Parallel Algorithms for Hierarchical Clustering and Cluster Validity
Xiaobo Li

Abstract—This correspondence proposes parallel algorithms on SIMD machines for hierarchical clustering and cluster validity computation. The machine model uses a parallel memory system and an alignment network to facilitate parallel access of both pattern matrix and proximity matrix. For a problem with \( N \) patterns, the number of memory accesses is reduced from \( O(N^3) \) on a sequential machine to \( O(N^2) \) on an SIMD machine with \( N \) PE's.

Index Terms—Alignment network, cluster validity, hierarchical clustering, parallel algorithms, SIMD machine.

I. INTRODUCTION

In exploratory data analysis, partitional clustering divides the data set into several clusters according to some criterion, and hierarchical clustering produces a hierarchy of clusters represented by a dendrogram. Both partitional and hierarchical clustering require intensive computation, even for a modest number of patterns. Several parallel approaches have been developed to take advantage of the computing power of multiple processor systems. Parallel clustering algorithms have been proposed on hypercube SIMD machines with local memory modules [8]. Parallel access of both the pattern matrix and the proximity matrix poses an interesting and challenging task. This correspondence describes an SIMD machine model with an alignment network connecting the array of processing elements (PE's) and a shared memory system [5], and proposes hierarchical clustering algorithms on this machine. It also presents a parallel cluster validity algorithm for computing one measure of global fit (CPCC) between the result dendrogram and the original proximity.

Many numerical computation problems with matrices use rows, columns, and diagonals as aggregate data structures. Although clustering algorithms and cluster validity algorithms also use matrices, they often involve only half of a matrix. Instead of using twice as much memory and twice as many accessing cycles as necessary, this correspondence presents several accessing schemes for the matrices stored in memory which are suitable for clustering type applications. These storage schemes and accessing methods can also be used in other pattern analysis applications.

Section II gives a simple definition of the clustering procedures and notations. The SIMD computer model is introduced in Section III. Section IV presents a parallel hierarchical clustering algorithm, including the global fit computation, for a conceptually simple case. Section V gives the generalized clustering algorithm, which allows different problem sizes and machine sizes. A brief conclusion is given in Section VI.

II. HIERARCHICAL CLUSTERING

Pattern analysis problems start with either a pattern matrix \( P (N \times M) \) or a proximity matrix \( D (N \times N) \). Each of the \( N \) rows of \( P \) represents a pattern (data object), and each of the \( M \) columns represents a feature (characteristic measure). Let \( P(i) \) represent the \( i \)th row of \( P \) for pattern vector \( i \) and \( p(i,j) \) represent the \( i,j \)th entry of \( P \) for the \( j \)th feature of pattern \( i \). The entry \( D(i,j) \) is a proximity index such as the Euclidean distance between pattern \( i \) and pattern \( j \). In this correspondence, we only consider Euclidean distance, and \( D(i,j) \) is set to \( \infty \) and is to be excluded from the computation of minimum. When we refer to a general matrix, either \( P \) or \( D \), notation \( A \) is used. We consider single-link and complete-link algorithms (described by the Johnson scheme [6]) because of their popularity in various applications. If the number of patterns is \( N \), there are \( N - 1 \) levels. Each level involves finding a minimum from matrix \( D \) with time complexity \( O(N^2) \). Therefore, the total time complexity of the direct sequential implementation of the Johnson scheme is \( O(N^3) \). In the case of single-link clustering, if the dissimilarity of the nearest neighbor to each pattern is stored so that the process of finding global minimum from \( D \) at each iteration is faster (\( O(N) \) time), the time complexity can be easily reduced to \( O(N^2) \).

A cluster validity study is usually performed after obtaining the clustering result [6]. To validate a resulting dendrogram, some measure of correspondence between the dendrogram and the original proximity matrix has to be computed. We use an \( N \times N \) cophenetic matrix \( C = [C(i,j)] \), where \( C(i,j) \) is the first level in the clustering dendrogram at which pattern \( i \) and pattern \( j \) occur in the same cluster. For the usual case of interval proximity, the well-known CPCC, the product-moment correlation coefficient between the entries of \( D \) and \( C \) is used. In our algorithm here, we have both the cophenetic matrix and an \( N \times 2 \) matrix, node, as the clustering result. With node, the dendrogram can be easily generated for presentation.

