summation vector is stored in only eight bits. Thus, we obtain the following C code:

```c
score_t score(iterator from, iterator to) {
    score_t sc=0;
    base_t sum[K];
    iterator it;
    set_all_to_zero(sum);
    for (it=from; it!=to; ++it)
        add_to(*it, sum);
    for (it=from; it!=to; ++it)
        sc+= (K*dot_prod(sum, *it)) / magnitude(*it);
    sc/=distance(from, to);
    return sc;
}
```

The asymptotically dominant operations here are:

1) Calls to the `add_to` function, which adds a \(k\)-wide vector of `base_t`s with a \(k\)-wide vector of bits.

2) Calls to the `dot_prod` function, which computes the dot product of a \(k\)-wide vector of `base_t`s with a \(k\)-wide vector of bits.
Fig. 1. Clustering results for a subset of the 20-newsgroup dataset. In parentheses are the total number of documents in a node’s subtree. Leaves show the most category present (e.g., motorcycles) and its frequency.

Experimentally, almost all of the algorithm’s execution time is evenly divided between add, dot, and dot_prod.

IV. THE LIQUID ARCHITECTURE PLATFORM AND THE LEON2 PROCESSOR

Liquid architecture platform is developed by the Liquid architecture group [9] of Washington University. The following description is from [14].

The platform aims to measures and improves application performance, by providing an easily and efficiently reconfigurable architecture, along with software support to expedite its use.

The Liquid architecture system has been implemented as an extensible hardware module on the Field-programmable Port Extender (FPX) platform [12], which is described in Section V.

The Liquid platform instantiates LEON V2, a soft core processor used by European Space Agency [6] for embedded systems development. LEON is open source and hence can be downloaded for free.

A. LEON Processor

Figure IV-A shows the fairly sophisticated computer architecture model of LEON - separate instruction and data caches, full SPARC V8 instruction set, 5-stage pipeline, generic co-processor interface and separate (standard AMBA) buses for high-speed memory access and low-speed peripheral control.

We use the co-processor interface to integrate our scoring circuit.

B. Co-processor Interface

The following description of the co-processor interface is from LEON user manual [2].

The generic co-processor interface of LEON allows an execution unit to operate in parallel to increase performance. One coprocessor instruction can be started each cycle as long as there are no data dependencies. When finished, the result is written back to the co-processor register file.

The timing (waveform) diagram for the execution unit interface can be seen in Figure V. The execution unit is started by asserting the start signal together with a valid opcode. The operands are driven on the following cycle together with the load signal. If the instruction will take more than one cycle to complete, the execution unit must drive busy from the cycle after the start signal was asserted, until the cycle before the result is valid. The result, condition codes and exception information are valid from the cycle after the de-assertion of busy, and until the next assertion of start. The opcode (cpi.opcode[9:0]) is the concatenation of bits [19,13:5] of the instruction. If execution of a co-processor instruction need to be prematurely aborted (due to an IU trap), cpi.flush will be asserted for two clock cycles. The execution unit should then be reset to its idle condition.
C. Implementation of Scoring through Co-processor Interface

Implementing the integration of our scoring circuit through the co-processor interface, involved the following.

We thank Phillip Jones of Liquid architecture group for his help with integrating our scoring circuit through the co-processor interface.

- device.vhd was modified to enable co-processor interface.
- hclust1eu.vhd was created based on fp1eu.vhd provided with LEON2 distribution. This component gets instantiated in proc.vhd. It contains the implementation of co-processor interface integration, in particular, the co-processor register file management which is being done in the opcode decode process.
 Opcode encodings for scoring circuit

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCLUST_STG1</td>
<td>111100001</td>
</tr>
<tr>
<td>HCLUST_STG2</td>
<td>111100010</td>
</tr>
<tr>
<td>HCLUST_STG1_RESET</td>
<td>111100110</td>
</tr>
<tr>
<td>HCLUST_STG3</td>
<td>111100011</td>
</tr>
<tr>
<td>HCLUST_RESET</td>
<td>111111111</td>
</tr>
<tr>
<td>HCLUST_NOP</td>
<td>111100000</td>
</tr>
</tbody>
</table>

- `sparcv8.vhd` was modified to include opcode definitions for the scoring circuit. Opcodes are defined for stage1, stage2, stage1_reset, stage3, reset and nop. Their encodings are shown in Figure IV-C.
- `hcluster_cp_core.vhd` was created to instantiate the scoring circuit. The scoring circuit itself is defined Section VI.

