An efficient parallel approach to reduce sparse matrices with invariant entries
M.P. Bekakos, O.B. Efremides
Department of Informatics, Athens University of Economics and Business, Athens, Greece

Abstract

This paper investigates an efficient parallel technique for reducing sparse matrices that can be applied to analysis tables. This kind of matrices take up a great amount of memory space by the zero entries and, hence, a subtle compaction scheme is necessary. The benefit of the parallel approach introduced herein is that a very compact form results which will contribute to a greatly reduced time when accessing the given data structure.

1 Introduction

Some sequential techniques have been proposed in Knuth[3] and Aho[1] only for the nonzero entries to be represented as list data structures. However, although these methods are quite suited to insert a new entry and delete a redundant one in the matrix, they are not always effective because of the considerable amount of time required to access a random entry.

Various reduction methods for analysis tables have been proposed in Aho[2], Aoe[5, 6], and Joliat[7].

In this work, three parallel algorithms are proposed for reducing static sparse matrices, which can be applied to any analyses tables. Through the first algorithm the number of the connected entries is decreased. The second algorithm indirectly reduces the huge space of the matrix by replacing only the connected entries of each row with one-dimensional array. The last parallel algorithm consists of the retrieval algorithm of the stored entries in the array. Through the experimental results obtained it is proved that the final data structures are of very reasonable size and that the time to access them has been greatly reduced.
Finally, this reduction scheme can be effectively applied to sparse matrices met in circuit analysis, graph theory, etc.

2 Definitions and notations

Let $M$ be an $(n \times n)$ static sparse matrix. $M(a, b)$ represents the entry of $M$ corresponding to the $a$th row and the $b$th column. Conventionally, the similar definitions, procedures, and so forth, associated with both the row and the column of $M$ are described only on the row. In the first place, the essential terms for $M$ are defined in Aoe[6].

**Definition 1**

Let $a_1, a_2, ..., a_n$ be the entries of the $a$th row of $M$. We say that $a_r, a_{r+1}, ..., a_s$, for $1 \leq r \leq s \leq m$, are C (connected)-entries of the $a$th row, if for these entries the following conditions hold:

1. $a_1 = a_2 = ... = a_{r-1} = 0$,
2. $a_{s+1} = a_{s+2} = ... = a_m = 0$,
3. $a_r \neq 0$, and
4. $a_s \neq 0$.

A string $a_r, a_{r+1}, ..., a_s$ is called a CE (connected entries) - string of the $a$th row (denoted by $S(a)$). The set of the CE-strings corresponding to all the rows of $M$ is denoted by $P_M$. Each zero entry in the CE-string is called a CZ (connected zero)-entry and $N(a)$ represents the number of the CZ-entries in the $a$th row.

**Definition 2**

For a given $(n \times n)$ matrix $M$, we define the following sets:

1. $F(a) = \{ b \mid M(a, b) \neq 0 \}$,
2. $U_M = \{ F(a) \mid 1 \leq a \leq n \}$,
3. Let, $Q_1, Q_2, ..., Q_t$, for $t \geq 1$, be disjoint sets, where the element of $q_i$, for $1 \leq i \leq t$, is the column number of $M$. The ordered-sets-sequence

$$R = [Q_1, Q_2, ..., Q_t],$$

and the union is denoted by

$$\text{TOTAL}(R) = Q_1 \cup Q_2 \cup ... \cup Q_t.$$

3 A new parallel compaction approach

Herein, a different strategy to compact a sparse matrix is introduced. According to this new approach the initial matrix $M$ is partitioned into four sub-matrices $M_i$, for $i=1,2,3,4$, as is described in Figure 1. The parallel scheme for compaction presented is based on the three sequential algorithms proposed in Aoe[6]. All of these sequential algorithms are executed in parallel for each one of the four blocks of the initial matrix.
Note that, for the implementation of this new strategy a simulation software tools environment, (rf. Lester[4]), has been utilized, where a time unit of the simulated time is approximately equivalent to one microsecond of the real execution time on a general purpose multiprocessor.