III. AN SIMD COMPUTER MODEL

We consider an SIMD computer which consists of \( Q \) PE's and \( P \) memory modules where \( Q = 2^p \). A central control unit CU broadcasts an instruction to all PE's, and all enabled PE's simultaneously execute this instruction. An alignment network is used to connect PE's and memory modules, and is also used for the PE-PE communication, since these two communications never require the network at the same time in the clustering algorithms considered here. Fig. 1 illustrates this computer model. It is basically the model of Configuration II in [5, p. 326]. The PE's are indexed 0 through \( Q-1 \) with index \( r = r_0r_1r_2\cdots r_p \). Each PE has several registers, denoted by \( R(X), R(0), R(1), \ldots \), Register RX always stores the PE index, i.e., \( RX(r) = r \). Location \( z \) of memory module \( s \) (mem \( s(z) \)) is denoted by mem \([s,z] \). The alignment network is the standard shuffle-exchange network, chosen for its simplicity and powerful functions [11]. In describing parallel algorithms, we follow the notation common in the literature [2], [6], i.e., \( " := " \) for broadcast, \( " \leftarrow " \) for local assignment, and the symbol \( " \Rightarrow " \) denotes transfer through the network. The inter-PE communication statements \( R2(\text{shuff}(r)) \leftarrow R2(r) \), \( R2(\text{unshuff}(r)) \leftarrow R2(r) \), and \( R2(r^{10}) \leftarrow R2(r) \) can be performed in one pass of the network. Each of the statements \( R2(r \oplus c) \leftarrow R1(r) \), \( R2(r \oplus c) \leftarrow \text{mem} [s,z] \), and \( \text{mem}[r \oplus c,z] \rightarrow R(r) \) can be performed in log \( Q \) shuffle-exchange passes, where \( c \) is the network control vector, \( z \) is the local address in that memory module, and the logic operation \( \oplus \) is defined as a bit-wise exclusive-OR operation. Note that the time required to access memory modules is longer than that for PE registers. Each module has a control unit and a few registers and can carry out logical (exclusive-OR) and arithmetic (comparison) operations for local address computation. All the address computations in the modules are executed in parallel.

We now describe the parallel memory system for a simple case where an \( N \times N \) matrix \( A \) is to be stored, where \( N = Q = 2^p \).
This simple case can help us explore maximum parallelism in memory access and operation. More general cases can be derived easily and are given in later sections. We store matrix $A$ according to the rule first stated in [1].

Storage Rule 1: To store matrix $A$ of size $N \times N$ in $Q(=N)$ memory modules, $A(x,y)$ is stored at location $y$ of module $x$. As it turns out, location $y$ of $mem(x\oplus y)$ stores matrix element $A(2x\oplus y)$. Fig. 2 illustrates the storage rule for matrix $A$ when $N = 16$. For example, $A(3,6)$ is stored in module 5. The rows, columns, and some other subsets ("stencils") of this matrix can be accessed without memory conflict [1]. If the matrix stored in parallel memory modules is a pattern matrix, parallel fetch of a row provides an easy access of a pattern; parallel fetch of a column provides an easy access of a feature. A procedure for fetching row $x$ based on Storage Rule 1 can be given as $R(s \oplus x) \leftarrow mem[s, max(x, s \oplus y)]$ ($s > 0$). The simple enable mask ($s = 0$) ensures that only $N - 1$ fetches are executed in this procedure. The PE connected to $mem(0)$ receives nothing and sets its register to $\infty$. Procedure store_upper-fold can be given by reversing the arrow in Procedure fetch_upper_fold1. Procedures fetch_lower-fold1 and store_lower-fold use the same statement of local address computation as above, with "$min$" replacing "max." Since "$\oplus$" and "max" take one unit-step each, any one of these four procedures requires one memory access and $2 \log Q + 2$ unit-steps.

The computation of CPCG, we need to access every element of the half matrix $D$ (and the half matrix $C$) exactly once. If the above procedures are used, each element will be accessed twice. Since the order of summation is immaterial, we propose the following accessing mechanism.