D. Cross Compiler

Liquid platform uses sparc-elf cross compiler suite distributed with LEON2. The following is a description in the distributed document.

The compiler is based on the GNU compiler tools and the Newlib standalone C-library. The cross-compiler allows compilation of sequential (non-tasking) C and C++ applications. It supports both hard and soft floating-point operations, as well as both V7 and V8 multiply and divide instructions.

Packages distributed in the suite include:
- GNU GCC C/C++ compiler v3.2.3
- Newlib C-library v 1.12.0
- Libio low-level I/O routines for LEON2
- GDB debugger v5.3

E. Liquid Platform User Interface

Liquid platform provides a web interface as well as command line options for the following:
- LEON status - to check if LEON has started up.
- Load program - to load a program into memory, at a specific address.
- Start LEON - to instruct LEON to execute the program that has been loaded into memory at a given address.

- Read memory - to read a specified number of bytes at a given address. This can be used to verify the program that was loaded, or to read the results of a program which writes results to memory.
- Get statistics - to get program execution time (number of hardware clock cycles), cache statistics (number of cache reads, writes, hits, misses), etc.
- Reset LEON - to reset LEON.

V. FPX PLATFORM

The following description is from [14].

The FPX platform, developed by Washington University’s Reconfigurable Network Group [13], provides an environment where a circuit implemented in FPGA hardware can be interfaced with SRAM, SDRAM, and high speed network interfaces. Hardware modules on the FPX can use some or all of a large Xilinx Virtex XCV2000E FPGA to implement a logic function [8]. By using the FPX platform, resulting hardware modules can be rapidly implemented, deployed and tested with live data [11].

The Liquid architecture system leverages FPX platform to evaluate customizations of the soft core processor and to control experiments remotely over the Internet.
VI. A SCORING CIRCUIT

We would thus like to design a specialized scoring circuit to speed up hierarchical partitioning (i.e., parallelization). Consider a set of \( n \) documents, represented as vectors in a \( k \) dimensional space. Assume that we can parallel-process on words of \( m \) bits at a time, and that \( m < k \). We can consider a \( k \)-dimensional vector \( \vec{a} \) as \( k/m \) \( m \)-dimensional subvectors \( \vec{a}^1, \vec{a}^2, ..., \vec{a}^{k/m} \), where \( \vec{a}^i \) contains \( \vec{a} \)'s values along dimensions \( i(k - 1) + 1 \) through \( ik \).

Our asymptotically dominant operations – computing the dot product of each vector with the sum of all vectors in a set – can occur in \( k/m \) independent \( m \)-bit wide slices (if \( k \) is not a multiple of \( m \), we can zero-pad it). Specifically, for every \( \vec{a} \) in a set,

\[
(\sum_{i=1}^{n} \vec{a}_i) \cdot \vec{a}
\]

will be computed as

\[
\sum_{i=1}^{k} \sum_{j=1}^{k/m} \vec{a}_j \cdot \vec{a}^i.
\]

In our current implementation for LEON2 (see above), \( m = 64 \). Slices are processed one at a time (i.e., left-to-right, from 1 to \( m/k \)). Since the coprocessor interface allows for two 64-bit values as simultaneous inputs, corresponding slices from two vectors are sent simultaneously, until a corresponding slice has been sent for every vector in the set (if there is an odd number of vectors, we simply send a vector of zeros along with the first vector). After the \( i \)th slice has been sent, the scoring circuit may compute \( \sum_{j=1}^{k} \vec{a}_j \), and, for each vector \( \vec{a}_i \), \( (\sum_{j=1}^{k} \vec{a}_j) \cdot \vec{a}^i \). These partial dot products are accumulated as more slices are processed, such that when the entire set of vectors has been sent, the full dot product be may read back for every vector. These values are then normalized (divided by the magnitude of the corresponding vector) and summed to obtain the final score.