More analytically, the sequential algorithm A described in Figure 2 is executed, in parallel, for each one of the related sub-matrices. The new parallel algorithm A decreases the number of CZ-entries for every submatrix M; by permuting the columns; it consequently generates four ordered-sets-sequence R. Depending on these R’s the new arrangements of the column (of each sub-matrix) are determined. Finally, a new matrix M' is produced. This matrix includes all the four submatrices and has the C-strings compacted in each of the four blocks.

**Parallel Algorithm A**

**Input**: U_M  
**Output**: R’s and, finally, the new matrix M  
**Method**: Described in Figure 5.

This algorithm eventually produces twice as fast timing results, even when it runs on a single processor, as compared with the sequential algorithm, which is executed for each block, when it is utilized to compact the initial matrix M without any partitioning. The results obtained for small matrices are depicted in Tables 1 and 2.  
Note that, the number of utilized processors always equals the number of the matrix partitions.

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Serial time for sequential algorithm A</th>
<th>Serial time for parallel algorithm A</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>6x6</td>
<td>51700</td>
<td>22248</td>
<td>2.32</td>
</tr>
<tr>
<td>8x8</td>
<td>86072</td>
<td>45583</td>
<td>1.89</td>
</tr>
<tr>
<td>10x10</td>
<td>121044</td>
<td>58193</td>
<td>2.08</td>
</tr>
<tr>
<td>12x12</td>
<td>156616</td>
<td>79507</td>
<td>1.99</td>
</tr>
<tr>
<td>14x14</td>
<td>192788</td>
<td>101841</td>
<td>1.89</td>
</tr>
</tbody>
</table>

**Table 2. Relative Speedups**

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Serial time for parallel algorithm A</th>
<th>Parallel time for parallel algorithm A</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>6x6</td>
<td>22248</td>
<td>7373</td>
<td>3.02</td>
</tr>
<tr>
<td>8x8</td>
<td>45583</td>
<td>15664</td>
<td>2.91</td>
</tr>
<tr>
<td>10x10</td>
<td>58193</td>
<td>17961</td>
<td>3.24</td>
</tr>
<tr>
<td>12x12</td>
<td>79507</td>
<td>24171</td>
<td>3.29</td>
</tr>
<tr>
<td>14x14</td>
<td>101841</td>
<td>30682</td>
<td>3.32</td>
</tr>
</tbody>
</table>

The technique proposed rearranges the columns of M and produces a new matrix. We may call, without any confusion, the new matrix as M. In the second parallel algorithm, the new matrix M is partitioned again into four blocks, as before, and the sequential algorithm, described in Figure 3, is ap-
14 High-Performance Computing in Engineering

plied to each one of them. The C-entries of each row, of the submatrices $M_i$, are stored in a local one-dimensional array (called VALUE), while three local one-dimensional arrays (called BASE, HEAD, and TAIL) are used for the retrieval. The HEAD and TAIL arrays for each block are obtained via the next procedure.

**Procedure 1**

Suppose that $S(a)=a_r \, a_{r+1} \ldots \, a_s$, for the $a_{th}$ row in each $M_i$. Set HEAD($a$) = $r$ and TAIL($a$) = $s$.

**Parallel Algorithm B**

Input: $P_M$ and HEAD array
Output: VALUE and BASE array for each submatrix $M_i$
Method: Described in Figure 6.

Tables 3 and 4 present the execution timing results obtained for the sequential and parallel algorithms.

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Serial time for sequential algorithm B</th>
<th>Serial time for parallel algorithm B</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>6x6</td>
<td>27529</td>
<td>13918</td>
<td>1.98</td>
</tr>
<tr>
<td>8x8</td>
<td>31998</td>
<td>18670</td>
<td>1.71</td>
</tr>
<tr>
<td>10x10</td>
<td>36995</td>
<td>24118</td>
<td>1.53</td>
</tr>
<tr>
<td>12x12</td>
<td>42520</td>
<td>30246</td>
<td>1.40</td>
</tr>
<tr>
<td>14x14</td>
<td>48573</td>
<td>37054</td>
<td>1.31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Serial time for parallel algorithm B</th>
<th>Parallel time for parallel algorithm B</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>6x6</td>
<td>13918</td>
<td>4961</td>
<td>2.81</td>
</tr>
<tr>
<td>8x8</td>
<td>18670</td>
<td>6531</td>
<td>2.86</td>
</tr>
<tr>
<td>10x10</td>
<td>24118</td>
<td>8334</td>
<td>2.89</td>
</tr>
<tr>
<td>12x12</td>
<td>30246</td>
<td>10351</td>
<td>2.92</td>
</tr>
<tr>
<td>14x14</td>
<td>37054</td>
<td>12581</td>
<td>2.95</td>
</tr>
</tbody>
</table>