Definition: Upper-fold $x$, $0 \leq x < N$, can be given as $R(s \oplus x) \leftarrow mem[s, max(x, s \oplus y)]$ ($s > 0$). The simple enable mask ($s = 0$) ensures that only $N - 1$ fetches are executed in this procedure. The PE connected to $mem(0)$ receives nothing and sets its register to $\infty$. Procedure store_upper-fold can be given by reversing the arrow in Procedure fetch_upper_fold1. Procedures fetch_lower-fold1 and store_lower-fold use the same statement of local address computation as above, with "$min$" replacing "max." Since "$\oplus$" and "max" take one unit-step each, any one of these four procedures requires one memory access and $2 \log Q + 2$ unit-steps.

The computations of CPCG, we need to access every element of the half matrix $D$ (and the half matrix $C$) exactly once. If the above procedures are used, each element will be accessed twice. Since the order of summation is immaterial, we propose the following accessing mechanism.

Procedure fetch-pcl can be given by changing $\phi_{unit}$ to $\phi_{unit}$. We use the number of operators appearing in that statement. Different machines have different processing times for various operations, and this correspondence intends to propose the general algorithm, thus the time complexity given here serves only as an approximation. The time complexity given here serves only as an approximation. The time complexity given here serves only as an approximation. The time complexity given here serves only as an approximation.
We use two procedures (minQ.a, minQ.b) for computing the minimum across the PE’s. Both procedures involve several registers of each PE. Vector \( B \) denotes a set of \( Q \) values, which could be a row, column, or fold of \( A \), stored in a specified register of the PE’s. The input to Procedure \( \text{minQ.a} \) is \( R(x) = \{\min_y B(y)\} \) and \( R(z) = x \). The key steps are shuffle-exchange pairs \( R(z) \leftrightarrow f(e(x)) \rightarrow R(1) \) and \( R(x) \rightarrow \text{Ry}(z) \). Procedure \( \text{minQ.b} \) is similar to \( \text{minQ.a} \) but passes one more index in \( R4 \). Procedure \( \text{minQQ} \) which computes the minimum of the entire \( Q \times Q \) matrix \( D \), employs a loop of \( N \) iterations. Each iteration fetches a row from matrix \( D \) and applies \( \text{minQ.a} \). Procedure \( \text{minQ.b} \) is called after the loop. Registers \( R5 \) and \( R6 \) are reserved for indexes which correspond to the local minimum. This information is specifically useful for single-link clustering. After each iteration, only one pattern per new cluster needs to compute its nearest neighbor. Table I lists the time complexities of all procedures in Sections IV with and without the operations of matrix \( C \). The time complexities are generally higher with \( C \). Using \( C \) we can have an NN index for each pattern \( P(i) \) and only store those distances. The following procedure merges two clusters and updates \( D \).

```
Procedure \( \text{mergel} \) \( (k, \text{func}) \):
begin
  //Phase 1: update the dendrogram record in module 0
  z:=k+3; mem(s,z)\(+R4(s@0)
  //Phase 2: update matrix \( D \)
  fetch_upper-foldl(R2, R7); fetch_upper-foldl(R4, R9);
  for x=0 to \( N-1 \) do
    if R12=R4 then //update R12 of related patterns
      R5:=\( RX = RX \)
    else
      //Phase 3: update the cophenetic matrix \( C \)
      \( R_s(e, s)=R_s(RX) \)
      for x=0 to \( N-1 \) do
        fetch_upper-foldl(x, R21) (R12=R2);
        store_upper-foldl(R2, R7); store_upper-foldl(R4, x) (R9 \( \neq \infty \));
        //Phase 4
        for y=0 to \( N-1 \) do
          R5:=\( RX = RX \)
        endfor;
  end;
end;
```

Fetching upper-folds instead of entire rows or columns has two advantages: 1) Half the original proximity matrix is modified, the other half is left unchanged for the future cluster validity computation. 2) Only half the memory accesses are required. Procedure \( \text{SINGLE-LINK}1 \) can be given as basically a loop with each iteration merges two clusters by accessing upper-folds. At each of the \( N-1 \) iterations, a merge is performed to eliminate a cluster and to update matrix \( D \). Procedure \( \text{minQ.a} \) finds the local minimum among the entries of \( D \) related to the newly formed cluster, then Procedure \( \text{minQ.b} \) locates the minimum among the nearest neighbors. If the cophenetic clustering method is used, a global minimum has to be searched for after each merge. It is not enough to just look at the set of nearest neighbor distances, because the distance from a cluster to the newly formed cluster can get larger. Procedure \( \text{COMPLETE-LINK1} \) using \( \text{minQ.b} \) is straightforward and similar to the sequential version.

