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 (nxn) 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 $a_1, a_2, \dots, a_n$ be the entries of the a_{th} row of M. We say that, $a_r, a_{r+1}, \dots, a_s$, 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) $a_1 = a_2 = \dots = a_{r-1} = 0$,
(2) $a_{s+1} = a_{s+2} = \dots = a_m = 0$,
(3) $a_r \neq 0$, and
(4) $a_s \neq 0$.

A string $a_r\, a_{r+1} \dots a_s$ 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 (nxn) 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, $Q_1, Q_2, \dots, Q_t$, 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

$$R = [Q_1, Q_2, \dots, Q_t]$$

and the union is denoted by

$$TOTAL(R) = Q_1 \cup Q_2 \cup \dots \cup Q_T .$$

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.

Transactions on Information and Communications Technologies vol 9, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

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_i 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 1. Sequential Algorithm Enhancement

Matrix Size	**Serial time for sequential algorithm A**	**Serial time for parallel algorithm A**	**Speedup**
6x6	51700	22248	2.32
8x8	86072	45583	1.89
10x10	121044	58193	2.08
12x12	156616	79507	1.99
14x14	192788	101841	1.89

Table 2. Relative Speedups

Matrix Size	**Serial time for parallel algorithm A**	**Parallel time for parallel algorithm A**	**Speedup**
6x6	22248	7373	3.02
8x8	45583	15664	2.91
10x10	58193	17961	3.24
12x12	79507	24171	3.29
14x14	101841	30682	3.32

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 applied 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 3. Sequential Algorithm Enhancement

Matrix Size	**Serial time for sequential algorithm B**	**Serial time for parallel algorithm B**	**Speedup**
6x6	27529	13918	1.98
8x8	31998	18670	1.71
10x10	36995	24118	1.53
12x12	42520	30246	1.40
14x14	48573	37054	1.31

Table 4. Relative Speedups

Matrix Size	**Serial time for parallel algorithm B**	**Parallel time for parallel algorithm B**	**Speedup**
6x6	13918	4961	2.81
8x8	18670	6531	2.86
10x10	24118	8334	2.89
12x12	30246	10351	2.92
14x14	37054	12581	2.95

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

Matrix Size	**Serial time for sequential algorithm C**	**Serial time for parallel algorithm C**	**Speedup**
6x6	315	888	0.35
8x8	313	925	0.33
10x10	343	986	0.34
12x12	373	1008	0.37
14x14	403	1092	0.36

Table 6. Relative Speedups

Matrix Size	**Serial time for parallel algorithm C**	**Parallel time for parallel algorithm C**	**Speedup**
6x6	888	663	1.34
8x8	925	712	1.30
10x10	986	781	1.26
12x12	1008	861	1.17
14x14	1092	941	1.16

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:

1. Aho, A.V., Hopcroft, J.E. and Ullman, J.D. *The Design and Analysis of Computer Algorithms*, pp. 44-52, Addison-Wesley, 1974.
2. Aho, A.V. and Ullman, J.D. *The Theory of Parsing, Translation and Compiling*, Vol. I, pp. 368-399, Vol. II, pp. 579-675, Prentice-Hall, 1972, 1973.
3. Knuth, D.E. *The Art of Computer Programming*, Vol. I, pp. 299-301, Addison-Wesley, 1973.
4. Lester, B.P. *The Art of Parallel Programming*, Prentice-Hall Int. Inc., Englewood Cliffs, N. Jersey, 1993.

2. Paper in journal:

5. Aoe, J.,. Fukuoka, I, Yamamoto, Y. and Shimada, R. *A Practical Method for Reducing Lexical and Syntax Analyzer Tables*, Bulletin of Faculty of Engineering, Tokushima University, 17, pp. 11-27, 1980.
6. Aoe, J., Yamamoto, Y. and Shimada, R. *A Practical Method for Reducing Sparse Matrices with Invariant Entries*, Int. J. Computer Math., Gordon and Breach Science Publishers Inc., Vol.12, pp. 97-111, 1982

3. Paper in conference proceedings:

7. Joliat, M.L. *Practical Minimization of LR(k) Parser Tables*, Information Processing 74, pp. 376-380, NorthHolland, 1974.

$$M = \left[\begin{array}{ccc:ccc} a_{11} & a_{21} & a_{31} & a_{41} & a_{51} & a_{61} \\ a_{21} & a_{22} & a_{32} & a_{42} & a_{52} & a_{62} \\ a_{31} & a_{23} & a_{33} & a_{43} & a_{53} & a_{63} \\ \hdashline a_{41} & a_{24} & a_{34} & a_{44} & a_{54} & a_{64} \\ a_{51} & a_{25} & a_{35} & a_{45} & a_{55} & a_{65} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} \end{array}\right]$$

Figure 1 : Matrix partitioning method

```
BEGIN
  FOR F(b) in $U_M$ such that MEMBER(F(b) $\cap$ F(c)) is maximum DO
    BEGIN
        R:= [$Q_1$=F(b)];
        $U_M$ := $U_M$ - {F(b)}
    END;

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

Figure 2 : Sequential Algorithm A

```
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
        TEMP:=CONNECT(TEMP,S(a));
        PAI:= PAI - {S(a)}
      END
  UNTIL PAI = Ø ;

  FOR all e for $1 \le e \le k$ DO VALUE(e):= $d_e$ ;
  FOR all S(a)= $a_r a_{r+1} \ldots a_s$ in $P_M$ such that $d_u = a_r$, $d_{u+1} = a_{r+1}, \ldots, d_{u+(s-r)} = a_s$ 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 Algoritm C

```
Program Parallel__Algorithm__A;
  Architecture Shared(5);
Declarations...
Function Member (a : list) : integer; ...
Function Neq(a,b : list):boolean; ...
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(s1,e1,s2,e2 : integer; var M : typeM); ...
Procedure Update(s1,s2 : integer; NM : typeM); ...
Procedure Find__F(M : typeM; i:integer; var F:typeF); ...
Procedure Find__Max(A,B : list; i, max : integer;
                    var maxset : integer; var ALP :list); ...
Procedure Renumber(var R : rlist); ...
Procedure Empty(var R : rlist); ...
Procedure Total(R : rlist; var totalR : list); ...
Procedure Find__Q(R : rlist; ALP:list; var qi,qj : integer); ...
Procedure Find__Num(R : rlist; ALP:list; qi,qj : integer;
    var NUM1,NUM2: integer; var BE1, BE2 :list); ...
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]:3);
      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(s1,e1,s2,e2 : integer; var M : typeM); ...
Procedure CreateS(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 S1, S2 : list; var C : list; i,j : integer;
                  var li,lj, maxlength : integer); ...
Procedure Find__Pos (var A, B : list; var position : char; var f, l : list); ...
Procedure Cut(A,f,l : list; position: char; var ans, Y : list); ...
Procedure Connect(A, B,X,Y,Z, f1, f2, l1, 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