The parallel algorithm C utilizes the BASE, HEAD and TAIL arrays of each submatrix and decides in which one it will search for the required elements. It is a slow retrieval algorithm, as depicted in Table 5, but the time it requires to retrieve the appropriate elements is too small to negatively affect the overall speedup of the new approach. In Figure 4 is described the corresponding sequential algorithm proposed in Aoe[6], while in Table 6 are presented the timing results obtained by the new algorithm when it is executed sequentially and in parallel utilizing four different processors.

**Parallel Algorithm C**

Input: Row number $a$ and column number $b$ for each $M_i$
Output: DET
Method: Described in Figure 7
Table 5. Sequential Algorithm Enhancement

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Serial time for sequential algorithm C</th>
<th>Serial time for parallel algorithm C</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>6x6</td>
<td>315</td>
<td>888</td>
<td>0.35</td>
</tr>
<tr>
<td>8x8</td>
<td>313</td>
<td>925</td>
<td>0.33</td>
</tr>
<tr>
<td>10x10</td>
<td>343</td>
<td>986</td>
<td>0.34</td>
</tr>
<tr>
<td>12x12</td>
<td>373</td>
<td>1008</td>
<td>0.37</td>
</tr>
<tr>
<td>14x14</td>
<td>403</td>
<td>1092</td>
<td>0.36</td>
</tr>
</tbody>
</table>

Table 6. Relative Speedups

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Serial time for parallel algorithm C</th>
<th>Parallel time for parallel algorithm C</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>6x6</td>
<td>888</td>
<td>663</td>
<td>1.34</td>
</tr>
<tr>
<td>8x8</td>
<td>925</td>
<td>712</td>
<td>1.30</td>
</tr>
<tr>
<td>10x10</td>
<td>986</td>
<td>781</td>
<td>1.26</td>
</tr>
<tr>
<td>12x12</td>
<td>1008</td>
<td>861</td>
<td>1.17</td>
</tr>
<tr>
<td>14x14</td>
<td>1092</td>
<td>941</td>
<td>1.16</td>
</tr>
</tbody>
</table>

4 Discussion and conclusive remarks

In this work, a new parallel method is investigated for reducing sparse matrices, while a modified retrieval algorithm is presented. The experimental results proved that even though the retrieval algorithm is slow when compared with the sequential algorithm proposed in Aoe[6], the overall access time to the data structure is very fast.

The more important advantage is that our method works effectively for any analysis tables. This reduction technique can be well used for all static sparse matrices in graph theory, circuit analysis, and so on.

To conclude, the new parallel approach can be generalized by partitioning the initial matrix M into more than four blocks. More specifically, the matrix under consideration can be partitioned into several submatrices depending on its initial size n. The only restriction is that we should not have a submatrix partitioning of size less than (3x3). Further partitioning will eventually lead to the process of very small size matrices, a fact which will contribute to the minimization of the effectiveness of the approach.

References

1. Book:

2. Paper in journal:


3. Paper in conference proceedings:


![Figure 1: Matrix partitioning method](image)

BEGIN
FOR F(b) in U_M such that MEMBER(F(b) ∩ F(c)) is maximum DO
BEGIN
R := [Q=F(b)];
U_M := U_M - {F(b)}
END;
END;

REPEAT
FOR F(a) in U_M such that MEMBER(TOTAL(R) ∩ F(a)) is maximum DO
BEGIN
R := NEW(EMPTY(MERGE(R,F(a))));
U_M := U_M - {F(a)}
END;
UNTIL U_M = Ø ;