The cluster validity procedure \( \text{VALID1} \) computes CPCC between the cophenetic matrix \( C \) of a dendrogram and the original proximity matrix \( D \).

```
Procedure \( \text{VALID1} \):
begin
  R5:=0; R7:=0; R9:=0; R11:=0; R13:=0;
  for x=0 to \( N/2-1 \) do
    //compute the sums for each partial-row-pair and partial-column-pair
    //fetch_prl(x, R1); fetch_pcl(x, R3);
endfor;
```

<table>
<thead>
<tr>
<th>Table I</th>
<th>Time Complexity of Each Procedure (( N ) is the Number of Patterns, ( M ) is the Number of Features, ( Q ) is the Number of PE's and Memory Modules, and ( q = \log_2 Q ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure</td>
<td>( \Phi_{\text{com}} )</td>
</tr>
<tr>
<td>fetch_row1</td>
<td>1</td>
</tr>
<tr>
<td>fetch_upper-fold1</td>
<td>1</td>
</tr>
<tr>
<td>fetch_prl</td>
<td>1</td>
</tr>
<tr>
<td>minQ.a</td>
<td>0</td>
</tr>
<tr>
<td>minQ.b</td>
<td>0</td>
</tr>
<tr>
<td>minQQ</td>
<td>( N )</td>
</tr>
</tbody>
</table>

(without \( C \) computation)

- merge1: \( 7N + 9 \)
- \( \Phi_{\text{com}} = 39N + 22N - 39.5q - 20 \)
- \( \Phi_{\text{add}} = 1.5N^2 - 1.5N \)
- \( \text{COMPLETE-LINK1} \)
- \( N^2 + 6N - 7 \)
- \( 8Nq + 4N^2 + 14.5Nq + 11N - 22.5q - 14 \)
- \( \Phi_{\text{com}} = 0.5N^3 + 3N^2 - 2.5N - 1 \)
- \( \Phi_{\text{add}} = 0.5N^3 + 2N^2 - 2N^2 - 2.5N + 1 \)

(with \( C \) computation)

- merge1: \( 2N + 7 \)
- \( \Phi_{\text{com}} = 4Nq + 5N + 14q + 16 \)
- \( \Phi_{\text{add}} = 2N^2 + 7N - 9 \)
- \( 4Nq + 5N + 35q + 21N - 39.5q - 24 \)
- \( N^3 + 2.5N^2 - 2.5N - 1 \)
- \( \Phi_{\text{com}} = 0.5N^3 + 0.5N^2 - N \)
- \( \Phi_{\text{add}} = 12Nq + 9N^2 + 10.5Nq + 10N - 22.5q - 18 \)
- \( \Phi_{\text{com}} = 1.5N^3 + 1.5N^2 - 1.5N - 1 \)
- \( \Phi_{\text{add}} = 2N + 1 \)
- \( \Phi_{\text{com}} = 2Nq + 7N + 10q + 20 \)
- \( \Phi_{\text{add}} = N^2 - 5N + 16 \)
- \( \Phi_{\text{com}} = 3.5N^2 + 1.5N^2 - 1.5Nq - 0.5N - 4q - 1 \)
- \( \Phi_{\text{add}} = MN^2 + 0.5N^2 - MN - 0.5N \)