The circuit was implemented as a coprocessor to the LEON processor on the liquid architecture. The coprocessor communicates with LEON via the coprocessor interface using commands comprised of two 64 bit inputs, a 9 bit opcode, and a 64 bit result. Additionally, the coprocessor is limited in that it cannot access main memory itself, it must go through LEON anytime a memory access is necessary. The circuit is responsible for performing two major operations of the clustering algorithm: computing the bitwise sum of the document bit-vectors and calculating the dot-product of each of the documents and the bitwise sums. These two operations are accomplished in three separate stages. The first stage computes the bitwise sum, the second stage computes the dot product, and the third stage returns the final score for each of the documents. The following circuit description uses as an example a cluster comprised of four documents (A,B,C,D) each of which is represented as a bit-vector of length 4000. The first stage of the circuit begins by receiving the first 64 bits of the first two documents in the cluster (A & B). Each bit of the first document is then added to the corresponding bit in the second document and the result is stored to a register. This creates a total of 64 registers containing the bitwise sum of the first 64 bits of the first two documents. As the sums are being computed, the two 64 bit portions of the two documents are also stored in a FIFO local to the coprocessor (FIFO1). This allows for the coprocessor to access this data in the future without having to go through LEON. This entire process is then repeated with the first 64 bits of the third and fourth documents (C & D). The registers containing the bitwise sums would be updated to reflect not only the sums of the first two documents, but also of the third and fourth documents. It will eventually be necessary to repeat this stage (4000/64) * 4 = 250 times, each call reading in the next 64 bit s of each document until the entire document has been processed. After the first 64 bits of all of the documents in the cluster have passed through the first stage of the circuit, the second stage is ready for execution. This stage is responsible for computing the partial dot-product of each of the documents and the bitwise sums. The stage begins by first copying the 64 bitwise sums computed in stage one. Next, it reads the first 64 bits of the first document from FIFO1 and computes the dot-product of it and the 64 bitwise sums. This partial dot-product of the document is then stored in an additional FIFO (FIFO2). This entire process continues until all of the documents entered in stage one have had their partial dot-product computed.
After the first two stages of the circuit have been executed the required number of times, the final stage of the circuit is ready for execution. Stage three reads each of the partial dot-products from FIFO2 (which at this point have become the full dot-products) and returns it as the result to LEON. The stage that the scoring circuit executes is based on opcodes that are sent in the software developer’s code. Each of the aforementioned 3 stages has a corresponding opcode which the software developer sends in order to start those stages. In addition to the opcodes for the three main stages, there are three additional opcodes: no-operation opcode, stage one reset opcode, and a full reset opcode. The following is a pseudo code example of how these opcodes would be used to score the previously mentioned example cluster.

```plaintext
for index=0 to 64
    Stg1[ A(index to index+63), B(index to index+63) ]
    Stg1[ C(index to index+63), D(index to index+63) ]
    Stg2[ ]
    Stg1Reset[ ]
// 1 nop needed for each document
NOP[ ]
NOP[ ]
NOP[ ]
NOP[ ]
ScoreForDocA = Stg3[ ]
ScoreForDocB = Stg3[ ]
ScoreForDocC = Stg3[ ]
ScoreForDocD = Stg3[ ]
FullReset[ ]
```

VII. PERFORMANCE GAINS

To estimate the performance gain from the circuit, let’s first consider the case of the unaccelerated algorithm; we need to first compute \( \sum a \), then compute \( \sum \cdot a \) for every \( a \) in our set. Given a set of \( n \) \( k \)-dimensional vectors, the summation requires \( k \) add operation for every vector, giving a total of \( nk \) operations (we are not counting operations that are common to both the unaccelerated and accelerated versions). Computing the dot products requires an add and a multiply for each element (thus \( 2nk \) operations)—giving an overall computational overhead of \( 3nk \) operations.

For the accelerated circuit, we require \( (1.5nk)/64 \) operations to send the data to the circuit, simultaneously computing \( \sum a \). Computing the dot-products is pipelined with this operation, so the addition overhead from this step is only \( 2k/64 \) operations. Finally, \( 4n \) operations are required to read back the results. The total is thus \( 0.023nk + 0.31k + 4n \) operations.

The actual speedup for clustering \( N \) vectors will depend on \( k \), as well as how deep a clustering hierarchy we wish to construct. Consider \( k = 4000 \); in this case, the unaccelerated algorithm will take 12000 operations, while the accelerated version will take \( 0.023\cdot4000 \cdot k + 0.31\cdot4000 + 4n = 99n + 125 \) operations. Thus, the speedup factor will be between 54 (12000/54) and 121 (12000/99).

VIII. CONCLUSIONS

We have presented novel work which addresses the demands of high-volume data mining by moving the bottleneck operations of a hierarchical clustering algorithm to specialized hardware. A significant speedup is achieved via this specialization. We expect that future implementations following this design methodology on more powerful hardware will be able to exploit an even greater level of parallelism, and to significantly outperform commodity PCs.

REFERENCES

Fig. 5. The scoring circuit