![Figure 2: Sequential Algorithm A](image)
BEGIN
   PAI := PM;
   FOR S(a) and S(c) in PAI such that COMMON(S(a), S(c)) is maximum DO
      BEGIN
         TEMP := CONNECT(S(a), S(c));
         PAI := PAI - {S(a), S(c)};
      END;
   REPEAT
      FOR S(a) in PAI such that COMMON(TEMP, S(a)) is maximum DO
         BEGIN
            TEMR := CONNECT(TEMP, S(a));
            PAD := PAI - {S(a)};
         END;
      UNTIL PAI = \emptyset;
   FOR all e for 1 ≤ e ≤ k DO VALUE(e) := d_e;
   FOR all S(a) := a, a_{e+1}, ... a, in P^, such that d_e = a, d_{i+1} = a_{e+1}, ... d_{i+(e-1)} = a_i DO
      BASE(a) := u - HEAD(a);
   END

Figure 3: Sequential Algorithm B

BEGIN
   IF HEAD(a) ≤ b ≤ TAIL(a) THEN
      DET := VALUE(BASE(a) + B)
   ELSE
      DET := 0
   END

Figure 4: Sequential Algorithm C

Program Parallel Algorithm A;
   Architecture Shared(5);
   Declarations...
   Function Member (a : list) : integer; ...
   Function Eq(a, b : list) : boolean; ...
   Procedure Copy (a : list; var b : list); ...
   Procedure Delete(var a: list; element: integer); ...
   Procedure Remove(var a: list); ...
   Procedure Sort(var a: list); ...
   Procedure Union (a, b : list; var c : list); ...
   Procedure InterSection (a, b : list; var c : list); ...
   Procedure Minus (a, b : list; var c : list); ...
   Procedure MainInitial; ...
   Procedure Initial(s, e1 s2 e2 : integer; var M : typeM); ...
   Procedure Update(s1 s2: integer; NM : typeM); ...
   Procedure Find_1(M : typeM; ti :integer; var F: typeF); ...
   Procedure Find_2(M : typeM; ti :integer; var F: typeF); ...
   Procedure Reunimber(var R : rlist); ...
   Procedure Empty(var R : rlist); ...
   Procedure Total(R : rlist; var totalR : list); ...
   Procedure Find_1(Q(R : rlist; ALPlist; var q): integer); ...
   Procedure Find_2(Nim(R : rlist; ALPlist; q); tj); ...
   Procedure Merge(var R : rlist; Fa, ALP : list); ...
   Procedure Main (par: integer); ...
   BEGIN (* main program *)
      MainInitial;
      forall par=1 to processors do
         (@par) Main(par);
      for i:=1 to n do
         begin
            for j:=1 to n do
               write(NIM[i,j]);
            writeln;
         end;
   End (* Parallel Algorithm A *).

Figure 5: Parallel Algorithm A
Program Parallel__Algorithm__B;
  Architecture Shared(5);
  Declarations;
  Function Member (a : list) : integer;
  Procedure Copy (a : list; var b : list);
  Procedure Remove(var a: list);
  Procedure MainInitial;
  Procedure Initial(a, b : list; var M : typeM);
  Procedure Create8(var S : typeS; M : typeM; i : integer;
    var Head, Tail : typeA);
  Procedure Find__Best(A, B : list; var y: list; var length : integer);
  Procedure Common(var SI, S2 : list; var C : list; i, j : integer;
    var li, lj, max length : integer);
  Procedure Find__Pos (var A, B : list; var position : char; var f, l : list);
  Procedure Cut(f, l : list; position: char; var ans, Y : list);
  Procedure Connect(A, B, X, Y, f, l2 : list; pos1, pos2 : char;
    var TEMP : list);
  Procedure Find__BASE(A: list; i: integer; V: typeV;
    H integer, var BASE : typeA);
  Procedure Main (par : integer);
Begin (* main program *)
  MainInitial;
  forall par= 1 to processors do
    (@par) Main(par);
End (* Parallel__Algorithm__B *).

Figure 6: Parallel Algorithm B

Program Parallel__Algorithm__C;
  Architecture Shared(5);
  Declarations;
  Procedure Initial;
  Procedure Main(par: integer);
Begin (* main program *)
  Initial;
  forall par=1 to processors do
    (@par) Main(par);
End (* Parallel__Algorithm__C *).

Figure 7: Parallel Algorithm C