\( R5=R5+R1*R3; R7=R7+R1; R9=R9+R1*R1; R11=R11+R3; R13=R13+R3*R3; \)

for y=1 to \( \log_2 Q \) do // compute the sums
  \( R15(\text{shuffle}(r))\rightarrow R5(r); R17(\text{shuffle}(r))\rightarrow R7(r); R19(\text{shuffle}(r))\rightarrow R9(r); R21(\text{shuffle}(r))\rightarrow R11(r); R23(\text{shuffle}(r))\rightarrow R13(r); R5(\text{shuffle}(r))\rightarrow R15(r); R7(\text{shuffle}(r))\rightarrow R17(r); R9(\text{shuffle}(r))\rightarrow R19(r); \)
The time complexity of this distribution is $O(NM)$. The cases with $M < Q$, especially the cases with $M \leq Q$, are treated in Section V. A procedure $PtoD1$ for directly computing $D$ from $P$ consists of a double loop of fetching rows and log $Q$ shuffle-exchange pairs for computing the sum. Euclidean distance is assumed. It can be modified easily for other types of dissimilarities. The only operation for computing the sum is $\sqrt{R_1}$.

The number of patterns $N$ is usually in the range of hundreds, such as the well-known IMOX data ($N = 192$) [10], a pattern matrix of questionnaire data used in [9] ($N = 145$), or the number of pixels being clustered in a range image ($N = 1000$) [6]. Parallel machines with over 1000 PE's, such as the Connection Machine ($Q = 65536$) and MPP ($Q = 16384$), are widely used. Machines with 1024 32-bit PE's have already been built, and machines with 16000 or more PE's are becoming more available. Therefore, $N > Q$ (cases 1, 2, 3, 4) becomes more likely. The procedures proposed in Section IV can be used in all these cases, with some PE's setting their registers (e.g., R1 in most cases) to $\infty$. The time complexity analysis is the same. Note that the number of memory fetches for each procedure is a hypercube machine. This procedure could be converted to apply to shuffle-exchange networks.

### V. General Parallel Algorithms

This section discusses the cases 2)–6) referred to at the beginning of Section IV.

#### A. When $M < Q$

The algorithm proposed in Section IV assumes that matrix $P$ or matrix $D$ is stored in the parallel memory modules. In most applications, $P$ or $D$ is stored in the control memory $con_mem$ first, and is distributed to the modules later. This distribution is sequential and is not of interest to us. As an exploratory analysis tool, the clustering is only one of the first steps of data processing. Before and after clustering, other treatments of $P$ and $D$ are necessary. Therefore, the initial sequential data distribution does not slow down the entire process very much. However, we could take advantage of the communication from $con_mem$ to make the computation of $D$ from $P$ more efficient. When $M < Q$, we could use dummy features with the $PtoD1$. When $M \leq Q$, especially when $M < 10$, or $M \approx \log Q$, $PtoD1$ is inefficient. A procedure $PtoD2$ with complexity $O(NM + N \log Q)$ can be given to directly compute $D$ when we send the feature values from the control memory $con_mem$ to all PE’s. A more complex procedure for exchanging data elements among PE’s to compute $D$ from $P$ is given in [8] in the context of hypercube machines. This procedure could be converted to apply to shuffle-exchange networks.

### B. When $N < Q$

The number of patterns $N$ is usually in the range of hundreds, such as the well-known IMOX data ($N = 192$) [10], a pattern matrix of questionnaire data used in [9] ($N = 145$), or the number of pixels being clustered in a range image ($N = 1000$) [6]. Parallel machines with over 1000 PE’s, such as the Connection Machine ($Q = 65536$) and MPP ($Q = 16384$), are widely used. Machines with 1024 32-bit PE’s have already been built, and machines with 16000 or more PE’s are becoming more available. Therefore, $N > Q$ (cases 1, 2, 3, 4) becomes more likely. The procedures proposed in Section IV can be used in all these cases, with some PE’s setting their registers (e.g., R1 in most cases) to $\infty$. The time complexity analysis is the same. Note that the number of memory fetches for each procedure is a hypercube machine. This procedure could be converted to apply to shuffle-exchange networks.

#### TABLE II Computation Times of Clustering Algorithms

<table>
<thead>
<tr>
<th>Parameters</th>
<th>without C</th>
<th>Parallel</th>
<th>Sequential</th>
<th>with C</th>
<th>Parallel</th>
<th>Sequential</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu$</td>
<td>$Q$</td>
<td>$N$</td>
<td></td>
<td>$\mu$</td>
<td>$Q$</td>
<td>$N$</td>
</tr>
<tr>
<td>2</td>
<td>256</td>
<td>100</td>
<td>34846</td>
<td>104148</td>
<td>441142</td>
<td>2554398</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>89758</td>
<td>665950</td>
<td>2767258</td>
<td>4209150</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>100</td>
<td>42567</td>
<td>104148</td>
<td>528063</td>
<td>2554398</td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>171567</td>
<td>1676598</td>
<td>7995563</td>
<td>16877598</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>439887</td>
<td>11001342</td>
<td>5174207</td>
<td>268475603</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>256</td>
<td>100</td>
<td>36628</td>
<td>193446</td>
<td>482524</td>
<td>4603896</td>
</tr>
<tr>
<td>256</td>
<td>95438</td>
<td>1273980</td>
<td>3032968</td>
<td>76182780</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>44349</td>
<td>193446</td>
<td>569445</td>
<td>4603896</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>178749</td>
<td>3113796</td>
<td>8639145</td>
<td>289675596</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>458301</td>
<td>20431356</td>
<td>35592649</td>
<td>4802357009</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### TABLE II Computation Times of Clustering Algorithms

<table>
<thead>
<tr>
<th>Parameters</th>
<th>without C</th>
<th>Parallel</th>
<th>Sequential</th>
<th>with C</th>
<th>Parallel</th>
<th>Sequential</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu$</td>
<td>$Q$</td>
<td>$N$</td>
<td></td>
<td>$\mu$</td>
<td>$Q$</td>
<td>$N$</td>
</tr>
<tr>
<td>2</td>
<td>256</td>
<td>100</td>
<td>713692</td>
<td>1569249</td>
<td>1119988</td>
<td>4024999</td>
</tr>
<tr>
<td>256</td>
<td>4622896</td>
<td>25622655</td>
<td>7300396</td>
<td>67236655</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>876547</td>
<td>1569249</td>
<td>1362043</td>
<td>4024999</td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>13826847</td>
<td>79116999</td>
<td>21648943</td>
<td>25617999</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>9034315</td>
<td>1617945087</td>
<td>41683455</td>
<td>4207039377</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>256</td>
<td>100</td>
<td>734878</td>
<td>2628747</td>
<td>11800774</td>
<td>7039197</td>
</tr>
<tr>
<td>256</td>
<td>4757026</td>
<td>42791805</td>
<td>7095646</td>
<td>117700605</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>897733</td>
<td>2628747</td>
<td>1422829</td>
<td>7039197</td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>14151733</td>
<td>162074997</td>
<td>22612129</td>
<td>448636797</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>92458741</td>
<td>2147483647</td>
<td>147983089</td>
<td>7520378877</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### TABLE II Computation Times of Clustering Algorithms

<table>
<thead>
<tr>
<th>Parameters</th>
<th>without C</th>
<th>Parallel</th>
<th>Sequential</th>
<th>with C</th>
<th>Parallel</th>
<th>Sequential</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu$</td>
<td>$Q$</td>
<td>$N$</td>
<td></td>
<td>$\mu$</td>
<td>$Q$</td>
<td>$N$</td>
</tr>
<tr>
<td>2</td>
<td>256</td>
<td>100</td>
<td>2600</td>
<td>69316</td>
<td>456976</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>6500</td>
<td>456976</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>100</td>
<td>3020</td>
<td>69316</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>400</td>
<td>11720</td>
<td>1117216</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>29816</td>
<td>7332880</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>256</td>
<td>100</td>
<td>2800</td>
<td>89116</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>7012</td>
<td>587536</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>100</td>
<td>3220</td>
<td>89116</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>400</td>
<td>12520</td>
<td>1436416</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>31664</td>
<td>9427984</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
function of $N$ (Table 1), which will get significantly smaller when $N$ is significantly smaller, regardless of the value of $Q$. The inefficient part is the computation complexity which usually depends on $\log Q$ no matter how small $N$ is. However, $\log Q$ is usually below 20, which is much less than the actual value of $N$ in most cases.

C. When $N > Q$

Cases 5) and 6) $(Q < N)$ are more complicated and require separate treatment. If some application involves a huge number of patterns, but is significantly smaller, regardless of the value of $Q$, only a small number of PE’s and memory modules are available, the procedures of the previous section can not be used. We now propose the following cyclic storage scheme and procedures for this situation.

Storage Rule 2: To store matrix $A$ of size $N \times N$ in $Q \times N$ memory modules, $A(x,y)$ is stored at location $z$ of memory module $s$, where $s = (x \mod y) \mod Q$ and $z = N \times [x/Q] + y$.

An example is given in Fig. 3 for $Q = 4$ and $N = 10$. Note that Storage Rule 1 is a special case ($Q = N$) of Storage Rule 2. The operations involved in the module number computation ("mod", "floor," and "and") are simple operations. According to this storage rule, each row, column, upper-fold, lower-fold, partial-row-pair, and partial-column-pair can be accessed in $[N/Q]$ steps. We now give a fetch procedure for this situation.

**Procedure** fetch_row($2x, R$)

//fetch row $x$, $x \in N \rightarrow 1$, of matrix $A$ from memory to PE's so that $//\begin{align*}
&\text{if } R(x)=m(x,y) \text{ with } r=y \mod Q \text{ and } k=\lfloor y/Q \rfloor \text{ then } //\end{align*}$

**begin**

for $k=0$ to $[N/Q]-2$

$z:=N \times [x/Q] + k \times \lfloor x \mod Q \rfloor$;

$R(k \mod Q) \leftarrow \text{mem}[z]$;

endfor

$k:=\lfloor N/Q \rfloor - 1$;

$z:=N \times [x/Q] + [x \mod Q]$;

if $z/x \mod Q = N \mod Q$ then $z:=z-1$;

$R(k \mod Q) \leftarrow \text{mem}[z]$;

**end**

Other procedures, such as fetch-upper-fold2, minQ2Q, merge2, SINGLE-LINK2, and COMPLETE-LINK2 can be obtained similarly and are not given here because of their length. The cyclic distribution of the matrices allows the maximum utilization of the inherent parallelism. The number of memory accesses in the general single-link algorithm is proportional to $N \times [N/Q]$. The other time complexities can be obtained by substituting the one-step fetching/storing with $[N/Q]$ steps.

VI. CONCLUSION AND DISCUSSION

Traditional hierarchical clustering requires a large memory storage for the proximity matrix and intensive computation. The sequential algorithm has time complexity of $O(N^2)$. This correspondence proposes general parallel algorithms for single-link and complete-link clustering on SIMD machines. There are $Q$ PE's and $Q$ parallel memory modules. An alignment network is used for both the inter-PE communication and PE-memory communication. Two storage rules are proposed to store the pattern matrix or the proximity matrix in the memory modules. Rows, columns, "fold"s, partial-row-pairs, and partial-column-pairs can all be accessed in parallel. These storage and accessing schemes reduce the space requirement and facilitate parallel operations. When the number of patterns $N$ is smaller than $Q$, the time complexity of the single-link and the complete-link algorithms are $O(N \log Q)$ and $O(N^2 \log Q)$ without the computation of cluster validity measures, and $O(N^2 \log Q)$ and $O(N^2 \log Q)$ with the validity study. When the number of patterns is larger than $Q$, the above complexities become $O(N^2 [N/Q] \log Q)$, $O(N^2 [N/Q] \log Q)$, $O(N^2 [N/Q] \log Q)$, and $O(N^2 [N/Q] \log Q)$, respectively. Modification is required if the proposed algorithms are to work for other clustering schemes. The storage and accessing scheme proposed in this paper can also be used for other pattern analysis algorithms, such as feature extraction and classification.

REFERENCES


An Iterated Estimation of the Motion Parameters of a Rigid Body from Noisy Displacement Vectors

Janet Aisbett

Abstract—An important and still controversial problem is the robust recovery of the parameters of motion of a rigid body from noisy point matches made on perspective views of the body. In this correspondence, we formulate a minimization problem which permits relatively robust parameter estimation and helps overcome poor zoom estimation when the field-of-view is small. We then assume that the motion has a small rotary component, interframe differences are small, and the errors in the system are due to independently distributed errors in the components of the motion.

Manuscript received May 31, 1988; revised May 16, 1990. Recommended for acceptance by W. B. Thompson.

The author is with the Electronic Research Laboratory, Defence Science and Technology Organisation, P.O. Box 1600, Salisbury 5108, South Australia.

IEEE Log Number 903756S.